Crypto Forum
Internet-Draft
Intended status: Informational
Expires: 3 September 2026

M. Orr湛
CNRS
C. Yun
Apple, Inc.

2 March 2026

Interactive Sigma Proofs
draft-irtf-cfrg-sigma-protocols-02

Abstract

A Sigma Protocol is an interactive zero-knowledge proof of knowledge that allows a prover to convince a verifier of the validity of a statement. It satisfies the properties of completeness, soundness, and zero-knowledge, as described in Section 3.

This document describes Sigma Protocols for proving knowledge of pre-images of linear maps in prime-order elliptic curve groups. Examples include zero-knowledge proofs for discrete logarithm relations, ElGamal encryptions, Pedersen commitments, and range proofs.

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.

Copyright Notice

Table of Contents

1. Introduction

Any Sigma Protocol must define a _commitment_ (computed by the prover), a _challenge_ (randomly sampled from a specific distribution), and a _response_ (computed by the prover). One of the advantages of Sigma Protocols is their composability, which enables the construction of more complex protocols. A classic example is the OR composition [CramerDS94]. Given a Sigma Protocol for N relations, it is possible to prove knowledge of one of N witnesses for those relations . The composed sigma protocols can be made non-interactive using the Fiat-Shamir transformation [Cramer97]. However, such compositions must be handled carefully to preserve security properties as discussed in Section 3.

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 a 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 as defined in Section 1.1.1. This step generally requires access to a high-quality entropy source to perform the commitment. Leakage of even just 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 return value response is a public value and is transmitted to the verifier.

* 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.

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

The simulated transcript (commitment, challenge, response) must be indistinguishable from the one generated using the prover algorithms.

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.

1.1.1. Randomized algorithms

The generation of proofs involves randomized algorithms that take as input a source of randomness, denoted as rng. The functionality required in this document is a secure way to sample non-zero scalars uniformly at random. Algorithms access this functionality through the following interface.

```
class CSRNG(ABC):
    def getrandom(self, length: int) -> bytes:
        pass

    def random_scalar(self) -> groups.Scalar:
        pass
```

Implementations MUST use a cryptographically secure pseudorandom number generator (CSPRNG) to sample non-zero scalars either by using rejection sampling methods or reducing a large bitstring modulo the group order. Refer to Section A.4 of [FIPS.186-5] for guidance about these methods.

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 [CS97] 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 needs 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(): returns the order of the group p.

* 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], fails if the input is not the valid canonical byte representation of an array of elements 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(rng): samples a scalar from the RNG. Securely decoding random bytes into a random scalar is described in Section 9.1.4 of [fiat-shamir].

* 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 array of elements of the scalar field. 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. Witness representation

A witness is an array of scalar elements. The length of the array is denoted num_scalars.

Witness = [Scalar; num_scalars]

2.2.2. Linear map

A _linear map_ takes a Witness (an array of num_scalars in the scalar field) and maps it to an array of group elements. The length of the image is denoted num_elements.

Linear maps can be represented as matrix-vector multiplications, where the multiplication is the elliptic curve scalar multiplication defined in Section 2.1.

Since the matrix is oftentimes sparse, it is stored 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; num_scalars]) -> list[Group; num_eleme
nts]
```

2.2.2.1. Initialization

The linear map LinearMap is initialized with

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

2.2.2.2. Linear map evaluation

A witness can be mapped to a vector of group elements via:

```
map(self, scalars: [Scalar; num_scalars]) -> list[Group; num_elements]

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
```

2.2.3. Statements for linear relations

A LinearRelation encodes a proof statement of the form linear_map(witness) = image, and is used to prove knowledge of a witness that produces image under linear map. It internally stores linear_map (cf. Section 2.2.2) and an image (an array of num_elements Group elements).

```
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
```

2.2.3.1. 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)

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

Procedure:

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

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
```

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

2.2.4. Core protocol

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

2.2.5. Prover procedures

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

2.2.5.1. Prover commitment

```
prover_commit(self, witness, rng)

Inputs:

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

Outputs:

- A (private) prover state, holding the information of the interactive prover necessary f
or 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 = [rng.random_scalar() 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)
```

2.2.5.2. 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)]

2.2.6. 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

2.2.7. 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. [var_G, var_X, var_H, var_Y] = statement.allocate_elements(4)
4. statement.set_elements([(var_G, G), (var_H, H), (var_X, X), (var_Y, Y)])
5. statement.append_equation(X, [(var_x, G)])
6. 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:

```
1. statement = LinearRelation()
2. var_x, var_r = statement.allocate_scalars(2)
3. [var_G, var_H, var_C] = statement.allocate_elements(3)
4. statement.set_elements([(var_G, G), (var_H, H), (var_C, C)])
5. statement.append_equation(C, [(var_x, G), (var_r, H)])
```

2.3. Ciphersuites

We consider ciphersuites of prime-order elliptic curve groups.

