Workgroup: CFRG
Internet-Draft:
draft-irtf-cfrg-bls-signature-05
Published: 16 June 2022
Intended Status: Informational
Expires: 18 December 2022
Authors: D. Boneh
Stanford University
S. Gorbunov
University of Waterloo
R. Wahby
Carnegie Mellon University
H. Wee
NTT Research and ENS, Paris
C. Wood
Cloudflare, Inc.
Z. Zhang
Algorand

BLS Signatures

Abstract

BLS is a digital signature scheme with aggregation properties. Given set of signatures (signature_1, ..., signature_n) anyone can produce an aggregated signature. Aggregation can also be done on secret keys and public keys. Furthermore, the BLS signature scheme is deterministic, non-malleable, and efficient. Its simplicity and cryptographic properties allows it to be useful in a variety of use-cases, specifically when minimal storage space or bandwidth are required.

Status of This Memo

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

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

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 18 December 2022.

Table of Contents

1. Introduction

A signature scheme is a fundamental cryptographic primitive that is used to protect authenticity and integrity of communication. Only the holder of a secret key can sign messages, but anyone can verify the signature using the associated public key.

Signature schemes are used in point-to-point secure communication protocols, PKI, remote connections, etc. Designing efficient and secure digital signature is very important for these applications.

This document describes the BLS signature scheme. The scheme enjoys a variety of important efficiency properties:

1. The public key and the signatures are encoded as single group elements.

2. Verification requires 2 pairing operations.

3. A collection of signatures (signature_1, ..., signature_n) can be aggregated into a single signature. Moreover, the aggregate signature can be verified using only n+1 pairings (as opposed to 2n pairings, when verifying n signatures separately).

Given the above properties, the scheme enables many interesting applications. The immediate applications include

*Authentication and integrity for Public Key Infrastructure (PKI) and blockchains.

-The usage is similar to classical digital signatures, such as ECDSA.

*Aggregating signature chains for PKI and Secure Border Gateway Protocol (SBGP).

  -Concretely, in a PKI signature chain of depth n, we have n signatures by n certificate authorities on n distinct certificates. Similarly, in SBGP, each router receives a list of n signatures attesting to a path of length n in the network. In both settings, using the BLS signature scheme would allow us to aggregate the n signatures into a single signature.

*consensus protocols for blockchains.

  -There, BLS signatures are used for authenticating transactions as well as votes during the consensus protocol, and the use of aggregation significantly reduces the bandwidth and storage requirements.

1.1. Comparison with ECDSA

The following comparison assumes BLS signatures with curve BLS12-381, targeting 126 bits of security [GMT19].

For 128 bits security, ECDSA takes 37 and 79 micro-seconds to sign and verify a signature on a typical laptop. In comparison, for a similar level of security, BLS takes 370 and 2700 micro-seconds to sign and verify a signature.

In terms of sizes, ECDSA uses 32 bytes for public keys and 64 bytes for signatures; while BLS uses 96 bytes for public keys, and 48 bytes for signatures. Alternatively, BLS can also be instantiated with 48 bytes of public keys and 96 bytes of signatures. BLS also allows for signature aggregation. In other words, a single signature is sufficient to authenticate multiple messages and public keys.

1.2. Organization of this document

This document is organized as follows:

*The remainder of this section defines terminology and the high-level API.

*Section 2 defines primitive operations used in the BLS signature scheme. These operations MUST NOT be used alone.

*Section 3 defines three BLS Signature schemes giving slightly different security and performance properties.

*Section 4 defines the format for a ciphersuites and gives recommended ciphersuites.

*The appendices give test vectors, etc.

1.3. Terminology and definitions

The following terminology is used through this document:

*SK: The secret key for the signature scheme.

*PK: The public key for the signature scheme.

*message: The input to be signed by the signature scheme.

*signature: The digital signature output.

*aggregation: Given a list of signatures for a list of messages and public keys, an aggregation algorithm generates one signature that authenticates the same list of messages and public keys.

*rogue key attack: An attack in which a specially crafted public key (the "rogue" key) is used to forge an aggregated signature. Section 3 specifies methods for securing against rogue key attacks.

The following notation and primitives are used:

*a || b denotes the concatenation of octet strings a and b.

*A pairing-friendly elliptic curve defines the following primitives (see [I-D.irtf-cfrg-pairing-friendly-curves] for detailed discussion):

  -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.

  -P1, P2: distinguished points that generate G1 and G2, respectively.

  -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.

*For the above pairing-friendly curve, this document writes 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.

* For each of E1 and E2 defined by the above pairing-friendly curve, we assume that the pairing-friendly elliptic curve definition provides several primitives, described below.

  Note that these primitives are named generically. When referring to one of these primitives for a specific group, this document appends the name of the group, e.g., point_to_octets_E1, subgroup_check_E2, etc.

