Workgroup: Network Working Group
Internet-Draft: draft-irtf-cfrg-cpace-11
Published: 27 March 2024
Intended Status: Informational
Expires: 28 September 2024
Authors: M. Abdalla
DFINITY - Zurich
B. Haase
Endress + Hauser Liquid Analysis - Gerlingen
J. Hesse
IBM Research Europe - Zurich

# CPace, a balanced composable PAKE

## Abstract

This document describes CPace which is a protocol that allows two parties that share a low-entropy secret (password) to derive a strong shared key without disclosing the secret to offline dictionary attacks. The CPace protocol was tailored for constrained devices and can be used on groups of prime- and non-prime order.

## Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the Crypto Forum Research Group mailing list (cfrg@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=cfrg.

Source for this draft and an issue tracker can be found at https://github.com/cfrg/draft-irtf-cfrg-cpace.

## 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 28 September 2024.

## Copyright Notice



## Table of Contents

## 1. Introduction

This document describes CPace which is a balanced Password-Authenticated-Key-Establishment (PAKE) protocol for two parties where both parties derive a cryptographic key of high entropy from a shared secret of low-entropy. CPace protects the passwords against offline dictionary attacks by requiring adversaries to actively interact with a protocol party and by allowing for at most one single password guess per active interaction.

The CPace design was tailored considering the following main objectives:

* Efficiency: Deployment of CPace is feasible on resource-constrained devices.

* Versatility: CPace supports different application scenarios via versatile input formats, and by supporting applications with and without clear initiator and responder roles.

* Implementation error resistance: CPace aims at avoiding common implementation pitfalls already by-design, such as avoiding incentives for insecure execution-time speed optimizations. For smooth integration into different cryptographic library ecosystems, this document provides a variety of cipher suites.

* Post-quantum annoyance: CPace comes with mitigations with respect to adversaries that become capable of breaking the discrete logarithm problem on elliptic curves.

### 1.1. Outline of this document

* Section 3 describes the expected properties of an application using CPace, and discusses in particular which application-level aspects are relevant for CPace's security.

* Section 4 gives an overview of the recommended cipher suites for CPace which were optimized for different types of cryptographic library ecosystems.

* Section 5 introduces the notation used throughout this document.

* Section 6 specifies the CPace protocol.

* The final section provides explicit reference implementations and test vectors of all of the functions defined for CPace in the appendix.

As this document is primarily written for implementers and application designers, we would like to refer the theory-inclined reader to the scientific paper [AHH21] which covers the detailed security analysis of the different CPace instantiations as defined in this document via the cipher suites.

## 2. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

## 3. High-level application perspective

CPace enables balanced password-authenticated key establishment. CPace requires a shared secret octet string, the password-related string (PRS), is available for both parties A and B. PRS can be a low-entropy secret itself, for instance a clear-text password encoded according to [RFC8265], or any string derived from a common secret, for instance by use of a password-based key derivation function.

Applications with clients and servers where the server side is storing account and password information in its persistent memory are recommended to use augmented PAKE protocols such as OPAQUE [I-D.irtf-cfrg-opaque].

In the course of the CPace protocol, A sends one message MSGa to B and B sends one message MSGb to A. CPace does not mandate any ordering of these two messages. We use the term "initiator-responder" for CPace where A always speaks first, and the term "symmetric" setting where anyone can speak first.

CPace's output is an intermediate session key (ISK), but any party might abort in case of an invalid received message. A and B will produce the same ISK value only if both sides did initiate the

protocol using the same protocol inputs, specifically the same PRS string and the same value for the optional input parameters CI, ADa, ADb and sid that will be specified in the upcoming sections.

The naming of ISK key as "intermediate" session key highlights the fact that it is RECOMMENDED that applications process ISK by use of a suitable strong key derivation function KDF (such as defined in [RFC5869]) before using the key in a higher-level protocol.

### 3.1. Optional CPace inputs

For accomodating different application settings, CPace offers the following OPTIONAL inputs, i.e. inputs which MAY also be the empty string:

* Channel identifier (CI). CI can be used to bind a session key exchanged with CPace to a specific networking channel which interconnects the protocol parties. Both parties are required to have the same view of CI. CI will not be publicly sent on the wire and may also include confidential information.

* Associated data fields (ADa and ADb). These fields can be used to authenticate public associated data alongside the CPace protocol. The values ADa (and ADb, respectively) are guaranteed to be authenticated in case both parties agree on a key.

  ADa and ADb can for instance include party identities or protocol version information of an application protocol (e.g. to avoid downgrade attacks).

  If party identities are not encoded as part of CI, party identities SHOULD be included in ADa and ADb (see Section 9.1). In a setting with clear initiator and responder roles, identity information in ADa sent by the initiator can be used by the responder for choosing the right PRS string (respectively password) for this identity.

* Session identifier (sid). CPace comes with a security analysis [AHH21] in the framework of universal composability. This framework allows for modular analysis of a larger application protocol which uses CPace as a building block. For such analysis the CPace protocol is bound to a specific session of the larger protocol by use of a sid string that is globally unique. As a result, when used with a unique sid, CPace instances remain secure when running concurrently with other CPace instances, and even arbitrary other protocols.

  For this reason, it is RECOMMENDED that applications establish a unique session identifier sid prior to running the CPace protocol. This can be implemented by concatenating random bytes

produced by A with random bytes produced by B. If such preceding round is not an option but parties are assigned clear initiator-responder roles, it is RECOMMENDED to let the initiator A choose a fresh random sid and send it to B together with the first message. If a sid string is used it SHOULD HAVE a length of at least 8 bytes.

### 3.2. Responsibilities of the application layer

The following tasks are out of the scope of this document and left to the application layer

* Setup phase:
  - The application layer is responsible for the handshake that makes parties agree on a common CPace cipher suite.
  - The application layer needs to specify how to encode the CPace byte strings Ya / Yb and ADa / ADb defined in section Section 6 for transfer over the network. For CPace it is RECOMMENDED to encode network messages by using MSGa = lv_cat(Ya,ADa) and MSGb = lv_cat(Yb,ADb) using the length-value concatenation function lv_cat speficied in Section 5.3. This document provides test vectors for lv_cat-encoded messages. Alternative network encodings, e.g., the encoding method used for the client hello and server hello messages of the TLS protocol, MAY be used when considering the guidance given in Section 9.
* This document does not specify which encodings applications use for the mandatory PRS input and the optional inputs CI, sid, ADa and ADb. If PRS is a clear-text password or an octet string derived from a clear-text password, e.g. by use of a key-derivation function, the clear-text password SHOULD BE encoded according to [RFC8265].
* The application needs to settle whether CPace is used in the initiator-responder or the symmetric setting, as in the symmetric setting transcripts must be generated using ordered string concatenation. In this document we will provide test vectors for both, initiator-responder and symmetric settings.

## 4. CPace cipher suites

In the setup phase of CPace, both communication partners need to agree on a common cipher suite. Cipher suites consist of a combination of a hash function H and an elliptic curve environment G.

For naming cipher suites we use the convention "CPACE-G-H". We RECOMMEND the following cipher suites:

- CPACE-X25519-SHA512. This suite uses the group environment G_X25519 defined in Section 7.2 and SHA-512 as hash function. This cipher suite comes with the smallest messages on the wire and a low computational cost.

- CPACE-P256_XMD:SHA-256_SSWU_NU_-SHA256. This suite instantiates the group environment G as specified in Section 7.4 using the encode_to_curve function P256_XMD:SHA-256_SSWU_NU_ from [RFC9380] on curve NIST-P256, and hash function SHA-256.

The following RECOMMENDED cipher suites provide higher security margins.

- CPACE-X448-SHAKE256. This suite uses the group environment G_X448 defined in Section 7.2 and SHAKE-256 as hash function.

- CPACE-P384_XMD:SHA-384_SSWU_NU_-SHA384. This suite instantiates G as specified in Section 7.4 using the encode_to_curve function P384_XMD:SHA-384_SSWU_NU_ from [RFC9380] on curve NIST-P384 with H = SHA-384.

- CPACE-P521_XMD:SHA-512_SSWU_NU_-SHA512. This suite instantiates G as specified in Section 7.4 using the encode_to_curve function P521_XMD:SHA-512_SSWU_NU_ from [RFC9380] on curve NIST-P521 with H = SHA-512.

CPace can also securely be implemented using the cipher suites CPACE-RISTR255-SHA512 and CPACE-DECAF448-SHAKE256 defined in Section 7.3. Section 9 gives guidance on how to implement CPace on further elliptic curves.

## 5. Definitions and notation

### 5.1. Hash function H

Common choices for H are SHA-512 [RFC6234] or SHAKE-256 [FIPS202]. (I.e. the hash function outputs octet strings, and not group elements.) For considering both variable-output-length hashes and fixed-output-length hashes, we use the following convention. In case that the hash function is specified for a fixed-size output, we define H.hash(m,l) such that it returns the first l octets of the output.

We use the following notation for referring to the specific properties of a hash function H:

- H.hash(m,l) is a function that operates on an input octet string m and returns a hashing result of l octets.

- H.b_in_bytes denotes the minimum output size in bytes for collision resistance for the security level target of the hash function. E.g. H.b_in_bytes = 64 for SHA-512 and SHAKE-256 and H.b_in_bytes = 32 for SHA-256 and SHAKE-128. We use the notation H.hash(m) = H.hash(m, H.b_in_bytes) and let the hash operation output the default length if no explicit length parameter is given.

- H.bmax_in_bytes denotes the *maximum* output size in octets supported by the hash function. In case of fixed-size hashes such as SHA-256, this is the same as H.b_in_bytes, while there is no such limit for hash functions such as SHAKE-256.

- H.s_in_bytes denotes the *input block size* used by H. This number denotes the maximum number of bytes that can be processed in a single block before applying the compression function or permutation becomes necessary. (See also [RFC2104] for the corresponding block size concepts). For instance, for SHA-512 the input block size s_in_bytes is 128 as the compression function can process up to 128 bytes, while for SHAKE-256 the input block size amounts to 136 bytes before the permutation of the sponge state needs to be applied.

### 5.2. Group environment G

The group environment G specifies an elliptic curve group (also denoted G for convenience) and associated constants and functions as detailed below. In this document we use additive notation for the group operation.

- G.calculate_generator(H,PRS,CI,sid) denotes a function that outputs a representation of a generator (referred to as "generator" from now on) of the group which is derived from input octet strings PRS, CI, and sid and with the help of the hash function H.

- G.sample_scalar() is a function returning a representation of an integer (referred to as "scalar" from now on) appropriate as a private Diffie-Hellman key for the group.

- G.scalar_mult(y,g) is a function operating on a scalar y and a group element g. It returns an octet string representation of the group element Y = g*y.

*G.I denotes a unique octet string representation of the neutral element of the group. G.I is used for detecting and signaling certain error conditions.

*G.scalar_mult_vfy(y,g) is a function operating on a scalar y and a group element g. It returns an octet string representation of the group element g*y. Additionally, scalar_mult_vfy specifies validity conditions for y,g and g*y and outputs G.I in case they are not met.

*G.DSI denotes a domain-separation identifier octet string which SHALL be uniquely identifying the group environment G.

### 5.3. Notation for string operations

*bytes1 || bytes2 and denotes concatenation of octet strings.

*len(S) denotes the number of octets in an octet string S.

*nil denotes an empty octet string, i.e., len(nil) = 0.

*This document uses quotation marks "" both for general language (e.g. for citation of notation used in other documents) and as syntax for specifying octet strings as in b"CPace25519".

We use a preceeding lower-case letter b"" in front of the quotation marks if a character sequence is representing an octet string sequence. I.e. we use the notation for byte string representations with single-byte ASCII character encodings from the python programming language.

*prepend_len(octet_string) denotes the octet sequence that is obtained from prepending the length of the octet string to the string itself. The length shall be prepended by using an LEB128 encoding of the length. This will result in a single-byte encoding for values below 128. (Test vectors and reference implementations for prepend_len and the LEB128 encodings are given in the appendix.)

*lv_cat(a0,a1, ...) is the "length-value" encoding function which returns the concatenation of the input strings with an encoding of their respective length prepended. E.g. lv_cat(a0,a1) returns prepend_len(a0) || prepend_len(a1). The detailed specification of lv_cat and a reference implementations are given in the appendix.

*network_encode(Y,AD) denotes the function specified by the application layer that outputs an octet string encoding of the input octet strings Y and AD for transfer on the network. The implementation of MSG = network_encode(Y,AD) SHALL allow the receiver party to parse MSG for the individual subcomponents Y

and AD. For CPace we RECOMMEND to implement network_encode(Y,AD) as network_encode(Y,AD) = lv_cat(Y,AD).

Other encodings, such as the network encoding used for the client-hello and server-hello messages in TLS MAY also be used when following the guidance given in the security consideration section.

* sample_random_bytes(n) denotes a function that returns n octets, each of which is to be independently sampled from an uniform distribution between 0 and 255.

* zero_bytes(n) denotes a function that returns n octets with value 0.

* o_cat(bytes1,bytes2) denotes a function for ordered concatenation of octet strings. It places the lexiographically larger octet string first and prepends the two bytes from the octet string b"oc" to the result. (Explicit reference code for this function is given in the appendix.)

* transcript(MSGa,MSGb) denotes function outputing a string for the protocol transcript with messages MSGa and MSGb. In applications where CPace is used without clear initiator and responder roles, i.e. where the ordering of messages is not enforced by the protocol flow, transcript(MSGa,MSGb) = o_cat(MSGa,MSGb) SHALL be used. In the initiator-responder setting transcript(MSGa,MSGb) SHALL BE implemented such that the later message is appended to the earlier message, i.e., transcript(MSGa,MSGb) = MSGa||MSGb if MSGa is sent first.

### 5.4. Notation for group operations

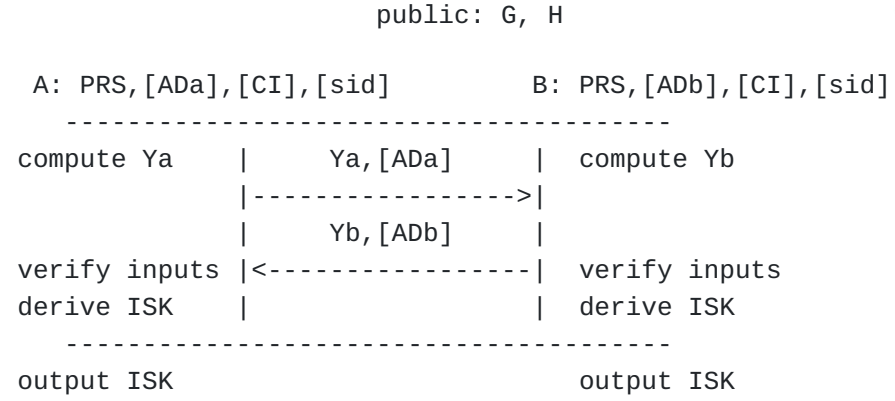
We use additive notation for the group, i.e., X*2 denotes the element that is obtained by computing X+X, for group element X and group operation +.

## 6. The CPace protocol

CPace is a one round protocol between two parties, A and B. At invocation, A and B are provisioned with PRS,G,H and OPTIONAL CI,sid,ADa (for A) and CI,sid,ADb (for B). A sends a message MSGa to B. MSGa contains the public share Ya and OPTIONAL associated data ADa (i.e. an ADa field that MAY have a length of 0 bytes). Likewise, B sends a message MSGb to A. MSGb contains the public share Yb and OPTIONAL associated data ADb (i.e. an ADb field that MAY have a length of 0 bytes). Both A and B use the received messages for deriving a shared intermediate session key, ISK.

### 6.1. Protocol flow

Optional parameters and messages are denoted with [].




### 6.2. CPace protocol instructions

A computes a generator g = G.calculate_generator(H,PRS,CI,sid), scalar ya = G.sample_scalar() and group element Ya = G.scalar_mult (ya,g). A then transmits MSGa = network_encode(Ya, ADa) with optional associated data ADa to B.

B computes a generator g = G.calculate_generator(H,PRS,CI,sid), scalar yb = G.sample_scalar() and group element Yb = G.scalar_mult(yb,g). B sends MSGb = network_encode(Yb, ADb) with optional associated data ADb to A.

Upon reception of MSGa, B checks that MSGa was properly generated in conformity with the chosen encoding of network messages (notably correct length fields). If this parsing fails, then B MUST abort. (Testvectors of examples for invalid messages when using lv_cat() as network_encode function for CPace are given in the appendix.) B then computes K = G.scalar_mult_vfy(yb,Ya). B MUST abort if K=G.I. Otherwise B calculates ISK = H.hash(lv_cat(G.DSI || b"_ISK", sid, K)||transcript(MSGa, MSGb)). B returns ISK and terminates.

Likewise upon reception of MSGb, A parses MSGb for Yb and ADb and checks for a valid encoding. If this parsing fails, then A MUST abort. A then computes K = G.scalar_mult_vfy(ya,Yb). A MUST abort if K=G.I. Otherwise A calculates ISK = H.hash(lv_cat(G.DSI || b"_ISK", sid, K) || transcript(MSGa, MSGb)). A returns ISK and terminates.

The session key ISK returned by A and B is identical if and only if the supplied input parameters PRS, CI and sid match on both sides and transcript view (containing of MSGa and MSGb) of both parties match.

(Note that in case of a symmetric protocol execution without clear initiator/responder roles, transcript(MSGa, MSGb) needs to be implemented using ordered concatenation for generating a matching view by both parties.)

## 7. Implementation of recommended CPace cipher suites

### 7.1. Common function for computing generators

The different cipher suites for CPace defined in the upcoming sections share the same method for deterministically combining the individual strings PRS, CI, sid and the domain-separation identifier DSI to a generator string that we describe here.

- generator_string(DSI, PRS, CI, sid, s_in_bytes) denotes a function that returns the string lv_cat(DSI, PRS, zero_bytes(len_zpad), CI, sid).

- len_zpad = MAX(0, s_in_bytes - len(prepend_len(PRS)) - len(prepend_len(G.DSI)) - 1)

The zero padding of length len_zpad is designed such that the encoding of DSI and PRS together with the zero padding field completely fills at least the first input block (of length s_in_bytes) of the hash. As a result for the common case of short PRS the number of bytes to hash becomes independent of the actual length of the password (PRS). (A reference implementation and test vectors are provided in the appendix.)

The introduction of a zero-padding within the generator string also helps mitigating attacks of a side-channel adversary that analyzes correlations between publicly known variable information with a short low-entropy PRS string. Note that the hash of the first block is intentionally made independent of session-specific inputs, such as sid or CI and that there is no limitation regarding the maximum length of the PRS string.

### 7.2. CPace group objects G_X25519 and G_X448 for single-coordinate Ladders on Montgomery curves

In this section we consider the case of CPace when using the X25519 and X448 Diffie-Hellman functions from [RFC7748] operating on the Montgomery curves Curve25519 and Curve448 [RFC7748]. CPace implementations using single-coordinate ladders on further Montgomery curves SHALL use the definitions in line with the specifications for X25519 and X448 and review the guidance given in Section 9.

For the group environment G_X25519 the following definitions apply:

- *G_X25519.field_size_bytes = 32

- *G_X25519.field_size_bits = 255

- *G_X25519.sample_scalar() =
  sample_random_bytes(G.field_size_bytes)

- *G_X25519.scalar_mult(y,g) = G.scalar_mult_vfy(y,g) = X25519(y,g)

- *G_X25519.I = zero_bytes(G.field_size_bytes)

- *G_X25519.DSI = b"CPace255"

CPace cipher suites using G_X25519 MUST use a hash function producing at least H.b_max_in_bytes >= 32 bytes of output. It is RECOMMENDED to use G_X25519 in combination with SHA-512.

For X448 the following definitions apply:

- *G_X448.field_size_bytes = 56

- *G_X448.field_size_bits = 448

- *G_X448.sample_scalar() = sample_random_bytes(G.field_size_bytes)

- *G_X448.scalar_mult(y,g) = G.scalar_mult_vfy(y,g) = X448(y,g)

- *G_X448.I = zero_bytes(G.field_size_bytes)

- *G_X448.DSI = b"CPace448"

CPace cipher suites using G_X448 MUST use a hash function producing at least H.b_max_in_bytes >= 56 bytes of output. It is RECOMMENDED to use G_X448 in combination with SHAKE-256.

For both G_X448 and G_X25519 the G.calculate_generator(H, PRS,sid,CI) function shall be implemented as follows.

- *First gen_str = generator_string(G.DSI,PRS,CI,sid, H.s_in_bytes) SHALL BE calculated using the input block size of the chosen hash function.

