

Network Working Group
Internet-Draft

Intended status: Informational

Expires: 18 July 2022

M. Abdalla
DFINITY - Zurich

B. Haase

J. Hesse

IBM Research Europe - Zurich

14 January 2022

Endress + Hauser Liquid Analysis - Gerlingen

CPlace, a balanced composable PAKE
draft-irtf-cfrg-cpace-05

Abstract

This document describes CPlace which is a protocol for two parties that share a low-entropy secret (password) to derive a strong shared key without disclosing the secret to offline dictionary attacks. This method was tailored for constrained devices, is compatible with any group of both prime- and non-prime order, and comes with a security proof providing composability guarantees.

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 18 July 2022.

Internet-Draft

CPace

January 2022

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

| | | |
|------------------------|---|--------------------|
| 1. | Introduction | 5 |
| 2. | Requirements Notation | 5 |
| 3. | Definition CPace | 6 |
| 3.1. | Setup | 6 |
| 3.1.1. | Hash function H | 6 |
| 3.1.2. | Group environment G | 7 |
| 3.2. | Inputs | 7 |
| 3.3. | Notation | 8 |
| 3.3.1. | Notation for group operations | 9 |
| 4. | The CPace protocol | 9 |
| 4.1. | Session identifier establishment | 10 |
| 4.2. | Protocol flow | 10 |
| 4.3. | CPace protocol instructions | 10 |
| 5. | CPace cipher suites | 11 |
| 6. | Implementation of recommended CPace cipher suites | 12 |
| 6.1. | Common function for computing generators | 12 |
| 6.2. | CPace group objects G_X25519 and G_X448 for single-coordinate Ladders on Montgomery curves | 13 |
| 6.2.1. | Verification tests | 14 |
| 6.3. | CPace group objects G_Ristretto255 and G-Decaf448 for prime-order group abstractions | 15 |
| 6.3.1. | Verification tests | 17 |
| 6.4. | CPace group objects for curves in Short-Weierstrass representation | 17 |
| 6.4.1. | Curves and associated functions | 17 |
| 6.4.2. | Suitable encode_to_curve methods | 18 |
| 6.4.3. | Definition of the group environment G for | |

| | |
|--|--------------------|
| Short-Weierstrass curves | 18 |
| 6.4.4. Verification tests | 20 |
| 7. Implementation verification | 20 |
| 8. Security Considerations | 20 |
| 8.1. Party identifiers and relay attacks | 20 |

| | |
|---|--------------------|
| 8.2. Hashing and key derivation | 21 |
| 8.3. Key confirmation | 21 |
| 8.4. Sampling of scalars | 22 |
| 8.5. Single-coordinate CPace on Montgomery curves | 22 |
| 8.6. Nonce values | 23 |
| 8.7. Side channel attacks | 23 |
| 8.8. Quantum computers | 23 |
| 9. IANA Considerations | 24 |
| 10. Acknowledgements | 24 |
| 11. References | 24 |
| 11.1. Normative References | 24 |
| 11.2. Informative References | 25 |
| Appendix A. CPace function definitions | 26 |
| A.1. Definition and test vectors for string utility functions | 26 |
| A.1.1. prepend_len function | 26 |
| A.1.2. prepend_len test vectors | 26 |
| A.1.3. prefix_free_cat function | 27 |
| A.1.4. Testvector for prefix_free_cat() | 27 |
| A.1.5. Examples for invalid encoded messages | 27 |
| A.2. Definition of generator_string function. | 28 |
| A.3. Definitions and test vector ordered concatenation | 28 |
| A.3.1. Definitions for lexicographical ordering | 28 |
| A.3.2. Definitions for ordered concatenation | 28 |
| A.3.3. Test vectors ordered concatenation | 28 |
| A.4. Decoding and Encoding functions according to RFC7748 | 29 |
| A.5. Elligator 2 reference implementation | 29 |
| Appendix B. Test vectors | 30 |
| B.1. Test vector for CPace using group X25519 and hash SHA-512 | 30 |
| B.1.1. Test vectors for calculate_generator with group X25519 | 30 |
| B.1.2. Test vector for MSGa | 31 |
| B.1.3. Test vector for MSGb | 31 |
| B.1.4. Test vector for secret points K | 32 |
| B.1.5. Test vector for ISK calculation initiator/ | |

| | |
|---|--------------------|
| responder | 32 |
| B.1.6. Test vector for ISK calculation parallel execution . | 32 |
| B.1.7. Corresponding ANSI-C initializers | 33 |
| B.1.8. Test vectors for G_X25519.scalar_mult_vfy: low order points | 34 |
| B.2. Test vector for CPace using group X448 and hash SHAKE-256 | 35 |
| B.2.1. Test vectors for calculate_generator with group X448 | 35 |
| B.2.2. Test vector for MSGa | 36 |
| B.2.3. Test vector for MSGb | 36 |
| B.2.4. Test vector for secret points K | 37 |

| | |
|---|--------------------|
| B.2.5. Test vector for ISK calculation initiator/ responder | 37 |
| B.2.6. Test vector for ISK calculation parallel execution . | 38 |
| B.2.7. Corresponding ANSI-C initializers | 38 |
| B.2.8. Test vectors for G_X448.scalar_mult_vfy: low order points | 40 |
| B.3. Test vector for CPace using group ristretto255 and hash SHA-512 | 41 |
| B.3.1. Test vectors for calculate_generator with group ristretto255 | 41 |
| B.3.2. Test vector for MSGa | 42 |
| B.3.3. Test vector for MSGb | 42 |
| B.3.4. Test vector for secret points K | 43 |
| B.3.5. Test vector for ISK calculation initiator/ responder | 43 |
| B.3.6. Test vector for ISK calculation parallel execution . | 43 |
| B.3.7. Corresponding ANSI-C initializers | 44 |
| B.3.8. Test case for scalar_mult with valid inputs | 45 |
| B.3.9. Invalid inputs for scalar_mult_vfy | 46 |
| B.4. Test vector for CPace using group decaf448 and hash SHAKE-256 | 46 |
| B.4.1. Test vectors for calculate_generator with group decaf448 | 46 |
| B.4.2. Test vector for MSGa | 47 |
| B.4.3. Test vector for MSGb | 47 |
| B.4.4. Test vector for secret points K | 48 |
| B.4.5. Test vector for ISK calculation initiator/ responder | 48 |
| B.4.6. Test vector for ISK calculation parallel execution . | 49 |

| | | |
|------------------------|--|--------------------|
| B.4.7. | Corresponding ANSI-C initializers | 49 |
| B.4.8. | Test case for scalar_mult with valid inputs | 51 |
| B.4.9. | Invalid inputs for scalar_mult_vfy | 51 |
| B.5. | Test vector for CPace using group NIST P-256 and hash SHA-256 | 51 |
| B.5.1. | Test vectors for calculate_generator with group NIST P-256 | 51 |
| B.5.2. | Test vector for MSGa | 52 |
| B.5.3. | Test vector for MSGb | 52 |
| B.5.4. | Test vector for secret points K | 53 |
| B.5.5. | Test vector for ISK calculation initiator/ responder | 53 |
| B.5.6. | Test vector for ISK calculation parallel execution . | 54 |
| B.5.7. | Corresponding ANSI-C initializers | 55 |
| B.5.8. | Test case for scalar_mult_vfy with correct inputs . . | 56 |
| B.5.9. | Invalid inputs for scalar_mult_vfy | 57 |
| B.6. | Test vector for CPace using group NIST P-384 and hash SHA-384 | 57 |

| | | |
|------------------------|--|--------------------|
| B.6.1. | Test vectors for calculate_generator with group NIST P-384 | 57 |
| B.6.2. | Test vector for MSGa | 58 |
| B.6.3. | Test vector for MSGb | 59 |
| B.6.4. | Test vector for secret points K | 59 |
| B.6.5. | Test vector for ISK calculation initiator/ responder | 59 |
| B.6.6. | Test vector for ISK calculation parallel execution . | 60 |
| B.6.7. | Corresponding ANSI-C initializers | 61 |
| B.6.8. | Test case for scalar_mult_vfy with correct inputs . . | 63 |
| B.6.9. | Invalid inputs for scalar_mult_vfy | 63 |
| B.7. | Test vector for CPace using group NIST P-521 and hash SHA-512 | 64 |
| B.7.1. | Test vectors for calculate_generator with group NIST P-521 | 64 |
| B.7.2. | Test vector for MSGa | 64 |
| B.7.3. | Test vector for MSGb | 65 |
| B.7.4. | Test vector for secret points K | 66 |
| B.7.5. | Test vector for ISK calculation initiator/ responder | 66 |
| B.7.6. | Test vector for ISK calculation parallel execution . | 67 |
| B.7.7. | Corresponding ANSI-C initializers | 68 |

| | | |
|------------------------|---|--------------------|
| B.7.8. | Test case for scalar_mult_vfy with correct inputs . . | 70 |
| B.7.9. | Invalid inputs for scalar_mult_vfy | 71 |
| Authors' Addresses | | 71 |

[1.](#) Introduction

This document describes CPace which is a protocol for two parties for deriving a strong shared secret from a shared low-entropy secret (password) without exposing the secret to offline dictionary attacks. The CPace design was tailored for efficiency on constrained devices such as secure-element chipsets and considers mitigations with respect to adversaries that might become capable of breaking the discrete logarithm problem on elliptic curves by quantum computers. CPace comes with both game-based and simulation-based proofs, where the latter provides composability guarantees that let CPace run securely in concurrent settings.

[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.](#) Definition CPace

[3.1.](#) Setup

For 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 . We assume both G and H to come with associated constants and functions as detailed below. To access these we use an object-style notation such as, e.g., $H.b_in_bytes$ and $G.sample_scalar()$.

[3.1.1.](#) Hash function H

With H we denote a hash function. 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-length output 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 default output size in bytes corresponding to the symmetric security level 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 . For instance, for SHA-512 the input block size s_in_bytes is 128, while for SHAKE-256 the input block size amounts to 136 bytes.

[3.1.2.](#) 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 multiplicative 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 a scalar (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 string which SHALL be uniquely identifying the group environment `G`.

[3.2.](#) Inputs

- * PRS denotes a password-related octet string which is a MANDATORY input for all CPace instantiations and needs to be available to both parties. Typically PRS is derived from a low-entropy secret such as a user-supplied password (pw) or a personal identification number, e.g. by use of a password-based key derivation function $PRS = \text{PBKDF}(pw)$.

- * CI denotes an OPTIONAL octet string identifying a communication

channel that needs to be available to both parties. CI can be used for binding a CPace execution to one specific channel. Typically CI is obtained by concatenating strings that uniquely identify the protocol partner's identities, such as their networking addresses.

- * sid denotes an OPTIONAL octet string serving as session identifier that needs to be available to both parties. In application scenarios where a higher-level protocol has established a unique sid value, this parameter can be used to ensure strong composability guarantees of CPace, and to bind a CPace execution to the application.
- * ADa and ADb denote OPTIONAL octet strings containing arbitrary associated data, each available to one of the parties. They are not required to be equal, and are publicly transmitted as part of the protocol flow. ADa and ADb can for instance include party identifiers or protocol version information (to avoid, e.g., downgrade attacks). In a setting with initiator and responder roles, the information ADa sent by the initiator can be used by the responder for identifying which among possibly several different PRS to use for the CPace session.

[3.3.](#) Notation

- * bytes1 || bytes2 denotes concatenation of octet strings.
- * oCat(bytes1,bytes2) denotes ordered concatenation of octet strings, which places the lexicographically larger octet string first. (Explicit code for this function is given in the appendix.)
- * concat(MSGa,MSGb) denotes a concatenation method allows both parties to concatenate CPace's protocol messages in the same way. 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, concat(MSGa,MSGb) = oCat(MSGa,MSGb) SHALL be used. In settings where the protocol flow enforces ordering, concat(MSGa,MSGb) SHOULD BE implemented such that the later message is appended to the earlier message, i.e., concat(MSGa,MSGb) = MSGa||MSGb if MSGa is sent first.
- * len(S) denotes the number of octets in a string S.
- * nil denotes an empty octet string, i.e., len(nil) = 0.

- * `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 are given in the appendix.)
- * `prefix_free_cat(a0,a1, ...)` denotes a function that outputs the prefix-free encoding of all input octet strings as the concatenation of the individual strings with their respective length prepended: `prepend_len(a0) || prepend_len(a1) || ...`. Such prefix-free encoding of multiple substrings allows for parsing individual subcomponents of a network message. (Test vectors and reference implementations are given in the appendix.)
- * `sample_random_bytes(n)` denotes a function that returns `n` octets uniformly distributed between 0 and 255.
- * `zero_bytes(n)` denotes a function that returns `n` octets with value 0.

[3.3.1.](#) Notation for group operations

We use multiplicative 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 $*$.

[4.](#) 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 public 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. Naming of this key as "intermediate" session key highlights the fact that 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.

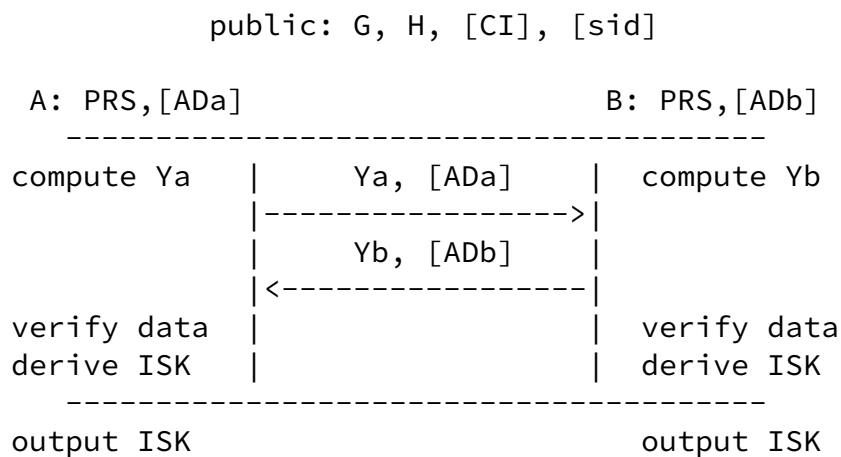
[4.1.](#) Session identifier establishment

It is RECOMMENDED to establish a unique session identifier `sid` in the course of the higher-level protocol that invokes CPace, by concatenating random bytes produced by A with random bytes produced by B. In settings where such establishment is not an option, we can let initiator A choose a fresh random `sid` and send it to B together with the first message. This method works whenever the message produced by party A comes first.

