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

M. Orrテケ
CNRS

2 March 2026

Fiat-Shamir Transformation
draft-irtf-cfrg-fiat-shamir-02

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 duplex sponge object.

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 duplex sponge's internal state, while the squeeze operation produces variable-length, unpredictable outputs. This interface can be instantiated with different constructions based on permutation or compression functions.

This specification also defines codecs to securely map prover messages into the duplex sponge domain, from the duplex sponge domain into verifier messages. It also establishes how the non-interactive argument string should be serialized.

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 duplex sponge to convert a public-coin interactive protocol between a prover and a verifier into a corresponding non-interactive argument. The term "public-coin" here refers to interactive protocols where all verifier messages are essentially random values sent in the clear. 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 _duplex sponge instantiation_ 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. Security Considerations

The Fiat-Shamir transformation carries over the soundness and witness hiding properties of the interactive proof:

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

* *Soundness*: If the interactive proof is sound, then so is the non-interactive proof. In particular, valid proofs cannot be generated without possession of the corresponding witness.

* *Zero-Knowledge*: If the interactive proof is honest-verifier zero-knowledge, then so is the non-interactive proof. In particular, the resulting argument string does not reveal any information beyond what can be directly inferred from the statement being valid. This ensures that verifiers gain no knowledge about the witness.

In particular, the Fiat-Shamir transformation of Sigma Protocols is a zero-knowledge and sound argument of knowledge.

Note that non-interactive Sigma Protocols do not have deniability, as the non-interactive nature of the protocol implies transferable message authenticity.

3. The Duplex Sponge Interface

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

```
class DuplexSponge:
  def init(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 duplex 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 duplex sponge. This function takes as input an integral length and squeezes a list of Unit elements of length length.

4. The Codec interface

A codec is a collection of: - functions that map prover messages into Units, - functions that map Units into verifier messages, preserving the uniform distribution

A codec provides the following interface.

```
class Codec:
    def prover_message(self, state, elements)
    def verifier_challenge(self, state) -> verifier_challenge
```

Where:

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

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

The verifier_challenge function must generate a challenge from the underlying scalar field that is statistically close to uniform, from the public inputs given to the verifier, as described in Section 9.1.4.

5. Initialization of the Duplex Sponge State

The duplex sponge state is initialized by sequentially absorbing:

* A protocol_id: the unique identifier for the interactive protocol and the associated relation being proven. This identifier MUST be 64 bytes.

* 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). This identifier is currently generated as 32 zero-bytes concatenated with a 32-byte digest derived using the duplex sponge.

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

The session_id is computed as:

```
state = DuplexSponge.init(b"fiat-shamir/session-id".ljust(64, b"\x00"))
state.absorb(session)
session_id = [0] * 32 || state.squeeze(32)
```

The protocol instance label is absorbed without an explicit length prefix. Therefore, the encoding used to produce instance_label MUST be prefix-free.

6. Fiat-Shamir transformation for Sigma Protocols

We describe how to construct non-interactive proofs for sigma protocols. The Fiat-Shamir transformation is parameterized 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 duplex sponge for computing challenges.

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

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

    def __init__(self, session, instance):
        protocol_id = self.get_protocol_id()
        assert len(protocol_id) == 64
        self.sigma_protocol = self.Protocol(instance)
        self.codec = self.Codec()
        instance_label = self.sigma_protocol.get_instance_label()
        session_state = self.DuplexSponge(b"fiat-shamir/session-id".ljust(64, b"\x00"))
        session_state.absorb(session)
        session_id = [0] * 32 || session_state.squeeze(32)
        self.state = self.DuplexSponge(protocol_id)
        self.state.absorb(session_id)
        self.state.absorb(instance_label)

    def _prove(self, witness, rng):
        # Core proving logic that returns commitment, challenge, and response.
```

```
        # The challenge is generated via the duplex sponge.
        (prover_state, commitment) = self.sigma_protocol.prover_commit(witness, rng)
        self.codec.prover_message(self.state, commitment)
        challenge = self.codec.verifier_challenge(self.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)
        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 Nc + response_bytes_len,
        Nc = self.sigma_protocol.instance.Domain.scalar_byte_length()
        assert len(proof) == Nc + self.sigma_protocol.instance.response_bytes_len

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

        # - the re-computed challenge equals the serialized challenge.
        self.codec.prover_message(self.state, commitment)
        expected_challenge = self.codec.verifier_challenge(self.state)
        if challenge != expected_challenge:
            return False

        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.state, commitment)
        challenge = self.codec.verifier_challenge(self.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.

6.1. NISigmaProtocol instances (ciphersuites)

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

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

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

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

7. 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):
           state.absorb(self.GG.serialize(elements))

       def verifier_challenge(self, state):
           # see https://eprint.iacr.org/2025/536.pdf, Appendix C.
           Ns = self.GG.ScalarField.scalar_byte_length()
           uniform_bytes = state.squeeze(
               Ns + 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()
```

8. Duplex Sponge Interfaces

8.1. SHAKE128

SHAKE128 is a variable-length extendable-output function based on the Keccak sponge construction [SHA3]. It belongs to the SHA-3 family and is used here to provide a duplex sponge interface.

