Workgroup: CFRG
Internet-Draft: draft-irtf-cfrg-bbs-signatures-latest
Published: 23 October 2022
Intended Status: Informational
Expires: 26 April 2023
Authors: T. Looker, *MATTR*; V. Kalos, *MATTR*; A. Whitehead, *Portage*; M. Lodder, *CryptID*

The BBS Signature Scheme

Abstract

BBS is a digital signature scheme categorized as a form of short group signature that supports several unique properties. Notably, the scheme supports signing multiple messages whilst producing a single output digital signature. Through this capability, the possessor of a signature is able to generate proofs that selectively disclose subsets of the originally signed set of messages, whilst preserving the verifiable authenticity and integrity of the messages. Furthermore, these proofs are said to be zero-knowledge in nature as they do not reveal the underlying signature; instead, what they reveal is a proof of knowledge of the undisclosed signature.

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.

The name BBS is derived from the authors of the original academic work of Dan Boneh, Xavier Boyen, and Hovav Shacham, where the scheme was first described.

Beyond the core properties of a digital signature scheme, BBS signatures 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 holder/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 known as 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 proof generated is indistinguishable from random even for two proofs 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 (holder/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.

Refer to the Appendix B 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

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.
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.
msg	An input message to be signed by the signature

scheme.

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.
nonce	A cryptographic nonce
presentation_header (ph)	A payload generated and bound to the context of a specific spk.
nizk	A non-interactive zero-knowledge proof from fiat-shamir heuristic.
dst	The domain separation tag.
I2OSP	As defined by Section 4 of [RFC8017]
OS2IP	As defined by Section 4 of [RFC8017].

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.
range(a, b)	For integers a and b, with a <= b, denotes the ascending ordered list of all integers between a and b inclusive (i.e., the integers "i" such that a <= i <= b).
utf8(ascii_string)	Encoding the inputted ASCII string to an octet string using UTF-8 character encoding.
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.

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.

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.

P1, P2
: points on G1 and G2 respectively. 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. Operations in GT are written in multiplicative notation, i.e., a * b is field multiplication.

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 [I-D.irtf-cfrg-hash-to-curve] 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_g1(P) -> ostr, point_to_octets_g2(P) -> ostr
: returns the canonical representation of the point P for the respective subgroup as an octet string. This operation is also known as serialization.

octets_to_point_g1(ostr) -> P, octets_to_point_g2(ostr) -> P
: returns the point P for the respective subgroup corresponding to the canonical representation ostr, or INVALID if ostr is not a valid output of the respective point_to_octets_g* function. This operation is also known as deserialization.

subgroup_check(P) -> VALID or INVALID
: returns VALID when the point P is an element of the subgroup of order r, 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].

1.3. Organization of this document

This document is organized as follows:

- Scheme Definition defines the core operations and parameters for the BBS signature scheme.
- Utility Operations defines utilities used by the BBS signature scheme.
- Security Considerations describes a set of security considerations associated to the signature scheme.

- Ciphersuites 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 ciphersuite.

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 [I-D.irtf-cfrg-hash-to-curve], using the aforementioned pairing-friendly curve. This defines the hash_to_curve and expand_message operations, used by this document.
- PRF(n): a pseudo-random function similar to [RFC4868]. Returns n pseudo randomly generated bytes.

3.2. Considerations

3.2.1. 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 things. 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.

3.2.2. Messages and generators

Throughout the operations of this signature scheme, each message that is signed is paired with a specific generator (point in G1). 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).

Aside from the message generators, the scheme uses two additional generators: Q_1 and Q_2. The first (Q_1), is used for the blinding value (s) of the signature. The second generator (Q_2), is used 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.).

3.3. Key Generation Operations

3.3.1. KeyGen

This operation generates a secret key (SK) deterministically from a secret octet string (IKM).

KeyGen uses an HKDF [RFC5869] instantiated with the hash function hash.

For security, IKM MUST be infeasible to guess, e.g. generated by a trusted source of randomness.

IKM MUST be at least 32 bytes long, but it MAY be longer.

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

KeyGen takes an optional parameter, key_info. This parameter MAY be used to derive multiple independent keys from the same IKM. By default, key_info is the empty string.

```
SK = KeyGen(IKM, key_info)

Inputs:

- IKM (REQUIRED), a secret octet string. See requirements above.
- key_info (OPTIONAL), an octet string. if this is not supplied, it
                       MUST default to an empty string.

Definitions:

- HKDF-Extract is as defined in [@!RFC5869], instantiated with hash function
hash.
- HKDF-Expand is as defined in [@!RFC5869], instantiated with hash function
hash.
- I2OSP and OS2IP are as defined in [@!RFC8017], Section 4.
- L is the integer given by ceil((3 * ceil(log2(r))) / 16).
- INITSALT is the ASCII string "BBS-SIG-KEYGEN-SALT-".

Outputs:

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

Procedure:

1. salt = INITSALT
2. SK = 0
3. while SK == 0:
4.     salt = hash(salt)
5.     PRK = HKDF-Extract(salt, IKM || I2OSP(0, 1))
6.     OKM = HKDF-Expand(PRK, key_info || I2OSP(L, 2), L)
7.     SK = OS2IP(OKM) mod r
8. return SK
```

Note 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 KeyGen procedure. For security, such an alternative MUST output a secret key that is statistically close to uniformly random in the range 0 < SK < r.

3.3.2. SkToPk

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 * P2
2. return point_to_octets_g2(W)
```

3.4. Core Operations

The operations in this section make use of a "Precomputations" set of steps. The "Precomputations" steps must be executed before the steps in the "Procedure" of each operation and include computations that can be cached and re-used multiple times (like creating the generators etc.) or procedural steps like de-structuring inputted arrays.

3.4.1. Sign

This operation computes a deterministic signature from a secret key (SK) and optionally over a header and or a vector of messages.

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

Inputs:

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

Parameters:

- ciphersuite_id, ASCII string. The unique ID of the ciphersuite.
- P1, fixed point of G1, defined by the ciphersuite.

Definitions:

- L, is the non-negative integer representing the number of messages to
     be signed e.g length(messages). If no messages are supplied as an
     input, the value of L MUST evaluate to zero (0).

Outputs:

- signature, a signature encoded as an octet string.

Precomputations:

1. msg_1, ..., msg_L = messages[1], ..., messages[L]
2. (Q_1, Q_2, H_1, ..., H_L) = create_generators(L+2)

Procedure:

1.  dom_array = (PK, L, Q_1, Q_2, H_1, ..., H_L, ciphersuite_id, header)
2.  dom_for_hash = encode_for_hash(dom_array)
3.  if dom_for_hash is INVALID, return INVALID
4.  domain = hash_to_scalar(dom_for_hash, 1)
5.  e_s_for_hash = encode_for_hash((SK, domain, msg_1, ..., msg_L))
6.  if e_s_for_hash is INVALID, return INVALID
7.  (e, s) = hash_to_scalar(e_s_for_hash, 2)
8.  B = P1 + Q_1 * s + Q_2 * domain + H_1 * msg_1 + ... + H_L * msg_L
9.  A = B * (1 / (SK + e))
10. signature_octets = signature_to_octets(A, e, s)
11. return signature_octets
```

Note When computing step 9 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.

3.4.2. Verify

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

```
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 an empty string.
- messages (OPTIONAL), a vector of scalars. If not supplied, it defaults
                       to the empty array "()".

Parameters:

- ciphersuite_id, ASCII string. The unique ID of the ciphersuite.
- P1, fixed point of G1, defined by the ciphersuite.

Definitions:

- L, is the non-negative integer representing the number of messages to
     be signed e.g length(messages). If no messages are supplied as an
     input, the value of L MUST evaluate to zero (0).

Outputs:

- result, either VALID or INVALID.

Precomputations:

1. (msg_1, ..., msg_L) = messages
2. (Q_1, Q_2, H_1, ..., H_L) = create_generators(L+2)

Procedure:

1.  signature_result = octets_to_signature(signature)
2.  if signature_result is INVALID, return INVALID
3.  (A, e, s) = signature_result
4.  W = octets_to_pubkey(PK)
5.  if W is INVALID, return INVALID
6.  dom_array = (PK, L, Q_1, Q_2, H_1, ..., H_L, ciphersuite_id, header)
7.  dom_for_hash = encode_for_hash(dom_array)
8.  if dom_for_hash is INVALID, return INVALID
9.  domain = hash_to_scalar(dom_for_hash, 1)
10. B = P1 + Q_1 * s + Q_2 * domain + H_1 * msg_1 + ... + H_L * msg_L
11. if e(A, W + P2 * e) * e(B, -P2) != Identity_GT, return INVALID
12. return VALID
```

3.4.3. ProofGen

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, see Presentation header selection for more details.

The messages supplied in this operation MUST be in the same order as when supplied to Sign. 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 1 to `length(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 an empty string.
- ph (OPTIONAL), octet string containing the presentation header. If not
                 supplied, it defaults to an empty string.
- messages (OPTIONAL), a vector of scalars. 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:

- ciphersuite_id, ASCII string. The unique ID of the ciphersuite.
- P1, fixed point of G1, defined by the ciphersuite.

