Network Working Group T. Taubert
Internet-Draft Apple Inc.
Intended status: Informational C.A. Wood
Expires: 13 June 2021 10 December 2020

SPAKE2+, an Augmented PAKE
draft-bar-cfrg-spake2plus-02

Abstract

This document describes SPAKE2+, a Password Authenticated Key Exchange (PAKE) protocol run between two parties for deriving a strong shared key with no risk of disclosing the password. SPAKE2+ is an augmented PAKE protocol, as only one party has knowledge of the password. This method is simple to implement, compatible with any prime order group and is computationally efficient.

Discussion Venues

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

Source for this draft and an issue tracker can be found at https://github.com/chris-wood/draft-bar-cfrg-spake2plus (https://github.com/chris-wood/draft-bar-cfrg-spake2plus).

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 13 June 2021.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction 2
2. Requirements Notation 3
3. Definition of SPAKE2+ 3
3.1. Offline Initialization 3
3.2. Protocol Flow 4
3.3. SPAKE2+ 5
4. Key Schedule and Key Confirmation 7
5. Ciphersuites 8
6. IANA Considerations 10
7. Security Considerations 10
8. Acknowledgements 11
9. References 11
9.1. Normative References 11
9.2. Informative References 12
Appendix A. Algorithm used for Point Generation 12
Appendix B. Test Vectors 14
Authors' Addresses 18

1. Introduction

This document describes SPAKE2+, a Password Authenticated Key Exchange (PAKE) protocol run between two parties for deriving a strong shared key with no risk of disclosing the password. SPAKE2+ is an augmented PAKE protocol, as only one party makes direct use of the password during the execution of the protocol. The other party only needs a verification value at the time of the protocol execution instead of the password. The verification value can be computed once, during an offline initialization phase. The party using the password directly would typically be a client, and acts as a prover, while the other party would be a server, and acts as verifier.

The protocol is augmented in the sense that it provides some resilience to the compromise or extraction of the verification value. The design of the protocol forces the adversary to recover the password from the verification value to successfully execute the protocol. Hence this protocol can be advantageously combined with a salted Password Hashing Function to increase the cost of the recovery

and slow down attacks. The verification value cannot be used directly to successfully run the protocol as a prover, making this protocol more robust than balanced PAKEs which don't benefit from Password Hashing Functions to the same extent.

This augmented property is especially valuable in scenarios where the execution of the protocol is constrained and the adversary can not query the salt of the password hash function ahead of the attack. Constraints may consist in being in physical proximity through a local network or when initiation of the protocol requires a first authentication factor.

This password-based key exchange protocol appears in [TDH] and is proven secure in [UCAnalysis]. It is compatible with any prime-order group and relies only on group operations, making it simple and computationally efficient. Predetermined parameters for a selection of commonly used groups are also provided.

This document has content split out from a related document specifying SPAKE2 [I-D.irtf-cfrg-spake2].

2. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Definition of SPAKE2+

3.1. Offline Initialization

Let G be a group in which the computational Diffie-Hellman (CDH) problem is hard. Suppose G has order p*h where p is a large prime; h will be called the cofactor. Let I be the unit element in G, e.g., the point at infinity if G is an elliptic curve group. We denote the operations in the group additively. We assume there is a representation of elements of G as byte strings: common choices would be SEC1 uncompressed or compressed [SEC1] for elliptic curve groups or big endian integers of a fixed (per-group) length for prime field DH. We fix two elements M and N in the prime-order subgroup of G as defined in the table in this document for common groups, as well as a generator P of the (large) prime-order subgroup of G. P is specified in the document defining the group, and so we do not repeat it here.

|| denotes concatenation of strings. We also let len(S) denote the length of a string in bytes, represented as an eight-byte little-endian number. Finally, let nil represent an empty string, i.e., len(nil) = 0.

