Network Working Group
Internet-Draft
Intended status: Informational
Expires: April 5, 2020

W. Ladd
Cloudflare
B. Kaduk, Ed.
Akamai

October 3, 2019

SPAKE2, a PAKE
draft-irtf-cfrg-spake2-09

Abstract

This document describes SPAKE2 which is a protocol for two parties that share a password to derive a strong shared key with no risk of disclosing the password. This method is compatible with any group, is computationally efficient, and SPAKE2 has a security proof.

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 April 5, 2020.

Copyright Notice

Table of Contents

1. Introduction

This document describes SPAKE2, a means for two parties that share a password to derive a strong shared key with no risk of disclosing the password. This password-based key exchange protocol is compatible with any group (requiring only a scheme to map a random input of fixed length per group to a random group element), is computationally efficient, and has a security proof. Predetermined parameters for a selection of commonly used groups are also provided for use by other protocols.

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. Setup

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. In the case of a composite order group we will work in the quotient group. 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 H are SHA256 or SHA512 [RFC6234]. Let MHF be a memory-hard hash function designed to slow down brute-force attackers. Scrypt [RFC7914] is a common example of this function. The output length of MHF matches that of Hash. Parameter selection for MHF is out of scope for 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 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. We also assume A and B share an integer w; typically w = MHF(pw) mod p, for a user-supplied password pw. Standards such NIST.SP.800-56Ar3 suggest taking mod p of a hash value that is 64

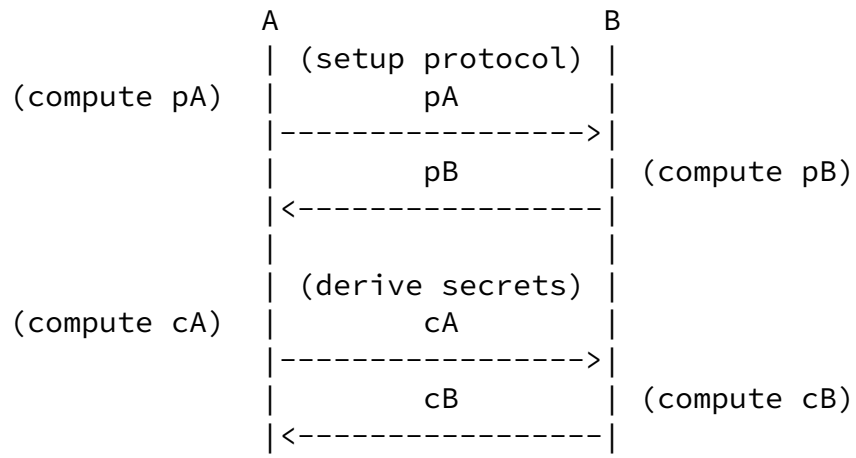
bits longer than that needed to represent p to remove statistical bias introduced by the modulation. Protocols using this specification must define the method used to compute w: it may be necessary to carry out various forms of normalization of the password before hashing [RFC8265]. The hashing algorithm SHOULD be a MHF so as to slow down brute-force attackers.

3.2. Protocol Flow

SPAKE2 is a one round protocols to establish 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. During the first round, A sends a public share pA to B, 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, A sends a key confirmation message cA to B, and B responds with its own key confirmation messgage cB. Both parties MUST NOT consider the protocol complete prior to receipt and validation of these key confirmation messages.

This sample trace is shown below.



3.3. SPAKE2

To begin, A picks x randomly and uniformly from the integers in [0,p), and calculates X=x*P and T=w*M+X, then transmits pA=T to B.

B selects y randomly and uniformly from the integers in [0,p), and calculates Y=y*P, S=w*N+Y, then transmits pB=S to A.

Both A and B calculate a group element K. A calculates it as h*x*(S-wN), while B calculates it as h*y*(T-w*M). A knows S because it has received it, and likewise B knows T. The multiplication by h prevents small subgroup confinement attacks by computing a unique value in the quotient group. (Any text on abstract algebra explains this notion.)

