CS334 – VirtIO Crypto Project Report

Jiarun Zhu, Shengli Zhou, Yicheng Xiao

2025-01-15

Github 仓库：Jaredanwolfgang/CS334_Operating_System_VirtIO

目录

1. 项目概述

1.1 实现功能

- 禁用的设备特性
  - 由于当前 qemu 设备不支持 `VIRTIO_CRYPTO_F_REVISION_1`，无法测试，因此将其禁用
  - 由于当前 qemu 设备不支持多条数据队列，因此仅使用第一条数据队列进行数据传输
- 支持的加密算法服务
  - Symmetric Algorithms
    - CIPHER
    - Chaining Algorithm (CIPHER + HASH/MAC)
  - HASH
  - MAC
  - AEAD
  - AKCIPHER
- 与设备之间的异步通信
  - DMA 空间的动态分配和回收
  - 已提交请求的记录与状态更新
- 支持用户态下调用加密算法
- AKCIPHER 算法中 RSA 密钥的自动生成

1.2 项目结构

```
crypto
| - device.rs # 定义 CryptoDevice 与相关方法
| - header.rs # 定义请求与响应结构与相关方法
| - config.rs # 定义设备配置空间与读取配置方法
| - mod.rs    # 注册各模块与设备名称
| - service   # 算法包文件夹
  | - services.rs # 定义各算法包与相关包管理方法
  | - sym.rs      # 定义 CIPHER 与 ALG_CHAIN 对称加密算法包的各类具体算法与操作
  | - hash.rs     # 定义 HASH 算法包的各类具体算法与操作
  | - mac.rs      # 定义 MAC 算法包的各类具体算法与操作
  | - aead.rs     # 定义 AEAD 算法包的各类具体算法与操作
  | - akcipher.rs # 定义 AKCIPHER 算法包的各类具体算法与操作
  | - mod.rs      # 注册各算法包模块
```

1.3 设备配置信息

```
virtio_crypto_config = VirtioCryptoConfig {
    status: 1,
    max_dataqueues: 1,
    crypto_services: 23,
    cipher_algo_l: 8,
    cipher_algo_h: 0,
    hash_algo: 4,
    mac_algo_l: 0,
    mac_algo_h: 0,
    aead_algo: 0,
    max_cipher_key_len: 64,
    max_auth_key_len: 512,
    akcipher_algo: 2,
    max_size: 4294967231,
}
```

2. API

2.1 CryptoDevice

CryptoDevice 是对 Virtio-Crypto 设备的抽象，实现对设备的配置管理、请求和响应缓冲区管理、控制和数据队列管理等操作。

2.1.1 字段

```rust
pub struct CryptoDevice {
    /// 用于管理虚拟加密设备配置
    pub config_manager: ConfigManager<VirtioCryptoConfig>,
    /// 用于发送请求数据的 DMA 流
    pub request_buffer: DmaStream,
    /// 用于接收响应数据的 DMA 流
    pub response_buffer: DmaStream,
    /// 用于分配请求缓冲区的分配器
    pub request_buffer_allocator: SpinLock<SliceAllocator>,
    /// 用于分配响应缓冲区的分配器
    pub response_buffer_allocator: SpinLock<SliceAllocator>,
    /// 用于管理控制队列中已提交请求的管理器
    pub controlq_manager: SpinLock<SubmittedReqManager>,
    /// 用于管理数据队列中已提交请求的管理器
    pub dataq_manager: SpinLock<SubmittedReqManager>,
    /// 数据队列的集合
    pub dataqs: Vec<SpinLock<VirtQueue>>,
    /// 控制队列
    pub controlq: SpinLock<VirtQueue>,
    /// 虚拟 I/O 设备的传输层接口
    pub transport: SpinLock<Box<dyn VirtioTransport>>,
    /// 表示设备支持的加密服务的映射关系，用于确定设备的功能范围
    pub supported_crypto_services: CryptoServiceMap
}
```

2.1.2 方法

2.1.2.1 negotiate_features

驱动与设备进行特性协商。

注意：不支持特性: `VIRTIO_CRYPTO_F_REVISION_1`。

2.1.2.2 init

初始化设备，创建设备的请求与响应 DMA 流、控制与数据队列、请求管理器等。

注意：根据定义，一个 Virtio-Crypto 设备可以拥有一条控制队列和多条数据队列（调用多条 Data_Queue 以实现更高性能）。但由于当前 qemu 的 Virtio-Crypto 设备仅支持一条数据队列，因此该 Driver 仅使用第 1 条数据队列（dataqs[0]）进行数据传输。

2.1.2.3 refresh_state

刷新加密请求的状态，处理控制队列和数据队列中已完成的请求。

此方法依次遍历控制队列（controlq）和第一个数据队列（dataqs[0]），检查是否有已完成的请求，并调用相应的管理器来完成请求的清理操作。

注意事项： 当前仅处理 `dataqs[0]` 队列，扩展到更多数据队列时需要相应调整。

2.1.2.4 test_device
测试设备能否正常工作。

2.1.2.5 print_supported_crypto_algorithms
根据配置空间内容，输出设备支持的所有密码算法。在 qemu 的 cryptodev-backend-builtin 设备上运行，结果如下：

```
Supported Crypto Services and Algorithms:
- CIPHER
  - AES CBC
- HASH
  - SHA1
- AKCIPHER
  - RSA
```

2.1.2.6 get_resp_slice_from
从指定队列中获取完成请求的响应数据切片。

该方法通过提供的队列索引和请求的令牌（token），获取与请求对应的响应切片，并返回包含切片和实际写入长度的元组。

2.1.2.6.1 定义:

```rust
pub fn get_resp_slice_from(
  &self,
  queue_index: u32,
  token: u16
) -> (
  resp_slice: DmaStreamSlice<&DmaStream>,
  write_len: u32
)
```

2.1.2.6.2 方法逻辑:
1. 根据队列索引确定队列管理器：
   - 如果队列索引为 `Self::CONTROLQ`，选择 `controlq_manager`。
   - 否则，如果队列索引为 `Self::DATAQ`，选择 `dataq_manager`。
2. 调用 `refresh_state` 刷新队列状态，确保请求完成状态最新。
3. 通过令牌从管理器中检索提交的请求记录，使用自旋等待直到请求成功检索。
4. 创建响应切片，记录其数据范围，并释放对应的请求和响应缓冲区。
5. 同步响应切片的数据并返回切片和写入长度。

2.1.2.6.3 参数:
- `queue_index: u32`
  - 队列索引，指定目标队列。可以是 `Self::CONTROLQ` 或 `Self::DATAQ`，分别表示控制队列和数据队列。
- `token: u16`
  - 请求的令牌，用于标识和检索指定请求。

2.1.2.6.4 返回值:
- `resp_slice: DmaStreamSlice<&DmaStream>`
  - 指向响应数据的切片，用于读取数据。
- `write_len: u32`

- 表示响应切片中的有效数据长度。

2.1.2.6.5 注意事项:

- 如果请求未完成，方法会自旋等待，直到请求完成或成功检索为止。
- 调用此方法会释放请求和响应的分配缓冲区，以便重用。
- 切片在同步后可以安全读取，但需要确保设备未修改数据。

2.2 Cipher

`Cipher` 是一个抽象包，提供对加密/解密服务的封装。它支持创建加密会话、销毁会话，以及对数据进行加密和解密操作。

`Cipher` 不包含任何字段，仅用于提供静态方法。

2.2.1 常量

- `Cipher::ENCRYPT: u32`
  - 值：`1`
  - 表示加密操作
- `Cipher::DECRYPT: u32`
  - 值：`2`
  - 表示解密操作

2.2.2 方法

2.2.2.1 `send_create_session_request`

向设备发送创建一个加密/解密会话的异步请求。

2.2.2.1.1 定义:

```
pub fn send_create_session_request(
    device: &CryptoDevice,
    algo: u32,
    encrypt_or_decrypt: u32,
    cipher_key: &[u8]
) → (
    queue_index: u32,
    token: u16
)
```

2.2.2.1.2 参数:

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符，用于指定加密算法（如 `VIRTIO_CRYPTO_AES_CBC`）
- `encrypt_or_decrypt: u32`
  - 操作类型，可为 `Cipher::ENCRYPT` 或 `Cipher::DECRYPT`
- `cipher_key: &[u8]`
  - 密钥，用于初始化加密/解密会话

2.2.2.1.3 返回值:

- `queue_index: u32`
  - 用于传输该请求的队列编号，可为 `CryptoDevice::CONTROLQ` 或 `CryptoDevice::DATAQ`

- `token: u16`
  - 请求的令牌，用于标识和检索指定的请求。

2.2.2.2 `create_session`

创建一个加密/解密会话。该方法是对多个子方法的打包，实现阻塞式创建会话操作。

2.2.2.2.1 定义：

```
pub fn create_session(
    device: &CryptoDevice,
    algo: u32,
    encrypt_or_decrypt: u32,
    cipher_key: &[u8]
) → (
    session_id: u64
)
```

2.2.2.2.2 方法逻辑：

1. 调用 `Cipher::send_create_session_request` 方法发送创建会话请求，获取请求对应 `queue_index` 和 `token`。
2. 调用 `device::get_resp_slice_from` 方法，通过 `queue_index` 和 `token` 获取设备响应 `resp_slice`。
3. 使用 `VirtioCryptoCreateSessionInput` 结构解析 `resp_slice`，返回 `session_id`。

2.2.2.2.3 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符，用于指定加密算法（如 `VIRTIO_CRYPTO_AES_CBC`）
- `encrypt_or_decrypt: u32`
  - 操作类型，可为 `Cipher::ENCRYPT` 或 `Cipher::DECRYPT`
- `cipher_key: &[u8]`
  - 密钥，用于初始化加密/解密会话

2.2.2.2.4 返回值：

- `session_id: u64`
  - 创建的会话 ID

2.2.2.3 `send_destroy_session_request`

向设备发送销毁一个加密/解密会话的异步请求。

2.2.2.3.1 定义：

```
pub fn send_destroy_session_request(
    device: &CryptoDevice,
    session_id: u64
) → (
    queue_index: u32,
    token: u16
)
```

2.2.2.3.2 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `session_id: u64`
  - 要销毁的会话 ID

2.2.2.3.3 返回值：

- `queue_index: u32`
  - 用于传输该请求的队列编号，可为 `CryptoDevice::CONTROLQ` 或 `CryptoDevice::DATAQ`
- `token: u16`
  - 请求的令牌，用于标识和检索指定的请求

2.2.2.4 `destroy_session`

销毁一个加密/解密会话。该方法是多个子方法的打包，实现阻塞式销毁会话操作。

2.2.2.4.1 定义：

```
pub fn destroy_session(
    device: &CryptoDevice,
    session_id: u64
)
```

2.2.2.4.2 方法逻辑：

1. 调用 `Cipher::send_destroy_session_request` 方法发送销毁会话请求，获取请求对应 `queue_index` 和 `token`。
2. 调用 `device::get_resp_slice_from` 方法，通过 `queue_index` 和 `token` 获取设备响应 `resp_slice`。
3. 使用 `VirtioCryptoDestroySessionInput` 结构解析 `resp_slice`，确认返回状态。

2.2.2.4.3 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `session_id: u64`
  - 要销毁的会话 ID

