# Rust-Keystone 可信执行环境

# 设计文档


项目成员：孙宇涛　清华大学

安一帆　清华大学

赖瀚宇　清华大学

指导教师：陈渝　清华大学

向勇　清华大学


2022　年　8　月

## 摘要

Rust-Keystone 可信执行环境的目标是在 RISC-V 指令集上，通过 Enclave 的设计模式，构建一个自底向上的安全执行环境。我们基于 UC Berkeley 的 Keystone 作为原型，利用 Rust 语言在 zCore 操作系统上，从 U 态的 SDK，S 态的 Driver，M 态的 Security Monitor 进行了重构，对原有的工程结构进行了优化，在性能和安全性上均有所提升。

在本届功能挑战赛的时间范围内，我们的预期工作目标与完成情况为：

1. 完成三个特权级的 TEE 模块的开发（完成）：我们利用 Rust 语言，在 unsafe 代码段极少的情况下，完成了三个主要模块的重构；
2. 将 TEE 的各个环节 Rust 化，打通完整的安全执行环境运行流程（基本完成）：在我们最终可以运行的系统上，我们可以成功在 zCore 的执行环境下，利用 Rust 版本的 SDK 执行 Rust 语言编写的用户程序，其余部分由于时间原因，暂时由 Keystone 的 C 语言版本替代；
3. 对 Host OS 和 Host App 交互的安全性与高效性进行优化（完成）：我们的系统相比于 Keystone 的原生版本，极大地减小了系统调用的执行次数，同时将安全敏感的操作迁移到底层执行；

我们希望 Rust-Keystone 作为一个参赛项目，在比赛结束之后仍然具有生命力，为开源社区可持续性的做出贡献：

1. 扩充了 zCore 操作系统的使用场景：作为一个全新的，用 Rust 语言开发的安全高效的操作系统，zCore 同时支持 Zicron 和 Linux 标准。我们在他们的标准之上，搭建起了一个可以执行 TEE 的体系；
2. 为 Rust 社区提供更多的样例：Rust 作为一个年轻的编程语言，其安全性和高效性是其在系统开发领域的优势，而我们可能是第一个将 Rust 语言引入 TEE 领域的团队；

# 目录

# 1 背景介绍

## 1.1 可信执行环境

机密计算（Confidential Computing）是近几年新兴的一项安全技术，它利用可信执行环境（简称 TEE）的强隔离和内存加密来保护 TEE 内的可信应用程序，确保了用户数据的机密性和完整性。机密计算一举解决了诸多应用场景中“信任”难题，包括但不限于公有云、区块链、多方数据分析等。

目前，所有主流的体系架构都已经推出了各自的 TEE 实现，比如 Intel SGX、ARM TrustZone、AMD SEV、Intel TDX 和 RISC-V Keystone 等。TEE 中要运行各种应用程序当然离不开 OS 的支撑；而 OS 的设计也必须充分考虑 TEE 的需求。

## 1.2 已有工作分析

对于一个完整的 TEE 来说，硬件的支持是必不可少的，我们无法通过纯粹的软件方法对可能的攻击进行隔离。对于不同的指令集，有些指令集推出了一些标准化的保护，而对于大部分成熟的指令集，业界都是通过扩展硬件功能的方式，为可信执行提供保障。

我们挑选一些有代表性的工作，分析他们在软硬件上的主要思想：

- Intel SGX 是 Intel 官方在 X86 指令集上推出的一套可信执行标准，它的核心思想是扩充若干条 U 态和 S 态的特殊指令，利用硬件限制对安全的程序进行隔离；
- AMD SEV 是 AMD 公司推出的一套 TEE，它的主要功能在于加密，其处理器中附加了 AES 加密引擎，用来快速地对用户内存进行加密，同时维护一套硬件的密钥管理系统，防止密钥本身被攻击者所窃取；
- ARM TrustZone 是一套软硬件结合的体系，它的设计模式也是 Enclave 模式，通过硬件的支持使二者进行连接；
- Keystone 是 UC Berkeley 在 RISC-V 指令集上，依靠 RISC-V 标准中的 pmp 内存隔离，通过 Enclave 的设计模式，对用户内存进行隔离、加密；
- Penglai 是上海交通大学团队在 RISC-V 指令集上推出的一套全新的 TEE 系统。相比于 Keystone，它在硬件上增加了 Guarded Page Table，可以以页为单位对页