Definitions:

- L, is the non-negative integer representing the number of messages,
     i.e., L = length(messages). If no messages are supplied, the
     value of L MUST evaluate to zero (0).
- R, is the non-negative integer representing the number of disclosed
     (revealed) messages, i.e., R = length(disclosed_indexes). If no
     messages are disclosed, R MUST evaluate to zero (0).
- U, is the non-negative integer representing the number of undisclosed
     messages, i.e., U = L - R.
- prf_len = ceil(ceil(log2(r))/8), where r defined by the ciphersuite.

Outputs:

- proof, octet string; or INVALID.

Precomputations:

1. (i1, ..., iR) = disclosed_indexes
2. (j1, ..., jU) = range(1, L) \ disclosed_indexes
3. (msg_1, ..., msg_L) = messages
4. (msg_i1, ..., msg_iR) = (messages[i1], ..., messages[iR])
5. (msg_j1, ..., msg_jU) = (messages[j1], ..., messages[jU])
6. (Q_1, Q_2, MsgGenerators) = create_generators(L+2)
7. (H_1, ..., H_L) = MsgGenerators
8. (H_j1, ..., H_jU) = (MsgGenerators[j1], ..., MsgGenerators[jU])

Procedure:

1.  signature_result = octets_to_signature(signature)
2.  if signature_result is INVALID, return INVALID
```

```
3.  (A, e, s) = signature_result
4.  dom_array = (PK, L, Q_1, Q_2, H_1, ..., H_L, ciphersuite_id, header)
5.  dom_for_hash = encode_for_hash(dom_array)
6.  if dom_for_hash is INVALID, return INVALID
7.  domain = hash_to_scalar(dom_for_hash, 1)
8.  (r1, r2, e~, r2~, r3~, s~) = hash_to_scalar(PRF(prf_len), 6)
9.  (m~_j1, ..., m~_jU) = hash_to_scalar(PRF(prf_len), U)
10. B = P1 + Q_1 * s + Q_2 * domain + H_1 * msg_1 + ... + H_L * msg_L
11. r3 = r1 ^ -1 mod r
12. A' = A * r1
13. Abar = A' * (-e) + B * r1
14. D = B * r1 + Q_1 * r2
15. s' = r2 * r3 + s mod r
16. C1 = A' * e~ + Q_1 * r2~
17. C2 = D * (-r3~) + Q_1 * s~ + H_j1 * m~_j1 + ... + H_jU * m~_jU
18. c_array = (A', Abar, D, C1, C2, R, i1, ..., iR,
                       msg_i1, ..., msg_iR, domain, ph)
19. c_for_hash = encode_for_hash(c_array)
20. if c_for_hash is INVALID, return INVALID
21. c = hash_to_scalar(c_for_hash, 1)
22. e^ = c * e + e~ mod r
23. r2^ = c * r2 + r2~ mod r
24. r3^ = c * r3 + r3~ mod r
25. s^ = c * s' + s~ mod r
26. for j in (j1, ..., jU): m^_j = c * msg_j + m~_j mod r
27. proof = (A', Abar, D, c, e^, r2^, r3^, s^, (m^_j1, ..., m^_jU))
28. return proof_to_octets(proof)
```

3.4.4. ProofVerify

This operation checks that a proof is valid for a header, vector of disclosed messages (along side their index corresponding to their original position when signed) and presentation header against a public key (PK).

The operation accepts the list of messages the prover indicated to be disclosed. Those messages MUST be in the same order as when supplied to Sign (as a subset of the signed messages list). The operation also requires the total number of signed messages (L). Lastly, it also accepts the indexes that the disclosed messages had in the original array of messages supplied to Sign (i.e., the `disclosed_indexes` list supplied to ProofGen). Every element in this list MUST be a non-negative integer in the range from 1 to L, in ascending order.

```
result = ProofVerify(PK, proof, L, 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.
- L (REQUIRED), non-negative integer. The number of signed messages.
- header (OPTIONAL), an optional octet string containing context and
                     application specific information. If not supplied,
                     it defaults to an empty string.
- ph (OPTIONAL), octet string containing the presentation header. If not
                 supplied, it defaults to an empty string.
- disclosed_messages (OPTIONAL), a vector of scalars. 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:

- ciphersuite_id, ASCII string. The unique ID of the ciphersuite.
- P1, fixed point of G1, defined by the ciphersuite.

Definitions:

- R, is the non-negative integer representing the number of disclosed
     (revealed) messages, i.e., R = length(disclosed_indexes). If no
     messages are disclosed, the value of R MUST evaluate to zero (0).
- U, is the non-negative integer representing the number of undisclosed
     messages, i.e., U = L - R.

Outputs:

- result, either VALID or INVALID.

Precomputations:

1. (i1, ..., iR) = disclosed_indexes
2. (j1, ..., jU) = range(1, L) \ disclosed_indexes
3. (msg_i1, ..., msg_iR) = disclosed_messages
4. (Q_1, Q_2, MsgGenerators) = create_generators(L+2)
5. (H_1, ..., H_L) = MsgGenerators
6. (H_i1, ..., H_iR) = (MsgGenerators[i1], ..., MsgGenerators[iR])
7. (H_j1, ..., H_jU) = (MsgGenerators[j1], ..., MsgGenerators[jU])

Preconditions:

1. for i in (i1, ..., iR), if i < 1 or i > L, return INVALID
2. if length(disclosed_messages) != R, return INVALID

Procedure:

1.  proof_result = octets_to_proof(proof)
2.  if proof_result is INVALID, return INVALID
3.  (A', Abar, D, c, e^, r2^, r3^, s^, (m^_j1,...,m^_jU)) = proof_result
4.  W = octets_to_pubkey(PK)
5.  if W is INVALID, return INVALID
6.  dom_array = (PK, L, Q_1, Q_2, H_1, ..., H_L, ciphersuite_id, header)
7.  dom_for_hash = encode_for_hash(dom_array)
8.  if dom_for_hash is INVALID, return INVALID
9.  domain = hash_to_scalar(dom_for_hash, 1)
10. C1 = (Abar - D) * c + A' * e^ + Q_1 * r2^
11. T = P1 + Q_2 * domain + H_i1 * msg_i1 + ... + H_iR * msg_iR
12. C2 = T * c - D * r3^ + Q_1 * s^ + H_j1 * m^_j1 + ... + H_jU * m^_jU
13. cv_array = (A', Abar, D, C1, C2, R, i1, ..., iR,
                         msg_i1, ..., msg_iR, domain, ph)
14. cv_for_hash = encode_for_hash(cv_array)
15. if cv_for_hash is INVALID, return INVALID
16. cv = hash_to_scalar(cv_for_hash, 1)
17. if c != cv, return INVALID
18. if A' == Identity_G1, return INVALID
19. if e(A', W) * e(Abar, -P2) != Identity_GT, return INVALID
20. return VALID
```

4. Utility Operations

4.1. Generator point computation

This operation defines how to create a set of generators that form a part of the public parameters used by the BBS Signature scheme to accomplish operations such as Sign, Verify, ProofGen and ProofVerify. It takes one input, the number of generator points to create, which is determined in part by the number of signed messages.

As an optimization, implementations MAY cache the result of `create_generators` for a specific `generator_seed` (determined by the ciphersuite) and `count`. The values `n` and `v` MAY also be cached in order to efficiently extend a existing list of generator points.

```
generators = create_generators(count)

Inputs:

- count (REQUIRED), unsigned integer. Number of generators to create.

Parameters:

- hash_to_curve_suite, the hash to curve suite id defined by the
                         ciphersuite.
- 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.
- expand_message, the expand_message operation defined by the suite
                    specified by the hash_to_curve_suite parameter.
- generator_seed, octet string. A seed value selected by the
                    ciphersuite.
- P1, fixed point of G1, defined by the ciphersuite.

Definitions:

- seed_dst, octet string representing the domain seperation tag:
            utf8(ciphersuite_id || "SIG_GENERATOR_SEED_"), where
            ciphersuite_id is defined by the ciphersuite.
- generator_dst, octet string representing the domain seperation tag:
                 utf8(ciphersuite_id || "SIG_GENERATOR_DST_"), where
                 ciphersuite_id is defined by the ciphersuite.
- seed_len = ceil((ceil(log2(r)) + k)/8), where r and k are defined by
                                          the ciphersuite.

Outputs:

- generators, an array of generators.

Procedure:

1.  v = expand_message(generator_seed, seed_dst, seed_len)
2.  n = 1
3.  for i in range(1, count):
4.     v = expand_message(v || I2OSP(n, 4), seed_dst, seed_len)
5.     n = n + 1
6.     generator_i = Identity_G1
7.     candidate = hash_to_curve_g1(v, generator_dst)
8.     if candidate in (generator_1, ..., generator_i):
9.        go back to step 4
10.    generator_i = candidate
11. return (generator_1, ..., generator_count)
```

4.2. MapMessageToScalar

There are multiple ways in which messages can be mapped to their respective scalar values, which is their required form to be used with the Sign, Verify, ProofGen and ProofVerify operations.

4.2.1. MapMessageToScalarAsHash

This operation takes an input message and maps it to a scalar value via a cryptographic hash function for the given curve. The operation takes also as an optional input a domain separation tag (dst). If a dst is not supplied, its value MUST default to the octet string returned from utf8(ciphersuite_id || "MAP_MSG_TO_SCALAR_AS_HASH_"), where ciphersuite_id is the ASCII string representing the unique ID of the ciphersuite.

