tls R. Cragie
Internet-Draft ARM Ltd.
Intended status: Informational F. Hao
Expires: November 12, 2016 Newcastle University (UK)
May 11, 2016

Elliptic Curve J-PAKE Cipher Suites for Transport Layer Security (TLS)
draft-cragie-tls-ecjpake-00

Abstract

This document defines new cipher suites based on an Elliptic Curve Cryptography (ECC) variant of Password Authenticated Key Exchange by Juggling (J-PAKE) for the Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) protocols.

Table of Contents

1. Introduction

This document defines new cipher suites based on an Elliptic Curve Cryptography (ECC) variant of Password Authenticated Key Exchange by Juggling (J-PAKE) for version 1.2 of Transport Layer Security (TLS) protocol [RFC5246] as well as version 1.2 of the Datagram Transport Layer Security (DTLS) protocol [RFC6347]. The cipher suites are AEAD cipher suites using AES-CCM [CCM] based on the cipher suites defined in [RFC7251], using ECJ-PAKE as an alternative key establishment mechanism.

The existing set of TLS cipher suites are typically aimed at more traditional client-server interactions, for example, a web browser to web server. However, TLS and DTLS are increasingly being specified for use in Internet-of-Things (IoT) standards for peer-to-peer application layer interaction. For example, DTLS is specified as a binding to provide security for the CoAP protocol [RFC7252], which is widely used in IoT applications.

J-PAKE is a balanced password-authenticated key exchange (PAKE) protocol resistant to off-line dictionary attack designed by Feng Hao and Peter Ryan in 2008 [HR08]. The use of a PAKE for IoT devices is highly appropriate as it allows a simple method of commissioning IoT devices onto a network without requiring certificates to be issued and maintained for each device. An ECC variant of J-PAKE [J-PAKE] is particularly suited to IoT devices, which are often constrained with regard to memory and processing power. The cipher suite TLS_ECJPAKE_WITH_AES_128_CCM_8 as defined in this document is currently being used in the Thread protocol [THREAD].

1.1. Requirements Language

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

1.2. Terminology

AEAD

Authenticated Encryption with Associated Data.

ECJ-PAKE

Elliptic Curve Cryptography (ECC) variant of Password Authenticated Key Exchange by Juggling (J-PAKE).

ZKP

Zero-knowledge proof.

2. ECJ-PAKE Based AES-CCM Cipher Suites

The cipher suites defined in this document are based on the AES-CCM Authenticated Encryption with Associated Data (AEAD) algorithms AEAD_AES_128_CCM and AEAD_AES_256_CCM described in [RFC5116]. The following cipher suites are defined:

```
TLS_ECJPAKE_WITH_AES_128_CCM = {0xTBD, 0xTBD}
TLS_ECJPAKE_WITH_AES_256_CCM = {0xTBD, 0xTBD}
TLS_ECJPAKE_WITH_AES_128_CCM_8 = {0xTBD, 0xTBD}
TLS_ECJPAKE_WITH_AES_256_CCM_8 = {0xTBD, 0xTBD}
```

These cipher suites make use of the AEAD capability in TLS 1.2 [RFC5246]. Cipher suites ending with "8" use eight-octet authentication tags; the other cipher suites have 16-octet authentication tags. The HMAC truncation option described in Section 7 of [RFC6066] (which negotiates the "truncated_hmac" TLS extension) does not have an effect on the cipher suites defined in this document, because they do not use HMAC to protect TLS records.

The "nonce" input to the AEAD algorithm is as defined in [RFC6655].

These cipher suites make use of the default TLS 1.2 Pseudorandom Function (PRF), which uses HMAC with the SHA-256 hash function.

The following stipulations apply to the use of elliptic curves:

- Curves with a cofactor equal to one SHOULD be used; this simplifies their use.

- The uncompressed point format MUST be supported. Other point formats MAY be used.

o Fundamental ECC algorithms [RFC6090] MAY be used as an implementation method.

