Workgroup: CFRG
Internet-Draft:
draft-irtf-cfrg-bbs-signatures-02
Published: 11 March 2023
Intended Status: Informational
Expires: 12 September 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

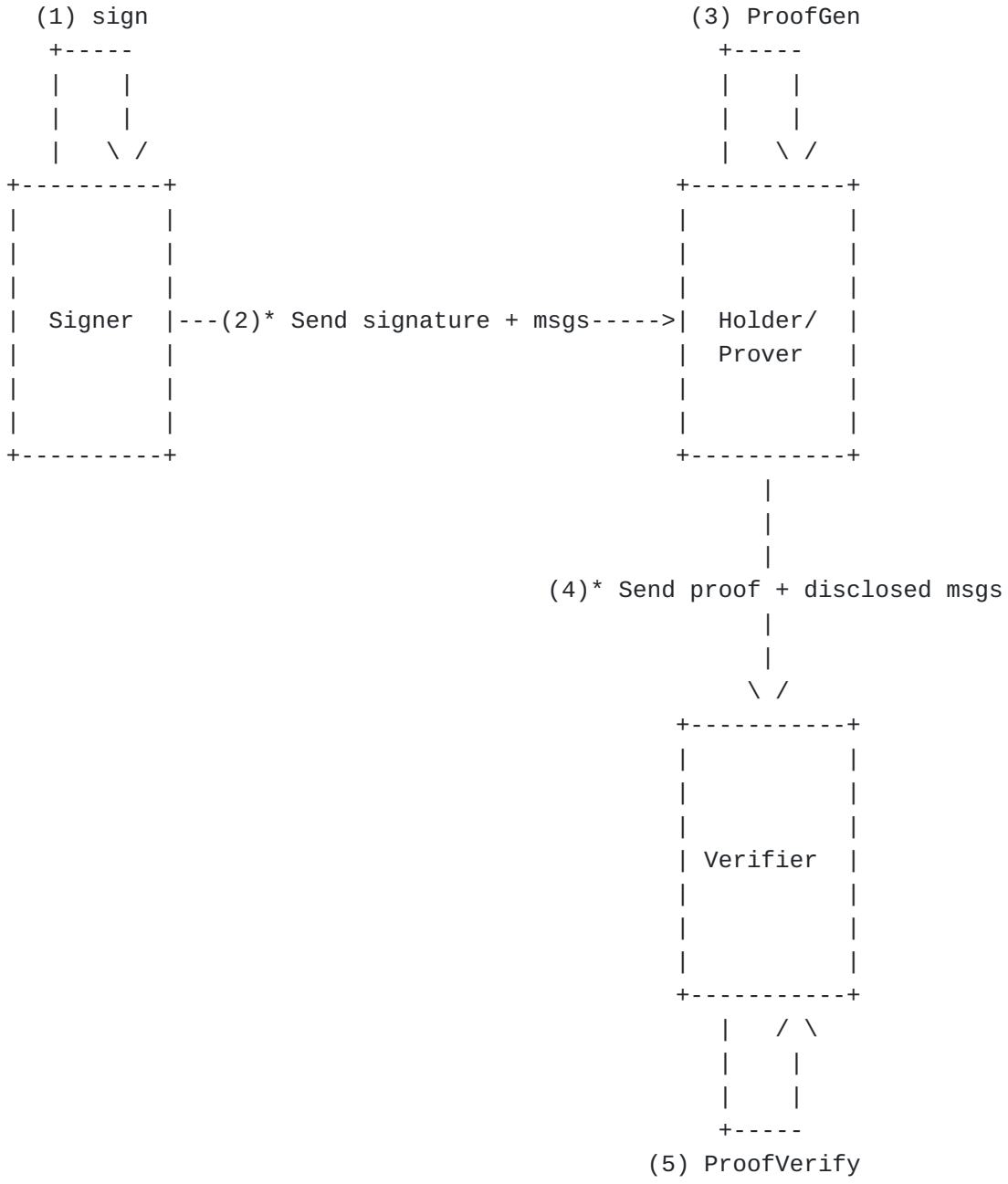

Figure 1: Basic diagram capturing the main entities involved in using the scheme

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.

scalar An integer between 0 and r-1, where r is the prime order of the selected groups, defined by each ciphersuite (see also Notation).

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 the Fiat-Shamir heuristic.

dst The domain separation tag.

I2OSP An operation that transforms a non-negative integer into an octet string, defined in Section 4 of [RFC8017]. Note, the output of this operation is in big-endian order.

OS2IP An operation that transforms a octet string into an non-negative integer, defined in Section 4 of [RFC8017]. Note, the input of this operation must be in big-endian order.

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).

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.

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

3.2. 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

Each of the core operations of the BBS signature scheme expect the inputted messages to be scalar values within a given range (specifically 1 and r-1, where r is the prime order of the G1 and G2 subgroups, defined by each ciphersuite, see Notation). There are multiple ways to transform a message from an octet string to a scalar value. This document defines the MapMessageToScalarAsHash operation, which hashes an octet string to a scalar (see MapMessageToScalarAsHash). An application can use a different MapMessageToScalar operation, but it MUST be clearly and unambiguously defined, for all parties involved. Before using the core operations, all messages MUST be mapped to their respective scalars using the same operation. The defined

MapMessageToScalarAsHash is the RECOMMENDED way of mapping octet strings to scalar values.

3.2.3. 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.).

3.2.4. Serializing to octet strings

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

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

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

Those rules will be used explicitly on every operation. See also Serialize.

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 func
- HKDF-Expand is as defined in [@!RFC5869], instantiated with hash funct
- 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 of this section make use of functions and sub-routines defined in Utility Operations. More specifically,

- *hash_to_scalar is defined in Section 4.3
- *calculate_domain and calculate_challenge are defined in Section 4.4 and Section 4.5 correspondingly.
- *serialize, signature_to_octets, octets_to_signature, proof_to_octets, octets_to_proof and octets_to_pubkey are defined in Section 4.6

The following operations also make use of the create_generators operation defined in Section 4.1, to create generator points on G1 (see Messages and Generators). Note that the values of those points depends only on a cipheruite defined seed. As a result, the output of that operation can be cached to avoid unnecessary calls to the create_generators procedure. See Section 4.1 for more details.

