云途 YUNTU

YUNTU AUTOMOTIVE SEMICONDUCTOR

真车规 高可靠 高性能

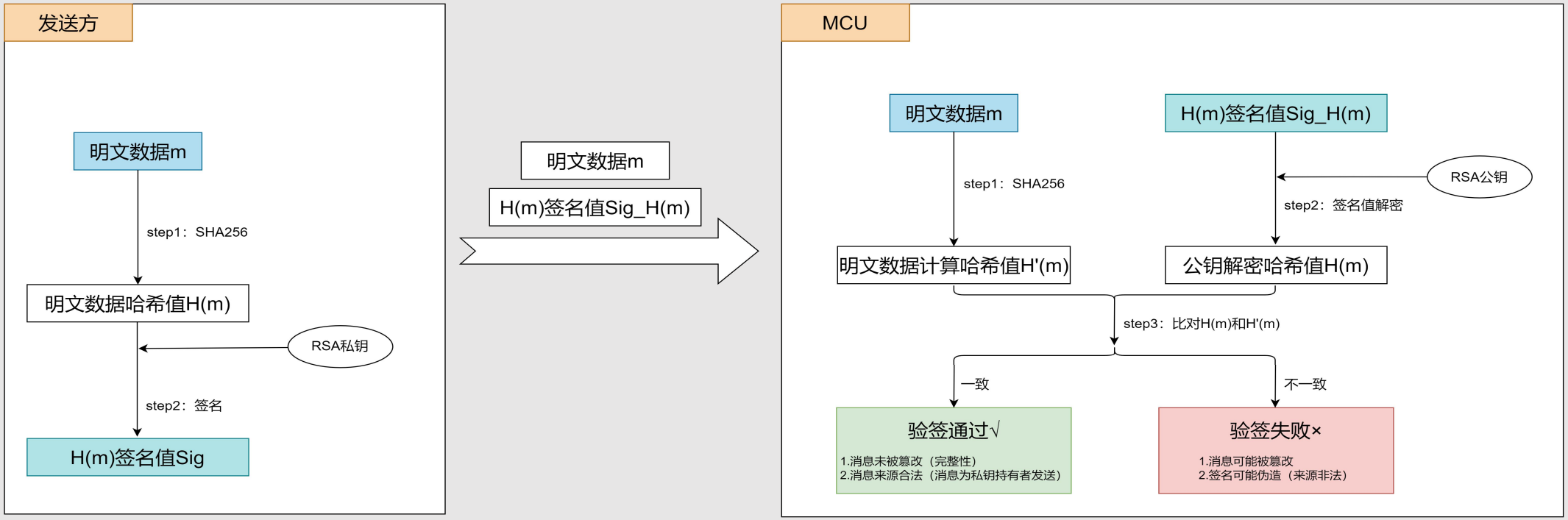
RSA验签原理及软件实现

目录

CONTENTS

（一）RSA验签介绍

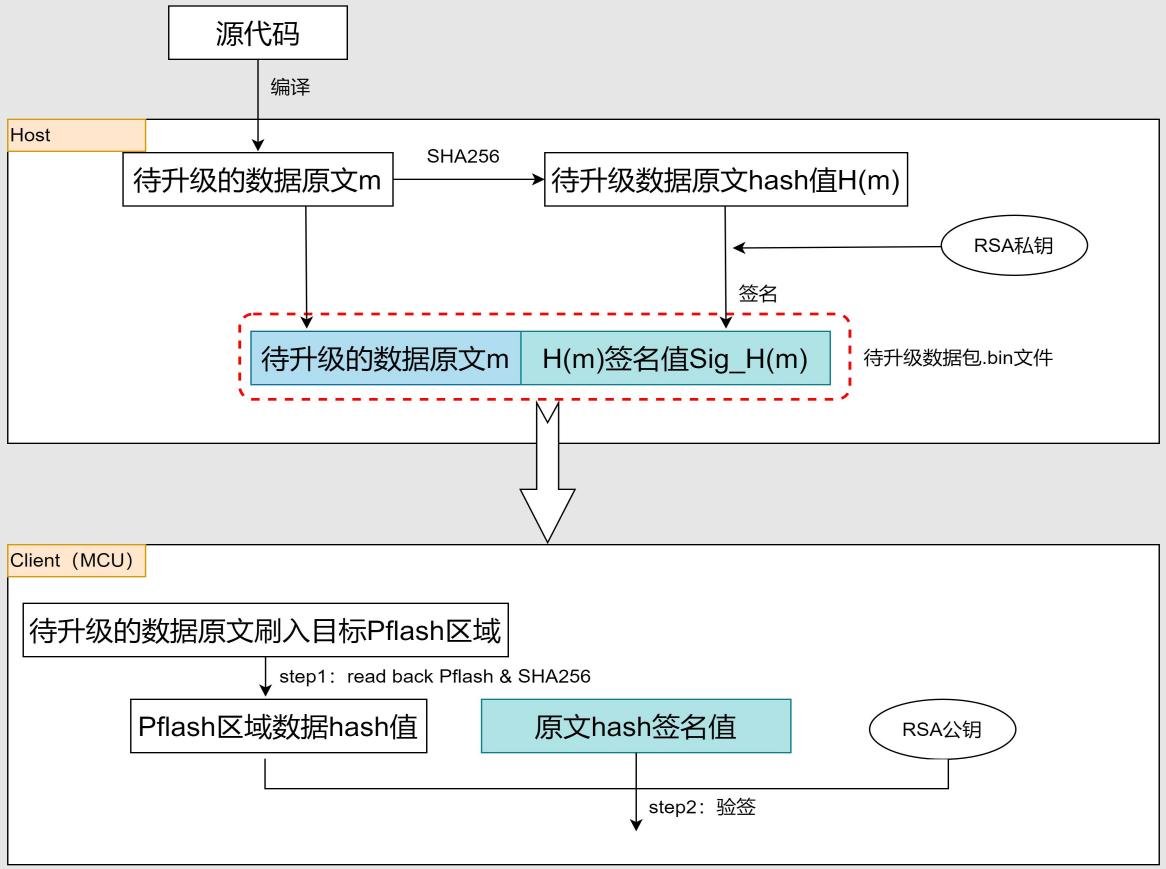
RSA是基于大整数因式分解难题的非对称加密算法，应用于信息安全领域，核心功能是让接收方确认消息的完整性和消息来源的合法性。