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 Protocols have the following properties:

* *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.

* *Honest verifier 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 honest verifiers gain no knowledge about the witness.

* *Completeness*: If the statement being proved is true, an honest verifier can be convinced of this fact by an honest prover via the proof.

* *Deniable*: Because Interactive Sigma Protocols don’t have transferable message authenticity, a third party (not the prover or verifier) cannot be convinced that the prover made the proof. This means that the Sigma Protocol interaction is not transferable as evidence to a third party.

3.1. Privacy Considerations

Sigma Protocols 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.

3.2. Constant-Time Requirements

The prover’s control flow and memory access patterns are typically influenced by the witness. To prevent side-channel leakage of witness information, which may reveal private values, it is important that the implementation of underlying group and field operations are constant-time. Operations such as modular reduction, scalar multiplication, random value generation, and all other group and field operations are required to be constant-time especially when working with inputs which are private to prevent side-channel attacks which may reveal their values. In some cases, such as keyed-verification credentials, also the verifier must be constant-time. Implementations MUST securely delete prover state as soon as it is no longer needed, and SHOULD minimize the lifetime of sensitive material (witness and instance), explicitly zeroize temporary buffers after proof generation, use secure de-allocation mechanisms when available, and reduce exposure in crash dumps, swap/page files, and diagnostic logging.

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.

Acknowledgments

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

References

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>.

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. Shoup, "A Graduate Course in Applied Cryptography", n.d., <https://toc.cryptobook.us/>.

[Cramer97] Cramer, R., "Modular Design of Secure yet Practical Cryptographic Protocols", 1997, <https://ir.cwi.nl/pub/21438>.

[CramerDS94] Cramer, R., Damgaard, I., and B. Schoenmakers, "Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols", 1994, <https://ir.cwi.nl/pub/1456/1456D.pdf>.

[CS97] Camenisch, J. and M. Stadler, "Proof Systems for General Statements about Discrete Logarithms", n.d., <https://crypto.ethz.ch/publications/files/CamSta97b.pdf>.

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

[FIPS-202] "SHA-3 standard :: permutation-based hash and extendable-output functions", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.202, 2015, <https://doi.org/10.6028/nist.fips.202>.

[FIPS.186-5] "Digital Signature Standard (DSS)", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.186-5, February 2023, <https://doi.org/10.6028/nist.fips.186-5>.

[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. Test Vectors

A.1. Seeded PRNG

For interoperability, the random number generator used for test vectors is implemented using the duplex sponge SHAKE128 instantiation in Section 8.1 of [fiat-shamir], absorbing a seed of 32 bytes. The Seeded PRNG is for reproducible test vectors; production implementations MUST use a CSPRNG.

Random scalars are generated squeezing Ns + 16 bytes, seen as a big-endian positive integer and reduced modulo p, as in Section 9.1.4 of [fiat-shamir].

```
class SeededPRNG:
    def __init__(self, seed: bytes, order: int):
        assert(len(seed) == 32)
        self.order = order
        self.hash_state = SHAKE128(b"sigma-proofs/TestDRNG/SHAKE128".ljust(64, b"\x00"))
        self.hash_state.absorb(seed)

    def random_scalar(self) -> Scalar:
        Ns = (self.order.bit_length() + 7) // 8
        random_integer  = OS2IP(self.hash_state.squeeze(Ns + 16))
        return Scalar(random_integer % self.order)
```

The following sections contain test vectors for the Sigma Protocols specified in this document.

A.2. discrete_logarithm

Ciphersuite = sigma-proofs_Shake128_BLS12381
SessionId = 64697363726574655f6c6f6761726974686d
Statement = 0100000001000000010000000000000000000000097f1d3a73197d794
2695638c4fa9ac0fc3688c4f9774b905a14e3a3f171bac586c55e83ff97a1aeffb3a
f00adb22c6bbb2fa861063d133109d361486d5105a7e9c676a7831f8707b940cde05
514a18ca60f09d5d253c4b7b1b4b349d8a8c108f
Witness = 14de3306fc5f57e5d9e2e89caaf03a261f668b621093c17da407ee7462
43a421
Proof = 06a4c2c6e672c645b22be579a8c85df51582866b3af4ac4498d4c0a3253c
e7fe1c079022962b5a9ff682c728754e1e5984727d6e41b9fc7a48fc804a08538e88
Batchable Proof = 936241c2ed1da3b385294db75a499e96ffc71b5014a01db263
b993b718a901259f0d97700216c683fd97edb99ecac9e8423f70c52c0ea33b3037e6
2ffb3cfae8fd20cc5f3da8981aad1e5900deb7ee8c

