Глава 3: Проектирование и реализация доверенной операционной системы

3.1. Соображения по выбору интерфейса

3.1.1. Базовое API ДСИ: Интерфейс взаимодействия между мирами

3.1.1.1. Введение в TEE API

Базовое API TEE (Trusted Execution Environment Internal Core API) определяет функции, которые клиенты нормального мира используют для взаимодействия с доверенными приложениями, работающими в Secure OS. Этот интерфейс должен предоставлять механизмы для установки соединения, обмена данными, вызова защищённых функций и управления жизненным циклом безопасных сессий.

Рассмотрим варианты, на которые можно обратить внимание при выборе интерфейса для доверенной операционной системы.

3.1.1.2. OP-TEE для RISC-V

OP-TEE (Open Portable TEE) - это известная среда доверенного исполнения с открытым исходным кодом, изначально разработанная для технологии Arm TrustZone. OP-TEE реализует значительную часть спецификаций GlobalPlatform TEE, предоставляя богатый набор функций, включая внутренний API ядра TEE (TEE Internal Core API) для разработки TA и полнофункциональный клиентский API TEE.

Хотя OP-TEE представляет собой зрелое и широко распространённое решение TEE, его архитектура сопряжена с определёнными особенностями. OP-TEE сама по себе является небольшой безопасной операционной системой, обычно работающей параллельно с Rich OS. Для данного проекта, прямое использование API OP-TEE потребовало бы либо значительных усилий по портированию и адаптации, либо диктовало бы архитектурные решения, менее соответствующие целям проекта по созданию минимальной, глубоко проработанной доверенной кодовой базы (TCB), адаптированной под World Guard.

3.1.1.3. Разработка собственного минимального интерфейса на базе GlobalPlatform TEE

Альтернативным подходом рассматривалась разработка собственного, минимально необходимого интерфейса TEE, основанного на спецификациях GlobalPlatform TEE Internal Core API. Этот путь предполагает реализацию только ключевых функций API, необходимых для базового взаимодействия с TA: управление контекстами, сессиями, разделяемой памятью и вызовом команд.

Разработка собственного минимального интерфейса имела несколько преимуществ:

- Снижение сложности:
  Меньший набор функций API упрощает реализацию Secure OS, что потенциально ведёт к уменьшению доверенной кодовой базы (TCB) и поверхности атак.

- Адаптация под World Guard:
  Интерфейс мог бы быть специально спроектирован для эффективного использования двухмировой модели расширения World Guard и его примитивов межмирового взаимодействия. Например, парадигма обмена данными через разделяемую память, предусмотренная проектом, может быть напрямую отражена в дизайне API.

- Согласованность с архитектурой Безопасной ОС:
  Интерфейс мог бы быть оптимально интегрирован с микроядерной архитектурой Secure OS и её handle-based моделью безопасности.

- Контролируемый объём работ:
  такой подход позволяет сфокусировать усилия на реализации ключевых функций TEE и их взаимодействии с World Guard.

Основной проблемой этого подхода является обеспечение соответствия стандартам, даже для подмножества функций, и предоставление достаточной функциональности для создания значимых TA.

3.1.1.4. Экспериментальные и исследовательские прототипы

В академическом и исследовательском сообществе существуют различные архитектуры TEE и механизмы межмирового взаимодействия, часто фокусируясь на новых аппаратных возможностях или специфических свойствах безопасности. Некоторые прототипы исследуют собственные механизмы RPC, отличающиеся модели разделения памяти или совершенно новые парадигмы API, отличные от GlobalPlatform.

Хотя такие исследовательские прототипы предлагают ценные идеи для потенциальных улучшений TEE и специализированных решений безопасности, им часто не хватает стандартизации и широкой поддержки со стороны экосистемы, что критично для разработки переносимых TA.

Внедрение полностью экспериментального API отклонилось бы от признанных в индустрии стандартов, повысило бы порог вхождения для потенциальных разработчиков и ограничило бы совместимость.

3.1.1.5. Обоснование выбора подмножества API на базе GlobalPlatform

После оценки альтернатив было принято решение использовать тщательно подобранное подмножество GlobalPlatform TEE Internal Core API для межмирового взаимодействия. Этот подход призван сбалансировать стремление к стандартизации со специфическими требованиями и ограничениями проекта.

Выбор стоится на следущих причинах:

- Стандартизация и привычность:
  Спецификации GlobalPlatform TEE Internal Core API предоставляют хорошо определённый, признанный в индустрии стандарт для взаимодействия между клиентом TEE и TA. Использование подмножества этого стандарта позволяет опереться на существующую базу знаний, облегчая разработчикам TA, знакомым с концепциями TEE, понимание и использование системы.

- Ключевая функциональность:
  GP TEE API определяет такие основополагающие концепции, как контексты,

сессии, разделяемая память и вызов команд, которые являются фундаментальными для большинства взаимодействий с TEE. Реализация подмножества, содержащего эти ключевые элементы, обеспечивает достаточную функциональность для широкого круга Доверенных Приложений.

- Уменьшение трудоёмкости реализации и ДКБ:
  Выбирая только необходимые вызовы API, значительно снижается сложность реализации в Secure OS по сравнению с полной имплементацией GP. Это способствует уменьшению TCB, что критично для безопасности.

- Возможность будущего расширения:
  Начиная с минимального подмножества, фреймворк GP предлагает чёткий путь для потенциальных будущих расширений, если потребуется больше функций. Модульная природа спецификаций GP позволяет добавлять функциональность постепенно.

Выбранное подмножество включает функции: - TEEC_InitializeContext - TEEC_FinalizeContext - TEEC_OpenSession - TEEC_CloseSession - TEEC_InvokeCommand - TEEC_AllocateSharedMemory - TEEC_ReleaseSharedMemory

Обеспечивая базовые компоненты для безопасного взаимодействия и вычислений, минимизируя при этом интерфейс, предоставляемый Secure OS.

3.2. Обзор архитектуры системы

3.2.1. Высокоуровневая архитектура

3.2.1.1. Архитектурные уровни

Архитектура системы имеет четкую уровневую структуру, где каждый уровень выполняет специфические функции и работает на своем уровне привилегий. В основе системы лежит аппаратная платформа RISC-V, инициализацию которой осуществляет OpenSBI (Open Supervisor Binary Interface), выполняющий функции SEE. OpenSBI предоставляет необходимую низкоуровневую аппаратную абстракцию, выполняет начальную конфигурацию безопасности и управляет процессом передачи управления операционным системам.

Над уровнем встроенного ПО функционирует Secure OS, работающая в выделенном безопасном мире (Secure World). Эта функциональность обеспечивается расширением RISC-V World Guard. Данное микроядро разработано с минимальной TCB. Его основные задачи включают изолированное выполнение доверенных приложений, управление защищенными ресурсами и принудительное применение политик безопасности системы.

Параллельно, в отдельном домене, работает ОС нормального мира - это полнофункциональная операционная система. Она предоставляет среду для выполнения приложений общего назначения и взаимодействия с пользователем.

3.2.1.2. Обзор безопасного и нормального миров

Система использует статическое распределение ресурсов центрального процессора для поддержания строгого разделения между доменами безопасности. Secure OS постоянно находится в памяти и исполняется исключительно на первом ядре ЦП. Это ядро

полностью выделено для выполнения защищенных операций и доверенных приложений.

Все остальные ядра ЦП отводятся нормальному миру, в котором функционирует операционная система Linux. Нормальный мир предоставляет многофункциональную среду для вычислений общего назначения, однако с точки зрения безопасного ира он считается недоверенным.

Основные роли и обязанности четко разграничены:

1. Безопасный мир (Secure OS):

- Предоставляет изолированную среду исполнения для доверенных приложений.
- Управляет защищенными системными ресурсами, такими как аппаратное обеспечение или защищенные области памяти.
- Применяет capability-based модель безопасности к ТА.
- Обрабатывает критически важные операции и сервисы, запрашиваемые нормальным миром через строго определенный интерфейс.

1. Нормальный мир (Linux):

- Выполняет приложения общего назначения и управляет недоверенными системными ресурсами.
- Взаимодействует с безопасным миром исключительно через предопределенный интерфейс для запроса к защищенным сервисам.
- Не имеет прямого доступа к памяти или ресурсам, принадлежащим безопасному миру, если это явно не разрешено и не опосредовано Secure OS.

Контроль соблюдения границ обеспечивается комбинацией аппаратных и программных механизмов:

- Аппаратное расширение RISC-V World Guard обеспечивает принудительную изоляцию на аппаратном уровне, предотвращая прямой доступ нормального мира к областям памяти или периферийным устройствам, назначенным безопасному миру.

- Обмен данными между мирами (inter-world communication) строго контролируется через специализированные очереди в разделяемой памяти (для запросов и ответов) и межпроцессорные прерывания (IPI) для сигнализации. Этот контролируемый канал гарантирует, что все взаимодействия могут быть проверены.

3.2.2. Ключевые компоненты системы

3.2.2.1. Ядро, менеджеры ресурсов и сервисы TEE

Архитектура Secure OS построена на базе микроядра, исполняющегося на выделенном для него процессорном ядре. Микроядро предоставляет базовые сервисы, необходимые для функционирования Доверенной Среды Исполнения. К ним относятся безопасное переключение контекста между Доверенными Приложениями и внутренними задачами ядра, обработка прерываний, специфичных для Защищенного Мира, а также реализация механизмов capability-based модели безопасности. Микроядро отвечает за управление низкоуровневыми аппаратными ресурсами, выделенными для Защищенного Мира.

Поверх микроядра функционируют несколько ключевых менеджеров ресурсов:

- Менеджер задач (Task Manager):
  Контролирует жизненный цикл ТА и внутренних задач Защищенной ОС. Он отвечает за их создание (инициируемое Корневой Задачей (Root Task) по запросам из Обычного Мира), планирование и завершение. Каждое ДП, а также другие управляемые сущности, такие как потоки (threads), представляются в виде объектов ядра, доступ к которым и управление которыми осуществляется через дескрипторы в соответствии с capability-based моделью.

- Менеджер памяти:
  Управляет выделением, освобождением и защитой памяти в пределах Защищенного Мира. Это включает память для самого ядра, для отдельных ДП и для буферов, используемых при межмировом взаимодействии. Он использует базовые аппаратные механизмы, такие как PMP совместно с расширением World Guard, для обеспечения строгой изоляции памяти между различными ДП, а также между Защищенным и Обычным Мирами. Ресурсы памяти управляются как Объекты Виртуальной Памяти (VMO), доступ к которым также осуществляется через дескрипторы.

- Менеджер IPC (Inter-Process Communication Manager):
  Обеспечивает безопасные каналы взаимодействия внутри Защищенного Мира, например, между различными ДП или между ДП и сервисами Защищенной ОС. Это достигается с использованием объектов ядра, таких как каналы, где доступ и операции контролируются через capability, представленные дескрипторами. Менеджер IPC также поддерживает Root task в ее роли по ретрансляции сообщений в Обычный Мир и из него.

Сервисы TEE реализованы поверх ядра и менеджеров ресурсов. Сервисы предоставляют функциональность Защищенной ОС для ДП через определенный API, определяемый самими сервисами. Сервисы включают управление сессиями ДП, обработку вызовов команд из Обычного Мира и управление криптографическими операциями.

3.2.2.2. Разделяемая память и взаимодействие на основе IPI

Взаимодействие между Обычным Миром (Linux) и Защищенной ОС опирается на два основных механизма: разделяемую память и Межпроцессорные Прерывания.

Основу для обмена данными составляют две предварительно выделенные страницы физической памяти. Одна страница предназначена для очереди запросов (Request Queue), позволяя Обычному Миру помещать в нее запросы для Защищенной ОС. Вторая страница служит очередью ответов (Response Queue), через которую Защищенная ОС передает результаты и уведомления обратно в Обычный Мир. Эти очереди реализуются как lock-free обеспечения эффективной передачи данных и снижения конкуренции за доступ.

Доступ к этим областям разделяемой памяти осуществляется только в соответствии с заданной системной политикой безопасности: Обычный Мир имеет право записи в очередь запросов, из которой читает Защищенная ОС, а Защищенная ОС - право записи в очередь ответов, из которой читает Обычный Мир.

Стандартные для RISC-V (IPI) используются в качестве механизма сигнализации для уведомления другой стороны о наличии сообщений в разделяемых очередях.

- Сигнализация из Обычного Мира в Защищенный:
  Когда Linux (работающий на ядре Обычного Мира) помещает новый запрос в разделяемую очередь запросов, он затем отправляет IPI на выделенное ядро Защищенного Мира (CPU0), уведомляя Защищенную ОС о доступности нового запроса для обработки.