2.2.2.4.4 返回值：

- 无

2.2.2.5 `send_encrypt_or_decrypt_request`

向设备发送在某个会话对数据执行加密或解密操作的异步请求。

2.2.2.5.1 定义：

```
pub fn send_encrypt_or_decrypt_request(
    device: &CryptoDevice,
    algo: u32,
    session_id: u64,
    encrypt_or_decrypt: u32,
    iv: &[u8],
    src_data: &[u8],
    dst_data_len: u32
) → (
    queue_size: u32,
    token: u16
)
```

2.2.2.5.2 返回值：

- `queue_index: u32`
  - 用于传输该请求的队列编号，可为 `CryptoDevice::CONTROLQ` 或 `CryptoDevice::DATAQ`
- `token: u16`
  - 请求的令牌，用于标识和检索指定的请求

2.2.2.5.3 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符
- `session_id: u64`
  - 当前加密/解密会话 ID
- `encrypt_or_decrypt: u32`
  - 操作类型，可为 `Cipher::ENCRYPT` 或 `Cipher::DECRYPT`
- `iv: &Vec<u8>`
  - 初始化向量（IV）
- `src_data: &Vec<u8>`
  - 原始数据
- `dst_data_len: u32`
  - 输出数据的长度

2.2.2.6 `encrypt_or_decrypt`

在某个会话对数据执行加密或解密操作。该方法是多个子方法的打包，实现阻塞式对数据加密或解密。

2.2.2.6.1 定义：

```rust
pub fn encrypt_or_decrypt(
    device: &CryptoDevice,
    algo: u32,
    session_id: u64,
    encrypt_or_decrypt: u32,
    iv: &[u8],
    src_data: &[u8],
    dst_data_len: u32
) -> (
    dst_data: Vec<u8>
)
```

2.2.2.6.2 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符
- `session_id: u64`
  - 当前加密/解密会话 ID
- `encrypt_or_decrypt: u32`
  - 操作类型，可为 `Cipher::ENCRYPT` 或 `Cipher::DECRYPT`
- `iv: &Vec<u8>`
  - 初始化向量（IV）
- `src_data: &Vec<u8>`
  - 原始数据
- `dst_data_len: u32`
  - 输出数据的长度

2.2.2.6.3 返回值：

- `dst_data: Vec<u8>`
  - 加密或解密后的数据

2.2.2.7 `encrypt`

创建会话，执行加密操作，然后销毁会话。该方法是对多个子方法的打包，实现阻塞式的伪无状态加密操作。

2.2.2.7.1 定义：

```rust
pub fn encrypt(
    device: &CryptoDevice,
    algo: u32,
    cipher_key: &[u8],
    iv: &[u8],
    src_data: &[u8]
) -> (
    dst_data: Vec<u8>
)
```

2.2.2.7.2 方法逻辑：

1. 调用 `create_session` 创建加密会话。

2. 调用 `encrypt_or_decrypt` 执行加密操作。
3. 调用 `destroy_session` 销毁会话。
4. 返回加密后的数据。

2.2.2.7.3 参数：

- `device: &CryptoDevice` 使用的加密设备
- `algo: u32` 算法标识符
- `cipher_key: &[u8]` 密钥
- `iv: &Vec<u8>` 初始化向量（IV）
- `src_data: &Vec<u8>` 原始数据

2.2.2.7.4 返回值：

- `encrypted_data: Vec<u8>`：加密后的数据。

2.2.2.7.5 注意事项：

- 每次调用该方法进行加密时都需要执行创建/销毁会话操作，多次重复调用可能导致性能损耗。如果对加密性能有更高的要求，可以通过调用 `create_session`、`encrypt_or_decrypt`、`destroy_session` 手动管理会话并加密数据。

2.2.2.8 `decrypt`

创建会话，执行解密操作，然后销毁会话。该方法是对多个子方法的打包，实现阻塞式的伪无状态解密操作。

定义：

```
pub fn decrypt(
    device: &CryptoDevice,
    algo: u32,
    cipher_key: &[u8],
    iv: &[u8],
    src_data: &[u8]
) → (
  dst_data: Vec<u8>
)
```

参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符
- `cipher_key: &[u8]`
  - 密钥
- `iv: &Vec<u8>`
  - 初始化向量（IV）
- `src_data: &Vec<u8>`
  - 原始数据

返回值：

- `decrypted_data: Vec<u8>`
  - 解密后的数据。

逻辑：

1. 调用 `create_session` 创建解密会话。
2. 调用 `encrypt_or_decrypt` 执行解密操作。
3. 调用 `destroy_session` 销毁会话。
4. 返回解密后的数据。

2.2.2.8.1 注意事项：

- 每次调用该方法进行解密时都需要执行创建/销毁会话操作，多次重复调用可能导致性能损耗。如果对解密性能有更高的要求，可以通过调用 `create_session`、`encrypt_or_decrypt`、`destroy_session` 手动管理会话并解密数据。

2.2.3 用法示例

```rust
fn main() {
    let device = ...;
    let algo = VIRTIO_CRYPTO_CIPHER_AES_CBC; // 加密算法
    let cipher_key = b"mysecretkey"; // 密钥
    let iv = vec![...]; // 初始化向量
    let data = vec![...]; // 原始数据

    let encrypted_data =
        Cipher::encrypt(&device, algo, &cipher_key, &iv, &data);
    let decrypted_data =
        Cipher::decrypt(&device, algo, &cipher_key, &iv, &encrypted_data);

    assert_eq!(data, decrypted_data,
        "The initial data and decrypted data of CIPHER are inconsistent");

}
```

2.2.4 功能测试

2.2.4.1 测试代码

```rust
let cipher_key = [0_u8; 16];
let iv = vec![0x00; 16];
let data1: Vec<u8> = vec![
    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10,
    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10,
];

early_println!("[Test] Original data for CIPHER: {:?}", data1);
let cipher_encrypt_result = Cipher::encrypt(
    &device,
    VIRTIO_CRYPTO_CIPHER_AES_CBC,
    &cipher_key,
    &iv,
    &data1,
);
early_println!(
    "[Test] Encrypted data for CIPHER: {:?}",
    cipher_encrypt_result
);
let cipher_decrypt_result = Cipher::decrypt(
    &device,
    VIRTIO_CRYPTO_CIPHER_AES_CBC,
    &cipher_key,
    &iv,
    &cipher_encrypt_result,
);
early_println!(
    "[Test] Decrypted data for CIPHER: {:?}",
    cipher_decrypt_result
);

assert_eq!(
    data1, cipher_decrypt_result,
    "[Test] The initial data and decrypted data of CIPHER are inconsistent"
);
early_println!("[Test] CIPHER test pass!");
```

测试方式：加密再解密一段明文，如果解密的结果与初始明文相同则通过测试。

2.2.4.2 测试结果

```
[Test] Original data for CIPHER: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1, 2, 3, 4, 5,
6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
request in controlq with token: 0 has finished
VirtioCryptoCreateSessionInput { session_id: 0, status: 0, padding: 0 }
request in dataq with token: 0 has finished
Data: [65, 9F, 2A, BF, C5, BC, 7, 8, FB, 96, E7, A4, 4A, F9, D5, 95, 22, A9, 7C, BA, 80, 92, D5, D4, A5,
 F2, 68, D2, AC, 88, E7, C0]
[Test] Encrypted data for CIPHER: [101, 159, 42, 191, 197, 188, 7, 8, 251, 150, 231, 164, 74, 249, 213,
149, 34, 169, 124, 186, 128, 146, 213, 212, 165, 242, 104, 210, 172, 136, 231, 192]
request in controlq with token: 0 has finished
request in controlq with token: 2 has finished
VirtioCryptoCreateSessionInput { session_id: 0, status: 0, padding: 0 }
request in dataq with token: 0 has finished
Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10]
[Test] Decrypted data for CIPHER: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1, 2, 3, 4, 5,
 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
[Test] CIPHER test pass!
```

将数组 `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]` 加密再解密后得到的结果与原始数组相同，测试通过。

2.3 Hash

`Hash` 是一个抽象包，提供对加密/解密服务的封装。它支持创建加密会话、销毁会话，以及对数据进行加密和解密操作。

`Hash` 不包含任何字段，仅用于提供静态方法。

注意：由于当前 qemu 设备不支持 Hash 算法包，因此该算法包功能无法测试。

2.3.1 方法

2.3.1.1 `send_create_session_request`

向设备发送创建一个加密/解密会话的异步请求。

2.3.1.1.1 定义：

```rust
pub fn send_create_session_request(
  device: &CryptoDevice,
  algo: u32,
) -> (
  queue_index: u32,
  token: u16
)
```

2.3.1.1.2 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符，用于指定加密算法（如 `VIRTIO_CRYPTO_HASH_MD5`）

2.3.1.1.3 返回值：

- `queue_index: u32`
  - 用于传输该请求的队列编号，可为 `CryptoDevice::CONTROLQ` 或 `CryptoDevice::DATAQ`
- `token: u16`
  - 请求的令牌，用于标识和检索指定的请求。

2.3.1.2 `create_session`

创建一个加密/解密会话。该方法是对多个子方法的打包，实现阻塞式创建会话操作。

2.3.1.2.1 定义：

```
pub fn create_session(
  device: &CryptoDevice,
  algo: u32
) → (
  session_id: u64
)
```

2.3.1.2.2 方法逻辑：

1. 调用 `Hash::send_create_session_request` 方法发送创建会话请求，获取请求对应 `queue_index` 和 `token`。
2. 调用 `device::get_resp_slice_from` 方法，通过 `queue_index` 和 `token` 获取设备响应 `resp_slice`。
3. 使用 `VirtioCryptoCreateSessionInput` 结构解析 `resp_slice`，返回 `session_id`。

2.3.1.2.3 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符，用于指定加密算法（如 `VIRTIO_CRYPTO_HASH_MD5`）

2.3.1.2.4 返回值：

- `session_id: u64`
  - 创建的会话 ID

2.3.1.3 `send_destroy_session_request`

向设备发送销毁一个加密/解密会话的异步请求。

2.3.1.3.1 定义：

```
pub fn send_destroy_session_request(
  device: &CryptoDevice,
  session_id: u64
) → (
  queue_index: u32,
  token: u16
)
```

2.3.1.3.2 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `session_id: u64`
  - 要销毁的会话 ID

2.3.1.3.3 返回值：

- `queue_index: u32`
  - 用于传输该请求的队列编号，可为 `CryptoDevice::CONTROLQ` 或 `CryptoDevice::DATAQ`
- `token: u16`

- 请求的令牌，用于标识和检索指定的请求

2.3.1.4 `destroy_session`

销毁一个加密/解密会话。该方法是多个子方法的打包，实现阻塞式销毁会话操作。

2.3.1.4.1 定义：

```
pub fn destroy_session(
  device: &CryptoDevice,
  session_id: u64
)
```

2.3.1.4.2 方法逻辑：

1. 调用 `Hash::send_destroy_session_request` 方法发送销毁会话请求，获取请求对应 `queue_index` 和 `token`。
2. 调用 `device::get_resp_slice_from` 方法，通过 `queue_index` 和 `token` 获取设备响应 `resp_slice`。
3. 使用 `VirtioCryptoDestroySessionInput` 结构解析 `resp_slice`，确认返回状态。

2.3.1.4.3 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `session_id: u64`
  - 要销毁的会话 ID

2.3.1.4.4 返回值：

- 无

2.3.1.5 `send_hash_request`

向设备发送在某个会话对数据执行哈希操作的异步请求。

2.3.1.5.1 定义：

```
pub fn send_hash_request(
  device: &CryptoDevice,
  algo: u32,
  session_id: u64,
  src_data: &Vec<u8>
) → (
  queue_index: u32,
  token: u16
)
```