  - point_to_octets(P) -> ostr: returns the canonical representation of the point P as an octet string. This operation is also known as serialization.

  - octets_to_point(ostr) -> P: returns the point P corresponding to the canonical representation ostr, or INVALID if ostr is not a valid output of point_to_octets. 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].

* I2OSP and OS2IP are the functions defined in [RFC8017], Section 4.

* hash_to_point(ostr) -> P: a cryptographic hash function that takes as input an arbitrary octet string and returns a point on an elliptic curve. Functions of this kind are defined in [I-D.irtf-cfrg-hash-to-curve]. Each of the ciphersuites in Section 4 specifies the hash_to_point algorithm to be used.

1.4. API

The BLS signature scheme defines the following API:

* KeyGen(IKM) -> SK: a key generation algorithm that takes as input an octet string comprising secret keying material, and outputs a secret key SK.

* SkToPk(SK) -> PK: an algorithm that takes as input a secret key and outputs the corresponding public key.

* Sign(SK, message) -> signature: a signing algorithm that generates a deterministic signature given a secret key SK and a message.

*Verify(PK, message, signature) -> VALID or INVALID: a verification algorithm that outputs VALID if signature is a valid signature of message under public key PK, and INVALID otherwise.

*Aggregate((signature_1, ..., signature_n)) -> signature: an aggregation algorithm that aggregates a collection of signatures into a single signature.

*AggregateVerify((PK_1, ..., PK_n), (message_1, ..., message_n), signature) -> VALID or INVALID: an aggregate verification algorithm that outputs VALID if signature is a valid aggregated signature for a collection of public keys and messages, and outputs INVALID otherwise.

1.5. Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Core operations

This section defines core operations used by the schemes defined in Section 3. These operations MUST NOT be used except as described in that section.

2.1. Variants

Each core operation has two variants that trade off signature and public key size:

1. Minimal-signature-size: signatures are points in G1, public keys are points in G2. (Recall from Section 1.3 that E1 has a more compact representation than E2.)

2. Minimal-pubkey-size: public keys are points in G1, signatures are points in G2.

   Implementations using signature aggregation SHOULD use this approach, since the size of (PK_1, ..., PK_n, signature) is dominated by the public keys even for small n.

2.2. Parameters

The core operations in this section depend on several parameters:

*A signature variant, either minimal-signature-size or minimal-pubkey-size. These are defined in Section 2.1.

*A pairing-friendly elliptic curve, plus associated functionality given in Section 1.3.

*H, a hash function that MUST be a secure cryptographic hash function, e.g., SHA-256 [FIPS180-4]. For security, H MUST output at least ceil(log2(r)) bits, where r is the order of the subgroups G1 and G2 defined by the pairing-friendly elliptic curve.

*hash_to_point, a function whose interface is described in Section 1.3. When the signature variant is minimal-signature-size, this function MUST output a point in G1. When the signature variant is minimal-pubkey size, this function MUST output a point in G2. For security, this function MUST be either a random oracle encoding or a nonuniform encoding, as defined in [I-D.irtf-cfrg-hash-to-curve].

In addition, the following primitives are determined by the above parameters:

*P, an elliptic curve point. When the signature variant is minimal-signature-size, P is the distinguished point P2 that generates the group G2 (see Section 1.3). When the signature variant is minimal-pubkey-size, P is the distinguished point P1 that generates the group G1.

*r, the order of the subgroups G1 and G2 defined by the pairing-friendly curve.

*pairing, a function that invokes the function e of Section 1.3, with argument order depending on signature variant:

-For minimal-signature-size:

pairing(U, V) := e(U, V)

-For minimal-pubkey-size:

pairing(U, V) := e(V, U)

*point_to_pubkey and point_to_signature, functions that invoke the appropriate serialization routine (Section 1.3) depending on signature variant:

-For minimal-signature-size:

point_to_pubkey(P) := point_to_octets_E2(P)

point_to_signature(P) := point_to_octets_E1(P)

-For minimal-pubkey-size:

point_to_pubkey(P) := point_to_octets_E1(P)

point_to_signature(P) := point_to_octets_E2(P)

*pubkey_to_point and signature_to_point, functions that invoke the appropriate deserialization routine (Section 1.3) depending on signature variant:

-For minimal-signature-size:

pubkey_to_point(ostr) := octets_to_point_E2(ostr)

signature_to_point(ostr) := octets_to_point_E1(ostr)

-For minimal-pubkey-size:

pubkey_to_point(ostr) := octets_to_point_E1(ostr)

signature_to_point(ostr) := octets_to_point_E2(ostr)

*pubkey_subgroup_check and signature_subgroup_check, functions that invoke the appropriate subgroup check routine (Section 1.3) depending on signature variant:

-For minimal-signature-size:

pubkey_subgroup_check(P) := subgroup_check_E2(P)

signature_subgroup_check(P) := subgroup_check_E1(P)

-For minimal-pubkey-size:

pubkey_subgroup_check(P) := subgroup_check_E1(P)

signature_subgroup_check(P) := subgroup_check_E2(P)

2.3. KeyGen

The KeyGen procedure described in this section generates a secret key SK deterministically from a secret octet string IKM. SK is guaranteed to be nonzero, as required by KeyValidate (Section 2.5).

