M4521 ECC Certification

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example code realizes Elliptic Curve Cryptography (ECC) encryption and decryption by software on the M4521 series microcontroller (MCU) by Mbed TLS library.
BSP Version	M4521_BSP_CMSIS_V3.00.003
Hardware	NuTiny-SDK-M4521S

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1. Overview

Mbed TLS is an available SSL (Secure Sockets Layer) library as open-source under the Apache 2.0 license or the GPL 2.0 license, containing a wealth of encryption and decryption functions that can be used for secure communication to avoid data eavesdropping.

This example code performs Elliptic Curve Cryptography (ECC) encryption and decryption operation with pure software by Mbed TLS library, alleviating the problem that some evaluation boards cannot implement encryption and decryption functions in hardware.

1.1 Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) is the public key encryption system based on the mathematical principle of the elliptic curve and belongs to symmetric encryption technology. Under the same security level, ECC has a shorter key than other encryption algorithms; in addition, it also provides higher security level encryption. For example, the ECC key strength of 256 bits is equivalent to 3072 bits of RSA key.

Because of the complex mathematical logic, the ECC algorithm has the disadvantage of time-consuming encryption and decryption compared with other methods. However, it is generally believed that ECC is the most powerful asymmetric encryption algorithm among other crypto algorithms with the same key length. The ECC algorithm is based on the discrete logarithm problem on elliptic curves and takes a plane curve in a finite field as the application for cryptography[1,2]. The most commonly used elliptic curve equation for ECC is as follows:

$$y^2 = x^3 + ax + b \quad (\mathrm{a}, \mathrm{b} \in \mathrm{GF}(\mathrm{p}), 4a^3 + 27b^2 \neq 0)$$

Where *a* and *b* are any real numbers, the elliptic curve will have different shapes on the plane according to the values of *a* and *b*, but the elliptic curve will be always symmetrical to the x-axis, and the equation must satisfy no singularity, that is, the graph has no cusps or self-Intersection[3]. The random elliptic curve function is shown in Figure 1-1 and Figure 1-2.

1 https://zh.m.wikipedia.org/zh-tw/%E6%A4%AD%E5%9C%86%E6%9B%B2%E7%BA%BF%E5%AF%86%E7%A0%81%E5%AD%A6
2 https://zhuanlan.zhihu.com/p/42629724
3 https://zh.m.wikipedia.org/zh-tw/%E6%A4%AD%E5%9C%86%E6%9B%B2%E7%BA%BF

