Workgroup: CFRG
Internet-Draft:
draft-irtf-cfrg-bbs-signatures-05
Published: 21 December 2023
Intended Status: Informational
Expires: 23 June 2024
Authors: T. Looker (MATTR), V. Kalos (MATTR), A. Whitehead (Portage), M. Lodder (CryptID)

The BBS Signature Scheme

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 it self, while at the same time, guarantying 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

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, each 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. 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

```
  (1) sign                                      (3) ProofGen
   +-----                                         +-----
   |    |                                         |    |
   |    |                                         |    |
   |   \ /                                        |   \ /
+----------+                                   +-----------+
|          |                                   |           |
|          |                                   |           |
|          |                                   |           |
|  Signer  |---(2)* Send signature + msgs----->|  Holder/  |
|          |                                   |  Prover   |
|          |                                   |           |
|          |                                   |           |
+----------+                                   +-----------+
                                                     |
                                                     |
                                                     |
                                    (4)* Send proof + disclosed msgs
                                                     |
                                                     |
                                                    \ /
                                               +-----------+
                                               |           |
                                               |           |
                                               |           |
                                               | Verifier  |
                                               |           |
                                               |           |
                                               |           |
                                               +-----------+
                                                  |   / \
                                                  |    |
                                                  |    |
                                                  +-----
                                             (5) ProofVerify
```

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.

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]. Note, 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]. Note, 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 For sets I and J, denotes the difference of the two sets 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 x * P denotes scalar multiplication.

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.

e 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).

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.

* subgroup_check_G1(P) and subgroup_check_G2(P): operations that return VALID if the point P is in the subgroup G1 or G2 correspondingly, and INVALID otherwise, as defined in Section 1.2.

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.

Each of the core operations (see Section 3.6), expect 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

In definition of this signature scheme there are two possible variations based upon the sub-group 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 cryptography 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 the case of this scheme, due to the operations involved in both signature and proof generation being computational in-efficient when performed in G2 and in the pursuit of simplicity, the scheme 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 etc.). This document uses the procedures defined in [I-D.irtf-cfrg-hash-to-curve] 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 the BBS proofs, which was 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).

3.4. 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 inputted 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 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, header, message_scalars,
                                              generators, 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 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 will be bound the the resulting proof (see Section 3.3.6). 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 = 2 * octet_point_length + 3 * 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. 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.
* e 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.

```
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. signature_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, generators, header, api_id)

2. e = hash_to_scalar(serialize((SK, domain, msg_1, ..., msg_L)),
                                                        signature_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 12 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 9, and or A != Identity_G1 after step 9 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, generators, header, api_id)
2. B = P1 + Q_1 * domain + H_1 * msg_1 + ... + H_L * msg_L
3. if e(A, W + BP2 * e) * 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 (ph). 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_indexes,
                                      disclosed_messages, ph)
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 (ph) 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 2 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,
                                     messages, disclosed_indexes, api_id)
2. if init_res is INVALID, return INVALID
3. challenge = ProofChallengeCalculate(init_res, disclosed_indexes,
                                                    messages, ph, api_id)
4. if challenge is INVALID, return INVALID
```

```
5. if cp != challenge, return INVALID
6. if e(Abar, W) * e(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 3 more items than the list of undisclosed indexes (i.e., it must hold that length(random_scalars) = length(undisclosed_indexes) + 3).

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 3 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 (ph).

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 (REQUIRED), 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. challenge_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 = (Abar, Bbar, D, T1, T2, R, i1, ..., iR,
                                            msg_i1, ..., msg_iR, domain)
2. c_octs = serialize(c_arr) || I2OSP(length(ph), 8) || ph
3. return hash_to_scalar(c_octs, challenge_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 radom 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 to 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 values 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. domain_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, domain_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, CoreProofGen Section 3.6.3 etc.), 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, 1):
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. The following sections will describe possible threats, resulting from side chanel information or identifying disclosed messages, that could compromise the unlinkability property of the BBS proof. Such threats, if exploited, could lead to correlation of the Prover's interactions with different Verifiers, resulting to fingerprinting attacks on the Prover's activity.

Note that, the following sections describe ways to minimize possible identifying information revealed during a BBS proof presentation. 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. 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 index 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.

5.2. 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 5.1).

5.3. 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 government IDs, email addresses, phone numbers etc. 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 is 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.

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 udnisclosed 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 the undisclosed 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 need 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 chanel 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 2 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], Base-64 decoding [RFC4648] etc.,) should 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 understating for 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 etc.,) 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 compine 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 refered 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 optaining a valid BBS signature on those messages (in academic works, this is refered to as the BBS proof being a proof-of-knowledge of a BBS signature [CDL16] [TZ23]).

Data confidenciality 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 refered to as the zero-knowledge proeprty 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 Signatures. As a result, an adversary with access to such a quantum computer, will not be able to reveal neither the messages undisclosed by a BBS proof, nor the hidden signature value. 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).

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 etc.,), the serialization operations (point_to_octets_E1 etc.,) 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]).

*e: 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 leven 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 e 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, 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 ciphersuite makes use of expand_message_xof with SHAKE-256, while 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 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 = "8929dfbc7e6642c4ed9cba0856e493f8b9d7d5fcb0c31ef8fdcd34d50648a5
      6c795e106e9eada6e0bda386b414150755"
```

* e: 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 = "a8ce256102840821a3e94ea9025e4662b205762f9776b3a766c872b948f1fd
      225e7c59698588e70d11406d161b4e28c9"
```

* e: 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 format.

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 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""
```

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 = "746869732d49532d6a7573742d616e2d546573742d494b4d2d746f2d
                67656e65726174652d246528724074232d6b6579"
```

the following key_info value

```
key_info = "746869732d49532d736f6d652d6b65792d6d657461646174612d746f2d62
            652d757365642d696e2d746573742d6b65792d67656e"
```

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,

```
key_dst = "4242535f424c53313233383147315f584f463a5348414b452d3235365f535
           357555f524f5f4832475f484d32535f4b455947454e5f4453545f"
```

Outputs the following SK value

```
SK = "2eee0f60a8a3a8bec0ee942bfd46cbdae9a0738ee68f5a64e7238311cf09a079"
```

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

```
PK = "92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18f
      b0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179eb00
      1963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5"
```

8.3.2. Map Messages to Scalars

The messages in Section 3.3.3 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 3.3.3. Each output scalar value is encoded to octets using I2OSP and represented in big endian order,

