CFRG
Internet-Draft
Intended status: Informational
Expires: 12 July 2026

T. Looker
V. Kalos
MATTR
A. Whitehead
Portage
M. Lodder
CryptID

8 January 2026

# The BBS Signature Scheme
# draft-irtf-cfrg-bbs-signatures-10

## Abstract

This document describes the BBS Signature scheme, a secure, multi-message digital signature protocol, supporting proving knowledge of a signature while selectively disclosing any subset of the signed messages. Concretely, the scheme allows for signing multiple messages whilst producing a single, constant size, digital signature. Additionally, the possessor of a BBS signatures is able to create zero-knowledge, proofs of knowledge of a signature, while selectively disclosing subsets of the signed messages. Being zero-knowledge, the BBS proofs do not reveal any information about the undisclosed messages or the signature itself, while at the same time, guaranteeing the authenticity and integrity of the disclosed messages.

## Discussion Venues

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

Source for this draft and an issue tracker can be found at https://github.com/decentralized-identity/bbs-signature.

## Status of This Memo

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

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

## Copyright Notice



## Table of Contents

## 1. Introduction

A digital signature scheme is a fundamental cryptographic primitive that is used to provide data integrity and verifiable authenticity in various protocols. The core premise of digital signature technology is built upon asymmetric cryptography where-by the possessor of a private key is able to sign a message, where anyone in possession of the corresponding public key matching that of the private key is able to verify the signature.

Beyond the core properties of a digital signature scheme, the BBS signatures and proofs provide multiple additional unique properties. Three key ones are:

*Selective Disclosure* - The scheme allows a Signer to sign multiple messages and produce a single -constant size- output signature. A Prover then possessing the messages and the signature can generate a proof whereby they can choose which messages to disclose, while revealing no information about the undisclosed messages. The proof itself guarantees the integrity and authenticity of the disclosed messages (e.g. that they were originally signed by the Signer).

*Unlinkable Proofs* - The proofs generated by the scheme are zero-knowledge, proofs of knowledge of the signature, meaning a verifying party in receipt of a proof is unable to determine which signature was used to generate the proof, removing a common source of correlation. In general, the value of a BBS proof is indistinguishable from random even if generated from the same signature.

*Proof of Possession* - The proofs generated by the scheme prove to a Verifier that the party who generated the proof (Prover) was in possession of a signature without revealing it. The scheme also supports binding a presentation_header to the generated proof, which acts as a message signed by the Prover. The presentation_header can include arbitrary information such as a cryptographic nonce, an audience/domain identifier and or time based validity information (for more details on the presentation_header, see Section 3.3.6).

Refer to the Appendix C for an elaboration on situations where these properties are useful.

Below is a basic diagram describing the main entities involved in the scheme

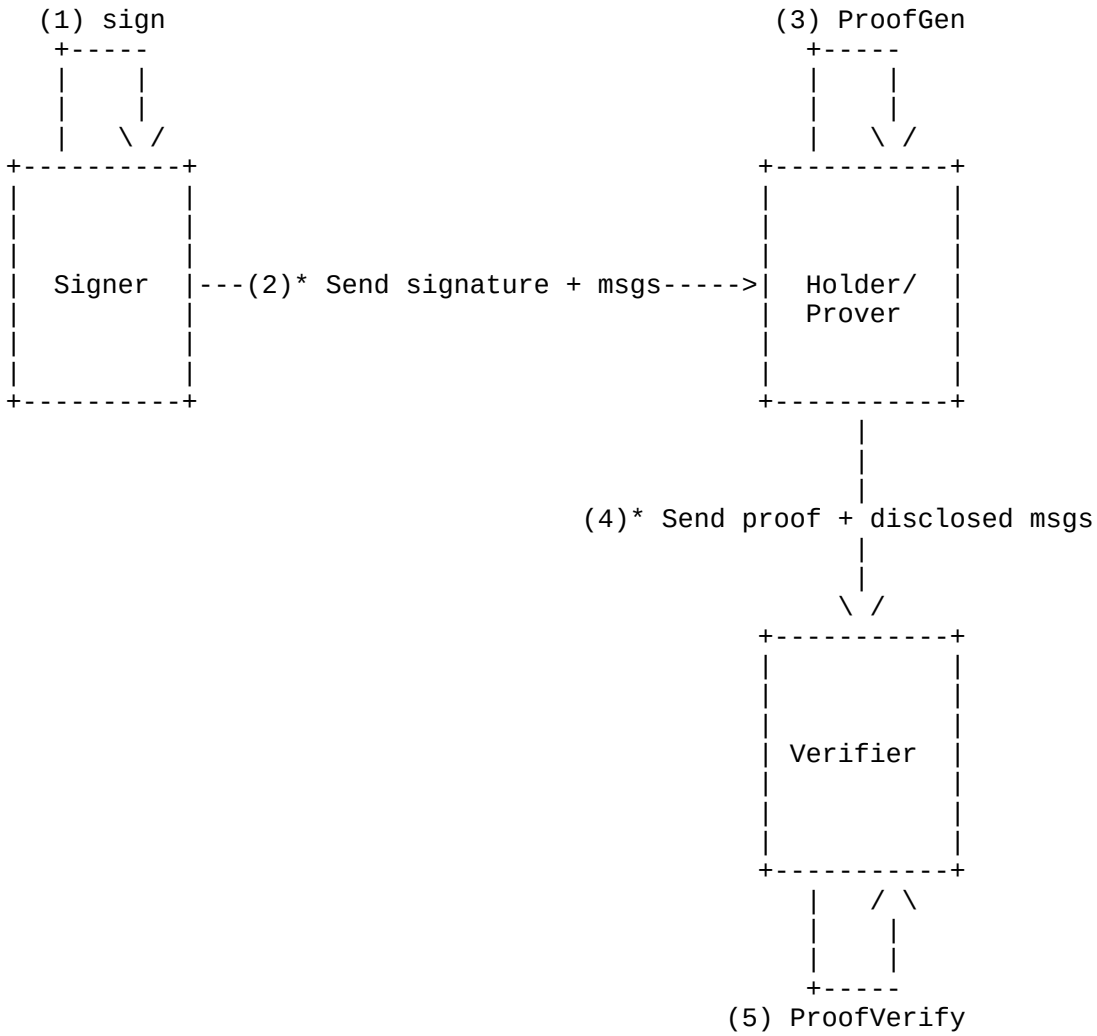


Figure 1: Basic diagram capturing the main entities involved in using the scheme

*Note* The protocols implied by the items annotated by an asterisk are out of scope for this specification

The name BBS is derived from the authors of the original academic work by Dan Boneh, Xavier Boyen, and Hovav Shacham [BBS04], where the scheme was first described as part of a group signatures protocol. Soon after, the scheme was described by Camenisch and Lysyanskaya as a stand-alone signatures scheme in [CL04], for anonymous credentials applications. Later, Au, Susilo an Mu presented the first, provably

secure version of BBS Signatures in [ASM06]. Following, works by Camenisch, Drijvers and Lehmann [CDL16] and by Barki, Brunet, Desmoulins and Traore [BBDT16], proved the security of the scheme in settings where more efficient computations are possible, thereby improving performance. Finally, in 2023, Tessaro and Zhu, presented in [TZ23] further performance improvements, shrinking the BBS signature. This document is mainly based on that work. More specifically, the BBS signature generation and verification algorithms are as defined in Section 3.1 of [TZ23] while the BBS proof generation and verification algorithms are as defined in Appendix B of the same work.

Note that the BBS Signatures scheme is based on the discrete logarithm problem. This means that it is not "post-quantum secure". However, the privacy and hiding properties of BBS proofs are resilient even against an attacker utilizing a Cryptographically Relevant Quantum Computer ([I-D.ietf-pquip-pqc-engineers]). See Section 6.9 for an elaboration on the security properties of BBS Signatures against such a computer.

## 1.1. Terminology

The following terminology is used throughout this document:

SK The secret key for the signature scheme.
PK The public key for the signature scheme.
message An octet string, representing a signed message.
L The total number of signed messages.
R The number of message indexes that are disclosed (revealed) in a proof-of-knowledge of a signature.
U The number of message indexes that are undisclosed in a proof-of-knowledge of a signature.
scalar An integer between 0 and r-1, where r is the prime order of the selected groups, defined by each ciphersuite (see also Section 1.2).
generator A valid point on the selected subgroup of the curve being used that is employed to commit a value.
signature The digital signature output.
header A payload chosen by the Signer and bound to a BBS signature, as well as the BBS proofs generated using that signature.
presentation_header (ph) A payload generated and bound to a specific BBS proof.
dst The domain separation tag.
I2OSP An operation that transforms a non-negative integer into an octet string, defined in Section 4 of [RFC8017]. The output of this operation is in big-endian order.
OS2IP An operation that transforms a octet string into an non-

negative integer, defined in Section 4 of [RFC8017]. The input of this operation must be in big-endian order.

INVALID, ABORT Error indicators. INVALID refers to an error encountered during the Deserialization or Procedure steps of an operation. An INVALID value can be returned by a subroutine and handled by the calling operation. ABORT indicates that one or more of the initial constraints defined by the operation are not met. In that case, the operation will stop execution. An operation calling a subroutine that aborted must also immediately abort.

## 1.2. Notation

The following notation and primitives are used:

a || b Denotes the concatenation of octet strings a and b.

I \ J Denotes the difference of the two sets I and J, i.e., all the elements of I that do not appear in J, in the same order as they were in I.

X[a..b] Denotes a slice of the array X containing all elements from and including the value at index a until and including the value at index b. Note when this syntax is applied to an octet string, each element in the array X is assumed to be a single byte.

length(input) Takes as input either an array or an octet string. If the input is an array, returns the number of elements of the array. If the input is an octet string, returns the number of bytes of the inputted octet string.

X[i] Denotes the element of array X at index i. Note that arrays in this document are considered "zero-indexed", meaning that element indexing starts from 0 rather than 1. For example, if X = [a, b, c, d] then X[0] = a, X[1] = b, X[2] = c and X[3] = d.

Terms specific to pairing-friendly elliptic curves that are relevant to this document are restated below, originally defined in [I-D.irtf-cfrg-pairing-friendly-curves].

E1, E2 elliptic curve groups defined over finite fields. This document assumes that E1 has a more compact representation than E2, i.e., because E1 is defined over a smaller field than E2. For a pairing-friendly curve, this document denotes operations in E1 and E2 in additive notation, i.e., P + Q denotes point addition and P * x denotes scalar multiplication, where x is a scalar.

G1, G2 subgroups of E1 and E2 (respectively) having prime order r.

GT a subgroup, of prime order r, of the multiplicative group of a field extension.

h G1 x G2 -> GT: a non-degenerate bilinear map.

r The prime order of the G1 and G2 subgroups.

BP1, BP2 base (constant) points on the G1 and G2 subgroups

respectively.

Identity_G1, Identity_G2, Identity_GT The identity element for the G1, G2, and GT subgroups respectively.

hash_to_curve_g1(ostr, dst) -> P A cryptographic hash function that takes an arbitrary octet string as input and returns a point in G1, using the hash_to_curve operation defined in [RFC9380] and the inputted dst as the domain separation tag for that operation (more specifically, the inputted dst will become the DST parameter for the hash_to_field operation, called by hash_to_curve).

expand_message(msg, dst, len) -> ostr returns a uniformly random octet string of len octets, deterministically on input a message msg to be hashed as an octet string, another octet string representing a domain separation tag dst and the number of octets to be returned (len).

point_to_octets_E1(P) -> ostr, point_to_octets_E2(P) -> ostr returns the canonical representation of the point P of the elliptic curve E1 or E2 as an octet string. This operation is also known as serialization. Note that we assume that when the point is valid, all the serialization operations will always succeed to return the octet string representation of the point.

octets_to_point_E1(ostr) -> P, octets_to_point_E2(ostr) -> P returns the point P for the respective elliptic curve corresponding to the canonical representation ostr, or INVALID if ostr is not a valid output of the respective point_to_octets_E* function. This operation is also known as deserialization.

subgroup_check_G1(P), subgroup_check_G2(P) -> VALID or INVALID returns VALID when the point P is an element of the subgroup G1 or G2 correspondingly, and INVALID otherwise. This function can always be implemented by checking that r * P is equal to the identity element. In some cases, faster checks may also exist, e.g., [Bowe19]. Note that these functions should always return VALID, on input the Identity point of the corresponding subgroup.

## 1.3. Document Organization

This document is organized as follows:

* Scheme Definition (Section 3), defines the core operations and parameters for the BBS signature scheme.

* Utility Operations (Section 4), defines utilities used by the BBS signature scheme.

* Security Considerations (Section 6), describes a set of security considerations associated to the signature scheme.

* Ciphersuites (Section 7), defines the format of a ciphersuite, alongside a concrete ciphersuite based on the BLS12-381 curve.

## 2. Conventions

The keywords MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, when they appear in this document, are to be interpreted as described in [RFC2119].

## 3. Scheme Definition

This section defines the BBS signature scheme, including the parameters required to define a concrete instantiation of the protocol.

### 3.1. Parameters

The schemes operations defined in this section depend on the following parameters:

* A pairing-friendly elliptic curve, plus associated functionality given in Section 1.2.

* A hash-to-curve suite as defined in [RFC9380], using the aforementioned pairing-friendly curve. This defines the hash_to_curve and expand_message operations, used by this document.

* get_random(n): returns a random octet string with a length of n bytes, sampled uniformly at random using a cryptographically secure pseudo-random number generator (CSPRNG) or a pseudo random function. See [RFC4086] for recommendations and requirements on the generation of random numbers.

### 3.2. Interfaces

The BBS signature scheme is organized as follows:

* A set of low level (core) operations, taking care of the main cryptographic functionality.
* An Application Interface, that uses the core operations in a secure way.

Together with a set of utility procedures, defining functionality that is common between different interface procedures or core operations, a full BBS Signatures deployment can be defined.

Each of the core operations (see Section 3.6) expects a list of points (called the generators, see Section 3.3.2) and a list of messages represented as scalar values (see Section 3.3.3). It is the job of the Interface to:

1. Create the necessary generators.
2. Map the inputted messages to scalars.

This allows for extensibility of the core scheme without exposing the resulting complexity to all applications. To ensure proper separation between BBS Interfaces with distinct functionality, each Interface is parametrized by a unique identifier (called api_id) that will be used as a domain separation tag (dst) by the core (Section 3.6) and utility (Section 4.1) procedures. A document extending the core functionality of BBS Signatures by defining a new Interface, MUST ensure that it adheres to the requirements described in Section 3.8.

## 3.3. Considerations

### 3.3.1. Subgroup Selection

For defining BBS signatures there are two possible variations regarding the subgroup selection, namely where public keys are defined in G2 and signatures in G1 OR the opposite where public keys are defined in G1 and signatures in G2. Some pairing-based digital signature schemes such as [I-D.irtf-cfrg-bls-signature] elect to allow for both variations, because they optimize for different use cases. However, in case of BBS Signatures, due to the operations involved in both signature and proof generation being computationally inefficient when performed in G2 and in the pursuit of simplicity, the BBS Signatures scheme as defined in this document is limited to a construction where public keys are in G2 and signatures in G1.

### 3.3.2. Generators

Throughout the operations of this signature scheme, each message that is signed is paired with a specific point of G1, called a generator. Specifically, if a generator H_1 is multiplied with msg_1 during signing, then H_1 MUST be multiplied with msg_1 in all other operations (signature verification, proof generation and proof verification). As a result, the messages must be passed to the operations of the BBS scheme in the same order.

Aside from the message generators, the scheme uses one additional generator Q_1 to sign the signature’s domain, which binds both the signature and generated proofs to a specific context and cryptographically protects any potential application-specific information (for example, messages that must always be disclosed, ciphersuite parameters or an application identifier). This document uses the procedures defined in [RFC9380] to create the generators. See Section 4.1.1 on more details.

#### 3.3.3. Messages

In this document, the messages to be signed are defined as octet strings. Each message must be mapped to a scalar value before passed to one of the core BBS operations (Section 3.6). There are various ways to map a message to a scalar value. The BBS Signatures Interface defined in this document (see Section 3.5), makes use of a hash function (see Section 4.1.2). See Section 4.1.2 on further details on how the each message is mapped to a scalar value and Section 6.8 for more details and guidance on using alternative mapping methods.

#### 3.3.4. Indexing of Arrays

Note that arrays in this document use the zero-based numbering common in many programming languages, meaning that element indexing starts from 0 (see Section 1.2). This is distinct from naming used during deserialization of arrays, where natural (one-based) numbering might be used as part of the names of the array's elements for clarity in that context.

For example, if X is an array of n elements, we may write,

```
[a_1, a_2, ..., a_n] = X
```

The above would indicate that

```
X[0] = a_1
X[1] = a_2
// ... and so on, up to
X[n-1] = a_n
```

#### 3.3.5. Serializing to Octets

When serializing one or more values to produce an octet string, each element will be encoded using a specific operation determined by its type. More concretely,

* Points in E* will be serialized using the point_to_octets_E* implementation for a particular ciphersuite.
* Non-negative integers will be serialized using I2OSP with an output length of 8 bytes.
* Scalars will be serialized using I2OSP with a constant output length defined by a particular ciphersuite.

We also use strings in double quotes to represent ASCII-encoded literals. For example "BBS" will be used to refer to the octet string, 010000100100001001010011.

Those rules will be used explicitly on every operation. See also serialize defined in Section 4.2.4.1.

### 3.3.6. Header and Presentation Header Usage

There are two special values defined by the BBS Scheme; the header and the presentation_header. The header value is chosen by the Signer and is bound to both a BBS signature and to any BBS proofs generated using that signature. Specifically, the Prover is required to reveal the header to the proof Verifier during every BBS proof presentation. As a result, the Signer SHOULD NOT include in the header any identifying information that may have the potential of compromising the Prover's privacy (see Section 5). Suitable use cases taking advantage of the header value include binding a BBS signature (and subsequent BBS proofs) to a specific application, deployment or domain, (in general, binding the signature to specific sets of metadata).

Similarly, the Prover can choose a presentation_header value to be bound to the BBS proof (in contrast to the header value that is chosen by the Signer and is bound to both BBS proof and signature). Verifying a BBS proof will guarantee the authenticity and integrity of the presentation_header value. This makes it suitable for ensuring the freshness of a BBS proof, for example, by including in it a (possibly supplied by the Verifier) random value. Other use cases include binding the BBS proof to a certain domain/audience or validity period. The presentation_header can also be used by the Prover to sign a message. In this case, the Prover will add to the presentation_header the message they want to sign. A valid BBS proof guarantees that the message contained in the presentation_header was signed by the same Prover that generated that proof (similar to how group signatures work [BBS04], where the group in this case will be all the Provers having received valid signatures under a specific public key).

### 3.3.7. Unlinkability

As mentioned in the Introduction, a BBS proof is unlinkable. In this section we will define the term in more detail. Formally, we use unlinkability to refer to the fact that a BBS proof is zero-knowledge [TZ23]. In practice, this guarantees that an adversary (a Verifier, the Issuer or coalitions between one or more Verifiers and the Issuer) will not be able to infer any information from the BBS proof value, other than what the Prover decided to provide, even with access to multiple proof values. Consequently, the Verifier will not be able to correlate multiple proofs generated by the same signature or Prover. Note however, that this holds only for the value of the BBS proof. In other words, other values revealed by the Prover

during their interaction with a Verifier, may still be used to correlate their activity and compromise their privacy. Examples of such values include the disclosed messages (if the same message of high enough entropy is revealed between multiple proofs), the header and presentation_header values (see Section Section 3.3.6), or the total number of signed messages. See Section Section 5 for privacy considerations and recommendations on minimizing these sources of correlation.

## 3.4. Key Generation Operations

### 3.4.1. Secret Key

This operation generates a secret key (SK) deterministically from a secret octet string (key_material). This operation is the RECOMMENDED way of generating a secret key, but its use is not required for compatibility, and implementations MAY use a different key generation procedure. For security, such an alternative MUST output a secret key that is statistically close to uniformly random in the range from 1 to r - 1. An example of an HKDF-based alternative is the KeyGen operation defined in Section 2.3 of [I-D.irtf-cfrg-bls-signature] (with an appropriate, BBS specific, salt value, like "BBS_SIG_KEYGEN_SALT_").

For security, key_material MUST be random and infeasible to guess, e.g. generated by a trusted source of randomness and with enough entropy. See [RFC4086] for suggestions on generating randomness. key_material MUST be at least 32 bytes long, but it MAY be longer.

KeyGen takes an optional input, key_info. This parameter MAY be used to derive distinct keys from the same key material.

Because KeyGen is deterministic, implementations MAY choose either to store the resulting SK or to store key_material and key_info and call KeyGen to derive SK when necessary.

```
SK = KeyGen(key_material, key_info, key_dst)

Inputs:

- key_material (REQUIRED), a secret octet string. See requirements
                           above.
- key_info (OPTIONAL), an octet string. Defaults to an empty string if
                       not supplied.
- key_dst (OPTIONAL), an octet string representing the domain separation
                      tag. Defaults to the octet string
                      ciphersuite_id || "KEYGEN_DST_" if not supplied.

Outputs:

- SK, a uniformly random integer such that 0 < SK < r.

Procedure:

1. if length(key_material) < 32, return INVALID
2. if length(key_info) > 65535, return INVALID
3. derive_input = key_material || I2OSP(length(key_info), 2) || key_info
4. SK = hash_to_scalar(derive_input, key_dst)
5. if SK is INVALID, return INVALID
6. return SK
```

#### 3.4.2. Public Key

This operation takes a secret key (SK) and outputs a corresponding public key (PK).

```
PK = SkToPk(SK)

Inputs:

- SK (REQUIRED), a secret integer such that 0 < SK < r.

Outputs:

- PK, a public key encoded as an octet string.

Procedure:

1. W = SK * BP2
2. return point_to_octets_E2(W)
```

### 3.5. BBS Signatures Interface

This section defines a BBS Signatures Interface (see Section 3.2), that makes use of the core operations defined in Section 3.6, to perform the functions of signing and verifying the signature, as well as generating and validating the BBS proof. To create the generators (see Section 3.3.2) it uses the create_generators operation defined in Section 4.1.1. Each input message is an octet string (see Section 3.3.3). To map the messages to scalars, it uses the messages_to_scalars operation defined in Section 4.1.2. Generated signatures and proofs may optionally be bound to a header value. A BBS proof may additionally be bound to a presentation_header value. See Section 3.3.6 for more details on the header and presentation_header usage.

The api_id parameter for this Interface is defined as,

api_id = ciphersuite_id || "H2G_HM2S_"

where ciphersuite_id is defined by the ciphersuite and "H2G_HM2S_" is an ASCII string comprised of 9 bytes, wherein "H2G_" refers to the identifier of the create_generators operation used (see Section 4.1.1) and "HM2S_" is the identifier of the used messages_to_scalars mapping (see Section 4.1.2).

#### 3.5.1. Signature Generation (Sign)

The Sign operation returns a BBS signature from a secret key (SK), over a header and a set of messages.

```
signature = Sign(SK, PK, header, messages)

Inputs:

- SK (REQUIRED), a secret key in the form outputted by the KeyGen
                 operation.
- PK (REQUIRED), an octet string of the form outputted by SkToPk
                 provided the above SK as input.
- header (OPTIONAL), an octet string containing context and application
                     specific information. If not supplied, it defaults
                     to the empty octet string ("").
- messages (OPTIONAL), a vector of octet strings. If not supplied, it
                       defaults to the empty array ("()").

Parameters:

- api_id, the octet string ciphersuite_id || "H2G_HM2S_", where
          ciphersuite_id is defined by the ciphersuite and "H2G_HM2S_"
          is an ASCII string comprised of 9 bytes.

Outputs:

- signature, a signature encoded as an octet string; or INVALID.

Procedure:

1. message_scalars = messages_to_scalars(messages, api_id)
2. generators = create_generators(length(messages)+1, api_id)

3. signature = CoreSign(SK, PK, generators, header, message_scalars,
                                                                api_id)
4. if signature is INVALID, return INVALID
5. return signature
```

### 3.5.2. Signature Verification (Verify)

The Verify operation validates a BBS signature, given a public key (PK), a header and a set of messages.

```
result = Verify(PK, signature, header, messages)

Inputs:

- PK (REQUIRED), an octet string of the form outputted by the SkToPk
                 operation.
- signature (REQUIRED), an octet string of the form outputted by the
                        Sign operation.
- header (OPTIONAL), an octet string containing context and application
                     specific information. If not supplied, it defaults
                     to the empty octet string ("").
- messages (OPTIONAL), a vector of octet strings. If not supplied, it
                       defaults to the empty array ("()").

Parameters:

- api_id, the octet string ciphersuite_id || "H2G_HM2S_", where
          ciphersuite_id is defined by the ciphersuite and "H2G_HM2S_"
          is an ASCII string comprised of 9 bytes.

Outputs:

- result, either VALID or INVALID.

Procedure:

1. message_scalars = messages_to_scalars(messages, api_id)
2. generators = create_generators(length(messages)+1, api_id)

3. result = CoreVerify(PK, signature, generators, header,
                                          message_scalars, api_id)
4. return result
```

#### 3.5.3. Proof Generation (ProofGen)

The ProofGen operation creates a BBS proof, which is a zero-knowledge, proof-of-knowledge of a BBS signature, while optionally disclosing any subset of the signed messages. Validating the proof (see ProofVerify defined in Section 3.5.4) guarantees authenticity and integrity of the header and disclosed messages, as well as knowledge of a valid BBS signature.