3. Notations

This section describes the notations used in this document.

3.1. Elliptic Curve Points

The generator (base point) of an elliptic curve is represented by the letter 'G':

G

A modified generator is represented by the letter 'G' concatenated with a single uppercase character:

GB

Elliptic curve points are represented using a single uppercase character or a single uppercase character concatenated with a single lowercase character or decimal digit, for example:

X

Xc

X2

Conversion to and from elliptic curve points to octet strings is as specified in Sections 2.3.3 and 2.3.4 of [SEC1].

Point multiplication is shown as an elliptic curve point multiplied by a scalar integer using the '*' operator, for example:

G*x

Point addition or subtraction is shown as the addition or subtraction of elliptic curve points or scalar multiplied elliptic curve points using the '+' and '-' operators respectively, for example:

X1 + X3 + X4

X*h + G*r

Xs - X4*x2*s

3.2. Integers

Integers are represented using a single lowercase character or a single lowercase character followed by a single lowercase character or decimal digit, for example:

x

xc

x2

Where expressed, integers are shown in hexadecimal and/or decimal form. Hexadecimal numbers have an '0x' prefix. For example:

0x12ab34cd

3132110061

Integer multiplication is shown as two integers multiplied together using the '*' operator:

x*s

Integer addition or subtraction is shown as the addition or subtraction of integers or multiplied integers using the '+' and '-' operators respectively:

v - x*h

3.3. Octet Strings

Octet strings are expressed in a hexadecimal form, with no '0x' prefix and with a space separator, first octet leftmost, for example:

12 ab 34 cd

3.4. Integer to Octet String Conversion

Integer to octet string conversion SHALL be performed as stated in Section 2.3.7 of [SEC1]. It is represented as follows:

M = str(mlen, x)

where x, mlen, and M are the parameters as stated in Section 2.3.7 of [SEC1].

3.5. Octet String to Integer Conversion

Octet string to integer conversion SHALL be as stated in section 2.3.8 of [SEC1]. It is represented as follows:

x = int(mlen, M)

where x, mlen, and M are the parameters as stated in Section 2.3.8 of [SEC1].

4. Handshake

The TLS-ECJ-PAKE handshake is as follows, augmented with parameters in braces to show the ECJ-PAKE material conveyed in each case:

```
Client                                            Server
------                                            ------
ClientHello          -------->
{(X1,ZKP(X1)),
(X2,ZKP(X2))}                                ServerHello
                                         {(X3, ZKP(X3)),
                                          (X4, ZKP(X4))}
                                       ServerKeyExchange
                                           {Xs, ZKP(Xs)}
                     <--------           ServerHelloDone
ClientKeyExchange
{Xc, ZKP(Xc)}
[ChangeCipherSpec]
Finished             -------->
                                      [ChangeCipherSpec]
                     <--------                  Finished
Application Data     <------->          Application Data
```

Figure 1: Message flow in a TLS-ECJ-PAKE handshake

5. Failure processing

If there are failures for any reason on client or server side, for example, Schnorr ZKP verification or missing extensions, the handshake SHALL abort immediately and send a TLS Error Alert message to the peer, using code 40 (handshake_failure) (see Section 7.2 of [RFC5246]).

6. ECJ-PAKE TLS Extensions and Modification

This section describes existing and newly-defined extensions required for ECJ-PAKE-TLS.

6.1. New Structure Definitions

TLS-ECJ-PAKE requires new structure definitions for:

o Public key and Schnorr ZKP pair

o Schnorr ZKP

6.1.1. Public Key and Schnorr ZKP Pair

The TLS structure is as follows:

```
struct {
    ECPoint X;
    ECSchnorrZKP zkp;
} ECJPAKEKeyKP;
```

X

Public key represented as an elliptic curve point. ECPoint is defined in [RFC4492].

zkp

ECSchnorrZKP is defined in Section 6.1.2.