```
result = MapMessageToScalarAsHash(msg, dst)

Inputs:

- msg (REQUIRED), octet string.
- dst (OPTIONAL), an octet string representing a domain separation tag.
                  If not supplied, it default to the octet string
                  utf8(ciphersuite_id || "MAP_MSG_TO_SCALAR_AS_HASH_")
                  where ciphersuite_id is defined by the ciphersuite.

Outputs:

- result, a scalar value.

Procedure:

1. msg_for_hash = encode_for_hash(msg)
2. if msg_for_hash is INVALID, return INVALID
3. if length(dst) > 255, return INVALID
4. return hash_to_scalar(msg_for_hash, 1, dst)
```

4.3. Hash to Scalar

This operation describes how to hash an arbitrary octet string to n scalar values in the multiplicative group of integers mod r (i.e., values in the range [1, r-1]). This procedure acts as a helper function, used internally in various places within the operations described in the spec. To map a message to a scalar that would be passed as input to the Sign, Verify, ProofGen and ProofVerify functions, one must use MapMessageToScalarAsHash instead.

This operation makes use of expand_message defined in [I-D.irtf-cfrg-hash-to-curve], in a similar way used by the hash_to_field operation of Section 5 from the same document (with the additional checks for getting a scalar that is 0). If an implementer wants to use hash_to_field instead, they MUST use the multiplicative group of integers mod r (Fr), as the target group (F). Note however, that the hash_to_curve document, makes use of hash_to_field with the target group being the multiplicative group of integers mod p (Fp). For this reason, we don't directly use hash_to_field here, rather we define a similar operation (hash_to_scalar), making direct use of the expand_message function, that will be defined by the hash-to-curve suite used (i.e., either expand_message_xmd or expand_message_xof). If someone also has a hash_to_field implementation available, with the target group been Fr, they can use this instead (adding the check for a scalar been 0).

The operation takes as input an octet string representing the message to hash (msg), the number of the scalars to return (count) as well as an optional domain separation tag (dst). If a dst is not supplied, its value MUST default to the octet string returned from utf8(ciphersuit_id || "H2S_"), where ciphersuite_id is the ASCII string representing the unique ID of the ciphersuite.

```
scalars = hash_to_scalar(msg_octets, count, dst)

Inputs:

- msg_octets (REQUIRED), octet string. The message to be hashed.
- count (REQUIRED), an integer greater or equal to 1. The number of
                    scalars to output.
- dst (OPTIONAL), an octet string representing a domain separation tag.
                  If not supplied, it defaults to the octet string given
                  by utf8(ciphersuite_id || "H2S_"), where
                  ciphersuite_id is defined by the ciphersuite.

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.

Definitions:

- expand_len = ceil((ceil(log2(r))+k)/8), where r and k are defined by
                                          the ciphersuite.

Outputs:

- scalars, an array of non-zero scalars mod r.

Procedure:

1.  len_in_bytes = count * expand_len
2.  t = 0
3.  msg_prime = msg_octets || I2OSP(t, 1) || I2OSP(count, 4)
4.  uniform_bytes = expand_message(msg_prime, dst, len_in_bytes)
5.  for i in (1, ..., count):
6.      tv = uniform_bytes[(i-1)*expand_len..i*expand_len-1]
7.      scalar_i = OS2IP(tv) mod r
8.  if 0 in (scalar_1, ..., scalar_count):
9.      t = t + 1
10.     go back to step 3
11. return (scalar_1, ..., scalar_count)
```

4.4. Serialization

4.4.1. OctetsToSignature

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), octet string of the form output from
                               signature_to_octets operation.

Outputs:

signature, a signature in the form (A, e, s), where A is a point in G1
           and e and s are non-zero scalars mod r.

Procedure:

1.  expected_len = octet_point_length + 2 * 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_g1(A_octets)
5.  if A is INVALID, return INVALID
6.  if A == Identity_G1, return INVALID
7.  index = octet_point_length
8.  end_index = index + octet_scalar_length - 1
9.  e = OS2IP(signature_octets[index..end_index])
10. if e = 0 OR e >= r, return INVALID
11. index += octet_scalar_length
12. end_index = index + octet_scalar_length - 1
13. s = OS2IP(signature_octets[index..end_index])
14. if s = 0 OR s >= r, return INVALID
15. return (A, e, s)
```

4.4.2. SignatureToOctets

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`, `e` and `s` because its assumed these are done prior to its invocation, e.g as is the case with the Sign operation.

```
signature_octets = signature_to_octets(signature)

Inputs:

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

Outputs:

- signature_octets, octet string.

Procedure:

1. (A, e, s) = signature
2. A_octets = point_to_octets_g1(A)
3. e_octets = I2OSP(e, octet_scalar_length)
4. s_octets = I2OSP(s, octet_scalar_length)
5. return (A_octets || e_octets || s_octets)
```

4.4.3. OctetsToProof

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. Five (5) 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., "()").

```
proof = octets_to_proof(proof_octets)

Inputs:

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

Parameters:

- r (REQUIRED), non-negative integer. The prime order of the G1 and
                G2 groups, defined by the ciphersuite.
- octet_scalar_length (REQUIRED), non-negative integer. The length of
                                  a scalar octet representation, defined
                                  by the ciphersuite.
- octet_point_length (REQUIRED), non-negative integer. The length of
                                 a point in G1 octet representation,
                                 defined by the ciphersuite.

Outputs:

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

Procedure:

1.  proof_len_floor = 3 * octet_point_length + 5 * octet_scalar_length
2.  if length(proof_octets) < proof_len_floor, return INVALID

// Points (i.e., (A', Abar, D) in ProofGen) de-serialization.
3.  index = 0
4.  for i in range(0, 2):
5.      end_index = index + octet_point_length - 1
6.      A_i = octets_to_point_g1(proof_octets[index..end_index])
7.      if A_i is INVALID or Identity_G1, return INVALID
8.      index += octet_point_length

// Scalars (i.e., (c, e^, r2^, r3^, s^, (m^_j1, ..., m^_jU)) in
// ProofGen) de-serialization.
9.  j = 0
10. while index < length(proof_octets):
11.     end_index = index + octet_scalar_length - 1
12.     s_j = OS2IP(proof_octets[index..end_index])
13.     if s_j = 0 or if s_j >= r, return INVALID
14.     index += octet_scalar_length
15.     j += 1

16. if index != length(proof_octets), return INVALID
17. msg_commitments = ()
18. If j > 5, set msg_commitments = (s_5, ..., s_(j-1))
19. return (A_0, A_1, A_2, s_0, s_1, s_2, s_3, s_4, msg_commitments)
```

4.4.4. ProofToOctets

This operation describes how to encode a proof, as computed at step 25 in ProofGen, to an octet string. The input to the operation MUST be a valid proof.

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

1. Three (3) valid compressed points of the G1 subgroup, different from the identity point of G1 (i.e., `A', Abar, D`, in ProofGen)
2. Five (5) integers representing scalars in the range of 1 to r-1 inclusive (i.e., `c, e^, r2^, r3^, s^`, 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).

```
proof_octets = proof_to_octets(proof)

Inputs:

- proof (REQUIRED), a BBS proof in the form calculated by ProofGen in
                    step 25 (see above).

Parameters:

- octet_scalar_length (REQUIRED), non-negative integer. The length of
                                  a scalar octet representation, defined
                                  by the ciphersuite.

Outputs:

- proof_octets, octet string.

Procedure:

1. (A', Abar, D, c, e^, r2^, r3^, s^, (m^_1, ..., m^_U)) = proof
2. Let proof_octets be an empty octet string.

// Points Serialization.
3. for point in (A', Abar, D):
4.     point_octets = point_to_octets_g1(point)
5.     proof_octets = proof_octets || point_octets

// Scalar Serialization.
6. for scalar in (c, e^, r2^, r3^, s^, m^_1, ..., m^_U):
7.     scalar_octets = I2OSP(scalar, octet_scalar_length)
8.     proof_octets = proof_octets || scalar_octets
9. return proof_octets
```

4.4.5. OctetsToPublicKey

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, octet string. A public key in the form ouputted by the SkToPK
      operation

Outputs:

- W, a valid point in G2 or INVALID

Procedure:

1. W = octets_to_point_g2(PK)
2. If W is INVALID, return INVALID
3. if subgroup_check(W) is INVALID, return INVALID
4. If W == Identity_G2, return INVALID
5. return W
```

4.4.6. EncodeForHash

This document uses the `hash_to_scalar` function to hash elements to scalars in the multiplicative group mod r (see Section 5.3). To avoid ambiguity, elements passed to that operation, must first be encoded appropriately using `encode_for_hash`. The following procedure describes how to encode each element accordingly by serializing it to an appropriate format depending on its type and concatenating the results. Specifically,

- Points in G1 or G2 will be encoded using the `point_to_octets_g*` implementation for a particular ciphersuite.
- Non-negative integers will be encoded using `I2OSP` with an output length of 8 bytes.
- Scalars will be zero-extended to a fixed length, defined by a particular ciphersuite.
- Octet strings will be zero-extended to a length that is a multiple of 8 bits. Then, the extended value is encoded directly.
- ASCII strings will be transformed into octet strings using UTF-8 encoding.

After encoding, octet strings will be prepended with a value representing the length of their binary representation in the form of the number of bytes. This length must be encoded to octets using I2OSP with output length of 8 bytes. The combined value (encoded value + length prefix) binary representation is then encoded as a single octet string. For example, the string `0x14d` will be encoded as `0x00000000000000002014d`. If the length of the octet string is larger than 2^64 - 1, the octet string must be rejected. Similarly, ASCII strings, after encoded to octets (using utf8), will also be appended with the length of their octet-string representation.

An exception to the above rule is octet strings that represent a public key or ASCII strings that represent a ciphersuite ID, since those have a constant, well defined by each ciphersuite, length.

Optional input/parameters to operations that feature in a call to hash_to_scalar, that are not supplied to the operation should default to an empty octet string. For example, if X is an optional input/parameter that is not supplied, whilst A and B are required, then the procedural step of `hash(A || X || B)` MUST be evaluated to `hash(A || "" || B)`.

The above is further described in the following procedure.

