Internet Research Task Force
Internet-Draft
Intended status: Informational
Expires: December 4, 2021

W. Ladd
Cloudflare
B. Kaduk, Ed.
Akamai

June 2, 2021

SPAKE2, a PAKE
draft-irtf-cfrg-spake2-20

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. This document predated the CFRG PAKE competition and it was not selected. This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

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. While not selected as the result of the PAKE selection competition, because of existing use of variants in Kerberos and other applications it was felt publication was beneficial. This RFC represents the individual opinion(s) of one or more members of the Crypto Forum Research Group of the Internet Research Task Force (IRTF).

Many of these applications predated methods to hash to elliptic curves being available or predated the publication of the PAKEs that were chosen as an outcome of the PAKE selection competition. In cases where a symmetric PAKE is needed, and hashing onto an elliptic curve at protocol execution time is not available, SPAKE2 is useful.

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 gap Diffie-Hellman (GDH) 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 [SEC1] uncompressed or compressed 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.

For elliptic curves other than the ones in this document the methods of [I-D.irtf-cfrg-hash-to-curve] SHOULD be used to generate M and N, e.g. via M = h2c("M SPAKE2 seed OID x"), N= h2c("N SPAKE2 seed OID x") where x is an OID for the curve.

|| denotes concatenation of byte 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. Text strings in double quotes are treated as their ASCII encodings throughout this document.

KDF(ikm, salt, info) 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 6 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 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 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 protocol 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 message 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.

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

3.3. SPAKE2

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

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

Both A and B calculate a group element K. A calculates it as h*x*(T-w*N), while B calculates it as h*y*(S-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. This is a common mitigation against this kind of attack.

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 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
```

Here w is encoded as a big endian number padded to the length of p. This representation prevents timing attacks that otherwise would reveal the length of w. len(w) is thus a constant. We include it for consistency.

If an identity is absent, it is encoded as a zero-length string. 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 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), with |Ke| = |Ka|. The length of each key is equal to half of the digest output, e.g., 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(Ka,nil, "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 underlying hash output length, 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) = Ke || Ka
AAD -> KDF(Ka, nil, "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. Per-User M and N

To avoid concerns that an attacker needs to solve a single ECDH instance to break the authentication of SPAKE2, a variant based on using [I-D.irtf-cfrg-hash-to-curve] is also presented. In this variant, M and N are computed as follows:

```
M = h2c(Hash("M for SPAKE2" || len(A) || A || len(B) || B))
N = h2c(Hash("N for SPAKE2" || len(A) || A || len(B) || B))
```

In addition M and N may be equal to have a symmetric variant. The security of these variants is examined in [MNVAR]. This variant may not be suitable for protocols that require the messages to be exchanged symmetrically and do not know the exact identity of the parties before the flow begins.

6. 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)
```

7. Security Considerations

A security proof of SPAKE2 for prime order groups is found in [REF], reducing the security of SPAKE2 to the gap Diffie-Hellman assumption. Note that the choice of M and N is critical for the security proof. The generation methods specified in this document are 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. To generate these uniform numbers rejection sampling is recommended. Some implementations of elliptic curve multiplication may leak information about the length of the scalar: these MUST NOT be used.

SPAKE2 does not support augmentation. As a result, the server has to store a password equivalent. This is considered a significant drawback in some use cases.

The HMAC keys in this document are shorter than recommended in [RFC8032]. This is appropriate as the difficulty of the discrete logarithm problem is comparable with the difficulty of brute forcing the keys.

8. IANA Considerations

No IANA action is required.

9. Acknowledgments

Special thanks to Nathaniel McCallum and Greg Hudson for generation of M and N, and Cris Wood for 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,Liliya Akhmetzyanova, and the members of the CFRG for comments and advice. Thanks to Scott Fluhrer and those Crypto Panel experts involved in the PAKE selection process (https://github.com/cfrg/pake-selection) who have provided valuable comments. Chris Wood contributed substantial text and reformatting to address the excellent review comments from Kenny Paterson.

