

# AN12132

## A71CH for secure connection to OEM cloud

Rev. 1.1 — 7 March 2018  
464211

Application note  
COMPANY PUBLIC

### Document information

Info	Content
<b>Keywords</b>	Security IC, IoT, PSP, Cloud authentication, Secure authentication
<b>Abstract</b>	This document describes how the A71CH security IC can be used to establish a secure connection with an OEM cloud.



**Revision history**

Rev	Date	Description
1.0	20180219	Initial version
1.1	20180302	Updated sections 3.2, 4.3.1 and 4.3.2

**Contact information**

For more information, please visit: <http://www.nxp.com>

## 1. Introduction

---

This document describes how the A71CH security IC can be used to establish a secure connection between an IoT device and an OEM cloud. It introduces ECC cryptography and SSL/TLS protocol fundamentals, it describes the mechanisms and credentials involved to create a secure TLS connection between an IoT device and the OEM cloud servers. And finally, for A71CH evaluation and demonstration purposes, it details step by step, how a TLS/SSL based communication can be initiated using A71CH OpenSSL Engine example scripts.

## 2. A71CH overview

---

The A71CH is a ready-to-use solution enabling ease-of-use security for IoT device makers. It is a secure element capable of securely storing and provisioning credentials, securely connecting IoT devices to public or private clouds and performing cryptographic device authentication.

The A71CH solution provides basic security measures protecting the IC against many physical and logical attacks. It can be integrated in various host platforms and operating systems to secure a broad range of applications. In addition, it is complemented by a comprehensive product support package, offering easy design-in with plug & play host application code, easy-to-use development kits, documentation and IC samples for product evaluation.

## 3. Public key infrastructure and ECC fundamentals

---

Security is an essential requirement for any IoT design. Thus, security should not be considered as differentiator option but rather a standard feature for the IoT designers. IoT devices must follow a secure-by-design approach, ensuring protected storage of credentials, device authentication, secure code execution and safe connections to remote servers among others. In this security context, the A71CH security IC is designed specifically to offer protected access to credentials, secure connection to private or public clouds and cryptographic device proof-of-origin verification.

Asymmetric cryptography, also known as public key cryptography, is any cryptographic algorithms based on a pair of keys: a public key and a private key. The private key must be kept secret, while the public key can be shared.

RSA (River, Shamir and Adleman) and Elliptical-Curve Cryptography (ECC) are two of the most widely used asymmetric cryptography algorithms. In the case of ECC cryptography, it is based on the algebraic structure of elliptic curves over finite fields. Therefore, each key pair (public and private key) is generated from a certain elliptical curve.

The digital signature, digital certificates, Elliptic Curve Digital Signature Algorithm (ECDSA) and Elliptic Curve Diffie-Hellman (ECDH) key agreement algorithm are briefly explained in the next sections.

### 3.1 Digital signature

A digital signature is used to guarantee the authenticity, the integrity and non-repudiation of a message. A signing algorithm generates a signature given a message and a private key. A signature verifying algorithm accepts or rejects a message given the public key and the signature.

Fig 1 illustrates an example of digital signature. In this case, the message is signed with the sender private key. The receiver will validate the signature using both the message and the sender public.

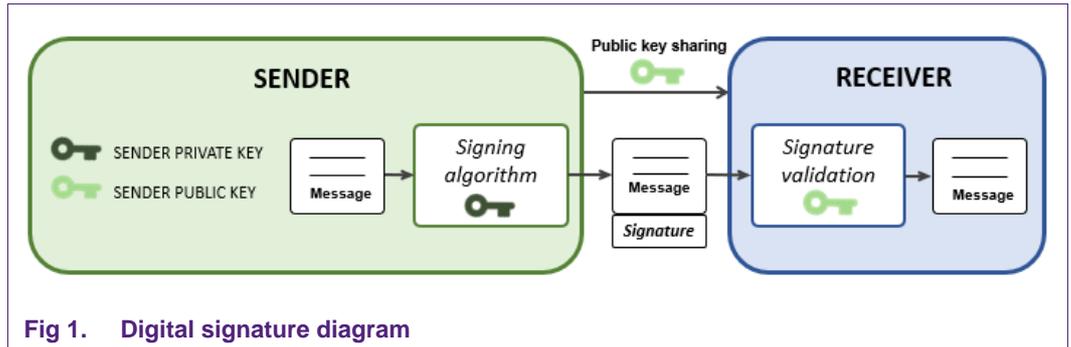


Fig 1. Digital signature diagram

### 3.2 Digital certificate, Certification Authority (CA) and Certificate Signing Request (CSR)

Digital certificates are used to prove the authenticity of shared public keys. Digital certificates are electronic documents that include information about the sender public key, identity of its owner and the signature of a trusted entity that has verified the contents of the certificate, normally called Certificate Authority (CA).

A Certificate Authority (CA) is an entity that issues digital certificates. The CA is trusted by both the certificate sender and the certificate receiver, and it is typically in charge of receiving a Certificate Signing Request (CSR) and generating a new certificate based upon information contained in the CSR and signed with the CA private key.

Therefore, a CSR is a request that contains all the necessary information, e.g., sender public key and relevant information to generate a new digital certificate.

Fig 2 shows digital certificates generation steps. First, the interested device (sender) creates a Certificate Signing Request. The CSR is then sent to the CA and a new digital certificate is created and signed with the CA private key. Also, the basic contents of this new digital certificate are illustrated in the figure.

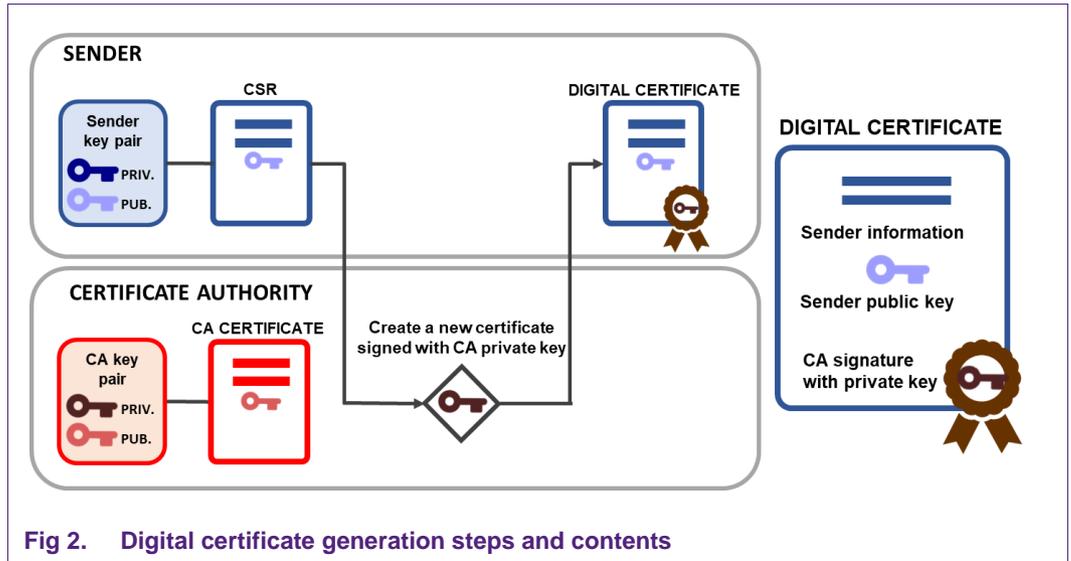


Fig 2. Digital certificate generation steps and contents

### 3.3 Elliptic Curve Digital Signature Algorithm (ECDSA)

The Elliptic Curve Digital Signature Algorithm (ECDSA) algorithm uses ECC to provide a variant of the Digital Signature Algorithm (DSA). A pair of keys (public and private) are generated from an elliptic curve, and these can be used both for signing or verifying a message's signature. Fig 3 illustrates an example of ECDSA application. In this example, the sender device generates a signature with its private key. The signed message is sent together with the sender digital certificate to the receiver. Finally, the receiver retrieves the sender public key from the digital certificate and uses it to validate the signature of the received message.

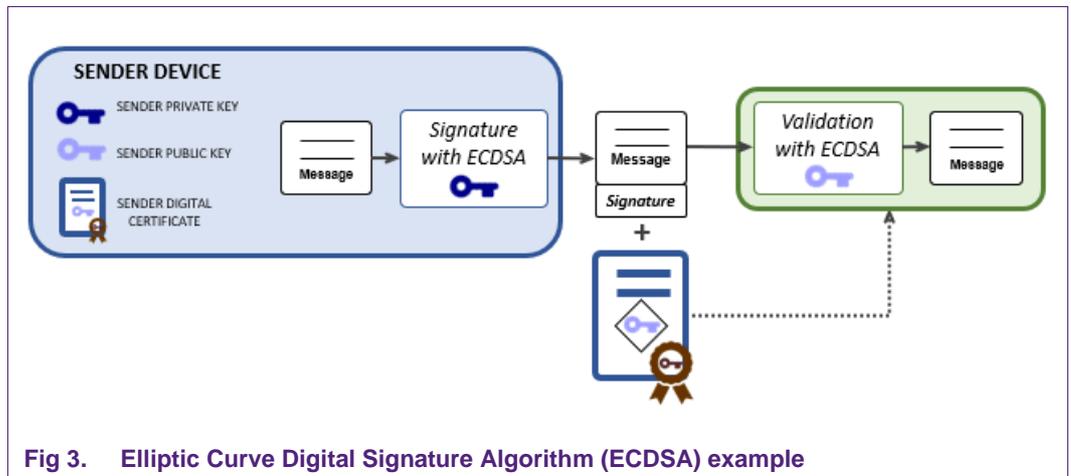


Fig 3. Elliptic Curve Digital Signature Algorithm (ECDSA) example

### 3.4 Elliptic Curve Diffie-Hellman (ECDH)

Elliptic Curve Diffie-Hellman algorithm (ECDH) is a key-agreement protocol. The goal of ECDH is to reach a key agreement between two parties, each having an elliptic-curve key pair generated from the same domain parameters. When the agreement has been reached, a shared secret key, usually referred to as the 'master key', is derived to obtain

session keys. These session keys will be employed to establish a communication using symmetric-key encryption algorithms.

The sender and the receiver have its own elliptical key pair. Both the sender and receiver public keys are shared with each other. In this case, the exchange has been represented with digital certificates. Each party can compute the secret key using their own private key and the public key obtained from the received certificate. Due to the elliptical curve properties and the fact that both key pairs have been generated from the same domain parameters, the computed secret key is the same for both parties. This common secret key will be further used for establishing a communication and encrypt messages based on symmetrical cryptography. Fig 4 illustrates the usage of ECDH for a shared secret key agreement.