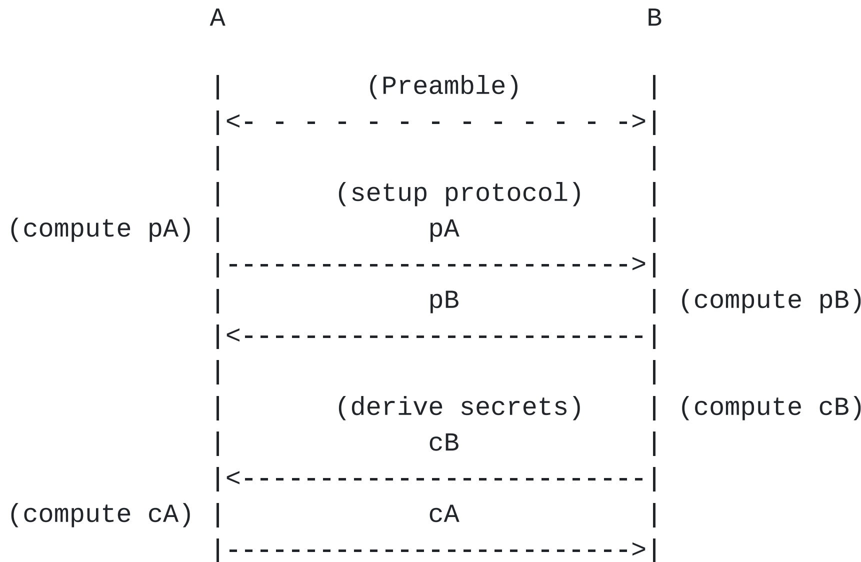
KDF is a key-derivation function that takes as input a salt, intermediate keying material (IKM), info string, and derived key length L to derive a cryptographic key of length L. MAC is a Message Authentication Code algorithm that takes a secret key and message as input to produce an output. Let Hash be a hash function from arbitrary strings to bit strings of a fixed length. Common choices for Hash are SHA256 or SHA512 [RFC6234]. Let PBKDF be a Password-Based Key Derivation Function designed to slow down brute-force attackers. Brute-force resistance may be obtained through various computation hardness parameters such as memory or CPU cycles, and are typically configurable. Scrypt [RFC7914] and Argon2 are common examples of PBKDF functions. PBKDF and hardness parameter selection are out of scope of this document. Section 5 specifies variants of KDF, MAC, and Hash suitable for use with the protocols contained herein.

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). A and B may share additional data (the context) separate from their identities which they may want to include in the protocol transcript. One example of additional data is a list of supported protocol versions if SPAKE2+ were used in a higher-level protocol which negotiates the use of a particular PAKE. Another example is the inclusion of PBKDF parameters and the application name. Including those would ensure that both parties agree upon the same set of supported protocols and use the same PBKDF parameters and therefore prevent downgrade and cross-protocol attacks. Specification of precise context values is out of scope for this document.

3.2. Protocol Flow

SPAKE2+ is a two round protocol that establishes a shared secret with an additional round for key confirmation. Prior to invocation, A and B are provisioned with information such as the input password needed to run the protocol. A preamble exchange may occur in order to communicate identities, protocol version and PBKDF parameters related to the verification value. Details of the preamble phase are out of scope of this document. During the first round, A, the prover, sends a public share pA to B, the verifier, and B responds with its own public share pB. Both A and B then derive a shared secret used to produce encryption and authentication keys. The latter are used during the second round for key confirmation. (Section 4 details the key derivation and confirmation steps.) In particular, B sends a key confirmation message cB to A, and A responds with its own key confirmation message cA. (Note that pB and cB MAY be sent in the same message.) Both parties MUST NOT consider the protocol complete prior to receipt and validation of these key confirmation messages.

A sample trace is shown below.



3.3. SPAKE2+

Let w0 and w1 be two integers derived by hashing the password pw with the identities of the two participants, A and B. Specifically, compute:

```
w0s || w1s = PBKDF(len(pw) || pw || len(A) || A || len(B) || B)
w0 = w0s mod p
w1 = w1s mod p
```

If both identities A and B are absent, i.e. len(A) = len(B) = 0, then the length prefixes are omitted as in Section 3.1.

```
w0s || w1s = PBKDF(pw)
```

If one or both identities A and B are unknown at the time of deriving w0 and w1, w0s and w1s are computed as if the unknown identities were absent. They however SHOULD be included in the transcript TT if the parties exchange those prior to or as part of the protocol flow.

The party B stores the verification value pair L and w0.

```
L = w1*P
```

Note that standards such as NIST.SP.800-56Ar3 suggest taking mod p of a hash value that is 64 bits longer than that needed to represent p to remove statistical bias introduced by the modulation. Protocols using this specification must define the method used to compute w0 and w1: it may be necessary to carry out various forms of

normalization of the password before hashing [RFC8265]. The hashing algorithm SHOULD be a PBKDF so as to slow down brute-force attackers.

When executing SPAKE2+, A selects x uniformly at random from the integers in [0, p), computes the public share pA=X, and transmits it to B.

```
x <- [0, p)
X = x*P + w0*M
```

Upon receipt of X, B computes h*X and aborts if the result is equal to I to ensure that X is in the large prime-order subgroup of G. B then selects y uniformly at random from the integers in [0, p), computes the public share pB=Y and transmits it to A. Upon receipt of Y, A computes h*Y and aborts if the result is equal to I.

```
y <- [0, p)
Y = y*P + w0*N
```

Parties A and B compute Z and V that are now shared as common values. Party A computes:

```
Z = h*x*(Y - w0*N)
V = h*w1*(Y - w0*N)
```

Party B computes:

```
Z = h*y*(X - w0*M)
V = h*y*L
```

All proofs of security hold even if the discrete log of the fixed group element N is known to the adversary. In particular, one MAY set N=I, i.e. set N to the unit element in G.