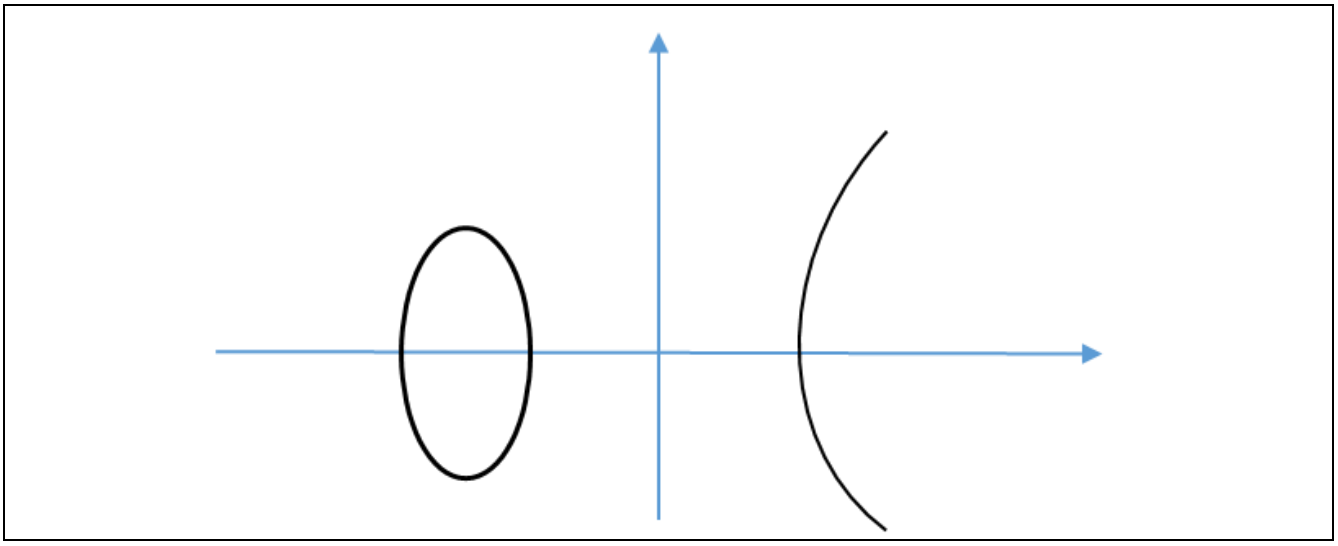

Figure 1-1 Elliptic Curve Graph ($y^2 = x^3 - 1$)

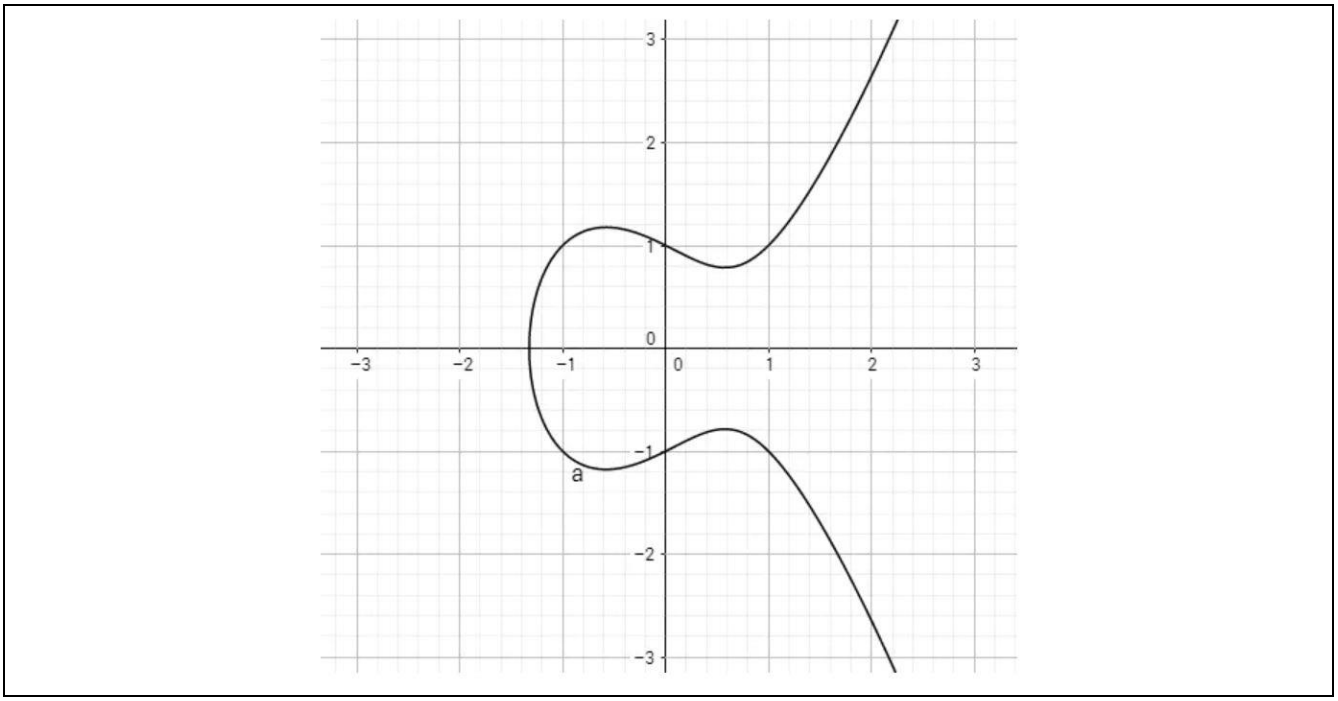

Figure 1-2 Elliptic Curve Graph ($y^2 = x^3 - x + 1$) [4]

4 https://www.jianshu.com/p/899077117ff8

1.2 Elliptic Curve Digital Signature Algorithm (ECDSA)

1.2.1 Principle and Operation

In order to build an elliptic curve-based encryption mechanism, it is necessary to find difficult problems such as RSA prime factorization or other discrete logarithms. The discrete logarithm problem on the elliptic curve needs to find x for the known G and xG on the elliptic curve, which is a very difficult problem so the Elliptic Curve Digital Signature Algorithm was born from this.

