Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 10, 2020

B. Haase
Endress + Hauser Liquid Analysis

February 7, 2020

CPace, a balanced composable PAKE
draft-haase-cpace-01

Abstract

This document describes CPace 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.

Status of This Memo

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

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

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

This Internet-Draft will expire on August 10, 2020.

Table of Contents

1. Introduction

This document describes CPace which is a protocol for two parties that share a low-entropy secret (password) to derive a to derive a strong shared key without disclosing the secret to offline dictionary attacks. The CPace method was tailored for constrained devices and specifically considers efficiency and hardware side-channel attack mitigations at the protocol level. CPace is designed to be compatible with any group of both prime- and non-prime order and explicitly handles the complexity of cofactor clearing on the protcol level. CPace comes with a security proof providing composability guarantees. As a protocol, CPace is designed to be compatible with so-called "x-coordinate-only" Diffie-Hellman implementations on elliptic curve groups.

CPace is designed to be suitable as both, a building block within a larger protocol construction using CPace as substep, and as a standalone protocol.

It is considered, that for composed larger protocol constructions, the CPace subprotocol might be best executed in a separate cryptographic hardware, such as secure element chipsets. The CPace protocol design aims at considering the resulting constraints.

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

Let C be a group in which there exists a subgroup of prime order p where the computational simultaneous Diffie-Hellman (SDH) problem [VTBPEKE] is hard. C has order p*c where p is a large prime; c will be called the cofactor. Let I be the unit element in C, e.g., the point at infinity in if C is an elliptic curve group. We denote the operations in the group using addition and multiplication operators, e.g. P + (P + P) = P + 2 * P = 3 * P. We refer to a sequence of n additions of an element in P as scalar multiplication by n and use the notation scalar_multiply(P,n).

With F we denote a field that may be associated with C, e.g. the prime base field used for representing the coordinates of points on an elliptic curve.

We assume that for any element P in C there is a representation modulo negation, encode_group_element_mod_neg(P) as a byte string such that for any Q in C with Q != P and Q != -P, encode_group_element_mod_neg(P) != encode_group_element_mod_neg(Q). It is recommended that encodings of the elements P and -P share the

same result string. Common choices would be a fixed (per-group) length encoding of the x-coordinate of points on an elliptic curve C or its twist C' in Weierstrass form, e.g. according to [IEEE1363] in case of short Weierstrass form curves. For curves in Montgomery form correspondingly the u-coordinate would be encoded, as specified, e.g., by the encodeUCoordinate function from [RFC7748].

With J we denote the group modulo negation associated to C. Note that in J the scalar multiplication operation scalar_multiply is well defined since scalar_multiply(P,s) == -scalar_multiply(-P,s) while arbitrary additions of group elements are no longer available.

With J' be denote a second group modulo negation that might share the byte-string encoding function encode_group_element_mod_neg with J such for a given byte string either an element in J or J' is encoded. If the x-coordinate of an elliptic curve point group is used for the encoding, J' would commonly be corresponding to the group of points on the elliptic curve's quadratic twist. Correspondingly, with p' we denote the largest prime factor of the order of J' and its cofactor with c'.

Let scalar_cofactor_clearing(s) be a cofactor clearing function taking an integer input argument and returning an integer as result. For any s, scalar_cofactor_clearing(s) is REQUIRED to be of the form c * s1. I.e. it MUST return a multiple of the cofactor. An example of such a function may be the cofactor clearing and clamping functions decodeScalar25519 and decodeScalar448 as used in the X25519 and X448 protocols definitions of [RFC7748]. In case of prime-order groups with c == 1, it is RECOMMENDED to use the identity function with scalar_cofactor_clearing(s) = s.

Let scalar_mult_cc(P,s) be a joint "scalar multiplication and cofactor clearing" function of an integer s and an string-encoded value P, where P could represent an element either on J or J'. If P is an element in J or J', the scalar_mult_cc function returns a string encoding of an element in J or J' respectively, such that the result of scalar_mult_cc(P,s) encodes (scalar_cofactor_clearing(s) * P).

