## Types of Devices and Network Structure

There are three roles a participant can act as:

• Border Router

• Commissioner

• End Device

## Definitions:

### Border Router

Any device capable of forwarding between a Mesh Network and a non-Mesh Network. The Border Router also serves as an interface point for the Commissioner when the Commissioner is on a non-Mesh Network. The Border Router requires a Mesh interface.

### Commissioner

The authentication server for new Devices and the authorizer for providing the Network Credentials they require to join the network. A device capable of being elected as a Commissioner is called a Commissioner Candidate. Devices without mesh network interfaces may perform this role. This device may be, for example, a cell phone or a server in the cloud, and typically provides the interface by which a human administrator manages joining a new device to the Thread Network.

### Commissioner Candidate

A device that is capable of becoming the Commissioner, and either intends or is currently petitioning the Border Router to

become the Commissioner, but has not yet been formally assigned the role of Commissioner.

### Commissioning Credential

A human-scaled passphrase for use in authenticating that a device may petition to become the Commissioner of the network. This passphrase is encoded in utf-8 format and has a length of six (6) bytes minimum and 255 bytes maximum. This credential can be thought of as the network administrator password for a Thread Network.

The first device in a network, typically the Border Router, MUST be out-of-band commissioned to inject the correct user generated Commissioning Credential into the Thread Network, or provide a known default Commissioning Credential to be changed later. This credential is used to derive an enhanced key using key stretching called the PSKc (Pre-Shared Key for the Commissioner) which is used to establish the Commissioner Session.

### Commissioning Key (PSKc)

The PSKc (Pre-Shared Key for the Commissioner) is a derived key based on the Commissioning Credential which is used to establish the Commissioner Session. The PSKc is used as a passphrase input to a PAKE cipher suite, such as J-PAKE [draft-hao-jpake-03], to establish a secure session between the Commissioner and Border Agent. All devices in a mesh Network will store the PSKc.

### End Device

A Mesh device that transmits and receives unicast traffic only with a Parent device.

### Frame counter

A value that is incremented with each new secured message and used to detect replayed messages.

### Joiner

The device to be added by a human administrator to a commissioned mesh Network. This role requires a mesh Network interface to operate.

### Joining Device Credential

A human-scaled passphrase for use in authenticating that a new Joiner device is the correct one. This passphrase is used in conjunction with a PAKE cipher suite to establish a secure session between the Commissioner and the Joiner. The Commissioner and the Joiner typically share the Joining Device Credential using some out-of-band mechanism such as scanning a barcode or entering a serial number from the Joiner device label. This credential when specifically encoded in binary is referred to as the PSKd (Pre-Shared Key for the Device). A Joining Device Credential is encoded as uppercase alphanumeric characters (base32-thread: 0-9,A-Y excluding I,O,Q, and Z for readability) with a minimum length of 6 such characters and a maximum length of 32 such characters

### Network Credentials

All the security and operational parameters required for a device to be part of a Thread Network as contained in the Joiner Entrust message, including the Network Master Key, Mesh Local Prefix, etc. All devices on a Thread Network store the Network Credentials, including the Leader, Routers, and End Devices

### Server

A device that provides one or more services to a network. Border Routers are considered servers that provide a routing service.

The main goal of the protocol is to authenticate that the correct device is joined, and then to securely transfer the Network Credentials to that device over an unsecured link (secured only by DTLS).

## Commissioner

### Commissioner Authentication

The Commissioner Candidate, after having discovered the Mesh Network of interest, would then securely connect to the Thread Network using the Commissioning Credential.

The Commissioner Authentication step establishes a secured client/server connection between the Commissioner Candidate and a Border Router via DTLS (Datagram Transport Layer Security, [RFC 6347] or optionally TLS (Transport Layer Security, [RFC 5246]. This secured session is hereby known as the Commissioning Session.

The credential used to establish the Commissioning Session is known as the PSKc and is generated as follows.

### Derivation of PSKc

Because the PSKc is stored by all devices in the Thread Network, a key-stretching algorithm is used to protect the actual user-generated passphrase, or Commissioning Credential, that is used to derive it. A Commissioner will locally generate the PSKc, and only authenticate with the Thread Network using that PSKc. This provides some protection for a user that may reuse the same password across various services. The key stretching uses the [PBKDF2](https://ru.wikipedia.org/wiki/PBKDF2) [RFC 2898], AES-CMAC-PRF-128 [RFC 4615].

PSKc = PBKDF2(

PRF=AES-CMAC-PRF-128,

P=<Commissioning Credential>,

S=”Mesh” || <Network Name>,

c=16384, dkLen=16)

Where the **Network Name** is all characters up to the maximum size or null terminator, and 27 || is the concatenation operator.

### Commissioner Registration

After the Commissioner Authentication is successful, the Commissioner Candidate registers its identity with its Border Router by sending message via the secure Commissioning Session.

Message contains a human-readable string that identifies the Commissioner.

Failed attempts by a Commissioner to establish its authority with a Mesh Network shall be rate-limited to RETRY\_DELAY second between attempts up to RETRY\_COUNT attempts and ATTEMPT\_DELAY seconds between such groups of attempts.

### Commissioning

The commissioning process begins when an off-network or on-mesh commissioning device becomes an active Commissioner for a Mesh Network Partition. There can be only one active Commissioner at a time for a Mesh Network, so the commissioning device petitions the Border Router to become the active Commissioner.

An off-network commissioning device, typically a mobile phone, will generally initiate commissioning by discovering the Mesh Network. The device then sets up a DTLS connection with the Border Agent using the network's Commissioning key (PSKc).

Once active, the Commissioner can enable joining on the network and optionally provide data that indicates the ID of the devices expected to join.

A joining device (Joiner) can then communicate with the Commissioner via a relaying protocol through Border Router to which the Commissioner is connected. The Joiner and the Commissioner then perform a DTLS handshake using the Joiner’s passphrase, which the Commissioner MUST have received out-of-band, typically by being entered by a user. Once the handshake is complete, the shared secret it produces is used to pass the Mesh Network’s commissioning material from the Commissioner (Router in case of OT) to the joining device.