```
msg_scalar_1 = "1e0dea6c9ea8543731d331a0ab5f64954c188542b33c5bbc8ae5b3a8
                30f2d99f"
msg_scalar_2 = "3918a40fb277b4c796805d1371931e08a314a8bf8200a92463c06054
                d2c56a9f"
msg_scalar_3 = "6642b981edf862adf34214d933c5d042bfa8f7ef343165c325131e2f
                fa32fa94"
msg_scalar_4 = "33c021236956a2006f547e22ff8790c9d2d40c11770c18cce6037786
                c6f23512"
msg_scalar_5 = "52b249313abbe323e7d84230550f448d99edfb6529dec8c4e783dbd6
                dd2a8471"
msg_scalar_6 = "2a50bdcbe7299e47e1046100aadffe35b4247bf3f059d525f9215374
                84dd54fc"
msg_scalar_7 = "0e92550915e275f8cfd6da5e08e334d8ef46797ee28fa29de40a1ebc
                cd9d95d3"
msg_scalar_8 = "4c28f612e6c6f82f51f95e1e4faaf597547f93f6689827a6dcda3cb9
                4971d356"
msg_scalar_9 = "1db51bedc825b85efe1dab3e3ab0274fa82bbd39732be3459525faf7
                0f197650"
msg_scalar_10 = "27878da72f7775e709bb693d81b819dc4e9fa60711f4ea927740e40
                 073489e78"
```

8.3.3. Message Generators

Following the procedure defined in Section 4.1.1 with an input count value of 11, for the BLS12-381-SHAKE-256 suite, 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 = "a9d40131066399fd41af51d883f4473b0dcd7d028d3d34ef17f3241d204e28507
       d7ecae032afa1d5490849b7678ec1f8"
H_1 = "903c7ca0b7e78a2017d0baf74103bd00ca8ff9bf429f834f071c75ffe6bfdec6d
       6dca15417e4ac08ca4ae1e78b7adc0e"
H_2 = "84321f5855bfb6b001f0dfcb47ac9b5cc68f1a4edd20f0ec850e0563b27d2acce
       e6edff1a26b357762fb24e8ddbb6fcb"
H_3 = "b3060dff0d12a32819e08da00e61810676cc9185fdd750e5ef82b1a9798c7d76d
       63de3b6225d6c9a479d6c21a7c8bf93"
H_4 = "8f1093d1e553cdead3c70ce55b6d664e5d1912cc9edfdd37bf1dad11ca396a0a8
       bb062092d391ebf8790ea5722413f68"
H_5 = "990824e00b48a68c3d9a308e8c52a57b1bc84d1cf5d3c0f8c6fb6b1230e4e5b8e
       b752fb374da0b1ef687040024868140"
H_6 = "b86d1c6ab8ce22bc53f625d1ce9796657f18060fcb1893ce8931156ef992fe568
       56199f8fa6c998e5d855a354a26b0dd"
H_7 = "b4cdd98c5c1e64cb324e0c57954f719d5c5f9e8d991fd8e159b31c8d079c76a67
       321a30311975c706578d3a0ddc313b7"
H_8 = "8311492d43ec9182a5fc44a75419b09547e311251fe38b6864dc1e706e29446cb
       3ea4d501634eb13327245fd8a574f77"
H_9 = "ac00b493f92d17837a28d1f5b07991ca5ab9f370ae40d4f9b9f2711749ca20011
       0ce6517dc28400d4ea25dddc146cacc"
H_10 = "965a6c62451d4be6cb175dec39727dc665762673ee42bf0ac13a37a74784fbd6
        1e84e0915277a6f59863b2bb4f5f6005"
```

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 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"

SK = "2eee0f60a8a3a8bec0ee942bfd46cbdae9a0738ee68f5a64e7238311cf09a079"
PK = "92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18f
      b0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179eb00
      1963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5"
header = "11223344556677889900aabbccddeeff"

B = "8bbc8c123d3f128f206dd0d2dae490e82af08b84e8d70af3dc291d32a6e98f635be
     efcc4533b2599804a164aabe68d7c"
domain = "2f18dd269c11c512256a9d1d57e61a7d2de6ebcf41cac3053f37afedc4e650
          a9"

signature = "98eb37fceb31115bf647f2983aef578ad895e55f7451b1add02fa738224
             cb89a31b148eace4d20d001be31d162c58d12574f30e68665b6403956a8
             3b23a16f1daceacce8c5fde25d3defd52d6d5ff2e1"
```

8.3.4.2. Valid Multi-Message Signature

```
m_1 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""

SK = "2eee0f60a8a3a8bec0ee942bfd46cbdae9a0738ee68f5a64e7238311cf09a079"
PK = "92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18f
      b0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179eb00
      1963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5"
header = "11223344556677889900aabbccddeeff"

B = "ae8d4ebe248b9ad9c933d5661bfb46c56721fba2a1182ddda7e8fb443bda3c0a571
     ad018ad31d0b6d1f4e8b985e6c58d"
domain = "6f7ee8de30835599bb540d2cb4dd02fd0c6cf8246f14c9ee9a8463f7fd400f
          7b"

signature = "97a296c83ed3626fe254d26021c5e9a087b580f1e8bc91bb51efb04420b
             fdaca215fe376a0bc12440bcc52224fb33c696cca9239b9f28dcddb7bd8
             50aae9cd1a9c3e9f3639953fe789dbba53b8f0dd6f"
```

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 =
      "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 = "1004262112c3eaa95941b2b0d1311c09c845db0099a50e67eda62
                   8ad26b43083"
random_scalar_2 = "6da7f145a94c1fa7f116b2482d59e4d466fe49c955ae8726e7945
                   3065156a9a4"
random_scalar_3 = "05017919b3607e78c51e8ec34329955d49c8c90e4488079c43e74
                   824e98f1306"
random_scalar_4 = "4d451dad519b6a226bba79e11b44c441f1a74800eecfec6a2e2d7
                   9ea65b9d32d"
random_scalar_5 = "5e7e4894e6dbe68023bc92ef15c410b01f3828109fc72b3b5ab15
                   9fc427b3f51"
random_scalar_6 = "646e3014f49accb375253d268eb6c7f3289a1510f1e9452b612dd
                   73a06ec5dd4"
random_scalar_7 = "363ecc4c1f9d6d9144374de8f1f7991405e3345a3ec49dd485a39
                   982753c11a4"
random_scalar_8 = "12e592fe28d91d7b92a198c29afaa9d5329a4dcfdaf8b08557807
                   412faeb4ac6"
random_scalar_9 = "513325acdcdec7ea572360587b350a8b095ca19bdd8258c5c69d3
                   75e8706141a"
random_scalar_10 = "6474fceba35e7e17365dde1a0284170180e446ae96c82943290d
                    7baa3a6ed429"
```

8.3.5.1. Valid Single Message Proof

```
m_0 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"

public_key = "92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd4594
              9cdeb18fb0490edcd4429adff56e65cbce42cf188b31bddbd619e419b9
              9c2c41b38179eb001963bc3decaae0d9f702c7a8c004f207f46c734a5e
              ae2e8e82833f3e7ea5"
signature = "98eb37fceb31115bf647f2983aef578ad895e55f7451b1add02fa738224
             cb89a31b148eace4d20d001be31d162c58d12574f30e68665b6403956a8
             3b23a16f1daceacce8c5fde25d3defd52d6d5ff2e1"
header = "11223344556677889900aabbccddeeff"
presentation_header = "bed231d880675ed101ead304512e043ade9958dd0241ea70b
                       4b3957fba941501"
revealed_indexes = "[ 0 ]"

random scalars:
    r1 = "1308e6f945f663b96de1c76461cf7d7f88b92eb99a9034685150db443d7338
          81"
    r2 = "25f81cb69a8fac6fb55d44a084557258575d1003be2bd94f1922dad2c3e447
          fd"
    e_tilde = "5e8041a7ab02976ee50226c4b062b47d38829bbf42ee7eb899b297203
               77a584c"
    r1_tilde = "3bbf1d5dc2904dbb7b2ba75c5dce8a5ad2d56a359c13ff0fa5fcb133
                9cd2fe58"
    r3_tilde = "016b1460eee7707c524a86a4aedeb826ce9597b42906dccaa96c6b49
                a8ea7da2"
    m_tilde_scalars: "[  ]"

T1 = "aa74110474fcb00285be4fef3189da207720a7fbc84e3afae2c75b12d936f365c8
      6c9ac5fa39119ef5e094d151bfef0f"
T2 = "988f3d473186634e41478dc4527cf240e64de23a763037454d39a876862ebc6177
      38ba6c458142e3746b01eab58ca8d7"
domain = "2f18dd269c11c512256a9d1d57e61a7d2de6ebcf41cac3053f37afedc4e650
          a9"

proof = "89b485c2c7a0cd258a5d265a6e80aae416c52e8d9beaf0e38313d6e5fe31e7f
         7dcf62023d130fbc1da747440e61459b1929194f5527094f56a7e812afb7d92
         ff2c081654c6d5a70e369474267f1c7f769d47160cd92d79f66bb86e994c999
         226b023d58ee44d660434e6ba60ed0da1a5d2cde031b483684cd7c5b13295a8
         2f57e209b584e8fe894bcc964117bf3521b468cc9c6ba22419b3e567c7f72b6
         af815ddeca161d6d5270c3e8f269cdabb7d60230b3c66325dcf6caf39bcca06
         d889f849d301e7f30031fdeadc443a7575de547259ffe5d21a45e5a0da9b113
         512f7b124f031b0b8329a8625715c9245033ae13dfadd6bdb0b4364952647db
         3d7b91faa4c24cbb65344c03473c5065bb414ff7"
```

