SDK应用_HCU 原理与应用（二）

版本：

Config Tool Version：2.7.6

YTM32B1MD2 SDK version：1.3.2

前言

由于 **YTM32B1MD24** 新增了 RSA 与 ECC，本文主要针对 AES，SHA，RSA，ECC 进行原理性质的说明，以及 Demo 示例解释。

另外最后附上了全系列芯片的算法功能对比，以及算法效率对比

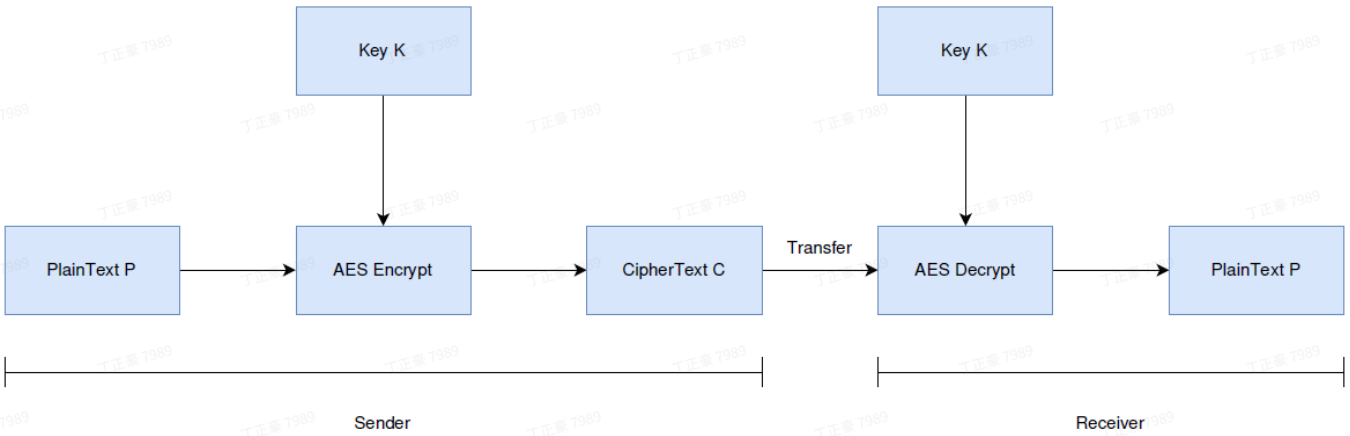
AES

AES：高级加密标准（Advanced Encryption Standard）

对称密钥加密（加密和解密用相同的密钥）

AES 的区块长度固定为 **128** 位，密钥长度则可以是128，192 或256 位，对应的加密轮数为10、12、14 轮，相比较而言，AES-256 的安全性最高，AES-128 的性能最高。