- *This string SHALL then BE hashed to the required length gen_str_hash = H.hash(gen_str, G.field_size_bytes). Note that this implies that the permissible output length H.maxb_in_bytes MUST BE larger or equal to the field size of the group G for making a hashing function suitable.

*This result is then considered as a field coordinate using the u = decodeUCoordinate(gen_str_hash, G.field_size_bits) function from [RFC7748] which we repeat in the appendix for convenience.

*The result point g is then calculated as (g,v) = map_to_curve_elligator2(u) using the function from [RFC9380]. Note that the v coordinate produced by the map_to_curve_elligator2 function is not required for CPace and discarded. The appendix repeats the definitions from [RFC9380] for convenience.

In the appendix we show sage code that can be used as reference implementation.

#### 7.2.1. Verification tests

For single-coordinate Montgomery ladders on Montgomery curves verification tests according to Section 8 SHALL check for proper handling of the abort conditions, when a party is receiving u coordinate values that encode a low-order point on either the curve or the quadratic twist.

In addition to that in case of G_X25519 the tests SHALL also verify that the implementation of G.scalar_mult_vfy(y,g) produces the expected results for non-canonical u coordinate values with bit #255 set, which may also encode low-order points.

Corresponding test vectors are provided in the appendix.

### 7.3. CPace group objects G_Ristretto255 and G_Decaf448 for prime-order group abstractions

In this section we consider the case of CPace using the Ristretto255 and Decaf448 group abstractions [I-D.draft-irtf-cfrg-ristretto255-decaf448]. These abstractions define an encode and decode function, group operations using an internal encoding and an element-derivation function that maps a byte string to a group element. With the group abstractions there is a distinction between an internal representation of group elements and an external encoding of the same group element. In order to distinguish between these different representations, we prepend an underscore before values using the internal representation within this section.

For Ristretto255 the following definitions apply:

*G_Ristretto255.DSI = b"CPaceRistretto255"

*G_Ristretto255.field_size_bytes = 32

  *G_Ristretto255.group_size_bits = 252

  *G_Ristretto255.group_order = 2^252 +
   27742317777372353535851937790883648493

CPace cipher suites using G_Ristretto255 MUST use a hash function producing at least H.b_max_in_bytes >= 64 bytes of output. It is RECOMMENDED to use G_Ristretto255 in combination with SHA-512.

For decaf448 the following definitions apply:

  *G_Decaf448.DSI = b"CPaceDecaf448"

  *G_Decaf448.field_size_bytes = 56

  *G_Decaf448.group_size_bits = 445

  *G_Decaf448.group_order = l = 2^446 -
   13818066809895115352007386748515426880336692474882178609894547503885

CPace cipher suites using G_Decaf448 MUST use a hash function producing at least H.b_max_in_bytes >= 112 bytes of output. It is RECOMMENDED to use G_Decaf448 in combination with SHAKE-256.

For both abstractions the following definitions apply:

  *It is RECOMMENDED to implement G.sample_scalar() as follows.

     -Set scalar = sample_random_bytes(G.group_size_bytes).

     -Then clear the most significant bits larger than
      G.group_size_bits.

     -Interpret the result as the little-endian encoding of an
      integer value and return the result.

  *Alternatively, if G.sample_scalar() is not implemented according to the above recommendation, it SHALL be implemented using uniform sampling between 1 and (G.group_order - 1). Note that the more complex uniform sampling process can provide a larger side-channel attack surface for embedded systems in hostile environments.

  *G.scalar_mult(y,_g) SHALL operate on a scalar y and a group element _g in the internal representation of the group abstraction environment. It returns the value Y = encode((_g) * y), i.e. it returns a value using the public encoding.

*G.I = is the public encoding representation of the identity element.

*G.scalar_mult_vfy(y,X) operates on a value using the public encoding and a scalar and is implemented as follows. If the decode(X) function fails, it returns G.I. Otherwise it returns encode( decode(X) * y ).

*The G.calculate_generator(H, PRS,sid,CI) function SHALL return a decoded point and SHALL BE implemented as follows.

-First gen_str = generator_string(G.DSI,PRS,CI,sid, H.s_in_bytes) is calculated using the input block size of the chosen hash function.

-This string is then hashed to the required length gen_str_hash = H.hash(gen_str, 2 * G.field_size_bytes). Note that this implies that the permissible output length H.maxb_in_bytes MUST BE larger or equal to twice the field size of the group G for making a hash function suitable.

-Finally the internal representation of the generator _g is calculated as _g = element_derivation(gen_str_hash) using the element derivation function from the abstraction.

Note that with these definitions the scalar_mult function operates on a decoded point _g and returns an encoded point, while the scalar_mult_vfy(y,X) function operates on an encoded point X (and also returns an encoded point).

#### 7.3.1. Verification tests

For group abstractions verification tests according to Section 8 SHALL check for proper handling of the abort conditions, when a party is receiving encodings of the neutral element or receives an octet string that does not decode to a valid group element.

### 7.4. CPace group objects for curves in Short-Weierstrass representation

The group environment objects G defined in this section for use with Short-Weierstrass curves, are parametrized by the choice of an elliptic curve and by choice of a suitable encode_to_curve function. encode_to_curve must map an octet string to a point on the curve.

#### 7.4.1. Curves and associated functions

Elliptic curves in Short-Weierstrass form are considered in [IEEE1363]. [IEEE1363] allows for both, curves of prime and non-

prime order. However, for the procedures described in this section any suitable group MUST BE of prime order.

The specification for the group environment objects specified in this section closely follow the ECKAS-DH1 method from [IEEE1363]. I.e. we use the same methods and encodings and protocol substeps as employed in the TLS [RFC5246] [RFC8446] protocol family.

For CPace only the uncompressed full-coordinate encodings from [SEC1] (x and y coordinate) SHOULD be used. Commonly used curve groups are specified in [SEC2] and [RFC5639]. A typical representative of such a Short-Weierstrass curve is NIST-P256. Point verification as used in ECKAS-DH1 is described in Annex A.16.10. of [IEEE1363].

For deriving Diffie-Hellman shared secrets ECKAS-DH1 from [IEEE1363] specifies the use of an ECSVDP-DH method. We use ECSVDP-DH in combination with the identy map such that it either returns "error" or the x-coordinate of the Diffie-Hellman result point as shared secret in big endian format (fixed length output by FE2OSP without truncating leading zeros).

#### 7.4.2. Suitable encode_to_curve methods

All the encode_to_curve methods specified in [RFC9380] are suitable for CPace. For Short-Weierstrass curves it is RECOMMENDED to use the non-uniform variant of the SSWU mapping primitive from [RFC9380] if a SSWU mapping is available for the chosen curve. (We recommend non-uniform maps in order to give implementations the flexibility to opt for x-coordinate-only scalar multiplication algorithms.)

#### 7.4.3. Definition of the group environment G for Short-Weierstrass curves

In this paragraph we use the following notation for defining the group object G for a selected curve and encode_to_curve method:

- With G.group_order we denote the order of the elliptic curve which MUST BE a prime.

- With is_valid(X) we denote a method which operates on an octet stream according to [SEC1] of a point on the group and returns true if the point is valid and returns false otherwise. This is_valid(X) method SHALL be implemented according to Annex A. 16.10. of [IEEE1363]. I.e. it shall return false if X encodes either the neutral element on the group or does not form a valid encoding of a point on the group.

- With encode_to_curve(str,DST) we denote a mapping function from [RFC9380]. I.e. a function that maps octet string str to a point

on the group using the domain separation tag DST. [RFC9380] considers both, uniform and non-uniform mappings based on several different strategies. It is RECOMMENDED to use the nonuniform variant of the SSWU mapping primitive within [RFC9380].

* G.DSI denotes a domain-separation identifier octet string. G.DSI which SHALL BE obtained by the concatenation of b"CPace" and the associated name of the cipher suite used for the encode_to_curve function as specified in [RFC9380]. E.g. when using the map with the name P384_XMD:SHA-384_SSWU_NU_ on curve NIST-P384 the resulting value SHALL BE G.DSI = b"CPaceP384_XMD:SHA-384_SSWU_NU_".

Using the above definitions, the CPace functions required for the group object G are defined as follows.

* G.DST denotes the domain-separation tag value to use in conjunction with the encode_to_curve function from [RFC9380]. G.DST shall be obtained by concatenating G.DSI and b"_DST".

* G.sample_scalar() SHALL return a value between 1 and (G.group_order - 1). The sampling SHALL BE indistinguishable from uniform random selection between 1 and (G.group_order - 1). It is RECOMMENDED to use a constant-time rejection sampling algorithm for converting a uniform bitstring to a uniform value between 1 and (G.group_order - 1).

* G.calculate_generator(H, PRS,sid,CI) function SHALL be implemented as follows.

  - First gen_str = generator_string(G.DSI,PRS,CI,sid, H.s_in_bytes) is calculated.

  - Then the output of a call to encode_to_curve(gen_str, G.DST) is returned, using the selected suite from [RFC9380].

* G.scalar_mult(s,X) is a function that operates on a scalar s and an input point X. The input X shall use the same encoding as produced by the G.calculate_generator method above. G.scalar_mult(s,X) SHALL return an encoding of either the point X*s or the point X*(-s) according to [SEC1]. Implementations SHOULD use the full-coordinate format without compression, as important protocols such as TLS 1.3 removed support for compression. Implementations of scalar_mult(s,X) MAY output either X*s or X*(-s) as both points X*s and X*(-s) have the same x-coordinate and result in the same Diffie-Hellman shared secrets K. (This allows implementations to opt for x-coordinate-only scalar multiplication algorithms.)

*G.scalar_mult_vfy(s,X) merges verification of point X according to [IEEE1363] A.16.10. and the the ECSVDP-DH procedure from [IEEE1363]. It SHALL BE implemented as follows:

-If is_valid(X) = False then G.scalar_mult_vfy(s,X) SHALL return "error" as specified in [IEEE1363] A.16.10 and 7.2.1.

-Otherwise G.scalar_mult_vfy(s,X) SHALL return the result of the ECSVDP-DH procedure from [IEEE1363] (section 7.2.1). I.e. it shall either return "error" (in case that X*s is the neutral element) or the secret shared value "z" (otherwise). "z" SHALL be encoded by using the big-endian encoding of the x-coordinate of the result point X*s according to [SEC1].

*We represent the neutral element G.I by using the representation of the "error" result case from [IEEE1363] as used in the G.scalar_mult_vfy method above.

#### 7.4.4. Verification tests

For Short-Weierstrass curves verification tests according to Section 8 SHALL check for proper handling of the abort conditions, when a party is receiving an encoding of the point at infinity and an encoding of a point not on the group.

## 8. Implementation verification

Any CPace implementation MUST be tested against invalid or weak point attacks. Implementation MUST be verified to abort upon conditions where G.scalar_mult_vfy functions outputs G.I. For testing an implementation it is RECOMMENDED to include weak or invalid point encodings within MSGa and MSGb and introduce this in a protocol run. It SHALL be verified that the abort condition is properly handled.

Moreover regarding the network format any implementation MUST be tested with respect to invalid encodings of MSGa and MSGb. E.g. when lv_cat is used as network format for encoding MSGa and MSGb, the sum of the prepended lengths of the fields must be verified to match the actual length of the message. Tests SHALL verify that a party aborts in case that incorrectly encoded messages are received.

Corresponding test vectors are given in the appendix for all recommended cipher suites.

## 9. Security Considerations

A security proof of CPace is found in [AHH21]. This proof covers all recommended cipher suites included in this document. In the following sections we describe how to protect CPace against several

attack families, such as relay-, length extension- or side channel attacks. We also describe aspects to consider when deviating from recommended cipher suites.

### 9.1. Party identifiers and relay attacks

If unique strings identifying the protocol partners are included either as part of the channel identifier CI, the session id sid or the associated data fields ADa, ADb, the ISK will provide implicit authentication also regarding the party identities. Incorporating party identifier strings is important for fending off relay attacks. Such attacks become relevant in a setting where several parties, say, A, B and C, share the same password PRS. An adversary might relay messages from a honest user A, who aims at interacting with user B, to a party C instead. If no party identifier strings are used, and B and C use the same PRS value, A might be establishing a common ISK key with C while assuming to interact with party B. Including and checking party identifiers can fend off such relay attacks.

### 9.2. Network message encoding and hashing protocol transcripts

It is RECOMMENDED to encode the (Ya,ADa) and (Yb,ADb) fields on the network by using network_encode(Y,AD) = lv_cat(Y,AD). I.e. we RECOMMEND to prepend an encoding of the length of the subfields. Prepending the length of all variable-size input strings results in a so-called prefix-free encoding of transcript strings, using terminology introduced in [CDMP05]. This property allows for disregarding length-extension imperfections that come with the commonly used Merkle-Damgard hash function constructions such as SHA256 and SHA512.

Other alternative network encoding formats which prepend an encoding of the length of variable-size data fields in the protocol messages are equally suitable. This includes, e.g., the type-length-value format specified in the DER encoding standard (X.690) or the protocol message encoding used in the TLS protocol family for the TLS client-hello or server-hello messages.

In case that an application uses another form of network message encoding which is not prefix-free, the guidance given in [CDMP05] SHOULD BE considered (e.g. by replacing hash functions with the HMAC constructions from[RFC2104]).

### 9.3. Key derivation

Although already K is a shared value, it MUST NOT itself be used as an application key. Instead, ISK MUST BE used. Leakage of K to an adversary can lead to offline dictionary attacks.

As noted already in Section 6 it is RECOMMENDED to process ISK by use of a suitable strong key derivation function KDF (such as defined in [RFC5869]) first, before using the key in a higher-level protocol.

### 9.4. Key confirmation

In many applications it is advisable to add an explicit key confirmation round after the CPace protocol flow. However, as some applications might only require implicit authentication and as explicit authentication messages are already a built-in feature in many higher-level protocols (e.g. TLS 1.3) the CPace protocol described here does not mandate use of a key confirmation on the level of the CPace sub-protocol.

Already without explicit key confirmation, CPace enjoys weak forward security under the sCDH and sSDH assumptions [AHH21]. With added explicit confirmation, CPace enjoys perfect forward security also under the strong sCDH and sSDH assumptions [AHH21].

Note that in [ABKLX21] it was shown that an idealized variant of CPace also enjoys perfect forward security without explicit key confirmation. However this proof does not explicitly cover the recommended cipher suites in this document and requires the stronger assumption of an algebraic adversary model. For this reason, we recommend adding explicit key confirmation if perfect forward security is required.

When implementing explicit key confirmation, it is recommended to use an appropriate message-authentication code (MAC) such as HMAC [RFC2104] or CMAC [RFC4493] using a key mac_key derived from ISK.

One suitable option that works also in the parallel setting without message ordering is to proceed as follows.

* First calculate mac_key as mac_key = H.hash(b"CPaceMac" || ISK).

* Then let each party send an authenticator tag Ta, Tb that is calculated over the protocol message that it has sent previously. I.e. let party A calculate its transmitted authentication code Ta as Ta = MAC(mac_key, MSGa) and let party B calculate its transmitted authentication code Tb as Tb = MAC(mac_key, MSGb).

* Let the receiving party check the remote authentication tag for the correct value and abort in case that it's incorrect.

### 9.5. Sampling of scalars

For curves over fields F_q where q is a prime close to a power of two, we recommend sampling scalars as a uniform bit string of length

field_size_bits. We do so in order to reduce both, complexity of the implementation and the attack surface with respect to side-channels for embedded systems in hostile environments. The effect of non-uniform sampling on security was demonstrated to be begnin in [AHH21] for the case of Curve25519 and Curve448. This analysis however does not transfer to most curves in Short-Weierstrass form.

As a result, we recommend rejection sampling if G is as in Section 7.4. Alternatively an algorithm designed allong the lines of the hash_to_field() function from [RFC9380] can also be used. There oversampling to an integer significantly larger than the curve order is followed by a modular reduction to the group order.

### 9.6. Preconditions for using the simplified CPace specification from Section 7.2

The security of the algorithms used for the recommended cipher suites for the Montgomery curves Curve25519 and Curve448 in Section 7.2 rely on the following properties [AHH21]:

- The curve has order (p * c) with p prime and c a small cofactor. Also the curve's quadratic twist must be of order (p' * c') with p' prime and c' a cofactor.

- The cofactor c of the curve MUST BE EQUAL to or an integer multiple of the cofactor c' of the curve's quadratic twist. Also, importantly, the implementation of the scalar_mult and scalar_mult_vfy functions must ensure that all scalars actually used for the group operation are integer multiples of c (e.g. such as asserted by the specification of the decodeScalar functions in [RFC7748]).

- Both field order q and group order p MUST BE close to a power of two along the lines of [AHH21], Appendix E. Otherwise the simplified scalar sampling specified in Section 7.2 needs to be changed.

- The representation of the neutral element G.I MUST BE the same for both, the curve and its twist.

- The implementation of G.scalar_mult_vfy(y,X) MUST map all c low-order points on the curve and all c' low-order points on the twist to G.I.

Algorithms for curves other than the ones recommended here can be based on the principles from Section 7.2 given that the above properties hold.

### 9.7. Nonce values

Secret scalars ya and yb MUST NOT be reused. Values for sid SHOULD NOT be reused since the composability guarantees established by the simulation-based proof rely on the uniqueness of session ids [AHH21].

If CPace is used in a concurrent system, it is RECOMMENDED that a unique sid is generated by the higher-level protocol and passed to CPace. One suitable option is that sid is generated by concatenating ephemeral random strings contributed by both parties.

### 9.8. Side channel attacks

All state-of-the art methods for realizing constant-time execution SHOULD be used. Special care is RECOMMENDED specifically for elliptic curves in Short-Weierstrass form as important standard documents including [IEEE1363] describe curve operations with non-constant-time algorithms.

In case that side channel attacks are to be considered practical for a given application, it is RECOMMENDED to pay special attention on computing the secret generator G.calculate_generator(PRS,CI,sid). The most critical substep to consider might be the processing of the first block of the hash that includes the PRS string. The zero-padding introduced when hashing the sensitive PRS string can be expected to make the task for a side-channel attack somewhat more complex. Still this feature alone is not sufficient for ruling out power analysis attacks.

Even though the calculate_generator operation might be considered to form the primary target for side-channel attacks as information on long-term secrets might be exposed, also the subsequent operations on ephemeral values, such as scalar sampling and scalar multiplication should be protected from side-channels.

### 9.9. Quantum computers

CPace is proven secure under the hardness of the strong computational Simultaneous Diffie-Hellmann (sSDH) and strong computational Diffie-Hellmann (sCDH) assumptions in the group G (as defined in [AHH21]). These assumptions are not expected to hold any longer when large-scale quantum computers (LSQC) are available. Still, even in case that LSQC emerge, it is reasonable to assume that discrete-logarithm computations will remain costly. CPace with ephemeral session id values sid forces the adversary to solve one computational Diffie-Hellman problem per password guess [ES21]. In this sense, using the wording suggested by Steve Thomas on the CFRG mailing list, CPace is "quantum-annoying".

## 10. IANA Considerations

No IANA action is required.

## 11. Acknowledgements

We would like to thank the participants on the CFRG list for comments and advice. Any comment and advice is appreciated.

## 12. References

### 12.1. Normative References

**[I-D.draft-irtf-cfrg-ristretto255-decaf448]** de Valence, H., Grigg, J., Hamburg, M., Lovecruft, I., Tankersley, G., and F. Valsorda, "The ristretto255 and decaf448 Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-ristretto255-decaf448-08, 5 September 2023, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-ristretto255-decaf448-08>.

**[I-D.irtf-cfrg-opaque]** Bourdrez, D., Krawczyk, H., Lewi, K., and C. A. Wood, "The OPAQUE Augmented PAKE Protocol", Work in Progress, Internet-Draft, draft-irtf-cfrg-opaque-14, 24 March 2024, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-opaque-14>.

**[IEEE1363]** "Standard Specifications for Public Key Cryptography, IEEE 1363", 2000.

**[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/rfc/rfc2119>.

**[RFC7748]** Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/rfc/rfc7748>.

**[RFC8174]** Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

**[SEC1]** Standards for Efficient Cryptography Group (SECG), "SEC 1: Elliptic Curve Cryptography", May 2009, <http://www.secg.org/sec1-v2.pdf>.

### 12.2. Informative References

**[ABKLX21]**

Abdalla, M., Barbosa, M., Katz, J., Loss, J., and J. Xu, "Algebraic Adversaries in the Universal Composability Framework.", n.d., <https://eprint.iacr.org/2021/1218>.

**[AHH21]** Abdalla, M., Haase, B., and J. Hesse, "Security analysis of CPace", n.d., <https://eprint.iacr.org/2021/114>.