```
result = encode_for_hash(input_array)

Inputs:

- input_array, an array of elements to be hashed. All elements of this
               array that are octet strings MUST be multiples of 8 bits.

Parameters:

- octet_scalar_length, non-negative integer. The length of a scalar
                       octet representation, defined by the ciphersuite.

Outputs:

- result, an octet string or INVALID.

Procedure:

1.  let octets_to_hash be an empty octet string.
2.  for el in input_array:
3.      if el is an ASCII string: el = utf8(el)
4.      if el is an octet string representing a public key:
5.          el_octs = el
6.      else if el is representing a utf8 encoded Ciphersuite ID:
7.          el_octs = el
8.      else if el is an octet string:
9.          if length(el) > 2^64 - 1, return INVALID
10.         el_octs = I2OSP(length(el), 8) || el
11.     else if el is a Point in G1: el_octs = point_to_octets_g1(el)
12.     else if el is a Point in G2: el_octs = point_to_octets_g2(el)
10.     else if el is a Scalar: el_octs = I2OSP(el, octet_scalar_length)
11.     else if el is a non-negative integer: el_octs = I2OSP(el, 8)
12.     else: return INVALID
13.     octets_to_hash = octets_to_hash || el_octs
14. return octets_to_hash
```

5. Security Considerations

5.1. Validating public keys

It is RECOMENDED for any operation in Core Operations involving public keys, that they deserialize the public key first using the OctetsToPublicKey operation, even if they only require the octet-string representation of the public key. If the `octets_to_pubkey` procedure (see the OctetsToPublicKey section) returns INVALID, the calling operation should also return INVALID and abort. An example of where this recommendation applies is the Sign operation. An example of where an explicit invocation to the `octets_to_pubkey` operation is already defined and therefore required is the Verify operation.

5.2. Point de-serialization

This document makes use of `octet_to_point_g*` to parse octet strings to elliptic curve points (either in G1 or G2). It is assumed (even if not explicitly described) that the result of this operation will not be INVALID. If `octet_to_point_g*` returns INVALID, then the calling operation should immediately return INVALID as well and abort the operation. Note that the only place where the output is assumed to be VALID implicitly is in the EncodingForHash section.

5.3. Skipping membership checks

Some existing implementations skip the subgroup_check invocation in Verify, whose purpose is ensuring that the signature is an element of a prime-order subgroup. This check is REQUIRED of conforming implementations, for two reasons.

1. For most pairing-friendly elliptic curves used in practice, the pairing operation e Section 1.2 is undefined when its input points are not in the prime-order subgroups of E1 and E2. The resulting behavior is unpredictable, and may enable forgeries.
2. Even if the pairing operation behaves properly on inputs that are outside the correct subgroups, skipping the subgroup check breaks the strong unforgeability property [ADR02].

5.4. Side channel attacks

Implementations of the signing algorithm SHOULD protect the secret key from side-channel attacks. One method for protecting against certain side-channel attacks is ensuring that the implementation executes exactly the same sequence of instructions and performs exactly the same memory accesses, for any value of the secret key. In other words, implementations on the underlying pairing-friendly elliptic curve SHOULD run in constant time.

5.5. Randomness considerations

The IKM input to KeyGen MUST be infeasible to guess and MUST be kept secret. One possibility is to generate IKM from a trusted source of randomness. Guidelines on constructing such a source are outside the scope of this document.

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.

BBS proofs are nondeterministic, meaning care must be taken against attacks arising from using bad randomness, for example, the nonce reuse attack on ECDSA [HDWH12]. It is RECOMMENDED that the presentation header used in this specification contain a nonce chosen at random from a trusted source of randomness, see the Section 5.6 for additional considerations.

When a trusted source of randomness is used, signatures and proofs are much harder to forge or break due to the use of multiple nonces.

5.6. 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.

5.7. 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 [I-D.irtf-cfrg-hash-to-curve].

In addition, ciphersuites MUST specify unique domain separation tags for hash_to_curve. Some guidance around defining this can be found in Section 6.

5.8. Choice of underlying curve

BBS signatures can be implemented on any pairing-friendly curve. However care MUST be taken when selecting one that is appropriate, this specification defines a ciphersuite for using the BLS12-381 curve in Section 6 which as a curve achieves around 117 bits of security according to a recent NCC ZCash cryptography review [ZCASH-REVIEW].

5.9. Security of proofs generated by ProofGen

The proof, as returned by ProofGen, is a zero-knowledge proof-of-knowledge [CDL16]. This guarantees that no information will be revealed about the signature itself or the undisclosed messages, from the output of ProofGen. Note that the security proofs in [CDL16] work on type 3 pairing setting. This means that G1 should be different from G2 and with no efficient isomorphism between them.

6. 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].

6.1. Ciphersuite Format

6.1.1. Ciphersuite ID

The following section defines the format of the unique identifier for the ciphersuite denoted `ciphersuite_id`. 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 string indicating any additional information used to uniquely qualify the ciphersuite. When present this value MUST only contain ASCII 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).

6.1.2. 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_g1 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_g*` function. It is RECOMMENDED that this value is set to `ceil(log2(p)/8)`.
- hash_to_curve_suite: The hash-to-curve ciphersuite id, in the form defined in [I-D.irtf-cfrg-hash-to-curve]. This defines the hash_to_curve_g1 (the hash_to_curve operation for the G1 subgroup, see the Notation section) and the expand_message (either expand_message_xmd or expand_message_xof) operations used in this document.
- P1: A fixed point in the G1 subgroup.

Serialization functions:

- point_to_octets_g1: a function that returns the canonical representation of the point P for the G1 subgroup as an octet string.
- point_to_octets_g2: a function that returns the canonical representation of the point P for the G2 subgroup as an octet string.
- octets_to_point_g1: a function that returns the point P in the subgroup G1 corresponding to the canonical representation ostr, or INVALID if ostr is not a valid output of `point_to_octets_g1`.
- octets_to_point_g2: a function that returns the point P in the subgroup G2 corresponding to the canonical representation ostr, or INVALID if ostr is not a valid output of `point_to_octets_g2`.

Generator parameters:

- generator_seed: The seed used to determine the generator points which form part of the public parameters used by the BBS signature scheme. Note there are multiple possible scopes for this seed, including: a globally shared seed (where the resulting message generators are common across all BBS signatures); a signer specific seed (where the message generators are specific to a signer); and a signature specific seed (where the message generators are specific per signature). The ciphersuite MUST define this seed OR how to compute it as a pre-cursor operation to any others.

6.2. 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 first ciphersuite makes use of an extendable output function, and most specifically of SHAKE-256, as defined in Section 6.2 of [SHA3]. It also uses the hash-to-curve suite defined by this document in Appendix A.1, which also makes use of the SHAKE-256 function.

The second ciphersuite uses SHA-256, as defined in Section 6.2 of [SHA2] and the BLS12-381 G1 hash-to-curve suite defined in Section 8.8.1 of the [I-D.irtf-cfrg-hash-to-curve] document.

Note that these two ciphersuites differ only in the hash function (SHAKE-256 vs SHA-256) and 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.

6.2.1. BLS12-381-SHAKE-256

Basic parameters:

- ciphersuite_id: "BBS_BLS12381G1_XOF:SHAKE-256_SSWU_RO_"
- hash: SHAKE-256 as defined in [SHA3].
- 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.
- P1: The G1 point returned from the `create_generators` procedure, with generator_seed = "BBS_BLS12381G1_XOF:SHAKE-256_SSWU_RO_BP_MESSAGE_GENERATOR_SEED". More specifically,

