Crypto Forum
Internet-Draft
Intended status: Informational
Expires: 23 April 2026

M. Orrテケ
CNRS

20 October 2025

Fiat-Shamir Transformation
draft-irtf-cfrg-fiat-shamir-01

Abstract

This document describes how to construct a non-interactive proof via the Fiat窶鉄hamir transformation, using a generic procedure that compiles an interactive proof into a non-interactive one by relying on a stateful hash object that provides a duplex sponge interface.

The duplex sponge interface requires two methods: absorb and squeeze, which respectively read and write elements of a specified base type. The absorb operation incrementally updates the sponge’s internal hash state, while the squeeze operation produces variable-length, unpredictable outputs. This interface can be instantiated with various hash functions based on permutation or compression functions.

This specification also defines codecs to securely map elements from the prover into the duplex sponge domain, and from the duplex sponge domain into verifier messages.

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-fiat-shamir.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-irtf-cfrg-fiat-shamir/.

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

Copyright Notice

Table of Contents

1. Introduction

The Fiat-Shamir transformation is a technique that uses a hash function to convert a public-coin interactive protocol between a prover and a verifier into a corresponding non-interactive protocol. It depends on:

* An _initialization vector_ (IV) uniquely identifying the protocol, the session, and the statement being proven.

* An _interactive protocol_ supporting a family of statements to be proven.

* A _hash function_ implementing the duplex sponge interface, capable of absorbing inputs incrementally and squeezing variable-length unpredictable messages.

* A _codec_, which securely remaps prover elements into the base alphabet, and outputs of the duplex sponge into verifier messages (preserving the distribution).

2. The Duplex Sponge Interface

The duplex sponge interface defines the space (the Unit) where the hash function operates in, plus a function for absorbing and squeezing prover messages. It provides the following interface.

```
class DuplexSponge:
  def new(iv: bytes) -> DuplexSponge
  def absorb(self, x: list[Unit])
  def squeeze(self, length: int) -> list[Unit]
```

Where:

* init(iv: bytes) -> DuplexSponge denotes the initialization function. This function takes as input a 64-byte initialization vector iv and initializes the state of the duplex sponge.

* absorb(self, values: list[Unit]) denotes the absorb operation of the sponge. This function takes as input a list of Unit elements and mutates the DuplexSponge internal state.

* squeeze(self, length: int) denotes the squeeze operation of the sponge. This function takes as input an integral length and squeezes a list of Unit elements of length length.

3. The Codec interface

A codec is a collection of: - functions that map prover messages into the hash function domain, - functions that map hash outputs into a message output by the verifier in the Sigma protocol In addition, the "init" function initializes the hash state with a session ID and an instance label. For byte-oriented codecs, this is just the concatenation of the two prefixed by their lengths.

A codec provides the following interface.

```
class Codec:
    def init(session_id, instance_label) -> hash_state
    def prover_message(self, hash_state, elements)
    def verifier_challenge(self, hash_state) -> verifier_challenge
```

Where:

* init(session_id, instance_label) -> hash_state denotes the initialization function. This function takes as input a session ID and an instance label, and returns the initial hash state.

* prover_message(self, hash_state, elements) -> self denotes the absorb operation of the codec. This function takes as input the hash state, and elements with which to mutate the hash state.

* verifier_challenge(self, hash_state) -> verifier_challenge denotes the squeeze operation of the codec. This function takes as input the hash state to produce an unpredictable verifier challenge verifier_challenge.

The verifier_challenge function must generate a challenge uniformly from the underlying scalar field, from the public inputs given to the verifier. The default way to generate the challenge is by sampling a random (|log_2(p)| + 128)-bit string, parsing it as a big integer, and reducing it modulo the prime order of the group p.

4. Generation of the Initialization Vector

The initialization vector is a 64-byte string that embeds:

* A protocol_id: the unique identifier for the interactive protocol and the associated relation being proven.

* A session_id: the session identifier, for user-provided contextual information about the context where the proof is made (e.g. a URL, or a timestamp).

* An instance_label: the instance identifier for the statement being proven.