Let scalar_mult_ccv(P,s) be a "scalar multiplication cofactor clearing and verify" function of an integer s and an encoding of a

group element P. Unlike scalar_mult_cc, scalar_mult_ccv additionally carries out a verification that checks that the computational simultaneous Diffie-Hellman problem (SDH) is hard in the subgroup (in J or J') generated by the encoded element SP = scalar_mult_cc(P,s). In case that the verification fails (SP might be of low order or on the wrong curve), scalar_mult_ccv is REQUIRED to return the encoding of the identity element I. Otherwise scalar_mult_ccv(P,S) is REQUIRED to return the result of scalar_mult_cc(P,s). A common choice for scalar_mult_ccv for Montgomery curves with twist security would be the X25519 and X448 Diffie-Hellman functions as specified in [RFC7748]. For curves in short Weierstrass form, scalar_mult_ccv could be implemented by the combination of a point verification of the input point with a scalar multiplication. Here scalar_mult_ccv SHALL return the encoding of the neutral element I if the input point P was not on the curve C.

Let P=map_to_group_mod_neg(r) be a mapping operation that maps a string r to an encoding of an element P in J. Common choices would be the combination of map_to_base and map_to_curve methods as defined

in the hash2curve draft [HASH2CURVE]. Note that we don't require and RECOMMEND cofactor clearing here since this complexity is already included in the definition of the scalar multiplication operation calar_mult_cc above. Additionally requiring cofactor clearing also in map_to_group_mod_neg() would result in efficiency loss.

|| denotes concatenation of strings. We also let len(S) denote the length of a string in bytes. Finally, let nil represent an empty string, i.e., len(nil) = 0.

Let H(m) be a hash function from arbitrary strings m to bit strings of a fixed length. Common choices for H are SHA256 or SHA512 [RFC6234]. H is assumed to segment messages m into blocks m_i of byte length H_block. E.g. the blocks used in SHA512 have a size of 128 bytes.

Let strip_sign_information(P) be function that takes a string encoding of an element P in J and strips any information regarding the sign of P, such that strip_sign_information(P) = strip_sign_information(-P). For short Weierstrass (Montgomery) curves this function will return a string encoding the x-coordinate (u-coordinate). The purpose of defining this function is for

allowing for x-coordinate only scalar multiplication algorithms. The sign is to be stripped before generating the intermediate session key ISK.

With ISK we denote the intermediate session key output string provided by CPace that is generated by a hash operation on the Diffie-Hellman result. It is RECOMMENDED to apply ISK to a KDF function prior to using the key in a higher-level protocol.

KDF(Q) is a key-derivation function that takes an string and derives key of length L. A common choice for a KDF would be HMAC-SHA512.

With DSI we denote domain-separation identifier strings that may be prepended to the inputs of Hash and KDF functions.

Let IHF(salt, username, pw, sigma) be an iterated hash function that take a salt value, a user name and a password as input. IHF is designed to slow down brute-force attackers as controlled by a workload parameter set sigma. State of the art iterated hash functions are designed for requiring a large amount of memory for its operation and will be referred to as memory-hard hash functions (MHF). Scrypt [RFC7914] or Argon2 are common examples of a MHF primitive.

Let A and B be two parties. A and B may also have digital representations of the parties' identities such as Media Access

Control addresses or other names (hostnames, usernames, etc). We denote the parties' representation and the parties themselves both by using the identifiers A and B.

With CI we denote a string that SHALL be formed by the concatenation of the identifiers A and B and an an OPTIONAL associated data string. AD includes information which A and B might want to authenticate in the protocol execution. CI = A || B || AD; . One first example of CI data is an encoding of the concatenation of IP addresses and port numbers of both parties. AD might include a list of supported protocol versions if CPace were used in a higher-level protocol which negotiates use of a particular version. Including this list would ensure that both parties agree upon the same set of supported protocols and therefore prevent downgrade attacks.

We also assume that A and B share a common encoding of a password related string PRS. Typically PRS is a low-entropy secret such as a user-supplied password (pw) or a personal identification number. Note that CPace is NOT RECOMMENDED to be used in conjunction with user datbases that include more than one user account. CPace does not provide mechanisms for agreeing on user names, deriving salt values and agreeing on workload parameters, as required by the MHF functions that should be used for such databases. In such settings it is RECOMMENDED to use CPace as a subcomponent of the higher-level AuCPace protocol.

Let sid be a session id byte string chosen for each protocol session before protocol execution; The length len(sid) SHOULD be larger or equal to 16 bytes.