The `sid` string SHOULD HAVE a length of at least 8 bytes and it MAY also be the empty string, `nil`. I.e., use of the `sid` string is OPTIONAL.

[4.2.](#) Protocol flow

Optional parameters and messages are denoted with `[]`.



[4.3.](#) 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 = prefix_free_cat(Ya, ADa)` with optional associated data `ADa` to B. `ADa` MAY have length zero.

B computes a generator `g = G.calculate_generator(H, PRS, CI, sid)`,

scalar $y_b = G.sample_scalar()$ and group element $Y_b = G.scalar_mult(y_b, g)$. B sends $MSG_b = prefix_free_cat(Y_b, ADB)$ to A.

Note that as `prefix_free_cat` prepends the respectively length of the input fields, the receivers can parse MSG_a and MSG_b for subcomponents.

Upon reception of MSG_a , B checks that MSG_a was properly generated by `prefix_free_cat`. I.e. it checks that the actual length of MSG_a matches the sum of the decoded prepended lengths for Y_a and AD_a . If this parsing fails, then B MUST abort. (Testvectors of examples for invalid messages are given in the appendix.) B then computes $K = G.scalar_mult_vfy(y_b, Y_a)$. B MUST abort if $K=G.I$. Otherwise B returns $ISK = H.hash(prefix_free_cat(G.DSI || _ISK, sid, K) || concat(MSG_a, MSG_b))$. B returns ISK and terminates.

Upon reception of MSG_b , A parses MSG_b for Y_b and AD_b . I.e. it checks that the actual length of MSG_b matches the sum of the decoded prepended lengths for Y_b and AD_b . If this parsing fails, then A MUST abort. A then computes $K = G.scalar_mult_vfy(y_a, Y_b)$. A MUST abort if $K=G.I$. Otherwise A returns $ISK = H.hash(prefix_free_cat(G.DSI || _ISK, sid, K) || concat(MSG_a, MSG_b))$. 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 MSG_a and MSG_b) of both parties match.

We note that the above protocol instructions implement a parallel setting with no specific initiator/responder and no assumptions about the order in which messages arrive. If implemented as initiator-responder protocol, the responder, say, B, starts with computation of the generator only upon reception of MSG_a .

[5.](#) CPace cipher suites

This section documents RECOMMENDED CPace cipher suite configurations. Any cipher suite configuration for CPace is REQUIRED to specify

- * A group environment G specified by

- Functions `G.sample_scalar()`, `G.scalar_mult()`, `G.scalar_mult_vfy()` and `G.calculate_generator()`
- A neutral element `G.I`
- A domain separation identifier string `G.DSI` unique for this cipher suite.
- * A hash function `H` specified by
 - Function `H.hash()`
 - Constants `H.b_in_bytes`, `H.bmax_in_bytes` and `H.s_in_bytes`

For naming cipher suites we use the convention "CPACE-G-H". Currently, test vectors are available for the following RECOMMENDED cipher suites:

- * CPACE-X25519-SHA512. This suite uses curve `G_X25519` defined in [Section 6.2](#) and SHA-512 as hash function.
- * CPACE-X448-SHAKE256. This suite uses curve `G_X448` defined in [Section 6.2](#) and SHAKE-256 as hash function.
- * CPACE-P256_XMD:SHA-256_SSWU_NU_-SHA256. This suite instantiates `G` as specified in [Section 6.4](#) using the `encode_to_curve` function `P256_XMD:SHA-256_SSWU_NU_` from [\[I-D.irtf-cfrg-hash-to-curve\]](#) on curve NIST-P256, and hash function SHA-256.
- * CPACE-P384_XMD:SHA-384_SSWU_NU_-SHA384. This suite instantiates `G` as specified in [Section 6.4](#) using the `encode_to_curve` function `P384_XMD:SHA-384_SSWU_NU_` from [\[I-D.irtf-cfrg-hash-to-curve\]](#) on curve NIST-P384 with `H = SHA-384`.
- * CPACE-P521_XMD:SHA-512_SSWU_NU_-SHA512. This suite instantiates `G` as specified in [Section 6.4](#) using the `encode_to_curve` function `P521_XMD:SHA-384_SSWU_NU_` from [\[I-D.irtf-cfrg-hash-to-curve\]](#) on curve NIST-P384 with `H = SHA-512`.

- * CPACE-RISTR255-SHA512. This suite uses $G_{\text{ristretto255}}$ defined in [Section 6.3](#) and $H = \text{SHA-512}$.
- * CPACE-DECAF448-SHAKE256 This suite uses G_{decaf448} defined in [Section 6.3](#) and $H = \text{SHAKE-256}$.

CPace can securely be implemented on further elliptic curves when following the guidance given in [Section 8](#).

[6](#). Implementation of recommended CPace cipher suites

[6.1](#). 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. Let CPACE-G-H denote the cipher suite.

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

- * $\text{len_zpad} = \text{MAX}(0, s_{\text{in_bytes}} - \text{len}(\text{prepend_len}(\text{PRS})) - \text{len}(\text{prepend_len}(\text{G.DSI})) - 1)$

The zero padding of length `len_zpad` is designed such that the encoding of `G.DSI` and `PRS` together with the zero padding field completely fills the first input block (of length `s_in_bytes`) of the hash. As a result 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 the 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`.

[6.2](#). 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 8](#).

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 = "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 = "CPace448"

CPace cipher suites using G_X448 MUST use a hash function producing

at least $H.b_max_in_bytes \geq 56$ 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 [\[I-D.irtf-cfrg-hash-to-curve\]](#). Note that the `v` coordinate produced by the `map_to_curve_elligator2` function is not required for CSpace and discarded. The appendix repeats the definitions from [\[I-D.irtf-cfrg-hash-to-curve\]](#) for convenience.

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

[6.2.1](#). Verification tests

For single-coordinate Montgomery ladders on Montgomery curves verification tests according to [Section 7](#) SHALL consider the `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 also encode low-order points.

Corresponding test vectors are provided in the appendix.

6.3. 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 a one-way-map. 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 = "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 \geq 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 = "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 $H.b_max_in_bytes \geq 112$ 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.

Internet-Draft

CPace

January 2022

- Finally the internal representation of the generator `_g` is calculated as `_g = one_way_map(gen_str_hash)` using the one-way map 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).

[6.3.1](#). Verification tests

For group abstractions verification tests according to [Section 7](#) SHALL consider encodings of the neutral element and an octet string that does not decode to a valid group element.

[6.4](#). 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(str)` function. `encode_to_curve(str)` must map an octet string `str` to a point on the curve.

[6.4.1](#). 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 identity 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).

[6.4.2.](#) Suitable encode_to_curve methods

All the encode_to_curve methods specified in [\[I-D.irtf-cfrg-hash-to-curve\]](#) are suitable for CPace. For Short-Weierstrass curves it is RECOMMENDED to use the non-uniform variant of the SSWU mapping primitive from [\[I-D.irtf-cfrg-hash-to-curve\]](#) 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.)

[6.4.3.](#) 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 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 or 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) we denote a selected mapping function from [\[I-D.irtf-cfrg-hash-to-curve\]](#). I.e. a function that maps octet string str to a point on the group. [\[I-D.irtf-cfrg-hash-to-curve\]](#) 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 [[I-D.irtf-cfrg-hash-to-curve](#)].

- * G.DSI denotes a domain-separation identifier string. G.DSI which SHALL BE obtained by the concatenation of "CPace" and the associated name of the cipher suite used for the encode_to_curve function as specified in [[I-D.irtf-cfrg-hash-to-curve](#)]. 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 = "CPaceP384_XMD:SHA-384_SSWU_NU_".

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

- * G.sample_scalar() SHALL return a value between 1 and (G.group_order - 1). The value sampling MUST BE uniformly random. It is RECOMMENDED to use rejection sampling 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) is returned, using the selected function from [[I-D.irtf-cfrg-hash-to-curve](#)].
- * 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.

[6.4.4.](#) Verification tests

For Short-Weierstrass curves verification tests according to [Section 7](#) SHALL consider encodings of the point at infinity and an encoding of a point not on the group.

[7.](#) 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 points in `MSGa` and `MSGb` and introduce this in a protocol run. It SHALL be verified that the abort condition is properly handled.

Moreover any implementation MUST be tested with respect invalid encodings of `MSGa` and `MSGb` where the length of the message does not match the specified encoding (i.e. where the sum of the prepended length information does not match the actual length of the message).

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

[8.](#) 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.

[8.1.](#) 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.

[8.2.](#) Hashing and key derivation

In order to prevent analysis of length extension attacks on hash functions, all hash input strings in CPace are designed to be prefix-free strings which have the length of individual substrings prepended, enforced by the `prefix_free_cat()` function. This choice was made in order to make CPace suitable also for hash function instantiations using Merkle-Damgard constructions such as SHA-256 or SHA-512 along the lines of [[CDMP05](#)]. In case that an application wishes to use another form of encoding, the guidance given in [[CDMP05](#)] SHOULD BE considered.

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 4](#) 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.

[8.3](#). 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 T_a , T_b that is calculated over the protocol message that it has sent previously. I.e. let party A calculate its transmitted authentication code T_a as $T_a = \text{MAC}(\text{mac_key}, \text{MSG}_a)$ and let party B calculate its transmitted authentication code T_b as $T_b = \text{MAC}(\text{mac_key}, \text{MSG}_b)$.
- * Let the receiving party check the remote authentication tag for the correct value and abort in case that it's incorrect.

[8.4.](#) Sampling of scalars

For curves over fields F_p where p 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 reducing 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 begning 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 6.4](#).

[8.5.](#) Single-coordinate CPace on Montgomery curves

The recommended cipher suites for the Montgomery curves Curve25519 and Curve448 in [Section 6.2](#) 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 twist MUST BE EQUAL to or an integer multiple of the cofactor c of the curve.
- * Both field order q and group order p MUST BE close to a power of two along the lines of [\[AHH21\]](#), [Appendix E](#).
- * 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 $G.I$.

Montgomery curves other than the ones recommended here can use the specifications given in [Section 6.2](#), given that the above properties hold.

[8.6.](#) Nonce values

Secret scalars y_a and y_b 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.

[8.7.](#) Side channel attacks

All state-of-the-art methods for realizing constant-time execution SHOULD be used. 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.

[8.8.](#) 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".

Internet-Draft

CPace

January 2022

[9.](#) IANA Considerations

No IANA action is required.

[10.](#) Acknowledgements

Thanks to the members of the CFRG for comments and advice. Any comment and advice is appreciated.

[11.](#) References

[11.1.](#) Normative References

[I-D.[draft-irtf-cfrg-ristretto255-decaf448](#)]

Valence, H. D., Grigg, J., Tankersley, G., Valsorda, F., Lovecruft, I., and M. Hamburg, "The ristretto255 and decaf448 Groups", Work in Progress, Internet-Draft, [draft-irtf-cfrg-ristretto255-decaf448-01](#), 4 August 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-ristretto255-decaf448-01>>.

[I-D.irtf-cfrg-hash-to-curve]

Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", Work in Progress, Internet-Draft, [draft-irtf-cfrg-hash-to-curve-13](#), 10 November 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-13>>.

[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>>.

Abdalla, et al.

Expires 18 July 2022

[Page 24]

Internet-Draft

CPace

January 2022

[11.2](#). 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>>.

Abdalla, et al.

Expires 18 July 2022

[Page 25]

Internet-Draft

CPace

January 2022

[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>>.

[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>>.

[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.](#) prefix_free_cat function

```

def prefix_free_cat(*args):
    result = b''

```

```

    for arg in args:
        result += prepend_len(arg)
    return result

```

[A.1.4.](#) Testvector for prefix_free_cat()

```

prefix_free_cat(b"1234",b"5",b"",b"6789"):
(length: 13 bytes)
04313233340135000436373839

```

[A.1.5.](#) Examples for invalid encoded messages

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 for the prefix free concatenation does not give the correct length of the message. Parties MUST abort upon reception of such invalid messages as MSGa or MSGb.

```

Inv_MSG1 with invalid encoded length: (length: 3 bytes)
fffff
Inv_MSG2 with invalid encoded length: (length: 3 bytes)
ffff03
Inv_MSG3 with invalid encoded length: (length: 4 bytes)
00ffff03
Inv_MSG4 with invalid encoded length: (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 prefix_free_cat(DSI, PRS, zero_bytes(len_zpad),
                           CI, sid)

```

[A.3.](#) Definitions and test vector ordered concatenation

[A.3.1.](#) Definitions for lexicographical ordering

For ordered concatenation lexicographical ordering of byte sequences is used:

```
def lexicographically_larger(bytes1,bytes2):
    "Returns True if bytes1 > bytes2 using lexicographical 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 lexicographical ordering ordered concatenation is specified as follows.

