M4521 RSA Certification

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example code realizes Rivest-Shamir-Adleman (RSA) 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 encryption and decryption operation of RSA with pure software way by Mbed TLS library, alleviating the problem that some evaluation boards cannot implement encryption and decryption functions in hardware.

1.1 Rivest–Shamir–Adleman Algorithm

The Rivest-Shamir-Adleman (RSA) algorithm is derived from a difficult mathematical problem in integer factorization of maximum value. The difficulty of factorization of integers determines the reliability of RSA, the more difficult it is, the more reliable the algorithm is.

RSA is an asymmetric encryption algorithm widely used in electronic commerce. It requires two keys: public key and private key, used for encryption and decryption respectively. The ciphertext obtained by encrypted with the public key can only be decrypted to obtain the original plaintext by the corresponding private key. Conversely, plaintext can also be encrypted with the private key and decrypted with the public key. Since encryption and decryption require two different keys, it was called asymmetric encryption.

The public key can be released publicly to many, but the private key must be kept strictly secret by the user, never provided to anyone through any means. Based on the characteristics of encrypted by public key, it can also provide the function of a digital signature, so that the digital file can get the effect of handwriting on the paper file.[1]

[1] https://zh.wikipedia.org/zh-tw/%E5%85%AC%E5%BC%80%E5%AF%86%E9%92%A5%E5%8A%A0%E5%AF%86

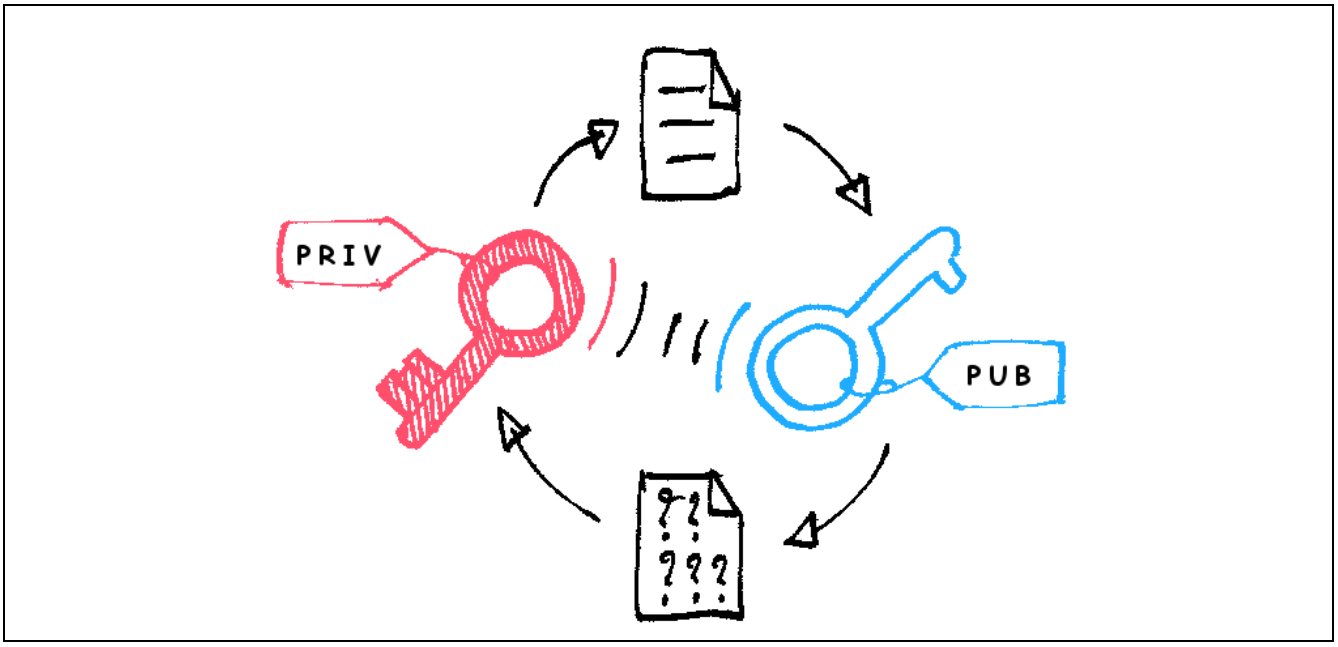

Figure 1-1 One-way Process of Encryption and Decryption[2]

1.1.1 Digital Signature

The digital signature is a method similar to a normal signature written on paper, but using techniques from the field of public key encryption, for authenticating digital messages. A set of digital signatures usually defines two complementary operations, one for signing and the other for verification. Electronic signatures refer to those attached to digital documents, identifying and confirming the identity, qualifications, and authenticity of the digital document signatories; nevertheless, the digital signature is a digital signature formed by encrypting it with a mathematical algorithm or other methods.[3]

Generally speaking, people use the public key to encrypt the message and encrypt it by private key; however, the digital signature is the contrary. In the digital signature, the private key is used to encrypt (equivalent to generating a signature), and the public key is decrypted (equivalent to verifying the signature). The message can be signed directly (that is, encrypted with the private key, and the purpose of encryption at this time is for signature, not for confidentiality), and the verifier decrypts the message correctly with the public key. If it is consistent with the original message, the signature is verified successfully. To greatly improve the efficiency of the signature, usually signed on the hash value of the message, because its length is much smaller than the original message.