Other than the Signer’s public key (PK), the BBS signature and the signed header and messages, the operation also accepts a presentation_header value. That value, chosen by the Prover, will also be integrity protected (signed) by the resulting proof (see Section 3.3.6). Finally, to indicate which of the messages should be disclosed, the operation accepts a list of integers in ascending order, representing the indexes of those messages.

```
proof = ProofGen(PK, signature, header, ph, messages, disclosed_indexes)

Inputs:

- PK (REQUIRED), an octet string of the form outputted by the SkToPk
                 operation.
- signature (REQUIRED), an octet string of the form outputted by the
                        Sign operation.
- header (OPTIONAL), an octet string containing context and application
                     specific information. If not supplied, it defaults
                     to the empty octet string ("").
- ph (OPTIONAL), an octet string containing the presentation_header. If
                 not supplied, it defaults to the empty octet
                 string ("").
- messages (OPTIONAL), a vector of octet strings. If not supplied, it
                       defaults to the empty array ("()").
- disclosed_indexes (OPTIONAL), vector of unsigned integers in ascending
                                order. Indexes of disclosed messages. If
                                not supplied, it defaults to the empty
                                array ("()").

Parameters:

- api_id, the octet string ciphersuite_id || "H2G_HM2S_", where
          ciphersuite_id is defined by the ciphersuite and "H2G_HM2S_"
          is an ASCII string comprised of 9 bytes.

Outputs:

- proof, an octet string; or INVALID.

Procedure:

1. message_scalars = messages_to_scalars(messages, api_id)
2. generators = create_generators(length(messages) + 1, api_id)

3. proof = CoreProofGen(PK, signature, generators, header, ph,
                             message_scalars, disclosed_indexes, api_id)
4. if proof is INVALID, return INVALID
5. return proof
```

### 3.5.4. Proof Verification (ProofVerify)

The ProofVerify operation validates a BBS proof, given the Signer's public key (PK), a header and presentation_header values, the disclosed messages and the indexes those messages had in the original vector of signed messages.

```
result = ProofVerify(PK, proof, header, ph,
                     disclosed_messages,
                     disclosed_indexes)
```

Inputs:

- PK (REQUIRED), an octet string of the form outputted by the SkToPk operation.
- proof (REQUIRED), an octet string of the form outputted by the ProofGen operation.
- header (OPTIONAL), an optional octet string containing context and application specific information. If not supplied, it defaults to the empty octet string ("").
- ph (OPTIONAL), an octet string containing the presentation_header. If not supplied, it defaults to the empty octet string ("").
- disclosed_messages (OPTIONAL), a vector of octet strings. If not supplied, it defaults to the empty array ("()").
- disclosed_indexes (OPTIONAL), vector of unsigned integers in ascending order. Indexes of disclosed messages. If not supplied, it defaults to the empty array ("()").

Parameters:

- api_id, the octet string ciphersuite_id || "H2G_HM2S_", where ciphersuite_id is defined by the ciphersuite and "H2G_HM2S_" is an ASCII string comprised of 9 bytes.
- (octet_point_length, octet_scalar_length), defined by the ciphersuite.

Outputs:

- result, either VALID or INVALID.

Deserialization:

1. proof_len_floor = 3 * octet_point_length + 4 * octet_scalar_length
2. if length(proof) < proof_len_floor, return INVALID
3. U = floor((length(proof) - proof_len_floor) / octet_scalar_length)
4. R = length(disclosed_indexes)

Procedure:

```
1. message_scalars = messages_to_scalars(disclosed_messages, api_id)
2. generators = create_generators(U + R + 1, api_id)

3. result = CoreProofVerify(PK, proof, generators, header, ph,
                            message_scalars, disclosed_indexes, api_id)
4. return result
```

### 3.6. Core Operations

The operations defined in this section perform the low-level cryptographic functionality of BBS Signatures. Those core functions MUST only be invoked by an Application Interface that conform to the requirements outlined in Section 3.8.

The operations of this section make use of functions and sub-routines defined in Utility Operations (#utility-operations). More specifically,

* hash_to_scalar is defined in Section 4.2.2
* calculate_domain is defined in Section 4.2.3.
* serialize, signature_to_octets, octets_to_signature, proof_to_octets, octets_to_proof and octets_to_pubkey are defined in Section 4.2.4.
* h is the pairing operation used (see Section 1.2), defined as part of the ciphersuite.

Each core operation will accept a vector of generators (points of G1) and optionally, a vector of messages. The generators MUST be unique and pseudo-random i.e., with no known relationship to each other. See Section 4.1.1.1 for more details. Each message is represented as a scalar value. See Section 4.1.2 for ways to map a message to a scalar and the corresponding security requirements.

Furthermore, all core operations accept the Signer’s public key (PK) as well as an optional octet string representing an Interface identifier (api_id).

*Note* Some of the utility functions used by the core operations of this section could fail (ABORT). In that case, the calling operation MUST also immediately abort.

#### 3.6.1. CoreSign

This operation computes a deterministic signature from a secret key (SK), a set of generators (points of G1) and optionally a header and a vector of messages. Note that signature generation is deterministic, in contrast to the academic literature, where signature generation, and more specifically the calculation of the e value (Procedure step 2 below), is randomized (i.e., the e value is drawn at random, instead of been deterministically calculated by hashing the Signer's secret key and the list of messages). This alteration protects the scheme (at least the signature generation part) from vulnerabilities related to bad entropy sources, as well as some of the the best currently known attacks, as suggested in [TZ23]. Additionally, it makes testing of the CoreSign operation easier, as it avoids the need for a mocked random scalar.

```
signature = CoreSign(SK, PK, generators, header, messages, api_id)
```

Inputs:

- SK (REQUIRED), a secret key in the form outputted by the KeyGen operation.
- PK (REQUIRED), an octet string of the form outputted by SkToPk provided the above SK as input.
- generators (REQUIRED), vector of pseudo-random points in G1.
- header (OPTIONAL), an octet string containing context and application specific information. If not supplied, it defaults to the empty octet string ("").
- messages (OPTIONAL), a vector of scalars representing the messages. If not supplied, it defaults to the empty array ("()").
- api_id (OPTIONAL), an octet string. If not supplied it defaults to the empty octet string ("").

Parameters:

- P1, fixed point of G1, defined by the ciphersuite.

Outputs:

- signature, a vector comprised of a point of G1 and a scalar.

Definitions:

1. hash_to_scalar_dst, an octet string representing the domain separation tag: api_id || "H2S_" where "H2S_" is an ASCII string comprised of 4 bytes.

Deserialization:

```
1. L = length(messages)
2. if length(generators) != L + 1, return INVALID
3. (msg_1, ..., msg_L) = messages
4. (Q_1, H_1, ..., H_L) = generators
```

Procedure:

```
1. domain = calculate_domain(PK, Q_1, (H_1, ..., H_L), header, api_id)

2. e = hash_to_scalar(serialize((SK, msg_1, ..., msg_L, domain)),
                                                    hash_to_scalar_dst)
3. B = P1 + Q_1 * domain + H_1 * msg_1 + ... + H_L * msg_L
4. A = B * (1 / (SK + e))
5. return signature_to_octets((A, e))
```

*Note* When computing step 4 of the above procedure there is an extremely small probability (around 2^(-r)) that the condition (SK + e) = 0 mod r will be met. How implementations evaluate the inverse of the scalar value 0 may vary, with some returning an error and others returning 0 as a result. If the returned value from the inverse operation 1/(SK + e) does evaluate to 0 the value of A will equal Identity_G1 thus an invalid signature. Implementations MAY elect to check (SK + e) = 0 mod r prior to step 4, and or A != Identity_G1 after step 4 to prevent the production of invalid signatures.

### 3.6.2. CoreVerify

This operation checks that a signature is valid for a given set of generators, header and vector of messages, against a supplied public key (PK). The set of messages MUST be supplied in this operation in the same order they were supplied to CoreSign (Section 3.6.1) when creating the signature.

```
result = CoreVerify(PK, signature, generators, header, messages, api_id)

Inputs:

- PK (REQUIRED), an octet string of the form outputted by the SkToPk
                 operation.
- signature (REQUIRED), an octet string of the form outputted by the
                        Sign operation.
- generators (REQUIRED), vector of pseudo-random points in G1.
- header (OPTIONAL), an octet string containing context and application
                     specific information. If not supplied, it defaults
                     to the empty octet string ("").
- messages (OPTIONAL), a vector of scalars representing the messages.
                       If not supplied, it defaults to the empty
                       array ("()").
- api_id (OPTIONAL), an octet string. If not supplied it defaults to the
                     empty octet string ("").

Parameters:

- P1, fixed point of G1, defined by the ciphersuite.

Outputs:

- result, either VALID or INVALID.

Deserialization:

1. signature_result = octets_to_signature(signature)
2. if signature_result is INVALID, return INVALID
3. (A, e) = signature_result
4. W = octets_to_pubkey(PK)
5. if W is INVALID, return INVALID
6. L = length(messages)
7. if length(generators) != L + 1, return INVALID
8. (msg_1, ..., msg_L) = messages
9. (Q_1, H_1, ..., H_L) = generators

Procedure:

1. domain = calculate_domain(PK, Q_1, (H_1, ..., H_L), header, api_id)
2. B = P1 + Q_1 * domain + H_1 * msg_1 + ... + H_L * msg_L
3. if h(A, W) * h(A * e - B, BP2) != Identity_GT, return INVALID
4. return VALID
```

### 3.6.3. CoreProofGen

This operation computes a zero-knowledge proof-of-knowledge of a signature, while optionally selectively disclosing from the original set of signed messages. The Prover may also supply a presentation_header (denoted as ph on the input definitions of the CoreProofGen operation). See Section 3.3.6 for more details. Validating the resulting proof (using the CoreProofVerify algorithm defined in Section 3.6.4), guarantees the integrity and authenticity of the revealed messages, as well as the possession of a valid signature (for the public key PK) by the Prover. See Appendix E for a high level explanation on the inner-workings of the algorithm.

The CoreProofGen operation will accept that signature as an input. It is RECOMMENDED to validate that signature, using the inputted public key PK and generators set, against the supplied messages and header, with the CoreVerify operation defined in Section 3.6.2.

The messages supplied in this operation MUST be in the same order as when supplied to CoreSign (Section 3.6.1). To specify which of those messages will be disclosed, the Prover can supply the list of indexes (disclosed_indexes) that the disclosed messages have in the array of signed messages. Each element in disclosed_indexes MUST be a non-negative integer, in the range from 0 to length(messages) - 1.

The operation works by first calculating a set of random scalars using the calculate_random_scalars operation defined in Section 4.2.1, utilized to blind the signature and the undisclosed messages (see Section 6.7 for considerations and requirements on random scalars generation). It then initializes the proof using the ProofInit subroutine defined in Section 3.7.1. The result will be passed to the challenge calculation operation (ProofChallengeCalculate, defined in Section 3.7.4). The outputted challenge, together with the initialization result, will be used by the ProofFinalize subroutine defined in Section 3.7.2, which will return the proof value.

```
proof = CoreProofGen(PK, signature, generators, header, ph, messages,
                                                 disclosed_indexes, api_id)
```

Inputs:

- PK (REQUIRED), an octet string of the form outputted by the SkToPk operation.
- signature (REQUIRED), an octet string of the form outputted by the Sign operation.
- generators (REQUIRED), vector of pseudo-random points in G1.
- header (OPTIONAL), an octet string containing context and application

```
                    specific information. If not supplied, it defaults
                    to the empty octet string ("").
- ph (OPTIONAL), an octet string containing the presentation_header. If
                 not supplied, it defaults to the empty octet
                 string ("").
- messages (OPTIONAL), a vector of scalars representing the messages.
                       If not supplied, it defaults to the empty
                       array ("()").
- disclosed_indexes (OPTIONAL), vector of non-negative integers in
                                ascending order. Indexes of disclosed
                                messages. If not supplied, it defaults
                                to the empty array ("()").
- api_id (OPTIONAL), an octet string. If not supplied it defaults to the
                     empty octet string ("").

Outputs:

- proof, an octet string; or INVALID.

Deserialization:

1.  signature_result = octets_to_signature(signature)
2.  if signature_result is INVALID, return INVALID
3.  (A, e) = signature_result

4.  L = length(messages)
5.  R = length(disclosed_indexes)
6.  if R > L, return INVALID
7.  U = L - R
8.  for i in disclosed_indexes, if i < 0 or i > L - 1, return INVALID
9.  undisclosed_indexes = (0, 1, ..., L - 1) \ disclosed_indexes
10. (i1, ..., iR) = disclosed_indexes
11. (j1, ..., jU) = undisclosed_indexes

12. disclosed_messages = (messages[i1], ..., messages[iR])
13. undisclosed_messages = (messages[j1], ..., messages[jU])

Procedure:

1. random_scalars = calculate_random_scalars(5+U)
2. init_res = ProofInit(PK,
                        signature_result,
                        generators,
                        random_scalars,
                        header,
                        messages,
                        undisclosed_indexes,
                        api_id)
```

```
3. if init_res is INVALID, return INVALID
4. challenge = ProofChallengeCalculate(init_res, disclosed_messages,
                                                 disclosed_indexes,
                                                 ph,
                                                 api_id)
5. if challenge is INVALID, return INVALID
6. proof = ProofFinalize(init_res, challenge, e, random_scalars,
                                                   undisclosed_messages)
7. return proof
```

#### 3.6.4. CoreProofVerify

This operation checks that a proof is valid for a header, vector of disclosed messages (disclosed_messages) along side their index corresponding to their original position when signed (disclosed_indexes) and presentation_header (denoted as ph on the input definitions of the CoreProofVerify operation) against a public key (PK).

The inputted disclosed messages (disclosed_messages) MUST be supplied to this operation in the same order as they had as part of the messages input of the CoreSign operation defined in Section 3.6.1. Similarly, the indexes of the disclosed messages (disclosed_indexes) MUST be the same and in the same order as the disclosed_indexes input of CoreProofGen (Section 3.6.3). Failure to comply with these requirements will result to the proof verification procedure returning INVALID.

The operation works by first initializing the proof verification procedure using the ProofVerifyInit subroutine defined in Section 3.7.3. The result will be inputted to the challenge calculation operation (ProofChallengeCalculate, defined in Section 3.7.4). The resulting challenge and the two first components of the received proof (points of G1) will be checked for correctness (steps 5 and 6 in the following procedure), to verify the proof.

```
result = CoreProofVerify(PK, proof, generators, header, ph,
                         disclosed_messages, disclosed_indexes, api_id)
```

Inputs:

- PK (REQUIRED), an octet string of the form outputted by the SkToPk operation.
- proof (REQUIRED), an octet string of the form outputted by the ProofGen operation.
- generators (REQUIRED), vector of pseudo-random points in G1.
- header (OPTIONAL), an optional octet string containing context and application specific information. If not supplied,

```
                    it defaults to the empty octet string ("").
- ph (OPTIONAL), an octet string containing the presentation_header. If
                 not supplied, it defaults to the empty octet
                 string ("").
- disclosed_messages (OPTIONAL), a vector of scalars representing the
                                 messages. If not supplied, it defaults
                                 to the empty array ("()").
- disclosed_indexes (OPTIONAL), vector of non-negative integers in
                                ascending order. Indexes of disclosed
                                messages. If not supplied, it defaults
                                to the empty array ("()").
- api_id (OPTIONAL), an octet string. If not supplied it defaults to the
                     empty octet string ("").
```

Parameters:

- P1, fixed point of G1, defined by the ciphersuite.

Outputs:

- result, either VALID or INVALID.

Deserialization:

```
1. proof_result = octets_to_proof(proof)
2. if proof_result is INVALID, return INVALID
3. (Abar, Bbar, D, e^, r1^, r3^, commitments, cp) = proof_result
4. W = octets_to_pubkey(PK)
5. if W is INVALID, return INVALID
```

Procedure:

```
1. init_res = ProofVerifyInit(PK, proof_result, generators, header,
                                                 disclosed_messages,
                                                 disclosed_indexes,
                                                 api_id)
2. if init_res is INVALID, return INVALID
3. challenge = ProofChallengeCalculate(init_res, disclosed_messages,
                                       disclosed_indexes, ph, api_id)
4. if challenge is INVALID, return INVALID
5. if cp != challenge, return INVALID
6. if h(Abar, W) * h(Bbar, -BP2) != Identity_GT, return INVALID
7. return VALID
```

## 3.7. Proof Protocol Subroutines

This section describes the subroutines used by the CoreProofGen (Section 3.6.3) and CoreProofVerify (Section 3.6.4) operations. See Appendix E, for a high-level intuitive overview of the procedure used to generate and verify a BBS proof.

### 3.7.1. Proof Initialization

This operation initializes the proof and returns one of the inputs passed to the challenge calculation operation (i.e., ProofChallengeCalculate, Section 3.7.4), during the CoreProofGen operation defined in Section 3.6.3.

The inputted messages MUST be supplied to this operation in the same order they had when inputted to the CoreSign operation (Section 3.6.1).

The defined procedure needs the messages the Prover decided to not disclose. For this purpose, along the list of signed messages, the operation also accepts a set of integers in the range from 0 to length(messages) - 1 (inclusive) in ascending order, representing the indexes of the undisclosed messages (undisclosed_indexes). To blind the inputted signature and the undisclosed messages, the operation will also accept a set of uniformly random scalars (random_scalars). This set must have exactly 5 more items than the list of undisclosed indexes (i.e., it must hold that length(random_scalars) = length(undisclosed_indexes) + 5).

This operation makes use of the calculate_domain function defined in Section 4.2.3.

```
init_res = ProofInit(PK, signature, generators, random_scalars,
                          header, messages, undisclosed_indexes, api_id)
```

Inputs:

- PK (REQUIRED), an octet string of the form outputted by the SkToPk operation.
- signature (REQUIRED), vector representing a BBS signature, consisting of a point of G1 and a scalar, in that order.
- generators (REQUIRED), vector of points in G1.
- random_scalars (REQUIRED), vector of scalar values.
- header (OPTIONAL), octet string. If not supplied it defaults to the empty octet string ("").
- messages (OPTIONAL), vector of scalar values. If not supplied, it defaults to the empty array ("()").
- undisclosed_indexes (OPTIONAL), vector of non-negative integers in

ascending order. If not supplied, it defaults to the empty array ("()").
- api_id (OPTIONAL), an octet string. If not supplied it defaults to the empty octet string ("").

Parameters:

- P1, fixed point of G1, defined by the ciphersuite.

Outputs:

- init_res, vector consisting of 5 points of G1 and a scalar, in that order; or INVALID.

Deserialization:

```
1.  (A, e) = signature
2.  L = length(messages)
3.  U = length(undisclosed_indexes)
4.  (j1, ..., jU) = undisclosed_indexes
5.  if length(random_scalars) != U + 5, return INVALID
6.  (r1, r2, e~, r1~, r3~, m~_j1, ..., m~_jU) = random_scalars
7.  (msg_1, ..., msg_L) = messages

8.  if length(generators) != L + 1, return INVALID
9.  (Q_1, MsgGenerators) = generators
10. (H_1, ..., H_L) = MsgGenerators
11. (H_j1, ..., H_jU) = (MsgGenerators[j1], ..., MsgGenerators[jU])
```

ABORT if:

```
1. for i in undisclosed_indexes, i < 0 or i > L - 1
2. U > L
```

Procedure:

```
1. domain = calculate_domain(PK, Q_1, (H_1, ..., H_L), header, api_id)

2. B = P1 + Q_1 * domain + H_1 * msg_1 + ... + H_L * msg_L
3. D = B * r2
4. Abar = A * (r1 * r2)
5. Bbar = D * r1 - Abar * e

6. T1 = Abar * e~ + D * r1~
7. T2 = D * r3~ + H_j1 * m~_j1 + ... + H_jU * m~_jU

8. return (Abar, Bbar, D, T1, T2, domain)
```

#### 3.7.2. Proof Finalization

This operation finalizes the proof calculation during the CoreProofGen operation defined in Section 3.6.3 and returns the serialized proof value.

As inputs, this operation accepts the proof initialization result as returned by the ProofInit operation defined in Section 3.7.1 (init_res) as well as a scalar value representing the proof's challenge as calculated by the ProofChallengeCalculate operation defined in Section 3.7.4. It also requires the scalar part of the BBS signature (e_value), the random scalars used to generate the proof (random_scalars, as inputted to the ProofInit operation) and a set of scalars, representing the messages the Prover decided to not disclose (undisclosed_messages). Those messages MUST be supplied to this operation in the same order as they had as part of the messages input of the CoreSign operation (Section 3.6.1).

This operation makes use of the proof_to_octets function defined in Section 4.2.4.4.

```
proof = ProofFinalize(init_res, challenge, e_value, random_scalars,
                                                    undisclosed_messages)
```

Inputs:

- init_res (REQUIRED), vector representing the value returned after initializing the proof generation or verification operations, consisting of 5 points of G1 and a scalar value, in that order.
- challenge (REQUIRED), scalar value.
- e_value (REQUIRED), scalar value.
- random_scalars (REQUIRED), vector of scalar values.
- undisclosed_messages (OPTIONAL), vector of scalar values. If not supplied, it defaults to the empty array ("()").

Outputs:

- proof, an octet string; or INVALID.

Deserialization:

```
1. U = length(undisclosed_messages)
2. if length(random_scalars) != U + 5, return INVALID
3. (r1, r2, e~, r1~, r3~, m~_j1, ..., m~_jU) = random_scalars
4. (undisclosed_1, ..., undisclosed_U) = undisclosed_messages
5. (Abar, Bbar, D) = (init_res[0], init_res[1], init_res[2])
```

Procedure:

```
1. r3 = r2^-1 (mod r)

2. e^ = e~ + e_value * challenge
3. r1^ = r1~ - r1 * challenge
4. r3^ = r3~ - r3 * challenge
5. for j in (1, ..., U): m^_j = m~_j + undisclosed_j * challenge (mod r)

6. proof = (Abar, Bbar, D, e^, r1^, r3^, (m^_j1, ..., m^_jU), challenge)
7. return proof_to_octets(proof)
```

### 3.7.3. Proof Verification Initialization

This operation initializes the proof verification operation and returns part of the input that will be passed to the challenge calculation operation (i.e., ProofChallengeCalculate, Section 3.7.4), during the CoreProofVerify operation defined in Section 3.6.4.

Note that, the scalars representing the disclosed messages (disclosed_messages) MUST be supplied to this operation in the same order as they had as part of the messages input of the CoreSign operation defined in Section 3.6.1 (otherwise, proof verification will fail). Similarly, the indexes of the disclosed messages in the set of signed messages MUST be supplied to this operation as a set of integers in accenting order (disclosed_indexes).

This operation makes use of the calculate_domain function defined in Section 4.2.3.

```
init_res = ProofVerifyInit(PK,
                           proof,
                           generators,
                           header,
                           disclosed_messages,
                           disclosed_indexes,
                           api_id)
```

Inputs:

- PK (REQUIRED), an octet string of the form outputted by the SkToPk operation.
- proof (REQUIRED), vector representing a BBS proof, consisting of 3 points of G1, 3 scalars, another nested but possibly empty vector of scalars and another scalar, in that order.
- generators (REQUIRED), vector of points in G1.
- header (OPTIONAL), octet string. If not supplied it defaults to the empty octet string ("").
- disclosed_messages (OPTIONAL), vector of scalar values. If not supplied, it defaults to the empty array ("()").
- disclosed_indexes (OPTIONAL), vector of non-negative integers in ascending order. If not supplied, it defaults to the empty array ("()").
- api_id (OPTIONAL), an octet string. If not supplied it defaults to the empty octet string ("").

Parameters:

- P1, fixed point of G1, defined by the ciphersuite.

Outputs:

- init_res, vector consisting of 5 points of G1 and a scalar, in that order.

Deserialization:

```
1.  (Abar, Bbar, D, e^, r1^, r3^, commitments, c) = proof
2.  U = length(commitments)
3.  R = length(disclosed_indexes)
4.  L = R + U
5.  (i1, ..., iR) = disclosed_indexes
6.  for i in disclosed_indexes, if i < 0 or i > L - 1, return INVALID
7.  (j1, ..., jU) = (0, 1, ..., L - 1) \ disclosed_indexes
8.  if length(disclosed_messages) != R, return INVALID
9.  (msg_i1, ..., msg_iR) = disclosed_messages
10. (m^_j1, ...., m^_jU) = commitments

11. if length(generators) != L + 1, return INVALID
12. (Q_1, MsgGenerators) = generators
13. (H_1, ..., H_L) = MsgGenerators
14. (H_i1, ..., H_iR) = (MsgGenerators[i1], ..., MsgGenerators[iR])
15. (H_j1, ..., H_jU) = (MsgGenerators[j1], ..., MsgGenerators[jU])
```

Procedure:

```
1. domain = calculate_domain(PK, Q_1, (H_1, ..., H_L), header, api_id)

2. T1 = Bbar * c + Abar * e^ + D * r1^
3. Bv = P1 + Q_1 * domain + H_i1 * msg_i1 + ... + H_iR * msg_iR
4. T2 = Bv * c + D * r3^ + H_j1 * m^_j1 + ... +  H_jU * m^_jU

5. return (Abar, Bbar, D, T1, T2, domain)
```

#### 3.7.4. Challenge Calculation

This operation calculates the challenge scalar value, used during the CoreProofGen (Section 3.6.3) and CoreProofVerify (Section 3.6.4), as part of the Fiat-Shamir heuristic, for making the proof protocol non-interactive (in a interactive setting, the challenge would be a random value supplied by the Verifier).