It is essential that both Z and V be used in combination with the transcript to derive the keying material. The protocol transcript encoding is shown below.

```
TT = len(Context) || Context ||
  || len(A) || A || len(B) || B
  || len(M) || M || len(N) || N
  || len(X) || X || len(Y) || Y
  || len(Z) || Z || len(V) || V
  || len(w0) || w0
```

Context is an application-specific customization string shared between both parties and MUST precede the remaining transcript. It might contain the name and version number of the higher-level

protocol, or simply the name and version number of the application. The context MAY include additional data such as the chosen ciphersuite and PBKDF parameters like the iteration count or salt. The context and its length prefix MAY be omitted.

If an identity is absent, its length is given as zero and the identity itself the empty octet string. If both A and B are absent, then both lengths are zero and both A and B will be empty octet strings. In applications where identities are not implicit, A and B SHOULD always be non-empty. Otherwise, the protocol risks Unknown Key Share attacks (discussion of Unknown Key Share attacks in a specific protocol is given in [I-D.ietf-mmusic-sdp-uks]).

Upon completion of this protocol, A and B compute shared secrets Ka, Ke, KcA, and KcB as specified in Section 4. B MUST send A a key confirmation message cB so both parties can confirm that they agree upon these shared secrets. After receipt and verification of B's confirmation message, A MUST send B a confirmation message. B MUST NOT send application data to A until it has received and verified the confirmation message. Key confirmation verification requires recomputation of cA or cB and checking for equality against that which was received.

4. Key Schedule and Key Confirmation

The protocol transcript TT, as defined in Section 3.3, is unique and secret to A and B. Both parties use TT to derive shared symmetric secrets Ke and Ka. The length of each key is equal to half of the digest output, e.g., |Ke| = |Ka| = 128 bits for SHA-256.

```
Ka || Ke = Hash(TT)
KcA || KcB = KDF(nil, Ka, "ConfirmationKeys")
```

A and B output Ke as the shared secret from the protocol. Ka and its derived KcA and KcB are not used for anything except key confirmation and MUST be discarded after the protocol execution.

Both endpoints MUST either exchange cA=KcA and cB=KcB directly, or employ a secure PRF, acting as a MAC that produces pseudorandom tags, for key confirmation. In the latter case, KcA and KcB are symmetric keys used to compute tags cA and cB over data shared between the participants. That data could for example be an encoding of the key shares exchanged earlier, or simply a fixed string.

```
cA = MAC(KcA, ...)
cB = MAC(KcB, ...)
```

Once key confirmation is complete, applications MAY use Ke as an authenticated shared secret as needed. For example, applications MAY derive one or more AEAD keys and nonces from Ke for subsequent application data encryption.

5. Ciphersuites

This section documents SPAKE2+ ciphersuite configurations. A ciphersuite indicates a group, cryptographic hash algorithm, and pair of KDF and MAC functions, e.g., SPAKE2+-P256-SHA256-HKDF-HMAC. This ciphersuite indicates a SPAKE2+ protocol instance over P-256 that uses SHA256 along with HKDF [RFC5869] and HMAC [RFC2104] for G, Hash, KDF, and MAC functions, respectively.

If no MAC algorithm is used in the key confirmation phase, its respective column in the table below can be ignored and the ciphersuite name will contain no MAC identifier.

G	Hash	KDF	MAC
P-256	SHA256 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]
P-256	SHA512 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]
P-384	SHA256 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]
P-384	SHA512 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]
P-521	SHA512 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]
edwards25519	SHA256 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]
edwards448	SHA512 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]
P-256	SHA256 [RFC6234]	HKDF [RFC5869]	CMAC-AES-128 [RFC4493]
P-256	SHA512 [RFC6234]	HKDF [RFC5869]	CMAC-AES-128 [RFC4493]

Table 1

The following points represent permissible point generation seeds for the groups listed in the Table above, using the algorithm presented in Appendix A. These bytestrings are compressed points as in [SEC1] for curves from [SEC1].