- RSA密钥：由发送方提供公、私钥，私钥（发送方唯一持有）签名，公钥（发送方提供给接收方）验签
- 消息完整性：明文数据m正确且完整传输到接收方，才能确保接收方使用明文数据m计算的hash值H’ (m)与H(m)一致，验签通过可证明消息的完整性
- 消息来源合法性：由私钥签名的明文数据hash值H(m)，必须使用相应的公钥才能正确解密出明文的hash值，验签通过可证明用于签名的公钥和用于验签的私钥为有效的密钥对，消息来源合法

（二）RSA验签应用介绍

ECU开发过程中，RSA通常用于固件的OTA升级场景，即在固件刷写完成后回读Pflash对固件进行验签，验签通过后复位运行。



- Host：通常由主机厂提供

（1）RSA公、私钥：私钥绝对保密，仅由主机厂持有，公钥提供给供应商

（2）待升级数据原文hash值H(m)：由数据原文m计算SHA256获得

（3）H(m)签名值Sig_H(m)：由host计算，随待升级数据原文m传输给MCU

- Client（MCU）

（1）step1：固件刷写后计算对应Pflash区域SHA256值H’（m)

（2）step2：调用RSA软件验签接口函数，输入：

①Pflash区域数据hash值H’（m)、

②host提供的hash签名值Sig_H(m)、

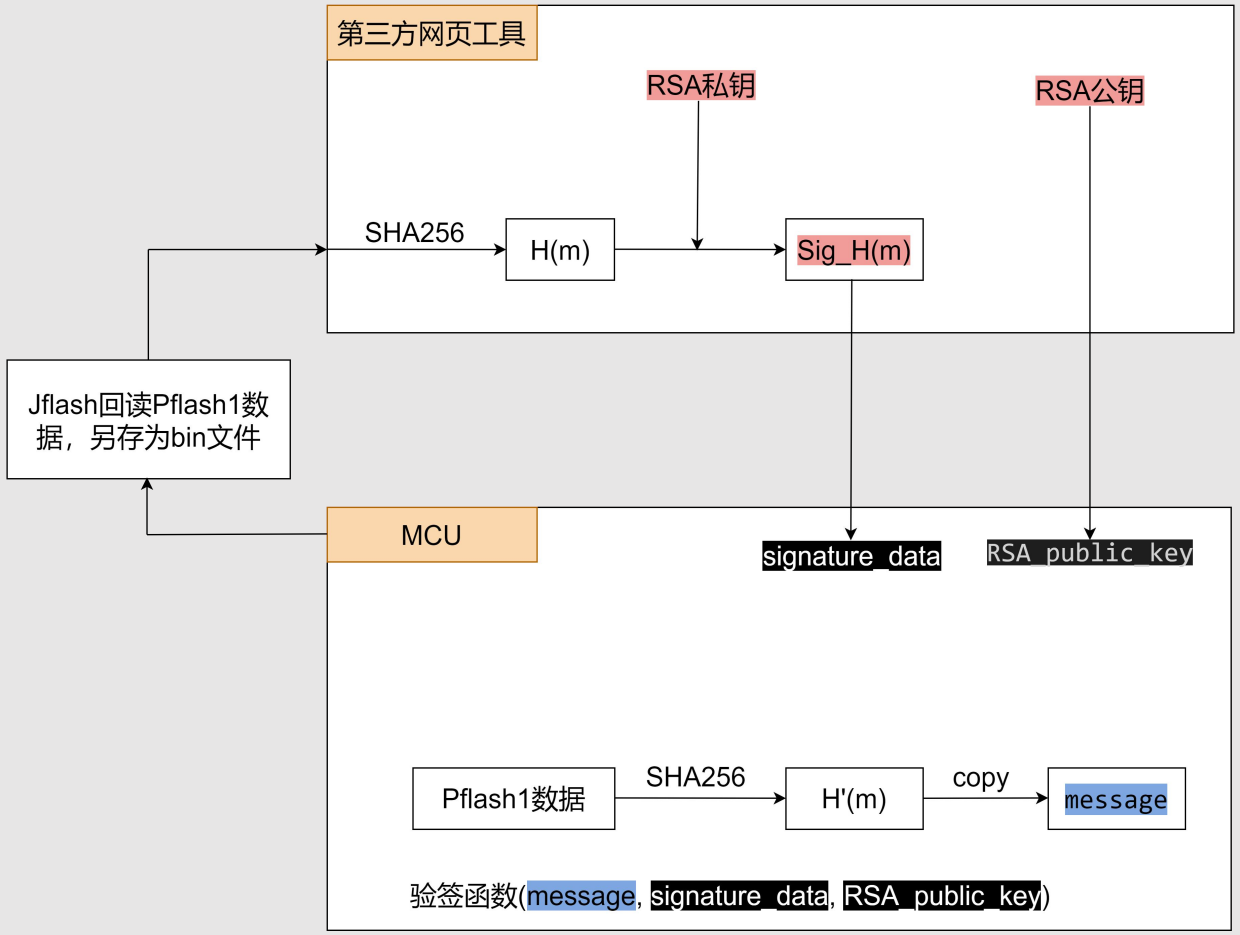
③host提供的RSA公钥

最终根据函数返回值判断是否验签通过。

（三）RSA验签应用demo实现——概述

本文档提供一个基于ME05 EVB板、YCT工具MbedTls_with_PSA_Demo工程实现的RSA4096验签示例。

它由网页端工具生成公、私钥，并假定MCU的Pflash1区域为升级数据（均为0xFF），用户可参考本示例验证或移植。



- 网页端工具

（1）生成RSA公、私钥

（2）对Pflash1区域数据算SHA256得到H(m)，私钥签名获得Der编码格式的签名值Sig_H(m)

- MCU

（1）对Pflash1数据SHA256计算得到H’（m），将H’（m)值填入数组***message***，作为待验签消息。

（2）将Sig_H(m)填入***signature_data_upper_der***

（3）导入公钥***RSA_public_key***，用***psa_verify_message***函数进行验签。

NOTE:

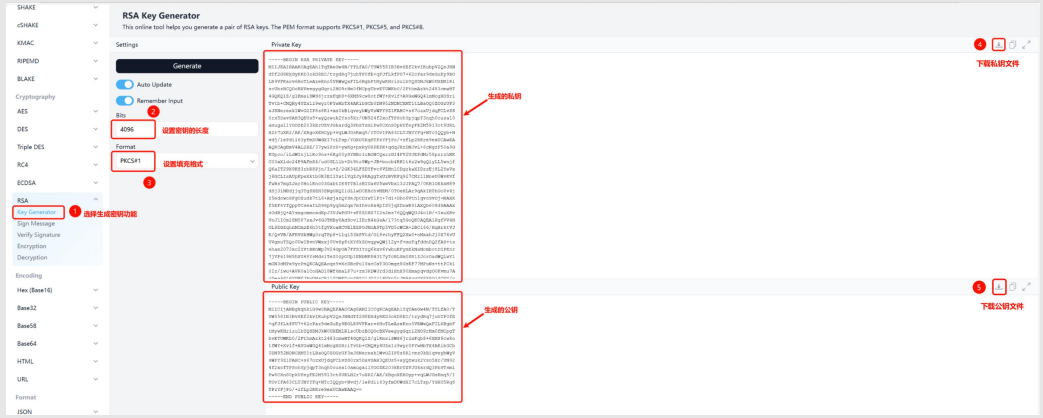
（1）当前demo软件仅支持SHA256格式的验签

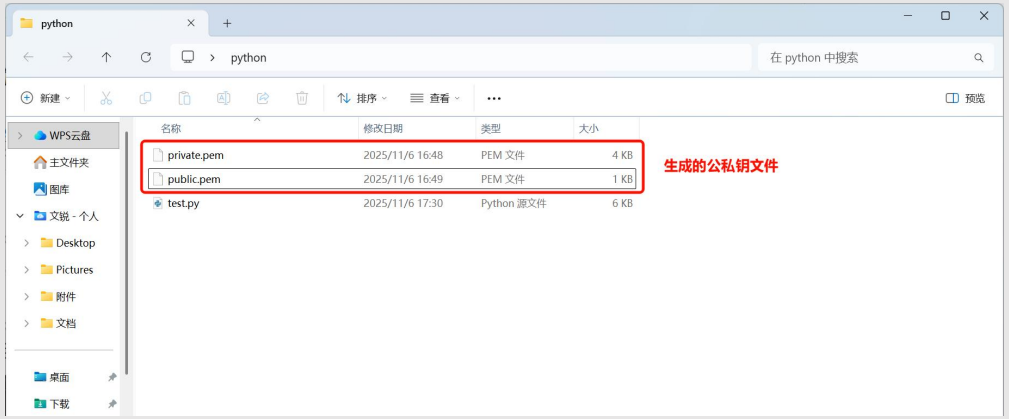
（2）网页端工具计算的H(m)值与MCU端计算的H’（m)值应当一致

（3）ME05的SHA256可通过mbedtls密码库软件实现或HCU模块硬件实现

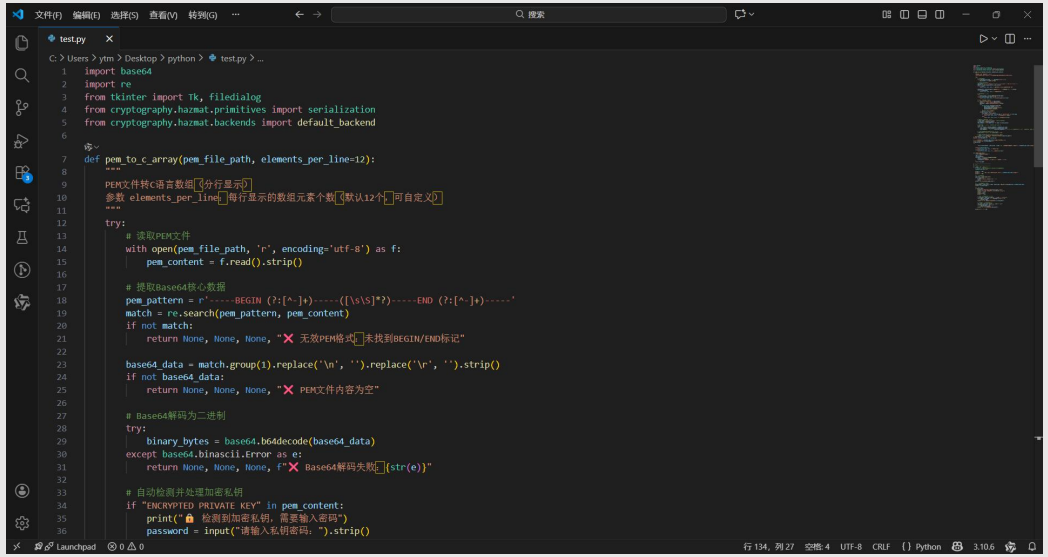
（三）RSA验签应用demo实现——网页端工具生成公、私钥

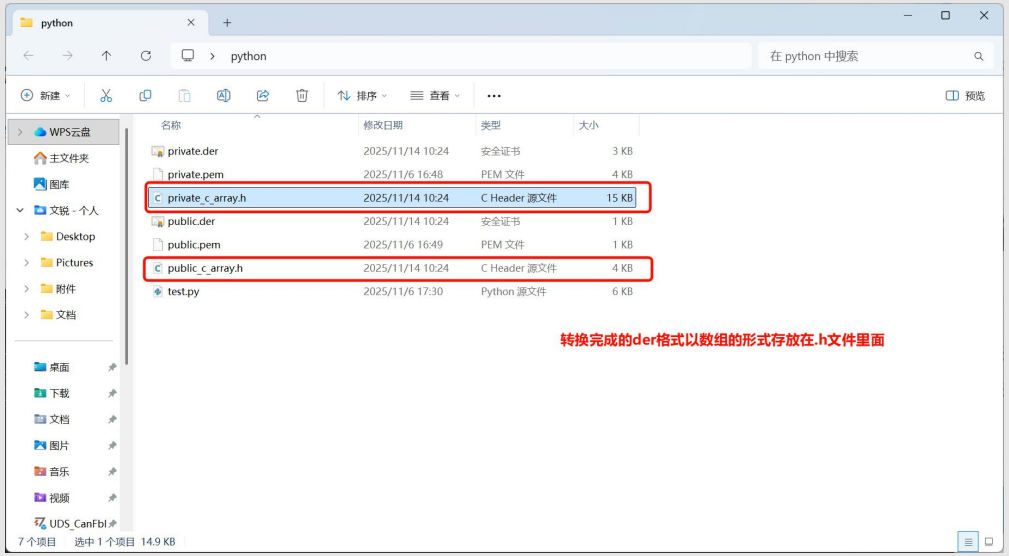
1.使用网页端工具（https://emn178.github.io/online-tools/rsa/key-generator/）生成公私钥。