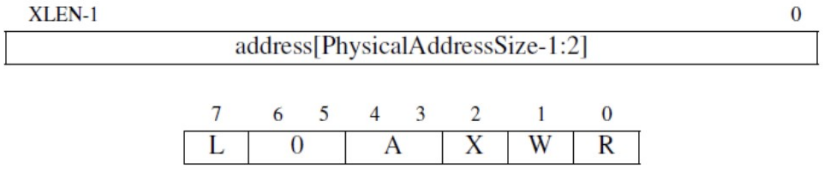


图 1: PMP 地址和配置寄存器。地址寄存器右移两位，如果物理地址位宽小于 XLEN-2，则高位为 0。R、W 和 X 域分别对应读、写和执行权限。A 域设置是否启用此 PMP，L 域锁定了 PMP 和对应的地址寄存器。

表上的物理地址进行保护；同时，它还在硬件中集成了 Mountable Merkle Tree，用于对内存的加密；

## 1.3 RISC-V 标准

前文提到了，在 RISC-V 标准中，已经存在了对内存进行隔离的指令，这一部分指令被设定在 M 态中执行，用来防止运行与上层特权级的程序对其他物理内存进行读写。我们的项目的隔离方法也是基于 RISC-V 标准的，这样就可以快速在 QEMU 平台上进行模拟。

实现了 M 和 U 模式的处理器具有一个叫做物理内存保护（PMP，Physical Memory Protection）的功能，允许 M 模式指定 U 模式可以访问的内存地址。PMP 包括几个地址寄存器（通常为 8 到 16 个）和相应的配置寄存器。这些配置寄存器可以授予或拒绝读、写和执行权限。当处于 U 模式的处理器尝试取指或执行 load 或 store 操作时，将地址和所有的 PMP 地址寄存器比较。如果地址大于等于 PMP 地址 i，但小于 PMP 地址 i+1，则 PMP i+1 的配置寄存器决定该访问是否可以继续，如果不能将会引发访问异常。

图 1 显示了 PMP 地址寄存器和配置寄存器的布局。两者都是 CSR，地址寄存器名为 pmpaddr0 到 pmpaddrN，其中 N+1 是实现的 PMP 个数。地址寄存器右移两位，因为 PMP 以四字节为单位。配置寄存器密集地填充在 CSR 中以加速上下文切换，如图 2 所示。PMP 的配置由 R、W 和 X 位组成，他们分别对于 load，store 和 fetch 操作，还有另一个域 A，当它为 0 时禁用此 PMP，当它为 1 时启用。PMP 配置还支持其他模式，还可以加锁，等等。

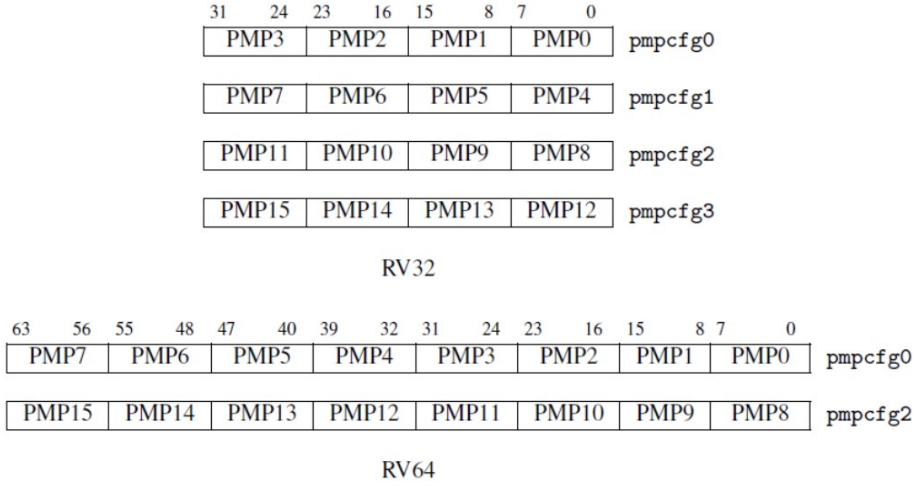


图 2: pmpcfg CSR 中 PMP 配置的布局。对于 RV32（上半部分），16 个配置寄存器被分配到 4 个 CSR 中。对于 RV64（下半部分），它们则分配到了两个偶数编号的 CSR 中。