**[CDMP05]** Coron, J.-S., Dodis, Y., Malinaud, C., and P. Puniya, "Merkle-Damgaard Revisited: How to Construct a Hash Function", In Advances in Cryptology - CRYPTO 2005, pages 430-448, DOI 10.1007/11535218_26, 2005, <https://doi.org/10.1007/11535218_26>.

**[ES21]** Eaton, E. and D. Stebila, "The 'quantum annoying' property of password-authenticated key exchange protocols.", n.d., <https://eprint.iacr.org/2021/696>.

**[FIPS202]** National Institute of Standards and Technology (NIST), "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", August 2015, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.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/rfc/rfc2104>.

**[RFC4493]** Song, JH., Poovendran, R., Lee, J., and T. Iwata, "The AES-CMAC Algorithm", RFC 4493, DOI 10.17487/RFC4493, June 2006, <https://www.rfc-editor.org/rfc/rfc4493>.

**[RFC5246]** Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/rfc/rfc5246>.

**[RFC5639]** Lochter, M. and J. Merkle, "Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation", RFC 5639, DOI 10.17487/RFC5639, March 2010, <https://www.rfc-editor.org/rfc/rfc5639>.

**[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/rfc/rfc5869>.

**[RFC6234]** Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/rfc/rfc6234>.

**[RFC8265]** Saint-Andre, P. and A. Melnikov, "Preparation, Enforcement, and Comparison of Internationalized Strings Representing Usernames and Passwords", RFC 8265, DOI 10.17487/RFC8265, October 2017, <https://www.rfc-editor.org/rfc/rfc8265>.

**[RFC8446]** Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

**[RFC9380]** Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", RFC 9380, DOI 10.17487/RFC9380, August 2023, <https://www.rfc-editor.org/rfc/rfc9380>.

**[SEC2]** Standards for Efficient Cryptography Group (SECG), "SEC 2: Recommended Elliptic Curve Domain Parameters", January 2010, <http://www.secg.org/sec2-v2.pdf>.

## Appendix A. CPace function definitions

### A.1. Definition and test vectors for string utility functions

#### A.1.1. prepend_len function

```
def prepend_len(data):
    "prepend LEB128 encoding of length"
    length = len(data)
    length_encoded = b""
    while True:
        if length < 128:
            length_encoded += bytes([length])
        else:
            length_encoded += bytes([(length & 0x7f) + 0x80])
        length = int(length >> 7)
        if length == 0:
            break;
    return length_encoded + data
```

### A.1.2. prepend_len test vectors

```
prepend_len(b""): (length: 1 bytes)
  00
prepend_len(b"1234"): (length: 5 bytes)
  0431323334
prepend_len(bytes(range(127))): (length: 128 bytes)
  7f000102030405060708090a0b0c0d0e0f101112131415161718191a1b
  1c1d1e1f202122232425262728292a2b2c2d2e2f303132333435363738
  393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f505152535455
  565758595a5b5c5d5e5f606162636465666768696a6b6c6d6e6f707172
  737475767778797a7b7c7d7e
prepend_len(bytes(range(128))): (length: 130 bytes)
  8001000102030405060708090a0b0c0d0e0f101112131415161718191a
  1b1c1d1e1f202122232425262728292a2b2c2d2e2f3031323334353637
  38393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f5051525354
  55565758595a5b5c5d5e5f606162636465666768696a6b6c6d6e6f7071
  72737475767778797a7b7c7d7e7f
```

### A.1.3. lv_cat function

```
def lv_cat(*args):
    result = b""
    for arg in args:
        result += prepend_len(arg)
    return result
```

### A.1.4. Testvector for lv_cat()

```
lv_cat(b"1234",b"5",b"",b"6789"): (length: 13 bytes)
  043132333401350004363738 39
```

### A.1.5. Examples for messages not obtained from a lv_cat-based encoding

The following messages are examples which have invalid encoded length fields. I.e. they are examples where parsing for the sum of the length of subfields as expected for a message generated using lv_cat(Y,AD) does not give the correct length of the message. Parties MUST abort upon reception of such invalid messages as MSGa or MSGb.

```
Inv_MSG1 not encoded by lv_cat: (length: 3 bytes)
  ffffff
Inv_MSG2 not encoded by lv_cat: (length: 3 bytes)
  ffff03
Inv_MSG3 not encoded by lv_cat: (length: 4 bytes)
  00ffff03
Inv_MSG4 not encoded by lv_cat: (length: 4 bytes)
  00ffffff
```

### A.2. Definition of generator_string function.

```
def generator_string(DSI,PRS,CI,sid,s_in_bytes):
    # Concat all input fields with prepended length information.
    # Add zero padding in the first hash block after DSI and PRS.
    len_zpad = max(0,s_in_bytes - 1 - len(prepend_len(PRS))
                    - len(prepend_len(DSI)))
    return lv_cat(DSI, PRS, zero_bytes(len_zpad),
                          CI, sid)
```

### A.3. Definitions and test vector ordered concatenation

#### A.3.1. Definitions for lexiographical ordering

For ordered concatenation lexiographical ordering of byte sequences is used:

```
def lexiographically_larger(bytes1,bytes2):
   "Returns True if bytes1 > bytes2 using lexiographical ordering."
   min_len = min (len(bytes1), len(bytes2))
   for m in range(min_len):
       if bytes1[m] > bytes2[m]:
           return True;
       elif bytes1[m] < bytes2[m]:
           return False;
   return len(bytes1) > len(bytes2)
```

#### A.3.2. Definitions for ordered concatenation

With the above definition of lexiographical ordering ordered concatenation is specified as follows.

```
def o_cat(bytes1,bytes2):
    if lexiographically_larger(bytes1,bytes2):
        return b"oc" + bytes1 + bytes2
    else:
        return b"oc" + bytes2 + bytes1
```

### A.3.3. Test vectors ordered concatenation

```
  string comparison for o_cat:
    lexiographically_larger(b"\0", b"\0\0") == False
    lexiographically_larger(b"\1", b"\0\0") == True
    lexiographically_larger(b"\0\0", b"\0") == True
    lexiographically_larger(b"\0\0", b"\1") == False
    lexiographically_larger(b"\0\1", b"\1") == False
    lexiographically_larger(b"ABCD", b"BCD") == False

  o_cat(b"ABCD",b"BCD"): (length: 9 bytes)
    6f6342434441424344
  o_cat(b"BCD",b"ABCDE"): (length: 10 bytes)
    6f634243444142434445
```

## A.4. Decoding and Encoding functions according to RFC7748

```
   def decodeLittleEndian(b, bits):
       return sum([b[i] << 8*i for i in range((bits+7)/8)])

   def decodeUCoordinate(u, bits):
       u_list = [ord(b) for b in u]
       # Ignore any unused bits.
       if bits % 8:
           u_list[-1] &= (1<<(bits%8))-1
       return decodeLittleEndian(u_list, bits)

   def encodeUCoordinate(u, bits):
       return ''.join([chr((u >> 8*i) & 0xff)
                       for i in range((bits+7)/8)])
```

## A.5. Elligator 2 reference implementation

The Elligator 2 map requires a non-square field element Z which shall be calculated as follows.

```
    def find_z_ell2(F):
        # Find nonsquare for Elligator2
        # Argument: F, a field object, e.g., F = GF(2^255 - 19)
        ctr = F.gen()
        while True:
            for Z_cand in (F(ctr), F(-ctr)):
                # Z must be a non-square in F.
                if is_square(Z_cand):
                    continue
                return Z_cand
            ctr += 1
```

The values of the non-square Z only depend on the curve. The algorithm above results in a value of Z = 2 for Curve25519 and Z=-1 for Ed448.

The following code maps a field element r to an encoded field element which is a valid u-coordinate of a Montgomery curve with curve parameter A.

```
 def elligator2(r, q, A, field_size_bits):
     # Inputs: field element r, field order q,
     #         curve parameter A and field size in bits
     Fq = GF(q); A = Fq(A); B = Fq(1);

     # get non-square z as specified in the hash2curve draft.
     z = Fq(find_z_ell2(Fq))
     powerForLegendreSymbol = floor((q-1)/2)

     v = - A / (1 + z * r^2)
     epsilon = (v^3 + A * v^2 + B * v)^powerForLegendreSymbol
     x = epsilon * v - (1 - epsilon) * A/2
     return encodeUCoordinate(Integer(x), field_size_bits)
```

## Appendix B. Test vectors

### B.1. Test vector for CPace using group X25519 and hash SHA-512

#### B.1.1. Test vectors for calculate_generator with group X25519

```
  Inputs
    H   = SHA-512 with input block size 128 bytes.
    PRS = b'Password' ; ZPAD length: 109 ; DSI = b'CPace255'
    CI = b'\nAinitiator\nBresponder'
    CI = 0a41696e69746961746f720a42726573706f6e646572
    sid = 7e4b4791d6a8ef019b936c79fb7f2c57
  Outputs
    generator_string(G.DSI,PRS,CI,sid,H.s_in_bytes):
    (length: 168 bytes)
      0843506163653235350850617373776f72646d00000000000000000
      0000000000000000000000000000000000000000000000000000000
      0000000000000000000000000000000000000000000000000000000
      0000000000000000000000000000000000000000000000000000000
      0000000000000000000000000000000160a41696e69746961746f72
      0a42726573706f6e646572107e4b4791d6a8ef019b936c79fb7f2c57
    hash generator string: (length: 32 bytes)
      10047198e8c4cacf0ab8a6d0ac337b8ae497209d042f7f3a50945863
      94e821fc
    decoded field element of 255 bits: (length: 32 bytes)
      10047198e8c4cacf0ab8a6d0ac337b8ae497209d042f7f3a50945863
      94e8217c
    generator g: (length: 32 bytes)
      4e6098733061c0e8486611a904fe5edb049804d26130a44131a6229e
      55c5c321
```

#### B.1.2. Test vector for MSGa

```
  Inputs
    ADa = b'ADa'
    ya (little endian): (length: 32 bytes)
      21b4f4bd9e64ed355c3eb676a28ebedaf6d8f17bdc365995b3190971
      53044080
  Outputs
    Ya: (length: 32 bytes)
      f970e36f37cfcd9a39e37dd2d1fbc9156d6d2f9ae422f4722cbd9d32
      e9b1e704
    MSGa = lv_cat(Ya,ADa): (length: 37 bytes)
      20f970e36f37cfcd9a39e37dd2d1fbc9156d6d2f9ae422f4722cbd9d
      32e9b1e70403414461
```

### B.1.3. Test vector for MSGb

```
  Inputs
    ADb = b'ADb'
    yb (little endian): (length: 32 bytes)
      848b0779ff415f0af4ea14df9dd1d3c29ac41d836c7808896c4eba19
      c51ac40a
  Outputs
    Yb: (length: 32 bytes)
      0178bbbab0804a4455b8f02e5d6e7d80997c6470bfb3618d7e74c396
      47af5a29
    MSGb = lv_cat(Yb,ADb): (length: 37 bytes)
      200178bbbab0804a4455b8f02e5d6e7d80997c6470bfb3618d7e74c3
      9647af5a2903414462
```

### B.1.4. Test vector for secret points K

```
    scalar_mult_vfy(ya,Yb): (length: 32 bytes)
      42ba4c6dc4c184a1cf405d4503f64bf7f015e2a0107450e38b9efff3
      bee52412
    scalar_mult_vfy(yb,Ya): (length: 32 bytes)
      42ba4c6dc4c184a1cf405d4503f64bf7f015e2a0107450e38b9efff3
      bee52412
```

### B.1.5. Test vector for ISK calculation initiator/responder

```
    unordered cat of transcript : (length: 74 bytes)
      20f970e36f37cfcd9a39e37dd2d1fbc9156d6d2f9ae422f4722cbd9d
      32e9b1e70403414461200178bbbab0804a4455b8f02e5d6e7d80997c
      6470bfb3618d7e74c39647af5a2903414462
    DSI = G.DSI_ISK, b'CPace255_ISK': (length: 12 bytes)
      435061636532353 55f49534b
    lv_cat(DSI,sid,K)||MSGa||MSGb: (length: 137 bytes)
      0c435061636532353 55f49534b107e4b4791d6a8ef019b936c79fb7f
      2c572042ba4c6dc4c184a1cf405d4503f64bf7f015e2a0107450e38b
      9efff3bee5241220f970e36f37cfcd9a39e37dd2d1fbc9156d6d2f9a
      e422f4722cbd9d32e9b1e70403414461200178bbbab0804a4455b8f0
      2e5d6e7d80997c6470bfb3618d7e74c39647af5a2903414462
    ISK result: (length: 64 bytes)
      f5ef3c13fdb9dfe839bdbf8a9256e8cee7db8a8f1dfa74958a925450
      cf8089cd560d9a4e7956b7334b6f625c8559b75ea0764ac2be894b8f
      3d434b30e87797d5
```

### B.1.6. Test vector for ISK calculation parallel execution

```
ordered cat of transcript : (length: 76 bytes)
  6f6320f970e36f37cfcd9a39e37dd2d1fbc9156d6d2f9ae422f4722c
  bd9d32e9b1e70403414461200178bbbab0804a4455b8f02e5d6e7d80
  997c6470bfb3618d7e74c39647af5a2903414462
DSI = G.DSI_ISK, b'CPace255_ISK': (length: 12 bytes)
  435061636532353 55f49534b
lv_cat(DSI,sid,K)||o_cat(MSGa,MSGb): (length: 139 bytes)
  0c435061636532353 55f49534b107e4b4791d6a8ef019b936c79fb7f
  2c572042ba4c6dc4c184a1cf405d4503f64bf7f015e2a0107450e38b
  9efff3bee524126f6320f970e36f37cfcd9a39e37dd2d1fbc9156d6d
  2f9ae422f4722cbd9d32e9b1e70403414461200178bbbab0804a4455
  b8f02e5d6e7d80997c6470bfb3618d7e74c39647af5a2903414462
ISK result: (length: 64 bytes)
  f4051edc63b2620e10d5ecf76d9f0c5ccd1447858a98d4bf847fafac
  737478c1350e14619bc0fcd4f028d10e4102dfca39f91fe9b829a503
  ab3e0549bd835edf
```

### B.1.7. Corresponding C programming language initializers

```
const unsigned char tc_PRS[] = {
 0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const unsigned char tc_CI[] = {
 0x0a,0x41,0x69,0x6e,0x69,0x74,0x69,0x61,0x74,0x6f,0x72,0x0a,
 0x42,0x72,0x65,0x73,0x70,0x6f,0x6e,0x64,0x65,0x72,
};
const unsigned char tc_sid[] = {
 0x7e,0x4b,0x47,0x91,0xd6,0xa8,0xef,0x01,0x9b,0x93,0x6c,0x79,
 0xfb,0x7f,0x2c,0x57,
};
const unsigned char tc_g[] = {
 0x4e,0x60,0x98,0x73,0x30,0x61,0xc0,0xe8,0x48,0x66,0x11,0xa9,
 0x04,0xfe,0x5e,0xdb,0x04,0x98,0x04,0xd2,0x61,0x30,0xa4,0x41,
 0x31,0xa6,0x22,0x9e,0x55,0xc5,0xc3,0x21,
};
const unsigned char tc_ya[] = {
 0x21,0xb4,0xf4,0xbd,0x9e,0x64,0xed,0x35,0x5c,0x3e,0xb6,0x76,
 0xa2,0x8e,0xbe,0xda,0xf6,0xd8,0xf1,0x7b,0xdc,0x36,0x59,0x95,
 0xb3,0x19,0x09,0x71,0x53,0x04,0x40,0x80,
};
const unsigned char tc_ADa[] = {
 0x41,0x44,0x61,
};
const unsigned char tc_Ya[] = {
 0xf9,0x70,0xe3,0x6f,0x37,0xcf,0xcd,0x9a,0x39,0xe3,0x7d,0xd2,
 0xd1,0xfb,0xc9,0x15,0x6d,0x6d,0x2f,0x9a,0xe4,0x22,0xf4,0x72,
 0x2c,0xbd,0x9d,0x32,0xe9,0xb1,0xe7,0x04,
};
const unsigned char tc_yb[] = {
 0x84,0x8b,0x07,0x79,0xff,0x41,0x5f,0x0a,0xf4,0xea,0x14,0xdf,
 0x9d,0xd1,0xd3,0xc2,0x9a,0xc4,0x1d,0x83,0x6c,0x78,0x08,0x89,
 0x6c,0x4e,0xba,0x19,0xc5,0x1a,0xc4,0x0a,
};
const unsigned char tc_ADb[] = {
 0x41,0x44,0x62,
};
const unsigned char tc_Yb[] = {
 0x01,0x78,0xbb,0xba,0xb0,0x80,0x4a,0x44,0x55,0xb8,0xf0,0x2e,
 0x5d,0x6e,0x7d,0x80,0x99,0x7c,0x64,0x70,0xbf,0xb3,0x61,0x8d,
 0x7e,0x74,0xc3,0x96,0x47,0xaf,0x5a,0x29,
};
const unsigned char tc_K[] = {
 0x42,0xba,0x4c,0x6d,0xc4,0xc1,0x84,0xa1,0xcf,0x40,0x5d,0x45,
 0x03,0xf6,0x4b,0xf7,0xf0,0x15,0xe2,0xa0,0x10,0x74,0x50,0xe3,
 0x8b,0x9e,0xff,0xf3,0xbe,0xe5,0x24,0x12,
};
const unsigned char tc_ISK_IR[] = {
 0xf5,0xef,0x3c,0x13,0xfd,0xb9,0xdf,0xe8,0x39,0xbd,0xbf,0x8a,
```

```
 0x92,0x56,0xe8,0xce,0xe7,0xdb,0x8a,0x8f,0x1d,0xfa,0x74,0x95,
 0x8a,0x92,0x54,0x50,0xcf,0x80,0x89,0xcd,0x56,0x0d,0x9a,0x4e,
 0x79,0x56,0xb7,0x33,0x4b,0x6f,0x62,0x5c,0x85,0x59,0xb7,0x5e,
 0xa0,0x76,0x4a,0xc2,0xbe,0x89,0x4b,0x8f,0x3d,0x43,0x4b,0x30,
 0xe8,0x77,0x97,0xd5,
};
const unsigned char tc_ISK_SY[] = {
 0xf4,0x05,0x1e,0xdc,0x63,0xb2,0x62,0x0e,0x10,0xd5,0xec,0xf7,
 0x6d,0x9f,0x0c,0x5c,0xcd,0x14,0x47,0x85,0x8a,0x98,0xd4,0xbf,
 0x84,0x7f,0xaf,0xac,0x73,0x74,0x78,0xc1,0x35,0x0e,0x14,0x61,
 0x9b,0xc0,0xfc,0xd4,0xf0,0x28,0xd1,0x0e,0x41,0x02,0xdf,0xca,
 0x39,0xf9,0x1f,0xe9,0xb8,0x29,0xa5,0x03,0xab,0x3e,0x05,0x49,
 0xbd,0x83,0x5e,0xdf,
};
```

### B.1.8. Test vectors for G_X25519.scalar_mult_vfy: low order points

Test vectors for which G_X25519.scalar_mult_vfy(s_in,ux) must return the neutral element or would return the neutral element if bit #255 of field element representation was not correctly cleared. (The decodeUCoordinate function from RFC7748 mandates clearing bit #255 for field element representations for use in the X25519 function.).

```
u0: 0000000000000000000000000000000000000000000000000000000000000000
u1: 0100000000000000000000000000000000000000000000000000000000000000
u2: ecffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff7f
u3: e0eb7a7c3b41b8ae1656e3faf19fc46ada098deb9c32b1fd866205165f49b800
u4: 5f9c95bca3508c24b1d0b1559c83ef5b04445cc4581c8e86d8224eddd09f1157
u5: edffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff7f
u6: daffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
u7: eeffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff7f
u8: dbffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
u9: d9ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ua: cdeb7a7c3b41b8ae1656e3faf19fc46ada098deb9c32b1fd866205165f49b880
ub: 4c9c95bca3508c24b1d0b1559c83ef5b04445cc4581c8e86d8224eddd09f11d7

u0 ... ub MUST be verified to produce the correct results q0 ... qb:

Additionally, u0,u1,u2,u3,u4,u5 and u7 MUST trigger the abort case
when included in MSGa or MSGb.

s = af46e36bf0527c9d3b16154b82465edd62144c0ac1fc5a18506a2244ba449aff
qN = G_X25519.scalar_mult_vfy(s, uX)
q0: 0000000000000000000000000000000000000000000000000000000000000000
q1: 0000000000000000000000000000000000000000000000000000000000000000
q2: 0000000000000000000000000000000000000000000000000000000000000000
q3: 0000000000000000000000000000000000000000000000000000000000000000
q4: 0000000000000000000000000000000000000000000000000000000000000000
q5: 0000000000000000000000000000000000000000000000000000000000000000
q6: d8e2c776bbacd510d09fd9278b7edcd25fc5ae9adfba3b6e040e8d3b71b21806
q7: 0000000000000000000000000000000000000000000000000000000000000000
q8: c85c655ebe8be44ba9c0ffde69f2fe10194458d137f09bbff725ce58803cdb38
q9: db64dafa9b8fdd136914e61461935fe92aa372cb056314e1231bc4ec12417456
qa: e062dcd5376d58297be2618c7498f55baa07d7e03184e8aada20bca28888bf7a
qb: 993c6ad11c4c29da9a56f7691fd0ff8d732e49de6250b6c2e80003ff4629a175
```