```
def oCAT(bytes1,bytes2):
    if lexicographically_larger(bytes1,bytes2):
        return bytes1 + bytes2
    else:
        return bytes2 + bytes1
```

[A.3.3.](#) Test vectors ordered concatenation

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



```
oCAT(b"ABCD",b"BCD"): (length: 7 bytes)
    42434441424344
oCAT(b"BCD",b"ABCDE"): (length: 8 bytes)
    4243444142434445
```

[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):
    u = u % p
    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

Internet-Draft

CPace

January 2022

Inputs

ADb = b'ADb'

yb (little endian): (length: 32 bytes)

a145e914b347002d298ce2051394f0ed68cf3623dfe5db082c78ffa5
a667acdc

Outputs

Yb: (length: 32 bytes)

e1b730a4956c0f853d96c5d125cebeeea46952c07c6f66da65bd9ffd
2f71a462

MSGb: (length: 37 bytes)

20e1b730a4956c0f853d96c5d125cebeeea46952c07c6f66da65bd9f
fd2f71a46203414462[B.1.4.](#) Test vector for secret points K

scalar_mult_vfy(ya,Yb): (length: 32 bytes)

2a905bc5f0b93ee72ac4b6ea8723520941adfc892935bf6f86d9e199
befa6024

scalar_mult_vfy(yb,Ya): (length: 32 bytes)

2a905bc5f0b93ee72ac4b6ea8723520941adfc892935bf6f86d9e199
befa6024[B.1.5.](#) Test vector for ISK calculation initiator/responder

unordered cat of transcript : (length: 74 bytes)

206f7fd31863b18b0cc9830fc842c60dea80120ccf2fd375498225e4
5a520653610341446120e1b730a4956c0f853d96c5d125cebeeea469
52c07c6f66da65bd9ffd2f71a46203414462

DSI = G.DSI_ISK, b'CPace255_ISK': (length: 12 bytes)

43506163653235355f49534b

prefix_free_cat(DSI,sid,K)||MSGa||MSGb: (length: 137 bytes)

0c43506163653235355f49534b107e4b4791d6a8ef019b936c79fb7f
2c57202a905bc5f0b93ee72ac4b6ea8723520941adfc892935bf6f86
d9e199befa6024206f7fd31863b18b0cc9830fc842c60dea80120ccf
2fd375498225e45a520653610341446120e1b730a4956c0f853d96c5
d125cebeeea46952c07c6f66da65bd9ffd2f71a46203414462

ISK result: (length: 64 bytes)
99a9e0ff35acb94ad8af1cd6b32ac409dc7d00557ccd9a7d19d3b462
9e5f1f084f9332096162438c7ecc78331b4eda17e1a229a47182ecc
9ea58cd9cdcd8e9a

[B.1.6.](#) Test vector for ISK calculation parallel execution

ordered cat of transcript : (length: 74 bytes)
20e1b730a4956c0f853d96c5d125cebeeea46952c07c6f66da65bd9f
fd2f71a46203414462206f7fd31863b18b0cc9830fc842c60dea8012
0ccf2fd375498225e45a5206536103414461
DSI = G.DSI_ISK, b'CPace255_ISK': (length: 12 bytes)
43506163653235355f49534b
prefix_free_cat(DSI,sid,K) || oCAT(MSGa,MSGb):
(length: 137 bytes)
0c43506163653235355f49534b107e4b4791d6a8ef019b936c79fb7f
2c57202a905bc5f0b93ee72ac4b6ea8723520941adfc892935bf6f86
d9e199befa602420e1b730a4956c0f853d96c5d125cebeeea46952c0
7c6f66da65bd9ffd2f71a46203414462206f7fd31863b18b0cc9830f
c842c60dea80120ccf2fd375498225e45a5206536103414461
ISK result: (length: 64 bytes)
3cd6a9670fa3ff211d829b845baa0f5ba9ad580c3ba0ee790bd0e9cd
556290a8ffce44419fbf94e4cb8e7fe9f454fd25dc13e689e4d6ab0a
c2211c70a8ac0062

[B.1.7.](#) Corresponding ANSI-C initializers

```
const uint8_t tc_PRS[] = {  
    0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,  
};  
const uint8_t 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 uint8_t tc_sid[] = {  
    0x7e,0x4b,0x47,0x91,0xd6,0xa8,0xef,0x01,0x9b,0x93,0x6c,0x79,
```

```

    0xfb,0x7f,0x2c,0x57,
};
const uint8_t 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 uint8_t tc_ya[] = {
    0x45,0xac,0xf9,0x31,0x16,0xae,0x5d,0x3d,0xae,0x99,0x5a,0x7c,
    0x62,0x7d,0xf2,0x92,0x43,0x21,0xa8,0xe8,0x57,0xd9,0xa2,0x00,
    0x80,0x71,0x31,0xe3,0x88,0x39,0xb0,0xc2,
};
const uint8_t tc_ADa[] = {
    0x41,0x44,0x61,
};
const uint8_t tc_Ya[] = {
    0x6f,0x7f,0xd3,0x18,0x63,0xb1,0x8b,0x0c,0xc9,0x83,0x0f,0xc8,
    0x42,0xc6,0x0d,0xea,0x80,0x12,0x0c,0xcf,0x2f,0xd3,0x75,0x49,
    0x82,0x25,0xe4,0x5a,0x52,0x06,0x53,0x61,

```

```

};
const uint8_t tc_yb[] = {
    0xa1,0x45,0xe9,0x14,0xb3,0x47,0x00,0x2d,0x29,0x8c,0xe2,0x05,
    0x13,0x94,0xf0,0xed,0x68,0xcf,0x36,0x23,0xdf,0xe5,0xdb,0x08,
    0x2c,0x78,0xff,0xa5,0xa6,0x67,0xac,0xdc,
};
const uint8_t tc_ADb[] = {
    0x41,0x44,0x62,
};
const uint8_t tc_Yb[] = {
    0xe1,0xb7,0x30,0xa4,0x95,0x6c,0x0f,0x85,0x3d,0x96,0xc5,0xd1,
    0x25,0xce,0xbe,0xee,0xa4,0x69,0x52,0xc0,0x7c,0x6f,0x66,0xda,
    0x65,0xbd,0x9f,0xfd,0x2f,0x71,0xa4,0x62,
};
const uint8_t tc_K[] = {
    0x2a,0x90,0x5b,0xc5,0xf0,0xb9,0x3e,0xe7,0x2a,0xc4,0xb6,0xea,
    0x87,0x23,0x52,0x09,0x41,0xad,0xfc,0x89,0x29,0x35,0xbf,0x6f,
    0x86,0xd9,0xe1,0x99,0xbe,0xfa,0x60,0x24,
};
const uint8_t tc_ISK_IR[] = {
    0x99,0xa9,0xe0,0xff,0x35,0xac,0xb9,0x4a,0xd8,0xaf,0x1c,0xd6,
    0xb3,0x2a,0xc4,0x09,0xdc,0x7d,0x00,0x55,0x7c,0xcd,0x9a,0x7d,

```

`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: d8e2c776bbacd510d09fd9278b7edcd25fc5ae9adfb3b6e040e8d3b71b21806
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
00000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000160a4169
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)
21b4f4bd9e64ed355c3eb676a28ebdaf6d8f17bdc365995b3190971
53044080516bd083bfcce66121a3072646994c8430cc382b8dc543e8

```

Outputs

```

Ya: (length: 56 bytes)
396bd11daf223711e575cac6021e3fa31558012048a1cec7876292b9
6c61eda353fe04f33028d2352779668a934084da776c1c51a58ce4b5
MSGa: (length: 61 bytes)
38396bd11daf223711e575cac6021e3fa31558012048a1cec7876292
b96c61eda353fe04f33028d2352779668a934084da776c1c51a58ce4
b503414461

```

[B.2.3.](#) Test vector for MSGb

```

ADb = b'ADb'
yb (little endian): (length: 56 bytes)
848b0779ff415f0af4ea14df9dd1d3c29ac41d836c7808896c4eba19
c51ac40a439caf5e61ec88c307c7d619195229412eaa73fb2a5ea20d
Outputs
Yb: (length: 56 bytes)
53c519fb490fde5a04bda8c18b327d0fc1a9391d19e0ac00c59df9c6
0422284e593d6b092eac94f5aa644ed883f39bd4f04e4beb6af86d58
MSGb: (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)
43506163653434385f49534b
prefix_free_cat(DSI,sid,K)||MSGa||MSGb: (length: 209 bytes)
0c43506163653434385f49534b105223e0cdc45d6575668d64c55200
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: 122 bytes)
3853c519fb490fde5a04bda8c18b327d0fc1a9391d19e0ac00c59df9
c60422284e593d6b092eac94f5aa644ed883f39bd4f04e4beb6af86d
580341446238396bd11daf223711e575cac6021e3fa31558012048a1
cec7876292b96c61eda353fe04f33028d2352779668a934084da776c
1c51a58ce4b503414461
DSI = G.DSI_ISK, b'CPace448_ISK': (length: 12 bytes)
43506163653434385f49534b
prefix_free_cat(DSI,sid,K) || oCAT(MSGa,MSGb):
(length: 209 bytes)
0c43506163653434385f49534b105223e0cdc45d6575668d64c55200
412438e00af217556a40ccbc9822cc27a43542e45166a653aa4df746
d5f8e1e8df483e9baff71c9eb03ee20a688ad4e4d359f70ac9ec3f6a
6599973853c519fb490fde5a04bda8c18b327d0fc1a9391d19e0ac00
c59df9c60422284e593d6b092eac94f5aa644ed883f39bd4f04e4beb
6af86d580341446238396bd11daf223711e575cac6021e3fa3155801
2048a1cec7876292b96c61eda353fe04f33028d2352779668a934084
da776c1c51a58ce4b503414461
ISK result: (length: 64 bytes)
925e95d1095dad1af6378d5ef8b9a998bd3855bfc7d36cb5ca05b0a7
a93346abcb8cef04bceb28c38fdaf0cc608fd1dcd462ab523f3b7f75
2c77c411be3ac8fb
```

[B.2.7.](#) Corresponding ANSI-C initializers

```
const uint8_t tc_PRS[] = {
    0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const uint8_t 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 uint8_t tc_sid[] = {
    0x52,0x23,0xe0,0xcd,0xc4,0x5d,0x65,0x75,0x66,0x8d,0x64,0xc5,
    0x52,0x00,0x41,0x24,
};
const uint8_t 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 uint8_t 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 uint8_t tc_ADa[] = {  
    0x41,0x44,0x61,  
};  
const uint8_t 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 uint8_t 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 uint8_t tc_ADb[] = {  
    0x41,0x44,0x62,  
};  
const uint8_t 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 uint8_t 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 uint8_t tc_ISK_IR[] = {
```



```
c1ae06a0871f9bf48545f49b2a792eed255ac04f52758c9c60448306
810b44e986e3dcbb
encoded generator g: (length: 32 bytes)
9c5712178570957204d89ac11acbef789dd076992ba361429acb2bc3
8c71d14c
```

[B.3.2.](#) Test vector for MSGa

Inputs

```
ADa = b'ADa'
ya (little endian): (length: 32 bytes)
1433dd19359992d4e06d740d3993d429af6338ffb4531ce175d22449
853a790b
```

Outputs

```
Ya: (length: 32 bytes)
a8fc42c4d57b3c7346661011122a00563d0995fd72b62123ae244400
e86d7b1a
MSGa: (length: 37 bytes)
20a8fc42c4d57b3c7346661011122a00563d0995fd72b62123ae2444
00e86d7b1a03414461
```

[B.3.3.](#) Test vector for MSGb

Inputs

```
ADb = b'ADb'
yb (little endian): (length: 32 bytes)
0e6566d32d80a5a1135f99c27f2d637aa24da23027c3fa76b9d1cfd9
742fdc00
```

Outputs

```
Yb: (length: 32 bytes)
fc8e84ae4ab725909af05a56ef9714db6930e4a5589b3fee6cdd2662
36676d63
MSGb: (length: 37 bytes)
20fc8e84ae4ab725909af05a56ef9714db6930e4a5589b3fee6cdd26
```


6236676d6303414462

[B.3.4.](#) Test vector for secret points K

```
scalar_mult_vfy(ya,Yb): (length: 32 bytes)
  3efef1706f42efa354020b087b37fbd9f81cf72a16f4947e4a042a7f
  1aaa2b6f
scalar_mult_vfy(yb,Ya): (length: 32 bytes)
  3efef1706f42efa354020b087b37fbd9f81cf72a16f4947e4a042a7f
  1aaa2b6f
```

[B.3.5.](#) Test vector for ISK calculation initiator/responder

```
unordered cat of transcript : (length: 74 bytes)
  20a8fc42c4d57b3c7346661011122a00563d0995fd72b62123ae2444
  00e86d7b1a0341446120fc8e84ae4ab725909af05a56ef9714db6930
  e4a5589b3fee6cdd266236676d6303414462
DSI = G.DSI_ISK, b'CPaceRistretto255_ISK':
(length: 21 bytes)
  435061636552697374726574746f3235355f49534b
prefix_free_cat(DSI,sid,K)||MSGa||MSGb: (length: 146 bytes)
  15435061636552697374726574746f3235355f49534b107e4b4791d6
  a8ef019b936c79fb7f2c57203efef1706f42efa354020b087b37fbd9
  f81cf72a16f4947e4a042a7f1aaa2b6f20a8fc42c4d57b3c73466610
  11122a00563d0995fd72b62123ae244400e86d7b1a0341446120fc8e
  84ae4ab725909af05a56ef9714db6930e4a5589b3fee6cdd26623667
  6d6303414462
ISK result: (length: 64 bytes)
  0e33c5822bd495dea94ba7af161501f1b2d6a16d464b5d6e1a53dcbf
  b9244b9ba66c09c430fffdfe4fb4e99b4ea46f991a272de0431c132c
  2c79fd6de1a7e5e4
```

[B.3.6.](#) Test vector for ISK calculation parallel execution

```
ordered cat of transcript : (length: 74 bytes)
  20fc8e84ae4ab725909af05a56ef9714db6930e4a5589b3fee6cdd26
  6236676d630341446220a8fc42c4d57b3c7346661011122a00563d09
  95fd72b62123ae244400e86d7b1a03414461
```

```

DSI = G.DSI_ISK, b'CPaceRistretto255_ISK':
(length: 21 bytes)
435061636552697374726574746f3235355f49534b
prefix_free_cat(DSI,sid,K) || oCAT(MSGa,MSGb):
(length: 146 bytes)
15435061636552697374726574746f3235355f49534b107e4b4791d6
a8ef019b936c79fb7f2c57203efef1706f42efa354020b087b37fbd9
f81cf72a16f4947e4a042a7f1aaa2b6f20fc8e84ae4ab725909af05a
56ef9714db6930e4a5589b3fee6cdd266236676d630341446220a8fc
42c4d57b3c7346661011122a00563d0995fd72b62123ae244400e86d
7b1a03414461
ISK result: (length: 64 bytes)
ca36335be682a480a9fc63977d044a10ff7adfcda0f2978fbcf8713d
2a4e23e25c05a9a02edcfbff2ede65b752f8ea1f4454d764ad8ed860
7c158ef662614567