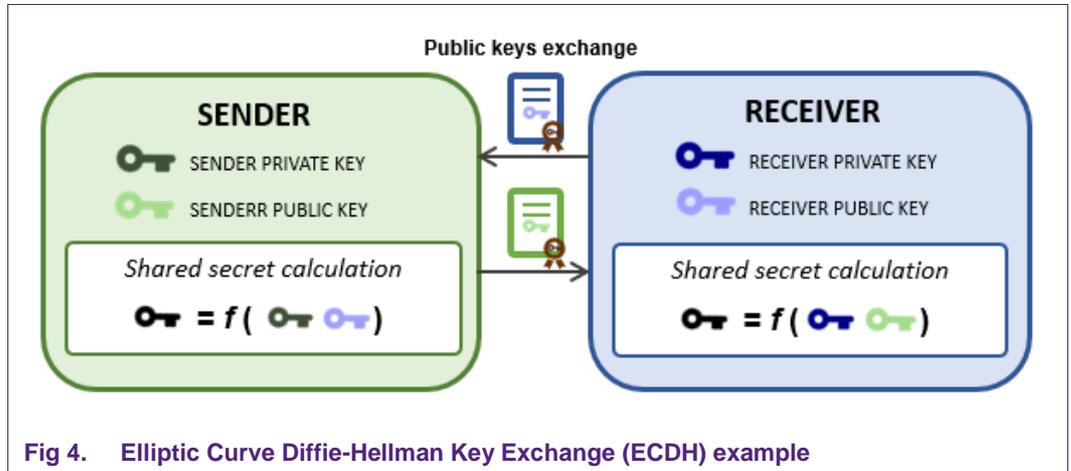


Fig 4. Elliptic Curve Diffie-Hellman Key Exchange (ECDH) example

In the Elliptic-curve Diffie-Hellman Ephemeral (ECDHE) algorithm case, a new elliptical key pair is generated for each key agreement instead of sharing the already existing public keys.

### 3.5 A71CH ECC supported functionality

The A71CH security IC supports the following ECC functionality:

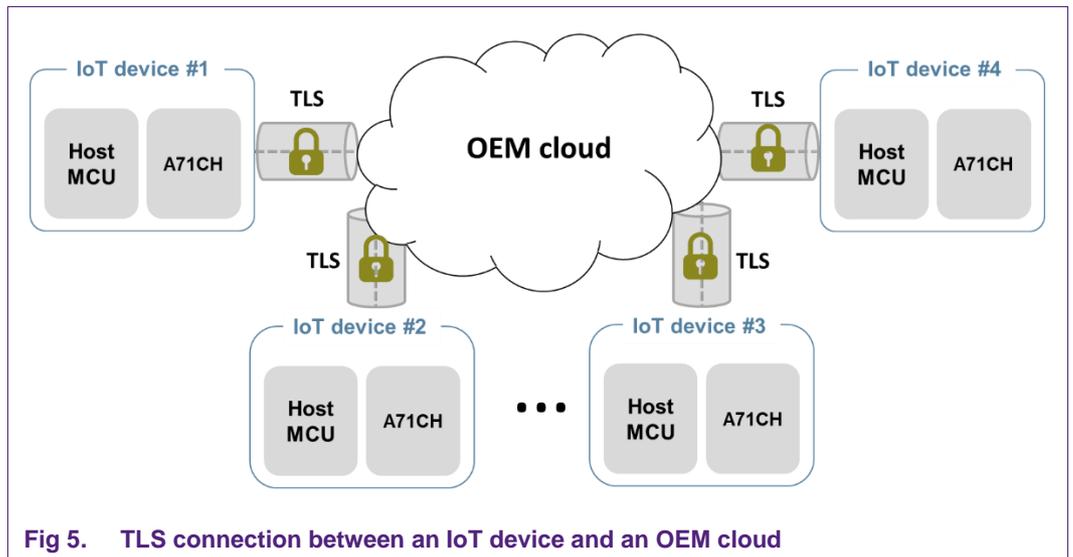
- Signature generation and verification (ECDSA).
- Shared secret calculation using Key Agreement (ECDH or ECDH-E).
- Protected storage, generation, insertion or deletion of key pairs (NIST-P256 elliptical curve).

## 4. A71CH for secure connection to OEM cloud

This section describes the steps and credentials involved so that an IoT device with an A71CH security IC can establish a secure, end-to-end TLS connection with a server in the cloud (OEM cloud).

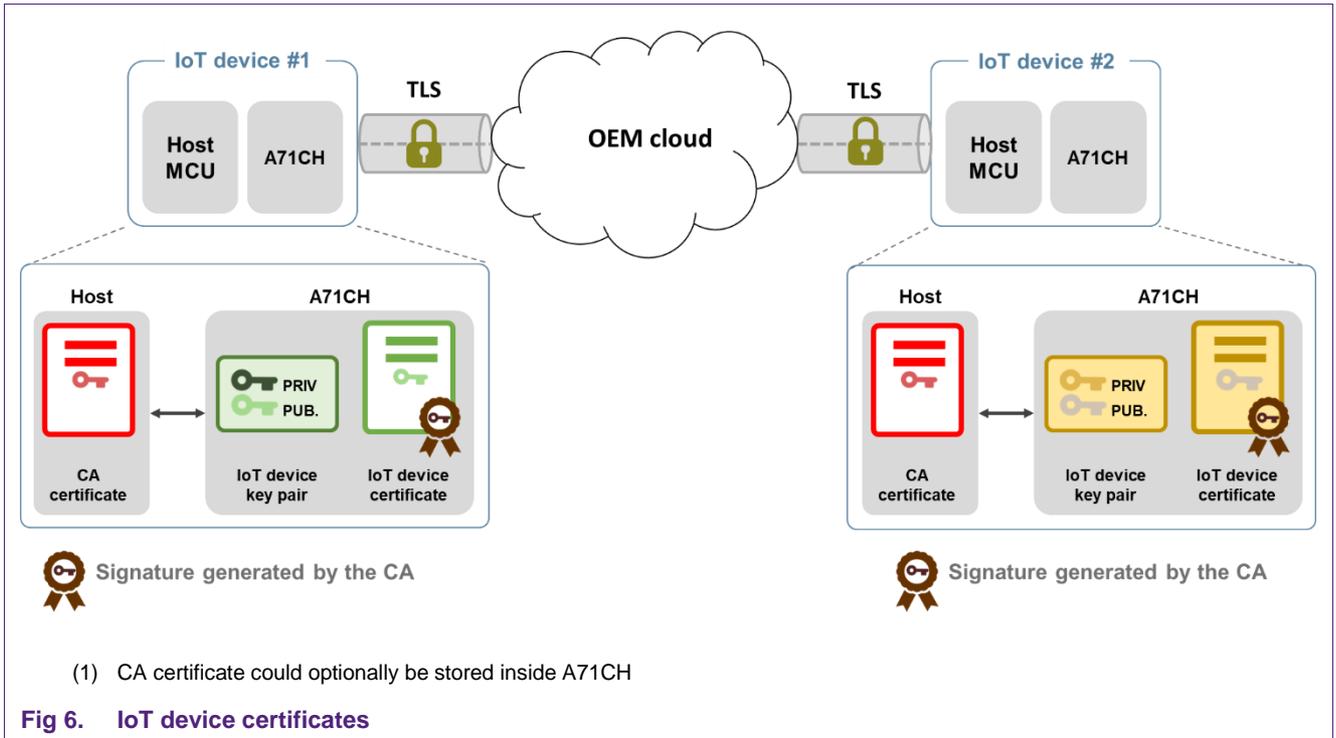
A channel established with TLS protocol guarantees authenticity of the device, confidentiality and integrity in the communication between the IoT device and the OEM server. The credentials required to establish this TLS connection are stored, and never leave, the A71CH security IC.

Fig 5 shows a network composed of an OEM cloud and several IoT devices. Each IoT device features an A71CH security IC and the communication between these and the OEM cloud is secured with TLS protocol.



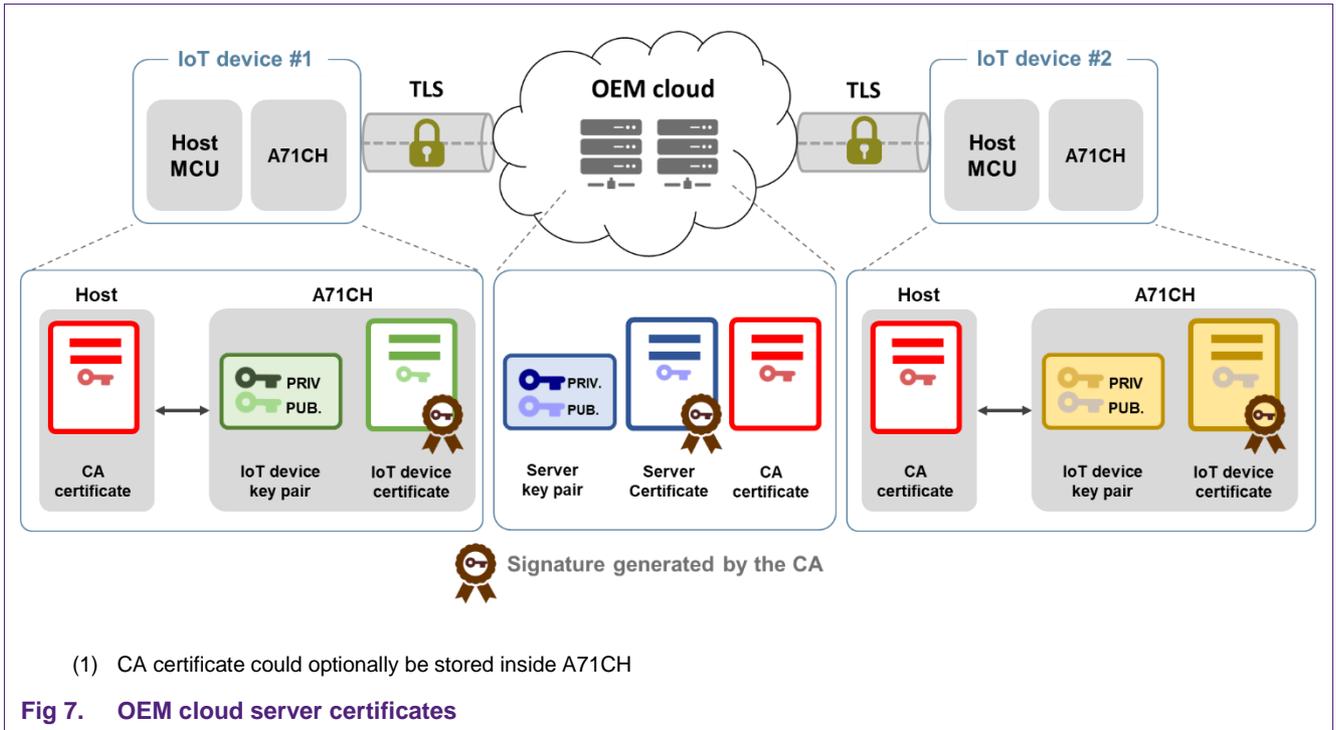
### 4.1 IoT device credentials

Each IoT device stores a unique elliptical key pair (IoT node key pair) and its digital certificate (IoT certificate) signed by a trusted CA. It should also contain the CA certificate or CA public key for a Server authentication. The IoT device key pair and digital certificate will be securely stored in each A71CH respectively. Fig 6 illustrates the contents of each IoT device in the communication between these and the OEM cloud. The contents of the A71CH security IC (IoT key pair and digital certificate) have been painted in different colors to remark that these credentials are unique per device.



