Глава 2. Основные принципы доверенной среды исполнения и модель угроз

2.1. Обзор доверенной среды исполнения

2.1.1. Определение и основные принципы

2.1.1.1. Доверенная среда исполнения

Доверенная среда исполнения (ДСИ), или же Trusted Execution Environment (TEE) - это безопасная, изолированная область внутри центрального процессора. Она предназначена для выполнения кода и обработки данных с повышенным уровнем конфиденциальности и целостности по сравнению со стандартной операционной средой, называемой универсальной средой исполнения (УСИ), или же Rich Execution Environment (REE).

В рамках ДСИ работает специализированная безопасная операционная система, называемая доверенной операционной системой (Secure OS), которая работает параллельно обыкновенной операционной системе в УСИ, называемой универсальной операционной системой (Rich OS).

ДСИ гарантирует защиту таких активов, как конфиденциальные данные и критически важный код, которые хранятся и обрабатываются в её пределах и остаются недоступными, даже в случае компроментации REE операционной системы или приложений УСИ.

Защита реализуется посредством сочетания аппаратных и программных механизмов, которые обеспечивают строгую изоляцию между ДСИ и УСИ. Функционирование и гарантии безопасности ДСИ базируются на ряде ключевых принципов, детально рассмотренных далее.

2.1.1.2 Изоляция

Изоляция в контексте доверенной среды исполнения представляет собой фундаментальный принцип разделения вычислительной среды на две физически разграниченные области: нормальный мир (Normal World), и защищённый мир (Secure World). Такое разделение гарантирует, что операции и данные в защищённом мире защищены от вмешательства или несанкционированного доступа со стороны программного обеспечения, работающего в обычном мире, включая основную операционную систему и её приложения. Для поддержания этого разделения обычно используются аппаратные механизмы изоляции, которые предотвращают прямой доступ кода из одного мира к ресурсам или памяти, выделенным другому миру, за исключением случаев, когда это явно разрешено через строго контролируемые интерфейсы.

2.1.1.3 Целостность

Целостность в TEE гарантирует, что код и данные, находящиеся и обрабатываемые в защищённом мире, остаются неизменными и сохраняют своё первоначальное состояние. Это означает, что ни внешние субъекты из обычного мира, ни неавторизованные процессы внутри защищённого мира не могут модифицировать критически важный код, такой как операционная система TEE или доверенные приложения, а также управляемые ими чувствительные данные. Целостность обычно обеспечивается такими механизмами, как безопасная загрузка (secure boot), подписанный и зашифрованый код, защита памяти во время исполнения и использование криптографических контрольных сумм. Это гарантирует выполнение только проверенных и аутентичных программных компонентов и защиту данных от злонамеренных изменений.

2.1.1.4 Конфиденциальность

Конфиденциальность - это принцип TEE, обеспечивающий защиту данных и кода в защищённом мире от несанкционированного раскрытия. Это означает, что никакой программный компонент, работающий за пределами TEE, включая потенциально скомпрометированную операционную систему обычного мира или привилегированные приложения, не может считывать или получать доступ к приватным областям памяти TEE. Конфиденциальность распространяется как на данные в процессе обработки (например, обрабатываемые криптографические ключи), так и на хранимые данные.

2.1.1.5 Защищённое хранилище

Защищённое хранилище (Secure Storage) - это функциональная возможность TEE, предоставляющая механизм для постоянного хранения чувствительных данных, таких как криптографические ключи, учетные данные пользователей или другая конфиденциальная информация, способом, который защищает их конфиденциальность и целостность даже при выключенном питании системы. Данные, сохраняемые с использованием этого механизма, шифруются и защищаются с использованием ключей, управляемых внутри TEE. Доступ к этим сохраненным данным строго контролируется TEE, гарантируя, что только авторизованные доверенные приложения могут извлекать или изменять принадлежащую им чувствительную информацию.

2.1.1.6 Аттестация

Аттестация позволяет TEE доказать свою подлинность и целостность исполняемой в ней программной среды удалённой стороне. Этот процесс обычно включает генерацию TEE подписанного отчёта, содержащего измерения её аппаратной конфигурации, микропрограмм, ОС и загруженных доверенных приложений. Удалённая сторона может проверить этот отчёт с использованием доверенного ключа, тем самым получая уверенность в том, что она взаимодействует с подлинной TEE и что конфигурация программного обеспечения соответствует ожиданиям и не была скомпрометирована.

2.1.1.7 Доверенное исполнение

Доверенное исполнение (Trusted Execution) гарантирует, что только авторизованный и проверенный код допускается к запуску в пределах TEE. Это включает механизмы аутентификации кода (например, с помощью цифровых подписей) перед его загрузкой и выполнением в защищённом мире. Ограничивая исполнение только известными и проверенными программными компонентами, TEE поддерживает контролируемую и безопасную среду, предотвращая выполнение вредоносного или недоверенного кода.

2.1.1.8 Минимальная доверенная вычислительная база

Принцип минимальной доверенной вычислительной базы (Minimal Trusted Computing Base, TCB) заключается в том, что совокупность всех аппаратных, микропрограммных и программных компонентов, критически важных для обеспечения политики безопасности системы, должна быть как можно меньше. Минимизация TCB сокращает поверхность атаки, упрощает анализ и верификацию безопасности, а также снижает вероятность наличия уязвимостей, способных скомпрометировать всю TEE. В TCB обычно входят защищённые аппаратные элементы, доверенные загрузчики, ядро Secure OS и набор основных доверенных сервисов. Основой доверия к самой TCB служит аппаратный корень доверия (RoT).

Chapter 3: Design and Implementation of the Secure Operating System

Interface Considerations

TEE Client API: Inter-World Communication Interface

OP-TEE on RISC-V

- *1.0 from TEE-Client-Api.md*

Develop own minimal GlobalPlatform TEE interface

- *2.0 from TEE-Client-Api.md*

Experemental of Research Prototypes

- *3.0 from TEE-Client-Api.md*

Rationale for Adopting a Global Platform-based API Subset

- Presents the justification for selecting a carefully chosen subset of the Global Platform TEE Client API.

System Architecture Overview

- section provides a high-level perspective on how the Secure OS is structured and how it interacts with the Normal World and hardware.

High-Level Architecture

Architectural Layers

- Introduces the layered nature of the system, from hardware/firmware (OpenSBI) to the Secure OS, and then to the Normal World OS (Linux).
- Emphasizes the isolation between the Secure World and the Normal World.

Secure vs. Normal World Overview

- Explains how the Secure OS permanently occupies the first CPU core while Linux runs on the remaining cores.
- Highlights the roles and responsibilities of each world, along with the boundary-enforcement mechanisms.

Core System Components

Kernel, Resource Managers, and TEE Services

- Details the internal architecture of the Secure OS, covering the Secure Kernel, resource managers (for tasks, memory, and IPC), and TEE service layers.
- Describes how these components collectively provide security, resource allocation, and runtime services to Trusted Applications.

Shared Memory and IPI-Based Communication

- Introduces the fundamental inter-world communication channels, such as shared rings/buffers used for request and response queues.
- Describes how RISC-V inter-processor interrupts (IPIs) are employed for signaling events and synchronizing data transfer between Normal and Secure Worlds.

Memory Layout and Addressing

Physical and Virtual Addressing

- Provides a high-level overview of how the Secure OS configures its page tables and manages physical/virtual addresses.
- Explains how memory mappings differ between the Secure World and the Normal World.

Isolation Mechanisms

- Details how World Guard extension enforces secure boundaries at the hardware level.
- Shows how the Secure and Normal Worlds remain isolated, preventing unauthorized access to protected pages.

Shared Memory Queues

- Explains the reserved memory regions that serve as shared buffers for secure–normal communication.
- Highlights concurrency concerns and locking strategies for ring-buffer manipulation.

Secure OS Execution Flow

Boot Process Overview

- Summarizes the critical steps in transitioning from OpenSBI to the Secure OS, and eventually handing over the remaining cores to Linux.
- Points to more detailed discussion in the “Secure Boot Process and Initialization” section.

Inter-World Transitions

- Outlines the mechanism by which execution moves between Secure and Normal Worlds (e.g., SMC calls, interrupts).
- Covers validation checks before granting world transitions and how the OS ensures secure state persistence.

Scheduling in Secure OS

- Highlights how the Secure OS manages tasks and threads in a uniprocessor environment (the first core).
- Discusses scheduling policies, context switching logic, and how TEE tasks do not interfere with Linux scheduling.

Security and Policy Enforcement

Capability-Based Security Model

- Introduces the core concepts behind object handles, secure syscalls, and fine-grained access control.
- Explains how capabilities are validated and enforced at runtime to prevent privilege escalation.