K is a shared value, though it MUST NOT be used as a shared secret. Both A and B must derive two shared secrets from K and the protocol transcript. This prevents man-in-the-middle attackers from inserting themselves into the exchange. The transcript TT is encoded as follows:

```
TT = len(A) || A || len(B) || B || len(S) || S
  || len(T) || T || len(K) || K || len(w) || w
```


If an identity is absent, it is omitted from the transcript entirely. For example, if both A and B are absent, then TT = len(S) || S || len(T) || T || len(K) || K || len(w) || w. Likewise, if only A is absent, TT = len(B) || B || len(S) || S || len(T) || T || len(K) || K || len(w) || w. 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. A MUST send B a key confirmation message so both parties agree upon these shared secrets. This confirmation message F is computed as a MAC over the protocol transcript TT using KcA, as follows: F = MAC(KcA, TT). Similarly, B MUST send A a confirmation message using a MAC computed equivalently except with the use of KcB. 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 if none was provided. The length of each of KcA and KcB is equal to half of the KDF output, e.g., |KcA| = |KcB| = 128 bits for HKDF(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 are not used for anything except key confirmation.

5. Ciphersuites

This section documents SPAKE2 and 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

A security proof of SPAKE2 for prime order groups is found in [REF]. Note that the choice of M and N is critical for the security proof. The generation method specified in this document is designed to eliminate concerns related to knowing discrete logs of M and N.

Elements received from a peer MUST be checked for group membership: failure to properly validate group elements can lead to attacks 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.

SPAKE2 does not support augmentation. As a result, the server has to store a password equivalent. This is considered a significant drawback, and so SPAKE2+ also appears in this document.

7. IANA Considerations

No IANA action is required.

8. Acknowledgments

Special thanks to Nathaniel McCallum and Greg Hudson for generation of test vectors. Thanks to Mike Hamburg for advice on how to deal with cofactors. Greg Hudson also suggested the addition of warnings on the reuse of x and y. Thanks to Fedor Brunner, Adam Langley, and the members of the CFRG for comments and advice. Chris Wood contributed substantial text and reformatting to address the excellent review comments from Kenny Paterson.

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>.

[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 and SPAKE2+ using the P256-SHA256-HKDF-HMAC ciphersuite. (Choice of MHF 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')
w = 0x7741cf8c80b9bee583abac3d38daa6b807fed38b06580cb75ee85319d25fed
e6
X = 0x04ac6827b3a9110d1e663bcd4f5de668da34a9f45e464e99067bbea53f1ed4
d8abbdd234c05b3a3a8a778ee47f244cca1a79acb7052d5e58530311a9af077ba179
T = 0x04e02acfbbfb081fc38b5bab999b5e25a5ffd0b1ac48eae24fcc8e49ac5e0d
8a790914419a100e205605f9862daa848e99cea455263f0c6e06bc5a911f3e10a16b
```



```
Y = 0x0413c45ab093a75c4b2a6e71f957eec3859807858325258b0fa43df5a6efd2
63c59b9c1fbfd55bc5e75fd3e7ba8af6799a99b225fe6c30e6c2a2e0ab4962136ba8
S = 0x047aad50ba7bd6a5eacbead7689f7146f1a4219fa071cce1755f80280cc6c3
a5a73cf469f2a294a0b74a5c07054585ccd447f3f633d8631f3bf43442449e9efeba
TT = 0x0600000000000000636c69656e74060000000000000007365727665724100
0000000000047aad50ba7bd6a5eacbead7689f7146f1a4219fa071cce1755f80280
cc6c3a5a73cf469f2a294a0b74a5c07054585ccd447f3f633d8631f3bf43442449e9
efeba41000000000000004e02acfbbfb081fc38b5bab999b5e25a5ffd0b1ac48eae
24fcc8e49ac5e0d8a790914419a100e205605f9862daa848e99cea455263f0c6e06b
c5a911f3e10a16b41000000000000004d01fc08bbae9b6abe2f4d6893cc9f810433
2e19be5f5881c6b9f077e1feff55023da74db65fae320fad8f0dd38e1323f5336f3f
53c9c9dec06710f18f556bd2020000000000000007741cf8c80b9bee583abac3d38d
aa6b807fed38b06580cb75ee85319d25fede6
Ka = 0x2b5e350c58d530c3586f75bf2a155c4b
Ke = 0x238509f7adf0dc72500b2d1315737a27
KcA = 0xc33d2ef8e37a7e545c14c7fcfdc9db94
KcB = 0x18a81cec7eb83416db6615cb3bc03fcb
MAC(A) = 0x29e9a63d243f2f0db5532d2eb0dbaa617803b85feb31566d0cb9457e3
```