2.3.1.5.2 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符
- `session_id: u64`
  - 当前加密/解密会话 ID
- `src_data: &Vec<u8>`
  - 原始数据

2.3.1.5.3 返回值：

- `queue_index: u32`
  - 用于传输该请求的队列编号，可为 `CryptoDevice::CONTROLQ` 或 `CryptoDevice::DATAQ`
- `token: u16`
  - 请求的令牌，用于标识和检索指定的请求

2.3.1.6 `do_hash`

在某个会话对数据执行哈希操作。该方法是多个子方法的打包，实现阻塞式对数据进行哈希操作。

2.3.1.6.1 定义：

```
pub fn do_hash(
  device: &CryptoDevice,
  algo: u32,
  session_id: u64,
  src_data: &Vec<u8>
) → (
  dst_data: Vec<u8>
)
```

2.3.1.6.2 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符
- `session_id: u64`
  - 当前加密/解密会话 ID
- `src_data: &Vec<u8>`
  - 原始数据

2.3.1.6.3 返回值：

- `dst_data: Vec<u8>`
  - 加密或解密后的数据

2.3.1.7 `hash`

创建会话，执行哈希操作，然后销毁会话。该方法是对多个子方法的打包，实现阻塞式的伪无状态加密操作。

2.3.1.7.1 定义：

```
pub fn hash(
  device: &CryptoDevice,
  algo: u32,
  src_data: &Vec<u8>
) → (
  dst_data: Vec<u8>
)
```

2.3.1.7.2 方法逻辑：

1. 调用 `create_session` 创建会话。
2. 调用 `do_hash` 执行哈希操作。

3. 调用 `destroy_session` 销毁会话。
4. 返回哈希后的数据。

2.3.1.7.3 参数：

- `device: &CryptoDevice` 使用的加密设备
- `algo: u32` 算法标识符
- `src_data: &Vec<u8>` 原始数据

2.3.1.7.4 返回值：

- `dst_data: Vec<u8>`：加密后的数据。

2.3.1.7.5 注意事项：

- 每次调用该方法进行加密时都需要执行创建/销毁会话操作，多次重复调用可能导致性能损耗。如果对加密性能有更高的要求，可以通过调用 `create_session`、`do_hash`、`destroy_session` 手动管理会话并加密数据。

2.4 Mac

`Mac` 是一个抽象包，提供对加密/解密服务的封装。它支持创建加密会话、销毁会话，以及对数据进行加密和解密操作。

`Mac` 不包含任何字段，仅用于提供静态方法。

注意：由于当前 qemu 设备不支持 Mac 算法包，因此该算法包功能无法测试。

2.4.1 方法

2.4.1.1 `send_create_session_request`

向设备发送创建一个加密/解密会话的异步请求。

2.4.1.1.1 定义：

```
pub fn send_create_session_request(
  device: &CryptoDevice,
  algo: u32,
  auth_key: &[u8],
) -> (
  queue_index: u32,
  token: u16
)
```

2.4.1.1.2 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符，用于指定加密算法（如 `VIRTIO_CRYPTO_HASH_MD5`）
- `auth_key: &[u8]`
  - 认证密钥

2.4.1.1.3 返回值：

- `queue_index: u32`
  - 用于传输该请求的队列编号，可为 `CryptoDevice::CONTROLQ` 或 `CryptoDevice::DATAQ`
- `token: u16`

- 请求的令牌，用于标识和检索指定的请求。

2.4.1.2 `create_session`

创建一个加密/解密会话。该方法是对多个子方法的打包，实现阻塞式创建会话操作。

2.4.1.2.1 定义：

```
pub fn create_session(
  device: &CryptoDevice,
  algo: u32,
  auth_key: &[u8]
) → (
  session_id: u64
)
```

2.4.1.2.2 方法逻辑：

1. 调用 `Mac::send_create_session_request` 方法发送创建会话请求，获取请求对应 `queue_index` 和 `token`。
2. 调用 `device::get_resp_slice_from` 方法，通过 `queue_index` 和 `token` 获取设备响应 `resp_slice`。
3. 使用 `VirtioCryptoCreateSessionInput` 结构解析 `resp_slice`，返回 `session_id`。

2.4.1.2.3 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符，用于指定加密算法（如 `VIRTIO_CRYPTO_HASH_MD5`）
- `auth_key: &[u8]`
  - 认证密钥

2.4.1.2.4 返回值：

- `session_id: u64`
  - 创建的会话 ID

2.4.1.3 `send_destroy_session_request`

向设备发送销毁一个加密/解密会话的异步请求。

2.4.1.3.1 定义：

```
pub fn send_destroy_session_request(
  device: &CryptoDevice,
  session_id: u64
) → (
  queue_index: u32,
  token: u16
)
```

2.4.1.3.2 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `session_id: u64`
  - 要销毁的会话 ID

2.4.1.3.3 返回值：

- `queue_index: u32`
  - 用于传输该请求的队列编号，可为 `CryptoDevice::CONTROLQ` 或 `CryptoDevice::DATAQ`
- `token: u16`
  - 请求的令牌，用于标识和检索指定的请求

2.4.1.4 `destroy_session`

销毁一个加密/解密会话。该方法是多个子方法的打包，实现阻塞式销毁会话操作。

2.4.1.4.1 定义：

```
pub fn destroy_session(
  device: &CryptoDevice,
  session_id: u64
)
```

2.4.1.4.2 方法逻辑：

1. 调用 `Mac::send_destroy_session_request` 方法发送销毁会话请求，获取请求对应 `queue_index` 和 `token`。
2. 调用 `device::get_resp_slice_from` 方法，通过 `queue_index` 和 `token` 获取设备响应 `resp_slice`。
3. 使用 `VirtioCryptoDestroySessionInput` 结构解析 `resp_slice`，确认返回状态。

2.4.1.4.3 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `session_id: u64`
  - 要销毁的会话 ID

2.4.1.4.4 返回值：

- 无

2.4.1.5 `send_mac_request`

向设备发送在某个会话对数据执行 MAC 加密的异步请求。

2.4.1.5.1 定义：

```
pub fn send_mac_request(
  device: &CryptoDevice,
  algo: u32,
  session_id: u64,
  src_data: &Vec<u8>
) -> (
  queue_index: u32,
  token: u16
)
```

2.4.1.5.2 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符

- `session_id: u64`
  - 当前加密/解密会话 ID
- `src_data: &Vec<u8>`
  - 原始数据

2.4.1.5.3 返回值：

- `queue_index: u32`
  - 用于传输该请求的队列编号，可为 `CryptoDevice::CONTROLQ` 或 `CryptoDevice::DATAQ`
- `token: u16`
  - 请求的令牌，用于标识和检索指定的请求

2.4.1.6 `do_mac`

在某个会话对数据执行 MAC 加密操作。该方法是多个子方法的打包，实现阻塞式对数据进行 MAC 加密操作。

2.4.1.6.1 定义：

```
pub fn do_mac(
  device: &CryptoDevice,
  algo: u32,
  session_id: u64,
  src_data: &Vec<u8>
) → (
  dst_data: Vec<u8>
)
```

2.4.1.6.2 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符
- `session_id: u64`
  - 当前加密/解密会话 ID
- `src_data: &Vec<u8>`
  - 原始数据

2.4.1.6.3 返回值：

- `dst_data: Vec<u8>`
  - 加密或解密后的数据

2.4.1.7 `mac`

创建会话，执行 MAC 操作，然后销毁会话。该方法是对多个子方法的打包，实现阻塞式的伪无状态加密操作。

2.4.1.7.1 定义：

```rust
pub fn mac(
  device: &CryptoDevice,
  algo: u32,
  src_data: &Vec<u8>
) -> (
  dst_data: Vec<u8>
)
```

2.4.1.7.2 方法逻辑：

1. 调用 `create_session` 创建会话。
2. 调用 `do_mac` 执行 MAC 操作。
3. 调用 `destroy_session` 销毁会话。
4. 返回 MAC 后的数据。

2.4.1.7.3 参数：

- `device: &CryptoDevice` 使用的加密设备
- `algo: u32` 算法标识符
- `src_data: &Vec<u8>` 原始数据

2.4.1.7.4 返回值：

- `dst_data: Vec<u8>`：加密后的数据。

2.4.1.7.5 注意事项：

- 每次调用该方法进行加密时都需要执行创建/销毁会话操作，多次重复调用可能导致性能损耗。如果对加密性能有更高的要求，可以通过调用 `create_session`、`do_hash`、`destroy_session` 手动管理会话并加密数据。

2.5 Aead

`Aead` 是一个抽象包，提供对加密/解密服务的封装。它支持创建加密会话、销毁会话，以及对数据进行加密和解密操作。

`Aead` 不包含任何字段，仅用于提供静态方法。

注意：由于当前 qemu 设备不支持 Aead 算法包，因此该算法包功能无法测试。

2.5.1 常量

- `Aead::ENCRYPT: u32`
  - 值：`1`
  - 表示加密操作
- `Aead::DECRYPT: u32`
  - 值：`2`
  - 表示解密操作

2.5.2 方法

2.5.2.1 `send_create_session_request`

向设备发送创建一个加密/解密会话的异步请求。

2.5.2.1.1 定义：

```
pub fn send_create_session_request(
    device: &CryptoDevice,
    algo: u32,
    encrypt_or_decrypt: u32,
    key: &[u8],
    tag_len: u32,
    aad_len: u32,
) → (
    queue_index: u32,
    token: u16
)
```

2.5.2.1.2 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符，用于指定加密算法（如 `VIRTIO_CRYPTO_AES_CBC`）
- `encrypt_or_decrypt: u32`
  - 操作类型，可为 `Aead::ENCRYPT` 或 `Aead::DECRYPT`
- `key: &[u8]`
  - 密钥，用于初始化加密/解密会话
- `tag_len: u32`
  - 标签信息长度
- `aad_len: u32`
  - 附加认证信息长度

2.5.2.1.3 返回值：

- `queue_index: u32`
  - 用于传输该请求的队列编号，可为 `CryptoDevice::CONTROLQ` 或 `CryptoDevice::DATAQ`
- `token: u16`
  - 请求的令牌，用于标识和检索指定的请求。

2.5.2.2 `create_session`

创建一个加密/解密会话。该方法是对多个子方法的打包，实现阻塞式创建会话操作。

2.5.2.2.1 定义：

```
pub fn create_session(
    device: &CryptoDevice,
    algo: u32,
    encrypt_or_decrypt: u32,
    key: &[u8],
    tag_len: u32,
    aad_len: u32
) → (
    session_id: u64
)
```

2.5.2.2.2 方法逻辑：

1. 调用 `Aead::send_create_session_request` 方法发送创建会话请求，获取请求对应 `queue_index` 和 `token`。

2. 调用 `device::get_resp_slice_from` 方法，通过 `queue_index` 和 `token` 获取设备响应 `resp_slice`。
3. 使用 `VirtioCryptoCreateSessionInput` 结构解析 `resp_slice`，返回 `session_id`。

2.5.2.2.3 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符，用于指定加密算法（如 `VIRTIO_CRYPTO_AES_CBC`）
- `encrypt_or_decrypt: u32`
  - 操作类型，可为 `Aead::ENCRYPT` 或 `Aead::DECRYPT`
- `key: &[u8]`
  - 密钥，用于初始化加密/解密会话
- `tag_len: u32`
  - 标签信息长度
- `aad_len: u32`
  - 附加认证信息长度