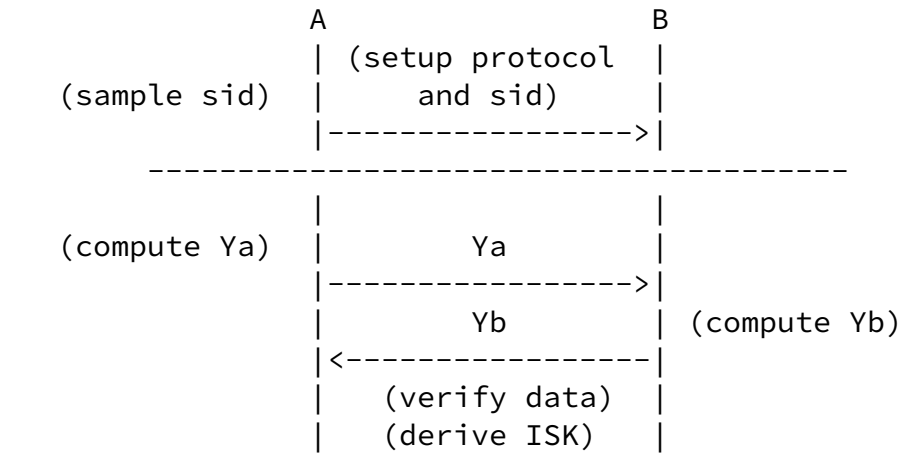
With ZPAD we denote a zero-padding string that is appended to PRS such that DSI||PRS has a length of at least H_block. CPace aims at mixing in entropy of PRS into the full internal state of the hash function before any adversary-known variable information (ADVI) enters the hashing algorithm. ADVI such as party identities or session IDs might be partially controlled by an adversary. Correlations of ADVI with the bare PRS string are considered to be easier exploitable by side-channel methods in comparison to a pre-hashed representation of PRS.

3.2. Protocol Flow

CPace is a one round protocol to establish an intermediate shared secret ISK with implicit mutual authentication. Prior to invocation, A and B are provisioned with public (CI) and secret information (PRS) as prerequisite for running the protocol. During the first round, A sends a public share Ya to B, and B responds with its own public share Yb. Both A and B then derive a shared secret ISK. ISK is

meant to be used for producing encryption and authentication keys by a KDF function outside of the scope of CPace. Prior to entering the protocol, A and B agree on a sid string. sid is typically pre-established by a higher-level protocol invocing CPace. If no such sid is available from a higher-level protocol, a suitable approach is to let A choose a fresh random sid string and send it to B together with Ya. This method is shown in the setup protocol section below.

This sample trace is shown below.



3.3. CPace

Both parties start with agreed values on the sid string, the channel identifier CI and the password-related string PRS.

The channel identifier, CI, SHOULD include an encoding of the communication channel used by both parties A and B, such as, e.g., IP and port numbers of both parties.

To begin, A calculates a generator G = map_to_group_mod_neg(DSI1 || PRS || ZPAD || sid || CI).

A picks ya randomly and uniformly according to the requirement of the group J and calculates Ya=scalar_mult_cc (G,ya). A then transmits Ya to B.

B picks yb randomly and uniformly. B then calculates G = map_to_group_mod_neg(DSI1 || PRS || ZPAD || sid || CI) and Yb = scalar_mult_cc(G,yb). B then calculates K = scalar_mult_ccv(Ya,yb). B MUST abort if K is the encoding of the neutral element I. Otherwise B sends Yb to A and proceeds as follows. B strips the sign information from K, Ya and Yb to obtain the strings Ks, Yas and Ybs by using the strip_sign_information() function. B returns ISK = H(DSI2 || sid || Ks || Yas || Ybs).

Upon reception of Yb, A calculates K = scalar_mult_ccv(Yb,ya). A

MUST abort if K is the neutral element I. If K is different from I, A strips the sign information from K, Ya and Yb and returns ISK = H(DSI2 || sid || Ks || Yas || Ybs).

K and Ks are shared values, though they MUST NOT be used as a shared secret key. Note that calculation of ISK from Ks includes the protocol transcript and prevents key malleability with respect to man-in-the-middle attacks from active adversaries.

Upon completion of this protocol, the session key ISK returned by A and B will be identical by both parties if and only if the supplied input parameters sid, PRS and CI match on both sides and the information on the public elements in J were not modified by an adversary.

4. Ciphersuites

This section documents CPACE ciphersuite configurations. A ciphersuite is REQUIRED to specify all of,

- a group modulo negation J with an associated encode_group_element_mod_neg function
- scalar_mult_cc(P,s) and scalar_mult_ccv(P,s) functions operating on encodings of elements P in J
- a mapping function map_to_group_mod_neg(r) converting byte strings r into elements in J
- a strip_sign_information(Q) function operating on string representations of elements Q
- a hash function H
- and domain separation strings DSI1, DSI2

Currently, detailed specifications are available for CPACE-X25519-ELLIGATOR2_SHA512-SHA512 and CPACE-P256-SSWU_SHA256-SHA256. These cipher suites are specifically designed for suitability also with constrained hardware. It is recommended that cipher suites for short Weierstrass curves are specified in line with the corresponding definitions for NIST-P256. Cipher suites for modern Montgomery or Edwards curves are recommended to be specified in line with the definitions for Curve25519.