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 (as scalar values, see 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:

- P1, fixed point of G1, defined by the ciphersuite.
- expand_message, the expand_message operation defined by the suite
                  specified by the hash_to_curve_suite parameter.
- octet_scalar_length, non-negative integer. The length of a scalar
                       octet representation, defined by the ciphersuite.

Definitions:

- L, is the non-negative integer representing the number of messages to
     be signed.
- expand_dst, an octet string representing the domain separation tag:
              utf8(ciphersuite_id || "SIG_DET_DST_"), where
              ciphersuite_id is defined by the ciphersuite.

Outputs:

- signature, a signature encoded as an octet string.

Deserialization:

1. L = length(messages)
2. (msg_1, ..., msg_L) = messages

Procedure:

1.  (Q_1, Q_2, H_1, ..., H_L) = create_generators(L+2)
2.  domain = calculate_domain(PK, Q_1, Q_2, (H_1, ..., H_L), header)
3.  if domain is INVALID, return INVALID
4.  e_s_octs = serialize((SK, domain, msg_1, ..., msg_L))
5.  if e_s_octs is INVALID, return INVALID
6.  e_s_len = octet_scalar_length * 2
7.  e_s_expand = expand_message(e_s_octs, expand_dst, e_s_len)
8.  if e_s_expand is INVALID, return INVALID
9.  e = hash_to_scalar(e_s_expand[0..(octet_scalar_length - 1)])
```

```
10. s = hash_to_scalar(e_s_expand[octet_scalar_length..(e_s_len - 1)])
11. if e or s is INVALID, return INVALID
12. B = P1 + Q_1 * s + Q_2 * domain + H_1 * msg_1 + ... + H_L * msg_L
13. A = B * (1 / (SK + e))
14. return signature_to_octets(A, e, s)
```

Note When computing step 12 of the above procedure there is an extremely small probability (around 2^(-r)) that the condition (SK + e) = 0 mod r will be met. How implementations evaluate the inverse of the scalar value 0 may vary, with some returning an error and others returning 0 as a result. If the returned value from the inverse operation 1/(SK + e) does evaluate to 0 the value of A will equal Identity_G1 thus an invalid signature. Implementations MAY elect to check (SK + e) = 0 mod r prior to step 9, and or A != Identity_G1 after step 9 to prevent the production of invalid signatures.

3.4.2. 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:

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

Definitions:

- L, is the non-negative integer representing the number of messages to
     be signed.

Outputs:

- result, either VALID or INVALID.

Deserialization:

1. signature_result = octets_to_signature(signature)
2. if signature_result is INVALID, return INVALID
3. (A, e, s) = signature_result
4. W = octets_to_pubkey(PK)
5. if W is INVALID, return INVALID
6. L = length(messages)
7. (msg_1, ..., msg_L) = messages

Procedure:

1. (Q_1, Q_2, H_1, ..., H_L) = create_generators(L+2)
2. domain = calculate_domain(PK, Q_1, Q_2, (H_1, ..., H_L), header)
3. if domain is INVALID, return INVALID
4. B = P1 + Q_1 * s + Q_2 * domain + H_1 * msg_1 + ... + H_L * msg_L
5. if e(A, W + P2 * e) * e(B, -P2) != Identity_GT, return INVALID
6. 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).

The operation calculates multiple random scalars using the calculate_random_scalars utility operation defined in Section 4.1. See also Section 5.10 for considerations and requirements on random scalars generation.

To allow for flexibility in implementations, although ProofGen defines a specific value for expand_len, applications may use any value larger than ceil((ceil(log2(r))+k)/8) (for example, for the BLS12-381-SHAKE-256 and BLS12-381-SHA-256 ciphersuites, an implementation can elect to use a value of 64, instead of 48, as to allow for certain optimizations).

```
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), an octet string containing the presentation header. If
                 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:

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

Definitions:

- L, is the non-negative integer representing the number of messages.
- R, is the non-negative integer representing the number of disclosed
    (revealed) messages.
- U, is the non-negative integer representing the number of undisclosed
    messages, i.e., U = L - R.

Outputs:

- proof, an octet string; or INVALID.

Deserialization:

1.  signature_result = octets_to_signature(signature)
2.  if signature_result is INVALID, return INVALID
3.  (A, e, s) = signature_result
4.  L = length(messages)
5.  R = length(disclosed_indexes)
6.  U = L - R
7.  (i1, ..., iR) = disclosed_indexes
8.  (j1, ..., jU) = range(1, L) \ disclosed_indexes
9.  (msg_1, ..., msg_L) = messages
10. (msg_i1, ..., msg_iR) = (messages[i1], ..., messages[iR])
11. (msg_j1, ..., msg_jU) = (messages[j1], ..., messages[jU])
```

Procedure:

```
1.  (Q_1, Q_2, MsgGenerators) = create_generators(L+2)
2.  (H_1, ..., H_L) = MsgGenerators
3.  (H_j1, ..., H_jU) = (MsgGenerators[j1], ..., MsgGenerators[jU])

4.  domain = calculate_domain(PK, Q_1, Q_2, (H_1, ..., H_L), header)
5.  if domain is INVALID, return INVALID
6.  random_scalars = calculate_random_scalars(6+U)
7.  (r1, r2, e~, r2~, r3~, s~, m~_j1, ..., m~_jU) = random_scalars
8.  B = P1 + Q_1 * s + Q_2 * domain + H_1 * msg_1 + ... + H_L * msg_L
9.  r3 = r1 ^ -1 mod r
10. A' = A * r1
11. Abar = A' * (-e) + B * r1
12. D = B * r1 + Q_1 * r2
13. s' = r2 * r3 + s mod r
14. C1 = A' * e~ + Q_1 * r2~
15. C2 = D * (-r3~) + Q_1 * s~ + H_j1 * m~_j1 + ... + H_jU * m~_jU
16. c = calculate_challenge(A', Abar, D, C1, C2, (i1, ..., iR),
                                    (msg_i1, ..., msg_iR), domain, ph)
17. if c is INVALID, return INVALID
18. e^ = c * e + e~ mod r
19. r2^ = c * r2 + r2~ mod r
20. r3^ = c * r3 + r3~ mod r
21. s^ = c * s' + s~ mod r
22. for j in (j1, ..., jU): m^_j = c * msg_j + m~_j mod r
23. proof = (A', Abar, D, c, e^, r2^, r3^, s^, (m^_j1, ..., m^_jU))
24. 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, header, ph,
                     disclosed_messages,
                     disclosed_indexes)

