Network Working Group T. Taubert
Internet-Draft C. Wood
Intended status: Informational Apple, Inc.
Expires: September 10, 2020 March 9, 2020

# SPAKE2+, an Augmented PAKE
## draft-bar-cfrg-spake2plus-00

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.

Status of This Memo



Copyright Notice

Table of Contents



## 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 successful 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 extend.

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 is compatible with any group. It only relies on group operations making it simple and computationally efficient. It also has a security proof. Predetermined parameters for a selection of commonly used groups are also provided.

This document has content split out from a related document specifying SPAKE2 [SPAKE2].

## 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 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 parameters selection for the PBKDF 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 Authenticated Data (AAD) of length at most 2^16 - 1 bits that is separate from their identities which they may want to include in the protocol execution. One example of AAD is a list of supported protocol versions if SPAKE2+ were used in a higher-level protocol which negotiates the use of a particular PAKE. Including this list would ensure that both parties agree upon the same set of supported protocols and therefore prevent downgrade attacks.

### 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 is 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.

This sample trace is shown below.

```
                    A                           B

                    |          (Preamble)        |
                    |<- - - - - - - - - - - - - ->|
                    |                            |
                    |       (setup protocol)     |
     (compute pA)   |             pA             |
                    |--------------------------->|
                    |             pB             | (compute pB)
                    |<---------------------------|
                    |                            |
                    |       (derive secrets)     | (compute cB)
                    |             cB             |
                    |<---------------------------|
     (compute cA)   |             cA             |
                    |--------------------------->|
```

### 3.3. SPAKE2+

This protocol appears in [TDH]. Let w0 and w1 be two integers derived by hashing the password pw with the identities of the two participants, A and B. Specifically, w0s || w1s = PBKDF(len(pw) || pw || len(A) || A || len(B) || B), and then computing w0 = w0s mod p and w1 = w1s mod p. If both identities A and B are absent, then w0s || w1s = PBKDF(pw), i.e., the length prefix is omitted as in Section 3.1. The party B stores the verification value pair L=w1*P and w0.

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 numbers in the range [0, p), and lets X=x*P+w0*M, then transmits pA=X to B. Upon receipt of X, A computes h*X and aborts if the result is equal to I. B then selects y uniformly at random from the numbers in [0, p), then computes Y=y*P+w0*N, and transmits pB=Y to A.

A computes Z as h*x*(Y-w0*N), and V as h*w1*(Y-w0*N). B computes Z as h*y*(X- w0*M) and V as h*y*L. Both share Z and V as common values. 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(A) || A || len(B) || B || len(X) || X
  || len(Y) || Y || len(Z) || Z || len(V) || V
  || len(w0) || w0
```

If an identity is absent, it is omitted from the transcript entirely. For example, if both A and B are absent, then TT = len(X) || X || len(Y) || Y || len(Z) || Z || len(w0) || w0. Likewise, if only A is absent, TT = len(B) || B || len(X) || X || len(Y) || Y || len(Z) || Z || len(w0) || w0. This must only be done for applications in which identities are implicit. 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 Ke, KcA, and KcB as specified in Section 4. B MUST send A a key confirmation message Fb so both parties agree upon these shared secrets. This confirmation message Fb is computed as a MAC over the received share (pA) using KcB. Specifically, B computes Fb = MAC(KcB, pA). After receipt and verification of B's confirmation message, A MUST send B a confirmation message using a MAC computed equivalently except with the use of pB and KcA. Key confirmation verification requires computing F and checking for equality against that which was received.

## 4. Key Schedule and Key Confirmation

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

Both endpoints use Ka to derive subsequent MAC keys for key confirmation messages. Specifically, let KcA and KcB be the MAC keys used by A and B, respectively. A and B compute them as KcA || KcB = KDF(nil, Ka, "ConfirmationKeys" || AAD), where AAD is the associated data each given to each endpoint, or nil (empty string) if none was provided. AAD may also include a string identifying the protocol, ciphersuite and all its parameters, including the definition of the group, and the element M and N. It may be omitted. The length of each of KcA and KcB is equal to half of the KDF output, e.g., |KcA| = |KcB| = 128 bits for HKDF with SHA256.

The resulting key schedule for this protocol, given transcript TT and additional associated data AAD, is as follows.

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

A and B output Ke as the shared secret from the protocol. Ka and its derived keys (KcA and KcB) are not used for anything except key confirmation.

## 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.

| 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-512 | SHA512 [RFC6234] | HKDF [RFC5869] | HMAC [RFC2104] |
| edwards25519 [RFC7748] | SHA256 [RFC6234] | HKDF [RFC5869] | HMAC [RFC2104] |
| edwards448 [RFC7748] | 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: SPAKE2+ Ciphersuites

The following points represent permissible point generation seeds for the groups listed in the Table Table 1, 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. Security Considerations

SPAKE2+ appears in [TDH] along with a path to a proof that server compromise does not lead to password compromise under the DH assumption (though the corresponding model excludes pre-computation attacks).

Elements received from a peer MUST be checked for group membership: failure to properly validate group elements can lead to attacks. Beyond the cofactor multiplication checks to ensure that these elements are in the prime order subgroup of G, it is essential that endpoints verify received points are members of G.

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.

## 7. IANA Considerations

No IANA action is required.

## 8. Acknowledgments

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

## 9. References

### 9.1. Normative References

[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>.

[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.

### 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)", draft-ietf-mmusic-sdp-uks-07 (work in progress), August 2019.

[REF] Abdalla, M. and D. Pointcheval, "Simple Password-Based Encrypted Key Exchange Protocols.", Feb 2005.

Appears in A. Menezes, editor. Topics in Cryptography-CT-RSA 2005, Volume 3376 of Lecture Notes in Computer Science, pages 191-208, San Francisco, CA, US. Springer-Verlag, Berlin, Germany.

[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>.

[SPAKE2] Ladd, W. and B. Kaduk, "SPAKE2, a PAKE.", Feb 2020.

SPAKE2, a PAKE

[TDH] Cash, D., Kiltz, E., and V. Shoup, "The Twin-Diffie Hellman Problem and Applications", 2008.

EUROCRYPT 2008. Volume 4965 of Lecture notes in Computer Science, pages 127-145. Springer-Verlag, Berlin, Germany.

## 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 ciphersuite.  (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.

### B.1.  SPAKE2+ Test Vectors