加密流程大致如下图所示。

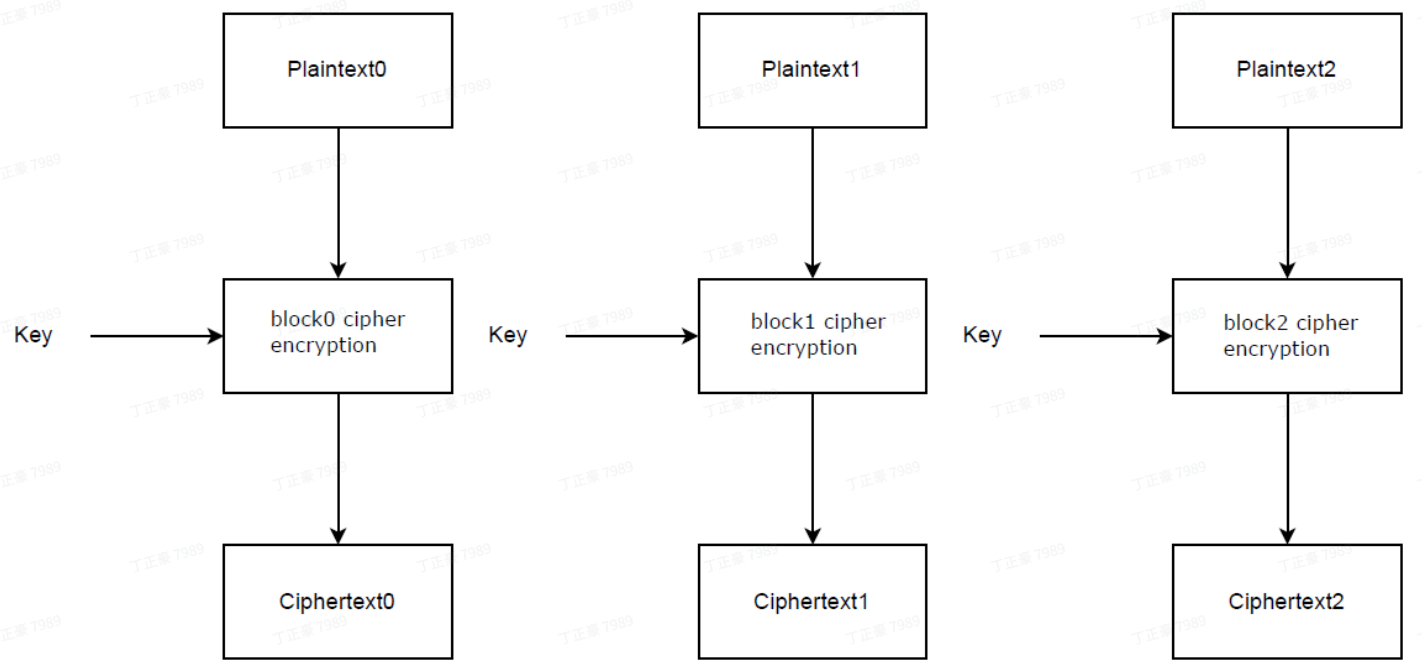


ECB

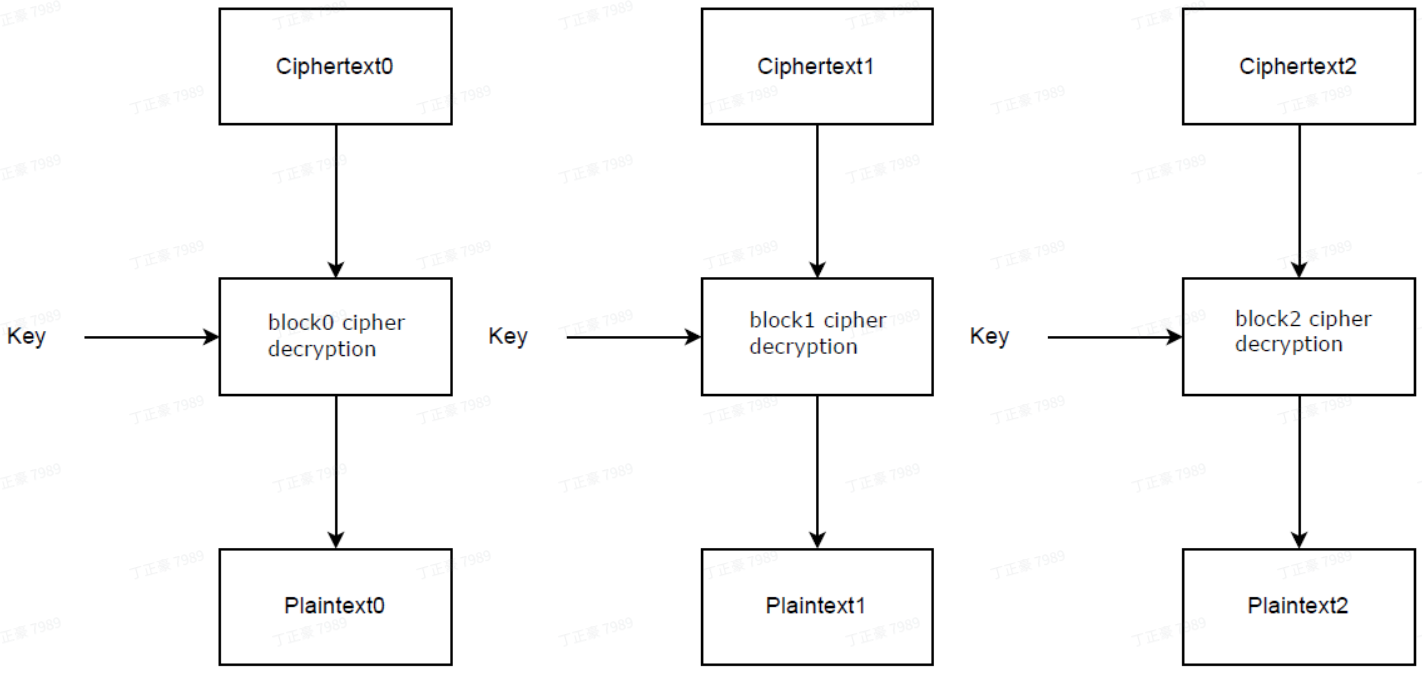
电码本模式Electronic Codebook

最简单的工作模式，在该模式下，每一个明文块的加密都是完全独立，互不干涉的。只能加密块数据，加密解密可以并行执行。

ECB 模式一般只适用于小数据量的字符信息的安全性保护，例如密钥保护。



Electronic Codebook (ECB) mode encryption



Electronic Codebook (ECB) mode decryption

优点：

- 简单
- 有利于并行计算

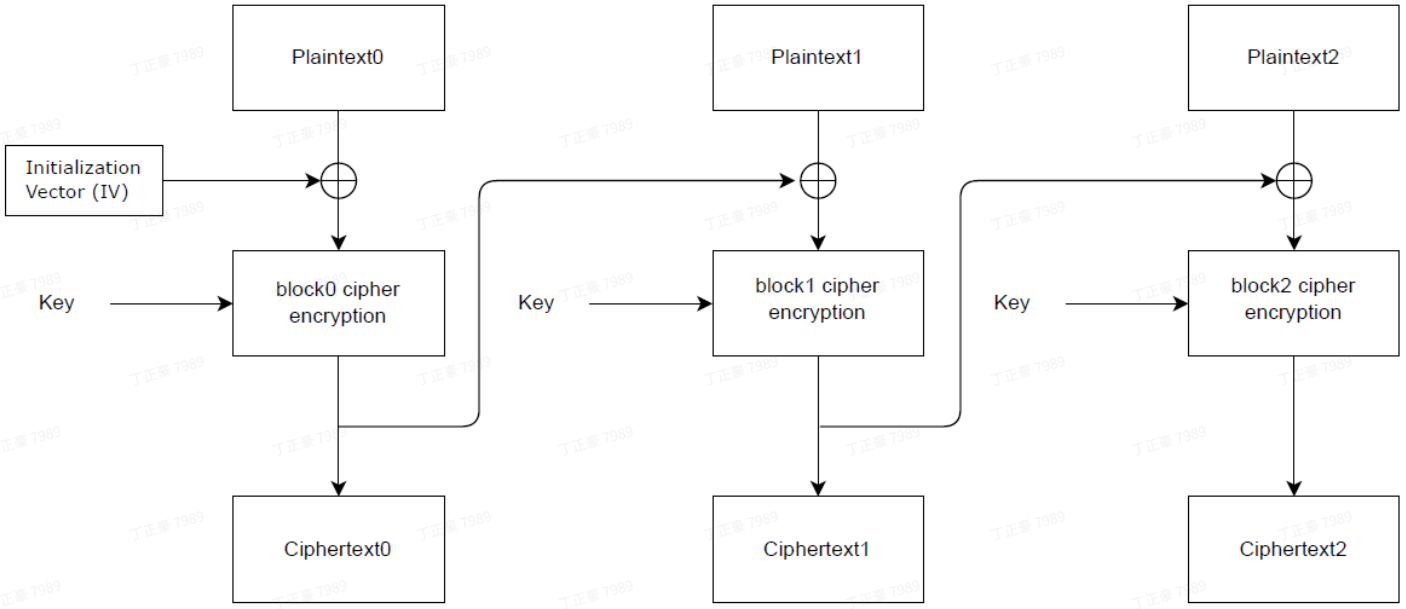
缺点：

- 相同的明文块经过加密会变成相同的密文块，因此安全性较差，难以抵抗统计分析攻击。

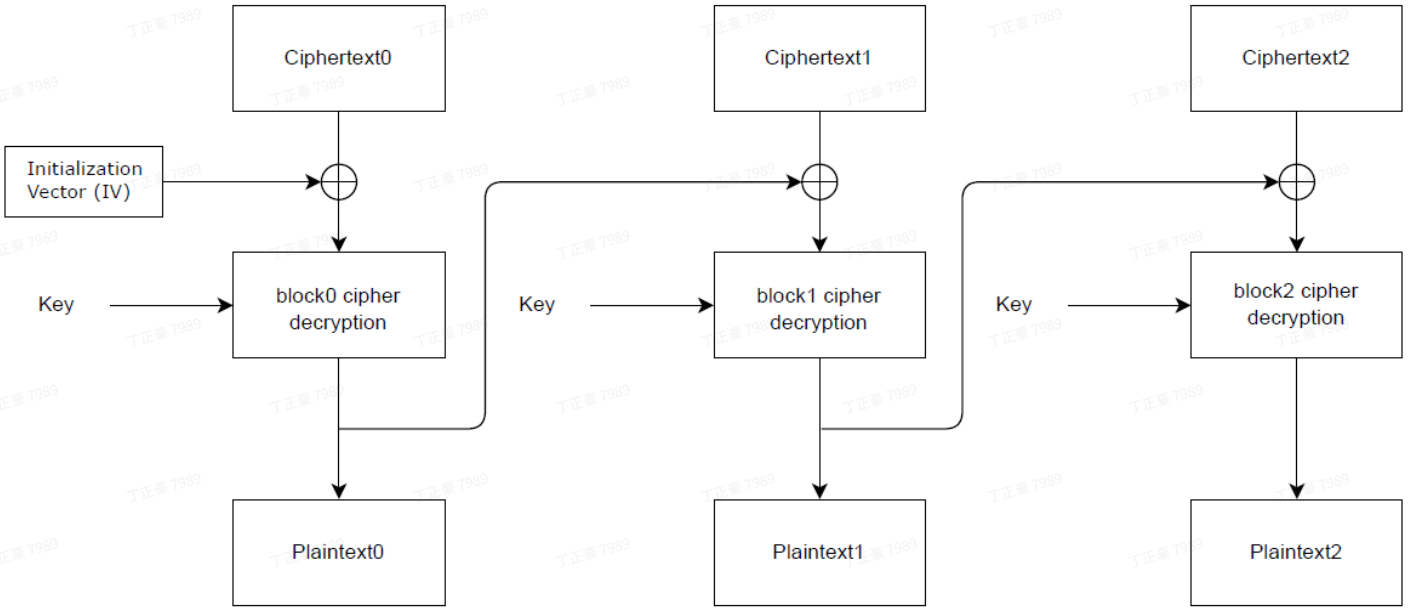
CBC

密码分组链接模式 Cipher Block Chaining

为了克服 ECB 的缺点，引入初始向量IV，可以防止同样的明文块始终加密成同样的密文块，IV 作为初始化变量，参与第一个明文块的异或，后续的每一个明文块和它前一个明文块所加密出的密文块相异或。只能加密块数据，加密是串行，解密可以并行。



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

优点：

- 安全性更高

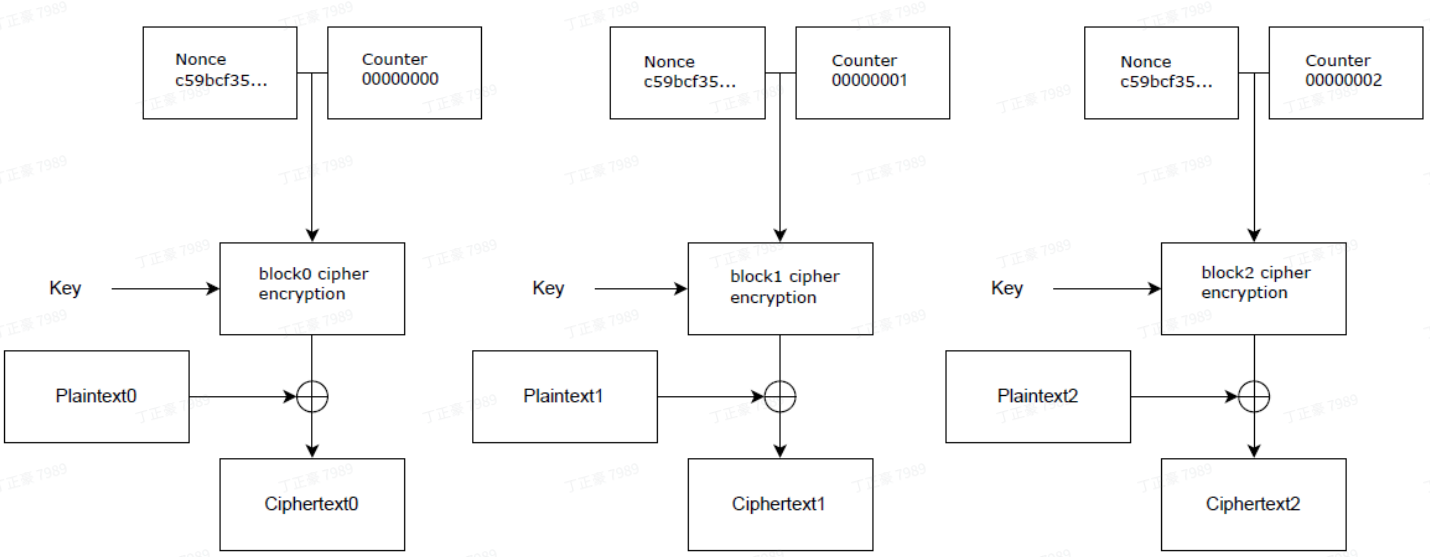
缺点：

- 无法并行计算，性能上不如ECB。

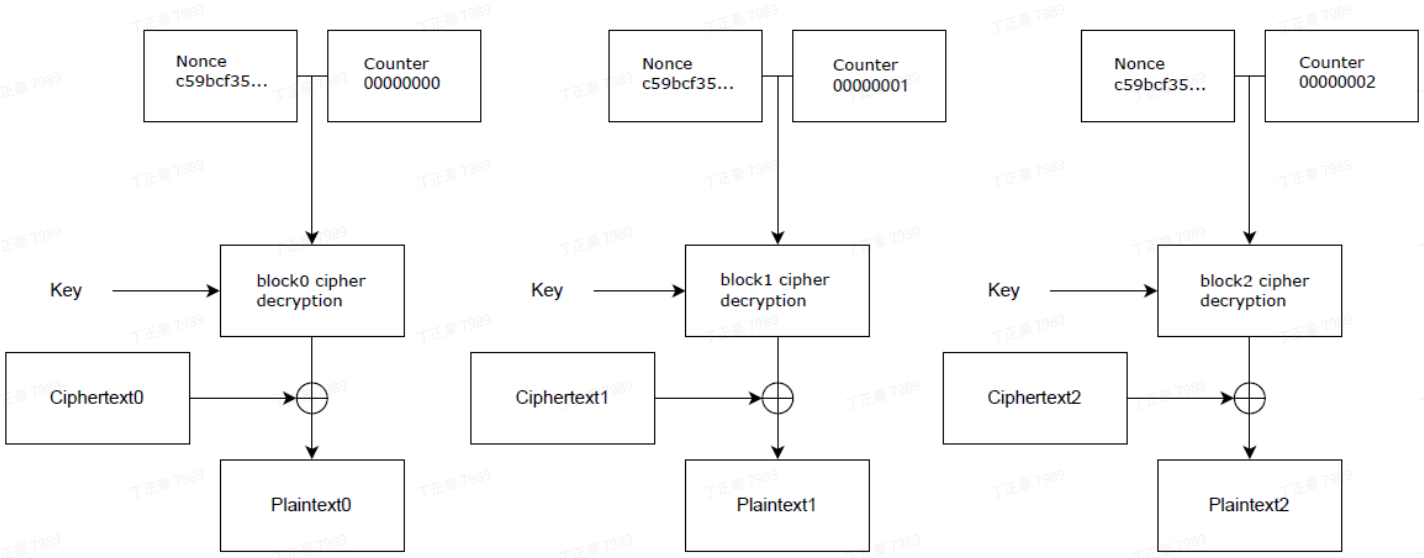
CTR

计算器模式 Counter

内部有一个自增的算子，这个算子用密钥加密之后的输出和明文异或的结果得到密文，相当于一次一密。这种加密方式简单快速，安全可靠，而且可以并行加密。CTR 涉及参量：Nonce 随机数、Counter 计数器和密钥。Nonce 随机数和Counter 计数器整体可看作计数器。