Sending master key from Commissioner is easier than from Router because: 1) protocol is simpler; 2) Commissioner is allowed to obtain master key from Border Router.

The Commissioner may also query and set Mesh Network parameters, such as the Mesh Network name and security configuration (i.e. master key).

The high level view of commissioning-joining procedure is depicted on the UML sequence diagram below:



## Border Router. Role in Commissioning

Border Routers play a role in commissioning of new devices to the Mesh Network. A Border Router implements a commissioning Border Agent which acts as a relay for messages between external Commissioners (such as a mobile device) and devices on the interior network.

Router MUST be specifically provisioned for immunity to Denial of Service (DoS) situations that may be based on generation of multiple requests in their radio vicinity.

Where a random number is required as specified in this document, a device MUST implement a True Random Number Generator (TRNG) where the entropy source can be proven to be from a verifiably random source, e.g., from thermal noise, avalanche noise, or atmospheric noise.

## Joiner

### Joining a Thread Network

To join a mesh Network a device must first acquire the following commissioning information:

*thrKeyRotation*

Key rotation period in hours. The minimum key rotation period SHALL be 1 hour and a key rotation period of 0 SHALL NOT be allowed.

*thrKeySequenceCounter*

Monotonically increasing counter used for key rotation.

• Master Key

• Commissioning Key (PSKc)

• Network Name (used in derivation of PSKc - provides some protection for a user that may reuse the same password across various services.)

These may be obtained either through in-band commissioning or through out-of-band commissioning. Out-of-band commissioning is not defined in this specification, other than a requirement that it be at least as secure as in-band commissioning.

### Joiner Authentication

The Joiner, throughout the Joining process, sends DTLS handshake messages via unsecured channel to a Router. A Router relays these messages on to Commissioner. Any Router has no knowledge of the content of these Joiner messages, and only relay data on the unsecured channel.

## Regular traffic

### Network-wide Key

The Mesh Network is protected with a network-wide key (master key), from which derived key are used (in OT at the MAC sub-layers to protect the 802.15.4 MAC data frames). This is an elementary form of security used to prevent casual eavesdropping and targeted disruption of the Network from outsiders without knowledge of the network-wide key. Because it is a network-wide key, compromise of any Mesh device could potentially reveal the key. (As a result, the network-wide key is not typically used as the only form of protection within the OT Network). From the point of view of joining the network, it is used to discriminate between an authenticated and authorized device and the joining device (in its initial state).

Because the joining device is untrusted at the point of joining, it is common practice to enforce some sort of policing mechanism to ensure the joining device can be verified and at the same time limit the effect of rogue devices attempting to join the Mesh Network. In a Mesh Network, this requires the joining device to identify a trusted device and to communicate solely in a point-to-point fashion with this trusted device. The trusted device polices any traffic from the joining device and forwards it to the commissioning device in a controlled manner to allow the authentication protocol (DTLS handshake) to execute.

In the usual case where the commissioning device is not in direct communication with the joining device, the trusted device MUST relay the DTLS handshake with the commissioning device through one relay agent - Router. A relay protocol provides encapsulation of the DTLS handshake and relaying of the DTLS handshake from the joining device all the way to the commissioning device in a simple manner.

### Master Key

Key used to derive security material for MAC protection in OT. For simplicity in this paper key rotation period and sequence counter (that participate in OT in key derivation) are not used because this would require complex protocol for propagation of counter to network devices – synchronization of counter.

***Not used in this version*** *but used in OT:*

*thrKeyRotation*

Key rotation period in hours. The minimum key rotation period SHALL be 1 hour and a key rotation period of 0 SHALL NOT be allowed.

*thrKeySequenceCounter*

Monotonically increasing counter used for key rotation.

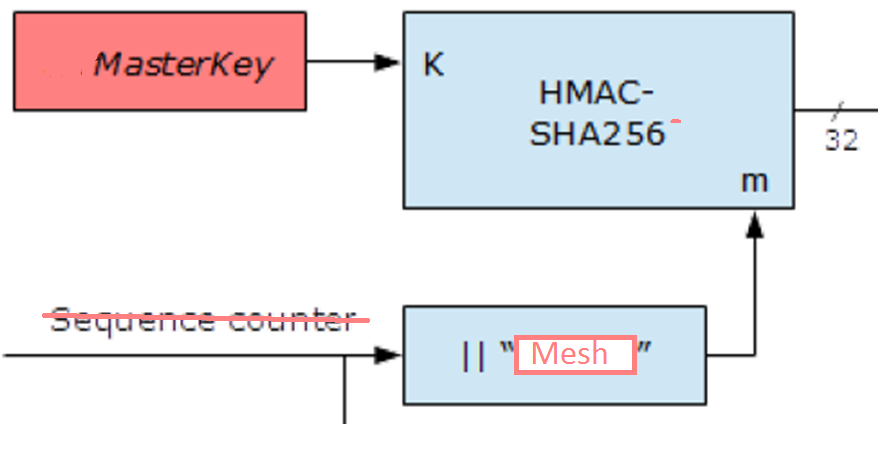
### **Key Generation**

**(**may be omitted if master key is always equal to necessary length for AES cipher**)**

Each Thread node receives the Master Key when joining and assigns it to the *MasterKey* attribute. In OT - *MasterKey* is used in conjunction with a sequence counter to derive two separate keys for use by the MAC sublayer and MLE. **This version** uses key derivation to make **derived master key (DMK) of particular length** (key stretching hash algorithm is used if master key may be of variable length. If master key is equal to length of AES key – then derivation may be omitted. Master key should be generated by Router (not user) therefore master key may be always of the same length). The use of Hashed Message Authentication Mode with the SHA-256 algorithm (HMAC-14 SHA256) as the keyed hash function produces an output of 32 bytes [RFC 6234]. Therefore, DMK, used for encryption of regular traffic maybe up to 32 bytes. The 32-byte DMK key is generated as follows:

HMAC-SHA256(MasterKey, ~~sequence counter ||~~ “Mesh”)

where || is the concatenation operator and “Mesh” is the byte sequence.