6.1.2. Schnorr ZKP

The TLS structure is as follows:

```
struct {
    ECPoint V;
    opaque r<1..2^8-1>;
} ECSchnorrZKP;
```

V

Ephemeral public key represented as an elliptic curve point. ECPoint is defined in [RFC4492].

r

Schnorr signature.

6.2. ClientHello and ServerHello TLS Extensions

6.2.1. Existing Extensions

The following TLS extensions defined in Section 4 of [RFC4492] SHALL be present in ClientHello:

- o Supported Elliptic Curves Extension (NamedCurve, EllipticCurveList)

- o Supported Point Formats Extension (ECPointFormat, ECPointFormatList)

and the following TLS extension defined in Section 4 of [RFC4492] SHALL be present in ServerHello:

- o Supported Point Formats Extension (ECPointFormat, ECPointFormatList)

6.2.2. Additional Extensions

The following extension SHALL additionally be present in both ClientHello and ServerHello:

```
enum { ecjpake_key_kp_pair(TBC) } ExtensionType;

struct {
    opaque identity<0..2^16-1>;
    ECJPAKEKeyKP ecjpake_key_kp_pair_list[2];
} ECJPAKEKeyKPPairList;
```

identity

MAY be included if the Client or Server needs to uniquely identify themselves to the other party. An identity is used in the Schnorr ZKP hash calculation (see Section 7.2).

ecjpake_key_kp_pair_list

The list is precisely two elements long. The list in a ClientHello extension conveys public keys X1 and X2 and the list in a ServerHello extension conveys public keys X3 and X4, with associated Schnorr ZKPs.

Note: When used in conjunction with DTLS and denial-of-service countermeasures as described in Section 4.2.1 of [RFC6347], the ECJPAKEKeyKPPairList in the subsequent ClientHello message SHALL be the same as the ECJPAKEKeyKPPairList in initial ClientHello message,

i.e. the public keys X1 and X2 and associated Schnorr ZKPs SHALL be the same.

6.3. ServerKeyExchange

ServerKeyExchange is extended as follows:

```
enum { ecjpake } KeyExchangeAlgorithm;
```

ecjpake

Indicates the ServerKeyExchange message contains ServerECJPAKEParams.

ServerKeyExchange for ecjpake SHALL be formatted as follows:

```
struct {
    ECParameters curve_params;
    ECJPAKEKeyKP ecjpake_key_kp;
} ServerECJPAKEParams;

select (KeyExchangeAlgorithm) {
    case ecjpake:
        ServerECJPAKEParams params;
} ServerKeyExchange;
```

6.4. ClientKeyExchange

ClientKeyExchange is extended as follows:

```
enum { ecjpake } KeyExchangeAlgorithm;
```

ecjpake

Indicates the ClientKeyExchange message contains ClientECJPAKEParams.

ClientKeyExchange for ecjpake SHALL be formatted as follows:

```
struct {
    ECJPAKEKeyKP ecjpake_key_kp;
} ClientECJPAKEParams;

select (KeyExchangeAlgorithm) {
    case ecjpake:
        ClientECJPAKEParams params;
} ClientKeyExchange;
```

7. Calculations

This section describes the calculations required to populate the data conveyed between Client and Server and also calculations required to verify knowledge proofs.

The following notation is used throughout this section:

Order of the base point: n

7.1. User Identity Selection

The Schnorr ZKP hash calculation requires non-confidential user identities. These identities need to be unique in the context of a transaction and be different for each party. In a peer-to-peer transaction where there is no ambiguity of identity, the identities can be a simple string representing the Client and Server respectively:

Originator	Name	Identity	Length of identity
Client	"client"	63 6c 69 65 6e 74	6
Server	"server"	73 65 72 76 65 72	6

Table 1: Simple Client and Server identities

In a multi-party transaction, each party SHOULD additionally provide an identity in the ClientHello and/or ServerHello to uniquely distinguish their user identity.

