

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 ( $\sigma$ ) 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 ( $\sigma$ ). 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  $\sigma$ . 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  $Verify\_Aggregated((PK_1, msg_1), \dots, (PK_n, msg_n), (PK, msg), \sigma)$  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.  $PK = x \cdot P_2$
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.  $H = \text{hash\_to\_G1}(\text{suite\_string}, \text{msg})$
3.  $\Gamma = x \cdot H$
4.  $\text{sigma} = \text{El\_to\_string}(\Gamma)$
5. Output sigma

## 2.4. Verify: Signature Verification

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

1.  $H = \text{hash\_to\_G1}(\text{suite\_string}, \text{msg})$
2.  $\Gamma = \text{string\_to\_El}(\text{sigma})$
3. If  $\Gamma$  is "INVALID", output "INVALID" and stop
4. If  $r \cdot \Gamma \neq 0$ , output "INVALID" and stop
5. Compute  $c = e(\Gamma, P_2)$
6. Compute  $c' = e(H, PK)$
7. If  $c$  and  $c'$  are equal, output "VALID", else output "INVALID"

## 2.5. Aggregate

Input:  $(PK_1, \text{sigma}_1), \dots, (PK_n, \text{sigma}_n)$       Output: sigma

1. Output  $\text{sigma} = \text{sigma}_1 + \text{sigma}_2 + \dots + \text{sigma}_n$



### 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  $\Gamma$  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 * padded_pbits`

Steps:

1. If  $P$  is the point at infinity, output `0b1000...0`
2. Parse `y` as `y_1, ..., y_k`; set `y_bit` as `y_1 mod 2`
3. Parse `x` as `x_1, ..., 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  $\text{padded\_pbits}/8$  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*padded_pbts` 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](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](mailto:dabo@cs.stanford.edu)

Sergey Gorbunov  
Algorand and University of Waterloo  
Boston, MA  
USA

Email: [sgorbunov@uwaterloo.ca](mailto:sgorbunov@uwaterloo.ca)

Hoeteck Wee  
Algorand and ENS, Paris  
Boston, MA  
USA

Email: [hoeteck.wee@ens.fr](mailto:hoeteck.wee@ens.fr)

Zhenfei Zhang  
Algorand  
Boston, MA  
USA

Email: [zhenfei@algorand.com](mailto:zhenfei@algorand.com)

Network Working Group  
Internet-Draft  
Intended status: Informational  
Expires: September 2, 2018

C. Cremers  
L. Garratt  
University of Oxford  
S. Smyshlyaev  
CryptoPro  
N. Sullivan  
Cloudflare  
C. Wood  
Apple Inc.  
March 01, 2018

Randomness Improvements for Security Protocols  
draft-cremers-cfrg-randomness-improvements-00

Abstract

Randomness is a crucial ingredient for TLS and related security protocols. Weak or predictable "cryptographically-strong" pseudorandom number generators (CSPRNGs) can be abused or exploited for malicious purposes. The Dual EC random number backdoor and Debian bugs are relevant examples of this problem. This document describes a way for security protocol participants to mix their long-term private key into the entropy pool(s) from which random values are derived. This augments and improves randomness from broken or otherwise subverted CSPRNGs.

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 September 2, 2018.

## Copyright Notice

Copyright (c) 2018 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 . . . . .	2
2. Randomness Wrapper . . . . .	3
3. Tag Generation . . . . .	4
4. Application to TLS . . . . .	4
5. IANA Considerations . . . . .	4
6. Security Considerations . . . . .	4
7. Normative References . . . . .	5
Authors' Addresses . . . . .	6

## 1. Introduction

Randomness is a crucial ingredient for TLS and related transport security protocols. TLS in particular uses Random Number Generators (RNGs) to generate several values: session IDs, ephemeral key shares, and ClientHello and ServerHello random values. RNG failures such as the Debian bug described in [DebianBug] can lead to insecure TLS connections. RNGs may also be intentionally weakened to cause harm [DualEC]. In such cases where RNGs are poorly implemented or insecure, an adversary may be able to predict its output and recover secret Diffie-Hellman key shares that protect the connection.