For P256: ~~~ M =
02886e2f97ace46e55ba9dd7242579f2993b64e16ef3dcab95afd497333d8fa12f
seed: 1.2.840.10045.3.1.7 point generation seed (M)

N =
03d8bbd6c639c62937b04d997f38c3770719c629d7014d49a24b4f98baa1292b49
seed: 1.2.840.10045.3.1.7 point generation seed (N) ~~~

For P384: ~~~ M =
030ff0895ae5ebf6187080a82d82b42e2765e3b2f8749c7e05eba366434b363d3dc

36f15314739074d2eb8613fceec2853 seed: 1.3.132.0.34 point generation
seed (M)

N =
02c72cf2e390853a1c1c4ad816a62fd15824f56078918f43f922ca21518f9c543bb
252c5490214cf9aa3f0baab4b665c10 seed: 1.3.132.0.34 point generation
seed (N) ~~~

For P521: ~~~ M =
02003f06f38131b2ba2600791e82488e8d20ab889af753a41806c5db18d37d85608
cfae06b82e4a72cd744c719193562a653ea1f119eef9356907edc9b56979962d7aa
seed: 1.3.132.0.35 point generation seed (M)

N =
0200c7924b9ec017f3094562894336a53c50167ba8c5963876880542bc669e494b25
32d76c5b53dfb349fdf69154b9e0048c58a42e8ed04cef052a3bc349d95575cd25
seed: 1.3.132.0.35 point generation seed (N) ~~~

For edwards25519: ~~~ M =
d048032c6ea0b6d697ddc2e86bda85a33adac920f1bf18e1b0c6d166a5cecdaf
seed: edwards25519 point generation seed (M)

N = d3bfb518f44f3430f29d0c92af503865a1ed3281dc69b35dd868ba85f886c4ab
seed: edwards25519 point generation seed (N) ~~~

For edwards448: ~~~ M =
b6221038a775ecd007a4e4dde39fd76ae91d3cf0cc92be8f0c2fa6d6b66f9a12
942f5a92646109152292464f3e63d354701c7848d9fc3b8880 seed: edwards448
point generation seed (M)

N = 6034c65b66e4cd7a49b0edec3e3c9ccc4588afd8cf324e29f0a84a072531c4db
f97ff9af195ed714a689251f08f8e06e2d1f24a0ffc0146600 seed: edwards448
point generation seed (N) ~~~

6. IANA Considerations

No IANA action is required.

7. Security Considerations

SPAKE2+ appears in [TDH] and is proven secure in [UCAnalysis].

Beyond the cofactor multiplication checks to ensure that elements
received from a peer are in the prime order subgroup of G, they also
MUST be checked for group membership as failure to properly validate
group elements can lead to attacks.

The choices of random numbers MUST BE uniform. Randomly generated values (e.g., x and y) MUST NOT be reused; such reuse may permit dictionary attacks on the password.

8. Acknowledgements

Thanks to Ben Kaduk and Watson Ladd, from which this specification originally emanated.

9. References

9.1. Normative References

[I-D.irtf-cfrg-spake2]
Ladd, W. and B. Kaduk, "SPAKE2, a PAKE", Work in Progress, Internet-Draft, draft-irtf-cfrg-spake2-15, 23 November 2020, <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-spake2-15.txt>.

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

[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>.

[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/info/rfc4493>.

[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>.

[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>.

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

[SEC1] "Elliptic Curve Cryptography, Standards for Efficient Cryptography Group, ver. 2", 2009, <https://secg.org/sec1-v2.pdf>.

[TDH] "The Twin-Diffie Hellman Problem and Applications", EUROCRYPT 2008, Volume 4965 of Lecture notes in Computer Science, pages 127-145, Springer-Verlag, Berlin, Germany , 2008.

[UCAnalysis] "Security analysis of SPAKE2+", 2020, <https://eprint.iacr.org/2020/313.pdf>.

9.2. Informative References

[I-D.ietf-mmusic-sdp-uks] Thomson, M. and E. Rescorla, "Unknown Key Share Attacks on uses of TLS with the Session Description Protocol (SDP)", Work in Progress, Internet-Draft, draft-ietf-mmusic-sdp-uks-07, 9 August 2019, <http://www.ietf.org/internet-drafts/draft-ietf-mmusic-sdp-uks-07.txt>.

Appendix A. Algorithm used for Point Generation

This section describes the algorithm that was used to generate the points (M) and (N) in the table in Section 5.

For each curve in the table below, we construct a string using the curve OID from [RFC5480] (as an ASCII string) or its name, combined with the needed constant, for instance "1.3.132.0.35 point generation seed (M)" for P-512. This string is turned into a series of blocks by hashing with SHA256, and hashing that output again to generate the

next 32 bytes, and so on. This pattern is repeated for each group and value, with the string modified appropriately.

A byte string of length equal to that of an encoded group element is constructed by concatenating as many blocks as are required, starting from the first block, and truncating to the desired length. The byte string is then formatted as required for the group. In the case of Weierstrass curves, we take the desired length as the length for representing a compressed point (section 2.3.4 of [SEC1]), and use the low-order bit of the first byte as the sign bit. In order to obtain the correct format, the value of the first byte is set to 0x02 or 0x03 (clearing the first six bits and setting the seventh bit), leaving the sign bit as it was in the byte string constructed by concatenating hash blocks. For the [RFC8032] curves a different procedure is used. For edwards448 the 57-byte input has the least-significant 7 bits of the last byte set to zero, and for edwards25519 the 32-byte input is not modified. For both the [RFC8032] curves the (modified) input is then interpreted as the representation of the group element. If this interpretation yields a valid group element with the correct order (p), the (modified) byte string is the output. Otherwise, the initial hash block is discarded and a new byte string constructed from the remaining hash blocks. The procedure of constructing a byte string of the appropriate length, formatting it as required for the curve, and checking if it is a valid point of the correct order, is repeated until a valid element is found.