- Сигнализация из Защищенного Мира в Обычный:
  После того как Защищенная ОС обработала запрос и поместила ответ в разделяемую очередь ответов, она не отправляет IPI, обратная передача запроса реализована на базе поллинга со стороны Linux драйвера.

Такое сочетание явно определенных областей разделяемой памяти для передачи данных и IPI для нотификации о событиях формирует надежный и эффективный канал связи.

3.2.3. Схема распределения памяти и адресация

3.2.3.1. Физическая и виртуальная адресация

Secure OS формирует и обслуживает свой собственный, отдельный набор таблиц страниц, который управляет трансляцией виртуальных адресов в физические в пределах Безопасного Мира. В процессе инициализации Secure OS настраивает регистр SATP (Supervisor Address Translation and Protection) для выделенного безопасного процессорного ядра таким образом, чтобы он указывал на ее корневую таблицу страниц. Это активирует виртуальную адресацию для ядра Secure OS, Доверенных Приложений и любых защищенных периферийных устройств, назначенных Безопасному Миру.

Каждое ДП работает в своем собственном изолированном виртуальном адресном пространстве, управляемом Secure OS. Это гарантирует, что ДП не могут напрямую обращаться к памяти друг друга или к памяти ядра, если это явно не разрешено через мандаты объектов разделяемой памяти.

Физическая память секционируется ОС с использованием расширения World Guard. Определенные области физической памяти назначаются для эксклюзивного использования Безопасным Миром, другие выделяются Нормальному Миру, а некоторые, специально ограниченные области, определяются как разделяемые.

Следовательно, отображения памяти в Безопасном Мире (устанавливаемые Secure OS) предоставляют доступ к защищенным областям физической памяти, которые невидимы, для конфигурации MMU из Нормального Мира. MMU Нормального Мира функционирует независимо на других процессорных ядрах под управлением Linux.

3.2.3.2. Механизмы изоляции

Основным аппаратным механизмом обеспечения изоляции памяти между Безопасным и Нормальным Мирами является расширение RISC-V World Guard. World Guard связывает идентификатор мира (world ID) с транзакциями памяти, исходящими от процессорных ядер, и с обращениями к периферийным устройствам.

Secure OS в процессе своей загрузки и при дальнейшей работе на выделенном безопасном ядре конфигурирует аппаратные контроллеры World Guard. Эти контроллеры программируются набором правил, которые определяют, какие области

физической памяти и периферийные устройства доступны для какого идентификатора мира.

Например, страницы памяти, выделенные для ядра Secure OS, Доверенных Приложений и приватных данных Безопасного Мира, конфигурируются таким образом, чтобы быть доступными только в том случае, если транзакция исходит от процессорного ядра, работающего в Безопасном Мире (т.е. обладающего идентификатором Безопасного Мира).

Если операционная система Нормального Мира, функционирующая на других ядрах с идентификатором Нормального Мира, попытается получить доступ к странице физической памяти, эксклюзивно назначенной Безопасному Миру, аппаратные контроллеры World Guard обнаружат это нарушение политики. Подобные попытки несанкционированного доступа приводят к аппаратному исключению, которая обрабатывается в соответствии с конфигурацией механизма сообщений об ошибках World Guard. Это эффективно предотвращает прямое чтение, запись или исполнение кода Нормальным Миром в пределах защищенных страниц Безопасного Мира.

3.2.3.3. Очереди в разделяемой памяти

Взаимодействие между Безопасным и Нормальным Мирами осуществляется через зарезервированные области физической памяти, сконфигурированные как разделяемые буферы. В частности, используются две выделенные страницы физической памяти, выровненные по границе размера страницы: одна для запросов от Нормального Мира к Безопасной ОС (очередь запросов), а другая — для ответов от Безопасной ОС обратно в Нормальный Мир (очередь ответов).

Безопасная ОС, совместно с конфигурациями контроллеров World Guard, предоставляет обоим мирам соответствующие права доступа к этим специальным разделяемым страницам.

Данные очереди реализованы в виде lock-free кольцевых буферов, что позволяет управлять конкурентным доступом без накладных расходов, свойственных традиционным механизмам блокировок. Использование lock-free алгоритмов минимизирует уровень конкуренции за ресурсы и сокращает задержки, особенно с учетом того, что Нормальный Мир может помещать запросы в очередь из различных контекстов исполнения (хотя они и обрабатываются последовательно одноядерной Безопасной ОС).

Фиксированный размер и заранее определенное расположение этих очередей упрощают управление памятью для межмирового обмена данными.

3.2.4. Процесс исполнения Безопасной ОС

3.2.4.1. Обзор процесса загрузки Secure OS

Процесс загрузки начинается с OpenSBI, который выполняет начальную настройку платформы. OpenSBI идентифицирует и загружает Безопасную ОС, передавая ей управление на выделенном первом ядре ЦП (ядре 0). Затем Безопасная ОС инициализирует свои ключевые компоненты, включая подсистему управления памятью и настраивает расширение World Guard для изоляции ядра 0 в качестве Безопасного Мира.

Это включает настройку контроллеров WorldGuard для защиты памяти Безопасного Мира. После своей инициализации Secure OS возвращается обратно OpenSBI. После этого OpenSBI приступает к загрузке ОС Обычного Мира (Linux) на остальных ядрах ЦП. Эти ядра настраиваются OpenSBI

3.2.4.2. Переходы между мирами

Переходы исполнения между Безопасным и Обычным Мирами строго контролируются. Когда Обычному Миру (Linux) необходимо вызвать сервис из Безопасной ОС, он подготавливает запрос в выделенной очереди запросов в разделяемой памяти, а затем, инициирует межпроцессорное прерывание (IPI) на ядро 0. Это IPI служит сигналом для Безопасной ОС о необходимости проверить очередь запросов.

Безопасная ОС обрабатывает запрос и помещает ответ в очередь ответов в разделяемой памяти. Обычный Мир, опрашивает очередь ответов для получения результата.

Перед обработкой любого запроса из Обычного Мира Безопасная ОС выполняет проверки достоверности на основе содержимого и источника сообщения. Сохранность состояния Безопасного Мира во время переходов обеспечивается расширением World Guard, которое управляет аппаратным контекстом для текущего мира, исполняющегося на ядре. Безопасная ОС отвечает за сохранение и восстановление контекстов конкретных Доверенных Приложений (TA) в Безопасном Мире, если под её управлением находится несколько TA или сервисов.

3.2.4.3. Планирование в Безопасной ОС

Безопасная ОС функционирует на единственном выделенном ядре ЦП (ядре 0) и использует однопроцессорную стратегию планирования для управления своими внутренними задачами и любыми размещаемыми ею Доверенными Приложениями (TA).

Реализована вытесняющая, round-robin политика планирования для обеспечения справедливого доступа к ЦП для всех готовых к выполнению сущностей в Безопасном Мире. Переключение контекста в Безопасной ОС включает сохранение контекста исполнения (регистров, указателя стека, счетчика команд) текущей TA или сервиса и восстановление контекста следующей запланированной сущности.

Учитывая, что Безопасная ОС и её TA исполняются исключительно на ядре 0, их деятельность по планированию полностью независима и не конфликтует с планировщиком ядра Linux, который управляет задачами на других ядрах ЦП, работающих в Обычном Мире. Архитектура нацелена на то, чтобы операции Безопасней ОС были относительно кратковременными для обеспечения своевременных ответов Доверенным Приложениям и запросам из Обычного Мира.

3.2.5. Обеспечение безопасности и реализация политик

3.2.5.1. Модель безопасности

Безопасная ОС применяет многоуровневый подход к обеспечению безопасности и реализации политик, сочетая механизмы изоляции на аппаратном уровне, предоставляемые расширением RISC-V World Guard, с программно-определяемой моделью гранулярного контроля доступа. Такой подход гарантирует защиту как Безопасного Мира, так и работающих в нем Доверенных Приложений от недоверенного

Нормального Мира, и, что особенно важно, друг от друга, в соответствии с принципом наименьших привилегий.

В Безопасной ОС реализована мандатно-дискреционная модель безопасности, где доступ ко всем системным ресурсам и сервисам ядра опосредован через дескрипторы, или же хендлы (handles).

Дескриптор выступает в роли неподделываемого токена, представляющего определенное полномочие (capability) - право на выполнение заданного набора операций над связанным объектом ядра (например, областями памяти, каналами связи, блоками управления задачами). Эта модель отличается от традиционного контроля доступа, фокусируясь на конкретных разрешениях, привязанных к экземплярам объектов, а не на общих привилегиях пользователя.

Изначально Доверенные Приложения получают набор дескрипторов через предопределенный Манифест, предоставляемый root таской системы при запуске ДП. Новые объекты и, соответственно, новые дескрипторы с конкретными полномочиями, создаются путем вызова методов объекта-фабрики, для чего ДП также должно обладать соответствующим дескриптором.

Операции над этими объектами запрашиваются через безопасные системные вызовы (secure syscalls), которые являются основным интерфейсом между ДП и ядром. Предъявляя дескриптор при системном вызове, ДП неявно подтверждает свое право на выполнение запрошенной операции. Этот механизм обеспечивает детальный контроль доступа, поскольку каждый дескриптор явно определяет допустимые взаимодействия с соответствующим ему объектом.

Центральное место в этой модели занимает контроль соблюдения разрешений на этапе выполнения. При вызове системного вызова ядро выполняет строгую проверку. Сначала оно проверяет аутентичность и валидность предъявленного дескриптора, убеждаясь, что он соответствует активному объекту ядра и может использоваться вызывающим ДП в его текущем контексте. Затем ядро проверяет, позволяет ли полномочие, инкапсулированное в дескрипторе, выполнить запрошенную операцию над целевым объектом. Если полномочия недостаточны или дескриптор недействителен, операция отклоняется, и ДП получает сообщение об ошибке. Такая проверка при каждой операции критически важна для предотвращения подделки полномочий ДП или эскалации привилегий сверх явно предоставленных. Поскольку ДП не могут напрямую манипулировать содержимым дескрипторов или произвольно создавать новые полномочия, они ограничены теми правами, которые были им предоставлены через первоначальный Манифест или законно получены от объекта-фабрики, что надежно обеспечивает соблюдение принципа наименьших привилегий.

3.2.5.2. Интеграция с World Guard

Программно-определяемая мандатно-дискреционная модель безопасности Безопасной ОС дополняет аппаратную изоляцию, обеспечивая детальный контроль доступа внутри Безопасного Мира.

В то время как World Guard предотвращает несанкционированный доступ Нормального Мира к ресурсам Безопасного Мира и вмешательство в его работу, мандатно-дискреционная модель тщательно регулирует взаимодействие ДП и компонентов Безопасной ОС с объектами и ресурсами, управляемыми ядром.

Такой многоуровневый подход создает стратегию эшелонированной обороны (defense-

in-depth): World Guard устанавливает надежный межмировой периметр, а мандатно-дискреционная система обеспечивает строгую компартментализацию и соблюдение политик внутри этого периметра.

Когда аппаратный контроллер (checker) WorldGuard обнаруживает попытку несанкционированного доступа к памяти (например, код из NW пытается прочитать приватную память SW, или же наоборот) или недействительный переход между мирами (например, некорректная инструкция пытается незаконно изменить текущее состояние мира для данного ядра), он генерирует аппаратное исключение (fault). Реакция системы на такое событие определяется обработчиками, сконфигурированными Безопасной ОС.

Нарушения, исходящие из Нормального Мира, приводят к прерыванию на ядре Нормального Мира, вызвавшем нарушение. Специфическая обработка (например, сигнализация об ошибке ядру Linux) позволяет Нормальному Миру управлять сбоем в своем собственном контексте, не компрометируя Безопасный Мир.

Если программный компонент внутри самого Безопасного Мира спровоцирует нарушение World Guard это также приведет к прерыванию, обрабатываемому Безопасной ОС, что вызоыет завершение сбойного ДП или сервиса, или контролируемую остановку системы, если нарушение произошло из ядра Secure OS.

Таким образом, World Guard выступает как механизм принудительного применения политики разделения миров, с определенными путями обработки сбоев при нарушениях этих аппаратно-обеспечиваемых правил. Механизмы сообщения об ошибках гарантируют, что такие нарушения могут быть зарегистрированы.

3.2.6. Жизненный цикл доверенных приложений

3.2.6.1. Создание TA

Жизненный цикл Доверенного Приложения (ДП) в Безопасной ОС инициируется поступлением запроса TEEC_OpenSession из Обычного Мира. Обработкой данного запроса занимается Root Task, ответственная за управление созданием экземпляров ДП.