```
03bcfa6
MAC(B) = 0x487e4cbe98b6287272d043e169a19b6c4682d0481c92f53f1ee03d4b8
6c3f43e

SPAKE2(A='client', B='')
w = 0x7741cf8c80b9bee583abac3d38daa6b807fed38b06580cb75ee85319d25fed
e6
X = 0x048b5d7b44b02c4c868f4486ec55bd2380ec34cd5fa5dbff1079a79097e305
0b34fa91272331729357c86cbb30d371e252dc915aeaa314921b1f09f74816f96a12
T = 0x04839f44931b88d12769e601d0ec480b6c9ea95e70ba361ba14bf513e5186a
6c302e6f409bd01f1030ad3cdac1e08965217e430ca7f9bce698111ae8a4d0530efd
Y = 0x0446419d63037d0bbaca224f89987c776bfea2e0913ccda0790079212f476d
6fd1ec997a02821a804f885e4f29b172b27c92251d883efe201cae106c239108c0c7
S = 0x042926b2dbcc5d0cb23ca123cc4133242f2998439af5380434a4bd5fd76fbb
c030b5563218d0184fa3fd303482a679c9555ccea41098b26b6ee16fe35c792b1fda
TT = 0x0600000000000000636c69656e74410000000000000000042926b2dbcc5d0cb
23ca123cc4133242f2998439af5380434a4bd5fd76fbbc030b5563218d0184fa3fd3
03482a679c9555ccea41098b26b6ee16fe35c792b1fda4100000000000000004839f4
4931b88d12769e601d0ec480b6c9ea95e70ba361ba14bf513e5186a6c302e6f409bd
01f1030ad3cdac1e08965217e430ca7f9bce698111ae8a4d0530efd4100000000000
000041d9e3c88db68247ab50264a6090e2e524bda3049dbc53c4df708e37bd76913b
8cf5954c4d0f835331f185fef4ff1c6115cf0eb8ce27e8224bf5f76c75b182308200
0000000000007741cf8c80b9bee583abac3d38daa6b807fed38b06580cb75ee8531
9d25fede6
Ka = 0xfc8482d5d7623a75ad09721d631d1392
Ke = 0x93f618fe24d0d5a54b320f498dbd3ecb
KcA = 0x75b20fc4205d6217a22156f918dd03b1
KcB = 0x3bf3a5d3876d9d12dc54cab927acd5f7
MAC(A) = 0xd4994b751eb832b2836ad674cd615c643053278864a63e263bc2f324b
9a04ddd
```



```
MAC(B) = 0x23cf761999b7603adf5507b50c9bda4eaabe8fa7a9ad0280729dfcd00
8b2bf05

SPAKE2(A='', B='server')
w = 0x7741cf8c80b9bee583abac3d38daa6b807fed38b06580cb75ee85319d25fed
e6
X = 0x0465e8b4709ba622bc97af5dde3b41757c2114bfc5abb10141245cb01d62ca
0d7360e1169cd518f9351bbfa44a66cc5f3bcb60661a04f39b04a3d504046db67884
T = 0x0482f64286419ff46362faf781776edf908740b8ff612e0bfe3c90cdc553ba
db7f882a4110ee71fa13a693b5ce96ceba5798636555d074648d4521e3b63dc14872
Y = 0x041aa11299692627a7cac122d4c14606ff700a8be6a0fb1c42f3762d629893
```