In cryptography, user needs to generate a pair of keys: a public key and a private key. The encryption mechanism of the elliptic curve is set k as a private key on the elliptic curve, and set k = kG as a public key. The private key is responsible for encryption, and kept by the sender; the public key is announced to the public and used for decryption.

When the plaintext is digested by the signature algorithm, it is encrypted with the private key to form a digital signature before it is published (the plaintext is not directly encrypted by the private key), and the user who has the corresponding public key can decrypt the signature with the public key. The deciphered digest will be consistent with the sender's content before encryption, and thus confirm that the data has not been tampered with. The process of encryption and decryption is shown in Figure 1-3 and Figure 1-4, respectively.

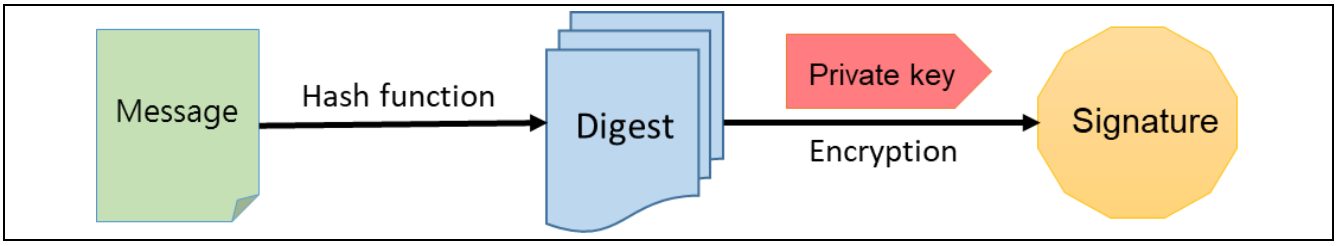

Figure 1-3 Encryption to Form a Signature

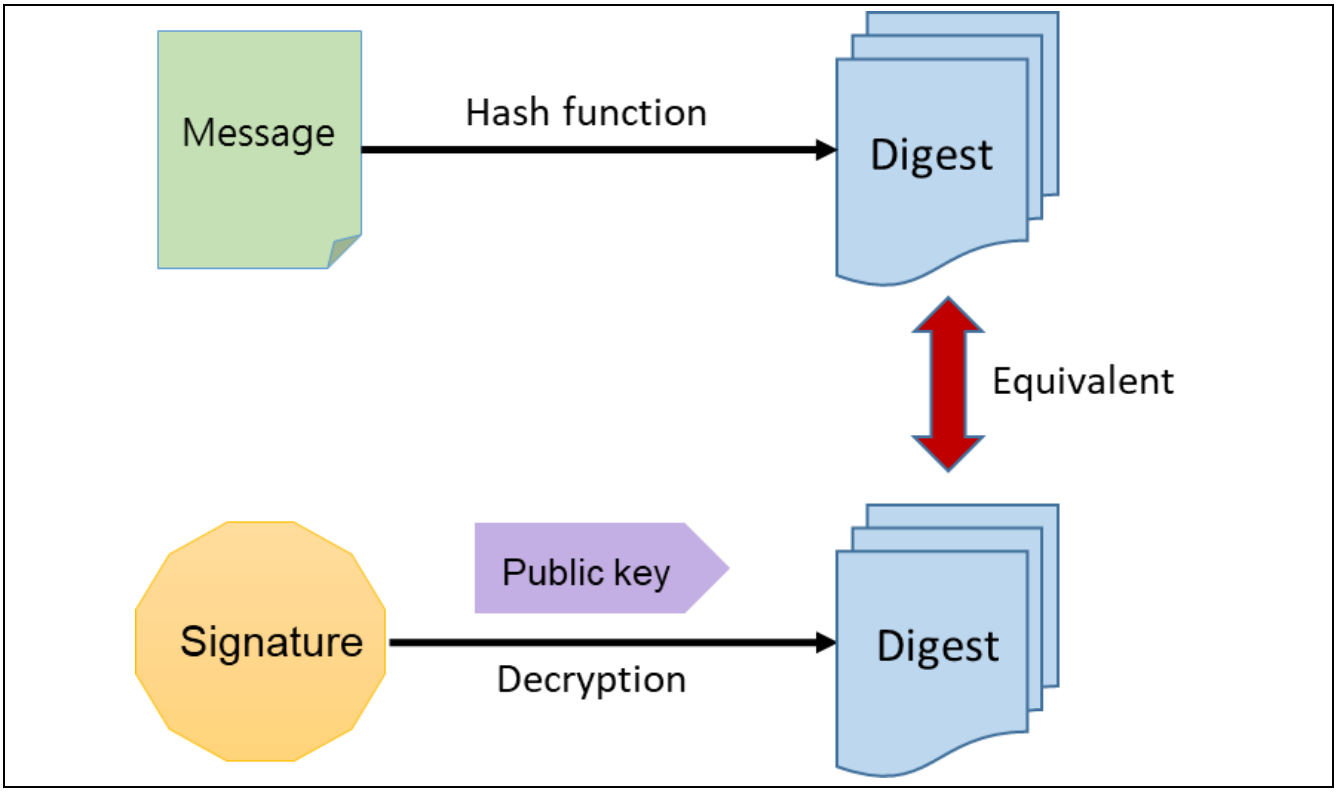

Figure 1-4 Decryption Signature by Public Key

1.2.2 Generation of Keys

First, select a set of elliptic curves according to the requirements (number of bits, signature compliance, etc.), and call ***mbedtls_ecp_curve_info_from_grp_id()*** of Mbed TLS library to obtain the relevant information of the curve. Then, call the function ***mbedtls_ecdsa_genkey()*** to generate the elliptic curve with the processed parameter ***gid***. Finally, generate the ECDSA key by ***mbedtls_ecdsa_from_keypair()***. The detailed function description is shown in Table 1-1 to Table 1-3.

*mbedtls_ecp_curve_info_from_grp_id(mbedtls_ecp_group_id grp_id)		
parameter :	grp_id	Parameter of Elliptic Curve