Процесс создания ДП включает несколько этапов, координируемых Корневой Задачей совместно с основными службами Безопасной ОС:

1. Загрузка образа ДП:
   Безопасная ОС загружает двоичный образ ДП (ELF-файла, указанный в предопределенном манифесте и размещенный в файловой системе Secure OS) в выделенное, изолированное адресное пространство. Это включает синтаксический анализ исполняемого формата и отображение его сегментов кода, данных и BSS в виртуальное адресное пространство ДП.

2. Выделение памяти:
   Специализированные области памяти выделяются для стека, кучи и приватных данных ДП. Эта память защищена и изолирована от других ДП и самого ядра Безопасной ОС с использованием таких механизмов, как PMP и MMU.

3. Инициализация дескрипторов:
   На основе манифеста ДП Корневая Задача предоставляет доверенному приложению начальный набор дескрипторов.

4. Создание процесса/потока:
   В рамках Безопасной ОС для размещения ДП создается новая структура процесса (task). В этом процессе порождается начальный поток, с точкой входа, установленной на соответствующую стартовую процедуру ДП (TA_CreateEntryPoint).

5. Начальное состояние:
   После успешного создания и инициализации процесс ДП переводится в состояние, в котором он уже выполнил свою процедуру TA_CreateEntryPoint, провел необходимую самоинициализацию и готов к приему команд, но изначально неактивен.

Вызов TEEC_OpenSession возвращает идентификатор сессии клиенту в Обычном Мире в случае успешного создания ДП и установления сессии. Этот идентификатор сессии используется для последующих взаимодействий с данным экземпляром ДП.

3.2.6.2. Исполнение ТА

После того как сессия с ДП успешно установлена (через TEEC_OpenSession) и его процедура TA_CreateEntryPoint завершила свою работу, ДП переходит в состояние ожидания и уступает управление планировщику Безопасной ОС. ТА остается неактивной, ожидая конкретных команд из Обычного Мира.

Дальнейшее взаимодействие осуществляется через вызовы TEEC_InvokeCommand, поступающие от клиента из Обычного Мира. При поступлении такого вызова происходит слудующая последовательность действий:

1. Активация и диспетчеризация:
   Root task получает запрос InvokeCommand. Она идентифицирует целевую сессию ДП и конкретный идентификатор исполняемой команды (ID метода). После этого поток ДП активируется (ставится в очередь на исполнение).

2. Передача аргументов:
   Параметры, связанные с InvokeCommand, включая идентификатор команды и любые буферы ввода/вывода, становятся доступными ДП. Безопасная ОС гарантирует, что ДП получает эти аргументы безопасным и определённым образом, через его контекст исполнения или заранее подготовленные участки памяти.

3. Исполнение ДП:
   ДП исполняет свою функцию TA_InvokeCommandEntryPoint. На основе полученного идентификатора команды ДП выполняет запрошенное вычисление. Это может включать: - Доступ к своим приватным данным и ресурсам. - Использование предоставленных ему дескрипторов для взаимодействия со службами Безопасной ОС (например, криптографическими операциями). - Обмен данными с другими ДП или Root task через защищённые каналы межпроцессного взаимодействия (IPC), используя дескрипторы, полученные при создании. - Запрос на выделение новых областей разделяемой памяти, если команда требует обмена большими объемами данных с Обычным Миром. Вызов API TEEC_AllocateSharedMemory, инициированный клиентом Обычного Мира, может быть связан с потребностями в памяти, выявленными во время выполнения TEEC_InvokeCommand. Обмен данными затем осуществляется клиентом Обычного Мира и ДП путем записи и чтения из этой разделяемой памяти.

По завершении выполнения команды ДП подготавливает любые результаты (в выходных буферах или путем модификации разделяемой памяти) и возвращает код состояния. Затем Безопасная ОС передаёт этот статус и любые релевантные данные обратно клиенту Обычного Мира в качестве ответа на TEEC_InvokeCommand. После этого ДП может снова уступить управление, ожидая дальнейших команд.

3.2.6.3. Завершение работы ТА

Завершение работы экземпляра Доверенного Приложения и освобождение занимаемых им ресурсов инициируются, когда клиентское приложение Обычного Мира явным образом закрывает сессию с помощью вызова TEEC_CloseSession. Этот запрос сигнализирует о том, что сервисы ДП для данной конкретной сессии более не требуются.

Процесс завершения включает следующие ключевые этапы, управляемые Безопасной ОС, при содействии и с уведомлением Корневой Задачи:

1. Уведомление о закрытии сессии:
   ДП уведомляется о закрытии своей сессии, путем вызова его функции TA_CloseSessionEntryPoint. Это позволяет ДП выполнить специфичную для сессии очистку, например, освободить ресурсы, выделенные конкретно для этой сессии.

2. Уничтожение ДП:
   Если с экземпляром ДП не связано других активных сессий (или если ДП предназначено для работы в режиме одной сессии), Безопасная ОС приступает к полному прекращению работы ДП путем вызова его функции TA_DestroyEntryPoint. Эта функция предоставляет ДП последнюю возможность выполнить глобальную очистку.

3. Освобождение ресурсов: Освобождаются следующие типы ресурсов: - Освобождение объектов:
   Все дескрипторы, принадлежащие ДП, аннулируются и освобождаются. На соответствующие объекты ядра, связанные с этими дескрипторами (например, объекты памяти, каналы связи) снимаются ссылки. Если ДП было последним держателем ссылки, сам объект удаляется соответствующим менеджером ресурсов Безопасной ОС. - Освобождение разделяемой памяти:
   Любые области разделяемой памяти, явно выделенные для связи между данным экземпляром ДП и Обычным Миром (например, через TEEC_AllocateSharedMemory) освобождаются: отменяется их отображение (unmap) из адресного пространства ДП в Безопасном Мире и помечаются для последующего освобождения. Драйвер Linux на стороне Обычного Мира, как правило, также отвечает за отмену отображения этих регионов. - Освобождение приватной памяти:
   Приватные области памяти, выделенные для кода, данных, стека и кучи ДП, возвращаются менеджеру памяти Безопасной ОС. Структура задачи/процесса ДП и связанные с ней потоки уничтожаются.

После успешного выполнения этих шагов экземпляр ДП прекращает свое существование, а его ресурсы становятся доступными для повторного выделения.

Код результата, указывающий на успех или неудачу операции TEEC_CloseSession, возвращается клиенту Обычного Мира. Безопасная ОС гарантирует, что процесс завершения работы является упорядоченным и не нарушает целостность Безопасного Мира или других активных ДП.

3.3. Интеграция WorldGuard

3.3.1. Конфигурирование WorldGuard

3.3.1.1. Конфигурация миров (двухмировая модель)

Фундаментом архитектуры безопасности системы является разделение аппаратных и программных ресурсов на два принципиально различных мира: Защищённый мир и Обычный мир. Такое разделение реализуется с помощью расширения WorldGuard.

- Разделение системных ресурсов:
  Расширение WorldGuard обеспечивает фундаментальное разделение системных ресурсов. Физическая память разделяется на области, исключительно доступные Защищённому миру, области, исключительно доступные Обычному миру, и специальные, ограниченные области, выделенные под разделяемую память для межмирового взаимодействия. Одно ядро ЦП статически назначается Защищённому миру для выполнения Безопасной ОС и её Доверенных приложений. Все остальные ядра ЦП отводятся Обычному миру для работы Linux.

- Обоснование двухмировой архитектуры: <br Такая двухмировая модель обеспечивает строгую, аппаратно усиленную изоляцию между доверенной средой (Защищённый мир) и средой общего назначения, потенциально подверженной компрометации (Обычный мир). Этот архитектурный выбор продиктован проверенными концепциями ДСИ и предлагает чёткое разделение окружений. Он упрощает Доверенную вычислительную базу, выделяя отдельный мир для критически важных с точки зрения безопасности операций, отделяя их от сложностей многофункциональной операционной среды.

- Роли каждого мира:

  - Защищённый мир (ЗМ):
    Работает с идентификатором мира 0 (WID 0). В нём размещается Безопасная ОС, отвечающая за выполнение Доверенных приложений, управление криптографическими ключами и другими секретами, выполнение чувствительных к безопасности вычислений и реализацию мандатной модели безопасности. Все операции в Защищённом мире рассматриваются как привилегированные с точки зрения безопасности.
  - Обычный мир (ОМ):
    Работает с идентификатором мира 1 (WID 1). В нём размещаются Универсальная ОС (Linux) и её приложения. Обычный мир рассматривается Защищённым миром как недоверенный. Он взаимодействует с Защищённым миром только через строго определённый интерфейс, как правило, для запроса служб у ДП.

- Назначение и управление идентификаторами миров (World ID):
  WorldGuard связывает идентификатор мира (WID) с транзакциями, исходящими от ядер ЦП. В данной системе WID 0 назначается Защищённому миру, а WID 1 — Обычному миру. Назначение WID ядрам ЦП выполняется OpenSBI на ранних этапах загрузки, используя его привилегии M-режима. OpenSBI настраивает ядро ЦП 0 для работы с WID 0 (Защищённый мир) перед передачей управления Безопасной ОС. Остальные ядра ЦП настраиваются с WID 1 (Обычный мир) перед загрузкой на них Linux. Безопасная ОС, после инициализации и запуска в Защищённом мире на ядре 0, работает под WID 0. Она гарантирует, что её

обращения к памяти и, следовательно, обращения управляемых ею ДП, корректно ассоциированы с этим WID. CSR-регистр mlwid (Machine Lower-privilege World ID) используется M-режимом (OpenSBI) для установки WID для S-режима (в котором работает Безопасная ОС). Это статическое назначение WID формирует основу аппаратной изоляции.

3.3.1.2. Конфигурирование контроллеров WorldGuard

Контроллеры WorldGuard (также называемые чекерами) - это аппаратные блоки, интегрированные в коммутационную матрицу памяти системы. Они отслеживают транзакции памяти, сопоставляют WID инициатора с заранее запрограммированными правилами для областей памяти и принудительно применяют разрешения на доступ. Безопасная ОС отвечает за настройку этих контроллеров на этапе своей инициализации для установления и поддержания надёжной изоляции.

Конфигурирование защищённой RAM с использованием слотов WG:

- Подход к разделению защищённой памяти:
  Физическое адресное пространство разделяется путём программирования слотов контроллера WorldGuard. Каждый слот определяет правило для конкретного диапазона памяти, связывая его с разрешениями для различных WID. Безопасная ОС выделяет отдельные регионы для собственного кода и данных ядра, приватной памяти каждого ДП и страниц разделяемой памяти, используемых для межмирового взаимодействия. Эти разделения транслируются в правила для слотов контроллера WG. Такое разделение статично и устанавливается во время загрузки.

- Блокировка регионов памяти для Защищённого мира:
  Области памяти, предназначенные для эксклюзивного использования Защищённым миром (например, образ Безопасной ОС, приватные стеки и кучи ДП), настраиваются в слотах контроллера WG. Эти слоты указывают, что только транзакции, инициированные WID 0 (Защищённый мир), имеют разрешение на доступ для чтения и/или записи. Любая попытка доступа к этим регионам со стороны WID 1 (Обычный мир) перехватывается контроллером WG, что приводит к исключению. Для обеспечения неизменности этих критически важных конфигураций безопасности бит L (lock - блокировка) в конфигурационном регистре (cfg) каждого слота контроллера WG устанавливается Безопасной ОС после инициализации. После блокировки конфигурация слота не может быть изменена до следующего аппаратного сброса системы.

- Установка границ разделов:
  Основной раздел, принудительно устанавливаемый WorldGuard - это раздел между Защищённым и Обычным мирами. Эта граница определяется совокупностью правил, запрограммированных в слотах контроллера WG.

  - Память Обычного мира:
    Регионы, предназначенные для использования Обычным миром (например, основная RAM для Linux), настраиваются так, чтобы разрешать чтение/запись для WID 1 и запрещать доступ для WID 0. Это предотвращает непреднамеренный или злонамеренный доступ Защищённого мира к приватным данным Обычного мира, следуя принципу наименьших привилегий.
  - Регионы разделяемой памяти:
    Для межмирового взаимодействия определённые страницы (например, для

очередей запросов и ответов) настраиваются в слотах WG так, чтобы быть доступными как для WID 0, так и для WID 1. Однако разрешения могут быть асимметричными: для очереди запросов WID 1 может иметь право на запись, тогда как WID 0 (ЗМ) - право на чтение, и наоборот для очереди ответов.
    - Политика контроля доступа: Политика достаточно прямолинейна:
        - Доступ из REE к приватной памяти TEE строго запрещён (чтение, запись, выполнение).
        - Доступ из TEE к приватной памяти REE, запрещён для поддержания изоляции и предотвращения превращения TEE в вектор атак на REE.
        - Доступ к разделяемой памяти явно разрешён с тщательно определёнными правами.