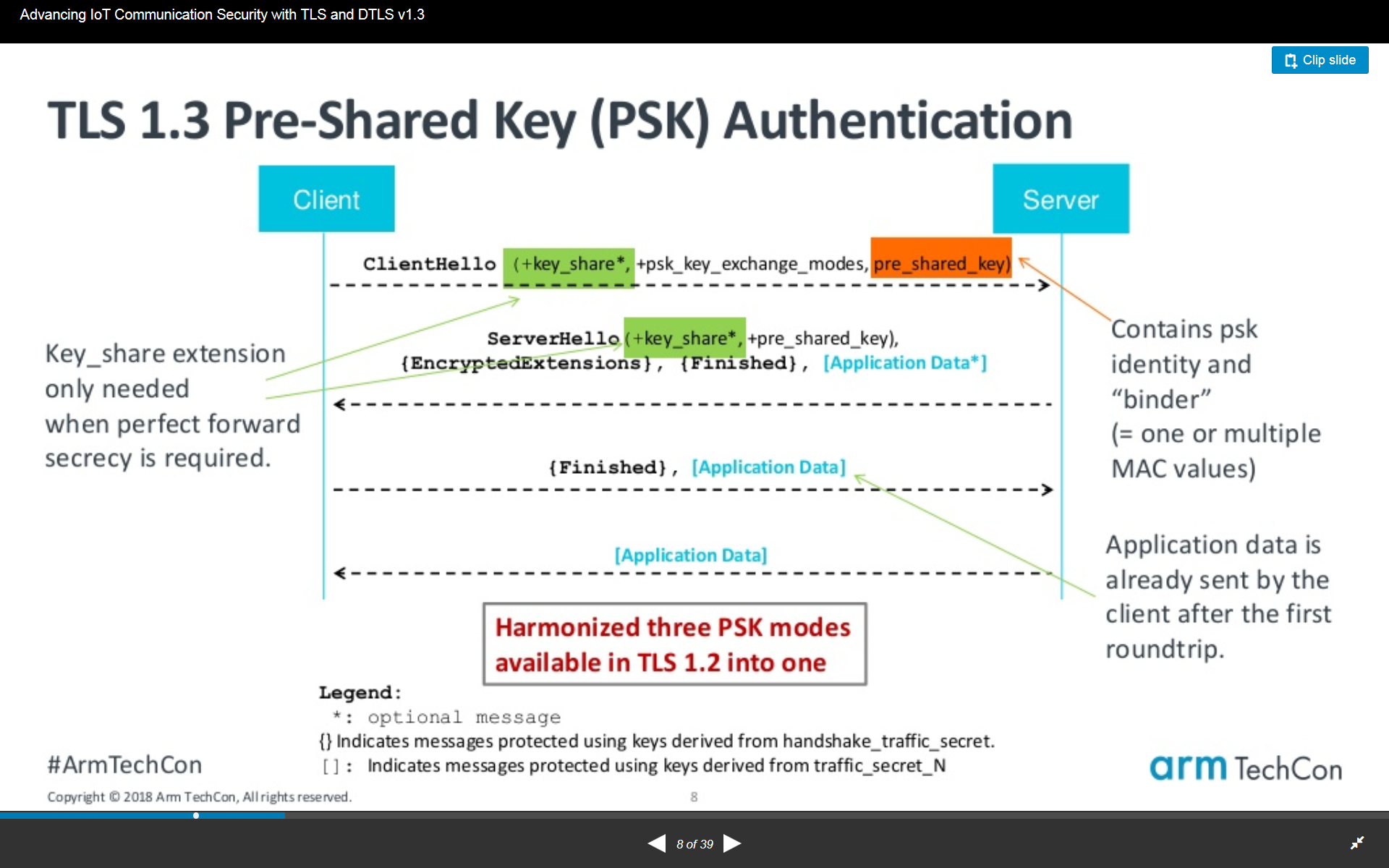


### Symmetric encryption for confidentiality of regular traffic.

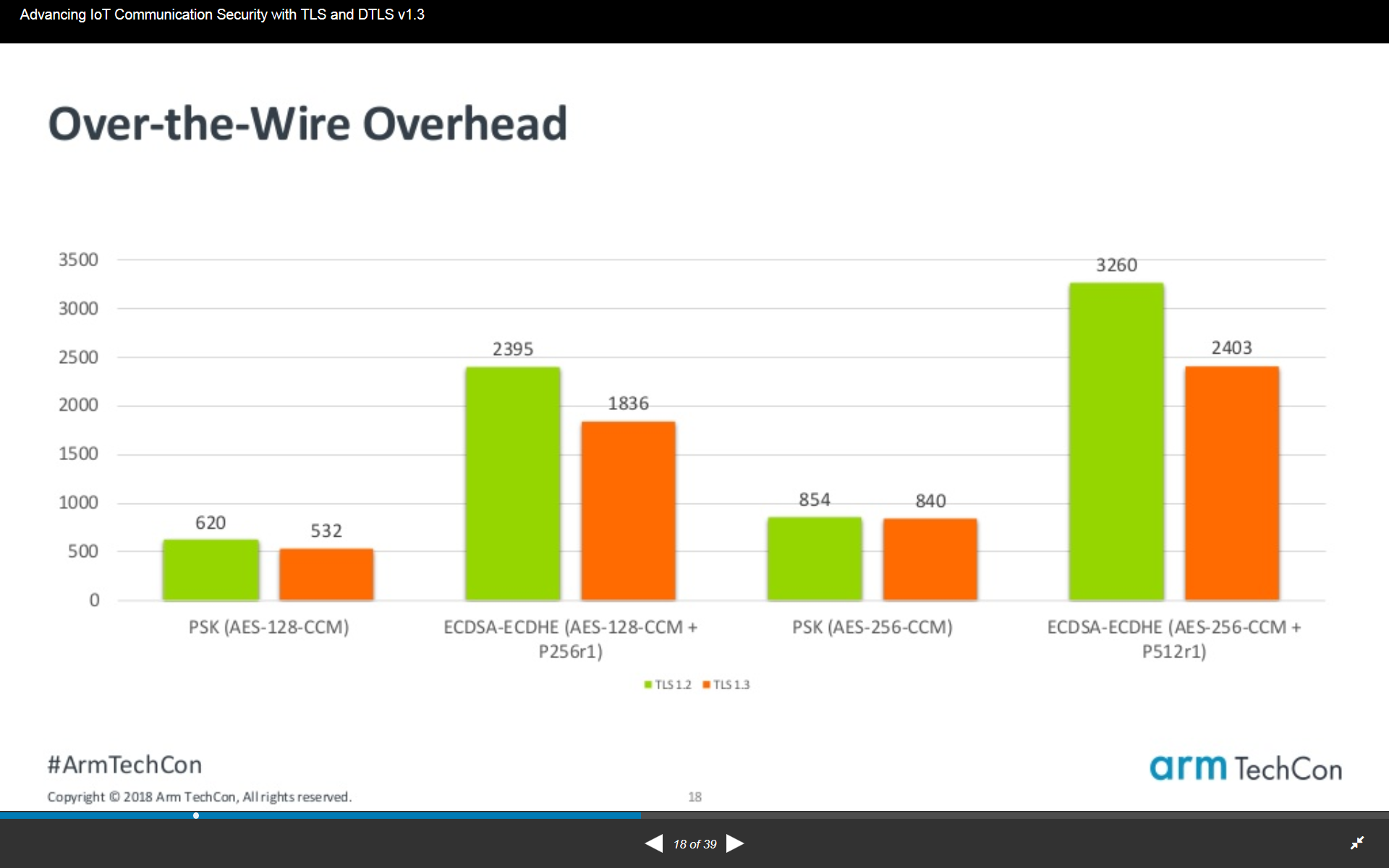
Since devices share key there are two options to encrypt regular traffic (between joined devices or Router):