```
P1 =
91b784eaac4b2b2c6f9bfb2c9eae97e817dd12bba49a0821d175a50f1632465b319ca9f
b
81dda3fb0434412185e2cca5
```

Serialization functions:

- point_to_octets_g1: follows the format documented in Appendix C section 1 of [I-D.irtf-cfrg-pairing-friendly-curves] for the G1 subgroup, using compression (i.e., setting C_bit = 1).
- point_to_octets_g2: follows the format documented in Appendix C section 1 of [I-D.irtf-cfrg-pairing-friendly-curves] for the G2 subgroup, using compression (i.e., setting C_bit = 1).
- octets_to_point_g1: follows the format documented in Appendix C section 2 of [I-D.irtf-cfrg-pairing-friendly-curves] for the G1 subgroup.
- octets_to_point_g2: follows the format documented in Appendix C section 2 of [I-D.irtf-cfrg-pairing-friendly-curves] for the G2 subgroup.

Generator parameters:

- generator_seed: A global seed value of utf8("BBS_BLS12381G1_XOF:SHAKE-256_SSWU_RO_MESSAGE_GENERATOR_SEED") which is used by the create_generators operation to compute the required set of message generators.

6.2.2. BLS12-381-SHA-256

Basic parameters:

- Ciphersuite_ID: "BBS_BLS12381G1_XMD:SHA-256_SSWU_RO_"
- hash: SHA-256 as defined in [SHA2].
- 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 [I-D.irtf-cfrg-hash-to-curve] for the G1 subgroup.
- P1: The G1 point returned from the `create_generators` procedure, with generator_seed = "BBS_BLS12381G1_XMD:SHA-256_SSWU_RO_BP_MESSAGE_GENERATOR_SEED". More specifically,

```
P1 =
8533b3fbea84e8bd9ccee177e3c56fbe1d2e33b798e491228f6ed65bb4d1e0ada07bcc4
4
89d8751f8ba7a1b69b6eecd7
```

Serialization functions:

- point_to_octets_g1: follows the format documented in Appendix C section 1 of [I-D.irtf-cfrg-pairing-friendly-curves] for the G1 subgroup, using compression (i.e., setting C_bit = 1).
- point_to_octets_g2: follows the format documented in Appendix C section 1 of [I-D.irtf-cfrg-pairing-friendly-curves] for the G2 subgroup, using compression (i.e., setting C_bit = 1).

- octets_to_point_g1: follows the format documented in Appendix C section 2 of [I-D.irtf-cfrg-pairing-friendly-curves] for the G1 subgroup.
- octets_to_point_g2: follows the format documented in Appendix C section 2 of [I-D.irtf-cfrg-pairing-friendly-curves] for the G2 subgroup.

Generator parameters:

- generator_seed: A global seed value of utf8("BBS_BLS12381G1_XMD:SHA-256_SSWU_RO_MESSAGE_GENERATOR_SEED") which is used by the create_generators operation to compute the required set of message generators.

7. Test Vectors

The following section details a basic set of test vectors that can be used to confirm an implementations correctness

NOTE All binary data below is represented as octet strings encoded in hexadecimal format

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

7.1. Key Pair

The following key pair will be used for the test vectors of both ciphersuites. Note that it is made based on the BLS12-381-SHA-356 ciphersuite, meaning that it uses SHA-256 as a hash function. Although KeyGen is not REQUIRED for ciphersuite compatibility, it is RECOMMENDED that implementations will NOT re-use keys across different ciphersuites (even if they are based on the same curve).

NOTE: this is work in progress and in the future, we may add different key pairs per ciphersuite for the test vectors.

Following the procedure defined in Section 3.3.1 with an input IKM value as follows

```
746869732d49532d6a7573742d616e2d546573742d494b4d2d746f2d67656e6572617465
2d246528724074232d6b6579
```

Outputs the following SK value

```
47d2ede63ab4c329092b342ab526b1079dbc2595897d4f2ab2de4d841cbe7d56
```

Following the procedure defined in Section 3.3.2 with an input SK value as above produces the following PK value

```
b65b7cbff4e81b723456a13936b6bcc77a078bf6291765f3ae13170072249dd7daa7ec1b
d82b818ab60198030b45b8fa159c155fc3841a9ad4045e37161c9f0d9a4f361b93cfdc67
d365f3be1a398e56aa173d7a55e01b4a8dd2494e7fb90da7
```

7.2. Messages

The following messages are used by the test vectors of both ciphersuites (unless otherwise stated).

```
9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02

87a8bd656d49ee07b8110e1d8fd4f1dcef6fb9bc368c492d9bc8c4f98a739ac6

96012096adda3f13dd4adbe4eea481a4c4b5717932b73b00e31807d3c5894b90

ac55fb33a75909edac8994829b250779298aa75d69324a365733f16c333fa943

d183ddc6e2665aa4e2f088af9297b78c0d22b4290273db637ed33ff5cf703151

515ae153e22aae04ad16f759e07237b43022cb1ced4c176e0999c6a8ba5817cc

496694774c5604ab1b2544eababcf0f53278ff5040c1e77c811656e8220417a2

77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c23364568523f8b91

7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b7320912416

c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80
```

7.3. BLS12-381-SHAKE-256 Test Vectors

Test vectors of the BLS12-381-SHAKE-256 ciphersuite. Further fixtures are available in additional BLS12-381-SHAKE-256 test vectors.

7.3.1. Map Messages to Scalars

The messages in Section 7.2 must be mapped to scalars before passed to the Sign, Verify, ProofGen and ProofVerify operations. For the purpose of the test vectors presented in this document we are using the MapMessageToScalarAsHash operation to map each message to a scalar. For the BLS12-381-SHAKE-256 ciphersuite, on input each message in Section 7.2 and the following default dst

```
4242535f424c53313233383147315f584f463a5348414b452d3235365f535357555f524f
5f4d41505f4d53475f544f5f5343414c41525f41535f484153485f
```

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

```
4e67c49cf68df268bca0624880770bb57dbe8460c89883cc0ac496785b68bbe9

12d92c990f37ffab1c6ac4b0cd83378ffb8a8610259d62d3b885fc4c1bc50f7f

41a157520e8752ca100a365ffde4683fb9610bf105b40933bb98dcacbbd56ace

3344daad11febac28f0f8e3740cd2921fd6da18ebc7e9692a8287cedea5f4bf4

0407198a8ffc4640b840fc924e5308f405ca86035d05366718aafd0b688876f3

1918fa78c85628cb3ac705cc4843197d3fce88c8132d9242d87201e65a4d3743

0a272f853369d70526d7bd37281bb87d1c8db7d0975dd833812bb9d264f4b0eb

00776f91d1ecb5cc01ffe155ae05efea0b820f3d40bada5142bb852f9922b7e1

3902ced42427bca88822f818912d2f4c0d88ba1d1fc7a9b0e2321674a5d53f27

397864d9292b1f4a5fff5fa33088ed8e1a9ec52346dbd5f66ee0f978bd67595d
```

Note that in both the following test vectors, as well as the additional BLS12-381-SHAKE-256 test vectors in Appendix C.1, when we are referring to a message that will be passed to one of the Sign, Verify, ProofGen or ProofVerify operations, we assume that it will first be mapped into one of the above scalars, using the MapMessageToScalarAsHash operation.

7.3.2. Message Generators

Following the procedure defined in Section 4.1 with an input count value of 12, for the BLS12-381-SHAKE-256 suite, outputs the following values (note that the first 2 correspond to `Q_1` and `Q_2`, while the next 10, to the message generators `H_1, ..., H_10`).

```
b60acd4b0dc13b580394d2d8bc6c07d452df8e2a7eff93bc9da965b57e076cae640c2858
fb0c2eaf242b1bd11107d635

ad03f655b4c94f312b051aba45977c924bc5b4b1780c969534c183784c7275b70b876db6
41579604328c0975eaa0a137

b63ae18d3edd64a2edd381290f0c68bebabaf3d37bc9dbb0bd5ad8daf03bbd2c48260255
ba73f3389d2d5ad82303ac25

b0b92b79a3e1fc59f39c6b9f78f00b873121c6a4c1814b94c07848efd172762fefbc4844
7a16f9ba8ed1b638e2933029

b671ed7256777fb5b82f66d1268d03492a1cecc19fd327d56e100cce69c2e15fcd03dcdc
fe6b2d42aa039edcd58092f4

867009da287e1186884084ed71477ce9bd401e0bf4a7be48e2af0a3a4f2e7e21d2b7bb0f
fdc4c03b5aa9672c3c76e0c9

a3a10489bf1a244753e864454fd24ed8c312f737c0c2a529905222509199a0b48715a048
cd93d134dac2cd4934c549bb

81d548904ec8aa58b3f56f69c3f543fb73f339699a33df82c338cad9657b70c457b735c4
ae96e8ea0c1ea0da65059d95

b4bbc2a56104c2289fc7688fef30222746467df27698b6c2d53dad5477fd05b7ec8a8412
2b8122c1de2d2f16750d2a92

ae22a4e89029d3507b8e40af3531b114b564cc77375c249036926e6973f69d21b356e734
cdeda47fd320035781eda7df

98b266b03b9cea3d466bafbcd2e1c600c40cba8817d52d46ea77612df911a6e6c0406352
11fc1bffd4ca914afca1ce55

b458cd3d7af0b5ceea335436a66e2015b216467c204b850b15547f68f6f2a209e8229d15
4d4f998c7b96aa4f88cdca15
```

7.3.3. Valid Single Message Signature

Using the following header