Inputs:

- PK (REQUIRED), an octet string of the form outputted by the SkToPk
                 operation.
- proof (REQUIRED), an octet string of the form outputted by the
                    ProofGen operation.
- header (OPTIONAL), an optional octet string containing context and
                     application specific information. If not supplied,
                     it defaults to an empty string.
- ph (OPTIONAL), an octet string containing the presentation header. If
                 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:

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

Definitions:

- R, is the non-negative integer representing the number of disclosed
     (revealed) messages.
- U, is the non-negative integer representing the number of undisclosed
     messages.
- L, is the non-negative integer representing the number of total,
     messages i.e., L = U + R.

Outputs:

- result, either VALID or INVALID.

Deserialization:

1.  proof_result = octets_to_proof(proof)
2.  if proof_result is INVALID, return INVALID
3.  (A', Abar, D, c, e^, r2^, r3^, s^, commitments) = proof_result
4.  W = octets_to_pubkey(PK)
5.  if W is INVALID, return INVALID
6.  U  = length(commitments)
7.  R = length(disclosed_indexes)
8.  L = R + U
9.  (i1, ..., iR) = disclosed_indexes
```

```
10. (j1, ..., jU) = range(1, L) \ disclosed_indexes
11. (msg_i1, ..., msg_iR) = disclosed_messages
12. (m^_j1, ...., m^_jU) = commitments

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.  (Q_1, Q_2, MsgGenerators) = create_generators(L+2)
2.  (H_1, ..., H_L) = MsgGenerators
3.  (H_i1, ..., H_iR) = (MsgGenerators[i1], ..., MsgGenerators[iR])
4.  (H_j1, ..., H_jU) = (MsgGenerators[j1], ..., MsgGenerators[jU])

5.  domain = calculate_domain(PK, Q_1, Q_2, (H_1, ..., H_L), header)
6.  if domain is INVALID, return INVALID
7.  C1 = (Abar - D) * c + A' * e^ + Q_1 * r2^
8.  T = P1 + Q_2 * domain + H_i1 * msg_i1 + ... + H_iR * msg_iR
9.  C2 = T * c - D * r3^ + Q_1 * s^ + H_j1 * m^_j1 + ... + H_jU * m^_jU
10. cv = calculate_challenge(A', Abar, D, C1, C2, (i1, ..., iR),
                                    (msg_i1, ..., msg_iR), domain, ph)
11. if cv is INVALID, return INVALID
12. if c != cv, return INVALID
13. if A' == Identity_G1, return INVALID
14. if e(A', W) * e(Abar, -P2) != Identity_GT, return INVALID
15. return VALID
```

4. Utility Operations

4.1. Random scalars computation

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

The required length of the get_random output is defined as expand_len. Each value returned by the get_random function is reduced modulo the group order r. To avoid biased results when creating the random scalars, the output of get_random MUST be at least (ceil(log2(r))+k bytes long, where k is the targeted security level specified by the ciphersuite (see Section 5 in [I-D.irtf-cfrg-hash-to-curve] for more details). ProofGen defines expand_len = ceil((ceil(log2(r))+k)/8). For both the BLS12-381-SHAKE-256 and BLS12-381-SHA-256 ciphersuites, log2(r) = 255 and k = 128 resulting to expand_len = 48. See Section 5.10 for further security considerations and requirements around the generated randomness.

Note: The security of the proof generation algorithm (ProofGen) is highly dependant on the quality of the get_random function. Care must be taken to ensure that a cryptographically secure pseudo-random generator is chosen, and that its outputs are not leaked to an adversary. See also Section 5.10 for more details.

```
random_scalars = calculate_random_scalars(count)

Inputs:

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

Parameters:

- get_random, a pseudo random function with extendable output, returning
              uniformly distributed pseudo random bytes.
- expand_len = ceil((ceil(log2(r))+k)/8), where r and k are defined by
                                          the ciphersuite.

Outputs:

- random_scalars, a list of pseudo random scalars,

Procedure:

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

4.2. 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 (which can be arbitrarily large, depending on the application). Then, during the execution of one of the Core Operations, if K generators are needed with K <= count, the application can use the K first of the cached generators (in place of the direct call to create_generators(K)).

NOTE: If the generator points are retrieved from cache, the order in which they are retrieved MUST be the same as the order they were originally returned by the create_generators operation.

For example, an application can save 100 generator points H_1, H_2, ..., H_100 returned from create_generators(100). Then if one of the core operations needs 30 of them, the application instead of calling create_generators again, can just retrieve the 30 first generators

H_1, H_2, ..., H_30 from the cache instead, in the same order they where originally created (starting from the first one).

The values n and v MAY also be cached in order to efficiently extend an existing list of cached 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, an octet string. A seed value selected by the
                  ciphersuite.
- P1, fixed point of G1, defined by the ciphersuite.

Definitions:

- seed_dst, an octet string representing the domain separation tag:
            ciphersuite_id || "SIG_GENERATOR_SEED_" where
            ciphersuite_id is defined by the ciphersuite and
            "SIG_GENERATOR_SEED_" is an ASCII string comprised of 19
            bytes.
- generator_dst, an octet string representing the domain separation tag:
                 ciphersuite_id || "SIG_GENERATOR_DST_", where
                 ciphersuite_id is defined by the ciphersuite and
                 "SIG_GENERATOR_DST_" is an ASCII string comprised of
                 18 bytes.
- 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)
```

4.3. 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.

4.3.1. 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 ciphersuite_id || "MAP_MSG_TO_SCALAR_AS_HASH_", where ciphersuite_id is the ASCII string representing the unique ID of the ciphersuite "MAP_MSG_TO_SCALAR_AS_HASH_" is an ASCII string comprised of 26 bytes.