```

B.3.7. Corresponding ANSI-C initializers

```

const uint8_t tc_PRS[] = {
    0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const uint8_t 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 uint8_t tc_sid[] = {
    0x7e,0x4b,0x47,0x91,0xd6,0xa8,0xef,0x01,0x9b,0x93,0x6c,0x79,
    0xfb,0x7f,0x2c,0x57,
};
const uint8_t tc_g[] = {
    0x9c,0x57,0x12,0x17,0x85,0x70,0x95,0x72,0x04,0xd8,0x9a,0xc1,
    0x1a,0xcb,0xef,0x78,0x9d,0xd0,0x76,0x99,0x2b,0xa3,0x61,0x42,
    0x9a,0xcb,0x2b,0xc3,0x8c,0x71,0xd1,0x4c,
};
const uint8_t tc_ya[] = {
    0x14,0x33,0xdd,0x19,0x35,0x99,0x92,0xd4,0xe0,0x6d,0x74,0x0d,
    0x39,0x93,0xd4,0x29,0xaf,0x63,0x38,0xff,0xb4,0x53,0x1c,0xe1,
    0x75,0xd2,0x24,0x49,0x85,0x3a,0x79,0x0b,
};
const uint8_t tc_ADa[] = {
    0x41,0x44,0x61,
};
const uint8_t tc_Ya[] = {
    0xa8,0xfc,0x42,0xc4,0xd5,0x7b,0x3c,0x73,0x46,0x66,0x10,0x11,

```

```

    0x12,0x2a,0x00,0x56,0x3d,0x09,0x95,0xfd,0x72,0xb6,0x21,0x23,
    0xae,0x24,0x44,0x00,0xe8,0x6d,0x7b,0x1a,
};
const uint8_t tc_yb[] = {
    0x0e,0x65,0x66,0xd3,0x2d,0x80,0xa5,0xa1,0x13,0x5f,0x99,0xc2,
    0x7f,0x2d,0x63,0x7a,0xa2,0x4d,0xa2,0x30,0x27,0xc3,0xfa,0x76,
    0xb9,0xd1,0xcf,0xd9,0x74,0x2f,0xdc,0x00,
};
const uint8_t tc_ADb[] = {
    0x41,0x44,0x62,
};
const uint8_t tc_Yb[] = {
    0xfc,0x8e,0x84,0xae,0x4a,0xb7,0x25,0x90,0x9a,0xf0,0x5a,0x56,
    0xef,0x97,0x14,0xdb,0x69,0x30,0xe4,0xa5,0x58,0x9b,0x3f,0xee,
    0x6c,0xdd,0x26,0x62,0x36,0x67,0x6d,0x63,
};
const uint8_t tc_K[] = {
    0x3e,0xfe,0xf1,0x70,0x6f,0x42,0xef,0xa3,0x54,0x02,0x0b,0x08,
    0x7b,0x37,0xfb,0xd9,0xf8,0x1c,0xf7,0x2a,0x16,0xf4,0x94,0x7e,
    0x4a,0x04,0x2a,0x7f,0x1a,0xaa,0x2b,0x6f,
};
const uint8_t tc_ISK_IR[] = {
    0x0e,0x33,0xc5,0x82,0x2b,0xd4,0x95,0xde,0xa9,0x4b,0xa7,0xaf,
    0x16,0x15,0x01,0xf1,0xb2,0xd6,0xa1,0x6d,0x46,0x4b,0x5d,0x6e,
    0x1a,0x53,0xdc,0xbf,0xb9,0x24,0x4b,0x9b,0xa6,0x6c,0x09,0xc4,
    0x30,0xff,0xfd,0xfe,0x4f,0xb4,0xe9,0x9b,0x4e,0xa4,0x6f,0x99,
    0x1a,0x27,0x2d,0xe0,0x43,0x1c,0x13,0x2c,0x2c,0x79,0xfd,0x6d,
    0xe1,0xa7,0xe5,0xe4,
};
const uint8_t tc_ISK_SY[] = {
    0xca,0x36,0x33,0x5b,0xe6,0x82,0xa4,0x80,0xa9,0xfc,0x63,0x97,
    0x7d,0x04,0x4a,0x10,0xff,0x7a,0xdf,0xcd,0xa0,0xf2,0x97,0x8f,
    0xbc,0xf8,0x71,0x3d,0x2a,0x4e,0x23,0xe2,0x5c,0x05,0xa9,0xa0,
    0x2e,0xdc,0xfb,0xff,0x2e,0xde,0x65,0xb7,0x52,0xf8,0xea,0x1f,
    0x44,0x54,0xd7,0x64,0xad,0x8e,0xd8,0x60,0x7c,0x15,0x8e,0xf6,
    0x62,0x61,0x45,0x67,
};

```

[B.3.8](#). Test case for scalar_mult with valid inputs

Internet-Draft

CPace

January 2022

```

s: (length: 32 bytes)
  7cd0e075fa7955ba52c02759a6c90dbbfc10e6d40aea8d283e407d88
  cf538a05
X: (length: 32 bytes)
  021ca069484e890c9e494d8ed6bb0f66cbd9a8f0ef67168f36c51e0e
  feb8f347
G.scalar_mult(s,decode(X)): (length: 32 bytes)
  62aaa018755dc881902097c2a993c0b7c0a4fe33bce2c0182b46a44c
  40b95119
G.scalar_mult_vfy(s,X): (length: 32 bytes)
  62aaa018755dc881902097c2a993c0b7c0a4fe33bce2c0182b46a44c
  40b95119

```

[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)
  011ca069484e890c9e494d8ed6bb0f66cbd9a8f0ef67168f36c51e0e
  feb8f347
Y_i2 == G.I: (length: 32 bytes)
  0000000000000000000000000000000000000000000000000000000000000000
  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

```
ADa = b'ADa'
ya (little endian): (length: 56 bytes)
    d8d2e26c821a12d7f59a8dee023d3f6155976152e16c73cbf68c303d
    f0404399f0a7b614a65df50a9788f00b410586b443f738ad7ff03930
```

Outputs

Ya: (length: 56 bytes)

223f95a5430a2f2a499431696d23ea2d0a90f432e5491e45e4005f3d
d785e7be1235b79252670099bc993c2df5c261dfb7a8989f091e2be3

MSGa: (length: 61 bytes)

38223f95a5430a2f2a499431696d23ea2d0a90f432e5491e45e4005f
3dd785e7be1235b79252670099bc993c2df5c261dfb7a8989f091e2b
e303414461

[B.4.3.](#) Test vector for MSGb

Inputs

ADb = b'ADb'

yb (little endian): (length: 56 bytes)

91bae9793f4a8aceb1b5c54375a7ed1858a79a6e72dab959c8bdf3a7
5ac9bb4de2a25af4d4a9a5c5bc5441d19b8e3f6fcce7196c6afc2236

Outputs

Yb: (length: 56 bytes)

b6ba0a336c103c6c92019ae4cfbcb88d8f6bfc361e979c9e0d3a0967
e630094ba3d1555821ac1f979996ef5ce79f012ffe279ac89b287bee

MSGb: (length: 61 bytes)

38b6ba0a336c103c6c92019ae4cfbcb88d8f6bfc361e979c9e0d3a09
67e630094ba3d1555821ac1f979996ef5ce79f012ffe279ac89b287b
ee03414462

[B.4.4.](#) Test vector for secret points K

scalar_mult_vfy(ya,Yb): (length: 56 bytes)

dc504938fb70eb13916697aa3e076e82537c171aa326121399c896fe
ea0e198b41b6bae300bb86f8c61d4b170eee4717b5497016f34364a9

scalar_mult_vfy(yb,Ya): (length: 56 bytes)

dc504938fb70eb13916697aa3e076e82537c171aa326121399c896fe
ea0e198b41b6bae300bb86f8c61d4b170eee4717b5497016f34364a9

[B.4.5.](#) Test vector for ISK calculation initiator/responder

unordered cat of transcript : (length: 122 bytes)

```

38223f95a5430a2f2a499431696d23ea2d0a90f432e5491e45e4005f
3dd785e7be1235b79252670099bc993c2df5c261dfb7a8989f091e2b
e30341446138b6ba0a336c103c6c92019ae4cfbc88d8f6bfc361e97
9c9e0d3a0967e630094ba3d1555821ac1f979996ef5ce79f012ffe27
9ac89b287bee03414462
DSI = G.DSI_ISK, b'CPaceDecaf448_ISK': (length: 17 bytes)
435061636544656361663434385f49534b
prefix_free_cat(DSI,sid,K)||MSGa||MSGb: (length: 214 bytes)
11435061636544656361663434385f49534b105223e0cdc45d657566
8d64c55200412438dc504938fb70eb13916697aa3e076e82537c171a
a326121399c896feea0e198b41b6bae300bb86f8c61d4b170eee4717
b5497016f34364a938223f95a5430a2f2a499431696d23ea2d0a90f4
32e5491e45e4005f3dd785e7be1235b79252670099bc993c2df5c261
dfb7a8989f091e2be30341446138b6ba0a336c103c6c92019ae4cfbc
b88d8f6bfc361e979c9e0d3a0967e630094ba3d1555821ac1f979996
ef5ce79f012ffe279ac89b287bee03414462
ISK result: (length: 64 bytes)
ebe28369491f8899a5af3b339d4993881b69d22607c58719da6eaab3
8f0d9025eae413ca2b072b156ce4a0d4778ff471a63c4d908cab70bc
2081951d504cbb03

```

B.4.6. Test vector for ISK calculation parallel execution

```

ordered cat of transcript : (length: 122 bytes)
38b6ba0a336c103c6c92019ae4cfbc88d8f6bfc361e979c9e0d3a09
67e630094ba3d1555821ac1f979996ef5ce79f012ffe279ac89b287b
ee0341446238223f95a5430a2f2a499431696d23ea2d0a90f432e549
1e45e4005f3dd785e7be1235b79252670099bc993c2df5c261dfb7a8
989f091e2be303414461
DSI = G.DSI_ISK, b'CPaceDecaf448_ISK': (length: 17 bytes)
435061636544656361663434385f49534b
prefix_free_cat(DSI,sid,K)||oCAT(MSGa,MSGb):
(length: 214 bytes)
11435061636544656361663434385f49534b105223e0cdc45d657566
8d64c55200412438dc504938fb70eb13916697aa3e076e82537c171a
a326121399c896feea0e198b41b6bae300bb86f8c61d4b170eee4717
b5497016f34364a938b6ba0a336c103c6c92019ae4cfbc88d8f6bfc
361e979c9e0d3a0967e630094ba3d1555821ac1f979996ef5ce79f01
2ffe279ac89b287bee0341446238223f95a5430a2f2a499431696d23
ea2d0a90f432e5491e45e4005f3dd785e7be1235b79252670099bc99

```

```
3c2df5c261dfb7a8989f091e2be303414461
ISK result: (length: 64 bytes)
2996d1953320581b587f473cfd5c974c5a8597b22b37fefe49bdb7b8
4073424f7f7a6e456498665a69530741398c6010bdb346f79944acc9
0c5c537fa35cd29a
```

[B.4.7.](#) Corresponding ANSI-C initializers

```
const uint8_t tc_PRS[] = {
    0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const uint8_t 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 uint8_t tc_sid[] = {
    0x52,0x23,0xe0,0xcd,0xc4,0x5d,0x65,0x75,0x66,0x8d,0x64,0xc5,
    0x52,0x00,0x41,0x24,
};
const uint8_t tc_g[] = {
    0xc8,0x11,0xb3,0xf6,0xb0,0xd2,0x7b,0x58,0xa7,0x4d,0x82,0x74,
    0xbf,0x5f,0x9c,0xa6,0xb7,0xad,0xa1,0x5b,0x0b,0xf5,0x7b,0x79,
    0xa6,0xb4,0x5c,0x13,0x2e,0xb0,0xc2,0x8b,0xdc,0xc3,0xab,0xf4,
    0xe5,0x93,0x2c,0xea,0x97,0xa8,0x09,0x97,0xea,0xd1,0xc1,0x46,
    0xb9,0x8b,0x1a,0x1f,0x1d,0xef,0x30,0xf3,
};
const uint8_t tc_ya[] = {
    0xd8,0xd2,0xe2,0x6c,0x82,0x1a,0x12,0xd7,0xf5,0x9a,0x8d,0xee,
    0x02,0x3d,0x3f,0x61,0x55,0x97,0x61,0x52,0xe1,0x6c,0x73,0xcb,
```

```
    0xf6,0x8c,0x30,0x3d,0xf0,0x40,0x43,0x99,0xf0,0xa7,0xb6,0x14,
    0xa6,0x5d,0xf5,0x0a,0x97,0x88,0xf0,0x0b,0x41,0x05,0x86,0xb4,
    0x43,0xf7,0x38,0xad,0x7f,0xf0,0x39,0x30,
};
const uint8_t tc_ADa[] = {
    0x41,0x44,0x61,
};
const uint8_t tc_Ya[] = {
    0x22,0x3f,0x95,0xa5,0x43,0x0a,0x2f,0x2a,0x49,0x94,0x31,0x69,
    0x6d,0x23,0xea,0x2d,0x0a,0x90,0xf4,0x32,0xe5,0x49,0x1e,0x45,
    0xe4,0x00,0x5f,0x3d,0xd7,0x85,0xe7,0xbe,0x12,0x35,0xb7,0x92,
    0x52,0x67,0x00,0x99,0xbc,0x99,0x3c,0x2d,0xf5,0xc2,0x61,0xdf,
```



```

    0xb7,0xa8,0x98,0x9f,0x09,0x1e,0x2b,0xe3,
};
const uint8_t tc_yb[] = {
    0x91,0xba,0xe9,0x79,0x3f,0x4a,0x8a,0xce,0xb1,0xb5,0xc5,0x43,
    0x75,0xa7,0xed,0x18,0x58,0xa7,0x9a,0x6e,0x72,0xda,0xb9,0x59,
    0xc8,0xbd,0xf3,0xa7,0x5a,0xc9,0xbb,0x4d,0xe2,0xa2,0x5a,0xf4,
    0xd4,0xa9,0xa5,0xc5,0xbc,0x54,0x41,0xd1,0x9b,0x8e,0x3f,0x6f,
    0xcc,0xe7,0x19,0x6c,0x6a,0xfc,0x22,0x36,
};
const uint8_t tc_ADb[] = {
    0x41,0x44,0x62,
};
const uint8_t tc_Yb[] = {
    0xb6,0xba,0x0a,0x33,0x6c,0x10,0x3c,0x6c,0x92,0x01,0x9a,0xe4,
    0xcf,0xbc,0xb8,0x8d,0x8f,0x6b,0xfc,0x36,0x1e,0x97,0x9c,0x9e,
    0x0d,0x3a,0x09,0x67,0xe6,0x30,0x09,0x4b,0xa3,0xd1,0x55,0x58,
    0x21,0xac,0x1f,0x97,0x99,0x96,0xef,0x5c,0xe7,0x9f,0x01,0x2f,
    0xfe,0x27,0x9a,0xc8,0x9b,0x28,0x7b,0xee,
};
const uint8_t tc_K[] = {
    0xdc,0x50,0x49,0x38,0xfb,0x70,0xeb,0x13,0x91,0x66,0x97,0xaa,
    0x3e,0x07,0x6e,0x82,0x53,0x7c,0x17,0x1a,0xa3,0x26,0x12,0x13,
    0x99,0xc8,0x96,0xfe,0xea,0x0e,0x19,0x8b,0x41,0xb6,0xba,0xe3,
    0x00,0xbb,0x86,0xf8,0xc6,0x1d,0x4b,0x17,0x0e,0xee,0x47,0x17,
    0xb5,0x49,0x70,0x16,0xf3,0x43,0x64,0xa9,
};
const uint8_t tc_ISK_IR[] = {
    0xeb,0xe2,0x83,0x69,0x49,0x1f,0x88,0x99,0xa5,0xaf,0x3b,0x33,
    0x9d,0x49,0x93,0x88,0x1b,0x69,0xd2,0x26,0x07,0xc5,0x87,0x19,
    0xda,0x6e,0xaa,0xb3,0x8f,0x0d,0x90,0x25,0xea,0xe4,0x13,0xca,
    0x2b,0x07,0x2b,0x15,0x6c,0xe4,0xa0,0xd4,0x77,0x8f,0xf4,0x71,
    0xa6,0x3c,0x4d,0x90,0x8c,0xab,0x70,0xbc,0x20,0x81,0x95,0x1d,
    0x50,0x4c,0xbb,0x03,
};
const uint8_t tc_ISK_SY[] = {
    0x29,0x96,0xd1,0x95,0x33,0x20,0x58,0x1b,0x58,0x7f,0x47,0x3c,