As inputs, this operation will accept the proof generation or verification initialization result, as outputted by the ProofInit (Section 3.7.1) or ProofVerifyInit (Section 3.7.3) operations (init_res). It will additionally accept the set of scalars representing the messages the Prover disclosed (disclosed_messages) as well as the list of indexes those messages had in the vector of signed messages (disclosed_indexes), together with the presentation_header (denoted as ph on the inputs of the ProofChallengeCalculate operation).

At a high level, the challenge will be calculated as the digest (using hash_to_scalar defined in Section 4.2.2, to map it to a scalar value) of the following values:

* The total number of disclosed messages R.
* Each index in the disclosed_indexes list, followed by the corresponding disclosed message (i.e., if disclosed_indexes = [i1, i2] and disclosed_messages = [msg_i1, msg_i2], the input to the challenge digest, after R, will include i1 || msg_i1 || i2 || msg_i2).
* The points Abar, Bbar, D, T1, T2 and the domain scalar, calculated during the proof initialization phase of CoreProofGen (see Section 3.6.3).
* The input presentation_header (ph) values.

This operation makes use of the serialize function, defined in Section 4.2.4.1.

```
challenge = ProofChallengeCalculate(init_res, disclosed_messages,
                                    disclosed_indexes, ph, api_id)

Inputs:
- init_res (REQUIRED), vector representing the value returned after
                       initializing the proof generation or verification
                       operations, consisting of 5 points of G1 and a
                       scalar value, in that order.
- disclosed_messages (OPTIONAL), vector of scalar values. If not
                                 supplied, it defaults to the empty
                                 array ("()").
- disclosed_indexes (OPTIONAL), vector of non-negative integers in
                                ascending order. If not supplied, it
                                defaults to the empty array ("()").
- ph (OPTIONAL), an octet string. If not supplied, it must default to
                 the empty octet string ("").
- api_id (OPTIONAL), an octet string. If not supplied it defaults to the
                     empty octet string ("").

Outputs:

- challenge, a scalar.

Definitions:

1. hash_to_scalar_dst, an octet string representing the domain
                       separation tag: api_id || "H2S_" where "H2S_" is
                       an ASCII string comprised of 4 bytes.

Deserialization:
```

```
1. R = length(disclosed_indexes)
2. (i1, ..., iR) = disclosed_indexes
3. if length(disclosed_messages) != R, return INVALID
3. (msg_i1, ..., msg_iR) = disclosed_messages
4. (Abar, Bbar, D, T1, T2, domain) = init_res

ABORT if:

1. R > 2^64 - 1
2. length(ph) > 2^64 - 1

Procedure:

1. c_arr = (R, i1, msg_i1, i2, msg_i2, ..., iR, msg_iR, Abar, Bbar,
                                                        D, T1, T2, domain)
2. c_octs = serialize(c_arr) || I2OSP(length(ph), 8) || ph
3. return hash_to_scalar(c_octs, hash_to_scalar_dst)
```

*Note*: If the presentation_header (ph) is not supplied in ProofChallengeCalculate, 8 bytes representing a length of 0 (i.e., 0x0000000000000000), must still be appended after the serialize(c_arr) value, during the concatenation step of the above procedure (step 2).

### 3.8. Defining New Interfaces

This document defines a BBS Interface to be a set of operations that use the core functions defined in Section 3.6, to generate and validate BBS signatures and proofs. These core operations require a set of generators, and optionally, a set of scalars representing the messages.

The Interface operations are tasked with creating the generators, as well as mapping the received set of messages to a set of scalar values. The created generators MUST follow the requirements listed in Section 4.1.1.1. If a set of messages is supplied, the mapping to scalars procedure MUST follow the requirements listed in Section 4.1.2.1.

Each Interface MUST also define a unique identifier as a parameter, called api_id. It is RECOMMENDED from the operations that create generators and map messages to scalars, to also define a unique identifiers (see Section 4.1). Assuming that CREATE_GENERATORS_ID is the unique identifier of the operation that creates the generators and MAP_TO_SCALAR_ID is the unique identifier of the operation that maps the messages to scalars, the RECOMMENDED format for the api_id is the following:

```
ciphersuite_id || CREATE_GENERATORS_ID || MAP_TO_SCALAR_ID || ADD_INFO
```

Where ciphersuite_id is defined by the ciphersuite and the ADD_INFO value is an optional octet string indicating any additional information used to uniquely qualify the Interface. When ADD_INFO is present, it MUST only contain ASCII encoded characters with codes between 0x21 and 0x7e (inclusive) and MUST end with an underscore (ASCII code: 0x5f), other than the last character the string MUST NOT contain any other underscores (ASCII code: 0x5f). The api_id value, MUST be used by all subroutines an Interface calls, to ensure proper domain separation.

Interfaces are meant to make it easier to use BBS Signature as part of other protocols with different requirements (for example, different types of input messages or different ways to create the generators), or to extend BBS Signatures with additional functionality (for example, using blinded messages as in [CDL16]). Documents defining new BBS Interfaces, other than adhering to the requirements listed in this section, should also include a detailed and peer reviewed analyses showcasing that, under reasonable cryptographic assumptions, the documented scheme is secure under the required security definitions and threat model of each protocol. In other words, Interfaces must be treated like Ciphersuites (Section 7), in the sense that applications should avoid creating their own, proprietary Interfaces.

# 4. Utility Operations

This section defines utility operations that are used by either the BBS Interface or the BBS Core Operations.

## 4.1. Interface Utilities

This section defines the create_generators and messages_to_scalars operations that are used by the BBS Signatures Interface defined in Section 3.5. It also defines requirements for alternative operations that calculate generators and map messages to scalars.

It is RECOMMENDED that the create_generators and messages_to_scalars operations define a unique identifier, called CREATE_GENERATORS_ID and MAP_TO_SCALAR_ID respectively. Those identifiers will be used to construct the Interface identifier (see Section 3.8).

#### 4.1.1. Generators Calculation

The create_generators procedure defines how to create a set of randomly sampled points from the G1 subgroup, called the generators. It makes use of the primitives defined in [RFC9380] (more specifically of hash_to_curve and expand_message) to hash a seed to a set of generators. Those primitives are implicitly defined by the ciphersuite, through the choice of a hash-to-curve suite (see the hash_to_curve_suite parameter in Section 7.1).

Since create_generators generates constant points, as an optimization, implementations MAY cache its result for a specific count (which can be arbitrarily large, depending on the application). Care must be taken, to guarantee that the generators will be fetched from the cache in the same order they had when they where created (i.e., an application should not sort or in any way rearrange the cached generators).

```
generators = create_generators(count, api_id)

Inputs:

- count (REQUIRED), unsigned integer. Number of generators to create.
- api_id (OPTIONAL), octet string. If not supplied it defaults to the
                     empty octet string ("").

Parameters:

- hash_to_curve_g1, the hash_to_curve operation for the G1 subgroup,
                    defined by the suite specified by the
                    hash_to_curve_suite parameter of the ciphersuite.
- expand_message, the expand_message operation defined by the suite
                  specified by the hash_to_curve_suite parameter of the
                  ciphersuite.
- expand_len, defined by the ciphersuite.

Outputs:

- generators, an array of generators.

Definitions:

1. seed_dst, an octet string representing the domain separation tag:
             api_id || "SIG_GENERATOR_SEED_" where "SIG_GENERATOR_SEED_"
             is an ASCII string comprised of 19 bytes.
2. generator_dst, an octet string representing the domain separation
                  tag: api_id || "SIG_GENERATOR_DST_", where
                  "SIG_GENERATOR_DST_" is an ASCII string comprised of
                  18 bytes.
3. generator_seed, an octet string representing the domain separation
                   tag: api_id || "MESSAGE_GENERATOR_SEED", where
                   "MESSAGE_GENERATOR_SEED" is an ASCII string comprised
                   of 22 bytes.

ABORT if:

1. count > 2^64 - 1

Procedure:

1. v = expand_message(generator_seed, seed_dst, expand_len)
2. for i in (1, 2, ..., count):
3.    v = expand_message(v || I2OSP(i, 8), seed_dst, expand_len)
4.    generator_i = hash_to_curve_g1(v, generator_dst)
5. return (generator_1, ..., generator_count)
```

The value of v MAY also be cached in order to efficiently extend an existing list of cached generator points.

The CREATE_GENERATORS_ID of the above operation is define as,

CREATE_GENERATORS_ID = "H2G_"

#### 4.1.1.1. Defining new Generators

When defining a new create_generators procedure, the most important property is that the points are pseudo-randomly chosen from the G1 group, with no known relationship to each other, given reasonable assumptions and cryptographic primitives. More specifically, the required properties are

* The generators should be indistinguishable from uniformly random points of G1 (even given the knowledge of the system's public parameters, like the generator_seed value in Section 4.1.1). This means that given only the points H_1, ..., H_i it should be infeasible to guess H_(i+1) (or any H_j with j > i), for any i. This also means that it should be infeasible to represent any of the generators as multi-exponentiation product (i.e., of the form H_i1 * a_1 + H_i2 * a_2 + ... + H_in * a_n) of any of the other generators.
* The returned points must be unique with very high probability, that would not lessen the targeted security level of the ciphersuite. Specifically, for a security level k, the probability of a collision should be at most 1/2^k.
* The returned points must be different from the Identity point of G1 as well as the constant point P1 defined by the ciphersuite.

Every operation that is used to return generator points for use with the core BBS operations (Section 3.6), MUST return points that conform to the aforementioned rules. Such operation must also follow the rules outlined bellow,

* It MUST be deterministic and constant time for a specific number of generators.
* It MUST use proper domain separation for both the create_generators procedure, as well as all of the internally-called procedures.

### 4.1.2. Messages to Scalars

The messages_to_scalars operation is used to map a list of messages to their respective scalar values, which are required by the core BBS operations defined in Section 3.6.

```
msg_scalar = messages_to_scalars(messages, api_id)

Inputs:

- messages (REQUIRED), a vector of octet strings.
- api_id (OPTIONAL), octet string. If not supplied it defaults to the
                     empty octet string ("").

Outputs:

- msg_scalars, a list of scalars.

Definitions:

1. map_dst, an octet string representing the domain separation tag:
            api_id || "MAP_MSG_TO_SCALAR_AS_HASH_" where
            "MAP_MSG_TO_SCALAR_AS_HASH_" is an ASCII string comprised of
            26 bytes.

ABORT if:

1. length(messages) > 2^64 - 1

Procedure:

1. L =  length(messages)
2. for i in (1, ..., L):
3.     msg_scalar_i = hash_to_scalar(messages[i], map_dst)
4. return (msg_scalar_1, ..., msg_scalar_L)
```

The MAP_TO_SCALAR_ID of the above operation is defines as,

```
MAP_TO_SCALAR_ID = "HM2S_"
```

#### 4.1.2.1. Define a new Map to Scalar

The most important property that a new operation that will map a set of messages to a set of scalars must have, is that each message should be mapped to a scalar independently from all the other messages. More specifically, the following MUST hold,

For every set of messages and every message msg',
let messages' be the list of messages with msg' appended at the end and
C1 = messages_to_scalars(messages').

Let also msg_prime_scalar = messages_to_scalars((msg')),
and C2 = messages_to_scalars(messages).

If we append msg_prime_scalar at the end of C2, it must always hold that
C1 == C2.

Note that the above property ensures that if a message is mapped to a scalar on its own or as part of a set of messages, it will not affect the resulting scalar value.

Additionally, the new operation MUST conform to the following requirements:

* The returned scalars MUST be independent. More specifically, knowledge of any subset of the returned scalars MUST NOT reveal any information about the scalars not in that subset.
* Unique inputs MUST result in unique outputs.
* If the inputted vector of messages does not include any duplicates, the outputted scalars MUST NOT include any duplicates either.
* It MUST be deterministic and constant time on the length of the inputted vector of messages.

### 4.2. Core Utilities

This section defines utility procedures that are used by the Core operations defined in Section 3.6.

#### 4.2.1. Random Scalars

This operation returns the requested number of pseudo-random scalars, using the get_random operation (see Section 3.1). The operation makes multiple calls to get_random. It is REQUIRED that each call will be independent from each other, as to ensure independence of the returned pseudo-random scalars.

*Note*: The security of the proof generation algorithm (ProofGen defined in Section 3.5.3) is highly dependant on the quality of the get_random function. Care must be taken to ensure that a cryptographically secure pseudo-random generator is chosen, and that its outputs are not leaked to an adversary. See also Section 6.7 for more details and guidance.

```
random_scalars = calculate_random_scalars(count)

Inputs:

- count (REQUIRED), non negative integer. The number of pseudo random
                    scalars to return.

Parameters:

- get_random, a pseudo random function with extendable output, returning
              uniformly distributed pseudo random bytes.
- expand_len, defined by the ciphersuite.

Outputs:

- random_scalars, a list of pseudo random scalars,

Procedure:

1. for i in (1, 2, ..., count):
2.     r_i = OS2IP(get_random(expand_len)) mod r
3. return (r_1, r_2, ..., r_count)
```

#### 4.2.2. Hash to Scalar

This operation describes how to hash an arbitrary octet string to a scalar value in the multiplicative group of integers mod r (i.e., values in the range from 1 to r - 1). This procedure acts as a helper function, used internally in various places within the operations described in the spec.

The operation takes as input an octet string representing the octet string to hash (msg) and a domain separation tag (dst). The length of the dst MUST be less than 255 octets. See section 5.3.3 of [RFC9380] for guidance on using larger dst values.

*Note* This operation makes use of expand_message defined in [RFC9380]. The operation expand_message may fail (abort). In that case, hash_to_scalar MUST also ABORT.

```
hashed_scalar = hash_to_scalar(msg_octets, dst)

Inputs:

- msg_octets (REQUIRED), an octet string. The message to be hashed.
- dst (REQUIRED), an octet string representing a domain separation tag.

Parameters:

- hash_to_curve_suite, the hash to curve suite id defined by the
                       ciphersuite.
- expand_message, the expand_message operation defined by the suite
                  specified by the hash_to_curve_suite parameter.
- expand_len, defined by the ciphersuite.

Outputs:

- hashed_scalar, a scalar.

ABORT if:

- length(dst) > 255

Procedure:

1. uniform_bytes = expand_message(msg_octets, dst, expand_len)
2. return OS2IP(uniform_bytes) mod r
```

#### 4.2.3. Domain Calculation

This operation calculates the domain value, a scalar representing the distillation of all essential contextual information for a signature. The same domain value must be calculated by all parties (the Signer, the Prover and the Verifier) for both the signature and proofs to be validated.

The input to the domain value includes the header value chosen by the Signer to encode any information that is required to be revealed by the Prover (such as an expiration date, or an identifier for the target audience). This is in contrast to the signed message values, which may be withheld during a proof.

When a signature is calculated, the domain value is combined with a specific generator point (Q_1, see CoreSign defined in Section 3.6.1) to protect the integrity of the public parameters and the header.

This operation makes use of the serialize function, defined in Section 4.2.4.1.

```
domain = calculate_domain(PK, Q_1, H_Points, header, api_id)

Inputs:

- PK (REQUIRED), an octet string, representing the public key of the
                 Signer of the form outputted by the SkToPk operation.
- Q_1 (REQUIRED), point of G1 (the first point returned from
                  create_generators).
- H_Points (REQUIRED), array of points of G1.
- header (OPTIONAL), an octet string. If not supplied, it must default
                     to the empty octet string ("").
- api_id (OPTIONAL), octet string. If not supplied it defaults to the
                     empty octet string ("").

Outputs:

- domain, a scalar.

Definitions:

1. hash_to_scalar_dst, an octet string representing the domain
                       separation tag: api_id || "H2S_" where "H2S_" is
                       an ASCII string comprised of 4 bytes.

Deserialization:

1. L = length(H_Points)
2. (H_1, ..., H_L) = H_Points

ABORT if:

1. length(header) > 2^64 - 1 or L > 2^64 - 1

Procedure:

1. dom_array = (L, Q_1, H_1, ..., H_L)
2. dom_octs = serialize(dom_array) || api_id
3. dom_input = PK || dom_octs || I2OSP(length(header), 8) || header
4. return hash_to_scalar(dom_input, hash_to_scalar_dst)
```

*Note*: If the header is not supplied in calculate_domain, it defaults to the empty octet string (""). This means that in the concatenation step of the above procedure (step 3), 8 bytes representing a length of 0 (i.e., 0x0000000000000000), will still need to be appended at the end, even though a header value is not provided.

#### 4.2.4. Serialization

##### 4.2.4.1. Serialize

This operation describes how to transform multiple elements of different types (i.e., elements that are not already in a octet string format) to a single octet string (see Section 3.3.5). The inputted elements can be points, scalars (see Section 1.1) or integers between 0 and 2^64-1. The resulting octet string will then either be used as an input to a hash function (i.e., in CoreSign Section 3.6.1, CoreVerify Section 3.6.2, CoreProofGen Section 3.6.3 and CoreProofVerify Section 3.6.4), or to serialize a signature or proof (see signature_to_octets Section 4.2.4.2 and proof_to_octets Section 4.2.4.4).

```
octets_result = serialize(input_array)

Inputs:

- input_array (REQUIRED), an array of elements to be serialized. Each
                          element must be either a point of G1 or G2, a
                          scalar, an ASCII string or an integer value
                          between 0 and 2^64 - 1.

Parameters:

- octet_scalar_length, non-negative integer. The length of a scalar
                       octet representation, defined by the ciphersuite.
- r, the prime order of the subgroups G1 and G2, defined by the
     ciphersuite.
- point_to_octets_E*, operations that serialize a point of E1 or E2 to
                      an octet string of fixed length, defined by the
                      ciphersuite.

Outputs:

- octets_result, a scalar value or INVALID.

Procedure:

1.  let octets_result be an empty octet string.
2.  for el in input_array:
3.      if el is a point of G1: el_octs = point_to_octets_E1(el)
4.      else if el is a point of G2: el_octs = point_to_octets_E2(el)
5.      else if el is a scalar: el_octs = I2OSP(el, octet_scalar_length)
6.      else if el is an integer between 0 and 2^64 - 1:
7.          el_octs = I2OSP(el, 8)
8.      else: return INVALID
9.      octets_result = octets_result || el_octs
10. return octets_result
```

#### 4.2.4.2. Signature to Octets

This operation describes how to encode a signature to an octet string.

_Note_ this operation deliberately does not perform the relevant checks on the inputs A and e because its assumed these are done prior to its invocation, e.g., as is the case with the CoreSign Section 3.6.1 operation.

signature_octets = signature_to_octets(signature)

Inputs:

- signature (REQUIRED), a valid signature, in the form (A, e), where
  A is a point in G1 and e is a non-zero
  scalar mod r.

Outputs:

- signature_octets, an octet string or INVALID.

Procedure:

1. (A, e) = signature
2. return serialize((A, e))

#### 4.2.4.3. Octets to Signature

This operation describes how to decode an octet string, validate it and return the underlying components that make up the signature.

```
signature = octets_to_signature(signature_octets)

Inputs:

- signature_octets (REQUIRED), an octet string of the form output from
                               signature_to_octets operation.

Parameters:

- octets_to_point_E1, operations that deserializes an octet string to a
                      a point of the elliptic curve E1, or INVALID,
                      defined by the ciphersuite.
- subgroup_check_G1, operation that on input a point P returns VALID if
                     P is a valid point of the G1 subgroup, otherwise it
                     returns INVALID (see (#notation)).

Outputs:

signature, a signature in the form (A, e), where A is a point in G1
           and e is a non-zero scalar mod r; or INVALID.

Procedure:

1.  expected_len = octet_point_length + octet_scalar_length
2.  if length(signature_octets) != expected_len, return INVALID
3.  A_octets = signature_octets[0..(octet_point_length - 1)]
4.  A = octets_to_point_E1(A_octets)
5.  if A is INVALID, return INVALID
6.  if A == Identity_G1, return INVALID
7.  if subgroup_check_G1(A) returns INVALID, return INVALID

8.  index = octet_point_length
9.  end_index = index + octet_scalar_length - 1
10. e = OS2IP(signature_octets[index..end_index])
11. if e = 0 or e >= r, return INVALID
12. return (A, e)
```

#### 4.2.4.4. Proof to Octets

This operation describes how to encode as an octet string, a proof as computed by CoreProofGen in Section 3.6.3 (or, more precisely, by step 5 of the ProofFinalize operation defined in Section 3.7.2).

The inputted proof value must consist of the following components, in that order:

1. Three (3) valid points of the G1 subgroup, different from the identity point of G1 (i.e., Abar, Bbar, D, in ProofGen)

2. Three (3) integers representing scalars in the range of 1 to r - 1 inclusive (i.e., e^, r1^, r3^, in ProofGen).
3. A number of integers representing scalars in the range of 1 to r - 1 inclusive, corresponding to the undisclosed from the proof messages (i.e., m^_j1, ..., m^_jU, in ProofGen, where U the number of undisclosed messages).
4. One (1) integer representing a scalar in the range 1 to r-1 inclusive (i.e., c in ProofGen).

proof_octets = proof_to_octets(proof)

Inputs:

- proof (REQUIRED), a BBS proof in the form calculated by ProofGen in step 27 (see above).

Outputs:

- proof_octets, an octet string or INVALID.

Procedure:

1. (Abar, Bbar, D, e^, r1^, r3^, (m^_1, ..., m^_U), c) = proof
2. return serialize((Abar, Bbar, D, e^, r1^, r3^, m^_1, ..., m^_U, c))

#### 4.2.4.5. Octets to Proof

This operation describes how to decode an octet string representing a proof, validate it and return the underlying components that make up the proof value.

The proof value outputted by this operation consists of the following components, in that order:

1. Three (3) valid points of the G1 subgroup, each of which must not equal the identity point.
2. Three (3) integers representing scalars in the range of 1 to r - 1 inclusive.
3. A set of integers representing scalars in the range of 1 to r - 1 inclusive, corresponding to the undisclosed from the proof message commitments. This set can be empty (i.e., "()").
4. One (1) integer representing a scalar in the range of 1 to r - 1 inclusive, corresponding to the proof’s challenge (c).

```
proof = octets_to_proof(proof_octets)

Inputs:

- proof_octets (REQUIRED), an octet string of the form outputted from
                           the proof_to_octets operation.

Parameters:

- r, non-negative integer. The prime order of the G1 and G2 groups,
      defined by the ciphersuite.
- octet_scalar_length, non-negative integer. The length of a scalar
                       octet representation, defined by the ciphersuite.
- octet_point_length, non-negative integer. The length of a point in G1
                      octet representation, defined by the ciphersuite.
- subgroup_check_G1, operation that on input a point P returns VALID if
                     P is a valid point of the G1 subgroup, otherwise it
                     returns INVALID (see (#notation)).

Outputs:

- proof, a proof value in the form described above or INVALID

Procedure:

1.  proof_len_floor = 3 * octet_point_length + 4 * octet_scalar_length
2.  if length(proof_octets) < proof_len_floor, return INVALID

// Points (i.e., (Abar, Bbar, D) in ProofGen) de-serialization.
3.  index = 0
4.  for i in (0, 2):
5.      end_index = index + octet_point_length - 1
6.      A_i = octets_to_point_E1(proof_octets[index..end_index])
7.      if A_i is INVALID or Identity_G1, return INVALID
8.      if subgroup_check_G1(A_i) returns INVALID, return INVALID
9.      index += octet_point_length

// Scalars (i.e., (e^, r1^, r3^, m^_j1, ..., m^_jU, c) in
// ProofGen) de-serialization.
10. j = 0
11. while index < length(proof_octets):
12.     end_index = index + octet_scalar_length - 1
13.     s_j = OS2IP(proof_octets[index..end_index])
14.     if s_j = 0 or if s_j >= r, return INVALID
15.     index += octet_scalar_length
16.     j += 1

17. if index != length(proof_octets), return INVALID
```

18. msg_commitments = ()
19. if j > 4, set msg_commitments = (s_3, ..., s_(j-2))
20. return (A_0, A_1, A_2, s_0, s_1, s_2, msg_commitments, s_(j-1))

#### 4.2.4.6. Octets to Public Key

This operation describes how to decode an octet string representing a public key, validates it and returns the corresponding point in G2. Steps 2 to 5 check if the public key is valid. As an optimization, implementations MAY cache the result of those steps, to avoid unnecessarily repeating validation for known public keys.

W = octets_to_pubkey(PK)

Inputs:

- PK, an octet string. A public key in the form outputted by the SkToPK operation

Parameters:

- subgroup_check_G2, operation that on input a point P returns VALID if P is a valid point of the G2 subgroup, otherwise it returns INVALID (see (#notation)).