Counter (CTR) mode encryption

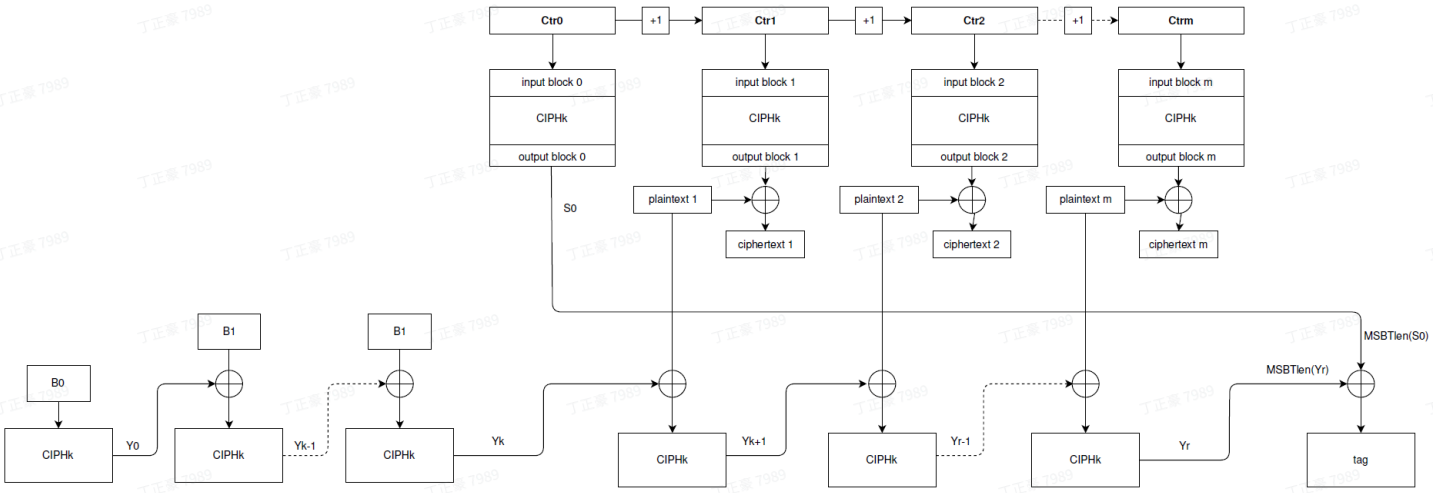


Counter (CTR) mode decryption

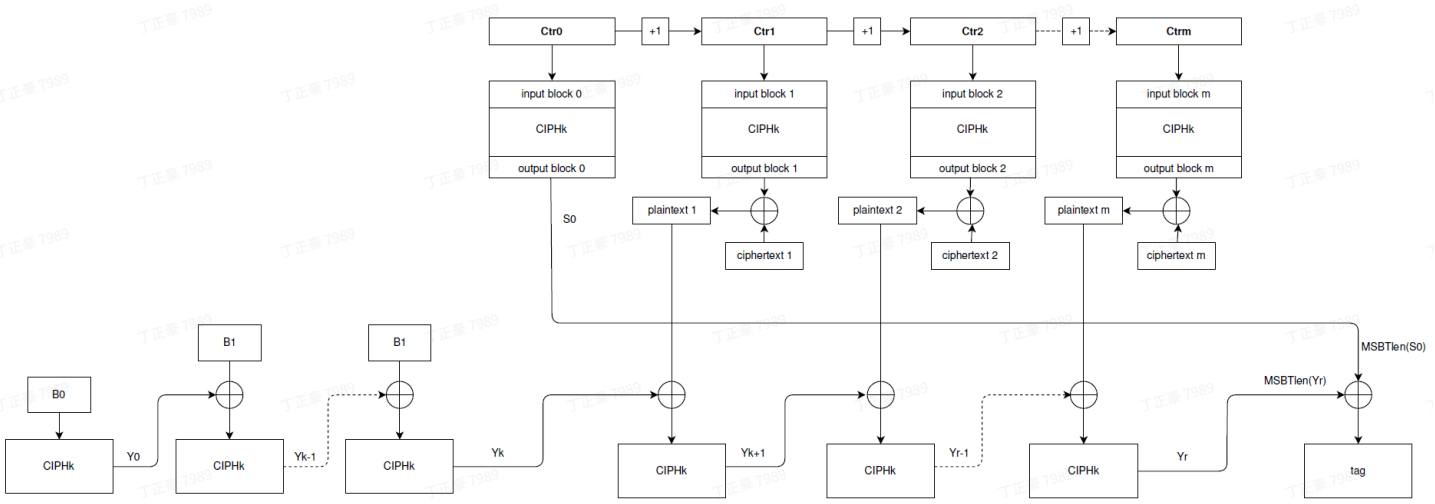
CCM

CCM 模式（Counter with CBC-MAC）

CCM 首先使用CBC-MAC 模式来认证传输帧，然后使用CTR 模式来加密帧，CBC-MAC 实际上就是对消息使用CBC 模式进行加密，再取密文的最后一块作为认证码，MAC 为消息认证码，MAC 值是使用明文和密钥计算得出的，在攻击者不知道密钥但修改了消息内容时，使用MAC 可以让接收方知道消息是否被篡改。



Counter with CBC-MAC (CCM) mode encryption



Counter with CBC-MAC (CCM) mode decryption

构造B0、B1……、Bn，每组16 个字节，B0、B1 有特定结构，具体内容请参考相关协议，以B0 加密结果作为IV 分别对B1 B2……. Bn 做CBC 计算，结果与分段明文做异或，下一个结果与计数器初值Ctr0 的CTR 加密结果做异或得到tag，明文P1、P2……、Pn 做CTR 加密得到密文C1、C2……、Cn。

CMAC

CMAC 模式（Cipher-based MAC）

只用于生成消息认证码，对于待加密消息M，应用CBC-MAC 算法。在CMAC 操作中有两种情况：如果输入消息长度等于Block 的整数倍，最后的Block M_n 需要先与K1 异或再进行处理；如果输入的消息长度不等于Block 的整数倍，最后的Block M_n 需要补齐到一个Block 的大小，与K2 异或，再进行处理。最后得到认证码T。