World Guard Integration

- Consolidates the hardware-based checks provided by the World Guard extension with the Secure OS's software policy.
- Provides an overview of failure handling when unauthorized accesses or invalid world transitions occur.

TA Lifecycle

Creation

- Describes how Trusted Applications (TAs) are registered or loaded by the Secure OS.
- Explores memory allocation, initial code setup, and the procedure for spawning a TA process or thread.

Compute

- Outlines how a TA executes in the Secure OS, including interaction with system calls, access to resources, and concurrency.
- Discusses how TAs can communicate with other tasks or the Normal World during their operational phases.

Teardown

- Explains the orderly shutdown of a TA, covering handle cleanup, memory deallocation, and final status reporting.
- Ensures that no sensitive data remains accessible and that the system reclaims all resources.

WorldGuard Integration

WorldGuard Configuration

World Configuration (Two-World Model)

- Overview of how the system hardware and memory are split between Secure World and Normal World.

- Explanation of the two-world design rationale, focusing on isolation guarantees.
- Definition of the roles of each world (e.g., Secure OS vs. Linux).
- Description of how World IDs (or similar identifiers) are assigned and managed.

WorldGuard Checker Configuration for Secure Isolation

- Overview of the hardware/software checker mechanism for enforcing world-based isolation.
- Configuration of Secure RAM slots and memory regions:
    - Secure memory partitioning approach.
    - Locking down memory regions to the Secure World
- Setting up enclave/partition boundaries:
    - Handling multiple enclaves (if applicable) within the Secure World.
    - Policy for controlling access across enclaves or from Normal World.
- Integration of memory attributes (e.g., read/write/exec permissions) with WorldGuard checks.

Integration with the Secure OS

Error Reporting

- Mechanisms to detect and report WorldGuard-related violations (e.g., unauthorized access attempts).
- Logging and reporting structure within the Secure OS for debugging and auditing.
- strategies for halting offending tasks in case of critical errors.

Managing World Transitions

- Description of the control flow when switching between Normal World and Secure World.
- Handling interrupt-driven transitions across worlds.
- Use of specific CPU instructions or registers to invoke transitions (if applicable).
- Ensuring minimal overhead while maintaining security guarantees.

Communication Pages

- Shared memory pages allocated with permissions for both worlds:
    - Shared memory region layout and alignment considerations.
    - Ensuring read/write restrictions are enforced by WorldGuard.

Secure Boot Process and Initialization

- This section describes how the Secure OS is bootstrapped, transitioning from platform firmware (OpenSBI) to a fully initialized secure environment.
- It covers early assembly-level initialization, kernel relocation, MMU enablement, and higher-level subsystem initialization, ultimately concluding with the handover to any “rich OS” components.

Secure OS Early Initialization

OpenSBI Handover

- Explanation of the OpenSBI boot protocol, which provides the Secure OS with the initial register context (e.g., a0, a1 containing specific parameters).
- SBI boots FW_PAYLOAD_PATH (TEEOS futher) on boot core, making this core secure
- High-level overview of how the Secure OS entry point (_start) is invoked by OpenSBI.
- Handling or storing system parameters (such as the device tree pointer) for further use.

Setting Up the Stack and Basic Memory Layout

- Allocating a stack in physical memory for secure execution.
- How the assembly code (head.S) calculates the stack location (via PAGE_SIZE * 6).
- Ensuring stack alignment for correct RISC-V operation.

First Kernel Relocation

- performing a “relocation” step due to pie (position-independent executable) nuances.
- Creating an identity mapping (physical == virtual) at the kernel load address while also mapping the kernel at its designated virtual base (KERNEL_VIRTUAL_BASE).
- Use of large page mappings (e.g., 2MB or 1GB mappings) for simplicity during early boot.

Enabling the MMU

- Explanation of how the SATP register is configured
- Ensuring the kernel text, data, and bss segments are accessible at both the physical region and the kernel virtual address.

Secure OS Initialization

- Once the minimal MMU and basic mapping are established, the Secure OS transitions to its primary C environment for final setup.

Register Console

- Initializing and registering the console driver (e.g., SBI console) as the primary I/O channel.
- Setting up early debug/log printing to assist with error reporting.

Initialize Page Tables

- Creation and configuration of more granular page tables beyond the initial large block mappings.
- Structures for dynamic region registration and page-level protections.

Second Kernel Relocation (If Needed)

- Further re-mapping kernel virtual addresses after early-boot.
- Cleanup of temporary mappings used during the first relocation phase.

Initialize Trap Handler

- Setting up the vector table or exception table to handle synchronous exceptions and interrupts.
- Registering fault handlers, system call handlers, and other critical exception vectors.

Initialize Timers

- Configuring RISC-V timer CSRs or platform-specific timer hardware.
- Setting up the early tick or scheduling timers.

Initialize Page Allocator

- Creation of a physical page allocator (pmm_init()) to manage secure RAM.
- Data structures (e.g., contiguous free-lists, bitmaps) for tracking page usage.

Initialize Slab Allocator

- A higher-level memory allocator (kmalloc or slab-based).
- Allocation of kernel objects (e.g., tasks, threads, pipes) efficiently.

Initialize Scheduler

- Setup of the scheduler data structures to manage secure OS threads or tasks.
- Timer-driven scheduler hooks using the timer subsystem.

Initialize Root Task

- Creation of the root task (or initial user-mode process in the Secure World).
- Loading or spawning any essential system services.

Initialize Normal World Communication Channel

- Setting up shared memory regions or queues for Normal World <-> Secure World communication.
- Configuring interrupt mechanisms or other signaling channels (e.g., IPI).

Initialize Trusted Applications

- Loading and initializing built-in or pre-installed Trusted Applications (TAs).
- Setting up an environment for TAs, including memory isolation, scheduling, and system call interfaces.

Rich OS Initialization

Initialization by OpenSBI

- Handing control back to OpenSBI to continue its normal boot flow for a Linux or other rich OS.
- TEEOS setups itself and does special ecall that indicates that it has finished
- SBI boots NWD_FW_PAYLOAD_PATH (REEOS futher) on other cpus

Core Startup

- after returing to sbi on secure core - sbi will start second non secure core
- second core will start Linux Kernel, and Linux will hotplug other cores by itself
- Linux will not try to run on first secure core, because it was marked "secure" at the begining of OpenSBI startup

OpenSBI modifications

...

Cross-World Communication

- This chapter describes the mechanisms enabling data exchange and signaling between the Secure OS running on the primary CPU core(s) and the Normal World (e.g., Linux). It focuses on the shared memory queues, the message structure used for TEE commands, and the IPI mechanism for sending interrupts across RISC-V cores.

Shared Memory Queues

- One of the fundamental mechanisms for communication between the Secure World (SWd) and Normal World (NWd) is through shared memory queues. This approach allows concurrent message passing without requiring complex locking operations.

Lock-Free Queue Algorithm

- *https://pskrgag.github.io/post/mpmc_vuykov/*

Shared Memory Ring Buffers

- Overview of how the queues are physically placed in shared memory pages accessible to both SWd and NWd.
- Ring buffer layout: circular array of message slots, head/tail pointers, and optional “sequence” fields for synchronization.
- Memory alignment considerations for preventing false sharing or alignment-related issues.

Requests Queue

- A dedicated ring buffer where the Normal World places requests that the Secure World must handle.
- Steps for enqueuing:

    1. Normal World driver writes the message into the ring slot.
    2. Driver updates the queue head pointer using an atomic operation.
    3. IPI sent (or polling mechanism invoked) to notify Secure World.

Responses Queue

- A separate ring buffer for the Secure World to provide responses or event notifications back to the Normal World.

- The Secure World writes its response into the ring slot, increments the tail pointer, and relies on NWd polling.

Canary Around Shared Pages

- Canary pages are placed around Shared pages with no access bits, so any access by overflowing will trap

Shared Memory Regions

- Aside from the primary queues, certain larger buffers or data structures may be shared.