8.3.5.2. Valid Multi-Message, All Messages Disclosed Proof

```
m_1 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""

public_key = "92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd4594
              9cdeb18fb0490edcd4429adff56e65cbce42cf188b31bddbd619e419b9
              9c2c41b38179eb001963bc3decaae0d9f702c7a8c004f207f46c734a5e
              ae2e8e82833f3e7ea5"
signature = "97a296c83ed3626fe254d26021c5e9a087b580f1e8bc91bb51efb04420b
             fdaca215fe376a0bc12440bcc52224fb33c696cca9239b9f28dcddb7bd8
             50aae9cd1a9c3e9f3639953fe789dbba53b8f0dd6f"
header = "11223344556677889900aabbccddeeff"
presentation_header = "bed231d880675ed101ead304512e043ade9958dd0241ea70b
                       4b3957fba941501"
revealed_indexes = "[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]"

random scalars:
    r1 = "1308e6f945f663b96de1c76461cf7d7f88b92eb99a9034685150db443d7338
          81"
    r2 = "25f81cb69a8fac6fb55d44a084557258575d1003be2bd94f1922dad2c3e447
          fd"
    e_tilde = "5e8041a7ab02976ee50226c4b062b47d38829bbf42ee7eb899b297203
               77a584c"
    r1_tilde = "3bbf1d5dc2904dbb7b2ba75c5dce8a5ad2d56a359c13ff0fa5fcb133
                9cd2fe58"
    r3_tilde = "016b1460eee7707c524a86a4aedeb826ce9597b42906dccaa96c6b49
                a8ea7da2"
    m_tilde_scalars: "[  ]"

T1 = "8aae12173b9fc9032a603c9e61b0c3dfa9b8d0c4428d7acba4317aa90354ed3fff
      1afb720cd0e15a912eb2d7ece8037f"
T2 = "a49f953636d3651a3ae6fe45a99a2e4fec079eef3be8b8a6a4ba70885d7e028642
      f7224e9f451529915c88a7edc59fbe"
domain = "6f7ee8de30835599bb540d2cb4dd02fd0c6cf8246f14c9ee9a8463f7fd400f
          7b"

proof = "80ff9367fda28896618e8ede02481d660fe80bfce51a46bebe7e1d6a4c751d6
         0e09e87cd8d1e2a078d0838de56b6a7ca94651eec82e5f689b4dfc7e3c879ff
         7e33906271b17af20eab678d64903515971e39484e712fd3c8a45f279c1e058
         955b3dd7ed57aaadc348361e2501a17317352e555a333e014e8e7d71eef808a
         e4f8fbdf45cd19fde45038bb310d5135f5205611672c8d50d505af8a6e03872
         9230458a6ceb663fa048f4ce3a7a92998de4200882156ba6b6e60d855c0645d
```

```
2fdd628518d2e6fc5221b7456ccbc1c5210a1704e4d662dddd1f99a767344a7
944ab7f9b6f9d9069de4a132e4feebb6d70a87b0856635e1b8b8ca49e2992f8
c80221398e08935824f959a821b4120cdfb5e6be"
```

8.3.5.3. Valid Multi-Message, Some Messages Disclosed Proof

```
m_1 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""

public_key = "92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd4594
              9cdeb18fb0490edcd4429adff56e65cbce42cf188b31bddbd619e419b9
              9c2c41b38179eb001963bc3decaae0d9f702c7a8c004f207f46c734a5e
              ae2e8e82833f3e7ea5"
signature = "97a296c83ed3626fe254d26021c5e9a087b580f1e8bc91bb51efb04420b
             fdaca215fe376a0bc12440bcc52224fb33c696cca9239b9f28dcddb7bd8
             50aae9cd1a9c3e9f3639953fe789dbba53b8f0dd6f"
header = "11223344556677889900aabbccddeeff"
presentation_header = "bed231d880675ed101ead304512e043ade9958dd0241ea70b
                       4b3957fba941501"
revealed_indexes = "[ 0, 2, 4, 6 ]"

random scalars:
    r1 = "5ee9426ae206e3a127eb53c79044bc9ed1b71354f8354b01bf410a02220be7
          d0"
    r2 = "280d4fcc38376193ffc777b68459ed7ba897e2857f938581acf95ae5a68988
          f3"
    e_tilde = "39966b00042fc43906297d692ebb41de08e36aada8d9504d4e0ae02ad
               59e9230"
    r1_tilde = "61f5c273999b0b50be8f84d2380eb9220fc5a88afe144efc4007545f
                0ab9c089"
    r3_tilde = "63af117e0c8b7d2f1f3e375fcf5d9430e136ff0f7e879423e49dadc4
                01a50089"
    m_tilde_scalars:
        m~_1 = "020b83ca2ab319cba0744d6d58da75ac3dfb6ba682bfce2587c5a6d8
                6a4e4e7b"
        m~_3 = "5bf565343611c08f83e4420e8b1577ace8cc4df5d5303aeb3c4e425f
                1080f836"
        m~_5 = "049d77949af1192534da28975f76d4f211315dce1e36f93ffcf2a555
                de516b28"
        m~_7 = "407e5a952f145de7da53533de8366bbd2e0c854721a204f03906dc82
                fde10f48"
        m~_8 = "1c925d9052849edddcf04d5f1f0d4ff183a66b66eb820f59b675aee1
                21cfc63c"
        m~_9 = "07d7c41b02158a9c5eac212ed6d7c2cddeb8e38baea6e93e1a00b2e8
                3e2a0995"

T1 = "8bec86c26337655162b39f97e38ee5c0bbd2b6e8900d1d68fc4c27679dbe88dc76
```

```
      f313526bc800dd3209bef6b8907e95"
T2 = "8655584d3da1313f881f48c239384a5623d2d292f08dae7ac1d8129c19a02a89b8
      2fa45de3f6c2c439510fce5919656f"
domain = "6f7ee8de30835599bb540d2cb4dd02fd0c6cf8246f14c9ee9a8463f7fd400f
          7b"

proof = "853f4927bd7e4998af27df65566c0a071a33a5207d1af33ef7c3be04004ac5d
         a860f34d35c415498af32729720ca4d92977bbbbd60fdc70ddbb2588878675b
         90815273c9eaf0caa1123fe5d0c4833fefc459d18e1dc83d669268ec702c0e1
         6a6b73372346feb94ab16189d4c525652b8d3361bab43463700720ecfb0ee75
         e595ea1b13330615011050a0dfcffdb21af36ac442df87545e0e8303260a97a
         0d251de15fc1447b82fff6b47ffb0ff94022869b315dc48c9302523b2715dde
         c9f56975a0892f5f3aeed3203c29c7a03cfc79187eef45f72b7c5bf0d4fc852
         adcc7528c05b0ba9554f2eb9b39c168a4dd6bdc3ac603ce14856184f6d71313
         9f9d3930efcc9842e724517dbccff6912088b399447ff786e2f9db8b1061cc8
         9a1636ba9282344729bcd19228ccde2318286c5a115baaf317b48341ac7906c
         6cc957f94b060351563907dca7f598a4cbdaeab26c4a4fcb6aa7ff6fd999c5f
         9bc0c9a9b0e4f4a3301de901a6c68b174ed24ccf5cd0cac6726766c91aded69
         47c4b446a9dfc8ec0aa11ec9ddda57dcc22c554a83a25471be93ae69ad9234b
         1fc3d133550d7ff570a4bc6555cd0bf23ee1b2a994b2434ea222bc221ba1615
         adc53b47ba99fc5a66495585d4c86f1f0aecb18df802b8"
```

8.4. BLS12381-SHA-256 Test Vectors

Test vectors of the BLS12-381-SHA-256 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 = "746869732d49532d6a7573742d616e2d546573742d494b4d2d746f2d
                67656e65726174652d246528724074232d6b6579"
```

the following key_info value