```
msg_scalar = MapMessageToScalarAsHash(msg, dst)

Inputs:

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

Outputs:

- msg_scalar, a scalar value.

Procedure:

1. if length(msg) > 2^64 - 1 or length(dst) > 255 return INVALID
2. msg_scalar = hash_to_scalar(msg, dst)
3. if msg_scalar is INVALID, return INVALID
4. return msg_scalar
```

4.4. 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 hash 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 ciphersuit_id || "H2S_", where ciphersuite_id is the octet string representing the unique ID of the ciphersuite and "H2S_" is an ASCII string comprised of 4 bytes.

Note It is possible that the hash_to_scalar procedure will return an error, if the underlying expand_message operation aborts. See [I-D.irtf-cfrg-hash-to-curve], Section 5.3, for more details on the cases that expand_message will abort (note that the input term len_in_bytes of expand_message in the Hash-to-Curve document equals count * expand_len in our case).

```
hashed_scalar = hash_to_scalar(msg_octets, dst)

Inputs:

- msg_octets (REQUIRED), an octet string. The message to be hashed.
- dst (OPTIONAL), an octet string representing a domain separation tag.
                  If not supplied, it defaults to the octet string given
                  by 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:

- hashed_scalar, a non-zero scalar mod r.

Procedure:

1.  counter = 0
2.  hashed_scalar = 0
3.  while hashed_scalar == 0:
4.      if counter > 255, return INVALID
5.      msg_prime = msg_octets || I2OSP(counter, 1)
6.      uniform_bytes = expand_message(msg_prime, dst, expand_len)
7.      if uniform_bytes is INVALID, return INVALID
8.      hashed_scalar = OS2IP(uniform_bytes) mod r
9.      counter = counter + 1
10. return hashed_scalar
```

4.5. Domain Calculation

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

The input to the domain value includes an octet string called the header, chosen by the signer and meant to encode any information that is required to be revealed by the prover (such as an expiration

date, or an identifier for the target audience). This is in contrast to the signed message values, which may be withheld during a proof.

When a signature is calculated, the domain value is combined with a specific generator point (Q_2, see Sign) to protect the integrity of the public parameters and the header.

This operation makes use of the serialize function, defined in Section 4.6.1.

```
domain = calculate_domain(PK, Q_1, Q_2, H_Points, header)

Inputs:

- PK (REQUIRED), an octet string, representing the public key of the
                 Signer of the form outputted by the SkToPk operation.
- (Q_1, Q_2) (REQUIRED), points of G1 (the first 2 points returned from
                         create_generators).
- H_Points (REQUIRED), array of points of G1.
- header (OPTIONAL), an octet string. If not supplied, it must default t
                     the empty octet string ("").

Parameters:

- ciphersuite_id, an octet string. The unique ID of the ciphersuite.

Outputs:

- domain, a scalar value or INVALID.

Procedure:

1.  L = length(H_Points)
2.  if length(header) > 2^64 - 1 or L > 2^64 - 1, return INVALID
3.  (H_1, ..., H_L) = H_Points
4.  dom_array = (L, Q_1, Q_2, H_1, ..., H_L)
5.  dom_octs = serialize(dom_array) || ciphersuite_id
6.  if dom_octs is INVALID, return INVALID
7.  dom_input = PK || dom_octs || I2OSP(length(header), 8) || header
8.  domain = hash_to_scalar(dom_input)
9.  if domain is INVALID, return INVALID
10. return domain
```

Note: If the header is not supplied in calculate_domain, it defaults to the empty octet string (""). This means that in the concatenation step of the above procedure (step 7), 8 bytes representing a length of 0 (i.e., 0x0000000000000000), will still need to be appended at the end, even though a header value is not provided.

4.6. Challenge Calculation

This operation calculates the challenge scalar value, used during ProofGen and ProofVerify, as part of the Fiat-Shamir heuristic, for making the proof protocol non-interactive (in a interactive sating, the challenge would be a random value supplied by the verifier).

This operation makes use of the serialize function, defined in Section 4.6.1.

```
challenge = calculate_challenge(A', Abar, D, C1, C2, i_array,
                                                  msg_array, domain, ph)
```

Inputs:

- (A', Abar, D, C1, C2) (REQUIRED), points of G1, as calculated in ProofGen.
- i_array (REQUIRED), array of non-negative integers (the indexes of the disclosed messages).
- msg_array (REQUIRED), array of scalars (the disclosed messages).
- domain (REQUIRED), a scalar.
- ph (OPTIONAL), an octet string. If not supplied, it must default to th empty octet string ("").

Outputs:

- challenge, a scalar or INVALID.

Procedure:

1. R = length(i_array)
2. if R > 2^64 - 1 or R != length(msg_array), return INVALID
3. if length(ph) > 2^64 - 1, return INVALID
4. (i1, ..., iR) = i_array
5. (msg_i1, ..., msg_iR) = msg_array
6. c_array = (A', Abar, D, C1, C2, R, i1, ..., iR, msg_i1, ..., msg_iR, domain)
7. c_octs = serialize(c_array)
8. if c_octs is INVALID, return INVALID
9. c_input = c_octs || I2OSP(length(ph), 8) || ph
10. challenge = hash_to_scalar(c_input)
11. if challenge is INVALID, return INVALID
12. return challenge

Note: Similarly to the header value in Domain Calculation, if the presentation header (ph) is not supplied in calculate_challenge, 8 bytes representing a length of 0 (i.e., 0x0000000000000000), must still be appended after the c_octs value, during the concatenation step of the above procedure (step 9).

4.7. Serialization

4.7.1. Serialize

This operation describes how to transform multiple elements of different types (i.e., elements that are not already in a octet string format) to a single octet string (see Section 3.2.3). The inputted elements can be points, scalars (see Terminology) or integers between 0 and 2^64-1. The resulting octet string will then either be used as an input to a hash function (i.e., in Sign, ProofGen etc.), or to serialize a signature or proof (see SignatureToOctets and ProofToOctets).

```
octets_result = serialize(input_array)

Inputs:

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

Parameters:

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

Outputs:

- octets_result, a scalar value or INVALID.