[2] https://zh.wikipedia.org/zh-tw/%E5%85%AC%E5%BC%80%E5%AF%86%E9%92%A5%E5%8A%A0%E5%AF%86

[3] https://zh.wikipedia.org/zh-tw/%E6%95%B8%E4%BD%8D%E7%B0%BD%E7%AB%A0

1.2 Principle and Operation of RSA

1.2.1 Generation of Key Pairs

If users want to receive messages over an unreliable medium, they can generate public and private keys in the following ways:

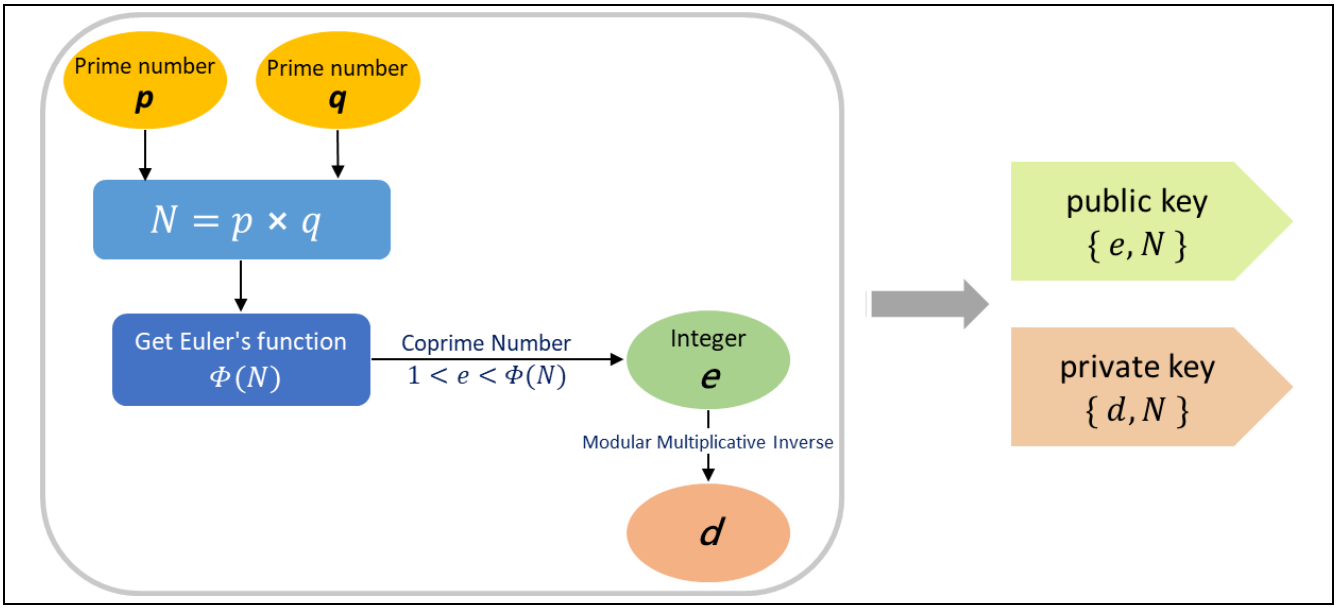

Figure 1-2 Process of Key Generation

1. Choose two large prime numbers *p* and *q* ($p \neq q$), and get the value: $N = pq$.
2. According to the Euler function, get the value: $r = \varphi(N) = \varphi(p)\varphi(q) = (p-1)(q-1)$
3. Choose an integer *e* less than *r*, such that *e* is relatively prime to *r*, satisfy $1 < e < r$, and then find the modulo inverse element of *e* with correspond to *r*, called *d*. (satisfy that: $ed \equiv 1 (\mathrm{mod}\, r)$)
4. Destroy the records of *p* and *q.*

Where (N, e) is the public key and (N, d) is the private key. Users can send their public key to the object to receive the encrypted message and hide the private key.

If anyone uses the public key to encrypt the plaintext, the resulting ciphertext can be transmitted through insecure channels (such as the Internet), and only the user who has the corresponding private key can decrypt the plaintext.