A.3. dleq

```
Ciphersuite = sigma-proofs_Shake128_BLS12381
SessionId = 646c6571
Statement = 020000000100000001000000000000000000000030000000100000000000000097f1d3a73197d7942695638c4fa9ac0fc3688c4f9774b905a14e3a3f171bac586c55e83ff97a1aeffb3af00adb22c6bbb24cff92be94ce84df8a18bd8f9c7e2f271bc9091002ff1196a7281283c87b563d9c3cf55173d30f57cac60e76830fe4b2fa861063d133109d361486d5105a7e9c676a7831f8707b940cde05514a18ca60f09d5d253c4b7b1b4b349d8a8c108f81eba50cd26d9e72c32af73e57f9f201b76b6c19061210eba4018d488830508c15d8862e09d24b19008a91c85d0aab2b
Witness = 054b258f4428690087c110387c5a27b3036847c4eb3021dacf604bbb697ec4a6
Proof = 2a29d448b76a5511f8ef616b0fb548a237211e6c40404c9e7522ef6d9b8f9a3756f6239886c671da3b45e5deca4b23ee37947b859ebec21e8b9d535b712abf12
Batchable Proof = 936241c2ed1da3b385294db75a499e96ffc71b5014a01db263b993b718a901259f0d97700216c683fd97edb99ecac9e8b6ed99afa6262bb161ee5ee7a9c1ac4d63adb6aa983af069ee60b1c48927f6e4a5609d4a982f35c9cf69aecc3d8f93992db48a698a9154b4f7339afc9830f258d923c9f69683f6d259dca5669e3e901b
```

A.4. pedersen_commitment

```
Ciphersuite = sigma-proofs_Shake128_BLS12381
SessionId = 706564657273656e5f636f6d6d69746d656e74
Statement = 010000000200000002000000000000000000000001000000010000000097f1d3a73197d7942695638c4fa9ac0fc3688c4f9774b905a14e3a3f171bac586c55e83ff97a1aeffb3af00adb22c6bbb2fa861063d133109d361486d5105a7e9c676a7831f8707b940cde05514a18ca60f09d5d253c4b7b1b4b349d8a8c108fb40c3e47aae45fa96715c58ddf5715c96c0765d7fdb919dc08c9fc0b3649b5054d71706a5bf1980df11e5e70390d20d6
Witness = 054b258f4428690087c110387c5a27b3036847c4eb3021dacf604bbb697ec4a628b20b5b8bbc7e534ac549882000877da9b475cc0725b403998a139355ab8af6
Proof = 1152702d85a1a11b53dbbdcf86fc27e31bfd1478d1192da60de113341cfc357547dc2e89734b54c68845324fe71951a75f73c4e31c7ea3136539e9292afd75d268551cec3d44ba6d02fa80f6e88fe9aa59663d68985b41aaa63003221741afed
Batchable Proof = b277abf8285d0c764f8b57cb326399bf9bed4224d698832dd420d63080db3cea5d8e4b5fbd2417c88b76969b344c9afe1b6151cb572ec64038634aa6b2977b86db6c7a01528051b8d18675b6488fb7010d06945f6bba1103b427d25e636a041a4f65ba6236e110b6c897730cbd9f1a1d
```