8.1.1. Initialization

new(self, iv)

Inputs:

- iv, a byte array

Outputs:

- a duplex sponge instance

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

8.1.2. SHAKE128 Absorb

absorb(state, x)

Inputs:

- state, a duplex sponge state
- x, a byte array

1. h.update(x)

8.1.3. SHAKE128 Squeeze

squeeze(state, length)

Inputs:

- state, the duplex sponge state
- length, the number of elements to be squeezed

1. return self.state.copy().digest(length)

8.2. 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.

8.2.1. 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

8.2.2. 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:]
```

8.2.3. 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
```

8.2.4. 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.

9. Codecs registry

9.1. Elliptic curves

9.1.1. 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.

* OS2IP and I2OSP: Convert a byte string to and from a non-negative integer, as described in [RFC8017]. 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‘ corresponding to its affine coordinates, represented as integers modulo the coordinate field order.

Outputs:

A byte array

Constants:

Ng, the number of bytes to represent an element in the coordinate field, equal to ‘ceil(log2(field.order())/8)‘.

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

9.1.2. Absorb scalars

absorb_scalars(state, scalars)

Inputs:

- state, the duplex sponge
- scalars, a list of elements of the elliptic curve’s scalar field

Constants:

- Ns, the number of bytes to represent a scalar element, equal to ‘ceil(log2(p)/8)‘.

1. for scalar in scalars:
2. state.absorb(I2OSP(scalar, Ns))

9.1.3. Absorb elements

absorb_elements(state, elements)

Inputs:

- state, the duplex sponge
- elements, a list of group elements

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

9.1.4. Decoding random bytes as scalars

Given Ns + 16 bytes, it is possible to generate a scalar modulo p that is statistically close to uniform. Interpret the bytes as a big-endian integer, then reduce it modulo p, where p is the order of the group.

```
squeeze_scalars(state, length)

Inputs:

- state, the duplex sponge
- length, an unsigned integer of 64 bits determining the number of scalars to output.

Constants:

- Ns, the number of bytes to represent a scalar, equal to ‘ceil(log2(p)/8)‘.

1. for i in range(length):
2.     scalar_bytes = state.squeeze(Ns + 16)
3.     scalars.append(OS2IP(scalar_bytes) % p)
```

10. References

10.1. 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>.

10.2. 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>.

Appendix A. Test Vectors

A.1. test_keccak_duplex_sponge_SHAKE128

```
DuplexSponge = SHAKE128
IV = 756e69745f74657374735f6b656363616b5f697600000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
Operation1 = absorb:6261736963206475706c65782073706f6e67652074657374
Operation2 = squeeze:64
Expected = f845c3ef4231a4d6e09c29b1eea0055842246fd57558fd7d93e1302f7
799dd9593d2e4d06eda72d5252ca5b2feff4b8cb324ec96673a7417cf70fa77b1898
991
```

A.2. test_absorb_empty_before_does_not_break_SHAKE128

```
DuplexSponge = SHAKE128
IV = 756e69745f74657374735f6b656363616b5f697600000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
Operation1 = absorb:656d707479206d65737361676520616674657 2
Operation2 = absorb:
Operation3 = squeeze:64
Expected = 3953e577d9e5d4dc7b86d1a62e881f2d1eb750ea3550fcae315854d16
6136ae816ca922a4c7e54d711b8721c8969598449922122768c50313f47eef35020b
73c
```

A.3. test_absorb_empty_after_does_not_break_SHAKE128

```
DuplexSponge = SHAKE128
IV = 756e69745f74657374735f6b656363616b5f697600000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
Operation1 = absorb:
Operation2 = absorb:656d707479206d65737361676520626566 6f7265
Operation3 = squeeze:64
Expected = 6e475edd3c400bec314d5891af570841a547c95d1a651adff9a8bfb70
719a79b5afde316386da13fa83525662df3c5b2367d987bf3dc4199efdb9d0612572
785
```

A.4. test_squeeze_zero_behavior_SHAKE128

```
DuplexSponge = SHAKE128
IV = 756e69745f74657374735f6b656363616b5f697600000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
Operation1 = squeeze:0
Operation2 = absorb:7a65726f2073717565657a652074657374
Operation3 = squeeze:0
Operation4 = squeeze:64
Expected = 4cf7f008057b63cb615547a143f42cf793b86b239f404d2f28b3f0919
7d850eb029df3024ad468be5aceb2fa60e9fb7add98436236be69ddb34314ce7a905
f23
```