```
SPAKE2+(A='client', B='server')
w0 = 0x4f9e28322a64f9dc7a01b282cc51e2abc4f9ed568805ca84f4ed3ef806516
cf8
w1 = 0x8d73e4ca273859c873d809431d15f30e2b722007964e32699160b54fda3ee
855
L = 0x0491bb1e6672e71ad80b17d13f7a72ca2fe7f882d4bd734e2d140f67ab49d2
c3e76dbcf706954bd9ada4e3a7fc50cf9294729f93b130ada3d3a4ae98cc7e7b6971
```

```
X = 0x04879567d09560c02be565429036ed1d2fc3ca53f2eb6fadda4dba09eff3a0
096f032f0e227207ebebe05e1e95de325dfffe579c8aae76054030e5435fd5298c75
Y = 0x04b595a25588a2fba757195a756d289c191240296699f61fee8f15a7a741a4
23d48bd44cf544b409bbe4262a8045051e734567548ba43b3117efd6fb03acf41aff
Z = 0x047bb4661db7085d019cffa8495aba73d22f87ab8ba22e789477ef933b916f
412863aeb2dbc8003e4f1c2193290338ea0c7d786d30ca47a48eea273375a0c72ca1
V = 0x0417658e1e9707a29d429a4733d3bee703574aec222e781a6e7e5f5e504908
11aabf28e112fee32a37c228df9b53e6220468a2f6f07427604d8917870ac965eec7
TT = 0x0600000000000000636c69656e740600000000000000736572766572410000
00000000000004879567d09560c02be565429036ed1d2fc3ca53f2eb6fadda4dba09e
ff3a0096f032f0e227207ebebe05e1e95de325dfffe579c8aae76054030e5435fd52
98c75410000000000000004b595a25588a2fba757195a756d289c191240296699f61
fee8f15a7a741a423d48bd44cf544b409bbe4262a8045051e734567548ba43b3117e
fd6fb03acf41aff4100000000000000047bb4661db7085d019cffa8495aba73d22f8
7ab8ba22e789477ef933b916f412863aeb2dbc8003e4f1c2193290338ea0c7d786d3
0ca47a48eea273375a0c72ca141000000000000000417658e1e9707a29d429a4733d
3bee703574aec222e781a6e7e5f5e50490811aabf28e112fee32a37c228df9b53e62
20468a2f6f07427604d8917870ac965eec720000000000000004f9e28322a64f9dc7
a01b282cc51e2abc4f9ed568805ca84f4ed3ef806516cf8
Ka = 0xbf800062847c5182bf5c549b05ea6cce
Ke = 0xce9acf88ff9440777bda3e34fa4993cd
KcA = 0x73c6a5597096e99b8025172bb45b4a2f
KcB = 0x96a801673bd07b51d61fbaea03ef17cf
MAC(A) = 0xcab37c89192f9ad90ca5e6b8eadb130d313b51d24b7889e2536f7c800
26e076a
MAC(B) = 0xf7076a78a3d16f0c62cb9e40bd1a91b68dee144b87016e2dae81c36e9
73f3b2e