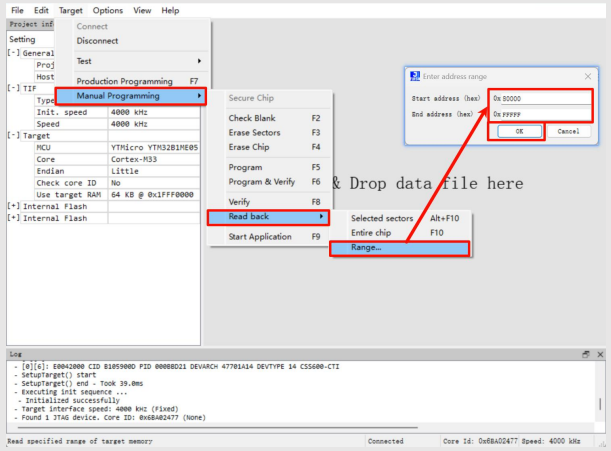
2.使用提供的python脚本将pem格式的文件转换成数组格式

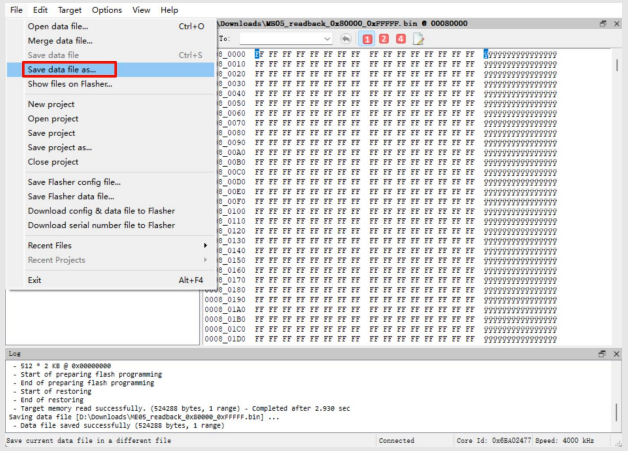




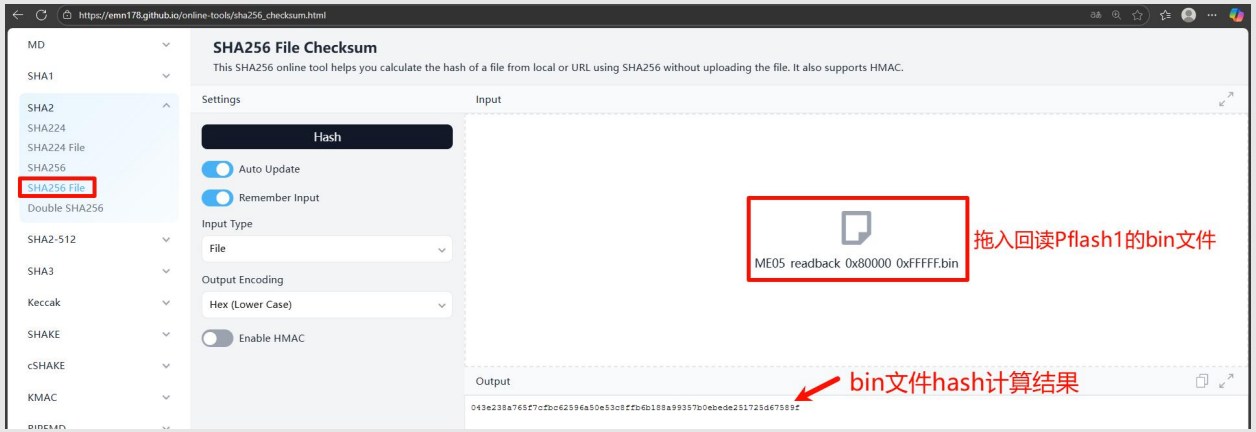
（三）RSA验签应用demo实现——回读Pflash1数据并由网页端工具计算hash值

将示例工程烧录到ME05芯片中，使用Pflash1中的数据作为测试数据，通过Jflash软件回读Pflash1区域的数据并保存为bin文件（该区域数据全为0xFF）。



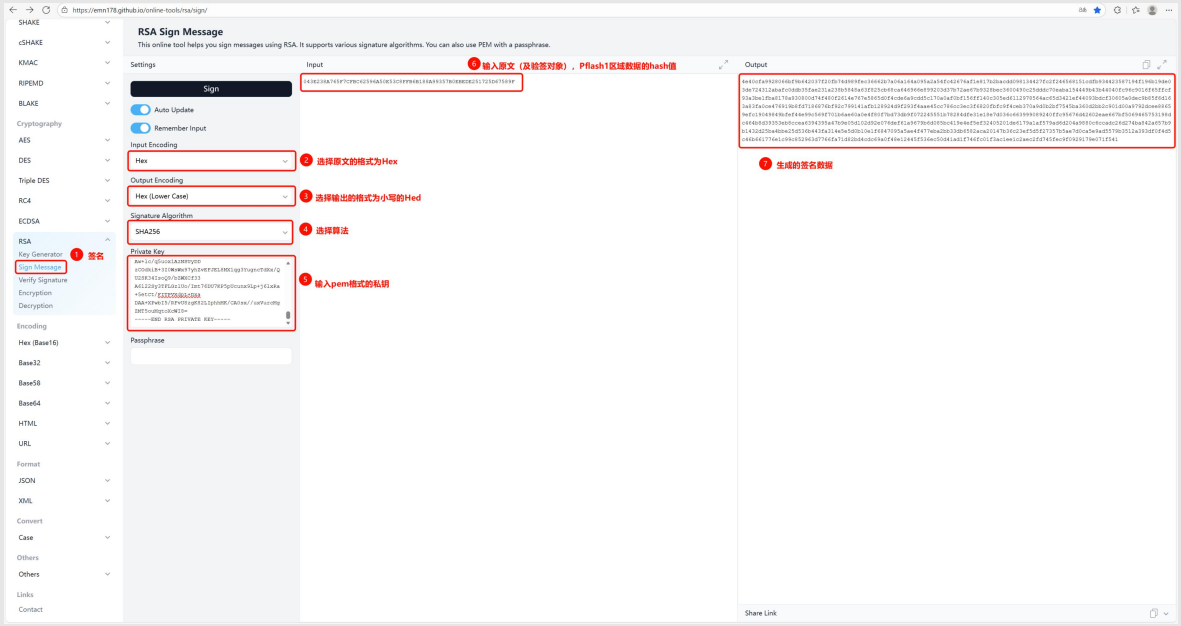


使用网页端工具（https://emn178.github.io/online-tools/sha256_checksum.html）计算bin文件的hash值。



（三）RSA验签demo实现——网页端工具使用私钥对hash值签名

使用网页端工具（https://emn178.github.io/online-tools/rsa/sign/）对hash值签名生成签名值。



NOTE:

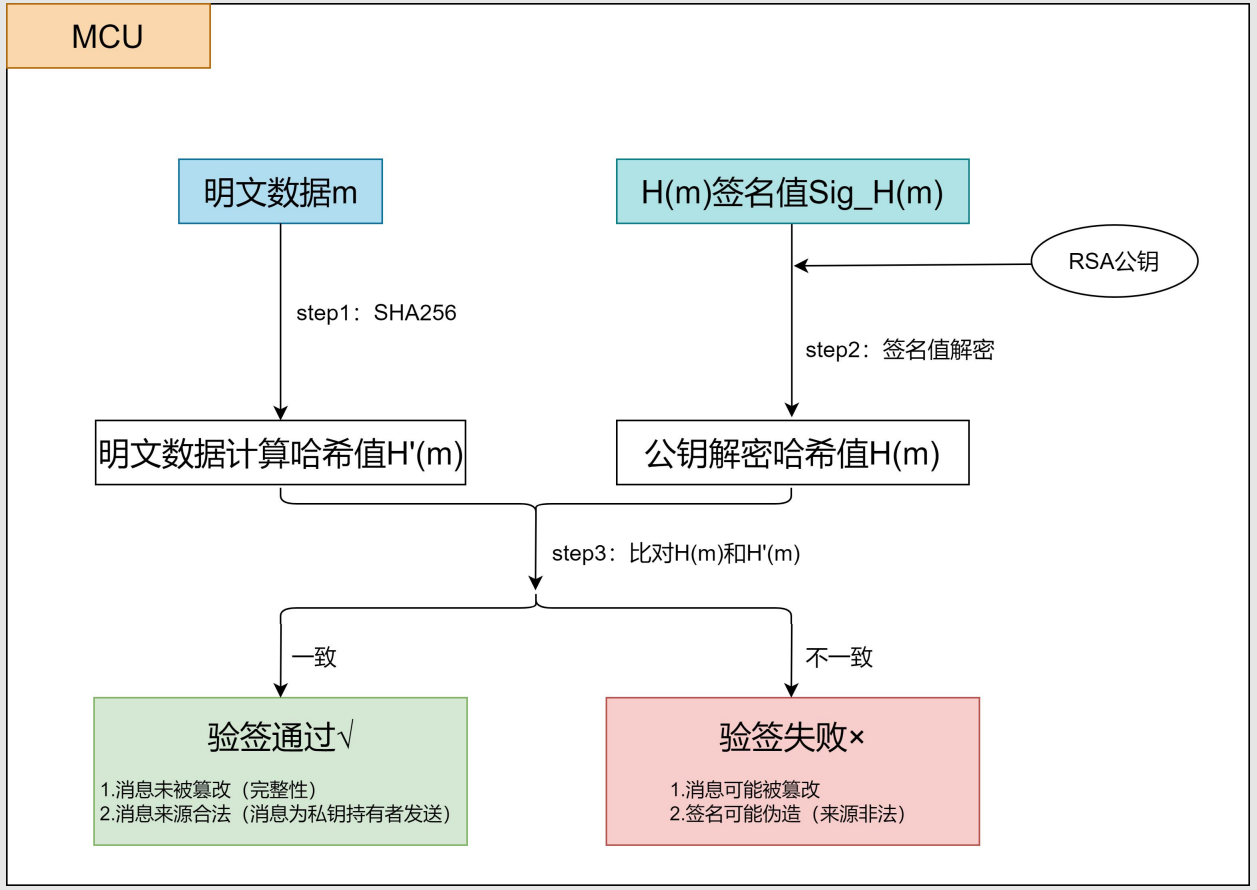
本示例使用PKCS#1 v1.5标准，使用同一个私钥对同一个明文多次签名时，每次生成的签名值相同。

工具签名生成的某个签名值（der编码格式）：

4e40cfa9928066bf9b642037f20fb74d989fec36662b7a06a164a095a2a54fc426
76af1e817b2bacdd098134427fc2f246568151cdfb934423587194f196b19de03
de724312abafc0ddb35fae231a238b5848a63f825cb68ca646966e899203d37b
72ae67b9328bec3600490c25dddc70eaba154449b43b44040fc96c9016f65ffcf9
3a3be1fba8178a930800d74f480f2614e767e5865d0f4cde6a9cdd5c170a0af0bf
156ff140c305ed6112978564ac65d3421ef44093bdcf30605a0dec9b85f6d163a
83fa0ce476919b8fd7186876bf92c799141afb128924d9f293f4aae45cc786cc3e
c3f6820fbfc9f4ceb370a9d0b2bf7545ba360d2bb2c901d00a9792dcee88659efc
19049849bfef44e99c569f701b6ae60a0e4f80f7bd73db9f072245551b78284dfe
31e18e7d036c663999089240ffc95676d42602eae667bf5069465753198dc464
b8d39353eb8ccea6394395a47b9e05d102d92e076def61a9679b6d085bc419e
4ef5ef32405201de6179a1af579ad6d204a9880c6ccadc26d274ba842a657b9b1
432d25ba4bbe25d536b443fa314e5e5d0b10e1f6847095a5ae4f477eba2bb33d
b6582aca20147b36c23ef5d5f27357b5ae7d0ca5e9ad5579b3512a393df0f4d5c
46b661776e1c99c852963d7766fa71d82bd4cdc69a0f48e12445f536ec50d41ad
1f746fc01f3ac1ee1c2aec2fd745fec9f0929179e071f541

（三）RSA验签demo实现——MCU端验签流程

如下图，”明文数据m“是将Pflash1的全部数据计算成32Byte的哈希值，而”H（m）签名值Sig_H（m）则是通过网页端的签名工具将”明文数据“通过私钥生成签名值。



（三）RSA验签demo实现——明文数据m（MCU端计算Pflash1数据hash值）

如下图，设置需要进行hash计算的起始地址和长度。其中：

（1）HASH_FLASH_STARTADDRESS为ME05 Pflash1的起始地址，注意避免跨 4 字节边界，建议使用sector的起始地址。

（2）ONECE_READ_FLASH_SIZE为分块计算hash时每一块的大小，建议选sector的整数倍，此处设为8K，用户可调整。

（3）READ_SECTOR_NUM为需要计算hash的整个Plfash大小，建议选sector的整数倍，此处设为512K，用户可调整。

```
#define HASH_FLASH_STARTADDRESS       (0x80000)
#define ONECE_READ_FLASH_SIZE         (8 * 1024)
#define READ_SECTOR_NUM               (512 * 1024)
```

- 方式1：mbedtls库软件计算hash

调用***StepReadFlash_And_SoftWare_CalculateSHA***函数软件计算Pflash1区域数据的hash值，结果存储在***swshaResult***数组，将其copy到***message***数组，验签时使用。