```
11223344556677889900aabbccddeeff
```

And the following message (the first message defined in Section 7.2)

```
9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02
```

After it is mapped to the first scalar in Section 7.3.1, along with the SK value as defined in Section 7.1 as inputs into the Sign operations, yields the following output signature

```
8d80b2aab1e4ddaa15194313e55e28b22cf09e42e5d47819720d56d9cb2f85b5a84918a0
0fcc1e21d95f9b80fb66cd4a66bdea8d576240067be369751a7359b1d2318531a29fbdfe
58e59a13e1e3d803627cb3197313b3b891a5facf95ad4cf8d7750fc99e036da6d1f8267f
163986b6
```

7.3.4. Valid Multi-Message Signature

Using the following header

```
11223344556677889900aabbccddeeff
```

And the messages defined in Section 7.2 (**Note** the ordering of the messages MUST be preserved), after they are mapped to the scalars in Section 7.3.1, along with the SK value as defined in Section 7.1 as inputs into the Sign operations, yields the following output signature

```
9157456791e4f9cae1130372f7cf37709ba661e43df5c23cc1c76be91abff7e2603e2dda
aa71fc42bd6f9d44bd58315b09ee5cc4e7614edde358f2c497b6b05c8b118fae3f71a52a
f482dceffccb37851907573c03d2890dffbd1f660cdf89c425d4e0498bbf73dd96ff15ad
9a8b581a
```

7.4. BLS12381-SHA-256 Test Vectors

Test vectors of the BLS12-381-SHA-256 ciphersuite. Further fixtures are available in additional BLS12-381-SHA-256 test vectors.

7.4.1. Map Messages to Scalars

Similarly to how messages are mapped to scalars in BLS12381-SHAKE-256 Test Vectors, we are using the MapMessageToScalarAsHash operation to map each message to a scalar. For the BLS12-381-SHA-256 ciphersuite, on input each message in Section 7.2 and the following default dst

```
4242535f424c53313233383147315f584d443a5348412d3235365f535357555f524f5f4d
41505f4d53475f544f5f5343414c41525f41535f484153485f
```

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

```
360097633e394c22601426bd9f8d5b95f1c64f89689deee230e817925dee4724

1e68fedccc3d68236c7e4ccb508ebb3da5d5de1864eac0a0f683de22752e6d28

4d20cb09c2ac1e1c572e83f355b90ea996c7a6ab98a03a98098c1abeb8c7a195

038f1892656f7753eb2be1ab3679dd0d0331fb5e7be0f72550dbe22b0f36df02

144cb3d17379b746217a93910a8ca07ca7248be3d1972b562010a4e09a27b7f3

5da7360f11d5133c6ed6e54c1a1fd230a2c9256d5b6be0b41219808bfc964d28

100e944c0a82da79b062a9cc2b014d16b345b4e4624e7408106fe282da0635cc

2004000723ef8997256f5f2a86cbef353c3034ab751092033fa0c0a844d639af

68a1f58bb5aaa3bc89fba6c40ccd761879fdadf336565cef9812ed5dba5d56ca

1aefbeb8e206723a37fc2e7f8eded8227d960bed44b7089fec0d7e6da93e5d38
```

Note that in both the following test vectors, as well as the additional BLS12-381-SHA-256 test vectors in Appendix C.2, when we are referring to a message that will be passed to one of the Sign, Verify, ProofGen or ProofVerify operations, we assume that it will first be mapped into one of the above scalars, using the MapMessageToScalarAsHash operation.

7.4.2. Message Generators

Following the procedure defined in Section 4.1 with an input count value of 12, for the BLS12-381-SHA-256 suite, outputs the following values (note that the first 2 correspond to `Q_1` and `Q_2`, while the next 10, to the message generators `H_1, ..., H_10`).

```
b57ec5e001c28d4063e0b6f5f0a6eee357b51b64d789a21cf18fd11e73e73577910182d4
21b5a61812f5d1ca751fa3f0

909573cbb9da401b89d2778e8a405fdc7d504b03f0158c31ba64cdb9b648cc35492b18e5
6088b44c8b4dc6310afb5e49

90248350d94fd550b472a54269e28b680757d8cbbe6bb2cb000742c07573138276884c28
72a8285f4ecf10df6029be15

8fb7d5c43273a142b6fc445b76a8cdfc0f96c5fdac7cdd73314ac4f7ec4990a0a6f28e4a
d97fb0a3a22efb07b386e3ff

8241e3e861aaac2a54a8d7093301143d7d3e9911c384a2331fcc232a3e64b4882498ce4d
9da8904ffcbe5d6eadafc82b

99bb19d202a4019c14a36933264ae634659994076bf02a94135e1026ea309c7d3fd6da60
c7929d30b656aeaba7c0dcec

81779fa5268e75a980799c0a01677a763e14ba82cbf0a66c653edc174057698636507ac5
8e73522a59585558dca80b42

98a3f9af71d391337bc6ae5d26980241b6317d5d71570829ce03d63c17e0d2164e1ad793
645e1762bfcc049a17f5994b

aca6a84770bb1f515591b4b95d69777856ddc52d5439325839e31ce5b6237618a9bc01a0
4b0057d33eab14341504c7e9

b96e206d6cf32b51d2f4d543972d488a4c4cbc5d994f6ebb0bdffbc5459dcb9a8e5ab045
c5949dc7eb33b0545b62aae3

8edf840b56ecf8d7c5a9c4a0aaf8a5525f3480df735743298dd2f4ae1cbb56f56ed6a04e
f6fa7c92cd68d9101c7b8c8f

86d4ae04738dc082eb37e753bc8ec35a8d982e463559214d0f777599f71aa1f95780b3dc
cbdcae45e146e5c7623dfe7d
```

7.4.3. Valid Single Message Signature

Using the following header

```
11223344556677889900aabbccddeeff
```

And the following message (the first message defined in Section 7.2)

```
9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02
```

After it is mapped to the first scalar in Section 7.4.1, along with the SK value as defined in Section 7.1 as inputs into the Sign operations, yields the following output signature

```
8d80b2aab1e4ddaa15194313e55e28b22cf09e42e5d47819720d56d9cb2f85b5a84918a0
0fcc1e21d95f9b80fb66cd4a66bdea8d576240067be369751a7359b1d2318531a29fbdfe
58e59a13e1e3d803627cb3197313b3b891a5facf95ad4cf8d7750fc99e036da6d1f8267f
163986b6
```

7.4.4. Valid Multi-Message Signature

Using the following header

```
11223344556677889900aabbccddeeff
```

And the messages defined in Section 7.2 (**Note** the ordering of the messages MUST be preserved), after they are mapped to the scalars in Section 7.4.1, along with the SK value as defined in Section 7.1 as inputs into the Sign operations, yields the following output signature

```
9157456791e4f9cae1130372f7cf37709ba661e43df5c23cc1c76be91abff7e2603e2dda
aa71fc42bd6f9d44bd58315b09ee5cc4e7614edde358f2c497b6b05c8b118fae3f71a52a
f482dceffccb37851907573c03d2890dffbd1f660cdf89c425d4e0498bbf73dd96ff15ad
9a8b581a
```

8. IANA Considerations

This document does not make any requests of IANA.

9. 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] and in [CDL16]. 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 and Tomislav Markovski

10. Normative References

[I-D.irtf-cfrg-hash-to-curve] Faz-Hernandez, A., Scott, S., Sullivan, N., Riad Wahby, S., and A. Christopher Wood, "Hashing to Elliptic Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-hash-to-curve-16, 15 June 2022, <https://www.ietf.org/archive/id/draft-irtf-cfrg-hash-to-curve-16.txt>.

[I-D.irtf-cfrg-pairing-friendly-curves] Sakemi, Y., Kobayashi, T., Saito, T., and S. Riad Wahby, "Pairing-Friendly Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-pairing-friendly-curves-10, 30 July 2021, <https://www.ietf.org/archive/id/draft-irtf-cfrg-pairing-friendly-curves-10.txt>.

[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>.

[RFC4868] Kelly, S. and S. Frankel, "Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec", RFC 4868, DOI 10.17487/RFC4868, May 2007, <https://www.rfc-editor.org/info/rfc4868>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-

Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[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>.

[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>.

11. Informative References

[ADR02] An, J. H., Dodis, Y., and T. Rabin, "On the Security of Joint Signature and Encryption", 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", Springer, Berlin, Heidelberg, 2006, <https://link.springer.com/chapter/10.1007/11832072_8>.

[BBS04] Boneh, D., Boyen, X., and H. Shacham, "Short Group Signatures", 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", Springer, Cham, 2016, <https://eprint.iacr.org/2016/663.pdf>.

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

[I-D.irtf-cfrg-bls-signature] Boneh, D., Gorbunov, S., Riad Wahby, S., Wee, H., Christopher Wood, A., and Z. Zhang, "BLS Signatures", Work in Progress, Internet-Draft, draft-irtf-cfrg-bls-signature-05, 16 June 2022, <https://www.ietf.org/archive/id/draft-irtf-cfrg-bls-signature-05.txt>.

[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_20101-30_v1.3.pdf>.

Appendix A. BLS12-381 hash_to_curve definition using SHAKE-256

The following defines a hash_to_curve suite [I-D.irtf-cfrg-hash-to-curve] 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 [I-D.irtf-cfrg-hash-to-curve].

Note the notation used in the below definitions is sourced from [I-D.irtf-cfrg-hash-to-curve].

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
                 [@!I-D.irtf-cfrg-hash-to-curve])

* 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
                  [@!I-D.irtf-cfrg-hash-to-curve])

* hash: SHAKE-256

* L: 64

* f: Simplified SWU for AB == 0 (Section 6.6.3 of
     [@!I-D.irtf-cfrg-hash-to-curve])

* 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
            [@!I-D.irtf-cfrg-hash-to-curve]

*  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 [I-D.irtf-cfrg-hash-to-curve].

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

Appendix B. Use Cases

B.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 JWP format and serialization). 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.

B.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 (https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop-04) 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.

B.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 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 C. Additional Test Vectors

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

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

C.1.1. Modified Message Signature

Using the following header

