

CFRG
Internet-Draft
Expires: August 12, 2019

D. Boneh
Stanford University
S. Gorbunov
Algorand and University of Waterloo
H. Wee
Algorand and ENS, Paris
Z. Zhang
Algorand
February 8, 2019

BLS Signature Scheme
draft-boneh-bls-signature-00

Abstract

The BLS signature scheme was introduced by Boneh-Lynn-Shacham in 2001. The signature scheme relies on pairing-friendly curves and supports non-interactive aggregation properties. That is, given a collection of signatures ($\sigma_1, \dots, \sigma_n$), anyone can produce a short signature (σ) that authenticates the entire collection. BLS signature scheme is simple, efficient and can be used in a variety of network protocols and systems to compress signatures or certificate chains. This document specifies the BLS signature and the aggregation algorithms.

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 August 12, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Terminology	3
1.2. Signature Scheme Algorithms and Properties	4
1.2.1. Aggregation	4
1.2.2. Security	5
2. BLS Signature	6
2.1. Preliminaries	7
2.2. Keygen: Key Generation	8
2.3. Sign: Signature Generation	8
2.4. Verify: Signature Verification	8
2.5. Aggregate	8
2.5.1. Verify-Aggregated-1	9
2.5.2. Verify-Aggregated-n	9
2.5.3. Implementation optimizations	9
2.6. Auxiliary Functions	9
2.6.1. Preliminaries	10
2.6.2. Type conversions	10
2.6.3. Hash to groups	14
2.7. Security analysis	15
3. Security Considerations	15
3.1. Verifying public keys	15
3.2. Skipping membership check	15
3.3. Side channel attacks	15
3.4. Randomness considerations	16
3.5. Implementing the hash function	16
4. Implementation Status	16
5. Related Standards	16
6. IANA Considerations	17
7. Appendix A. Test Vectors	17
8. Appendix B. Reference	17
9.1. URIs	17
Authors' Addresses	18

1. Introduction

A signature scheme is a fundamental cryptographic primitive used on the Internet that is used to protect integrity of communication. Only holder of the 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 ($\sigma_1, \dots, \sigma_n$) can be compressed into a single signature (σ). Moreover, the compressed signature can be verified using only $n+1$ pairings (as opposed to $2n$ pairings, when verifying naively n signatures).

Given the above properties, we believe the scheme will find very interesting applications. The immediate applications include compressing signature chains in Public Key Infrastructure (PKI) and in the 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 compress the n signatures into a single signature.

In addition, the BLS signature scheme is already integrated into major blockchain projects such as Ethereum, Algorand, Chia and Dfinity. 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. Terminology

The following terminology is used through this document:

- o SK: The private key for the signature scheme.
- o PK: The public key for the signature scheme.

- o msg: The input to be signed by the signature scheme.
- o sigma : The digital signature output.
- o Signer: The Signer generates a pair (SK, PK), publishes PK for everyone to see, but keeps the private key SK.
- o Verifier: The Verifier holds a public key PK. It receives (msg, sigma) that it wishes to verify.
- o Aggregator: The Aggregator receives a collection of signatures (sigma_1, ..., sigma_n) that it wishes to compress into a short signature.

1.2. Signature Scheme Algorithms and Properties

A signature scheme comes with a key generation algorithm that generates a public key PK and a private key SK.

The Signer, given an input msg, uses the private key SK to obtain and output a signature sigma.

```
sigma = Sign(SK, msg)
```

The signing algorithm may be deterministic or randomized, depending on the scheme. Looking ahead, BLS instantiates a deterministic signing algorithm.

The signature sigma allows a Verifier holding the public key PK to verify that sigma is indeed produced by the signer holding the associated secret key. Thus, the digital scheme also comes with an algorithm

```
Verify(PK, msg, sigma)
```

that outputs VALID if sigma is a valid signature of msg, and INVALID otherwise.

We require that PK, sigma and msg are octet strings.

1.2.1. Aggregation

An aggregatable signature scheme includes an algorithm that allows to compress a collection of signatures into a short signature.

```
sigma = Aggregate((PK_1, sigma_1), ..., (PK_n, sigma_n))
```

Note that the aggregator does not need to know the messages corresponding to individual signatures.