SPAKE2+(A='client', B='')
w0 = 0x4f9e28322a64f9dc7a01b282cc51e2abc4f9ed568805ca84f4ed3ef806516
cf8
w1 = 0x8d73e4ca273859c873d809431d15f30e2b722007964e32699160b54fda3ee
855
L = 0x0491bb1e6672e71ad80b17d13f7a72ca2fe7f882d4bd734e2d140f67ab49d2
c3e76dbcf706954bd9ada4e3a7fc50cf9294729f93b130ada3d3a4ae98cc7e7b6971
X = 0x0426fbedb3b9ccea93d609838dcc1d4baebdbb9c287763ed4cdb2d3cc76f78
8d3388db3da1f63e945f3f1ba17f7b986ab9ed3170359ee406cbb40f3e3719453b15
Y = 0x04d4960922990acb87809e734fed2c2ccb72fd26ed173e8207cdc6220073ac
5017660788e96db275f6edf2ba400d4e090273c24dc907d80ff9cad7f42fd9f79c3f
Z = 0x0421996ff4d9c05b2389ae05118c519679df5d6de258b31f2a17da7604c8e3
c17bb3c4aae2ae4217951aa82144cb8b677be8061f28893f70216c1e11ba2bacd50d
V = 0x04729f7c6c5bd68310345b1a10b84ea7db64c70441da2255992208b7a8e0b3
9d4f0e634acf7d440b4552a41df291ac6a409f8cf5a47cec9fed5f85fea1241379a4
TT = 0x0600000000000000636c69656e74410000000000000000426fbedb3b9ccea9
3d609838dcc1d4baebdbb9c287763ed4cdb2d3cc76f788d3388db3da1f63e945f3f1
ba17f7b986ab9ed3170359ee406cbb40f3e3719453b154100000000000000004d4960
922990acb87809e734fed2c2ccb72fd26ed173e8207cdc6220073ac5017660788e96
db275f6edf2ba400d4e090273c24dc907d80ff9cad7f42fd9f79c3f4100000000000
```

```
0000421996ff4d9c05b2389ae05118c519679df5d6de258b31f2a17da7604c8e3c17
bb3c4aae2ae4217951aa82144cb8b677be8061f28893f70216c1e11ba2bacd50d410
000000000000004729f7c6c5bd68310345b1a10b84ea7db64c70441da2255992208b
7a8e0b39d4f0e634acf7d440b4552a41df291ac6a409f8cf5a47cec9fed5f85fea12
41379a420000000000000004f9e28322a64f9dc7a01b282cc51e2abc4f9ed568805c
a84f4ed3ef806516cf8
Ka = 0xfd19104b836b0ba9dfaaeab88610be57
Ke = 0x90337374f974f673707de5ba1b98e5b8
KcA = 0x2e10249c566677c8826b48ad10b19bb5
KcB = 0x4fcaf8fd0bfcaeeabb9d6f48e264e4a3
MAC(A) = 0xaaef200ea5f5c41e1fdb9b3455dde715cd8aa96f8afd3274f7159c3c5
4887f2c
MAC(B) = 0x926eadbf4b720b46ea622d7100e0013eb24d1591496846a604cf90c14
46fe0e4

SPAKE2+(A='', B='server')
w0 = 0x4f9e28322a64f9dc7a01b282cc51e2abc4f9ed568805ca84f4ed3ef806516
cf8
w1 = 0x8d73e4ca273859c873d809431d15f30e2b722007964e32699160b54fda3ee
855
L = 0x0491bb1e6672e71ad80b17d13f7a72ca2fe7f882d4bd734e2d140f67ab49d2
c3e76dbcf706954bd9ada4e3a7fc50cf9294729f93b130ada3d3a4ae98cc7e7b6971
X = 0x0463a7531acd204e7d83ac6562278d7ced01a715eff937a25520bd2220c626
33db0ea510591c5cd23159a7a97181ec24433aac6e628f16d42c455fcae668411e34
Y = 0x0433625217e2ccc0c545126f756d999c16df68b14b73b3fe473593c1d3a0d7
287b43b353177806c641588ec969852b56b17190d6ebe80313de74e5eee0c1403025
Z = 0x049ef5ea46e8ca42f3e822c598858ca347bf19cc74a8a1fbfd836ec4d77bee
7f0cd4d42f4f817caa3360c918d2538d7c96de5db47a72949ca2888d02c18ea6f92b
V = 0x0408a70fc9dca87b70a7d4a074bdcca0222806f0caa0542d8d62aecf535ea8
ffbc5e48419c5127a0f7f03685013c09d22f797523d26e7db159fecaccebc54ed2a7
TT = 0x06000000000000007365727665724100000000000000000463a7531acd204e7
d83ac6562278d7ced01a715eff937a25520bd2220c62633db0ea510591c5cd23159a
7a97181ec24433aac6e628f16d42c455fcae668411e3441000000000000000433625
217e2ccc0c545126f756d999c16df68b14b73b3fe473593c1d3a0d7287b43b353177
806c641588ec969852b56b17190d6ebe80313de74e5eee0c14030254100000000000
000049ef5ea46e8ca42f3e822c598858ca347bf19cc74a8a1fbfd836ec4d77bee7f0
cd4d42f4f817caa3360c918d2538d7c96de5db47a72949ca2888d02c18ea6f92b410
000000000000000408a70fc9dca87b70a7d4a074bdcca0222806f0caa0542d8d62aec
f535ea8ffbc5e48419c5127a0f7f03685013c09d22f797523d26e7db159fecaccebc
54ed2a72000000000000000004f9e28322a64f9dc7a01b282cc51e2abc4f9ed568805c
a84f4ed3ef806516cf8
Ka = 0x5c85900898b2079c9de09ebef63cebd1
Ke = 0x13c812476859e909682c3be7436bfef0
KcA = 0x77bd636ab9bf153339c5724ee04f87a7
KcB = 0x194325b27d7c291c94a689ddafeaaa3c
MAC(A) = 0x3bb61248a1fd2946743314848fc501eb3455eb113bd8966e200de14d5
e412688
MAC(B) = 0x3e7912bd2a85a1f56d36fbb16de29834b000d49e50d4c17f992942ee5
```