- Интеграция атрибутов памяти (чтение/запись/выполнение) с проверками WorldGuard:
    - Контроллеры WorldGuard в основном обеспечивают соблюдение разрешений на основе типов доступа "чтение" и "запись" для указанных WID в определённых диапазонах памяти. Поле perm в каждом слоте контроллера напрямую кодирует эти разрешения R/W для каждого мира. Нарушение этих разрешений немедленно вызывает исключение, регистрируемый контроллером.
    - Выборка инструкций ядром ЦП рассматривается шинной структурой и, следовательно, контроллерами WorldGuard как операция чтения из памяти. Если WID лишён права на чтение определённой области памяти контроллером WG, ему также будет неявно отказано в праве на выборку инструкций (выполнение) из этой области на уровне WorldGuard.
    - Более гранулярные разрешения на выполнение (например, биты NX для страниц данных) внутри мира управляются Модулем Управления Памятью (MMU) этого мира. Например, Безопасная ОС использует свой MMU (настраиваемый через записи в таблицах страниц, PTE) для маркировки своих страниц данных как неисполняемых для ДП, даже если WorldGuard разрешает WID 0 читать эти страницы. Таким образом, WorldGuard обеспечивает крупнозернистую межмировую изоляцию тогда как MMU предоставляет более мелкозернистую внутримировую защиту и управление виртуальной памятью.

3.3.2. Интеграция доверенной операционной системы

3.3.2.1. Информирование об ошибках

Интеграция с Защищённой Операционной Системой в части обработки ошибок расширения WorldGuard включает немедленное обнаружение проблем. Аппаратные контроллеры WorldGuard выявляют попытки несанкционированного доступа или ошибки конфигурации, записывая в свои регистры errcause и erraddr подробности инцидента, такие как идентификатор мира-нарушителя (WID), тип доступа (чтение/запись) и целевой физический адрес.

При обнаружении нарушения, относящегося к домену Защищённого Мира или его ресурсам:

1. Обнаружение и уведомление о нарушении:
   Контроллер WorldGuard настроен с установленными битами IR (прерывание при

чтении) и IW (прерывание при записи) для областей памяти. Нарушение инициирует соответствующую реакцию. Прерывание, если будет направлено в Secure OS. Обработчик прерываний затем опрашивает регистры errcause и erraddr контроллера WorldGuard (доступные через отображение в память) для получения информации о сбое.

2. Структура информирования:
   Secure OS логгирует информацию о нарушении в консоль через UART драйвер:

   - Временная метка события.
   - Содержимое errcause: WID нарушителя, тип доступа (чтение/запись), был ли сигнализирован контроллером сбой шины (be) или прерывание (ip).
   - Содержимое erraddr: физический адрес попытки доступа.
   - Идентификатор компонента ЗОС или контекста Доверенного Приложения, если сбой произошёл внутри Защищённого Мира и его источник может быть надёжно определён.

3. Стратегии реагирования:

   - Нарушения со стороны Обычного Мира (ОМ): Если WID нарушителя указывает на попытку Обычного Мира получить несанкционированный доступ к памяти Защищённого Мира (ЗМ), CPU 0 ну будет проинформирован об этом, и со стороны Secure OS никаких действий произведено не будетб, поскольку нормальный мир сам отвечает за обработку сбоев на своих ядрах. Основное противодействие осуществляется аппаратными средствами WorldGuard, блокирующими доступ.
   - Нарушения со стороны ДП:
     Если ДП внутри Защищённого Мира инициирует нарушение WorldGuard (например, пытаясь получить доступ к памяти за пределами разрешённых ему областей). Secure OS в этом случае:

     - Регистрирует критическую ошибку.
     - Немедленно завершает работу ДП-нарушителя для предотвращения дальнейшей компрометации или нестабильности.
     - Высвобождает ресурсы, удерживаемые ДП.
     - Сообщает об ошибке клиенту Обычного Мира, у которого была активная сессия с этим ДП.

   - Нарушения со стороны ядра Защищённой ОС:
     Если нарушение WorldGuard связано с самим ядром Secure OS, это указывает на серьёзную внутреннюю несогласованность или ошибку. В этом случае Secue OS:

     - Попытается зарегистрировать диагностическую информацию, если это возможно.
     - Переходит в контролируемое состояние паники, прекращая дальнейшие операции в Защищённом Мире для предотвращения потенциального распространения небезопасного состояния.

3.3.2.2. Управление переходами между мирами

В реализованной двухмировой модели с WorldGuard, где Защищённая ОС статически привязана к ядру 0, а Linux - к остальным ядрам, под переходами между мирами в первую очередь понимается контролируемое взаимодействие и потоки сигналов между

этими статически разделёнными средами исполнения, а не динамические изменения WID на одном ядре для обычных вызовов.

1. Направление передачи данных ОМ -> ЗМ:

    - Драйвер Linux, подготавливает сообщение-запрос и помещает его в общую очередь запросов.
    - Затем Обычный Мир инициирует Межпроцессорное Прерывание (МПП), нацеленное на ядро 0. Это делается через вызов SBI (sbi_send_ipi).
    - Защищённая ОС, работающая на ядре 0, получает это МПП. Её обработчик МПП, затем проверяет очередь запросов, извлекает сообщение и передаёт его соответствующему ДП или внутренней службе.

2. Направление передачи данных ЗМ -> ОМ:

    - После обработки запроса Защищённая ОС или ДП помещает ответное сообщение в общую очередь ответов.
    - В текущей реализации ЗОС не отправляет МПП обратно в Обычный Мир. Вместо этого драйвер Linux в Обычном Мире использует механизм опроса через, выделенный поток ядра, который периодически проверяет очередь ответов на наличие готовых сообщений.

3. Обработка прерываний:

    - Основной переход, инициируемый прерыванием, - это МПП из ОМ в ЗМ, сигнализирующее о новом запросе.
    - Исключения по вине WorldGuard настроены на генерацию прерываний для ядра 0, также являются формой события, которое ЗОС обрабатывает для управления нарушениями безопасности.

3.3.2.3. Страницы обмена данными

1. Определение областей разделяемой памяти:
   Предварительно определены две основные области разделяемой памяти, каждая состоит из одной физической страницы, выровненных по границе 4 КиБ:

    - Страница очереди запросов:
      Используется Обычным Миром для отправки запросов в Защищённый Мир.
    - Страница очереди ответов:
      Используется Защищённым Миром для отправки ответов обратно в Обычный Мир.

2. Конфигурация WorldGuard для разделяемых страниц:
   Во время своей инициализации Защищённая ОС программирует контроллер WorldGuard для принудительного применения прав доступа для этих страниц обмена данными:

    - Права доступа для страницы очереди запросов:
      Конфигурируется слот в контроллере WorldGuard, соответствующий диапазону физических адресов очереди запросов. Поле perm для этого слота выставляется:

        - Обычный Мир (WID 1):
          право на запись и, на чтение, т.к. реализация очереди на стороне ОМ

этого требует для управления собственной метаинформацией.
    - Защищённый Мир (WID 0): право на чтение.
    - Права на выполнение запрещены для обоих миров.

- Права доступа для страницы очереди ответов:
  Конфигурируется слот, соответствующий диапазону физических адресов очереди ответов. Поле perm для этого слота выставляется:

    - Защищённый Мир (WID 0):
      право на запись и на чтение для управления своей очередью.
    - Обычный Мир (WID 1): право на чтение.
    - Права на выполнение запрещены для обоих миров.

3. Расположение и выравнивание:
   Разделяемые области являются физически смежными и выровнены по границам страниц (4 КиБ), чтобы соответствовать гранулярности управления памятью и упростить конфигурацию правил WorldGuard. Для предотвращения случайных переполнений, которые могут повредить смежную память, ЗОС настраивает канареечные страницы (недоступные защитные страницы-маркеры) вокруг этих разделяемых страниц обмена данными, используя дополнительные правила WorldGuard.

3.4. Процесс безопасной загрузки и инициализации

3.4.1. Ранняя инициализация Безопасной ОС

3.4.1.1. Передача управления от OpenSBI

OpenSBI, выступающий в роли среды исполнения уровня супервизора (Supervisor Execution Environment, SEE), инициирует процесс загрузки. OpenSBI загружает образ прошивки Безопасной ОС, указанный в FW_PAYLOAD_PATH, в физическую память на основном ядре процессора (hart 0), выделенном для выполнения безопасных операций. Затем управление передается точке входа Безопасной ОС - процедуре _start. OpenSBI передает Безопасной ОС ключевые системные параметры через регистры общего назначения: a0 содержит идентификатор загружаемого ядра (hart ID), а a1 - физический адрес дерева устройств (Flattened Device Tree, FDT).

Процедура _start, реализованная в файле head.S, получает эти параметры и сохраняет их для последующего использования на более высоких уровнях инициализации на языке C, в частности, функцией early_boot, которая применяет FDT для конфигурации платформы.

3.4.1.2. Настройка стека и базовой схемы памяти

Перед активацией MMU и созданием полноценной среды выполнения C, для ассемблерных подпрограмм и ранних вызовов функций C необходим начальный стек. Этот стек выделяется статически в образе памяти Безопасной ОС, а его размер определяется равным 6 физическим страницам.

Процедура _start вычисляет виртуальный адрес вершины этого стека. Сначала она определяет физическую вершину области стека (на основе символа stacks и STACK_SIZE), а затем прибавляет KERNEL_VIRTUAL_OFFSET (разницу между KERNEL_VIRTUAL_BASE и физическим адресом загрузки ядра), чтобы установить

указатель стека (sp) по его окончательному виртуальному адресу.

Базовый адрес стека выравнивается по границе страницы, что обеспечивает соблюдение требований RISC-V ABI к выравниванию указателя стека. На данном этапе ядро работает, исходя из информации о своем актуальном физическом расположении в памяти, определенном при загрузке OpenSBI.

3.4.1.3. Первичная релокация ядра

Безопасная ОС компилируется как позиционно-независимый исполняемый файл (PIE), что позволяет загружать её по произвольным физическим адресам, в то время как её внутренние компоненты компонуются для работы из верхнего диапазона виртуального адресного пространства, заданного параметром KERNEL_VIRTUAL_BASE. Для реализации этой возможности на начальном этапе настройки MMU выполняется первичная релокация путем создания отображения для сегментов кода и данных ядра в начальных таблицах страниц

Создается тождественное отображение (физический адрес равен виртуальному) для той области, куда физически загружено ядро. Это гарантирует непрерывность выборки инструкций с физического адреса загрузки ядра сразу после активации MMU.

3.4.1.4. Активация MMU

После того как таблицы страниц заполнены тождественным отображением и отображением на KERNEL_VIRTUAL_BASE для ядра, активируется модуль управления памятью (MMU). Это достигается путем конфигурирования регистра SATP (Supervisor Address Translation and Protection). Процедура включения MMU записывает в SATP:

1. Номер физической страницы (PPN) корневой таблицы страниц (полученный из физического адреса структуры ptable_cache).

2. Режим трансляции адресов (Sv39 для 64-разрядных систем RISC-V, что соответствует значению MODE = 8 в регистре SATP).

Сразу после записи в SATP трансляция адресов становится активной для всех обращений к памяти из режима супервизора на данном ядре. Предварительно созданные отображения гарантируют непрерывность выполнения текущего кода.

3.4.2. Инициализация Безопасной ОС

После перехода в среду C, завершив начальную настройку на уровне ассемблера и активацию MMU, Безопасная ОС запускает процедуру инициализации для запуска своих ключевых компонентов.

3.4.2.1. Регистрация консоли

Первым шагом в среде C становится настройка канала вывода диагностической информации и журналов.

Драйвер консоли, использующий сервисы консоли SBI инициализируется и регистрируется в качестве основного канала ввода-вывода посредством вызова sbi_register_console().

Это позволяет использовать функции вывода на ранних этапах работы в S-режиме, что важно для отладки и информирования о ходе или ошибках на последующих этапах инициализации.

3.4.2.2. Инициализация таблиц страниц

Хотя базовые отображения MMU уже активны, Безопасная ОС теперь детализирует свою схему распределения памяти, настраивая таблицы страниц более гранулярно.

Основные структуры таблиц страниц ядра доинициализируются функцией mmu_init_kernel_page_table(), выходя за рамки первоначальных крупных, грубо гранулированных отображений, созданных в head.S.