The following python snippet generates the above points, assuming an elliptic curve implementation following the interface of Edwards25519Point.stdbase() and Edwards448Point.stdbase() in Appendix A of [RFC8032]:

```
def iterated_hash(seed, n):
    h = seed
    for i in range(n):
        h = hashlib.sha256(h).digest()
    return h

def bighash(seed, start, sz):
    n = -(-sz // 32)
    hashes = [iterated_hash(seed, i) for i in range(start, start + n)]
    return b''.join(hashes)[:sz]

def canon_pointstr(ecname, s):
    if ecname == 'edwards25519':
        return s
    elif ecname == 'edwards448':
        return s[:-1] + bytes([s[-1] & 0x80])
    else:
        return bytes([(s[0] & 1) | 2]) + s[1:]

def gen_point(seed, ecname, ec):
    for i in range(1, 1000):
        hval = bighash(seed, i, len(ec.encode()))
        pointstr = canon_pointstr(ecname, hval)
        try:
            p = ec.decode(pointstr)
            if p != ec.zero_elem() and p * p.l() == ec.zero_elem():
                return pointstr, i
        except Exception:
            pass
```

Appendix B. Test Vectors

This section contains test vectors for SPAKE2+ using the P256-SHA256-HKDF-HMAC and P256-SHA256-HKDF-CMAC ciphersuites. (Choice of PBKDF is omitted and values for w and w0,w1 are provided directly.) All points are encoded using the uncompressed format, i.e., with a 0x04 octet prefix, specified in [SEC1] A and B identity strings are provided in the protocol invocation.

```
[Context=b'SPAKE2+-P256-SHA256-HKDF draft-01']
[A=b'client']
[B=b'server']
w0 = 0xe6887cf9bdfb7579c69bf47928a84514b5e355ac034863f7ffaf4390e67d7
98c
w1 = 0x24b5ae4abda868ec9336ffc3b78ee31c5755bef1759227ef5372ca139b94e
512
L = 0x0495645cfb74df6e58f9748bb83a86620bab7c82e107f57d6870da8cbcb2ff
9f7063a14b6402c62f99afcb9706a4d1a143273259fe76f1c605a3639745a92154b9
```

```
x = 0x8b0f3f383905cf3a3bb955ef8fb62e24849dd349a05ca79aafb18041d30cbd
b6
X = 0x04af09987a593d3bac8694b123839422c3cc87e37d6b41c1d630f000dd6498
0e537ae704bcede04ea3bec9b7475b32fa2ca3b684be14d11645e38ea6609eb39e7e
y = 0x2e0895b0e763d6d5a9564433e64ac3cac74ff897f6c3445247ba1bab40082a
91
Y = 0x04417592620aebf9fd203616bbb9f121b730c258b286f890c5f19fea833a9c
900cbe9057bc549a3e19975be9927f0e7614f08d1f0a108eede5fd7eb5624584a4f4
Z = 0x0471a35282d2026f36bf3ceb38fcf87e3112a4452f46e9f7b47fd769cfb570
145b62589c76b7aa1eb6080a832e5332c36898426912e29c40ef9e9c742eee82bf30
V = 0x046718981bf15bc4db538fc1f1c1d058cb0eececf1dbe1b1ea08a4e25275d3
82e82b348c8131d8ed669d169c2e03a858db7cf6ca2853a4071251a39fbe8cfc39bc
TT = 0x210000000000000005350414b45322b2d503235362d5348413235362d484b4
4462064726166742d30310600000000000000636c69656e7406000000000000000736
57276657241000000000000000004886e2f97ace46e55ba9dd7242579f2993b64e16ef
3dcab95afd497333d8fa12f5ff355163e43ce224e0b0e65ff02ac8e5c7be09419c78
5e0ca547d55a12e2d20410000000000000004d8bbd6c639c62937b04d997f38c3770
719c629d7014d49a24b4f98baa1292b4907d60aa6bfade45008a636337f5168c64d9
bd36034808cd564490b1e656edbe7410000000000000004af09987a593d3bac8694b