### 4.2 OEM cloud server credentials

The OEM server in the cloud stores a unique key pair (Server key pair) and a digital certificate (Server certificate) signed by a trusted CA. The server can either behave as the CA (thus store the self-signed root CA certificate and root CA key pair) or trust in a third-party CA. Fig 7 completes Fig 6 by representing the contents of the OEM cloud.



### 4.3 Transport Layer Security protocol (TLS)

IoT devices own several connectivity features that allow them to exchange data with the cloud. Therefore, the network link between these IoT devices and the cloud or server should be secure. Transport Layer Security protocol (TLS), and its predecessor Secure Sockets Layer (SSL), are cryptographic protocols that provide communications security over unsecure networks. These protocols are created from the necessity of establishing a connection preserving confidentiality, integrity and authenticity.

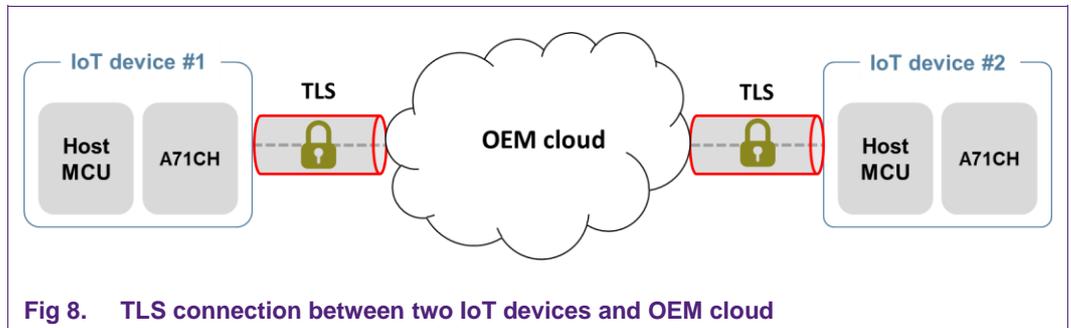
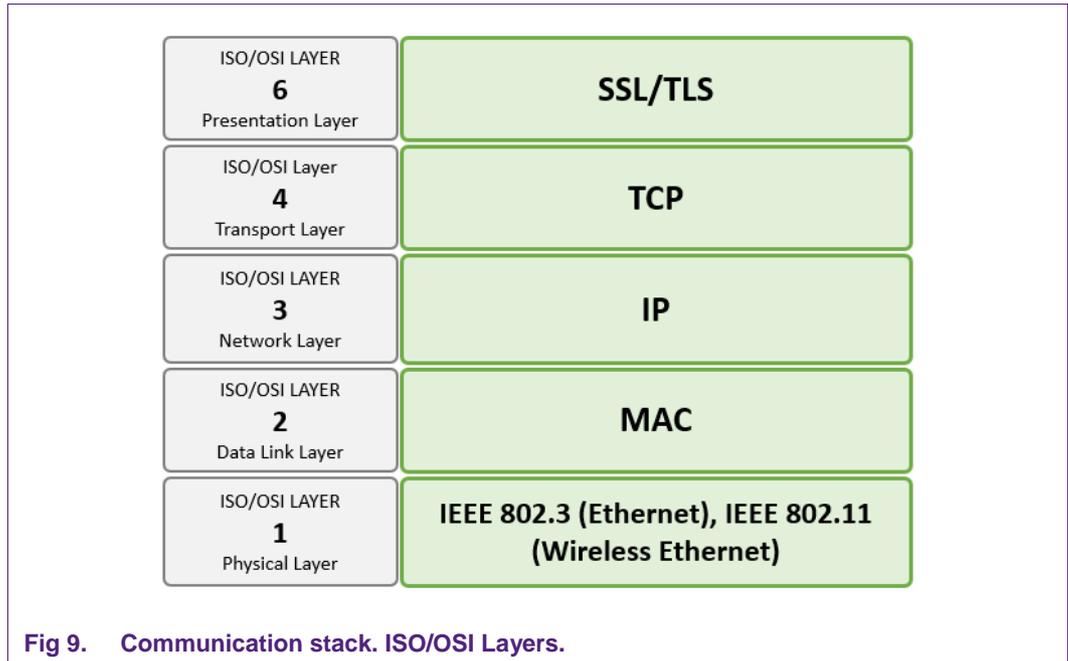


Fig 9 illustrates the protocol stack of a TLS communication over a TCP/IP network. In the well-known ISO/OSI layer architecture, SSL/TLS would belong to the presentation layer in charge of encrypting and securing the entire communication. The transport and network protocol TCP/IP and the medium access control (MAC) would fall in layers from 4 to 2, respectively. Finally, data would be electrically transferred according to ethernet (or wireless ethernet) protocols.



### 4.3.1 Transport Layer Security Handshake protocol

Before the IoT device and the server in the cloud begin exchanging data over TLS, the tunnel encryption must be negotiated. This negotiation is referred to as TLS Handshake. The TLS Handshake Protocol is responsible for the authentication and key exchange necessary to establish or resume secure sessions. When establishing a secure session, the TLS Handshake Protocol manages the following:

- Agree on the TLS protocol version to be used.
- Select cipher suite.
- Authenticate each other by exchanging and validating digital certificates.
- Use asymmetric encryption techniques to generate a shared secret key, which avoids the key distribution problem. SSL or TLS then uses the shared key for the symmetric encryption of messages, which is faster than asymmetric encryption.

The TLS Handshake Protocol involves the following steps:

- Exchange Hello messages to agree on algorithms, exchange random values, and check for resumption.
- Exchange the necessary cryptographic parameters to allow the client and server to agree on a pre-master secret.
- Exchange certifications and cryptographic information to allow the client and server to authenticate themselves.
- Generate a master secret from the pre-master secret and exchanged random value.
- Provide security parameters to the record layer.
- Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.

The A71CH security IC supports the TLS Handshake Protocol version 1.2 with the following options:

- Pre-Shared Key Cipher suites for TLS as described in [RFC4279]: A set of cipher suites for supporting TLS using pre-shared symmetric keys (*TLS\_PSK\_WITH\_xxx*)
- ECDHE\_PSK Cipher suites for TLS as described in [RFC5489]: A set of cipher suites that use a pre-shared key to authenticate an Elliptic Curve Diffie-Hellman exchange with Ephemeral keys (*TLS\_ECDHE\_PSK\_WITH\_xxx*).

The Fig 10 represents an overview of the TLS 1.2 handshake with ECDSA-ECDHE. More information about the TLS 1.2 handshake protocol can be obtained from the standard specifications document [RFC5246] or from [AN\_A71CH\_HOST\_SW].

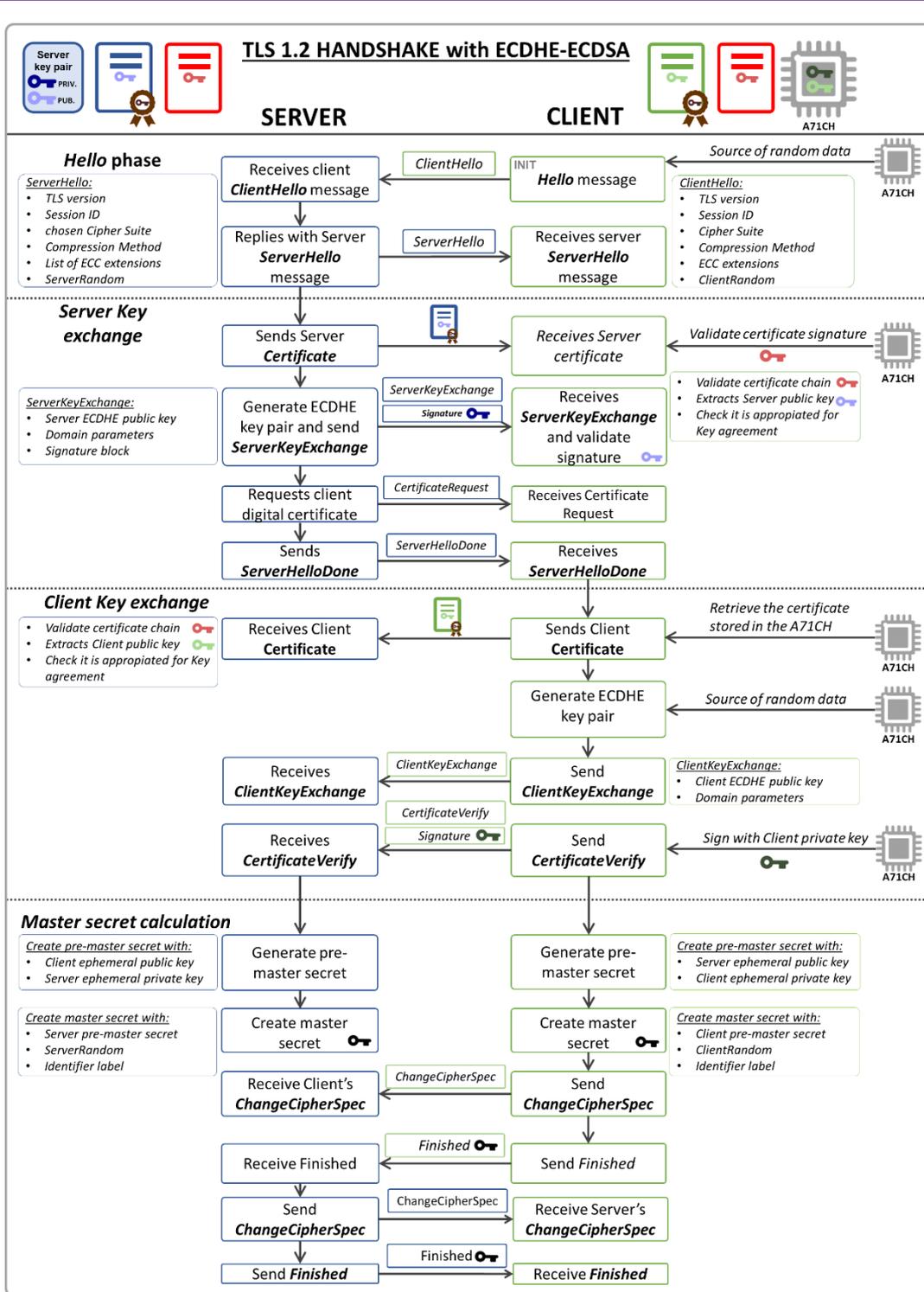


Fig 10. TLS 1.2 Handshake diagram with ECDHE-ECDSA

Once a TLS connection has been established between two devices, all data exchange is secured.