2.5.2.2.4 返回值：

- `session_id: u64`
  - 创建的会话 ID

2.5.2.3 `send_destroy_session_request`

向设备发送销毁一个加密/解密会话的异步请求。

2.5.2.3.1 定义：

```
pub fn send_destroy_session_request(
    device: &CryptoDevice,
    session_id: u64
) → (
    queue_index: u32,
    token: u16
)
```

2.5.2.3.2 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `session_id: u64`
  - 要销毁的会话 ID

2.5.2.3.3 返回值：

- `queue_index: u32`
  - 用于传输该请求的队列编号，可为 `CryptoDevice::CONTROLQ` 或 `CryptoDevice::DATAQ`
- `token: u16`
  - 请求的令牌，用于标识和检索指定的请求

2.5.2.4 `destroy_session`

销毁一个加密/解密会话。该方法是多个子方法的打包，实现阻塞式销毁会话操作。

2.5.2.4.1 定义：

```
pub fn destroy_session(
    device: &CryptoDevice,
    session_id: u64
)
```

2.5.2.4.2 方法逻辑：

1. 调用 `Aead::send_destroy_session_request` 方法发送销毁会话请求，获取请求对应 `queue_index` 和 `token`。
2. 调用 `device::get_resp_slice_from` 方法，通过 `queue_index` 和 `token` 获取设备响应 `resp_slice`。
3. 使用 `VirtioCryptoDestroySessionInput` 结构解析 `resp_slice`，确认返回状态。

2.5.2.4.3 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `session_id: u64`
  - 要销毁的会话 ID

2.5.2.4.4 返回值：

- 无

2.5.2.5 `send_encrypt_or_decrypt_request`

向设备发送在某个会话对数据执行加密或解密操作的异步请求。

2.5.2.5.1 定义：

```
pub fn send_encrypt_or_decrypt_request(
    device: &CryptoDevice,
    algo: u32,
    session_id: u64,
    encrypt_or_decrypt: u32,
    iv: &[u8],
    aad: &[u8],
    src_data: &[u8],
    dst_data_len: u32,
    tag: &[u8],
) → (
    queue_size: u32,
    token: u16
)
```

2.5.2.5.2 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符
- `session_id: u64`
  - 当前加密/解密会话 ID
- `encrypt_or_decrypt: u32`
  - 操作类型，可为 `Cipher::ENCRYPT` 或 `Cipher::DECRYPT`

- iv: &[u8]
  - 初始化向量（IV）
- aad: &[u8]
  - 补充认证信息
- src_data: &[u8]
  - 原始数据
- dst_data_len: u32
  - 输出数据的长度
- tag: &[u8]
  - 标签信息

2.5.2.5.3 返回值：

- queue_index: u32
  - 用于传输该请求的队列编号，可为 CryptoDevice::CONTROLQ 或 CryptoDevice::DATAQ
- token: u16
  - 请求的令牌，用于标识和检索指定的请求

2.5.2.6 encrypt_or_decrypt

在某个会话对数据执行加密或解密操作。该方法是多个子方法的打包，实现阻塞式对数据加密或解密。

2.5.2.6.1 定义：

```rust
pub fn encrypt_or_decrypt(
    device: &CryptoDevice,
    algo: u32,
    session_id: u64,
    encrypt_or_decrypt: u32,
    iv: &[u8],
    aad: &[u8],
    src_data: &[u8],
    dst_data_len: u32,
    tag: &[u8]
) -> (
    dst_data: Vec<u8>
)
```

2.5.2.6.2 参数：

- device: &CryptoDevice
  - 使用的加密设备
- algo: u32
  - 算法标识符
- session_id: u64
  - 当前加密/解密会话 ID
- encrypt_or_decrypt: u32
  - 操作类型，可为 Cipher::ENCRYPT 或 Cipher::DECRYPT
- iv: &[u8]
  - 初始化向量（IV）
- aad: &[u8]
  - 补充认证信息
- src_data: &[u8]

  - 原始数据
- `dst_data_len: u32`
  - 输出数据的长度
- `tag: &[u8]`
  - 标签信息

2.5.2.6.3 返回值：

- `dst_data: Vec<u8>`
  - 加密或解密后的数据

2.5.2.7 `encrypt`

创建会话，执行加密操作，然后销毁会话。该方法是对多个子方法的打包，实现阻塞式的伪无状态加密操作。

2.5.2.7.1 定义：

```rust
pub fn encrypt(
    device: &CryptoDevice,
    algo: u32,
    key: &[u8],
    iv: &[u8],
    aad: &[u8],
    src_data: &[u8],
    dst_data_len: u32,
    tag: &[u8],
) -> (
    dst_data: Vec<u8>
)
```

2.5.2.7.2 方法逻辑：

1. 调用 `create_session` 创建加密会话。
2. 调用 `encrypt_or_decrypt` 执行加密操作。
3. 调用 `destroy_session` 销毁会话。
4. 返回加密后的数据。

2.5.2.7.3 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符
- `session_id: u64`
  - 当前加密/解密会话 ID
- `encrypt_or_decrypt: u32`
  - 操作类型，可为 `Cipher::ENCRYPT` 或 `Cipher::DECRYPT`
- `iv: &[u8]`
  - 初始化向量（IV）
- `aad: &[u8]`
  - 补充认证信息
- `src_data: &[u8]`
  - 原始数据
- `dst_data_len: u32`

  - 输出数据的长度
- `tag: &[u8]`
  - 标签信息

2.5.2.7.4 返回值：

- `dst_data: Vec<u8>`：加密后的数据。

2.5.2.7.5 注意事项：

- 每次调用该方法进行加密时都需要执行创建/销毁会话操作，多次重复调用可能导致性能损耗。如果对加密性能有更高的要求，可以通过调用 `create_session`、`encrypt_or_decrypt`、`destroy_session` 手动管理会话并加密数据。

2.5.2.8 `decrypt`

创建会话，执行解密操作，然后销毁会话。该方法是对多个子方法的打包，实现阻塞式的伪无状态解密操作。

定义：

```rust
pub fn decrypt(
    device: &CryptoDevice,
    algo: u32,
    key: &[u8],
    iv: &[u8],
    aad: &[u8],
    src_data: &[u8],
    dst_data_len: u32,
    tag: &[u8],
) -> (
    dst_data: Vec<u8>
)
```

2.5.2.8.1 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符
- `session_id: u64`
  - 当前加密/解密会话 ID
- `encrypt_or_decrypt: u32`
  - 操作类型，可为 `Cipher::ENCRYPT` 或 `Cipher::DECRYPT`
- `iv: &[u8]`
  - 初始化向量（IV）
- `aad: &[u8]`
  - 补充认证信息
- `src_data: &[u8]`
  - 原始数据
- `dst_data_len: u32`
  - 输出数据的长度
- `tag: &[u8]`
  - 标签信息

返回值：

- `decrypted_data: Vec<u8>`
  - 解密后的数据。

逻辑：
1. 调用 `create_session` 创建解密会话。
2. 调用 `encrypt_or_decrypt` 执行解密操作。
3. 调用 `destroy_session` 销毁会话。
4. 返回解密后的数据。

2.5.2.8.2 注意事项：
- 每次调用该方法进行解密时都需要执行创建/销毁会话操作，多次重复调用可能导致性能损耗。如果对解密性能有更高的要求，可以通过调用 `create_session`、`encrypt_or_decrypt`、`destroy_session` 手动管理会话并解密数据。

2.6 Akcipher
`Akcipher` 是一个抽象包，提供对加密/解密服务的封装。它支持创建加密会话、销毁会话，以及对数据进行加密、解密、签名和验证操作。

`Akcipher` 不包含任何字段，仅用于提供静态方法。

2.6.1 常量
- `Akcipher::PUBLIC: u32`
  - 值：`1`
  - 表示使用公钥
- `Akcipher::PRIVATE: u32`
  - 值：`2`
  - 表示使用私钥
- `Akcipher::ENCRYPT: u32`
  - 值：`1`
  - 表示加密操作
- `Akcipher::DECRYPT: u32`
  - 值：`2`
  - 表示解密操作
- `Akcipher::SIGN: u32`
  - 值：`3`
  - 表示签名操作
- `Akcipher::VERIFY: u32`
  - 值：`4`
  - 表示验证操作

2.6.2 方法

2.6.2.1 `send_create_session_rsa_request`
向设备发送创建一个 RSA 会话的异步请求。

2.6.2.1.1 定义：

```
pub fn send_create_session_rsa_request(
  device: &CryptoDevice,
  padding_algo: u32,
  hash_algo: u32,
  public_or_private: u32,
  key: &[u8],
) → (
  queue_index: u32,
  token: u16
)
```

2.6.2.1.2 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `padding_algo: u32`
  - 使用的填充算法，可以为 `VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_RAW_PADDING` 或 `VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_PKCS1_PADDING`
- `hash_algo`
  - 进行填充时使用的哈希算法，可以为 `VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_NO_HASH`、`VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_MD2` 等；当 `padding_algo` 为 `VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_RAW_PADDING` 时该参数将被设备忽略
- `algo: u32`
  - 算法标识符，用于指定加密算法（如 `VIRTIO_CRYPTO_AES_CBC`）
- `public_or_private: u32`
  - 使用公钥或私钥，可为 `Akcipher::PUBLIC`、`Akcipher::PRIVATE`
- `key: &[u8]`
  - 用于初始化会话

2.6.2.1.3 返回值：

- `queue_index: u32` 用于传输该请求的队列编号，可为 `CryptoDevice::CONTROLQ` 或 `CryptoDevice::DATAQ`
- `token: u16` 请求的令牌，用于标识和检索指定的请求

2.6.2.2 `create_session_rsa`

创建一个 RSA 会话。该方法是对多个子方法的打包，实现阻塞式创建 RSA 会话操作。

2.6.2.2.1 定义：

```
pub fn create_session_rsa(
  device: &CryptoDevice,
  padding_algo: u32,
  hash_algo: u32,
  public_or_private: u32,
  key: &[u8],
) → (
  session_id: u64
)
```

2.6.2.2.2 方法逻辑：

1. 调用 `Akcipher::send_create_session_rsa_request` 方法发送创建 RSA 会话请求，获取请求对应 `queue_index` 和 `token`。
2. 调用 `device::get_resp_slice_from` 方法，通过 `queue_index` 和 `token` 获取设备响应 `resp_slice`。
3. 使用 `VirtioCryptoCreateSessionInput` 结构解析 `resp_slice`，返回 `session_id`。

2.6.2.2.3 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `padding_algo: u32`
  - 使用的填充算法，可以为 `VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_RAW_PADDING` 或 `VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_PKCS1_PADDING`
- `hash_algo`
  - 进行填充时使用的哈希算法，可以为 `VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_NO_HASH`、`VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_MD2` 等；当 `padding_algo` 为 `VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_RAW_PADDING` 时该参数将被设备忽略
- `public_or_private: u32`
  - 操作类型，可为 `Akcipher::PUBLIC` 或 `Akcipher::PRIVATE`
- `key: &[u8]`
  - 密钥，用于初始化加密/解密会话

2.6.2.2.4 返回值：

- `session_id: u64`：创建的会话 ID

2.6.2.3 `send_create_session_ecdsa_request`

向设备发送创建一个 ECDSA 会话的异步请求。

2.6.2.3.1 定义：

```
pub fn send_create_session_ecdsa_request(
  device: &CryptoDevice,
  curve_id: u32,
  public_or_private: u32,
  key: &[u8],
) -> (
  queue_size: u32,
  token: u16
)
```