```
11223344556677889900aabbccddeeff
```

And the following message (the first message defined in Section 7.2)

```
c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80
```

After is mapped to the first scalar in Section 7.3.1, and with the following signature

```
8d80b2aab1e4ddaa15194313e55e28b22cf09e42e5d47819720d56d9cb2f85b5a84918a0
0fcc1e21d95f9b80fb66cd4a66bdea8d576240067be369751a7359b1d2318531a29fbdfe
58e59a13e1e3d803627cb3197313b3b891a5facf95ad4cf8d7750fc99e036da6d1f8267f
163986b6
```

Along with the PK value as defined in Section 7.1 as inputs into the Verify operation should fail signature validation due to the message value being different from what was signed

C.1.2. Extra Unsigned Message Signature

Using the following header

```
11223344556677889900aabbccddeeff
```

And the following messages (the two first messages defined in Section 7.2)

```
9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02

87a8bd656d49ee07b8110e1d8fd4f1dcef6fb9bc368c492d9bc8c4f98a739ac6
```

After they are mapped to the first 2 scalars in Section 7.3.1, and with the following signature (which is a signature to only the first of the above two messages)

```
8d80b2aab1e4ddaa15194313e55e28b22cf09e42e5d47819720d56d9cb2f85b5a84918a0
0fcc1e21d95f9b80fb66cd4a66bdea8d576240067be369751a7359b1d2318531a29fbdfe
58e59a13e1e3d803627cb3197313b3b891a5facf95ad4cf8d7750fc99e036da6d1f8267f
163986b6
```

Along with the PK value as defined in Section 7.1 as inputs into the Verify operation should fail signature validation due to an additional message being supplied that was not signed.

C.1.3. Missing Message Signature

Using the following header

```
11223344556677889900aabbccddeeff
```

And the following messages (the two first messages defined in Section 7.2)

```
9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02

87a8bd656d49ee07b8110e1d8fd4f1dcef6fb9bc368c492d9bc8c4f98a739ac6
```

After they are mapped to the first 2 scalars in Section 7.3.1, and with the following signature (which is a signature on all the messages defined in Section 7.2)

```
9157456791e4f9cae1130372f7cf37709ba661e43df5c23cc1c76be91abff7e2603e2dda
aa71fc42bd6f9d44bd58315b09ee5cc4e7614edde358f2c497b6b05c8b118fae3f71a52a
f482dceffccb37851907573c03d2890dffbd1f660cdf89c425d4e0498bbf73dd96ff15ad
9a8b581a
```

Along with the PK value as defined in Section 7.1 as inputs into the Verify operation should fail signature validation due to missing messages that were originally present during the signing.

C.1.4. Reordered Message Signature

Using the following header

```
11223344556677889900aabbccddeeff
```

And the following messages (re-ordering of the messages defined in Section 7.2)

```
c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80

7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b7320912416

77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c23364568523f8b91

496694774c5604ab1b2544eababcf0f53278ff5040c1e77c811656e8220417a2

515ae153e22aae04ad16f759e07237b43022cb1ced4c176e0999c6a8ba5817cc

d183ddc6e2665aa4e2f088af9297b78c0d22b4290273db637ed33ff5cf703151

ac55fb33a75909edac8994829b250779298aa75d69324a365733f16c333fa943

96012096adda3f13dd4adbe4eea481a4c4b5717932b73b00e31807d3c5894b90

87a8bd656d49ee07b8110e1d8fd4f1dcef6fb9bc368c492d9bc8c4f98a739ac6

9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02
```

After they are mapped to the corresponding scalars in Section 7.3.1, and with the following signature

```
9157456791e4f9cae1130372f7cf37709ba661e43df5c23cc1c76be91abff7e2603e2dda
aa71fc42bd6f9d44bd58315b09ee5cc4e7614edde358f2c497b6b05c8b118fae3f71a52a
f482dceffccb37851907573c03d2890dffbd1f660cdf89c425d4e0498bbf73dd96ff15ad
9a8b581a
```

Along with the PK value as defined in Section 7.1 as inputs into the Verify operation should fail signature validation due to messages being re-ordered from the order in which they were signed

C.1.5. Wrong Public Key Signature

Using the following header

```
11223344556677889900aabbccddeeff
```

And the messages as defined in Section 7.2, mapped to the scalars in Section 7.3.1 and with the following signature

```
9157456791e4f9cae1130372f7cf37709ba661e43df5c23cc1c76be91abff7e2603e2dda
aa71fc42bd6f9d44bd58315b09ee5cc4e7614edde358f2c497b6b05c8b118fae3f71a52a
f482dceffccb37851907573c03d2890dffbd1f660cdf89c425d4e0498bbf73dd96ff15ad
9a8b581a
```

Along with the PK value as defined in Section 7.1 as inputs into the Verify operation should fail signature validation due to public key used to verify is incorrect

C.1.6. Wrong Header Signature

Using the following header

```
ffeeddccbbaa00998877665544332211
```

And the messages as defined in Section 7.2, mapped to the scalars in Section 7.3.1 and with the following signature

```
9157456791e4f9cae1130372f7cf37709ba661e43df5c23cc1c76be91abff7e2603e2dda
aa71fc42bd6f9d44bd58315b09ee5cc4e7614edde358f2c497b6b05c8b118fae3f71a52a
f482dceffccb37851907573c03d2890dffbd1f660cdf89c425d4e0498bbf73dd96ff15ad
9a8b581a
```

Along with the PK value as defined in Section 7.1 as inputs into the Verify operation should fail signature validation due to header value being modified from what was originally signed

C.1.7. Hash to Scalar Test Vectors

Using the following input message,

```
9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02
```

And the default dst defined in hash-to-scalar, i.e.,

```
4242535f424c53313233383147315f584f463a5348414b452d3235365f535357555f524f
5f4832535f
```

With an output count of `1`, we get the following scalar, encoded with I2OSP and represented in big endian order,

```
260cab748e24ccc2bbd66f5b834d692622fa131f5ce898fa57217434c9ed14fa
```

With the same input message and dst but with an output count of `10` we get the following scalars (again encoded with I2OSP and represented in big endian order),

```
5c6e62607c16397ee6d9624673be9a7ddacbc7b7dd290bdb853cf4c74a34de0a

2a3524e43413a5d1b34c4c8ed119c4c5a2f9b84392ff0fea0d34e1be44ceafbc

4b649b82eed1e62117d91cd8d22438e72f3f931a0f8ad683d1ade253333c472a

64338965f1d37d17a14b6f431128c0d41a7c3924a5f484c282d20205afdfdb8f

0dfe01c01ff8654e43a611b76aaf4faec618a50d85d34f7cc89879b179bde3d5

6b6935016e64791f5d719f8206284fbe27dbb8efffb4141512c3fbfbfa861a0f

0dfe13f85a36df5ebfe0efac3759becfcc2a18b134fd22485c151db85f981342

5071751012c142046e7c3508decb0b7ba9a453d06ce7787189f4d93a821d538e

5cdae3304e745553a75134d914db5b282cc62d295e3ed176fb12f792919fd85e

32b67dfbba729831798279071a39021b66fd68ee2e68684a0f6901cd6fcb8256
```

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

C.2.1. Modified Message Signature

Using the following header

```
11223344556677889900aabbccddeeff
```

And the following message (the first message defined in Section 7.2)

```
c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80
```

After is maped to the first scalar in Section 7.4.1, and with the following signature

```
acb90d6e4db7b6eb260f1ece4fd2c1b9b92f190c9bd2be900ad5a92b80cc23db9b6d3144
447cbb5f7af7c615d9a6b9ee0ba0e713ad2b77feba7c40a18b02e50ffbe0ecb13849c1c6
ed496f6a21300c145a1e7e0d4cb283607f6300e40ea411b6d6255967b3298765b22c6f1d
c91fa5ab
```

Along with the PK value as defined in Section 7.1 as inputs into the Verify operation should fail signature validation due to the message value being different from what was signed.

C.2.2. Extra Unsigned Message Signature

Using the following header

```
11223344556677889900aabbccddeeff
```

And the following messages (the two first messages defined in Section 7.2)