Procedure:

1.  let octets_result be an empty octet string.
2.  for el in input_array:
3.      if el is a point of G1: el_octs = point_to_octets_g1(el)
4.      else if el is a point of G2: el_octs = point_to_octets_g2(el)
5.      else if el is a scalar: el_octs = I2OSP(el, octet_scalar_length)
6.      else if el is an integer between 0 and 2^64 - 1:
7.          el_octs = I2OSP(el, 8)
8.      else: return INVALID
9.      octets_result = octets_result || el_octs
10. return octets_result
```

4.7.2. 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, an octet string or INVALID.

Procedure:

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

4.7.3. 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), an 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)
```

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

1. Three (3) valid points of the G1 subgroup, different from the identity point of G1 (i.e., 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 27 (see above).

Parameters:

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

Outputs:

- proof_octets, an octet string or INVALID.

Procedure:

1. (A', Abar, D, c, e^, r2^, r3^, s^, (m^_1, ..., m^_U)) = proof
2. return serialize((A', Abar, D, c, e^, r2^, r3^, s^, m^_1, ..., m^_U))
```

4.7.5. 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), an octet string of the form outputted from th
                           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)
```

4.7.6. 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, an octet string. A public key in the form outputted 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

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.

5.10. Randomness requirements

ProofGen is by its nature a randomized algorithm, requiring the generation of multiple uniformly distributed, pseudo random scalars. This makes ProofGen vulnerable to bad entropy in certain applications. As an example of such application, consider systems that need to monitor and potentially restrict outbound traffic, in order to minimize data leakage during a breach. In such cases, the attacker could manipulate couple of bits in the output of the

get_random function to create an undetected chanel out of the system. Although the applicability of such attacks is limited for most of the targeted use cases of the BBS scheme, some applications may want to take measures towards mitigating them. To that end, it is RECOMMENDED to use a deterministic RNG (like a ChaCha20 based deterministic RNG), seeded with a unique, uniformly random, single seed [DRBG]. This will limit the amount of bits the attacker can manipulate (note that some randomness is always needed).

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

6. 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, which will be represented as an ASCII encoded octet string. The REQUIRED format for this string is

"BBS_" || H2C_SUITE_ID || ADD_INFO

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

- ADD_INFO is an optional octet string indicating any additional information used to uniquely qualify the ciphersuite. When present this value MUST only contain ASCII encoded characters with codes between 0x21 and 0x7e (inclusive) and MUST end with an underscore (ASCII code: 0x5f), 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 = {{ $generatorFixtures.bls12-381-shake-256.generators.BP }}

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 "BBS_BLS12381G1_XOF:SHAKE-256_SSWU_RO_MESSAGE_GENERATOR_SEED" (an ASCII string comprised of 59 bytes) 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 = {{ $generatorFixtures.bls12-381-sha-256.generators.BP }}

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 "BBS_BLS12381G1_XMD:SHA-256_SSWU_RO_MESSAGE_GENERATOR_SEED" (an ASCII string comprised of 57 bytes) 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 in big endian order, encoded in hexadecimal format.

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

7.1. Mocked random scalars

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

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

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

The mocked_calculate_random_scalars operation will then be used in place of calculate_random_scalars during the ProofGen operation's procedure.