## 1.4 Rust 语言

Rust 语言是一门全新的编程语言，它可以在不包含垃圾回收（GC）模块的基础上保证内存安全，因此无论是在 Web、嵌入式、系统开发领域，都有其独特的优势：

- 高效性：Rust 速度惊人且内存利用率极高。由于没有运行时和垃圾回收，它能够胜任对性能要求特别高的服务，可以在嵌入式设备上运行，还能轻松和其他语言集成；
- Rust 丰富的类型系统和所有权模型保证了内存安全和线程安全，让您在编译期就能够消除各种各样的错误；

因此，对于额外强调安全性的 TEE 语言，Rust 语言的优势就更加重要——我们必须防止系统本身的不安全性给系统带来的整个隐患。除此以外，由于现代的语法特性，Rust 相比于 C 语言能够在性能上不落下风。

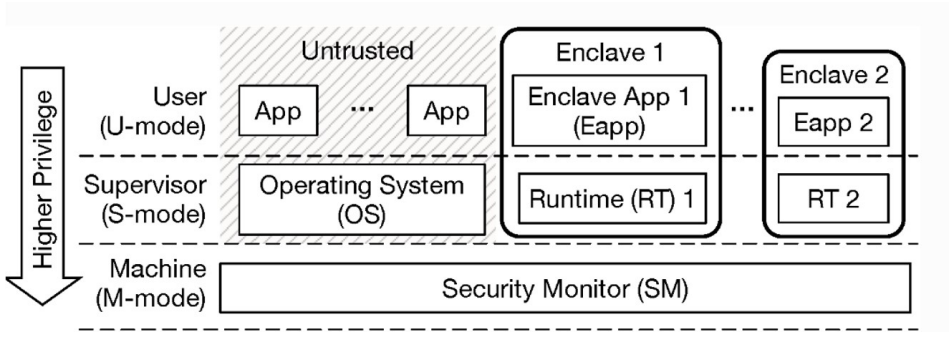


图 3: Enclave 可信执行环境的整体架构

# 2 系统运行逻辑

## 2.1 整体运行框架

Enclave 的设计模式的核心思想在于，利用底层的支持，将安全的 Runtime 和不安全的 Host OS 隔离开来，在二者之上分别运行独立的安全的/不安全的用户程序。

图 3 展示了整个的运行框架。其中，由安全监控系统来负责整个硬件的控制权调度，不同的 Enclave 和 Unstrusted 环境之间相互隔离，每一部分只能运行它们独立的内存空间。每一个小型的 Runtime 对每一个 Eapp 提供最基本的支持。

由于它功能比较弱且不能完全控制硬件，因此在 Enclave 的运行过程中还需要与不安全的环境进行交互。这个时候就需要一片共享的内存区域可以同时被 Enclave 和 Host 环境进行读写，这一部分称为共享内存。基于这部分内存，Enclave 就可以通过事先约定好的方式调用函数，完成 Runtime 所限制的功能。

## 2.2 创建与销毁流程

### 2.2.1 创建 Enclave

1. Host SDK: Host App 计算 Enclave 所需的内存空间，加载 Enclave 中需要的 runtime 和 eapp 到内存中;

2. Host OS: Host OS 在内存中开辟一块连续的内存，并且解析 elf，写 enclave 页表，并且保存必要的地址信息;

### 2.2.2 创建共享内存

1. Host SDK: Host App 计算所需要的共享内存空间，用于 host app 和 enclave app 的通信;
2. Host OS: Host OS 在内存中开辟一块连续的内存，将这一块物理地址同时写入两边的页表;

### 2.2.3 确认创建完成

1. Host SDK：发出 syscall，传递用于通信的共享内存的地址范围，交给 S 态处理;
2. Host OS：将若干参数传递给 sbi 层，包括私有内存段、共享内存段的物理地址，runtime 和 enclave app 的启动地址;
3. SM：分配和存储 Enclave 的必要信息，设置 PMP 内存隔离，至此以后，host 的 OS 和 app 无法访问 enclave 的私有内存段（但可以访问共享内存段）;