Return :		Null

Table 1-1 API of Elliptic Curve Information Extraction

mbedtls_ecdsa_genkey(mbedtls_ecdsa_context *ctx, mbedtls_ecp_group_id gid, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng)		
parameter :	*ctx	Initialized ECDSA context
	gid	Parameter of Elliptic Curve
	*f_rng	Ramdom function
	*p_rng	Additional functions of *f_rng* (Can be *NULL* when it unnecessary)

Return :	Success = 0 Failure = MBEDTLS_ERR_ECP_XXX

Table 1-2 API of Generating Key Pair from Elliptic Curve

mbedtls_ecdsa_from_keypair(mbedtls_ecdsa_context *ctx, const mbedtls_ecp_keypair *key)		
parameter :	*ctx	Initialized ECDSA context
	*key	The ECC key to use.
Return :		Success = 0 Failure = MBEDTLS_ERR_ECP_XXX

Table 1-3 API of Generating Parameters of ECDSA Key

1.2.3 Hash Value

Hash is a method of processing data in computer science. Through a specific function or algorithm, the item to be retrieved is associated with the index used for retrieval (called a hash, or a hash value) to generate a data structure that is easy to search, called hash table).

Hash is also commonly used as an implementation method for information security. It is a data fingerprint calculated by a hash algorithm in a string of data. It is often used to identify whether files and data have been tampered with, so as to ensure that files and data are indeed created by the original creator. Because the value calculated by the hash algorithm is irreversible (the original value cannot be reversely calculated), and the hash value must follow the changes in the plaintext, it can effectively protect the password. For example, SHA-256 is one of the most widely used cryptographic functions, which generates a hash value of 256-bit[5].