1. [DTLS PSK ciphersuite](https://en.wikipedia.org/wiki/TLS-PSK) - DTLS based on pre-shared symmetric key
2. or direct use of AES - to minimize traffic between sleepy devices (battery should be alive for long period)

DTLS PSK has less flights (handshakes) than DTLS-JPAKE until client can send application data (compare with J-Pake diagram below):



Below is comparison of overheads between DTLS-PSK (that doesn’t use asymmetric algorithms) and predominantly asymmetric ECDSA-ECDHE(not used here but is close to DTLS-JPAKE):



With confidence it’s possible to state that usage of DTLS-PSK is safe but requires additional overheads each time device starts data session – data transfer. Threats of direct usage of AES may be mitigated. Say using AES in CBC or CCM or another mode and changing IV each session). These will add message authentication and will prevent replay attacks that is possible if using AES in ECB mode. There is no sense to reinvent DTLS protocol manually and not implementing all the steps with the same overheads as DTLS has. More certainty requires deeper investigation. If DTLS-PSK is affordable considering bandwidth limits – better to use it as out of the box solution. Example of usage is presented in delivered prototype. Prototype mbedTLS solution also contains AES usage example – aescrypt2 project.

Code size:



## Authentication and Key Agreement

The Mesh Network is designed to provide a high level of security during the process of adding devices to the network and during normal network operation. During the joining process, devices MUST BE specifically authenticated and authorized, and required to complete a key agreement mechanism. Once on the network, all communications are secured with a network key (DMK). The fundamental security used during the joining for authentication and key agreement is an elliptic curve variant of J-PAKE (EC-JPAKE), using the NIST P-256 elliptic curve. J-PAKE is a PAKE (Password Authenticated Key Exchange) with “juggling” (hence the “J”). It essentially uses elliptic curve Diffie-Hellmann for key agreement and Schnorr signatures as a NIZK (Non-Interactive Zero-Knowledge) proof mechanism to authenticate two peers and to establish a shared secret between them based on the passphrase. Authorization is implied through the entry of the specific passphrase into the commissioning device. A TLS (Transport Layer Security) handshake is used for EC-JPAKE, which can be used in both TLS and DTLS (Datagram Transport Layer Security). DTLS is a variant of TLS with additional fields in the records to make it suitable for use over an unreliable datagram secured with a network key.

### DTLS

Commissioning and joining use the same secure algorithm of secure session: authentication, exchange of encrypted data – DTLS.

MbedTLS is the library that implements DTLS and proposed in this paper.

MbedTLS implements J-PAKE authentication.

Internet of Things (IoT) nodes participating in the implementation of IoT applications will need standard methods to authenticate each other and secure their communications. However, the use of certificates and public key infrastructure (PKI) technologies, widely employed in the Internet, is not straightforward in this context because of the limitations of the involved entities – restricted devices.

### J-PAKE

In [cryptography](https://en.wikipedia.org/wiki/Cryptography), a password-authenticated key agreement method is an interactive method for two or more parties to establish cryptographic keys based on one or more party's knowledge of a [password](https://en.wikipedia.org/wiki/Password).



An important property is that an eavesdropper or man in the middle cannot obtain enough information to be able to brute force guess a password without further interactions with the parties for each (few) guesses. This means that strong security can be obtained using weak passwords. User should not type long cryptographic key – it’s sufficient to type reasonably short password because number of guesses is limited – at most one password can be tested per interaction – only online dictionary attack works (limit of attempts make such attack rather impossible)

**Password authenticated key exchange** (PAKE) is where two or more parties, based only on their knowledge of a password, establish a cryptographic key using an exchange of messages, such that an unauthorized party (one who controls the communication channel but does not possess the password) cannot participate in the method and is constrained as much as possible from brute force guessing the password.

The **Password Authenticated Key Exchange by Juggling** (or J-PAKE) is a [password-authenticated key agreement](https://en.wikipedia.org/wiki/Password-authenticated_key_agreement) protocol

allows two parties to establish private and authenticated communication solely based on their shared (low-entropy) password without requiring a [Public Key Infrastructure](https://en.wikipedia.org/wiki/Public_Key_Infrastructure). It provides [mutual authentication](https://en.wikipedia.org/wiki/Mutual_authentication) to the key exchange, a feature that is lacking in the [Diffie–Hellman key exchange](https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange) protocol.