It is implemented as follows.

```
hash_state = DuplexSponge.init([0] * 64)
hash_state.absorb(I2OSP(len(protocol_id), 4))
hash_state.absorb(protocol_id)
hash_state.absorb(I2OSP(len(session_id), 4))
hash_state.absorb(session_id)
```

This will be expanded in future versions of this specification.

5. Fiat-Shamir transformation for Sigma Protocols

We describe how to construct non-interactive proofs for sigma protocols. The Fiat-Shamir transformation is parametrized by:

* a SigmaProtocol, which specifies an interactive 3-message protocol as defined in Section 2 of [SIGMA];

* a Codec, which specifies how to absorb prover messages and how to squeeze verifier challenges;

* a DuplexSpongeInterface, which specifies a hash function for computing challenges.

Upon initialization, the protocol receives as input: - session_id, which identifies the session being proven - instance, the sigma protocol instance for proving or verifying

```
class NISigmaProtocol:
    Protocol: SigmaProtocol = None
    Codec: Codec = None
    Hash: DuplexSpongeInterface = None

    def __init__(self, session_id, instance):
```

```
        self.hash_state = self.Codec(iv)
        self.ip = self.Protocol(instance)

    def _prove(self, witness, rng):
        # Core proving logic that returns commitment, challenge, and response.
        # The challenge is generated via the hash function.
        (prover_state, commitment) = self.sigma_protocol.prover_commit(witness, rng)
        self.codec.prover_message(self.hash_state, commitment)
        challenge = self.codec.verifier_challenge(self.hash_state)
        response = self.sigma_protocol.prover_response(prover_state, challenge)
        return (commitment, challenge, response)

    def prove(self, witness, rng):
        # Default proving method using challenge-response format.
        (commitment, challenge, response) = self._prove(witness, rng)
        assert self.sigma_protocol.verifier(commitment, challenge, response)
        assert self.sigma_protocol.verifier(commitment, challenge, response)
        return self.sigma_protocol.serialize_challenge(challenge) + self.sigma_protocol.s
erialize_response(response)

    def verify(self, proof):
        # Before running the sigma protocol verifier, one must also check that:
        # - the proof length is exactly challenge_bytes_len + response_bytes_len
        challenge_bytes_len = self.sigma_protocol.instance.Domain.scalar_byte_length()
        assert len(proof) == challenge_bytes_len + self.sigma_protocol.instance.response_
bytes_len

        # - proof deserialization successfully produces a valid challenge and a valid res
ponse
        challenge_bytes = proof[:challenge_bytes_len]
        response_bytes = proof[challenge_bytes_len:]
        challenge = self.sigma_protocol.deserialize_challenge(challenge_bytes)
        response = self.sigma_protocol.deserialize_response(response_bytes)

        commitment = self.sigma_protocol.simulate_commitment(response, challenge)
        return self.sigma_protocol.verifier(commitment, challenge, response)

    def prove_batchable(self, witness, rng):
        # Proving method using commitment-response format.
        # Allows for batching.
        (commitment, challenge, response) = self._prove(witness, rng)
        # running the verifier here is just a sanity check
        assert self.sigma_protocol.verifier(commitment, challenge, response)
        return self.sigma_protocol.serialize_commitment(commitment) + self.sigma_protocol
.serialize_response(response)

    def verify_batchable(self, proof):
        # Before running the sigma protocol verifier, one must also check that:
        # - the proof length is exactly commit_bytes_len + response_bytes_len
        assert len(proof) == self.sigma_protocol.instance.commit_bytes_len + self.sigma_p
rotocol.instance.response_bytes_len

        # - proof deserialization successfully produces a valid commitment and a valid re
sponse
```

```
        commitment_bytes = proof[:self.sigma_protocol.instance.commit_bytes_len]
        response_bytes = proof[self.sigma_protocol.instance.commit_bytes_len:]
        commitment = self.sigma_protocol.deserialize_commitment(commitment_bytes)
        response = self.sigma_protocol.deserialize_response(response_bytes)

        self.codec.prover_message(self.hash_state, commitment)
        challenge = self.codec.verifier_challenge(self.hash_state)
        return self.sigma_protocol.verifier(commitment, challenge, response)
```

Serialization and deserialization of scalars and group elements are defined by the ciphersuite chosen in the Sigma Protocol. In particular, serialize_challenge, deserialize_challenge, serialize_response, and deserialize_response call into the scalar serialize and deserialize functions. Likewise, serialize_commitment and deserialize_commitment call into the group element serialize and deserialize functions.