Memory Region Allocation

- allocation is done by calling secure operation TEE_CMD_ID_MAP_SHARED_MEM
- allocation is done in Secure World, it will allocate pages and set access to Secure world and normal world
- then Secure OS will map pages to Secure Kernel address space to be able to access them
- them Linux should map these pages to Linux Kernel address space

Memory Region Deallocation

- deallocation is done by calling secure operation TEE_CMD_ID_UNMAP_SHARED_MEM
- Secure World will deallocate pages, and remove access from both Secure world and Normal world
- them Secure os will unmap pages from Secure Kernel address space
- then Linux should unmap pages from Linux Kernel address space

Data transfer

- since memory is mapped to Linux Kernel and Secure OS, Operating Systems can transfer data just by regular memory reads and writes

Message Structure

- All commands passed through the request/response queues typically adhere to a consistent message format. This section details the wg_tee_cmd structure, which encapsulates TEE operation parameters and results.

struct wg_tee_cmd

This structure holds command identifiers, session tracking, error codes, and additional parameters.

field id

- uint32_t id
- Identifies the type of TEE operation.
- Possible values include:
    - TEE_CMD_ID_OPEN_SESSION
    - TEE_CMD_ID_CLOSE_SESSION

- TEE_CMD_ID_INVOKE_CMD
- TEE_CMD_ID_MAP_SHARED_MEM
- TEE_CMD_ID_UNMAP_SHARED_MEM

field seq

- uint32_t seq
- filed seq is a unique identifier of command
- it's value is generated just by atomically incremented sequence counter

field session_id

- uint32_t session_id
- Identifies which session within a TA this command belongs to.
- Allows a single TA to manage multiple open sessions simultaneously.

field func_i

- uint32_t func_id
- each Trusted Application implements it's own functionality, and TA can do multiple actions
- field func_id describes what action to do inside TA

field err

- uint32_t err
- Used by the Secure World to return error codes or status results.
- possible results include indicating success, permission failures, or other errors.

field uuid

- uint8_t uuid[16]
- A unique identifier for the Trusted Application.
- Used during TEE_CMD_ID_OPEN_SESSION to select the correct TA.

field paddr

- uint64_t paddr
- A physical address relevant to TEE_CMD_ID_MAP_SHARED_MEM; indicates the start page to map as shared.
- field remain unused for other command IDs.

field num_pages

- uint32_t num_pages
- The number of contiguous pages to map starting at paddr, for TEE_CMD_ID_MAP_SHARED_MEM.
- field remain unused for other command IDs.

field shmem_id

- uint32_t shmem_id
- A handle returned by the Secure OS to reference a mapped shared memory region.
- Allows subsequent TEE_CMD_ID_UNMAP_SHARED_MEM to unmap region

struct wg_param params

- Holds 4 arguments (each is 24 bytes).
- Simple arguments are stored directly
- memory references are stored as three 64-bit values (size, offset, world_id).

padding

- field padding has size of 96 bytes
- Reserved space to align the structure to 256 bytes overall.
- Prevents unwanted compiler padding from interfering with the queue alignment.

IPI Based Signaling

- While the shared queues provide a data structure for messages, an Inter-Processor Interrupt (IPI) mechanism triggers real-time notifications.

RISC-V IPI Mechanism

- High-level overview of the RISC-V interrupt controller and how software sets an IPI to a target hart.
- Explanation of relevant CSRs, memory-mapped interrupt lines, or OpenSBI calls for sending IPIs.
- Typical flows: setting a bit in the IPI register or invoking sbi_send_ipi with a hart mask.

Normal to Secure World Signaling

- Procedure in which the Linux driver or NWd service:

    1. Fills out wg_tee_cmd struct
    2. push struct in the request queue.
    3. Triggers an IPI to the secure hart via an OpenSBI call.
    4. Waits for response by polling the response queue

Secure to Normal World Signaling

- Due to the RISC-V architecture constraints, the simplest approach is for the Secure OS to place responses in the response queue without any other signaling of Normal World
- An IPI back is restricted because of limitations of RISC-V ISA - we can not distinguish Secure OS notification IPI from other types of IPI, so Linux will not be able to handle IPI correctly
- so the NWd driver periodically checks the response queue
- The requesting thread is then woken, a result is available.

TEE API

Global Platform API

Introduction to Global Platform API

- *introduction from OP-TEE Global Platfrom API spec*

TEE Client API

TEE Contexts

- *contexts chapter*

TEE Sessions

- *sessions chapter*

TEE Shared Memory

- *shared memory chapter*

TEE API Specification

TEEC_UUID

- describe ...

TEEC_Result

- describe ...

TEEC_Context;

- describe ...

TEEC_Session;

- describe ...

TEEC_Value;

- describe ...

TEEC_RegisteredMemoryReference;

- describe ...

TEEC_Parameter;

- describe ...

TEEC_Operation;

- describe ...

TEEC_SharedMemory;

- describe ...

TEEC_InitializeContext

- describe ...

TEEC_FinalizeContext

- describe ...

TEEC_OpenSession

- describe ...

TEEC_CloseSession

- describe ...

TEEC_InvokeCommand

- describe ...

TEEC_AllocateSharedMemory

- describe ...

TEEC_ReleaseSharedMemory

- describe ...

Linux Driver Implementation

- This section details the design and implementation of the Linux driver responsible for communication with the Secure OS.
- The driver uses the kernel's TEE subsystem interfaces to expose the Secure OS functionality to user-space applications.
- It establishes shared queues for request and response messages, initializes kernel threads for communication polling, handles incoming replies, and provides TEE-driver–compliant operations such as open session, close session, and invoke command.

Driver Overview and Registration

- Before diving into the shared queues and communication routines, the driver must be discoverable by the Linux kernel and properly registered within the TEE subsystem:

Linux Driver Initialization

1. The device tree node "riscv-wg/nwd_channel" is parsed to read the physical addresses of the shared request and response queues (SQ and CQ).
2. Memory for the driver control structure (struct wgtee) is allocated.
3. Shared queues are remapped into kernel space (ioremap) and initialized with the lock-free MPMC queue mechanism.
4. A TEE device (struct tee_device) is allocated and registered using tee_device_alloc() and tee_device_register().
5. The driver registers a set of callbacks (wgtee_ops), including open_session, invoke_func, etc., which allow user space to interact with the Secure OS through the standard TEE_IOCTL API.

- Key driver entry points: - module_init(wgtee_driver_init) — For device instantiation and registration. - module_exit(wgtee_driver_exit) — For cleanup and teardown.

Linux Driver Interface

- The Linux driver implements TEE-driver–compliant operations exposed via wgtee_ops: - get_version() — Returns the TEE driver version (implementation ID and capabilities). - open() / release() — Allocate or free per-context data (in wg_user_context). - open_session() — Handle TEE_CMD_ID_OPEN_SESSION. - close_session() — (Currently returns -ENODEV as a placeholder). - invoke_func() — Handle TEE_CMD_ID_INVOKE_CMD, forwarding parameters to the Secure OS. - cancel_req() — Not currently implemented.
- To accommodate the TEE subsystem's generic parameter structures (struct tee_param), the driver provides helper routines: - wg_convert_params_in() — Converts Linux TEE parameters into wg_param structures. - wg_convert_params_out() — Converts results back into Linux TEE param outputs.

Shared Queues from the Linux Side

- Shared memory queues form the primary communication channel between the Normal World (Linux) and the Secure World (Secure OS). There are two distinct types of queues in use:

Requests Queue

- The Requests Queue (SQ - Submission Queue) holds commands that the Normal World wishes to send to the Secure OS. For instance, when a user space application issues a TEE_IOCTL command (such as open session or invoke function), that request is packaged into a wgtee_cmd structure and placed into this queue.
- The queue is initialized via wg_communication_init().
- The driver uses mpmc_queue_push() to insert a command (wgtee_cmd) onto the queue.
- Each command is tagged with a unique sequence number (seq) to match responses.

Responses Queue

- The Responses Queue (CQ - Completion Queue) collects asynchronous responses sent back by the Secure OS. Whenever the Secure OS finishes handling a command from the SQ, it places a completed wgtee_cmd structure, including any output parameters or errors, onto the CQ.
- The driver polls this queue in a dedicated kernel thread.
- The mpmc_queue_pop() routine removes the next response command.

- The driver matches responses to ongoing requests by seq number.

Linux Communication Interface

- Communication with the Secure OS involves several key steps: initialization, creating the polling thread, enqueueing commands, waiting for replies, and finalizing when the system is shut down or the module is removed.

Communication Initialization

- The wg_communication_init() routine is responsible for:

1. Mapping the physical memory for the CQ and SQ into kernel virtual addresses (ioremap).
2. Initializing both mpmc queue data structures with mpmc_queue_init().
3. Starting a dedicated kernel thread (polling_thread) that continuously checks for completed commands in the CQ.

Queue Initialization

- Each queue (SQ and CQ) is implemented using the lock-free MPMC (Multiple Producer Multiple Consumer) circular queue: - Memory layout is backed by a contiguous region (one page for SQ, one page for CQ in the current setup). - mpmc_queue_init() sets up the ring buffer indices (head and tail) and ensures alignment. - ioremap_prot() is used to obtain a kernel virtual address for these pages.