```

```

    0xfd,0x5c,0x97,0x4c,0x5a,0x85,0x97,0xb2,0x2b,0x37,0xfe,0xfe,
    0x49,0xbd,0xb7,0xb8,0x40,0x73,0x42,0x4f,0x7f,0x7a,0x6e,0x45,
    0x64,0x98,0x66,0x5a,0x69,0x53,0x07,0x41,0x39,0x8c,0x60,0x10,
    0xbd,0xb3,0x46,0xf7,0x99,0x44,0xac,0xc9,0x0c,0x5c,0x53,0x7f,
    0xa3,0x5c,0xd2,0x9a,

```

```
};
```

[B.4.8.](#) Test case for scalar_mult with valid inputs

```
s: (length: 56 bytes)
  dd1bc7015daabb7672129cc35a3ba815486b139deff9bdeca7a4fc61
  34323d34658761e90ff079972a7ca8aa5606498f4f4f0ebc0933a819
X: (length: 56 bytes)
  c803a6c8171ac38b66c5306553f45a487a24eb8581414444715bd2e5
  cf4c749a3b56a550f3c9a6ea3efa6e11ae6a6da12b98ef2f51174b9a
G.scalar_mult(s,decode(X)): (length: 56 bytes)
  b831a1f804fd3c902ae82f731d298aebf9152ea855f5b5da5ee88584
  84c55a7f65cc3ccf5f678496dc4cb1c8d6bc7ed17d2fe535fdc8f60e
G.scalar_mult_vfy(s,X): (length: 56 bytes)
  b831a1f804fd3c902ae82f731d298aebf9152ea855f5b5da5ee88584
  84c55a7f65cc3ccf5f678496dc4cb1c8d6bc7ed17d2fe535fdc8f60e
```

[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)
  c703a6c8171ac38b66c5306553f45a487a24eb8581414444715bd2e5
  cf4c749a3b56a550f3c9a6ea3efa6e11ae6a6da12b98ef2f51174b9a
Y_i2 == G.I: (length: 56 bytes)
  0000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000000
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_'
CI = b'\nAinitiator\nBresponder'
CI = 0a41696e69746961746f720a42726573706f6e646572
sid = 34b36454cab2e7842c389f7d88ecb7df
```

Outputs

```
generator_string(PRS,G.DSI,CI,sid,H.s_in_bytes):
(length: 104 bytes)
1e4350616365503235365f584d443a5348412d3235365f535357555f
4e555f0850617373776f7264170000000000000000000000000000
000000000000000000000000160a41696e69746961746f720a42726573706f6e
6465721034b36454cab2e7842c389f7d88ecb7df
generator g: (length: 65 bytes)
04993b46e30ba9cfc3dc2d3ae2cf9733cf03994e74383c4e1b4a92e8
d6d466b321c4a642979162fbde9e1c9a6180bd27a0594491e4c231f5
1006d0bf7992d07127
```

[B.5.2.](#) Test vector for MSGa

Inputs

```
ADa = b'ADa'
ya (big endian): (length: 32 bytes)
c9e47ca5debd2285727af47e55f5b7763fa79719da428f800190cc66
59b4eafb
```

Outputs

```
Ya: (length: 65 bytes)
0478ac925a6e3447a537627a2163be005a422f55c08385c1ef7d051c
a94593df5946314120faa87165cba131c1da3aac429dc3d99a9bac7d
4c4cbb8570b4d5ea10
Alternative correct value for Ya:  $g^{(-ya)}$ :
(length: 65 bytes)
0478ac925a6e3447a537627a2163be005a422f55c08385c1ef7d051c
a94593df59b9cebede05578e9b345ece3e25c553bd623c2666645382
b3b3447a8f4b2a15ef
MSGa: (length: 70 bytes)
410478ac925a6e3447a537627a2163be005a422f55c08385c1ef7d05
1ca94593df5946314120faa87165cba131c1da3aac429dc3d99a9bac
7d4c4cbb8570b4d5ea1003414461
```

[B.5.3.](#) Test vector for MSGb

Internet-Draft

CPace

January 2022

Inputs

ADb = b'ADb'

yb (big endian): (length: 32 bytes)

```
a0b768ba7555621d133012d1dee27a0013c1bcfddd675811df12771e
44d77b10
```

Outputs

Yb: (length: 65 bytes)

```
04df13ffa89b0ce3cc553b1495ff027886564d94b8d9165cd50e5f65
4247959951bfac90839fca218bf8e2d1258eb7d7d9f733fe4cd558e6
fa57bf1f801aae7d3a
```

Alternative correct value for Yb: $g^{(-yb)}$:

(length: 65 bytes)

```
04df13ffa89b0ce3cc553b1495ff027886564d94b8d9165cd50e5f65
424795995140536f7b6035de75071d2eda7148282608cc01b42aa719
05a840e07fe55182c5
```

MSGb: (length: 70 bytes)

```
4104df13ffa89b0ce3cc553b1495ff027886564d94b8d9165cd50e5f
654247959951bfac90839fca218bf8e2d1258eb7d7d9f733fe4cd558
e6fa57bf1f801aae7d3a03414462
```

[B.5.4.](#) Test vector for secret points K

scalar_mult_vfy(ya,Yb): (length: 32 bytes)

```
27f7059d88f02007dc18c911c9b4034d3c0f13f8f7ed9603b0927f23
fbab1037
```

scalar_mult_vfy(yb,Ya): (length: 32 bytes)

```
27f7059d88f02007dc18c911c9b4034d3c0f13f8f7ed9603b0927f23
fbab1037
```

[B.5.5.](#) Test vector for ISK calculation initiator/responder

Internet-Draft

CPace

January 2022

```
unordered cat of transcript : (length: 140 bytes)
  410478ac925a6e3447a537627a2163be005a422f55c08385c1ef7d05
  1ca94593df5946314120faa87165cba131c1da3aac429dc3d99a9bac
  7d4c4cbb8570b4d5ea10034144614104df13ffa89b0ce3cc553b1495
  ff027886564d94b8d9165cd50e5f654247959951bfac90839fca218b
  f8e2d1258eb7d7d9f733fe4cd558e6fa57bf1f801aae7d3a03414462
DSI = G.DSI_ISK, b'CPaceP256_XMD:SHA-256_SSWU_NU__ISK':
(length: 34 bytes)
  4350616365503235365f584d443a5348412d3235365f535357555f4e
  555f5f49534b
prefix_free_cat(DSI,sid,K)||MSGa||MSGb: (length: 225 bytes)
  224350616365503235365f584d443a5348412d3235365f535357555f
  4e555f5f49534b1034b36454cab2e7842c389f7d88ecb7df2027f705
  9d88f02007dc18c911c9b4034d3c0f13f8f7ed9603b0927f23fbab10
  37410478ac925a6e3447a537627a2163be005a422f55c08385c1ef7d
  051ca94593df5946314120faa87165cba131c1da3aac429dc3d99a9b
  ac7d4c4cbb8570b4d5ea10034144614104df13ffa89b0ce3cc553b14
  95ff027886564d94b8d9165cd50e5f654247959951bfac90839fca21
  8bf8e2d1258eb7d7d9f733fe4cd558e6fa57bf1f801aae7d3a034144
  62
ISK result: (length: 32 bytes)
  ddc1b133c387ecf344c0b496bc1223656cd6e7d99a5def8b3b026796
  50811fc9
```

[B.5.6.](#) Test vector for ISK calculation parallel execution

Internet-Draft

CPace

January 2022

```
ordered cat of transcript : (length: 140 bytes)
4104df13ffa89b0ce3cc553b1495ff027886564d94b8d9165cd50e5f
654247959951bfac90839fca218bf8e2d1258eb7d7d9f733fe4cd558
e6fa57bf1f801aae7d3a03414462410478ac925a6e3447a537627a21
63be005a422f55c08385c1ef7d051ca94593df5946314120faa87165
cba131c1da3aac429dc3d99a9bac7d4c4cbb8570b4d5ea1003414461
DSI = G.DSI_ISK, b'CPaceP256_XMD:SHA-256_SSWU_NU__ISK':
(length: 34 bytes)
4350616365503235365f584d443a5348412d3235365f535357555f4e
555f5f49534b
prefix_free_cat(DSI,sid,K) || oCAT(MSGa,MSGb):
(length: 225 bytes)
224350616365503235365f584d443a5348412d3235365f535357555f
4e555f5f49534b1034b36454cab2e7842c389f7d88ecb7df2027f705
9d88f02007dc18c911c9b4034d3c0f13f8f7ed9603b0927f23fbab10
374104df13ffa89b0ce3cc553b1495ff027886564d94b8d9165cd50e
5f654247959951bfac90839fca218bf8e2d1258eb7d7d9f733fe4cd5
58e6fa57bf1f801aae7d3a03414462410478ac925a6e3447a537627a
2163be005a422f55c08385c1ef7d051ca94593df5946314120faa871
65cba131c1da3aac429dc3d99a9bac7d4c4cbb8570b4d5ea10034144
61
ISK result: (length: 32 bytes)
6ea775b0fb3c31502687565a52150fc595c63fe901a11d5fc1995cd5
089a17ae
```

[B.5.7.](#) Corresponding ANSI-C initializers

```
const uint8_t tc_PRS[] = {
```

```

    0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const uint8_t 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 uint8_t tc_sid[] = {
    0x34,0xb3,0x64,0x54,0xca,0xb2,0xe7,0x84,0x2c,0x38,0x9f,0x7d,
    0x88,0xec,0xb7,0xdf,
};
const uint8_t tc_g[] = {
    0x04,0x99,0x3b,0x46,0xe3,0x0b,0xa9,0xcf,0xc3,0xdc,0x2d,0x3a,
    0xe2,0xcf,0x97,0x33,0xcf,0x03,0x99,0x4e,0x74,0x38,0x3c,0x4e,
    0x1b,0x4a,0x92,0xe8,0xd6,0xd4,0x66,0xb3,0x21,0xc4,0xa6,0x42,
    0x97,0x91,0x62,0xfb,0xde,0x9e,0x1c,0x9a,0x61,0x80,0xbd,0x27,
    0xa0,0x59,0x44,0x91,0xe4,0xc2,0x31,0xf5,0x10,0x06,0xd0,0xbf,
    0x79,0x92,0xd0,0x71,0x27,
};
const uint8_t tc_ya[] = {
    0xc9,0xe4,0x7c,0xa5,0xde,0xbd,0x22,0x85,0x72,0x7a,0xf4,0x7e,

```

```

    0x55,0xf5,0xb7,0x76,0x3f,0xa7,0x97,0x19,0xda,0x42,0x8f,0x80,
    0x01,0x90,0xcc,0x66,0x59,0xb4,0xea,0xfb,
};
const uint8_t tc_ADa[] = {
    0x41,0x44,0x61,
};
const uint8_t tc_Ya[] = {
    0x04,0x78,0xac,0x92,0x5a,0x6e,0x34,0x47,0xa5,0x37,0x62,0x7a,
    0x21,0x63,0xbe,0x00,0x5a,0x42,0x2f,0x55,0xc0,0x83,0x85,0xc1,
    0xef,0x7d,0x05,0x1c,0xa9,0x45,0x93,0xdf,0x59,0x46,0x31,0x41,
    0x20,0xfa,0xa8,0x71,0x65,0xcb,0xa1,0x31,0xc1,0xda,0x3a,0xac,
    0x42,0x9d,0xc3,0xd9,0x9a,0x9b,0xac,0x7d,0x4c,0x4c,0xbb,0x85,
    0x70,0xb4,0xd5,0xea,0x10,
};
const uint8_t tc_yb[] = {
    0xa0,0xb7,0x68,0xba,0x75,0x55,0x62,0x1d,0x13,0x30,0x12,0xd1,
    0xde,0xe2,0x7a,0x00,0x13,0xc1,0xbc,0xfd,0xdd,0x67,0x58,0x11,
    0xdf,0x12,0x77,0x1e,0x44,0xd7,0x7b,0x10,
};
const uint8_t tc_ADb[] = {
    0x41,0x44,0x62,

```