J	map_to_group_mod_neg	KDF
X25519	ELLIGATOR2_SHA512	SHA512 [RFC6234]
NIST P-256	SSWU_SHA256 [HASH2CURVE]	SHA256 [RFC6234]

Table 1: CPace Ciphersuites

4.1. CPACE-X25519-ELLIGATOR2_SHA512-SHA512

This cipher suite targets particularly constrained targets and implements specific optimizations. It uses the group of points on the Montgomery curve Curve25519 for constructing J. The base field F is the prime field built upon the prime 2^255 - 19. The Diffie-Hellmann protocol X25519 and the group are specified in [RFC7748]. The encode_group_element_mod_neg(P) is implemented by the encodeUCoordinate(P) function defined in [RFC7748]. The neutral element I is encoded as a 32 byte zero-filled string.

The domain separation strings are defined as DSI1 = "CPace25519-1", DSI2 = "CPace25519-2" (twelve-byte ASCII encoding without ANSI-C style trailing zeros).

Both, scalar_mult_cc and scalar_mult_ccv, are implemented by the X25519 function specified in [RFC7748].

The secret scalars ya and yb used for X25519 shall be sampled as uniformly distributed 32 byte strings.

The map_to_group_mod_neg function is implemented as follows. First the byte length of the ZPAD zero-padding string is determined such that len(ZPAD) = max(0, H_block_SHA512 - len(DSI1 || PRS)), with H_block_SHA512 = 128 bytes. Then a byte string u is calculated by use of u = SHA512(DSI1||PRS||ZPAD||sid||CI). The resulting string is interpreted as 512-bit integer in little-endian format according to the definition of decodeLittleEndian() from [RFC7748]. The resulting integer is then reduced to the base field as input to the Elligator2 map specified in [HASH2CURVE] to yield the secret generator G = Elligator2(u).

CPace25519 returns a session key ISK of 64 bytes length by a single invocation of SHA512(DSI2||sid||K||Ya||Yb). Since the encoding does not incorporate the sign from the very beginning Qs = strip_sign_information(Q) == Q for this cipher suite.

The following sage code could be used as reference implementation for the mapping and key derivation functions.

<CODE BEGINS>

```
def littleEndianStringToInteger(k):
    bytes = [ord(b) for b in k]
    return sum((bytes[i] << (8 * i)) for i in range(len(bytes)))

def map_to_group_mod_neg_CPace25519(sid, PRS, CI):
    m = hashlib.sha512()
    p = 2^255 - 19

    H_block_SHA512 = 128
    DSI1 = b"CPace25519-1"
    ZPAD_len = max(0,H_block_SHA512 - len(CI) - len(PRS))
    ZPAD = ZPAD_len * "\0"

    m.update(DSI1)
    m.update(PRS)
    m.update(ZPAD)
    m.update(sid)
    m.update(CI)
    u = littleEndianStringToInteger(m.digest())
    return map_to_curve_elligator2_curve25519(u % p)

def generate_ISK_CPace25519(sid,K,Ya,Yb):
    m = hashlib.sha512(b"CPace25519-2")
    m.update(sid)
    m.update(K)
    m.update(Ya)
    m.update(Yb)
    return m.digest()
```

<CODE ENDS>

The definitions above aim at making the protocol suitable for outsourcing CPace to secure elements (SE) where nested hash function constructions such as defined in [RFC5869] have to be considered to

be particularly costly. As a result, the task of generating session keys by a strong KDF function is left out of the scope of the CPace protocol. This fact is expressed by the naming of the intermediate shared Key ISK. The definitions above regarding the mapping deviate from the definition in the encode_to_curve function from [HASH2CURVE] by significantly reducing the amount of hash invocations. Moreover, the CPace protocol specification, unlike the hash-to-curve draft specification also considers the risk of side-channel leakage during the hashing of PRS by introducing the ZPAD padding. Mitigating

attacks of an adversary that analyzes correlations between publicly known information with the low-entropy PRS strings was considered relevant in important settings. We also avoid the overhead of redundant cofactor clearing, by making the Diffie-Hellman protocol responsible for this task (and not the mapping algorithm). Due to its use in Ed25519 [RFC8032], SHA512 is considered to be the natural hash choice for Curve25519. The 512 bit output of SHA512 moreover allows for removing any statistical bias stemming from the non-canonical base field representations, such that the overhead of the HKDF_extract/HKDF_expand sequences from [HASH2CURVE] are considered not necessary (in line with the assessments regarding Curve25519 in [HASH2CURVE]).