1.2.2 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.[4]

1.2.3 Encryption

PEM (Privacy-Enhanced Mail) is an industry standard for storing and transmitting cryptographic keys, public key certificates and other data file formats. Therefore, PEM is the most common data format for public key and private key.[5]

For the key format of PEM, user can call ***mbedtls_rsa_pkcs1_encrypt()*** in the Mbed TLS library to encrypt the data, the function defined in Table 1-1:

mbedtls_rsa_pkcs1_encrypt(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, size_t ilen, const unsigned char *input, unsigned char *output)		
Parameters :	*ctx	Initialized RSA context
	*f_rng	Ramdom function
	*p_rng	Additional functions of **f_rng* (Can be *NULL* when it unnecessary)
	ilen	Length of the plaintext buffer (Bytes)
	*input	Input data for encryption
	*output	Output buffer
Returns :		Success = 0 Failure = MBEDTLS_ERR_RSA_XXX

Table 1-1 API_mbedtls_rsa_pkcs1_encrypt()

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

[5] https://zh.wikipedia.org/zh-tw/Pem%E6%A0%BC%E5%BC%8F

1.2.4 Decryption

After obtaining the ciphertext, the user can use the private key to decrypt data by calling ***mbedtls_rsa_pkcs1_decrypt()*** from Mbed TLS library, which is defined in Table 1-2.

mbedtls_rsa_pkcs1_decrypt(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, size_t *olen, const unsigned char *input, unsigned char *output, size_t output_max_len)		
Parameters :	*ctx	Initialized RSA context
	*f_rng	Random function
	*p_rng	Additional functions of *f_rng* (Can be *NULL* when it unnecessary)
	*olen	The address at which to store the length of the plaintext.
	*input	The ciphertext buffer
	*output	The buffer used to hold the plaintext
Returns :		Success = 0 Failure = MBEDTLS_ERR_RSA_XXX

Table 1-2 API_mbedtls_rsa_pkcs1_decrypt()

1.2.5 Verification of Digital Signature

RSA can also be used for message signing. When users want to transition a signed message to others, they can calculate a hash value for the message, then encrypt the hash value with the private key and add this signature at the end of the message. The detailed steps are shown in section 1.2.3.

The signed message can only be decrypted by the corresponding public key (the steps are shown in section 0). When users receive the ciphertext, they can use the public key to decrypt the hash value and compare the result of the decryption and the hash value they had calculated. If matched, it means that the sender holds the customer's private key, and the message hasn't been tampered with during the transmission process.

Users can sign the message with the private key by calling ***mbedtls_rsa_pkcs1_sign()*** in the Mbed TLS library, which is defined in Table 1-3. Moreover, users can also operate the verification of the public key by calling ***mbedtls_rsa_pkcs1_verify()***, which is defined in Table 1-4.

mbedtls_rsa_pkcs1_sign(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, mbedtls_md_type_t md_alg, unsigned int hashlen, const unsigned char *hash, unsigned char *sig)		
Parameters :	*ctx	Initialized RSA context
	*f_rng	Random function
	*p_rng	Additional functions of *f_rng*

		(Can be *NULL* when it unnecessary)
	md_alg	Message-digest algorithm used to hash the original data.
	hashlen	Length of the message digest or raw data (Bytes)
	*hash	The buffer holding the message digest or raw data.
	*sig	The buffer to hold the signature.
Returns：		Success = 0 Failure = MBEDTLS_ERR_RSA_XXX

Table 1-3 API_mbedtls_rsa_pkcs1_sign()

mbedtls_rsa_pkcs1_verify(mbedtls_rsa_context *ctx, mbedtls_md_type_t md_alg, unsigned int hashlen, const unsigned char *hash, const unsigned char *sig)		
Parameters：	*ctx	Initialized RSA context
	md_alg	Message-digest algorithm used to hash the original data.
	hashlen	Length of the message digest or raw data (Bytes)
	*hash	The buffer holding the message digest or raw data.
	*sig	The buffer to hold the signature.
Returns：		Success = 0 Failure = MBEDTLS_ERR_RSA_XXX

Table 1-4 API_mbedtls_rsa_pkcs1_verify()

1.3 Demo Result

The demo result is shown below.

```
MBEDTLS RSA self test ...
Pure software crypto running.

  RSA encrypt with hash test start

  mbedtls_pk_parse_key      : passed     takes 0.30 seconds,  30 ticks
  mbedtls_rsa_check_privkey : passed     takes 0.63 seconds,  63 ticks
  PKCS#1 encryption         : passed     takes 0.197 seconds,  197 ticks
  PKCS#1 decryption         : passed     takes 7.7783 seconds,  7783 ticks
  PKCS#1 data sign          : passed     takes 6.6554 seconds,  6554 ticks
  PKCS#1 sig. verify        : passed     takes 0.156 seconds,  156 ticks


  RSA encrypt with hash test done

  Test OK
```