```

};
const uint8_t tc_Yb[] = {
    0x04,0xdf,0x13,0xff,0xa8,0x9b,0x0c,0xe3,0xcc,0x55,0x3b,0x14,
    0x95,0xff,0x02,0x78,0x86,0x56,0x4d,0x94,0xb8,0xd9,0x16,0x5c,
    0xd5,0x0e,0x5f,0x65,0x42,0x47,0x95,0x99,0x51,0xbf,0xac,0x90,
    0x83,0x9f,0xca,0x21,0x8b,0xf8,0xe2,0xd1,0x25,0x8e,0xb7,0xd7,
    0xd9,0xf7,0x33,0xfe,0x4c,0xd5,0x58,0xe6,0xfa,0x57,0xbf,0x1f,
    0x80,0x1a,0xae,0x7d,0x3a,
};
const uint8_t tc_K[] = {
    0x27,0xf7,0x05,0x9d,0x88,0xf0,0x20,0x07,0xdc,0x18,0xc9,0x11,
    0xc9,0xb4,0x03,0x4d,0x3c,0x0f,0x13,0xf8,0xf7,0xed,0x96,0x03,
    0xb0,0x92,0x7f,0x23,0xfb,0xab,0x10,0x37,
};
const uint8_t tc_ISK_IR[] = {
    0xdd,0xc1,0xb1,0x33,0xc3,0x87,0xec,0xf3,0x44,0xc0,0xb4,0x96,
    0xbc,0x12,0x23,0x65,0x6c,0xd6,0xe7,0xd9,0x9a,0x5d,0xef,0x8b,
    0x3b,0x02,0x67,0x96,0x50,0x81,0x1f,0xc9,
};
const uint8_t tc_ISK_SY[] = {
    0x6e,0xa7,0x75,0xb0,0xfb,0x3c,0x31,0x50,0x26,0x87,0x56,0x5a,
    0x52,0x15,0x0f,0xc5,0x95,0xc6,0x3f,0xe9,0x01,0xa1,0x1d,0x5f,
    0xc1,0x99,0x5c,0xd5,0x08,0x9a,0x17,0xae,
};

```

[B.5.8.](#) Test case for scalar_mult_vfy with correct inputs

```

s: (length: 32 bytes)
  f012501c091ff9b99a123fffe571d8bc01e8077ee581362e1bd21399
  0835643b
X: (length: 65 bytes)
  0476ab88669dc640ca098b3d19ed87084d22d7e7c86b3b87451554d6
  93a7d98fb6bf0a6938fe0cec7be7563499ba3792909c8b9f4c936ef5
  2828b78a8d6254f49c
G.scalar_mult(s,X) (full coordinates): (length: 65 bytes)
  0492b0eb1fe6a988797a85e6de8ec5de7ec685c83164570d79f0d568
  b918bfe7718b049dac20ea4631d8c4f321ddb48d70416f4929eb9a85
  2528114d3a560537c7
G.scalar_mult_vfy(s,X) (only X-coordinate):
(length: 32 bytes)
  92b0eb1fe6a988797a85e6de8ec5de7ec685c83164570d79f0d568b9

```


18bfe771

[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)
  0476ab88669dc640ca098b3d19ed87084d22d7e7c86b3b87451554d6
  93a7d98fb6bf0a6938fe0cec7be7563499ba3792909c8b9f4c936ef5
  2828b78a8d6254f4f3
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_'
CI  = b'\nAinitiator\nBresponder'
CI  = 0a41696e69746961746f720a42726573706f6e646572
sid = 5b3773aa90e8f23c61563a4b645b276c
```

```
generator_string(PRS,G.DSI,CI,sid,H.s_in_bytes):
(length: 168 bytes)
1e4350616365503338345f584d443a5348412d3338345f5353575551
4e555f0850617373776f72645700000000000000000000000000
000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000
0a42726573706f6e646572105b3773aa90e8f23c61563a4b645b276c
generator g: (length: 97 bytes)
04bb6f046a601d0a0b134c6221e20e83c3f9ac0390be56c5a95b68eb
f41c82ade6f4977ea21341239d194c38dabd1a7eb5887d9fed2550a1
d5e6789327f2a039cd9c41239b240f775f5f2bef8744561b3a7e98f3
2234cb1b318f66616de777aeef
```

```
Inputs
ADa = b'ADa'
ya (big endian): (length: 48 bytes)
ef433dd5ad142c860e7cb6400dd315d388d5ec5420c550e9d6f0907f
375d988bc4d704837e43561c497e7dd93edc9db9d

Outputs
Ya: (length: 97 bytes)
047214fc512921b3fa0b555b41d841c9c20227fa1ab0dda5bfc051f6
de9be7983e6df11d4e8da738b739adfbdb85d8f5e80b2b4bbc66f3dff
c02136ee19773d05f9c0242c0dd51857763de98a2fdfec73a4b1010c
bc419c7b23b50adedbb3ff6644
Alternative correct value for Ya: g^(-ya):
(length: 97 bytes)
047214fc512921b3fa0b555b41d841c9c20227fa1ab0dda5bfc051f6
de9be7983e6df11d4e8da738b739adfbdb85d8f5e804d4b443990c200
3fdec911e688c2fa063fdbd3f22ae7a889c21675d020138c5a4efef3
42be6384dc4af521254c0099bb
MSGa: (length: 102 bytes)
61047214fc512921b3fa0b555b41d841c9c20227fa1ab0dda5bfc051
f6de9be7983e6df11d4e8da738b739adfbdb85d8f5e80b2b4bbc66f3d
ffc02136ee19773d05f9c0242c0dd51857763de98a2fdfec73a4b101
0cbc419c7b23b50adedbb3ff664403414461
```

[B.6.3.](#) Test vector for MSGb

Inputs

ADb = b'ADb'

yb (big endian): (length: 48 bytes)

50b0e36b95a2edfaa8342b843dddc90b175330f2399c1b36586dedda
3c255975f30be6a750f9404fccc62a6323b5e471

Outputs

Yb: (length: 97 bytes)

04e34cbd45b13ad11552ea7100b19899fa52662e268f2086e21262f7
46efcb18e4b51ecfaf2e8ebab82addb6245f9bb1ff8138317c8045c4
d2550e1566832b94acb91b670c4c4c00e59f5c15c74d4260e490caca
aa860c11b8f369b72d5871bd94

Alternative correct value for Yb: $g^{(-yb)}$:

(length: 97 bytes)

04e34cbd45b13ad11552ea7100b19899fa52662e268f2086e21262f7
46efcb18e4b51ecfaf2e8ebab82addb6245f9bb1ff7ec7ce837fba3b
2daaf1ea997cd46b5346e498f3b3b3ff1a60a3ea38b2bd9f1a6f3535
5479f3ee470c9648d3a78e426b

MSGb: (length: 102 bytes)

6104e34cbd45b13ad11552ea7100b19899fa52662e268f2086e21262
f746efcb18e4b51ecfaf2e8ebab82addb6245f9bb1ff8138317c8045
c4d2550e1566832b94acb91b670c4c4c00e59f5c15c74d4260e490ca
caaa860c11b8f369b72d5871bd9403414462

[B.6.4.](#) Test vector for secret points K

scalar_mult_vfy(ya,Yb): (length: 48 bytes)

e5ef578c410effb4ec114998a59fa5832f6101be479f1a97b021f224
e378c3fb1f77f87a92e39fb415edf5458b3815bf

scalar_mult_vfy(yb,Ya): (length: 48 bytes)

e5ef578c410effb4ec114998a59fa5832f6101be479f1a97b021f224
e378c3fb1f77f87a92e39fb415edf5458b3815bf

[B.6.5.](#) Test vector for ISK calculation initiator/responder

Internet-Draft

CPace

January 2022

```
unordered cat of transcript : (length: 204 bytes)
61047214fc512921b3fa0b555b41d841c9c20227fa1ab0dda5bfc051
f6de9be7983e6df11d4e8da738b739adfbfd85d8f5e80b2b4bbc66f3d
ffc02136ee19773d05f9c0242c0dd51857763de98a2fdfec73a4b101
0cbc419c7b23b50adedbb3ff6644034144616104e34cbd45b13ad115
52ea7100b19899fa52662e268f2086e21262f746efcb18e4b51ecfaf
2e8ebab82addb6245f9bb1ff8138317c8045c4d2550e1566832b94ac
b91b670c4c4c00e59f5c15c74d4260e490cacaaa860c11b8f369b72d
5871bd9403414462
DSI = G.DSI_ISK, b'CPaceP384_XMD:SHA-384_SSWU_NU__ISK':
(length: 34 bytes)
4350616365503338345f584d443a5348412d3338345f535357555f4e
555f5f49534b
prefix_free_cat(DSI,sid,K)||MSGa||MSGb: (length: 305 bytes)
224350616365503338345f584d443a5348412d3338345f535357555f
4e555f5f49534b105b3773aa90e8f23c61563a4b645b276c30e5ef57
8c410effb4ec114998a59fa5832f6101be479f1a97b021f224e378c3
fb1f77f87a92e39fb415edf5458b3815bf61047214fc512921b3fa0b
555b41d841c9c20227fa1ab0dda5bfc051f6de9be7983e6df11d4e8d
a738b739adfbfd85d8f5e80b2b4bbc66f3dffc02136ee19773d05f9c0
242c0dd51857763de98a2fdfec73a4b1010cbc419c7b23b50adedbb3
ff6644034144616104e34cbd45b13ad11552ea7100b19899fa52662e
268f2086e21262f746efcb18e4b51ecfaf2e8ebab82addb6245f9bb1
ff8138317c8045c4d2550e1566832b94acb91b670c4c4c00e59f5c15
c74d4260e490cacaaa860c11b8f369b72d5871bd9403414462
ISK result: (length: 48 bytes)
401601de4a9f25bd57fc85985c9abf1de75191d68306b584547e6ac9
e959cf2df49a9bf2205c3617ce99a169971bdbf8
```

B.6.6. Test vector for ISK calculation parallel execution

```
ordered cat of transcript : (length: 204 bytes)
6104e34cbd45b13ad11552ea7100b19899fa52662e268f2086e21262
f746efcb18e4b51ecfaf2e8ebab82addb6245f9bb1ff8138317c8045
c4d2550e1566832b94acb91b670c4c4c00e59f5c15c74d4260e490ca
caaa860c11b8f369b72d5871bd940341446261047214fc512921b3fa
0b555b41d841c9c20227fa1ab0dda5bfc051f6de9be7983e6df11d4e
8da738b739adfb85d8f5e80b2b4bbc66f3dfffc02136ee19773d05f9
c0242c0dd51857763de98a2fdfec73a4b1010cbc419c7b23b50adedb
b3ff664403414461
DSI = G.DSI_ISK, b'CPaceP384_XMD:SHA-384_SSWU_NU__ISK':
(length: 34 bytes)
4350616365503338345f584d443a5348412d3338345f535357555f4e
555f5f49534b
prefix_free_cat(DSI,sid,K) || oCAT(MSGa,MSGb):
(length: 305 bytes)
224350616365503338345f584d443a5348412d3338345f535357555f
4e555f5f49534b105b3773aa90e8f23c61563a4b645b276c30e5ef57
8c410effb4ec114998a59fa5832f6101be479f1a97b021f224e378c3
fb1f77f87a92e39fb415edf5458b3815bf6104e34cbd45b13ad11552
ea7100b19899fa52662e268f2086e21262f746efcb18e4b51ecfaf2e
8ebab82addb6245f9bb1ff8138317c8045c4d2550e1566832b94acb9
1b670c4c4c00e59f5c15c74d4260e490caca860c11b8f369b72d58
71bd940341446261047214fc512921b3fa0b555b41d841c9c20227fa
1ab0dda5bfc051f6de9be7983e6df11d4e8da738b739adfb85d8f5e
80b2b4bbc66f3dfffc02136ee19773d05f9c0242c0dd51857763de98a
2fdfec73a4b1010cbc419c7b23b50adedbb3ff664403414461
ISK result: (length: 48 bytes)
1eb17f7f7126a07acd510e9d60c84f63dc0113ac34f8d359e8f692a9
06f828bde926d9ff65202c9801e9884aa05a43b6
```

[B.6.7.](#) Corresponding ANSI-C initializers

```
const uint8_t tc_PRS[] = {
    0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const uint8_t 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 uint8_t tc_sid[] = {
    0x5b,0x37,0x73,0xaa,0x90,0xe8,0xf2,0x3c,0x61,0x56,0x3a,0x4b,
    0x64,0x5b,0x27,0x6c,
};
const uint8_t tc_g[] = {
    0x04,0xbb,0x6f,0x04,0x6a,0x60,0x1d,0x0a,0x0b,0x13,0x4c,0x62,
    0x21,0xe2,0x0e,0x83,0xc3,0xf9,0xac,0x03,0x90,0xbe,0x56,0xc5,
    0xa9,0x5b,0x68,0xeb,0xf4,0x1c,0x82,0xad,0xe6,0xf4,0x97,0x7e,
    0xa2,0x13,0x41,0x23,0x9d,0x19,0x4c,0x38,0xda,0xbd,0x1a,0x7e,