Outputs:

- W, a valid point in G2 or INVALID

Procedure:

1. W = octets_to_point_E2(PK)
2. if W is INVALID, return INVALID
3. if subgroup_check_G2(W) is INVALID, return INVALID
4. if W == Identity_G2, return INVALID
5. return W

## 5. Privacy Considerations

This section will go through threats to the Prover’s privacy. Note that a BBS proof is unlinkable against both the Verifiers and the Signer, as well as multiple Verifiers colluding with each other and Verifiers colluding with the Signer. Bear in mind that those guarantees concern only the proof value, as outputted by the CoreProofGen (Section Section 3.6.3) and of course the ProofGen (Section Section 3.5.3) operations. Correspondingly, the unlinkability property does not include other values that a Prover could either knowingly or unknowingly provide to a Verifier. Those

values can include disclosed messages of high entropy, the header and presentation_header values, their network address, or generally any information disclosed during an interaction with the Verifier having the potential to identify the user. Such threats, if exploited, could lead to correlation of the Prover's interactions with different Verifiers, resulting to fingerprinting attacks against the Prover's activity.

The following sections will describe possible privacy threats, resulting from such values and side channels, that could compromise the unlinkability property of the BBS proof. Note that, the following sections describe ways to minimize possible identifying information revealed during a BBS proof presentation, related to the BBS Signatures scheme. To minimize the privacy threats of an entire system, other protections may also need to be employed, for example, using an IP hiding proxy network like TOR ([DMS04]).

### 5.1. Header and Presentation Header

As mentioned in Section Section 5.1, the header value is chosen by the Signer and bound to a BBS Signature and proof. Consequently, it must be revealed to the Verifier, together with a BBS proof. If that header value is chosen to have high entropy (i.e., unique per credential, Prover or small group of Provers), it can be used as a correlation vector to trace and link together all BBS proofs made by a signature bound to that header value. This will result in significantly worse privacy guarantees, by allowing adversaries to trace and link together all generated proofs bound to that header value (for example, if a random header is used during each BBS signature generation, adversaries will be able to link and trace the BBS proofs generated from that signature). The Issuer MUST choose a low entropy header value and it MUST be the same for a large number of users (Provers). Examples of acceptable values include, an application identifier, a country identifier or a low cardinality version number. Examples of unacceptable values include, random values, high cardinality expiration dates, the Prover's email address or any other identifying information.

On the other hand, the misuse of a presentation_header, chosen by the Prover and only bound to a BBS proof, does not incur as many privacy risks as the header value. For example, since a new presentation_header can be chosen each time a BBS proof is generated, random values are a viable choice. Still, to not break the unlinkability property, care must be taken that the presentation_header does not identify a single or small group of Provers. If the presentation_header is chosen to have high entropy (for example, to be a random value, a high accuracy locality identifier or a specific software build number), then the same value

must not be used for more than one Proof generations. Note however, that even though the presentation_header can include high entropy values (as long as they are used only once), its a good practice for the Prover to avoid revealing personally identifying information (like their name, email address or phone number), to minimize the danger of correlating that information with other data sources, potentially unrelated to the specific application.

## 5.2. Total Number and Index of Signed Messages

When a Prover presents a BBS proof to a Verifier, other than the messages they decide to disclose, there are two additional pieces of information that will be revealed. First, the total number of signed messages, which can be inferred from the size of the BBS proof and the length of the disclosed messages list. Second, the indexes that the disclosed messages had in the list of signed messages (see Section 3.5.3). This information, if unique to each Prover, could be employed to correlate multiple proof presentations together. As a result, the Signer should not sign lists of messages with unique lengths or unique indexing. For this reason, it is RECOMMENDED that signed lists of messages are padded to a common length (using either random, or an unused by the application message, like 0 or 1). It is also RECOMMENDED that a constant ordering of messages will be preserved when possible. For example, if an application creates signatures for the messages [<user_name>, <user_affiliation>, <user_country>], then those messages should always be signed in the same order, i.e., first message should always be the user's name (<user_name>), second message should always be the user's affiliation (<user_affiliation>) and the last message should always be the user's country of origins (<user_country>). Provers can employ consistency validation mechanisms, like the ones described in [I-D.ietf-privacypass-key-consistency], to validate that those values are not used to correlate them.

## 5.3. Signer Public Keys

As with most systems based on public key cryptography, multiple BBS signatures (and the subsequent BBS proofs) could be correlated with each other, if the Signer does not use the same key for a large set of produced signatures. For example, the Signer could use a different key to generate the signatures intended for a specific user, or a small set of users. Every proof generated by that set of users would then be linked to that group (since it will be validated by a different public key). To avoid fragmentation of the user space by different public keys, an application could use the same mechanisms that where proposed to check the consistency of the total number of messages and their indexes (i.e., [I-D.ietf-privacypass-key-consistency], see Section 5.2).

### 5.4. Disclosed Messages

Although multiple BBS proofs cannot be linked to each other, privacy also depends on the uniqueness of the disclosed messages during proof generation. If a unique message (or unique combination of messages) is revealed multiple times, it could be used to link the corresponding proofs together. Examples of such messages include full names, government IDs, email addresses and phone numbers. If not required by the use case, the Prover should avoid disclosing such information when constructing a BBS proof.

For certain types of message values, set membership proofs (for example, [VB22]) or range proofs (for example, [BBB17]) could be used to further mitigate the above issue. With a set membership proof, the BBS proof Verifier will be able to validate that one of the Prover's signed (and undisclosed) messages, belongs to a pre-defined set (for example that the Prover's government ID belongs to a set of valid government IDs). The inverse is also possible, where the Prover showcases that one of the undisclosed messages is not part of a set (for example, that a signed unique revocation identifier is not part of the set of revoked identifiers). If a message is represented by a numeric value (see Section 6.8), range proofs can be used to prove that it is within a specific range. As an example, a Prover, instead of revealing their age, they could use a range proof to showcase that they are over 18 years old.

## 6. Security Considerations

### 6.1. Validating Public Keys

Note that all core operations as defined in Section 3.6 expect the Signer's public key as input. It is RECOMMENDED for all those operations, that they deserialize the public key first using the octets_to_pubkey procedure defined in Section 4.2.4.6, even if they only require the octet string representation of the public key. If the octets_to_pubkey procedure returns INVALID, the calling operation should also return INVALID and abort. This recommendation applies to the CoreSign (Section 3.6.1) and CoreProofGen (Section 3.6.3) operations. An explicit invocation to the octets_to_pubkey operation is already defined and therefore required in the CoreVerify (Section 3.6.2) and CoreProofVerify (Section 3.6.4) operations. If the required checks for the validity of the Signer's public key are not performed, the results are unpredictable, leading to unexpected vulnerabilities (for example, the output of the pairing operation on input of an invalid elliptic curve point can be highly irregular and implementation-dependent, with some returning the identity point of the elliptic curve and others returning errors).

### 6.2. Skipping Membership Checks

The subgroup check subgroup_check_G* invocation during either signature deserialization (octets_to_signature, defined in Section 4.2.4.3), proof deserialization (octets_to_proof, defined in Section 4.2.4.5) or public key deserialization (octets_to_pubkey, define in Section 4.2.4.6) is REQUIRED by all implementations. Failure to comply would lead to unpredicted behavior and vulnerabilities. Note that some libraries implementing the pairing-friendly curves functionality, may incorporate that check as part of a octets_to_point_G1 or octet_to_point_G2 operation (i.e., operations that both deserialize an octet string to get an elliptic curve point and then check if the resulting point is part of the G1 or G2 group accordingly). In those cases, the implementer must make sure that those checks are executed correctly.

Note that checking that the points are in the correct subgroup is essential to avoid possible forgeries of a BBS signature or proof ([ADR02]). Furthermore, the pairing operation Section 1.2 is undefined when its input points are not in G1 and G2. As a result, applications MUST execute all the subgroup checks defined by this document.

### 6.3. Side Channel Attacks

There are two places where side channel attacks could be relevant in the BBS Signatures scheme. First, against the Signer, where side channel leakage during signature generation could reveal their secret key. Second, against the Prover, where a side channel attack could be used during proof generation to either directly reveal the undisclosed messages and signature value, or reveal the random scalars used, leading again to the leakage of the undisclosed messages or the hidden signature. Therefore, implementations MUST apply proper side channel attack protection. One method to achieve this, is by using elliptic curve implementations that execute curve operations in constant time.

### 6.4. Presentation Header Selection

The signature proofs of knowledge generated in this specification are created using a specified presentation_header. A Verifier-specified cryptographically random value (e.g., a nonce) featuring in the presentation_header provides strong protections against replay attacks, and is RECOMMENDED in most use cases. In some settings, proofs can be generated in a non-interactive fashion, in which case verifiers MUST be able to verify the uniqueness of the presentation_header values.

### 6.5. Implementing hash_to_curve_g1

The security analysis models hash_to_curve_g1 as random oracles. It is crucial that these functions are implemented using a cryptographically secure hash function. For this purpose, implementations MUST meet the requirements of [RFC9380].

In addition, ciphersuites MUST specify unique domain separation tags for hash_to_curve. Some guidance around defining this can be found in Section 7.

### 6.6. Choice of Underlying Curve

BBS signatures can be implemented on any pairing-friendly curves suitable for type 3 pairing computations. However care must be taken when selecting one that is appropriate, to guarantee the desired security level for the targeted application. This specification defines a ciphersuite for using the BLS12-381 curve in Section 7 which as a curve achieves around 117 bits of security [ZCASH-REVIEW].

### 6.7. Randomness Requirements

The key_material input to the KeyGen operation defined in Section 3.4.1 MUST be infeasible to guess and MUST be kept secret. One possibility is to generate the key_material from a trusted, cryptographically secure pseudo random function [RFC4086]. Secret keys MAY be generated using other methods; in this case they MUST be infeasible to guess and MUST be indistinguishable from uniformly random modulo r.

The ProofGen operation defined in Section 3.5.3 is by its nature a randomized algorithm, requiring the generation of multiple uniformly distributed, pseudo random scalars. This makes ProofGen vulnerable to attacks caused by bad entropy (like the ones described in [HDWH12]). If randomness is re-used or is in any way predictable or maliciously constructed, an adversary may be able to unveil undisclosed information from the proof messages or the hidden signature value. More subtle attacks are also possible, where the security properties of the BBS proof may not be broken, but a system making use of the BBS scheme may still be compromised. As an example, consider systems that needs to monitor and potentially restrict outbound traffic, in order to minimize data leakage during a breach. In such cases, the attacker could manipulate couple of bits in the output of the get_random function (Section 3.1) to create an undetected channel out of the system. Although the applicability of such attacks is limited for most of the targeted use cases of the BBS scheme, some applications may want to take measures towards mitigating them. To that end, it is RECOMMENDED to use a

deterministic RNG (like a ChaCha20 based deterministic RNG), seeded with a unique, uniformly random, single seed [DRBG]. This will limit the amount of bits the attacker can manipulate (note that some randomness is always needed).

In any case, the randomness used in ProofGen MUST be unique in each call and MUST have a distribution that is indistinguishable from uniform. If the random scalars are re-used, created from "bad randomness" (for example with a known relationship to each other) or are in any way predictable, the undisclosed messages or the signature value may be compromised. Naturally, a cryptographically secure pseudorandom number generator or pseudo random function is REQUIRED to implement the get_random functionality. See [RFC4086] for guidance on implementing such functionality. See also [RFC8937], for recommendations on generating good randomness in cases where the Prover has direct or in-direct access to a secret key.

### 6.8. Mapping Messages to Scalars

In an application using BBS Signatures, there are two places where messages could be processed. First, before the messages are passed to the BBS Interface operations, and second, after they are passed to the BBS Interface operations but before they are passed to the BBS Core operations.

To allow for re-usability of software, it is RECOMMENDED that application specific processing (like UTF-8 encoding [RFC3629] or Base-64 decoding [RFC4648]) would happen before messages are passed to the BBS Interface operations. In those cases, the application should ensure that all protocol participants have a clear and consistent understanding of which method should be used to process a message. This can be achieved by associating specific Interfaces (with unique api_id values, see Section 3.8) or unique header values (see Section 3.5.1) with different pre-processing methodologies.

Note that the BBS Interface defined in this document (see Section 3.5) only accepts messages that are represented as octet strings. However, in some more advanced applications, like the ones using range proofs ([BBB17]) to prove that a signed message is within some range (without disclosing that message), the pre-processing of messages may result to some of them being mapped to scalar values, before they are passed to the BBS Interface (for example, an application could use [ISO8601] to represent dates as integers or map the user’s age directly to a number) that should directly be signed (e.g., to not be further processed by hash_to_scalar).

If a BBS Interface accepts both octet strings and scalar values as messages, where depending on the message's type different operations will be used to map it to a scalar (e.g., hash_to_scalar for octet strings and the identity operation for scalars), it must still ensure that the properties described in Section 4.1.2.1 holds. To that end, the application MUST ensure that it is clear to all participants, which message should be considered an octet string and which a scalar.

As an example, if the type (i.e., octet string or scalar) of the messages inputted to the BBS Interface, is uniquely determined by its index in the messages list (for example, first message is an octet string, second message a scalar etc.,), the map between message index and message type (determined by the Signer), could be made available as part of the Signer's public parameters (similar to [UPROVE]). This map would then be passed to the BBS Interface, which will use it to correctly map each message to a scalar. Another option, is to sign such configurations as part of the header parameter of the BBS signature (see Section 3.5.1). In this case, the map does not need to be published by the Signer.

If the application defines that the first (or last) n messages will be scalars and everything else octet strings, it could just publish the n value as part of the Signer's public parameters or again sign it as part of the header value.

In any case, the privacy considerations described in Section 5 MUST NOT be violated, for example, by using unique pre-processing rules or maps between message index and type. To validate the consistency of the message processing rules, the Prover could use mechanisms like the ones described in [I-D.ietf-privacypass-key-consistency].

### 6.9. Post-quantum Security

BBS Signatures combine two security properties; data authenticity and data confidentiality.

Data authenticity refers to the inability of anyone other that the Signer being able to generate BBS signatures that are valid under the Signer's public key (this property is often referred to as unforgeability, or in the case of BBS Signatures, strong unforgeability, e.g., by [TZ23]). It also means that no one should be able to generate valid BBS proofs disclosing sets of messages, without first obtaining a valid BBS signature on those messages (in academic works, this is referred to as the BBS proof being a proof-of-knowledge of a BBS signature [CDL16] [TZ23]).

Data confidentiality means that no one (not even the Signer) should be able to use a BBS proof to extract information about the messages the Prover decided not to disclose during the proof generation process, or the signature that was used to generate that proof (something that is referred to as the zero-knowledge property of the BBS proof [BBDT16] [CDL16] [TZ23]).

On the presence of a Cryptographically Relevant Quantum Computer (CRQC), meaning a computer that will be able to break the discrete logarithm problem in the groups used by BBS Signatures (see [I-D.ietf-pquip-pqc-engineers]), the data authenticity property will not hold. Specifically, an adversary could use a CRQC to reveal the Signer’s secret key from their public key, hence giving them the ability to generate BBS signatures on behalf of that Signer, for messages of their choosing, as well as BBS proofs using those signatures.

On the other hand, data confidentiality cannot be broken, even by adversaries with unbounded computational resources and in possession of the Signer’s secret key. This means that even by utilizing a CRQC, adversaries will not be able to compromise the data confidentiality property of BBS proofs. As a result, an adversary with access to such a quantum computer, will not be able to reveal either the messages undisclosed by a BBS proof, or the hidden signature value (which the Prover showcases possession of). This guarantees that the privacy and hiding properties of BBS proofs that are currently used, will not be compromised by future quantum-attacks (a property that is often referred to as everlasting privacy). Note that this only considers BBS proofs, not BBS signatures, which do not possess the same hiding properties as the BBS proofs.

# 7. Ciphersuites

This section defines the format for a BBS ciphersuite. It also gives concrete ciphersuites based on the BLS12-381 pairing-friendly elliptic curve [I-D.irtf-cfrg-pairing-friendly-curves].

## 7.1. Ciphersuite Format

### 7.1.1. Ciphersuite ID

The following section defines the format of the unique identifier for the ciphersuite denoted ciphersuite_id, which will be represented as an ASCII encoded octet string. The REQUIRED format for this string is

```
"BBS_" || H2C_SUITE_ID || ADD_INFO
```

* H2C_SUITE_ID is the suite ID of the hash-to-curve suite used to define the hash_to_curve function.

* ADD_INFO is an optional octet string indicating any additional information used to uniquely qualify the ciphersuite. When present this value MUST only contain ASCII encoded characters with codes between 0x21 and 0x7e (inclusive) and MUST end with an underscore (ASCII code: 0x5f). The last character MUST be the only underscore.

#### 7.1.2. Additional Parameters

The parameters that each ciphersuite needs to define are generally divided into three main categories; the basic parameters (a hash function, a pairing operation, the octet length of points and scalars, the hash to curve [RFC9380] related operations and parameters as well as the base point of the G1 subgroup), the serialization operations (mapping points from each elliptic curve to an octet string and vice versa) and the generator parameters. See below for more details.

*Basic parameters*:

* hash: a cryptographic hash function.

* octet_scalar_length: Number of bytes to represent a scalar value, in the multiplicative group of integers mod r, encoded as an octet string. It is RECOMMENDED this value be set to ceil(log2(r)/8).

* octet_point_length: Number of bytes to represent a point encoded as an octet string outputted by the point_to_octets_E* function.

* hash_to_curve_suite: The hash-to-curve ciphersuite id, in the form defined in [RFC9380]. This defines the hash_to_curve_g1 (the hash_to_curve operation for the G1 subgroup, see the Notation defined in Section 1.2) and the expand_message (either expand_message_xmd or expand_message_xof) operations used in this document.

* expand_len: Must be defined to be at least ceil((ceil(log2(r))+k)/8), where log2(r) and k are defined by each ciphersuite (see Section 5 in [RFC9380] for a more detailed explanation of this definition).

* P1: A fixed point in the G1 subgroup, different from the point BP1 (i.e., the base point of G1, see Section 1.1). This leaves the base point "free", to be used with other protocols, like key commitment and proof of possession schemes (for example, like the one described in Section 3.3 of [I-D.irtf-cfrg-bls-signature]).

* h: The pairing operation used.

*Serialization functions*:

* point_to_octets_E1: a function that returns the canonical representation of the point P of the E1 elliptic curve as an octet string.

* point_to_octets_E2: a function that returns the canonical representation of the point P of the E2 elliptic curve as an octet string.

* octets_to_point_E1: a function that returns the point P in the elliptic curve E1 corresponding to the canonical representation ostr, or INVALID if ostr is not a valid output of point_to_octets_E1.

* octets_to_point_E2: a function that returns the point P in the elliptic curve E2 corresponding to the canonical representation ostr, or INVALID if ostr is not a valid output of point_to_octets_E2.

## 7.2. BLS12-381 Ciphersuites

The following two ciphersuites are based on the BLS12-381 elliptic curves defined in Section 4.2.1 of [I-D.irtf-cfrg-pairing-friendly-curves]. The targeted security level of both suites in bits is k = 128 (the actual security level is closer to 126 bits). The number of bits of the order r, of the G1 and G2 subgroups, is log2(r) = 255. The base points BP1 and BP2 of G1 and G2 are the points BP and BP’ correspondingly, as defined in Section 4.2.1 of [I-D.irtf-cfrg-pairing-friendly-curves]. For completeness, BLS12-381 and the relevant functionality (base points BP1 and BP2, the pairing h as well as the point encoding and decoding operations) are defined in Appendix B.

The first ciphersuite uses the hash-to-curve suite BLS12381G1_XOF:SHAKE-256_SSWU_RO_, defined by this document in Appendix A.1 (#bls12-381-hash_to_curve-def), which is based on the SHAKE-256 extendable output function, as defined in Section 6.2 of [SHA3].

The second ciphersuite uses the hash-to-curve suite BLS12381G1_XMD:SHA-256_SSWU_RO_, defined in Section 8.8.1 of the [RFC9380] document, which is based on the SHA-256, as defined in Section 6.2 of [SHA2] .

For both ciphersuites defined in this section, the fixed point P1 of G1 is defined as the output of the create_generators procedure defined in Section 4.1.1 instantiated with the parameters defined by each ciphersuite, with the inputs count = 1, not supplying an api_id value and making use of the following "Definitions" for the seed_dst, generator_dst and generator_seed variables;

- seed_dst: ciphersuite_id || "H2G_HM2S_SIG_GENERATOR_SEED_" where "H2G_HM2S_SIG_GENERATOR_SEED_" is an ASCII string comprised of 28 bytes.
- generator_dst: ciphersuite_id || "H2G_HM2S_SIG_GENERATOR_DST_", where "H2G_HM2S_SIG_GENERATOR_DST_" is an ASCII string comprised of 27 bytes.
- generator_seed: ciphersuite_id || "H2G_HM2S_BP_MESSAGE_GENERATOR_SEED" where "H2G_HM2S_BP_MESSAGE_GENERATOR_SEED" is an ASCII string comprised of 34 bytes.

In the above, ciphersuite_id is the unique identifier defined by each ciphersuite. Note that the P1 point is independent from the BBS Interface that may use it and it remains constant for each ciphersuite. The similarity of the above "Definitions" with the Interface identifier (api_id) defined in Section 3.5, is only for compatibility reasons with previous versions of this document.

Note that these two ciphersuites differ only in the hash-to-curve suites used. The hash-to-curve suites differ in the expand_message variant and underlying hash function. More concretely, the BLS12-381-SHAKE-256 (#bls12-381-shake-256) ciphersuite makes use of expand_message_xof with SHAKE-256, while BLS12-381-SHA-256 (#bls12-381-sha-256) makes use of expand_message_xmd with SHA-256. Curve parameters are common between the two ciphersuites.

#### 7.2.1. BLS12-381-SHAKE-256

*Basic parameters*:

* ciphersuite_id: "BBS_BLS12381G1_XOF:SHAKE-256_SSWU_RO_"

* octet_scalar_length: 32, based on the RECOMMENDED approach of ceil(log2(r)/8).

* octet_point_length: 48, based on the RECOMMENDED approach of ceil(log2(p)/8).

* hash_to_curve_suite: "BLS12381G1_XOF:SHAKE-256_SSWU_RO_" as defined in Appendix A.1 (#bls12-381-hash-to-curve-definition-using-shake-256) for the G1 subgroup.

* expand_len: 48 ( = ceil((ceil(log2(r))+k)/8))

* P1: the following point of G1, serialized using the point_to_octets_E1 procedure defined by this ciphersuite and hex encoded

P1 = h'8929dfbc7e6642c4ed9cba0856e493f8b9d7d5fcb0c31ef8fdcd34d50648a
56c795e106e9eada6e0bda386b414150755'

* h: the optimal Ate pairing (Appendix A.2 of [I-D.irtf-cfrg-pairing-friendly-curves]), defined in Appendix B.1.

*Serialization functions*:

* point_to_octets_E1: as defined in Appendix B.2.1 for points of the curve E1 (which follows the format documented in Appendix C.1 of [I-D.irtf-cfrg-pairing-friendly-curves] for the E1 elliptic curve, using compression).

* point_to_octets_E2: as defined in Appendix B.2.1 for points of the curve E2 (which follows the format documented in Appendix C.1 of [I-D.irtf-cfrg-pairing-friendly-curves] for the E2 elliptic curve, using compression).

* octets_to_point_E1: as defined in Appendix B.2.2 (which follows the format documented in Appendix C.2 of [I-D.irtf-cfrg-pairing-friendly-curves]), returning INVALID if the resulting point is not in E1.

* octets_to_point_E2: as defined in Appendix B.2.2 (which follows the format documented in Appendix C.2 of [I-D.irtf-cfrg-pairing-friendly-curves]), returning INVALID if the resulting point is not in E2.

#### 7.2.2. BLS12-381-SHA-256

*Basic parameters*:

* Ciphersuite_ID: "BBS_BLS12381G1_XMD:SHA-256_SSWU_RO_"

* octet_scalar_length: 32, based on the RECOMMENDED approach of ceil(log2(r)/8).

* octet_point_length: 48, based on the RECOMMENDED approach of ceil(log2(p)/8).

* hash_to_curve_suite: "BLS12381G1_XMD:SHA-256_SSWU_RO_" as defined in Section 8.8.1 of the [RFC9380] for the G1 subgroup.

* expand_len: 48 ( = ceil((ceil(log2(r))+k)/8))

* P1: the following point of G1, serialized using the point_to_octets_E1 procedure defined by this ciphersuite and hex encoded

P1 = h'a8ce256102840821a3e94ea9025e4662b205762f9776b3a766c872b948f1f
d225e7c59698588e70d11406d161b4e28c9'

* h: the optimal Ate pairing (Appendix A.2 of [I-D.irtf-cfrg-pairing-friendly-curves]), defined in Appendix B.1.

*Serialization functions*:

* point_to_octets_E1: as defined in Appendix B.2.1 for points of the curve E1 (which follows the format documented in Appendix C.1 of [I-D.irtf-cfrg-pairing-friendly-curves] for the E1 elliptic curve, using compression).

* point_to_octets_E2: as defined in Appendix B.2.1 for points of the curve E2 (which follows the format documented in Appendix C.1 of [I-D.irtf-cfrg-pairing-friendly-curves] for the E2 elliptic curve, using compression).