123839422c3cc87e37d6b41c1d630f000dd64980e537ae704bcede04ea3bec9b7475
b32fa2ca3b684be14d11645e38ea6609eb39e7e410000000000000004417592620ae
bf9fd203616bbb9f121b730c258b286f890c5f19fea833a9c900cbe9057bc549a3e1
9975be9927f0e7614f08d1f0a108eede5fd7eb5624584a4f44100000000000000047
1a35282d2026f36bf3ceb38fcf87e3112a4452f46e9f7b47fd769cfb570145b62589
c76b7aa1eb6080a832e5332c36898426912e29c40ef9e9c742eee82bf30410000000
0000000046718981bf15bc4db538fc1f1c1d058cb0eececf1dbe1b1ea08a4e25275d
382e82b348c8131d8ed669d169c2e03a858db7cf6ca2853a4071251a39fbe8cfc39b
c2000000000000000e6887cf9bdfb7579c69bf47928a84514b5e355ac034863f7ffa
f4390e67d798c
Ka = 0xf9cab9adcc0ed8e5a4db11a8505914b2
Ke = 0x801db297654816eb4f02868129b9dc89
KcA = 0x0d248d7d19234f1486b2efba5179c52d
KcB = 0x556291df26d705a2caedd6474dd0079b
HMAC(KcA, Y) = 0xd4376f2da9c72226dd151b77c2919071155fc22a2068d90b5fa
a6c78c11e77dd
HMAC(KcB, X) = 0x0660a680663e8c5695956fb22dff298b1d07a526cf3cc591adf
ecd1f6ef6e02e
CMAC(KcA, Y) = 0xad04419077d806572fd7c8ab6d78656a
CMAC(KcB, X) = 0xaa076038a84938018a276e673ee7583e

[Context=b'SPAKE2+-P256-SHA256-HKDF draft-01']
[A=b'client']
[B=b'']
w0 = 0xe6887cf9bdfb7579c69bf47928a84514b5e355ac034863f7ffaf4390e67d7
98c
w1 = 0x24b5ae4abda868ec9336ffc3b78ee31c5755bef1759227ef5372ca139b94e
512
L = 0x0495645cfb74df6e58f9748bb83a86620bab7c82e107f57d6870da8cbcb2ff
```

```
9f7063a14b6402c62f99afcb9706a4d1a143273259fe76f1c605a3639745a92154b9
x = 0xec82d9258337f61239c9cd68e8e532a3a6b83d12d2b1ca5d543f44def17dfb
8d
X = 0x04230779960824076d3666a7418e4d433e2fa15b06176eabdd572f43a32ecc
79a192b243d2624310a7356273b86e5fd9bd627d3ade762baeff1a320d4ad7a4e47f
y = 0xeac3f7de4b198d5fe25c443c0cd4963807add767815dd02a6f0133b4bc2c9e
b0
Y = 0x044558642e71b616b248c9583bd6d7aa1b3952c6df6a9f7492a06035ca5d92
522d84443de7aa20a59380fa4de6b7438d925dbfb7f1cfe60d79acf961ee33988c7d
Z = 0x04b4e8770f19f58ddf83f9220c3a9305792665e0c60989e6ee9d7fa449c775
d6395f6f25f307e3903ac045a013fbb5a676e872a6abfcf4d7bb5aac69efd6140eed
V = 0x04141db83bc7d96f41b636622e7a5c552ad83211ff55319ac25ed0a09f0818
bd942e8150319bfbfa686183806dc61911183f6a0f5956156023d96e0f93d275bf50
TT = 0x21000000000000005350414b45322b2d503235362d5348413235362d484b4
4462064726166742d3031060000000000000000636c69656e74000000000000000410
000000000000004886e2f97ace46e55ba9dd7242579f2993b64e16ef3dcab95afd49
7333d8fa12f5ff355163e43ce224e0b0e65ff02ac8e5c7be09419c785e0ca547d55a
12e2d20410000000000000004d8bbd6c639c62937b04d997f38c3770719c629d7014
d49a24b4f98baa1292b4907d60aa6bfade45008a636337f5168c64d9bd36034808cd
564490b1e656edbe7410000000000000004230779960824076d3666a7418e4d433e2
fa15b06176eabdd572f43a32ecc79a192b243d2624310a7356273b86e5fd9bd627d3
ade762baeff1a320d4ad7a4e47f410000000000000044558642e71b616b248c9583
bd6d7aa1b3952c6df6a9f7492a06035ca5d92522d84443de7aa20a59380fa4de6b74
38d925dbfb7f1cfe60d79acf961ee33988c7d41000000000000000004b4e8770f19f58
ddf83f9220c3a9305792665e0c60989e6ee9d7fa449c775d6395f6f25f307e3903ac
045a013fbb5a676e872a6abfcf4d7bb5aac69efd6140eed4100000000000000004141