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

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.

KeyGen takes two parameters. The first parameter, salt, is required; see below for further discussion of this value. The second parameter, key_info, is optional; it MAY be used to derive multiple independent keys from the same IKM. By default, key_info is the empty string.

```
SK = KeyGen(IKM)

Inputs:
- IKM, a secret octet string. See requirements above.

Outputs:
- SK, a uniformly random integer such that 1 <= SK < r.

Parameters:
- salt, a required octet string.
- key_info, an optional octet string.
  If key_info is not supplied, it defaults to the empty string.

Definitions:
- HKDF-Extract is as defined in RFC5869, instantiated with hash H.
- HKDF-Expand is as defined in RFC5869, instantiated with hash H.
- I2OSP and OS2IP are as defined in RFC8017, Section 4.
- L is the integer given by ceil((3 * ceil(log2(r))) / 16).

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

KeyGen is the RECOMMENDED way of generating secret keys, but its use is not required for compatibility, and implementations MAY use a different KeyGen procedure. For security, such an alternative KeyGen procedure MUST output SK that is statistically close to uniformly random in the range 1 <= SK < r.

For compatibility with prior versions of this document, implementations SHOULD allow applications to choose the salt value. Setting salt to the value H("BLS-SIG-KEYGEN-SALT-") (i.e., the hash of an ASCII string comprising 20 octets) results in a KeyGen algorithm that is compatible with version 4 of this document. Setting salt to the value "BLS-SIG-KEYGEN-SALT-" (i.e., an ASCII string comprising 20 octets) results in a KeyGen algorithm that is compatible with versions of this document prior to number 4. See Section 5.1 for more information on choosing a salt value.

2.4. SkToPk

The SkToPk algorithm takes a secret key SK and outputs the corresponding public key PK. Section 2.3 discusses requirements for SK.

```
PK = SkToPk(SK)

Inputs:
- SK, a secret integer such that 1 <= SK < r.

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

Procedure:
1. xP = SK * P
2. PK = point_to_pubkey(xP)
3. return PK
```

2.5. KeyValidate

The KeyValidate algorithm ensures that a public key is valid. In particular, it ensures that a public key represents a valid, non-identity point that is in the correct subgroup. See Section 5.2 for further discussion.

As an optimization, implementations MAY cache the result of KeyValidate in order to avoid unnecessarily repeating validation for known keys.

```
result = KeyValidate(PK)

Inputs:
- PK, a public key in the format output by SkToPk.

Outputs:
- result, either VALID or INVALID

Procedure:
1. xP = pubkey_to_point(PK)
2. If xP is INVALID, return INVALID
3. If xP is the identity element, return INVALID
4. If pubkey_subgroup_check(xP) is INVALID, return INVALID
5. return VALID
```

2.6. CoreSign

The CoreSign algorithm computes a signature from SK, a secret key, and message, an octet string.

```
signature = CoreSign(SK, message)

Inputs:
- SK, a secret key in the format output by KeyGen.
- message, an octet string.

Outputs:
- signature, an octet string.

Procedure:
1. Q = hash_to_point(message)
2. R = SK * Q
3. signature = point_to_signature(R)
4. return signature
```

2.7. CoreVerify

The CoreVerify algorithm checks that a signature is valid for the octet string message under the public key PK.

```
result = CoreVerify(PK, message, signature)

Inputs:
- PK, a public key in the format output by SkToPk.
- message, an octet string.
- signature, an octet string in the format output by CoreSign.

Outputs:
- result, either VALID or INVALID.

Procedure:
1. R = signature_to_point(signature)
2. If R is INVALID, return INVALID
3. If signature_subgroup_check(R) is INVALID, return INVALID
4. If KeyValidate(PK) is INVALID, return INVALID
5. xP = pubkey_to_point(PK)
6. Q = hash_to_point(message)
7. C1 = pairing(Q, xP)
8. C2 = pairing(R, P)
9. If C1 == C2, return VALID, else return INVALID
```

2.8. Aggregate

The Aggregate algorithm aggregates multiple signatures into one.

```
signature = Aggregate((signature_1, ..., signature_n))

Inputs:
- signature_1, ..., signature_n, octet strings output by
  either CoreSign or Aggregate.