```
9255f1e

SPAKE2+(A='', B='')
w0 = 0x4f9e28322a64f9dc7a01b282cc51e2abc4f9ed568805ca84f4ed3ef806516
cf8
w1 = 0x8d73e4ca273859c873d809431d15f30e2b722007964e32699160b54fda3ee
855
L = 0x0491bb1e6672e71ad80b17d13f7a72ca2fe7f882d4bd734e2d140f67ab49d2
c3e76dbcf706954bd9ada4e3a7fc50cf9294729f93b130ada3d3a4ae98cc7e7b6971
X = 0x04f60f506cfa07506d4bfd2b3f56038b1c001fe6826374122c30e914747eab
647988702cc70210eb2aa625e603d56961af16ec543ee3d4d2cb90d6fe2f3c1d1180
Y = 0x046898fafef34fff9936217608151af08313305cf8e6f9add10d721c04a018
607f5b5aca327e150cd5d588de83e46491ec766e2cf87da9fb07dc3745c0630b03bb
Z = 0x042adeeea1417cc6c592fef772da8ba0f3aea69a5fb15923d0e9ae7c3301c7
ff87e9ff9fba292ad410e4af71770858e9a314f1deb75f77bde276d3cc8b45ffd70c
V = 0x04845c130c8c20865828e21ed3400abea726b07fdeb7533fa6017accc37e0b
e4922241dad44846112e42bee999501fdb4d09fc798e4677d403d10bfa862928584e
TT = 0x41000000000000004f60f506cfa07506d4bfd2b3f56038b1c001fe682637
4122c30e914747eab647988702cc70210eb2aa625e603d56961af16ec543ee3d4d2c
b90d6fe2f3c1d118041000000000000000046898fafef34fff9936217608151af0831
3305cf8e6f9add10d721c04a018607f5b5aca327e150cd5d588de83e46491ec766e2
cf87da9fb07dc3745c0630b03bb41000000000000000042adeeea1417cc6c592fef77
2da8ba0f3aea69a5fb15923d0e9ae7c3301c7ff87e9ff9fba292ad410e4af7177085
8e9a314f1deb75f77bde276d3cc8b45ffd70c41000000000000000004845c130c8c208
65828e21ed3400abea726b07fdeb7533fa6017accc37e0be4922241dad44846112e4
2bee999501fdb4d09fc798e4677d403d10bfa862928584e20000000000000004f9e2
8322a64f9dc7a01b282cc51e2abc4f9ed568805ca84f4ed3ef806516cf8
Ka = 0x850a18a77b14ef5e71b4a239413630a8
Ke = 0x4454819282b3e886a7e65b7b0de7cc62
KcA = 0x05df6196c12d6203768c73d875e2bfc5
KcB = 0xb58e61c322f685add02c125767e4fbb7
MAC(A) = 0x33e50d29f8eacc67bfdab4a6c46c88d75ac3308416c64dfbb0d7fb1c0
feda5b0
MAC(B) = 0x55434e5e501ad2d476aa1ae334ef27ba437a5dea87683defac575a63b
548ca64
```

Authors' Addresses

Tim Taubert
Apple, Inc.

Email: ttaubert@apple.com

Christopher A. Wood
Apple, Inc.

Email: cawood@apple.com