```

```

    0xb5,0x88,0x7d,0x9f,0xed,0x25,0x50,0xa1,0xd5,0xe6,0x78,0x93,
    0x27,0xf2,0xa0,0x39,0xcd,0x9c,0x41,0x23,0x9b,0x24,0x0f,0x77,
    0x5f,0x5f,0x2b,0xef,0x87,0x44,0x56,0x1b,0x3a,0x7e,0x98,0xf3,
    0x22,0x34,0xcb,0x1b,0x31,0x8f,0x66,0x61,0x6d,0xe7,0x77,0xae,
    0xef,
};
const uint8_t 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 uint8_t tc_ADa[] = {
    0x41,0x44,0x61,
};
const uint8_t tc_Ya[] = {
    0x04,0x72,0x14,0xfc,0x51,0x29,0x21,0xb3,0xfa,0x0b,0x55,0x5b,
    0x41,0xd8,0x41,0xc9,0xc2,0x02,0x27,0xfa,0x1a,0xb0,0xdd,0xa5,
    0xbf,0xc0,0x51,0xf6,0xde,0x9b,0xe7,0x98,0x3e,0x6d,0xf1,0x1d,
    0x4e,0x8d,0xa7,0x38,0xb7,0x39,0xad,0xfb,0xd8,0x5d,0x8f,0x5e,
    0x80,0xb2,0xb4,0xbb,0xc6,0x6f,0x3d,0xff,0xc0,0x21,0x36,0xee,
    0x19,0x77,0x3d,0x05,0xf9,0xc0,0x24,0x2c,0x0d,0xd5,0x18,0x57,
    0x76,0x3d,0xe9,0x8a,0x2f,0xdf,0xec,0x73,0xa4,0xb1,0x01,0x0c,
    0xbc,0x41,0x9c,0x7b,0x23,0xb5,0x0a,0xde,0xdb,0xb3,0xff,0x66,
    0x44,
};
const uint8_t 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 uint8_t tc_ADb[] = {
    0x41,0x44,0x62,
};
const uint8_t tc_Yb[] = {
    0x04,0xe3,0x4c,0xbd,0x45,0xb1,0x3a,0xd1,0x15,0x52,0xea,0x71,
    0x00,0xb1,0x98,0x99,0xfa,0x52,0x66,0x2e,0x26,0x8f,0x20,0x86,
    0xe2,0x12,0x62,0xf7,0x46,0xef,0xcb,0x18,0xe4,0xb5,0x1e,0xcf,
    0xaf,0x2e,0x8e,0xba,0xb8,0x2a,0xdd,0xb6,0x24,0x5f,0x9b,0xb1,
    0xff,0x81,0x38,0x31,0x7c,0x80,0x45,0xc4,0xd2,0x55,0x0e,0x15,
    0x66,0x83,0x2b,0x94,0xac,0xb9,0x1b,0x67,0x0c,0x4c,0x4c,0x00,
    0xe5,0x9f,0x5c,0x15,0xc7,0x4d,0x42,0x60,0xe4,0x90,0xca,0xca,
    0xaa,0x86,0x0c,0x11,0xb8,0xf3,0x69,0xb7,0x2d,0x58,0x71,0xbd,
    0x94,
};
const uint8_t tc_K[] = {
    0xe5,0xef,0x57,0x8c,0x41,0x0e,0xff,0xb4,0xec,0x11,0x49,0x98,

```

```

    0xa5,0x9f,0xa5,0x83,0x2f,0x61,0x01,0xbe,0x47,0x9f,0x1a,0x97,
    0xb0,0x21,0xf2,0x24,0xe3,0x78,0xc3,0xfb,0x1f,0x77,0xf8,0x7a,
    0x92,0xe3,0x9f,0xb4,0x15,0xed,0xf5,0x45,0x8b,0x38,0x15,0xbf,
};
const uint8_t tc_ISK_IR[] = {
    0x40,0x16,0x01,0xde,0x4a,0x9f,0x25,0xbd,0x57,0xfc,0x85,0x98,
    0x5c,0x9a,0xbf,0x1d,0xe7,0x51,0x91,0xd6,0x83,0x06,0xb5,0x84,
    0x54,0x7e,0x6a,0xc9,0xe9,0x59,0xcf,0x2d,0xf4,0x9a,0x9b,0xf2,
    0x20,0x5c,0x36,0x17,0xce,0x99,0xa1,0x69,0x97,0x1b,0xdb,0xf8,
};
const uint8_t tc_ISK_SY[] = {
    0x1e,0xb1,0x7f,0x7f,0x71,0x26,0xa0,0x7a,0xcd,0x51,0x0e,0x9d,
    0x60,0xc8,0x4f,0x63,0xdc,0x01,0x13,0xac,0x34,0xf8,0xd3,0x59,
    0xe8,0xf6,0x92,0xa9,0x06,0xf8,0x28,0xbd,0xe9,0x26,0xd9,0xff,
    0x65,0x20,0x2c,0x98,0x01,0xe9,0x88,0x4a,0xa0,0x5a,0x43,0xb6,
};

```

[B.6.8.](#) Test case for scalar_mult_vfy with correct inputs

s: (length: 48 bytes)

```

6e8a99a5cdd408eae98e1b8aed286e7b12adbbdac7f2c628d9060ce9
2ae0d90bd57a564fd3500fbcce3425dc94ba0ade

```

```

X: (length: 97 bytes)
  04a32d8d8e1057d37b090d92f46d0bac1874e6cd7c13774774385c30
  39fa8fa3539884b436e49743d2d6279f5bd69dda5fe79fc6ecfb8547
  bf32d8c64ac51f177a70041a1300944f255eea38ca7e964c9d02c5e7
  e28d744e7cdc0bd80437363999
G.scalar_mult(s,X) (full coordinates): (length: 97 bytes)
  045eb8202664ec20fed23ed6005c7be398174946a0f6a8a2e5fd2fed
  9ca159f22652899f820a2d472f926f57de30035a9d11e8006fb66e79
  f3db5d58bd5688954c7284d1e4a616a935dfb761955be13d29de5745
  074a863140dcc9a5c0056ced3b
G.scalar_mult_vfy(s,X) (only X-coordinate):
(length: 48 bytes)
  5eb8202664ec20fed23ed6005c7be398174946a0f6a8a2e5fd2fed9c
  a159f22652899f820a2d472f926f57de30035a9d

```

[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)
  04a32d8d8e1057d37b090d92f46d0bac1874e6cd7c13774774385c30
  39fa8fa3539884b436e49743d2d6279f5bd69dda5fe79fc6ecfb8547
  bf32d8c64ac51f177a70041a1300944f255eea38ca7e964c9d02c5e7
  e28d744e7cdc0bd80437363938
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


```
H = SHA-512 with input block size 128 bytes.
PRS = b'Password' ; ZPAD length: 87 ;
DSI = b'CPaceP521_XMD:SHA-512_SSWU_NU_'
CI = b'\nAinitiator\nBresponder'
CI = 0a41696e69746961746f7220a42726573706f6e646572
sid = 7e4b4791d6a8ef019b936c79fb7f2c57
```

[illegible]

Abdalla, et al. Expires 18 July 2022 [Page 64]

```
Inputs
  ADa = b'ADa'
  ya (big endian): (length: 66 bytes)
    016fac7bb757452e7b788d68a1510eda90113c65db1213fa08927d50
    bcf2635fd66ca254e82927071001353e265082fd609af47ad06fab42
    0c2295df4056ee9ff997
Outputs
  Ya: (length: 133 bytes)
```

```
0400484dcee6d54cb356830cd764079360a03b06a7db1a82188e09c9
2e02d7e78a1e3710da9554db11697d242893e2114d6cbee89f5999b7
e545d9fdf59f4c9acd408901ad73e01ec22ae6ecc122cf257e81826e
348cd410ddb9245c61889fe97b2bbb98b2038eb2ed23e989ec7013a6
10fb2f3b4fb958cc860dd10c98745b9d89e37f2bf9
```

Alternative correct value for Ya: $g^{(-ya)}$:

(length: 133 bytes)

```
0400484dcee6d54cb356830cd764079360a03b06a7db1a82188e09c9
2e02d7e78a1e3710da9554db11697d242893e2114d6cbee89f5999b7
e545d9fdf59f4c9acd408900528c1fe13dd519133edd30da817e7d91
cb732bef2246dba39e77601684d444674dfc714d12dc1676138fec59
ef04d0c4b046a73379f22ef3678ba462761c80d406
```

MSGa: (length: 139 bytes)

```
85010400484dcee6d54cb356830cd764079360a03b06a7db1a82188e
09c92e02d7e78a1e3710da9554db11697d242893e2114d6cbee89f59
99b7e545d9fdf59f4c9acd408901ad73e01ec22ae6ecc122cf257e81
826e348cd410ddb9245c61889fe97b2bbb98b2038eb2ed23e989ec70
13a610fb2f3b4fb958cc860dd10c98745b9d89e37f2bf903414461
```

B.7.3. Test vector for MSGb

```

Adb = b'Adb'
yb (big endian): (length: 66 bytes)
011a946e2d0f48dc440ae3f4fd9126198237042fd1d41d037068c284
6d43ec130cbc55ef1208496be068f8682bcaf6156e51598e27c1fb24
d77b43957bbc129bab80
Outputs
Yb: (length: 133 bytes)
0401edf767bd7d9e67ff137b8f3210c55e9192e9ac8a10f32a2f0eef
9ce34524a543e0d4eb9b3328ca114b02ab23b291f61b5bc814639a9e
caff07e870733131747637004c2df1bec8abe6b252e7fe91bdb6f724
2e65c36e7b960646c89aaf0262a4803ee4c90d1b58775a409a135bd1
8fedbf4ba0eae172b4fe8a0fada83d699e44f2f861
Alternative correct value for Yb: g^(-yb):
(length: 133 bytes)
0401edf767bd7d9e67ff137b8f3210c55e9192e9ac8a10f32a2f0eef
9ce34524a543e0d4eb9b3328ca114b02ab23b291f61b5bc814639a9e
caff07e87073313174763701b3d20e413754194dad18016e424908db
d19a3c918469f9b9376550fd9d5b7fc11b36f2e4a788a5bf65eca42e
701240b45f151e8d4b0175f05257c29661bb0d079e
MSGb: (length: 139 bytes)
85010401edf767bd7d9e67ff137b8f3210c55e9192e9ac8a10f32a2f
0eef9ce34524a543e0d4eb9b3328ca114b02ab23b291f61b5bc81463
9a9ecaff07e870733131747637004c2df1bec8abe6b252e7fe91bdb6
f7242e65c36e7b960646c89aaf0262a4803ee4c90d1b58775a409a13
5bd18fedbf4ba0eae172b4fe8a0fada83d699e44f2f86103414462

```

B.7.4. Test vector for secret points K

```

scalar_mult_vfy(ya,Yb): (length: 66 bytes)
0070a7460122c65d86bf9dd012ab45fc94be362619d1a1f0e75f1433
3ed8b873b5724616b88dadaaba5f28bb783aeb01f60df5fdb8c0a237
45900f462f405debfd51
scalar_mult_vfy(yb,Ya): (length: 66 bytes)
0070a7460122c65d86bf9dd012ab45fc94be362619d1a1f0e75f1433
3ed8b873b5724616b88dadaaba5f28bb783aeb01f60df5fdb8c0a237
45900f462f405debfd51

```

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

```
unordered cat of transcript : (length: 278 bytes)
85010400484dcee6d54cb356830cd764079360a03b06a7db1a82188e
09c92e02d7e78a1e3710da9554db11697d242893e2114d6cbee89f59
99b7e545d9fdf59f4c9acd408901ad73e01ec22ae6ecc122cf257e81
826e348cd410ddb9245c61889fe97b2bbb98b2038eb2ed23e989ec70
13a610fb2f3b4fb958cc860dd10c98745b9d89e37f2bf90341446185
010401edf767bd7d9e67ff137b8f3210c55e9192e9ac8a10f32a2f0e
ef9ce34524a543e0d4eb9b3328ca114b02ab23b291f61b5bc814639a
9ecaff07e870733131747637004c2df1bec8abe6b252e7fe91bdb6f7
242e65c36e7b960646c89aaf0262a4803ee4c90d1b58775a409a135b
d18fedbf4ba0eae172b4fe8a0fada83d699e44f2f86103414462
DSI = G.DSI_ISK, b'CPaceP521_XMD:SHA-512_SSWU_NU__ISK':
(length: 34 bytes)
4350616365503532315f584d443a5348412d3531325f535357555f4e
555f5f49534b
prefix_free_cat(DSI,sid,K)||MSGa||MSGb: (length: 397 bytes)
224350616365503532315f584d443a5348412d3531325f535357555f
4e555f5f49534b107e4b4791d6a8ef019b936c79fb7f2c57420070a7
460122c65d86bf9dd012ab45fc94be362619d1a1f0e75f14333ed8b8
73b5724616b88dadaaba5f28bb783aeb01f60df5fdb8c0a23745900f
462f405debfd5185010400484dcee6d54cb356830cd764079360a03b
06a7db1a82188e09c92e02d7e78a1e3710da9554db11697d242893e2
114d6cbee89f5999b7e545d9fdf59f4c9acd408901ad73e01ec22ae6
ecc122cf257e81826e348cd410ddb9245c61889fe97b2bbb98b2038e
b2ed23e989ec7013a610fb2f3b4fb958cc860dd10c98745b9d89e37f
2bf90341446185010401edf767bd7d9e67ff137b8f3210c55e9192e9
ac8a10f32a2f0eef9ce34524a543e0d4eb9b3328ca114b02ab23b291
f61b5bc814639a9ecaff07e870733131747637004c2df1bec8abe6b2
52e7fe91bdb6f7242e65c36e7b960646c89aaf0262a4803ee4c90d1b
58775a409a135bd18fedbf4ba0eae172b4fe8a0fada83d699e44f2f8
6103414462
ISK result: (length: 64 bytes)
2b2c534c352c446773bd334fac2f2c50ef8cd7991bd4e070f85b0367
a2f7ffca445066cf20b756773687e1038b170896ec2677fe722acb0f
9e6c2f11830e808a
```