Communication Polling kthread

- A single kernel thread (wg_polling_thread) handles responses from the Secure OS by:

1. Checking the CQ queue for any ready responses.
2. Matching each retrieved response to a waiting request via the seq field.
3. Signaling the appropriate completion object (struct completion) so the requesting context can proceed.

- The thread runs until the driver is unloaded or the system is halted.

Message Sending

- When a Normal World request is issued to the Secure OS (e.g., open session, invoke command, or close session):

1. The driver constructs a wgtee_cmd structure, populating fields such as id, func_id, parameters, etc.
2. A unique sequence number is generated with atomic_fetch_inc(&seq_number).
3. The request is placed into the SQ using wg_queue_push().
4. An IPI (sbi_send_ipi) is dispatched to wake up the Secure OS core.

Getting the Result

- Each inflight request has an associated completion entry (wg_completion_entry). The list of waiting requests is protected by a spinlock (waiting_lock). When the Secure OS

writes a response onto the CQ:

1. The polling thread pops it from the CQ.
2. The polling thread scans the waiting_requests list for a matching seq.
3. Once found, it copies fields into the original wgtee_cmd, calls complete() on the associated completion, and removes it from the waiting list.

Communication Finalization

- During driver removal (module_exit) or general teardown:

1. The polling thread is stopped (kthread_stop).
2. Mapped I/O memory for both queues is unmapped (iounmap).
3. The TEE device is unregistered, and any allocated kernel structures are freed.

Дизайн и реализация ядра Secure OS

- В данной секции рассматриваются основные аспекты проектирования и реализации ядра защищённой операционной системы (Secure OS).
- Предложенный подход основан на использовании объектно-ориентированного (capability-based) подхода и позволяет обеспечить высокий уровень изоляции между задачами (tasks) и сервисами (threads), а также гарантировать безопасность при взаимодействии с внешними ресурсами и другими компонентами системы.
- Рассмотрим детальную структуру, начиная с ключевых сущностей и механизмов управления.

Kernel Objects and Handles

- Взаимодействие с ресурсами внутри ядра построено вокруг объектной модели. Каждый объект в системе (например, задача, поток, виртуальный объект памяти, канал связи и т.д.) имеет свой уникальный дескриптор (handle). Доступ к функциональным методам объекта и внутреннему состоянию определяется набором capability-флагов (прав доступа).

Tasks (Processes)

- Задачи (tasks) являются основными изолированными сущностями в Secure OS.
- Они содержат собственное адресное пространство (vm_space) и набор потоков (threads).
- Кодовая составляющая задачи находится в пользовательском адресном пространстве безопасной среды (для Trusted Applications).
- Каждая задача имеет таблицу объектов (object_table), где регистрируются все ресурсы (включая каналы, объекты памяти и др.), предназначенные для её использования.
- создание задачи состоит из следующих этапов:

1. Создание пустой пользовательской задачи (task_create_empty) – формирование объектов структуры task, включая инициализацию отдельных полей (handle_page, object_table и т.д.).
2. Запуск задачи (task_run) – настройка защиты памяти для страницы дескрипторов (handle_page), установка состояния (TASK_SPAWNED) и добавление главного потока задачи в планировщик (sched_insert).

3. Уничтожение задачи (task_destroy) – освобождение адресного пространства (vm_space_destroy) и освобождение динамически выделенной памяти.

- Пример создания задачи на основе системного вызова task_spawn показывает, как пользователь может передать в ядро указатель на точку входа (ep) и handle задачи, а ядро создаёт в ней начальный поток и переводит задачу в состояние выполнения.
- *пример*

Threads

- Каждая задача содержит по крайней мере один поток (thread).
- Потоки отвечают за выполнение кода внутри адресного пространства задачи.
- При создании пользовательского потока ядро настраивает контекст выполнения (регистры, стек, текущее окружение и т.д.).
- При создании процесса (task), ядро создаёт «главный поток» задачи, устанавливая ему точку входа (ep). После этого поток регистрируется в списке потоков (list_head_add_tail) и может быть запланирован планировщиком.

Pipes (или Channels)

- Вместо классических «каналов» (pipes) в ядре используются объекты «channel» (двухсторонние каналы связи). Они создаются фабрикой (см. factory.c, syscall channel_create) и позволяют процессам или компонентам ядра обмениваться сообщениями по дескрипторам с установленными привилегиями.
- Канал представлен двумя концами (rx/tx), которые могут принадлежать одной или разным задачам.

Virtual Memory Objects

- Объекты виртуальной памяти (VM Object) отображают некоторый участок памяти (обычно физической) в адресное пространство задачи.
- В ОС реализован системный вызов vmo_create, который позволяет создавать vm_object через фабрику (factory_object).
- После создания объект регистрируется в таблице дескрипторов с соответствующими правами (OBJECT_CAP_TRANSFER, VMO_CAP_FULL и пр.), что позволяет разграничивать операции чтения, записи и копирования.

Synchronization Primitives

- В ядре имеется набор примитивов синхронизации
- spin_lock
- mutex
- semaphore
- Очереди ожидания (wait_queue_init, wait_object_many) – абстракция, позволяющая потокам ждать появление событий (например, данных в канале).

Task Management

- Механизм управления задачами (Task Management) в Secure OS определяет систему, в рамках которой задачи создаются, запускаются и завершаются.
- В коде ядра представлены несколько ключевых подсистем, реализующих данный

функционал.

Process Model

- Процессная (task) модель предполагает, что каждая задача имеет собственное адресное пространство и набор ресурсов, зарегистрированных в ядре.
- Создание задачи обычно происходит по запросу пользовательского процесса либо при создании сессии для новой TA, через системный вызов (task_create). Перед запуском задачи ядро выделяет нужные структуры данных, инициализирует объект задачи и подключает его к планировщику. При этом задача находится в состоянии TASK_CREATED, пока не будет вызвана функция task_spawn, переводящая её в состояние TASK_SPAWNED.

IPC Service

- В системе используется механизм IPC на основе каналов (channels). Каждая задача может получить дескрипторы двух сторон канала, позволящие выполнять операции чтения/записи (channel_read()/channel_send()).
- Кроме того, для организации группового ожидания сообщений служит вызов wait_object_many, позволяющий одним системным вызовом ожидать события от нескольких объектов.
- Пример кода root_task демонстрирует задачу, которая ожидает сообщения в канале kernel_channel. При появлении нового сообщения конструкция wait_object_many(...) возвращает события от одного или нескольких объектов.
- Затем сообщение извлекается (channel_read) и, результат работы, отправляется обратно (channel_send).

Root Task

- Root Task (root_task.c) является важной задачей, поддерживающей цикл обработки входящих запросов и сообщений от других процессов и ядровых сервисов. Здесь можно заметить:
- Использование структуры wait_entry для инициализации нескольких объектов ожидания (каналов).
- Циклическую обработку arriving-сообщений.
- Вызов nwd_proccess_message для обработки поступивших команд от Normal World
- Таким образом, root task служит центральной точкой обмена сообщениями между Secure OS и Normal World.

Scheduling

- Планировщик отвечает за распределение ресурсов процессора между потоками, находящимися в состоянии готовности к выполнению. В ядре реализованы базовые механизмы планирования, ориентированные на простоту и расширяемость.

Scheduling Service

- Scheduling Service управляет структурами данных, хранящими информацию о потоках (thread) и их состояниях.
- При создании потока он добавляется в очередь работы планировщика (sched_insert), где ему присваивается базовый приоритет и используется политика

Round-Robin (SCHED_RR), которая обеспечивает циклическое распределение времени процессора между всеми runnable-потоками.

Scheduling Policies

- В качестве политики планирования используется Round-Robin, чтобы обеспечить максимальную простоту и уменьшить количество доверенного кода, так как доверенные приложения не обладают сложной логикой, требующей более точной диспетчеризации
- В данной реализации (task_create_initial_thread инициализирует потоки с SCHED_RR) применяется классическая политика Round-Robin. Однако архитектура планировщика может быть расширена для поддержки: - Приоритетного планирования (приоритеты на основе критичности задачи). - Планирования на основе квантования времени (time-slices). - Специальных политик для реального времени (real-time scheduling).

Memory Management Subsystem

- Подсистема управления памятью (Memory Management Subsystem) обеспечивает надежную изоляцию памяти между задачами, а также предоставляет безопасный интерфейс распределения памяти в пространстве ядра и пользовательских задач.