### 4.3.2 Transport Layer Security software libraries

There are several full-featured TLS software libraries that can be used in both server cloud and IoT devices such as openssl, mbedTLS, WolfTLS, etc. OpenSSL [OPEN\_SSL] is an open-source implementation of SSL/TLS protocol. It is written in C language, although there are several wrappers to use this library in other languages. It implements all the cryptography functions needed and it is widely used. Starting with OpenSSL 0.9.6 an ‘Engine interface’ was added allowing support for alternative cryptographic implementations. This Engine interface can be used to interface with external crypto devices as e.g. HW accelerator cards or security ICs like the A71CH.

The OpenSSL toolkit including an A71CH OpenSSL Engine is available as part of the A71CH Host software package [A71CH\_OPENSSL\_ENGINE]. The A71CH OpenSSL Engine gives access to several A71CH features via the A71CH Host Library not natively supported by OpenSSL implementation. In other words, the Engine links the OpenSSL libraries to the A71CH Host API, and overwrites some of the native OpenSSL functions in order to include the use of the A71CH crypto functionality such as sign, verify and key exchange operations or random messages generation, that can be used for instance during the TLS Handshake protocol.

The A7CH OpenSSL Engine is fully compatible with the i.MX6UltraLite embedded platform. Nevertheless, more support will be added in future revisions.

Fig 11 illustrates the Host MCU software architecture. As it can be observed, the software stack is formed by an application that will call OpenSSL functions. Some of these functions will be overwritten by the A71CH OpenSSL Engine, thus the A71CH crypto functionality will be used through the A71CH Host Library over I<sup>2</sup>C.

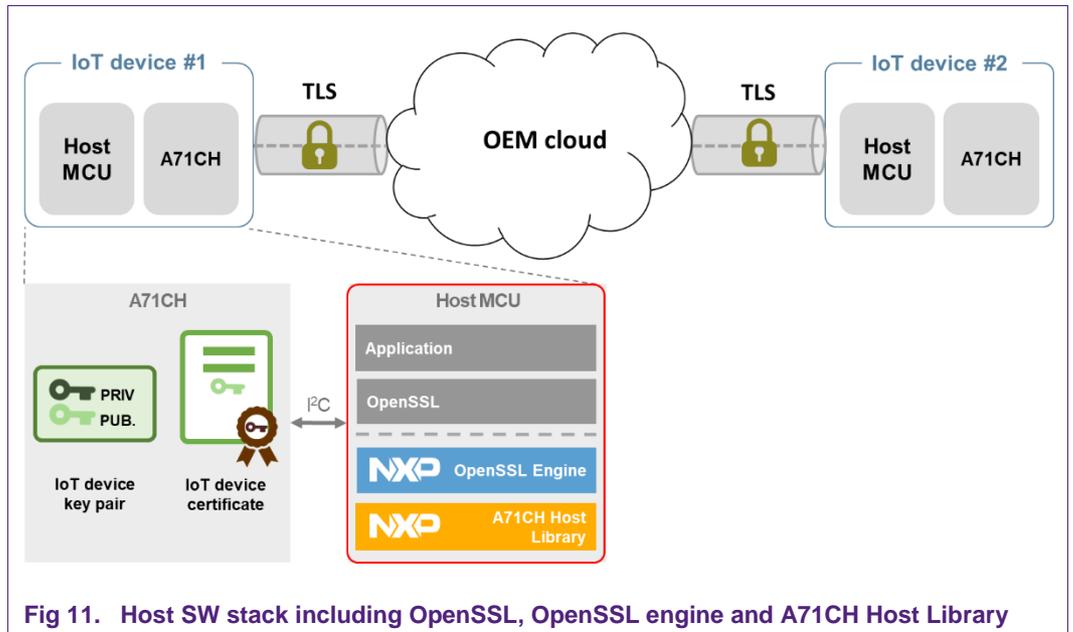


Fig 11. Host SW stack including OpenSSL, OpenSSL engine and A71CH Host Library

## 5. Evaluating A71CH for secure connection to OEM cloud

---

For easy product evaluation, the A71CH product support package (PSP) includes a group of scripts demonstrating the establishment of a communication secured with TLS between two end-points.

In this section it is explained, step by step, how to execute the A71CH OpenSSL Engine example scripts that establish a TLS connection between a development PC acting as a server and the i.MX6UltraLite embedded platform acting as an IoT device client.

---

**Note:** This section explains how to execute the A71CH OpenSSL Engine example scripts and associated key material provided as part of the A71CH product support package. These scripts illustrate how to initiate a TLS/SSL based communication between a development PC acting as a server and the i.MX6UltraLite embedded platform acting as an IoT device client. Therefore, the following description refers purely to demonstration purposes and needs to be adjusted and adapted for commercial deployment.

---

### 5.1 Demo setup

This sample demo setup consists of the following items:

- IoT device: Represented by an i.MX6UltraLite (MCIMX6UL-EVKB) host board and an A71CH security IC connected to the OM3710/A71CHPCB contained in the OM3710/A71CHARD Arduino compatible development kit.
- OEM Cloud sever: Represented by a Linux Ubuntu machine.
- A development PC running a terminal application to interact with the i.MX6UltraLite (MCIMX6UL-EVKB) board.

Fig 12 illustrates these items and how are they connected.

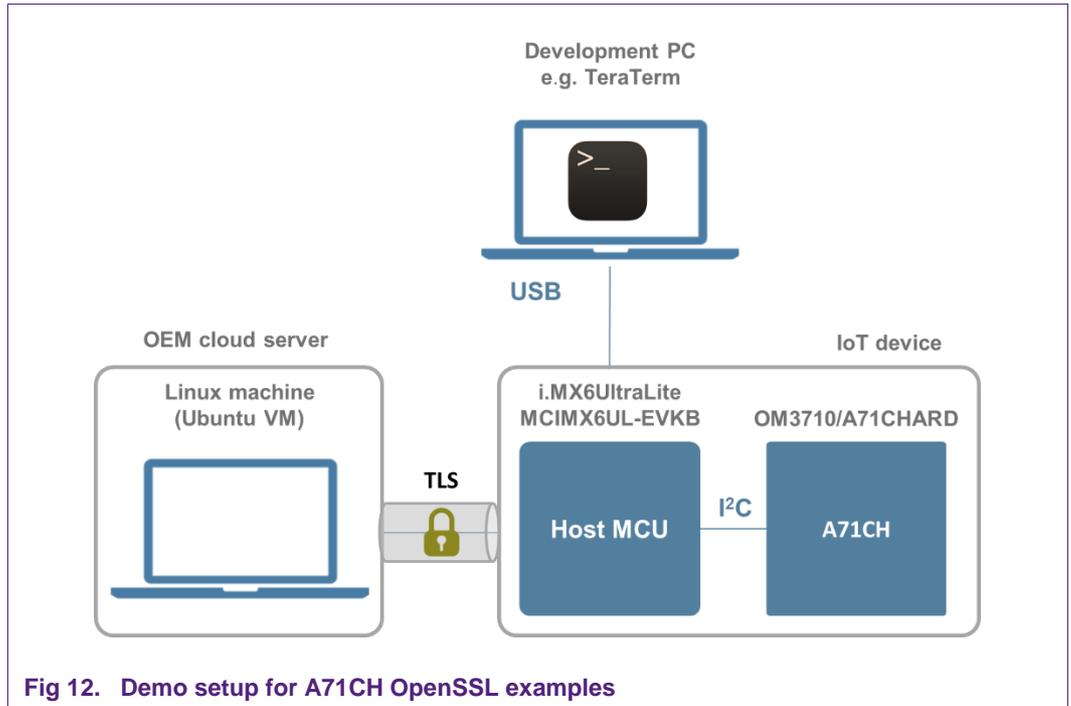


Fig 12. Demo setup for A71CH OpenSSL examples

The A71CH product support packages includes a precompiled image containing a Linux distribution with the A71CH Host Library and the OpenSSL Engine examples included in it. These scripts use the Configure Tool and the A71CH OpenSSL Engine library. More information on this can be found in [AN\_A71CH\_HOST\_SW].

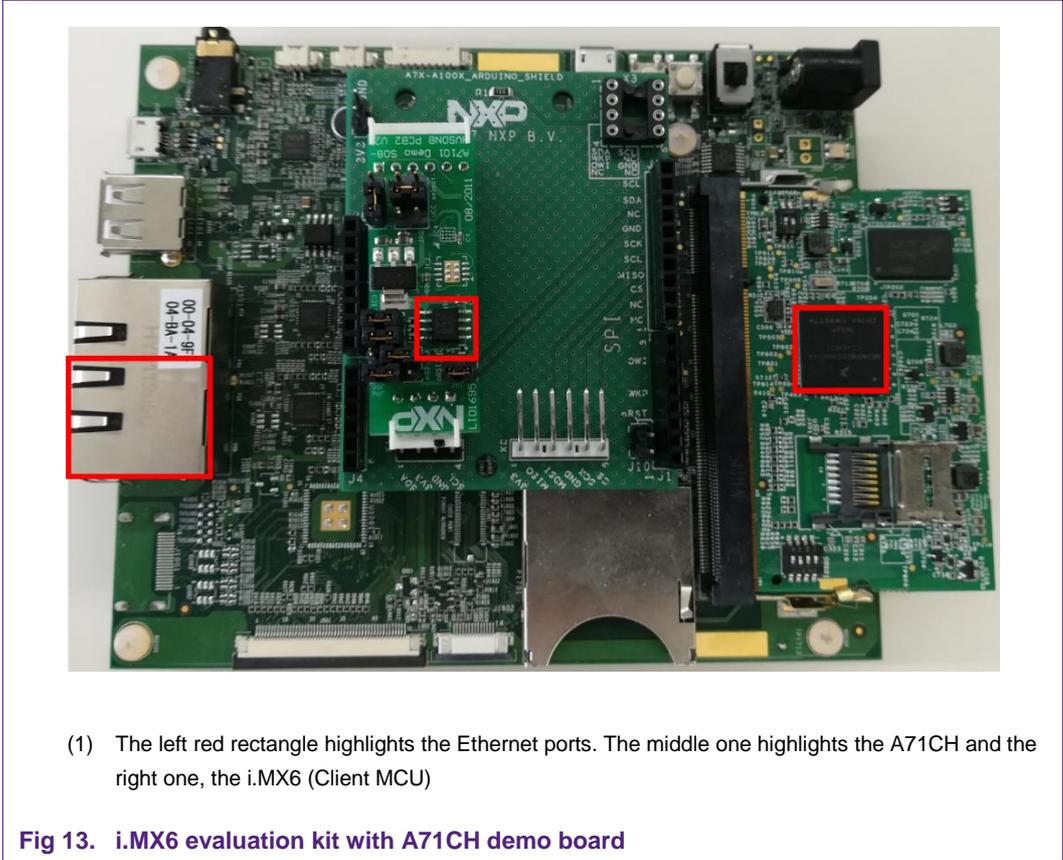
In this sample demo, the precompiled Linux image should be installed in a micro SD memory card and connected to i.MX6UltraLite evaluation board micro SD slot. On the other side, the Linux machine (Ubuntu) should also have the A71CH Host software package [A71CH\_HOST\_SW].