Elliptic curve cryptography is used in J-PAKE implementation in mbedTLS. Also

Schnorr ZKP (Zero Knowledge Proof) is part of J-PAKE.

The two-round J-PAKE protocol is completely symmetric. This helps significantly simplify the security analysis. For example, the proof that one party does not leak any password information in the data exchange must hold true for the other party based on the symmetry. This reduces the number of the needed security proofs by half.

In practice, it is more likely to implement J-PAKE in three flows since one party shall normally take the initiative. This shouldn’t be an issue because this step is performed once per joining.

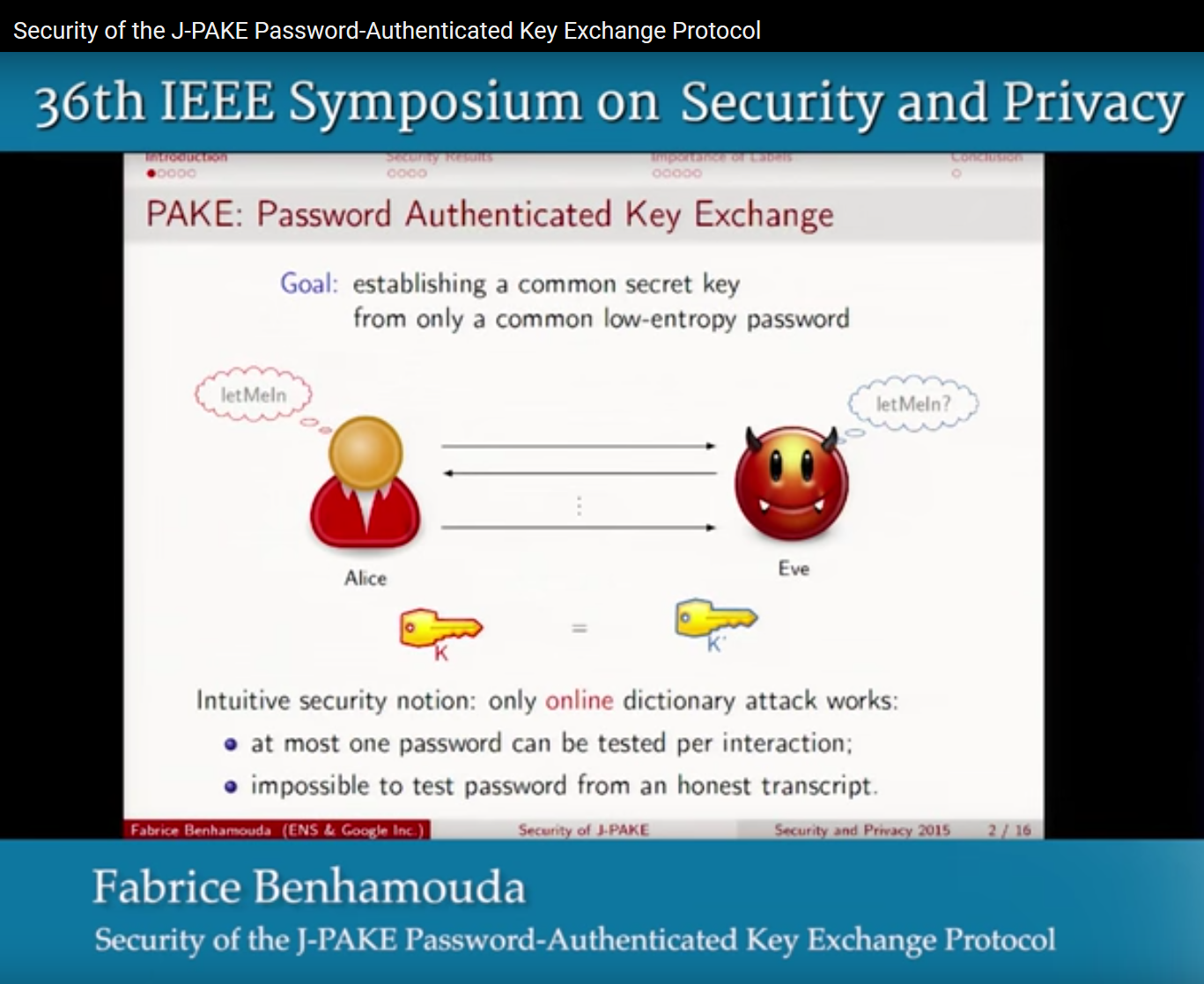
#### Security properties of J-PAKE

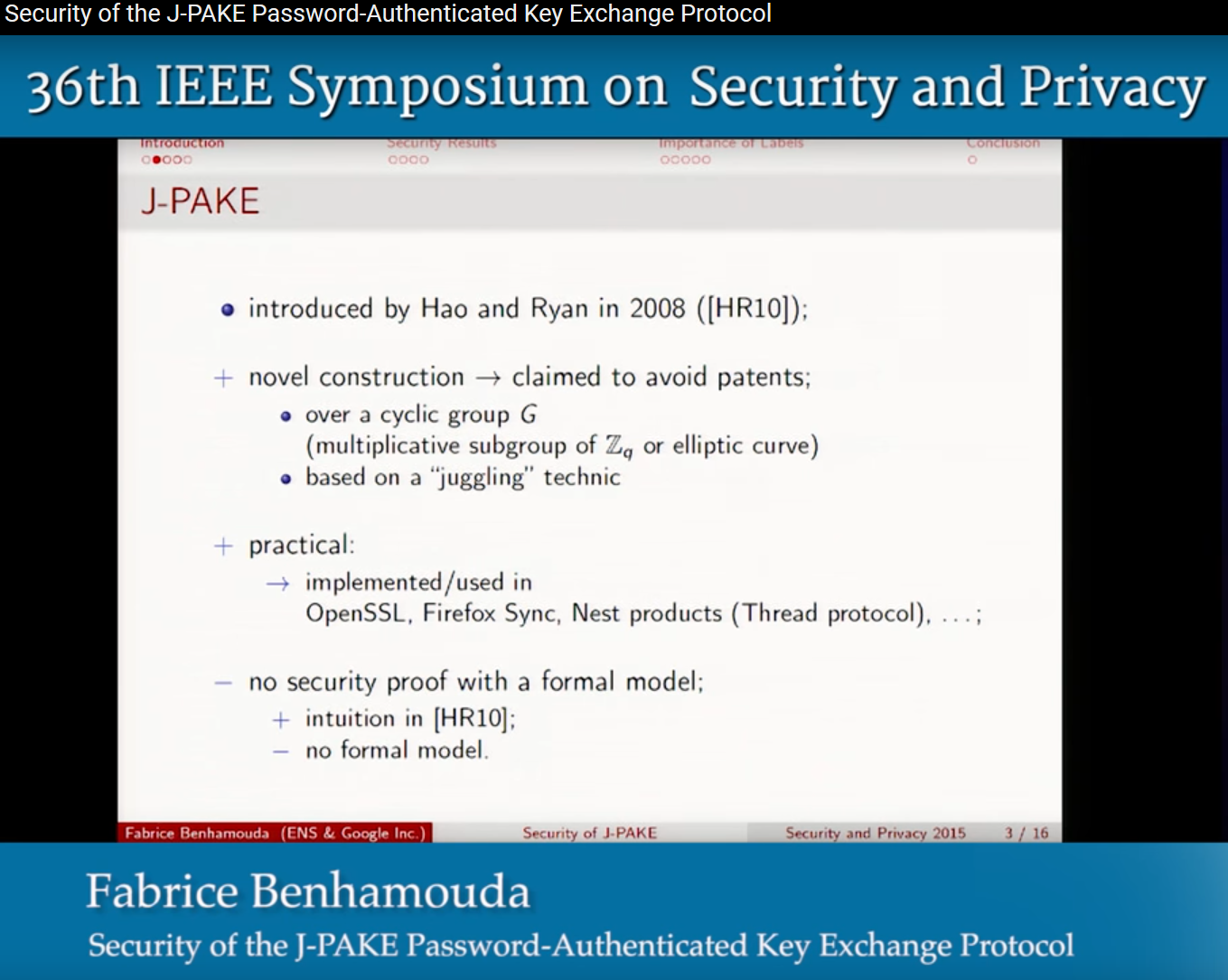
Given that the underlying Schnorr non-interactive [zero-knowledge proof](https://en.wikipedia.org/wiki/Zero-knowledge_proof) is secure, the J-PAKE protocol is proved to satisfy the following properties:[[3]](https://en.wikipedia.org/wiki/Password_Authenticated_Key_Exchange_by_Juggling" \l "cite_note-3)

1. Off-line dictionary attack resistance - It does not leak any password verification information to a passive/active attacker. Basically, this implies that the attacker is not able to obtain data with which to perform an offline dictionary attack, in which the attacker would run through a dictionary of possible passwords offline, checking each one for consistency with the data.
2. [Forward secrecy](https://en.wikipedia.org/wiki/Forward_secrecy) - It produces session keys that remain secure even when the password is later disclosed.
3. Known-key security - It prevents a disclosed session key from affecting the security of other sessions.
4. On-line dictionary attack resistance - It limits an active attacker to test only one password per protocol execution.

In 2015, [Abdalla, Benhamouda and MacKenzie](https://www.normalesup.org/~fbenhamo/files/publications/SP_AbdBenMac15.pdf) conducted an independent formal analysis of J-PAKE to prove its security in a random oracle model assuming algebraic adversaries.[[4]](http://www.normalesup.org/~fbenhamo/files/publications/SP_AbdBenMac15.pdf)

this work provides proof and states: J-PAKE is an efficient password-authenticated key exchange protocol that is included in the OpenSSL library and is currently being used in practice. We present **the first proof of security for this protocol in a well-known and accepted model** for authenticated key-exchange, that incorporates offline password guessing, concurrent sessions, forward secrecy, server compromise, and loss of session