* octets_to_point_E1: as defined in Appendix B.2.2 (which follows the format documented in Appendix C.2 of [I-D.irtf-cfrg-pairing-friendly-curves]), returning INVALID if the resulting point is not in E1.

* octets_to_point_E2: as defined in Appendix B.2.2 (which follows the format documented in Appendix C.2 of [I-D.irtf-cfrg-pairing-friendly-curves]), returning INVALID if the resulting point is not in E2.

## 8. Test Vectors

The following section details a basic set of test vectors that can be used to confirm an implementation's correctness.

*NOTE* All binary data below is represented as octet strings in big endian order, encoded in hexadecimal (base 16) format. Strings prefixed with h and enclosed by single quotes ' represent base 16

encoded byte strings. On the other hand, strings prefixed with 0x (not enclosed by single quotes), represent base 16 encoded decimal numbers (scalars).

*NOTE* These fixtures are a work in progress and subject to change.

## 8.1. Mocked Random Scalars

For the purpose of presenting fixtures for the ProofGen operation (Section 3.5.3), we describe here a way to mock the calculate_random_scalars operation (Section 4.2.1), used by CoreProofGen (Section 3.6.3) to create all the necessary random scalars.

To that end, the seeded_random_scalars operation is defined, which will deterministically calculate count random-looking scalars from a single SEED, given a domain separation tag (DST). The proof test vector will then define a SEED (as a nothing-up-my-sleeve value) and a DST and then set

```
mocked_calculate_random_scalars(count) :=
                              seeded_random_scalars(SEED, DST, count)
```

The mocked_calculate_random_scalars operation will be used in place of calculate_random_scalars during the CoreProofGen operation.

*Note* For the BLS12-381-SHA-256 ciphersuite (Section 7.2.2), if more than 170 mocked random scalars are required, the operation will return INVALID. Similarly, for the BLS12-381-SHAKE-256 ciphersuite (Section 7.2.1), if more than 1365 mocked random scalars are required, the operation will return INVALID. For the purpose of describing ProofGen (Section 3.5.3) test vectors, those limits are inconsequential.

```
seeded_scalars = seeded_random_scalars(SEED, DST, count)

Inputs:

- SEED (REQUIRED), an octet string. The random seed from which to
                   generate the scalars.
- DST (REQUIRED), octet string representing a domain separation tag.
- count (REQUIRED), non negative integer. The number of scalars to
                    return.

Parameters:

- expand_message, the expand_message operation defined by the
                  ciphersuite.
- expand_len, defined by the ciphersuite.

Outputs:

- mocked_random_scalars, a list of "count" pseudo random scalars

ABORT if:

1. count * expand_len > 65535

Procedure:

1. out_len = expand_len * count
2. v = expand_message(SEED, DST, out_len)
3. if v is INVALID, return INVALID

4. for i in (1, ..., count):
5.     start_idx = (i-1) * expand_len
6.     end_idx = i * expand_len - 1
7.     r_i = OS2IP(v[start_idx..end_idx]) mod r
8. return (r_1, ...., r_count)
```

### 8.2. Messages

The following messages are used by the test vectors of both ciphersuites (unless otherwise stated). All the listed messages represent hex-encoded octet strings.

```
   m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
            5f02'
   m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
            9b80'
   m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
   m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
   m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
   m_6 = h'515ae153e22aae04ad16f759e07237b4'
   m_7 = h'd183ddc6e2665aa4e2f088af'
   m_8 = h'ac55fb33a75909ed'
   m_9 = h'96012096'
   m_10 = h''
```

### 8.3. BLS12-381-SHAKE-256 Test Vectors

Test vectors of the BLS12-381-SHAKE-256 ciphersuite defined in Appendix D.1 ciphersuite. Further fixtures are available in Appendix D.1.

#### 8.3.1. Key Pair

Following the procedure defined in Section 3.4.1 with an input key_material value as follows

```
key_material = h'746869732d49532d6a7573742d616e2d546573742d494b4d2d746f2
                 d67656e65726174652d246528724074232d6b6579'
```

the following key_info value

```
key_info = h'746869732d49532d736f6d652d6b65792d6d657461646174612d746f2d6
              2652d757365642d696e2d746573742d6b65792d67656e'
```

and the following key_dst value, defined by api_id || KEYGEN_DST_, where api_id the identifier of the BBS Interface defined in Section 3.5, using the BLS12-381-SHAKE-256 ciphersuite defined in Section 7.2.1, meaning that api_id = BBS_BLS12381G1_XOF:SHAKE-256_SSWU_RO_H2G_HM2S_,

```
key_dst = h'4242535f424c533132333383147315f584f463a5348414b452d3235365f53
             5357555f524f5f4832475f484d32535f4b455947454e5f4453545f'
```

Outputs the following SK value

```
 SK = 0x2eee0f60a8a3a8bec0ee942bfd46cbdae9a0738ee68f5a64e7238311cf09a079
```

Following the procedure defined in Section 3.4.2 with an input SK value as above produces the following PK value

```
PK = h'92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18
       fb0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179e
       b001963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5'
```

#### 8.3.2. Map Messages to Scalars

The messages in Section 8.2 are mapped to scalars during the Sign, Verify, ProofGen and ProofVerify operations. Presented below, are the output scalar values of the messages_to_scalars operation (Section 4.1.2), on input the messages defined in Section 8.2 and the api_id defined in Section 3.5, using the BLS12-381-SHAKE-256 ciphersuite defined in Section 7.2.1, meaning that api_id = BBS_BLS12381G1_XOF:SHAKE-256_SSWU_RO_H2G_HM2S_. Each output scalar value is encoded to octets using I2OSP and represented in big endian order,

```
msg_scalar_1 = 0x1e0dea6c9ea8543731d331a0ab5f64954c188542b33c5bbc8ae5b3a
                  830f2d99f
msg_scalar_2 = 0x3918a40fb277b4c796805d1371931e08a314a8bf8200a92463c0605
                  4d2c56a9f
msg_scalar_3 = 0x6642b981edf862adf34214d933c5d042bfa8f7ef343165c325131e2
                  ffa32fa94
msg_scalar_4 = 0x33c021236956a2006f547e22ff8790c9d2d40c11770c18cce603778
                  6c6f23512
msg_scalar_5 = 0x52b249313abbe323e7d84230550f448d99edfb6529dec8c4e783dbd
                  6dd2a8471
msg_scalar_6 = 0x2a50bdcbe7299e47e1046100aadffe35b4247bf3f059d525f921537
                  484dd54fc
msg_scalar_7 = 0x0e92550915e275f8cfd6da5e08e334d8ef46797ee28fa29de40a1eb
                  ccd9d95d3
msg_scalar_8 = 0x4c28f612e6c6f82f51f95e1e4faaf597547f93f6689827a6dcda3cb
                  94971d356
msg_scalar_9 = 0x1db51bedc825b85efe1dab3e3ab0274fa82bbd39732be3459525faf
                  70f197650
msg_scalar_10 = 0x27878da72f7775e709bb693d81b819dc4e9fa60711f4ea927740e4
                   0073489e78
```

#### 8.3.3. Message Generators

Following the procedure defined in Section 4.1.1 for the BLS12-381-SHAKE-256 (#bls12-381-shake-256) suite, with an input count value of 11 and an api_id value of api_id = BBS_BLS12381G1_XOF:SHAKE-256_SSWU_RO_H2G_HM2S_ (as defined in Section 3.5 for the BLS12-381-SHAKE-256 ciphersuite), outputs the following values (note that the first one corresponds to Q_1, while the next 10, to the message generators H_1, ..., H_10).

```
Q_1 = h'a9d40131066399fd41af51d883f4473b0dcd7d028d3d34ef17f3241d204e2850
        7d7ecae032afa1d5490849b7678ec1f8'
H_1 = h'903c7ca0b7e78a2017d0baf74103bd00ca8ff9bf429f834f071c75ffe6bfdec6
        d6dca15417e4ac08ca4ae1e78b7adc0e'
H_2 = h'84321f5855bfb6b001f0dfcb47ac9b5cc68f1a4edd20f0ec850e0563b27d2acc
        ee6edff1a26b357762fb24e8ddbb6fcb'
H_3 = h'b3060dff0d12a32819e08da00e61810676cc9185fdd750e5ef82b1a9798c7d76
        d63de3b6225d6c9a479d6c21a7c8bf93'
H_4 = h'8f1093d1e553cdead3c70ce55b6d664e5d1912cc9edfdd37bf1dad11ca396a0a
        8bb062092d391ebf8790ea5722413f68'
H_5 = h'990824e00b48a68c3d9a308e8c52a57b1bc84d1cf5d3c0f8c6fb6b1230e4e5b8
        eb752fb374da0b1ef687040024868140'
H_6 = h'b86d1c6ab8ce22bc53f625d1ce9796657f18060fcb1893ce8931156ef992fe56
        856199f8fa6c998e5d855a354a26b0dd'
H_7 = h'b4cdd98c5c1e64cb324e0c57954f719d5c5f9e8d991fd8e159b31c8d079c76a6
        7321a30311975c706578d3a0ddc313b7'
H_8 = h'8311492d43ec9182a5fc44a75419b09547e311251fe38b6864dc1e706e29446c
        b3ea4d501634eb13327245fd8a574f77'
H_9 = h'ac00b493f92d17837a28d1f5b07991ca5ab9f370ae40d4f9b9f2711749ca2001
        10ce6517dc28400d4ea25dddc146cacc'
H_10 = h'965a6c62451d4be6cb175dec39727dc665762673ee42bf0ac13a37a74784fbd
         61e84e0915277a6f59863b2bb4f5f6005'
```

#### 8.3.4. Signature Fixtures

This section presents test vectors for the Sign operation, as defined in Section 3.5.1, for the BLS12-381-SHAKE-256 ciphersuite (Section 7.2.1).

##### 8.3.4.1. Valid Single Message Signature

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
        5f02'

SK = 0x2eee0f60a8a3a8bec0ee942bfd46cbdae9a0738ee68f5a64e7238311cf09a079
PK = h'92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18
       fb0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179e
       b001963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5'
header = h'11223344556677889900aabbccddeeff'

B = h'8bbc8c123d3f128f206dd0d2dae490e82af08b84e8d70af3dc291d32a6e98f635b
      eefcc4533b2599804a164aabe68d7c'
domain = 0x2f18dd269c11c512256a9d1d57e61a7d2de6ebcf41cac3053f37afedc4e65
           0a9

signature = h'b9a622a4b404e6ca4c85c15739d2124a1deb16df750be202e2430e169b
              c27fb71c44d98e6d40792033e1c452145ada95030832c5dc778334f2f
              1b528eced21b0b97a12025a283d78b7136bb9825d04ef'
```

#### 8.3.4.2. Valid Multi-Message Signature

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
        5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
        9b80'
m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_6 = h'515ae153e22aae04ad16f759e07237b4'
m_7 = h'd183ddc6e2665aa4e2f088af'
m_8 = h'ac55fb33a75909ed'
m_9 = h'96012096'
m_10 = h''

SK = 0x2eee0f60a8a3a8bec0ee942bfd46cbdae9a0738ee68f5a64e7238311cf09a079
PK = h'92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18
       fb0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179e
       b001963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5'
header = h'11223344556677889900aabbccddeeff'

B = h'ae8d4ebe248b9ad9c933d5661bfb46c56721fba2a1182ddda7e8fb443bda3c0a57
      1ad018ad31d0b6d1f4e8b985e6c58d'
domain = 0x6f7ee8de30835599bb540d2cb4dd02fd0c6cf8246f14c9ee9a8463f7fd400
           f7b

signature = h'956a3427b1b8e3642e60e6a7990b67626811adeec7a0a6cb4f770cdd7c
              20cf08faabb913ac94d18e1e92832e924cb6e202912b624261fc6c59b
              0fea801547f67fb7d3253e1e2acbcf90ef59a6911931e'
```

### 8.3.5. Proof Fixtures

This section presents test vectors for the ProofGen operation, as defined in Section 3.5.3, for the BLS12-381-SHAKE-256 ciphersuite (Section 7.2.1).

For the generation of the following test vectors, the mocked_calculate_random_scalars defined in Section 8.1 is used, in place of the calculate_random_scalars operation, with the following SEED value (hex encoding of the ASCII-encoded 30 first digits of pi)

```
SEED =
     h'332e313431353932363533353839373933323338343632363433333833323739'
```

and the domain separation tag DST = api_id || "MOCK_RANDOM_SCALARS_DST_", where api_id is the identifier of the BBS Interface defined in Section 3.5, i.e., api_id = ciphersuite_id || H2G_HM2S_, where ciphersuite_id is the unique identifier of the

BLS12-381-SHAKE-256 ciphersuite as defined in Section 7.2.1 and "MOCK_RANDOM_SCALARS_DST_" is an ASCII string composed of 24 bytes. More specifically,

```
DST =
"BBS_BLS12381G1_XOF:SHAKE-256_SSWU_RO_H2G_HM2S_MOCK_RANDOM_SCALARS_DST_"
```

Given the above SEED and DST values, the first 10 scalars (i.e., with count = 10) returned by the mocked_calculate_random_scalars operation will be,

```
random_scalar_1 = 0x1004262112c3eaa95941b2b0d1311c09c845db0099a50e67eda6
                     28ad26b43083
random_scalar_2 = 0x6da7f145a94c1fa7f116b2482d59e4d466fe49c955ae8726e794
                     53065156a9a4
random_scalar_3 = 0x05017919b3607e78c51e8ec34329955d49c8c90e4488079c43e7
                     4824e98f1306
random_scalar_4 = 0x4d451dad519b6a226bba79e11b44c441f1a74800eecfec6a2e2d
                     79ea65b9d32d
random_scalar_5 = 0x5e7e4894e6dbe68023bc92ef15c410b01f3828109fc72b3b5ab1
                     59fc427b3f51
random_scalar_6 = 0x646e3014f49accb375253d268eb6c7f3289a1510f1e9452b612d
                     d73a06ec5dd4
random_scalar_7 = 0x363ecc4c1f9d6d9144374de8f1f7991405e3345a3ec49dd485a3
                     9982753c11a4
random_scalar_8 = 0x12e592fe28d91d7b92a198c29afaa9d5329a4dcfdaf8b0855780
                     7412faeb4ac6
random_scalar_9 = 0x513325acdcdec7ea572360587b350a8b095ca19bdd8258c5c69d
                     375e8706141a
random_scalar_10 = 0x6474fceba35e7e17365dde1a0284170180e446ae96c82943290
                      d7baa3a6ed429
```

#### 8.3.5.1. Valid Single Message Proof

```
m_0 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'

public_key = h'92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd459
             49cdeb18fb0490edcd4429adff56e65cbce42cf188b31bddbd619e41
             9b99c2c41b38179eb001963bc3decaae0d9f702c7a8c004f207f46c7
             34a5eae2e8e82833f3e7ea5'
signature = h'b9a622a4b404e6ca4c85c15739d2124a1deb16df750be202e2430e169b
            c27fb71c44d98e6d40792033e1c452145ada95030832c5dc778334f2f
            1b528eced21b0b97a12025a283d78b7136bb9825d04ef'
header = h'11223344556677889900aabbccddeeff'
presentation_header = h'bed231d880675ed101ead304512e043ade9958dd0241ea70
                      b4b3957fba941501'
revealed_indexes = [ 0 ]

random scalars:
    r1 = 0x1308e6f945f663b96de1c76461cf7d7f88b92eb99a9034685150db443d733
         881
    r2 = 0x25f81cb69a8fac6fb55d44a084557258575d1003be2bd94f1922dad2c3e44
         7fd
    e_tilde = 0x5e8041a7ab02976ee50226c4b062b47d38829bbf42ee7eb899b29720
              377a584c
    r1_tilde = 0x3bbf1d5dc2904dbb7b2ba75c5dce8a5ad2d56a359c13ff0fa5fcb13
               39cd2fe58
    r3_tilde = 0x016b1460eee7707c524a86a4aedeb826ce9597b42906dccaa96c6b4
               9a8ea7da2
    m_tilde_scalars: [  ]

T1 = h'91a10e73cf4090812e8ea25f31aaa61be53fcb42ce86e9f0e5df6f6dac4c3eee6
     2ac846b0b83a5cfcbe78315175a4961'
T2 = h'988f3d473186634e41478dc4527cf240e64de23a763037454d39a876862ebc617
     738ba6c458142e3746b01eab58ca8d7'
domain = 0x2f18dd269c11c512256a9d1d57e61a7d2de6ebcf41cac3053f37afedc4e65
         0a9

proof = h'89e4ab0c160880e0c2f12a754b9c051ed7f5fccfee3d5cbbb62e1239709196
        c737fff4303054660f8fcd08267a5de668a2e395ebe8866bdcb0dff9786d7
        014fa5e3c8cf7b41f8d7510e27d307f18032f6b788e200b9d6509f40ce1d2
        f962ceedb023d58ee44d660434e6ba60ed0da1a5d2cde031b483684cd7c5b
        13295a82f57e209b584e8fe894bcc964117bf3521b43d8e2eb59ce31f34d6
        8b39f05bb2c625e4de5e61e95ff38bfd62ab07105d016414b45b01625c699
        65ad3c8a933e7b25d93daeb777302b966079827a99178240e6c3f13b7db2f
        b1f14790940e239d775ab32f539bdf9f9b582b250b05882996832652f7f5d
        3b6e04744c73ada1702d6791940ccbd75e719537f7ace6ee817298d'
```

8.3.5.2. Valid Multi-Message, All Messages Disclosed Proof

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
      9b80'
m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_6 = h'515ae153e22aae04ad16f759e07237b4'
m_7 = h'd183ddc6e2665aa4e2f088af'
m_8 = h'ac55fb33a75909ed'
m_9 = h'96012096'
m_10 = h''

public_key = h'92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd459
               49cdeb18fb0490edcd4429adff56e65cbce42cf188b31bddbd619e41
               9b99c2c41b38179eb001963bc3decaae0d9f702c7a8c004f207f46c7
               34a5eae2e8e82833f3e7ea5'
signature = h'956a3427b1b8e3642e60e6a7990b67626811adeec7a0a6cb4f770cdd7c
              20cf08faabb913ac94d18e1e92832e924cb6e202912b624261fc6c59b
              0fea801547f67fb7d3253e1e2acbcf90ef59a6911931e'
header = h'11223344556677889900aabbccddeeff'
presentation_header = h'bed231d880675ed101ead304512e043ade9958dd0241ea70
                        b4b3957fba941501'
revealed_indexes = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]

random scalars:
    r1 = 0x1308e6f945f663b96de1c76461cf7d7f88b92eb99a9034685150db443d733
         881
    r2 = 0x25f81cb69a8fac6fb55d44a084557258575d1003be2bd94f1922dad2c3e44
         7fd
    e_tilde = 0x5e8041a7ab02976ee50226c4b062b47d38829bbf42ee7eb899b29720
              377a584c
    r1_tilde = 0x3bbf1d5dc2904dbb7b2ba75c5dce8a5ad2d56a359c13ff0fa5fcb13
               39cd2fe58
    r3_tilde = 0x016b1460eee7707c524a86a4aedeb826ce9597b42906dccaa96c6b4
               9a8ea7da2
    m_tilde_scalars: [  ]

T1 = h'8890adfc78da24768d59dbfdb3f380e2793e9018b20c23e9ba05baa60f1b21456
       bc047a5d27049dab5dc6a94696ce711'
T2 = h'a49f953636d3651a3ae6fe45a99a2e4fec079eef3be8b8a6a4ba70885d7e02864
       2f7224e9f451529915c88a7edc59fbe'
domain = 0x6f7ee8de30835599bb540d2cb4dd02fd0c6cf8246f14c9ee9a8463f7fd400
         f7b

proof = h'91b0f598268c57b67bc9e55327c3c2b9b1654be89a0cf963ab392fa9e1637c
          565241d71fd6d7bbd7dfe243de85a9bac8b7461575c1e13b5055fed0b51fd
          0ec1433096607755b2f2f9ba6dc614dfa456916ca0d7fc6482b39c679cfb7
```

```
            47a50ea1b3dd7ed57aaadc348361e2501a17317352e555a333e014e8e7d71
            eef808ae4f8fbdf45cd19fde45038bb310d5135f5205fc550b077e381fb3a
            3543dca31a0d8bba97bc0b660a5aa239eb74921e184aa3035fa01eaba32f5
            2029319ec3df4fa4a4f716edb31a6ce19a19dbb971380099345070bd0fdee
            cf7c4774a33e0a116e069d5e215992fb637984802066dee6919146ae50b70
            ea52332dfe57f6e05c66e99f1764d8b890d121d65bfcc2984886ee0'
```

#### 8.3.5.3. Valid Multi-Message, Some Messages Disclosed Proof

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
        5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
        9b80'
m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_6 = h'515ae153e22aae04ad16f759e07237b4'
m_7 = h'd183ddc6e2665aa4e2f088af'
m_8 = h'ac55fb33a75909ed'
m_9 = h'96012096'
m_10 = h''

public_key = h'92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd459
                49cdeb18fb0490edcd4429adff56e65cbce42cf188b31bddbd619e41
                9b99c2c41b38179eb001963bc3decaae0d9f702c7a8c004f207f46c7
                34a5eae2e8e82833f3e7ea5'
signature = h'956a3427b1b8e3642e60e6a7990b67626811adeec7a0a6cb4f770cdd7c
               20cf08faabb913ac94d18e1e92832e924cb6e202912b624261fc6c59b
               0fea801547f67fb7d3253e1e2acbcf90ef59a6911931e'
header = h'11223344556677889900aabbccddeeff'
presentation_header = h'bed231d880675ed101ead304512e043ade9958dd0241ea70
                        b4b3957fba941501'
revealed_indexes = [ 0, 2, 4, 6 ]

random scalars:
    r1 = 0x5ee9426ae206e3a127eb53c79044bc9ed1b71354f8354b01bf410a02220be
           7d0
    r2 = 0x280d4fcc38376193ffc777b68459ed7ba897e2857f938581acf95ae5a6898
           8f3
    e_tilde = 0x39966b00042fc43906297d692ebb41de08e36aada8d9504d4e0ae02a
                d59e9230
    r1_tilde = 0x61f5c273999b0b50be8f84d2380eb9220fc5a88afe144efc4007545
                 f0ab9c089
    r3_tilde = 0x63af117e0c8b7d2f1f3e375fcf5d9430e136ff0f7e879423e49dadc
                 401a50089
    m_tilde_scalars:
        m~_1 = 0x020b83ca2ab319cba0744d6d58da75ac3dfb6ba682bfce2587c5a6d
                 86a4e4e7b
```

```
        m~_3 = 0x5bf565343611c08f83e4420e8b1577ace8cc4df5d5303aeb3c4e425
                  f1080f836
        m~_5 = 0x049d77949af1192534da28975f76d4f211315dce1e36f93ffcf2a55
                  5de516b28
        m~_7 = 0x407e5a952f145de7da53533de8366bbd2e0c854721a204f03906dc8
                  2fde10f48
        m~_8 = 0x1c925d9052849edddcf04d5f1f0d4ff183a66b66eb820f59b675aee
                  121cfc63c
        m~_9 = 0x07d7c41b02158a9c5eac212ed6d7c2cddeb8e38baea6e93e1a00b2e
                  83e2a0995

T1 = h'8b497dd4dcdcf7eb58c9b43e57e06bcea3468a223ae2fc015d7a86506a952d680
        55e73f5a5847e58f133ea154256d0da'
T2 = h'8655584d3da1313f881f48c239384a5623d2d292f08dae7ac1d8129c19a02a89b
        82fa45de3f6c2c439510fce5919656f'
domain = 0x6f7ee8de30835599bb540d2cb4dd02fd0c6cf8246f14c9ee9a8463f7fd400
           f7b

proof = h'b1f8bf99a11c39f04e2a032183c1ead12956ad322dd06799c50f20fb8cf6b0
           ac279210ef5a2920a7be3ec2aa0911ace7b96811a98f3c1cceba4a2147ae7
           63b3ba036f47bc21c39179f2b395e0ab1ac49017ea5b27848547bedd27be4
           81c1dfc0b73372346feb94ab16189d4c525652b8d3361bab43463700720ec
           fb0ee75e595ea1b13330615011050a0dfcffdb21af356dd39bf8bcbfd41bf
           95d913f4c9b2979e1ed2ca10ac7e881bb6a271722549681e398d29e9ba4ea
           c8848b168eddd5e4acec7df4103e2ed165e6e32edc80f0a3b28c36fb39ca1
           9b4b8acee570deadba2da9ec20d1f236b571e0d4c2ea3b826fe924175ed4d
           fffbf18a9cfa98546c241efb9164c444d970e8c89849bc8601e96cf228fde
           fe38ab3b7e289cac859e68d9cbb0e648faf692b27df5ff6539c30da17e544
           4a65143de02ca64cee7b0823be65865cdc310be038ec6b594b99280072ae0
           67bad1117b0ff3201a5506a8533b925c7ffae9cdb64558857db0ac5f5e0f1
           8e750ae77ec9cf35263474fef3f78138c7a1ef5cfbc878975458239824fad
           3ce05326ba3969b1f5451bd82bd1f8075f3d32ece2d61d89a064ab4804c3c
           892d651d11bc325464a71cd7aacc2d956a811aaff13ea4c35cef7842b656e
           8ba4758e7558'
```

### 8.4. BLS12381-SHA-256 Test Vectors

Test vectors of the BLS12-381-SHA-256 (#bls12-381-sha-256-ciphersuite) ciphersuite. Further fixtures are available in Appendix D.2.

