

Large APP demo：用 20 KiB 内存池校验 400 KiB 数据

这个 demo 在 ME0 MCU 上演示：从 Flash 读取约 400 KiB 数据，每次处理 1 KiB，最终通过 RSA4096 公钥验证整段数据的签名。Mbed TLS 内存池固定为 20480 字节，无需申请能装下整个 APP 的 RAM。

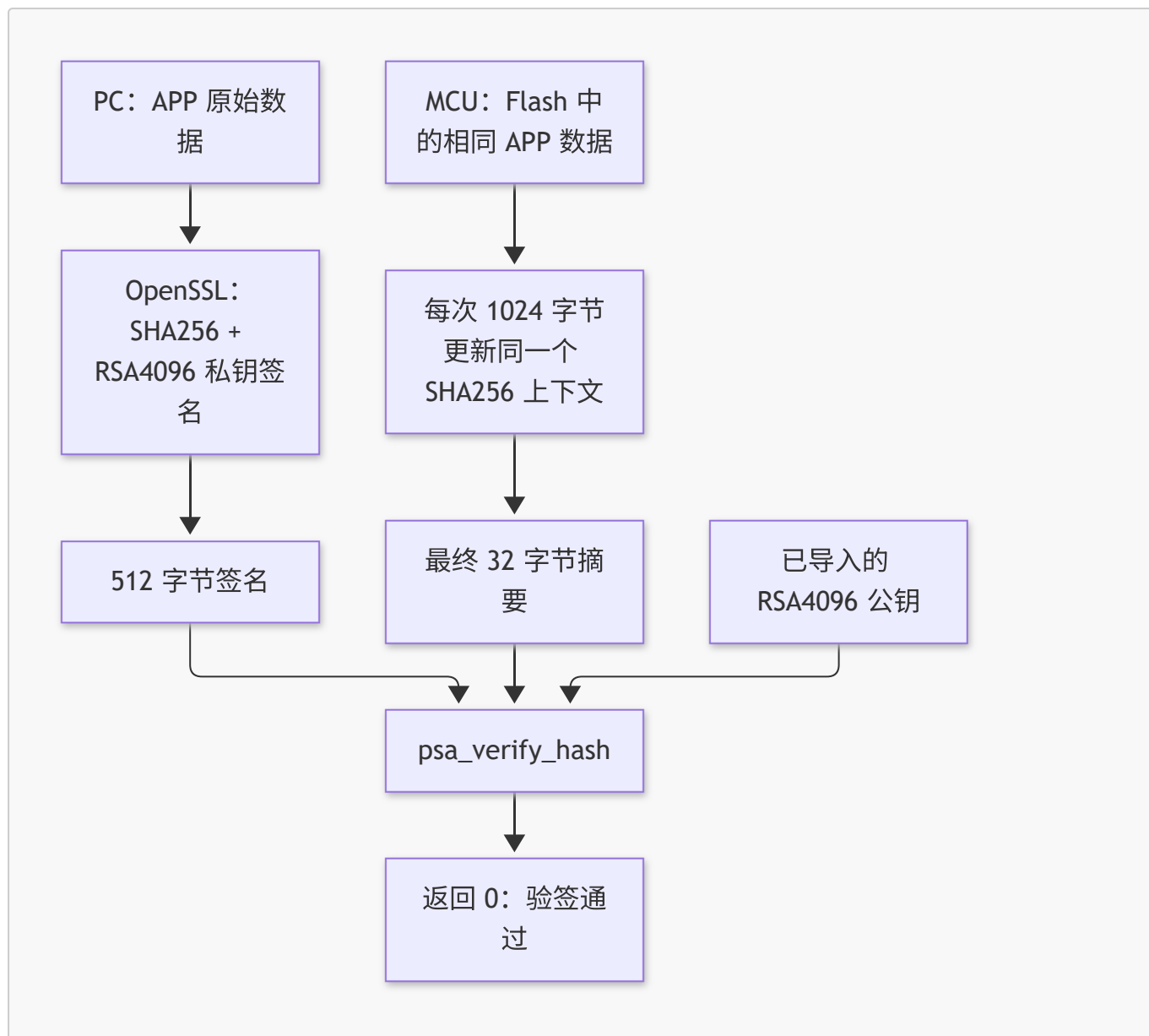
这里的“400K”具体指 **400 KiB = 400 × 1024 = 409600 字节**。Demo 使用模拟 APP 数据，实际项目可以替换为正式 APP 的 Flash 地址。

1. 先理解数据、hash 和签名

名称	本 demo 中的含义	大小
APP 数据	要检查的全部原始字节，存放在 Flash	409600 字节，另测试 409613 字节
SHA256 / hash / 摘要	按顺序处理全部 APP 字节后得到的结果	固定 32 字节
RSA 签名	PC 使用私钥对该数据进行 SHA256 + RSA 签名的结果	RSA4096 下固定 512 字节
RSA 公钥	MCU 用来验证签名的密钥	本 demo 使用 PKCS#1 DER 编码