A.5. pedersen_commitment_dleq

```
Ciphersuite = sigma-proofs_Shake128_BLS12381
SessionId = 706564657273656e5f636f6d6d69746d656e745f646c6571
Statement = 020000000200000002000000000000000000000001000000010000000
0500000002000000000000000300000001000000040000000b2fa861063d133109d36
1486d5105a7e9c676a7831f8707b940cde05514a18ca60f09d5d253c4b7b1b4b349d
8a8c108fb24cff92be94ce84df8a18bd8f9c7e2f271bc9091002ff1196a7281283c8
7b563d9c3cf55173d30f57cac60e76830fe498be754a4ad6f66dedcfca7e23f5d47f
4f913da328c25dd4506ac0ae3744115b2b3fca2dd3ef851faa74a4fdd82e947c8997
75bf2f7af10a80c0ad6cd35a2646bc2e9c8be292111073cc781d483e7eee325aba7b
e547ce566b071e5d463aca55b003bc4efdafcb4717d5b0ea62db9380edb54c6dd905
788216b21d4f3341dd82a038fe2d59bfca0ec4b8cfa801a4d76db289a73c5a7406c2
7d17f398edad6729f29ba1323b978b00e90abf824e0134e0f113ca9d04375df36caf
d59d51aa5437
Witness = 33c24a45ec7c5d15db45372751862cbd11487a6acc8599e1d4b09d85a5
32262f1baffc5c29eef8881343a7d0950a1ee46a39605e63f7cebf0fb2420385527f
53
Proof = 65fe2c4f0e97f2034c874bc141950d5bb8c70ab2bcf6d778aad6d5d5d8d6
3db1160571f08b1c51ba3f83ca80d132a32db2ddab8a0ddd6dc7b6ed7d4574842917
70c7cbeb3670ee9a523a1d1844a7c0d49369b9be4fe79f5e4b328f31ce85c5e1
Batchable Proof = 81c544e0ef1984f63a1b0b51f112d9f51af2bf76d1b5ecf32e
edaebab1ee9a97d63817756a61c3389331c1cf17ac7eb8823c7e780bad136e99c825
a0992b6d34e2b0c79ea945af382a5cde5596959df8127f58f39a4f1050a49aa94945
bb0c3e5b2a32a32f07b8b5834a9bbce6050b38e7a972fe8be310e76a0d1a72227b49
522d41bae56c2b50e65d4bfaff911fda1c40f68ff0df313c4737155271839f4997
```

A.6. bbs_blind_commitment_computation

```
   Ciphersuite = sigma-proofs_Shake128_BLS12381
   SessionId = 6262735f626c696e645f636f6d6d69746d656e745f636f6d70757461
   74696f6e
   Statement = 0100000004000000040000000000000000000000010000000100000
   0200000002000000030000000300000b2fa861063d133109d361486d5105a7e9c67
   6a7831f8707b940cde05514a18ca60f09d5d253c4b7b1b4b349d8a8c108fb24cff92
   be94ce84df8a18bd8f9c7e2f271bc9091002ff1196a7281283c87b563d9c3cf55173
   d30f57cac60e76830fe4899775bf2f7af10a80c0ad6cd35a2646bc2e9c8be2921110
   73cc781d483e7eee325aba7be547ce566b071e5d463aca55b003bc4efdafcb4717d5
   b0ea62db9380edb54c6dd905788216b21d4f3341dd82a038fe2d59bfca0ec4b8cfa8
   01a4d76d8789e9517c935cc3f345bec16dfbdf0777273de701583f2098c3020a10ca
   1f3f93c07f45fd64bb932423317e5a1e74e8
   Witness = 479b2e7e7b15a1d4118e1c887fe73fc4c7938cdd7d88422302b2c61794
   6b4bb633c24a45ec7c5d15db45372751862cbd11487a6acc8599e1d4b09d85a53226
   2f1baffc5c29eef8881343a7d0950a1ee46a39605e63f7cebf0fb2420385527f533b
   7e5ed38fe5ce6cf2dd67fb59e7739981a80aa62b7bc1abf6880346db4f28de
   Proof = 4bfb793c5b79281a5465ec59b8ba6f33bebb7a015181ecc65672ead554c4
   d1a52025fca49faed8cd469f81dbc53a5de3ca0e3dd46f95fed9829f8770e90e72fe
   5a6b5e4b091c5285037b0b56d1d2e13ff3511da0a3076066b5fa0456066287711bee
   58aaed34ad00252e625479e314722b73c2c1645a1fea7bed0c96f5ed89e95f38b662
   efd67399b248d0759691b1466a804d22f032ee08218196f324b9913b
   Batchable Proof = b123897f9ff891e048b0c83eb3d3dc45b8e1d4c5a29941b3a4
   727990ee1dd1028596f66fb5ef2934a7e6e083f0dd15e90139c8bf20adee43783325
   c39a4516bc9aa2c418d9d0d75584b3695d5a9b0ab343ddc80038226d4d4a0ea8b064
   2b55065b960c332af80d3dd52780cfbe3f064264e72e633ef92ed150c722d4af80dd
   34e268b56b6cc9caf54c36b3bdecf62bc831e62d2e3ec166aec3289ee605272df2d1
   69e69bba520326a082a299dca54110
```

Authors' Addresses

Michele Orr湛
CNRS
Email: m@orru.net

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