#### 8.4.1. Key Pair

Following the procedure defined in Section 3.4.1 with an input key_material value as follows

```
key_material = h'746869732d49532d6a7573742d616e2d546573742d494b4d2d746f2
                  d67656e65726174652d246528724074232d6b6579'
```

the following key_info value

```
key_info = h'746869732d49532d736f6d652d6b65792d6d657461646174612d746f2d6
             2652d757365642d696e2d746573742d6b65792d67656e'
```

and the following key_dst value, defined by api_id || KEYGEN_DST_, where api_id the identifier of the BBS Interface defined in Section 3.5, using the BLS12-381-SHA-256 ciphersuite defined in Section 7.2.2, meaning that api_id = BBS_BLS12381G1_XMD:SHA-256_SSWU_RO_H2G_HM2S_,

```
key_dst = h'4242535f424c533132333383147315f584d443a5348412d3235365f535357
            555f524f5f4832475f484d32535f4b455947454e5f4453545f'
```

Outputs the following SK value

```
 SK = 0x60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc
```

Following the procedure defined in Section 3.4.2 with an input SK value as above produces the following PK value

```
PK = h'a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f285
       1bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1
       e7c4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c'
```

### 8.4.2. Map Messages to Scalars

The messages in Section 8.2 are mapped to scalars during the Sign, Verify, ProofGen and ProofVerify operations. Presented below, are the output scalar values of the messages_to_scalars operation (Section 4.1.2), on input the messages defined in Section 8.2 and the api_id defined in Section 3.5, using the BLS12-381-SHA-256 ciphersuite defined in Section 7.2.2, meaning that api_id = BBS_BLS12381G1_XMD:SHA-256_SSWU_RO_H2G_HM2S_. Each output scalar value is encoded to octets using I2OSP and represented in big endian order,

```
dst = h'4242535f424c533132333383147315f584d443a5348412d3235365f535357555f
        524f5f4832475f484d32535f4d41505f4d53475f544f5f5343414c41525f415
        35f484153485f'
```

The output scalars, encoded to octets using I2OSP and represented in big endian order, are the following,

```
msg_scalar_1 = 0x1cb5bb86114b34dc438a911617655a1db595abafac92f47c5001799
                  cf624b430
msg_scalar_2 = 0x154249d503c093ac2df516d4bb88b510d54fd97e8d7121aede420a2
                  5d9521952
msg_scalar_3 = 0x0c7c4c85cdab32e6fdb0de267b16fa3212733d4e3a3f0d0f7516575
                  78b26fe22
msg_scalar_4 = 0x4a196deafee5c23f630156ae13be3e46e53b7e39094d22877b8cba7
                  f14640888
msg_scalar_5 = 0x34c5ea4f2ba49117015a02c711bb173c11b06b3f1571b88a2952b93
                  d0ed4cf7e
msg_scalar_6 = 0x4045b39b83055cd57a4d0203e1660800fabe434004dbdc8730c21ce
                  3f0048b08
msg_scalar_7 = 0x064621da4377b6b1d05ecc37cf3b9dfc94b9498d7013dc5c4a82bf3
                  bb1750743
msg_scalar_8 = 0x34ac9196ace0a37e147e32319ea9b3d8cc7d21870d3c3ba07124685
                  9cca49b02
msg_scalar_9 = 0x57eb93f417c43200e9784fa5ea5a59168d3dbc38df707a13bb597c8
                  71b2a5f74
msg_scalar_10 = 0x08e3afeb2b4f2b5f907924ef42856616e6f2d5f1fb373736db1cca
                   32707a7d16
```

#### 8.4.3. Message Generators

Following the procedure defined in Section 4.1.1 for the BLS12-381-SHA-256 (#bls12-381-sha-256) suite, with an input count value of 11 and an api_id value of api_id = BBS_BLS12381G1_XMD:SHA-256_SSWU_RO_H2G_HM2S_ (as defined in Section 3.5 for the BLS12-381-SHA-256 ciphersuite), outputs the following values (note that the first one corresponds to Q_1, while the next 10, to the message generators H_1, ..., H_10).

```
Q_1 = h'a9ec65b70a7fbe40c874c9eb041c2cb0a7af36ccec1bea48fa2ba4c2eb67ef7f
        9ecb17ed27d38d27cdeddff44c8137be'
H_1 = h'98cd5313283aaf5db1b3ba8611fe6070d19e605de4078c38df36019fbaad0bd2
        8dd090fd24ed27f7f4d22d5ff5dea7d4'
H_2 = h'a31fbe20c5c135bcaa8d9fc4e4ac665cc6db0226f35e737507e803044093f376
        97a9d452490a970eea6f9ad6c3dcaa3a'
H_3 = h'b479263445f4d2108965a9086f9d1fdc8cde77d14a91c856769521ad3344754c
        c5ce90d9bc4c696dffbc9ef1d6ad1b62'
H_4 = h'ac0401766d2128d4791d922557c7b4d1ae9a9b508ce266575244a8d6f32110d7
        b0b7557b77604869633bb49afbe20035'
H_5 = h'b95d2898370ebc542857746a316ce32fa5151c31f9b57915e308ee9d1de7db69
        127d919e984ea0747f5223821b596335'
H_6 = h'8f19359ae6ee508157492c06765b7df09e2e5ad591115742f2de9c08572bb284
        5cbf03fd7e23b7f031ed9c7564e52f39'
H_7 = h'abc914abe2926324b2c848e8a411a2b6df18cbe7758db8644145fefb0bf0a2d5
        58a8c9946bd35e00c69d167aadf304c1'
H_8 = h'80755b3eb0dd4249cbefd20f177cee88e0761c066b71794825c9997b551f2405
        1c352567ba6c01e57ac75dff763eaa17'
H_9 = h'82701eb98070728e1769525e73abff1783cedc364adb20c05c897a62f2ab2927
        f86f118dcb7819a7b218d8f3fee4bd7f'
H_10 = h'a1f229540474f4d6f1134761b92b788128c7ac8dc9b0c52d594931326796730
         32ac7db3fb3d79b46b13c1c41ee495bca'
```

#### 8.4.4. Signature Fixtures

This section presents test vectors for the Sign operation, as defined in Section 3.5.1, for the BLS12-381-SHA-256 ciphersuite (Section 7.2.2).

##### 8.4.4.1. Valid Single Message Signature

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
        5f02'

SK = 0x60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc
PK = h'a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f285
       1bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1
       e7c4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c'
header = h'11223344556677889900aabbccddeeff'

B = h'92d264aed02bf23de022ebe778c4f929fddf829f504e451d011ed89a313b8167ac
      947332e1648157ceffc6e6e41ab255'
domain = 0x25d57fab92a8274c68fde5c3f16d4b275e4a156f211ae34b3ab32fbaf506e
           d5c

signature = h'84773160b824e194073a57493dac1a20b667af70cd2352d8af241c7765
              8da5253aa8458317cca0eae615690d55b1f27164657dcafee1d5c1973
              947aa70e2cfbb4c892340be5969920d0916067b4565a0'
```

#### 8.4.4.2. Valid Multi-Message Signature

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
        5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
        9b80'
m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_6 = h'515ae153e22aae04ad16f759e07237b4'
m_7 = h'd183ddc6e2665aa4e2f088af'
m_8 = h'ac55fb33a75909ed'
m_9 = h'96012096'
m_10 = h''

SK = 0x60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc
PK = h'a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f285
       1bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1
       e7c4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c'
header = h'11223344556677889900aabbccddeeff'

B = h'84f48376f7df6af40bc329cf484cdbfd0b19d0b326fccab4e9d8f00d1dbcf48139
      d498b19667f203cf8a1d1f8340c522'
domain = 0x6272832582a0ac96e6fe53e879422f24c51680b25fbf17bad22a35ea93ce5
           b47

signature = h'8339b285a4acd89dec7777c09543a43e3cc60684b0a6f8ab335da4825c
              96e1463e28f8c5f4fd0641d19cec5920d3a8ff4bedb6c9691454597bb
              d298288abed3632078557b2ace7d44caed846e1a0a1e8'
```

### 8.4.5. Proof Fixtures

This section presents test vectors for the ProofGen operation, as defined in Section 3.5.3, for the BLS12-381-SHA-256 ciphersuite (Section 7.2.1).

For the generation of the following test vectors, the mocked_calculate_random_scalars defined in Section 8.1 is used, in place of the calculate_random_scalars operation, with the following SEED value (hex encoding of the ASCII-encoded 30 first digits of pi)

```
SEED =
     h'332e313431353932363533353839373933323338343632363433333833323739'
```

and the domain separation tag DST = api_id || "MOCK_RANDOM_SCALARS_DST_", where api_id is the identifier of the BBS Interface defined in Section 3.5, i.e., api_id = ciphersuite_id || H2G_HM2S_, where ciphersuite_id is the unique identifier of the

BLS12-381-SHA-256 ciphersuite as defined in Section 7.2.2 and "MOCK_RANDOM_SCALARS_DST_" is an ASCII string composed of 24 bytes. More specifically,

```
DST =
  "BBS_BLS12381G1_XMD:SHA-256_SSWU_RO_H2G_HM2S_MOCK_RANDOM_SCALARS_DST_"
```

Given the above SEED and DST values, the first 10 scalars (i.e., with count = 10) returned by the mocked_calculate_random_scalars operation will be,

```
random_scalar_1 = 0x04f8e2518993c4383957ad14eb13a023c4ad0c67d01ec86eeb90
                     2e732ed6df3f
random_scalar_2 = 0x5d87c1ba64c320ad601d227a1b74188a41a100325cecf0022372
                     9863966392b1
random_scalar_3 = 0x0444607600ac70482e9c983b4b063214080b9e808300aa4cc02a
                     91b3a92858fe
random_scalar_4 = 0x548cd11eae4318e88cda10b4cd31ae29d41c3a0b057196ee9cf3
                     a69d471e4e94
random_scalar_5 = 0x2264b06a08638b69b4627756a62f08e0dc4d8240c1b974c9c7db
                     779a769892f4
random_scalar_6 = 0x4d99352986a9f8978b93485d21525244b21b396cf61f1d71f7c4
                     8e3fbc970a42
random_scalar_7 = 0x5ed8be91662386243a6771fbdd2c627de31a44220e8d6f745bad
                     5d99821a4880
random_scalar_8 = 0x62ff1734b939ddd87beeb37a7bbcafa0a274cbc1b07384198f0e
                     88398272208d
random_scalar_9 = 0x05c2a0af016df58e844db8944082dcaf434de1b1e2e7136ec8a9
                     9b939b716223
random_scalar_10 = 0x485e2adab17b76f5334c95bf36c03ccf91cef77dcfcdc6b8a69
                      e2090b3156663
```

Note that the returned scalars will be unique for different count values, i.e., for different output lengths.

#### 8.4.5.1. Valid Single Message Proof

```
m_0 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'

public_key = h'a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8f
             a136f2851bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91a
             a8d460acee0e96f1e7c4cfb12d3ff9ab5d5dc91c277db75c845d649e
             f3c4f63aebc364cd55ded0c'
signature = h'84773160b824e194073a57493dac1a20b667af70cd2352d8af241c7765
            8da5253aa8458317cca0eae615690d55b1f27164657dcafee1d5c1973
            947aa70e2cfbb4c892340be5969920d0916067b4565a0'
header = h'11223344556677889900aabbccddeeff'
presentation_header = h'bed231d880675ed101ead304512e043ade9958dd0241ea70
                      b4b3957fba941501'
revealed_indexes = [ 0 ]

random scalars:
    r1 = 0x60ca409f6b0563f687fc471c63d2819f446f39c23bb540925d9d4254ac58f
         337
    r2 = 0x2ceff4982de0c913090f75f081df5ec594c310bb48c17cfdaab5332a682ef
         811
    e_tilde = 0x6101c4404895f3dff87ab39c34cb995af07e7139e6b3847180ffdd1b
              c8c313cd
    r1_tilde = 0x0dfcffd97a6ecdebef3c9c114b99d7a030c998d938905f357df6282
               2dee072e8
    r3_tilde = 0x639e3417007d38e5d34ba8c511e836768ddc2669fdd3faff5c14ad2
               7ac2b2da1
    m_tilde_scalars: [  ]

T1 = h'a862fa5d3ab4c264c22b8a02636fd4030e8b14ac20dee14e08fdb6cfc445432c0
     8abb49ec111c1eb9d90abef50134a60'
T2 = h'ab9543a6b04303e997621d3d5cbd85924e7e69da498a2a9e9d3a8b01f39259c9c
     5920bd530de1d3b0afb99eb0c549d5a'
domain = 0x25d57fab92a8274c68fde5c3f16d4b275e4a156f211ae34b3ab32fbaf506e
         d5c

proof = h'94916292a7a6bade28456c601d3af33fcf39278d6594b467e128a3f83686a1
        04ef2b2fcf72df0215eeaf69262ffe8194a19fab31a82ddbe06908985abc4
        c9825788b8a1610942d12b7f5debbea8985296361206dbace7af0cc834c80
        f33e0aadaeea5597befbb651827b5eed5a66f1a959bb46cfd5ca1a817a144
        75960f69b32c54db7587b5ee3ab665fbd37b506830a49f21d592f5e634f47
        cee05a025a2f8f94e73a6c15f02301d1178a92873b6e8634bafe4983c3e15
        a663d64080678dbf29417519b78af042be2b3e1c4d08b8d520ffab008cbaa
        ca5671a15b22c239b38e940cfeaa5e72104576a9ec4a6fad78c532381aeaa
        6fb56409cef56ee5c140d455feeb04426193c57086c9b6d397d9418'
```

#### 8.4.5.2. Valid Multi-Message, All Messages Disclosed Proof

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
      9b80'
m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_6 = h'515ae153e22aae04ad16f759e07237b4'
m_7 = h'd183ddc6e2665aa4e2f088af'
m_8 = h'ac55fb33a75909ed'
m_9 = h'96012096'
m_10 = h''

public_key = h'a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8f
             a136f2851bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91a
             a8d460acee0e96f1e7c4cfb12d3ff9ab5d5dc91c277db75c845d649e
             f3c4f63aebc364cd55ded0c'
signature = h'8339b285a4acd89dec7777c09543a43e3cc60684b0a6f8ab335da4825c
            96e1463e28f8c5f4fd0641d19cec5920d3a8ff4bedb6c9691454597bb
            d298288abed3632078557b2ace7d44caed846e1a0a1e8'
header = h'11223344556677889900aabbccddeeff'
presentation_header = h'bed231d880675ed101ead304512e043ade9958dd0241ea70
                      b4b3957fba941501'
revealed_indexes = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]

random scalars:
    r1 = 0x60ca409f6b0563f687fc471c63d2819f446f39c23bb540925d9d4254ac58f
         337
    r2 = 0x2ceff4982de0c913090f75f081df5ec594c310bb48c17cfdaab5332a682ef
         811
    e_tilde = 0x6101c4404895f3dff87ab39c34cb995af07e7139e6b3847180ffdd1b
              c8c313cd
    r1_tilde = 0x0dfcffd97a6ecdebef3c9c114b99d7a030c998d938905f357df6282
               2dee072e8
    r3_tilde = 0x639e3417007d38e5d34ba8c511e836768ddc2669fdd3faff5c14ad2
               7ac2b2da1
    m_tilde_scalars: [  ]

T1 = h'9881efa96b2411626d490e399eb1c06badf23c2c0760bd403f50f45a6b470c5a9
       dbeef53a27916f2f165085a3878f1f4'
T2 = h'b9f8cf9271d10a04ae7116ad021f4b69c435d20a5af10ddd8f5b1ec6b9b8b9160
       5aca76a140241784b7f161e21dfc3e7'
domain = 0x6272832582a0ac96e6fe53e879422f24c51680b25fbf17bad22a35ea93ce5
         b47

proof = h'b1f468aec2001c4f54cb56f707c6222a43e5803a25b2253e67b2210ab2ef9e
          ab52db2d4b379935c4823281eaf767fd37b08ce80dc65de8f9769d27099ae
          649ad4c9b4bd2cc23edcba52073a298087d2495e6d57aaae051ef741adf1c
```

```
            bce65c64a73c8c97264177a76c4a03341956d2ae45ed3438ce598d5cda4f1
            bf9507fecef47855480b7b30b5e4052c92a4360110c67327365763f5aa9fb
            85ddcbc2975449b8c03db1216ca66b310f07d0ccf12ab460cdc6003b677fe
            d36d0a23d0818a9d4d098d44f749e91008cf50e8567ef936704c8277b7710
            f41ab7e6e16408ab520edc290f9801349aee7b7b4e318e6a76e028e1dea91
            1e2e7baec6a6a174da1a22362717fbae1cd961d7bf4adce1d31c2ab'
```

#### 8.4.5.3. Valid Multi-Message, Some Messages Disclosed Proof

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
       5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
       9b80'
m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_6 = h'515ae153e22aae04ad16f759e07237b4'
m_7 = h'd183ddc6e2665aa4e2f088af'
m_8 = h'ac55fb33a75909ed'
m_9 = h'96012096'
m_10 = h''

public_key = h'a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8f
              a136f2851bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91a
              a8d460acee0e96f1e7c4cfb12d3ff9ab5d5dc91c277db75c845d649e
              f3c4f63aebc364cd55ded0c'
signature = h'8339b285a4acd89dec7777c09543a43e3cc60684b0a6f8ab335da4825c
             96e1463e28f8c5f4fd0641d19cec5920d3a8ff4bedb6c9691454597bb
             d298288abed3632078557b2ace7d44caed846e1a0a1e8'
header = h'11223344556677889900aabbccddeeff'
presentation_header = h'bed231d880675ed101ead304512e043ade9958dd0241ea70
                        b4b3957fba941501'
revealed_indexes = [ 0, 2, 4, 6 ]

random scalars:
    r1 = 0x44679831fe60eca50938ef0e812e2a9284ad7971b6932a38c7303538b712e
           457
    r2 = 0x6481692f89086cce11779e847ff884db8eebb85a13e81b2d0c79d6c106206
           9d8
    e_tilde = 0x721ce4c4c148a1d5826f326af6fd6ac2844f29533ba4127c3a43d222
                d51b7081
    r1_tilde = 0x1ecfaf5a079b0504b00a1f0d6fe8857291dd798291d7ad7454b3981
                 14393f37f
    r3_tilde = 0x0a4b3d59b34707bb9999bc6e2a6d382a2d2e214bff36ecd88639a14
                 124b1622e
    m_tilde_scalars:
        m~_1 = 0x7217411a9e329c7a5705e8db552274646e2949d62c288d7537dd62b
                 c284715e4
```

```
        m~_3 = 0x67d4d43660746759f598caac106a2b5f58ccd1c3eefaec31841a4f7
               7d2548870
        m~_5 = 0x715d965b1c3912d20505b381470ff1a528700b673e50ba89fd287e1
               3171cc137
        m~_7 = 0x4d3281a149674e58c9040fc7a10dd92cb9c7f76f6f0815a1afc3b09
               d74b92fe4
        m~_8 = 0x438feebaa5894ca0da49992df2c97d872bf153eab07e08ff73b2813
               1c46ff415
        m~_9 = 0x602b723c8bbaec1b057d70f18269ae5e6de6197a5884967b03b933f
               a80006121

T1 = h'84719c2b5bb275ee74913dbf95fb9054f690c8e4035f1259e184e9024544bc4bb
        ea9c244e7897f9db7c82b7b14b27d28'
T2 = h'8f5f191c956aefd5c960e57d2dfbab6761eb0ebc5efdba1aca1403dcc19e05296
        b16c9feb7636cb4ef2a360c5a148483'
domain = 0x6272832582a0ac96e6fe53e879422f24c51680b25fbf17bad22a35ea93ce5
           b47

proof = h'a2ed608e8e12ed21abc2bf154e462d744a367c7f1f969bdbf784a2a134c7db
           2d340394223a5397a3011b1c340ebc415199462ba6f31106d8a6da8b513b3
           7a47afe93c9b3474d0d7a354b2edc1b88818b063332df774c141f7a07c48f
           e50d452f897739228c88afc797916dca01e8f03bd9c5375c7a7c59996e514
           bb952a436afd24457658acbaba5ddac2e693ac481356918cd38025d86b286
           50e909defe9604a7259f44386b861608be742af7775a2e71a6070e5836f5f
           54dc43c60096834a5b6da295bf8f081f72b7cdf7f3b4347fb3ff19edaa9e7
           4055c8ba46dbcb7594fb2b06633bb5324192eb9be91be0d33e453b4d31274
           59de59a5e2193c900816f049a02cb9127dac894418105fa1641d5a206ec9c
           42177af9316f433417441478276ca0303da8f941bf2e0222a43251cf5c2bf
           6eac1961890aa740534e519c1767e1223392a3a286b0f4d91f7f25217a786
           2b8fcc1810cdcfddde2a01c80fcc90b632585fec12dc4ae8fea1918e9ddeb
           9414623a457e88f53f545841f9d5dcb1f8e160d1560770aa79d65e2eca8ed
           eaecb73fb7e995608b820c4a64de6313a370ba05dc25ed7c1d18519208496
           3652f2870341bdaa4b1a37f8c06348f38a4f80c5a2650a21d59f09e8305dc
           d3fc3ac30e2a'
```

## 9. IANA Considerations

This document does not make any requests of IANA.

## 10. Acknowledgements

The authors would like to acknowledge the significant amount of academic work that preceded the development of this document. In particular the original work of [BBS04] which was subsequently developed in [ASM06] [CL04] [BBDT16] [CDL16] and in [TZ23]. This last academic work is the one mostly used by this document.

The current state of this document is the product of the work of the Decentralized Identity Foundation Applied Cryptography Working group, which includes numerous active participants. In particular, the following individuals contributed ideas, feedback and wording that influenced this specification:

Orie Steele, Christian Paquin, Alessandro Guggino, Tomislav Markovski and Greg Bernstein.

Additionally, the authors would like to acknowledge Jacques Traore and Antoine Dumanois, for their crucial contributions to this document.

## 11. Normative References

[DRBG] NIST, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators", <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[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/info/rfc8017>.

[RFC8937] Cremers, C., Garratt, L., Smyshlyaev, S., Sullivan, N., and C. Wood, "Randomness Improvements for Security Protocols", RFC 8937, DOI 10.17487/RFC8937, October 2020, <https://www.rfc-editor.org/info/rfc8937>.

[RFC9380] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", RFC 9380, DOI 10.17487/RFC9380, August 2023, <https://www.rfc-editor.org/info/rfc9380>.

[SHA2] NIST, "Secure Hash Standard (SHS)", <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>.

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

## 12. Informative References

[ADR02] An, J. H., Dodis, Y., and T. Rabin, "On the Security of Joint Signature and Encryption", In EUROCRYPT, pages 83-107, April 2002, <https://doi.org/10.1007/3-540-46035-7_6>.

[ASM06] Au, M. H., Susilo, W., and Y. Mu, "Constant-Size Dynamic k-TAA", In International Conference on Security and Cryptography for Networks, pages 111-125, Springer, Berlin, Heidelberg, 2006, <https://link.springer.com/chapter/10.1007/11832072_8>.

[BBB17] Bunz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., and G. Maxwell, "Bulletproofs: Short Proofs for Confidential Transactions and More", In 2018 IEEE Symposium on Security and Privacy, 2017, <https://ia.cr/2017/1066>.

[BBDT16] Barki, A., Brunet, S., Desmoulins, N., and J. Traore, "Improved Algebraic MACs and Practical Keyed-Verification Anonymous Credentials", In International Conference on Selected Areas in Cryptography, 1016, <https://link.springer.com/chapter/10.1007/978-3-319-69453-5_20>.

[BBS04] Boneh, D., Boyen, X., and H. Shacham, "Short Group Signatures", In Advances in Cryptology, pages 41-55, 2004, <https://link.springer.com/chapter/10.1007/978-3-540-28628-8_3>.

[Bowe19] Bowe, S., "Faster subgroup checks for BLS12-381", July 2019, <https://eprint.iacr.org/2019/814>.

[CDL16] Camenisch, J., Drijvers, M., and A. Lehmann, "Anonymous Attestation Using the Strong Diffie Hellman Assumption Revisited", In International Conference on Trust and Trustworthy Computing, pages 1-20, Springer, Cham, 2016, <https://eprint.iacr.org/2016/663.pdf>.

[CL04] Camenisch, J. and A. Lysyanskaya, "Signature Schemes and Anonymous Credentials from Bilinear Maps", In Annual International Cryptology Conference, pages 56-72, 2004, <https://link.springer.com/chapter/10.1007/978-3-540-28628-8_4>.

[DMS04] Dingledine, R., Mathewson, N., and P. Syverson, "Tor: The Second-Generation Onion Router", 2004, <https://svn-archive.torproject.org/svn/projects/design-paper/tor-design.html>.

[HDWH12] Heninger, N., Durumeric, Z., Wustrow, E., and J. A. Halderman, "Mining your Ps and Qs: Detection of widespread weak keys in network devices", In USENIX Security, pages 205-220, August 2012, <https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final228.pdf>.