Outputs:
- signature, an octet string encoding a aggregated signature
  that combines all inputs; or INVALID.

Precondition: n >= 1, otherwise return INVALID.

Procedure:
1. aggregate = signature_to_point(signature_1)
2. If aggregate is INVALID, return INVALID
3. for i in 2, ..., n:
4.     next = signature_to_point(signature_i)
5.     If next is INVALID, return INVALID
6.     aggregate = aggregate + next
7. signature = point_to_signature(aggregate)
8. return signature
```

2.9. CoreAggregateVerify

The CoreAggregateVerify algorithm checks an aggregated signature over several (PK, message) pairs.

```
result = CoreAggregateVerify((PK_1, ..., PK_n),
                             (message_1, ... message_n),
                             signature)

Inputs:
- PK_1, ..., PK_n, public keys in the format output by SkToPk.
- message_1, ..., message_n, octet strings.
- signature, an octet string output by Aggregate.

Outputs:
- result, either VALID or INVALID.

Precondition: n >= 1, otherwise return INVALID.

Procedure:
1.  R = signature_to_point(signature)
2.  If R is INVALID, return INVALID
3.  If signature_subgroup_check(R) is INVALID, return INVALID
4.  C1 = 1 (the identity element in GT)
5.  for i in 1, ..., n:
6.      If KeyValidate(PK_i) is INVALID, return INVALID
7.      xP = pubkey_to_point(PK_i)
8.      Q = hash_to_point(message_i)
9.      C1 = C1 * pairing(Q, xP)
10. C2 = pairing(R, P)
11. If C1 == C2, return VALID, else return INVALID
```

3. BLS Signatures

This section defines three signature schemes: basic, message augmentation, and proof of possession. These schemes differ in the ways that they defend against rogue key attacks (Section 1.3).

All of the schemes in this section are built on a set of core operations defined in Section 2. Thus, defining a scheme requires fixing a set of parameters as defined in Section 2.2.

All three schemes expose the KeyGen, SkToPk, and Aggregate operations that are defined in Section 2. The sections below define the other API functions (Section 1.4) for each scheme.

3.1. Basic scheme

In a basic scheme, rogue key attacks are handled by requiring all messages signed by an aggregate signature to be distinct. This requirement is enforced in the definition of AggregateVerify.

The Sign and Verify functions are identical to CoreSign and CoreVerify (Section 2), respectively. AggregateVerify is defined below.

3.1.1. AggregateVerify

This function first ensures that all messages are distinct, and then invokes CoreAggregateVerify.

```
result = AggregateVerify((PK_1, ..., PK_n),
                         (message_1, ..., message_n),
                         signature)

Inputs:
- PK_1, ..., PK_n, public keys in the format output by SkToPk.
- message_1, ..., message_n, octet strings.
- signature, an octet string output by Aggregate.

Outputs:
- result, either VALID or INVALID.

Precondition: n >= 1, otherwise return INVALID.

Procedure:
1. If any two input messages are equal, return INVALID.
2. return CoreAggregateVerify((PK_1, ..., PK_n),
                              (message_1, ..., message_n),
                              signature)
```

3.2. Message augmentation

In a message augmentation scheme, signatures are generated over the concatenation of the public key and the message, ensuring that messages signed by different public keys are distinct.

3.2.1. Sign

To match the API for Sign defined in Section 1.4, this function recomputes the public key corresponding to the input SK. Implementations MAY instead implement an interface that takes the public key as an input.

Note that the point P and the point_to_pubkey function are defined in Section 2.2.

```
signature = Sign(SK, message)

Inputs:
- SK, a secret key in the format output by KeyGen.
- message, an octet string.

Outputs:
- signature, an octet string.

Procedure:
1. PK = SkToPk(SK)
2. return CoreSign(SK, PK || message)
```

3.2.2. Verify

```
result = Verify(PK, message, signature)

Inputs:
- PK, a public key in the format output by SkToPk.
- message, an octet string.
- signature, an octet string in the format output by CoreSign.

Outputs:
- result, either VALID or INVALID.

Procedure:
1. return CoreVerify(PK, PK || message, signature)
```

3.2.3. AggregateVerify

```
result = AggregateVerify((PK_1, ..., PK_n),
                         (message_1, ..., message_n),
                         signature)

Inputs:
- PK_1, ..., PK_n, public keys in the format output by SkToPk.
- message_1, ..., message_n, octet strings.
- signature, an octet string output by Aggregate.

Outputs:
- result, either VALID or INVALID.

Precondition: n >= 1, otherwise return INVALID.