The following steps are required to establish a TLS communication between the IoT device and the OEM cloud:

1. i.MX6UltraLite and OM3170/A71CHPCB board setup.
2. IoT device and OEM cloud server credentials preparation.
3. A71CH security IC key injection.
4. Linux machine preparation and credentials transfer.
5. Starting the server TLS connection.
6. Starting the client TLS connection.

## 5.2 i.MX6UltraLite and A71CH mini PCB board setup

The i.MX6UltraLite evaluation kit should be connected to the A71CH mini PCB board with the Arduino header adapter as is illustrated in Fig 13. The i.MX6UltraLite board should be flashed with the precompiled Linux image. Please, refer to [QUICK\_START\_IMX6] for instructions about getting started with i.MX6UltraLite and OM3710A71CHARD.



Additionally, an internet connection is needed in this scenario. An Ethernet cable should be plugged into the bottom port available on the board. The IP address should be reachable for the development PC (Ubuntu – server).

### 5.3 IoT device and OEM cloud server credentials preparation

The ECC key pairs and digital certificates used in this demo are generated using the ***tlsCreateCredentialsRunOnClientOnce.sh*** script. This bash script must be executed first to prepare all the required ECC keys and certificates for the TLS/SSL connection. Once the i.MX6UltraLite is initialized and its terminal is opened, the commands to change directory to the one containing the scripts and execute this script are:

```
#cd axHostSw/hostLib/embSeEngine/a71chDemo/scripts
#./tlsCreateCredentialsRunOnClientOnce.sh
```

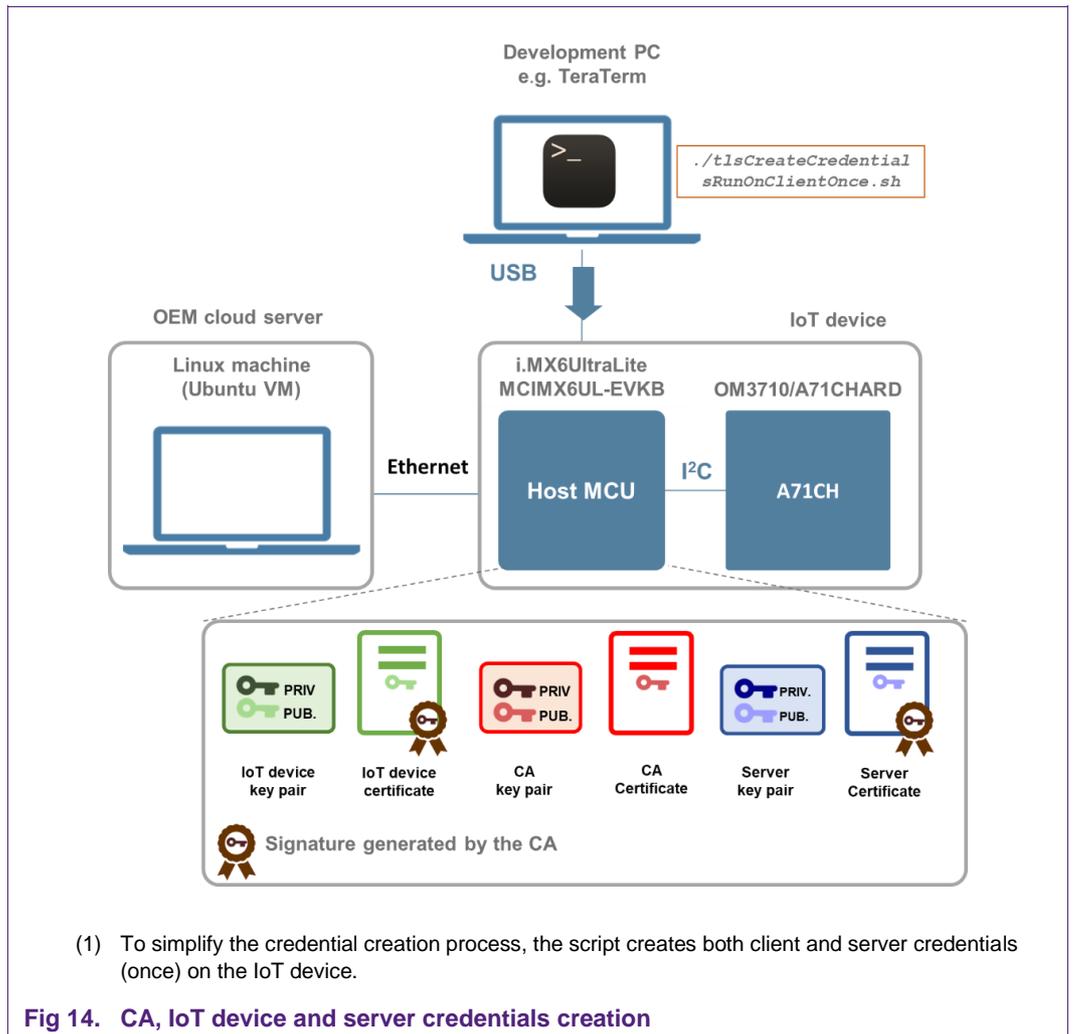
To simplify the credential creation process, the script creates both client and server credentials (once) on the client platform. One must transfer the server credentials created to the server platform.

First, the root CA is simulated and created by generating its root key pair and its root CA certificate using OpenSSL commands. In a real scenario, the CA is an external trusted entity whose root CA certificate and private key are securely stored on an HSM (Hardware Security Module).

After the CA certificate, the IoT device and OEM cloud credentials are created. The ECC private and public keys are generated using the P-NIST 256 ECC curves parameters as an OpenSSL function input argument. A CSR is then generated and sent to the CA. This CSR may contain, for instance, information about the IoT manufacturer, its main functionality, contact details, etc. Once the CSR has been used by the CA to generate the corresponding IoT device certificate signed with the root CA private key, it is sent back to the client.

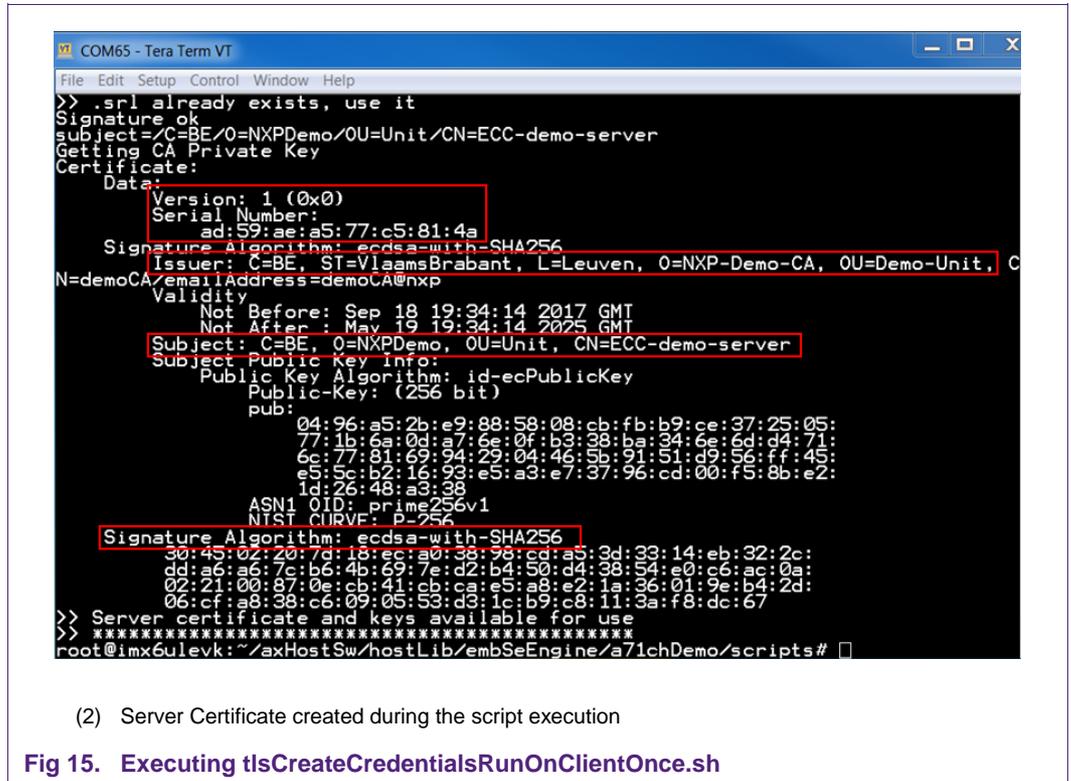
Similarly, the server ECC key pair is created and the CSR is prepared, sent and used by the CA to create server's certificate. At the end of the process, the client and the server credentials are ready.

Fig 14 illustrates the CA, IoT device and server credentials creation on the Host MCU (i.MX6 UltraLite of the MCIMX6UL-EVKB).



The Fig 15 shows a part of the script execution on the terminal. In this part, the server digital certificate is generated and printed in the terminal. The most interesting fields are

highlighted: the subject field, the issuer (Demo-CA), the signature algorithm and the serial number of the SE.