The scheme also includes an algorithm to verify an aggregated signature, given a collection of corresponding public keys, the aggregated signature, and one or more messages.

`Verify-Aggregated((PK_1, msg_1), ..., (PK_n, msg_n), sigma)`

that outputs `VALID` if `sigma` is a valid aggregated signature of messages `msg_1, ..., msg_n`, and `INVALID` otherwise.

The verification algorithm may also accept a simpler interface that allows to verify an aggregate signature of the same message. That is, `msg_1 = msg_2 = ... = msg_n`.

`Verify-Aggregated(PK_1, ..., PK_n, msg, sigma)`

1.2.2. Security

1.2.2.1. Message Unforgeability

Consider the following game between an adversary and a challenger. The challenger generates a key-pair (PK, SK) and gives PK to the adversary. The adversary may repeatedly query the challenger on any message `msg` to obtain its corresponding signature `sigma`. Eventually the adversary outputs a pair $(msg', sigma')$.

Unforgeability means no adversary can produce a pair $(msg', sigma')$ for a message `msg'` which he never queried the challenger and `Verify(PK, msg, sigma)` outputs `VALID`.

1.2.2.2. Strong Message Unforgeability

In the strong unforgeability game, the game proceeds as above, except no adversary should be able to produce a pair $(msg', sigma')$ that verifies (i.e. `Verify(PK, msg, sigma)` outputs `VALID`) given that he never queried the challenger on `msg'`, or if he did query and obtained a reply `sigma`, then `sigma != sigma'`.

More informally, the strong unforgeability means that no adversary can produce a different signature (not provided by the challenger) on a message which he queried before.

1.2.2.3. Aggregation Unforgeability

Consider the following game between an adversary and a challenger. The challenger generates a key-pair (PK, SK) and gives PK to the adversary. The adversary may repeatedly query the challenger on any message msg to obtain its corresponding signature σ . Eventually the adversary outputs a sequence $((PK_1, msg_1), \dots, (PK_n, msg_n), (PK, msg), \sigma)$.

Aggregation unforgeability means that no adversary can produce a sequence where it did not query the challenger on the message msg , and $Verify\text{-}Aggregated((PK_1, msg_1), \dots, (PK_n, msg_n), (PK, msg), \sigma)$ outputs `VALID`.

We note that aggregation unforgeability implies message unforgeability.

TODO: We may also consider a strong aggregation unforgeability property.

2. BLS Signature

BLS signatures require pairing-friendly curves given by $e : G_1 \times G_2 \rightarrow GT$, where G_1, G_2 are prime-order subgroups of elliptic curve groups E_1, E_2 . Such curves are described in [I-D.pairing-friendly-curves], one of which is BLS12-381.

There are two variants of the scheme:

1. (minimizing signature size) Put signatures in G_1 and public keys in G_2 , where G_1/E_1 has the more compact representation. For instance, when instantiated with the pairing-friendly curve BLS12-381, this yields signature size of 48 bytes, whereas the ECDSA signature over curve25519 has a signature size of 64 bytes.
2. (minimizing public key size) Put public keys in G_1 and signatures in G_2 . This latter case comes up when we do signature aggregation, where most of the communication costs come from public keys. This is particularly relevant in applications such as blockchains and compressing certificate chains, where the goal is to minimize the total size of multiple public keys and aggregated signatures.

The rest of the write-up assumes the first variant. It is straightforward to obtain algorithms for the second variant from those of the first variant where we simply swap G_1, E_1 with G_2, E_2 respectively.