db83bc7d96f41b636622e7a5c552ad83211ff55319ac25ed0a09f0818bd942e81503
19bfbfa686183806dc61911183f6a0f5956156023d96e0f93d275bf5020000000000
00000e6887cf9bdfb7579c69bf47928a84514b5e355ac034863f7ffaf4390e67d798
c
Ka = 0xe2cbee3ae19a4dbe9f146be6bee9bfa1
Ke = 0x6989d8f9177ef7df67da437987f07255
KcA = 0x2f9e0bb669d2c22645bce34da04ac16a
KcB = 0xeb7a35168759dd8e9ce44e4dc51277ce
HMAC(KcA, Y) = 0xe1b9258807ba4750dae1d7f3c3c294f13dc4fa60cde346d5de7
d200e2f8fd3fc
HMAC(KcB, X) = 0xb9c39dfa49c47757de778d9bedeaca2448b905be19a43b94ee2
4b770208135e3
CMAC(KcA, Y) = 0xf545e7af21e334de7389ddcf2174e822
CMAC(KcB, X) = 0x3fb3055e16b619fd3de0e1b2bd7a9383

[Context=b'SPAKE2+-P256-SHA256-HKDF draft-01']
[A=b'']
[B=b'server']
w0 = 0xe6887cf9bdfb7579c69bf47928a84514b5e355ac034863f7ffaf4390e67d7
98c
w1 = 0x24b5ae4abda868ec9336ffc3b78ee31c5755bef1759227ef5372ca139b94e
512
```

```
L = 0x0495645cfb74df6e58f9748bb83a86620bab7c82e107f57d6870da8cbcb2ff
9f7063a14b6402c62f99afcb9706a4d1a143273259fe76f1c605a3639745a92154b9
x = 0xba0f0f5b78ef23fd07868e46aeca63b51fda519a3420501acbe23d53c29187
48
X = 0x04c14d28f4370fea20745106cea58bcfb60f2949fa4e131b9aff5ea13fd5aa
79d507ae1d229e447e000f15eb78a9a32c2b88652e3411642043c1b2b7992cf2d4de
y = 0x39397fbe6db47e9fbd1a263d79f5d0aaa44df26ce755f78e092644b434533a
42
Y = 0x04d1bee3120fd87e86fe189cb952dc688823080e62524dd2c08dffe3d22a0a
8986aa64c9fe0191033cafbc9bcaefc8e2ba8ba860cd127af9efdd7f1c3a41920fe8
Z = 0x04aac71cf4c8df8181b867c9ecbee9d0963caf51f1534a823429c26fe52483
13ffc5c5e44ea8162161ab6b3d73b87704a45889bf6343d96fa96cd1641efa71607c
V = 0x04c7c9505365f7ce57293c92a37f1bbdc68e0322901e61edef59fee7876b17
b063e0fa4a126eae0a671b37f1464cf1ccad591c33ae944e3b1f318d76e36fea9966
TT = 0x21000000000000005350414b45322b2d503235362d5348413235362d484b4
4462064726166742d30310000000000000000060000000000000073657276657241 0
000000000000004886e2f97ace46e55ba9dd7242579f2993b64e16ef3dcab95afd49
7333d8fa12f5ff355163e43ce224e0b0e65ff02ac8e5c7be09419c785e0ca547d55a
12e2d20410000000000000004d8bbd6c639c62937b04d997f38c3770719c629d7014
d49a24b4f98baa1292b4907d60aa6bfade45008a636337f5168c64d9bd36034808cd
564490b1e656edbe7410000000000000004c14d28f4370fea20745106cea58bcfb60
f2949fa4e131b9aff5ea13fd5aa79d507ae1d229e447e000f15eb78a9a32c2b88652
e3411642043c1b2b7992cf2d4de410000000000000004d1bee3120fd87e86fe189cb
952dc688823080e62524dd2c08dffe3d22a0a8986aa64c9fe0191033cafbc9bcaefc
8e2ba8ba860cd127af9efdd7f1c3a41920fe8410000000000000004aac71cf4c8df8
181b867c9ecbee9d0963caf51f1534a823429c26fe5248313ffc5c5e44ea8162161a
b6b3d73b87704a45889bf6343d96fa96cd1641efa71607c410000000000000004c7c
9505365f7ce57293c92a37f1bbdc68e0322901e61edef59fee7876b17b063e0fa4a1
26eae0a671b37f1464cf1ccad591c33ae944e3b1f318d76e36fea996620000000000
00000e6887cf9bdfb7579c69bf47928a84514b5e355ac034863f7ffaf4390e67d798
c
Ka = 0xec8d19b807ffb1d1eea81a93ba35cdfe
Ke = 0x2ea40e4badfa5452b5744dc5983e99ba
KcA = 0x66de534d9bf1e44e96a53a4b48d6b353
KcB = 0x4945c38bb476cb0f347f3222be9b64a2