Затем менеджер виртуальной памяти (VMM) определяет и отображает различные области памяти:

- Информация об доступных областях памяти собирается с помощью vmm_collect_regions().
- Отдельные регионы, такие как Flattened Device Tree (FDT) и линейное отображение всей защищённой RAM, регистрируются как динамические области с использованием vmm_register_dynamic_region().
- Секции кода и данных ядра затем отображаются на свои окончательные виртуальные адреса в таблицах страниц функцией vmm_map_kernel_regions().

Этот процесс формирует окончательную схему виртуальной памяти ядра, открывая возможности для постраничной защиты и динамического выделения памяти.

3.4.2.3. Вторичное перемещение ядра

Далее ядро обеспечивает размещение всех своих компонентов по верхним виртуальным адресам, определённым на этапе компоновки:

- Функция vmm_relocate_kernel_regions() выполняет переотображение секций ядра на виртуальные адреса 2x верхних ГиБ виртуального адресного пространства.

- Временные тождественные отображения, созданные во время начальной активации MMU, для раннего выполнения кода, очищаются.

Затем ядро выполняет переход с текущего адреса счетчика команд на соответствующее смещение нового положения кода. Это действие полностью переносит исполнение в виртуальное адресное пространство ядра, обеспечивая корректную работу среды выполнения C и доступ к глобальным символам по их виртуальным адресам, заданным на этапе компоновки.

3.4.2.4. Инициализация обработчика перерываний

Для обработки исключений, прерываний и системных вызовов Безопасная ОС настраивает свои механизмы обработки прерываний.

Функция vector_table_init() вызывается для конфигурации вектора прерываний S-режима, направляя исключения и прерывания ЦП в центральный диспетчер прерываний Безопасной ОС.

Это включает настройку обработчиков для синхронных исключений (таких как

системные вызовы из Доверенных Приложений), асинхронных прерываний (например, от таймера, межпроцессорных прерываний - IPI) и ошибочных состояний.

3.4.2.5. Инициализация таймеров

Таймеры используются для планировании задач и учёте времени в Безопасной ОС.

Сначала инициализируется аппаратура таймеров, специфичная для платформы, с использованием информации из FDT, посредством вызова arch_timer_init(fdt). Это настраивает базовый аппаратный таймер для генерации периодических прерываний.

Затем подсистема таймеров ядра инициализируется функцией timer_init(). В ходе этого процесса настраиваются программные структуры таймеров, и система подготавливается к использованию аппаратного таймера для генерации тиков планировщика и управления событиями, привязанными ко времени.

3.4.2.6. Инициализация аллокатора страниц

Для управления физической памятью, доступной Защищённому Миру, используеься аллокатор физических страниц.

Схема распределения памяти Безопасной ОС, включая доступные регионы RAM, определяется функцией arch_mem_initilize(fdt).

Затем инициализируется менеджер физической памяти (PMM) с помощью pmm_init(). Эта функция создаёт структуры данных (списки свободных страниц и битовые карты) для отслеживания состояния всех страниц физической памяти, выделенных для использования в защищённом режиме, обеспечивая возможность динамического выделения и освобождения страниц.

3.4.2.7. Инициализация slab-аллокатора

Для эффективного управления небольшими, часто выделяемыми объектами ядра, поверх аллокатора страниц создаётся slab-аллокатор.

Функция kernel_slabs_init() инициализирует slab-аллокатор. Это подразумевает создание кэшей для объектов различных распространённых размеров, что уменьшает фрагментацию и повышает производительность выделения и освобождения структур данных ядра (например, блоков управления задачами, структур потоков, буферов сообщений IPC).

3.4.2.8. Инициализация планировщика

Планировщик отвечает за управление выполнением потоков в Безопасной ОС.

Базовые структуры для управления задачами подготавливаются функцией init_system_task(), которая настраивает начальные контексты задач ядра или idle поток.

Далее настраиваются структуры данных планировщика: очереди готовых задач, и планировщик интегрируется с подсистемой таймеров инициализированной через timer_init() для обеспечения вытесняющего планирования с квантованием времени на основе политики Round-Robin.

3.4.2.9. Инициализация Root Task

Корневая Задача - это компонент, выступающий в роли первого процесса в Защищённом Мире. Она отвечает за управление Доверенными Приложениями и обработку запросов взаимодействия между мирами.

После инициализации планировщика вызывается spawn_root_task() Это включает загрузку ELF-файла Корневой Задачи, настройку её выделенного адресного пространства, выделение начальных ресурсов (включая дескрипторы, указанные в её манифесте), создание основного потока, и подготовку к обработке запросов.

3.4.2.10. Инициализация канала связи с Обычным Миром

Для обеспечения взаимодействия с Linux инициализируется выделенная коммуникационная инфраструктура.

Функция nwd_channel_init(fdt) настраивает очереди в разделяемой памяти. Этот процесс включает:

- Определение областей физической памяти для очередей запросов и ответов на основе информации из FDT.
- Отображение разделяемых страниц в адресное пространство Безопасной ОС.
- Инициализацию структур данных для lock-free очередей MPMC как для запросов, так и для ответов.

3.4.2.11. Инициализация Доверенных Приложений

Хотя большинство Доверенных Приложений (ДП) запускаются динамически Корневой Задачей по запросу из Обычного Мира через TEEC_OpenSession, на данном этапе производится запуск серсисов ДСИ, таких, как сервисы криптостойкого генератора случайных чисел, сервиса вычисления криптографических хешей.

3.5. Межмировая коммуникация

3.5.1. Lock-Free Алгоритм Очереди

3.5.1.1. Введение в Lock-Free алгоритмы

Взаимодействие между Обычным миром (Linux) и Защищённым миром (Secure OS) основано на использовании очередей в разделяемой памяти. Для обеспечения эффективного и безопасного параллельного доступа из обоих миров без использования традиционных блокировок применяется Lock-Free алгоритм очереди типа "Множество Производителей - Множество Потребителей" (MPMC).

Lock-Free алгоритмы - это класс параллельных алгоритмов, гарантирующих общесистемный прогресс: если один или несколько потоков выполняют операции над структурой данных, по крайней мере один поток должен завершить свою операцию за конечное число собственных шагов, независимо от состояния (например, приостановка, сбой) других потоков.

Lock-Free алгоритмы обычно полагаются на атомарные аппаратные примитивы, такие

как CAS или LL/SC, для управления одновременными изменениями состояния.

3.5.1.2. Алгоритм MPMC-очереди

Выбранная MPMC-очередь представляет собой ограниченную очередь на основе массива (кольцевого буфера), разработанную для высокой производительности за счет использования атомарных операций и тщательного управления порядковыми номерами для координации производителей и потребителей.

3.5.1.3. Концепция Алгоритма

Каждая ячейка в кольцевом буфере очереди ассоциируется с порядковым номером seq. Производители и потребители используют глобальные атомарные счетчики enqueue и dequeue соответственно для получения доступа к определенным логическим ячейкам. Производитель, пытающийся записать данные в логическую ячейку pos, может сделать это только в том случае, если seq целевой физической ячейки совпадает с pos.

После записи производитель обновляет seq ячейки до pos + 1. Потребитель, пытающийся прочитать данные из логической ячейки pos, может сделать это только в том случае, если seq целевой физической ячейки совпадает с pos + 1.

После чтения потребитель обновляет seq ячейки до pos + N (где N - вместимость очереди), помечая ее доступной для будущей операции постановки в очередь.

Эта схема гарантирует, что ячейки обрабатываются по порядку и корректно передаются между производителями и потребителями.

3.5.1.4. Структуры данных

Реализация очереди использует две основные структуры:

mpmc_queue Представляет собой разделяемую очередь.

```
typedef struct {
    // Указатель на буфер с ячейками queue_cell
    void *data;

    // Маска для вычисления индекса в кольцевом буфере (ёмкость - 1)
    size_t size_mask;

    // Размер одной ячейки queue_cell (включая данные)
    size_t cell_size;

    // Размер пользовательских данных внутри ячейки
    size_t data_size;

    // Глобальный счетчик для следующей ячейки на постановку в очередь
    _Atomic size_t enqueue cacheline_aligned;

    // Глобальный счетчик для следующей ячейки на извлечение из очереди
    _Atomic size_t dequeue cacheline_aligned;
} mpmc_queue;
```

Счетчики enqueue и dequeue выровнены по границе кэш-линии (cacheline_aligned) во избежание ложного разделения (false sharing) между ядрами, которые могут одновременно пытаться добавлять и извлекать элементы.

Вместимость очереди должна быть степенью двойки, что позволяет эффективно вычислять индекс с помощью size_mask.

queue_cell Представляет отдельную ячейку в очереди.

```
typedef struct {
    // Порядковый номер для этой ячейки
    _Atomic int seq;

    // Данные пользователя
    unsigned char data[];
} queue_cell;
```

Поле seq является атомарным, так как это основной механизм синхронизации между производителями и потребителями для данной ячейки.

3.5.1.4.1. Инициализация

Функция mpmc_queue_init инициализирует очередь:

1. Передаются указатель на буфер для хранения элементов, его общий размер, размер элементов данных и вычисленный размер ячейки.

2. Атомарные счетчики enqueue и dequeue инициализируются нулем с использованием atomic_store_explicit с семантикой RELAXED.

3. Рассчитывается size_mask как ёмкость - 1.

4. Поле seq каждой структуры queue_cell с индексом i в буфере инициализируется значением i Такое состояние (cell[i].seq == i) означает, что ячейка i готова для i-ой операции постановки в очередь (т.е. когда глобальный счетчик enqueue равен i).

3.5.1.4.2. Операция постановки в очередь

Производитель, желающий добавить элемент в очередь, выполняет следующие шаги:

1. Атомарно считывается текущее значение счетчика enqueue, pos. pos - логическая ячейка, которую производитель намеревается заполнить.

2. Попытка занять ячейку:

    - Вычисляется индекс физической ячейки
    - Атомарно считывается seq из cell[idx] с семантикой ACQUIRE. Это гарантирует видимость любых предшествующих записей в эту ячейку (например, потребителем, освободившим её).
    - Проверяется, равно ли cell[idx].seq значению pos. Если да, то потребитель захватил ячейку pos. Иначе, попытка повторяется до достижения успеха.

3. После успешного захвата ячейки, данные пользователя копируются в ячейку

4. Атомарно записывается pos + 1 в cell[idx].seq с семантикой RELEASE. Это делает запись данных видимой для потребителей и сигнализирует, что pos-й элемент готов к извлечению.

3.5.1.4.3. Операция извлечения из очереди

Потребитель, желающий извлечь элемент из очереди, выполняет следующие шаги:

1. Атомарно считывается текущее значение счетчика enqueue, pos. pos - логическая ячейка, которую потребитель намеревается прочитать.

2. Попытка занять ячейку:

    - Вычисляется индекс физической ячейки
    - Атомарно считывается seq из cell[idx] с семантикой ACQUIRE. Это гарантирует видимость любых предшествующих записей в эту ячейку (например, потребителем, освободившим её).
    - Проверяется, равно ли cell[idx].seq значению pos. Если да, то потребитель захватил ячейку pos. Иначе, попытка повторяется до достижения успеха.

3. После успешного захвата элемента, данные копируются в буфер пользователя.

4. Атомарно записывается pos + queue->size_mask + 1 в cell[idx].seq с семантикой RELEASE. Это помечает ячейку как пустую и доступную для будущей операции enqueue.

3.5.2. Очереди в разделяемой памяти

3.5.2.1. Кольцевые буферы в разделяемой памяти

Межмировое взаимодействие основывается на использовании очередей в разделяемой памяти. Эти очереди реализованы как кольцевые буферы, расположенные в областях памяти, доступных как Обычному миру, так и Защищённому.

Коммуникационные очереди физически представляют собой кольцевые буферы, размещённые на двух специально выделенных страницах физической памяти.

Конфигурация этих страниц осуществляется Безопасной ОС с помощью расширения WorldGuard, которое предоставляет Защищённому и Обычному мирам разделяемый доступ на чтение и запись. При этом права доступа для каждой очереди настраиваются в соответствии с ее специфической ролью (производитель или потребитель).

Каждый кольцевой буфер - это циклический массив ячеек фиксированного размера, предназначенных для хранения структур wg_tee_cmd.

Важным аспектом является выравнивание памяти. Сама структура mpmc_queue выравнивает свои поля по границам кэш-линий. Это минимизирует ложное разделение (false sharing) между ядром ЦП, на котором работает Безопасная ОС, и ядрами, исполняющими Linux. Слоты для сообщений в кольцевом буфере выровнены по 256 байт для размещения структур, что предотвращает падение производительности из-за невыровненных обращений.