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

```
seeded_scalars = seeded_random_scalars(SEED, count)

Inputs:

- count (REQUIRED), non negative integer. The number of scalars to
                    return.
- SEED (REQUIRED), an octet string. The random seed from which to genera
                   the scalars.

Parameters:

- expand_message, the expand_message operation defined by the
                  ciphersuite.
- expand_len = ceil((ceil(log2(r))+k)/8), where r and k are defined by
                                          the ciphersuite.
- dst = utf8(ciphersuite_id || "MOCK_RANDOM_SCALARS_DST_"), where
      ciphersuite_id is defined by teh ciphersuite.

Outputs:

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

Preconditions:

1. if count * expand_len > 65535, return INVALID

Procedure:

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

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

7.2. 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

{{ $keyPair.ikm }}

and the following key_info value

{{ $keyPair.keyInfo }}

Outputs the following SK value

{{ $keyPair.keyPair.secretKey }}

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

{{ $keyPair.keyPair.publicKey }}

7.3. Messages

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

{{ $messages[0] }}

{{ $messages[1] }}

{{ $messages[2] }}

{{ $messages[3] }}

{{ $messages[4] }}

{{ $messages[5] }}

{{ $messages[6] }}

{{ $messages[7] }}

{{ $messages[8] }}

{{ $messages[9] }}

7.4. 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.

7.4.1. Map Messages to Scalars

The messages in Section 3.2.2 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 3.2.2 and the following default dst

{{ $MapMessageToScalarFixtures.bls12-381-shake-256.MapMessageToScalarAsH

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

{{ $MapMessageToScalarFixtures.bls12-381-shake-256.MapMessageToScalarAsH

{{ $MapMessageToScalarFixtures.bls12-381-shake-256.MapMessageToScalarAsH

{{ $MapMessageToScalarFixtures.bls12-381-shake-256.MapMessageToScalarAsH

{{ $MapMessageToScalarFixtures.bls12-381-shake-256.MapMessageToScalarAsH

{{ $MapMessageToScalarFixtures.bls12-381-shake-256.MapMessageToScalarAsH

{{ $MapMessageToScalarFixtures.bls12-381-shake-256.MapMessageToScalarAsH

{{ $MapMessageToScalarFixtures.bls12-381-shake-256.MapMessageToScalarAsH

{{ $MapMessageToScalarFixtures.bls12-381-shake-256.MapMessageToScalarAsH

{{ $MapMessageToScalarFixtures.bls12-381-shake-256.MapMessageToScalarAsH

{{ $MapMessageToScalarFixtures.bls12-381-shake-256.MapMessageToScalarAsH

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.

7.4.2. Message Generators

Following the procedure defined in Section 4.2 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).

{{ $generatorFixtures.bls12-381-shake-256.generators.Q1 }}

{{ $generatorFixtures.bls12-381-shake-256.generators.Q2 }}

{{ $generatorFixtures.bls12-381-shake-256.generators.MsgGenerators[0] }}

{{ $generatorFixtures.bls12-381-shake-256.generators.MsgGenerators[1] }}

{{ $generatorFixtures.bls12-381-shake-256.generators.MsgGenerators[2] }}

{{ $generatorFixtures.bls12-381-shake-256.generators.MsgGenerators[3] }}

{{ $generatorFixtures.bls12-381-shake-256.generators.MsgGenerators[4] }}

{{ $generatorFixtures.bls12-381-shake-256.generators.MsgGenerators[5] }}

{{ $generatorFixtures.bls12-381-shake-256.generators.MsgGenerators[6] }}

{{ $generatorFixtures.bls12-381-shake-256.generators.MsgGenerators[7] }}

{{ $generatorFixtures.bls12-381-shake-256.generators.MsgGenerators[8] }}

{{ $generatorFixtures.bls12-381-shake-256.generators.MsgGenerators[9] }}

7.4.3. Signature Fixtures

7.4.3.1. Valid Single Message Signature

Using the following header

{{ $signatureFixtures.bls12-381-shake-256.signature001.header }}

And the following message (the first message defined in Section 3.2.2)

{{ $signatureFixtures.bls12-381-shake-256.signature001.messages[0] }}

After it is mapped to the first scalar in Section 7.4.1, along with the SK value as defined in Section 7.2 as inputs into the Sign operations, yields the following output signature

{{ $signatureFixtures.bls12-381-shake-256.signature001.signature }}

7.4.3.2. Valid Multi-Message Signature

Using the following header

{{ $signatureFixtures.bls12-381-shake-256.signature004.header }}

And the messages defined in Section 3.2.2 (**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 7.2 as inputs into the Sign operations, yields the following output signature

{{ $signatureFixtures.bls12-381-shake-256.signature004.signature }}

7.4.4. Proof fixtures

For the generation of the following fixtures the mocked_calculate_random_scalars defined in Mocked Random Scalars is used, in place of the calculate_random_scalars operation, with the following seed value (hex encoding of utf8("<30 first digits of pi>"))

SEED = "332e313431353932363533353839373933323338343632363433333833323739

Given the above seed the first 10 scalars returned by the mocked_calculate_random_scalars operation will be,

{{ $MockRngFixtures.bls12-381-shake-256.mockedRng.mockedScalars[0] }}
{{ $MockRngFixtures.bls12-381-shake-256.mockedRng.mockedScalars[1] }}
{{ $MockRngFixtures.bls12-381-shake-256.mockedRng.mockedScalars[2] }}
{{ $MockRngFixtures.bls12-381-shake-256.mockedRng.mockedScalars[3] }}
{{ $MockRngFixtures.bls12-381-shake-256.mockedRng.mockedScalars[4] }}
{{ $MockRngFixtures.bls12-381-shake-256.mockedRng.mockedScalars[5] }}
{{ $MockRngFixtures.bls12-381-shake-256.mockedRng.mockedScalars[6] }}
{{ $MockRngFixtures.bls12-381-shake-256.mockedRng.mockedScalars[7] }}
{{ $MockRngFixtures.bls12-381-shake-256.mockedRng.mockedScalars[8] }}
{{ $MockRngFixtures.bls12-381-shake-256.mockedRng.mockedScalars[9] }}

7.4.4.1. Valid Single Message Proof

Using the header, message and signature used in Valid Single Message Signature to create a proof disclosing the message, with the following presentation header

{{ $proofFixtures.bls12-381-shake-256.proof001.presentationHeader }}

will result to the following proof value

{{ $proofFixtures.bls12-381-shake-256.proof001.proof }}

7.4.4.2. Valid Multi-Message, All Messages Disclosed Proof

Using the header, messages and signature used in Valid Multi Message Signature to create a proof disclosing all the messages, with the following presentation header

{{ $proofFixtures.bls12-381-shake-256.proof002.presentationHeader }}

will result to the following proof value

{{ $proofFixtures.bls12-381-shake-256.proof002.proof }}

7.4.4.3. Valid Multi-Message, Half of Messages Disclosed Proof

Using the same header, messages and signature as in Multi-Message, All Messages Disclosed Proof but this time with only every other messages disclosed (messages in index 0, 2, 4 and 6, in that order), with the following presentation header

{{ $proofFixtures.bls12-381-shake-256.proof003.presentationHeader }}

will result to the following proof value

{{ $proofFixtures.bls12-381-shake-256.proof003.proof }}

7.5. 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.

7.5.1. 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 3.2.2 and the following default dst

{{ $MapMessageToScalarFixtures.bls12-381-sha-256.MapMessageToScalarAsHas

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