### B.2. Test vector for CPace using group X448 and hash SHAKE-256

#### B.2.1. Test vectors for calculate_generator with group X448

```
  Inputs
    H   = SHAKE-256 with input block size 136 bytes.
    PRS = b'Password' ; ZPAD length: 117 ; DSI = b'CPace448'
    CI = b'\nAinitiator\nBresponder'
    CI = 0a41696e69746961746f720a42726573706f6e646572
    sid = 5223e0cdc45d6575668d64c552004124
  Outputs
    generator_string(G.DSI,PRS,CI,sid,H.s_in_bytes):
    (length: 176 bytes)
      0843506163653434380850617373776f726475000000000000000000
      0000000000000000000000000000000000000000000000000000000000
      0000000000000000000000000000000000000000000000000000000000
      0000000000000000000000000000000000000000000000000000000000
      0000000000000000000000000000000000000000000000000000000000
      000000000000000000000000000000000000000000000160a4169
      6e69746961746f720a42726573706f6e646572105223e0cdc45d6575
      668d64c552004124
    hash generator string: (length: 56 bytes)
      769e06d6c41c8cf1c87aa3df8e687167f6d0a2e41821e856276a0221
      d88272359d0b43204b546174c9179c83c107b707f296eafaa1c5a293
    decoded field element of 448 bits: (length: 56 bytes)
      769e06d6c41c8cf1c87aa3df8e687167f6d0a2e41821e856276a0221
      d88272359d0b43204b546174c9179c83c107b707f296eafaa1c5a293
    generator g: (length: 56 bytes)
      6fdae14718eb7506dd96e3f7797896efdb8db9ec0797485c9c48a192
      2e44961da097f2908b084a5de33ab671630660d27d79ffd6ee8ec846
```

#### B.2.2. Test vector for MSGa

```
  Inputs
    ADa = b'ADa'
    ya (little endian): (length: 56 bytes)
      21b4f4bd9e64ed355c3eb676a28ebedaf6d8f17bdc365995b3190971
      53044080516bd083bfcce66121a3072646994c8430cc382b8dc543e8
  Outputs
    Ya: (length: 56 bytes)
      396bd11daf223711e575cac6021e3fa31558012048a1cec7876292b9
      6c61eda353fe04f33028d2352779668a934084da776c1c51a58ce4b5
    MSGa = lv_cat(Ya,ADa): (length: 61 bytes)
      38396bd11daf223711e575cac6021e3fa31558012048a1cec7876292
      b96c61eda353fe04f33028d2352779668a934084da776c1c51a58ce4
      b503414461
```

### B.2.3. Test vector for MSGb

```
  Inputs
    ADb = b'ADb'
    yb (little endian): (length: 56 bytes)
      848b0779ff415f0af4ea14df9dd1d3c29ac41d836c7808896c4eba19
      c51ac40a439caf5e61ec88c307c7d619195229412eaa73fb2a5ea20d
  Outputs
    Yb: (length: 56 bytes)
      53c519fb490fde5a04bda8c18b327d0fc1a9391d19e0ac00c59df9c6
      0422284e593d6b092eac94f5aa644ed883f39bd4f04e4beb6af86d58
    MSGb = lv_cat(Yb,ADb): (length: 61 bytes)
      3853c519fb490fde5a04bda8c18b327d0fc1a9391d19e0ac00c59df9
      c60422284e593d6b092eac94f5aa644ed883f39bd4f04e4beb6af86d
      5803414462
```

### B.2.4. Test vector for secret points K

```
    scalar_mult_vfy(ya,Yb): (length: 56 bytes)
      e00af217556a40ccbc9822cc27a43542e45166a653aa4df746d5f8e1
      e8df483e9baff71c9eb03ee20a688ad4e4d359f70ac9ec3f6a659997
    scalar_mult_vfy(yb,Ya): (length: 56 bytes)
      e00af217556a40ccbc9822cc27a43542e45166a653aa4df746d5f8e1
      e8df483e9baff71c9eb03ee20a688ad4e4d359f70ac9ec3f6a659997
```

### B.2.5. Test vector for ISK calculation initiator/responder

```
    unordered cat of transcript : (length: 122 bytes)
      38396bd11daf223711e575cac6021e3fa31558012048a1cec7876292
      b96c61eda353fe04f33028d2352779668a934084da776c1c51a58ce4
      b5034144613853c519fb490fde5a04bda8c18b327d0fc1a9391d19e0
      ac00c59df9c60422284e593d6b092eac94f5aa644ed883f39bd4f04e
      4beb6af86d5803414462
    DSI = G.DSI_ISK, b'CPace448_ISK': (length: 12 bytes)
      435061636534343 85f49534b
    lv_cat(DSI,sid,K)||MSGa||MSGb: (length: 209 bytes)
      0c435061636534343 85f49534b105223e0cdc45d6575668d64c55200
      412438e00af217556a40ccbc9822cc27a43542e45166a653aa4df746
      d5f8e1e8df483e9baff71c9eb03ee20a688ad4e4d359f70ac9ec3f6a
      65999738396bd11daf223711e575cac6021e3fa31558012048a1cec7
      876292b96c61eda353fe04f33028d2352779668a934084da776c1c51
      a58ce4b5034144613853c519fb490fde5a04bda8c18b327d0fc1a939
      1d19e0ac00c59df9c60422284e593d6b092eac94f5aa644ed883f39b
      d4f04e4beb6af86d5803414462
    ISK result: (length: 64 bytes)
      4030297722c1914711da6b2a224a44b53b30c05ab02c2a3d3ccc7272
      a3333ce3a4564c17031b634e89f65681f52d5c3d1df7baeb88523d2e
      481b3858aed86315
```

### B.2.6. Test vector for ISK calculation parallel execution

```
ordered cat of transcript : (length: 124 bytes)
  6f633853c519fb490fde5a04bda8c18b327d0fc1a9391d19e0ac00c5
  9df9c60422284e593d6b092eac94f5aa644ed883f39bd4f04e4beb6a
  f86d580341446238396bd11daf223711e575cac6021e3fa315580120
  48a1cec7876292b96c61eda353fe04f33028d2352779668a934084da
  776c1c51a58ce4b503414461
DSI = G.DSI_ISK, b'CPace448_ISK': (length: 12 bytes)
  435061636534343 85f49534b
lv_cat(DSI,sid,K)||o_cat(MSGa,MSGb): (length: 211 bytes)
  0c435061636534343 85f49534b105223e0cdc45d6575668d64c55200
  412438e00af217556a40ccbc9822cc27a43542e45166a653aa4df746
  d5f8e1e8df483e9baff71c9eb03ee20a688ad4e4d359f70ac9ec3f6a
  6599976f633853c519fb490fde5a04bda8c18b327d0fc1a9391d19e0
  ac00c59df9c60422284e593d6b092eac94f5aa644ed883f39bd4f04e
  4beb6af86d580341446238396bd11daf223711e575cac6021e3fa315
  58012048a1cec7876292b96c61eda353fe04f33028d2352779668a93
  4084da776c1c51a58ce4b503414461
ISK result: (length: 64 bytes)
  4cd30768e2f75f0583449614bce823b421c31163c5a3bde4eed1c664
  284a32995ea3430b5c47fc7dd771b534ad38eaea5d8c8f97bd548966
  7facfc044615075f
```

### B.2.7. Corresponding C programming language initializers

```
const unsigned char tc_PRS[] = {
 0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const unsigned char tc_CI[] = {
 0x0a,0x41,0x69,0x6e,0x69,0x74,0x69,0x61,0x74,0x6f,0x72,0x0a,
 0x42,0x72,0x65,0x73,0x70,0x6f,0x6e,0x64,0x65,0x72,
};
const unsigned char tc_sid[] = {
 0x52,0x23,0xe0,0xcd,0xc4,0x5d,0x65,0x75,0x66,0x8d,0x64,0xc5,
 0x52,0x00,0x41,0x24,
};
const unsigned char tc_g[] = {
 0x6f,0xda,0xe1,0x47,0x18,0xeb,0x75,0x06,0xdd,0x96,0xe3,0xf7,
 0x79,0x78,0x96,0xef,0xdb,0x8d,0xb9,0xec,0x07,0x97,0x48,0x5c,
 0x9c,0x48,0xa1,0x92,0x2e,0x44,0x96,0x1d,0xa0,0x97,0xf2,0x90,
 0x8b,0x08,0x4a,0x5d,0xe3,0x3a,0xb6,0x71,0x63,0x06,0x60,0xd2,
 0x7d,0x79,0xff,0xd6,0xee,0x8e,0xc8,0x46,
};
const unsigned char tc_ya[] = {
 0x21,0xb4,0xf4,0xbd,0x9e,0x64,0xed,0x35,0x5c,0x3e,0xb6,0x76,
 0xa2,0x8e,0xbe,0xda,0xf6,0xd8,0xf1,0x7b,0xdc,0x36,0x59,0x95,
 0xb3,0x19,0x09,0x71,0x53,0x04,0x40,0x80,0x51,0x6b,0xd0,0x83,
 0xbf,0xcc,0xe6,0x61,0x21,0xa3,0x07,0x26,0x46,0x99,0x4c,0x84,
 0x30,0xcc,0x38,0x2b,0x8d,0xc5,0x43,0xe8,
};
const unsigned char tc_ADa[] = {
 0x41,0x44,0x61,
};
const unsigned char tc_Ya[] = {
 0x39,0x6b,0xd1,0x1d,0xaf,0x22,0x37,0x11,0xe5,0x75,0xca,0xc6,
 0x02,0x1e,0x3f,0xa3,0x15,0x58,0x01,0x20,0x48,0xa1,0xce,0xc7,
 0x87,0x62,0x92,0xb9,0x6c,0x61,0xed,0xa3,0x53,0xfe,0x04,0xf3,
 0x30,0x28,0xd2,0x35,0x27,0x79,0x66,0x8a,0x93,0x40,0x84,0xda,
 0x77,0x6c,0x1c,0x51,0xa5,0x8c,0xe4,0xb5,
};
const unsigned char tc_yb[] = {
 0x84,0x8b,0x07,0x79,0xff,0x41,0x5f,0x0a,0xf4,0xea,0x14,0xdf,
 0x9d,0xd1,0xd3,0xc2,0x9a,0xc4,0x1d,0x83,0x6c,0x78,0x08,0x89,
 0x6c,0x4e,0xba,0x19,0xc5,0x1a,0xc4,0x0a,0x43,0x9c,0xaf,0x5e,
 0x61,0xec,0x88,0xc3,0x07,0xc7,0xd6,0x19,0x19,0x52,0x29,0x41,
 0x2e,0xaa,0x73,0xfb,0x2a,0x5e,0xa2,0x0d,
};
const unsigned char tc_ADb[] = {
 0x41,0x44,0x62,
};
const unsigned char tc_Yb[] = {
 0x53,0xc5,0x19,0xfb,0x49,0x0f,0xde,0x5a,0x04,0xbd,0xa8,0xc1,
 0x8b,0x32,0x7d,0x0f,0xc1,0xa9,0x39,0x1d,0x19,0xe0,0xac,0x00,
 0xc5,0x9d,0xf9,0xc6,0x04,0x22,0x28,0x4e,0x59,0x3d,0x6b,0x09,
```

```
 0x2e,0xac,0x94,0xf5,0xaa,0x64,0x4e,0xd8,0x83,0xf3,0x9b,0xd4,
 0xf0,0x4e,0x4b,0xeb,0x6a,0xf8,0x6d,0x58,
};
const unsigned char tc_K[] = {
 0xe0,0x0a,0xf2,0x17,0x55,0x6a,0x40,0xcc,0xbc,0x98,0x22,0xcc,
 0x27,0xa4,0x35,0x42,0xe4,0x51,0x66,0xa6,0x53,0xaa,0x4d,0xf7,
 0x46,0xd5,0xf8,0xe1,0xe8,0xdf,0x48,0x3e,0x9b,0xaf,0xf7,0x1c,
 0x9e,0xb0,0x3e,0xe2,0x0a,0x68,0x8a,0xd4,0xe4,0xd3,0x59,0xf7,
 0x0a,0xc9,0xec,0x3f,0x6a,0x65,0x99,0x97,
};
const unsigned char tc_ISK_IR[] = {
 0x40,0x30,0x29,0x77,0x22,0xc1,0x91,0x47,0x11,0xda,0x6b,0x2a,
 0x22,0x4a,0x44,0xb5,0x3b,0x30,0xc0,0x5a,0xb0,0x2c,0x2a,0x3d,
 0x3c,0xcc,0x72,0x72,0xa3,0x33,0x3c,0xe3,0xa4,0x56,0x4c,0x17,
 0x03,0x1b,0x63,0x4e,0x89,0xf6,0x56,0x81,0xf5,0x2d,0x5c,0x3d,
 0x1d,0xf7,0xba,0xeb,0x88,0x52,0x3d,0x2e,0x48,0x1b,0x38,0x58,
 0xae,0xd8,0x63,0x15,
};
const unsigned char tc_ISK_SY[] = {
 0x4c,0xd3,0x07,0x68,0xe2,0xf7,0x5f,0x05,0x83,0x44,0x96,0x14,
 0xbc,0xe8,0x23,0xb4,0x21,0xc3,0x11,0x63,0xc5,0xa3,0xbd,0xe4,
 0xee,0xd1,0xc6,0x64,0x28,0x4a,0x32,0x99,0x5e,0xa3,0x43,0x0b,
 0x5c,0x47,0xfc,0x7d,0xd7,0x71,0xb5,0x34,0xad,0x38,0xea,0xea,
 0x5d,0x8c,0x8f,0x97,0xbd,0x54,0x89,0x66,0x7f,0xac,0xfc,0x04,
 0x46,0x15,0x07,0x5f,
};
```

### B.2.8. Test vectors for G_X448.scalar_mult_vfy: low order points

Test vectors for which G_X448.scalar_mult_vfy(s_in,ux) must return the neutral element. This includes points that are non-canonicaly encoded, i.e. have coordinate values larger than the field prime.

Weak points for X448 smaller than the field prime (canonical)

u0: (length: 56 bytes)
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
u1: (length: 56 bytes)
0100000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
u2: (length: 56 bytes)
fefffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Weak points for X448 larger or equal to the field prime (non-canonical)

u3: (length: 56 bytes)
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
u4: (length: 56 bytes)
00000000000000000000000000000000000000000000000000000000000000ff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

All of the above points u0 ... u4 MUST trigger the abort case when included in the protocol messages MSGa or MSGb.

Expected results for X448 resp. G_X448.scalar_mult_vfy

```
scalar s: (length: 56 bytes)
  af8a14218bf2a2062926d2ea9b8fe4e8b6817349b6ed2feb1e5d64d7a4
  523f15fceec70fb111e870dc58d191e66a14d3e9d482d04432cadd
G_X448.scalar_mult_vfy(s,u0): (length: 56 bytes)
  00000000000000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000
G_X448.scalar_mult_vfy(s,u1): (length: 56 bytes)
  00000000000000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000
G_X448.scalar_mult_vfy(s,u2): (length: 56 bytes)
  00000000000000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000
G_X448.scalar_mult_vfy(s,u3): (length: 56 bytes)
  00000000000000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000
G_X448.scalar_mult_vfy(s,u4): (length: 56 bytes)
  00000000000000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000

 Test vectors for scalar_mult with nonzero outputs

scalar s: (length: 56 bytes)
  af8a14218bf2a2062926d2ea9b8fe4e8b6817349b6ed2feb1e5d64d7a4
  523f15fceec70fb111e870dc58d191e66a14d3e9d482d04432cadd
point coordinate u_curve on the curve: (length: 56 bytes)
  ab0c68d772ec2eb9de25c49700e46d6325e66d6aa39d7b65eb84a68c55
  69d47bd71b41f3e0d210f44e146dec8926b174acb3f940a0b82cab
G_X448.scalar_mult_vfy(s,u_curve): (length: 56 bytes)
  3b0fa9bc40a6fdc78c9e06ff7a54c143c5d52f365607053bf0656f5142
  0496295f910a101b38edc1acd3bd240fd55dcb7a360553b8a7627e

point coordinate u_twist on the twist: (length: 56 bytes)
  c981cd1e1f72d9c35c7d7cf6be426757c0dc8206a2fcfa564a8e7618c0
  3c0e61f9a2eb1c3e0dd97d6e9b1010f5edd03397a83f5a914cb3ff
G_X448.scalar_mult_vfy(s,u_twist): (length: 56 bytes)
  d0a2bb7e9c5c2c627793d8342f23b759fe7d9e3320a85ca4fd61376331
  50ffd9a9148a9b75c349fac43d64bec49a6e126cc92cbfbf353961
```

### B.3. Test vector for CPace using group ristretto255 and hash SHA-512

#### B.3.1. Test vectors for calculate_generator with group ristretto255

```
  Inputs
    H   = SHA-512 with input block size 128 bytes.
    PRS = b'Password' ; ZPAD length: 100 ;
    DSI = b'CPaceRistretto255'
    CI = b'\nAinitiator\nBresponder'
    CI = 0a41696e69746961746f720a42726573706f6e646572
    sid = 7e4b4791d6a8ef019b936c79fb7f2c57
  Outputs
    generator_string(G.DSI,PRS,CI,sid,H.s_in_bytes):
    (length: 168 bytes)
      1143506163655269737472657474 6f32353085061737377 6f726464
      0000000000000000000000000000000000000000000000000000000000
      0000000000000000000000000000000000000000000000000000000000
      0000000000000000000000000000000000000000000000000000000000
      0000000000000000000000000000160a41696e69746961746f72
      0a42726573706f6e646572107e4b4791d6a8ef019b936c79fb7f2c57
    hash result: (length: 64 bytes)
      a5ce446f63a1ae6d1fee80fa67d0b4004a4b1283ec5549a462bf33a6
      c1ae06a0871f9bf48545f49b2a792eed255ac04f52758c9c60448306
      810b44e986e3dcbb
    encoded generator g: (length: 32 bytes)
      5e25411ca1ad7c9debfd0b33ad987a95cefef2d3f15dcc8bd26415a5
      dfe2e15a
```

#### B.3.2. Test vector for MSGa

```
  Inputs
    ADa = b'ADa'
    ya (little endian): (length: 32 bytes)
      da3d23700a9e5699258aef94dc060dfda5ebb61f02a5ea77fad53f4f
      f0976d08
  Outputs
    Ya: (length: 32 bytes)
      383a85dd236978f17f8c8545b50dabc52a39fcdab2cf8bc531ce040f
      f77ca82d
    MSGa = lv_cat(Ya,ADa): (length: 37 bytes)
      20383a85dd236978f17f8c8545b50dabc52a39fcdab2cf8bc531ce04
      0ff77ca82d03414461
```

### B.3.3. Test vector for MSGb

```
  Inputs
    ADb = b'ADb'
    yb (little endian): (length: 32 bytes)
      d2316b454718c35362d83d69df6320f38578ed5984651435e2949762
      d900b80d
  Outputs
    Yb: (length: 32 bytes)
      a6206309c0e8e5f579295e35997ac4300ab3fecec3c17f7b604f3e69
      8fa1383c
    MSGb = lv_cat(Yb,ADb): (length: 37 bytes)
      20a6206309c0e8e5f579295e35997ac4300ab3fecec3c17f7b604f3e
      698fa1383c03414462
```

### B.3.4. Test vector for secret points K

```
    scalar_mult_vfy(ya,Yb): (length: 32 bytes)
      fa1d0318864e2cacb26875f1b791c9ae83204fe8359addb53e95a2e9
      8893853f
    scalar_mult_vfy(yb,Ya): (length: 32 bytes)
      fa1d0318864e2cacb26875f1b791c9ae83204fe8359addb53e95a2e9
      8893853f
```