3.5.2.2. Очередь запросов

Один из кольцевых буферов, расположенный на выделенной странице разделяемой памяти, используется в качестве Очереди Запросов. В этой схеме Обычный мир (драйвер Linux) выступает в роли производителя, помещая в очередь запросы, предназначенные для обработки Защищённым миром. Безопасная ОС, соответственно, является потребителем данной очереди.

Процесс постановки запроса в очередь со стороны Обычного мира выглядит следующим образом:

1. Драйвер Linux формирует команду TEE в виде структуры wg_tee_cmd.

2. Драйвер пытается поместить данную структуру в свободный слот Очереди Запросов, используя операцию mpmc_queue_push. Эта функция сама находит свободный слот и атомарно обновляет счётчики данной очереди.

3. В случае успешного добавления команды в очередь, драйвер Linux отправляет Межпроцессорное Прерывание (IPI) ядру Безопасной ОС, уведомляя его о появлении нового запроса

3.5.2.3. Очередь ответов

Второй кольцевой буфер отведен под Очередь Ответов и располагается на другой странице разделяемой памяти. Здесь Безопасная ОС выступает производителем, помещая в очередь результаты обработки команд TEE или уведомления о событиях. Обычный мир (драйвер Linux), в свою очередь, является потребителем.

Безопасная ОС помещает ответ в очередь следующим образом:

1. Заполняет структуру wg_tee_cmd данными, содержащими результат выполненной операции.

2. Помещает эту структуру в Очередь Ответов, используя функцию mpmc_queue_push. При этом атомарно обновляя счётчики Очереди Ответов.

3. Драйвер Обычного мира использует механизм опроса (polling) для периодической проверки Очереди Ответов на наличие готовых результатов.

3.5.2.4. Канареечные страницы вокруг разделяемых страниц

Для обнаружения и предотвращения ситуаций переполнения или опустошения буфера (buffer overflow/underflow) на страницах очередей в разделяемой памяти используются так называемые канареечные страницы.

В процессе инициализации Безопасная ОС настраивает правила WorldGuard для физических страниц, непосредственно примыкающих (с обеих сторон) к двум страницам разделяемой памяти, используемым для очередей.

Этим канареечным страницам WorldGuard назначает права доступа, полностью запрещающие любые операции (чтение, запись, исполнение) как со стороны WID 0 (Защищённый мир), так и со стороны WID 1 (Обычный мир). Если в результате ошибочной операции какой-либо из миров попытается обратиться к памяти за пределами выделенных границ разделяемой очереди, это приведёт к обращению к одной из таких канареечных страниц. Попытка несанкционированного доступа вызовет аппаратное исключение.

Это позволяет системе обнаружить подобные нарушения безопасности памяти и предпринять адекватные меры, например, зарегистрировать ошибку или принудительно завершить вызвавший её компонент.

3.5.3. Области разделяемой памяти

Помимо основных очередей, используемых для обмена управляющими командами и ответами, система поддерживает динамическое выделение и совместное использование более крупных областей памяти. Эти области предназначены для эффективной передачи больших объемов данных между Обычным миром и доверенными приложениями, исполняющимися в Безопасном мире.

3.5.3.1. Выделение области памяти

Выделение области разделяемой памяти инициируется Обычным миром (Linux) путем вызова защищенной операции TEE_CMD_ID_MAP_SHARED_MEM. Безопасная ОС обрабатывает этот запрос следующим образом:

1. Выделение страниц в Безопасном мире:
   Безопасная ОС выделяет запрошенное количество страниц физической памяти из собственного управляемого пула.

2. Настройка доступа:
   Для выделенных страниц конфигурируются правила контроллера WorldGuard, предоставляющие права на одновременный доступ (например, чтение, запись) как Безопасному миру (действующему от имени доверенного приложения), так и Обычному миру.

3. Отображение в Безопасной ОС:
   Безопасная ОС отображает выделенные физические страницы в адресное пространство своего ядра. Это позволяет самой Безопасной ОС, а также доверенному приложению, для которого предназначается память, обращаться к этой разделяемой области.

4. Отображение в Обычном мире:
   После успешного выделения памяти и настройки прав доступа в Безопасном мире, драйвер Linux получает физический адрес и размер разделяемой области. Затем драйвер отвечает за отображение этих физических страниц в виртуальное адресное пространство ядра Linux, делая регион доступным для Обычного мира. В Обычный мир также возвращается идентификатор этой области разделяемой памяти (shmem_id), который используется для последующих обращений к ней.

3.5.3.2. Освобождение области памяти

Освобождение памяти инициируется Обычным миром через вызов защищенной операции TEE_CMD_ID_UNMAP_SHARED_MEM, с указанием идентификатора shmem_id, полученного на этапе выделения. Безопасная ОС выполняет этот запрос, следуя следующим шагам:

1. Отзыв прав в Безопасном мире:
   Безопасная ОС сначала удостоверяется, что соответствующее доверенное приложение (если оно есть) прекратило использовать данную область памяти. После этого она перенастраивает правила контроллера WorldGuard для

физических страниц этой разделяемой области, отзывая все права доступа как для Безопасного, так и для Обычного мира.

2. Отмена отображения в Безопасной ОС:
   Безопасная ОС удаляет отображение региона из адресного пространства своего ядра.

3. Физическое освобождение страниц:
   Физические страницы возвращаются в пул свободной памяти, управляемый Безопасной ОС.

4. Отмена отображения в Обычном мире:
   После того, как Безопасная ОС подтвердит успешное освобождение, драйвер Linux удаляет отображение соответствующей области памяти из виртуального адресного пространства ядра Linux, тем самым освобождая занимаемые ею виртуальные адреса.

3.5.3.3. Передача данных

Когда область разделяемой памяти выделена и отображена в виртуальные адресные пространства ядра Linux и Безопасной ОС (что обеспечивает к ней доступ для доверенного приложения), передача данных между мирами осуществляется напрямую, с помощью стандартных операций чтения и записи в память.

3.5.4. Общая структура сообщений

Все команды, передаваемые через очереди запросов и ответов, имеют унифицированный формат. Структура wg_tee_cmd инкапсулирует параметры и результаты операций TEE. Эта структура служит основной единицей обмена данными и содержит идентификаторы команд, информацию для отслеживания сессий, коды ошибок и параметры полезной нагрузки для операций TEE.

3.5.4.1. Структура wg_tee_cmd

Структура wg_tee_cmd - пакет данных для межмирового взаимодействия через очереди в разделяемой памяти. Она стандартизирует формат всех командных запросов TEE, исходящих из Обычного мира, и ответов от Защищённой ОС.

Компоненты структуры включают идентификатор команды, порядковый номер для отслеживания, идентификаторы сессии и Доверенного Приложения, операционные параметры или детали отображения памяти, также поле для кодов возврата. Структура дополняется до фиксированного размера (256 байт) для обеспечения выравнивания в очередях разделяемой памяти.

3.5.4.2. Поле id

Поле uint32_t id определяет тип запрашиваемой или обрабатываемой операции TEE. Это позволяет принимающему миру направить команду соответствующему обработчику.

Определены следующие идентификаторы команд:

- TEE_CMD_ID_OPEN_SESSION:

Инициирует новую сессию с Доверенным Приложением (ДП).

- TEE_CMD_ID_CLOSE_SESSION:
  Завершает существующую сессию с ДП.
- TEE_CMD_ID_INVOKE_CMD:
  Вызывает определённую функцию в рамках активной сессии ДП.
- TEE_CMD_ID_MAP_SHARED_MEM:
  Запрашивает у Защищённой ОС отображение области физической памяти в качестве разделяемой между Обычным и Защищённым мирами.
- TEE_CMD_ID_UNMAP_SHARED_MEM:
  Запрашивает у Защищённой ОС отмену отображения ранее разделяемой области памяти.

3.5.4.3. Поле seq

Поле uint32_t seq служит уникальным идентификатором для каждого экземпляра команды. Этот порядковый номер генерируется Обычным миром путем атомарного инкрементирования счетчика для каждого нового запроса. Защищённая ОС включает этот номер seq в свой ответ, позволяя клиенту Обычного мира сопоставлять ответы с исходными запросами, особенно в условиях асинхронного взаимодействия.

3.5.4.4. Поле session_id

Поле uint32_t session_id идентифицирует конкретную сессию, к которой относится команда. После успешного выполнения команды TEE_CMD_ID_OPEN_SESSION Защищённая ОС возвращает session_id Обычному миру. Последующие команды TEE_CMD_ID_INVOKE_CMD и TEE_CMD_ID_CLOSE_SESSION должны включать этот параметр для обращения к корректному экземпляру ДП и связанному с ним контексту. Это позволяет одному Клиентскому Приложению в Обычном мире управлять несколькими одновременными сессиями с одним или несколькими ДП.

3.5.4.5. Поле func_id

Поле uint32_t func_id используется в команде TEE_CMD_ID_INVOKE_CMD. Каждое Доверенное Приложение может предоставлять несколько функций или команд. Это поле позволяет клиенту Обычного мира указать, какое конкретное действие или функция, идентифицируемая func_id, должно быть выполнено ДП во время операции InvokeCommand.

3.5.4.6. Поле err

Поле uint32_t err используется в ответах от Защищённой ОС Обычному миру. Оно передает результат выполнения запрошенной операции TEE. Значение TEEC_SUCCESS означает успешное завершение, в то время как другие значения указывают на различные ошибки, такие как TEEC_ERROR_BAD_PARAMETERS (некорректные параметры), TEEC_ERROR_ACCESS_DENIED (доступ запрещён) или специфичные ошибки ДП.

3.5.4.7. Поле uuid

Поле uint8_t uuid[16] содержит 128-битный универсальный уникальный идентификатор (UUID), однозначно идентифицирующий Доверенное Приложение. Это поле используется при операции TEE_CMD_ID_OPEN_SESSION, когда клиент Обычного мира предоставляет UUID Доверенного Приложения, к которому он хочет подключиться.

Защищённая ОС использует этот UUID для поиска и создания экземпляра нужного ДП.

3.5.4.8. Поле paddr

Поле uint64_t paddr используется для операций с разделяемой памятью, в частности, для команды TEE_CMD_ID_MAP_SHARED_MEM. Оно указывает начальный физический адрес области памяти в Обычном мире, которая должна быть отображена в Защищённый мир как разделяемая. Для других идентификаторов команд это поле обычно не используется.

3.5.4.9. Поле num_pages

Поле uint32_t num_pages дополняет поле paddr для операций TEE_CMD_ID_MAP_SHARED_MEM. Оно определяет количество смежных страниц физической памяти, начиная с адреса paddr, которые должны быть включены в отображение разделяемой памяти. Это поле не используется для других идентификаторов команд.

3.5.4.10. Поле shmem_id

Поле uint32_t shmem_id служит дескриптором для успешно отображённой области разделяемой памяти. Когда Защищённая ОС обрабатывает запрос TEE_CMD_ID_MAP_SHARED_MEM, она возвращает shmem_id в ответе. Этот идентификатор должен быть предоставлен клиентом Обычного мира в последующих запросах TEE_CMD_ID_UNMAP_SHARED_MEM для указания, какую область разделяемой памяти необходимо освободить и отменить её отображение.

3.5.4.11. Поле params

Поле struct wg_param params[4] представляет собой массив из четырёх структур по 24 байта каждая, предназначенный для передачи параметров в Доверенное Приложение и из него во время операции TEE_CMD_ID_INVOKE_CMD. Это соответствует структуре TEEC_Operation из клиентского API GlobalPlatform TEE, которая также допускает до четырёх параметров.

Каждая структура wg_param может представлять либо небольшое значение, передаваемое напрямую, либо ссылку на область памяти.

В частности:

- Простые аргументы (значения):
  Небольшие значения данных могут храниться непосредственно в 24-байтном пространстве wg_param. Интерпретация этих значений зависит от конкретного ДП.

- Ссылки на память:
  Для передачи больших буферов данных wg_param может представлять ссылку на

память. В данной реализации ссылка на память структурирована как три 64-битных значения: size (размер адресуемой области памяти), offset (смещение внутри области разделяемой памяти, идентифицированной shmem_id) и world_id (идентификатор мира, указывающий, находится ли память в Обычном или Защищённом мире, или является ли она разделяемой).

Тип каждого параметра (значение или направление ссылки на память: входной, выходной, входной/выходной) передаётся через отдельное поле param_types.

3.5.4.12. Поле padding

Поле uint8_t padding[108] представляет собой зарезервированное пространство, используемое для выравнивания общего размера структуры wg_tee_cmd до 256 байт.

Этот фиксированный общий размер упрощает управление очередями, гарантируя, что каждое сообщение занимает предсказуемое количество кеш-линий и облегчает вычисление смещений слотов в кольцевых буферах разделяемой памяти.