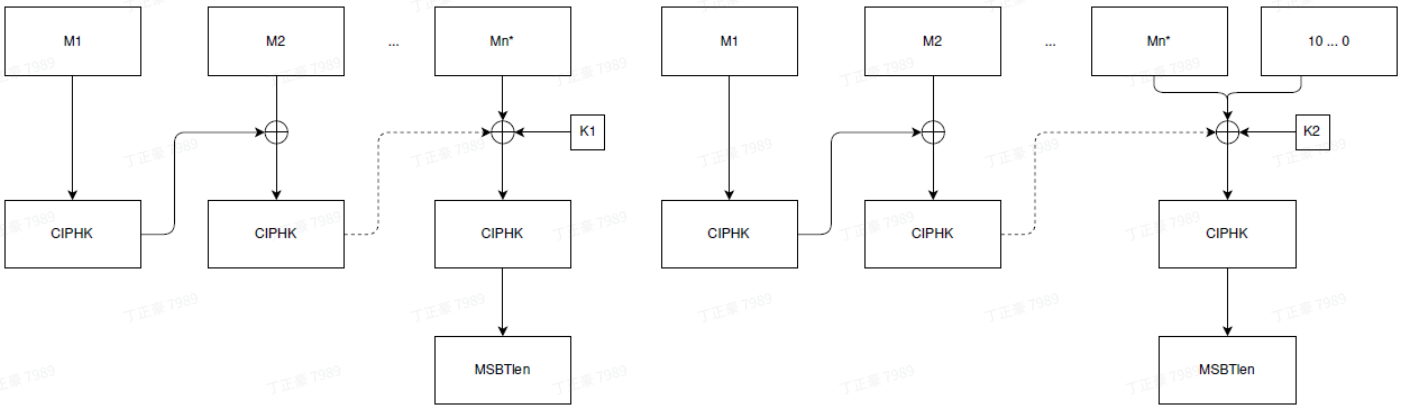


Illustration of the two cases of MAC Generation

输入密钥K，输出子密钥K1 与K2。过程如下：

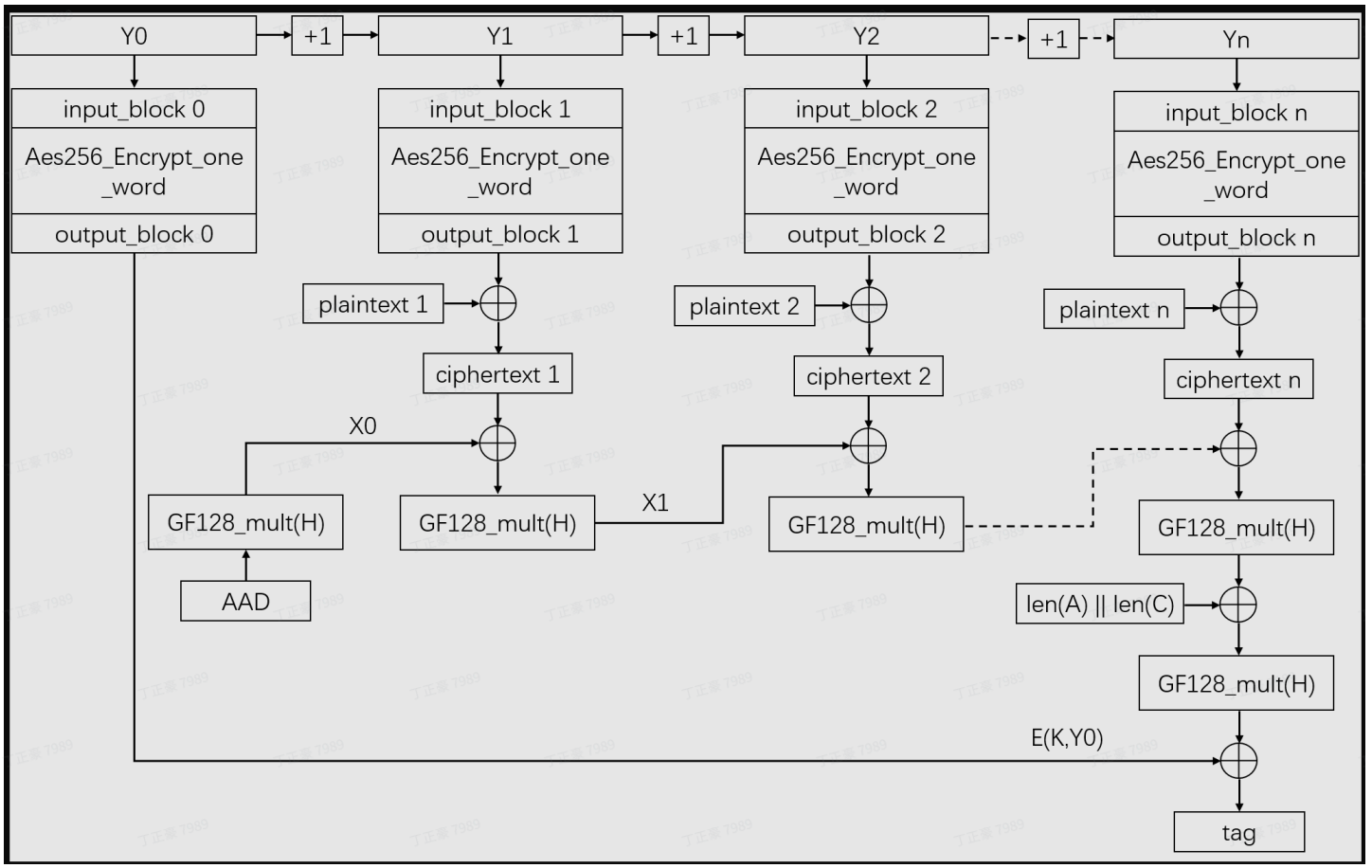
1. 用密钥K 对全零输入块加密得到L
2. K1 通过如下方式导出：如果L 的最高有效位为0，则K1：=L 左移1 位；否则，K1 等于const_Rb 与左移1 位的L 异或。
3. K2 通过如下方式导出：如果K1 的最高有效位为0，则K2：=K1 左移1 位；否则，K2 等于const_Rb 与左移1 位的K1 异或。

const_Rb 为固定值，只与块大小有关，128bit 时const_Rb = 012010000111，64bit 时const_Rb = 05911011

GCM

GCM模式（GMAC CTR Mode）

先进行CTR加密，得到的密文会与GMAC乘法处理后的附加消息及计数器初值做异或，得到MAC值。最后，密文接收者会收到密文、IV（计数器CTR的初始值）、MAC值。

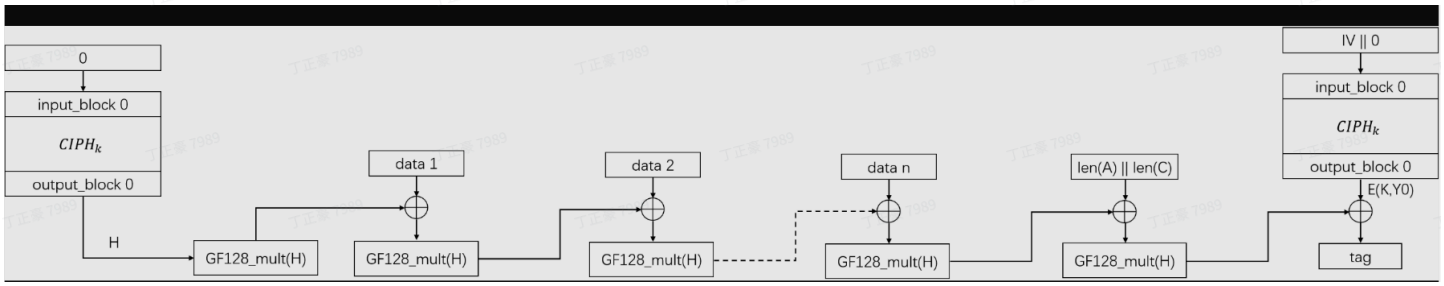


GMAC

GMAC (Galois Message Authentication Code)

GMAC 是基于 AES 和 GHASH 的高效、仅用于认证的算法。它利用 GCM 模式的认证部分，为消息和可选的关联数据生成一个认证标签（MAC），以验证数据的完整性和来源真实性。

注意，GCM 得到的 MAC 有明文数据的参与，所以与 GMAC 计算同一段内容时，最终得到的 MAC 不相同。



SHA

安全散列算法（英语：Secure Hash Algorithm，缩写为SHA）是一个密码散列函数家族，是FIPS所认证的安全散列算法。能计算出一个数字消息所对应到的，长度固定的字符串（又称消息摘要）的算法。且若输入的消息不同，它们对应到不同字符串的机率很高。

SHA家族的五个算法，分别是`SHA-1`、`SHA-224`、`SHA-256`、`SHA-384`，和`SHA-512`，由美国国家安全局（NSA）所设计，并由美国国家标准与技术研究院（NIST）发布；是美国的政府标

准。后四者有时并称为 `SHA-2`。`SHA-1` 在许多安全协定中广为使用，包括 `TLS` 和 `SSL`、`PGP`、`SSH`、`S/MIME` 和 `IPsec`，曾被视为是 `MD5`（更早之前被广为使用的杂凑函数）的后继者。但 `SHA-1` 的安全性如今被密码学家严重质疑；虽然至今尚未出现对 `SHA-2` 有效的攻击，它的算法跟 `SHA-1` 基本上仍然相似；因此有些人开始发展其他替代的杂凑算法。

HMAC

HMAC Hash-based Message Authentication Code, 哈希消息认证码.

HMAC是密钥相关的哈希运算消息认证码，HMAC运算利用哈希算法，以一个密钥和一个消息为输入，生成一个消息摘要作为输出。

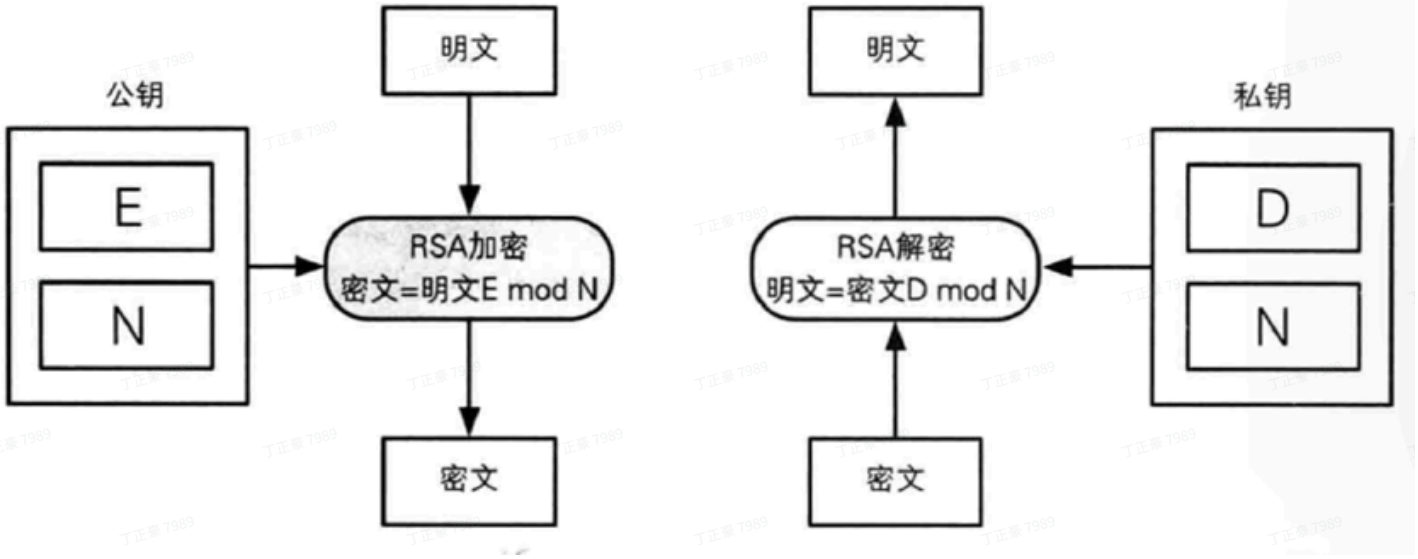
HMAC中的H代指Hash散列算法，HMAC可以使用多种单项散列式，例如使用SHA-1，则构成HMAC-1，选用SHA-256散列算法，则构成HMAC-256。

RSA

RSA 简介

RSA (Rivest Shamir Adleman)

RSA 是非对称算法，有公钥与私钥，加密与解密流程如下图所示。常见的有 RSA-1024/2048/3072/4096



RSA 密钥生成

1. 准备两个非常大的素数 p 和 q
2. 利用字符串模拟计算大素数 p 和 q 的乘积 $n = pq$
3. 同样方法计算 $m = (p-1)(q-1)$，这里 m 为 n 的欧拉函数
4. 找到一个数 $e(1 < e < m)$，满足 $gcd(m, e) = 1$（即 e 和 m 互素）

5. 计算 e 在模 m 域上的逆元 d （即满足 $edmodm = 1$ ）
6. 至此，公钥和私钥生成完毕：(n, e) 为公钥，(n, d) 为私钥

RSA 加密

对于明文 x ，用公钥 (n, e) 对 x 加密的过程，就是将 x 转换成数字（字符串的话取其 ASCII码或者 unicode 值），然后通过幂取模计算出 ，其中 y 就是密文；

$$y = x^e modn$$

RSA 解密

对于密文 y ，用私钥 (n, d) 对 y 进行解密的过程和加密类似，同样是计算幂取模；

$$x = y^d modn$$

RSA Padding

RSA 原始加密算法具有确定性和乘法同态性，直接对明文加密存在安全隐患：相同明文使用相同公钥加密始终得到相同密文，攻击者可通过预计算进行选择明文攻击；同时 RSA 的乘法性质（ $E(m_1) \times E(m_2) = E(m_1 \times m_2 \mod n)$ ）使密文具有可锻造性。因此需要在加密或签名前对消息进行填充（Padding），使输出概率化并破坏同态结构。