2.1. Preliminaries

Notation and primitives used:

- o E_1, E_2 - elliptic curves (EC) defined over a field
- o P_1, P_2 - elements of E_1, E_2 of prime order r
- o G_1, G_2 - prime-order subgroups of E_1, E_2 generated by P_1, P_2
- o GT - order r subgroup of the multiplicative group over a field
- o We require an efficient pairing $e : (G_1, G_2) \rightarrow GT$ that is bilinear and non-degenerate.
- o Elliptic curve operations in E_1 and E_2 are written in additive notation, with $P+Q$ denoting point addition and $x*P$ denoting scalar multiplication of a point P by a scalar x .
TBD: [I-D.pairing-friendly-curves] uses the notation $x[P]$.
- o Field operations in GT are written in multiplicative notation, with $a*b$ denoting field element multiplication.
- o $||$ - octet string concatenation
- o `suite_string` - an identifier for the ciphersuite. May include an identifier of the curve, for example BLS12-381, an identifier of the hash function, for example SHA512, and the algorithm in use, for example, try-and-increment.

Type conversions:

- o `int_to_string(a, len)` - conversion of nonnegative integer a to octet string of length len .
- o `string_to_int(a_string)` - conversion of octet string `a_string` to nonnegative integer.
- o `E1_to_string` - conversion of E_1 point to octet string
- o `string_to_E1` - conversion of octet string to E_1 point. Returns `INVALID` if the octet string does not convert to a valid E_1 point.

Hashing Algorithms

- o `hash_to_G1` - cryptographic hashing of octet string to G_1 element. Must return a valid G_1 element. Specified in Section {{auxiliary}}.

2.2. Keygen: Key Generation

Output: PK, SK

1. SK = x, chosen as a random integer in the range 1 and r-1
2. PK = x*P2
3. Output PK, SK

2.3. Sign: Signature Generation

Input: SK = x, msg Output: sigma

1. Input a secret key SK = x and a message digest msg
2. H = hash_to_G1(suite_string, msg)
3. Gamma = x*H
4. sigma = El_to_string(Gamma)
5. Output sigma

2.4. Verify: Signature Verification

Input: PK, msg, sigma Output: "VALID" or "INVALID"

1. H = hash_to_G1(suite_string, msg)
2. Gamma = string_to_El(sigma)
3. If Gamma is "INVALID", output "INVALID" and stop
4. If r*Gamma != 0, output "INVALID" and stop
5. Compute c = e(Gamma, P2)
6. Compute c' = e(H, PK)
7. If c and c' are equal, output "VALID", else output "INVALID"

2.5. Aggregate

Input: (PK_1, sigma_1), ..., (PK_n, sigma_n) Output: sigma

1. Output sigma = sigma_1 + sigma_2 + ... + sigma_n

2.5.1. Verify-Aggregated-1

Input: $(PK_1, \dots, PK_n), msg, sigma$ Output: "VALID" or "INVALID"

1. $PK' = PK_1 + \dots + PK_n$
2. Output $Verify(PK', msg, sigma)$

2.5.2. Verify-Aggregated-n

Input: $(PK_1, msg_1), \dots, (PK_n, msg_n), sigma$
Output: "VALID" or "INVALID"

1. $H_i = hash_to_G1(suite_string, msg_i)$
2. $\Gamma = string_to_E1(sigma)$
3. If Γ is "INVALID", output "INVALID" and stop
4. If $r * \Gamma \neq 0$, output "INVALID" and stop
5. Compute $c = e(\Gamma, P_2)$
6. Compute $c' = e(H_1, PK_1) * \dots * e(H_n, PK_n)$
7. If c and c' are equal, output "VALID", else output "INVALID"

2.5.3. Implementation optimizations

There are several optimizations we should use to speed up verification. First, we can use multi-pairings instead of a normal pairing. Roughly speaking, this means that we can reuse the "final exponentiation" step in all of the pairing operations. In addition, we can carry out pre-computation on the public keys for aggregate verification.

2.6. Auxiliary Functions

Here, we describe the auxiliary functions relating to serialization and hashing to the elliptic curves E , where E may be $E1$ or $E2$.

(Authors' note: this section is extremely preliminary and we anticipate substantial updates pending feedback from the community. We describe a generic approach for hashing, in order to cover hashing into curves defined over prime power extension fields, which are not covered in [I-D.irtf-cfrg-hash-to-curve]. We expect to support several different hashing algorithms specified via the `suite_string`.)