### 2.2.4 销毁 Enclave

1. Host SDK：通知操作系统可以进行销毁;
2. Host OS：释放私有内存与共享内存，清除保存的相应数据结构;
3. SM：将 pmp 保护解除，之后 host 可以自由访问相应内存段，也需要清楚保存的数据结构;

## 2.3 运行流程

### 2.3.1 运行 Enclave

1. Host SDK：Host App 发出 run/resume 的系统调用;
2. Host OS：调用 sbi call，将控制权交出;
3. SM：设置 pmp 内存可用，切换 csr 寄存器与通用寄存器，将 mepc 指向 runtime 上一次/入口地址，切换到 runtime;

4. Runtime：第一次执行：初始化用户栈，时钟中断，切换 stvec 的地址，设置共享内存段的地址以便之后的交互；恢复执行：从上一次 trap 的地址恢复执行即可；

5. Enclave SDK：正常执行安全的用户程序；

#### 2.3.2 时钟中断

1. Runtime：调用 sbi call，暂时退出 enclave；

2. SM：设置 pmp 保护私有内存，切换 csr 寄存器与通用寄存器，将控制权交还给 Host OS；

3. Host OS：实现上表现为 do nothing，直接将返回值返回给 Host App，实际上 Host 会执行正常的进程调度，执行 Host 上的其他程序；

4. Host App：识别返回值发现是时钟中断，调用 resume syscall，继续 Enclave 的运行。

#### 2.3.3 交互函数调用

1. Enclave SDK：调用 Ocall syscall，将控制权转交给 runtime；

2. Runtime：将必要的信息复制到与 Host App 共享的内存中，将控制权转交给 sbi；

3. SM：sbi 将控制权转交给 Host OS，其余切换细节与上文所述相同；

4. Host OS：将 sbi call 的返回值返回给 Host App，其中包含了 ocall 的信息；

5. Host SDK：根据预先设置好的方式进行函数调用，调用结束之后发出 resume syscall，恢复 Enclave 的运行。

# 3 用户程序 SDK 设计

## 3.1 创建 Enclave

### 3.1.1 创建准备

创建 Enclave 时，首先要在 Host OS 中加载 Enclave App 和 Enclave Runtime 的可执行文件到内存中。

由于 Enclave 环境并不掌握系统资源，许多任务需要借助 Host OS 或 Host App 来完成，这种机制被称为 Outbound Call (OCall)。在 Host App 中可以注册 OCall Handler 用于处理 OCall，这些函数将被保存在 Enclave 对象内。

### 3.1.2 创建 Enclave

在 Keystone 原有的设计中，在 Host OS 上运行的用户程序承担了大量的初始化任务，将 Enclave App 和 Enclave Runtime 加载后，为其分配内存并执行初始化，包括初始化页表和数据等。

在我们的实现中，直接将这部分代码整合进系统内核中，因此 Host App 只需将 Enclave App 和 Enclave Runtime 加载后交给系统内核执行相关的内存操作即可。Host SDK 与 Host OS 确认 Enclave 成功创建时，返回 Enclave 对象，Host App 可以通过该对象启动和销毁 Enclave。

## 3.2 运行 Enclave

Host App 通过 Enclave 对象的方法发起系统调用，由 Host OS 启动 Enclave 并转移控制权。

为 Enclave App 设计的 SDK 中包含了若干系统调用，这部分内容与 Enclave Runtime 相耦合。目前来说，SDK 提供了读写共享内存、发起 OCall 和结束 Enclave App 运行（Enclave App 发起系统调用，由 Enclave Runtime 发起 Sbi Call 将控制权转交给 Host OS，终止 Enclave Runtime 运行并返回 Enclave App 运行结果）的系统调用封装。

除读写共享内存由 Enclave Runtime 处理外，其他的操作都借助 Sbi Call 交给 Host OS 处理，在 Host OS 中，这一中断会交给 Host App 处理，即回到 Enclave 对象内部。对于 OCall 中断，Enclave 对象会根据创建 Enclave 时注册的 OCall Handler 处

理，处理结果保存在共享内存中，并通过相同的方式将控制权交回到 Enclave Runtime 中。

# 4 Host OS 的功能扩展

## 4.1 新增系统调用

在 Keystone 的设计中，由于需要对 Linux 进行适配，因此他的实现是以 Linux module 的形式完成的，通过重载 mmap 和 iotcl 这两个函数，借助文件句柄，就可以在完全兼容 Linux 系统调用的基础上实现相应的功能。我们同样参考这个设计，但是不同点在于，我们通过对内核的修改来实现，我们约定一个固定的文件句柄（类似 STDIN，STDOUT），完成了系统调用这一环节，可见图 4。