```
key_info = "746869732d49532d736f6d652d6b65792d6d657461646174612d746f2d62
            652d757365642d696e2d746573742d6b65792d67656e"
```

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,

```
key_dst = "4242535f424c53313233383147315f584d443a5348412d3235365f5353575
           55f524f5f4832475f484d32535f4b455947454e5f4453545f"
```

Outputs the following SK value

```
SK = "60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc"
```

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

```
PK = "a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f2851
      bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1e7c
      4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c"
```

8.4.2. Map Messages to Scalars

The messages in Section 3.3.3 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). Each output scalar value is encoded to octets using I2OSP and represented in big endian order,

```
dst = "4242535f424c53313233383147315f584d443a5348412d3235365f535357555f5
       24f5f4832475f484d32535f4d41505f4d53475f544f5f5343414c41525f41535f
       484153485f"
```

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

```
msg_scalar_1 = "1cb5bb86114b34dc438a911617655a1db595abafac92f47c5001799c
                f624b430"
msg_scalar_2 = "154249d503c093ac2df516d4bb88b510d54fd97e8d7121aede420a25
                d9521952"
msg_scalar_3 = "0c7c4c85cdab32e6fdb0de267b16fa3212733d4e3a3f0d0f75165757
                8b26fe22"
msg_scalar_4 = "4a196deafee5c23f630156ae13be3e46e53b7e39094d22877b8cba7f
                14640888"
msg_scalar_5 = "34c5ea4f2ba49117015a02c711bb173c11b06b3f1571b88a2952b93d
                0ed4cf7e"
msg_scalar_6 = "4045b39b83055cd57a4d0203e1660800fabe434004dbdc8730c21ce3
                f0048b08"
msg_scalar_7 = "064621da4377b6b1d05ecc37cf3b9dfc94b9498d7013dc5c4a82bf3b
                b1750743"
msg_scalar_8 = "34ac9196ace0a37e147e32319ea9b3d8cc7d21870d3c3ba071246859
                cca49b02"
msg_scalar_9 = "57eb93f417c43200e9784fa5ea5a59168d3dbc38df707a13bb597c87
                1b2a5f74"
msg_scalar_10 = "08e3afeb2b4f2b5f907924ef42856616e6f2d5f1fb373736db1cca3
                 2707a7d16"
```

8.4.3. Message Generators

Following the procedure defined in Section 4.1.1 with an input count value of 11, for the BLS12-381-SHA-256 suite, 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 = "a9ec65b70a7fbe40c874c9eb041c2cb0a7af36ccec1bea48fa2ba4c2eb67ef7f9
       ecb17ed27d38d27cdeddff44c8137be"
H_1 = "98cd5313283aaf5db1b3ba8611fe6070d19e605de4078c38df36019fbaad0bd28
       dd090fd24ed27f7f4d22d5ff5dea7d4"
H_2 = "a31fbe20c5c135bcaa8d9fc4e4ac665cc6db0226f35e737507e803044093f3769
       7a9d452490a970eea6f9ad6c3dcaa3a"
H_3 = "b479263445f4d2108965a9086f9d1fdc8cde77d14a91c856769521ad3344754cc
       5ce90d9bc4c696dffbc9ef1d6ad1b62"
H_4 = "ac0401766d2128d4791d922557c7b4d1ae9a9b508ce266575244a8d6f32110d7b
       0b7557b77604869633bb49afbe20035"
H_5 = "b95d2898370ebc542857746a316ce32fa5151c31f9b57915e308ee9d1de7db691
       27d919e984ea0747f5223821b596335"
H_6 = "8f19359ae6ee508157492c06765b7df09e2e5ad591115742f2de9c08572bb2845
       cbf03fd7e23b7f031ed9c7564e52f39"
H_7 = "abc914abe2926324b2c848e8a411a2b6df18cbe7758db8644145fefb0bf0a2d55
       8a8c9946bd35e00c69d167aadf304c1"
H_8 = "80755b3eb0dd4249cbefd20f177cee88e0761c066b71794825c9997b551f24051
       c352567ba6c01e57ac75dff763eaa17"
H_9 = "82701eb98070728e1769525e73abff1783cedc364adb20c05c897a62f2ab2927f
       86f118dcb7819a7b218d8f3fee4bd7f"
H_10 = "a1f229540474f4d6f1134761b92b788128c7ac8dc9b0c52d5949313267967303
        2ac7db3fb3d79b46b13c1c41ee495bca"
```

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 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"

SK = "60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc"
PK = "a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f2851
      bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1e7c
      4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c"
header = "11223344556677889900aabbccddeeff"

B = "92d264aed02bf23de022ebe778c4f929fddf829f504e451d011ed89a313b8167ac9
     47332e1648157ceffc6e6e41ab255"
domain = "25d57fab92a8274c68fde5c3f16d4b275e4a156f211ae34b3ab32fbaf506ed
          5c"

signature = "88c0eb3bc1d97610c3a66d8a3a73f260f95a3028bccf7fff7d9851e2acd
             9f3f32fdf58a5b34d12df8177adf37aa318a20f72be7d37a8e8d8441d1b
             c0bc75543c681bf061ce7e7f6091fe78c1cb8af103"
```

8.4.4.2. Valid Multi-Message Signature

```
m_1 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""

SK = "60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc"
PK = "a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f2851
      bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1e7c
      4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c"
header = "11223344556677889900aabbccddeeff"

B = "84f48376f7df6af40bc329cf484cdbfd0b19d0b326fccab4e9d8f00d1dbcf48139d
     498b19667f203cf8a1d1f8340c522"
domain = "6272832582a0ac96e6fe53e879422f24c51680b25fbf17bad22a35ea93ce5b
          47"

signature = "895cd9c0ccb9aca4de913218655346d718711472f2bf1f3e68916de106a
             0d93cf2f47200819b45920bbda541db2d91480665df253fedab2843055b
             dc02535d83baddbbb2803ec3808e074f71f199751e"
```

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 =
      "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 = "04f8e2518993c4383957ad14eb13a023c4ad0c67d01ec86eeb902
                   e732ed6df3f"
random_scalar_2 = "5d87c1ba64c320ad601d227a1b74188a41a100325cecf00223729
                   863966392b1"
random_scalar_3 = "0444607600ac70482e9c983b4b063214080b9e808300aa4cc02a9
                   1b3a92858fe"
random_scalar_4 = "548cd11eae4318e88cda10b4cd31ae29d41c3a0b057196ee9cf3a
                   69d471e4e94"
random_scalar_5 = "2264b06a08638b69b4627756a62f08e0dc4d8240c1b974c9c7db7
                   79a769892f4"
random_scalar_6 = "4d99352986a9f8978b93485d21525244b21b396cf61f1d71f7c48
                   e3fbc970a42"
random_scalar_7 = "5ed8be91662386243a6771fbdd2c627de31a44220e8d6f745bad5
                   d99821a4880"
random_scalar_8 = "62ff1734b939ddd87beeb37a7bbcafa0a274cbc1b07384198f0e8
                   8398272208d"
random_scalar_9 = "05c2a0af016df58e844db8944082dcaf434de1b1e2e7136ec8a99
                   b939b716223"
random_scalar_10 = "485e2adab17b76f5334c95bf36c03ccf91cef77dcfcdc6b8a69e
                    2090b3156663"
```