1.2.4 Digital Signature

The digital signature is written and generated by encrypting the message by the hash function with the private key, and can only be decrypted with the public key released by the sender. User can call ***mbedtls_ecdsa_write_signature()*** in the Mbed TLS library to calculate and write the signature, and record the executed program step by step with ***mbedtls_ecdsa_write_signature_restartable()***. If an error occurs, the error code will be returned immediately, and then restart at the specified location to save total blocking time. The detailed function description is shown in Figure 1-4 and Table 1-5.

mbedtls_ecdsa_write_signature(mbedtls_ecdsa_context *ctx,mbedtls_md_type_t md_alg, const unsigned char *hash, size_t hlen, unsigned char *sig, size_t sig_size, size_t *slen, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng)		
parameter :	*ctx	Initialized ECDSA context
	md_alg	Message-digest algorithm used to hash the original data.
	*hash	The buffer holding the message digest or raw

[5] https://zh.wikipedia.org/zh-tw/%E6%95%A3%E5%88%97%E5%87%BD%E6%95%B8

		data.
	hlen	Length of the message digest or raw data (Bytes)
	*sig	The buffer to hold the signature.
	sig_size	The size of the *sig buffer in bytes.
	*slen	The address at which to store the actual length of the signature written.
	*f_rng	Ramdom function
	*p_rng	*p_rng Additional functions of *f_rng (It can be NULL when unnecessary.)
Return :		Success = 0 Failure = MBEDTLS_ERR_ECP_XXX, MBEDTLS_ERR_MPI_XXX, MBEDTLS_ERR_ASN1_XXX

Table 1-4 API of Writing the Signature

mbedtls_ecdsa_write_signature_restartable(mbedtls_ecdsa_context *ctx, mbedtls_md_type_t md_alg, const unsigned char *hash, size_t hlen, unsigned char *sig, size_t sig_size, size_t *slen, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, mbedtls_ecdsa_restart_ctx *rs_ctx)		
parameter :	*ctx	Initialized ECDSA context
	md_alg	Message-digest algorithm used to hash the original data.
	*hash	The buffer holding the message digest or raw data.
	hlen	Length of the message digest or raw data (Bytes)
	*sig	The buffer to hold the signature.
	sig_size	The size of the *sig buffer in bytes.
	*slen	The address at which to store the actual length of the signature written.
	*f_rng	Ramdom function
	*p_rng	Additional functions of *f_rng* (Can be *NULL* when it unnecessary)
	*rs_ctx	The restart context to use. (Can be NULL when it unnecessary)
Return :		Success = 0 Operations reached =

	MBEDTLS_ERR_ECP_IN_PROGRESS Failure = MBEDTLS_ERR_ECP_XXX, MBEDTLS_ERR_MPI_XXX, MBEDTLS_ERR_ASN1_XXX

Table 1-5 API of Restarting the Signature

1.2.5 Verification of Digital Signature

Since the digital signature processes the data with the private key of the sender and can only pass the verification with the corresponding public key, it is often used to prevent forgery, denial, and detection of tampering. As shown in Figure 1-4, the point of the digital signature is: after the public key is decrypted, whether the digest obtained is consistent with the digest obtained by the sender after hashing the message. If consistent, it means the signature is legal; otherwise, it is illegal.

Users can call ***mbedtls_ecdsa_read_signature()*** in the Mbed TLS library to verify the digital signature, it also includes ***mbedtls_ecdsa_write_signature_restartable()*** for assistance. The detailed function description is shown in Table 1-6 and Table 1-7.

mbedtls_ecdsa_read_signature(mbedtls_ecdsa_context *ctx, const unsigned char *hash, size_t hlen, const unsigned char *sig, size_t slen)		
parameter	*ctx	Initialized ECDSA context
	*hash	The buffer holding the message digest or raw data.
	hlen	Length of the message digest or raw data (Bytes)
	*sig	The buffer to hold the signature.
	slen	The address at which to store the actual length of the signature written.
Return		Success = 0 Failure = MBEDTLS_ERR_ECP_XXX, MBEDTLS_ERR_MPI_XXX

Table 1-6 API of Verification Signature

mbedtls_ecdsa_read_signature_restartable(mbedtls_ecdsa_context *ctx, const unsigned char *hash, size_t hlen, const unsigned char *sig, size_t slen, mbedtls_ecdsa_restart_ctx *rs_ctx)		
parameter	*ctx	Initialized ECDSA context
	*hash	The buffer holding the message digest or raw data.
	hlen	Length of the message digest or raw data (Bytes)
	*sig	The buffer to hold the signature.
	slen	The address at which to store the actual length of the signature written.
	*rs_ctx	The restart context to use.