```
ab3ca51e4a48c798fc8c6b9dcfda1ad33099ed2f73abe6b3500ce383f54011430c26
S = 0x04adba3c3b9a74d9769651d09aedb37d22b9471b9e408e2b98fdd4188c12fa
c731e9dc87e029f7dee0213660ddf0791f50dd8fd32f7152015be0489125b3831b4b
TT = 0x0600000000000000073657276657241000000000000000004adba3c3b9a74d97
69651d09aedb37d22b9471b9e408e2b98fdd4188c12fac731e9dc87e029f7dee0213
660ddf0791f50dd8fd32f7152015be0489125b3831b4b4100000000000000000482f64
286419ff46362faf781776edf908740b8ff612e0bfe3c90cdc553badb7f882a4110e
e71fa13a693b5ce96ceba5798636555d074648d4521e3b63dc148724100000000000
00004a406929024a5275372531c85c54fd222f35bfdb1cdf1bd1abe82d5c837744d9
3ea2979962eb374d4feda37b178e91711c52edd453178cf69748e0a3d9ef073c2200
0000000000007741cf8c80b9bee583abac3d38daa6b807fed38b06580cb75ee8531
9d25fede6
Ka = 0xcd9c33c6329761919486d0041faccb56
Ke = 0xa08125eeed51c61ad93b2ff7d8ec3cd5
KcA = 0x60056386cbe06ba199fa6aef81dfb273
KcB = 0x5e5a591b4426d47190aecb2fc4527140
MAC(A) = 0xf0dcfb4fa874e3fcbadd44b6eb26a64d1d5c6e50034934934551f172d
3cdc50e
MAC(B) = 0x52e7a505c0b73db656108554a854c3f33bfb01edcc1ee52aa27ceb1cb
ef7f47b

SPAKE2(A='', B='')
w = 0x7741cf8c80b9bee583abac3d38daa6b807fed38b06580cb75ee85319d25fed
e6
X = 0x04fbeb44d6b772fa390fcced51be7316107e608ddf4ab5dcc9f1b2e24bf667
7f3232cdeeb39a61621a9e48028997d449894212eb54b6f12bdbd9baf8f1c909a740
T = 0x04887af8439d743215f26d48314835b024b9301ea508eac3a339241672fbba
09f63e155b1df5d31ccc63babafc00ffff6e258c692aed84a859fd4960d99fcec777
Y = 0x04bb4727c5c5c50ae34d5148ec6797e5ebf93ae51c5c6cfd48579c41436823
1ac8769142bf6a0109bd2b86dd901c6054629ce2c6b982326c9cd9a3685c4cf0640d
S = 0x04665b5101132528be32f4b4762d6ae80273bbe74e151fc2320da373e146ee
cd33038ff8099782f3781160244672cb43b4d9f2007da9b617c1890845440da0ca53
TT = 0x4100000000000000004665b5101132528be32f4b4762d6ae80273bbe74e151
fc2320da373e146eecd33038ff8099782f3781160244672cb43b4d9f2007da9b617c
1890845440da0ca53410000000000000000004887af8439d743215f26d48314835b024b
9301ea508eac3a339241672fbba09f63e155b1df5d31ccc63babafc00ffff6e258c6
92aed84a859fd4960d99fcec77741000000000000000004aacd2378990cecd338c7cac
```



```
d132ce633bc424ac5d4ab32f539ccf31f15deef62463253790e139b461c5137944fc
6a5ffd895dbe0d3960b01f6d662fc41057a702000000000000000007741cf8c80b9bee
583abac3d38daa6b807fed38b06580cb75ee85319d25fede6
Ka = 0x16b10f1541c24c630f462f7e0aa57ddf
```

```
Ke = 0xb7ae8b61938e3dfad8b9ce1d2865533f
KcA = 0x3398d6c7de402a9ae89a4594d5576c21
KcB = 0x6894ab44d7ba7f3a40a772d1476593d9
MAC(A) = 0x12fce7f0aecc1dba393a7e5612e6357becc5e3d07cd41ffd35c6d652f
29cde60
MAC(B) = 0xac36c6d186c3b824f4cfe099f035cf3aed4162d08886d32fa1806e5bf
4015255
```

B.2. 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 = 0x0600000000000000636c69656e74060000000000000007365727665724100
0000000000004879567d09560c02be565429036ed1d2fc3ca53f2eb6fadda4dba09e
ff3a0096f032f0e227207ebebe05e1e95de325dfffe579c8aae76054030e5435fd52
98c754100000000000000004b595a25588a2fba757195a756d289c191240296699f61
fee8f15a7a741a423d48bd44cf544b409bbe4262a8045051e734567548ba43b3117e
fd6fb03acf41aff4100000000000000047bb4661db7085d019cffa8495aba73d22f8
7ab8ba22e789477ef933b916f412863aeb2dbc8003e4f1c2193290338ea0c7d786d3
0ca47a48eea273375a0c72ca1410000000000000000417658e1e9707a29d429a4733d
3bee703574aec222e781a6e7e5f5e50490811aabf28e112fee32a37c228df9b53e62
20468a2f6f07427604d8917870ac965eec7200000000000000004f9e28322a64f9dc7
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 = 0x0600000000000000636c69656e744100000000000000000426fbedb3b9ccea9
3d609838dcc1d4baebdbb9c287763ed4cdb2d3cc76f788d3388db3da1f63e945f3f1
ba17f7b986ab9ed3170359ee406cbb40f3e3719453b1541000000000000000004d4960
922990acb87809e734fed2c2ccb72fd26ed173e8207cdc6220073ac5017660788e96
db275f6edf2ba400d4e090273c24dc907d80ff9cad7f42fd9f79c3f4100000000000
0000421996ff4d9c05b2389ae05118c519679df5d6de258b31f2a17da7604c8e3c17
bb3c4aae2ae4217951aa82144cb8b677be8061f28893f70216c1e11ba2bacd50d410
0000000000004729f7c6c5bd68310345b1a10b84ea7db64c70441da2255992208b
7a8e0b39d4f0e634acf7d440b4552a41df291ac6a409f8cf5a47cec9fed5f85fea12
41379a4200000000000000004f9e28322a64f9dc7a01b282cc51e2abc4f9ed568805c
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
```