Procedure:
1. for i in 1, ..., n:
2.     mprime_i = PK_i || message_i
3. return CoreAggregateVerify((PK_1, ..., PK_n),
                              (mprime_1, ..., mprime_n),
                              signature)
```

3.3. Proof of possession

A proof of possession scheme uses a separate public key validation step, called a proof of possession, to defend against rogue key attacks. This enables an optimization to aggregate signature verification for the case that all signatures are on the same message.

The Sign, Verify, and AggregateVerify functions are identical to CoreSign, CoreVerify, and CoreAggregateVerify (Section 2), respectively. In addition, a proof of possession scheme defines three functions beyond the standard API (Section 1.4):

- PopProve(SK) -> proof: an algorithm that generates a proof of possession for the public key corresponding to secret key SK.

- PopVerify(PK, proof) -> VALID or INVALID: an algorithm that outputs VALID if proof is valid for PK, and INVALID otherwise.

- FastAggregateVerify((PK_1, ..., PK_n), message, signature) -> VALID or INVALID: a verification algorithm for the aggregate of multiple signatures on the same message. This function is faster than AggregateVerify.

All public keys used by Verify, AggregateVerify, and FastAggregateVerify MUST be accompanied by a proof of possession, and the result of evaluating PopVerify on each public key and its proof MUST be VALID.

3.3.1. Parameters

In addition to the parameters required to instantiate the core operations (Section 2.2), a proof of possession scheme requires one further parameter:

- hash_pubkey_to_point(PK) -> P: a cryptographic hash function that takes as input a public key and outputs a point in the same subgroup as the hash_to_point algorithm used to instantiate the core operations.

  For security, this function MUST be domain separated from the hash_to_point function. In addition, this function MUST be either a random oracle encoding or a nonuniform encoding, as defined in [I-D.irtf-cfrg-hash-to-curve].

  The RECOMMENDED way of instantiating hash_pubkey_to_point is to use the same hash-to-curve function as hash_to_point, with a different domain separation tag (see [I-D.irtf-cfrg-hash-to-curve], Section 3.1). Section 4.1 discusses the RECOMMENDED way to construct the domain separation tag.

3.3.2. PopProve

This function recomputes the public key coresponding to the input SK. Implementations MAY instead implement an interface that takes the public key as input.

Note that the point P and the point_to_pubkey and point_to_signature functions are defined in Section 2.2. The hash_pubkey_to_point function is defined in Section 3.3.1.

```
proof = PopProve(SK)

Inputs:
- SK, a secret key in the format output by KeyGen.

Outputs:
- proof, an octet string.

Procedure:
1. PK = SkToPk(SK)
2. Q = hash_pubkey_to_point(PK)
3. R = SK * Q
4. proof = point_to_signature(R)
5. return proof
```

3.3.3. PopVerify

PopVerify uses several functions defined in Section 2. The hash_pubkey_to_point function is defined in Section 3.3.1.

As an optimization, implementations MAY cache the result of PopVerify in order to avoid unnecessarily repeating validation for known keys.

```
result = PopVerify(PK, proof)

Inputs:
- PK, a public key in the format output by SkToPk.
- proof, an octet string in the format output by PopProve.

Outputs:
- result, either VALID or INVALID

Procedure:
1. R = signature_to_point(proof)
2. If R is INVALID, return INVALID
3. If signature_subgroup_check(R) is INVALID, return INVALID
4. If KeyValidate(PK) is INVALID, return INVALID
5. xP = pubkey_to_point(PK)
6. Q = hash_pubkey_to_point(PK)
7. C1 = pairing(Q, xP)
8. C2 = pairing(R, P)
9. If C1 == C2, return VALID, else return INVALID
```

3.3.4. FastAggregateVerify

FastAggregateVerify uses several functions defined in Section 2.

All public keys passed as arguments to this algorithm MUST have a corresponding proof of possession, and the result of evaluating PopVerify on each public key and its proof MUST be VALID. The caller is responsible for ensuring that this precondition is met. If it is violated, this scheme provides no security against aggregate signature forgery.

```
result = FastAggregateVerify((PK_1, ..., PK_n), message, signature)

Inputs:
- PK_1, ..., PK_n, public keys in the format output by SkToPk.
- message, an octet string.
- signature, an octet string output by Aggregate.

Outputs:
- result, either VALID or INVALID.

Preconditions:
- n >= 1, otherwise return INVALID.
- The caller MUST know a proof of possession for all PK_i, and the
  result of evaluating PopVerify on PK_i and this proof MUST be VALID.
  See discussion above.