Явное дополнение предотвращает вариации из-за специфического поведения выравнивания компилятора в различных инструментальных цепочках или на разных платформах.

3.5.5. Уведомление на основе межпроцессорных прерываний

3.5.5.1. Механизм IPI в RISC-V

Хотя очереди в разделяемой памяти предоставляют структуры данных для обмена сообщениями команд и ответов между Обычным и Защищённым мирами, для запуска уведомлений в реальном времени о новых сообщениях используется механизм межпроцессорных прерываний (IPI). Это гарантирует, что принимающий мир сможет оперативно обрабатывать входящие сообщения, не прибегая к непрерывному высокочастотному опросу очередей, снижая тем самым нагрузку на ЦП.

Архитектура RISC-V предоставляет механизмы для отправки IPI между ядрами. Программное обеспечение, работающее на одном ядре, может инициировать прерывание на другом целевом ядре.

Это достигается путем записи в регистр, отображенный в память и связанный с контроллером прерываний целевого ядра: установкой бита Supervisor Software Interrupt Pending (SSIP) в CSR-регистре sip, или с помощью платформенных контроллеров прерываний, таких как Core Local Interruptor (CLINT), через его регистр программных прерываний msip, отображенный в память.

3.5.5.2. Сигнализация из Обычного мира в Защищённый

Когда драйвер Linux в Обычном мире должен отправить команду Защищённой ОС, он выполняет следующую процедуру:

1. Драйвер заполняет структуру wg_tee_cmd деталями команды и параметрами.
2. Эта структура затем помещается в общую очередь запросов с использованием очереди MPMC.
3. После успешного добавления команды в очередь драйвер инициирует IPI,

направленное на ядро, на котором исполняется Защищённая ОС. Это IPI отправляется через вызов sbi_send_ipi, предоставляемый OpenSBI.

4. IPI служит сигналом для Защищённой ОС, побуждая её проверить очередь запросов на наличие новых сообщений. Запрашивающий поток в драйвере Linux затем ожидает (блокируется на условной переменной), пока ответ не появится в очереди ответов.

Такое уведомление на основе IPI минимизирует задержку между отправкой команды и началом её обработки Защищённой ОС.

3.5.5.3. Сигнализация из Защищённого мира в Обычный

В обратном направлении, когда Защищённая ОС обработала команду и поместила ответ в общую очередь ответов, механизм сигнализации отличается.

Исходя из архитектурных соображений и стремления упростить обработку прерываний в ядре Linux Обычного мира, Защищённая ОС не отправляет IPI обратно на ядра Обычного мира.

Основная причина такого решения - потенциальная сложность для ядра Linux надёжно отличить IPI, исходящее именно от Защищённой ОС для взаимодействия с TEE, от других типов IPI, которые оно может получать (например, для балансировки нагрузки планировщика, инвалидации TLB или других подсистем ядра). Некорректная обработка или неверная интерпретация таких IPI может привести к нестабильности системы.

Следовательно, поток обмена данными для ответов выглядит следующим образом:

1. Защищённая ОС обрабатывает запрос и помещает структуру ответа wg_tee_cmd в общую очередь ответов.

2. Драйвер Linux в Обычном мире использует активный опрос (polling). Выделенный поток ядра проверяет очередь ответов на наличие готовых команд.

3. Когда механизм опроса находит ответ, соответствующий ожидающему запросу (сопоставление происходит по полю seq), он извлекает ответ и пробуждает исходный запрашивающий поток, доставляя результат.

Хотя опрос может вносить некоторую задержку по сравнению с прямым прерыванием, он упрощает управление прерываниями на стороне Linux и позволяет избежать сложностей двунаправленного протокола сигнализации IPI для данного конкретного взаимодействия с TEE. Частоту опроса можно настраивать для достижения баланса между скоростью отклика и нагрузкой на ЦП.

3.6. API ДСИ

3.6.1. Global Platform API

3.6.1.1. Introduction to Global Platform API

Global Platform работает в различных отраслях промышленности, определяя, разрабатывая и публикуя спецификации, которые облегчают безопасное и совместимое

развертывание и управление несколькими встроенными приложениями на основе технологии secure chip.

Secure OS поддерживает спецификацию API GlobalPlatform TEE версии 1.0 (GPD_SPE_007), а также исправления 2.0 (GPD_EPR_028) и спецификацию внутреннего базового API TEE версии 1.3.1 (GPD_SPE_010).

3.6.1.2. Введение в пользовательский интрефейс ДСИ

Клиентский API TEE описывает и определяет, как клиент, работающий в Rich OS (REE), должен взаимодействовать с TEE.

Чтобы определить используемое доверенное приложение (TA), клиент предоставляет UUID. Все TA предоставляют доступ к одной или нескольким функциям. Эти функции соответствуют так называемому command_id, который также отправляется клиентом.

3.6.2. Пользовательский интрефейс ДСИ

3.6.2.1. Контекст ДСИ

Контекст TEE используется для создания логического соединения между клиентом и TEE. Контекст должен быть инициализирован до того, как можно будет создать сеанс TEE. Когда клиент завершит выполнение задания в безопасном мире, он должен закрыть контекст и, таким образом, освободить ресурсы.

3.6.2.2. Сессия ДСИ

Сеансы используются для создания логических соединений между клиентом и определенным доверенным приложением. После установления сеанса клиент открывает канал связи с указанным Доверенным приложением, идентифицируемым с помощью UUID. На этом этапе клиент и Доверенное приложение могут начать обмениваться данными.

3.6.2.3. Разделяемая память ДСИ

Клиентский API TEE описывает множество способов совместного использования памяти между клиентом и TEE. Некоторые способы более эффективны, чем другие, из-за того, как они реализованы, но у них есть и свои преимущества. Например, использование ссылки на временную память часто удобно, но в зависимости от ситуации часто не является наиболее эффективным. Временная ссылка на память устанавливается внутри клиентской библиотеки TEE перед ее использованием, и когда запрос в безопасный мир завершен, она снова удаляется.

Для более эффективной коммуникации следует использовать блок общей памяти, поскольку его можно повторно использовать между вызовами, а также настраивать другими способами. Блок общей памяти может быть инициализирован либо с помощью TEEC_RegisterSharedMemory(), либо с помощью TEEC_AllocateSharedMemory().

TEEC_RegisterSharedMemory() иногда не удается установить общую память с без копии, и в таких случаях приходится использовать временный теневой буфер. Платформа TEE, например, откажется регистрировать блок памяти, который отображается только для чтения в клиенте.

TEEC_AllocateSharedMemory() - лучший выбор для создания общей памятиб без копирования. Если вместо этого необходимо использовать TEEC_RegisterSharedMemory(), поскольку буфер выделен заранее или извне, есть еще несколько вещей, которые помогают избежать возврата к теневому буферу.

3.6.3. Спецификация TEE API

3.6.3.1. TEEC_UUID

Структура TEEC_UUID используется для однозначной идентификации доверенных приложений (Trusted Applications, TA). Она соответствует стандартному 128-битному формату UUID и включает поля:

- timeLow (32 бита)
- timeMid (32 бита)
- timeHiAndVersion (16 бит)
- clockSeqAndNode (массив из 8 байт).

Эта структура используется при установлении сессии (TEEC_OpenSession) для указания целевого TA в безопасной ОС.

3.6.3.2. TEEC_Result

TEEC_Result - это алиас на тип uint32_t, используемый в качестве стандартного возвращаемого значения для всех функций TEE Client API. Он указывает на результат выполнения операции, сигнализируя об успехе (TEEC_SUCCESS) или конкретном состоянии ошибки.

Допустимые значения:

- TEEC_SUCCESS
- TEEC_ERROR_BAD_PARAMETERS
- TEEC_ERROR_ITEM_NOT_FOUND
- TEEC_ERROR_GENERIC

Для сообщения детального статуса вызывающей стороне.

3.6.3.3. TEEC_Context

Структура TEEC_Context представляет собой соединение от клиентского приложения (Client Application) из обычного мира (Normal World) к фреймворку TEE, управляемому безопасной ОС.

Структура инкапсулирует файловый дескриптор, полученный при открытии устройства TEE (/dev/tee0). Этот файловый дескриптор используется для всех последующих взаимодействий с драйвером Linux на основе ioctl, который, в свою очередь, взаимодействует с безопасной ОС.

Контекст должен быть инициализирован перед выполнением любых других операций TEE.

3.6.3.4. TEEC_Session

Структура TEEC_Session представляет активное соединение между клиентским приложением и конкретным экземпляром доверенного приложения, исполняемого в безопасной ОС. Она содержит указатель на родительский TEEC_Context (ctx) и идентификатор сессии (session_id), присваиваемый TEE при успешном создании сессии через TEEC_OpenSession.

В рамках одного контекста могут сосуществовать несколько сессий к различным или одному и тому же TA.

3.6.3.5. TEEC_Value

Структура TEEC_Value предназначена для прямой передачи небольших данных фиксированного размера между клиентским приложением и доверенным приложением. Она состоит из двух членов типа uint32_t: a и b.

Структура используется как часть TEEC_Parameter, когда тип параметра указывает на прямую передачу значения (входного, выходного или входного/выходного).

3.6.3.6. TEEC_RegisteredMemoryReference

Структура TEEC_RegisteredMemoryReference определяет сегмент ранее выделенного блока TEEC_SharedMemory. Она включает указатель на родительскую структуру TEEC_SharedMemory (parent), размер (size) сегмента и его смещение (offset) от начала родительского блока общей памяти.

Структура позволяет как клиентскому приложению, так и доверенному приложению ссылаться на определённые участки разделяемого буфера без необходимости передавать весь буфер целиком или повторно регистрировать его части.

3.6.3.7. TEEC_Parameter

TEEC_Parameter - это объединение (union), которое может содержать либо TEEC_Value (для небольших, передаваемых по значению параметров), либо TEEC_RegisteredMemoryReference (для параметров, ссылающихся на области разделяемой памяти).

Массив, именуемый params, содержащий до четырёх экземпляров TEEC_Parameter, встроен в структуру TEEC_Operation. Активный член объединения для каждого параметра определяется соответствующим paramType в TEEC_Operation.

3.6.3.8. TEEC_Operation

Структура TEEC_Operation инкапсулирует все параметры для вызова команды в доверенном приложении или для открытия сессии. Содержит массив из четырёх объединений TEEC_Parameter (params) и поле paramTypes типа uint32_t.

Поле paramTypes кодирует тип и направление (например, TEEC_VALUE_INPUT, TEEC_MEMREF_TEMP_OUTPUT) для каждого из четырёх параметров с использованием макросов, таких как TEEC_PARAM_TYPE_GET.

3.6.3.9. TEEC_SharedMemory

Структура TEEC_SharedMemory представляет область памяти, предназначенную для

совместного использования между клиентским приложением из обычного мира и доверенным приложением из безопасного мира.

Структура включает:

- void *buffer:
  виртуальный адрес в пользовательском пространстве клиентского приложения, куда отображается общая память (после выделения и отображения).

- size_t size:
  размер области общей памяти в байтах.

- uint32_t flags: определяет права доступа, флаги выделения или свойства регистрации, используемые TA и безопасной ОС (например, для указания направления передачи данных для всего буфера при глобальной регистрации).

- int fd:
  файловый дескриптор, полученный от драйвера TEE после выделения (например, через TEE_IOC_SHM_ALLOC), используемый клиентским приложением для отображения памяти.

- int32_t id:
  идентификатор блока общей памяти, присвоенный TEE, который используется для ссылки на этот блок в последующих операциях, таких как TEEC_RegisteredMemoryReference.

3.6.3.10. TEEC_InitializeContext

Функция с прототипом

```
TEEC_Result TEEC_InitializeContext(const char *name, TEEC_Context *cont
```

устанавливает соединение с подсистемой TEE.

- Параметры:
  - name (необязательный, обычно NULL для имени TEE по умолчанию)
  - указатель на структуру TEEC_Context для инициализации.

- Поведение:
  Перебирает возможные узлы устройств TEE (например, /dev/tee0 до /dev/tee(TEEC_MAX_DEV_SEQ-1)) и пытается открыть один из них. В случае успеха файловый дескриптор открытого устройства сохраняется в context->fd.

- Возвращаемое значение:
  TEEC_SUCCESS в случае успеха. TEEC_ERROR_BAD_PARAMETERS, если context равен NULL. TEEC_ERROR_ITEM_NOT_FOUND, если ни одно устройство TEE не может быть открыто.

3.6.3.11. TEEC_FinalizeContext

Функция с прототипом

```
TEEC_Result TEEC_FinalizeContext(TEEC_Context *context)
```

завершает соединение с подсистемой TEE и освобождает ресурсы, связанные с указанным TEEC_Context.