2.6.2.3.2 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `curve_id: u32`
  - 使用的加密曲线，可为 `VirtioCryptoEcdsaSessionPara::VIRTIO_CRYPTO_CURVE_NIST_P192`、`VirtioCryptoEcdsaSessionPara::VIRTIO_CRYPTO_CURVE_NIST_P224` 等
- `public_or_private: u32`
  - 操作类型，可为 `Akcipher::PUBLIC`、`Akcipher::PRIVATE`

- `key: &[u8]`
  - 密钥，用于初始化加密/解密会话

2.6.2.3.3 返回值：

- `queue_index: u32` 用于传输该请求的队列编号，可为 `CryptoDevice::CONTROLQ` 或 `CryptoDevice::DATAQ`
- `token: u16` 请求的令牌，用于标识和检索指定的请求

2.6.2.4 `create_session_ecdsa`

创建一个 ECDSA 会话。该方法是对多个子方法的打包，实现阻塞式创建 ECDSA 会话操作。

2.6.2.4.1 定义：

```
pub fn create_session_ecdsa(
  device: &CryptoDevice,
  curve_id: u32,
  public_or_private: u32,
  key: &[u8],
) → (
  session_id: u64
)
```

2.6.2.4.2 方法逻辑：

1. 调用 `Akpher::send_create_session_ecdsa_request` 方法发送创建 ECDSA 会话请求，获取请求对应 `queue_index` 和 `token`。
2. 调用 `device::get_resp_slice_from` 方法，通过 `queue_index` 和 `token` 获取设备响应 `resp_slice`。
3. 使用 `VirtioCryptoCreateSessionInput` 结构解析 `resp_slice`，返回 `session_id`。

2.6.2.4.3 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `curve_id: u32`
  - 使用的加密曲线，可为 `VirtioCryptoEcdsaSessionPara::VIRTIO_CRYPTO_CURVE_NIST_P192`、`VirtioCryptoEcdsaSessionPara::VIRTIO_CRYPTO_CURVE_NIST_P224` 等
- `public_or_private: u32`
  - 使用公钥或私钥，可为 `Akcipher::PUBLIC`、`Akcipher::PRIVATE`
- `key: &[u8]`
  - 用于初始化会话

2.6.2.4.4 返回值：

- `session_id: u64`：创建的会话 ID

2.6.2.5 `send_destroy_session_request`

向设备发送一个销毁会话的异步请求。

2.6.2.5.1 定义：

```
pub fn send_destroy_session_request(
  device: &CryptoDevice,
  session_id: u64
) → (
  queue_index: u32,
  token: u16
)
```

2.6.2.5.2 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `session_id: u64`
  - 要销毁的会话 ID

2.6.2.5.3 返回值：

- `queue_index: u32` 用于传输该请求的队列编号，可为 `CryptoDevice::CONTROLQ` 或 `CryptoDevice::DATAQ`
- `token: u16` 请求的令牌，用于标识和检索指定的请求

2.6.2.6 `destroy_session`

销毁一个 RSA/ECDSA 会话。该方法是多个子方法的打包，实现阻塞式销毁会话操作。

2.6.2.6.1 定义：

```
pub fn destroy_session(device: &CryptoDevice, session_id: u64)
```

2.6.2.6.2 参数：

- `device: &CryptoDevice` 使用的加密设备
- `session_id: u64` 要销毁的会话 ID

2.6.2.6.3 返回值：

- 无

2.6.2.7 `send_encrypt_or_decrypt_or_sign_or_verify_request`

执行异步加密、解密、签名、验证操作。

2.6.2.7.1 定义：

```
pub fn send_encrypt_or_decrypt_or_sign_or_verify_request(
  device: &CryptoDevice,
  session_id: u64,
  algo: u32,
  op: u32,
  src_data: &[u8],
  dst_data_len: u32,
) → (
  queue_index: u32,
  token: u16,
)
```

2.6.2.7.2 参数：

- device: &CryptoDevice
  - 使用的加密设备
- algo: u32
  - 算法标识符
- session_id: u64
  - 当前加密/解密会话 ID
- op: u32
  - 操作类型，可为 Akcipher::ENCRYPT 或 Akcipher::DECRYPT 或 Akcipher::SIGN 或 Akcipher::VERIFY
- src_data: &Vec<u8>
  - 原始数据
- dst_data_len: u32
  - 输出数据的长度

2.6.2.7.3 返回值：

- queue_index: u32
  - 用于传输该请求的队列编号，可为 CryptoDevice::CONTROLQ 或 CryptoDevice::DATAQ
- token: u16
  - 请求的令牌，用于标识和检索指定的请求

2.6.2.8 encrypt_or_decrypt_or_sign_or_verify

对数据执行加密或解密操作。该方法是对多个子方法的调用，实现阻塞式的数据加解密操作。

2.6.2.8.1 定义：

```
pub fn encrypt_or_decrypt_or_sign_or_verify(
  device: &CryptoDevice,
  session_id: u64,
  algo: u32,
  op: u32,
  src_data: &[u8],
  dst_data_len: u32,
) → (
  dst_data: Vec<u8>
)
```

2.6.2.8.2 参数：

- device: &CryptoDevice
  - 使用的加密设备
- algo: u32
  - 算法标识符
- session_id: u64
  - 当前加密/解密会话 ID
- op: u32
  - 操作类型，可为 Akcipher::ENCRYPT 或 Akcipher::DECRYPT 或 Akcipher::SIGN 或 Akcipher::VERIFY
- src_data: &Vec<u8>
  - 原始数据
- dst_data_len: u32
  - 输出数据的长度

2.6.2.8.3 返回值：

- `dst_data: Vec<u8>`
  - 加密或解密后的数据

2.6.2.8.4 方法逻辑：

1. 调用 `Akcipher::send_encrypt_or_decrypt_or_sign_or_verify_request` 方法发送请求，获取请求对应 `queue_index` 和 `token`。
2. 调用 `device::get_resp_slice_from` 方法，通过 `queue_index` 和 `token` 获取设备响应 `resp_slice` 和写入长度 `write_len`。
3. 解析 `resp_slice`，返回输出数据。

2.6.2.9 akcipher

创建会话，执行加密/解密/签名/验证操作，然后销毁会话。该方法是对多个子方法的打包，实现阻塞式的伪无状态操作。

2.6.2.9.1 方法逻辑：

1. 调用 `Self::create_session` 创建会话。
2. 调用 `Self::encrypt_or_decrypt_or_sign_or_verify` 执行操作。
3. 调用 `Self::destroy_session` 销毁会话。
4. 返回结果数据。

2.6.2.9.2 定义：

```rust
pub fn akcipher(
  device: &CryptoDevice,
  padding_algo: u32,
  hash_algo: u32,
  public_or_private: u32,
  akcipher_key: &[u8],
  algo: u32,
  op: u32,
  src_data: &[u8],
) -> (
  dst_data: Vec<u8>
)
```

2.6.2.9.3 参数：

- `device: &CryptoDevice`
  - 使用的加密设备
- `algo: u32`
  - 算法标识符
- `session_id: u64`
  - 当前加密/解密会话 ID
- `op: u32`
  - 操作类型，可为 `Akcipher::ENCRYPT` 或 `Akcipher::DECRYPT` 或 `Akcipher::SIGN` 或 `Akcipher::VERIFY`
- `src_data: &Vec<u8>`
  - 原始数据
- `dst_data_len: u32`
  - 输出数据的长度

2.6.2.9.4 返回值：

- `dst_data: Vec<u8>`
  - 加密或解密后的数据

2.6.3 用法示例及功能测试

2.6.3.1 加密-解密测试

```rust
// 密钥：生成方法详见附录
let pub_key: Vec<u8> = vec![ ... ]
let private_key: Vec<u8> = vec![ ... ]

early_println!("[Test] Original data for AKCIPHER: {:?}", data2);
let akcipher_encrypt_result = Akcipher::akcipher(
    &device,
    VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_PKCS1_PADDING,
    VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_SHA256,
    Akcipher::PUBLIC,
    &pub_key,
    VIRTIO_CRYPTO_AKCIPHER_RSA,
    Akcipher::ENCRYPT,
    &data2,
);
early_println!(
    "[Test] Encrypted data for AKCIPHER: {:?}",
    akcipher_encrypt_result
);

let akcipher_decrypt_result = Akcipher::akcipher(
    &device,
    VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_PKCS1_PADDING,
    VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_SHA256,
    Akcipher::PRIVATE,
    &private_key,
    VIRTIO_CRYPTO_AKCIPHER_RSA,
    Akcipher::DECRYPT,
    &akcipher_encrypt_result,
);
early_println!(
    "[Test] Decrypted data for AKCIPHER: {:?}",
    akcipher_decrypt_result
);

assert_eq!(
    data2, akcipher_decrypt_result,
    "[Test] The initial data and decrypted data of AKCIPHER are inconsistent"
);
early_println!("[Test] AKCIPHER encrypt-decrypt test pass!");
```

```
[Test] Original data for AKCIPHER: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
request in controlq with token: 2 has finished
request in controlq with token: 4 has finished
Status: VirtioCryptoCreateSessionInput { session_id: 0, status: 0, padding: 0 }
request in dataq with token: 0 has finished
dst_data_len_buf: 257
Data: [65, 3D, CD, BA, A8, 94, 5A, AE, 87, 5F, C3, 5B, D4, 41, F1, 10, 5, 49, EE, 81, 12, D0, 3, FC, E9,
 2, 93, 9C, FD, FC, B2, 6, 1F, 3B, AF, 6B, DE, F, 99, 58, 84, 64, 19, B4, 1D, B1, 3, 10, 4D, 3D, C4, 38,
 F6, 57, 1B, 1E, D0, 64, 49, BF, 1C, D7, 8B, 9B, DD, C0, 79, 69, 7A, 99, 8C, 80, 4D, 43, 73, CA, 30, 1C,
 2E, 11, 7D, E0, 1B, BD, 8F, CD, 12, 6E, FF, F1, EE, 5E, 62, 2D, DE, E6, 22, F1, BB, 62, 11, 7C, 75, E,
5D, 99, 53, 41, D7, DD, 4C, 7B, 16, 77, AD, C5, 1E, 66, EA, 53, 67, 4B, 48, 6A, 11, 1, DD, BD, 1, E1, 5E
, 32, D9, 26, 20, 79, B6, C0, 77, FF, E9, BC, FF, E6, 73, A, F2, 7E, BC, 35, 7C, 3B, CD, 38, 24, 1C, 64,
 A3, 5E, 5E, EA, D, 8E, 2D, 62, 38, 6E, B, F, 8C, 74, 26, B7, C3, 31, B5, 91, 85, AE, D5, FE, 9D, D7, 5E
, B9, A9, 62, A9, E3, 68, 19, 8A, BD, 81, 1E, 6, F5, 43, 24, 8D, F7, 7D, 2A, CE, 68, 36, 74, D0, 96, CA,
 26, B8, AC, 63, AD, A6, 61, AB, 12, 44, 3F, 2, 29, 27, 30, 2, 28, D9, 2F, BE, A3, 7B, E4, A, 45, 55, 99
, 7F, C5, CA, 9A, 3E, 21, 6, 93, 8A, 3C, 73, 95, 79, AD, 5, 1F, 5F, 16, 8E]
request in controlq with token: 4 has finished
Status: VirtioCryptoDestroySessionInput { status: 0 }
[Test] Encrypted data for AKCIPHER: [101, 61, 205, 186, 168, 148, 90, 174, 135, 95, 195, 91, 212, 65, 24
1, 16, 5, 73, 238, 129, 18, 208, 3, 252, 233, 2, 147, 156, 253, 252, 178, 6, 31, 59, 175, 107, 222, 15,
153, 88, 132, 100, 25, 180, 29, 177, 3, 16, 77, 61, 196, 56, 246, 87, 27, 30, 208, 100, 73, 191, 28, 215
, 139, 155, 221, 192, 121, 105, 122, 153, 140, 128, 77, 67, 115, 202, 48, 28, 46, 17, 125, 224, 27, 189,
 143, 205, 18, 110, 255, 241, 238, 94, 98, 45, 222, 230, 34, 241, 187, 98, 17, 124, 117, 14, 93, 153, 83
, 65, 215, 221, 76, 123, 22, 119, 173, 197, 30, 102, 234, 83, 103, 75, 72, 106, 17, 1, 221, 189, 1, 225,
 94, 50, 217, 38, 32, 121, 182, 192, 119, 255, 233, 188, 255, 230, 115, 10, 242, 126, 188, 53, 124, 59,
205, 56, 36, 28, 100, 163, 94, 94, 234, 13, 142, 45, 98, 56, 110, 11, 15, 140, 116, 38, 183, 195, 49, 18
1, 145, 133, 174, 213, 254, 157, 215, 94, 185, 169, 98, 169, 227, 104, 25, 138, 189, 129, 30, 6, 245, 67
, 36, 141, 247, 125, 42, 206, 104, 54, 116, 208, 150, 202, 38, 184, 172, 99, 173, 166, 97, 171, 18, 68,
63, 2, 41, 39, 48, 2, 40, 217, 47, 190, 163, 123, 228, 10, 69, 85, 153, 127, 197, 202, 154, 62, 33, 6, 1
47, 138, 60, 115, 149, 121, 173, 5, 31, 95, 22, 142]
```

```
request in controlq with token: 4 has finished
Status: VirtioCryptoCreateSessionInput { session_id: 0, status: 0, padding: 0 }
request in dataq with token: 0 has finished
dst_data_len_buf: 14
Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D]
request in controlq with token: 4 has finished
Status: VirtioCryptoDestroySessionInput { status: 0 }
[Test] Decrypted data for AKCIPHER: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[Test] AKCIPHER encrypt-decrypt test pass!
```