V = 0x0408a70fc9dca87b70a7d4a074bdcca0222806f0caa0542d8d62aecf535ea8
ffbc5e48419c5127a0f7f03685013c09d22f797523d26e7db159fecaccebc54ed2a7
TT = 0x0600000000000000073657276657241000000000000000463a7531acd204e7
d83ac6562278d7ced01a715eff937a25520bd2220c62633db0ea510591c5cd23159a
7a97181ec24433aac6e628f16d42c455fcae668411e3441000000000000000433625
217e2ccc0c545126f756d999c16df68b14b73b3fe473593c1d3a0d7287b43b353177
806c641588ec969852b56b17190d6ebe80313de74e5eee0c14030254100000000000
000049ef5ea46e8ca42f3e822c598858ca347bf19cc74a8a1fbfd836ec4d77bee7f0
cd4d42f4f817caa3360c918d2538d7c96de5db47a72949ca2888d02c18ea6f92b410
000000000000000408a70fc9dca87b70a7d4a074bdcca0222806f0caa0542d8d62aec
f535ea8ffbc5e48419c5127a0f7f03685013c09d22f797523d26e7db159fecaccebc
54ed2a720000000000000004f9e28322a64f9dc7a01b282cc51e2abc4f9ed568805c
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 = 0x4100000000000000004f60f506cfa07506d4bfd2b3f56038b1c001fe682637
4122c30e914747eab647988702cc70210eb2aa625e603d56961af16ec543ee3d4d2c
b90d6fe2f3c1d1180410000000000000000046898fafef34fff9936217608151af0831

```
   3305cf8e6f9add10d721c04a018607f5b5aca327e150cd5d588de83e46491ec766e2
   cf87da9fb07dc3745c0630b03bb4100000000000000042adeeea1417cc6c592fef77
   2da8ba0f3aea69a5fb15923d0e9ae7c3301c7ff87e9ff9fba292ad410e4af7177085
   8e9a314f1deb75f77bde276d3cc8b45ffd70c4100000000000000004845c130c8c208
   65828e21ed3400abea726b07fdeb7533fa6017accc37e0be4922241dad44846112e4
   2bee999501fdb4d09fc798e4677d403d10bfa862928584e20000000000000004f9e2
   8322a64f9dc7a01b282cc51e2abc4f9ed568805ca84f4ed3ef806516cf8
   Ka = 0x850a18a77b14ef5e71b4a239413630a8
```



```
   Ke = 0x4454819282b3e886a7e65b7b0de7cc62
   KcA = 0x05df6196c12d6203768c73d875e2bfc5
   KcB = 0xb58e61c322f685add02c125767e4fbb7
   MAC(A) = 0x33e50d29f8eacc67bfdab4a6c46c88d75ac3308416c64dfbb0d7fb1c0
   feda5b0
   MAC(B) = 0x55434e5e501ad2d476aa1ae334ef27ba437a5dea87683defac575a63b
   548ca64
```

Authors' Addresses

Watson Ladd
Cloudflare

Email: watsonbladd@gmail.com

Benjamin Kaduk (editor)
Akamai Technologies

Email: kaduk@mit.edu