Secure Memory Allocator

- Для работы с динамически распределяемой памятью в ядре используются следующие механизмы:

1. Kmalloc/kvfree (или kfree) для управления небольшими блоками памяти в пространстве ядра (см. пример в factory_destroy или task_create_empty).
2. Специализированные аллокаторы страниц (vm_allocate, vm_space_init_kernel, vm_space_init_user), которые выделяют виртуальные адреса и сопоставляют их с физической памятью, учитывая защитные атрибуты (VMA_READ, VMA_WRITE, VMA_USER).

- В момент инициализации задачи (task_create_empty) вызывается vm_space_init_user для подготовки пользовательского адресного пространства

Memory Isolation

- Механизм изоляции достигается с помощью:

1. Различных адресных пространств (space) для каждой задачи.
2. Управления таблицами страниц (mmu_switch_space), что позволяет ядру переключаться между контекстами задач и гарантировать, что одна задача не может получить доступ к памяти другой.
3. Контроля прав доступа к памяти при вызове vm_protect. Данная функция устанавливает соответствующие флаги (VMA_READ, VMA_WRITE) и обеспечивает недоступность памяти для неавторизованных задач.

- В коде виден пример, когда при запуске задачи (task_run) вызывается vm_protect для установки в памяти read-only доступа к странице дескрипторов (handle page). Таким образом, даже сама задача ограничена в манипуляции с полями, ответственными за управление дескрипторами, если это не предусмотрено

соглашениями по доступу.

File System

linear RAM fs

elf files

Capability-Based Security Model

- This chapter focuses on the design and implementation of the capability-based security model within the Secure OS.
- It explains how the system uses handles to encapsulate capabilities, how these handles are created and managed, and how capability-based access control is enforced through task manifests and the root task.

Handles as Encapsulated Capabilities

- Handles in the Secure OS function as references to system resources (objects). Each handle has associated permissions and metadata defining how it may be accessed or manipulated. Internally, handles map to kernel-managed descriptors that maintain the state, permission bits, and relevant object pointers.

Design Rationale

- Least Privilege Principle: Capabilities ensure that tasks and trusted applications only have the minimum set of privileges needed.
- Fine-Grained Access Control: Provides precise control over which resources can be accessed and how they are used.
- Composability: The handle-based model allows different system components (tasks, services, etc.) to dynamically create and share capabilities in a structured manner.

Objects

- Definition: Objects represent protected resources (e.g., memory regions, tasks, communication channels).
- Creation: created by a specialized factory object or created initally by kernel
- Management: The kernel and corresponding resource managers maintain object lifecycles (allocation, reference counting, destruction).

Object Handles

- Semantics: An object handle is an token referencing an underlying object.
- Handle Table: Each task or trusted application maintains a handle table
- Security Properties: Handles cannot be duplicated or guessed; only the kernel can create valid handles.

Factory Objects

- Factory Concept: There is a singleton act as “factory” capable of creating other objects (e.g., tasks, pipes, or memory objects).

- Controlled Creation: A Manifest ensures that only permitted tasks can spawn or instantiate new objects.
- Lifecycle: Factoty themselve is created by the kernel.

Object Methods

- Method Calls: Operations on objects (e.g., read, write, map) are exposed as system calls.
- Capability Checks: Before performing any operation, the kernel verifies that the caller's handle has sufficient permissions.
- Extensibility: New object types can not define custom methods, which stricts permission volations

Capability-Based Access Control

- The system enforces a strict capability-based security policy, ensuring only authorized handles may invoke methods on objects.

Permissions

- since syscalls act as object methods - there is a fixed number of methods that can be executed on object
- each handle has its own permission bits for each syscall
- Permission Propagation: When a handle is shared between tasks, permissions can only be stricted, to increased.
- Revocation: The kernel can invalidate or downgrade a handle's permissions at runtime if security conditions change.

Task Manifests

- Manifest Format: Each task has a manifest specifying its initial handles and allowed permissions on those handles.
- Initialization: On task creation, the kernel reads the manifest to populate the task's handle table.
- Dynamic Policy: The root task or a privileged controller can update or revoke handles from TA

Root Task

- Privilege Level: The root task is endowed with the highest level of privilege, including the ability to create new tasks and objects
- Handle Distribution: Upon launching a new trusted application, the root task provides the necessary initial handles listed in the manifest.
- Security Enforcement: The root task can audit or modify the capabilities of any other task if required.

Method Invocation

- Invocation Flow:

    1. Trusted application issues a syscall to invoke a method on a handle.
    2. Kernel checks handle validity and permission bits.
    3. Kernel executes the method if authorized; otherwise returns an error.

- Parameter Passing: Depending on the object type, additional data (e.g., memory buffer addresses or message payload) must be specified.
- Audit Logging: A log of handle usage may be maintained for debugging, accountability, and forensics.

Performance Implications

- Lookup Overheads: A balanced design attempts to keep handle operations lightweight to avoid excessive overhead.

Secure Syscalls

- The Secure Operating System exposes a set of privileged system calls (“secure syscalls”) available only to code running in the Trusted Execution Environment (TEE). These syscalls form the backbone of the secure OS abstraction layer and are fundamental to the capability-based model which enforces strict access and isolation. In this section, we describe the secure syscall mechanism, their capability enforcement, and the secure object operations made available to Trusted Applications (TAs).

Secure Entry Points

- The secure syscall interface is the only gateway through which TAs and system objects interact. These system calls are verified and dispatched via central syscall routing infrastructure based on a syscall table indexed by syscall number.

Background on OS System Calls

- A system call facilitates user mode code (in this case, Trusted Applications) to invoke kernel functionality in a controlled and verified manner. Syscalls operate strictly on handles referencing kernel-managed secure objects. The handle list is process-local and authorized through task manifests at TA launch time.

Secure Syscall Lifecycle

- Each secure syscall follows the canonical secure OS object lifecycle:

1. User passes handle(s) and optional state.
2. Kernel resolves the handle reference and asserts access rights using capability tags.
3. Operation is executed atomically.
4. Ownership of resulting objects or memory copies is well defined (copy vs. move semantics).
5. Errors from any stage are returned to the user-space Trusted Application. Syscalls involving inter-task communication (e.g., SYS_CHANNEL_WRITE or SYS_TASK_SHARE_HANDLE) often cooperate with internal kernel structures like queues or per-process handle pages. These components are designed to be strictly partitioned and race-free.

Argument Passing Format

- Syscall arguments are passed following the calling convention of the Secure OS, and include the handle identifiers, pointers to user data, data lengths, and capability-specific flags. Data is validated before being dereferenced or mutation occurs. Shared

memory is always accessed via secure copies using kernel-managed copy_from_user and copy_to_user primitives.

Validation of Handle Permissions

- Each syscall entry validates whether the calling task has the needed capabilities for the given object type. Handles are looked up in the invoking task's object table, and if the lookup fails or lacks the proper capability flags (TRANSFER, WAIT, SEND, RECV, etc.), the syscall returns an error. This fine-grained check ensures robust per-object and per-action filtering in line with the capability-based security model.
- Syscalls rely entirely on the underlying capability-based object model:
- Handles are opaque 32-bit values resolved into kernel pointers via per-task object tables.
- Each handle links to an internal object and permission mask.
- All syscall-side object accesses invoke a type + capability check. For example, in the `channel_write` syscall, the following enforcement occurs:

1. Validate caller owns the provided handle with `CHANNEL_CAP_SEND`.
2. Validate all message-passed handles include the `TRANSFER` capability.
3. Receiver will only receive transferred handles if it has adequate handle table space. This guarantees that:

- Object accesses are never implicit — they must be manifested in the task manifest.
- No object leaks across Trusted Application boundaries.
- Origin and access pathway of each resource is traceable through the handle fabric.

Syscall Specification

- Each system call operates on one or more kernel object handles. Object types include tasks, threads, virtual memory objects (VMOs), factory objects, and channels. Below is an overview of the currently defined syscalls.

SYS_LOG

- Logging interface for debugging output from the secure world.

SYS_VM_ALLOCATE

- Allocates anonymous virtual memory inside a task's virtual address space.

SYS_VMO_CREATE

- Requests creation of a virtual memory object (VMO) via a factory. The new handle is stored in user-writable memory. Capability `FACTORY_CREATE_VMO_CAP` must be present.

SYS_CHANNEL_CREATE

- Asks the factory to produce bidirectional channel endpoints. Returns two handle values referencing peer-connected `channel_endpoint` objects. Requires `FACTORY_CREATE_CHANNEL_CAP`.

SYS_CHANNEL_READ

- Attempts to retrieve a pending message from the associated channel endpoint. Caller must possess `CHANNEL_CAP_RECV`. The syscall verifies receiver-side buffer, optional handle array, and copies message from kernel space.

SYS_CHANNEL_WRITE

- Enqueues a message to be sent over a channel endpoint. Requires capability `CHANNEL_CAP_SEND`. Handles being transferred are verified for `OBJECT_CAP_TRANSFER`.