将数组 `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]` 加密再解密后得到的结果与原始数组相同，测试通过。

2.6.3.2 签名-验证测试

```rust
let data3 = vec![
    49, 95, 91, 219, 118, 208, 120, 196, 59, 138, 192, 6, 78, 74, 1, 100, 97,
43, 31, 206,
    119, 200, 105, 52, 91, 252, 148, 199, 88, 148, 237, 211,
];

early_println!("[Test] Original data for AKCIPHER: {:?}", data3);
let akcipher_sign_result = Akcipher::akcipher(
    &device,
    VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_PKCS1_PADDING,
    VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_SHA256,
    Akcipher::PRIVATE,
    &private_key,
    VIRTIO_CRYPTO_AKCIPHER_RSA,
    Akcipher::SIGN,
    &data3,
);
early_println!("[Test] Signature for AKCIPHER: {:?}", akcipher_sign_result);

let verify_input = [akcipher_sign_result.as_slice(), data3.as_slice()].concat();
early_println!(
    "[Test] Sign_result length: {:?}, Data_result length: {:?}",
    akcipher_sign_result.len(),
    data3.len()
);
let akcipher_verify_result = Akcipher::akcipher(
    &device,
    VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_PKCS1_PADDING,
    VirtioCryptoRsaSessionPara::VIRTIO_CRYPTO_RSA_SHA256,
    Akcipher::PUBLIC,
    &pub_key,
    VIRTIO_CRYPTO_AKCIPHER_RSA,
    Akcipher::VERIFY,
    &verify_input,
);
early_println!(
    "[Test] Verification for AKCIPHER: {:?}",
    akcipher_verify_result
);
assert!(
    akcipher_verify_result[0] == 0,
    "[Test] AKCIPHER sign-verify test failed!"
);
early_println!("[Test] AKCIPHER sign-verify test pass!");
```

```
[Test] Original data for AKCIPHER: [49, 95, 91, 219, 118, 208, 120, 196, 59, 138, 192, 6, 78, 74, 1, 100
, 97, 43, 31, 206, 119, 200, 105, 52, 91, 252, 148, 199, 88, 148, 237, 211]
request in controlq with token: 4 has finished
Status: VirtioCryptoCreateSessionInput { session_id: 0, status: 0, padding: 0 }
request in dataq with token: 0 has finished
dst_data_len_buf: 257
Data: [5, B4, 9F, 5D, 8F, 24, 8A, 30, 85, BE, A8, 60, F1, 93, 44, E7, ED, 69, E2, AE, E4, E0, 41, 56, 12
, 2E, 73, 81, D1, CB, 1C, 18, 5F, B0, 8F, B, 8E, 98, 8B, 72, EB, 28, 10, CC, 9F, 5D, F9, 4D, 4F, 1E, C0,
 E, 6, 77, 55, B, 96, 1B, 24, 9, 25, D9, DC, 85, F1, 1A, CE, 22, B1, FA, B5, 92, 6C, D, BD, 78, AE, AC,
D3, 41, 79, EF, 6D, 41, 19, 4B, ED, CC, 10, 93, 53, 35, 85, B2, D6, 15, B7, 7A, 94, 19, F4, 64, 98, 96,
18, 1E, 84, 2F, 48, BB, 9, 13, 4C, A4, A6, 3E, A8, 8C, F1, 18, B1, D2, BD, 11, 31, 12, CA, E6, 1D, 47, 6
C, 60, 36, F6, B1, 97, B1, 71, B8, 20, 25, 83, BB, 61, 71, B6, 2, C4, AA, 27, 0, BF, 65, 3, 2C, 9D, 52,
4A, 0, EF, 8B, F5, 3C, 7C, 45, D8, 39, 29, 48, 22, E3, 34, 6B, 9F, D9, 83, F5, 1E, 19, 9F, 98, 70, 3C, 6
1, 20, 6D, 6B, A3, AB, 1D, AE, 39, C, 5A, DD, 8E, 80, 76, F8, 7E, 0, 1E, 70, 1E, 3, 55, 8F, B5, 68, B7,
38, 2A, 87, 4F, AD, A6, EA, C6, 76, 1F, D9, 97, 69, 3, 7E, 42, CD, 99, E9, 74, B7, 6E, 11, 60, 30, 87, 6
8, 54, A3, B2, 25, 4B, 87, 94, B2, 6C, F1, 4B, 6A, 3E, 6E, E, 3C, 54, FE, DF]
request in controlq with token: 4 has finished
Status: VirtioCryptoDestroySessionInput { status: 0 }
[Test] Signature for AKCIPHER: [5, 180, 159, 93, 143, 36, 138, 48, 133, 190, 168, 96, 241, 147, 68, 231,
 237, 105, 226, 174, 228, 224, 65, 86, 18, 46, 115, 129, 209, 203, 28, 24, 95, 176, 143, 11, 142, 152, 1
39, 114, 235, 40, 16, 204, 159, 93, 249, 77, 79, 30, 192, 14, 6, 119, 85, 11, 150, 27, 36, 9, 37, 217, 2
20, 133, 241, 26, 206, 34, 177, 250, 181, 146, 108, 13, 189, 120, 174, 172, 211, 65, 121, 239, 109, 65,
25, 75, 237, 204, 16, 147, 83, 53, 133, 178, 214, 21, 183, 122, 148, 25, 244, 100, 152, 150, 24, 30, 132
, 47, 72, 187, 9, 19, 76, 164, 166, 62, 168, 140, 241, 24, 177, 210, 189, 17, 49, 18, 202, 230, 29, 71,
108, 96, 54, 246, 177, 151, 177, 113, 184, 32, 37, 131, 187, 97, 113, 182, 2, 196, 170, 39, 0, 191, 101,
 3, 44, 157, 82, 74, 0, 239, 139, 245, 60, 124, 69, 216, 57, 41, 72, 34, 227, 52, 107, 159, 217, 131, 24
5, 30, 25, 159, 152, 112, 60, 97, 32, 109, 107, 163, 171, 29, 174, 57, 12, 90, 221, 142, 128, 118, 248,
126, 0, 30, 112, 30, 3, 85, 143, 181, 104, 183, 56, 42, 135, 79, 173, 166, 234, 198, 118, 31, 217, 151,
105, 3, 126, 66, 205, 153, 233, 116, 183, 110, 17, 96, 48, 135, 104, 84, 163, 178, 37, 75, 135, 148, 178
, 108, 241, 75, 106, 62, 110, 14, 60, 84, 254, 223]
```

```
[Test] Sign_result length: 256, Data_result length: 32
request in controlq with token: 4 has finished
Status: VirtioCryptoCreateSessionInput { session_id: 0, status: 0, padding: 0 }
qemu-system-x86_64: Signature length is greater than key size: 256
request in dataq with token: 0 has finished
dst_data_len_buf: 257
Data: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
request in controlq with token: 4 has finished
Status: VirtioCryptoDestroySessionInput { status: 0 }
[Test] Verification for AKCIPHER: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[Test] AKCIPHER sign-verify test pass!
```

在验证签名的过程中，设备返回的 signal 为 0，说明签名验证成功，测试通过。

2.7 ChainAlg (Hash & Mac)

`ChainAlg` 包是基于 Cipher 和 Hash(Mac) 两种加密算法的链式加密算法。它支持创建加密会话、销毁会话，以及对数据进行加密操作。

2.7.1 字段

```rust
// 当前加密/解密会话 ID
pub session_id: u64,
// 加密（ChainAlg::ENCRYPT）或解密（ChainAlg::DECRYPT）
pub service: u32,
// 算法顺序：VIRTIO_CRYPTO_SYM_ALG_CHAIN_ORDER_HASH_THEN_CIPHER
// 或 VIRTIO_CRYPTO_SYM_ALG_CHAIN_ORDER_CIPHER_THEN_HASH
pub alg_chain_order: u32,
// 哈希算法：VIRTIO_CRYPTO_SYM_HASH_MODE_PLAIN（对应 Hash）或
VIRTIO_CRYPTO_SYM_HASH_MODE_AUTH（对应 Mac）
pub hash_mode: u32,
// HASH algorithm for VIRTIO_CRYPTO_SYM_HASH_MODE_PLAIN, MAC algorithm for
VIRTIO_CRYPTO_SYM_HASH_MODE_AUTH
pub hash_algo: u32,
// Cipher 算法（详见 Cipher 部分）
pub cipher_algo: u32,
```

2.7.2 常量

- `VIRTIO_CRYPTO_SYM_ALG_CHAIN_ORDER_HASH_THEN_CIPHER: u32`
  - 值：1
  - 表示加密时先使用哈希算法再使用 Cipher 算法
- `VIRTIO_CRYPTO_SYM_ALG_CHAIN_ORDER_CIPHER_THEN_HASH: u32`
  - 值：2
  - 表示加密时先使用 Cipher 算法再使用哈希算法
- `VIRTIO_CRYPTO_SYM_HASH_MODE_PLAIN: u32`
  - 值：1
  - 表示使用 Hash 作为哈希算法加密
- `VIRTIO_CRYPTO_SYM_HASH_MODE_AUTH: u32`
  - 值：2
  - 表示使用 Mac 作为哈希算法加密
- `VIRTIO_CRYPTO_SYM_HASH_MODE_NESTED: u32`
  - 值：3
  - 表示使用 Nested Hash 作为哈希算法加密