- 方式2：HCU模块SHA256硬件计算hash

调用***StepReadFlash_And_HardWare_CalculateSHA***函数硬件计算Pflash1区域数据的hash值，结果存储在***hwshaResult***数组，将其copy到***message***数组，验签时使用。

NOTE：ME05的HCU支持SHA256，才能使用方式2计算hash，需要在配置工具中开启HCU_CLK，并使能HCU模块。方式1更具有通用性。

（三）RSA验签demo实现——明文数据计算哈希值H'(m)

```
#if READ_FLASH_TEST_APPLICATION
    //初始化mbedtls sha56
    PINS_DRV_SetPins(base: GPIOE, pins: (1 << 23));
    #if hash_calculate_MODE
        // 1. 读取Flash并使用硬件计算SHA256作为明文
        StepReadFlash_And_HardWare_CalculateSHA();
        PINS_DRV_ClearPins(base: GPIOE, pins: (1 << 23));
        //将硬件计算的hash值填到本工程的message数组
        for(uint8_t i = 0; i < 32; i++){
            message[i] = hwshaResult[i];
        }

        // 2. 计算明文的hash（SHA256）值
        HCU_DRV_GenerateSHA(msg: message, msgLen: sizeof(message), totalLen: sizeof(message), shaType: HCU_SHA_256, msgType: MSG_ALL, result: hashResult);
    #else
        PINS_DRV_SetPins(GPIOE, (1 << 23));
        // 1. 读取Flash并使用软件计算SHA256作为明文
        StepReadFlash_And_SoftWare_CalculateSHA();
        PINS_DRV_ClearPins(GPIOE, (1 << 23));
        //将软件计算的hash值填到本工程的message数组
        for(uint8_t i = 0; i < 32; i++){
            message[i] = swshaResult[i];
        }

        // 2. 计算明文的hash（SHA256）值
        Message_And_SoftWare_CalculateSHA(message, sizeof(message), hashResult);
    #endif
#endif
```

硬件计算Pflash1里数据的hash（SHA256）值作为明文

硬件计算明文的hash（SHA256）值

软件计算Pflash1里数据的hash（SHA256）值作为明文

软件计算明文的hash（SHA256）值

- 方式1：mbedtls库软件计算hash

调用***Message_And_SoftWare_CalculateSHA***函数软件计算明文的hash值，结果存储在***hashResult***数组，验签时使用。

- 方式2：HCU模块SHA256硬件计算hash

调用***HCU_DRV_GenerateSHA***函数硬件计算明文的hash值，结果存储在***hashResult***数组，验签时使用。

（三）RSA验签demo实现——公钥的导入与解析

```c
// 3. 解析数据
PRINTF(format: "Parsing RSA DER...\n\r");
int parse_ret = mbedtls_pk_parse_public_key(
    ctx: &pk_ctx,
    key: RSA_public_key_4096_der,
    keylen: sizeof(RSA_public_key_4096_der)
);

if (parse_ret != 0) {
    PRINTF(format: "公钥解析失败\n");
    status = parse_ret;
    mbedtls_pk_free(ctx: &pk_ctx);
    return status;
}

RSA_Context = mbedtls_pk_rsa(pk: pk_ctx);

// 4. 设置填充方式和哈希算法
PRINTF(format: "Setting RSA padding...\n\r");
mbedtls_rsa_set_padding(ctx: RSA_Context, padding: MBEDTLS_RSA_PKCS_V15, hash_id: MBEDTLS_MD_SHA256);
if(parse_ret != 0)
{
    PRINTF(format: "设置 RSA 填充方式失败\n");
    status = parse_ret;
    mbedtls_pk_free(ctx: &pk_ctx);
    mbedtls_rsa_free(ctx: RSA_Context);
    return status;
}

// 5. 完成 RSA 公钥结构
PRINTF(format: "Completing RSA public key structure...\n\r");
mbedtls_rsa_complete(ctx: RSA_Context);
if(parse_ret != 0)
{
    PRINTF(format: "完成 RSA 公钥结构失败\n");
    status = parse_ret;
    mbedtls_pk_free(ctx: &pk_ctx);
    mbedtls_rsa_free(ctx: RSA_Context);
    return status;
}
```

公钥的导入和解析主要分为三部：

- 第一步：调用***mbedtls_pk_parse_public_key***函数解析公钥，并通过***mbedtls_pk_rsa***函数提取出RSA的相关内容（模数、指数）存放在RSA_Context里面
- 第二步：调用***mbedtls_rsa_set_padding***设置签名数据的填充方式和哈希算法
- 第三步：调用***mbedtls_rsa_complete***函数验证导入参数是否正确并补全缺失的参数

（三）RSA验签demo实现——签名数据的验签

```
// 6. 验证签名
PRINTF(format: "Verifying RSA signature...\n\r");
parse_ret = mbedtls_rsa_pkcs1_verify(ctx: RSA_Context,
                                     md_alg: MBEDTLS_MD_SHA256,
                                     hashlen: 32,
                                     hash: hashResult,
                                     sig: signature_data);
if(parse_ret != 0)
{
    PRINTF(format: "RSA 签名验证失败\n");
}
else
{
    PRINTF(format: "RSA 签名验证成功\n");
}
mbedtls_pk_free(ctx: &pk_ctx);
mbedtls_rsa_free(ctx: RSA_Context);
```

签名数据的验签是通过比对明文数据哈希值的H'(m)和公钥解密哈希值H(m)来决定验签是否通过，MCU验签的实现则是通过***mbedtls_rsa_pkcs1_verify***函数来实现。

最后调用***mbedtls_pk_free***函数和***mbedtls_rsa_free***函数释放***pk_ctx***和***RSA_Context***。