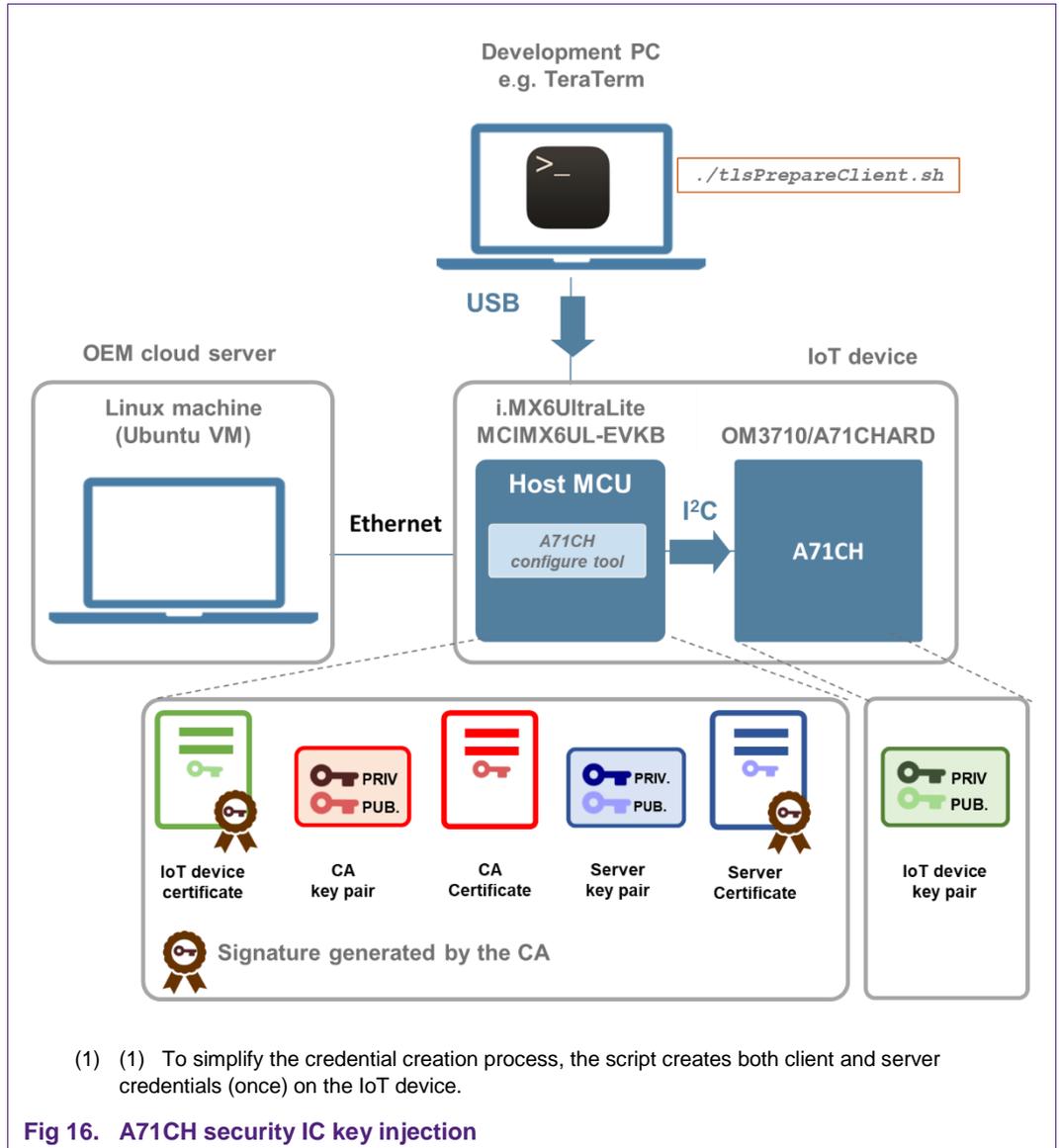
### 5.4 A71CH security IC key injection

The created IoT device ECC private and public keys should be injected into the A71CH security IC. This will ensure that these are kept safe and protected: once injected, the private key will never leave the module. Whenever these keys are required to digitally sign or verify a file, it is possible to use them within the A71CH, using the A71CH OpenSSL Engine functionality. The keys injection is carried out using the A71CH Configure Tool commands line, provisioning the A71CH with the client keys.

The IoT device contains the digital certificates and key pairs created in the previous step. The IoT device key pair is then stored inside the A71CH. It is possible to store the device and CA certificates or the CA public key inside the A71CH, although it is not done this way in this demo example. The command to be executed to store IoT device keys into the A71CH is:

```
#!/tlsPrepareClient.sh
```

Fig 16 illustrates the injection of the IoT device key pair into the A71CH. The script `tlsPrepareClient.sh` will use the Configure Tool application included in the A71CH Host software package.



The Fig 17 shows the end of the *tlsPrepareClient.sh* execution. In this case, the A71CH Configure Tool command *info pair* is executed to show the key pairs that have been stored inside the A71CH. The most relevant information is highlighted with a red rectangle.

```

COM65 - Tera Term VT
File Edit Setup Control Window Help
I2C driver: PEC flag cleared
I2C driver supports plain i2c-level commands.
I2C driver supports Read Block.
SCI2C_ATR=0xB8.04.11.01.05.04.B9.02.01.01.BA.01.01.BB.0D.41.37.31.30.31.43.48.20
.32.34.32.52.31.BC.00.
HostLib Version          : 0x0110
Applet-Rev:SecureBox-Rev : 0x0121:0x0000
ECCPrivateKey: hidden
ECCPublicKey (LEN=65):
04:1D:84:0B:B0:25:B4:70:80:28:5E:B9:90:95:9F:EB:D8:61:B1:13:83:90:A9:93:2C:CB:31
:5C:7A:21:82:CA:
CC:24:84:DB:3C:C2:D6:05:38:58:78:3E:85:E5:BF:D1:36:5E:06:1F:31:D2:86:E5:D0:E5:05
:F6:94:9F:70:88:
03
>> ../linux/a71chConfig_i2c_imx_info pair
a71chConfig (Rev 0.94) .. connect to A71CH. Chunksize at link layer = 256.
I2CInit: opening /dev/i2c-1
I2C driver: PEC flag cleared
I2C driver supports plain i2c-level commands.
I2C driver supports Read Block.
SCI2C_ATR=0xB8.04.11.01.05.04.B9.02.01.01.BA.01.01.BB.0D.41.37.31.30.31.43.48.20
.32.34.32.52.31.BC.00.
HostLib Version          : 0x0110
Applet-Rev:SecureBox-Rev : 0x0121:0x0000
Public Keys from ECC key pairs:
idx=0x00 ECC PUB (LEN=65):
04:1D:84:0B:B0:25:B4:70:80:28:5E:B9:90:95:9F:EB:D8:61:B1:13:83:90:A9:93:2C:CB:31
:5C:7A:21:82:CA:
CC:24:84:DB:3C:C2:D6:05:38:58:78:3E:85:E5:BF:D1:36:5E:06:1F:31:D2:86:E5:D0:E5:05
:F6:94:9F:70:88:
03
idx=0x01 n.a.
root@imx6ulevk:~/axHostSw/hostLib/embSeEngine/a71chDemo/scripts#
    
```

(1) First red rectangle highlights the execution on info pair command. Second one shows how the public keys from ECC key pairs look like

Fig 17. Executing tlsPrepareClient.sh

5.5 Linux machine preparation and credentials transfer (OEM cloud)

On the OEM cloud server side, there is some preparation to be done before executing the corresponding script:

- Transfer the most updated version of the Host Library to the Server machine and uncompress it.
- Update Linux repositories and upgrade the installed packages running the following commands on the Linux Terminal:

```
#sudo apt-get update
#sudo apt-get upgrade
```

- Install additional packages for OpenSSL:

```
#sudo apt-get install libssl-dev
```

- Navigate to the Host Library folder and compile the library:

```
#cd ~/<...>/axHostSw/linux
#make -f Makefile_A71CH default applet=A71CH conn=i2c platf=native
```

- Navigate to the scripts folder:

```
#cd axHostSw/hostLib/embSeEngine/a71chDemo/scripts
```

The next step is to transfer the server credentials and a copy of the CA certificate from the IoT device (i.MX6UltraLite) to the Linux Ubuntu machine (e.g. using a USB drive, or any other mechanisms). In other words, to transfer the Server credentials from the IoT device storage to the Server storage.

For simplicity, this guide shows how to transfer all the credentials folder. With the following commands, the user can mount the USB drive into a folder inside the i.MX6UltraLite file system and copy the folder with the credentials.

```
#cd ~  
#mkdir usb  
#mount /dev/sdc1 ./usb  
#cp ~/axHostSw/hostLib/embSeEngine/a71chDemo/ecc ~/usb
```

Now the USB drive can be unmounted and connected to the PC running Linux (or virtual machine) and copy the folder into the same path of the Host library they were in the IoT device. Fig 18 shows the server credentials already transferred from the IoT device to the OEM Cloud server.

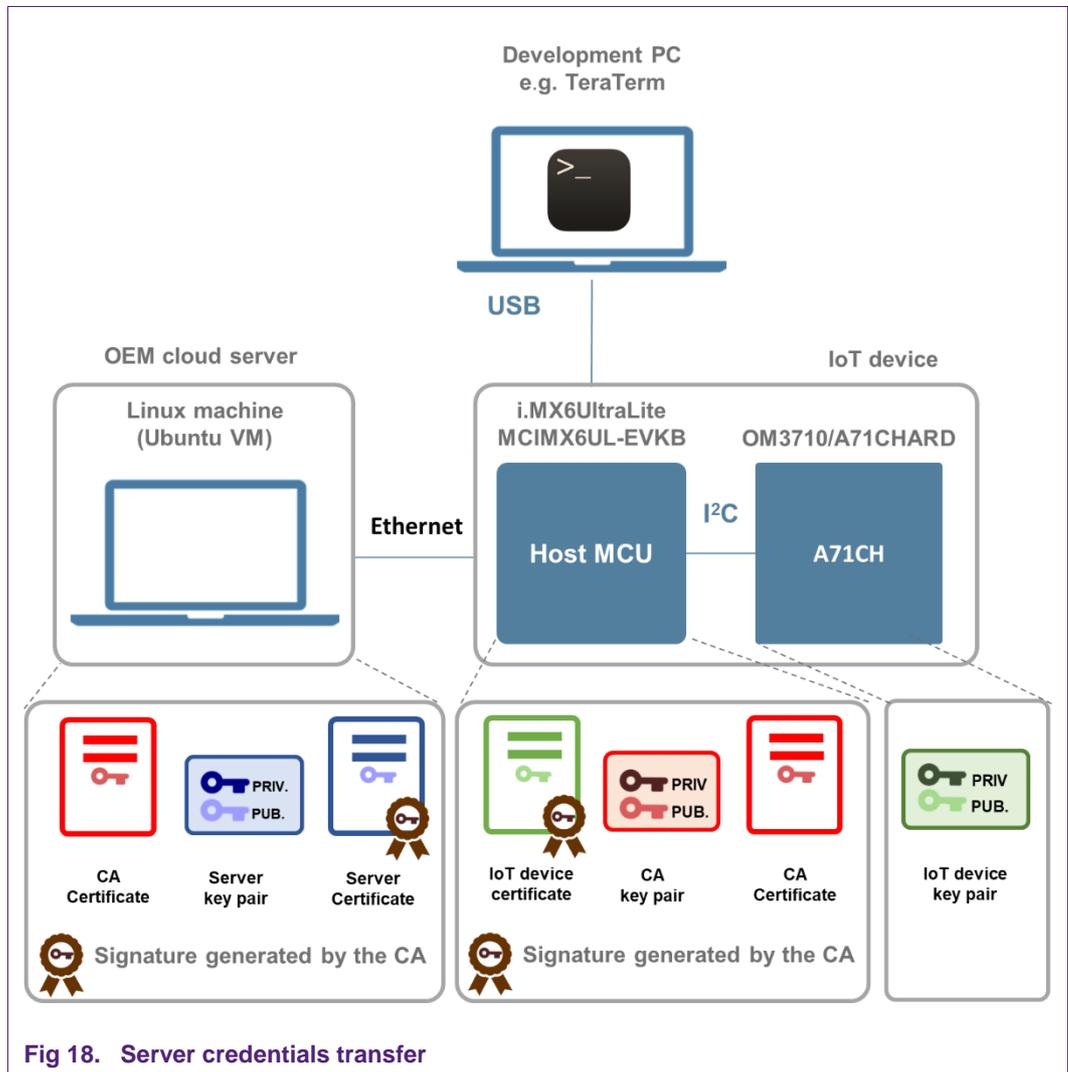
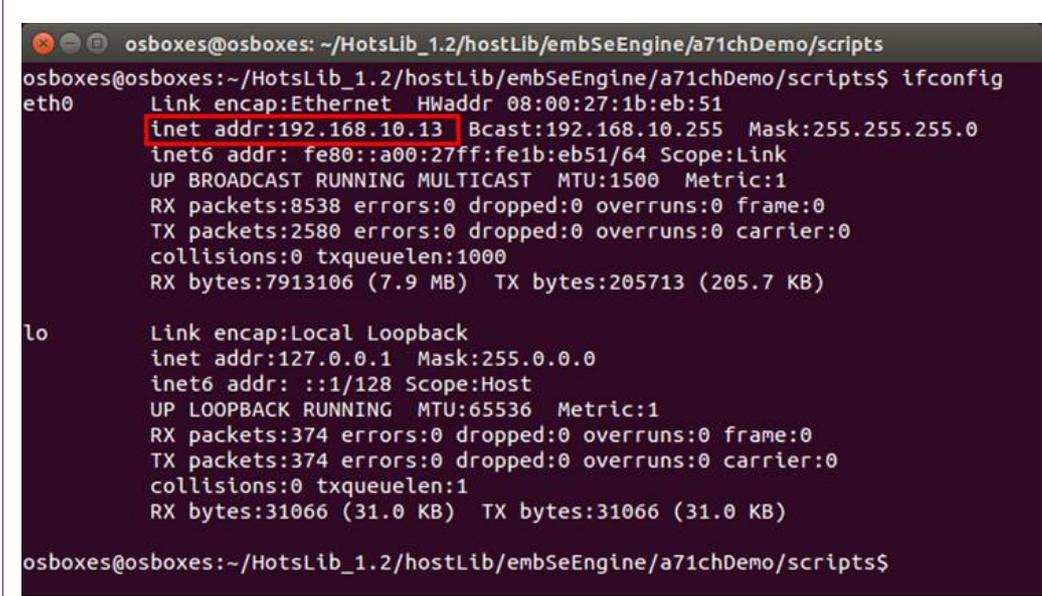