SYS_TASK_CREATE

- Asks a factory to create an empty task object (stub process). Returns a handle with `TASK_GET_SPACE_CAP` and `TRANSFER`. Initial state is `TASK_CREATED`.

SYS_TASK_GET_SPACE

- Grants the caller access to another task's virtual memory address space (typically for explicit handle passing or object serialization). Requires handle with `TASK_GET_SPACE_CAP`.

SYS_VM_MAP_VMO

- Maps an existing VMO into a task address space with specified offset and permissions.

SYS_VM_FREE

- Unmaps a virtual address range from a VM space.

SYS_TASK_SPAWN

- Spawns a previously created task. Initializes the main thread with a provided entry point. Transitions the task's state from `CREATED` to `SPAWNED`.

SYS_OBJECT_CLOSE

- Releases the given handle in the caller's object table.

SYS_OBJECT_WAIT_MANY

- Waits on multiple kernel objects, useful for synchronization/IPC.

SYS_PHYSMAPPER_MAP

- Maps physical memory ranges for I/O accesses, used by device drivers or MMIO facilities (access controlled based on TA manifest and task capabilities).

SYS_TASK_SHARE_HANDLE

- Allows handle transfer from one task to another. The handle is written along with an identifier string into a memory area expected by the recipient. Available only when receiver is in `CREATED` state. Enforced via handle-page layout in receiver's address context.

SYS_OBJECT_COPY

- Duplicates a handle within the same task, assigning the requested capability mask. Used to derive restricted-view handles (e.g. remove `TRANSFER`).

Trusted Application Framework

- The Trusted Application (TA) Framework provides a lightweight, secure runtime environment and development interface for writing user-mode Trusted Applications atop the Secure OS.
- It defines a standard C runtime environment enriched with system capabilities accessible via a handle-based capability model.
- Its primary role is to facilitate secure software development, ensuring alignment with both TEE security policies and performance constraints in constrained environments.

Standard Library for Trusted Applications

Standard Library Overview

- The standard library is a minimal libc equivalent tailored to the Secure OS TEE context. It provides essential functionality typically found in a standard C runtime, excluding non-secure system calls. Implemented entirely in secure world userspace, the library avoids dynamic linking or unnecessary runtime overhead. It includes: - Memory functions (e.g., memcpy, memset) - Formatting and I/O (e.g., printf) - Math functions (including support for hardware-accelerated routines if available) - Cryptographic primitives - Concurrent synchronization mechanisms - Container utilities (e.g., lists, maps) - Typed object and handle access abstraction

Handle Operations Specification

Channel Functions

- `channel_read` - Read data from a secure communication channel.
- `channel_write` - Send data over a secure communication channel.
- `channel_from_handle` - Cast a generic handle into a channel type.

Factory Functions (Fabric Object Handle Interface)

- `factory_init` - Prepare a factory object for spawning or object creation.
- `factory_create_vmo` - Create a Virtual Memory Object (VMO).
- `factory_channel_create` - Create a new communication channel.
- `factory_task_create` - Create and launch a new Trusted Application task.
- `factory_get_handle` - Retrieve system/manually assigned handles.

Object Functions

- `object_copy` - Duplicate a handle reference.
- `object_close` - Close and discard a handle.

Task Functions

- `task_spawn` - Spawn a new task using a manifest.
- `task_share_handle` - Share handle(s) with another task securely.

Memory Management Functions

- `vm_init` - Initialize virtual memory structures.
- `vm_map_vmp` - Map memory pages into a task's virtual space.
- `vm_free` - Free allocated virtual memory regions.

I/O Standard Library Specification

Printf-Compatible Functions

- `printf` - Wrapper using `tee_log` syscall.
- `sprintf` - Internal memory-safe string writing variant.
- `vprintf` - Variadic-style printf handler.

Logging Function

- `tee_log` - Internal secure log syscall (invokes `SYS_LOG`, tagged output).

Strings Standard Library Specification

String Utility Functions

- `memset`
- `memcmp`
- `memcpy`
- `memmove`
- `memchr`
- `strlen`
- `strchr`
- `strcmp`
- `strtol`
- These are implemented using size-optimized and alignment-aware techniques for low-overhead TA memory environments.

Math Standard Library Specification

Algebraic Functions

- `sqrt`, `pow`, `log`, `exp`, `abs`, `floor`, `ceil`

Trigonometric Functions

- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`

Mathematical Constants

- `pi`, `e`, `inf`, `nan`

Complex Math Functions

- Complex number support is syntactically mirrored from real-number APIs.

Crypto Standard Library Specification

Hashing Functions

- `sha256(data, len)`
- `sha512(data, len)`
- `md5(data, len)`

Encryption/Decryption Functions

- in future work support for functionality like:
- `aes_encrypt`, `aes_decrypt` - Support for AES-GCM/CTR if hardware-accelerated
- `chacha20_encrypt`, `chacha20_decrypt`

Key Management and Derivation

- in future work support for functionality like:
- `hkdf` implementation
- Insecure vs. hardware-sealed key storage distinction

Random Number Generation

- in future work support for functionality like:
- `crypto_rng` - Hardware-backed RNG where available
- `crypto_rng_init_seed` - Optional API for seed injection

Container Standard Library Specification

- Note: Trees are all reentrant and zero-alloc in TA context

List Functions

- Singly Linked List: Init, Push, Pop, Find, Remove
- Doubly Linked List: Bidirectional traversal APIs with embedded nodes

Radix Tree Functions

- Insertion, deletion, lookup optimized for dense ID spaces

WAVL Tree Functions

- Self-balancing tree, relaxed AVL variant, with logarithmic insert/remove

Red-Black Tree Functions

- Balanced binary tree implemented using node-color rules

Concurrency Standard Library Specification

Atomic Operations

- `atomic_add_fetch`
- `atomic_sub_fetch`
- `atomic_or_fetch`, `atomic_and_fetch`
- `atomic_read`, `atomic_write`
- Memory barrier primitives: `smp_rb()` (read barrier), `smp_wb()` (write barrier)

Mutex API

- `mutex_init()`, `mutex_lock()`, `mutex_unlock()`, `mutex_destroy()`

Spinlock API

- `spinlock_init()`, `spin_lock()`, `spin_unlock()`, `spin_trylock()`

Semaphore API

- `sem_init()`, `sem_wait()`, `sem_post()`, `sem_destroy()`

Conditional Variables

- `cond_init()`, `cond_wait()`, `cond_signal()`, `cond_broadcast()`

Misc Library Functions Specification

Align Macros

- `align_up(x, a)`
- `align_up_ptr(p, a)`
- `align_down(x, a)`
- `align_down_ptr(p, a)`
- `is_aligned(x, a)`
- `is_aligned_ptr(p, a)`

Bit Manipulation

- `bit32(n)`, `bit64(n)`
- `is_power_of_two(x)`
- `clz32(x)` - Count Leading Zeros (32-bit)
- `clz64(x)` - Count Leading Zeros (64-bit)
- `log2(x)`

Compiler and Intrinsic Macros

- `barrier()` - Compiler-level memory fence
- `container_of(ptr, type, member)` - Offset-based typed accessor
- Standardized `__attribute__` usage for alignment/enforced inlining.
- `same_type()` - Static type matching check (debug-mode only)

Implementation Challenges and Optimizations

- This section discusses the practical challenges encountered during the development of the Secure OS and outlines optimizations applied to ensure a balanced trade-off between performance, security, and maintainability.
- It also examines developer tooling, build considerations, and key lessons learned from implementation.
- The intent is to provide transparency about the engineering process and to offer insights to future developers working in similar environments.

Performance vs. Security Trade-Offs

Balancing Isolation with Speed

- Design decisions around compartmentalization, memory isolation, and privilege levels.
- Trade-offs in choosing monolithic kernel vs microkernel OS
- Trade-offs in choosing user/kernel boundaries for TAs vs. monolithic kernel TAs.
- Security isolation for TAs vs. higher communication overhead through system call boundaries.

Inter-World Communication Overhead

- Overhead from shared region polling, memory copy costs, and IPI-based signaling.
- Use of lock-free ring queues to reduce latency.
- Minimizing trap/return paths for better inter-world round-trip latency.

Scalability Limits on a Single-Core Secure OS

- Implications of limiting Secure OS execution to a single core (core 0).
- Managing multiple active TAs and long-running calls to maintain responsiveness.
- Use of concurrent threads within Secure OS vs. thread serialization.