2.6.1. Preliminaries

In all the pairing-friendly curves, E is defined over a field $\text{GF}(p^k)$. We also assume an explicit isomorphism that allows us to treat $\text{GF}(p^k)$ as $\text{GF}(p)$. In most of the curves in [I-D.pairing-friendly-curves], we have $k=1$ for E1 and $k=2$ for E2.

Each point (x,y) on E can be specified by the x -coordinate in $\text{GF}(p)^k$ plus a single bit to determine whether the point is (x,y) or $(x,-y)$, thus requiring $k \log(p) + 1$ bits [I-D.irtf-cfrg-hash-to-curve].

Concretely, we encode a point (x,y) on E as a string comprising k substrings s_1, \dots, s_k each of length $\log(p)+2$ bits, where

- o the first bit of s_1 indicates whether E is the point at infinity
- o the second bit of s_1 indicates whether the point is (x,y) or $(x,-y)$
- o the first two bits of s_2, \dots, s_k are 00
- o the x -coordinate is specified by the last $\log(p)$ bits of s_1, \dots, s_k

In fact, we will pad each substring with 0s so that the length of each substring is a multiple of 8.

This section uses the following constants:

- o `pbits`: the number of bits to represent integers modulo p .
- o `padded_pbits`: the smallest multiple of 8 that is greater than `pbits+2`.
- o `padlen`: `padded_pbits - padlen`

curve	pbits	padded_pbits	padlen
BLS-381	381	384	3

2.6.2. Type conversions

In general we view a string `str` as a vector of substrings s_1, \dots, s_k for $k \geq 1$; each substring is of `padded_pbits` bits; and k is set properly according to the individual curve. For example, for BLS12-381 curve, $k=1$ for E1 and 2 for E2. If the input string is not

a multiple of `padded_pbits`, we tail pad the string to meet the correct length.

A string that encodes an E1/E2 point may have the following structure: * for the first substring `s_1` * the first bit indicates if the point is the point at infinity * the second bit is either 0 or 1, denoted by `y_bit` * the third to `padlen` bits are 0

o for the rest substrings `s_2, ... s_k`

* the first `padlen` bits are 0s

TBD: some implementation uses an additional leading bit to indicate the string is in a compressed form (give x coordinate and the parity/sign of y coordinate) or in an uncompressed form (give both x and y coordinate).

2.6.2.1. curve-to-string

Input:

`input_string` - a point $P = (x, y)$ on the curve

Output:

a string of $k * \text{padded_pbits}$

Steps:

1. If P is the point at infinity, output `0b1000...0`
2. Parse y as y_1, \dots, y_k ; set `y_bit` as $y_1 \bmod 2$
3. Parse x as x_1, \dots, x_k
4. set the substring `s_1 = 0 | y_bit | padlen-2 of 0s | int_to_string(x_1)`
5. set substrings `s_i = padlen of 0s | int_to_string(x_i)` for $2 \leq i \leq k$
6. Output the string `s_1 | s_2 | ... | s_k`

2.6.2.2. string-to-curve

The algorithm takes as follows:

Input:

input_string - a single octet string.

Output:

Either a point P on the curve, or INVALID

Steps:

1. If `length(input_string)` is $< \text{padded_pbits}/8$ bytes, lead pad `input_string` with 0s;
2. If `length(input_string)` is not a multiple of `padded_pbits/8` bytes, tail pad with 0, ..., 0;
3. Parse `input_string` as a vector of substrings `s_1, ..., s_k`
4. `b = s_1[0]`; i.e., the first byte of the first substring;
5. If the first bit of `b` is 1, return `P = 0` (the point at infinity)
6. Set `y_bit` to be the second bit of `b` and then set the second bit of `b` to 0
7. If the third to plen bits of `input_string` are not 0, return INVALID
8. Set `x_1 = string_to_int(s_1)`
 1. if `x_1 > p` then return INVALID
9. for `i` in `[2 ... k]`
 1. `b = s_i[0]`
 2. if top plen bits of `b` is not 0, return INVALID
 3. set `x_i = string_to_int(s_i)`
 1. if `x_i > p` then return INVALID
10. Set `x = (x_1, ..., x_k)`

11. solve for y so that (x, y) satisfies elliptic curve equation;
 - * output INVALID if equation is not solvable with x
 - * parse y as (y_1, \dots, y_k)
 - * if solutions exist, there should be a pair of y s where y_1 -s differ by parity
 - * set y to be the solution where y_1 is odd if $y_{\text{bit}} = 1$
 - * set y to be the solution where y_1 is even if $y_{\text{bit}} = 0$
12. output $P = (x, y)$ as a curve point.