This document proposes an improvement to randomness generation in security protocols inspired by the "NAXOS trick" [NAXOS]. Specifically, instead of using raw entropy where needed, e.g., in generating ephemeral key shares, a party's long-term private key is mixed into the entropy pool. In the NAXOS key exchange protocol, raw entropy output  $x$  is replaced by  $H(x, sk)$ , where  $sk$  is the sender's private key. Unfortunately, as private keys are often isolated in HSMs, direct access to compute  $H(x, sk)$  is impossible. An alternate yet functionally equivalent construction is needed.

The approach described herein replaces the NAXOS hash with a keyed hash, or pseudorandom function (PRF), where the key is derived from raw entropy output and a private key signature. Implementations SHOULD apply this technique when indirect access to a private key is available and CSPRNG randomness guarantees are dubious, or to provide stronger guarantees about possible future issues with the randomness.

## 2. Randomness Wrapper

Let  $x$  be the raw entropy output of a CSPRNG. When properly instantiated,  $x$  should be indistinguishable from a random string of length  $|x|$ . However, as previously discussed, this is not always true. To mitigate this problem, we propose an approach for wrapping the CSPRNG output with a construction that artificially injects randomness into a value that may be lacking entropy.

Let  $\text{PRF}(k, m)$  be a cryptographic pseudorandom function, e.g., HMAC [RFC2104], that takes as input a key  $k$  of length  $L$  and message  $m$  and produces an output of length  $M$ . For example, when using HMAC with SHA256,  $L$  and  $M$  are 256 bits. Let  $\text{Sig}(sk, m)$  be a function that computes a signature of message  $m$  given private key  $sk$ . Let  $G$  be an algorithm that generates random numbers from raw entropy, i.e., the output of a CSPRNG. Let  $\text{tag}$  be a fixed, context-dependent string. Let  $\text{KDF}$  be a key derivation function, e.g., HKDF-Extract [RFC5869] (with first argument set to nil), that extracts a key of length  $L$  suitable for cryptographic use. Lastly, let  $H$  be a cryptographic hash function that produces output of length  $M$ .

The construction works as follows: instead of using  $x$  when randomness is needed, use:

```
PRF(KDF(G(x) || H(Sig(sk, tag1))), tag2)
```

Functionally, this computes the PRF of a string ( $\text{tag2}$ ) with a key derived from the CSPRNG output and signature over a fixed string ( $\text{tag1}$ ). See Section 3 for details about how " $\text{tag1}$ " and " $\text{tag2}$ " should be generated. The PRF behaves in a manner that is indistinguishable from a truly random function from  $\{0, 1\}^L$  to  $\{0, 1\}^M$  assuming the key is selected at random. Thus, the security of this construction depends upon the secrecy of  $H(\text{Sig}(sk, \text{tag1}))$  and  $G(x)$ . If the signature is leaked, then security reduces to the scenario wherein the PRF provides only a wrapper to  $G(x)$ .

In systems where signature computations are not cheap, these values may be precomputed in anticipation of future randomness requests. This is possible since the construction depends solely upon the CSPRNG output and private key.

$\text{Sig}(\text{sk}, \text{tag1})$  MUST NOT be used or exposed beyond its role in this computation. Moreover,  $\text{Sig}$  MUST be a deterministic signature function, e.g., deterministic ECDSA [RFC6979].

### 3. Tag Generation

Both tags SHOULD be generated such that they never collide with another accessor or owner of the private key. This can happen if, for example, one HSM with a private key is used from several servers, or if virtual machines are cloned.

