

Zkproof
Internet-Draft
Intended status: Informational
Expires: 8 January 2026

M. Orrテケ
CNRS
7 July 2025

Fiat-Shamir Transformation
draft-orry-zkproof-fiat-shamir-00

Abstract

This document describes how to construct a non-interactive proof via the Fiat-Shamir 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-zkproof-sigma-protocols/draft-orry-zkproof-fiat-shamir.html>. Status information for this document may be found at <https://datatracker.ietf.org/doc/draft-orry-zkproof-fiat-shamir/>.

Discussion of this document takes place on the Crypto Forum 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-zkproof-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/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 8 January 2026.

Copyright Notice

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
2. The Duplex Sponge Interface	3
3. The Codec interface	4
4. Generation of the Initialization Vector	4
5. Fiat-Shamir transformation for Sigma Protocols	5
5.1. Codec for Linear maps	6
6. Ciphersuites	7
6.1. SHAKE128	7
6.1.1. Initialization	7
6.1.2. SHAKE128 Absorb	7
6.1.3. SHAKE128 Squeeze	7
6.2. Duplex Sponge	8
6.2.1. Initialization	8
6.2.2. Absorb	8
6.2.3. Squeeze	9

6.2.4. Keccak-f[1600] Implementation	9
7. Codecs registry	9
7.1. Elliptic curves	9
7.1.1. Notation and Terminology	9
7.1.2. Absorb scalars	10
7.1.3. Absorb elements	10
7.1.4. Squeeze scalars	11
8. References	11
8.1. Normative References	11
8.2. Informative References	11
Test Vectors	11
Author's Address	11

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

A duplex sponge operates over an abstract `Unit` type and 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 32-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 provides the following interface.

```
class Codec:
    def new(iv: bytes) -> Codec
    def prover_message(self, prover_message)
    def verifier_challenge(self) -> verifier_challenge
```

Where:

- * `init(iv: bytes) -> DuplexSponge` denotes the initialization function. This function takes as input a 32-byte initialization vector `iv` and initializes the state of the codec.
- * `prover_message(self, prover_message) -> self` denotes the absorb operation of the codec. This function takes as input a prover message `prover_message` and mutates the codec's internal state.
- * `verifier_challenge(self) -> verifier_challenge` denotes the squeeze operation of the codec. This function takes no inputs and uses the codec's internal state to produce an unpredictable verifier challenge `verifier_challenge`.

4. Generation of the Initialization Vector

The initialization vector is a 32-bytes 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] * 32)
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 `Codec`, which specifies how to absorb prover messages and how to squeeze verifier challenges;
- * a `SigmaProtocol`, which specifies an interactive 3-message protocol.

Upon initialization, the protocol receives as input an iv of 32-bytes which uniquely identifies the protocol and the session being proven and (optionally) pre-processes some information about the protocol using the instance.

```

class NISigmaProtocol:
    Protocol: SigmaProtocol
    Codec: Codec

    def init(self, iv: bytes, instance):
        self.hash_state = self.Codec(iv)
        self.ip = self.Protocol(instance)

    def prove(self, witness, rng):
        (prover_state, commitment) = self.ip.prover_commit(witness, rng)
        challenge = self.hash_state.prover_message(commitment).verifier_challenge()
        response = self.ip.prover_response(prover_state, challenge)

        assert self.ip.verifier(commitment, challenge, response)
        return self.ip.serialize_commitment(commitment) + self.ip.serialize_response(response)

    def verify(self, proof):
        commitment_bytes = proof[:self.ip.instance.commit_bytes_len]
        response_bytes = proof[self.ip.instance.commit_bytes_len:]
        commitment = self.ip.deserialize_commitment(commitment_bytes)
        response = self.ip.deserialize_response(response_bytes)
        challenge = self.hash_state.prover_message(commitment).verifier_challenge()
        return self.ip.verifier(commitment, challenge, response)

```

5.1. Codec for Linear maps

We describe a codec for Schnorr proofs over groups of prime order p that is intended for duplex sponges where `Unit = u8`.

```

class LinearMapCodec:
    Group: groups.Group = None
    DuplexSponge: DuplexSpongeInterface = None

    def init(self, iv: bytes):
        self.hash_state = self.DuplexSponge(iv)

    def prover_message(self, elements: list):
        self.hash_state.absorb(self.Group.serialize(elements))
        # calls can be chained
        return self

    def verifier_challenge(self):
        uniform_bytes = self.hash_state.squeeze(
            self.Group.ScalarField.scalar_byte_length() + 16
        )
        scalar = OS2IP(uniform_bytes) % self.Group.ScalarField.order
        return scalar

```

6. Ciphersuites

6.1. 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.

6.1.1. Initialization

```
new(self, iv)
```

Inputs:

- iv, a byte array

Outputs:

- a hash state interface

1. h = shake_128(iv)
2. return h

6.1.2. SHAKE128 Absorb

```
absorb(hash_state, x)
```

Inputs:

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

1. h.update(x)

6.1.3. SHAKE128 Squeeze

```
squeeze(hash_state, length)
```

Inputs:

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

1. h.copy().digest(length)

6.2. 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.

6.2.1. Initialization

This is the constructor for a duplex sponge object. It is initialized with a 32-byte initialization vector.

```
new(iv)
```

Inputs:

- iv, a 32-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

6.2.2. 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_chunk
9. self.absorb_index += chunk_size
10. input = input[chunk_size:]

6.2.3. 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
```

6.2.4. 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.

7. Codecs registry

7.1. Elliptic curves

7.1.1. 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 $\text{ceil}(\text{ceil}(\log_2(\text{coordinate_field_order}))/8) + 1$ 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'` corresponding 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(log2(field.order()))'`.

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

7.1.2. 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 7.1.1

7.1.3. 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))
```

7.1.4. 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))
```

8. References

8.1. 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>>.

8.2. 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-zkproof-sigma-protocols/tree/main/poc/vectors>).

Author's Address

Michele Orrテケ
CNRS
Email: m@orru.net