2.7.3 方法

2.7.3.1 `create_session`

创建一个加密会话。该方法是对多个子方法的打包，实现阻塞式创建会话操作。

2.7.3.1.1 定义：

```rust
pub fn create_session(
  &self,
  device: &CryptoDevice,
  cipher_key: &[u8],
  auth_key: &[u8]
) → (
  session_id: u64
)
```

2.7.3.1.2 方法逻辑：

详见 Cipher 部分。

2.7.3.1.3 参数：

- `device: &CryptoDevice` 使用的加密设备
- `cipher_key: &[u8]` 密钥
- `auth_key: &[u8]` Authentication key

2.7.3.1.4 返回值：

- `session_id: u64` 创建的会话的 ID

2.7.3.2 `destroy_session`

销毁一个加密会话。该方法是多个子方法的打包，实现阻塞式销毁会话操作。

2.7.3.2.1 定义：

```
pub fn destroy_session(&self, device: &CryptoDevice)
```

2.7.3.2.2 方法逻辑：

详见 Cipher 部分。

2.7.3.2.3 参数：

- `device: &CryptoDevice` 使用的加密设备
- 需要销毁的会话的 ID `session_id` 为 `self.session_id`

2.7.3.2.4 返回值：

无

2.7.3.3 `hash_or_mac`

在某个会话对数据通过 chaining algorithm 进行加密操作。

2.7.3.3.1 定义：

```
pub fn hash_or_mac(
  &self,
  device: &CryptoDevice,
  iv: &Vec<u8>,
  src_data: &Vec<u8>,
  dst_data_len: u32,
) -> (
  dst_data: Vec<u8>
)
```

2.7.3.3.2 参数：

- `device: &CryptoDevice` 使用的加密设备
- `iv: &Vec<u8>` 初始化向量（IV）
- `src_data: &Vec<u8>` 原始数据
- `dst_data_len: u32` 加密后数据的长度

2.7.3.3.3 返回值：

- `dst_data: Vec<u8>` 加密后的数据

2.7.3.4 `chaining_algorithms`

在某个会话对数据通过 chaining algorithm 进行加密操作。该方法是多个子方法的打包，以便将操作作为一个整体进行调用。

2.7.3.4.1 定义：

```rust
pub fn chaining_algorithms(
  mut self,
  device: &CryptoDevice,
  cipher_key: &[u8],
  auth_key: &[u8],
  iv: &Vec<u8>,
  src_data: &Vec<u8>,
) -> (
  dst_data: Vec<u8>
)
```

2.7.3.4.2 方法逻辑：

- 创建会话并获取会话 ID
- 调用 `hash_or_mac` 方法进行加密操作
- 返回加密后的数据

2.7.3.4.3 参数：

- `device: &CryptoDevice` 使用的加密设备
- `cipher_key: &[u8]` 密钥
- `auth_key: &[u8]` Authentication key
- `iv: &Vec<u8>` 初始化向量（IV）
- `src_data: &Vec<u8>` 原始数据

2.7.3.4.4 返回值：

- `dst_data: Vec<u8>` 加密后的数据

2.7.4 用法示例

```rust
let cipher_key = [0_u8; 16];
let iv = vec![0x00; 16];
let data1: Vec<u8> = vec![
    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10,
    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10,
];

let mut chain_alg = ChainAlg::new(
    ChainAlg::ENCRYPT,
    VIRTIO_CRYPTO_SYM_ALG_CHAIN_ORDER_CIPHER_THEN_HASH,
    VIRTIO_CRYPTO_SYM_HASH_MODE_PLAIN,
    VIRTIO_CRYPTO_HASH_SHA1,
    VIRTIO_CRYPTO_CIPHER_AES_CBC,
);
let chain_alg_result = chain_alg.chaining_algorithms(
    &device, &cipher_key, &[], &iv, &data1
);
early_println!("Result for chaining algorithm: {:?}", chain_alg_result);
```

3. 异步支持

本驱动支持非阻塞式的异步发送请求、接收响应操作。

3.1 相关方法

3.1.1 `(Service)::send_(operation)_request -> queue_index, token`

- `(Service)` 为 `Cipher`、`Akcipher` 等算法服务
- `(operation)` 为 `create_session`、`encrypt_or_decrypt` 等具体操作
- 向设备发送一个操作相关的异步请求，获取请求被执行的 `queue_index` 和请求的 `token` 信息

3.1.2 `CryptoDevice::refresh_state`

- 通过循环调用 `VirtQueue::pop_used` 弹出所有已完成请求并更新记录状态

3.1.3 `CryptoDevice::is_finished -> bool`

- 通过 `queue_index` 和 `token` 查询某个请求是否已经被设备响应
- 调用该方法时会自动调用 `CryptoDevice::refresh_state` 方法刷新已提交请求的响应状态

3.1.4 `CryptoDevice::get_resp_from -> resp_slice, write_len`

- 对于已经完成的请求，该方法直接返回响应切片和写入长度信息
- 对于尚未完成的请求，`spin_loop` 直到请求完成并返回响应切片和写入长度信息

3.2 用法示例

以 `Cipher::create_session` 为例，异步执行过程分为三步：

1. 通过调用 `Cipher::send_create_session_request` 发送请求
2. 通过调用 `CryptoDevice::is_finished` 查询请求是否完成
   - 如果请求尚未完成，可以考虑先执行其他任务以提高 CPU 利用率
3. 通过调用 `CryptoDevice::get_resp_slice_from` 获取设备响应和写入长度数据
   - 如果调用该方法时请求尚未完成，驱动会 `spin_loop` 直到请求完成再返回数据

```rust
pub fn create_session(
  device: &CryptoDevice,
  algo: u32,
  encrypt_or_decrypt: u32,
  cipher_key: &[u8],
) -> u64 {
    // (b)步骤可以在执行(a)步骤后的任意时刻执行
    // (c)步骤可以在(a)步骤和(b)步骤之间的任何时刻执行

    // (a) 发送 create_session 请求，返回请求所对应的 queue_index 和 token
    let (queue_index, token) =
        Cipher::send_create_session_request(
            device,
            algo,
            encrypt_or_decrypt,
            cipher_key
        );

    // (b) 通过 token 从对应 queue 查询该请求是否已经完成
    let _finished = device.is_finished(queue_index, token);

    // (c) 通过 token 从对应 queue 获取该请求的 resp_slice 和 write_len
    // 如果该请求尚未完成，则 spin_loop 直到请求完成并返回
    let (resp_slice, _write_len) =
        device.get_resp_slice_from(queue_index, token);

    // 解析 resp_slice 为 resp
    let resp: VirtioCryptoCreateSessionInput =
        resp_slice.read_val(0).unwrap();
    resp.session_id
}
```

上面的代码仅仅作为一个方法调用的参考示例。在实际使用中，可以：

- 并发地执行多个(a)步骤，发送多个请求
- 随时通过(b)步骤查看运行状态，并据此做出调度策略
- 在合适的时机调用(c)步骤，返回数据

3.3 实现逻辑

1. 发送请求时为请求和响应动态分配 DMA 流空间
2. 将分配到的 DMA 流空间（切片）注册到已提交请求中
3. 驱动轮询 VirtQueue，弹出并标记已完成的请求
4. 需要返回数据时从已提交请求中弹出已完成的请求
5. 已完成请求弹出后回收并合并对应 DMA 流空间

3.3.1 动态分配 DMA 流空间

根据定义，由于 `Virtio-Crypto` 操作的请求和响应长度不定，传统的通过 `id_allocator` 分配 `id`，并根据 `id` 计算 `offset`、分配 DMA 流空间的方法不可行。

因此，驱动在设备初始化时创建 `request_buffer_allocator` 和 `response_buffer_allocator` 并传入设备中，用于动态分配和回收请求或响应的 DMA 流地址。

3.3.1.1 相关结构与方法

3.3.1.1.1 SliceAllocator

```rust
pub struct SliceAllocator {
  stream_size: usize, // DMA 流的空间大小
  id_allocator: SpinLock<IdAlloc>, // ID 分配器
  space_slices: Vec<SpaceSlice>, // 记录 DMA 流中的空余切片
}

impl SliceAllocator {
    // 创建一个新的 SliceAllocator
    // 传入参数：DMA 流的空间大小，最大 ID 个数
    pub fn new(
      stream_size: usize,
      id_capacity: usize
    ) -> (
        self: Self
    )

    // 分配空间
    // 传入参数：需要分配的空间大小
    // 返回值：Result<分配成功的空间信息, 分配失败的错误信息>
    // 通过 First Match 的方法为请求分配空间
    pub fn allocate(
      &mut self,
      size: usize
    ) -> (
        Result<SliceRecord, &'static str>
    )

    // 回收空间
    // 传入参数：需要回收的空间信息
    // 将连续的空余切片自动合并
    fn deallocate(
      &mut self,
      slice_record: &SliceRecord
    )
```

3.3.1.1.2 SliceRecord 与 SpaceSlice

```rust
pub struct SliceRecord {
  pub id: usize, // 切片 ID
  pub head: usize, // 切片起始地址
  pub tail: usize, // 切片结束地址
}

struct SpaceSlice {
  pub head: usize, // 空闲切片起始地址
  pub tail: usize, // 空闲切片结束地址
}
```

3.3.2 管理已提交请求

由于发送请求和接收响应的步骤异步进行，因此驱动需要记录所有已提交的请求信息，并实时更新状态。

3.3.2.1 相关结构和方法

3.3.2.1.1 SubmittedReqManager

```rust
pub struct SubmittedReqManager {
  pub records: Vec<SubmittedReq>,
}

impl SubmittedReqManager {
    // 未找到请求错误
    pub const NOT_FIND: u32 = 1;
    // 请求未完成错误
    pub const NOT_FINISHED: u32 = 2;

    // 创建一个新的 SubmittedReqManager
    pub fn new() -> (Self)

    // 添加记录
    pub fn add(&mut self, record: SubmittedReq)

    // 标记令牌为 token 的请求已经响应结束
    pub fn finish(&mut self, token: u16, write_len: u32)

    // 查询令牌为 token 的请求是否响应结束
    pub fn is_finished(&self, token: u16) -> bool

    // 弹出令牌为 token 的请求
    // 如果没有找到令牌为 token 的请求，返回 Err(NOT_FIND)
    // 如果请求尚未完成，返回 Err(NOT_FINISHED)
    pub fn pop(
        &mut self,
        token: u16
    ) -> (
        Result<SubmittedReq, u32>
    )
```

3.3.2.1.2 SubmittedReq

```rust
pub struct SubmittedReq {
  pub token: u16, // 请求令牌
  pub req_slice_record: SliceRecord, // 请求切片记录
  pub resp_slice_record: SliceRecord, // 请求响应切片记录
  pub finished: bool, // 记录该请求是否已经被响应
  pub write_len: u32, // 记录设备响应写入长度
}
```

4. 用户态调用

通过扩充 `ioctl` 支持的指令，我们可以通过 `C` 语言支持的 `sys/ioctl.h` 来实现对 `virtio-crypto` 设备方法的调用。目前，由于时间原因，我们只支持对于 Cipher AES_CBC 算法的用户态调用。