Procedure:
1. aggregate = pubkey_to_point(PK_1)
2. for i in 2, ..., n:
3.     next = pubkey_to_point(PK_i)
4.     aggregate = aggregate + next
5. PK = point_to_pubkey(aggregate)
6. return CoreVerify(PK, message, signature)
```

4. Ciphersuites

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

4.1. Ciphersuite format

A ciphersuite specifies all parameters from Section 2.2, a scheme from Section 3, and any parameters the scheme requires. In particular, a ciphersuite comprises:

*ID: the ciphersuite ID, an ASCII string. The REQUIRED format for this string is

"BLS_SIG_" || H2C_SUITE_ID || SC_TAG || "_"

-Strings in double quotes are ASCII-encoded literals.

-H2C_SUITE_ID is the suite ID of the hash-to-curve suite used to define the hash_to_point and hash_pubkey_to_point functions.

-SC_TAG is a string indicating the scheme and, optionally, additional information. The first three characters of this string MUST chosen as follows:

o"NUL" if SC is basic,

o"AUG" if SC is message-augmentation, or

o"POP" if SC is proof-of-possession.

oOther values MUST NOT be used.

SC_TAG MAY be used to encode other information about the ciphersuite, for example, a version number. When used in this way, SC_TAG MUST contain only ASCII characters between 0x21 and 0x7e (inclusive), except that it MUST NOT contain underscore (0x5f).

The RECOMMENDED way to add user-defined information to SC_TAG is to append a colon (':', ASCII 0x3a) and then the informational string. For example, "NUL:version=2" is an appropriate SC_TAG value.

Note that hash-to-curve suite IDs always include a trailing underscore, so no field separator is needed between H2C_SUITE_ID and SC_TAG.

*SC: the scheme, one of basic, message-augmentation, or proof-of-possession.

*SV: the signature variant, either minimal-signature-size or minimal-pubkey-size.

*EC: a pairing-friendly elliptic curve, plus all associated functionality (Section 1.3).

*H: a cryptographic hash function.

*hash_to_point: a hash from arbitrary strings to elliptic curve points. hash_to_point MUST be defined in terms of a hash-to-curve suite [I-D.irtf-cfrg-hash-to-curve].

The RECOMMENDED hash-to-curve domain separation tag is the ciphersuite ID string defined above.

*hash_pubkey_to_point (only specified when SC is proof-of-possession): a hash from serialized public keys to elliptic curve points. hash_pubkey_to_point MUST be defined in terms of a hash-to-curve suite [I-D.irtf-cfrg-hash-to-curve].

The hash-to-curve domain separation tag MUST be distinct from the domain separation tag used for hash_to_point. It is RECOMMENDED that the domain separation tag be constructed similarly to the ciphersuite ID string, namely:

"BLS_POP_" || H2C_SUITE_ID || SC_TAG || "_"

4.2. Ciphersuites for BLS12-381

The following ciphersuites are all built on the BLS12-381 elliptic curve. The required primitives for this curve are given in Appendix A.

These ciphersuites use the hash-to-curve suites BLS12381G1_XMD:SHA-256_SSWU_RO_ and BLS12381G2_XMD:SHA-256_SSWU_RO_ defined in [I-D.irtf-cfrg-hash-to-curve], Section 8.7. Each ciphersuite defines a unique hash_to_point function by specifying a domain separation tag (see [@I-D.irtf-cfrg-hash-to-curve, Section 3.1).

4.2.1. Basic

BLS_SIG_BLS12381G1_XMD:SHA-256_SSWU_RO_NUL_ is defined as follows:

- *SC: basic
- *SV: minimal-signature-size
- *EC: BLS12-381, as defined in Appendix A.
- *H: SHA-256
- *hash_to_point: BLS12381G1_XMD:SHA-256_SSWU_RO_ with the ASCII-encoded domain separation tag

  BLS_SIG_BLS12381G1_XMD:SHA-256_SSWU_RO_NUL_

BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_NUL_ is identical to BLS_SIG_BLS12381G1_XMD:SHA-256_SSWU_RO_NUL_, except for the following parameters:

- *SV: minimal-pubkey-size
- *hash_to_point: BLS12381G2_XMD:SHA-256_SSWU_RO_ with the ASCII-encoded domain separation tag

  BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_NUL_

4.2.2. Message augmentation

BLS_SIG_BLS12381G1_XMD:SHA-256_SSWU_RO_AUG_ is defined as follows:

- *SC: message-augmentation
- *SV: minimal-signature-size
- *EC: BLS12-381, as defined in Appendix A.
- *H: SHA-256
- *hash_to_point: BLS12381G1_XMD:SHA-256_SSWU_RO_ with the ASCII-encoded domain separation tag

  BLS_SIG_BLS12381G1_XMD:SHA-256_SSWU_RO_AUG_

BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_AUG_ is identical to BLS_SIG_BLS12381G1_XMD:SHA-256_SSWU_RO_AUG_, except for the following parameters:

- *SV: minimal-pubkey-size
- *hash_to_point: BLS12381G2_XMD:SHA-256_SSWU_RO_ with the ASCII-encoded domain separation tag

  BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_AUG_

4.2.3. Proof of possession

BLS_SIG_BLS12381G1_XMD:SHA-256_SSWU_RO_POP_ is defined as follows:

- *SC: proof-of-possession
- *SV: minimal-signature-size