PKCS#1 标准定义了两类填充方案：

- 加密填充（RSAES）：RSAES-OAEP、RSAES-PKCS1-v1_5
- 签名填充（RSASSA）：RSASSA-PSS、RSASSA-PKCS1-v1_5

RSAES-OAEP

全称：Optimal Asymmetric Encryption Padding（最优非对称加密填充），PKCS#1 v2.0 引入，RFC 8017 定义。

核心思想：使用哈希函数与 MGF1（Mask Generation Function，掩码生成函数）对明文和随机种子进行两轮混淆，使加密结果概率化，在随机预言机模型下可证明达到 IND-CCA2 安全。

编码结构：$EM = 0x00 \parallel maskedSeed \parallel maskedDB$ ，其中 $DB = lHash \parallel PS \parallel 0x01 \parallel M$ 。

编码步骤：

1. 计算 $lHash = Hash(L)$ ，L 为可选标签，通常为空串
2. 生成随机种子 $seed$ ，长度为 $hLen$ （哈希输出字节数）
3. 计算 $dbMask = MGF1(seed, k - hLen - 1)$
4. 计算 $maskedDB = DB \oplus dbMask$
5. 计算 $seedMask = MGF1(maskedDB, hLen)$

6. 计算 $maskedSeed = seed \oplus seedMask$
7. 拼接得到 $EM = 0x00 \parallel maskedSeed \parallel maskedDB$

最大明文长度：$k - 2 \times hLen - 2$ 字节（k 为 RSA 模数字节数）。

安全性：抗选择密文攻击，是新应用的推荐加密填充方案。常用哈希函数为 SHA-1、SHA-256。

RSAES-PKCS1-v1_5

定位：PKCS#1 v1.5 定义的老式加密填充，结构简单但存在已知攻击，目前主要用于遗留系统兼容。

编码结构：$EM = 0x00 \parallel 0x02 \parallel PS \parallel 0x00 \parallel M$

- $0x02$ 为加密类型标记
- PS 为非零随机填充字节，长度至少 8 字节
- $0x00$ 为填充与明文的分隔符
- M 为明文消息

最大明文长度：$k - 11$ 字节。

安全性问题：存在 Bleichenbacher 攻击（1998）及后续 ROBOT 攻击，攻击者通过观察解密失败的不同错误提示，可逐字节恢复明文。实际部署中必须做严格的错误处理（不区分解密失败的具体原因），否则存在安全风险。

现状：因兼容性仍广泛存在，新项目应优先使用 OAEP。

RSASSA-PSS

全称：Probabilistic Signature Scheme（概率签名方案），PKCS#1 v2.1 引入，RFC 8017 定义。

核心思想：签名时引入随机盐（salt），使相同消息每次签名产生不同结果，在随机预言机模型下可证明安全。

编码结构：$EM = 0x00 \parallel maskedDB \parallel H \parallel 0xBC$，其中 $H = Hash(0x00^8 \parallel mHash \parallel salt)$，$DB = PS \parallel 0x01 \parallel salt$。

签名编码步骤：

1. 计算 $mHash = Hash(M)$
2. 生成随机盐 $salt$，长度为 $sLen$（通常等于 $hLen$）
3. 拼接 $M' = 0x00^8 \parallel mHash \parallel salt$
4. 计算 $H = Hash(M')$
5. 构造 $DB = PS \parallel 0x01 \parallel salt$（PS 为全零填充）
6. 计算 $dbMask = MGF1(H, k - hLen - 1)$
7. 计算 $maskedDB = DB \oplus dbMask$
8. 拼接得到 $EM = 0x00 \parallel maskedDB \parallel H \parallel 0xBC$

验签：逆过程从 EM 中恢复 H 和 salt，重新计算哈希值并比对。

推荐：新系统签名首选 PSS，可有效抵御各类签名伪造攻击。

RSASSA-PKCS1-v1_5

定位：PKCS#1 v1.5 签名填充，确定性签名，是目前部署最广泛的 RSA 签名方案。

编码结构：$EM = 0x00 \parallel 0x01 \parallel PS \parallel 0x00 \parallel T$

- $0x01$ 为签名类型标记
- PS 为全 $0xFF$ 填充字节
- $0x00$ 为填充与 DigestInfo 的分隔符
- $T = DigestInfo$，包含哈希算法标识符（OID）和哈希值

DigestInfo 结构（ASN.1）：

```
代码块
1  SEQUENCE {
2      digestAlgorithm AlgorithmIdentifier,
3      digest OCTET STRING
4  }
```

特点：确定性——相同消息配合相同密钥始终产生相同签名，便于验证和缓存。

安全性：存在 Bleichenbacher 签名伪造攻击（针对实现缺陷，如未严格校验 DigestInfo 前缀和填充结构）。正确实现下目前仍被认为安全，但长期推荐迁移到 PSS。

四种填充方案对比：

方案	类型	随机性	可证明安全	最大输入	推荐场景
RSAES-OAEP	加密	是	IND-CCA2	k - 2hLen - 2	新加密
RSAES-PKCS1-v1_5	加密	是	否	k - 11	遗留兼容
RSASSA-PSS	签名	是	是（ROM）	不限（哈希）	新签名
RSASSA-PKCS1-v1_5	签名	否	否	不限（哈希）	遗留兼容

ECC

ECC (Elliptic Curves Cryptography)

ECC算法以指定的椭圆曲线为基础，将消息映射成曲线上的一个点，通过“点加”和“点乘”操作来进行椭圆上点的变换，最后得到的点作为加密结果。

本章主要介绍 ECC 算法原理，以及 ECDSA 的加密流程

ECC 简介

并不是所有的椭圆曲线都适合加密，$y^2 = x^3 + ax + b$ 是一类可以用来加密的椭圆曲线，也是最为简单的一类。

针对曲线Ep(a,b)表示为$y^2 = x^3 + ax + b(modp)$，$x, y \in [0, p]$，p为质数
该曲线关于x轴对称。选择两个满足下列条件的小于p(p为素数)的非负整数a、b，要求满足以下条件 $3a^3 + 27b^2 \neq 0$

ECC 曲线的参数需要包含

- p：有限域
- a：多项式一次系数
- b：多项式常数系数
- G：基点
- n：椭圆曲线的阶数

例如比特币 Bitcoin 使用的 **secp256K1** 这条曲线的参数如下：

2.4.1 Recommended Parameters secp256k1

The elliptic curve domain parameters over $\mathbb{F}_p$ associated with a Koblitz curve `secp256k1` are specified by the sextuple $T = (p, a, b, G, n, h)$ where the finite field $\mathbb{F}_p$ is defined by:

$$p = \texttt{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F}$$
$$= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$$

The curve E: $y^2 = x^3 + ax + b$ over $\mathbb{F}_p$ is defined by:

$$a = \texttt{00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000}$$
$$b = \texttt{00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007}$$

The base point G in compressed form is:

$$G = \texttt{02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798}$$

and in uncompressed form is:

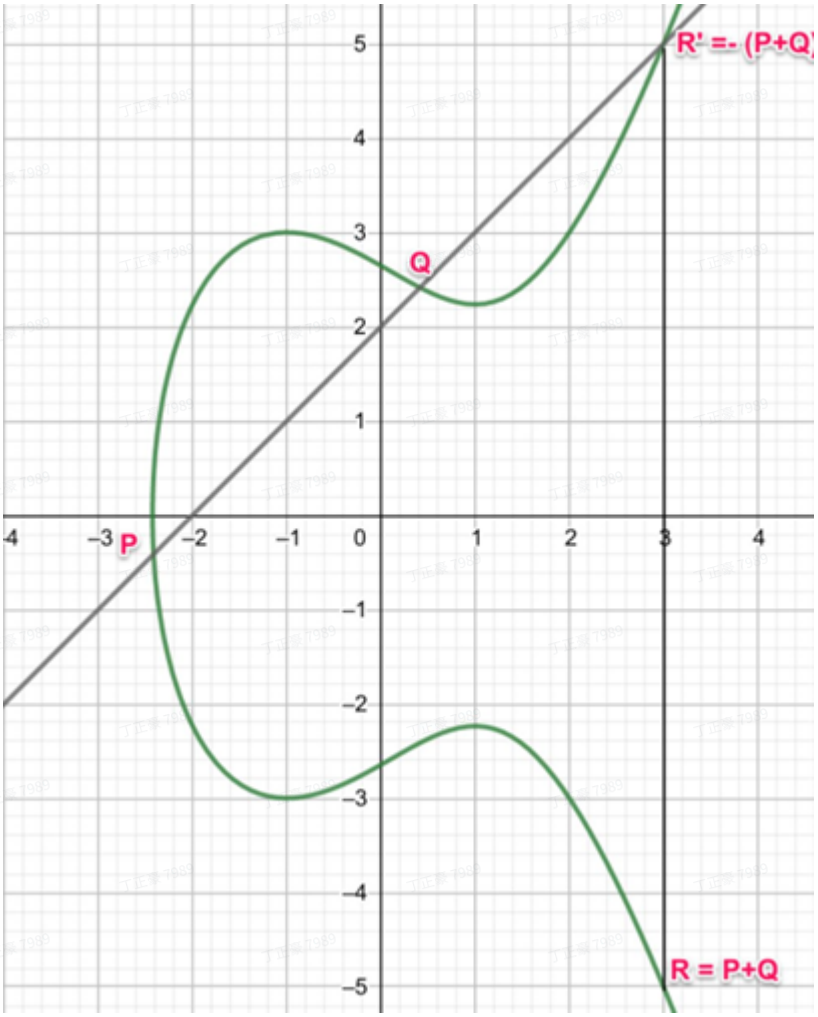
$$G = \texttt{04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8}$$