To mitigate collisions, tag strings SHOULD be constructed as follows:

- o tag1: Constant string bound to a specific device and protocol in use. This allows caching of  $\text{Sig}(\text{sk}, \text{tag1})$ . Device specific information may include, for example, a MAC address. See Section 4 for example protocol information that can be used in the context of TLS 1.3.
- o tag2: Non-constant string that includes a timestamp or counter. This ensures change over time even if randomness were to repeat.

### 4. Application to TLS

The PRF randomness wrapper can be applied to any protocol wherein a party has a long-term private key and also generates randomness. This is true of most TLS servers. Thus, to apply this construction to TLS, one simply replaces the "private" PRNG, i.e., the PRNG that generates private values, such as key shares, with:

```
HKMAC(HKDF-Extract(nil, G(x) || Sig(sk, tag1)), tag2)
```

Moreover, we fix tag1 to protocol-specific information such as "TLS 1.3 Additional Entropy" for TLS 1.3. Older variants use similarly constructed strings.

### 5. IANA Considerations

This document makes no request to IANA.

### 6. Security Considerations

A security analysis was performed by two authors of this document. Generally speaking, security depends on keeping the private key secret. If this secret is compromised, the scheme reduces to the scenario wherein the PRF provides only an outer wrapper on usual CSPRNG generation.

The main reason one might expect the signature to be exposed is via a side-channel attack. It is therefore prudent when implementing this construction to take into consideration the extra long-term key operation if equipment is used in a hostile environment when such considerations are necessary.

The signature in the construction as well as in the protocol itself MUST be deterministic: if the signatures are probabilistic, then with weak entropy, our construction does not help and the signatures are still vulnerable due to repeat randomness attacks. In such an attack, the adversary might be able to recover the long-term key used in the signature.

Under these conditions, applying this construction should never yield worse security guarantees than not applying it assuming that applying the PRF does not reduce entropy. We believe there is always merit in analysing protocols specifically. However, this construction is generic so the analyses of many protocols will still hold even if this proposed construction is incorporated.

## 7. Normative References

### [DebianBug]

Yilek, Scott, et al, ., "When private keys are public - Results from the 2008 Debian OpenSSL vulnerability", n.d., <<https://pdfs.semanticscholar.org/fcf9/fe0946c20e936b507c023bbf89160cc995b9.pdf>>.

### [DualEC]

Bernstein, Daniel et al, ., "Dual EC - A standardized back door", n.d., <<https://projectbullrun.org/dual-ec/documents/dual-ec-20150731.pdf>>.

### [NAXOS]

LaMacchia, Brian et al, ., "Stronger Security of Authenticated Key Exchange", n.d., <<https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/strongake-submitted.pdf>>.

### [RFC2104]

Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <<https://www.rfc-editor.org/info/rfc2104>>.

### [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>>.

- [RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <<https://www.rfc-editor.org/info/rfc6979>>.
- [X9.62] American National Standards Institute, ., "Public Key Cryptography for the Financial Services Industry -- The Elliptic Curve Digital Signature Algorithm (ECDSA). ANSI X9.62-2005, November 2005.", n.d..

## Authors' Addresses

Cas Cremers  
University of Oxford  
Wolfson Building, Parks Road  
Oxford  
England

Email: [cas.cremers@cs.ox.ac.uk](mailto:cas.cremers@cs.ox.ac.uk)

Luke Garratt  
University of Oxford  
Wolfson Building, Parks Road  
Oxford  
England

Email: [luke.garratt@cs.ox.ac.uk](mailto:luke.garratt@cs.ox.ac.uk)

Stanislav Smyshlyaev  
CryptoPro  
18, Suschevsky val  
Moscow  
Russian Federation

Email: [svs@cryptopro.ru](mailto:svs@cryptopro.ru)

Nick Sullivan  
Cloudflare  
101 Townsend St  
San Francisco  
United States of America

Email: [nick@cloudflare.com](mailto:nick@cloudflare.com)



Christopher A. Wood  
Apple Inc.  
One Apple Park Way  
Cupertino, California 95014  
United States of America

Email: cawood@apple.com