Fig 18. Server credentials transfer

## 5.6 Starting the server TLS connection

Before starting up the server, it is important to obtain the IP address that will be used in the IoT device commands. Executing the command `ifconfig` in the Linux machine terminal gives the user information about the network interfaces. The ethernet IP address is in the field `inet addr` (Fig 19).



```

osboxes@osboxes: ~/HotsLib_1.2/hostLib/embSeEngine/a71chDemo/scripts
osboxes@osboxes:~/HotsLib_1.2/hostLib/embSeEngine/a71chDemo/scripts$ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:1b:eb:51
          inet addr:192.168.10.13  Bcast:192.168.10.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe1b:eb51/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8538 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2580 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:7913106 (7.9 MB)  TX bytes:205713 (205.7 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:374 errors:0 dropped:0 overruns:0 frame:0
          TX packets:374 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:31066 (31.0 KB)  TX bytes:31066 (31.0 KB)

osboxes@osboxes:~/HotsLib_1.2/hostLib/embSeEngine/a71chDemo/scripts$

```

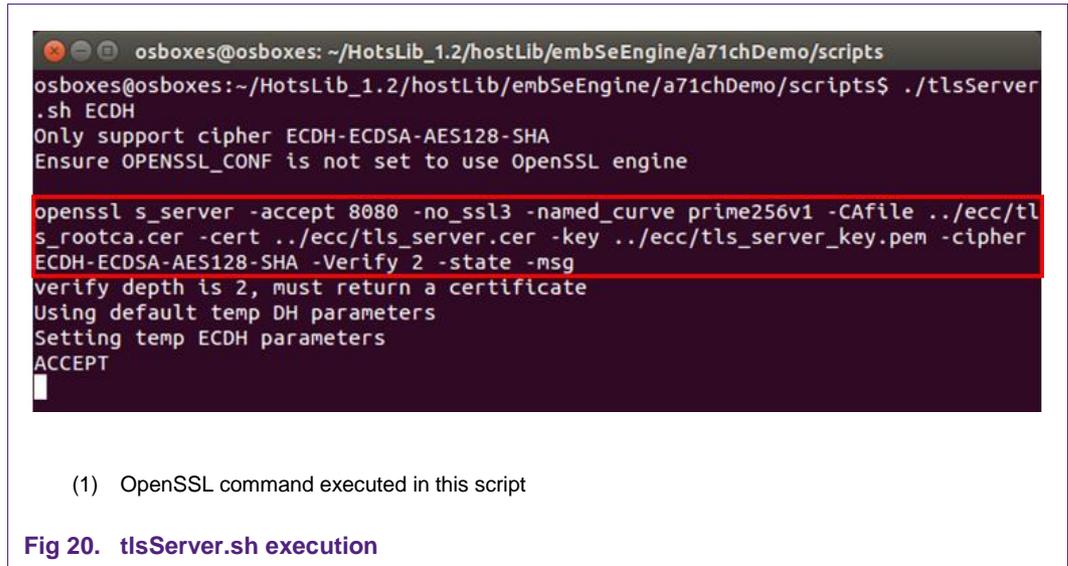
(1) Highlighted in red, the server IP address

**Fig 19. `ifconfig` command execution**

The script to start the server admits one parameter that can be chosen by the user. This parameter determine which algorithm is used in the key exchange phase of the TLS handshake protocol. The user can either choose ECDH, ECHDE or accept both key exchange algorithms. ECDH is used in this guide:

```
#./tlsServer.sh ECDH
```

The script uses an OpenSSL command whose input arguments are the server public key, the server certificate, the CA certificate and the port that it will be listening to (Fig 20). Now the server is ready to receive any client connection request.



### 5.7 Starting the client TLS connection.

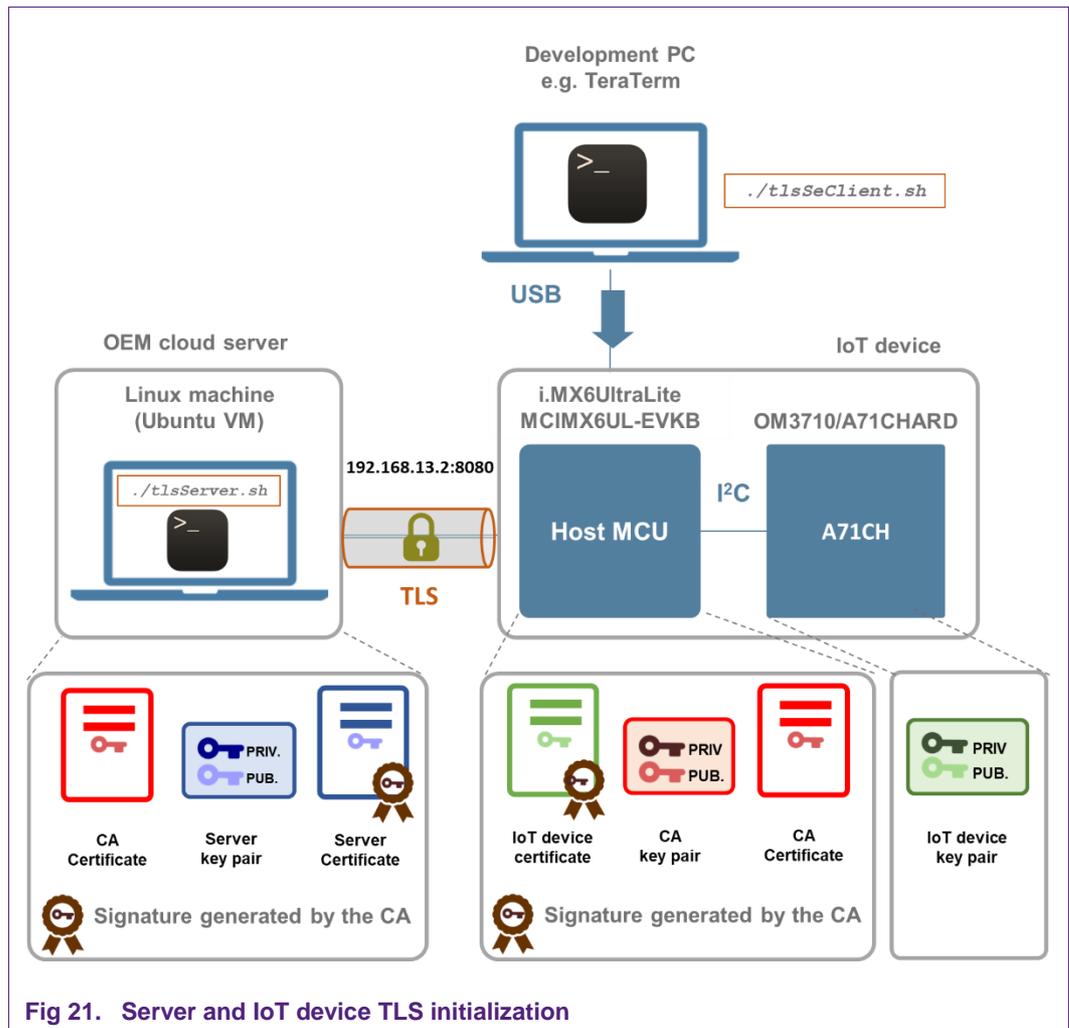
In the case of the IoT device, the script needs two input parameters: the first one is the Server IP address that was obtained in the previous step, and the second one is the algorithm for the key exchange that used for the *tlsServer.sh* script. This algorithm should be the same in both ends of the communication. In this example the script should be executed as:

```

#./tlsSeClient <IP-address> ECDH
    
```

Where the Server’s IP corresponds to the one obtained in Fig 19.

Correspondingly, the *tlsSeClient.sh* script is also started with an OpenSSL command that requires the IoT device public key, the IoT device certificate, the CA certificate and the IP address and port of the server. Once the IoT device and the server have been initialized, the TLS handshake starts. Fig 21 illustrates the connection between the IoT device and the Server through the port 8080 and the local IP address.



The result when the TLS connection has been successfully established is shown in the IoT device terminal (Tera Term, Fig 22).