摘要相当于数据的一个固定长度“指纹”。只比较摘要不能证明发布者身份；验签还需要公钥和签名。正式系统必须使用可信的公钥。

PC 保存私钥并生成签名；MCU 导入对应公钥，计算收到的 APP 的摘要，然后检查签名与摘要是否匹配。这里的 RSA 操作是签名/验签，不是把整个 APP 加密/解密。



2. 为什么直接校验 400 KiB 会内存不足

当前 SDK 的 Mbed TLS 3.6.2 默认在部分 PSA 接口内复制输入数据。因此，下面两种一次性调用都会尝试申请约 400 KiB 的输入副本：

```
psa_verify_message(key, algorithm, image, 409600, signature, 512);
psa_hash_compute(PSA_ALG_SHA_256, image, 409600, hash, 32, &hash_size);
```

内存池只有 20480 字节，申请失败就返回 `-141`，即 `PSA_ERROR_INSUFFICIENT_MEMORY`。即使 `image` 已经在 Flash，接口内部仍会尝试复制它。

`CRYPTO_ENGINE_BUF_SIZE` 控制的是 Mbed TLS 可分配的 **RAM 内存池大小**，不是 APP 最大长度。解决方法是缩小每次交给 PSA 的输入，而不是把内存池扩大到 APP 大小。

3. 分段处理如何覆盖完整 400 KiB

核心代码位于 `app_verify.c`，分成两个函数：

- `app_sha256()`：遍历全部数据，计算一个最终摘要。
- `app_verify()`：先调用 `app_sha256()`，再使用公钥调用 `psa_verify_hash()`。

`app_sha256()` 的正常执行流程是：

```
psa_hash_setup(SHA256)      只初始化一次
  update(image + 0,         1024) 第 1 块
  update(image + 1024,      1024) 第 2 块
  ...
  update(image + 408576, 1024) 第 400 块
psa_hash_finish()           只结束一次，输出整段数据的 32 字节摘要
psa_hash_abort()            清理操作状态；错误路径也会执行
```

实际循环取 `min(剩余字节数, 1024)`。对于 409613 字节的数据，处理完 400 个完整块后，还会处理最后 13 字节，再执行 `finish()`。

所有块共用同一个 SHA256 上下文。 分段只是改变数据送入的方式，结果与对整个文件一次性计算 SHA256 相同。不是分别对 400 个块计算 400 个 hash，也不需要每块一个签名。

本 demo 的实现直接传入 `image + offset` 指向的 Flash 数据，没有创建一个 400 KiB RAM 数组。当前 PSA 每次 `update()` 临时复制最多 1024 字节，调用返回时释放这次副本；SHA256 上下文保存累计计算状态。随后 RSA 验签使用 32 字节摘要和 512 字节签名。

因此，APP 长度增加主要增加处理次数，内存需求不会按 APP 长度等比例增加。20 KiB 池还用于密钥、SHA256 上下文和 RSA 运算，并非全部只用于那 1 KiB 数据；本配置是否足够已经通过此前 ME0 板测验证。

`app_verify()` 最后调用的是：

```
psa_verify_hash(key, APP_VERIFY_ALG,
                hash, sizeof(hash), signature, signature_size);
```

这里不能把 `hash` 交给 `psa_verify_message()`：后者会把这 32 字节再做一次 SHA256，导致与原始 APP 的签名不匹配。

4. `rsa_large_app_vectors.h` 中的数据从哪里来

`rsa_large_app_vectors.h` 由 PC 脚本 `generate_rsa_large_app_vectors.py` 生成。正常编译和上板运行不需要重新执行脚本。

测试数据的生成规则为：

```
payload = bytes(range(256)) * 1600 + bytes(range(13))
```

即 `00 01 ... FE FF` 重复 1600 次，随后追加 `00 01 ... 0C`，共 409613 字节。头文件用宏缩短源代码，但编译后仍是 Flash 中真实、连续的 409613 字节 `static const` 数组，并非在 RAM 中展开。

同一个数组取三个长度，分别生成对应摘要和签名：

数组后缀	数据范围	目的
4k	前 4096 字节	小数据基线，与一次性验签对照
400k	前 409600 字节	验证目标 400 KiB 长度
tail	全部 409613 字节	确认不足 1 KiB 的最后一块也被处理

demo_hash_* 使用 Python `hashlib.sha256(data).digest()` 生成。demo_signature_* 使用 OpenSSL 对同一组原始数据签名：

```
openssl dgst -sha256 -keyform DER -sign sample.der -out signature.bin image.bin
```

脚本从原工程 `RSA_example.c` 的 `RSA_private_key_4096_der` 提取 SDK 示例私钥，临时写入 `sample.der`。OpenSSL 内部先计算 SHA256，再做 RSA4096 / PKCS#1 v1.5 签名。脚本同时导出 `demo_public_key`，并使用 OpenSSL 验签确认每组结果正确；结束后清理临时文件。

MCU 会将自己计算的摘要与 `demo_hash_*` 比较，以检查分段处理是否正确；之后再验签。**参考摘要只是 demo 的检查工具，正式 `app_verify()` 不依赖预先保存的参考 hash。** 公钥、APP 数据和签名就足以调用该接口。