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 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"

public_key = "a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa
              136f2851bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d
              460acee0e96f1e7c4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f
              63aebc364cd55ded0c"
signature = "88c0eb3bc1d97610c3a66d8a3a73f260f95a3028bccf7fff7d9851e2acd
             9f3f32fdf58a5b34d12df8177adf37aa318a20f72be7d37a8e8d8441d1b
             c0bc75543c681bf061ce7e7f6091fe78c1cb8af103"
header = "11223344556677889900aabbccddeeff"
presentation_header = "bed231d880675ed101ead304512e043ade9958dd0241ea70b
                       4b3957fba941501"
revealed_indexes = "[ 0 ]"

random scalars:
    r1 = "60ca409f6b0563f687fc471c63d2819f446f39c23bb540925d9d4254ac58f3
          37"
    r2 = "2ceff4982de0c913090f75f081df5ec594c310bb48c17cfdaab5332a682ef8
          11"
    e_tilde = "6101c4404895f3dff87ab39c34cb995af07e7139e6b3847180ffdd1bc
               8c313cd"
    r1_tilde = "0dfcffd97a6ecdebef3c9c114b99d7a030c998d938905f357df62822
                dee072e8"
    r3_tilde = "639e3417007d38e5d34ba8c511e836768ddc2669fdd3faff5c14ad27
                ac2b2da1"
    m_tilde_scalars: "[  ]"

T1 = "8ce960f5155d05a1795cc3422e6c975f6436a9b70c17ffbfd776346c93a9682bb6
      c74abd70d8c32781ae783ec45ea005"
T2 = "ab9543a6b04303e997621d3d5cbd85924e7e69da498a2a9e9d3a8b01f39259c9c5
      920bd530de1d3b0afb99eb0c549d5a"
domain = "25d57fab92a8274c68fde5c3f16d4b275e4a156f211ae34b3ab32fbaf506ed
          5c"

proof = "a7c217109e29ecab846691eaad757beb8cc93356daf889856d310af5fc5587e
         a4f8b70b0d960c68b7aefa62cae806baa8edeca19ca3dd884fb977fc43d946d
         c2a0be8778ec9ff7a1dae2b49c1b5d75d775ba37652ae759b9bb70ba484c74c
         8b2aeea5597befbb651827b5eed5a66f1a959bb46cfd5ca1a817a14475960f6
         9b32c54db7587b5ee3ab665fbd37b506830a0fdc9a7f71072daabd4cdb49038
         f5c55e84623400d5f78043a18f76b272fd65667373702763570c8a2f7c83757
         4f6c6c7d9619b0834303c0f55b2314cec804b33833c7047865587b8e5561912
         3183f832021dd97439f324fa3ad90ec45417070067fb8c56b2af454562358b1
         509632f92f2116c020fe7de1ba242effdb36e980"
```

8.4.5.2. Valid Multi-Message, All Messages Disclosed Proof

```
m_1 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""

public_key = "a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa
              136f2851bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d
              460acee0e96f1e7c4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f
              63aebc364cd55ded0c"
signature = "895cd9c0ccb9aca4de913218655346d718711472f2bf1f3e68916de106a
             0d93cf2f47200819b45920bbda541db2d91480665df253fedab2843055b
             dc02535d83baddbbb2803ec3808e074f71f199751e"
header = "11223344556677889900aabbccddeeff"
presentation_header = "bed231d880675ed101ead304512e043ade9958dd0241ea70b
                       4b3957fba941501"
revealed_indexes = "[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]"

random scalars:
    r1 = "60ca409f6b0563f687fc471c63d2819f446f39c23bb540925d9d4254ac58f3
          37"
    r2 = "2ceff4982de0c913090f75f081df5ec594c310bb48c17cfdaab5332a682ef8
          11"
    e_tilde = "6101c4404895f3dff87ab39c34cb995af07e7139e6b3847180ffdd1bc
               8c313cd"
    r1_tilde = "0dfcffd97a6ecdebef3c9c114b99d7a030c998d938905f357df62822
                dee072e8"
    r3_tilde = "639e3417007d38e5d34ba8c511e836768ddc2669fdd3faff5c14ad27
                ac2b2da1"
    m_tilde_scalars: "[  ]"

T1 = "815064df090feebe9d089343add9ce0c46c55c45a7a75913c3ffe980cd51dd5af5
      a6b45a10dcf7c56927b3a30c99adea"
T2 = "b9f8cf9271d10a04ae7116ad021f4b69c435d20a5af10ddd8f5b1ec6b9b8b91605
      aca76a140241784b7f161e21dfc3e7"
domain = "6272832582a0ac96e6fe53e879422f24c51680b25fbf17bad22a35ea93ce5b
          47"

proof = "a6faacf33f935d1910f21b1bbe380adcd2de006773896a5bd2afce31a138742
         98f92e602a4d35aef5880786cffc5aaf08978484f303d0c85ce657f463b7190
         5ee7c3c0c9038671d8fb925525f623745dc825b14fc50477f3de79ce8d915d8
         41ba73c8c97264177a76c4a03341956d2ae45ed3438ce598d5cda4f1bf9507f
         ecef47855480b7b30b5e4052c92a4360110c322b4cb2d9796ff2d7419792262
         49dc14d4b1fd5ca1a8f6fdfc16f726fc7683e3605d5ec28d331111a22ed8172
```

9cbb3c8c3732c7593e445f802fc3169c26857622ed31bc058fdfe68d25f0c3b
9615279719c64048ea9cdb74104b27757c2d01035507d39667d77d990ec5bda
22c866fcc9fe70bb5b7826a2b4e861b6b8124fbd"

8.4.5.3. Valid Multi-Message, Some Messages Disclosed Proof

```
m_1 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""

public_key = "a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa
              136f2851bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d
              460acee0e96f1e7c4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f
              63aebc364cd55ded0c"
signature = "895cd9c0ccb9aca4de913218655346d718711472f2bf1f3e68916de106a
             0d93cf2f47200819b45920bbda541db2d91480665df253fedab2843055b
             dc02535d83baddbbb2803ec3808e074f71f199751e"
header = "11223344556677889900aabbccddeeff"
presentation_header = "bed231d880675ed101ead304512e043ade9958dd0241ea70b
                       4b3957fba941501"
revealed_indexes = "[ 0, 2, 4, 6 ]"

random scalars:
    r1 = "44679831fe60eca50938ef0e812e2a9284ad7971b6932a38c7303538b712e4
          57"
    r2 = "6481692f89086cce11779e847ff884db8eebb85a13e81b2d0c79d6c1062069
          d8"
    e_tilde = "721ce4c4c148a1d5826f326af6fd6ac2844f29533ba4127c3a43d222d
               51b7081"
    r1_tilde = "1ecfaf5a079b0504b00a1f0d6fe8857291dd798291d7ad7454b39811
                4393f37f"
    r3_tilde = "0a4b3d59b34707bb9999bc6e2a6d382a2d2e214bff36ecd88639a141
                24b1622e"
    m_tilde_scalars:
        m~_1 = "7217411a9e329c7a5705e8db552274646e2949d62c288d7537dd62bc
                284715e4"
        m~_3 = "67d4d43660746759f598caac106a2b5f58ccd1c3eefaec31841a4f77
                d2548870"
        m~_5 = "715d965b1c3912d20505b381470ff1a528700b673e50ba89fd287e13
                171cc137"
        m~_7 = "4d3281a149674e58c9040fc7a10dd92cb9c7f76f6f0815a1afc3b09d
                74b92fe4"
        m~_8 = "438feebaa5894ca0da49992df2c97d872bf153eab07e08ff73b28131
                c46ff415"
        m~_9 = "602b723c8bbaec1b057d70f18269ae5e6de6197a5884967b03b933fa
                80006121"