10. References

10.1. Normative References

[I-D.irtf-cfrg-hash-to-curve]
Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R., and C. Wood, "Hashing to Elliptic Curves", draft-irtf-cfrg-hash-to-curve-05 (work in progress), November 2019.

[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>.

10.2. 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.

[MNVAR] Abdalla, M., Barbosa, M., Bradley, T., Jarecki, S., Katz, J., and J. Xu, "Universally Composable Relaxed Password Authentication", August 2020.

Appears in Micciancio D., Ristenpart T. (eds) Advances in Cryptology -CRYPTO 20202. Crypto 20202. Lecture notes in Computer Science volume 12170. Springer.

[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>.

[SEC1] Standards for Efficient Cryptography Group, "SEC 1: Elliptic Curve Cryptography", May 2009.

[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 6.

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 MHF is omitted and values for w,x and y 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='server', B='client'
w = 0x2ee57912099d31560b3a44b1184b9b4866e904c49d12ac5042c97dca461b1a5f
x = 0x43dd0fd7215bdcb482879fca3220c6a968e66d70b1356cac18bb26c84a78d729
S = 0x04a56fa807caaa53a4d28dbb9853b9815c61a411118a6fe516a8798434751470
f9010153ac33d0d5f2047ffdb1a3e42c9b4e6be662766e1eeb4116988ede5f912c
y = 0xdcb60106f276b02606d8ef0a328c02e4b629f84f89786af5befb0bc75b6e66be
T = 0x0406557e482bd03097ad0cbaa5df82115460d951e3451962f1eaf4367a420676
```

```
d09857ccbc522686c83d1852abfa8ed6e4a1155cf8f1543ceca528afb591a1e0b7
K = 0x0412af7e89717850671913e6b469ace67bd90a4df8ce45c2af19010175e37eed
69f75897996d539356e2fa6a406d528501f907e04d97515fbe83db277b715d3325
TT = 0x06000000000000007365727665720600000000000000636c69656e744100000
00000000004a56fa807caaa53a4d28dbb9853b9815c61a411118a6fe516a8798434751
470f9010153ac33d0d5f2047ffdb1a3e42c9b4e6be662766e1eeb4116988ede5f912c4
1000000000000000406557e482bd03097ad0cbaa5df82115460d951e3451962f1eaf43
67a420676d09857ccbc522686c83d1852abfa8ed6e4a1155cf8f1543ceca528afb591a
1e0b74100000000000000000412af7e89717850671913e6b469ace67bd90a4df8ce45c2a
f19010175e37eed69f75897996d539356e2fa6a406d528501f907e04d97515fbe83db2
77b715d33252000000000000000002ee57912099d31560b3a44b1184b9b4866e904c49d1
2ac5042c97dca461b1a5f
Hash(TT) = 0x0e0672dc86f8e45565d338b0540abe6915bdf72e2b35b5c9e5663168e960a91bKe
= 0x0e0672dc86f8e45565d338b0540abe69
Ka = 0x15bdf72e2b35b5c9e5663168e960a91b
KcA = 0x00c12546835755c86d8c0db7851ae86f
KcB = 0xa9fa3406c3b781b93d804485430ca27a
A conf = 0x58ad4aa88e0b60d5061eb6b5dd93e80d9c4f00d127c65b3b35b1b5281fee38f0
B conf = 0xd3e2e547f1ae04f2dbdbf0fc4b79f8ecff2dff314b5d32fe9fcef2fb26dc459b