```
{{ $MapMessageToScalarFixtures.bls12-381-sha-256.MapMessageToScalarAsHas

{{ $MapMessageToScalarFixtures.bls12-381-sha-256.MapMessageToScalarAsHas

{{ $MapMessageToScalarFixtures.bls12-381-sha-256.MapMessageToScalarAsHas

{{ $MapMessageToScalarFixtures.bls12-381-sha-256.MapMessageToScalarAsHas

{{ $MapMessageToScalarFixtures.bls12-381-sha-256.MapMessageToScalarAsHas

{{ $MapMessageToScalarFixtures.bls12-381-sha-256.MapMessageToScalarAsHas

{{ $MapMessageToScalarFixtures.bls12-381-sha-256.MapMessageToScalarAsHas

{{ $MapMessageToScalarFixtures.bls12-381-sha-256.MapMessageToScalarAsHas

{{ $MapMessageToScalarFixtures.bls12-381-sha-256.MapMessageToScalarAsHas

{{ $MapMessageToScalarFixtures.bls12-381-sha-256.MapMessageToScalarAsHas
```

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.

7.5.2. Message Generators

Following the procedure defined in Section 4.2 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).

{{ $generatorFixtures.bls12-381-sha-256.generators.Q1 }}

{{ $generatorFixtures.bls12-381-sha-256.generators.Q2 }}

{{ $generatorFixtures.bls12-381-sha-256.generators.MsgGenerators[0] }}

{{ $generatorFixtures.bls12-381-sha-256.generators.MsgGenerators[1] }}

{{ $generatorFixtures.bls12-381-sha-256.generators.MsgGenerators[2] }}

{{ $generatorFixtures.bls12-381-sha-256.generators.MsgGenerators[3] }}

{{ $generatorFixtures.bls12-381-sha-256.generators.MsgGenerators[4] }}

{{ $generatorFixtures.bls12-381-sha-256.generators.MsgGenerators[5] }}

{{ $generatorFixtures.bls12-381-sha-256.generators.MsgGenerators[6] }}

{{ $generatorFixtures.bls12-381-sha-256.generators.MsgGenerators[7] }}

{{ $generatorFixtures.bls12-381-sha-256.generators.MsgGenerators[8] }}

{{ $generatorFixtures.bls12-381-sha-256.generators.MsgGenerators[9] }}

7.5.3. Signature Fixtures

7.5.3.1. Valid Single Message Signature

Using the following header

{{ $signatureFixtures.bls12-381-sha-256.signature001.header }}

And the following message (the first message defined in Section 3.2.2)

{{ $signatureFixtures.bls12-381-sha-256.signature001.messages[0] }}

After it is mapped to the first scalar in Section 7.5.1, along with the SK value as defined in Section 7.2 as inputs into the Sign operations, yields the following output signature

{{ $signatureFixtures.bls12-381-shake-256.signature001.signature }}

7.5.3.2. Valid Multi-Message Signature

Using the following header

{{ $signatureFixtures.bls12-381-sha-256.signature004.header }}

And the messages defined in Section 3.2.2 (**Note** the ordering of the messages MUST be preserved), after they are mapped to the scalars in

Section 7.5.1, along with the SK value as defined in Section 7.2 as inputs into the Sign operations, yields the following output signature

{{ $signatureFixtures.bls12-381-sha-256.signature004.signature }}

7.5.4. Proof fixtures

Similarly to the proof fixtures for the BLS12381-SHA-256 ciphersuite, the generation of the following fixtures uses the mocked_calculate_random_scalars defined in Mocked Random Scalars, in place of the calculate_random_scalars operation, with the following seed value (hex encoding of utf8("<30 first digits of pi>"))

SEED = "332e313431353932363533353839373933323338343632363433333833323739

Given the above seed the first 10 scalars returned by the mocked_calculate_random_scalars operation will be,

{{ $MockRngFixtures.bls12-381-sha-256.mockedRng.mockedScalars[0] }}
{{ $MockRngFixtures.bls12-381-sha-256.mockedRng.mockedScalars[1] }}
{{ $MockRngFixtures.bls12-381-sha-256.mockedRng.mockedScalars[2] }}
{{ $MockRngFixtures.bls12-381-sha-256.mockedRng.mockedScalars[3] }}
{{ $MockRngFixtures.bls12-381-sha-256.mockedRng.mockedScalars[4] }}
{{ $MockRngFixtures.bls12-381-sha-256.mockedRng.mockedScalars[5] }}
{{ $MockRngFixtures.bls12-381-sha-256.mockedRng.mockedScalars[6] }}
{{ $MockRngFixtures.bls12-381-sha-256.mockedRng.mockedScalars[7] }}
{{ $MockRngFixtures.bls12-381-sha-256.mockedRng.mockedScalars[8] }}
{{ $MockRngFixtures.bls12-381-sha-256.mockedRng.mockedScalars[9] }}

7.5.4.1. Valid Single Message Proof

Using the header, message and signature used in Valid Single Message Signature to create a proof disclosing the message, with the following presentation header

{{ $proofFixtures.bls12-381-sha-256.proof001.presentationHeader }}

will result to the following proof value

{{ $proofFixtures.bls12-381-sha-256.proof001.proof }}

7.5.4.2. Valid Multi-Message, All Messages Disclosed Proof

Using the header, messages and signature used in Valid Multi Message Signature to create a proof disclosing all the messages, with the following presentation header

{{ $proofFixtures.bls12-381-shake-256.proof002.presentationHeader }}

will result to the following proof value

{{ $proofFixtures.bls12-381-shake-256.proof002.proof }}

7.5.4.3. Valid Multi-Message, Half of Messages Disclosed Proof

Using the same header, messages and signature as in Multi-Message, All Messages Disclosed Proof but this time with only every other messages disclosed (messages in index 0, 2, 4 and 6, in that order), with the following presentation header

{{ $proofFixtures.bls12-381-sha-256.proof003.presentationHeader }}

will result to the following proof value

{{ $proofFixtures.bls12-381-sha-256.proof003.proof }}

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

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

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

datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-16>.

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

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

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

[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>.

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

[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/sec12-final228.pdf>.

[I-D.irtf-cfrg-bls-signature] Boneh, D., Gorbunov, S., Wahby, R. S., Wee, H., Wood, C. A., and Z. Zhang, "BLS Signatures", Work in Progress, Internet-Draft, draft-irtf-cfrg-bls-signature-05, 16 June 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-05>.

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

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

The following defines a hash_to_curve suite [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

{{ $signatureFixtures.bls12-381-shake-256.signature002.header }}

And the following message (the first message defined in Section 3.2.2)

{{ $signatureFixtures.bls12-381-shake-256.signature002.messages[0] }}

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

{{ $signatureFixtures.bls12-381-shake-256.signature002.signature }}

Along with the PK value as defined in Section 7.2 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

{{ $signatureFixtures.bls12-381-shake-256.signature003.header }}

And the following messages (the two first messages defined in Section 3.2.2)

{{ $signatureFixtures.bls12-381-shake-256.signature003.messages[0] }}

{{ $signatureFixtures.bls12-381-shake-256.signature003.messages[1] }}

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)