Finally the order n of G and the cofactor are:

$$n = \texttt{FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141}$$
$$h = \texttt{01}$$

ECC 点加

椭圆曲线上的两个点相加，并非是横坐标与纵坐标的相加，例如下图中，椭圆曲线上的两点，P，Q 相加，取 过P，Q两点的直线，与椭圆曲线相交的点 R'，取 R' 作 X 轴对称的点 R 即得两点相加的结果。 R = P + Q

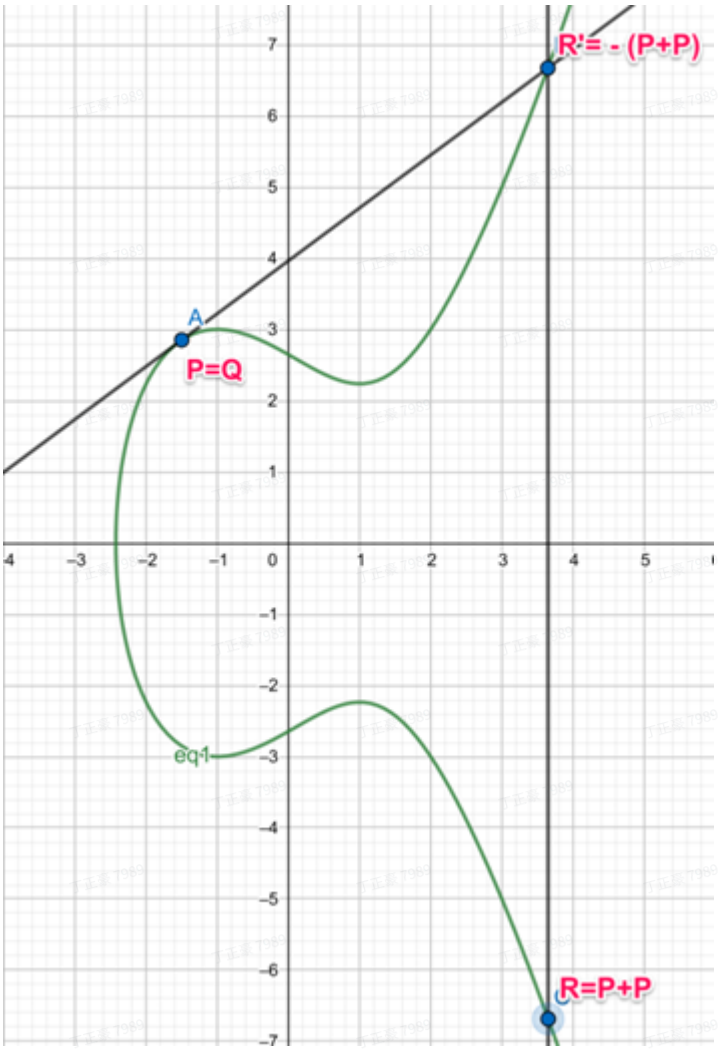


ECC 点乘

椭圆曲线的点乘运算会稍微复杂一点。例如下图，期望得到 2 * P 的结果，需要在 P 点作关于椭圆曲线的切线，与椭圆曲线相交的点 R'，取 R' 作 X 轴对称的点 R 即得点乘的结果。 R = 2 * P；如果需要计算多倍的值，例如希望计算 11 * P，则要将

$11 * P = 8 * P + 2 * P + P = 2 * (4 * P) + 2 * P + P = 2 * (2 * (2 * P)) + 2 * P + P$。

2P，4P，8P 的结果先计算出来，随后依次进行点加，即可得到 11 * P



ECDSA

ECDSA（Elliptic Curve Digital Signature Algorithm）

ECDSA 是一种基于椭圆曲线密码学（Elliptic Curve Cryptography, ECC）的数字签名算法，伴以摘要算法，即可完成对消息的加密。

下面对 ECDSA 算法流程进行简述

生成公钥与私钥

私钥

一个随机生成的大数据 d，要求 $1 <= d <= n - 1$，其中 n 是椭圆曲线的阶。

公钥

使用私钥 d 生成公钥 $Q = d * G$，其中 G 是曲线的基点

签名

1. 计算消息的 Hash 值，使用哈希函数（例如 SHA-256）对消息 M 进行计算，得到消息的摘要 $h = Hash(M)$

2. 从[1,n-1] 范围内选择任意一个随机数 k，该值同样是私密的，不能泄露，且尽量保证每一次签名生成，都使用不同的 k 来保证私钥不会被泄露。
3. 计算签名值，$R = k * G$，其中 G 是曲线的基点，取 r 为 R 点的横坐标值，即 $r = R_x mod n$
4. 计算 $s = k^{-1} * (h + d * r) mod n$，其中 k^{-1} 是 k 的乘法逆元
5. 签名对（r，s）即消息最终的数字签名
6. 签名对 **（r，s）**，原始消息 **M**，与公钥 **Q** 会一同发送给接收方供验签

验签

1. 计算消息的 Hash 值，使用哈希函数（例如 SHA-256）对消息 M 进行计算，得到消息的摘要 $h = Hash(M)$
2. 验证签名在有限域内，即检查签名对（r，s），两个值均在 [1,n] 范围内
3. 计算如下 $v_1 = s^{-1} mod n$，$v_2 = v_1 * h mod n$，$v_3 = v_1 * r mod n$
4. 计算曲线点 $P = v_2 * G + v_3 * Q$，其中 G 是曲线的基点，Q 是公钥
5. 最后验证 r 是否为 P 点的横坐标，即 $r mod n = P_x mod n$
6. 验证成功，则表示消息的可靠性，否则就证明，签名对，消息或公钥中至少有一项被篡改

国密算法

国密即国家密码局认定的国产密码算法。主要有SM1，SM2，SM3，SM4。目前云途硬件仅支持 **SM4-ECB** 算法，其余国密算法均不支持。开此章节仅作知识补充。

- SM1 为对称加密。其加密强度与AES相当。该算法不公开，调用该算法时，需要通过加密芯片的接口进行调用。
- SM2为非对称加密，基于ECC。该算法已公开。由于该算法基于ECC，故其签名速度与秘钥生成速度都快于RSA。ECC 256位（SM2采用的就是ECC 256位的一种）安全强度比RSA 2048位高，但运算速度快于RSA。
- SM3 消息摘要。可以用MD5作为对比理解。该算法已公开。校验结果为256位。
- SM4 无线局域网标准的分组数据算法。对称加密，密钥长度和分组长度均为128位。

软件应用

每款芯片支持的加密算法不完全相同，且 HCU (Hardware Cryptography Unit) 也有迭代，目前有两个版本的 HCU 驱动，需要注意区分。

HCU 初始化

代码块

```
status_t HCU_DRV_Init(const hcu_user_config_t * userConfig, hcu_state_t
*state);
```

初始化函数主要实现了如下功能：

1. 对状态机进行初始化
2. 配置明文与密文的交换方式，默认不交换
3. 配置明文与密文的数据搬运方式，轮询，中断 or DMA

AES 密钥加载

HCU v1

软件密钥

代码块

```
status_t HCU_DRV_LoadUserKey(const void *key, hcu_key_size_t keySize);
```

直接传入密钥，通过**软件**方式直接将密钥加载至 HCU 引擎供运算。

硬件密钥

代码块

```
status_t FLASH_DRV_LoadAESKey(uint32_t instance, uint32_t address);
......
static inline void HCU_SetKeySize(hcu_key_size_t size);
```

传入 AES 密钥在 HCU_NVR 中的存储地址，随后通过**硬件**将密钥加载至 HCU 引擎供运算。如果 HCU 支持不同长度的密钥长度，需要额外设置密钥长度。(AES-128/192/256)

HCU v2

密钥加载直接封在各个加密函数内部，不需要额外调用 API 去实现密钥加载。

AES-ECB

HCU v1

代码块

```
status_t HCU_DRV_EncryptECB(const void *plainText, uint16_t length, void
*cipherText);
......
status_t HCU_DRV_DecryptECB(const void *cipherText, uint16_t length, void
*plainText);
```

驱动提供了 AES-ECB 加密与解密函数，若加密，则需要输入明文，与明文长度，即可得到对应的密文。由于一次最大输入的明文长度有限，如果明文长度超出限制，可分多个 block 进行加密。

HCU v2

代码块

```
typedef struct
{
    uint32_t const *dataInputPtr;  /*!< Specifies current processing data
input pointer */
    uint32_t *dataOutputPtr;       /*!< Specifies current processing data
output pointer */
    uint32_t msgLen;               /*!< Specifies the length of message */
    hcu_msg_type_t msgType;        /*!< Specifies the type of message */
    bool hwKeySelected;            /*!< Specifies hardware key or software key
selected */
    uint32_t hwKeySlot;            /*!< Specifies index of hardware key in
HCU_NVR */
    uint32_t const * swKeyPtr;     /*!< Specifies current software key */
    hcu_key_size_t keySize;        /*!< Specifies the key size */
} aes_algorithm_config_t;

status_t HCU_DRV_EncryptECB(aes_algorithm_config_t * aesAlgCfg);
......
status_t HCU_DRV_DecryptECB(aes_algorithm_config_t * aesAlgCfg);
```

HCU v2 版本驱动在 AES-ECB 加密需要传入一个结构体指针，下面对这个指针进行解释：

1. dataInputPtr：待加密，或待解密的数据
2. dataOutputPtr：加密后，或解密后的结果
3. msgLen：输入数据的长度
4. msgType：当前数据块的类型，指示当前是第一个block，或最后一个 block，或中间的 block
5. hwKeySelected：是否选择加载硬件密钥
6. hwKeySlot：硬件密钥的序列号，即选择加载 HCU_NVR 第几组密钥
7. swKeyPtr：软件密钥