- *EC: BLS12-381, as defined in Appendix A.
- *H: SHA-256
- *hash_to_point: BLS12381G1_XMD:SHA-256_SSWU_RO_ with the ASCII-encoded domain separation tag

  BLS_SIG_BLS12381G1_XMD:SHA-256_SSWU_RO_POP_
- *hash_pubkey_to_point: BLS12381G1_XMD:SHA-256_SSWU_RO_ with the ASCII-encoded domain separation tag

  BLS_POP_BLS12381G1_XMD:SHA-256_SSWU_RO_POP_

BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_POP_ is identical to BLS_SIG_BLS12381G1_XMD:SHA-256_SSWU_RO_POP_, except for the following parameters:

- *SV: minimal-pubkey-size

- *hash_to_point: BLS12381G2_XMD:SHA-256_SSWU_RO_ with the ASCII-encoded domain separation tag

  BLS_SIG_BLS12381G2_XMD:SHA-256_SSWU_RO_POP_

- *hash_pubkey_to_point: BLS12381G2_XMD:SHA-256_SSWU_RO_ with the ASCII-encoded domain separation tag

  BLS_POP_BLS12381G2_XMD:SHA-256_SSWU_RO_POP_

5. Security Considerations

5.1. Choosing a salt value for KeyGen

KeyGen uses HKDF to generate keys. The security analysis of HKDF assumes that the salt value is either empty (in which case it is replaced by the all-zeros string) or an unstructured string. Versions of this document prior to number 4 set the salt parameter to "BLS-SIG-KEYGEN-SALT-", which does not meet this requirement. If, as is common, the hash function H is modeled as a random oracle, this requirement is obviated.

When choosing a salt value other than "BLS-SIG-KEYGEN-SALT-" or H("BLS-SIG-KEYGEN-SALT-"), the RECOMMENDED method is to fix a uniformly random octet string whose length equals the output length of H.

5.2. Validating public keys

All algorithms in Section 2 and Section 3 that operate on public keys require first validating those keys. For the basic and message augmentation schemes, the use of KeyValidate is REQUIRED. For the proof of possession scheme, each public key MUST be accompanied by a proof of possession, and use of PopVerify is REQUIRED.

KeyValidate requires all public keys to represent valid, non-identity points in the correct subgroup. A valid point and subgroup membership are required to ensure that the pairing operation is defined (Section 5.3).

A non-identity point is required because the identity public key has the property that the corresponding secret key is equal to zero, which means that the identity point is the unique valid signature for every message under this key. A malicious signer could take

advantage of this fact to equivocate about which message he signed. While non-equivocation is not a required property for a signature scheme, equivocation is infeasible for BLS signatures under any nonzero secret key because it would require finding colliding inputs to the hash_to_point function, which is assumed to be collision resistant. Prohibiting SK == 0 eliminates the exceptional case, which may help to prevent equivocation-related security issues in protocols that use BLS signatures.

5.3. Skipping membership check

Some existing implementations skip the signature_subgroup_check invocation in CoreVerify (Section 2.7), 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 1.3) 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

BLS signatures are deterministic. This protects against attacks arising from signing with bad randomness, for example, the nonce reuse attack on ECDSA [HDWH12].

As discussed in Section 2.3, 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.

5.6. Implementing hash_to_point and hash_pubkey_to_point

The security analysis models hash_to_point and hash_pubkey_to_point 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_point and hash_pubkey_to_point. The domain separation tag format used in Section 4 is the RECOMMENDED one.

6. Implementation Status

This section will be removed in the final version of the draft. There are currently several implementations of BLS signatures using the BLS12-381 curve.

*Algorand: bls_sigs_ref.

*Chia: spec python/C++. Here, they are swapping G1 and G2 so that the public keys are small, and the benefits of avoiding a membership check during signature verification would even be more substantial. The current implementation does not seem to implement the membership check. Chia uses the Fouque-Tibouchi hashing to the curve, which can be done in constant time.

*Dfinity: go BLS. The current implementations do not seem to implement the membership check.

*Ethereum 2.0: spec.

7. Related Standards

*Pairing-friendly curves, [I-D.irtf-cfrg-pairing-friendly-curves]

*Pairing-based Identity-Based Encryption IEEE 1363.3.

*Identity-Based Cryptography Standard rfc5901.

*Hashing to Elliptic Curves [I-D.irtf-cfrg-hash-to-curve], in order to implement the hash function hash_to_point.

*EdDSA rfc8032.

8. IANA Considerations

TBD (consider to register ciphersuite identifiers for BLS signature and underlying pairing curves)