```
9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02

87a8bd656d49ee07b8110e1d8fd4f1dcef6fb9bc368c492d9bc8c4f98a739ac6
```

After they are mapped to the first 2 scalars in Section 7.4.1, and with the following signature (which is a signature to only the first of the above two messages)

```
acb90d6e4db7b6eb260f1ece4fd2c1b9b92f190c9bd2be900ad5a92b80cc23db9b6d3144
447cbb5f7af7c615d9a6b9ee0ba0e713ad2b77feba7c40a18b02e50ffbe0ecb13849c1c6
ed496f6a21300c145a1e7e0d4cb283607f6300e40ea411b6d6255967b3298765b22c6f1d
c91fa5ab
```

Along with the PK value as defined in Section 7.1 as inputs into the Verify operation should fail signature validation due to an additional message being supplied that was not signed.

C.2.3. Missing Message Signature

Using the following header

```
11223344556677889900aabbccddeeff
```

And the following messages (the two first messages defined in Section 7.2)

```
9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02

87a8bd656d49ee07b8110e1d8fd4f1dcef6fb9bc368c492d9bc8c4f98a739ac6
```

After they are mapped to the first 2 scalars in Section 7.4.1, and with the following signature (which is a signature on all the messages defined in Section 7.2)

```
b9c68aa75ed3510d2c3dd06d962106b888073b9468db2bde45c42ed32d3a04ffc14e0854
ce219b77ce845fe7b06e200f66f64cb709e83a367586a70dc080b0fe242444b7cfd08977
d74d91be64b468485774792526992181bc8b2d40a913c9bf561b2eeb0e149bfb7dc05d36
07903513
```

Along with the PK value as defined in Section 7.1 as inputs into the Verify operation should fail signature validation due to missing messages that were originally present during the signing.

C.2.4. Reordered Message Signature

Using the following header

```
11223344556677889900aabbccddeeff
```

And the following messages (re-ordering of the messages defined in Section 7.2)

```
c344136d9ab02da4dd5908bbba913ae6f58c2cc844b802a6f811f5fb075f9b80

7372e9daa5ed31e6cd5c825eac1b855e84476a1d94932aa348e07b7320912416

77fe97eb97a1ebe2e81e4e3597a3ee740a66e9ef2412472c23364568523f8b91

496694774c5604ab1b2544eababcf0f53278ff5040c1e77c811656e8220417a2

515ae153e22aae04ad16f759e07237b43022cb1ced4c176e0999c6a8ba5817cc

d183ddc6e2665aa4e2f088af9297b78c0d22b4290273db637ed33ff5cf703151

ac55fb33a75909edac8994829b250779298aa75d69324a365733f16c333fa943

96012096adda3f13dd4adbe4eea481a4c4b5717932b73b00e31807d3c5894b90

87a8bd656d49ee07b8110e1d8fd4f1dcef6fb9bc368c492d9bc8c4f98a739ac6

9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02
```

After they are mapped to the corresponding scalars in Section 7.4.1, and with the following signature

```
b9c68aa75ed3510d2c3dd06d962106b888073b9468db2bde45c42ed32d3a04ffc14e0854
ce219b77ce845fe7b06e200f66f64cb709e83a367586a70dc080b0fe242444b7cfd08977
d74d91be64b468485774792526992181bc8b2d40a913c9bf561b2eeb0e149bfb7dc05d36
07903513
```

Along with the PK value as defined in Section 7.1 as inputs into the Verify operation should fail signature validation due to messages being re-ordered from the order in which they were signed.

C.2.5. Wrong Public Key Signature

Using the following header

```
11223344556677889900aabbccddeeff
```

And the messages as defined in Section 7.2, mapped to the scalars in Section 7.4.1 and with the following signature

```
b9c68aa75ed3510d2c3dd06d962106b888073b9468db2bde45c42ed32d3a04ffc14e0854
ce219b77ce845fe7b06e200f66f64cb709e83a367586a70dc080b0fe242444b7cfd08977
d74d91be64b468485774792526992181bc8b2d40a913c9bf561b2eeb0e149bfb7dc05d36
07903513
```

Along with the PK value as defined in Section 7.1 as inputs into the Verify operation should fail signature validation due to public key used to verify is incorrect.

C.2.6. Wrong Header Signature

Using the following header

```
ffeeddccbbaa00998877665544332211
```

And the messages as defined in Section 7.2, mapped to the scalars in Section 7.4.1 and with the following signature

```
b9c68aa75ed3510d2c3dd06d962106b888073b9468db2bde45c42ed32d3a04ffc14e0854
ce219b77ce845fe7b06e200f66f64cb709e83a367586a70dc080b0fe242444b7cfd08977
d74d91be64b468485774792526992181bc8b2d40a913c9bf561b2eeb0e149bfb7dc05d36
07903513
```

Along with the PK value as defined in Section 7.1 as inputs into the Verify operation should fail signature validation due to header value being modified from what was originally signed.

C.2.7. Hash to Scalar Test Vectors

Using the following input message,

```
9872ad089e452c7b6e283dfac2a80d58e8d0ff71cc4d5e310a1debdda4a45f02
```

And the default dst defined in hash-to-scalar, i.e.,

```
4242535f424c53313233383147315f584d443a5348412d3235365f535357555f524f5f48
32535f
```

With an output count of 1, we get the following scalar, encoded with I2OSP and represented in big endian order,

```
3805512ad3be0b912c70fb460ee39085ee05eda69c9eea1be4977543c0db7af5
```

With the same input message and dst but with an output count of `10` we get the following scalars (again encoded with I2OSP and represented in big endian order),

```
6d81d02d87e95d701e4eddeeed0b79d9fd5bf9233af3b67f264b090cd2ebd98b

67aeda49b69dcc80bd85ee48fac93bf4ae46c54d53cd89b43eda279e6292d91d

27865ba85cbcfe982050f2afbd594d092e28063b740b05604d552faab5026815

453df87fc2ba78af69015fa6bb7948037e2dc76cfe842028fadc13a04146c095

515d22ca9914192aae582e65f430fc520cf174580ad5f9573254f6cf0fa7e5f9

090bf41e05dbd70b3c2bfc56743c1b9778a422366c69629bf309973128be1e9e

4eff0f64ab4ebf959c65f8430b14a1ea1a5c46314defd98a797ff61065555b09

443284df7e4c0bde9d47b3ef788a1d4880971878374d4919c0c37290c6ed6ec7

0f00ca900618d3b4dcb1cace167939abcd0f558b745a1c7db3c132cbb194e94b

4757cb9e1a38941e21adcb1fdb8f38cef984ab65bcbfd543f4b753a5f870915d
```

Appendix D. Proof Generation and Verification Algorithmic Explanation

The following section provides an explanation of how the ProofGen and ProofVerify operations work.

Let the prover be in possession of a BBS signature `(A, e, s)` on messages `msg_1, ..., msg_L` and a `domain` value (see Sign). Let `A = B * (1/(e + SK))` where `SK` the signer's secret key and,

```
B = P1 + Q_1 * s + Q_2 * domain + H_1 * msg_1 + ... + H_L * msg_L
```

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

- Hide the signature by randomizing it. To randomize the signature `(A, e, s)`, take uniformly random `r1`, `r2` in `[1, r-1]`, and calculate,

  ```
  1.  A' = A * r1,
  2.  Abar = A' * (-e) + B * r1
  3.  D = B * r1 + Q_1 * r2.
  ```

  Also set,

  ```
  4.  r3 = r1 ^ -1 mod r
  5.  s' = r2 * r3 + s mod r.
  ```

  The values `(A', Abar, D)` will be part of the proof and are used to prove possession of a BBS signature, without revealing the signature itself. Note that; `e(A', PK) = e(Abar, P2)` where `PK` the signer's public key and `P2` the base element in `G2` (used to create the signer's `PK`, see SkToPk). This also serves to bind the proof to the signer's `PK`.

- Set the following,

  ```
  1.  C1 = Abar - D
  2.  C2 = P1 + Q_2 * domain + H_i1 * msg_i1 + ... + H_iR * msg_iR
  ```

Create a non-interactive zero-knowledge proof-of-knowledge (nizk) of the values e, r2, r3, s' and msg_j1, ..., msg_jU (the undisclosed messages) so that both of the following equalities hold,

```
EQ1.  C1 = A' * (-e) - Q_1 * r2
EQ2.  C2 = Q_1 * s' - D * r3 + H_j1 * msg_j1 + ... + H_jU * msg_jU.
```

Note that the verifier will know the elements in the left side of the above equations (i.e., C1 and C2) but not in the right side (i.e., s', r3 and the undisclosed messages: msg_j1, ..., msg_jU). However, using the nizk, the prover can convince the verifier that they (the prover) know the elements that satisfy those equations, without disclosing them. Then, if both EQ1 and EQ2 hold, and e(A', PK) = e(Abar, P2), an extractor can return a valid BBS signature from the signer's SK, on the disclosed messages. The proof returned is (A', Abar, D, nizk). To validate the proof, a verifier checks that e(A', PK) = e(Abar, P2) and verifies the nizk. Validating the proof, will guarantee the authenticity and integrity of the disclosed messages, as well as ownership of the undisclosed messages and of the signature.

Appendix E. 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

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