TBD: check the parity property remains true for E2. The Chia and Ethereum implementations use lexicographic ordering.

2.6.2.3. alt-str-to-curve

The algorithm takes as follows:

Input:

input_string - a single octet string.

Output:

Either a point P on the curve, or INVALID

Steps:

1. If $\text{length}(\text{input_string})$ is $< \text{padded_pbits}/8$ bytes, lead pad input_string with 0s;
2. If $\text{length}(\text{input_string})$ is not a multiple of 48 bytes, tail pad with 0, ..., 0s;
3. Parse input_string as a vector of substrings s_1, \dots, s_k
4. Set the first padlen bits except for the second bit of $s_1[0]$ to 0
5. Set the first padlen bits for $s_2[0], \dots, s_k[0]$ to 0
6. call `string_to_curve(input_string)`

2.6.3. Hash to groups

The following `hash_to_G1_try_and_increment` algorithm implements `hash_to_G1` in a simple and generic way that works for any pairing-friendly curve. It follows the try-and-increment approach [I-D.irtf-cfrg-hash-to-curve] and uses `alt_str_to_curve` as a subroutine. The running depends on `alpha_string`, and for the appropriate instantiations, is expected to find a valid G1 element after approximately two attempts (i.e., when `ctr=1`) on average.

The following pseudocode is adapted from draft-irtf-cfrg-vrf-03 Section 5.4.1.1.

Recall that $\text{cofactor} = |E1|/|G1|$. This algorithm also uses a hash functions that hashes arbitrary strings into strings of 384 bits.

`hash_to_G1_try_and_increment(suite_string, alpha_string)`

input: `suite_string` - an identifier to indicate the curves and a hash function that outputs `k*packed_pbits` bits
`alpha_string` - the input string to be hashed

Output:

H - hashed value, a point in G1

Steps:

1. `ctr = 0`
2. `one_string = 0x01 = int_to_string(1, 1)`, a single octet with value 1
3. `H = "INVALID"`
4. While H is "INVALID" or H is EC point at infinity:
 1. `ctr_string = int_to_string(ctr, 1)`
 2. `hash_string = Hash(suite_string || one_string || alpha_string || ctr_string)`
 3. `H = alt_str_to_curve(hash_string)`
 4. If H is not "INVALID" and `cofactor > 1`, set `H = cofactor * H`
 5. `ctr = ctr + 1`

5. Output H

Note that this hash to group function will never hash into the point at infinity. This does not affect the security since the output distribution is statistically indistinguishable from the uniform distribution over the group.

2.7. Security analysis

The BLS signature scheme achieves strong message unforgeability and aggregation unforgeability under the co-CDH assumption, namely that given P_1 , a_{P_1} , P_2 , b_{P_2} , it is hard to compute $\{ab\} * P_1$. [BLS01, BGLS03]

3. Security Considerations

3.1. Verifying public keys

When users register a public key, we should ensure that it is well-formed. This requires a G2 membership test. In applications where we use aggregation, we would further require that users prove knowledge of the corresponding secret key during registration to prevent rogue key attacks.

TBA: additional discussion on this, e.g. [Ristenpart-Yilek 06], and alternative mechanisms for securing aggregation against rogue key attacks, e.g. [Boneh-Drijvers-Neven 18]; there, pre-processing public keys would speed up verification.

3.2. Skipping membership check

Several existing implementations skip step 4 (membership in G1) in Verify. In this setting, the BLS signature remains unforgeable (but not strongly unforgeable) under a stronger assumption:

given P_1 , a_{P_1} , P_2 , b_{P_2} , it is hard to compute U in E_1 such that $e(U, P_2) = e(a_{P_1}, b_{P_2})$.