4.2. CPACE-P256-SSWU_SHA256-SHA256

This cipher suite targets applications that do not as agressively focus on efficiency, bandwidth and code size as the Curve25519 implementation. Instead it aims at reusing existing encoding and curve standards wherever possible.

It uses the group of points on the NIST P-256 curve which is defined in short Weierstrass form for constructing J [RFC5480]. The base field F is the prime field built upon the Solinas prime p = 2^256-2^224+2^192+2^96-1. Encoding of full group elements requires both, x and y coordinates. In order to facilitate point validation and in order to be in line with recent TLS 1.3 requirements, implementations MUST encode both, x and y coordinates. It is RECOMMENDED to use the uncompressed format from [SEC1] using the 0x04 octet prefix. The strip_sign_information() function returns the substring from the SEC1 representation encoding the x-coordinate of the curve point.

NIST P-256 is of prime order and does not require cofactor clearing. The scalar_cofactor_clearing function is the identity function with scalar_cofactor_clearing(s) == s

The domain separation strings are defined as DSI1 = "CPace-P256-1", DSI2 = "CPace-P256-2".

For the scalar_mult_cc function operating on the internally generated points, a conventional scalar multiplication on P-256 is used, i.e. without the need of further verification checks. The scalar_mult_ccv function that operates on remotely generated points includes the mandatory verification as follows. First from the encoded point the x and y coordinates are decoded. These points are used for verifying the curve equation. If the point is not on the curve, scalar_mult_ccv returns the neutral element I. If the point is on

the curve, scalar_mult_ccv calls scalar_mult_cc and returns the result of the scalar multiplication.

For P-256, the map_to_group_mod_neg function is implemented as follows. The zero-padding string length is calculated as len(ZPAD) = max(0, H_block_SHA256 - len(DSI1 || PRS)) with H_block_SHA256 = 64. For the mapping to the curve, a 32 byte string U1 = SHA256(DSI1 || PRS || ZPAD || sid || CI) is calculated. From U1 a second 32 byte value is calculated as U2 = SHA256(U1). The concatenation of U1 and U2 is interpreted as a 512 bit integer u by use of the u = OS2IP(U1 || U2) function from [HASH2CURVE]. This value is reduced to a 32 byte representation of a field element fu = u % p. The coordinates (x,y) in F of the secret generator G are calculated as (x,y) = map_to_curve_simple_swu_3mod4(fu) function from [HASH2CURVE].

As hash function H SHA256 is chosen, returning a session key ISK of 32 bytes length with ISK=SHA256(DSI2 || sid || Ks || Yas || Ybs).

The following sage code could be used as reference implementation for the mapping and key derivation functions.

<CODE BEGINS>

```
def map_to_group_mod_neg_CPace_P256(sid, PRS, CI):
    m = hashlib.sha256()

    H_block_SHA256 = 64
    DSI1 = b"CPace-P256-1"
    ZPAD_len = max(0,H_block_SHA256 - len(CI) - len(PRS))
    ZPAD = ZPAD_len * "\0"

    m.update(DSI1)
    m.update(PRS)
    m.update(ZPAD)
    m.update(sid)
    m.update(CI)
    U1 = m.digest()
    U2 = hashlib.sha256(U1).digest()
    u = OS2I(U1 + U2)
```

```
    return map_to_curve_simple_swu_3mod4(u)

def generate_ISK_CPace_P256(sid,K,Ya,Yb):
    m = hashlib.sha256(b"CPace-P256-2")
    m.update(sid)
    m.update(strip_sign_information(K))
    m.update(strip_sign_information(Ya))
    m.update(strip_sign_information(Yb))
    return m.digest()
```

<CODE ENDS>

Similarly to the Curve25519 implementation, the definitions above aim at making the protocol suitable for outsourcing to secure elements where hash function invocations have to be considered to be particularly costly. As a result, the task of generating session keys by a strong KDF function is left out of the scope of the CPace protocol. The naming of ISK as intermediate shared key reflects this fact. Also the method for calculating the generator has been optimized for reducing the number of hash calculations in comparison to the suggestions [HASH2CURVE].

5. Security Considerations

A security proof of CPace is found in [cpace_paper].