HMAC(KcA, Y) = 0xe564c93b3015efb946dc16d642bbe7d1c8da5be164ed9fc3bae
4e0ff86e1bd3c
HMAC(KcB, X) = 0x072a94d9a54edc201d8891534c2317cadf3ea3792827f479e87
3f93e90f21552
CMAC(KcA, Y) = 0x94aacd28128dc2ce1d7f5684119d553c
CMAC(KcB, X) = 0xbc6615eb68af10d329b2acb2d4545d97

[Context=b'SPAKE2+-P256-SHA256-HKDF draft-01']
[A=b'']
[B=b'']
w0 = 0xe6887cf9bdfb7579c69bf47928a84514b5e355ac034863f7ffaf4390e67d7
98c
w1 = 0x24b5ae4abda868ec9336ffc3b78ee31c5755bef1759227ef5372ca139b94e
```

```
512
L = 0x0495645cfb74df6e58f9748bb83a86620bab7c82e107f57d6870da8cbcb2ff
9f7063a14b6402c62f99afcb9706a4d1a143273259fe76f1c605a3639745a92154b9
x = 0x5b478619804f4938d361fbba3a20648725222f0a54cc4c876139efe7d9a217
86
X = 0x04a6db23d001723fb01fcfc9d08746c3c2a0a3feff8635d29cad2853e73586
23425cf39712e928054561ba71e2dc11f300f1760e71eb177021a8f85e78689071cd
y = 0x766770dad8c8eecba936823c0aed044b8c3c4f7655e8beec44a15dcbcaf78e
5e
Y = 0x04390d29bf185c3abf99f150ae7c13388c82b6be0c07b1b8d90d26853e8437
4bbdc82becdb978ca3792f472424106a2578012752c11938fcf60a41df75ff7cf947
Z = 0x040a150d9a62f514c9a1fedd782a0240a342721046cefb1111c3adb3be893c
e9fcd2ffa137922fcf8a588d0f76ba9c55c85da2af3f1c789ca17976810387fb1d7e
V = 0x04f8e247cc263a1846272f5a3b61b68aa60a5a2665d10cd22c89cd6bad05dc
0e5e650f21ff017186cc92651a4cd7e66ce88f529299f340ea80fb90a9bad094e1a6
TT = 0x2100000000000000005350414b45322b2d503235362d5348413235362d484b4
4462064726166742d303100000000000000000000000000000000041000000000000
004886e2f97ace46e55ba9dd7242579f2993b64e16ef3dcab95afd497333d8fa12f5
ff355163e43ce224e0b0e65ff02ac8e5c7be09419c785e0ca547d55a12e2d2041000
0000000000004d8bbd6c639c62937b04d997f38c3770719c629d7014d49a24b4f98b
aa1292b4907d60aa6bfade45008a636337f5168c64d9bd36034808cd564490b1e656
edbe7410000000000000004a6db23d001723fb01fcfc9d08746c3c2a0a3feff8635d
29cad2853e7358623425cf39712e928054561ba71e2dc11f300f1760e71eb177021a
8f85e78689071cd41000000000000000004390d29bf185c3abf99f150ae7c13388c82b
6be0c07b1b8d90d26853e84374bbdc82becdb978ca3792f472424106a2578012752c
11938fcf60a41df75ff7cf947410000000000000000040a150d9a62f514c9a1fedd782
a0240a342721046cefb1111c3adb3be893ce9fcd2ffa137922fcf8a588d0f76ba9c5
5c85da2af3f1c789ca17976810387fb1d7e4100000000000000004f8e247cc263a184
6272f5a3b61b68aa60a5a2665d10cd22c89cd6bad05dc0e5e650f21ff017186cc926
51a4cd7e66ce88f529299f340ea80fb90a9bad094e1a6200000000000000000e6887cf
9bdfb7579c69bf47928a84514b5e355ac034863f7ffaf4390e67d798c
Ka = 0x5929a3ce9822c81401bf0f764f69af08
Ke = 0xea3276d68334576097e04b19ee5a3a8b
KcA = 0x7f84b939d600117256b0c8a6d40cf181
KcB = 0xf7d7547ced93f681e8df4c258c4516fd
HMAC(KcA, Y) = 0x71d9412779b6c45a2c615c9df3f1fd93dc0aaf63104da8ece4a
a1b5a3a415fea
HMAC(KcB, X) = 0x095dc0400355cc233fde7437811815b3c1524aae80fd4e6810c
f531cf11d20e3
CMAC(KcA, Y) = 0xd66386ee8033bf56387db3543691064e
CMAC(KcB, X) = 0x391070acb88ecc74dfe079cd0b8b52dc
```

Authors' Addresses

Tim Taubert
Apple Inc.
One Apple Park Way
Cupertino, California 95014,

United States of America

Email: ttaubert@apple.com

Christopher A. Wood

Email: caw@heapingbits.net