3.3. Side channel attacks

It is important to protect the secret key in implementations of the signing algorithm. We can protect against some side-channel attacks by 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. To achieve this, we require that point multiplication in G1 should run in constant time with respect to the scalar.

3.4. Randomness considerations

BLS signatures are deterministic. This protects against attacks arising from signing with bad randomness.

3.5. Implementing the hash function

The security analysis models the hash function H as a random oracle, and it is crucial that we implement H using a cryptographically secure hash function. <!-- At the moment, hashing onto G_1 is typically implemented by hashing into E_1 and then multiplying by the cofactor; this needs to be taken into account in the security proof (namely, the reduction needs to simulate the corresponding E_1 element).-->

4. Implementation Status

There are currently several implementations of BLS signatures using the BLS12-381 curve.

- o Algorand: TBA
- o Chia: spec [1] python/C++ [2]. Here, they are swapping G_1 and G_2 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.
- o Dfinity: go [3] BLS [4]. The current implementations do not seem to implement the membership check.
- o Ethereum 2.0: spec [5]

5. Related Standards

- o Pairing-friendly curves draft-yonezawa-pairing-friendly-curves [6]
- o Pairing-based Identity-Based Encryption IEEE 1363.3 [7].
- o Identity-Based Cryptography Standard rfc5901 [8].
- o Hashing to Elliptic Curves draft-irtf-cfrg-hash-to-curve-02 [9], in order to implement the hash function H . The current draft does not cover pairing-friendly curves, where we need to handle curves over prime power extension fields $\text{GF}(p^k)$.

- o Verifiable random functions draft-irtf-cfrg-vrf-03 [10].
Section 5.4.1 also discusses instantiations for H.

- o EdDSA rfc8032 [11]

6. IANA Considerations

This document does not make any requests of IANA.

7. Appendix A. 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.

8. Reference

8.1. Normative References

[BLS 01] Dan Boneh, Ben Lynn, Hovav Shacham: Short Signatures from the Weil Pairing. ASIACRYPT 2001: 514-532.

[BGLS 03] Dan Boneh, Craig Gentry, Ben Lynn, Hovav Shacham: Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. EUROCRYPT 2003: 416-432.

[I-D.irtf-cfrg-hash-to-curve] S. Scott, N. Sullivan, and C. Wood: "Hashing to Elliptic Curves", draft-irtf-cfrg-hash-to-curve-01 (work in progress), July 2018.

[I-D.pairing-friendly-curves] S. Yonezawa, S. Chikara, T. Kobayashi, T. Saito: "Pairing-Friendly Curves", draft-yonezawa-pairing-friendly-curves-00, Jan 2019.

8.2. Informative References

9. References

9.1. URIs

- [1] <https://github.com/Chia-Network/bls-signatures/blob/master/SPEC.md>
- [2] <https://github.com/Chia-Network/bls-signatures>
- [3] <https://github.com/dfinity/go-dfinity-crypto>
- [4] <https://github.com/dfinity/bls>
- [5] https://github.com/ethereum/eth2.0-specs/blob/master/specs/bls_signature.md

- [6] <https://tools.ietf.org/html/draft-yonezawa-pairing-friendly-curves-00>
- [7] <https://ieeexplore.ieee.org/document/6662370>
- [8] <https://tools.ietf.org/html/rfc5091>
- [9] <https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-02>
- [10] <https://tools.ietf.org/html/draft-irtf-cfrg-vrf-03>
- [11] <https://tools.ietf.org/html/rfc8032>

Authors' Addresses

Dan Boneh
Stanford University
USA

Email: dabo@cs.stanford.edu

Sergey Gorbunov
Algorand and University of Waterloo
Boston, MA
USA

Email: sgorbunov@uwaterloo.ca

Hoeteck Wee
Algorand and ENS, Paris
Boston, MA
USA

Email: hoeteck.wee@ens.fr

Zhenfei Zhang
Algorand
Boston, MA
USA

Email: zhenfei@algorand.com