keys. This proof relies on the Decision Square Diffie-Hellman assumption, as well as a strong security assumption for the non-interactive zero-knowledge (NIZK) proofs in the protocol (specifically, simulation-sound extractability). We show that the Schnorr proof-of-knowledge protocol, which was recommended for the J-PAKE protocol, satisfies this strong security assumption in a model with algebraic adversaries and random oracles, and extend the full J-PAKE proof of security to this model. Finally, we show that by modifying the recommended labels in the Schnorr protocol used in J-PAKE, we can achieve a security proof for J-PAKE with atighter security reduction.





These slides are taken from [IEEE Symposium on Security and Privacy](https://www.youtube.com/channel/UC6pXMS7qre9GZW7A7FVM90Q)

These authors emphasize the potential vulnerability of J-PAKE: label of device should be unique, otherwise attacker can redirect session to another participant if two participants share the same passphrase. It seems that derivation of PSKc passphrase (not direct usage of Commissioner Credentials) is introduces to eliminate this vulnerability – PSKc will be different in different networks even user uses the same passphrase in two separate networks - and users often use the same passwords in different domains.

#### Standardization of J-PAKE

J-PAKE has been included in ISO/IEC 11770-4 (2017) as an international standard.[[11]](https://en.wikipedia.org/wiki/Password_Authenticated_Key_Exchange_by_Juggling#cite_note-11) It is also published in [RFC 8236](https://tools.ietf.org/html/rfc8236)

The J-PAKE protocol [24] is a PAKE protocol that has started seeing wide usage. It is included as an optional protocolin the OpenSSL library (enabled using a config parameter during install, see directory crypto/jpake), and has been used in various products, such as Firefox Sync and Nest products.

Its popularity is likely due not only to its easy description, straightforward implementation, and practical efficiency, but also because it seems to be based on a different paradigm than previous practical PAKE protocols. Those protocols basically used the password to obfuscate the inputs to a key exchange (e.g., the gx and gy values in a Diffie-Hellman key exchange), whereas the J-PAKE protocol uses ephemeral values like a standard Diffie-Hellman key exchange, but then combines them with a password in an extra round, such that use of the correct password makes certain randomization factors vanish. The J-PAKE designers call this the “juggling” technique and attribute the first use of the idea to Hao and Zielinski.