Figure 1-3 Demo Result

2. Code Description

First, check RSA key pairs by using the Mbed TLS library. The code is located in *rsa_test.c*.

```
int ret = 0;

/* Read the RSA key */
if((ret = mbedtls_pk_parse_key(
                  &pk,
                  (const unsigned char *)SSL_USER_PRIV_KEY_PEM,
                  sizeof(SSL_USER_PRIV_KEY_PEM),
                  NULL,  0,
                 myrand, NULL)) != 0)
   {
        printf("  failed\n ! mbedtls_pk_parse_key returned %d\n", ret);
        return -1;
   }

/* check RSA privkey key */
mbedtls_rsa_context *rsa = mbedtls_pk_rsa(pk);
if(mbedtls_rsa_check_privkey(rsa) != 0)
{
     printf("  failed\n !mbedtls_rsa_check_privkey\n");
     return -1;
}

/* check RSA pubkey key */
if(mbedtls_rsa_check_pubkey(rsa) != 0)
{
     printf("  failed\n !mbedtls_rsa_check_pubkey\n");
     return -1;
}
printf(" passed");
g_u32Time = get_timer0_counter();

/* TIMER0->CNT is the elapsed us */
printf("     takes %d.%d seconds,  %d ticks\n", g_u32Time / 1000000, g_u32Time / 1000,
g_u32Ticks);
```

After reading the key pairs, start the encryption and decryption test. The code is located in *rsa_test.c.*

```
printf("  PKCS#1 encryption      : ");

/* RSA encryption */
/* This function adds the message padding, then performs an RSA operation */
if(mbedtls_rsa_pkcs1_encrypt(rsa, myrand, NULL,
                                 PT_LEN, rsa_plaintext,
                                 rsa_ciphertext) != 0)
    {
        if(verbose != 0)
            printf("failed\n");

        ret = 1;
```

```
        goto cleanup;
    }

    if(verbose != 0)
    {
        printf("passed");
        g_u32Time = get_timer0_counter();
        printf("     takes %d.%d seconds,  %d ticks\n", g_u32Time / 1000000, g_u32Time /
1000, g_u32Ticks);
    }

    enable_sys_tick(1000);
    start_timer0();
    printf("  PKCS#1 decryption      : ");

    /* RSA decryption */
    /* This function performs an RSA operation, then removes the message padding */
    if(mbedtls_rsa_pkcs1_decrypt(rsa, myrand, NULL,
                                 &len, rsa_ciphertext, rsa_decrypted,
                                 sizeof(rsa_decrypted)) != 0)
    {
        if(verbose != 0)
            printf("failed\n");

        ret = 1;
        goto cleanup;
    }

    if(memcmp(rsa_decrypted, rsa_plaintext, len) != 0)
    {
        if(verbose != 0)
            printf("failed\n");

        ret = 1;
        goto cleanup;
    }

    if(verbose != 0)
    {
        printf("passed");
        g_u32Time = get_timer0_counter();
        printf("     takes %d.%d seconds,  %d ticks\n", g_u32Time / 1000000, g_u32Time /
1000, g_u32Ticks);
    }
```

Use the RSA private key to sign the message, and verify the digital signature by the public key. The code is located in *rsa_test.c.*

```
if(verbose != 0)
        printf("  PKCS#1 data sign       : ");

    if(mbedtls_sha1(test_image_bin, sizeof(test_image_bin), sha1sum) != 0)
{
        if(verbose != 0)
            printf("failed\n");
```

```
        return(1);
    }
    /* performs a private RSA operation to sign a message digest using PKCS#1 */
    if(mbedtls_rsa_pkcs1_sign(rsa, myrand, NULL,
                              MBEDTLS_MD_SHA1, sha_set_value,
                              sha1sum, rsa_ciphertext) != 0)
    {
        if(verbose != 0)
            printf("failed\n");

        ret = 1;
        goto cleanup;
    }

    if(verbose != 0)
    {
        printf("passed");
        g_u32Time = get_timer0_counter();

        printf("     takes %d.%d seconds,  %d ticks\n", g_u32Time / 1000000, g_u32Time /
1000, g_u32Ticks);
    }
    enable_sys_tick(1000);
    start_timer0();

    printf("  PKCS#1 sig. verify     : ");

    /* performs a public RSA operation and checks the message digest */
    if(mbedtls_rsa_pkcs1_verify(rsa, MBEDTLS_MD_SHA1, sha_set_value,
                                sha1sum, rsa_ciphertext) != 0)
    {
        if(verbose != 0)
            printf("failed\n");

        ret = 1;
        goto cleanup;
    }

    if(verbose != 0)
    {
        printf("passed");
        g_u32Time = get_timer0_counter();
        printf("     takes %d.%d seconds,  %d ticks\n", g_u32Time / 1000000, g_u32Time /
1000, g_u32Ticks);
    }
```

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_RSA_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 *RSA_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.