### B.3.5. Test vector for ISK calculation initiator/responder

```
    unordered cat of transcript : (length: 74 bytes)
      20383a85dd236978f17f8c8545b50dabc52a39fcdab2cf8bc531ce04
      0ff77ca82d0341446120a6206309c0e8e5f579295e35997ac4300ab3
      fecec3c17f7b604f3e698fa1383c03414462
    DSI = G.DSI_ISK, b'CPaceRistretto255_ISK':
    (length: 21 bytes)
      435061636552697374726574746f3235355f49534b
    lv_cat(DSI,sid,K)||MSGa||MSGb: (length: 146 bytes)
      154350616365526973747265747 46f3235355f49534b107e4b4791d6
      a8ef019b936c79fb7f2c5720fa1d0318864e2cacb26875f1b791c9ae
      83204fe8359addb53e95a2e98893853f20383a85dd236978f17f8c85
      45b50dabc52a39fcdab2cf8bc531ce040ff77ca82d0341446120a620
      6309c0e8e5f579295e35997ac4300ab3fecec3c17f7b604f3e698fa1
      383c03414462
    ISK result: (length: 64 bytes)
      e91ccb2c0f5e0d0993a33956e3be59754f3f2b07db57631f5394452e
      a2e7b4354674eb1f5686c078462bf83bec72e8743df440108e638f35
      26d9b90e85be096f
```

### B.3.6. Test vector for ISK calculation parallel execution

```
ordered cat of transcript : (length: 76 bytes)
  6f6320a6206309c0e8e5f579295e35997ac4300ab3fecec3c17f7b60
  4f3e698fa1383c0341446220383a85dd236978f17f8c8545b50dabc5
  2a39fcdab2cf8bc531ce040ff77ca82d03414461
DSI = G.DSI_ISK, b'CPaceRistretto255_ISK':
(length: 21 bytes)
  435061636552697374726574746f3235355f49534b
lv_cat(DSI,sid,K)||o_cat(MSGa,MSGb): (length: 148 bytes)
  154350616365526973747265747 46f3235355f49534b107e4b4791d6
  a8ef019b936c79fb7f2c5720fa1d0318864e2cacb26875f1b791c9ae
  83204fe8359addb53e95a2e98893853f6f6320a6206309c0e8e5f579
  295e35997ac4300ab3fecec3c17f7b604f3e698fa1383c0341446220
  383a85dd236978f17f8c8545b50dabc52a39fcdab2cf8bc531ce040f
  f77ca82d03414461
ISK result: (length: 64 bytes)
  1638fb6ff564a80a12af07c036870e10c4efb539fa847fdf3e9c4621
  7bf52cd4df4ca0fe51146492a9ba6dd6a42ac402bc2d60adb4084c81
  758d754d1d81482a
```

### B.3.7. Corresponding C programming language initializers

```
const unsigned char tc_PRS[] = {
 0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const unsigned char tc_CI[] = {
 0x0a,0x41,0x69,0x6e,0x69,0x74,0x69,0x61,0x74,0x6f,0x72,0x0a,
 0x42,0x72,0x65,0x73,0x70,0x6f,0x6e,0x64,0x65,0x72,
};
const unsigned char tc_sid[] = {
 0x7e,0x4b,0x47,0x91,0xd6,0xa8,0xef,0x01,0x9b,0x93,0x6c,0x79,
 0xfb,0x7f,0x2c,0x57,
};
const unsigned char tc_g[] = {
 0x5e,0x25,0x41,0x1c,0xa1,0xad,0x7c,0x9d,0xeb,0xfd,0x0b,0x33,
 0xad,0x98,0x7a,0x95,0xce,0xfe,0xf2,0xd3,0xf1,0x5d,0xcc,0x8b,
 0xd2,0x64,0x15,0xa5,0xdf,0xe2,0xe1,0x5a,
};
const unsigned char tc_ya[] = {
 0xda,0x3d,0x23,0x70,0x0a,0x9e,0x56,0x99,0x25,0x8a,0xef,0x94,
 0xdc,0x06,0x0d,0xfd,0xa5,0xeb,0xb6,0x1f,0x02,0xa5,0xea,0x77,
 0xfa,0xd5,0x3f,0x4f,0xf0,0x97,0x6d,0x08,
};
const unsigned char tc_ADa[] = {
 0x41,0x44,0x61,
};
const unsigned char tc_Ya[] = {
 0x38,0x3a,0x85,0xdd,0x23,0x69,0x78,0xf1,0x7f,0x8c,0x85,0x45,
 0xb5,0x0d,0xab,0xc5,0x2a,0x39,0xfc,0xda,0xb2,0xcf,0x8b,0xc5,
 0x31,0xce,0x04,0x0f,0xf7,0x7c,0xa8,0x2d,
};
const unsigned char tc_yb[] = {
 0xd2,0x31,0x6b,0x45,0x47,0x18,0xc3,0x53,0x62,0xd8,0x3d,0x69,
 0xdf,0x63,0x20,0xf3,0x85,0x78,0xed,0x59,0x84,0x65,0x14,0x35,
 0xe2,0x94,0x97,0x62,0xd9,0x00,0xb8,0x0d,
};
const unsigned char tc_ADb[] = {
 0x41,0x44,0x62,
};
const unsigned char tc_Yb[] = {
 0xa6,0x20,0x63,0x09,0xc0,0xe8,0xe5,0xf5,0x79,0x29,0x5e,0x35,
 0x99,0x7a,0xc4,0x30,0x0a,0xb3,0xfe,0xce,0xc3,0xc1,0x7f,0x7b,
 0x60,0x4f,0x3e,0x69,0x8f,0xa1,0x38,0x3c,
};
const unsigned char tc_K[] = {
 0xfa,0x1d,0x03,0x18,0x86,0x4e,0x2c,0xac,0xb2,0x68,0x75,0xf1,
 0xb7,0x91,0xc9,0xae,0x83,0x20,0x4f,0xe8,0x35,0x9a,0xdd,0xb5,
 0x3e,0x95,0xa2,0xe9,0x88,0x93,0x85,0x3f,
};
const unsigned char tc_ISK_IR[] = {
 0xe9,0x1c,0xcb,0x2c,0x0f,0x5e,0x0d,0x09,0x93,0xa3,0x39,0x56,
```

```
 0xe3,0xbe,0x59,0x75,0x4f,0x3f,0x2b,0x07,0xdb,0x57,0x63,0x1f,
 0x53,0x94,0x45,0x2e,0xa2,0xe7,0xb4,0x35,0x46,0x74,0xeb,0x1f,
 0x56,0x86,0xc0,0x78,0x46,0x2b,0xf8,0x3b,0xec,0x72,0xe8,0x74,
 0x3d,0xf4,0x40,0x10,0x8e,0x63,0x8f,0x35,0x26,0xd9,0xb9,0x0e,
 0x85,0xbe,0x09,0x6f,
};
const unsigned char tc_ISK_SY[] = {
 0x16,0x38,0xfb,0x6f,0xf5,0x64,0xa8,0x0a,0x12,0xaf,0x07,0xc0,
 0x36,0x87,0x0e,0x10,0xc4,0xef,0xb5,0x39,0xfa,0x84,0x7f,0xdf,
 0x3e,0x9c,0x46,0x21,0x7b,0xf5,0x2c,0xd4,0xdf,0x4c,0xa0,0xfe,
 0x51,0x14,0x64,0x92,0xa9,0xba,0x6d,0xd6,0xa4,0x2a,0xc4,0x02,
 0xbc,0x2d,0x60,0xad,0xb4,0x08,0x4c,0x81,0x75,0x8d,0x75,0x4d,
 0x1d,0x81,0x48,0x2a,
};
```

### B.3.8. Test case for scalar_mult with valid inputs

```
s: (length: 32 bytes)
  7cd0e075fa7955ba52c02759a6c90dbbfc10e6d40aea8d283e407d88
  cf538a05
X: (length: 32 bytes)
  2c3c6b8c4f3800e7aef6864025b4ed79bd599117e427c41bd47d93d6
  54b4a51c
G.scalar_mult(s,decode(X)): (length: 32 bytes)
  7c13645fe790a468f62c39beb7388e541d8405d1ade69d1778c5fe3e
  7f6b600e
G.scalar_mult_vfy(s,X): (length: 32 bytes)
  7c13645fe790a468f62c39beb7388e541d8405d1ade69d1778c5fe3e
  7f6b600e
```

### B.3.9. Invalid inputs for scalar_mult_vfy

For these test cases scalar_mult_vfy(y,.) MUST return the representation of the neutral element G.I. When points Y_i1 or Y_i2 are included in MSGa or MSGb the protocol MUST abort.

```
s: (length: 32 bytes)
  7cd0e075fa7955ba52c02759a6c90dbbfc10e6d40aea8d283e407d88
  cf538a05
Y_i1: (length: 32 bytes)
  2b3c6b8c4f3800e7aef6864025b4ed79bd599117e427c41bd47d93d6
  54b4a51c
Y_i2 == G.I: (length: 32 bytes)
  0000000000000000000000000000000000000000000000000000000
  00000000
G.scalar_mult_vfy(s,Y_i1) = G.scalar_mult_vfy(s,Y_i2) = G.I
```

### B.4. Test vector for CPace using group decaf448 and hash SHAKE-256

#### B.4.1. Test vectors for calculate_generator with group decaf448

```
  Inputs
    H   = SHAKE-256 with input block size 136 bytes.
    PRS = b'Password' ; ZPAD length: 112 ;
    DSI = b'CPaceDecaf448'
    CI = b'\nAinitiator\nBresponder'
    CI = 0a41696e69746961746f720a42726573706f6e646572
    sid = 5223e0cdc45d6575668d64c552004124
  Outputs
    generator_string(G.DSI,PRS,CI,sid,H.s_in_bytes):
    (length: 176 bytes)
      0d435061636544656361663434380850617373776f72647000000000
      0000000000000000000000000000000000000000000000000000000
      0000000000000000000000000000000000000000000000000000000
      0000000000000000000000000000000000000000000000000000000
      0000000000000000000000000000000000000000000160a4169
      6e69746961746f720a42726573706f6e646572105223e0cdc45d6575
      668d64c552004124
    hash result: (length: 112 bytes)
      8955b426ff1d3a22032d21c013cf94134cee9a4235e93261a4911edb
      f68f2945f0267c983954262c7f59badb9caf468ebe21b7e9885657af
      b8f1a3b783c2047ba519e113ecf81b2b580dd481f499beabd401cc77
      1d28915fb750011209040f5f03b2ceb5e5eb259c96b478382d5a5c57
    encoded generator g: (length: 56 bytes)
      682d1a4f49fc2a4834356ae4d7f58636bc9481521c845e66e6fb0b29
      69341df45fbaeaea9e2221b3f5babc54c5f8ce456988ffc519defaeb
```

#### B.4.2. Test vector for MSGa

```
  Inputs
    ADa = b'ADa'
    ya (little endian): (length: 56 bytes)
      33d561f13cfc0dca279c30e8cde895175dc25483892819eba132d58c
      13c0462a8eb0d73fda941950594bef5191d8394691f86edffcad6c1e
  Outputs
    Ya: (length: 56 bytes)
      e233867540319ec86eaecc09a85dec233745db729f61c36bde14c034
      200994fc4b6e8d263008c169585fd1d186d8ac560cb9f7ad0d166965
    MSGa = lv_cat(Ya,ADa): (length: 61 bytes)
      38e233867540319ec86eaecc09a85dec233745db729f61c36bde14c0
      34200994fc4b6e8d263008c169585fd1d186d8ac560cb9f7ad0d1669
      6503414461
```

### B.4.3. Test vector for MSGb

```
  Inputs
    ADb = b'ADb'
    yb (little endian): (length: 56 bytes)
      2523c969f68fa2b2aea294c2539ef36eb1e0558abd14712a7828f16a
      85ed2c7e77e2bdd418994405fb1b57b6bbaadd66849892aac9d81402
  Outputs
    Yb: (length: 56 bytes)
      5062a0f33478914bf162a80dad39b5b266c1dd02f408573b41827e38
      599b682afbf7a0735adfd68c39ab4994fd1b034846270e38332b4da9
    MSGb = lv_cat(Yb,ADb): (length: 61 bytes)
      385062a0f33478914bf162a80dad39b5b266c1dd02f408573b41827e
      38599b682afbf7a0735adfd68c39ab4994fd1b034846270e38332b4d
      a903414462
```

### B.4.4. Test vector for secret points K

```
    scalar_mult_vfy(ya,Yb): (length: 56 bytes)
      dc9edef7c127e79d32f2584f9fcd3269174fe32226c2082963879a6d
      eafefb9c14efcee9fc1245917ad3658037d2d62aff2d3f76fa4fca99
    scalar_mult_vfy(yb,Ya): (length: 56 bytes)
      dc9edef7c127e79d32f2584f9fcd3269174fe32226c2082963879a6d
      eafefb9c14efcee9fc1245917ad3658037d2d62aff2d3f76fa4fca99
```

### B.4.5. Test vector for ISK calculation initiator/responder

```
    unordered cat of transcript : (length: 122 bytes)
      38e233867540319ec86eaecc09a85dec233745db729f61c36bde14c0
      34200994fc4b6e8d263008c169585fd1d186d8ac560cb9f7ad0d1669
      6503414461385062a0f33478914bf162a80dad39b5b266c1dd02f408
      573b41827e38599b682afbf7a0735adfd68c39ab4994fd1b03484627
      0e38332b4da903414462
    DSI = G.DSI_ISK, b'CPaceDecaf448_ISK': (length: 17 bytes)
      435061636544656361663434385f49534b
    lv_cat(DSI,sid,K)||MSGa||MSGb: (length: 214 bytes)
      1143506163654465636166343438 5f49534b105223e0cdc45d657566
      8d64c55200412438dc9edef7c127e79d32f2584f9fcd3269174fe322
      26c2082963879a6deafefb9c14efcee9fc1245917ad3658037d2d62a
      ff2d3f76fa4fca9938e233867540319ec86eaecc09a85dec233745db
      729f61c36bde14c034200994fc4b6e8d263008c169585fd1d186d8ac
      560cb9f7ad0d1669650341446138506 2a0f33478914bf162a80dad39
      b5b266c1dd02f408573b41827e38599b682afbf7a0735adfd68c39ab
      4994fd1b034846270e38332b4da903414462
    ISK result: (length: 64 bytes)
      a752612fe6dec542e96629a6eb68ecb9bfe2257224975e916035aee7
      47c6aba32af2e6fe25eeb96261e6140100edcf95686e0aaa134026b4
      b5254fd271b7a4da
```

### B.4.6. Test vector for ISK calculation parallel execution

```
ordered cat of transcript : (length: 124 bytes)
  6f6338e233867540319ec86eaecc09a85dec233745db729f61c36bde
  14c034200994fc4b6e8d263008c169585fd1d186d8ac560cb9f7ad0d
  1669650341446138506 2a0f33478914bf162a80dad39b5b266c1dd02
  f408573b41827e38599b682afbf7a0735adfd68c39ab4994fd1b0348
  46270e38332b4da903414462
DSI = G.DSI_ISK, b'CPaceDecaf448_ISK': (length: 17 bytes)
  435061636544656361663434385f49534b
lv_cat(DSI,sid,K)||o_cat(MSGa,MSGb): (length: 216 bytes)
  1143506163654465636166343438 5f49534b105223e0cdc45d657566
  8d64c55200412438dc9edef7c127e79d32f2584f9fcd3269174fe322
  26c2082963879a6deafefb9c14efcee9fc1245917ad3658037d2d62a
  ff2d3f76fa4fca996f6338e233867540319ec86eaecc09a85dec2337
  45db729f61c36bde14c034200994fc4b6e8d263008c169585fd1d186
  d8ac560cb9f7ad0d1669650341446138506 2a0f33478914bf162a80d
  ad39b5b266c1dd02f408573b41827e38599b682afbf7a0735adfd68c
  39ab4994fd1b034846270e38332b4da903414462
ISK result: (length: 64 bytes)
  e6c79d30d4381a45bd47b14b769d41354211aff553ece937d4ac134f
  09844896c72a723b1f1b6da1ab281d759a15624d2bcd0e423b70b8b8
  50a4d0ed126a3026
```

### B.4.7. Corresponding C programming language initializers

```
const unsigned char tc_PRS[] = {
 0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const unsigned char tc_CI[] = {
 0x0a,0x41,0x69,0x6e,0x69,0x74,0x69,0x61,0x74,0x6f,0x72,0x0a,
 0x42,0x72,0x65,0x73,0x70,0x6f,0x6e,0x64,0x65,0x72,
};
const unsigned char tc_sid[] = {
 0x52,0x23,0xe0,0xcd,0xc4,0x5d,0x65,0x75,0x66,0x8d,0x64,0xc5,
 0x52,0x00,0x41,0x24,
};
const unsigned char tc_g[] = {
 0x68,0x2d,0x1a,0x4f,0x49,0xfc,0x2a,0x48,0x34,0x35,0x6a,0xe4,
 0xd7,0xf5,0x86,0x36,0xbc,0x94,0x81,0x52,0x1c,0x84,0x5e,0x66,
 0xe6,0xfb,0x0b,0x29,0x69,0x34,0x1d,0xf4,0x5f,0xba,0xea,0xea,
 0x9e,0x22,0x21,0xb3,0xf5,0xba,0xbc,0x54,0xc5,0xf8,0xce,0x45,
 0x69,0x88,0xff,0xc5,0x19,0xde,0xfa,0xeb,
};
const unsigned char tc_ya[] = {
 0x33,0xd5,0x61,0xf1,0x3c,0xfc,0x0d,0xca,0x27,0x9c,0x30,0xe8,
 0xcd,0xe8,0x95,0x17,0x5d,0xc2,0x54,0x83,0x89,0x28,0x19,0xeb,
 0xa1,0x32,0xd5,0x8c,0x13,0xc0,0x46,0x2a,0x8e,0xb0,0xd7,0x3f,
 0xda,0x94,0x19,0x50,0x59,0x4b,0xef,0x51,0x91,0xd8,0x39,0x46,
 0x91,0xf8,0x6e,0xdf,0xfc,0xad,0x6c,0x1e,
};
const unsigned char tc_ADa[] = {
 0x41,0x44,0x61,
};
const unsigned char tc_Ya[] = {
 0xe2,0x33,0x86,0x75,0x40,0x31,0x9e,0xc8,0x6e,0xae,0xcc,0x09,
 0xa8,0x5d,0xec,0x23,0x37,0x45,0xdb,0x72,0x9f,0x61,0xc3,0x6b,
 0xde,0x14,0xc0,0x34,0x20,0x09,0x94,0xfc,0x4b,0x6e,0x8d,0x26,
 0x30,0x08,0xc1,0x69,0x58,0x5f,0xd1,0xd1,0x86,0xd8,0xac,0x56,
 0x0c,0xb9,0xf7,0xad,0x0d,0x16,0x69,0x65,
};
const unsigned char tc_yb[] = {
 0x25,0x23,0xc9,0x69,0xf6,0x8f,0xa2,0xb2,0xae,0xa2,0x94,0xc2,
 0x53,0x9e,0xf3,0x6e,0xb1,0xe0,0x55,0x8a,0xbd,0x14,0x71,0x2a,
 0x78,0x28,0xf1,0x6a,0x85,0xed,0x2c,0x7e,0x77,0xe2,0xbd,0xd4,
 0x18,0x99,0x44,0x05,0xfb,0x1b,0x57,0xb6,0xbb,0xaa,0xdd,0x66,
 0x84,0x98,0x92,0xaa,0xc9,0xd8,0x14,0x02,
};
const unsigned char tc_ADb[] = {
 0x41,0x44,0x62,
};
const unsigned char tc_Yb[] = {
 0x50,0x62,0xa0,0xf3,0x34,0x78,0x91,0x4b,0xf1,0x62,0xa8,0x0d,
 0xad,0x39,0xb5,0xb2,0x66,0xc1,0xdd,0x02,0xf4,0x08,0x57,0x3b,
 0x41,0x82,0x7e,0x38,0x59,0x9b,0x68,0x2a,0xfb,0xf7,0xa0,0x73,
```

```
 0x5a,0xdf,0xd6,0x8c,0x39,0xab,0x49,0x94,0xfd,0x1b,0x03,0x48,
 0x46,0x27,0x0e,0x38,0x33,0x2b,0x4d,0xa9,
};
const unsigned char tc_K[] = {
 0xdc,0x9e,0xde,0xf7,0xc1,0x27,0xe7,0x9d,0x32,0xf2,0x58,0x4f,
 0x9f,0xcd,0x32,0x69,0x17,0x4f,0xe3,0x22,0x26,0xc2,0x08,0x29,
 0x63,0x87,0x9a,0x6d,0xea,0xfe,0xfb,0x9c,0x14,0xef,0xce,0xe9,
 0xfc,0x12,0x45,0x91,0x7a,0xd3,0x65,0x80,0x37,0xd2,0xd6,0x2a,
 0xff,0x2d,0x3f,0x76,0xfa,0x4f,0xca,0x99,
};
const unsigned char tc_ISK_IR[] = {
 0xa7,0x52,0x61,0x2f,0xe6,0xde,0xc5,0x42,0xe9,0x66,0x29,0xa6,
 0xeb,0x68,0xec,0xb9,0xbf,0xe2,0x25,0x72,0x24,0x97,0x5e,0x91,
 0x60,0x35,0xae,0xe7,0x47,0xc6,0xab,0xa3,0x2a,0xf2,0xe6,0xfe,
 0x25,0xee,0xb9,0x62,0x61,0xe6,0x14,0x01,0x00,0xed,0xcf,0x95,
 0x68,0x6e,0x0a,0xaa,0x13,0x40,0x26,0xb4,0xb5,0x25,0x4f,0xd2,
 0x71,0xb7,0xa4,0xda,
};
const unsigned char tc_ISK_SY[] = {
 0xe6,0xc7,0x9d,0x30,0xd4,0x38,0x1a,0x45,0xbd,0x47,0xb1,0x4b,
 0x76,0x9d,0x41,0x35,0x42,0x11,0xaf,0xf5,0x53,0xec,0xe9,0x37,
 0xd4,0xac,0x13,0x4f,0x09,0x84,0x48,0x96,0xc7,0x2a,0x72,0x3b,
 0x1f,0x1b,0x6d,0xa1,0xab,0x28,0x1d,0x75,0x9a,0x15,0x62,0x4d,
 0x2b,0xcd,0x0e,0x42,0x3b,0x70,0xb8,0xb8,0x50,0xa4,0xd0,0xed,
 0x12,0x6a,0x30,0x26,
};
```