（三）RSA验签demo实现——MCU端总体流程代码

```c
    // 1. 初始化mbedtls的配置并填充明文数据
    mbedtls_config_init();
    mbedtls_pk_init(ctx: &pk_ctx);

#if READ_FLASH_TEST_APPLICATION
    //初始化mbedtls sha56
    PINS_DRV_SetPins(base: GPIOE, pins: (1 << 23));
    #if hash_calculate_MODE
        // 1. 读取Flash并使用硬件计算SHA256作为明文
        StepReadFlash_And_HardWare_CalculateSHA();
        PINS_DRV_ClearPins(base: GPIOE, pins: (1 << 23));
        //将硬件计算的hash值填到本工程的message数组
        for(uint8_t i = 0; i < 32; i++){
            message[i] = hwshaResult[i];
        }

        // 2. 计算明文的hash（SHA256）值
        HCU_DRV_GenerateSHA(msg: message,
                            msgLen: sizeof(message),
                            totalLen: sizeof(message),
                            shaType: HCU_SHA_256,
                            msgType: MSG_ALL,
                            result: hashResult);
    #else
        PINS_DRV_SetPins(GPIOE, (1 << 23));
        // 1. 读取Flash并使用软件计算SHA256作为明文
        StepReadFlash_And_SoftWare_CalculateSHA();
        PINS_DRV_ClearPins(GPIOE, (1 << 23));
        //将软件计算的hash值填到本工程的message数组
        for(uint8_t i = 0; i < 32; i++){
            message[i] = swshaResult[i];
        }

        // 2. 计算明文的hash（SHA256）值
        Message_And_SoftWare_CalculateSHA(message, sizeof(message), hashResult);
    #endif
#endif


    // 3. 解析数据
    PRINTF(format: "Parsing RSA DER...\n\r");
    int parse_ret = mbedtls_pk_parse_public_key(
        ctx: &pk_ctx,
        key: RSA_public_key_4096_der,
        keylen: sizeof(RSA_public_key_4096_der)
    );

    if (parse_ret != 0) {
        PRINTF(format: "公钥解析失败\n");
        status = parse_ret;
        mbedtls_pk_free(ctx: &pk_ctx);
        return status;
    }

    RSA_Context = mbedtls_pk_rsa(pk: pk_ctx);
```

```c
// 4. 设置填充方式和哈希算法
PRINTF(format: "Setting RSA padding...\n\r");
mbedtls_rsa_set_padding(ctx: RSA_Context,
                        padding: MBEDTLS_RSA_PKCS_V15,
                        hash_id: MBEDTLS_MD_SHA256);
if(parse_ret != 0)
{
    PRINTF(format: "设置 RSA 填充方式失败\n");
    status = parse_ret;
    mbedtls_pk_free(ctx: &pk_ctx);
    mbedtls_rsa_free(ctx: RSA_Context);
    return status;
}

// 5. 完成 RSA 公钥结构
PRINTF(format: "Completing RSA public key structure...\n\r");
mbedtls_rsa_complete(ctx: RSA_Context);
if(parse_ret != 0)
{
    PRINTF(format: "完成 RSA 公钥结构失败\n");
    status = parse_ret;
    mbedtls_pk_free(ctx: &pk_ctx);
    mbedtls_rsa_free(ctx: RSA_Context);
    return status;
}

// 6. 验证签名
PRINTF(format: "Verifying RSA signature...\n\r");
parse_ret = mbedtls_rsa_pkcs1_verify(ctx: RSA_Context,
                                     md_alg: MBEDTLS_MD_SHA256,
                                     hashlen: 32,
                                     hash: hashResult,
                                     sig: signature_data);
if(parse_ret != 0)
{
    PRINTF(format: "RSA 签名验证失败\n");
}
else
{
    PRINTF(format: "RSA 签名验证成功\n");
}
mbedtls_pk_free(ctx: &pk_ctx);
mbedtls_rsa_free(ctx: RSA_Context);
/* USER CODE END 2 */
```

（四）各系列芯片RSA4096验签时间测量

RSA验签时间主要与以下两个操作相关，其中：

（1）Hash计算，消耗时间T(h)主要与hash计算方式（软件或硬件）和目标数据的size有关。

（2）软件验签操作，消耗时间T(v)主要与芯片系列（性能）有关。

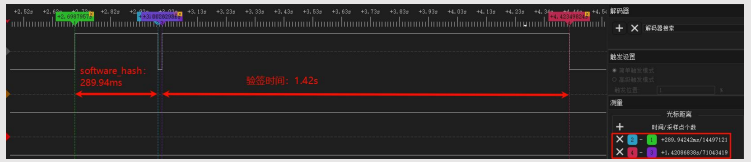
以**ME05**示例工程为例，IDE使用的是**Vscode+GCC**，工程编译等级：**O1**，目标数据size为**512K**，测量时间通过下图所示的IO翻转测量时间。

```c
#if READ_FLASH_SOFTWARE_SHA256 // 软件hash
    PRINTF(format: "SoftWare_Hash!\n");
    PINS_DRV_SetPins(base: GPIOE, pins: 1 << 23);
    //初始化mbedtls sha56
    mbedtls_sha256_init(ctx: &Mbedtls_Context);
    StepReadFlash_And_SoftWare_CalculateSHA();
    PINS_DRV_ClearPins(base: GPIOE, pins: 1 << 23);
    //将软件计算的hash值填到message数组
    for(uint8_t i = 0; i < 32; i++){
        message[i] = swshaResult[i];
    }
#else // 硬件hash
    PRINTF("HardWare_Hash!\n");
    PINS_DRV_SetPins(GPIOE, 1 << 23);
    //初始化HCU模块
    HCU_DRV_Init(&hcu_config0,&hcu_config0_State);
    StepReadFlash_And_HardWare_CalculateSHA();
    PINS_DRV_ClearPins(GPIOE, 1 << 23);
    //将硬件计算的hash值填到message数组
    for(uint8_t i = 0; i < 32; i++){
        message[i] = hwshaResult[i];
    }
#endif
```

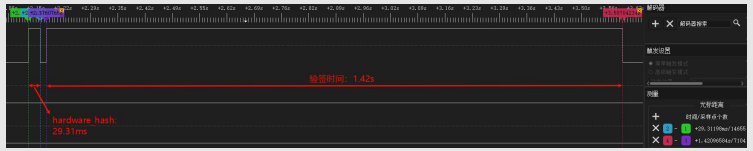
hash计算时间T(h)

```c
// 7. 验证签名
PRINTF(format: "Verifying RSA signature...\n\r");
PINS_DRV_SetPins(base: GPIOE, pins: (1 << 23));
parse_ret = mbedtls_rsa_pkcs1_verify(ctx: RSA_Context,
                                     md_alg: MBEDTLS_MD_SHA256,
                                     hashlen: 32,
                                     hash: hashResult,
                                     sig: signature_data);
PINS_DRV_ClearPins(base: GPIOE, pins: (1 << 23));
if(parse_ret != 0)
{
    PRINTF(format: "RSA 签名验证失败\n");
}
else
{
    PRINTF(format: "RSA 签名验证成功\n");
}
```

RSA验签时间T(v)