Elements received from a peer MUST be checked by a proper implementation of the scalar_mult_ccv method. Failure to properly validate group elements can lead to attacks. The Curve25519-based cipher suite employs the twist security feature of the curve for

point validation. As such, it is mandatory to check that all low-order points on both the curve and the twist are mapped on the neutral element by the X25519 function. Corresponding test vectors are provided in the appendix.

The choices of random numbers MUST be uniform. Randomly generated values (e.g., ya and yb) MUST NOT be reused.

CPace is NOT RECOMMENDED to be used in conjunction with applications supporting different username/password pairs. In this case it is RECOMMENDED to use CPace as building block of the augmented AuCPace

protocol.

If CPace is used as a building block of higher-level protocols, it is RECOMMENDED that sid is generated by the higher-level protocol and passed to CPace. It is RECOMMENDED sid, is generated by sampling ephemeral random strings.

Since CPace is designed to be used as a building block in higher-level protocols and for compatibility with constrained hardware, it does not by itself include a strong KDF construction. CPace uses a simple hash operation for generating its intermediate key ISK. It is RECOMMENDED that the ISK is post-processed by a KDF according the needs of the higher-level protocol. In case that the CPace protocol is delegated to a secure element hardware, it is RECOMMENDED that the main processing unit applies a KDF to the externally generated ISK.

In case that side-channel attacks are to be considered practical for a given application, it is RECOMMENDED to focus side-channel protections such as masking and redundant execution (faults) on the process of calculating the secret generator G. The most critical aspect to consider is the processing of the first block of the hash that includes the PRS string. The CPace protocol construction considers the fact that side-channel protections of hash functions might be particularly resource hungry. For this reason, CPace aims at minimizing the number of hash functions invocations in the specified mapping method.

CPace is proven secure under the hardness of the computational Simultaneous Diffie-Hellmann (SDH) assumption in the group J (as defined in [VTBPEKE]). Still, even for the event that large-scale quantum computers (LSQC) will become available, CPace forces an active adversary to solve one CDH per password guess. Using the wording suggested by S. Tobtu on the CFRG mailing list, CPace is "quantum-annoying". For the event that LSQC become ubiquitous, it is suggested to consider the replacement of the group operations used in CPace with a corresponding commutative group actions on isogenies, such as suggested in [IsogenyPAKE]. CPace does not require arbitrary

group operations but only the operation set available in a group modulo negation. This is considered to be a significant advantage when using commutative isogeny-based group action cryptography as a replacement for elliptic-curve Diffie-Hellmann.

6. IANA Considerations

No IANA action is required.

7. Acknowledgments

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

Comments are specifically invited regarding the following aspect. The CPace mapping function design is based on the following assessments. 1.) Masked, hardware-side-channel-protected hash function implementations should be considered highly desirable for the calculation of the generators G if an implementation might be exposed to physical attacks. 2.) The complexity of such protected hash implementations (possibly with lots of boolean-arithmetic masking conversions) was assessed critical for constrained hardware. Hash operation complexity was also assessed to be critical for secure element chipsets that often were assessed to run hash operations in software without hardware accellerator support.

This assessment is not in line with the assumptions for the hash-to-curve-05 draft. As a consequence, this draft aimed at more aggressively reducing the number of nested hash function invocations in comparison to the suggestions of the hash-to-curve-05 draft.

8. References

8.1. Normative References

[HASH2CURVE]
Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R., and C. Wood, "draft-irtf-cfrg-hash-to-curve-05", 2019.

IRTF draft standard

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

IEEE 1363

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, DOI 10.17487/RFC5480, March 2009, <https://www.rfc-editor.org/info/rfc5480>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[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/info/rfc6234>.

[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/info/rfc7748>.

[RFC7914] Percival, C. and S. Josefsson, "The scrypt Password-Based Key Derivation Function", RFC 7914, DOI 10.17487/RFC7914, August 2016, <https://www.rfc-editor.org/info/rfc7914>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[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/info/rfc8174>.

[SEC1] SEC, "STANDARDS FOR EFFICIENT CRYPTOGRAPHY, "SEC 1: Elliptic Curve Cryptography", version 2.0", May 2009.

8.2. Informative References

[cpace_paper] Haase, B. and B. Labrique, "AuCPace. PAKE protocol tailored for the use in the internet of things.", Feb 2018.

[IsogenyPAKE]
Taraskin, O., Soukharev, V., Jao, D., and J. LeGrow, "An Isogeny-Based Password-Authenticated Key Establishment Protocol.", Sep. 2018.

eprint.iacr.org/2018/886

[VTBPEKE] Pointcheval, D. and G. Wang, "VTBPEKE: Verifier-based Two-Basis Password ExponentialKey Exchange", 2017.