		(Can be NULL when it unnecessary)
Return		Success = 0 Failure = MBEDTLS_ERR_ECP_XXX, MBEDTLS_ERR_MPI_XXX

Table 1-7 API of Restarting Verification Signature

1.3 Demo Result

The demo result is shown below.

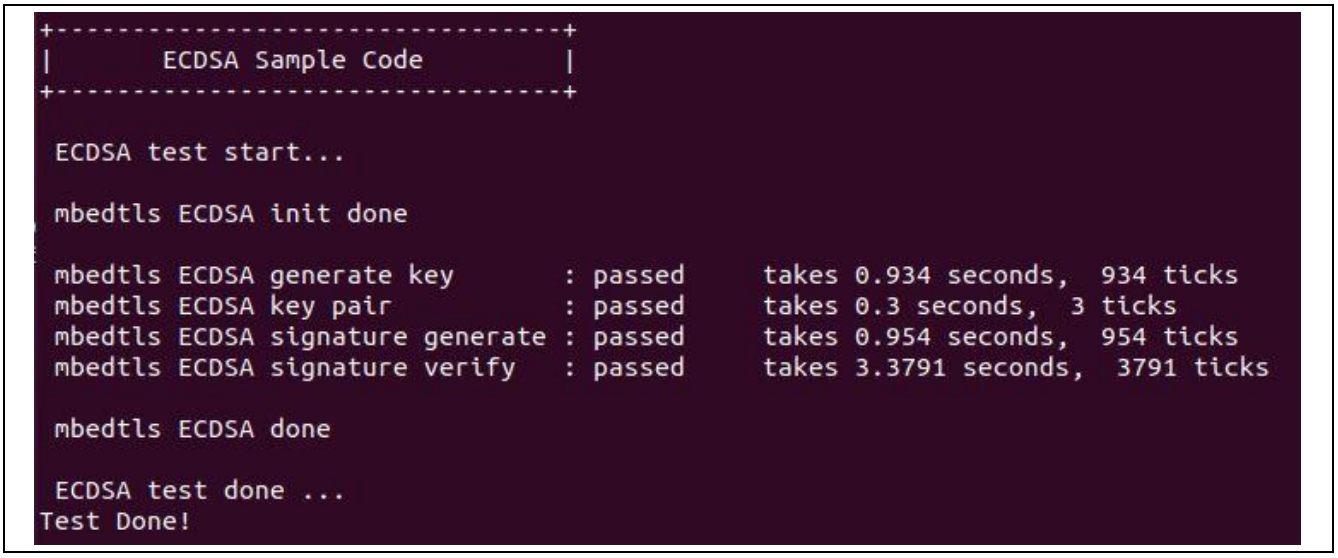

Figure 1-5 Demo Result

2. Code Description

Use the Mbed TLS library to generate the public and private keys of ECDSA. The code is located in *ecdsa_test.c*.

```
    mbedtls_ecdsa_context ecdsa;
    size_t sig_len;
    const mbedtls_ecp_curve_info *curve_info =
                              mbedtls_ecp_curve_info_from_grp_id(MBEDTLS_ECP_DP_BP256R1);
    mbedtls_pk_context pk;
    mbedtls_ecp_keypair *ecp;

    mbedtls_pk_init(&pk);

    /* Initializes an ECDSA context. */
    mbedtls_ecdsa_init(&ecdsa);
    printf(" mbedtls ECDSA init done  \n\n");

    printf(" mbedtls ECDSA generate key       : ");
       /* Generate the keypair on an elliptic curve */
    ret = mbedtls_ecdsa_genkey(&ecdsa, curve_info->grp_id, myrand, NULL);
    if(ret == 0) {
        printf("passed");
        u32Time = get_timer0_counter();
        printf("     takes %d.%d seconds,  %d ticks\n", u32Time / 1000000, u32Time / 1000,
g_u32Ticks);
    } else {
        printf("failed! ret[%d]\n", ret);
        mbedtls_ecdsa_free(&ecdsa);
        return ret;
    }

printf(" mbedtls ECDSA key pair           : ");
   /* Generate the public key and private key from key pair */
    ret = mbedtls_ecdsa_from_keypair(&ecdsa, &ecdsa);
    if(ret == 0) {
        printf("passed");
        u32Time = get_timer0_counter();
        printf("     takes %d.%d seconds,  %d ticks\n", u32Time / 1000000, u32Time / 1000,
g_u32Ticks);
    } else {
        printf("failed! ret[%d]\n", ret);
        mbedtls_ecdsa_free(&ecdsa);
        return ret;
}
```

After key pairs are completed, use the hash function to encrypt the data, and then create a digital signature by the private key. The code is located in *ecdsa_test.c.*

```
/* Cacualate the data hash */
    const mbedtls_md_info_t *md_info;
    md_info = mbedtls_md_info_from_type(MBEDTLS_MD_SHA256);
    mbedtls_md(md_info,(const unsigned char *) test_image_bin, sizeof(test_image_bin),
hash);
```

```
    /* Computes the ECDSA signature and writes it to a buffer */
    printf(" mbedtls ECDSA signature generate : ");