Due to its novelty, the designers of J-PAKE claim that it might be useful in avoiding patent issues around other PAKE protocols.

### Comparison with other authentication systems.

Other well-known auth systems we considered:

* Kerberos (symmetric encryption)
* PKI (asymmetric)

#### Kerberos

[Kerberos for IoT](https://kit.mit.edu/sites/default/files/documents/Kerberos_Internet_of%20Things.pdf): the **Pros**:

- Well understood protocol (cf. Needham-Schroeder);

- Symmetric-key approach suits constrained devices;

– Long-term keys can be installed by device manufacturer;

– Symmetric key operations cheaper/faster;

– Kerberos flows can be optimized for IoT devices;

- Integration with directories a well-trodden path;

- Open source code (20+ years) –MIT code written in C – several generations of coders –Active dev community

Kerberos in IoT: the Cons

- works excellent in corporate environment;

- RFC4120 will put you to sleep...

- No initial key distribution protocol –Use PKINIT (RFC6112) or similar

- Good C programmers hard to come-by

#### PKI

PKI requires expensive infrastructure: signing server, revocation services (CRL or OCSP), sometimes CMP service. There is a big concern with PKI because of the management of the invalidation of certificates.

Some risks:

* System Downtime Due to Certificate Expiries
* Management by Spreadsheet

One can only look at the news of IoT devices becoming inoperative, such as the Wink Hub issue in April of 2015, to see that maintaining expiring certificates can cause major problems in the deployment of IoT devices.

PKI for IoT is referred in articles:

1. “[PKIoT: A public key infrastructure for the Internet of Things](https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.3681)”
2. [“Technology Insight for X.509 Certificate Management”](https://www.gartner.com/en/documents/3969998/technology-insight-for-x-509-certificate-management)
3. “New research project aims to “shrink” public key infrastructure (PKI) [technology](https://www.nexusgroup.com/public-key-infrastructure-pki-secure-iot/) to secure the IoT”

##### Possible solution with PKI

Imagine that we are going to use simplified system based on public key certificates. Suppose we simplified PKI system – gave certificate with very long expiration period, suppose that we are confident that private key of root certificate is not possible to compromise and therefore we dropped support of revocation of certificates – that is we do not support CRL server or OCSP server. Supporting of revocation means that validation of certificate requires connection to internet. Connection from Border Router is possible – Router can validate devices certificate. Device unlikely will be able to authenticate Router or Commissioner – this will require connection to revocation server from device. In OT device authenticate Commissioner – because it will not establish DTLS joining session with Commissioner if Commissioner doesn’t know PSKd passphrase. Therefore J-PAKE has advantage here – using simple mechanism mutual authentication is achieved. We need to inject certificate to each device. Signed with root certificate. Can we join device automatically (without human intervention)? If device will found two mesh networks within range what network should it choose? User may have 2 networks or second network in range may be in owned by his neighbor. Therefore some human intervention is needed: user should accept joining of new device at least pressing OK button. Some Wi-Fi routers work in this simplified way: new device connects to selected network and user presses dedicated button to authorize connection. But again – network is selected from new device. How to select network from headless device… Maybe device should send requests to all fount routers and again user should perform some action to authorize it. Without specifying ID of device some attacks are possible – malicious user may try to connect its own device and he will obtain master key on his device. To avoid such kind of attacks user should enter not even ID of device – that maybe easily cloned but information derived from public key from device or some passphrase/pin from device label. Again passphrase like in J-Pake. Therefore PKI doesn’t give sufficient advantage but has drawbacks: it’s more expensive in deployment and maintenance; requires sending significant volumes of data with certificate chain (compact certificates are proposed in [PKIoT: A public key infrastructure for the Internet of Things](https://doi.org/10.1002/ett.3681)) and makes difficult to authenticate Commissioner by device.

## MbedTLS implementation of DTLS J-Pake (MBEDTLS\_TLS\_ECJPAKE\_WITH\_AES\_128\_CCM\_8)

mbed TLS is used in embedded software. mbed TLS offers an [SSL library](https://tls.mbed.org/ssl-library) with an intuitive API and readable source code, so you can actually understand what the code does. Also the mbed TLS modules are as loosely coupled as possible and written in the portable C language. This allows you to use the parts you need, without having to include the total library.

mbed TLS is available as open source under the Apache 2.0 license or the GPL 2.0 license. The Apache 2.0 license enables you to use mbed TLS in both open source and closed source projects.

MbedTLS has small memory and storage footprint. To reduce mbed TLS memory and storage footprint:

<https://tls.mbed.org/kb/how-to/reduce-mbedtls-memory-and-storage-footprint>

### Licensing

<https://tls.mbed.org/kb/licensing/using-mbedtls-in-a-non-gpl-project>

### Examples in mbedTLS prototype solution.

There are several cumulative projects in solution (pair of client and server).

<https://github.com/vnovakovsky/mbedtlsESP32.git>

Repository is private – Ownership is transferred to email kerryshih2@gmail.com

Each new project can what previous can plus new feature. For instance, mmf server can communicate with socket and with shared memory client. AIO server can communicate additionally with pipe client. Also AIO (all in one) server can communicate with JPAKE and with PSK client (in socket mode only).

Following table shows what client can communicate to what server considering DTLS mode and kind of messaging system:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **dtls\_server** | **dtls\_psk\_server** | **dtls\_mmf\_server** | **dtls\_aio\_server** |
| **dtls\_client** | socket |  | socket | socket |
| **dtls\_mmf\_client** |  |  | socket, mmf | socket, mmf |
| **dtls\_psk\_client** |  | socket | socket | socket |
| **dtls\_aio\_client** |  |  | socket, mmf | socket, mmf, pipe |
|  |  |  |  |  |

Defines of messaging system maybe changed in dtls\_config.props

Channel interface was introduced. Please find diagram below. Interface hides specific messaging implementation.Logs are in the github\mbedtls\visualc\VS2010\tests\ directoryThis directory contains batch files for launching demo projects – copy them to exe folder.