spake2: A='', B='client'
w = 0x0548d8729f730589e579b0475a582c1608138ddf7054b73b5381c7e883e2efae
x = 0x403abbe3b1b4b9ba17e3032849759d723939a27a27b9d921c500edde18ed654b
S = 0x04a897b769e681c62ac1c2357319a3d363f610839c4477720d24cbe32f5fd85f
44fb92ba966578c1b712be6962498834078262caa5b441ecfa9d4a9485720e918a
y = 0x903023b6598908936ea7c929bd761af6039577a9c3f9581064187c3049d87065
T = 0x04e0f816fd1c35e22065d5556215c097e799390d16661c386e0ecc84593974a6
1b881a8c82327687d0501862970c64565560cb5671f696048050ca66ca5f8cc7fc
K = 0x048f83ec9f6e4f87cc6f9dc740bdc2769725f923364f01c84148c049a39a735e
bda82eac03e00112fd6a5710682767cff5361f7e819e53d8d3c3a2922e0d837aa6
TT = 0x00000000000000000600000000000000636c69656e74410000000000000004a
897b769e681c62ac1c2357319a3d363f610839c4477720d24cbe32f5fd85f44fb92ba9
66578c1b712be6962498834078262caa5b441ecfa9d4a9485720e918a4100000000000
00004e0f816fd1c35e22065d5556215c097e799390d16661c386e0ecc84593974a61b8
81a8c82327687d0501862970c64565560cb5671f696048050ca66ca5f8cc7fc4100000
000000000048f83ec9f6e4f87cc6f9dc740bdc2769725f923364f01c84148c049a39a7
35ebda82eac03e00112fd6a5710682767cff5361f7e819e53d8d3c3a2922e0d837aa62
0000000000000000548d8729f730589e579b0475a582c1608138ddf7054b73b5381c7e
883e2efae
Hash(TT) = 0x642f05c473c2cd79909f9a841e2f30a70bf89b18180af97353ba198789c2b963Ke
= 0x642f05c473c2cd79909f9a841e2f30a7
Ka = 0x0bf89b18180af97353ba198789c2b963
KcA = 0xc6be376fc7cd1301fd0a13adf3e7bffd
KcB = 0xb7243f4ae60440a49b3f8cab3c1fba07
A conf = 0x47d29e6666af1b7dd450d571233085d7a9866e4d49d2645e2df975489521232b
B conf = 0x3313c5cefc361d27fb16847a91c2a73b766ffa90a4839122a9b70a2f6bd1d6df
```

```
spake2: A='server', B=''
w = 0x626e0cdc7b14c9db3e52a0b1b3a768c98e37852d5db30febe0497b14eae8c254
```

```
x = 0x07adb3db6bc623d3399726bfdbfd3d15a58ea776ab8a308b00392621291f9633
S = 0x04f88fb71c99bfffaea370966b7eb99cd4be0ff1a7d335caac4211c4afd855e2
e15a873b298503ad8ba1d9cbb9a392d2ba309b48bfd7879aefd0f2cea6009763b0
y = 0xb6a4fc8dbb629d4ba51d6f91ed1532cf87adec98f25dd153a75accafafedec16
T = 0x040c269d6be017dccb15182ac6bfcd9e2a14de019dd587eaf4bdfd353f031101
e7cca177f8eb362a6e83e7d5e729c0732e1b528879c086f39ba0f31a9661bd34db
K = 0x0445ee233b8ecb51ebd6e7da3f307e88a1616bae2166121221fdc0dadb986afa
f3ec8a988dc9c626fa3b99f58a7ca7c9b844bb3e8dd9554aafc5b53813504c1cbe
TT = 0x0600000000000000736572766572000000000000000000410000000000000004f
88fb71c99bfffaea370966b7eb99cd4be0ff1a7d335caac4211c4afd855e2e15a873b2
98503ad8ba1d9cbb9a392d2ba309b48bfd7879aefd0f2cea6009763b04100000000000
000040c269d6be017dccb15182ac6bfcd9e2a14de019dd587eaf4bdfd353f031101e7c
ca177f8eb362a6e83e7d5e729c0732e1b528879c086f39ba0f31a9661bd34db4100000
0000000000445ee233b8ecb51ebd6e7da3f307e88a1616bae2166121221fdc0dadb986
afaf3ec8a988dc9c626fa3b99f58a7ca7c9b844bb3e8dd9554aafc5b53813504c1cbe2
000000000000000626e0cdc7b14c9db3e52a0b1b3a768c98e37852d5db30febe0497b1
4eae8c254
Hash(TT) = 0x005184ff460da2ce59062c87733c299c3521297d736598fc0a1127600efa1afbKe
= 0x005184ff460da2ce59062c87733c299c
Ka = 0x3521297d736598fc0a1127600efa1afb
KcA = 0xf3da53604f0aeecea5a33be7bddf6edf
KcB = 0x9e3f86848736f159bd92b6e107ec6799
A conf = 0xbc9f9bbe99f26d0b2260e6456e05a86196a3307ec6663a18bf6ac825736533b2
B conf = 0xc2370e1bf813b086dff0d834e74425a06e6390f48f5411900276dcccc5a297ec