5.1. NISigmaProtocol instances (ciphersuites)

We describe noninteractive sigma protocol instances for combinations of protocols (SigmaProtocol), codec (Codec), and hash fuction (DuplexSpongeInterface). Descriptions of codecs and hash functions are in the following sections.

```
class NISchnorrProofShake128P256(NISigmaProtocol):
    Protocol = SchnorrProof
    Codec = P256Codec
    Hash = SHAKE128

class NISchnorrProofShake128Bls12381(NISigmaProtocol):
    Protocol = SchnorrProof
    Codec = Bls12381Codec
    Hash = SHAKE128

class NISchnorrProofKeccakDuplexSpongeBls12381(NISigmaProtocol):
    Protocol = SchnorrProof
    Codec = Bls12381Codec
    Hash = KeccakDuplexSponge
```

6. Codec for Schnorr proofs

We describe a codec for Schnorr proofs over groups of prime order p where Unit = u8.

```
class ByteSchnorrCodec(Codec):
    GG: groups.Group = None

    def prover_message(self, elements: list):
        hash_state.absorb(self.GG.serialize(elements))

    def verifier_challenge(self, hash_state):
        # see https://eprint.iacr.org/2025/536.pdf, Appendix C.
        uniform_bytes = hash_state.squeeze(
            self.GG.ScalarField.scalar_byte_length() + 16
        )
        scalar = OS2IP(uniform_bytes) % self.GG.ScalarField.order
        return scalar
```

We describe a codec for the P256 curve.

```
class P256Codec(ByteSchnorrCodec):
    GG = groups.GroupP256()
```

7. Duplex Sponge Interfaces

7.1. SHAKE128

SHAKE128 is a variable-length hash function based on the Keccak sponge construction [SHA3]. It belongs to the SHA-3 family but offers a flexible output length, and provides 128 bits of security against collision attacks, regardless of the output length requested.

7.1.1. Initialization

new(self, iv)

Inputs:

- iv, a byte array

Outputs:

- a hash state interface

```
1. initial_block = iv + b'\00' * 104  # len(iv) + 104 == SHAKE128 rate
2. self.hash_state = hashlib.shake_128()
3. self.hash_state.update(initial_block)
```

7.1.2. SHAKE128 Absorb

```
absorb(hash_state, x)

Inputs:

- hash_state, a hash state
- x, a byte array

1. h.update(x)
```

7.1.3. SHAKE128 Squeeze

```
squeeze(hash_state, length)

Inputs:

- hash_state, the hash state
- length, the number of elements to be squeezed

1. return self.hash_state.copy().digest(length)
```

7.2. Duplex Sponge

A duplex sponge in overwrite mode is based on a permutation function that operates on a state vector. It implements the DuplexSpongeInterface and maintains internal state to support incremental absorption and variable-length output generation.

7.2.1. Initialization

This is the constructor for a duplex sponge object. It is initialized with a 64-byte initialization vector.

```
new(iv)

Inputs:
- iv, a 64-byte initialization vector

Procedure:
1. self.absorb_index = 0
2. self.squeeze_index = self.permutation_state.R
3. self.rate = self.permutation_state.R
4. self.capacity = self.permutation_state.N - self.permutation_state.R
```

7.2.2. Absorb

The absorb function incorporates data into the duplex sponge state using overwrite mode.

```
absorb(self, input)

Inputs:
- self, the current duplex sponge object
- input, the input bytes to be absorbed

Procedure:
1. self.squeeze_index = self.rate
2. while len(input) != 0:
3.     if self.absorb_index == self.rate:
4.         self.permutation_state.permute()
5.         self.absorb_index = 0
6.     chunk_size = min(self.rate - self.absorb_index, len(input))
7.     next_chunk = input[:chunk_size]
8.     self.permutation_state[self.absorb_index:self.absorb_index + chunk_size] = next_ch
unk
9.     self.absorb_index += chunk_size
10.    input = input[chunk_size:]
```