B.7.6. Test vector for ISK calculation parallel execution

Internet-Draft

CPace

January 2022

```
ordered cat of transcript : (length: 278 bytes)
85010401edf767bd7d9e67ff137b8f3210c55e9192e9ac8a10f32a2f
0eef9ce34524a543e0d4eb9b3328ca114b02ab23b291f61b5bc81463
9a9ecaff07e870733131747637004c2df1bec8abe6b252e7fe91bdb6
f7242e65c36e7b960646c89aaf0262a4803ee4c90d1b58775a409a13
5bd18fedbf4ba0eae172b4fe8a0fada83d699e44f2f8610341446285
010400484dcee6d54cb356830cd764079360a03b06a7db1a82188e09
c92e02d7e78a1e3710da9554db11697d242893e2114d6cbee89f5999
b7e545d9fdf59f4c9acd408901ad73e01ec22ae6ecc122cf257e8182
6e348cd410ddb9245c61889fe97b2bbb98b2038eb2ed23e989ec7013
a610fb2f3b4fb958cc860dd10c98745b9d89e37f2bf903414461
DSI = G.DSI_ISK, b'CPaceP521_XMD:SHA-512_SSWU_NU__ISK':
(length: 34 bytes)
4350616365503532315f584d443a5348412d3531325f535357555f4e
555f5f49534b
prefix_free_cat(DSI,sid,K) || oCAT(MSGa,MSGb):
(length: 397 bytes)
224350616365503532315f584d443a5348412d3531325f535357555f
4e555f5f49534b107e4b4791d6a8ef019b936c79fb7f2c57420070a7
460122c65d86bf9dd012ab45fc94be362619d1a1f0e75f14333ed8b8
73b5724616b88dadaaba5f28bb783aeb01f60df5fdb8c0a23745900f
462f405debfd5185010401edf767bd7d9e67ff137b8f3210c55e9192
e9ac8a10f32a2f0eef9ce34524a543e0d4eb9b3328ca114b02ab23b2
91f61b5bc814639a9ecaff07e870733131747637004c2df1bec8abe6
b252e7fe91bdb6f7242e65c36e7b960646c89aaf0262a4803ee4c90d
1b58775a409a135bd18fedbf4ba0eae172b4fe8a0fada83d699e44f2
f8610341446285010400484dcee6d54cb356830cd764079360a03b06
a7db1a82188e09c92e02d7e78a1e3710da9554db11697d242893e211
4d6cbee89f5999b7e545d9fdf59f4c9acd408901ad73e01ec22ae6ec
c122cf257e81826e348cd410ddb9245c61889fe97b2bbb98b2038eb2
ed23e989ec7013a610fb2f3b4fb958cc860dd10c98745b9d89e37f2b
f903414461
ISK result: (length: 64 bytes)
78c4dd7136309a2bbe1fdef3cf24a08690006b0c9de253b770c147dd
0800681c82e4e67a388ed1cd9182e595b8e9e3f2976a0e6dab48b2cd
205b19489e20f571
```

B.7.7. Corresponding ANSI-C initializers

```

const uint8_t tc_PRS[] = {
    0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const uint8_t 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 uint8_t tc_sid[] = {
    0x7e,0x4b,0x47,0x91,0xd6,0xa8,0xef,0x01,0x9b,0x93,0x6c,0x79,

```

```

    0xfb,0x7f,0x2c,0x57,
};
const uint8_t tc_g[] = {
    0x04,0x00,0x52,0x3c,0x2b,0xe7,0x5a,0x6f,0xdb,0x50,0xe3,0x3d,
    0x91,0x75,0x97,0xf1,0x82,0x81,0x0e,0xa6,0xaf,0xe0,0x4b,0x72,
    0x97,0xfc,0xcd,0xfc,0xf8,0xc1,0xc9,0xf0,0xf1,0xa0,0xc7,0x94,
    0x05,0x6c,0x72,0x9c,0x27,0x5a,0x65,0x4d,0x1f,0x9f,0x52,0xcd,
    0x3d,0x1d,0x0e,0xcc,0x8f,0x2f,0x6a,0x1b,0xab,0x95,0x8d,0x36,
    0xcc,0x53,0x9c,0x55,0x84,0x96,0xa9,0x01,0xbb,0xe4,0xfd,0x57,
    0x3f,0x2a,0x6e,0x6c,0xc0,0xc9,0xaf,0xee,0x3e,0xe2,0x5c,0x4b,
    0x6f,0x04,0x74,0xdd,0x01,0x2e,0xff,0x5a,0xf0,0xcb,0xf5,0x5c,
    0x4e,0xc3,0xc0,0xab,0x4f,0x11,0x87,0x35,0x3f,0x81,0x5e,0xb2,
    0xa0,0x1e,0xbc,0x52,0xd0,0x76,0xd4,0x5a,0x77,0xa9,0xb8,0x6d,
    0x14,0xfb,0x21,0x06,0x6d,0xf1,0xd0,0x9f,0x10,0xb0,0xa9,0x75,
    0x46,
};
const uint8_t tc_ya[] = {
    0x01,0x6f,0xac,0x7b,0xb7,0x57,0x45,0x2e,0x7b,0x78,0x8d,0x68,
    0xa1,0x51,0x0e,0xda,0x90,0x11,0x3c,0x65,0xdb,0x12,0x13,0xfa,
    0x08,0x92,0x7d,0x50,0xbc,0xf2,0x63,0x5f,0xd6,0x6c,0xa2,0x54,
    0xe8,0x29,0x27,0x07,0x10,0x01,0x35,0x3e,0x26,0x50,0x82,0xfd,
    0x60,0x9a,0xf4,0x7a,0xd0,0x6f,0xab,0x42,0x0c,0x22,0x95,0xdf,
    0x40,0x56,0xee,0x9f,0xf9,0x97,
};
const uint8_t tc_ADa[] = {
    0x41,0x44,0x61,
};
const uint8_t tc_Ya[] = {
    0x04,0x00,0x48,0x4d,0xce,0xe6,0xd5,0x4c,0xb3,0x56,0x83,0x0c,
    0xd7,0x64,0x07,0x93,0x60,0xa0,0x3b,0x06,0xa7,0xdb,0x1a,0x82,
    0x18,0x8e,0x09,0xc9,0x2e,0x02,0xd7,0xe7,0x8a,0x1e,0x37,0x10,
    0xda,0x95,0x54,0xdb,0x11,0x69,0x7d,0x24,0x28,0x93,0xe2,0x11,

```

```

0x4d,0x6c,0xbe,0xe8,0x9f,0x59,0x99,0xb7,0xe5,0x45,0xd9,0xfd,
0xf5,0x9f,0x4c,0x9a,0xcd,0x40,0x89,0x01,0xad,0x73,0xe0,0x1e,
0xc2,0x2a,0xe6,0xec,0xc1,0x22,0xcf,0x25,0x7e,0x81,0x82,0x6e,
0x34,0x8c,0xd4,0x10,0xdd,0xb9,0x24,0x5c,0x61,0x88,0x9f,0xe9,
0x7b,0x2b,0xbb,0x98,0xb2,0x03,0x8e,0xb2,0xed,0x23,0xe9,0x89,
0xec,0x70,0x13,0xa6,0x10,0xfb,0x2f,0x3b,0x4f,0xb9,0x58,0xcc,
0x86,0x0d,0xd1,0x0c,0x98,0x74,0x5b,0x9d,0x89,0xe3,0x7f,0x2b,
0xf9,
};
const uint8_t tc_yb[] = {
0x01,0x1a,0x94,0x6e,0x2d,0x0f,0x48,0xdc,0x44,0x0a,0xe3,0xf4,
0xfd,0x91,0x26,0x19,0x82,0x37,0x04,0x2f,0xd1,0xd4,0x1d,0x03,
0x70,0x68,0xc2,0x84,0x6d,0x43,0xec,0x13,0x0c,0xbc,0x55,0xef,
0x12,0x08,0x49,0x6b,0xe0,0x68,0xf8,0x68,0x2b,0xca,0xf6,0x15,
0x6e,0x51,0x59,0x8e,0x27,0xc1,0xfb,0x24,0xd7,0x7b,0x43,0x95,
0x7b,0xbc,0x12,0x9b,0xab,0x80,

```

```

};
const uint8_t tc_ADb[] = {
0x41,0x44,0x62,
};
const uint8_t tc_Yb[] = {
0x04,0x01,0xed,0xf7,0x67,0xbd,0x7d,0x9e,0x67,0xff,0x13,0x7b,
0x8f,0x32,0x10,0xc5,0x5e,0x91,0x92,0xe9,0xac,0x8a,0x10,0xf3,
0x2a,0x2f,0x0e,0xef,0x9c,0xe3,0x45,0x24,0xa5,0x43,0xe0,0xd4,
0xeb,0x9b,0x33,0x28,0xca,0x11,0x4b,0x02,0xab,0x23,0xb2,0x91,
0xf6,0x1b,0x5b,0xc8,0x14,0x63,0x9a,0x9e,0xca,0xff,0x07,0xe8,
0x70,0x73,0x31,0x31,0x74,0x76,0x37,0x00,0x4c,0x2d,0xf1,0xbe,
0xc8,0xab,0xe6,0xb2,0x52,0xe7,0xfe,0x91,0xbd,0xb6,0xf7,0x24,
0x2e,0x65,0xc3,0x6e,0x7b,0x96,0x06,0x46,0xc8,0x9a,0xaf,0x02,
0x62,0xa4,0x80,0x3e,0xe4,0xc9,0x0d,0x1b,0x58,0x77,0x5a,0x40,
0x9a,0x13,0x5b,0xd1,0x8f,0xed,0xbf,0x4b,0xa0,0xea,0xe1,0x72,
0xb4,0xfe,0x8a,0x0f,0xad,0xa8,0x3d,0x69,0x9e,0x44,0xf2,0xf8,
0x61,
};
const uint8_t tc_K[] = {
0x00,0x70,0xa7,0x46,0x01,0x22,0xc6,0x5d,0x86,0xbf,0x9d,0xd0,
0x12,0xab,0x45,0xfc,0x94,0xbe,0x36,0x26,0x19,0xd1,0xa1,0xf0,
0xe7,0x5f,0x14,0x33,0x3e,0xd8,0xb8,0x73,0xb5,0x72,0x46,0x16,
0xb8,0x8d,0xad,0xaa,0xba,0x5f,0x28,0xbb,0x78,0x3a,0xeb,0x01,
0xf6,0x0d,0xf5,0xfd,0xb8,0xc0,0xa2,0x37,0x45,0x90,0x0f,0x46,
0x2f,0x40,0x5d,0xeb,0xfd,0x51,

```

```

};
const uint8_t tc_ISK_IR[] = {
    0x2b,0x2c,0x53,0x4c,0x35,0x2c,0x44,0x67,0x73,0xbd,0x33,0x4f,
    0xac,0x2f,0x2c,0x50,0xef,0x8c,0xd7,0x99,0x1b,0xd4,0xe0,0x70,
    0xf8,0x5b,0x03,0x67,0xa2,0xf7,0xff,0xca,0x44,0x50,0x66,0xcf,
    0x20,0xb7,0x56,0x77,0x36,0x87,0xe1,0x03,0x8b,0x17,0x08,0x96,
    0xec,0x26,0x77,0xfe,0x72,0x2a,0xcb,0x0f,0x9e,0x6c,0x2f,0x11,
    0x83,0x0e,0x80,0x8a,
};
const uint8_t tc_ISK_SY[] = {
    0x78,0xc4,0xdd,0x71,0x36,0x30,0x9a,0x2b,0xbe,0x1f,0xde,0xf3,
    0xcf,0x24,0xa0,0x86,0x90,0x00,0x6b,0x0c,0x9d,0xe2,0x53,0xb7,
    0x70,0xc1,0x47,0xdd,0x08,0x00,0x68,0x1c,0x82,0xe4,0xe6,0x7a,
    0x38,0x8e,0xd1,0xcd,0x91,0x82,0xe5,0x95,0xb8,0xe9,0xe3,0xf2,
    0x97,0x6a,0x0e,0x6d,0xab,0x48,0xb2,0xcd,0x20,0x5b,0x19,0x48,
    0x9e,0x20,0xf5,0x71,
};

```

[B.7.8.](#) Test case for scalar_mult_vfy with correct inputs

```

s: (length: 66 bytes)
0182dd7925f1753419e4bf83429763acd37d64000cd5a175edf53a15
87dd986bc95acc1506991702b6ba1a9ee2458fee8efc00198cf0088c
480965ef65ff2048b856
X: (length: 133 bytes)
0400bf0a2632f954515e65c55553e25cde4c8bf3a48e5df86a3ef845
fcf15c8d9a4640171188ff835df48b8f934070d225daa591e270a9cc
539b82e8dc145caf38aeb900c30b83a1c9792e95c4a25f75b58001d3
6331c2b71a86591e1b510a1740335bc9947da1f6bab91b86900c9258
b28ee7b5ea33af2a8138a75cde4287613ab6673bcc
G.scalar_mult(s,X) (full coordinates): (length: 133 bytes)
040100763e7ebe6a051e2195b1980686a2a5d7edbc1d9284e38d1e9e
13673b65b6b3b5cb1b1ab146a315c32425edee8fdca06a07cf72d26d
31e38ec6a38481b4f18d8600b2a7df9cc7db6cbf75b2eee98f9f14e5
e24a789d45b9709278e8b74b30eb32d55fb8cfea4258dcf9de7fb36a
67326584d5c8121c4802801115b908b937361c9828
G.scalar_mult_vfy(s,X) (only X-coordinate):
(length: 66 bytes)

```


0100763e7ebe6a051e2195b1980686a2a5d7edbc1d9284e38d1e9e13
673b65b6b3b5cb1b1ab146a315c32425edee8fdca06a07cf72d26d31
e38ec6a38481b4f18d86

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)
  0400bf0a2632f954515e65c55553e25cde4c8bf3a48e5df86a3ef845
  fcf15c8d9a4640171188ff835df48b8f934070d225daa591e270a9cc
  539b82e8dc145caf38aeb900c30b83a1c9792e95c4a25f75b58001d3
  6331c2b71a86591e1b510a1740335bc9947da1f6bab91b86900c9258
  b28ee7b5ea33af2a8138a75cde4287613ab6673b3a
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

Abdalla, et al.

Expires 18 July 2022

[Page 71]

Internet-Draft

CPace

January 2022

Bjoern Haase
Endress + Hauser Liquid Analysis - Gerlingen

Email: bjoern.m.haase@web.de

Julia Hesse
IBM Research Europe - Zurich

Email: JHS@zurich.ibm.com