软件hash + 软件验签



硬件hash + 软件验签

```
[build] [140/145  93% :: 9.671] Building C object CMakeFiles/mbedtls.dir/middleware/mbedtls-3.6.2/li
x509write_crt.c.o
[build] [140/145  93% :: 9.759] Building C object CMakeFiles/mbedtls.dir/middleware/mbedtls-3.6.2/li
c.o
[build] [140/145  94% :: 9.768] Building C object CMakeFiles/mbedtls.dir/middleware/mbedtls-3.6.2/li
c.o
[build] [140/145  95% :: 9.885] Building C object CMakeFiles/MbedTls_with_RSA_Demo.elf.dir/app/SHA25
[build] [140/145  95% :: 9.975] Building C object CMakeFiles/MbedTls_with_RSA_Demo.elf.dir/app/main.
[build] [140/145  96% :: 10.085] Building C object CMakeFiles/UTILITY_PRINT.dir/middleware/utility_p
printf.c.o
[build] [141/145  97% :: 11.431] Linking C static library libUTILITY_PRINT.a
[build] [142/145  97% :: 11.725] Linking C static library libmbedtls.a
[build] [143/145  98% :: 11.956] Linking C static library libGENERATED_SDK_TARGET.a
[build] [144/145  99% :: 12.173] Linking C static library libGENERATED_CONFIG_TARGET.a
[build] [145/145 100% :: 13.091] Linking C executable MbedTls_with_RSA_Demo.elf
[build]    text    data     bss     dec     hex filename
[build]   56788     600   45408  102796   1918c MbedTls_with_RSA_Demo.elf
[driver] 生成完毕: 00:00:13.271
[build] 生成已完成，退出代码为 0
```

工程文件大小

- 软件hash + 软件验签：289.94ms + 238ms
- 硬件hash + 软件验签：29.31ms + 238ms

（四）各系列芯片RSA4096验签时间测量

Hash：截止到2025.11.9，已量产芯片中仅ME05，HA01支持硬件SHA256。MC03，MD14需使用软件hash。

RSA验签：截止到2025.11.9，已量产芯片均未支持硬件RSA验签。

沿用上页ME05的时间性能测法，测量各芯片针对自身一半Pflash数据的hash计算及验签操作消耗的时间。

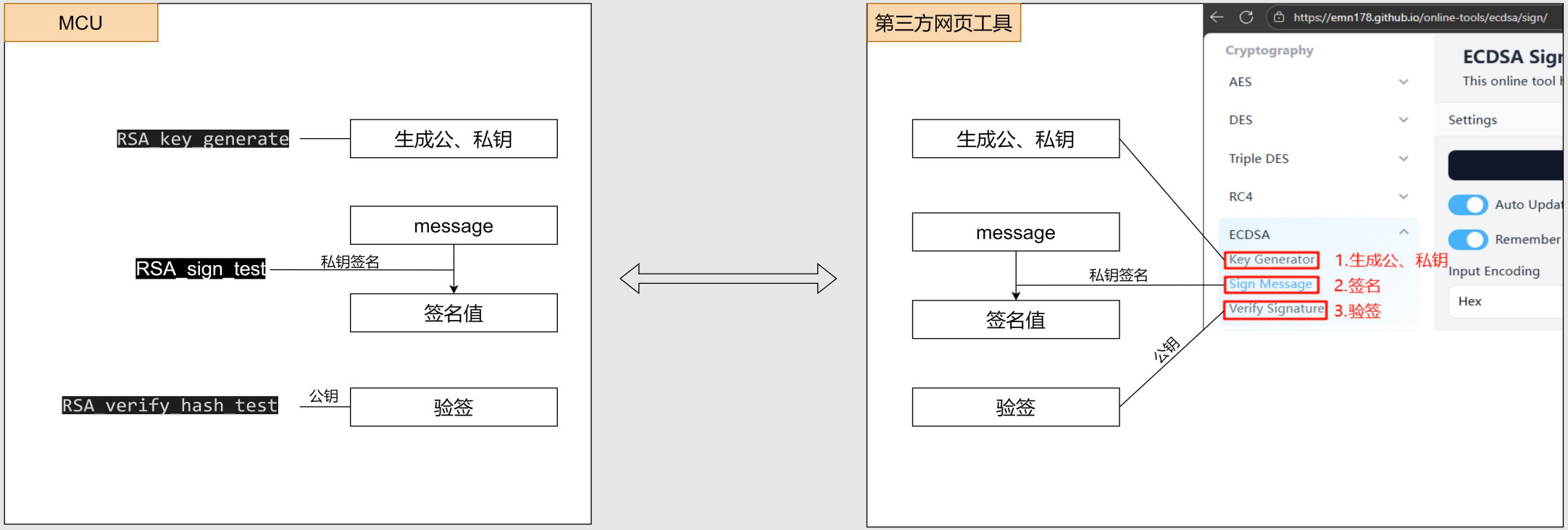
芯片系列	MCU主频/Mhz	Hash计算时间/ms			RSA验签时间(mbedtls库)/ms	总时间/ms
		hash计算空间/K	hash计算实现	hash计算时间/ms		
MC03	80	128	软件hash	116.55	382	498.55
MD14	120	256	软件hash	193.26	323	516.26
ME05	120	512	软件hash	289.94	238	522.94
			硬件hash	29.31	238	262.31
HA01	200	1024	软件hash（开Cache）	162.11	70	232.11
			硬件hash（开Cache）	32.53	70	102.53
			软件hash	811.31	496	1307.31
			硬件hash	107.27	496	603.27

NOTE：上述的测试工程IDE均使用的是Vscode+GCC，工程编译等级：O1

（五）YCT工具MbedTls_with_PSA_Demo用法补充（RSA）

NOTE：本章节是为了保证内容完整性所做的补充介绍，实际应用中公、私钥生成，签名操作均无需在MCU端完成，不需要的用户可忽略该章节

mbedtls库具有完整的MCU端的公私钥生成、message签名、RSA验签接口函数，用户可以在MCU端和第三方工具端进行多种组合交叉验证。



（五）MbedTls_with_PSA_Demo玩法补充（RSA）

例如：MCU端生成公、私钥，对message签名，再将公钥、message、message的签名值放在网页端工具进行验签

交叉验证要点：

（一）message可选十六进制字符串或ASCII字符串

选择ASCII字符串时，MCU端对message签名填入长度时注意填sizeof(message)-1，原因是sizeof会将字符串结尾空字符 '\0'计算在内，直接填sizeof(message)会导致网页端验签不通过。

```
145
146     // 2. 计算明文的 SHA-256 哈希值
147     mbedtls_sha256_starts(ctx: &Mbedtls_Context, is224: 0);
148     mbedtls_sha256_update(ctx: &Mbedtls_Context,
149                           input: message,
150                           ilen: sizeof(message));
151     mbedtls_sha256_finish(ctx: &Mbedtls_Context, output: hashResult);
152
```

1 当明文为16进制数字时，长度为sizeof(message)

2 当明文为字符串时，长度为sizeof(message)-1

THANK YOU