SDK 示例密钥是公开测试材料，正式产品应使用自己的 RSA4096 密钥。

5. 从 main 开始阅读代码

文件	作用
<code>main.c</code>	初始化时钟、引脚、串口、Mbed TLS 内存池及 PSA，然后运行 demo
<code>RSA_large_app_demo.c</code>	导入公钥，组织对比测试，输出每项结果，销毁密钥
<code>app_verify.c</code> / <code>app_verify.h</code>	可复用的分段摘要计算与验签接口
<code>rsa_large_app_vectors.h</code>	Flash 测试数据、公钥、三组参考摘要和签名
<code>mbedtls_config_yuntu.h</code>	Mbed TLS 功能和 20480 字节内存池配置
<code>mbedtls_config.c</code>	定义内存池并初始化分配器

`main.c` 中 `RUN_LARGE_APP_VERIFY_DEMO=1` 时运行本 demo；改成 0 可以运行原工程的 AES/RSA/ECDSA 示例。

Demo 先导入一个允许 `VERIFY_HASH` 和 `VERIFY_MESSAGE` 的 RSA4096 公钥，以便同一把钥匙同时参与一次性与分段验签。处理结束后调用 `psa_destroy_key()`。正式代码只调用 `app_verify()` 时，设置 `PSA_KEY_USAGE_VERIFY_HASH` 即可。

当前共 20 项检查：

检查内容	项数	预期结果
公钥导入	1	成功
4 KiB 一次性验签	1	成功
400 KiB 一次性验签、一次性 hash	2	返回 -141 ，复现内存不足
三个长度的分段 hash 与 PC 参考值比较	3	全部匹配
三个长度的分段验签	3	全部成功
公钥销毁	1	成功
8 轮重新导入 / 分段验签 / 销毁	8	全部成功
内存分配器元数据检查	1	通过

测试之间不会重新初始化内存池。当前已移除数据/签名篡改、截短及无效参数的测试。两个一次性调用仍使用完整、正确的数据和签名，其 -141 来自实际内存分配失败。

6. 上板运行和预期输出

- 编译项目，烧录 `build/MbedTls_with_PSA_Demo.hex`，或用调试器加载同名 ELF。
- 串口配置为 **115200、8N1、无流控**，当前 MCU 输出引脚为 PTA9 / LINFlexD2 TX，接收端与 MCU 共地。原 EVB 使用 J32。
- 打开串口后复位 MCU；调试器若停在 `main`，继续运行。

关键预期输出：

```
pool=20480, chunk=1024, flash payload=409613 bytes
[PASS] 400KiB verify_message reproduces OOM: status=-141 expected=-141
[PASS] 400KiB hash_compute reproduces OOM: status=-141 expected=-141
[PASS] 400KiB streamed SHA256 matches PC: status=0 expected=0
[PASS] 400KiB app_verify: status=0 expected=0
[PASS] 400KiB+13 app_verify: status=0 expected=0
ME0 LARGE APP DEMO: PASS (20 checks, 0 failures)
```

PASS 表示实际结果符合该项预期。因此上面两个 -141 是对比测试通过，不是分段方案失败。分段验签必须返回 `0`。

当前移除错误注入后的版本已由用户在 ME0 板上重新运行，提供的完整串口日志显示：**ME0 LARGE APP DEMO: PASS (20 checks, 0 failures)**。在 20480 字节内存池、1024 字节分块配置下，400 KiB 和 400 KiB + 13 字节分段验签、三个参考摘要比对及 8 轮重复运行全部通过；两个一次性调用均返回预期的 -141。结果来源为用户提供的板端串口日志。此前旧版 26 项板测也全部通过，记录与详细构建步骤见 [测试说明](#)。

7. 如何替换为正式 APP

PC 端先用自己的 RSA4096 私钥对实际二进制文件签名，并导出公钥：

```
openssl dgst -sha256 -sign private.pem -out app.sig app.bin
openssl rsa -in private.pem -RSAPublicKey_out -outform DER -out public.der
```

MCU 端保留 `app_verify.c/.h` 的处理流程，导入 `public.der`，将 `app.bin` 对应的真实 Flash 地址、文件长度和 512 字节 `app.sig` 传入：

```
/* 前提：内存池及 PSA 已初始化，key_id 为已导入的公钥。 */
psa_status_t status = app_verify(key_id,
                                app_start, app_actual_size,
                                signature_data, 512u);

if (status == PSA_SUCCESS) {
    /* 该范围的数据通过验签。 */
}

/* 不再使用这把密钥时调用 psa_destroy_key(key_id)。 */
```

实际地址范围需要由调用方检查，数据在计算期间应保持不变。PC 与 MCU 必须覆盖完全相同的字节，包括是否包含头部和尾部填充。签的是原始 `.bin` 内容，不是 `.hex` 文件的文本，也不是 hash 的十六进制字符串。

接入实际 APP 后不需要保留 demo 的 400 KiB 测试数组，也不需要保留 `demo_hash_*`。本 demo 验证的是公钥验签所需内存；板端私钥签名、密钥生成的内存需求应另行评估。