## 4.2 内存管理

前文提到了，Enclave 所需要的内存整体上可以分为两部分：私有内存 (epm) 和共享内存 (utm)。其中在私有内存中，存有 Runtime 和 Eapp 的代码段，栈空间，可以被 Runtime 管理的空闲空间，以及支撑这些虚拟地址的 Enclave 页表。

在 Enclave 创建时，我们不得不让 Host 环境对其进行初始化，因此在创建完成之前的阶段，私有内存是可以被 Host 环境访问的，我们需要在这个时间点之前完成以下操作：

- Runtime 和 Eapp 代码段的加载；
- 两个栈空间如果需要初始化，则对其进行初始化；
- 将初始化时所有用到的内存，包含共享内存，均写到页表中；

在 zicron 标准中，对于处理连续空间的地址分配是很谨慎的，因此我们无法通过 zCore 中对于中管理物理内存的结构体 VmObjectPaged 来实现。幸运的是，我们在这里不需要对于不同的内存块进行长期的、灵活的分配，只需要在创建时划分不同的部分用于不同的功能即可。因此，我们在初始化时 alloc 若干个连续的物理页，在实际需要时（mmap 时）再对其分配虚拟地址，再修改页表即可。因此，我们需要一个结构体来管理这一个连续的物理页，和一个分配算法，参见图5。

```rust
pub fn ioctl(cmd: Cmd, base: usize) -> LxResult<usize> {
    match cmd.match_field() {
        CREATE_ENCLAVE | DESTROY_ENCLAVE | FINALIZE_ENCLAVE | UTM_INIT => {
            if cmd.ioc_size() >= size_of::<CreateParams>() {
                let mut ptr: UserInOutPtr<CreateParams> = base.into();
                if let Ok(mut data : CreateParams ) = ptr.read() {
                    let ret : LxResult<usize>  = match cmd.match_field() {
                        CREATE_ENCLAVE => { create_enclave( params: &mut data) },
                        DESTROY_ENCLAVE => { destroy_enclave(&data) },
                        FINALIZE_ENCLAVE => { finalize_enclave(&data) },
                        UTM_INIT => { utm_init_ioctl(&mut data) },
                        _ => { Err(LxError::ENOSYS) }
                    };
                    if let Ok(_) = ptr.write( value: data) {
                        return ret;
                    }
                }
            }
            Err(LxError::EFAULT)
        },
        RUN_ENCLAVE | RESUME_ENCLAVE => {
            if cmd.ioc_size() >= size_of::<RunParams>() {
                let mut ptr: UserInOutPtr<RunParams> = base.into();
                if let Ok(mut data : RunParams ) = ptr.read() {
                    let ret : LxResult<usize>  = match cmd.match_field() {
                        RUN_ENCLAVE => { run_enclave(&mut data) },
                        RESUME_ENCLAVE => { resume_enclave(&mut data) },
                        _ => { Err(LxError::ENOSYS) }
                    };
                    if let Ok(_) = ptr.write( value: data) {
                        return ret;
                    }
                }
            }
            Err(LxError::EFAULT)
        }
        _ => { Err(LxError::ENOSYS) }
    }
}
```

图 4: zCore 中 TEE 对于 ioctl 的重载

```rust
pub struct MemoryRegion {
    // root_page_table: usize,
    // ptr: VirtAddr,
    pub size: usize,
    pub order: usize,
    pub pa: PhysAddr,
    pub frames: Vec<PhysFrame>
}
```

图 5: MemoryRegion 的实现

### 4.3 页表管理

在 Enclave 的运行中，我们自然需要一个独立的页表。而这个页表不能通过和 Host OS 相同的页表管理来实现，原因在于页表本身的位置是特殊的，它必须同样位于私有内存段，这样页表部分才可以受到保护。

Enclave 页表需要维护的虚拟地址段有：

- Runtime，Eapp 的代码段；
- 共享内存段；
- Runtime 的栈（这部分写到了 elf 中），Eapp 的栈（这部分指定一个虚拟地址，由 Host OS 进行分配）