Memory Footprint Optimizations

Static Allocation vs. Dynamic Allocation

- When and why static memory regions were used (boot sequence, early kernel sections).
- Justification for dynamic allocation fallback features (slab and page pool management).
- Minimal boot allocator and on-demand zeroing mechanisms.

Slab Allocator and Page Pool Efficiency

- Custom lightweight slab allocator optimized for isolated heaps.
- Fit-to-size classes to reduce fragmentation.

Minimal Kernel Subsystem Design

- Avoiding traditional bloated kernel designs (e.g., no sysfs, vfs).
- Core components only: task/thread scheduling, IPC, memory isolation, syscalls.
- Benefits of small Trusted Computing Base (TCB) in verification and auditability.

Debugging Considerations

Logging from Secure OS

- Secure and minimalistic logging channel
- Multiple logging levels (panic, error, info, debug).

Debugging TAs in Isolation

- tracing
- remote debugger hooks (gdb)

Instrumentation Techniques

- Internal counters for scheduling decisions, memory allocations, syscall hit counts.
- Tracing memory allocator usage (per-task page count or slab lifetime tracking).

Fault Isolation and Crash Analysis

- Handling invalid syscalls in Secure OS and malformed commands from untrusted Linux driver.
- Watchdog for runaway tasks and per-task isolation to reduce blast radius.

Build System and Packaging for TAs

Trusted Application Build Flow

- TA toolchain constraints (compiler hardening in libtacore).
- Source tree layout that enforces separation of kernel, libraries, and apps.
- Manifests for capabilities, memory regions, and initial handles.
- Options for static TA linking into kernel image vs. runtime TA loading.
- Binary format (e.g., ELF) parsing from kernel for runtime spawning.

Kernel Build System

- CMake based build system for Secure OS

Development Tooling Support

- QEMU and hardware launch wrappers (scripts for OpenSBI + Secure OS + Linux images).
- Secure OS runtime shell commands for debugging tasks and memory maps.
- Developer stubs and host-side tools for image generation, symbol resolution.

Testing and Validation

Unit Testing Secure OS Components

- Internal test cases for linked list manipulation, allocator correctness, timer behavior.
- Framework for checking invariants (vmspace protections, handle tables).
- Building unit tests into kernel image and controlled via boot parameters.

Integration Testing with Linux

- End-to-end TEE client interface validation: context creation, open session, invoke command.

Security-Oriented Tests

- Handle table fuzzing framework (e.g., reusing/releasing invalid handles).
- Fault injection infrastructure via Linux-side IPI storms, memory overwrites, syscall replay.

Summary of Implementation

Overview

- Summary of major challenges solved during implementation
- Design principles that proved effective (e.g., capability model, static memory layout).

Codebase Structure Summary

- Secure OS code structuring: split into - ta (contains trusted applications) - kernel (contains arch, lib, mem, sched, sync, tasks, tests) - libtee (contains user space library) - scripts (contains build scripts, codegeneration scripts, backtrace, etc)
- Tools and CMake-based build system granularity.

Opportunities for Improvement

- Feature hardening: memory encryption, exploit mitigations like stack canaries.
- Potential for multicore support within Secure OS given future WorldGuard changes.
- Technical debt around early bootloader glue code and MMU bring-up code.

Chapter 4: Evaluation and Security Analysis

Software Stack Setup

- This section provides a detailed description of the full software environment used to support and validate the Secure OS, focusing on emulation, build infrastructure, and integration with toolchains.
- This chapter is essential to reproduce the development setup and benchmark context.

Toolchains

- Description of RISC-V GCC or LLVM toolchain versions, secure OS and TA compilation flags, linker scripts used, and build script wrappers.

Development Environment

- Recommended dev environment setup: OS dependencies, Make/CMake/gcc versions, scripting helpers, debugging support (e.g., GDB hooks to Secure OS), and virtual machine setup, if applicable.

Emulation Environment

- A robust emulation setup using QEMU provides a virtual platform to simulate the RISC-V World Guard hardware and enables rapid development and testing.

QEMU with WorldGuard Support

- Explanation of QEMU version used and modifications or forks maintained to support the World Guard extension.

QEMU Configuration

- QEMU settings used during emulation: memory map, number of harts, device tree blob (DTB) settings, and boot arguments necessary to launch both Secure OS and Linux.
- Also covers usage of debugging options and UART output customization.

Linux

- A Secure World-aware Linux kernel build is a key part of the integration testing, providing the userland-controlled "Normal World" for Secure OS interaction.

Linux with WorldGuard Support

- Brief explanation of the version/fork of the Linux kernel used, including any upstream or out-of-tree patches to support Secure OS interaction and WorldGuard.

Linux Configuration

- Kernel config menu options (e.g., minimal init system, character device support, TEE driver integration) and explanation of chosen configurations.

Linux Image

- Process of creating the kernel image and initial RAM filesystem (initramfs); integration into QEMU boot flow and linkage with rootfs/init and Secure OS debug output collection.

Build System

- The build system is centralized and modular to build various components including the kernel, trusted applications, OpenSBI, and Linux.

CMake Configuration

- How the overall build system is managed using CMake files: compiler toolchains, cross-compilation targets, component path registration, and reusability across Secure OS kernel and TA build systems.

CMake Build System Design

- Code organization and dependency separation. Build phases: TA compilation, kernel linking, staging and image generation. Covers options or build presets (e.g., debug vs release), and the way it cooperates with toolchains and QEMU images.

Trusted Application (TA) Build Flow

- Explains how a TA is built, manifests are generated, linking to standard libraries, and embedding into final system images.

CI Integration

- Automated testing ensures regression-free development and reliable build stability.

Continuous Integration Setup

- Framework used for testing (e.g., GitHub Actions, GitLab CI, Jenkins), pipeline stages—such as QEMU boot test, TA invocation test—and artifacts generation.

Automated Testing Scripts

- Details on scripts and system outputs validated within CI. Boot success, basic syscall availability and secure/normal world boundary integrity.

Demonstration of Secure OS Functionality

- In this section, we showcase the practical use and integration of the Secure OS, including building the software stack, developing a minimal Trusted Application (TA), and demonstrating its execution using the Linux-side communication driver.

Building the Software Stack

- This subsection outlines detailed steps to build all required components for running the full software stack in a QEMU-based RISC-V emulated environment.

Cloning Project Repositories

- Steps to clone Secure OS, OpenSBI, Linux kernel, QEMU, and any required dependency sources.

Building the Cross Toolchain

- Building or fetching a RISC-V cross-compilation toolchain.
- Required versions and environment setup.

Building the WorldGuard-Enabled QEMU

- Building QEMU with required patches for WorldGuard support.
- Notes on configuration flags and validation of WorldGuard support.

Building the Patched OpenSBI

- Compilation of an OpenSBI fork with additional code to support WorldGuard boot flow.
- Integration of Secure OS handoff from M-mode.

Building a WorldGuard-Aware Linux Image

- Configuration options for enabling WorldGuard awareness in Linux.
- Applying required patches, configuring the kernel, and generating an image.

Building Secure OS

- Step-by-step instructions on configuring and compiling the Secure OS kernel.
- Overview of CMake-based build, memory layout specification, and output binaries.
- Preparing and including predefined TA manifests.

Assembling Bootable Image

- Integrating OpenSBI, Linux, and Secure OS into a QEMU-bootable image.
- Image layout overview and loading addresses.

Example Trusted Application: Simple Arithmetic TA

- Presents the practical development of a basic trusted application that offers simple arithmetic operations, to serve as a demonstrative example.

Writing a Simple Trusted Application

- Creating a minimal TA: hello world service (or similar).

Defining TA Manifest for Capability Model

- Creating a manifest describing required permissions and object handles.
- Integrating the TA in root task's manifest for launch.

Building the Trusted Application

- Using build system to compile and link the TA.
- Output format (ELF) and file placement for boot.

Demonstration and Execution

- This subsection illustrates booting full system with communication between Linux and Secure OS via the sample TA.

Booting the System

- Launching QEMU with appropriate flags and verifying CPU/world isolation.
- Boot logs highlighting OpenSBI handoff and Secure OS initialization.

Initializing the Linux Driver for Secure OS Communication

- Loading the Linux kernel module for Secure OS communication.
- Debug output validating setup of shared communication queues.

Opening a Session to Trusted Application

- Using the TEE Client API to initialize a session to the sample TA.
- Debug logs showing session creation and rendezvous with Secure OS.

Invoking the TA Function and Receiving Response

- Sending invoke command (e.g., multiplication request) from Linux-side.
- Observing execution through debug output from Secure OS and TA.
- Logging TA's output and Linux-side response resolution.

Visualizing Capability Enforcement (Optional)

- Showing an attempt to call unauthorized method or access restricted handle.
- Logging access denial via kernel enforcement logic.