9. Normative References

[ZCash] Electric Coin Company, "BLS12-381", July 2017, <https://github.com/zkcrypto/pairing/blob/

34aa52b0f7bef705917252ea63e5a13fa01af551/src/bls12_381/README.md#serialization>.

[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>.

10. Informative References

[Bol03] Boldyreva, A., "Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme", January 2003, <https://link.springer.com/chapter/10.1007%2F3-540-36288-6_3>.

[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-10, 30 July 2021, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-pairing-friendly-curves-10>.

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

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

[BLS01] Boneh, D., Lynn, B., and H. Shacham, "Short signatures from the Weil pairing", December 2001, <https://www.iacr.org/archive/asiacrypt2001/22480516.pdf>.

[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>.

[Scott21] Scott, M., "A note on group membership tests for G1, G2 and GT on BLS pairing-friendly curves", September 2021, <https://eprint.iacr.org/2021/1130.pdf>.

[LOSSW06] Lu, S., Ostrovsky, R., Sahai, A., Shacham, H., and B. Waters, "Sequential Aggregate Signatures and Multisignatures Without Random Oracles", May 2006, <https://link.springer.com/chapter/10.1007/11761679_28>.

[RY07] Ristenpart, T. and S. Yilek, "The Power of Proofs-of-Possession: Securing Multiparty Signatures against Rogue-

Key Attacks", May 2007, <https://link.springer.com/chapter/10.1007%2F978-3-540-72540-4_13>.

[GMT19] Guillevic, A., Masson, S., and E. Thome, "Cocks–Pinch curves of embedding degrees five to eight and optimal ate pairing computation", 2019.

[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>.

[BGLS03] Boneh, D., Gentry, C., Lynn, B., and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps", May 2003, <https://link.springer.com/chapter/10.1007%2F3-540-39200-9_26>.

[BNN07] Bellare, M., Namprempre, C., and G. Neven, "Unrestricted aggregate signatures", July 2007, <https://link.springer.com/chapter/10.1007%2F978-3-540-73420-8_37>.

[BDN18] Boneh, D., Drijvers, M., and G. Neven, "Compact multi-signatures for shorter blockchains", December 2018, <https://link.springer.com/chapter/10.1007/978-3-030-03329-3_15>.

[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>.

[FIPS180-4] National Institute of Standards and Technology (NIST), "FIPS Publication 180-4: Secure Hash Standard", August 2015, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.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", August 2012, <https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final228.pdf>.

Appendix A. BLS12-381

The ciphersuites in Section 4 are based upon the BLS12-381 pairing-friendly elliptic curve. The following defines the correspondence

between the primitives in Section 1.3 and the parameters given in Section 4.2.1 of [I-D.irtf-cfrg-pairing-friendly-curves].

*E1, G1: the curve E and its order-r subgroup.

*E2, G2: the curve E' and its order-r subgroup.

*GT: the subgroup G_T.

*P1: the point BP.

*P2: the point BP'.

*e: the optimal Ate pairing defined in Appendix A of [I-D.irtf-cfrg-pairing-friendly-curves].

*point_to_octets and octets_to_point use the compressed serialization formats for E1 and E2 defined by [ZCash].

*subgroup_check MAY use either the naive check described in Section 1.3 or the optimized checks given by [Bowe19] or [Scott21].

Appendix B. Test Vectors

TBA: (i) test vectors for both variants of the signature scheme (signatures in G2 instead of G1) , (ii) test vectors ensuring membership checks, (iii) intermediate computations ctr, hm.

Appendix C. Security analyses

The security properties of the BLS signature scheme are proved in [BLS01].

[BGLS03] prove the security of aggregate signatures over distinct messages, as in the basic scheme of Section 3.1.

[BNN07] prove security of the message augmentation scheme of Section 3.2.

[Bol03][LOSSW06][RY07] prove security of constructions related to the proof of possession scheme of Section 3.3.

[BDN18] prove the security of another rogue key defense; this defense is not standardized in this document.

Authors' Addresses

Dan Boneh
Stanford University

United States of America

Email: dabo@cs.stanford.edu

Sergey Gorbunov
University of Waterloo
Waterloo, ON
Canada

Email: sgorbunov@uwaterloo.ca

Riad S. Wahby
Carnegie Mellon University
United States of America

Email: rsw@cs.stanford.edu

Hoeteck Wee
NTT Research and ENS, Paris
Boston, MA,
United States of America

Email: wee@di.ens.fr

Christopher A. Wood
Cloudflare, Inc.
San Francisco, CA,
United States of America

Email: caw@heapingbits.net

Zhenfei Zhang
Algorand
Boston, MA,
United States of America

Email: zhenfei@algorand.com