8. keySize：密钥长度

AES-CMAC

HCU v1

代码块

```
typedef enum
{
    MSG_START = 0x02U,       /*!< Start of a message block */
    MSG_END = 0x01U,         /*!< End of a message block */
    MSG_ALL = 0x03U,         /*!< All message in one block */
    MSG_MIDDLE = 0x00U,      /*!< All message in one block */
} hcu_msg_type_t;

typedef struct
{
    uint8_t *macPtr;        /*!< Specifies the mac used for the CMAC operation */
    uint8_t macLen;         /*!< Specifies the length of the mac used for the CMAC operation */
} hcu_cmac_config_t;

status_t HCU_DRV_GenerateMAC(const void *msg, uint16_t msgLen, hcu_msg_type_t msgType,
                             hcu_cmac_config_t *cmacConfig);
......
status_t HCU_DRV_AuthorizeMAC(const void *msg, uint16_t msgLen, hcu_msg_type_t msgType,
                              hcu_cmac_config_t *cmacConfig);

/* Generate CMAC in one block */
HCU_DRV_GenerateMAC(plainText, 64, MSG_ALL, &cmacConfig);

/* Generate CMAC in four blocks */
HCU_DRV_GenerateMAC(plainText + 0, 16, MSG_START, &cmacConfig);
HCU_DRV_GenerateMAC(plainText + 4, 16, MSG_MIDDLE, &cmacConfig);
HCU_DRV_GenerateMAC(plainText + 8, 16, MSG_MIDDLE, &cmacConfig);
HCU_DRV_GenerateMAC(plainText +12, 16, MSG_END, &cmacConfig);
```

AES-CMAC 生成 MAC 签名，需要输入明文，明文长度，当前 block 类型，以及 MAC 的长度。

如果验签 MAC，同样需要输入明文，明文长度，当前 block 类型，以及 MAC 的长度，另外还需输入签名结果 MAC。

line22 演示了如何在一个 block 中完成签名验签。

line25~28 演示了当明文长度过大时，需要分多个 block 进行签名验签时的操作。

HCU v2

代码块

```
1  status_t HCU_DRV_GenerateMAC(aes_algorithm_config_t * aesAlgCfg,
   hcu_cmac_config_t *cmacConfig);
2
3  status_t HCU_DRV_AuthorizeMAC(aes_algorithm_config_t * aesAlgCfg,
   hcu_cmac_config_t *cmacConfig);
4
5  /* Generate CMAC in one block */
6  encryptSwCfg.msgType = MSG_ALL;
7  HCU_DRV_GenerateMAC(&encryptSwCfg, &cmacConfig);
8
9  /* Generate CMAC in two blocks */
10 encryptSwCfg.dataInputPtr = plainText;
11 encryptSwCfg.msgLen = 32
12 encryptSwCfg.msgType = MSG_START;
13 HCU_DRV_GenerateMAC(&encryptSwCfg, &cmacConfig);
14
15 encryptSwCfg.dataInputPtr = plainText + 8;
16 encryptSwCfg.msgLen = 32
17 encryptSwCfg.msgType = MSG_END;
18 HCU_DRV_GenerateMAC(&encryptSwCfg, &cmacConfig);
```

代码逻辑与 HCU v1 版本类似，只是在分多个 block 的时候，需要改变配置。

SHA

HCU v1

代码块

```
1  typedef enum
2  {
3      HCU_SHA_256 = 0x01U,        /*!< SHA 256 algorithm */
4      HCU_SHA_384 = 0x02U,        /*!< SHA 384 algorithm */
5  } hcu_sha_type_t;
6
7  status_t HCU_DRV_GenerateSHA(const void *msg, uint16_t msgLen, uint32_t
   totalLen,
```

```
                                hcu_sha_type_t shaType, hcu_msg_type_t msgType,
void *result);

status_t HCU_DRV_AuthorizeSHA(const void *msg, uint16_t msgLen, uint32_t
totalLen,
                                hcu_sha_type_t shaType, hcu_msg_type_t msgType,
void *result, void *trueResult);
```

因为 SHA 是非对称加密，需要提前知道消息的总长度，所以 SHA 签名需要输入 消息，当前 block 长度，消息总长度，SHA算法 (SHA-256/384)，当前 block 类型。

验签也类似，输入 消息，当前 block 长度，消息总长度，SHA算法 (SHA-256/384)，当前 block 类型，以及签名值，另外 MCU 可以将正确的签名值一并返回，即 trueResult.

分 Block 与 CMAC 类似，不再赘述

HCU v2

代码块

```
typedef struct
{
    uint32_t const *dataInputPtr;  /*!< Specifies current processing data
input pointer */
    uint32_t *dataOutputPtr;       /*!< Specifies current processing data
output pointer */
    uint32_t msgLen;               /*!< Specifies the length of message */
    uint32_t totalLen;             /*!< Specifies the total length of message
*/
    hcu_msg_type_t msgType;        /*!< Specifies the type of message */
    hcu_sha_type_t shaType;        /*!< Specifies the type of SHA */
} sha_algorithm_config_t;

status_t HCU_DRV_GenerateSHA(sha_algorithm_config_t * shaAlgCfg);

status_t HCU_DRV_AuthorizeSHA(sha_algorithm_config_t * shaAlgCfg);
```

与 AES 加密类似，SHA也有单独的配置结构体，只是少了密钥相关的参数，多了消息总长度，与 SHA 算法选择。

分 Block 与 CMAC 类似，不再赘述

RSA

代码块

```c
typedef struct
{
    uint8_t privateKeyLen;                          /*!< Specifies the words of private key length */
    uint32_t d[FEATURE_HCU_RSA_MAX_WORD_LENGTH]; /*!< Specifies the private key */
    uint8_t publicKeyLen;                           /*!< Specifies the words of public key length */
    uint32_t e[FEATURE_HCU_RSA_MAX_WORD_LENGTH]; /*!< Specifies the public key */
    hcu_rsa_alg_t opLength;                         /*!< Specifies the length of RSA operation */
    uint32_t n[FEATURE_HCU_RSA_MAX_WORD_LENGTH]; /*!< Specifies the N for RSA */
#if FEATURE_RSA_HAS_HARDWARE_KEY_LOAD
    bool rsaLoadHwPrivateKey;                       /*!< Specifies RSA if use hardware private key */
    uint8_t rsaHwPrivateKeySlot;                    /*!< Specifies RSA hardware private key select */
    bool rsaLoadHwPublicKey;                        /*!< Specifies RSA if use hardware public key */
    uint8_t rsaHwPublicKeySlot;                     /*!< Specifies RSA hardware public key select */
#endif /* FEATURE_RSA_HAS_HARDWARE_KEY_LOAD */
} hcu_rsa_config_t;

typedef struct
{
#if FEATURE_HCU_ECC_ENGINE
    hcu_ecc_config_t const *eccConfig; /*!< Specifies the ecc configuration  */
#endif /* FEATURE_HCU_ECC_ENGINE */
#if FEATURE_HCU_RSA_ENGINE
    hcu_rsa_config_t const *rsaConfig; /*!< Specifies the rsa configuration  */
#endif /* FEATURE_HCU_RSA_ENGINE */
} pke_user_config_t;

status_t PKE_DRV_Init(const pke_user_config_t * userConfig, pke_state_t *state);

status_t HCU_DRV_EncryptRSA(uint32_t const *plainText, uint32_t length, uint32_t *cipherText);

status_t HCU_DRV_DecryptRSA(uint32_t const *cipherText, uint32_t length, uint32_t *plainText);
```

在使用 RSA 计算前，需要调用 *PKE_DRV_Init* 来为 RSA 选择参数，下面对其参数进行说明：

1. privateKeyLen：私钥长度
2. d：私钥
3. publicKeyLen：公钥长度
4. e：公钥
5. opLength：选择 RSA-1024/2048/3072/4096
6. n：RSA 的参数之一

随后调用加密或解密 RSA 函数。

需要注意的是，该 API 为 RSA_NO_PADDING 模式，即 RSA 计算对明文长度有限制，例如 RSA-4096，则需要明文长度为 4096bit，即 512 bytes。若不足 512 bytes，需要自行补全，若超过 512 bytes，需自行切割。

ECC

代码块

```
typedef struct
{
    hcu_ecc_alg_t opLength;                          /*!< Specifies the length of ECC operation */
    uint32_t a[FEATURE_HCU_ECC_MAX_WORD_LENGTH];     /*!< Specifies the a of ellipse */
    uint32_t b[FEATURE_HCU_ECC_MAX_WORD_LENGTH];     /*!< Specifies the b of ellipse */
    uint32_t p[FEATURE_HCU_ECC_MAX_WORD_LENGTH];     /*!< Specifies the p of ellipse */
    uint32_t gx[FEATURE_HCU_ECC_MAX_WORD_LENGTH];    /*!< Specifies the Gx of base point G */
    uint32_t gy[FEATURE_HCU_ECC_MAX_WORD_LENGTH];    /*!< Specifies the Gy of base point G */
    uint32_t n[FEATURE_HCU_ECC_MAX_WORD_LENGTH];     /*!< Specifies the n of ellipse */
} hcu_ecc_config_t;

typedef struct
{
    uint32_t *pointX;   /*!< Pointer to the X of point */
    uint32_t *pointY;   /*!< Pointer to the Y of point */
} hcu_ecc_point_t;

typedef struct
{
#if FEATURE_HCU_ECC_ENGINE
    hcu_ecc_config_t const *eccConfig; /*!< Specifies the ecc configuration  */
```

```
#endif /* FEATURE_HCU_ECC_ENGINE */
#if FEATURE_HCU_RSA_ENGINE
    hcu_rsa_config_t const *rsaConfig; /*!< Specifies the rsa configuration  */
#endif /* FEATURE_HCU_RSA_ENGINE */
} pke_user_config_t;

status_t PKE_DRV_Init(const pke_user_config_t * userConfig, pke_state_t
*state);

status_t HCU_DRV_ECCPointMul(uint32_t const *K, hcu_ecc_point_t const *pointP,
                             hcu_ecc_point_t *point);

status_t HCU_DRV_ECCPointAdd(hcu_ecc_point_t const *pointP, hcu_ecc_point_t
const *pointQ,
                             hcu_ecc_point_t *pointR);
```

同样在 ECC 计算中，需要调用 *PKE_DRV_Init* 来为 ECC 选择曲线参数，ECC 参数与上文 ECC 章节中介绍的参数一致，只是多了个 ECC 长度选择，ECC-128/192/256。