### B.4.8. Test case for scalar_mult with valid inputs

```
s: (length: 56 bytes)
  dd1bc7015daabb7672129cc35a3ba815486b139deff9bdeca7a4fc61
  34323d34658761e90ff079972a7ca8aa5606498f4f4f0ebc0933a819
X: (length: 56 bytes)
  601431d5e51f43d422a92d3fb2373bde28217aab42524c341aa404ea
  ba5aa5541f7042dbb3253ce4c90f772b038a413dcb3a0f6bf3ae9e21
G.scalar_mult(s,decode(X)): (length: 56 bytes)
  388b35c60eb41b66085a2118316218681d78979d667702de105fdc1f
  21ffe884a577d795f45691781390a229a3bd7b527e831380f2f585a4
G.scalar_mult_vfy(s,X): (length: 56 bytes)
  388b35c60eb41b66085a2118316218681d78979d667702de105fdc1f
  21ffe884a577d795f45691781390a229a3bd7b527e831380f2f585a4
```

### B.4.9. Invalid inputs for scalar_mult_vfy

For these test cases scalar_mult_vfy(y,.) MUST return the representation of the neutral element G.I. When points Y_i1 or Y_i2 are included in MSGa or MSGb the protocol MUST abort.

```
s: (length: 56 bytes)
  dd1bc7015daabb7672129cc35a3ba815486b139deff9bdeca7a4fc61
  34323d34658761e90ff079972a7ca8aa5606498f4f4f0ebc0933a819
Y_i1: (length: 56 bytes)
  5f1431d5e51f43d422a92d3fb2373bde28217aab42524c341aa404ea
  ba5aa5541f7042dbb3253ce4c90f772b038a413dcb3a0f6bf3ae9e21
Y_i2 == G.I: (length: 56 bytes)
  00000000000000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000
G.scalar_mult_vfy(s,Y_i1) = G.scalar_mult_vfy(s,Y_i2) = G.I
```

### B.5. Test vector for CPace using group NIST P-256 and hash SHA-256

#### B.5.1. Test vectors for calculate_generator with group NIST P-256

```
  Inputs
    H   = SHA-256 with input block size 64 bytes.
    PRS = b'Password' ; ZPAD length: 23 ;
    DSI = b'CPaceP256_XMD:SHA-256_SSWU_NU_'
    DST = b'CPaceP256_XMD:SHA-256_SSWU_NU__DST'
    CI = b'\nAinitiator\nBresponder'
    CI = 0a41696e69746961746f720a42726573706f6e646572
    sid = 34b36454cab2e7842c389f7d88ecb7df
  Outputs
    generator_string(PRS,G.DSI,CI,sid,H.s_in_bytes):
    (length: 104 bytes)
      1e435061636550323536 5f584d443a5348412d3235365f535357555f
      4e555f0850617373776f726417000000000000000000000000000000
      0000000000000000160a41696e69746961746f720a42726573706f6e
      6465721034b36454cab2e7842c389f7d88ecb7df
    generator g: (length: 65 bytes)
      041b51433114e096c9d595f0955f5717a75169afb95557f4a6f51155
      035dee19c76887bce5c7c054fa1fe48a4a62c7fb96dc75e34259d2f7
      2b8d41f31b8e586bcd
```

#### B.5.2. Test vector for MSGa

```
  Inputs
    ADa = b'ADa'
    ya (big endian): (length: 32 bytes)
      37574cfbf1b95ff6a8e2d7be462d4d01e6dde2618f34f4de9df869b2
      4f532c5d
  Outputs
    Ya: (length: 65 bytes)
      04b75c1bcda84a0f324aabb7f25cf853ed7fb327c33f23db6aeb320d
      81df014649c2ac691925fce0eceac7dbc75eca25e6a1558066a610b4
      021488279e3b989d52
    Alternative correct value for Ya: g*(-ya):
    (length: 65 bytes)
      04b75c1bcda84a0f324aabb7f25cf853ed7fb327c33f23db6aeb320d
      81df0146493d5396e5da031f1415382438a135da195eaa7f9a59ef4b
      fdeb77d861c46762ad
    MSGa = lv_cat(Ya,ADa): (length: 70 bytes)
      4104b75c1bcda84a0f324aabb7f25cf853ed7fb327c33f23db6aeb32
      0d81df014649c2ac691925fce0eceac7dbc75eca25e6a1558066a610
      b4021488279e3b989d5203414461
```

### B.5.3. Test vector for MSGb

```
  Inputs
    ADb = b'ADb'
    yb (big endian): (length: 32 bytes)
      e5672fc9eb4e721f41d80181ec4c9fd9886668acc48024d33c82bb10
      2aecba52
  Outputs
    Yb: (length: 65 bytes)
      04bb2783a57337e74671f76452876b27839c0ea9e044e3aadaad2e64
      777ed27a90e80a99438e2f1c072462f2895c6dadf1b43867b92ffb65
      562b78c793947dcada
    Alternative correct value for Yb: g*(-yb):
    (length: 65 bytes)
      04bb2783a57337e74671f76452876b27839c0ea9e044e3aadaad2e64
      777ed27a9017f566bb71d0e3f9db9d0d76a392520e4bc79847d0049a
      a9d487386c6b823525
    MSGb = lv_cat(Yb,ADb): (length: 70 bytes)
      4104bb2783a57337e74671f76452876b27839c0ea9e044e3aadaad2e
      64777ed27a90e80a99438e2f1c072462f2895c6dadf1b43867b92ffb
      65562b78c793947dcada03414462
```

### B.5.4. Test vector for secret points K

```
    scalar_mult_vfy(ya,Yb): (length: 32 bytes)
      8fd12b283805750aeee6151bcd4211a6b71019e8fc416293ade24ed2
      bce12c39
    scalar_mult_vfy(yb,Ya): (length: 32 bytes)
      8fd12b283805750aeee6151bcd4211a6b71019e8fc416293ade24ed2
      bce12c39
```

### B.5.5. Test vector for ISK calculation initiator/responder

```
unordered cat of transcript : (length: 140 bytes)
  4104b75c1bcda84a0f324aabb7f25cf853ed7fb327c33f23db6aeb32
  0d81df014649c2ac691925fce0eceac7dbc75eca25e6a1558066a610
  b4021488279e3b989d52034144614104bb2783a57337e74671f76452
  876b27839c0ea9e044e3aadaad2e64777ed27a90e80a99438e2f1c07
  2462f2895c6dadf1b43867b92ffb65562b78c793947dcada03414462
DSI = G.DSI_ISK, b'CPaceP256_XMD:SHA-256_SSWU_NU__ISK':
(length: 34 bytes)
  4350616365503235365f584d443a5348412d3235365f535357555f4e
  555f5f49534b
lv_cat(DSI,sid,K)||MSGa||MSGb: (length: 225 bytes)
  2243506163655032353 65f584d443a5348412d3235365f535357555f
  4e555f5f49534b1034b36454cab2e7842c389f7d88ecb7df208fd12b
  283805750aeee6151bcd4211a6b71019e8fc416293ade24ed2bce12c
  394104b75c1bcda84a0f324aabb7f25cf853ed7fb327c33f23db6aeb
  320d81df014649c2ac691925fce0eceac7dbc75eca25e6a1558066a6
  10b4021488279e3b989d52034144614104bb2783a57337e74671f764
  52876b27839c0ea9e044e3aadaad2e64777ed27a90e80a99438e2f1c
  072462f2895c6dadf1b43867b92ffb65562b78c793947dcada034144
  62
ISK result: (length: 32 bytes)
  7ae1e916606e44652e3c0d7231198af6519226339c241e546afd0bbf
  48e1c96a
```

### B.5.6. Test vector for ISK calculation parallel execution

```
ordered cat of transcript : (length: 142 bytes)
  6f634104bb2783a57337e74671f76452876b27839c0ea9e044e3aada
  ad2e64777ed27a90e80a99438e2f1c072462f2895c6dadf1b43867b9
  2ffb65562b78c793947dcada034144624104b75c1bcda84a0f324aab
  b7f25cf853ed7fb327c33f23db6aeb320d81df014649c2ac691925fc
  e0eceac7dbc75eca25e6a1558066a610b4021488279e3b989d520341
  4461
DSI = G.DSI_ISK, b'CPaceP256_XMD:SHA-256_SSWU_NU__ISK':
(length: 34 bytes)
  435061636550323536 5f584d443a5348412d3235365f535357555f4e
  555f5f49534b
lv_cat(DSI,sid,K)||o_cat(MSGa,MSGb): (length: 227 bytes)
  224350616365503235365f584d443a5348412d3235365f535357555f
  4e555f5f49534b1034b36454cab2e7842c389f7d88ecb7df208fd12b
  283805750aeee6151bcd4211a6b71019e8fc416293ade24ed2bce12c
  396f634104bb2783a57337e74671f76452876b27839c0ea9e044e3aa
  daad2e64777ed27a90e80a99438e2f1c072462f2895c6dadf1b43867
  b92ffb65562b78c793947dcada034144624104b75c1bcda84a0f324a
  abb7f25cf853ed7fb327c33f23db6aeb320d81df014649c2ac691925
  fce0eceac7dbc75eca25e6a1558066a610b4021488279e3b989d5203
  414461
ISK result: (length: 32 bytes)
  5600a5c5bea5e92695dd68bd33d7f7b58326199c27c9b7326d76e4f9
  cb2fb276
```

### B.5.7. Corresponding C programming language initializers

```
const unsigned char tc_PRS[] = {
 0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const unsigned char tc_CI[] = {
 0x0a,0x41,0x69,0x6e,0x69,0x74,0x69,0x61,0x74,0x6f,0x72,0x0a,
 0x42,0x72,0x65,0x73,0x70,0x6f,0x6e,0x64,0x65,0x72,
};
const unsigned char tc_sid[] = {
 0x34,0xb3,0x64,0x54,0xca,0xb2,0xe7,0x84,0x2c,0x38,0x9f,0x7d,
 0x88,0xec,0xb7,0xdf,
};
const unsigned char tc_g[] = {
 0x04,0x1b,0x51,0x43,0x31,0x14,0xe0,0x96,0xc9,0xd5,0x95,0xf0,
 0x95,0x5f,0x57,0x17,0xa7,0x51,0x69,0xaf,0xb9,0x55,0x57,0xf4,
 0xa6,0xf5,0x11,0x55,0x03,0x5d,0xee,0x19,0xc7,0x68,0x87,0xbc,
 0xe5,0xc7,0xc0,0x54,0xfa,0x1f,0xe4,0x8a,0x4a,0x62,0xc7,0xfb,
 0x96,0xdc,0x75,0xe3,0x42,0x59,0xd2,0xf7,0x2b,0x8d,0x41,0xf3,
 0x1b,0x8e,0x58,0x6b,0xcd,
};
const unsigned char tc_ya[] = {
 0x37,0x57,0x4c,0xfb,0xf1,0xb9,0x5f,0xf6,0xa8,0xe2,0xd7,0xbe,
 0x46,0x2d,0x4d,0x01,0xe6,0xdd,0xe2,0x61,0x8f,0x34,0xf4,0xde,
 0x9d,0xf8,0x69,0xb2,0x4f,0x53,0x2c,0x5d,
};
const unsigned char tc_ADa[] = {
 0x41,0x44,0x61,
};
const unsigned char tc_Ya[] = {
 0x04,0xb7,0x5c,0x1b,0xcd,0xa8,0x4a,0x0f,0x32,0x4a,0xab,0xb7,
 0xf2,0x5c,0xf8,0x53,0xed,0x7f,0xb3,0x27,0xc3,0x3f,0x23,0xdb,
 0x6a,0xeb,0x32,0x0d,0x81,0xdf,0x01,0x46,0x49,0xc2,0xac,0x69,
 0x19,0x25,0xfc,0xe0,0xec,0xea,0xc7,0xdb,0xc7,0x5e,0xca,0x25,
 0xe6,0xa1,0x55,0x80,0x66,0xa6,0x10,0xb4,0x02,0x14,0x88,0x27,
 0x9e,0x3b,0x98,0x9d,0x52,
};
const unsigned char tc_yb[] = {
 0xe5,0x67,0x2f,0xc9,0xeb,0x4e,0x72,0x1f,0x41,0xd8,0x01,0x81,
 0xec,0x4c,0x9f,0xd9,0x88,0x66,0x68,0xac,0xc4,0x80,0x24,0xd3,
 0x3c,0x82,0xbb,0x10,0x2a,0xec,0xba,0x52,
};
const unsigned char tc_ADb[] = {
 0x41,0x44,0x62,
};
const unsigned char tc_Yb[] = {
 0x04,0xbb,0x27,0x83,0xa5,0x73,0x37,0xe7,0x46,0x71,0xf7,0x64,
 0x52,0x87,0x6b,0x27,0x83,0x9c,0x0e,0xa9,0xe0,0x44,0xe3,0xaa,
 0xda,0xad,0x2e,0x64,0x77,0x7e,0xd2,0x7a,0x90,0xe8,0x0a,0x99,
 0x43,0x8e,0x2f,0x1c,0x07,0x24,0x62,0xf2,0x89,0x5c,0x6d,0xad,
 0xf1,0xb4,0x38,0x67,0xb9,0x2f,0xfb,0x65,0x56,0x2b,0x78,0xc7,
```

```
 0x93,0x94,0x7d,0xca,0xda,
};
const unsigned char tc_K[] = {
 0x8f,0xd1,0x2b,0x28,0x38,0x05,0x75,0x0a,0xee,0xe6,0x15,0x1b,
 0xcd,0x42,0x11,0xa6,0xb7,0x10,0x19,0xe8,0xfc,0x41,0x62,0x93,
 0xad,0xe2,0x4e,0xd2,0xbc,0xe1,0x2c,0x39,
};
const unsigned char tc_ISK_IR[] = {
 0x7a,0xe1,0xe9,0x16,0x60,0x6e,0x44,0x65,0x2e,0x3c,0x0d,0x72,
 0x31,0x19,0x8a,0xf6,0x51,0x92,0x26,0x33,0x9c,0x24,0x1e,0x54,
 0x6a,0xfd,0x0b,0xbf,0x48,0xe1,0xc9,0x6a,
};
const unsigned char tc_ISK_SY[] = {
 0x56,0x00,0xa5,0xc5,0xbe,0xa5,0xe9,0x26,0x95,0xdd,0x68,0xbd,
 0x33,0xd7,0xf7,0xb5,0x83,0x26,0x19,0x9c,0x27,0xc9,0xb7,0x32,
 0x6d,0x76,0xe4,0xf9,0xcb,0x2f,0xb2,0x76,
};
```

### B.5.8. Test case for scalar_mult_vfy with correct inputs

```
s: (length: 32 bytes)
  f012501c091ff9b99a123fffe571d8bc01e8077ee581362e1bd21399
  0835643b
X: (length: 65 bytes)
  0424648eb986c2be0af636455cef0550671d6bcd8aa26e0d72ffa1b1
  fd12ba4e0f78da2b6d2184f31af39e566aef127014b6936c9a37346d
  10a4ab2514faef5831
G.scalar_mult(s,X) (full coordinates): (length: 65 bytes)
  04f5a191f078c87c36633b78c701751159d56c59f3fe9105b5720673
  470f303ab925b6a7fd1cdd8f649a21cf36b68d9e9c4a11919a951892
  519786104b27033757
G.scalar_mult_vfy(s,X) (only X-coordinate):
(length: 32 bytes)
  f5a191f078c87c36633b78c701751159d56c59f3fe9105b572067347
  0f303ab9
```

### B.5.9. Invalid inputs for scalar_mult_vfy

For these test cases scalar_mult_vfy(y,.) MUST return the representation of the neutral element G.I. When including Y_i1 or Y_i2 in MSGa or MSGb the protocol MUST abort.

```
s: (length: 32 bytes)
  f012501c091ff9b99a123fffe571d8bc01e8077ee581362e1bd21399
  0835643b
Y_i1: (length: 65 bytes)
  0424648eb986c2be0af636455cef0550671d6bcd8aa26e0d72ffa1b1
  fd12ba4e0f78da2b6d2184f31af39e566aef127014b6936c9a37346d
  10a4ab2514faef5857
Y_i2: (length: 1 bytes)
  00
G.scalar_mult_vfy(s,Y_i1) = G.scalar_mult_vfy(s,Y_i2) = G.I
```

### B.6. Test vector for CPace using group NIST P-384 and hash SHA-384

#### B.6.1. Test vectors for calculate_generator with group NIST P-384

```
  Inputs
    H   = SHA-384 with input block size 128 bytes.
    PRS = b'Password' ; ZPAD length: 87 ;
    DSI = b'CPaceP384_XMD:SHA-384_SSWU_NU_'
    DST = b'CPaceP384_XMD:SHA-384_SSWU_NU__DST'
    CI = b'\nAinitiator\nBresponder'
    CI = 0a41696e69746961746f720a42726573706f6e646572
    sid = 5b3773aa90e8f23c61563a4b645b276c
  Outputs
    generator_string(PRS,G.DSI,CI,sid,H.s_in_bytes):
    (length: 168 bytes)
      1e435061636550333834 5f584d443a5348412d3338345f535357555f
      4e555f0850617373776f726457000000000000000000000000000000
      00000000000000000000000000000000000000000000000000000000
      00000000000000000000000000000000000000000000000000000000
      00000000000000000000000000000160a41696e69746961746f72
      0a42726573706f6e646572105b3773aa90e8f23c61563a4b645b276c
    generator g: (length: 97 bytes)
      04f35a925fe82e54350e80b084a8013b1960cb3f73c49b0c2ae9b523
      997846ddd14c66f24f62223112cf35b866065f91ad86674cce2a2876
      84904e49f01287b54666bb518df2ea53cec627fa6e1283f14c6ed4bc
      d11b33fbb962da3e2e4ff1345c
```

### B.6.2. Test vector for MSGa

```
Inputs
  ADa = b'ADa'
  ya (big endian): (length: 48 bytes)
    ef433dd5ad142c860e7cb6400dd315d388d5ec5420c550e9d6f0907f
    375d988bc4d704837e43561c497e7dd93edcdb9d
Outputs
  Ya: (length: 97 bytes)
    04fd864c1a81f0e657a8a3f8e4ebafa421da712b6fb98f0abfa139ff
    971718cab474fa74c6a44b80a46468699280dd5d271252f3b9c05acc
    93dbd8b939152987cd5a8d1fb7b70c45512c993ec5456cc10f1797c9
    2fac2f1b7e363478a9ecd79e74
  Alternative correct value for Ya: g*(-ya):
  (length: 97 bytes)
    04fd864c1a81f0e657a8a3f8e4ebafa421da712b6fb98f0abfa139ff
    971718cab474fa74c6a44b80a46468699280dd5d27edad0c463fa533
    6c242746c6ead67832a572e04848f3baaed366c13aba933eefe86836
    cf53d0e481c9cb87571328618b
  MSGa = lv_cat(Ya,ADa): (length: 102 bytes)
    6104fd864c1a81f0e657a8a3f8e4ebafa421da712b6fb98f0abfa139
    ff971718cab474fa74c6a44b80a46468699280dd5d271252f3b9c05a
    cc93dbd8b939152987cd5a8d1fb7b70c45512c993ec5456cc10f1797
    c92fac2f1b7e363478a9ecd79e7403414461
```