T1 = "896e010e182f0718400b1e694ebc740215c2dd703f5988b7312be5a7f824f86b22
```

```
      1dd89d7a66f61b9fb238a73169e3bb"
T2 = "8f5f191c956aefd5c960e57d2dfbab6761eb0ebc5efdba1aca1403dcc19e05296b
      16c9feb7636cb4ef2a360c5a148483"
domain = "6272832582a0ac96e6fe53e879422f24c51680b25fbf17bad22a35ea93ce5b
          47"

proof = "a8da259a5ae7a9a8e5e4e809b8e7718b4d7ab913ed5781ebbff4814c762033e
         da4539973ed9bf557f882192518318cc4916fdffc857514082915a31df5bbb7
         9992a59fd68dc3b48d19d2b0ad26be92b4cf78a30f472c0fd1e558b9d03940b
         077897739228c88afc797916dca01e8f03bd9c5375c7a7c59996e514bb952a4
         36afd24457658acbaba5ddac2e693ac481352bb6fce6084eb1867c71caeac2a
         fc4f57f4d26504656b798b3e4009eb227c7fa41b6ae00daae0436d853e86b32
         b366b0a9929e1570369e9c61b7b177eb70b7ff27326c467c362120dfeacc069
         2d25ccdd62d733ff6e8614abd16b6b63a7b78d11632cf41bc44856aee370fee
         6690a637b3b1d8d8525aff01cd3555c39d04f8ee1606964c2da8b988897e3d2
         7cb444b8394acc80876d3916c485c9f36098fed6639f12a6a6e67150a641d74
         85656408e9ae22b9cb7ec77e477f71c1fe78cab3ee5dd62c34dd595edb15cbc
         e061b29192419dfadcdee179f134dd8feb9323c426c51454168ffacb6502199
         5848e368a5c002314b508299f67d85ad0eaaaac845cb029927191152edee034
         194cca3ae0d45cbd2f5e5afd1f9b8a3dd903adfa17ae43a191bf3119df57214
         f19e662c7e01e8cc2eb6b038bc7d707f2f3e13545909e0"
```

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 preceeded 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 acknoledge 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>.

[I-D.irtf-cfrg-hash-to-curve] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-hash-to-curve-16, 15 June 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-16>.

[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] Miller, J., Waite, D., and M. B. Jones, "JSON Web Proof", Work in Progress, Internet-Draft, draft-ietf-jose-json-web-proof-02, 21 October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-jose-json-web-proof-02>.

[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., and T. Hollebeek, "Post-Quantum Cryptography for Engineers", Work in Progress, Internet-Draft, draft-ietf-pquip-pqc-engineers-02, 20 October

2023, <https://datatracker.ietf.org/doc/html/draft-ietf-pquip-pqc-engineers-02>.

[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., 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-05, 16 June 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-05>.

[I-D.irtf-cfrg-pairing-friendly-curves] Sakemi, Y., Kobayashi, T., Saito, T., and R. S. Wahby, "Pairing-Friendly Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-pairing-friendly-curves-11, 6 November 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-pairing-friendly-curves-11>.

[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 = "97f1d3a73197d7942695638c4fa9ac0fc3688c4f9774b905a14e3a3f171bac586
       c55e83ff97a1aeffb3af00adb22c6bb"
BP2 = "93e02b6052719f607dacd3a088274f65596bd0d09920b61ab5da61bbdc7f50493
       34cf11213945d57e5ac7d055d042b7e024aa2b2f08f0a91260805272dc51051c6
       e47ad4fa403b02b4510b647ae3d1770bac0326a805bbefd48056c8c121bdb8"
```

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 e(P, Q) where e 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.

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 ofter 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

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

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

D.1.1. Signature Test Vectors

D.1.1.1. No Header Valid Signature

```
m_1 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""

SK = "2eee0f60a8a3a8bec0ee942bfd46cbdae9a0738ee68f5a64e7238311cf09a079"
PK = "92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18f
      b0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179eb00
      1963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5"
header = ""

B = "8607ebc413b397c1e27ce591d1daa39f73da329018bda0f90bf996355cc28c3cdba
     19feeb81e35be9e1503a018e4086e"
domain = "333d8686761cff65a3a2ef20bfa217d37bdf19105e87c210e9ce64ea1210a1
          57"

signature = "abfa513cdb323e47214b7c182fb623197a0681b753f897545a73d82ee13
             3a8ecf69db9aa09fe425df4e7687d99d779db5c66199c0dc9d2a442d331
             c43f56e060edc69a69ed2f13de3813b98ce6b05737"
```

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 = ""

PK = "92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18f
      b0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179eb00
      1963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5"
header = "11223344556677889900aabbccddeeff"

signature = "98eb37fceb31115bf647f2983aef578ad895e55f7451b1add02fa738224
             cb89a31b148eace4d20d001be31d162c58d12574f30e68665b6403956a8
             3b23a16f1daceacce8c5fde25d3defd52d6d5ff2e1"

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 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"

PK = "92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18f
      b0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179eb00
      1963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5"
header = "11223344556677889900aabbccddeeff"

signature = "98eb37fceb31115bf647f2983aef578ad895e55f7451b1add02fa738224
             cb89a31b148eace4d20d001be31d162c58d12574f30e68665b6403956a8
             3b23a16f1daceacce8c5fde25d3defd52d6d5ff2e1"

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 3.3.3 as input).

```
m_1 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"

PK = "92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18f
      b0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179eb00
      1963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5"
header = "11223344556677889900aabbccddeeff"

signature = "97a296c83ed3626fe254d26021c5e9a087b580f1e8bc91bb51efb04420b
             fdaca215fe376a0bc12440bcc52224fb33c696cca9239b9f28dcddb7bd8
             50aae9cd1a9c3e9f3639953fe789dbba53b8f0dd6f"

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 = ""
m_2 = "96012096"
m_3 = "ac55fb33a75909ed"
m_4 = "d183ddc6e2665aa4e2f088af"
m_5 = "515ae153e22aae04ad16f759e07237b4"
m_6 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_7 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_8 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_9 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_10 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02
        "

PK = "92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18f
      b0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179eb00
      1963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5"
header = "11223344556677889900aabbccddeeff"

signature = "97a296c83ed3626fe254d26021c5e9a087b580f1e8bc91bb51efb04420b
             fdaca215fe376a0bc12440bcc52224fb33c696cca9239b9f28dcddb7bd8
             50aae9cd1a9c3e9f3639953fe789dbba53b8f0dd6f"

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 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""

PK = "b24c723803f84e210f7a95f6265c5cbfa4ecc51488bf7acf24b921807801c0798b
      725b9a2dcfa29953efcdfef03328720196c78b2e613727fd6e085302a0cc2d8d7e
      1d820cf1d36b20e79eee78c13a1a5da51a298f1aef86f07bc33388f089d8"
header = "11223344556677889900aabbccddeeff"

signature = "97a296c83ed3626fe254d26021c5e9a087b580f1e8bc91bb51efb04420b
             fdaca215fe376a0bc12440bcc52224fb33c696cca9239b9f28dcddb7bd8
             50aae9cd1a9c3e9f3639953fe789dbba53b8f0dd6f"

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 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""

PK = "92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd45949cdeb18f
      b0490edcd4429adff56e65cbce42cf188b31bddbd619e419b99c2c41b38179eb00
      1963bc3decaae0d9f702c7a8c004f207f46c734a5eae2e8e82833f3e7ea5"
header = "ffeeddccbbaa00998877665544332211"

signature = "97a296c83ed3626fe254d26021c5e9a087b580f1e8bc91bb51efb04420b
             fdaca215fe376a0bc12440bcc52224fb33c696cca9239b9f28dcddb7bd8
             50aae9cd1a9c3e9f3639953fe789dbba53b8f0dd6f"

valid: "false"
reason: "different header"
```