spake2: A='', B=''
w = 0x7bf46c454b4c1b25799527d896508afd5fc62ef4ec59db1efb49113063d70cca
x = 0x8cef65df64bb2d0f83540c53632de911b5b24b3eab6cc74a97609fd659e95473
S = 0x04a65b367a3f613cf9f0654b1b28a1e3a8a40387956c8ba6063e8658563890f4
6ca1ef6a676598889fc28de2950ab8120b79a5ef1ea4c9f44bc98f585634b46d66
y = 0xd7a66f64074a84652d8d623a92e20c9675c61cb5b4f6a0063e4648a2fdc02d53
T = 0x04589f13218822710d98d8b2123a079041052d9941b9cf88c6617ddb2fcc0494
662eea8ba6b64692dc318250030c6af045cb738bc81ba35b043c3dcb46adf6f58d
K = 0x041a3c03d51b452537ca2a1fea6110353c6d5ed483c4f0f86f4492ca3f378d40
a994b4477f93c64d928edbbcd3e85a7c709b7ea73ee97986ce3d1438e135543772
TT = 0x00000000000000000000000000000000410000000000000004a65b367a3f613
cf9f0654b1b28a1e3a8a40387956c8ba6063e8658563890f46ca1ef6a676598889fc28
de2950ab8120b79a5ef1ea4c9f44bc98f585634b46d66410000000000000004589f132
18822710d98d8b2123a079041052d9941b9cf88c6617ddb2fcc0494662eea8ba6b6469
2dc318250030c6af045cb738bc81ba35b043c3dcb46adf6f58d4100000000000000041
a3c03d51b452537ca2a1fea6110353c6d5ed483c4f0f86f4492ca3f378d40a994b4477
f93c64d928edbbcd3e85a7c709b7ea73ee97986ce3d1438e135543772200000000000000
0007bf46c454b4c1b25799527d896508afd5fc62ef4ec59db1efb49113063d70cca
Hash(TT) = 0xfc6374762ba5cf11f4b2caa08b2cd1b9907ae0e26e8d6234318d91583cd74c86Ke
= 0xfc6374762ba5cf11f4b2caa08b2cd1b9
Ka = 0x907ae0e26e8d6234318d91583cd74c86
KcA = 0x5dbd2f477166b7fb6d61febbd77a5563
KcB = 0x7689b4654407a5faeffdc8f18359d8a3
```

```
A conf = 0xdfb4db8d48ae5a675963ea5e6c19d98d4ea028d8e898dad96ea19a80ade95dca
B conf = 0xd0f0609d1613138d354f7e95f19fb556bf52d751947241e8c7118df5ef0ae175
```

Authors' Addresses

Watson Ladd
Cloudflare

Email: watsonbladd@gmail.com

Benjamin Kaduk (editor)
Akamai Technologies

Email: kaduk@mit.edu