{{ $signatureFixtures.bls12-381-shake-256.signature003.signature }}

Along with the PK value as defined in Section 7.2 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

{{ $signatureFixtures.bls12-381-shake-256.signature005.header }}

And the following messages (the two first messages defined in Section 3.2.2)

{{ $signatureFixtures.bls12-381-shake-256.signature005.messages[0] }}

{{ $signatureFixtures.bls12-381-shake-256.signature005.messages[1] }}

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 3.2.2)

{{ $signatureFixtures.bls12-381-shake-256.signature005.signature }}

Along with the PK value as defined in Section 7.2 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

{{ $signatureFixtures.bls12-381-shake-256.signature006.header }}

And the following messages (re-ordering of the messages defined in Section 3.2.2)

{{ $signatureFixtures.bls12-381-shake-256.signature006.messages[0] }}

{{ $signatureFixtures.bls12-381-shake-256.signature006.messages[1] }}

{{ $signatureFixtures.bls12-381-shake-256.signature006.messages[2] }}

{{ $signatureFixtures.bls12-381-shake-256.signature006.messages[3] }}

{{ $signatureFixtures.bls12-381-shake-256.signature006.messages[4] }}

{{ $signatureFixtures.bls12-381-shake-256.signature006.messages[5] }}

{{ $signatureFixtures.bls12-381-shake-256.signature006.messages[6] }}

{{ $signatureFixtures.bls12-381-shake-256.signature006.messages[7] }}

{{ $signatureFixtures.bls12-381-shake-256.signature006.messages[8] }}

{{ $signatureFixtures.bls12-381-shake-256.signature006.messages[9] }}

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

{{ $signatureFixtures.bls12-381-shake-256.signature006.signature }}

Along with the PK value as defined in Section 7.2 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

{{ $signatureFixtures.bls12-381-shake-256.signature007.header }}

And the messages as defined in Section 3.2.2, mapped to the scalars in Section 7.4.1 and with the following signature

{{ $signatureFixtures.bls12-381-shake-256.signature007.signature }}

Along with the PK value as defined in Section 7.2 as inputs into the Verify operation should fail signature validation due to public key used to verify is in-correct

C.1.6. Wrong Header Signature

Using the following header

{{ $signatureFixtures.bls12-381-shake-256.signature008.header }}

And the messages as defined in Section 3.2.2, mapped to the scalars in Section 7.4.1 and with the following signature

{{ $signatureFixtures.bls12-381-shake-256.signature008.signature }}

Along with the PK value as defined in Section 7.2 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,

{{ $H2sFixture.bls12-381-shake-256.h2s.message }}

And the default dst defined in hash_to_scalar, i.e.,

{{ $H2sFixture.bls12-381-shake-256.h2s.dst }}

We get the following scalar, encoded with I2OSP and represented in big endian order,

{{ $H2sFixture.bls12-381-shake-256.h2s.scalar }}

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

C.2.1. Modified Message Signature

Using the following header

{{ $signatureFixtures.bls12-381-sha-256.signature002.header }}

And the following message (the first message defined in Section 3.2.2)

{{ $signatureFixtures.bls12-381-sha-256.signature002.messages[0] }}

After is maped to the first scalar in Section 7.5.1, and with the following signature

{{ $signatureFixtures.bls12-381-sha-256.signature002.signature }}

Along with the PK value as defined in Section 7.2 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

{{ $signatureFixtures.bls12-381-sha-256.signature003.header }}

And the following messages (the two first messages defined in Section 3.2.2)

{{ $signatureFixtures.bls12-381-sha-256.signature003.messages[0] }}

{{ $signatureFixtures.bls12-381-sha-256.signature003.messages[1] }}

After they are mapped to the first 2 scalars in Section 7.5.1, and with the following signature (which is a signature to only the first of the above two messages)

{{ $signatureFixtures.bls12-381-sha-256.signature003.signature }}

Along with the PK value as defined in Section 7.2 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

{{ $signatureFixtures.bls12-381-sha-256.signature005.header }}

And the following messages (the two first messages defined in Section 3.2.2)

{{ $signatureFixtures.bls12-381-sha-256.signature005.messages[0] }}

{{ $signatureFixtures.bls12-381-sha-256.signature005.messages[1] }}

After they are mapped to the first 2 scalars in Section 7.5.1, and with the following signature (which is a signature on all the messages defined in Section 3.2.2)

{{ $signatureFixtures.bls12-381-sha-256.signature005.signature }}

Along with the PK value as defined in Section 7.2 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

{{ $signatureFixtures.bls12-381-sha-256.signature006.header }}

And the following messages (re-ordering of the messages defined in Section 3.2.2)

{{ $signatureFixtures.bls12-381-sha-256.signature006.messages[0] }}

{{ $signatureFixtures.bls12-381-sha-256.signature006.messages[1] }}

{{ $signatureFixtures.bls12-381-sha-256.signature006.messages[2] }}

{{ $signatureFixtures.bls12-381-sha-256.signature006.messages[3] }}

{{ $signatureFixtures.bls12-381-sha-256.signature006.messages[4] }}

{{ $signatureFixtures.bls12-381-sha-256.signature006.messages[5] }}

{{ $signatureFixtures.bls12-381-sha-256.signature006.messages[6] }}

{{ $signatureFixtures.bls12-381-sha-256.signature006.messages[7] }}

{{ $signatureFixtures.bls12-381-sha-256.signature006.messages[8] }}

{{ $signatureFixtures.bls12-381-sha-256.signature006.messages[9] }}

After they are mapped to the corresponding scalars in Section 7.5.1, and with the following signature

{{ $signatureFixtures.bls12-381-sha-256.signature006.signature }}

Along with the PK value as defined in Section 7.2 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

{{ $signatureFixtures.bls12-381-sha-256.signature007.header }}

And the messages as defined in Section 3.2.2, mapped to the scalars in Section 7.5.1 and with the following signature

{{ $signatureFixtures.bls12-381-sha-256.signature007.signature }}

Along with the PK value as defined in Section 7.2 as inputs into the Verify operation should fail signature validation due to public key used to verify is in-correct.

C.2.6. Wrong Header Signature

Using the following header

{{ $signatureFixtures.bls12-381-sha-256.signature008.header }}

And the messages as defined in Section 3.2.2, mapped to the scalars in Section 7.5.1 and with the following signature

{{ $signatureFixtures.bls12-381-sha-256.signature008.signature }}

Along with the PK value as defined in Section 7.2 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,

{{ $H2sFixture.bls12-381-sha-256.h2s.message }}

And the default dst defined in hash_to_scalar, i.e.,

{{ $H2sFixture.bls12-381-sha-256.h2s.dst }}

We get the following scalar, encoded with I2OSP and represented in big endian order,

{{ $H2sFixture.bls12-381-sha-256.h2s.scalar }}

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

-02

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

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