A.5. test_squeeze_zero_after_behavior_SHAKE128

```
DuplexSponge = SHAKE128
IV = 756e69745f74657374735f6b656363616b5f697600000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
Operation1 = squeeze:0
Operation2 = absorb:7a65726f2073717565657a65206166746572
Operation3 = squeeze:64
Expected = bd9278e6f65cb854935b3f6b2c51ab158be8ea09744509519b8f06f0c
501d07c429e37f232b6f0955b620ff6226d9d02e4817b1447e7309023a3a14f73587
6ec
```

A.6. test_absorb_squeeze_absorb_consistency_SHAKE128

```
DuplexSponge = SHAKE128
IV = 656467652d636173652d746573742d646f6d61696e2d6162736f72620000000
0000000000000000000000000000000000000000000000000000000000000000000
Operation1 = absorb:696e7465726c65617665206669727374
Operation2 = squeeze:32
Operation3 = absorb:696e7465726c6561766520736 5636f6e64
Operation4 = squeeze:32
Expected = 4d31a75f29851f9f15cd54fa6f2335cbe07b947b9d3c28092c1ba7315
e295921
```

A.7. test_associativity_of_absorb_SHAKE128

```
DuplexSponge = SHAKE128
IV = 6162736f72622d6173736f6369617469766974792d646f6d61696e000000000
0000000000000000000000000000000000000000000000000000000000000000000
Operation1 = absorb:6173736f6369617469766974792074657374206675 6c6c
Operation2 = squeeze:32
Expected = c0faa351141d60678dceff4f3a5760381bb335ad113958b70edf7b242
df01c8a
```

A.8. test_iv_affects_output_SHAKE128

```
DuplexSponge = SHAKE128
IV = 646f6d61696e2d6f6e652d646966666572732d6865726500000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
Operation1 = absorb:697620646966666572656e63652074657374
Operation2 = squeeze:32
Expected = 7650642267cc544abf0e01ce28e2595aec4c2f5b5e5e3720ab5514496
37b35f2
```

A.9. test_multiple_blocks_absorb_squeeze_SHAKE128

```
DuplexSponge = SHAKE128
IV = 6d756c74692d626c6f636b2d6162736f72622d746573740000000000000000
0000000000000000000000000000000000000000000000000000000000000000
Operation1 = absorb:abababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
abababababababababababababababababababababababababababababababab
Operation2 = squeeze:600
Expected = 526d4f6cfca230e0654bf8749bddc0f4416a8a164c50f3c1b0bc1d527
2a88b9a524e73cafad76691a29c0e03a5255fd8fb9d778ef5a0c8c9e11e003011d25
6bf92dd36233e4c6c360baca0f8ac305d459adb1231a801742669efa051396e96417
814448b5328336d028a62dbddf24d1bb68496d27f1944eb24d4b2812d9ad4eae6c26
0b720c44ed2be8bfeeed3acc2640edbab987674f2cef8ceacda1e04f254170aba424
1dabc6364ed5afc09b58205682d5e8413bf5f9d97e9c799b97876ccd1c48d86759ad
e5871acc4c5d41d37f2b1843c8b6f9e0bade78342d56f9b1e8232d4c7553674d889e
69fe24dea31f42f0b02b70161876ceb12cc0b36868c262cbebb5e815a1eceaee97ae
d3402a518287c32f2f469c3a38a17afd0f0d82433acf695ae143ded9412b4e6b6144
bd6d4be6bb7de33c05f560480c63aa89336954f1cf5992399e6ed59d406adb4497bb
88aa897fd3d65646cf86e796da4f193c418a74d662f57e0e0c775386abdace02157e
519ba54495555145016c550ff32004981d0e34f0abe7d814ac4fe25260473ffa8746
0a736f20954e8d3b9f16140e79451953fe6cfc222cba6ad4f85a2e2efd6ff8f5fef6
5d8480e6af40baab298c4de57f30d08a5e1b4c10d123a5af7702ff26ba9a84a6fe92
f48391b23a7e8e8cb06deda74d1b10870611995f6bfe4df60320a0b7f2c891cad5a5
645ecec80868ed568591a74dafb35cabb42dae1a1085269b655db1ebf09929f63d5a
f775a24e43759f673b83aeefef382bc2b7bf175bb9d90e77911466ffb3b230754776
5cd5adc30a6b07881a88fd1511e5f8d2dcc4347c076e6c79676d8df
```

Author’s Address

Michele Orrテケ
CNRS
Email: m@orru.net