Debugging and Logging Support

- Tail of secure log buffer dumped through secure syscall.
- Logging from TAs printed using standard I/O libraries.

Security Analysis

- In this section, we evaluate the security posture of the Secure OS by analyzing the resilience of its architecture to a variety of threats according to the threat model defined in Chapter 2.
- For each scenario, we describe the setup, the simulated or real attack, and the system's actual response.

- The analysis is grounded in practical testing on an emulation environment backed by software instrumentation and tracing.

Resilience against Normal World Attacks

- This subsection evaluates the Secure OS against a potentially hostile rich OS (Linux) running in the Normal World.

Unauthorized Access to Secure Memory

- Attack Setup: Linux kernel driver attempts to read/write physical page mappings belonging to Secure OS.
- Observation: Memory protection enforced with World Guard prevents access; bus errors raised correctly.

Unauthorized Access to Secure OS/TA Code

- Attack Setup: Linux attempts to scan Trusted Application code or Secure OS binary via /dev/mem or similar - - methods. Observation: Memory remains inaccessible due to hardware separation and lack of mapping in the normal world.

Attempts to Corrupt Shared Memory Queues

- Attack Setup: Malformed or oversized requests injected into shared communication pages.
- Observation: Secure OS validates request format; invalid requests rejected and logged, avoiding buffer overflows.

Exploiting CWC Protocol (Cross World Communication)

- Attack Setup: Linux attempts to race against a Secure OS processing request by overwriting in-flight messages.
- Observation: Lock-free queue only uses relative indexes in accesses, so it prevents reads and writes from unexpected pointers. So if index is corrupted (out of bounds) - it will be ignored

Resilience against Buggy Trusted Applications

- Important to ensure Secure OS protects itself even in case of flawed Trusted Applications.

Inter-TA Isolation

Attack Setup: One TA attempts to access memory or channel of another TA. Observation: Physical and logical isolation enforced; failed access due to missing capabilities; Secure OS rejects request.

Capability Enforcement Engine

Attack Setup: Malformed syscall using an invalid or forged handle; fuzzing against the Secure OS syscall interface. Observation: Consistent rejection due to absence of capability

introspection in root task's manifest.

TA Resource Misuse Protection

Attack Setup: Malicious or buggy TA requests excess memory, opens too many handles. Observation: Secure OS enforces per-task quotas; resource exhaustion attempts fail gracefully.

Side-Channel Attacks

Scenario: Timing variation attacks on Secure OS execution; cache access pattern leakage. Observation: Not possible separation due to architecture-level isolation (e.g., cores/cache).

Additional Attack Scenarios and Limitations

- This section groups attacks that are currently outside the full mitigation scope or require future hardening efforts.

Physical Attacks

Scenario: Direct DRAM probing, bus sniffer or glitching attacks on OTP or memory controller. Observation: Physical attack resistance not yet applied; relies on external platform features.

Complexity of Trusted Computing Base (TCB)

Exploration: How large and auditable is the TCB? Observation: Secure OS kernel remains minimal and auditable, but Trusted Applications contribute to overall TCB and must be vetted.

Chain of Trust Attacks

Scenario: Malicious OpenSBI image or Secure OS loader. Observation: Evaluation based on boot integrity; Secure Boot implementation assumed but not fully integrated in prototype.

Performance Evaluation

- This section evaluates the runtime behavior, efficiency, and resource usage of the Secure OS, with emphasis on inter-world communication, Trusted Application (TA) execution overhead, and system resource constraints in a constrained RISC-V environment.
- Measurements are taken on an emulated platform using WorldGuard-enabled QEMU.

Latency of Secure OS Operations

- This subsection presents a detailed breakdown of the latency involved in interaction between the Normal World (Linux) and the Secure World (Secure OS), following the GlobalPlatform API lifecycle: session open, command invocation, session close.
- Note: All latency measurements should include mean, standard deviation, and max/min values with multiple runs.

Session Open Latency

- Time required to open a session to a TA via `TEEC_OpenSession()`
- Includes context switch, message construction, capability validation, and TA instantiation
- Factors influencing latency (TA manifest size, cold start vs. warm start)

Command Invocation Latency

- Latency of `TEEC_InvokeCommand()` to a previously opened TA Session
- Breakdown of fixed syscall overhead, scheduling delay, and parameter marshalling
- Synchronous vs. asynchronous (if supported)

Session Close Latency

- Time to teardown the session and reclaim resources
- Includes cleanup of handles, session state, and Secure OS bookkeeping

Communication Performance

- Evaluation of the throughput and timing characteristics of the CWC (Cross World Communication) mechanism, emphasizing Serializable Command Queues backed by Shared Memory.

Throughput of CWC Channel

- Maximum achievable throughput of request/response cycles through shared memory queues
- Impact of message size

IPI Signaling Overhead

- Cost of using Inter-Processor Interrupt (IPI) for signaling between worlds
- Measurement of context switch delay due to IPI
- Comparison of IPI costs under load and idle scenarios

TA Context Switch Overhead

- Cost of switching between multiple TAs or restoring TA context for a scheduled operation

Kernel Entry/Exit Transition Overhead

- Benchmark the time overhead to perform Secure OS syscalls (without actual work)
- Use of synthetic minimal syscall to isolate context switch cost

Memory and Resource Footprint

- Analysis of memory usage of the Secure OS and associated components under typical workloads.

Memory Footprint of Secure OS

- Static memory usage (text/data/bss sizes)
- Dynamic memory usage at runtime (heap usage, page tables)
- Total footprint with N TAs loaded

Per-TA Resource Consumption

- Memory consumption per individual TA context
- Number of handles, stack size, VMO usage

Shared Queue Overhead

- Memory and CPU overhead of shared ring buffers for communication queues
- Lock-free implementation impact

Scalability Limits and Bottlenecks

- Maximum number of concurrently active TAs
- System behavior under memory pressure
- Fragmentation and memory management limitations

Future Work

- This chapter outlines promising directions for extending and improving the Secure OS platform for the RISC-V WorldGuard architecture. It includes technical enhancements, additional security features, hardware support expansions, and rigorous verification procedures.

Advanced TA Features

- This section discusses enhancements to the Trusted Application (TA) framework that would allow TAs to offer more sophisticated services while still retaining minimal TCB and robust isolation.

Secure Storage

- Introduce support for tamper-resistant persistent storage to enable confidential data access, including sealed key-value storage SDK for TAs.

Attestation

- Add support for both local and remote attestation with cryptographic proof of measurement and TA identity, potentially backed by a platform Root-of-Trust chain.

Root of Trust

- Define or integrate a hardware/software Root of Trust, including secure provisioning mechanisms and interaction with system boot.

Cryptographic Services

- Provide Trusted Applications with a standardized, hardware-accelerated crypto runtime offering: symmetric encryption/decryption, asymmetric crypto, hashing, digital signatures, and secure RNG.

Porting to Real RISC-V Hardware with WorldGuard

- Move from QEMU-based evaluation to physical RISC-V hardware that implements the WorldGuard extension for real-world performance measurements and evaluation under physically observable systems.
- Identify WorldGuard-compatible silicon platforms
- Implement board-specific OpenSBI and bootloader adaptations
- Hardware debugging framework integration

Performance and memory

Multicore Support for Secure World

- Extend the Secure OS runtime to allow execution on multiple cores, introducing challenges around synchronization, inter-core TA instance affinity, and capability tracking in a multithreaded environment.
- Secure world scheduling policy for multiple harts
- Shared state consistency between cores
- Scalability tuning and bottleneck analysis

Dynamic TA Loading

- Introduce support for on-demand loading and unloading of TAs to reduce secure world memory usage and enable more complex applications.
- TA cryptographic signature verification before loading
- TA manifest validation and integration with runtime capability system
- Secure memory isolation upon load/unload

Security enhancments

Formal Verification of Secure Components

- Augment Secure OS with formal verification techniques for core components, especially the kernel, capability system, and secure IPC primitives, in order to reduce the Trusted Computing Base risk and improve assurance.
- Kernel API model verification
- Memory safety guarantees via static analysis
- Proofs for capability propagation correctness

Enhanced Hardening Against Attacks

- Additional work on increasing robustness of the Secure OS and its communication mechanisms against advanced offensive threats.
- Side-channel mitigation techniques (cache partitioning, temporal fuzzing, constant-time algorithms)
- Memory fault injection resilience
- Kernel fuzzing and semi-automated stress testing
- System defenses against speculative execution and timing inference

Заключение

References

Appendices

Sample TA Code

TA Manifests

Secure OS Code

Secure Standard Library Code