7.2.3. Squeeze

The squeeze operation extracts output elements from the sponge state, which are uniformly distributed and can be used as a digest, key stream, or other cryptographic material.

```
squeeze(self, length)

Inputs:
- self, the current duplex sponge object
- length, the number of bytes to be squeezed out of the sponge

Outputs:
- digest, a byte array of ‘length‘ elements uniformly distributed

Procedure:
1. output = b’’
2. while length != 0:
3.     if self.squeeze_index == self.rate:
4.         self.permutation_state.permute()
5.         self.squeeze_index = 0
6.         self.absorb_index = 0
7.     chunk_size = min(self.rate - self.squeeze_index, length)
8.     output += bytes(self.permutation_state[self.squeeze_index:self.squeeze_index+chunk
_size])
9.     self.squeeze_index += chunk_size
10.    length -= chunk_size
11. return output
```

7.2.4. Keccak-f[1600] Implementation

Keccak-f is the permutation function underlying [SHA3].

KeccakDuplexSponge instantiates DuplexSponge with Keccak-f[1600], using rate R = 136 bytes and capacity C = 64 bytes.

8. Codecs registry

8.1. Elliptic curves

8.1.1. Notation and Terminology

For an elliptic curve, we consider two fields, the coordinate fields, which indicates the base field, the field over which the elliptic curve equation is defined, and the scalar field, over which the scalar operations are performed.

The following functions and notation are used throughout the document.

* concat(x0, ..., xN): Concatenation of byte strings.

* bytes_to_int and scalar_to_bytes: Convert a byte string to and from a non-negative integer. bytes_to_int and scalar_to_bytes are implemented as OS2IP and I2OSP as described in [RFC8017], respectively. Note that these functions operate on byte strings in big-endian byte order.

* The function ecpoint_to_bytes converts an elliptic curve point in affine-form into an array string of length ceil(ceil(log2(coordinate_field_order))/ 8) + 1 using int_to_bytes prepended by one byte. This is defined as

```
ecpoint_to_bytes(element)
Inputs:
- ‘element‘, an elliptic curve element in affine form, with attributes ‘x‘ and ‘y‘ corres
ponding to its affine coordinates, represented as integers modulo the coordinate field or
der.

Outputs:

A byte array

Constants:

field_bytes_length, the number of bytes to represent the scalar element, equal to ‘ceil(l
og2(field.order()))‘.

1. byte = 2 if sgn0(element.y) == 0 else 3
2. return I2OSP(byte, 1) + I2OSP(x, field_bytes_length)
```

8.1.2. Absorb scalars

```
absorb_scalars(hash_state, scalars)

Inputs:

- hash_state, the hash state
- scalars, a list of elements of the elliptic curve’s scalar field

Constants:

- scalar_byte_length = ceil(384/8)

1. for scalar in scalars:
2.     hash_state.absorb(scalar_to_bytes(scalar))
```

Where the function scalar_to_bytes is defined in Section 8.1.1

8.1.3. Absorb elements

```
absorb_elements(hash_state, elements)

Inputs:

- hash_state, the hash state
- elements, a list of group elements

1. for element in elements:
2.     hash_state.absorb(ecpoint_to_bytes(element))
```

8.1.4. Squeeze scalars

```
squeeze_scalars(hash_state, length)

Inputs:

- hash_state, the hash state
- length, an unsigned integer of 64 bits determining the output length.

1. for i in range(length):
2.     scalar_bytes = hash_state.squeeze(field_bytes_length + 16)
3.     scalars.append(bytes_to_scalar_mod_order(scalar_bytes))
```

9. References

9.1. Normative References

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/rfc/rfc8017>.

[SIGMA] Orrテケ, M. and C. Yun, "Interactive Sigma Proofs", Work in Progress, Internet-Draft, draft-irtf-cfrg-sigma-protocols-00, 8 August 2025, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-sigma-protocols-00>.

9.2. Informative References

[SHA3] "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", n.d., <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>.

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-irtf-cfrg-sigma-protocols/tree/main/poc/vectors).

Author’s Address

Michele Orrテケ
CNRS
Email: m@orru.net