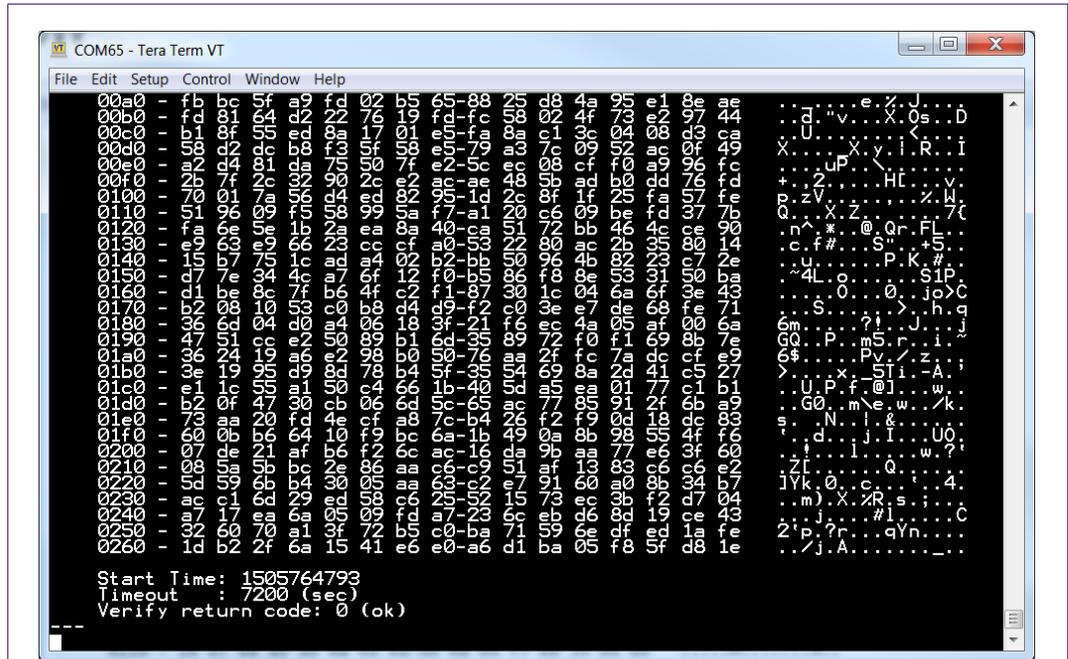


Fig 22. TLS connection success (IoT device side)

and server terminal (Linux terminal, Fig 23).

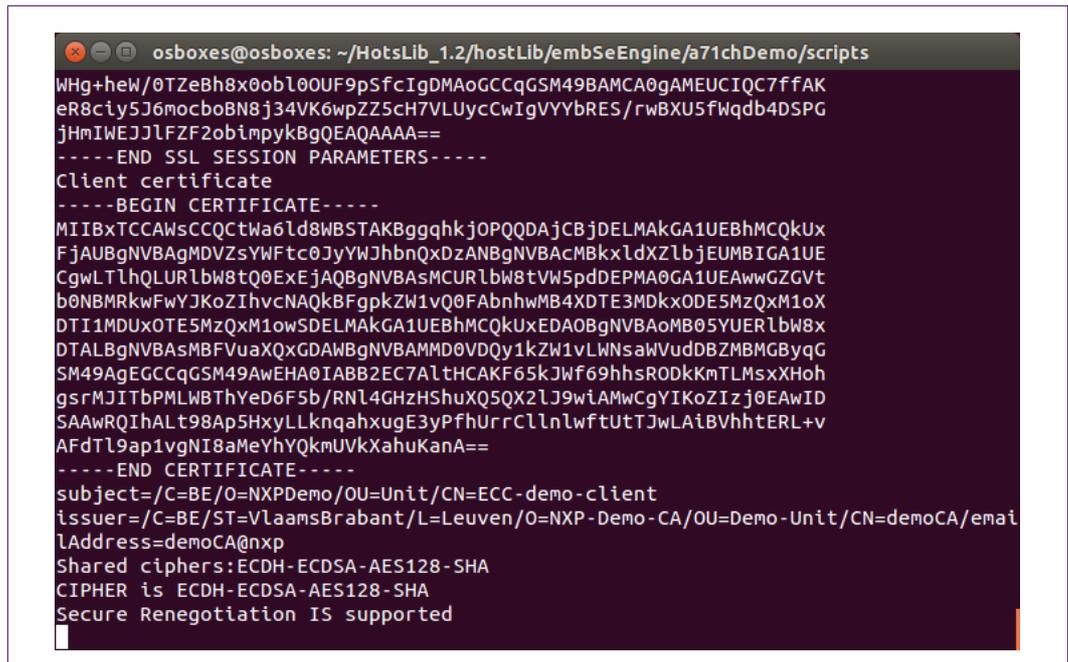


Fig 23. TLS connection success (Server side)

## 6. Referenced Documents

Table 1. Referenced Documents

[OPEN_SSL]	<b>OpenSSL Cryptography and SSL/TLS Toolkit information</b> - <a href="http://www.openssl.org">www.openssl.org</a>
[RFC4279]	<b>Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)</b> - December 2005
[RFC5489]	<b>ECDHE_PKE Cipher Suites for Transport Layer Security (TLS)</b> - March 2009
[RFC5246]	<b>The Transport Layer Security (TLS) Protocol</b> - Version 1.2, August 2008
[RFC4492]	<b>Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)</b> - May 2006
[A71CH_HOST_SW]	<b>A71CH Host Software Package (Windows Installer)</b> - DocStore sw4673xx <sup>1</sup> , Version 01.03.00 (or later) available on <a href="http://www.nxp.com/A71CH">www.nxp.com/A71CH</a> <b>A71CH Host Software Package (Bash installer)</b> - DocStore sw4672xx <sup>1</sup> , Version 01.03.00 (or later) available on <a href="http://www.nxp.com/A71CH">www.nxp.com/A71CH</a>
[A71CH_OPENSSL_ENGINE]	<b>A71CH OpenSSL Engine</b> – DocStore um4334 <sup>**1</sup>
[QUICK_START_IMX6]	<b>AN12119 Quick start guide for OM3710A71CHARD i.MX6</b> – Application note, document number 4582 <sup>**1</sup>
[AN_A71CH_HOST_SW]	<b>AN12133 A71CH Host software package documentation</b> – Application note, document number 4643 <sup>**1</sup>

<sup>1</sup> \*\* ... document version number

## 7. Legal information

### 7.1 Definitions

**Draft** — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

### 7.2 Disclaimers

**Limited warranty and liability** — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the *Terms and conditions of commercial sale* of NXP Semiconductors.

**Right to make changes** — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

**Suitability for use** — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

**Applications** — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the

customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

**Export control** — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

**Translations** — A non-English (translated) version of a document is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

**Evaluation products** — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out of the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US\$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

### 7.1 Licenses

#### ICs with DPA Countermeasures functionality



NXP ICs containing functionality implementing countermeasures to Differential Power Analysis and Simple Power Analysis are produced and sold under applicable license from Cryptography Research, Inc.

### 7.2 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

**FabKey** — is a trademark of NXP B.V.

**IC-bus** — logo is a trademark of NXP B.V.

## 8. List of figures

Fig 1.	Digital signature diagram .....	4
Fig 2.	Digital certificate generation steps and contents .....	5
Fig 3.	Elliptic Curve Digital Signature Algorithm (ECDSA) example.....	5
Fig 4.	Elliptic Curve Diffie-Hellman Key Exchange (ECDH) example.....	6
Fig 5.	TLS connection between an IoT device and an OEM cloud .....	7
Fig 6.	IoT device certificates .....	8
Fig 7.	OEM cloud server certificates .....	9
Fig 8.	TLS connection between two IoT devices and OEM cloud .....	9
Fig 9.	Communication stack. ISO/OSI Layers.....	10
Fig 10.	TLS 1.2 Handshake diagram with ECDHE- ECDSA.....	12
Fig 11.	Host SW stack including OpenSSL, OpenSSL engine and A71CH Host Library .....	13
Fig 12.	Demo setup for A71CH OpenSSL examples ..	15
Fig 13.	i.MX6 evaluation kit with A71CH demo board.	16
Fig 14.	CA, IoT device and server credentials creation .....	17
Fig 15.	Executing tlsCreateCredentialsRunOnClientOnce.sh.....	18
Fig 16.	A71CH security IC key injection.....	19
Fig 17.	Executing tlsPrepareClient.sh.....	20
Fig 18.	Server credentials transfer .....	21
Fig 19.	ifconfig command execution.....	22
Fig 20.	tlsServer.sh execution.....	23
Fig 21.	Server and IoT device TLS initialization .....	24
Fig 22.	TLS connection success (IoT device side).....	25
Fig 23.	TLS connection success (Server side).....	25

9. List of tables

---

Table 1. Referenced Documents .....26

## 10. Contents

<b>1.</b>	<b>Introduction .....</b>	<b>3</b>	<b>8.</b>	<b>List of figures .....</b>	<b>28</b>
<b>2.</b>	<b>A71CH overview .....</b>	<b>3</b>	<b>9.</b>	<b>List of tables .....</b>	<b>29</b>
<b>3.</b>	<b>Public key infrastructure and ECC fundamentals .....</b>	<b>3</b>	<b>10.</b>	<b>Contents .....</b>	<b>30</b>
3.1	Digital signature .....	4			
3.2	Digital certificate, Certification Authority (CA) and Certificate Signing Request (CSR) .....	4			
3.3	Elliptic Curve Digital Signature Algorithm (ECDSA) .....	5			
3.4	Elliptic Curve Diffie-Hellman (ECDH) .....	5			
3.5	A71CH ECC supported functionality .....	6			
<b>4.</b>	<b>A71CH for secure connection to OEM cloud ....</b>	<b>7</b>			
4.1	IoT device credentials .....	7			
4.2	OEM cloud server credentials .....	8			
4.3	Transport Layer Security protocol (TLS) .....	9			
4.3.1	Transport Layer Security Handshake protocol ..	10			
4.3.2	Transport Layer Security software libraries .....	13			
<b>5.</b>	<b>Evaluating A71CH for secure connection to OEM cloud.....</b>	<b>14</b>			
5.1	Demo setup.....	14			
5.2	i.MX6UltraLite and A71CH mini PCB board setup .....	15			
5.3	IoT device and OEM cloud server credentials preparation .....	16			
5.4	A71CH security IC key injection .....	18			
5.5	Linux machine preparation and credentials transfer (OEM cloud).....	20			
5.6	Starting the server TLS connection .....	22			
5.7	Starting the client TLS connection.....	23			
<b>6.</b>	<b>Referenced Documents .....</b>	<b>26</b>			
<b>7.</b>	<b>Legal information .....</b>	<b>27</b>			
7.1	Definitions .....	27			
7.2	Disclaimers.....	27			
7.1	Licenses.....	27			
7.2	Trademarks.....	27			

---

Please be aware that important notices concerning this document and the product(s) described herein, have been included in the section 'Legal information'.

---