在 zCore 的实现中，GenericPageTable 定义了一个页表的接口，定义结构体 EnclavePageTable，成员变量见图 6。注意到：

- 页表页的 alloc 依赖私有内存段对于物理页帧的管理；
- 写页表的操作必须在 finalize 之前，否则 sbi 写 pmp 寄存器之后将无法进行物理页帧的读写；
- Root page table 作为第一次调用 alloc 的位置，其物理地址与私有内存段 epm 的基地址相同；

其中，共享内存段需要同时写两个页表，我们通过 mmap 来实现，调用 Enclave 句柄的 mmap 时，OS 会写好 Enclave 的页表，同时 vmo 作为返回值，Host App 的页表会同时被写好。

```rust
pub struct EnclavePageTable {
    root: PhysAddr,
    epm: Arc<Mutex<MemoryRegion>>
}
```

图 6: EnclavePageTable 的实现

```rust
pub struct EnclavePageTable {
    root: PhysAddr,
    epm: Arc<Mutex<MemoryRegion>>
}
```

图 7: Enclave 的内核管理

### 4.4 Enclave 全局管理

内核需要管理一系列的 Enclave 的参数，以及 id 的分配，注意到：

- 参考 zCore 的其他栈式分配系统，利用 lazy_static 创建一个全局管理器；
- 由于需要频繁修改全局管理器所保存的参数，因此参考 rCore 的函数式编程来对其进行修改；
- 全局不可变 + 内部可变是 Rust 的常用设计，防止了预期之外的修改；

最终，我们设计的函数接口可见图 8。对于一个 Enclave 而言，我们需要对两段内存进行维护，记录 Enclave 的阶段信息，同时保存一些参数以便将来通过 sbi_call 传递给 SM，结构体设计可见图 7。

```rust
pub fn get_enclave_sbi_eid(&self, id: usize) -> LxResult<isize> {
    if let Some(enclave :&Enclave ) = self.enclave_map.get( k: &id) {
        Ok(enclave.eid)
    } else {
        Err(LxError::EINVAL)
    }
}

pub fn modify_enclave_by_id<F, T>(&mut self, id: usize, mut f: F) -> LxResult<T>
    where
        F: FnMut(&mut Enclave) -> LxResult<T>, {
    if let Some(enclave :&mut Enclave ) = self.enclave_map.get_mut( k: &id) {
        f(enclave)
    } else {
        Err(LxError::EINVAL)
    }
}
```

图 8: 全局管理器的修改函数

# 5 基于 Rustsbi 的安全监控系统

## 5.1 执行环境切换

安全监控系统 (Security Monitor, SM) 是运行于 M 态，基于 sbi 进行功能扩展的系统。不安全的 Host 环境和安全的 Enclave 环境之间需要来回进行切换，原因有：

- Ocall 调用、时钟中断会将执行权交还到 Host 环境；
- Enclave 的生命周期依赖 Host 的协助;

我们对于内存段的隔离保护也是在环境切换这一环节实现的。总的来说，在这一阶段需要维护以下上下文信息：

- 通用寄存器，这部分在 S 态与 M 态的切换中也存在；
- mepc 指向的运行位置，切换到另一个 S 态时，需要保存上一个 S 态的位置；
- 与中断异常有关的 csr 寄存器 (mstatus, mideleg 等)，这部分暂时遵循一个固定的标准；
- satp 寄存器，使得虚拟地址可以正常运行，Enclave 的页表在第一次切换时已经写好；

### 5.2 内存段保护

借助 RISC-V 标准中的 pmp 寄存器，我们可以保证在不安全的 Host 环境执行期间，无论是 OS 还是 App，都无法访问 Enclave 所保护的物理地址。这一部分依赖安全监控系统的实现，它需要在每次进入 Enclave 时，，使能相应内存段的 RXW 权限，而当 Host 运行时，禁止对应的权限。

在 Keystone 中，pmp 只是对物理内存最基础的一层保护，还可以在 Enclave 运行时，对内存进行一层加密。这一部分的实现位于 Runtime 模块，由于我们的项目没有对这一部分进行重构，因此不予赘述。但是可以肯定的是，pmp 阻止了软件上的攻击，而内存加密对于防止攻击者直接攻击物理内存本身，是有作用的。