D.1.2. Proof Test Vectors

D.1.2.1. No Header Valid Proof

```
m_1 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""

public_key = "92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd4594
              9cdeb18fb0490edcd4429adff56e65cbce42cf188b31bddbd619e419b9
              9c2c41b38179eb001963bc3decaae0d9f702c7a8c004f207f46c734a5e
              ae2e8e82833f3e7ea5"
signature = "abfa513cdb323e47214b7c182fb623197a0681b753f897545a73d82ee13
             3a8ecf69db9aa09fe425df4e7687d99d779db5c66199c0dc9d2a442d331
             c43f56e060edc69a69ed2f13de3813b98ce6b05737"
header = ""
presentation_header = "bed231d880675ed101ead304512e043ade9958dd0241ea70b
                       4b3957fba941501"
revealed_indexes = "[ 0, 2, 4, 6 ]"

T1 = "913b100fcf5f9ac2d83635a31d806d01d4bd2d10adf2e90f377852eece1d9c0834
      db5f062d2d4d4578c54338cd923eb1"
T2 = "9827a40454cdc90a70e9c927f097019dbdd84768babb10ebcb460c2d918e1ce1c0
      512bf2cc49ed7ec476dfcde7a6a10c"
domain = "333d8686761cff65a3a2ef20bfa217d37bdf19105e87c210e9ce64ea1210a1
          57"

proof = "ada2a57ae3d869255d1533f74317b131ad4f0f24cae413ac40028d70f0cf037
         2b503ff6e705220532727002b8958ebf987e2e8378984afe3214511b9feeee8
         30ffe3121ed005d2c382c04e6db37b646bc2f7002f3699648570fe9b67a0a5a
         ac995644ee738810772d90c1033f1dfe45c0b1b453d131170aafa8a99f812f3
         b90a5d1d9e6bd05a4dee6a50dd277ffc646f6b676faadceff172a0002325e7f
         22f47ed9b5125f30dd5fffe9ed1dc99dc283100cb702fa63aaef1bd1f530a53
         68ca4c7e78a01c7fcc3563b25c6c10c0e063092cbe2590fdfcc7b6a2859e482
         796f1f6783a41dfdf133ce28d13071b77cbe7fe06bf6e138bd3323e7edc4a6e
         c9942bfa0b6d1287836e2b1c2db84833d8325d145e6d2a3e94ddd5b6f58c1d1
         b2a15a854f7cf46711239ebe522bf5e428131e31e2f5f322eba2399fa7a8efe
         c4be722dcaf6ec6adaf84af72c3d7690072d07928045327f3a6587102b066fb
         9cf96b27aca7f5698a2ec66d04efa05ed57fd6ac27636322c013a168100b733
         269e9bd6f23d7562affebafc3d9b3c5f54a0c57216b733f8ecb24dc292c17e1
         8b6b8e0f3b8303dfaedee84fba02d491994b95f965deb3c1295545bb9802d98
         449d98d1af18e9c60536146cfa7aa267bd888b25552dd2"
```

D.1.2.2. No Presentation Header Valid Proof

```
m_1 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""

public_key = "92d37d1d6cd38fea3a873953333eab23a4c0377e3e049974eb62bd4594
              9cdeb18fb0490edcd4429adff56e65cbce42cf188b31bddbd619e419b9
              9c2c41b38179eb001963bc3decaae0d9f702c7a8c004f207f46c734a5e
              ae2e8e82833f3e7ea5"
signature = "97a296c83ed3626fe254d26021c5e9a087b580f1e8bc91bb51efb04420b
             fdaca215fe376a0bc12440bcc52224fb33c696cca9239b9f28dcddb7bd8
             50aae9cd1a9c3e9f3639953fe789dbba53b8f0dd6f"
header = "11223344556677889900aabbccddeeff"
presentation_header = ""
revealed_indexes = "[ 0, 2, 4, 6 ]"

T1 = "8bec86c26337655162b39f97e38ee5c0bbd2b6e8900d1d68fc4c27679dbe88dc76
      f313526bc800dd3209bef6b8907e95"
T2 = "8655584d3da1313f881f48c239384a5623d2d292f08dae7ac1d8129c19a02a89b8
      2fa45de3f6c2c439510fce5919656f"
domain = "6f7ee8de30835599bb540d2cb4dd02fd0c6cf8246f14c9ee9a8463f7fd400f
          7b"

proof = "853f4927bd7e4998af27df65566c0a071a33a5207d1af33ef7c3be04004ac5d
         a860f34d35c415498af32729720ca4d92977bbbbd60fdc70ddbb2588878675b
         90815273c9eaf0caa1123fe5d0c4833fefc459d18e1dc83d669268ec702c0e1
         6a6b73372346feb94ab16189d4c525652b8d3361bab43463700720ecfb0ee75
         e595ea1b13330615011050a0dfcffdb21af37286b5d6012208605b7c3fe5457
         936db502aa7eec43ae4a9d1bdf5f675153d521b1e587c6ddd195e80358667aa
         e42e64754595a0d35c1d6e72f147f67f591c823e75340360615b9c0173445af
         e53002d4face239979f697eff7183826449d4dc285a15e0c6afec9289b0b39e
         0741d0c4925c090f722569b8c64e2829904a02ec1ab6340cfe999a59196bbb8
         da2be2a89ddd84378dba0a22533e76fd6ac14f2b52a3972b041950539c19deb
         af7454e6ef3b9cec23086dc26b8a104e319aa4394e4e376c133d6c00133daf2
         f414e1df8ebca2de0a23e6ba37663f8074b9c8f440e37459bc08a8a4a587b78
         b2102c81b2f48f0fa73c331f7b6f64f6d8d50f3f8cb1424626f9cf3171cdea7
         f8cedb7bbb5a269856b37e8ba16ba8604fb1681be22dc6b64827a8326691524
         b7c05ac462ec8d8eee64bc6e09df622bb974fba93a75f8"
```

D.1.3. Hash to Scalar Test Vectors

Using the following input message,

```
msg = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
```

And following dst value,

```
dst = "4242535f424c53313233383147315f584f463a5348414b452d3235365f5353575
       55f524f5f4832475f484d32535f4832535f"
```

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

```
scalar = "0500031f786fde5326aa9370dd7ffe9535ec7a52cf2b8f432cad5d9acfb73c
          d3"
```

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

D.2.1. Signature Test Vectors

D.2.1.1. No Header Valid Signature

```
m_1 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""

SK = "60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc"
PK = "a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f2851
      bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1e7c
      4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c"
header = ""

B = "98e38eadb6a2232cf91f41861089cda14d7e3ddef0c6eaba4d11a2732f66408f394
     d58301ffcc8fcfb3c89bb75136f61"
domain = "41c5fe0290d0da734ce9bba57bfe0dfc14f3f9cfef18a0d7438cf2075fd71c
          c7"

signature = "ae0b1807865598b3884e3e9b110e8faec662050dc9b4d95309d957fd30f
             6fc24161f6f8b5680f1f5d1b547be221547915ca665c7b3087a336d5e0c
             5fcfea62576afd13e563b730ef6d6d81f9944ab95b"
```

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 = ""

SK = "60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc"
PK = "a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f2851
      bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1e7c
      4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c"
header = "11223344556677889900aabbccddeeff"

signature = "88c0eb3bc1d97610c3a66d8a3a73f260f95a3028bccf7fff7d9851e2acd
             9f3f32fdf58a5b34d12df8177adf37aa318a20f72be7d37a8e8d8441d1b
             c0bc75543c681bf061ce7e7f6091fe78c1cb8af103"

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 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"

SK = "60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc"
PK = "a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f2851
      bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1e7c
      4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c"
header = "11223344556677889900aabbccddeeff"

signature = "88c0eb3bc1d97610c3a66d8a3a73f260f95a3028bccf7fff7d9851e2acd
             9f3f32fdf58a5b34d12df8177adf37aa318a20f72be7d37a8e8d8441d1b
             c0bc75543c681bf061ce7e7f6091fe78c1cb8af103"