[I-D.ietf-jose-json-web-proof] Waite, D., Jones, M. B., and J. Miller, "JSON Web Proof", Work in Progress, Internet-Draft, draft-ietf-jose-json-web-proof-12, 4 November 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-jose-json-web-proof-12>.

[I-D.ietf-lwig-curve-representations] Struik, R., "Alternative Elliptic Curve Representations", Work in Progress, Internet-Draft, draft-ietf-lwig-curve-representations-23, 21 January 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-lwig-curve-representations-23>.

[I-D.ietf-pquip-pqc-engineers] Banerjee, A., Reddy.K, T., Schoinianakis, D., Hollebeek, T., and M. Ounsworth, "Post-Quantum Cryptography for Engineers", Work in Progress, Internet-Draft, draft-ietf-pquip-pqc-engineers-14, 25 August 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-pquip-pqc-engineers-14>.

[I-D.ietf-privacypass-key-consistency] Davidson, A., Finkel, M., Thomson, M., and C. A. Wood, "Key Consistency and Discovery", Work in Progress, Internet-Draft, draft-ietf-privacypass-key-consistency-01, 10 July 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-key-consistency-01>.

[I-D.irtf-cfrg-bls-signature]
Boneh, D., Bradley, J., Gorbunov, S., Wahby, R. S., Wee, H., Wood, C. A., and Z. Zhang, "BLS Signatures", Work in Progress, Internet-Draft, draft-irtf-cfrg-bls-signature-06, 2 November 2025, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-06>.

[I-D.irtf-cfrg-pairing-friendly-curves]
Sakemi, Y., Kanno, S., and R. S. Wahby, "Pairing-Friendly Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-pairing-friendly-curves-12, 2 November 2025, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-pairing-friendly-curves-12>.

[ISO8601] ISO, "Date and time - Representations for information interchange - Part 1: Basic rules", <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC9449] Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0 Demonstrating Proof of Possession (DPoP)", RFC 9449, DOI 10.17487/RFC9449, September 2023, <https://www.rfc-editor.org/info/rfc9449>.

[TZ23] Tessaro, S. and C. Zhu, "Revisiting BBS Signatures", In EUROCRYPT, 2023, <https://ia.cr/2023/275>.

[UPROVE] Microsoft Research, "U-Prove Cryptographic Specification V1.1 Revision 5", <https://github.com/microsoft/uprove-node-reference/blob/main/doc/U-Prove%20Cryptographic%20Specification%20V1.1%20Revision%205.pdf>.

[VB22] Giuseppe, V. and A. Biryukov, "Dynamic universal accumulator with batch update over bilinear groups", 2022, <https://link.springer.com/chapter/10.1007/978-3-030-95312-6_17>.

[ZCASH-REVIEW]
NCC Group, "Zcash Overwinter Consensus and Sapling Cryptography Review", <https://research.nccgroup.com/wp-content/uploads/2020/07/NCC_Group_Zcash2018_Public_Report_2019-01-30_v1.3.pdf>.

# Appendix A. BLS12-381 hash_to_curve Definition Using SHAKE-256

The following defines a hash_to_curve suite [RFC9380] for the BLS12-381 curve for both the G1 and G2 subgroups using the extendable output function (xof) of SHAKE-256 as per the guidance defined in section 8.9 of [RFC9380].

Note the notation used in the below definitions is sourced from [RFC9380].

## A.1. BLS12-381 G1

The suite of BLS12381G1_XOF:SHAKE-256_SSWU_RO_ is defined as follows:

* encoding type: hash_to_curve (Section 3 of [@!RFC9380])

* E: y^2 = x^3 + 4

* p: 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f624
  1eabfffeb153ffffb9feffffffffaaab

* r: 0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001

* m: 1

* k: 128

* expand_message: expand_message_xof (Section 5.3.2 of [@!RFC9380])

* hash: SHAKE-256

* L: 64

* f: Simplified SWU for AB == 0 (Section 6.6.3 of [@!RFC9380])

* Z: 11

* E': y'^2 = x'^3 + A' * x' + B', where

  - A' = 0x144698a3b8e9433d693a02c96d4982b0ea985383ee66a8d8e8981aef
    d881ac98936f8da0e0f97f5cf428082d584c1d

  - B' = 0x12e2908d11688030018b12e8753eee3b2016c1f0f24f4070a0b9c14f
    cef35ef55a23215a316ceaa5d1cc48e98e172be0

* iso_map: the 11-isogeny map from E' to E given in Appendix E.2 of [@!RFC9380]

* h_eff: 0xd201000000010001

  Note that the h_eff values for this suite are copied from that defined for the BLS12381G1_XMD:SHA-256_SSWU_RO_ suite defined in section 8.8.1 of [RFC9380].

  An optimized example implementation of the Simplified SWU mapping to the curve E' isogenous to BLS12-381 G1 is given in Appendix F.2 [RFC9380].

# Appendix B. The BLS12-381 Curve

This section defines BLS12-381. The definitions of this section have been originally described in [I-D.irtf-cfrg-pairing-friendly-curves], where they are discussed in greater detail.

BLS12-381 are Barreto-Lynn-Scott curves, defined by two elliptic curves E1 and E2, parameterized by an integer t. In the case of BLS12-381, t is defined as,

t = -2^63 - 2^62 - 2^60 - 2^57 - 2^48 - 2^16

The curves E1 and E2 are defined over the finite fields GF(p) and GF(p^2) correspondingly, where p is defined as,

p = (t - 1)^2 * (t^4 - t^2 + 1) / 3 + t

Let (1, I) be the bases of the finite field GF(p^2), where I ^ 2 + 1 = 0 in GF(p^2). We will denote an element y of GF(p^2) as a tuple y = (y_0, y_1), where y_0 and y_1 elements of GF(p) for which it holds y = y_0 * 1 + y_1 * I. The two elliptic curves are defined by the following equations,

E1: y ^ 2 = x ^ 3 + 4
E2: y ^ 2 = x ^ 3 + 4 * (I + 1)

The group G1 and G2 are defined as the the order r subgroup of E1 defined over GF(p) and E2 defined over GF(p^2) correspondingly, where r is defined as,

r = 0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001

Note that r is a prime factor of p. The target group G_T is defined as the finite group GF(p^12) minus the element 0.

The base points of BLS12-381, encoded to octets using the procedure defined in Appendix B.2.1 and then represented in hexadecimal format, are defined as,

```
BP1 = h'97f1d3a73197d7942695638c4fa9ac0fc3688c4f9774b905a14e3a3f171bac58
       6c55e83ff97a1aeffb3af00adb22c6bb'
BP2 = h'93e02b6052719f607dacd3a088274f65596bd0d09920b61ab5da61bbdc7f5049
      334cf11213945d57e5ac7d055d042b7e024aa2b2f08f0a91260805272dc51051
      c6e47ad4fa403b02b4510b647ae3d1770bac0326a805bbefd48056c8c121bdb8'
```

### B.1. Optimal Ate pairing

This section describes the optimal Ate pairing for BLS12-381. The pairing computation uses the following utility function.

```
res = Line_function(Q1, Q2, P)
```

Inputs:

- Q1 (REQUIRED), point of G2.
- Q2 (REQUIRED), point of G2.
- P (REQUIRED), point of G1.

Outputs:

- res: an element on the target group G_T.

Procedure:

```
1. (x_1, y_1) = Q1
2. (x_2, y_2) = Q2
3. (x, y) = P
4. if Q1 = Q2, set l = (3 * x_1^2) / (2 * y_1)
5. else if Q1 = - Q2, return x - x_1
6. else set l = (y_2 - y_1) / (x_2 - x_1)
7. return (l * (x - x_1) + y_1 - y)
```

Let c = t for t as defined above (Appendix B) and c_0, c_1, ... , c_L in (-1, 0, 1) such that the sum of c_i * 2^i for i = 0, 1, ..., L equals c.

Given a point P of G1, and a point Q of G2, the output h(P, Q) where h the Ate pairing for BLS12-381 is calculated as follows,

```
1.  set f = 1 and T = Q
2.  if c_L = -1, set T = -T
3.  for i in (L-1, L-2, ..., 1, 0)
4.      f = f^2 * Line_function(T, T, P)
5.      T = T + T
6.      if c_i = 1,
7.          f = f * Line_function(T, Q, P)
8.          T = T + Q
9.      else if c_i = -1,
10.         f = f * Line_function(T, -Q, P)
11.         T = T - Q
12. f = f ^ ((p ^ 12 - 1) / r)
13. return f
```

## B.2. Point Encoding

This section defines point encoding and decoding procedures for BLS12-381. Although more flexible point encoding procedures may exist (for example [I-D.ietf-lwig-curve-representations]), the vast majority of current libraries implementing BLS12-381 use (most of them explicitly) the encoding method defined in Appendix C of [I-D.irtf-cfrg-pairing-friendly-curves]. For this reason, the ciphersuites defined in Section 7.2, use those encoding and decoding procedures. For completeness, those operations are defined in this section as well. See [I-D.irtf-cfrg-pairing-friendly-curves] for a more detailed explanation of the encoding and decoding steps. Note also that we will only consider compressed point encoding (in contrast to [I-D.irtf-cfrg-pairing-friendly-curves], which supports both compressed and uncompressed point encoding).

In this section we will use the following notation,

* For an octet string x, x[0] will denote the first octet (i.e., 8 most significant bits) of x.
* On input an element y of GF(p) or GF(p^2), sqrt(y) will return the square root of that element in the respective group, i.e., an element a such that a^2 = y, or INVALID.
* For clarity, we will use Identity_E1, Identity_E2 to denote the identity points of E1 and E2 correspondingly (note that Identity_E1 is the same point as Identity_G1 and Identity_E2 is the same point as Identity_G2).

We first have to define the following utility operations.

The following procedure returns one bit corresponding to the sign of an element of GF(p).

res = sign_GF_p(y)

Inputs:

- y (REQUIRED), point of the GF(p) group

Outputs:

- res, either 0 or 1

Procedure:

1. if y > (p - 1) / 2, return 1
2. return 0

The following procedure returns one bit corresponding to the sign of an element in GF(p^2).

res = sign_GF_p^2(y)

Inputs:

- y (REQUIRED), point of the GF(p^2) group

Outputs:

- res, either 0 or 1

Procedure:

1. (y_0, y_1) = y
2. if y_1 is 0, return sign_GF_p(y_0)
3. if y_1 > (p - 1) / 2, return 1
4. return 0

#### B.2.1. Point Serialization

Let P = (x, y) the point to be serialized.

Compute three metadata bits C_bit, I_bit, and S_bit, as follows,

1. C_bit is set to 1 (indicating that point compression is used).
2. I_bit is 1 if P is either the Identity_E1 or Identity_E2 points, otherwise it is 0.
3. S_bit is 0 if I_bit is 1 (again note that the ciphersuites described in this document always use point compression). Otherwise (i.e., when point compression is used and P is not the identity point of its respective curve), if P is a point on E1, set S_bit = sign_GF_p(y), else if P is a point on E2, S_bit = sign_GF_p^2(y).

Let m = (C_bit * 2^7) + (I_bit * 2^6) + (S_bit * 2^5) and set m_byte = I2OSP(m, 1). Define x_string as follows,

1. If P = Identity_E1, set x_string = I2OSP(0, 48).
2. If P is a point on E1 and P != Identity_E1, set x_string = I2OSP(x, 48).
3. If P = Identity_E2, set x_string = I2OSP(0, 96).
4. If P is a point on E2 and P != Identity_E2, then let x_0 and x_1 elements of GF(p) such that x = (x_0, x_1) and set x_string = I2OSP(x_1, 48) || I2OSP(x_0, 48).

Let s_string = x_string. Set s_string[0] = x_string[0] OR m_byte, where OR is computed for each bit. Output s_string as the serialization result of the point P.

#### B.2.2. Point De-serialization

Let m_byte = s_string[0] AND 0xE0, where AND is computed bitwise. If m_byte equals 0x20 or 0x60 or 0xE0, output INVALID and abort the operation. Otherwise, let C_bit equal the most significant bit of m_byte, I_bit equal the second most significant bit of m_byte, and S_bit equal the third most significant bit of m_byte. If C_bit is 0 return INVALID and abort the operation (note again that we only consider compressed encoding).

1. Determine the curve of the encoded point as follows,
   * If s_string has length 48 octets, the encoded point is on the curve E1.
   * If s_string has length 96 octets, the encoded point is on the curve E2.
   * If s_string has any other length, output INVALID and abort the operation.
2. Let s_string[0] = s_string[0] AND 0x1F, where AND is computed bitwise (this will set the three most significant bits of s_string[0] to 0).
3. If I_bit is 1, then the encoded point must be the Identity point of the curve determined on step 1. If s_string is not the all zeros string, output INVALID and abort the operation. Otherwise, output the Identity point of the curve that was determined in step 1 (i.e., either Identity_E1 or Identity_E2).
4. Let x = OS2IP(s_string).
5. If the curve that was determined in step 1 is E1,
   * Let y2 = x^3 + 4 in GF(p).
   * If y2 is not square in GF(p), output INVALID and abort the operation. Otherwise, let y = sqrt(y2) in GF(p) and set Y_bit = sign_GF_p(y).
6. If the curve that was determined in step 1 is E2,
   * Let y2 = x^3 + 4 * (I + 1) in GF(p^2).
   * If y2 is not square in GF(p^2), output INVALID and abort the operation. Otherwise, let y = sqrt(y2) in GF(p^2) and set Y_bit = sign_GF_p^2(y).

7. If S_bit equals Y_bit, output P = (x, y). Otherwise, output P = (x, -y).

## Appendix C. Use Cases

### C.1. Non-correlating Security Token

In the most general sense BBS signatures can be used in any application where a cryptographically secured token is required but correlation caused by usage of the token is un-desirable.

For example in protocols like OAuth2.0 the most commonly used form of the access token leverages the JWT format alongside conventional cryptographic primitives such as traditional digital signatures or HMACs. These access tokens are then used by a relying party to prove authority to a resource server during a request. However, because the access token is most commonly sent by value as it was issued by the authorization server (e.g., in a bearer style scheme), the access token can act as a source of strong correlation for the relying party. Relevant prior art can be found here (https://www.ietf.org/archive/id/draft-private-access-tokens-01.html).

BBS Signatures due to their unique properties removes this source of correlation but maintains the same set of guarantees required by a resource server to validate an access token back to its relevant authority (note that an approach to signing JSON tokens with BBS that may be of relevance is the JSON Web Proofs (JWP) format and serialization described in [I-D.ietf-jose-json-web-proof]). In the context of a protocol like OAuth2.0 the access token issued by the authorization server would feature a BBS Signature, however instead of the relying party providing this access token as issued, in their request to a resource server, they generate a unique proof from the original access token and include that in the request instead, thus removing this vector of correlation.

### C.2. Improved Bearer Security Token

Bearer based security tokens such as JWT based access tokens used in the OAuth2.0 protocol are a highly popular format for expressing authorization grants. However their usage has several security limitations. Notably a bearer based authorization scheme often has to rely on a secure transport between the authorized party (client) and the resource server to mitigate the potential for a MITM attack or a malicious interception of the access token. The scheme also has to assume a degree of trust in the resource server it is presenting an access token to, particularly when the access token grants more than just access to the target resource server, because in a bearer

based authorization scheme, anyone who possesses the access token has authority to what it grants. Bearer based access tokens also suffer from the threat of replay attacks.

Improved schemes around authorization protocols often involve adding a layer of proof of cryptographic key possession to the presentation of an access token, which mitigates the deficiencies highlighted above as well as providing a way to detect a replay attack. However, approaches that involve proof of cryptographic key possession such as DPoP ([RFC9449]), suffer from an increase in protocol complexity. A party requesting authorization must pre-generate appropriate key material, share the public portion of this with the authorization server alongside proving possession of the private portion of the key material. The authorization server must also be-able to accommodate receiving this information and validating it.

BBS Signatures offer an alternative model that solves the same problems that proof of cryptographic key possession schemes do for bearer based schemes, but in a way that doesn’t introduce new up-front protocol complexity. In the context of a protocol like OAuth2.0 the access token issued by the authorization server would feature a BBS Signature, however instead of the client providing this access token as issued, in their request to a resource server, they generate a unique proof from the original access token and include that in the request instead. Because the access token is not shared in a request to a resource server, attacks such as MITM are mitigated. A resource server also obtains the ability to detect a replay attack by ensuring the proof presented is unique.

### C.3. Selectively Disclosure Enabled Identity Credentials

BBS signatures when applied to the problem space of identity credentials can help to enhance user privacy. For example a digital drivers license that is cryptographically signed with a BBS signature, allows the holder or subject of the license (acting as the Prover of the BBS scheme) to disclose different claims from their drivers license to different parties. Furthermore, the unlinkable presentations property of proofs generated by the scheme remove an important possible source of correlation for the holder across multiple presentations.

# Appendix D. Additional Test Vectors

## D.1. BLS12-381-SHAKE-256 Ciphersuite

### D.1.1. Signature Test Vectors

#### D.1.1.1. No Header Valid Signature

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
      9b80'
m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_6 = h'515ae153e22aae04ad16f759e07237b4'
m_7 = h'd183ddc6e2665aa4e2f088af'
m_8 = h'ac55fb33a75909ed'
m_9 = h'96012096'
m_10 = h''

SK = 0x2eee0f60a8a3a8bec0ee942bfd46cbdae9a0738ee68f5a64e7238311cf09a079
PK = h'92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18
     fb0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179e
     b001963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5'
header = h''

B = h'8607ebc413b397c1e27ce591d1daa39f73da329018bda0f90bf996355cc28c3cdb
    a19feeb81e35be9e1503a018e4086e'
domain = 0x333d8686761cff65a3a2ef20bfa217d37bdf19105e87c210e9ce64ea1210a
         157

signature = h'88beeb970f803160d3058eacde505207c576a8c9e4e5dc7c5249cbcf2a
            046c15f8df047031eef3436e04b779d92a9cdb1fe4c6cc035ba1634f1
            740f9dd49816d3ca745ecbe39f655ea61fb700137fded'
```

#### D.1.1.2. Modified Message Signature

The following fixture should fail signature validation due to the message value being different from what was signed.

```
m_1 = h''

PK = h'92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18
       fb0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179e
       b001963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5'
header = h'11223344556677889900aabbccddeeff'

signature = h'b9a622a4b404e6ca4c85c15739d2124a1deb16df750be202e2430e169b
              c27fb71c44d98e6d40792033e1c452145ada95030832c5dc778334f2f
              1b528eced21b0b97a12025a283d78b7136bb9825d04ef'

valid: false
reason: modified message
```

#### D.1.1.3. Extra Unsigned Message Signature

The following fixture should fail signature validation due to an additional message being supplied that was not signed.

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
       5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
       9b80'

PK = h'92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18
       fb0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179e
       b001963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5'
header = h'11223344556677889900aabbccddeeff'

signature = h'b9a622a4b404e6ca4c85c15739d2124a1deb16df750be202e2430e169b
              c27fb71c44d98e6d40792033e1c452145ada95030832c5dc778334f2f
              1b528eced21b0b97a12025a283d78b7136bb9825d04ef'

valid: false
reason: extra unsigned message
```

#### D.1.1.4. Missing Message Signature

The following fixture should fail signature validation due to missing messages that were originally present during the signing (the presented signature was generated with all the messages in Section 8.2 as input).

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
      9b80'

PK = h'92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18
     fb0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179e
     b001963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5'
header = h'11223344556677889900aabbccddeeff'

signature = h'956a3427b1b8e3642e60e6a7990b67626811adeec7a0a6cb4f770cdd7c
            20cf08faabb913ac94d18e1e92832e924cb6e202912b624261fc6c59b
            0fea801547f67fb7d3253e1e2acbcf90ef59a6911931e'

valid: false
reason: missing messages
```

#### D.1.1.5. Reordered Message Signature

The following fixture should fail signature validation due to messages being re-ordered from the order in which they were signed.

```
m_1 = h''
m_2 = h'96012096'
m_3 = h'ac55fb33a75909ed'
m_4 = h'd183ddc6e2665aa4e2f088af'
m_5 = h'515ae153e22aae04ad16f759e07237b4'
m_6 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_7 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_8 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_9 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
      9b80'
m_10 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f0
       2'

PK = h'92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18
     fb0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179e
     b001963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5'
header = h'11223344556677889900aabbccddeeff'

signature = h'956a3427b1b8e3642e60e6a7990b67626811adeec7a0a6cb4f770cdd7c
            20cf08faabb913ac94d18e1e92832e924cb6e202912b624261fc6c59b
            0fea801547f67fb7d3253e1e2acbcf90ef59a6911931e'

valid: false
reason: re-ordered messages
```

#### D.1.1.6. Wrong Public Key Signature

The following fixture should fail signature validation due to public key used to verify is in-correct.

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
      9b80'
m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_6 = h'515ae153e22aae04ad16f759e07237b4'
m_7 = h'd183ddc6e2665aa4e2f088af'
m_8 = h'ac55fb33a75909ed'
m_9 = h'96012096'
m_10 = h''

PK = h'b24c723803f84e210f7a95f6265c5cbfa4ecc51488bf7acf24b921807801c0798
     b725b9a2dcfa29953efcdfef03328720196c78b2e613727fd6e085302a0cc2d8
     d7e1d820cf1d36b20e79eee78c13a1a5da51a298f1aef86f07bc33388f089d8'
header = h'11223344556677889900aabbccddeeff'

signature = h'956a3427b1b8e3642e60e6a7990b67626811adeec7a0a6cb4f770cdd7c
            20cf08faabb913ac94d18e1e92832e924cb6e202912b624261fc6c59b
            0fea801547f67fb7d3253e1e2acbcf90ef59a6911931e'

valid: false
reason: wrong public key
```

#### D.1.1.7. Wrong Header Signature

The following fixture should fail signature validation due to header value being modified from what was originally signed.