## 6 功能评价

### 6.1 安全性

对于一个可信执行环境来说，在多大程度上可以防止外界攻击，是最核心的一个指标。对于 Enclave 模式的 TEE 而言，安全性依赖于以下几个方面：

- 在 Enclave 创建结束之前，Host OS 是安全的，不会篡改代码段，而开始运行之后可以是不安全的；
- M 态的安全监控系统是安全的，否则整个 pmp 的修改都失去了意义；
- 攻击者无法通过物理手段随意获取内存中的信息，并且在一定时间复杂度下进行计算。对于有限的内存信息，加密内存可以防止攻击者直接窃取内存信息，但是在 Keystone 系统中对于密钥的管理不是完全安全的，因此仍有潜在的风险；

### 6.2 高效性

对于我们的系统，通过分析得出，性能瓶颈取决于以下几个方面：

- 创建 Enclave 所需要的时间，在我们实现的系统中，为了尽量减少 Host App 对于 Host OS 的系统调用开销，将大部分可以集成到 Host OS 中的功能与 sdk 解耦，极大增强了性能；
- SM 对于两个 S 态之间切换的开销，这一部分主要依赖 sbi 的性能；

- Ocall 的调用频率，对于一个固定的 SM，Ocall 的调用频率在一定程度上取决于 runtime 可以支持的 syscall；
- Ocall 本身的性能开销目前来自于目前的结构设计（U-S-M-S-U）(在绝大多数情况下，进入 Host App 后会进行 syscall，这部分可以进行优化）

## 6.3 功能比较

由于我们的项目是参考 Keystone 的设计结构实现的，Keystone 本身运行于 Linux，Rust-Keystone 则运行于 zCore，因此直接对其进行定量的比较是不公平的，但是从理论上来说，可以比较出二者设计的不同，而我们的改进也来源于对 Keystone 结构的分析。

### 6.3.1 页表处理

在 Keystone 中，由 sdk 来对页表进行读写，每有一个新的页表产生时，它就会调用一次 mmap，这不仅极大地增加了系统调用的次数，也增加了潜在的安全风险。

在我们的系统中，用户态无法直接对 Enclave 页表进行读写操作，防止在创建 Enclave 环节其他 Host App 的攻击，减少了 syscall 的次数，保证了创建环节的高效性。

### 6.3.2 代码段处理

在 Keystone 中，由 sdk 对 runtime 和 eapp 的 elf 进行解析，加载到私有物理地址依赖不断调用 mmap 函数。

在我们的系统中，sdk 只负责将两个 elf 加载到内存中，其余的解析、加载到私有物理地址的操作均在 Host OS 中完成，这样又减少了 syscall 的次数，同时这部分内存不会泄露到不安全环境的页表中。

### 6.3.3 系统稳定性

在 Keystone 中，指针 + 全局变量承担了绝大部分操作，需要在所有指针操作处检验合法性，具有很大的不稳定因素。

在我们的系统中，Rust 语言保证了绝大多数安全性，极少数的 unsafe 操作体现为对物理地址的直接读写（页表），这部分比例极少，且不可避免。

# 7 未来展望

TEE 作为近年来学界和业界的关注热点之一，它的系统设计、安全保护，仍然在不断发展中。很多优秀的 TEE 系统仍然在不断涌现，跟进最新的研究，分析它们的设计模式，才能更好的对其进行改进、创新。

对于本项目而言，由于时间原因，我们无法做到系统的整个功能的完善和进一步的创新，但是我们认为，在未来，以下方面是有进一步发展空间的：

- 用 Rust 重构 runtime，增加一些系统调用。runtime 的能力一定程度上决定了整个系统的性能。为 Penglai 的设计中，runtime 的功能相比于 Keystone，就有了很大的长进，包括一个独立的文件系统。这样对于文件而言，也可以安全地进行加密。

- 改进用户态交互的机制，减少 Ocall 的性能开销。在 Enclave 的设计模式中，一次 Ocall 需要经历 5 个特权级，是十分低效的。我们认为，既可以直接调用 Host OS 的系统调用减小特权级切换的步骤，也可以采取异步调用的方式，将转化的开销下降到最低。

总的来说，在本次大赛中，由于缺乏系统级的开发经验，在环境配置、编译选项等环节踩了不少坑，在中间曾一度想放弃。但是柳暗花明的是，在所有的 Dirty Work 解决掉之后，整个的开发逐渐变得得心应手。

在这几个月的工作中，老师们为我们工作的时间规划、功能设置上提供了一些建议，实验室的学长们在环境配置与系统开发的经验上提供了很多帮助。在此致谢他们，为我们的项目保驾护航。