### B.6.3. Test vector for MSGb

```
Inputs
  ADb = b'ADb'
  yb (big endian): (length: 48 bytes)
    50b0e36b95a2edfaa8342b843dddc90b175330f2399c1b36586dedda
    3c255975f30be6a750f9404fccc62a6323b5e471
Outputs
  Yb: (length: 97 bytes)
    04822b9874755c51adfdf624101eb4dc12a8ae433750be4fd6f4f7eb
    f6954ddb57837752a4effa4a5b44627a64b62a2db9d3c9c031c4ad37
    dbe7bf180d6bcba54feb4e84eeb876ebfa64a85d4c5ac2063dc05ba7
    26810824c41e1893faa9373a84
  Alternative correct value for Yb: g*(-yb):
  (length: 97 bytes)
    04822b9874755c51adfdf624101eb4dc12a8ae433750be4fd6f4f7eb
    f6954ddb57837752a4effa4a5b44627a64b62a2db92c363fce3b52c8
    241840e7f294345ab014b17b11478914059b57a2b3a53df9c13fa458
    d87ef7db3be1e76c0656c8c57b
  MSGb = lv_cat(Yb,ADb): (length: 102 bytes)
    6104822b9874755c51adfdf624101eb4dc12a8ae433750be4fd6f4f7
    ebf6954ddb57837752a4effa4a5b44627a64b62a2db9d3c9c031c4ad
    37dbe7bf180d6bcba54feb4e84eeb876ebfa64a85d4c5ac2063dc05b
    a726810824c41e1893faa9373a8403414462
```

### B.6.4. Test vector for secret points K

```
scalar_mult_vfy(ya,Yb): (length: 48 bytes)
  374290a54e07015baad085b311b18fbae1a20652e137c7c4bd13d565
  7d8b1ace028eb5acfba8c68d6211a79fff0965c9
scalar_mult_vfy(yb,Ya): (length: 48 bytes)
  374290a54e07015baad085b311b18fbae1a20652e137c7c4bd13d565
  7d8b1ace028eb5acfba8c68d6211a79fff0965c9
```

### B.6.5. Test vector for ISK calculation initiator/responder

```
unordered cat of transcript : (length: 204 bytes)
  6104fd864c1a81f0e657a8a3f8e4ebafa421da712b6fb98f0abfa139
  ff971718cab474fa74c6a44b80a46468699280dd5d271252f3b9c05a
  cc93dbd8b939152987cd5a8d1fb7b70c45512c993ec5456cc10f1797
  c92fac2f1b7e363478a9ecd79e74034144616104822b9874755c51ad
  fdf624101eb4dc12a8ae433750be4fd6f4f7ebf6954ddb57837752a4
  effa4a5b44627a64b62a2db9d3c9c031c4ad37dbe7bf180d6bcba54f
  eb4e84eeb876ebfa64a85d4c5ac2063dc05ba726810824c41e1893fa
  a9373a8403414462
DSI = G.DSI_ISK, b'CPaceP384_XMD:SHA-384_SSWU_NU__ISK':
(length: 34 bytes)
  43506163650333 38345f584d443a5348412d3338345f535357555f4e
  555f5f49534b
lv_cat(DSI,sid,K)||MSGa||MSGb: (length: 305 bytes)
  22435061636550333 8345f584d443a5348412d3338345f535357555f
  4e555f5f49534b105b3773aa90e8f23c61563a4b645b276c30374290
  a54e07015baad085b311b18fbae1a20652e137c7c4bd13d5657d8b1a
  ce028eb5acfba8c68d6211a79fff0965c96104fd864c1a81f0e657a8
  a3f8e4ebafa421da712b6fb98f0abfa139ff971718cab474fa74c6a4
  4b80a46468699280dd5d271252f3b9c05acc93dbd8b939152987cd5a
  8d1fb7b70c45512c993ec5456cc10f1797c92fac2f1b7e363478a9ec
  d79e74034144616104822b9874755c51adfdf624101eb4dc12a8ae43
  3750be4fd6f4f7ebf6954ddb57837752a4effa4a5b44627a64b62a2d
  b9d3c9c031c4ad37dbe7bf180d6bcba54feb4e84eeb876ebfa64a85d
  4c5ac2063dc05ba726810824c41e1893faa9373a8403414462
ISK result: (length: 48 bytes)
  a62d337820ce9cc1195a1adfb3c1efc2d844c0d8c6bc44bd060fe3cd
  d4ee8d2343aca0168c2b58478354a37d8d8856bd
```

### B.6.6. Test vector for ISK calculation parallel execution

```
ordered cat of transcript : (length: 206 bytes)
  6f636104fd864c1a81f0e657a8a3f8e4ebafa421da712b6fb98f0abf
  a139ff971718cab474fa74c6a44b80a46468699280dd5d271252f3b9
  c05acc93dbd8b939152987cd5a8d1fb7b70c45512c993ec5456cc10f
  1797c92fac2f1b7e363478a9ecd79e74034144616104822b9874755c
  51adfdf624101eb4dc12a8ae433750be4fd6f4f7ebf6954ddb578377
  52a4effa4a5b44627a64b62a2db9d3c9c031c4ad37dbe7bf180d6bcb
  a54feb4e84eeb876ebfa64a85d4c5ac2063dc05ba726810824c41e18
  93faa9373a8403414462
DSI = G.DSI_ISK, b'CPaceP384_XMD:SHA-384_SSWU_NU__ISK':
(length: 34 bytes)
  4350616365503338345f584d443a5348412d3338345f535357555f4e
  555f5f49534b
lv_cat(DSI,sid,K)||o_cat(MSGa,MSGb): (length: 307 bytes)
  2243506163655033383 45f584d443a5348412d3338345f535357555f
  4e555f5f49534b105b3773aa90e8f23c61563a4b645b276c30374290
  a54e07015baad085b311b18fbae1a20652e137c7c4bd13d5657d8b1a
  ce028eb5acfba8c68d6211a79fff0965c96f636104fd864c1a81f0e6
  57a8a3f8e4ebafa421da712b6fb98f0abfa139ff971718cab474fa74
  c6a44b80a46468699280dd5d271252f3b9c05acc93dbd8b939152987
  cd5a8d1fb7b70c45512c993ec5456cc10f1797c92fac2f1b7e363478
  a9ecd79e74034144616104822b9874755c51adfdf624101eb4dc12a8
  ae433750be4fd6f4f7ebf6954ddb57837752a4effa4a5b44627a64b6
  2a2db9d3c9c031c4ad37dbe7bf180d6bcba54feb4e84eeb876ebfa64
  a85d4c5ac2063dc05ba726810824c41e1893faa9373a8403414462
ISK result: (length: 48 bytes)
  eebf988a62b5c854f0ba32822ab45d23329bd1c78c84a4a0e1b40704
  c99c0a6f6c01c29af5fc6943254b883ce8a65ea1
```

### B.6.7. Corresponding C programming language initializers

```
const unsigned char tc_PRS[] = {
 0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const unsigned char tc_CI[] = {
 0x0a,0x41,0x69,0x6e,0x69,0x74,0x69,0x61,0x74,0x6f,0x72,0x0a,
 0x42,0x72,0x65,0x73,0x70,0x6f,0x6e,0x64,0x65,0x72,
};
const unsigned char tc_sid[] = {
 0x5b,0x37,0x73,0xaa,0x90,0xe8,0xf2,0x3c,0x61,0x56,0x3a,0x4b,
 0x64,0x5b,0x27,0x6c,
};
const unsigned char tc_g[] = {
 0x04,0xf3,0x5a,0x92,0x5f,0xe8,0x2e,0x54,0x35,0x0e,0x80,0xb0,
 0x84,0xa8,0x01,0x3b,0x19,0x60,0xcb,0x3f,0x73,0xc4,0x9b,0x0c,
 0x2a,0xe9,0xb5,0x23,0x99,0x78,0x46,0xdd,0xd1,0x4c,0x66,0xf2,
 0x4f,0x62,0x22,0x31,0x12,0xcf,0x35,0xb8,0x66,0x06,0x5f,0x91,
 0xad,0x86,0x67,0x4c,0xce,0x2a,0x28,0x76,0x84,0x90,0x4e,0x49,
 0xf0,0x12,0x87,0xb5,0x46,0x66,0xbb,0x51,0x8d,0xf2,0xea,0x53,
 0xce,0xc6,0x27,0xfa,0x6e,0x12,0x83,0xf1,0x4c,0x6e,0xd4,0xbc,
 0xd1,0x1b,0x33,0xfb,0xb9,0x62,0xda,0x3e,0x2e,0x4f,0xf1,0x34,
 0x5c,
};
const unsigned char tc_ya[] = {
 0xef,0x43,0x3d,0xd5,0xad,0x14,0x2c,0x86,0x0e,0x7c,0xb6,0x40,
 0x0d,0xd3,0x15,0xd3,0x88,0xd5,0xec,0x54,0x20,0xc5,0x50,0xe9,
 0xd6,0xf0,0x90,0x7f,0x37,0x5d,0x98,0x8b,0xc4,0xd7,0x04,0x83,
 0x7e,0x43,0x56,0x1c,0x49,0x7e,0x7d,0xd9,0x3e,0xdc,0xdb,0x9d,
};
const unsigned char tc_ADa[] = {
 0x41,0x44,0x61,
};
const unsigned char tc_Ya[] = {
 0x04,0xfd,0x86,0x4c,0x1a,0x81,0xf0,0xe6,0x57,0xa8,0xa3,0xf8,
 0xe4,0xeb,0xaf,0xa4,0x21,0xda,0x71,0x2b,0x6f,0xb9,0x8f,0x0a,
 0xbf,0xa1,0x39,0xff,0x97,0x17,0x18,0xca,0xb4,0x74,0xfa,0x74,
 0xc6,0xa4,0x4b,0x80,0xa4,0x64,0x68,0x69,0x92,0x80,0xdd,0x5d,
 0x27,0x12,0x52,0xf3,0xb9,0xc0,0x5a,0xcc,0x93,0xdb,0xd8,0xb9,
 0x39,0x15,0x29,0x87,0xcd,0x5a,0x8d,0x1f,0xb7,0xb7,0x0c,0x45,
 0x51,0x2c,0x99,0x3e,0xc5,0x45,0x6c,0xc1,0x0f,0x17,0x97,0xc9,
 0x2f,0xac,0x2f,0x1b,0x7e,0x36,0x34,0x78,0xa9,0xec,0xd7,0x9e,
 0x74,
};
const unsigned char tc_yb[] = {
 0x50,0xb0,0xe3,0x6b,0x95,0xa2,0xed,0xfa,0xa8,0x34,0x2b,0x84,
 0x3d,0xdd,0xc9,0x0b,0x17,0x53,0x30,0xf2,0x39,0x9c,0x1b,0x36,
 0x58,0x6d,0xed,0xda,0x3c,0x25,0x59,0x75,0xf3,0x0b,0xe6,0xa7,
 0x50,0xf9,0x40,0x4f,0xcc,0xc6,0x2a,0x63,0x23,0xb5,0xe4,0x71,
};
const unsigned char tc_ADb[] = {
```

```
 0x41,0x44,0x62,
};
const unsigned char tc_Yb[] = {
 0x04,0x82,0x2b,0x98,0x74,0x75,0x5c,0x51,0xad,0xfd,0xf6,0x24,
 0x10,0x1e,0xb4,0xdc,0x12,0xa8,0xae,0x43,0x37,0x50,0xbe,0x4f,
 0xd6,0xf4,0xf7,0xeb,0xf6,0x95,0x4d,0xdb,0x57,0x83,0x77,0x52,
 0xa4,0xef,0xfa,0x4a,0x5b,0x44,0x62,0x7a,0x64,0xb6,0x2a,0x2d,
 0xb9,0xd3,0xc9,0xc0,0x31,0xc4,0xad,0x37,0xdb,0xe7,0xbf,0x18,
 0x0d,0x6b,0xcb,0xa5,0x4f,0xeb,0x4e,0x84,0xee,0xb8,0x76,0xeb,
 0xfa,0x64,0xa8,0x5d,0x4c,0x5a,0xc2,0x06,0x3d,0xc0,0x5b,0xa7,
 0x26,0x81,0x08,0x24,0xc4,0x1e,0x18,0x93,0xfa,0xa9,0x37,0x3a,
 0x84,
};
const unsigned char tc_K[] = {
 0x37,0x42,0x90,0xa5,0x4e,0x07,0x01,0x5b,0xaa,0xd0,0x85,0xb3,
 0x11,0xb1,0x8f,0xba,0xe1,0xa2,0x06,0x52,0xe1,0x37,0xc7,0xc4,
 0xbd,0x13,0xd5,0x65,0x7d,0x8b,0x1a,0xce,0x02,0x8e,0xb5,0xac,
 0xfb,0xa8,0xc6,0x8d,0x62,0x11,0xa7,0x9f,0xff,0x09,0x65,0xc9,
};
const unsigned char tc_ISK_IR[] = {
 0xa6,0x2d,0x33,0x78,0x20,0xce,0x9c,0xc1,0x19,0x5a,0x1a,0xdf,
 0xb3,0xc1,0xef,0xc2,0xd8,0x44,0xc0,0xd8,0xc6,0xbc,0x44,0xbd,
 0x06,0x0f,0xe3,0xcd,0xd4,0xee,0x8d,0x23,0x43,0xac,0xa0,0x16,
 0x8c,0x2b,0x58,0x47,0x83,0x54,0xa3,0x7d,0x8d,0x88,0x56,0xbd,
};
const unsigned char tc_ISK_SY[] = {
 0xee,0xbf,0x98,0x8a,0x62,0xb5,0xc8,0x54,0xf0,0xba,0x32,0x82,
 0x2a,0xb4,0x5d,0x23,0x32,0x9b,0xd1,0xc7,0x8c,0x84,0xa4,0xa0,
 0xe1,0xb4,0x07,0x04,0xc9,0x9c,0x0a,0x6f,0x6c,0x01,0xc2,0x9a,
 0xf5,0xfc,0x69,0x43,0x25,0x4b,0x88,0x3c,0xe8,0xa6,0x5e,0xa1,
};
```

### B.6.8. Test case for scalar_mult_vfy with correct inputs

```
s: (length: 48 bytes)
  6e8a99a5cdd408eae98e1b8aed286e7b12adbbdac7f2c628d9060ce9
  2ae0d90bd57a564fd3500fbcce3425dc94ba0ade
X: (length: 97 bytes)
  045b4cd53c4506cc04ba4c44f2762d5d32c3e55df25b8baa5571b165
  7ad9576efea8259f0684de065a470585b4be876748c7797054f3defe
  f21b77f83d53bac57c89d52aa4d6dd5872bd281989b138359698009f
  8ac1f301538badcce9d9f4036e
G.scalar_mult(s,X) (full coordinates): (length: 97 bytes)
  0465c28db05fd9f9a93651c5cc31eae49c4e5246b46489b8f6105873
  3173a033cda76c3e3ea5352b804e67fdbe2e334be8245dad5c8c993e
  63bacf0456478f29b71b6c859f13676f84ff150d2741f028f560584a
  0bdba19a63df62c08949c2fd6d
G.scalar_mult_vfy(s,X) (only X-coordinate):
(length: 48 bytes)
  65c28db05fd9f9a93651c5cc31eae49c4e5246b46489b8f610587331
  73a033cda76c3e3ea5352b804e67fdbe2e334be8
```

### B.6.9. Invalid inputs for scalar_mult_vfy

For these test cases scalar_mult_vfy(y,.) MUST return the representation of the neutral element G.I. When including Y_i1 or Y_i2 in MSGa or MSGb the protocol MUST abort.

```
s: (length: 48 bytes)
  6e8a99a5cdd408eae98e1b8aed286e7b12adbbdac7f2c628d9060ce9
  2ae0d90bd57a564fd3500fbcce3425dc94ba0ade
Y_i1: (length: 97 bytes)
  045b4cd53c4506cc04ba4c44f2762d5d32c3e55df25b8baa5571b165
  7ad9576efea8259f0684de065a470585b4be876748c7797054f3defe
  f21b77f83d53bac57c89d52aa4d6dd5872bd281989b138359698009f
  8ac1f301538badcce9d9f40302
Y_i2: (length: 1 bytes)
  00
G.scalar_mult_vfy(s,Y_i1) = G.scalar_mult_vfy(s,Y_i2) = G.I
```

### B.7. Test vector for CPace using group NIST P-521 and hash SHA-512

#### B.7.1. Test vectors for calculate_generator with group NIST P-521

```
  Inputs
    H   = SHA-512 with input block size 128 bytes.
    PRS = b'Password' ; ZPAD length: 87 ;
    DSI = b'CPaceP521_XMD:SHA-512_SSWU_NU_'
    DST = b'CPaceP521_XMD:SHA-512_SSWU_NU__DST'
    CI = b'\nAinitiator\nBresponder'
    CI = 0a41696e69746961746f720a42726573706f6e646572
    sid = 7e4b4791d6a8ef019b936c79fb7f2c57
  Outputs
    generator_string(PRS,G.DSI,CI,sid,H.s_in_bytes):
    (length: 168 bytes)
      1e435061636550353231 5f584d443a5348412d3531325f535357555f
      4e555f0850617373776f726457000000000000000000000000000000
      0000000000000000000000000000000000000000000000000000000000
      0000000000000000000000000000000000000000000000000000000000
      0000000000000000000000000000160a41696e69746961746f72
      0a42726573706f6e646572107e4b4791d6a8ef019b936c79fb7f2c57
    generator g: (length: 133 bytes)
      0400dc927958f0b69ccad8fb67ef008905354b58c7c9c92ad50060a9
      e6afb10437d6ca8a26164e8573702b897275a25d05ed4407af2a3849
      86dca7e243b92c5dd500d40057012121a9c8e34373fa619f918f7d47
      9c23f85f0485379ef0f05284398de26653b49a155324c9d7b138be84
      d0b49bb58e232b7bf697798de6ee8afd6b92b6fa2f
```

### B.7.2. Test vector for MSGa

```
  Inputs
    ADa = b'ADa'
    ya (big endian): (length: 66 bytes)
      006367e9c2aeff9f1db19af600cca73343d47cbe446cebbd1ccd783f
      82755a872da86fd0707eb3767c6114f1803deb62d63bdd1e613f67e6
      3e8c141ee5310e3ee819
  Outputs
    Ya: (length: 133 bytes)
      04003701ec35caafa3dd416cad29ba1774551f9d2ed89f7e1065706d
      ca230b86a11d02e4cee8b3fde64380d4a05983167d8a2414bc594ad5
      286c068792ab7ca60ff6ea00919c41c00e789dabc2f42fd94178d7bf
      d8fbe1aff1c1854b3dafb3a0ea13f5a5fc1703860f022bd271740469
      bb322b07c179c7c225499b31727c0ea3ee65578634
    Alternative correct value for Ya: g*(-ya):
    (length: 133 bytes)
      04003701ec35caafa3dd416cad29ba1774551f9d2ed89f7e1065706d
      ca230b86a11d02e4cee8b3fde64380d4a05983167d8a2414bc594ad5
      286c068792ab7ca60ff6ea016e63be3ff18762543d0bd026be872840
      27041e500e3e7ab4c2504c5f15ec0a5a03e8fc79f0fdd42d8e8bfb96
      44cdd4f83e86383ddab664ce8d83f15c119aa879cb
    MSGa = lv_cat(Ya,ADa): (length: 139 bytes)
      850104003701ec35caafa3dd416cad29ba1774551f9d2ed89f7e1065
      706dca230b86a11d02e4cee8b3fde64380d4a05983167d8a2414bc59
      4ad5286c068792ab7ca60ff6ea00919c41c00e789dabc2f42fd94178
      d7bfd8fbe1aff1c1854b3dafb3a0ea13f5a5fc1703860f022bd27174
      0469bb322b07c179c7c225499b31727c0ea3ee6557863403414461
```