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 3.3.3 as input).

```
m_1 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"

SK = "60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc"
PK = "a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f2851
      bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1e7c
      4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c"
header = "11223344556677889900aabbccddeeff"

signature = "895cd9c0ccb9aca4de913218655346d718711472f2bf1f3e68916de106a
             0d93cf2f47200819b45920bbda541db2d91480665df253fedab2843055b
             dc02535d83baddbbb2803ec3808e074f71f199751e"

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 = ""
m_2 = "96012096"
m_3 = "ac55fb33a75909ed"
m_4 = "d183ddc6e2665aa4e2f088af"
m_5 = "515ae153e22aae04ad16f759e07237b4"
m_6 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_7 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_8 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_9 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_10 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02
        "

SK = "60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc"
PK = "a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f2851
      bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1e7c
      4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c"
header = "11223344556677889900aabbccddeeff"

signature = "895cd9c0ccb9aca4de913218655346d718711472f2bf1f3e68916de106a
             0d93cf2f47200819b45920bbda541db2d91480665df253fedab2843055b
             dc02535d83baddbbb2803ec3808e074f71f199751e"

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 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""

SK = "60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc"
PK = "b064bd8d1ba99503cbb7f9d7ea00bce877206a85b1750e5583dd9399828a4d2061
      0cb937ea928d90404c239b2835ffb104220a9c66a4c9ed3b54c0cac9ea465d0429
      556b438ceefb59650ddf67e7a8f103677561b7ef7fe3c3357ec6b94d41c6"
header = "11223344556677889900aabbccddeeff"

signature = "895cd9c0ccb9aca4de913218655346d718711472f2bf1f3e68916de106a
             0d93cf2f47200819b45920bbda541db2d91480665df253fedab2843055b
             dc02535d83baddbbb2803ec3808e074f71f199751e"

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 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""

SK = "60e55110f76883a13d030b2f6bd11883422d5abde717569fc0731f51237169fc"
PK = "a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa136f2851
      bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d460acee0e96f1e7c
      4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f63aebc364cd55ded0c"
header = "ffeeddccbbaa00998877665544332211"

signature = "895cd9c0ccb9aca4de913218655346d718711472f2bf1f3e68916de106a
             0d93cf2f47200819b45920bbda541db2d91480665df253fedab2843055b
             dc02535d83baddbbb2803ec3808e074f71f199751e"

valid: "false"
reason: "different header"
```

D.2.2. Proof Test Vectors

D.2.2.1. No Header Valid Proof

```
m_1 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""

public_key = "a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa
              136f2851bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d
              460acee0e96f1e7c4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f
              63aebc364cd55ded0c"
signature = "ae0b1807865598b3884e3e9b110e8faec662050dc9b4d95309d957fd30f
             6fc24161f6f8b5680f1f5d1b547be221547915ca665c7b3087a336d5e0c
             5fcfea62576afd13e563b730ef6d6d81f9944ab95b"
header = ""
presentation_header = "bed231d880675ed101ead304512e043ade9958dd0241ea70b
                       4b3957fba941501"
revealed_indexes = "[ 0, 2, 4, 6 ]"

T = "undefined"
domain = "41c5fe0290d0da734ce9bba57bfe0dfc14f3f9cfef18a0d7438cf2075fd71c
          c7"
challenge = "1cc198830295ccc56e5f9527216765105eee34324c5f3834154943608a8
             ca652"

proof = "958783d7d535fe1860a71ad5a7cf42df6527246300e3f3d94d67639c7e8a7db
         cf3f082f63e3b1bcc1cdad71e1f6d5f0d821c4c6bb4b2dcdfe945491d4f4a23
         d10752431d364fcbdd199c753f0beee7ffe02abbad57384244294ef7c2031d9
         c50ac310574f509c712bb1a181d64ea3c1ee075c018a2bc773e2480b5c033cc
         b9bfea5af347a88ab83746c9342ba76db36771c74f1feec7f67b30e3805d71c
         8f893837b455d734d360c80e119b00dc63e2756b81a320d659a9a0f1ee57c41
         773f304c37c278d169faec5f6720bb9187e9333b793a57ba69f27e4b0c2ea35
         271276fc0011306d6c909cf4d4a7a50dbc9f6ef35d43e2043046dc3041ac0a9
         b893dfd2dcd147910d719e818b4189a76f791a3600acd76623573c1796262a3
         914921ec504d0f727c63e16b432f6256db62b9667016e516e97e2ef0bfa3bd1
         92306564df28e019af18c50ca86a0e1d8d6b08b0641e549accd5e34ada8903d
         55021780865edfa70f63b85f0ddaf50787f8ced8eee658f2dd61673d2cbeca2
         aa2a5b649c22501b72cc7ee2d10bc9fe3aa3a7e169dc070d90b37735488cd0c
         27517ffd634b99c1dc016a4086d24feff6f19f3c92fa11cc198830295ccc56e
         5f9527216765105eee34324c5f3834154943608a8ca652"
```

D.2.2.2. No Presentation Header Valid Proof

```
m_1 = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
m_2 = "c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80"
m_3 = "7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b73"
m_4 = "77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c"
m_5 = "496694774c5604ab1b2544eababcf0f53278ff50"
m_6 = "515ae153e22aae04ad16f759e07237b4"
m_7 = "d183ddc6e2665aa4e2f088af"
m_8 = "ac55fb33a75909ed"
m_9 = "96012096"
m_10 = ""

public_key = "a820f230f6ae38503b86c70dc50b61c58a77e45c39ab25c0652bbaa8fa
              136f2851bd4781c9dcde39fc9d1d52c9e60268061e7d7632171d91aa8d
              460acee0e96f1e7c4cfb12d3ff9ab5d5dc91c277db75c845d649ef3c4f
              63aebc364cd55ded0c"
signature = "895cd9c0ccb9aca4de913218655346d718711472f2bf1f3e68916de106a
             0d93cf2f47200819b45920bbda541db2d91480665df253fedab2843055b
             dc02535d83baddbbb2803ec3808e074f71f199751e"
header = "11223344556677889900aabbccddeeff"
presentation_header = ""
revealed_indexes = "[ 0, 2, 4, 6 ]"

T1 = "896e010e182f0718400b1e694ebc740215c2dd703f5988b7312be5a7f824f86b22
      1dd89d7a66f61b9fb238a73169e3bb"
T2 = "8f5f191c956aefd5c960e57d2dfbab6761eb0ebc5efdba1aca1403dcc19e05296b
      16c9feb7636cb4ef2a360c5a148483"
domain = "6272832582a0ac96e6fe53e879422f24c51680b25fbf17bad22a35ea93ce5b
          47"

proof = "a8da259a5ae7a9a8e5e4e809b8e7718b4d7ab913ed5781ebbff4814c762033e
         da4539973ed9bf557f882192518318cc4916fdffc857514082915a31df5bbb7
         9992a59fd68dc3b48d19d2b0ad26be92b4cf78a30f472c0fd1e558b9d03940b
         077897739228c88afc797916dca01e8f03bd9c5375c7a7c59996e514bb952a4
         36afd24457658acbaba5ddac2e693ac481356d60aa96c9b53ff5c63b3930bbc
         b3940f2132b7dcd800be4afbffd3325ecedaf033d354de52e12e924b32dd13c
         2f7cebef3614a4a519ff94d1bcceb7e22562ab4a5729a74cc3746558e254696
         51d7da37f714951c2ca03fc364a2272d13b2dee53412f97f42dfd6b57ae92fc
         7cb4859f418d6a912f5c446002cbf96ee6b8f4a849577a43ef303592c33e036
         08a9ca93066084bdfb3d3974ba322b7523d48fc9b35227e776c994b0e2da158
         7b496660836a7307a2125eae5912be3ea839bb4db16a21cc394c9a63fce9104
         0d8321b30313677f7cbc4a9119fd0849aacef25fe9336db2dcbd85a2e3fd2ca
         2efff623c13e6c48b832c9e07dbe4337320dd0264a573f25bb46876e8153db4
         7de2f0176db68cca1f55406a78c89c1a65716c00e9230098c6a9690a190b207
         20a7662ccd13b392fe08d045b99d5010f625cd74f7e90a"
```

D.2.3. Hash to Scalar Test Vectors

Using the following input message,

```
msg = "9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02"
```

And following dst value,

```
dst = "4242535f424c5331323333383147315f584d443a5348412d3235365f535357555f5
       24f5f4832475f484d32535f4832535f"
```

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

```
scalar = "0f90cbee27beb214e6545becb8404640d3612da5d6758dffeccd77ed716980
         7c"
```

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 e(Abar, PK) = e(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 e(A, PK + BP2 * e) = e(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 e(Abar, PK) = e(Bbar, BP2) we can get that,

```
e(Abar * r1' * r2', PK + BP2 * e) = e(D * r2', BP2) = e(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 e(Abar, PK) = e(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
* 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.

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