7.2. Schnorr ZKP Hash Calculation

The hash calculation is defined as follows:

Public Key	Calculation
X1, X2, X3 and X4	h = SHA-256(G, V, X, ID) mod n
Xs	h = SHA-256(GB, V, Xs, IDs) mod n
Xc	h = SHA-256(GA, V, Xc, IDc) mod n

Table 2: Schnorr ZKP Hash Calculation

Each item in the hash calculation is prepended with its length in octets represented an octet (length 4), formed by applying integer to

octet string conversion as defined in Section 3.4. For example, the length of an uncompressed octet string representation of a public key is 65 (decimal) therefore the octet string (length 4) representation of 65 in hexadecimal is:

- 00 00 00 41

Each public key (elliptic curve point) is first converted to an octet string according to Section 2.3.3 of [SEC1].

The concatentation order of the hash is as follows:

1. G (or GA, GB): Generator

2. V: ZKP ephemeral public key

3. X (or Xs, Xc): Public key to be verified

4. ID (or IDc, IDs): User ID (see Section 7.1)

The hash is therefore performed on the concatenation as follows:

- H = SHA-256(lenG || G || lenV || V || lenX || X || lenID || ID)

An integer representation of the hash (see Section 3.5) is produced:

- h = int(H)

7.3. Shared Secret

The shared secret for the ServerKeyExchange and ClientKeyExchange calculations is required to be an integer in the range 1 to n-1. This section shows an example of how this could be practically accomplished using an initial password. The initial password is usually represented visually as a variable length character string using a subset of internationally recognized characters from the UTF-8 character set, which prevents the possibility of the resulting shared secret having the value 0. The initial password is then be converted into an octet string <password> using UTF-8 conversion. The integer shared secret calculation is thus defined as follows, using the function defined in Section 3.5:

```
s = int(<password>) mod n
```

7.3.1. Example

Password:

"d45yj8e"

Equivalent octet string M using UTF-8 conversion (no null termination):

64 34 35 79 6a 38 65

Length mlen:

7

Shared secret:

0x643435796a3865

28204901945981028 (decimal)

7.4. ClientHello and ServerHello Calculations

The structure ECJPAKEKeyKPPairList conveys the public key and associated Schnorr ZKP for ClientHello (X1 and X2) and ServerHello (X3 and X4).

7.4.1. Public Key Generation

For X1, X2, X3 and X4, the value for the public key part X of the ECJPAKEKeyKP structure is generated as follows:

The inputs are:

- o Base point: G
- o Order of the base point: n

The public key of the key pair is calculated as follows:

1. A random integer in the range 1 to n-1 is assigned to private key x.

2. A public key associated with x is generated and assigned to X:

   X = G*x

3. X is assigned to the public key part X of the ECJPAKEKeyKP structure.

7.4.2. Schnorr ZKP Generation

For X1, X2, X3 and X4, the values for the ZKP part zkp.V and zkp.r of the ECJPAKEKeyKP structure are generated as follows:

The inputs are:

- Base point: G
- Order of the base point: n
- Identity of originator: ID (IDc or IDs depending on context)
- Key pair to provide a ZKP of: (X,x) (public key: X, private key: x), where X is X1, X2, X3, or X4 and x is x1, x2, x3, or x4, depending on context

The ZKP is generated as follows:

1. A random integer in the range 1 to n-1 is assigned to ephemeral private key v.

2. An ephemeral public key associated with v is generated and assigned to V:

   V = G*v

3. An integer representation of a hash (see Section 7.2) is generated and assigned to h:

   h = int(SHA-256(G, V, X, ID)) mod n

4. A signature is generated and assigned to r:

   r = v - x*h mod n

5. V and r are assigned to the ZKP part zkp.V and zkp.r of the ECJPAKEKeyKP structure respectively.

7.4.3. Schnorr ZKP Verification

For X1, X2, X3 and X4, the ECJPAKEKeyKP structure is verified as follows:

The inputs are:

- o Base point: G
- o Order of the base point: n
- o Identity of originator: ID (IDc or IDs depending on context)
- o Public key to be verified: X (X1, X2, X3, or X4 depending on context)
- o ZKP ephemeral public key: V
- o ZKP signature: r

The ZKP is verified as follows:

1. An integer representation of a hash (see Section 7.2) is generated and assigned to h:

   h = int(SHA-256(G, V, X, ID)) mod n

2. A check point is generated and assigned to V':

   V'= X*h + G*r

3. The points V' and V are compared. If equal then the ZKP verifies, otherwise it does not verify.

7.5. ServerKeyExchange Calculations

The structure ECJPAKEKeyKP conveys the public key and associated Schnorr ZKP for Xs.

7.5.1. Public Key Generation

For Xs, the value for the public key part X of the ECJPAKEKeyKP structure is generated as follows:

The inputs are:

- o Public keys: X1, X2 and X3
- o Private key: x4
- o Shared secret: s (integer format, see Section 7.3)
- o Order of the base point: n

The public key of the key pair is calculated as follows:

1. A new generator is generated and assigned to GB:

   GB = X1 + X2 + X3

2. A private key is generated and assigned to xs:

   xs = x4*s mod n

3. A public key associated with xs is generated and assigned to Xs:

   Xs = GB*xs

4. Xs is assigned to the public key part X of the ECJPAKEKeyKP structure.

7.5.2. Schnorr ZKP Generation

For Xs, the values for the ZKP part zkp.V and zkp.r of the ECJPAKEKeyKP structure are generated as follows:

The inputs are:

- New generator: GB

- Order of the base point: n

- Identity of originator: IDs

- Key pair to provide a ZKP of: (Xs,xs) (public key: Xs, private key: xs)

The ZKP is generated as follows:

1. A random integer in the range 1 to n-1 is assigned to ephemeral private key v.

2. An ephemeral public key associated with v is generated and assigned to V:

   V = GB*v

3. An integer representation of a hash (see Section 7.2) is generated and assigned to h:

   h = int(SHA-256(GB, V, Xs, IDs)) mod n

4. A signature is generated and assigned to r:

```
r = v - xs*h mod n
```

5. V and r are assigned to the ZKP part zkp.V and zkp.r of the ECJPAKEKeyKP structure respectively.

7.5.3. Schnorr ZKP Verification

For Xs, the ECJPAKEKeyKP structure is verified as follows:

The inputs are:

- New generator: GB
- Order of the base point: n
- Identity of originator: IDs
- Public key to be verified: Xs
- ZKP ephemeral public key: V
- ZKP signature: r

The ZKP is verified as follows:

1. An integer representation of a hash (see Section 7.2) is generated and assigned to h:

   ```
   h = int(SHA-256(GB, V, Xs, IDs)) mod n
   ```

2. A check point is generated and assigned to V':

   ```
   V'= X*h + GB*r
   ```

3. The points V' and V are compared. If equal then the ZKP verifies, otherwise it does not verify.

7.6. ClientKeyExchange Calculations

The structure ECJPAKEKeyKP conveys the public key and associated Schnorr ZKP for Xc.

7.6.1. Public Key Generation

For Xc, the value for the public key part X of the ECJPAKEKeyKP structure is generated as follows:

The inputs are:

- o Public keys: X1, X3 and X4

- o Private key: x2

- o Shared secret: s (integer format, see Section 7.3)

- o Order of the base point: n

The public key of the key pair is calculated as follows:

1. A new generator is generated and assigned to GA:

   GA = X1 + X3 + X4

2. A private key is generated and assigned to xc:

   xc = x2*s mod n

3. A public key associated with xs is generated and assigned to Xc:

   Xc = GA*xc

4. Xc is assigned to the public key part X of the ECJPAKEKeyKP structure.

7.6.2. Schnorr ZKP Generation

For Xc, the values for the ZKP part zkp.V and zkp.r of the ECJPAKEKeyKP structure are generated as follows:

The inputs are:

- o New generator: GA

- o Order of the base point: n

- o Identity of originator: IDc

- o Key pair to provide a ZKP of: (Xc,xc) (public key: Xc, private key: xc)

The ZKP is generated as follows:

1. A random integer in the range 1 to n-1 is assigned to ephemeral private key v.

2. An ephemeral public key associated with v is generated and assigned to V:

V = GA*v

3. An integer representation of a hash (see Section 7.2) is generated and assigned to h:

   h = int(SHA-256(GA, V, Xc, IDc)) mod n

4. A signature is generated and assigned to r:

   r = v - xc*h mod n

5. V and r are assigned to the ZKP part zkp.V and zkp.r of the ECJPAKEKeyKP structure respectively.

7.6.3. Schnorr ZKP Verification

For Xc, the ECJPAKEKeyKP structure is verified as follows:

The inputs are:

- o New generator: GA
- o Order of the base point: n
- o Identity of originator: IDc
- o Public key to be verified: Xc
- o ZKP ephemeral public key: V
- o ZKP signature: r

The ZKP is verified as follows:

1. An integer representation of a hash (see Section 7.2) is generated and assigned to h:

   h = int(SHA-256(GA, V, Xc, IDc)) mod n

2. A check point is generated and assigned to V':

   V'= X*h + GA*r

3. The points V' and V are compared. If equal then the ZKP verifies, otherwise it does not verify.

7.7. Premaster Secret Generation

The TLS-ECJ-PAKE handshake relies on the generation of identical premaster secrets at the client and server to verify the key establishment. The use of the protected Finished messages is therefore used for key confirmation purposes and to verify the handshake.

7.7.1. Server Premaster Secret Generation

The inputs are:

- Public key of the client: Xc
- Public key: X2
- Private key: x4
- Shared secret: s (integer format, see Section 7.3)

The premaster secret is generated as follows:

1. Compute PMSK:

   PMSK = (Xc - X2*x4*s)*x4

2. Compute PMS:

   PMS = SHA-256(str(32, X coordinate of PMSK))

3. The master secret and key expansion is generated according to Section 8.1 and Section 6.3 of [RFC5246].

7.7.2. Client Premaster Secret Generation

The inputs are:

- Public key of the server: Xs
- Public key: X4
- Private key: x2
- Shared secret: s (integer format, see Section 7.3)

The premaster secret is generated as follows:

1. Compute PMSK:

```
PMSK = (Xs - X4*x2*s)*x2
```

2. Compute PMS:

```
PMS = SHA-256(str(32, X coordinate of PMSK))
```

3. The master secret and key expansion is generated according to Section 8.1 and Section 6.3 of [RFC5246].

8. Acknowledgements

The authors would like to thank Sorin Aliciuc, Richard Kelsey, Maurizio Nanni, Manuel Pegourie-Gonnard and Martin Turon for their helpful comments and assistance.

9. IANA Considerations

9.1. Transport Layer Security (TLS) Parameters

9.1.1. TLS Cipher Suite Registry

IANA is requested to add the following entries in the TLS Cipher Suite Registry:

```
TLS_ECJPAKE_WITH_AES_128_CCM = {0xTBD, 0xTBD}
TLS_ECJPAKE_WITH_AES_256_CCM = {0xTBD, 0xTBD}
TLS_ECJPAKE_WITH_AES_128_CCM_8 = {0xTBD, 0xTBD}
TLS_ECJPAKE_WITH_AES_256_CCM_8 = {0xTBD, 0xTBD}
```

9.2. Transport Layer Security (TLS) Extensions

9.2.1. ExtensionType Values

IANA is requested to add the following entries in the ExtensionType Values:

```
ecjpake_key_kp_pair = TBD
```

10. Security Considerations

10.1. Security Proof

An independent study that proves security of J-PAKE in a model with algebraic adversaries and random oracles can be found in [ABM15].