4.1 相关参数定义

4.1.1 `IoctlCmd::CIOCCRYPT` 和 `IoctlCmd::CIOCXCRYPT`

```
// kernel/src/fs/utils/ioctl.rs
pub enum IoctlCmd {
   ... // Other IoctlCmds

   /// Test Cipher Encrypt
   CIOCCRYPT = 0x20006401,
   /// Test Cipher Decrypt
   CIOCXCRYPT = 0x20006402,
}
```

4.2 用法示例

由于 Asterinas 中没有对应 C 编译器，我们通过在外部实现 C 代码并且编译后，并将对应可执行文件放入 Asterinas 在 make build 后的 initramfs 中，并在 Asterinas 中执行，我们测试的 C 代码为。这个代码中我们调用 `sys/ioctl.h` 中的 `ioctl` 函数来实现对 `virtio-crypto` 中的方法进行调用，我们通过传入 user_string 对应的虚拟地址来取出我们想要加密的数据类型。

```
#include <sys/ioctl.h>
#include ... // Other includes

#define CIOCCRYPT  0x20006401
#define CIOCXCRYPT 0x20006402

int main() {
    char *null_ptr;
    char *user_string;
    user_string = (char *)malloc(256);
    strcpy(user_string, "Hello World!");
    int fd = open("/dev/tty", O_RDWR);
    if (ioctl(fd, CIOCCRYPT, &user_string) == -1) {
            perror("Encrypt Failed");
            close(fd);
            return EXIT_FAILURE;
        }
    if (ioctl(fd, CIOCXCRYPT, &null_ptr) == -1) {
            perror("Decrypt Failed");
            close(fd);
            return EXIT_FAILURE;
        }
    free(user_string);
    close(fd);
    return EXIT_SUCCESS;
}
```

在 Asterinas 执行后，我们可以看到我们成功从用户态调用了 Kernel 的设备。

```
/usr/bin # ls
busybox  test
/usr/bin # ./test
268440224 "Hello World!"
[Test] Original data for CIPHER: [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 33, 0, 0, 0, 0]
request in controlq with token: 0 has finished
VirtioCryptoCreateSessionInput { session_id: 0, status: 0, padding: 0 }
request in dataq with token: 0 has finished
[Test] Encrypted data for CIPHER: [30, 14, 103, 217, 19, 131, 99, 211, 222, 119, 251, 172, 73, 107, 49, 253]
request in controlq with token: 0 has finished
request in controlq with token: 2 has finished
VirtioCryptoCreateSessionInput { session_id: 0, status: 0, padding: 0 }
request in dataq with token: 0 has finished
[Test] Decrypted data for CIPHER: [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 33, 0, 0, 0, 0]
Decrypted string: Hello World!
/usr/bin #
```

4.3 实现逻辑

4.3.1 注册设备 Component

参考其他驱动设备的方式，我们首先需要在 `kernel/comps` 创建对应的 Component 文件夹，并且更新对应的 Module 路径。对于任意 `virtio-crypto` 设备，我们设计了 `cipher_encrypt` 和 `cipher_decrypt` 两个对应的接口，支持传入需要加密/解密的数据和 AES_CBC 算法需要的 key 信息。

```rust
pub trait AnyCryptoDevice: Send + Sync {
    fn cipher_encrypt(&self, data: &[u8], cipher_key: &[u8]) -> Vec<u8>;
    fn cipher_decrypt(&self, data: &[u8], cipher_key: &[u8]) -> Vec<u8>;
}
```

同时提供了 `register_device` 和 `get_device` 来注册和获取我们对应的设备信息。(Component 类同其他设备的完全一致)

```rust
static COMPONENT: Once<Component> = Once::new();
pub fn register_device(name: String, device: Arc<dyn AnyCryptoDevice>) {
    COMPONENT
        .get()
        .unwrap()
        .crypto_device_table
        .disable_irq()
        .lock()
        .insert(name, device);
}

pub fn get_device(name: &str) -> Option<Arc<dyn AnyCryptoDevice>> {
    let crypto_devs = COMPONENT
        .get()
        .unwrap()
        .crypto_device_table
        .disable_irq()
        .lock();
    crypto_devs.get(name).map(|device| device.clone())
}
```

4.3.2 Device 接口实现

在定义 Trait 后，我们需要在对应的 CryptoDevice 里实现对应的 Trait，并且调用对应的方法。

```rust
impl AnyCryptoDevice for CryptoDevice {
    // 接口对应信息请查看报告前面对应的部分
    fn cipher_encrypt(&self, data: &[u8], cipher_key: &[u8]) -> Vec<u8> {
        let iv = vec![0x00; 16];
        early_println!("[Test] Original data for CIPHER: {:?}", data);
        let cipher_encrypt_result = Cipher::encrypt(
            &self,
            VIRTIO_CRYPTO_CIPHER_AES_CBC,
            &cipher_key,
            &iv,
            &data,
        );
        early_println!(
            "[Test] Encrypted data for CIPHER: {:?}",
            cipher_encrypt_result
        );
        cipher_encrypt_result
    }

    fn cipher_decrypt(&self, data: &[u8], cipher_key: &[u8]) -> Vec<u8> {
        let iv = vec![0x00; 16];
        let cipher_decrypt_result = Cipher::decrypt(
            &self,
            VIRTIO_CRYPTO_CIPHER_AES_CBC,
            &cipher_key,
            &iv,
            &data,
        );
        early_println!(
            "[Test] Decrypted data for CIPHER: {:?}",
            cipher_decrypt_result
        );
        cipher_decrypt_result
    }
}

```

4.3.3 实现接口调用

在 `kernel/src` 创建 `crypto` 文件夹，在其中实现对 Device 的获取和对应方法的调用。在这里，我们对数据进行预处理，使其符合我们 Cipher 算法的输入规范。

```rust
use aster_crypto::get_device;
use ... // Other Import

static ENCRYPTED_DATA: Once<Vec<u8>> = Once::new();
static PADDING: Once<usize> = Once::new();

pub fn encrypt(input: &[u8]) {
    let device = get_device("virtio-crypto").unwrap();
    // AES_CBC 方法要求输入是 16 的整数倍，在输入前要做一次 Padding
    let mut padded_data = input.to_vec();
    let padding = 16 - (padded_data.len() % 16);
    padded_data.extend(vec![0_u8; padding]);
    let cipher_key = [0_u8; 16];
    let encrypted_data = device.cipher_encrypt(&padded_data, &cipher_key);
    ENCRYPTED_DATA.call_once(|| encrypted_data);
    PADDING.call_once(|| padding);
}

pub fn decrypt() {
    let device = get_device("virtio-crypto").unwrap();
    let encrypted_data = ENCRYPTED_DATA.get().unwrap();
    let cipher_key = [0_u8; 16];
    let decrypted_data = device.cipher_decrypt(encrypted_data, &cipher_key);
    // 在解析前要把 Padding 的长度去掉
    let padding = PADDING.get().unwrap();
    let decrypted_data = decrypted_data[..decrypted_data.len() -
padding].to_vec();
    match String::from_utf8(decrypted_data) {
        Ok(string) => {
            early_println!("Decrypted string: {}", string);
        },
        Err(e) => {
            early_println!("Failed to decode bytes: {}", e);
        }
    }
}
```

4.3.4 syscall 方法扩充

在 `kernel/src/syscall/ioctl.rs`，我们可以对 ioctl 所支持的 Ioctlcmd 进行扩充，使其支持我们的设备操作，实现如下：

```rust
use crate::crypto::crypto::{encrypt, decrypt};
pub fn sys_ioctl(fd: FileDesc, cmd: u32, arg: Vaddr, ctx: &Context) ->
Result<SyscallReturn> {
    let res = match ioctl_cmd {
        ... => {}
        IoctlCmd::CIOCCRYPT => {
            // Encrypt
            let max_string_len = 256;
            let input: CString = read_cstring_vec(arg, max_string_len, ctx)?;
            let input_bytes: &[u8] = input.as_bytes();
            encrypt(input_bytes);
            0
        }
        IoctlCmd::CIOCXCRYPT => {
            // Decrypt
            decrypt();
            0
        }
    }
}
```

`read_cstring_vec` 帮助我们可以从 userspace 中通过虚拟地址，拿到我们的输入数据“Hello World!”。(参考 `sys_execve` 方法)

```rust
fn read_cstring_vec(
  read_ptr: Vaddr,
  max_string_len: usize,
  ctx: &Context,
) -> Result<CString> {
    let mut res = CString::default();
    // On Linux, argv pointer and envp pointer can be specified as NULL.
    if read_ptr == 0 {
        return Ok(res);
    }
    let user_space = ctx.user_space();
    let cstring_ptr = user_space.read_val::<usize>(read_ptr)?;
    if cstring_ptr == 0 {
        return_errno_with_message!(Errno::E2BIG, "Cannot find null pointer in
vector");
    }
    let cstring = user_space.read_cstring(cstring_ptr, max_string_len)?;
    early_println!("{:?} {:?}", cstring_ptr, cstring);
    res = cstring;
    Ok(res)
}
```

附录

在 Akcipher 的加密解密过程中，我们需要用到一对 RSA 公钥密钥对，这个公钥密钥对一般是通过 `openssl` 之类的软件生成，不由操作系统自己提供，需要我们手动输入，因而我们实现了 `gen_akcipher` 来生成这样的密钥对。同时在 Akcipher 中，签名和认证并不是对应所有的的 `&[u8]`均可以实现，而是对于每一个对应的待签名数据取一个摘要 Digest 来作为签名的数据，然后在认证过程中，需要同时传入签名前后的数据，因此我们实现了 `gen_digest` 来帮我们完成这个任务。

gen_akcipher

```
// Cargo.toml
[package]
name = "gen_akcipher"
version = "0.1.0"
edition = "2024"

[dependencies]
rsa = "0.9.0"
rand = "0.8"

// main.rs
use rsa::{RsaPrivateKey, RsaPublicKey, pkcs1::EncodeRsaPrivateKey,
pkcs1::EncodeRsaPublicKey};
use rand::rngs::OsRng;

fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Generate private key
    let mut rng = OsRng;
    let private_key = RsaPrivateKey::new(&mut rng, 2048)?;

    // Convert private key to Vec<u8> (PKCS#1 format)
    let private_key_bytes = private_key.to_pkcs1_der()?.as_bytes().to_vec();
    println!("Private Key (PKCS#1) as Vec<u8>: vec![{:?}]", private_key_bytes);

    // Get public key from private key
    let public_key = RsaPublicKey::from(&private_key);

    // Convert public key to Vec<u8> (PKCS#1 format)
    let public_key_bytes = public_key.to_pkcs1_der()?.as_bytes().to_vec();
    println!("Public Key (PKCS#1) as Vec<u8>: vec![{:?}]", public_key_bytes);

    Ok(())
}
```

gen_digest

```
// Cargo.toml
[package]
name = "gen_digest"
version = "0.1.0"
edition = "2024"

[dependencies]
sha2 = "0.10"

// main.rs
use sha2::{Sha256, Digest};

fn main() {
    // The data to be hashed (digest)
    let data = b"Hello, world!";

    // Create a Sha256 hasher
    let mut hasher = Sha256::new();

    // Update the hasher with the data
    hasher.update(data);

    // Get the digest (hash) as a result
    let result = hasher.finalize();

    // Convert the result to Vec<u8>
    let result_vec: Vec<u8> = result.to_vec();

    // Print the Vec<u8>
    println!("Digest as Vec<u8>: vec![{:?}]", result_vec);
}
```