ECC 提供了点乘运算与点加运算，供用户自由发挥。

PKE 计算

代码块

```
status_t HCU_DRV_PreCalculate(uint32_t const *N, uint32_t *result);

status_t HCU_DRV_GetPKESub(uint32_t const *A, uint32_t const *E, uint32_t
*result);

status_t HCU_DRV_GetPKEAdd(uint32_t const *A, uint32_t const *E, uint32_t
*result);

status_t HCU_DRV_GetPKEDouble(uint32_t const *A, uint32_t *result);

status_t HCU_DRV_GetPKESquare(uint32_t const *A, uint32_t const *R2, uint32_t
*result);

status_t HCU_DRV_GetPKEMultiply(uint32_t const *A, uint32_t const *E, uint32_t
const *R2, uint32_t *result);

status_t HCU_DRV_GetPKEInverse(uint32_t const *A, uint32_t const *N, uint32_t
const *R2, uint32_t *result);

void hcu_pke_test(void)
```

```
{
    pkeUserConfig.eccConfig = &eccConfig;
    PKE_DRV_Init(&pkeUserConfig, &pkeState);

    /* Pre-calculation */
    HCU_DRV_PreCalculate(eccConfig.p, R2);
    /* (A + A) % N */
    HCU_DRV_GetPKEDouble(eccConfig.gx, result);

    /* (A + E) % N */
    HCU_DRV_GetPKEAdd(eccConfig.gx, eccConfig.gy, result);

    /* (A - E) % N */
    HCU_DRV_GetPKESub(eccConfig.gx, eccConfig.gy, result);

    /* A * A * R % N */
    HCU_DRV_GetPKESquare(eccConfig.gx, R2, result);

    /* A * E * R % N */
    HCU_DRV_GetPKEMultiply(eccConfig.gx, eccConfig.gy, R2, result);

    /* 1 / A % N */
    HCU_DRV_GetPKEInverse(eccConfig.gx, eccConfig.p, R2, result);
}
```

此外还提供了一系列大数据的运算，供用户 DIY 加密算法。

ECDSA

代码块

```
typedef struct
{
    uint32_t const *privateKey;             /*!< Specifies private key for ECDSA, ignore if during signature verification */
    hcu_ecc_point_t *publicKey;             /*!< Specifies public key generated by private key by ECC */
    uint32_t const *K;                      /*!< Specifies random number K for ECDSA signature generate, ignore if during signature verification */
    hcu_sha_type_t shaType;                 /*!< Specifies the type of SHA */
    uint32_t msgLen;                        /*!< Specifies the message length for ECDSA. */
    uint32_t const *message;                /*!< Specifies the message for ECDSA. */
    uint32_t *hashResult;                   /*!< Specifies the hash result (little endian) for message. */
```

```
    uint32_t *R;                        /*!< Specifies the R of signature
pair (r, s) for ECDSA */
    uint32_t *S;                        /*!< Specifies the S of signature
pair (r, s) for ECDSA */
} hcu_ecdsa_config_t;
#endif /* FEATURE_HCU_ECC_ENGINE */

status_t PKE_DRV_Init(const pke_user_config_t * userConfig, pke_state_t
*state);

status_t HCU_DRV_ECDSA_GenerateSignature(hcu_ecdsa_config_t * ecdsaCfg);

status_t HCU_DRV_ECDSA_VerifySignature(hcu_ecdsa_config_t * ecdsaCfg);
```

同样，使用 ECDSA 算法仍需要调用 *PKE_DRV_Init* 来为 ECC 选择曲线参数。另外关于 ECDSA 的参数说明如下：

1. privateKey：私钥，在签名时需要输入，在验签时不需要输入
2. publicKey：公钥，公钥是曲线上的一个点，在签名时不需要输入，在验签时需要输入
3. K：随机数 K，仅在签名时需要输入，可通过 **TRNG** 来产生一组随机数
4. shaType：SHA 算法选择 SHA-256/384
5. msgLen：消息的长度
6. message：消息
7. hashResult：消息的摘要内容，为方便用户 debug，特地将消息摘要返回
8. (R, S)：签名对

注意：不同的随机 K 仅对生成的签名对有影响，对验签结果无任何影响，但尽量保证每次签名时使用的 K 不同，依次来保护私钥信息。

各系列芯片对比

功能对比

	YTM32B1ME0	YTM32B1MD1	YTM32B1MC0	YTM32B1ME1	YTM32B1HA0	YT
KeySize	128/192/256	128	128	128/192/256	128/192/256	
AES-ECB	Y	Y	Y	Y	Y	
AES-CBC	Y	Y	Y	Y	Y	
AES-CTR	Y	N	N	Y	Y	
AES-CCM	Y	N	N	Y	Y	
AES-CMAC	Y	Y	Y	Y	Y	
SM4-ECB	Y	N	N	N	Y	
SHA-256	Y	N	N	Y	Y	
SHA-384	Y	N	N	Y	Y	
RSA	N	N	N	N	N	RSA-1
ECC	N	N	N	N	N	ECC-
支持DMA搬运SHA数据	N	N	N	Y	Y	
支持SHA认证	N	N	N	N	Y	
有数据FIFO	Y	Y	N	Y	Y	
AES单Block最大输入数据个数	0x7FFF	0x7FFF	0xFFFF	0xFFFF	0x7FFF	
SHA单Block最大输入数据个数	0xFFFF	-	-	0xFFFF	0xFFFF	
SDK 驱动版本	HCU_v1	HCU_v1	HCU_v2	HCU_v1	HCU_v1	
HCU_NVR Start address	0x1000_0000	0x1000_0200	0x1000_3800	0x1000_0800	0x1000_0800	0x
HCU_NVR End address	0x1000_03FF	0x1000_03FF	0x1000_39FF	0x1000_0BFF	0x1000_17FF	0x
HCU_NVR 密钥存储单元	32-bytes	16-bytes	16-bytes	32-bytes	32-bytes	
HCU_NVR 最多存储密钥组	32	32	32	32	128	
HCU_NVR 可擦除	Y	N	Y	Y	Y	
RSA_NVR Start address	-	-	-	-	-	0x
RSA_NVR End address	-	-	-	-	-	0x
RSA_NVR 密钥存储单元	-	-	-	-	-	5
RSA_NVR 最多存储密钥组	-	-	-	-	-	

RSA_NVR 可擦除	-	-	-	-	-	

计算效率对比

YTM32B1MC0 HCU 效率

80MHz FIRC 主频，各个加密算法的速度比较

	1KB Message (us)	4KB Message (us)
AES-ECB	176.75	697.91
AES-CBC	178.63	699.64
AES-CMAC	105.2	400.45

YTM32B1ME1 HCU 效率

120MHz PLL 主频，各个加密算法的速度比较

GCC 优化等级-O1 (-Ofast 还能少10us左右)

	1KB Message (us)	4KB Message (us)
AES-ECB	65.38/64.66	253.52/249.7
AES-CBC	66.44/65.02	254.54/250.08
AES-CTR	65.34/64.52	250.2/248.38
AES-CCM	70.4/64.72	256.42/248.74
AES-CMAC	52.46	168.44
SM4-ECB	-	-
SHA-256	45.46	168.26
SHA-384	47.68	167.46

YTM32B1ME0 HCU 效率

120MHz PLL 主频，各个加密算法的速度比较

	1KB Message (us)	4KB Message (us)
AES-ECB	87.92	342.2
AES-CBC	87.65	340.34
AES-CTR	88.18	342.49
AES-CCM	104.49	357.28
AES-CMAC	75.68	272.37
SM4-ECB	86.75	339.52
SHA-256	65.54	247.92
SHA-384	65.86	248.25

YTM32B1HA0 HCU 效率

200MHz PLL 主频，各个加密算法的速度比较

	未开 Cache		开 Cache	
	1KB Message (us)	4KB Message (us)	1KB Message (us)	4KB Message (us)
AES-ECB	160.55	620.24	45.74	164.55
AES-CBC	174.34	673.28	42.76	162.06
AES-CTR	175.74	678.47	43.2	161.68
AES-CCM	197.57	695.32	72.82	240.72
AES-CMAC	129.11	452.15	36.03	121.29
SM4-ECB	155.17	598.69	42.24	161.42
SHA-256	124.88	467.23	32.92	118.88
SHA-384	124.51	467.7	26.5	93.43

YTM32B1MD2 HCU 效率

80M PLL 主频

注意：以下测试结果单位均为 us，RSA 加密公钥均为 0x10001，ECDSA 仅支持 ECC-256 曲线 + SHA-256

	Flash Cache RAM Cache	Flash Cache RAM no Cache	Flash no Cache RAM Cache	Flash no Cache RAM no Cache	Flash Cache DMA Enable RAM Cache
AES-ECB 1kB	159	160	206	225	128

AES-ECB 4kB	599	602	794	878	397
AES-CBC 1kB	162	162	209	231	125
AES-CBC 4kB	611	604	803	894	394
AES-CMAC 1kB	92	110	162	97	90
AES-CMAC 4kB	327	395	613	342	262
SHA-256 1kB	90	120	138	162	77
SHA-256 4kB	319	430	513	606	186
ECC-192 点加	2324	2428	2888	2963	2327
ECC-192 点乘	268	282	321	334	270
ECC-256 点加	4079	4211	4804	4894	4082
ECC-256 点乘	442	458	505	515	444
ECC-384 点加	7406	7544	8160	8257	7409
ECC-384 点乘	1066	1088	1141	1158	1068
RSA-1024 加密	234	234	241	242	234
RSA-1024 解密	11460	11460	11473	11473	11460
RSA-2048 加密	855	855	873	874	857
RSA-2048 解密	86008	86008	86035	86035	86007
RSA-3072 加密	1917	1917	1942	1943	1918
RSA-3072 解密	277092	277092	277132	277133	277090
RSA-4096 加密	3347	3347	3381	3381	3348
RSA-4096 解密	657466	657466	657520	657520	657466
ECDSA 1kB 签名	9023	9328	10596	10819	9023
ECDSA 1kB 验签	9445	9670	10953	11200	9445
ECDSA 4kB 签名	9258	9649	10967	11263	9258

ECDSA 4kB 验签	9677	10008	11323	11644	9677

（注：部分内容由豆包 AI 生成）