    ret = mbedtls_ecdsa_write_signature(&ecdsa, MBEDTLS_MD_SHA256, hash, sizeof(hash),
tmp, sizeof(tmp), &sig_len, myrand, NULL);

    if(ret == 0) {
        printf("passed");
        u32Time = get_timer0_counter();
        printf("     takes %d.%d seconds,  %d ticks\n", u32Time / 1000000, u32Time / 1000,
g_u32Ticks);
    } else {
        printf("failed! ret[%d]\n", ret);
        mbedtls_ecdsa_free(&ecdsa);
        return ret;
    }
```

Read the contents of the digital signature and verify it by the public key. The code is located in *ecdsa_test.c*.

```
  /* Reads and verifies an ECDSA signature */
    printf(" mbedtls ECDSA signature verify   : ");
    ret = mbedtls_ecdsa_read_signature(&ecdsa, hash, sizeof(hash), tmp, sig_len);
    if(ret == 0) {
        printf("passed");
        u32Time = get_timer0_counter();
        printf("     takes %d.%d seconds,  %d ticks\n", u32Time / 1000000, u32Time / 1000,
g_u32Ticks);
    } else {
        printf("failed! ret[%d]\n", ret);
        mbedtls_ecdsa_free(&ecdsa);
        return ret;
    }

    mbedtls_ecdsa_free(&ecdsa);
    mbedtls_ecdsa_free(ecp);
    printf("\n mbedtls ECDSA done  \n");
```

3. Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - M4521_BSP_CMSIS_V3.00.003
- IDE version
  - Keil uVersion 6.10

3.2 Hardware Requirements

- Circuit components
  - NuTiny-SDK-M4521S

4. Directory Information

EC_ M4521_ECC_Certification _V1.00	
Library	Sample code header and source files
CMSIS	Cortex® Microcontroller Software Interface Standard (CMSIS) by Arm® Corp.
Device	CMSIS compliant device header file
StdDriver	All peripheral driver header and source files
SampleCode	
ExampleCode	Source file of example code
ThirdParty	
mbedtls-3.1.0	A portable, easy to use, readable and flexible SSL library. Mbed TLS is available as open source under the Apache 2.0 license or the GPL 2.0 license. Its official website: https://tls.mbed.org/

Figure 4-1 Directory Structure

5. Example Code Execution

1. Browse the sample code folder as described in the Directory Information section and double-click *ECC_Certification.uvproj*.
2. Enter Keil compile mode.
   - Build
   - Download
   - Start/Stop debug session
3. Enter debug mode.
   - Run

6. Revision History

Date	Revision	Description
2022.08.02	1.00	Initial version.

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.