As base for client and server code original examples from mbedTLS solution were taken. They were modified as less as possible. Logic of server and client is similar between projects server logic is depicted in diagram below. Socket implementation were moved from library to user application code and encapsulated in Channel interface the same way as shared memory (mmf) and pipe implementation were. This was done to identify and isolate transport (messaging system) code and make all messaging systems equal.



Messaging system (this text has been sent in one of the report already and may be skipped – bear in mind that Channel interface was added later and some statements below may be slightly different)

Replacement of transport consists of 3 main steps. Developer need to do following:

1. Implement you read and write functions and send them as callback functions to mbedtls\_ssl\_set\_bio.
2. Implement functions equivalent to socket functions: connect, accept, close – these functions are called from application code via Channel interface
3. Introduce synchronization for sending messages between client and server.

AIO implementation uses socket or shared memory or pipes for message exchange between client and server. Shared memory uses windows implementation: “memory mapped file” windows kernel object that uses swap file to avoid creating file in file system.(See file mbedtls\programs\ssl\mmf\_communication.c/mmf\_communication.h)

To switch between socket and shared memory macro is defined: USE\_SHARED\_MEMORY

Turning it on: shared memory will be used. Turning off: sockets are used.

Step 1.

Read/write callback are in mbedtls\programs\ssl\mmf\_communication.c

Those callbacks should be passed to following function (called from application code: dtls\_mmf\_server.c/ dtls\_mmf\_client.c) – parameters in grey are not used in current implementation. Description is copied from original functions and not all of parameters are used in this simple implementation. But may be important in other messaging systems.

mbedtls\_ssl\_set\_bio( &ssl, &client\_fd,

mbedtls\_net\_send\_mmf,

mbedtls\_net\_recv\_mmf,

mbedtls\_net\_recv\_timeout\_mmf);

/\*\*

\* \brief **Read** at most 'len' characters. If no error occurs,

\* the actual amount read is returned.

\*

\* \param ctx Socket

\* \param buf The buffer to write to

\* \param len Maximum length of the buffer

\*

\* \return the number of bytes received,

\* or a non-zero error code; with a non-blocking socket,

\* MBEDTLS\_ERR\_SSL\_WANT\_READ indicates read() would block.

\*/

int mbedtls\_net\_recv\_mmf(void\* ctx, unsigned char\* buf, size\_t len);

/\*\*

\* \brief **Write** at most 'len' characters. If no error occurs,

\* the actual amount read is returned.

\*

\* \param ctx Socket

\* \param buf The buffer to read from

\* \param len The length of the buffer

\*

\* \return the number of bytes sent,

\* or a non-zero error code; with a non-blocking socket,

\* MBEDTLS\_ERR\_SSL\_WANT\_WRITE indicates write() would block.

\*/

int mbedtls\_net\_send\_mmf(void\* ctx, const unsigned char\* buf, size\_t len);

/\*\*

\* \brief **Read** at most 'len' characters, blocking for at most

\* 'timeout' seconds. If no error occurs, the actual amount

\* read is returned.

\*

\* \param ctx Socket

\* \param buf The buffer to write to

\* \param len Maximum length of the buffer

\* \param timeout Maximum number of milliseconds to wait for data

\* 0 means no timeout (wait forever)

\*

\* \return the number of bytes received,

\* or a non-zero error code:

\* MBEDTLS\_ERR\_SSL\_TIMEOUT if the operation timed out,

\* MBEDTLS\_ERR\_SSL\_WANT\_READ if interrupted by a signal.

\*

\* \note This function will block (until data becomes available or

\* timeout is reached) even if the socket is set to

\* non-blocking. Handling timeouts with non-blocking reads

\* requires a different strategy.

\*/

int mbedtls\_net\_recv\_timeout\_mmf(void\* ctx, unsigned char\* buf, size\_t len,

uint32\_t timeout);

Step 2.

Functions below are specific to shared memory implementation. They creates/uses shared memory – between two processes – one process writes buffer of mmf – another reads from buffer. First int value (4 bytes) contains length of record.

|  |  |
| --- | --- |
| HANDLE create\_mmf(); | Create memory mapped file (mmf) - shared memory |
| PVOID map\_mmf (HANDLE hFileMap); | Maps file to process address space – in the begging of program |
| void unmap\_mmf (PVOID pView); | Unmap file in the end of program |
| void write\_mmf (PVOID pView, void\* buf, int nbytes); | Writes to mmf – called from callback above |
| int read\_mmf (PVOID pView, void\* buf); | Read from mmf – called from callback above |
| void close\_mmf (HANDLE hFileMap); | Closes mmf in the end. |

Step 3. Synchronization.

Following events are used:

ghConnectedEvent - signals to server that connection from client accepted.

ghSignalAboutEvent – signals that message is written – set in write operation

ghWaitForEvent – blocks read operation until message arrives.

|  |  |
| --- | --- |
| void create\_event\_mmf(enum PointOfView pointOfView); | Creates all 3 events. PointOfView meams point of view: server or client because the same event is treated differently – event set in write op of client is used for waiting for message in server. And vice versa. |
|  |  |
| void accept\_connection\_mmf(); | Server blocks on this operation waiting for connection from client |
| void connect\_mmf(); | Establishes connection with server by setting ghConnectedEvent to signaled state |
| void close\_connection\_mmf(); | Close all connections by resetting ghConnectedEvent to non-signaled state |

To verify/ test messaging implementation SysInternals [tcpview](https://docs.microsoft.com/en-us/sysinternals/downloads/tcpview) is used (or netstat) and following can be done:

1. Test socket version by undefining USE\_SHARED\_MEMORY . and rebuild. Run dtls\_mmf\_server.bat and dtls\_mmf\_commissioner\_J1.bat and observe that TCPView shows connection on port 4433. Output will show that protocol works.
2. Define USE\_SHARED\_MEMORY and rebuild. Repeat the same as in 1). Output will show additional records for mmf read and write functions and will show that protocol works – message sent from client and received back. TCPView will do not show connection because data is sent via shared memory.