10.2. Counter Reuse

The cipher suites described in this document are AES-CCM-based AEAD cipher suites, therefore the security considerations for counter reuse described in [RFC6655] also apply to these cipher suites.

10.3. Password

The password forming the basis of the shared secret SHOULD be distributed in a secure out-of-band channel. In the specific case of [THREAD], this is achieved by the user enabling the use of the password only through a commissioning session where the user is in control of adding details of devices they wish to add to the Thread network.

10.4. Rate Limiting

An attacker could attempt to engage repeatedly with a ECJ-PAKE server in an attempt to guess the password. Servers SHOULD take steps to ensure the opportunity for repeated contact is limited.

10.5. Usage Restrictions

The cipher suites described in this document have primarily been developed to enable authentication and authorization for network access for IoT devices, as described in [THREAD]. It is therefore RECOMMENDED that the use of these cipher suite is restricted to similar uses and SHOULD NOT be used in conjunction with web servers and web browsers unless consideration is given to secure entry of passwords in a browser.

11. References

11.1. Normative References

[CCM] National Institute of Standards and Technology, "Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality", SP 800-38C, May 2004, <http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C_updated-July20_2007.pdf>.

[SEC1] Standards for Efficient Cryptography Group, "Standards for Efficient Cryptography: SEC 1: Elliptic Curve Cryptography", SECG SEC1-v2, May 2004, <http://www.secg.org/sec1-v2.pdf>.

[THREAD] Thread Group, "Thread Commissioning", July 2015, <http://threadgroup.org/Portals/0/documents/whitepapers/Thread%20Commissioning%20white%20paper_v2_public.pdf>.

[HR08] Hao, F. and P. Ryan, "Password Authenticated Key Exchange by Juggling", 16th Workshop on Security Protocols (SPW'08), May 2008, <http://grouper.ieee.org/groups/1363/Research/contributions/hao-ryan-2008.pdf>.

[ABM15] Abdalla, M., Benhamouda, F., and P. MacKenzie, "Security of the J-PAKE Password-Authenticated Key Exchange Protocol", IEEE Symposium on Security and Privacy, May 2015, <https://www.normalesup.org/~fbenhamo/files/publications/SP_AbdBenMac15.pdf>.

[J-PAKE] Hao, F., "J-PAKE: Password Authenticated Key Exchange by Juggling", draft-hao-jpake-03 (work in progress), February 2016, <http://tools.ietf.org/id/draft-hao-jpake-03.txt>.

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

[RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, DOI 10.17487/RFC4492, May 2006, <http://www.rfc-editor.org/info/rfc4492>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <http://www.rfc-editor.org/info/rfc5116>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc5246>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <http://www.rfc-editor.org/info/rfc6066>.

[RFC6655] McGrew, D. and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)", RFC 6655, DOI 10.17487/RFC6655, July 2012, <http://www.rfc-editor.org/info/rfc6655>.

[RFC7251] McGrew, D., Bailey, D., Campagna, M., and R. Dugal, "AES-CCM Elliptic Curve Cryptography (ECC) Cipher Suites for TLS", RFC 7251, DOI 10.17487/RFC7251, June 2014, <http://www.rfc-editor.org/info/rfc7251>.

11.2. Informative References

[RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, DOI 10.17487/RFC6090, February 2011, <http://www.rfc-editor.org/info/rfc6090>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <http://www.rfc-editor.org/info/rfc6347>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <http://www.rfc-editor.org/info/rfc7252>.

Authors' Addresses

Robert Cragie
ARM Ltd.
110 Fulbourn Road
Cambridge CB1 9NJ
UK

Email: robert.cragie@arm.com

Feng Hao
Newcastle University (UK)
Claremont Tower, School of Computing Science, Newcastle University
Newcastle upon Tyne NE1 7RU
UK

Email: feng.hao@ncl.ac.uk