### B.7.3. Test vector for MSGb

```
  Inputs
    ADb = b'ADb'
    yb (big endian): (length: 66 bytes)
      009227bf8dc741dacc9422f8bf3c0e96fce9587bc562eaafe0dc5f6f
      82f28594e4a6f98553560c62b75fa4abb198cecbbb86ebd41b0ea025
      4cde78ac68d39a240ae7
  Outputs
    Yb: (length: 133 bytes)
      0400f5cb68bf0117bd1a65412a2bc800af92013f9969cf546e1ea6d3
      bcf08643fdc482130aec1eecc33a2b5f33600be51295047fa3399fa2
      82cc1a78de91f3a4e30b5d01a085b453f22bf3dc947386b042e5fc4e
      c691fee47fe3c3ec6408c22a17c26bc0ab73940910614d6fcee32daf
      bfd2d340d6e382d71b1fc763d7cec502fbcbcf93b4
    Alternative correct value for Yb: g*(-yb):
    (length: 133 bytes)
      0400f5cb68bf0117bd1a65412a2bc800af92013f9969cf546e1ea6d3
      bcf08643fdc482130aec1eecc33a2b5f33600be51295047fa3399fa2
      82cc1a78de91f3a4e30b5d005f7a4bac0dd40c236b8c794fbd1a03b1
      396e011b801c3c139bf73dd5e83d943f548c6bf6ef9eb290311cd250
      402d2cbf291c7d28e4e0389c28313afd0434306c4b
    MSGb = lv_cat(Yb,ADb): (length: 139 bytes)
      85010400f5cb68bf0117bd1a65412a2bc800af92013f9969cf546e1e
      a6d3bcf08643fdc482130aec1eecc33a2b5f33600be51295047fa339
      9fa282cc1a78de91f3a4e30b5d01a085b453f22bf3dc947386b042e5
      fc4ec691fee47fe3c3ec6408c22a17c26bc0ab73940910614d6fcee3
      2dafbfd2d340d6e382d71b1fc763d7cec502fbcbcf93b403414462
```

### B.7.4. Test vector for secret points K

```
    scalar_mult_vfy(ya,Yb): (length: 66 bytes)
      00503e75e38e012a6dc6f3561980e4cf540dbcff3de3a4a6f09d79c3
      2cc45764d3a6605eb45df1dc63fb7937b7879f2820da1b3266b69fa0
      99bf8720dd8f6a07e8ed
    scalar_mult_vfy(yb,Ya): (length: 66 bytes)
      00503e75e38e012a6dc6f3561980e4cf540dbcff3de3a4a6f09d79c3
      2cc45764d3a6605eb45df1dc63fb7937b7879f2820da1b3266b69fa0
      99bf8720dd8f6a07e8ed
```

### B.7.5. Test vector for ISK calculation initiator/responder

```
unordered cat of transcript : (length: 278 bytes)
  850104003701ec35caafa3dd416cad29ba1774551f9d2ed89f7e1065
  706dca230b86a11d02e4cee8b3fde64380d4a05983167d8a2414bc59
  4ad5286c068792ab7ca60ff6ea00919c41c00e789dabc2f42fd94178
  d7bfd8fbe1aff1c1854b3dafb3a0ea13f5a5fc1703860f022bd27174
  0469bb322b07c179c7c225499b31727c0ea3ee655786340341446185
  010400f5cb68bf0117bd1a65412a2bc800af92013f9969cf546e1ea6
  d3bcf08643fdc482130aec1eecc33a2b5f33600be51295047fa3399f
  a282cc1a78de91f3a4e30b5d01a085b453f22bf3dc947386b042e5fc
  4ec691fee47fe3c3ec6408c22a17c26bc0ab73940910614d6fcee32d
  afbfd2d340d6e382d71b1fc763d7cec502fbcbcf93b403414462
DSI = G.DSI_ISK, b'CPaceP521_XMD:SHA-512_SSWU_NU__ISK':
(length: 34 bytes)
  4350616365503532315f584d443a5348412d3531325f535357555f4e
  555f5f49534b
lv_cat(DSI,sid,K)||MSGa||MSGb: (length: 397 bytes)
  224350616365503532315f584d443a5348412d3531325f535357555f
  4e555f5f49534b107e4b4791d6a8ef019b936c79fb7f2c574200503e
  75e38e012a6dc6f3561980e4cf540dbcff3de3a4a6f09d79c32cc457
  64d3a6605eb45df1dc63fb7937b7879f2820da1b3266b69fa099bf87
  20dd8f6a07e8ed850104003701ec35caafa3dd416cad29ba1774551f
  9d2ed89f7e1065706dca230b86a11d02e4cee8b3fde64380d4a05983
  167d8a2414bc594ad5286c068792ab7ca60ff6ea00919c41c00e789d
  abc2f42fd94178d7bfd8fbe1aff1c1854b3dafb3a0ea13f5a5fc1703
  860f022bd271740469bb322b07c179c7c225499b31727c0ea3ee6557
  863403414461850104 00f5cb68bf0117bd1a65412a2bc800af92013f
  9969cf546e1ea6d3bcf08643fdc482130aec1eecc33a2b5f33600be5
  1295047fa3399fa282cc1a78de91f3a4e30b5d01a085b453f22bf3dc
  947386b042e5fc4ec691fee47fe3c3ec6408c22a17c26bc0ab739409
  10614d6fcee32dafbfd2d340d6e382d71b1fc763d7cec502fbcbcf93
  b403414462
ISK result: (length: 64 bytes)
  ed208a15af3ef8a67a5cac4acb360d03154570e3b1b1c54867f53a72
  53cb919d13aa47efc647375be2250cb39ad965afa4ddfcb6be47d586
  d28c7eef6d654525
```

### B.7.6. Test vector for ISK calculation parallel execution

```
ordered cat of transcript : (length: 280 bytes)
  6f6385010400f5cb68bf0117bd1a65412a2bc800af92013f9969cf54
  6e1ea6d3bcf08643fdc482130aec1eecc33a2b5f33600be51295047f
  a3399fa282cc1a78de91f3a4e30b5d01a085b453f22bf3dc947386b0
  42e5fc4ec691fee47fe3c3ec6408c22a17c26bc0ab73940910614d6f
  cee32dafbfd2d340d6e382d71b1fc763d7cec502fbcbcf93b4034144
  62850104003701ec35caafa3dd416cad29ba1774551f9d2ed89f7e10
  65706dca230b86a11d02e4cee8b3fde64380d4a05983167d8a2414bc
  594ad5286c068792ab7ca60ff6ea00919c41c00e789dabc2f42fd941
  78d7bfd8fbe1aff1c1854b3dafb3a0ea13f5a5fc1703860f022bd271
  740469bb322b07c179c7c225499b31727c0ea3ee6557863403414461
DSI = G.DSI_ISK, b'CPaceP521_XMD:SHA-512_SSWU_NU__ISK':
(length: 34 bytes)
  43506163655035323 15f584d443a5348412d3531325f535357555f4e
  555f5f49534b
lv_cat(DSI,sid,K)||o_cat(MSGa,MSGb): (length: 399 bytes)
  2243506163655035323 15f584d443a5348412d3531325f535357555f
  4e555f5f49534b107e4b4791d6a8ef019b936c79fb7f2c574200503e
  75e38e012a6dc6f3561980e4cf540dbcff3de3a4a6f09d79c32cc457
  64d3a6605eb45df1dc63fb7937b7879f2820da1b3266b69fa099bf87
  20dd8f6a07e8ed6f6385010400f5cb68bf0117bd1a65412a2bc800af
  92013f9969cf546e1ea6d3bcf08643fdc482130aec1eecc33a2b5f33
  600be51295047fa3399fa282cc1a78de91f3a4e30b5d01a085b453f2
  2bf3dc947386b042e5fc4ec691fee47fe3c3ec6408c22a17c26bc0ab
  73940910614d6fcee32dafbfd2d340d6e382d71b1fc763d7cec502fb
  cbcf93b4034144628501040037 01ec35caafa3dd416cad29ba177455
  1f9d2ed89f7e1065706dca230b86a11d02e4cee8b3fde64380d4a059
  83167d8a2414bc594ad5286c068792ab7ca60ff6ea00919c41c00e78
  9dabc2f42fd94178d7bfd8fbe1aff1c1854b3dafb3a0ea13f5a5fc17
  03860f022bd271740469bb322b07c179c7c225499b31727c0ea3ee65
  57863403414461
ISK result: (length: 64 bytes)
  e7b10b6da531d9a8fd47fdd08441e8bb803d16c59a93e366d5cd9a10
  277bbc543d943182889154704d80f2b0756ed62da87e0eb4e6d07920
  480100d5e800ca85
```

### B.7.7. Corresponding C programming language initializers

```
const unsigned char tc_PRS[] = {
 0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const unsigned char tc_CI[] = {
 0x0a,0x41,0x69,0x6e,0x69,0x74,0x69,0x61,0x74,0x6f,0x72,0x0a,
 0x42,0x72,0x65,0x73,0x70,0x6f,0x6e,0x64,0x65,0x72,
};
const unsigned char tc_sid[] = {
 0x7e,0x4b,0x47,0x91,0xd6,0xa8,0xef,0x01,0x9b,0x93,0x6c,0x79,
 0xfb,0x7f,0x2c,0x57,
};
const unsigned char tc_g[] = {
 0x04,0x00,0xdc,0x92,0x79,0x58,0xf0,0xb6,0x9c,0xca,0xd8,0xfb,
 0x67,0xef,0x00,0x89,0x05,0x35,0x4b,0x58,0xc7,0xc9,0xc9,0x2a,
 0xd5,0x00,0x60,0xa9,0xe6,0xaf,0xb1,0x04,0x37,0xd6,0xca,0x8a,
 0x26,0x16,0x4e,0x85,0x73,0x70,0x2b,0x89,0x72,0x75,0xa2,0x5d,
 0x05,0xed,0x44,0x07,0xaf,0x2a,0x38,0x49,0x86,0xdc,0xa7,0xe2,
 0x43,0xb9,0x2c,0x5d,0xd5,0x00,0xd4,0x00,0x57,0x01,0x21,0x21,
 0xa9,0xc8,0xe3,0x43,0x73,0xfa,0x61,0x9f,0x91,0x8f,0x7d,0x47,
 0x9c,0x23,0xf8,0x5f,0x04,0x85,0x37,0x9e,0xf0,0xf0,0x52,0x84,
 0x39,0x8d,0xe2,0x66,0x53,0xb4,0x9a,0x15,0x53,0x24,0xc9,0xd7,
 0xb1,0x38,0xbe,0x84,0xd0,0xb4,0x9b,0xb5,0x8e,0x23,0x2b,0x7b,
 0xf6,0x97,0x79,0x8d,0xe6,0xee,0x8a,0xfd,0x6b,0x92,0xb6,0xfa,
 0x2f,
};
const unsigned char tc_ya[] = {
 0x00,0x63,0x67,0xe9,0xc2,0xae,0xff,0x9f,0x1d,0xb1,0x9a,0xf6,
 0x00,0xcc,0xa7,0x33,0x43,0xd4,0x7c,0xbe,0x44,0x6c,0xeb,0xbd,
 0x1c,0xcd,0x78,0x3f,0x82,0x75,0x5a,0x87,0x2d,0xa8,0x6f,0xd0,
 0x70,0x7e,0xb3,0x76,0x7c,0x61,0x14,0xf1,0x80,0x3d,0xeb,0x62,
 0xd6,0x3b,0xdd,0x1e,0x61,0x3f,0x67,0xe6,0x3e,0x8c,0x14,0x1e,
 0xe5,0x31,0x0e,0x3e,0xe8,0x19,
};
const unsigned char tc_ADa[] = {
 0x41,0x44,0x61,
};
const unsigned char tc_Ya[] = {
 0x04,0x00,0x37,0x01,0xec,0x35,0xca,0xaf,0xa3,0xdd,0x41,0x6c,
 0xad,0x29,0xba,0x17,0x74,0x55,0x1f,0x9d,0x2e,0xd8,0x9f,0x7e,
 0x10,0x65,0x70,0x6d,0xca,0x23,0x0b,0x86,0xa1,0x1d,0x02,0xe4,
 0xce,0xe8,0xb3,0xfd,0xe6,0x43,0x80,0xd4,0xa0,0x59,0x83,0x16,
 0x7d,0x8a,0x24,0x14,0xbc,0x59,0x4a,0xd5,0x28,0x6c,0x06,0x87,
 0x92,0xab,0x7c,0xa6,0x0f,0xf6,0xea,0x00,0x91,0x9c,0x41,0xc0,
 0x0e,0x78,0x9d,0xab,0xc2,0xf4,0x2f,0xd9,0x41,0x78,0xd7,0xbf,
 0xd8,0xfb,0xe1,0xaf,0xf1,0xc1,0x85,0x4b,0x3d,0xaf,0xb3,0xa0,
 0xea,0x13,0xf5,0xa5,0xfc,0x17,0x03,0x86,0x0f,0x02,0x2b,0xd2,
 0x71,0x74,0x04,0x69,0xbb,0x32,0x2b,0x07,0xc1,0x79,0xc7,0xc2,
 0x25,0x49,0x9b,0x31,0x72,0x7c,0x0e,0xa3,0xee,0x65,0x57,0x86,
 0x34,
```

```
};
const unsigned char tc_yb[] = {
 0x00,0x92,0x27,0xbf,0x8d,0xc7,0x41,0xda,0xcc,0x94,0x22,0xf8,
 0xbf,0x3c,0x0e,0x96,0xfc,0xe9,0x58,0x7b,0xc5,0x62,0xea,0xaf,
 0xe0,0xdc,0x5f,0x6f,0x82,0xf2,0x85,0x94,0xe4,0xa6,0xf9,0x85,
 0x53,0x56,0x0c,0x62,0xb7,0x5f,0xa4,0xab,0xb1,0x98,0xce,0xcb,
 0xbb,0x86,0xeb,0xd4,0x1b,0x0e,0xa0,0x25,0x4c,0xde,0x78,0xac,
 0x68,0xd3,0x9a,0x24,0x0a,0xe7,
};
const unsigned char tc_ADb[] = {
 0x41,0x44,0x62,
};
const unsigned char tc_Yb[] = {
 0x04,0x00,0xf5,0xcb,0x68,0xbf,0x01,0x17,0xbd,0x1a,0x65,0x41,
 0x2a,0x2b,0xc8,0x00,0xaf,0x92,0x01,0x3f,0x99,0x69,0xcf,0x54,
 0x6e,0x1e,0xa6,0xd3,0xbc,0xf0,0x86,0x43,0xfd,0xc4,0x82,0x13,
 0x0a,0xec,0x1e,0xec,0xc3,0x3a,0x2b,0x5f,0x33,0x60,0x0b,0xe5,
 0x12,0x95,0x04,0x7f,0xa3,0x39,0x9f,0xa2,0x82,0xcc,0x1a,0x78,
 0xde,0x91,0xf3,0xa4,0xe3,0x0b,0x5d,0x01,0xa0,0x85,0xb4,0x53,
 0xf2,0x2b,0xf3,0xdc,0x94,0x73,0x86,0xb0,0x42,0xe5,0xfc,0x4e,
 0xc6,0x91,0xfe,0xe4,0x7f,0xe3,0xc3,0xec,0x64,0x08,0xc2,0x2a,
 0x17,0xc2,0x6b,0xc0,0xab,0x73,0x94,0x09,0x10,0x61,0x4d,0x6f,
 0xce,0xe3,0x2d,0xaf,0xbf,0xd2,0xd3,0x40,0xd6,0xe3,0x82,0xd7,
 0x1b,0x1f,0xc7,0x63,0xd7,0xce,0xc5,0x02,0xfb,0xcb,0xcf,0x93,
 0xb4,
};
const unsigned char tc_K[] = {
 0x00,0x50,0x3e,0x75,0xe3,0x8e,0x01,0x2a,0x6d,0xc6,0xf3,0x56,
 0x19,0x80,0xe4,0xcf,0x54,0x0d,0xbc,0xff,0x3d,0xe3,0xa4,0xa6,
 0xf0,0x9d,0x79,0xc3,0x2c,0xc4,0x57,0x64,0xd3,0xa6,0x60,0x5e,
 0xb4,0x5d,0xf1,0xdc,0x63,0xfb,0x79,0x37,0xb7,0x87,0x9f,0x28,
 0x20,0xda,0x1b,0x32,0x66,0xb6,0x9f,0xa0,0x99,0xbf,0x87,0x20,
 0xdd,0x8f,0x6a,0x07,0xe8,0xed,
};
const unsigned char tc_ISK_IR[] = {
 0xed,0x20,0x8a,0x15,0xaf,0x3e,0xf8,0xa6,0x7a,0x5c,0xac,0x4a,
 0xcb,0x36,0x0d,0x03,0x15,0x45,0x70,0xe3,0xb1,0xb1,0xc5,0x48,
 0x67,0xf5,0x3a,0x72,0x53,0xcb,0x91,0x9d,0x13,0xaa,0x47,0xef,
 0xc6,0x47,0x37,0x5b,0xe2,0x25,0x0c,0xb3,0x9a,0xd9,0x65,0xaf,
 0xa4,0xdd,0xfc,0xb6,0xbe,0x47,0xd5,0x86,0xd2,0x8c,0x7e,0xef,
 0x6d,0x65,0x45,0x25,
};
const unsigned char tc_ISK_SY[] = {
 0xe7,0xb1,0x0b,0x6d,0xa5,0x31,0xd9,0xa8,0xfd,0x47,0xfd,0xd0,
 0x84,0x41,0xe8,0xbb,0x80,0x3d,0x16,0xc5,0x9a,0x93,0xe3,0x66,
 0xd5,0xcd,0x9a,0x10,0x27,0x7b,0xbc,0x54,0x3d,0x94,0x31,0x82,
 0x88,0x91,0x54,0x70,0x4d,0x80,0xf2,0xb0,0x75,0x6e,0xd6,0x2d,
 0xa8,0x7e,0x0e,0xb4,0xe6,0xd0,0x79,0x20,0x48,0x01,0x00,0xd5,
```

```
 0xe8,0x00,0xca,0x85,
};
```

### B.7.8. Test case for scalar_mult_vfy with correct inputs

```
s: (length: 66 bytes)
  0182dd7925f1753419e4bf83429763acd37d64000cd5a175edf53a15
  87dd986bc95acc1506991702b6ba1a9ee2458fee8efc00198cf0088c
  480965ef65ff2048b856
X: (length: 133 bytes)
  0400dc5078b24c4af1620cc10fbecc6cd8cf1cab0b011efb73c782f2
  26dc21c7ca7eb406be74a69ecba5b4a87c07cfc6e687b4beca9a6eda
  c95940a3b4120573b26a80005e697833b0ba285fce7b3f1f25243008
  860b8f1de710a0dcc05b0d20341efe90eb2bcca26797c2d85ae6ca74
  c00696cb1b13e40bda15b27964d7670576647bfab9
G.scalar_mult(s,X) (full coordinates): (length: 133 bytes)
  040122f88ce73ec5aa2d1c8c5d04148760c3d97ba87daa10d8cb8bb7
  c73cf6e951fc922721bf1437995cfb13e132a78beb86389e60d3517c
  df6d99a8a2d6db19ef27bd0055af9e8ddcf337ce0a7c22a9c8099bc4
  a44faeded1eb72effd26e4f322217b67d60b944b267b3df5046078fd
  577f1785728f49b241fd5e8c83223a994a2d219281
G.scalar_mult_vfy(s,X) (only X-coordinate):
(length: 66 bytes)
  0122f88ce73ec5aa2d1c8c5d04148760c3d97ba87daa10d8cb8bb7c7
  3cf6e951fc922721bf1437995cfb13e132a78beb86389e60d3517cdf
  6d99a8a2d6db19ef27bd
```

### B.7.9. Invalid inputs for scalar_mult_vfy

For these test cases scalar_mult_vfy(y,.) MUST return the representation of the neutral element G.I. When including Y_i1 or Y_i2 in MSGa or MSGb the protocol MUST abort.

```
s: (length: 66 bytes)
  0182dd7925f1753419e4bf83429763acd37d64000cd5a175edf53a15
  87dd986bc95acc1506991702b6ba1a9ee2458fee8efc00198cf0088c
  480965ef65ff2048b856
Y_i1: (length: 133 bytes)
  0400dc5078b24c4af1620cc10fbecc6cd8cf1cab0b011efb73c782f2
  26dc21c7ca7eb406be74a69ecba5b4a87c07cfc6e687b4beca9a6eda
  c95940a3b4120573b26a80005e697833b0ba285fce7b3f1f25243008
  860b8f1de710a0dcc05b0d20341efe90eb2bcca26797c2d85ae6ca74
  c00696cb1b13e40bda15b27964d7670576647bfaf9
Y_i2: (length: 1 bytes)
  00
G.scalar_mult_vfy(s,Y_i1) = G.scalar_mult_vfy(s,Y_i2) = G.I
```

## Authors' Addresses


Michel Abdalla
DFINITY - Zurich

Email: michel.abdalla@gmail.com

Bjoern Haase
Endress + Hauser Liquid Analysis - Gerlingen

Email: bjoern.m.haase@web.de

Julia Hesse
IBM Research Europe - Zurich

Email: JHS@zurich.ibm.com