Proceedings of the 2017 {ACM} on Asia Conference on Computer and Communications Security, AsiaCCS 2017

Appendix A. CPace25519 Test Vectors

The test vectors for CPace25519 consist of three blocks.

First test vectors for X25519 are provided which is used as combined scalar multiplication, cofactor clearing and verification function. Specifically, test vectors for the small order points are provided for checking that all small order points are mapped to the neutral element

Then test vectors for the Elligator2 primitive are provided.

Then test vectors for the encoding of the secret generator are provided combining the hash operation and the encoding of the generator.

Finally test vectors for a honest party protocol execution are provided, including derivation of the session key ISK.

A.1. X25519 test vectors

```
############################ /X25519 ################################
Test vectors for X25519 include three values:
- The scalar encoding prior to co-factor clearing and clamping, s
- The little-endian byte string encoding of the input point, u
- The expected little-endian byte string encoding of the result, r

Test vector for X25519 with a coordinate on J:
s: a546e36bf0527c9d3b16154b82465edd62144c0ac1fc5a18506a2244ba449ac4
u: e6db6867583030db3594c1a424b15f7c726624ec26b3353b10a903a6d0ab1c4c
r: c3da55379de9c6908e94ea4df28d084f32eccf03491c71f754b4075577a28552

Test vector for X25519 with a coordinate on the twist J':
s: 4b66e9d4d1b4673c5ad22691957d6af5c11b6421e0ea01d42ca4169e7918ba0d
u: e5210f12786811d3f4b7959d0538ae2c31dbe7106fc03c3efc4cd549c715a413
r: 95cbde9476e8907d7aade45cb4b873f88b595a68799fa152e6f8f7647aac7957

Test vectors for X25519 with coordinates on J and J' that MUST all
yield the neutral element (0) independent of s:

s: a546e36bf0527c9d3b16154b82465edd62144c0ac1fc5a18506a2244ba449ac4

u: 0000000000000000000000000000000000000000000000000000000000000000
u: 0100000000000000000000000000000000000000000000000000000000000000
u: e0eb7a7c3b41b8ae1656e3faf19fc46ada098deb9c32b1fd866205165f49b800
u: 5f9c95bca3508c24b1d0b1559c83ef5b04445cc4581c8e86d8224eddd09f1157
u: ecffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff7f
u: edffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff7f
u: eeffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff7f
u: cdeb7a7c3b41b8ae1656e3faf19fc46ada098deb9c32b1fd866205165f49b880
u: 4c9c95bca3508c24b1d0b1559c83ef5b04445cc4581c8e86d8224eddd09f11d7
u: d9ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
u: daffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
u: dbffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

```
r: 0000000000000000000000000000000000000000000000000000000000000000

############################ X25519/ ################################
```

A.2. Elligator2 test vectors

Two test vectors are provided


```
################### /Elligator 2 ###################################

Vector set 1 as little endian byte strings:
in: bc149a46d293b0aeea34581349d72f8a5a96cd531102d67379cd9bfadd4ec800
out:66b68f7575cd282403fc2bd323ff04601203c1ec5516ce247f7c0adbef05d367

Vector set 1 as base 10 numbers:
in:  353913733711106373587640212589159940893929660
     218606101422693758398583131871 6
out: 469610691099713701930355044506778951666876820
     601074157241352710439876254742118

#####################################################################

Vector set 2 as little endian byte strings:
in: 89cf55d4b5d3f84b1634957ac503a32b84ba11471a96b227bca70a0c3bf26375
out:1db163c86ceca7621903c9412d6dc71b4ed263b687eed092b194b5e540bba308

Vector set 2 as base 10 numbers:

in:  530971929581761349677698694411105058011053992421528586742712709
     85522606362505
out: 390779123641965710057372702362153599533842156764537839663973068
     2305857171741

################### Elligator 2/ ###################################
```

A.3. Test vectors for the secret generator G


```
   ###################### /Secret generator G ##########################
   Inputs:
   DSI1 = 'CPace25519-1'
   PRS  = 'password'
   sid  = SHA512('sid'), bytes 0 to 15
   A    = 'Ainitiator'
   B    = 'Bresponder'
   AD   = 'AD'
   ######################################################################
   Outputs and intermediate results:

   DSI1 = 435061636532353531392d31 string ('CPace25519-1') of len(12)
   PRS  = 70617373776f7264 ('password') string of len(8)
   ZPAD = 98 zero bytes (before mixing in variable data)
   sid  = 7e4b4791d6a8ef019b936c79fb7f2c57 string of len(16)
   CI   = 41696e69746961746f7242726573706f6e6465724144
          ('AinitiatorBresponderAD') string of len(22)

   u = SHA512(DSI1||PRS||ZPAD||sid||CI) as 512 bit little-endian int:
```

```
    (0xced4bf3254970eaec9f304ed422d8fde59e8c4abb0a27c675b4820a0c2c8fd92
     << 256)
   + 0x6dd2899f728ed1620e01e3d7fb9f5cd86e06ee4b5d552bde1524e0cb1e9344e0
   u as reduced base field element coordinate:
     0x2166eb1800faff5408149f0fce62b7d9c6941fc79573a335a1d9b8a80868ed26
   u encoded as little endian byte string:
       26ed6808a8b8d9a135a37395c71f94c6d9b762ce0f9f140854fffa0018eb6621

   Elligator2 output G as base field element coordinate:
     0x307760941be97d7c68b037cb9d22d69838b60e194c50ded8b85873f9e1395126
   Elligator2 output G encoded as little endian byte string:
       265139e1f97358b8d8de504c190eb63898d6229dcb37b0687c7de91b94607730

   ###################### Secret generator G/ ##########################
```

A.4. Test vectors for CPace DH


```
   ###################### /CPace Diffie-Hellman ########################
   Inputs:

   Elligator2 output G as base field element coordinate:
     0x307760941be97d7c68b037cb9d22d69838b60e194c50ded8b85873f9e1395126
   Elligator2 output G encoded as little endian byte string:
       265139e1f97358b8d8de504c190eb63898d6229dcb37b0687c7de91b94607730

   Secret scalar ya=SHA512('ya'), bytes 0...31, as integer:
     0xbfec93334144994275a3eba9eb0adf3fe40d54e400d105d59724bee398b722d1
   ya encoded as little endian byte string:
       d122b798e3be2497d505d100e4540de43fdf0aeba9eba375429944413393ecbf
```

```
Secret scalar yb=SHA512('yb'), bytes 0...31, as integer:
  0xb16a6ff3fcaf874cb59058493cb1f28b3e20084ad6d46fcd3c053284d60cecc0
yb encoded as little endian byte string:
    c0ec0cd68432053ccd6fd4d64a08203e8bf2b13c495890b54c87affcf36f6ab1

################################################################
Outputs:

Public point Ya as integer:
  0x79f9f2c1245fd8c4ab38bc75082f2daf6f47ca53fd5f0de7af72fee9c7ddd993
Ya encoded as little endian byte string:
    93d9ddc7e9fe72afe70d5ffd53ca476faf2d2f0875bc38abc4d85f24c1f2f979

Public point Yb as integer:
  0x18ac9063b4419695db48028d2eda7b2b2e649d22f56a5987eba9f05941de1c74
Yb encoded as little endian byte string:
    741cde4159f0a9eb87596af5229d642e2b7bda2e8d0248db959641b46390ac18

DH point K as integer:
  0x276896a227a09f389a04b9656099aa05ef8ec2b394cf32cc50cca9ae56334215
K encoded as little endian byte string:
    15423356aea9cc50cc32cf94b3c28eef05aa996065b9049a389fa027a2966827

###################### CPace Diffie-Hellman/ ########################
```

A.5. Test vectors for intermediate session key generation


```
#################### /Session Key derivation ######################
Inputs:

DSI2 = 435061636532353531392d32 string ('CPace25519-2') of len(12)
sid  = 7e4b4791d6a8ef019b936c79fb7f2c57 string of len(16)
```

```
   strings of length 32:
   K = 15423356aea9cc50cc32cf94b3c28eef05aa996065b9049a389fa027a2966827
   Ya= 93d9ddc7e9fe72afe70d5ffd53ca476faf2d2f0875bc38abc4d85f24c1f2f979
   Yb= 741cde4159f0a9eb87596af5229d642e2b7bda2e8d0248db959641b46390ac18

   ######################################################################

   string of length 64:
   ISK = SHA512(DSI2 || sid || K || Ya || Yb)
     = de0be1eeb7e6453d8c961353cd333694866f5432f24b0d4ed393cb6473e835df
       265ce72613effa3368a907031d897c733d300dfdb364ff66d270b404cdfbcb0a

   ##################### Session Key derivation/ ######################
```

Author's Address

Bjoern Haase
Endress + Hauser Liquid Analysis

Email: bjoern.m.haase@web.de