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
      9b80'
m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_6 = h'515ae153e22aae04ad16f759e07237b4'
m_7 = h'd183ddc6e2665aa4e2f088af'
m_8 = h'ac55fb33a75909ed'
m_9 = h'96012096'
m_10 = h''

PK = h'92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18
     fb0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179e
     b001963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5'
header = h'ffeeddccbbaa09988877665544332211'

signature = h'956a3427b1b8e3642e60e6a7990b67626811adeec7a0a6cb4f770cdd7c
            20cf08faabb913ac94d18e1e92832e924cb6e202912b624261fc6c59b
            0fea801547f67fb7d3253e1e2acbcf90ef59a6911931e'

valid: false
reason: different header
```

#### D.1.2. Proof Test Vectors

##### D.1.2.1. No Header Valid Proof

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
      9b80'
m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_6 = h'515ae153e22aae04ad16f759e07237b4'
m_7 = h'd183ddc6e2665aa4e2f088af'
m_8 = h'ac55fb33a75909ed'
m_9 = h'96012096'
m_10 = h''

public_key = h'92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd459
             49cdeb18fb0490edcd4429adff56e65cbce42cf188b31bddbd619e41
             9b99c2c41b38179eb001963bc3decaae0d9f702c7a8c004f207f46c7
             34a5eae2e8e82833f3e7ea5'
signature = h'88beeb970f803160d3058eacde505207c576a8c9e4e5dc7c5249cbcf2a
            046c15f8df047031eef3436e04b779d92a9cdb1fe4c6cc035ba1634f1
```

```
                740f9dd49816d3ca745ecbe39f655ea61fb700137fded'
header = h''
presentation_header = h'bed231d880675ed101ead304512e043ade9958dd0241ea70
                         b4b3957fba941501'
revealed_indexes = [ 0, 2, 4, 6 ]

T1 = h'a5405cc2c5965dda18714ab35f4d4a7ae4024f388fa7a5ba71202d4455b50b316
       ec37b360659e3012234562fa8989980'
T2 = h'9827a40454cdc90a70e9c927f097019dbdd84768babb10ebcb460c2d918e1ce1c
       0512bf2cc49ed7ec476dfcde7a6a10c'
domain = 0x333d8686761cff65a3a2ef20bfa217d37bdf19105e87c210e9ce64ea1210a
           157

proof = h'8ac336eea1d278656372d9914483c3d3b3069dfa4a7862293ac021dfeeebca
          93cadd7eb2b818f7b89719cdeffa5aa85989a7d691be11b1929a2bf089bfe
          9f2adc2c06788edc30585546efb74877f34ad91f0d6923b4ed7a53c49051d
          da8d056a95644ee738810772d90c1033f1dfe45c0b1b453d131170aafa8a9
          9f812f3b90a5d1d9e6bd05a4dee6a50dd277ffc646f2429372f3ad9d5946f
          feb53f24d41ffcc83c32cbb68afc9b6e0b64eebd24c69c6a7bd3bca8a6394
          ed8ae315abd555a6996f34d9da7680447947b3f35f54c38b562e990ee4d17
          a21569af4fc02f2991e6db78cc32d3ef9f6069fc5c2d47c8d8ff116dfb8a5
          9641641961b854427f67649df14ab6e63f2d0d2a0cba2b2e1e835d20cd45e
          41f274532e9d50f31a690e5fef1c1456b65c668b80d8ec17b09bd5fb3b2c4
          edd6d6f5f790a5d6da22eb9a1aa2196d1a607f3c753813ba2bc6ece15d352
          63218fc7667c5f0fabfffe74745a8000e0415c8dafd5654ce6850ac2c6485
          d02433fdaebd9993f8b86a2eebb3beb10b4cc7735330384a3f4dfd4d5b219
          98ad0227b37e736cf9c144a0386f28cccf27a01e50aab45dda8275eb87772
          8e77d2055309dba8c6604e7cff0d2c46ce6026b8e232c192955f909da6e47
          c2130c7e3f4f'
```

D.1.2.2. No Presentation Header Valid Proof

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
      9b80'
m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_6 = h'515ae153e22aae04ad16f759e07237b4'
m_7 = h'd183ddc6e2665aa4e2f088af'
m_8 = h'ac55fb33a75909ed'
m_9 = h'96012096'
m_10 = h''

public_key = h'92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd459
             49cdeb18fb0490edcd4429adff56e65cbce42cf188b31bddbd619e41
             9b99c2c41b38179eb001963bc3decaae0d9f702c7a8c004f207f46c7
             34a5eae2e8e82833f3e7ea5'
signature = h'956a3427b1b8e3642e60e6a7990b67626811adeec7a0a6cb4f770cdd7c
            20cf08faabb913ac94d18e1e92832e924cb6e202912b624261fc6c59b
            0fea801547f67fb7d3253e1e2acbcf90ef59a6911931e'
header = h'11223344556677889900aabbccddeeff'
presentation_header = h''
revealed_indexes = [ 0, 2, 4, 6 ]

T1 = h'8b497dd4dcdcf7eb58c9b43e57e06bcea3468a223ae2fc015d7a86506a952d680
      55e73f5a5847e58f133ea154256d0da'
T2 = h'8655584d3da1313f881f48c239384a5623d2d292f08dae7ac1d8129c19a02a89b
      82fa45de3f6c2c439510fce5919656f'
domain = 0x6f7ee8de30835599bb540d2cb4dd02fd0c6cf8246f14c9ee9a8463f7fd400
         f7b

proof = h'b1f8bf99a11c39f04e2a032183c1ead12956ad322dd06799c50f20fb8cf6b0
        ac279210ef5a2920a7be3ec2aa0911ace7b96811a98f3c1cceba4a2147ae7
        63b3ba036f47bc21c39179f2b395e0ab1ac49017ea5b27848547bedd27be4
        81c1dfc0b73372346feb94ab16189d4c525652b8d3361bab43463700720ec
        fb0ee75e595ea1b13330615011050a0dfcffdb21af33fda9e14ba4cc0fcad
        8015bce3fecc4704799bef9924ab19688fc04f760c4da35017072a3e29578
        8eff1b0dc2311bb199c186f86ea0540379d5a2ac8b7bd02d22487f2acc0e2
        99115e16097b970badea802752a6fcb56cfbbcc2569916a8d3fe6d2d0fb1a
        e801cfc5ce056699adf23e3cd16b1fdf197deac099ab093da049a5b4451d0
        38c71b7cc69e8390967594f6777a855c7f5d301f0f0573211ac85e2e165ea
        196f78c33f54092645a51341b777f0f5342301991f3da276c04b0224f7308
        090ae0b290d428a0570a71605a27977e7daf01d42dfbdcec252686c3060a7
        3d81f6e151e23e3df2473b322da389f15a55cb2cd8a2bf29ef0d83d487611
        7735465fae956d8df56ec9eb0e4748ad3ef5587797368c51a0ccd67eb6da3
        8602a1c2d4fd411214efc6932334ba0bcbf562626e7c0e1ae0db912c28d99
        f194fa3cd3a2'
```

#### D.1.3. Hash to Scalar Test Vectors

Using the following input message,

```
msg = h’9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
        5f02’
```

And following dst value,

```
dst = h’4242535f424c533132333831473l5f584f463a5348414b452d3235365f535357
      555f524f5f4832475f484d32535f4832535f’
```

We get the following scalar output from hash_to_scalar (Section 4.2.2), encoded with I2OSP and represented in big endian order,

```
scalar = 0x0500031f786fde5326aa9370dd7ffe9535ec7a52cf2b8f432cad5d9acfb73
          cd3
```

### D.2. BLS12-381-SHA-256 Ciphersuite

#### D.2.1. Signature Test Vectors

##### D.2.1.1. No Header Valid Signature

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
      9b80'
m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_6 = h'515ae153e22aae04ad16f759e07237b4'
m_7 = h'd183ddc6e2665aa4e2f088af'
m_8 = h'ac55fb33a75909ed'
m_9 = h'96012096'
m_10 = h''

SK = 0x60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc
PK = h'a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f285
     1bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1
     e7c4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c'
header = h''

B = h'98e38eadb6a2232cf91f41861089cda14d7e3ddef0c6eaba4d11a2732f66408f39
    4d58301ffcc8fcfb3c89bb75136f61'
domain = 0x41c5fe0290d0da734ce9bba57bfe0dfc14f3f9cfef18a0d7438cf2075fd71
         cc7

signature = h'8c87e2080859a97299c148427cd2fcf390d24bea850103a97488790392
            62ecf4f42206f6ef767f298b6a96b424c1e86c26f8fba62212d0e05b9
            5261c2cc0e5fdc63a32731347e810fd12e9c58355aa0d'
```

D.2.1.2. Modified Message Signature

The following fixture should fail signature validation due to the message value being different from what was signed.

```
m_1 = h''

SK = 0x60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc
PK = h'a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f285
     1bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1
     e7c4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c'
header = h'11223344556677889900aabbccddeeff'

signature = h'84773160b824e194073a57493dac1a20b667af70cd2352d8af241c7765
            8da5253aa8458317cca0eae615690d55b1f27164657dcafee1d5c1973
            947aa70e2cfbb4c892340be5969920d0916067b4565a0'

valid: false
reason: modified message
```

#### D.2.1.3. Extra Unsigned Message Signature

The following fixture should fail signature validation due to an additional message being supplied that was not signed.

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
      9b80'

SK = 0x60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc
PK = h'a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f285
     1bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1
     e7c4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c'
header = h'11223344556677889900aabbccddeeff'

signature = h'84773160b824e194073a57493dac1a20b667af70cd2352d8af241c7765
            8da5253aa8458317cca0eae615690d55b1f27164657dcafee1d5c1973
            947aa70e2cfbb4c892340be5969920d0916067b4565a0'

valid: false
reason: extra unsigned message
```

#### D.2.1.4. Missing Message Signature

The following fixture should fail signature validation due to missing messages that were originally present during the signing (the presented signature was generated with all the messages in Section 8.2 as input).

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
      9b80'

SK = 0x60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc
PK = h'a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f285
     1bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1
     e7c4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c'
header = h'11223344556677889900aabbccddeeff'

signature = h'8339b285a4acd89dec7777c09543a43e3cc60684b0a6f8ab335da4825c
            96e1463e28f8c5f4fd0641d19cec5920d3a8ff4bedb6c9691454597bb
            d298288abed3632078557b2ace7d44caed846e1a0a1e8'

valid: false
reason: missing messages
```

#### D.2.1.5. Reordered Message Signature

The following fixture should fail signature validation due to messages being re-ordered from the order in which they were signed.

```
m_1 = h''
m_2 = h'96012096'
m_3 = h'ac55fb33a75909ed'
m_4 = h'd183ddc6e2665aa4e2f088af'
m_5 = h'515ae153e22aae04ad16f759e07237b4'
m_6 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_7 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_8 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_9 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
        9b80'
m_10 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f0
        2'

SK = 0x60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc
PK = h'a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f285
       1bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1
       e7c4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c'
header = h'11223344556677889900aabbccddeeff'

signature = h'8339b285a4acd89dec7777c09543a43e3cc60684b0a6f8ab335da4825c
              96e1463e28f8c5f4fd0641d19cec5920d3a8ff4bedb6c9691454597bb
              d298288abed3632078557b2ace7d44caed846e1a0a1e8'

valid: false
reason: re-ordered messages
```

#### D.2.1.6. Wrong Public Key Signature

The following fixture should fail signature validation due to public key used to verify is in-correct.

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
      9b80'
m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_6 = h'515ae153e22aae04ad16f759e07237b4'
m_7 = h'd183ddc6e2665aa4e2f088af'
m_8 = h'ac55fb33a75909ed'
m_9 = h'96012096'
m_10 = h''

SK = 0x60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc
PK = h'b064bd8d1ba99503cbb7f9d7ea00bce877206a85b1750e5583dd9399828a4d206
      10cb937ea928d90404c239b2835ffb104220a9c66a4c9ed3b54c0cac9ea465d0
      429556b438ceefb59650ddf67e7a8f103677561b7ef7fe3c3357ec6b94d41c6'
header = h'11223344556677889900aabbccddeeff'

signature = h'8339b285a4acd89dec7777c09543a43e3cc60684b0a6f8ab335da4825c
             96e1463e28f8c5f4fd0641d19cec5920d3a8ff4bedb6c9691454597bb
             d298288abed3632078557b2ace7d44caed846e1a0a1e8'

valid: false
reason: wrong public key
```

#### D.2.1.7. Wrong Header Signature

The following fixture should fail signature validation due to header value being modified from what was originally signed.

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
      9b80'
m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_6 = h'515ae153e22aae04ad16f759e07237b4'
m_7 = h'd183ddc6e2665aa4e2f088af'
m_8 = h'ac55fb33a75909ed'
m_9 = h'96012096'
m_10 = h''

SK = 0x60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc
PK = h'a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f285
     1bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1
     e7c4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c'
header = h'ffeeddccbbaa09988877665544332211'

signature = h'8339b285a4acd89dec7777c09543a43e3cc60684b0a6f8ab335da4825c
            96e1463e28f8c5f4fd0641d19cec5920d3a8ff4bedb6c9691454597bb
            d298288abed3632078557b2ace7d44caed846e1a0a1e8'

valid: false
reason: different header
```

### D.2.2. Proof Test Vectors

#### D.2.2.1. No Header Valid Proof

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
      9b80'
m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_6 = h'515ae153e22aae04ad16f759e07237b4'
m_7 = h'd183ddc6e2665aa4e2f088af'
m_8 = h'ac55fb33a75909ed'
m_9 = h'96012096'
m_10 = h''

public_key = h'a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8f
             a136f2851bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91a
             a8d460acee0e96f1e7c4cfb12d3ff9ab5d5dc91c277db75c845d649e
             f3c4f63aebc364cd55ded0c'
signature = h'8c87e2080859a97299c148427cd2fcf390d24bea850103a97488790392
            62ecf4f42206f6ef767f298b6a96b424c1e86c26f8fba62212d0e05b9
            5261c2cc0e5fdc63a32731347e810fd12e9c58355aa0d'
header = h''
presentation_header = h'bed231d880675ed101ead304512e043ade9958dd0241ea70
                      b4b3957fba941501'
revealed_indexes = [ 0, 2, 4, 6 ]

T = h''
domain = 0x41c5fe0290d0da734ce9bba57bfe0dfc14f3f9cfef18a0d7438cf2075fd71
         cc7
challenge = 0x4a70506add5b2eb0be9ff66e3ea8deae666f198edfbb1391c6834e6df4
            f1026d

proof = h'81925c2e525d9fbb0ba95b438b5a13fff5874c7c0515c193628d7d143ddc3b
        b487771ad73658895997a88dd5b254ed29abc019bfca62c09b8dafb37e5f0
        9b1d380e084ec3623d071ec38d6b8602af93aa0ddbada307c9309cca86be1
        6db53dc7ac310574f509c712bb1a181d64ea3c1ee075c018a2bc773e2480b
        5c033ccb9bfea5af347a88ab83746c9342ba76db3675ff70ce9006d166fd8
        13a81b448a632216521c864594f3f92965974914992f8d1845230915b1168
        0cf44b25886c5670904ac2d88255c8c31aea7b072e9c4eb7e4c3fdd38836a
        e9d2e9fa271c8d9fd42f669a9938aeeba9d8ae613bf11f489ce947616f5cb
        aee95511dfaa5c73d85e4ddd2f29340f821dc2fb40db3eae5f5bc08467eb1
        95e38d7d436b63e556ea653168282a23b53d5792a107f85b1203f82aab46f
        6940650760e5b320261ffc0ca5f15917b51e7d2ad4bcbec94de792e229db6
        63abff23af392a5e73ce115c27e8492ec24a0815091c69874dbd9dae2d2ee
        d000810c748a798a78a804a39034c6e745cee455812cc982eea7105948b2c
        b55b82278a77237fcbec4748e2d2255af0994dd09dba8ac60515a39b24632
        a2c1c840c4a70506add5b2eb0be9ff66e3ea8deae666f198edfbb1391c683
        4e6df4f1026d'
```

#### D.2.2.2. No Presentation Header Valid Proof

```
m_1 = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
      5f02'
m_2 = h'c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f
      9b80'
m_3 = h'7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73'
m_4 = h'77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c'
m_5 = h'496694774c5604ab1b2544eababcf0f53278ff50'
m_6 = h'515ae153e22aae04ad16f759e07237b4'
m_7 = h'd183ddc6e2665aa4e2f088af'
m_8 = h'ac55fb33a75909ed'
m_9 = h'96012096'
m_10 = h''

public_key = h'a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8f
             a136f2851bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91a
             a8d460acee0e96f1e7c4cfb12d3ff9ab5d5dc91c277db75c845d649e
             f3c4f63aebc364cd55ded0c'
signature = h'8339b285a4acd89dec7777c09543a43e3cc60684b0a6f8ab335da4825c
            96e1463e28f8c5f4fd0641d19cec5920d3a8ff4bedb6c9691454597bb
            d298288abed3632078557b2ace7d44caed846e1a0a1e8'
header = h'11223344556677889900aabbccddeeff'
presentation_header = h''
revealed_indexes = [ 0, 2, 4, 6 ]

T1 = h'84719c2b5bb275ee74913dbf95fb9054f690c8e4035f1259e184e9024544bc4bb
      ea9c244e7897f9db7c82b7b14b27d28'
T2 = h'8f5f191c956aefd5c960e57d2dfbab6761eb0ebc5efdba1aca1403dcc19e05296
      b16c9feb7636cb4ef2a360c5a148483'
domain = 0x6272832582a0ac96e6fe53e879422f24c51680b25fbf17bad22a35ea93ce5
         b47

proof = h'a2ed608e8e12ed21abc2bf154e462d744a367c7f1f969bdbf784a2a134c7db
        2d340394223a5397a3011b1c340ebc415199462ba6f31106d8a6da8b513b3
        7a47afe93c9b3474d0d7a354b2edc1b88818b063332df774c141f7a07c48f
        e50d452f897739228c88afc797916dca01e8f03bd9c5375c7a7c59996e514
        bb952a436afd24457658acbaba5ddac2e693ac48135672556358e78b5398f
        1a547a2a98dfe16230f244ba742dea737e4f810b4d94e03ac068ef840aaad
        f12b2ed51d3fb774c2a0a620019fd1f39c52c6f89a0e6067e3039413a9112
        9791b2af215a82ad2356b6bc305c1d7a828fe519619dd026eaaf07ea81cee
        52b21aab3e8320519bf37c2bb228a8b580f899d84327bdc5e84a66000e8ba
        c17d2fa039bb2246c8eacc623ccd9eb26e184a96a9e3a6702e1dbafe19477
        2394b05251f72bcd2d20f542b15b2406f899791f6f285c7b469e7c7b96241
        47f305c38c903273a949f6e85b9774aeeccfafa432e2cdd7c8f97d1687741
        ed30d725444428dd87d9884711d9a46baaf0c04b03a2a228b7033be084188
        0134b03b15f698756eca5f37503a0411a9586d3027a8b8b9118e95a9949b2
        719e85e4a669d9e4b7bb6d4544c8cc558c30d79f9c85a87e1a95611400b7c
        7dac5673d800'
```

### D.2.3. Hash to Scalar Test Vectors

Using the following input message,

```
msg = h'9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a4
        5f02'
```

And following dst value,

```
dst = h'4242535f424c533132333383147315f584d443a5348412d3235365f535357555f
      524f5f4832475f484d32535f4832535f'
```

We get the following scalar output from hash_to_scalar (Section 4.2.2), encoded with I2OSP and represented in big endian order,

```
scalar = 0x0f90cbee27beb214e6545becb8404640d3612da5d6758dffeccd77ed71698
         07c
```

## Appendix E. Proof Generation and Verification Algorithmic Explanation

The following section provides a high-level explanation of how the CoreProofGen and CoreProofVerify operations work, as presented in Appendix B of [TZ23] and used by this document. The CoreProofGen procedure uses a generic non-interactive zero-knowledge proof-of-knowledge (NIZK) protocol, executed between a Prover and a Verifier. A NIZK works as follows; Assume the group points J_0, J_1, ..., J_n and the exponents e_0, e_1, ..., e_n. Assume also that all the group points are publicly known, while only the exponent e_0 is known to the Verifier of the NIZK and the exponents e_1, ..., e_n are known only by the Prover of the protocol. The NIZK can be used to prove a relationship of the form,

```
J_O * e_0 = J_1 * e_1 + J_2 * e_2 + ... + J_n * e_n
```

While revealing nothing about the secret exponents (i.e., e_1, ..., e_n), other than the fact that the Prover knows them.

For BBS, let the Prover be in possession of a BBS signature (A, e) on messages msg_1, ..., msg_L and a domain value (see CoreSign defined in Section 3.6.1). Let A = B * (1/(e + SK)) where SK the Signer's secret key and,

```
[1]     B = P1 + Q_1 * domain + H_1 * msg_1 + ... + H_L * msg_L
```

Let (i1, ..., iR) be the indexes of the messages the Prover wants to disclose and (j1, ..., jU) be the indexes corresponding to undisclosed messages (i.e., (j1, ..., jU) = (1, 2, ..., L) \ (i1, ..., iR)). To prove knowledge of a signature on the disclosed messages, work as follows;

* Prove possession of a valid signature. As defined above, a signature (A, e), on messages msg_1, ..., msg_L is valid if A = B * 1/(e + SK), where B as in [1]. However, the Prover cannot reveal neither A, e nor B to the Verifier (signature is uniquely identifiable and B will reveal information about the signed messages, even the undisclosed ones). To get around this, the Prover needs to hide the signature (A, e) and the value of B, in a way that will allow proving knowledge of such elements with the aforementioned relationship (i.e., that A = B * 1/(e + SK)), without revealing their value. The Prover will do this by randomizing them. To do that, they take uniformly random r1, r2 in [1, r-1], and calculate,

  [2] Abar = A * (r1 * r2)
  [3] D = B * r2
  [4] Bbar = D * r1 + Abar * (-e)

  The values (Abar, D, Bbar) will be part of the proof and are used to prove possession of a BBS signature, without revealing the signature itself. Note that; if Abar and Bbar are constructed using a valid BBS signature as above, then Abar * SK = Bbar which is equivalent to h(Abar, PK) = h(Bbar, BP2), where SK, PK the Signer’s secret and public key and BP2 the base generator of G2 (used to create the Signer’s PK, see Section 3.4.2). This last equation is something that the Verifier can check using the Signer’s PK.

* Prove that the disclosed messages are signed as part of that signature. The Prover will start by setting the following,

  [5] r2’ = (1 / r2) mod r

  If the Abar, D and Bbar values are constructed using a valid BBS signature as in [2], [3] and [4], then the following will hold,

[6] P1 + Q_1 * domain + H_i1 * msg_i1 + ... + H_iR * msg_iR =
D * r2’ - H_ji * msg_j1 - ... - H_jU * msg_jU

Note that the Verifier will know the elements in the left side of [6] (i.e., P1, Q_1, H_i1, ..., H_iR and the disclosed messages: msg_i1, ..., msg_iR) as well as the base points of the right side (i.e., the points D and H_j1, ..., H_jU). They will not however know the

exponents on the right side of [6] (i.e., r2′ and the undisclosed messages: msg_j1, ..., msg_jU). The same holds for equation [4] where the Verifier will know the left side of the equation (i.e., Bbar) and the base points of the right side (i.e., D and Abar) but not the exponents (i.e., r1 and -e).

To convince the Verifier that both [4] and [6] hold, the Prover can use a NIZK, to prove that they know the exponents that satisfy those equations, without disclosing them.

Note that if the value D is constructed correctly (as in [3]), then B = D * r2′. Proving knowledge of [6] corresponds to proving knowledge of r2′, which means that the Prover does actually know a value B = D * r2′. If [6] holds, then that B value that the Prover knows (i.e., D * r2′) will also have the "correct form" for B (as in [1]), including all (the disclosed and "some" undisclosed) messages.

All that remains is proving that this B value the Prover knows, is also "signed" by the Signer i.e., that the Prover also knows values A and e, such that A = B * 1/(e + SK) or, equivalently, that h(A, PK + BP2 * e) = h(B, BP2), which is what CoreVerify checks to validate a signature (see Section 3.6.2).

Note that, the Prover will use a NIZK to showcase (among other things), knowledge of values r1 and e so that [4] holds (Bbar, D and Abar will be part of the proof and hence known to the Verifier). Setting r1′ = (1 / r1) mod r (note that proving knowledge of r1 indirectly proves knowledge of r1′ as well), using [4] and the fact that h(Abar, PK) = h(Bbar, BP2) we can get that,

h(Abar * r1′ * r2′, PK + BP2 * e) = h(D * r2′, BP2) = h(B, BP2)

Note that the above is what CoreVerify checks, for A = Abar * r1′ * r2′. Since the Prover showcased knowledge of r1′ and r2′ and revealed Abar as part of the proof, the Verifier can be assured that the Prover knows the value A = Abar * r1′ * r2′. So setting A = Abar * r1′ * r2′, the values A, e, B that the Prover showed knowledge of, will form a valid BBS signature. Note that the Verifier doesn’t know A (since they don’t know r1′ and r2′), e or B (since they don’t know r2′ or the undisclosed messages). However, they know that the prover knows them and as we saw above, these values form a valid signature on (among others) the disclosed messages.

To sum up; in order to validate the proof, a Verifier checks that h(Abar, PK) = h(Bbar, BP2) and verifies the NIZK. Validating the proof will guarantee the authenticity and integrity of the disclosed messages, as well as knowledge of the undisclosed messages and of the signature.

## Appendix F. Document History

-00

* Initial version

-01

* Populated fixtures
* Added SHA-256 based ciphersuite
* Fixed typo in ProofVerify
* Clarify ASCII string usage in DST
* Added MapMessageToScalar test vectors
* Fix typo in ciphersuite name

-02

* Variety of editiorial clarifications
* Clarified integer endianness
* Revised the encode for hash operation
* Shifted to using CSPRNG instead of PRF
* Removed total number of messages from proof verify operation
* Added deterministic proof fixtures
* Shifted to multiple CSPRNG calls to calculate random elements, instead of expand_message
* Updated hash_to_scalar to a single output

-03

* Updated core operation based on new academic paper (https://eprint.iacr.org/2023/275)
* Variety of editorial updates
* Updated exception and error handling
* Added extension point for the operation with which the generators are created, allowing ciphersuites to define different operations for creating the generator points.
* Added extension point for the operation with which the input messages are mapped to scalar values, allowing ciphersuites to define different message-to-scalar mapping operations
* Added signature/proof fixtures with an empty header or an empty presentation header input
* Updated the fixtures to use variable length messages (one of which is now the empty message "")

-04

* Restructure Proof Generation and Verification operation to different subroutines.

* Separate high-level (Interface) operations from low-level (Core) operations.
* Update the ciphersuite ID to remove from it the create_generators and map_message_to_scalar IDs, since those are defined as part of the high-level interface instead of the ciphersuite.
* Add a commitment optional value to the CoreSign operation. The commitment value is added to allow using BBS as part of other protocols but is ignored in this document.
* Update test-vectors display.

-05

* Proof Generation and Verification operations updated based on Appendix B of [TZ23].
* Test vectors updated based on the new proof generation procedure.
* Removed the optional commitment value from the CoreSign operation, as the intended use case (blind signatures) will be addressed differently and in another document.
* Changed the reference to [I-D.irtf-cfrg-pairing-friendly-curves] from Normative to Informative, by re-defining the relevant functionality to this document.
* Various editorial updates.

-06

* To support bounded memory implementations, the order of the inputs to the digest operation for the calculation of the e value during CoreSign and the challenge value during CoreProofGen and CoreProofVerify was updated.
* Updated the test vectores to match the above update.
* Renamed the pairing function from e to h, to avoid naming collisions with the scalar component of the signature.
* Renamed signature_dst, challenge_dst and domain_dst to hash_to_scalar_dst.

-07

* Editorial fixes (nizk -> NIZK, clarified scalar multiplication in Notation Section).
* Removed "subject to change" warning on additional test vectors.
* Fixed proof deserialization error.
* Fixed order of inputs in CoreSign call.
* Fixed wrong inputs in calculate_domain call in CoreSign and CoreVerify.

-08

* Editorial clarifications and fixes.

-09

* Editorial clarifications and fixing mistakes in api calls.
* Using CDDL style representation of test vector values.

-10

* Editorial and formatting fixes
* Clarifying the use of the h and 0x prefixes to represent test vectors

## Authors' Addresses

Tobias Looker
MATTR
Email: tobias.looker@mattr.global

Vasilis Kalos
MATTR
Email: vasilis.kalos@mattr.global

Andrew Whitehead
Portage
Email: andrew.whitehead@portagecybertech.com

Mike Lodder
CryptID
Email: redmike7@gmail.com