- Параметры:
  - Указатель на TEEC_Context, который необходимо закрыть.
- Поведение:
  закрывает файловый дескриптор (context->fd) и гарантирует, что все оставшиеся ресурсы TEE, управляемые этим контекстом (например, косвенное освобождение открытых сессий или разделяемой памяти драйвером, если это не было сделано явно ранее), будут очищены со стороны клиента и об этом будет сообщено драйверу TEE.
- Возвращаемое значение:
  TEEC_SUCCESS при успешной финализации.

3.6.3.12. TEEC_OpenSession

Функция с прототипом

```
TEEC_Result TEEC_OpenSession(TEEC_Context *ctx, TEEC_Session *session,
```

открывает новую сессию с указанным доверенным приложением.

- Параметры:
  - Инициализированный TEEC_Context,
  - указатель на структуру TEEC_Session для заполнения
  - TEEC_UUID целевого TA,
  - connection_method (метод подключения, например, TEEC_LOGIN_PUBLIC)
  - connection_data (необязательный) для передачи начальных данных
  - operation (необязательный)
  - указатель uint32_t для ret_origin (источник ошибки).
- Поведение:
  Заполняет структуру UUID-ом TA и другими параметрами (обработка connection_data, connection_method и operation для первоначального контакта с TA осуществляется через сам аргумент ioctl, но детальная обработка является специфичной частью драйвера Linux/Безопасной ОС). Затем выполняет системный вызов ioctl с командой TEE_IOC_OPEN_SESSION, используя файловый дескриптор контекста. В случае успеха session->session_id заполняется идентификатором, возвращенным драйвером,
- Возвращаемое значение:
  TEEC_SUCCESS в случае успеха, TEEC_ERROR_GENERIC, если ioctl завершается ошибкой.

3.6.3.13. TEEC_CloseSession

Функция с прототипом

```
TEEC_Result TEEC_CloseSession(TEEC_Session *session)
```

закрывает ранее открытую сессию с доверенным приложением.

- Параметры:
    - Указатель на TEEC_Session, которую необходимо закрыть.
- Поведение:
  Вызывает ioctl (TEE_IOC_CLOSE_SESSION) к драйверу TEE, передавая session->session_id, для информирования безопасной ОС о необходимости завершить сессию и освободить связанные с ней ресурсы TA.
- Возвращаемое значение:
  TEEC_SUCCESS при успешном закрытии.

3.6.3.14. TEEC_InvokeCommand

Функция с прототипом

```
TEEC_Result TEEC_InvokeCommand(TEEC_Session *session, uint32_t commandI
```

вызывает команду в контексте открытой сессии с доверенным приложением.

- Параметры:
    - Активная TEEC_Session,
    - идентификатор команды commandID, указывающий на выполняемую функцию TA
    - структура TEEC_Operation, содержащая параметры для команды
    - указатель для returnOrigin (выходной параметр, источник ошибки).
- Поведение:
    1. Подготавливает struct tee_ioctl_invoke_arg, устанавливая func в commandID, session в session->session_id, и копирует параметры.
    2. Вспомогательная функция prepare_ioctl_args преобразует TEEC_Operation (типы параметров, значения и детали memref, такие как ID родительской общей памяти, смещение и размер) в формат массива struct tee_ioctl_param, требуемый драйвером TEE Linux
    3. Выполняет системный вызов ioctl с командой TEE_IOC_INVOKE, используя файловый дескриптор контекста сессии.
    4. Если ioctl успешен, вспомогательная функция prepare_ioctl_args_out преобразует любые выходные значения или измененные параметры из массива struct tee_ioctl_param обратно в структуру TEEC_Operation. Поле *returnOrigin устанавливается драйвером в случае ошибки или может быть установлено на стороне клиента, если подготовка API завершилась неудачно.
- Возвращаемое значение:
  TEEC_SUCCESS в случае успеха. TEEC_ERROR_BAD_PARAMETERS, если преобразование параметров не удалось. TEEC_ERROR_GENERIC, если ioctl завершается ошибкой.

3.6.3.15. TEEC_AllocateSharedMemory

Функция с прототипом

```
TEEC_Result TEEC_AllocateSharedMemory(TEEC_Context *context, TEEC_Share
```

запрашивает выделение нового блока общей памяти, доступного как из обычного, так и из безопасного мира.

- Параметры:
  - Инициализированный TEEC_Context
  - указатель на структуру TEEC_SharedMemory.

Вызывающая сторона должна установить shared_mem->size в желаемый размер (если 0, используется размер по умолчанию, округлённый ядром в большую сторону).

- Поведение:
  1. Заполняет структуру struct tee_ioctl_shm_alloc_data.
  2. Выполняет системный вызов ioctl с командой TEE_IOC_SHM_ALLOC, используя файловый дескриптор контекста.
  3. В случае успеха драйвер ядра возвращает файловый дескриптор для выделенной общей памяти, который сохраняется в shared_mem->fd. Драйвер также возвращает уникальный идентификатор этого блока общей памяти, который сохраняется в shared_mem->id.
  4. Клиентское приложение отвечает за последующий вызов mmap() для shared_mem->fd, чтобы получить виртуальный адрес в пользовательском пространстве, который затем будет присвоен shared_mem->buffer
- Возвращаемое значение:
  TEEC_SUCCESS при успешном выделении обработчика, TEEC_ERROR_GENERIC, если ioctl завершается ошибкой.

3.6.3.16. TEEC_ReleaseSharedMemory

Функция с прототипом

```
TEEC_Result TEEC_ReleaseSharedMemory(TEEC_SharedMemory *shared_mem)
```

освобождает ранее выделенный блок TEEC_SharedMemory, делая его недоступным для дальнейшего использования.

- Параметры:
  - Указатель на структуру TEEC_SharedMemory, которую необходимо освободить.
- Поведение:
  1. Клиентское приложение сначала должно вызвать munmap() для shared_mem->buffer, если он был ранее отображен.
  2. Затем оно должно закрыть shared_mem->fd.
  3. Может потребоваться вызов ioctl (TEE_IOC_SHM_FREE) для информирования драйвера TEE и безопасной ОС о том, что разделяемая память, идентифицируемая shared_mem->id, больше не нужна, позволяя безопасной ОС освободить свои внутренние ресурсы, связанные с этой общей памятью.
- Возвращаемое значение:

TEEC_SUCCESS при успешном освобождении.

3.7. Дизайн и реализация ядра Secure OS

3.7.1. Объекты ядра и дескрипторы

Взаимодействие с ресурсами в ядре построено на основе объектной модели. Каждая отдельная сущность, управляемая ядром, такая как задача, поток, объект виртуальной памяти или канал связи, представляется в виде объекта.

Доступ к этим объектам и операции над ними осуществляются исключительно через идентификаторы - дескрипторы (handles). Доступ к функциональности объекта определяется этими дескрипторами и связанными с ними правами.

3.7.1.1. Задачи (Процессы)

Задачи являются основными единицами изоляции в Защищенной ОС. Каждая задача обладает собственным виртуальным адресным пространством (vm_space) и управляет одним или несколькими потоками исполнения.

Для Доверенных Приложений исполняемый код задачи располагается в пользовательской части ее защищенного адресного пространства.

Задачи имеют таблицу дескрипторов объектов ядра (handle_table) - индивидуальный для каждой задачи реестр дескрипторов ко всем объектам ядра, к которым она авторизована для доступа.

Жизненный цикл задачи включает несколько ключевых этапов, управляемых ядром:

1. Создание (task_create_empty):
   Ядро выделяет память и инициализирует основную структуру task, и адресное пространство для процесса.

2. Запуск (task_run):
   Состояние задачи изменяется на SPAWNED, создается ее главный поток и добавляется в очередь готовых к выполнению потоков планировщика

3. Уничтожение task_destroy:
   Все виртуальное адресное пространство задачи освобождается, вся динамически выделенная память ядра, связанная с задачей, возвращается системе, со всех объектов ядра связвнных с процессом отнимается единица в счетчике ссылок

3.7.1.2. Потоки

Каждая задача включает как минимум один поток, который выступает единицей исполнения в контексте этой задачи. Потоки выполняют код в изолированном виртуальном адресном пространстве своей родительской задачи.

При создании пользовательского потока ядро выделяет и инициализирует его контекст исполнения, включая регистры ЦП, стек в адресном пространстве задачи и другие специфичные для окружения атрибуты.

Во время создания задачи ядро создает главный поток, устанавливая его начальный

указатель инструкций на указанную точку входа задачи. Этот главный поток затем добавляется в список потоков задачи и становится доступным для планировщика.

3.7.1.3. Каналы

Защищенная ОС использует объекты channel для межзадачного взаимодействия и связи задач с ядром.

Каналы представляют собой двунаправленные примитивы связи. Экземпляры каналов создаются через объект фабрики, посредством системного вызова channel_create Обмен данными через каналы опосредован дескрипторами, что обеспечивает проверку прав доступа для обмена сообщениями.

Каждый канал состоит из двух конечных точек (одна для приема - rx, другая для передачи - tx), которые могут принадлежать одной задаче или быть распределены между двумя разными, обеспечивая различные схемы межпроцессного взаимодействия (IPC).

3.7.1.4. Объекты виртуальной памяти (VMO)

Объекты виртуальной памяти (VMO) представляют собой непрерывную область памяти, подкрепленную физической памятью, которая может быть отображена в виртуальное адресное пространство задачи.

Защищенная ОС предоставляет системный вызов vmo_create позволяющий Доверенным Приложениям запрашивать создание нового VMO через объект фабрики.

После создания дескриптор VMO регистрируется в таблице дескрипторов запросившей задачи. Этот дескриптор ассоциируется с определенными правами котрые определяют допустимые операции над VMO, такие как чтение, запись, отображение или дублирование дескриптора.

3.7.1.5. Примитивы синхронизации

Ядро Защищенной ОС предоставляет набор примитивов синхронизации для управления параллелизмом и защиты общих ресурсов:

- Спинлоки:
  механизм блокировки для коротких критических секций, используемый предотвращения одновременного доступа к общим данным, без вытеснения на время критической секции.

- Мьютексы:
  Блокировки взаимного исключения, используемые для защиты структур общих данных от одновременного доступа несколькими потоками, позволяя потоку быть вытесненым, если мьютекс уже занят.

- Семафоры:
  Счетные семафоры для контроля доступа к ресурсу с ограниченным количеством экземпляров или для сигнализации между потоками.

- Очереди ожидания:
  Механизм предоставляет абстракцию для потоков, позволяющую им блокироваться до наступления определенных событий на одном или нескольких

объектах ядра, например, прихода данных в канал или изменения состояния другого объекта.

3.7.2. Управление задачами

3.7.2.1. Модель процессов

Подсистема управления задачами в Безопасной ОС отвечает за создание, исполнение и завершение задач. Каждая задача функционирует в собственном выделенном виртуальном адресном пространстве и обладает уникальным набором ресурсов, управляемых ядром. Эти ресурсы регистрируются в ядре, а доступ к ним осуществляется через дескрипторы в соответствии с мандатной моделью безопасности системы.

Создание задачи инициируется запросом от существующего процесса пользовательского пространства, службы ядра (Корневой задачи) или в рамках установления сессии для нового ДП. Этот процесс опосредуется системным вызовом task_create(). При поступлении такого запроса ядро выполняет несколько шагов:

1. Выделяются и инициализируются необходимые структуры данных для новой задачи, включая ее основной поток управления.

2. Ядро инициализирует vm_space задачи, подготавливая его для загрузки кода и данных ДП.

3. Новый объект задачи связывается с планировщиком.

Изначально задача переводится в состояние TASK_CREATED. В этом состоянии она остается до тех пор, пока не будет вызвана функция tsk_spawn(). Вызов tas_spawn() перводит задачу в состояние TASK_SPAWNED, после чего создается ее главный поток, и он становится доступным для планирования.

3.7.2.2. Служба IPC

Межпроцессное взаимодействие (IPC) в Безопасной ОС реализуется через механизм каналов. Каналы служат примитивом IPC, обеспечивая структурированный обмен сообщениями между задачами.

При создании канала задача получает дескрипторы к двум его конечным точкам. Одна из них позволяет отправлять сообщения (channel_send(), другая - принимать (cannel_read()). Доступ к этим операциям регулируется мандатами, связанными с соответствующими дескрипторами.

Для сценариев, где задаче требуется одновременно ожидать события или сообщения от нескольких источников, система предоставляет системный вызов wait_object_many(). Это позволяет задаче ожидать на нескольких объектах (например, нескольких принимающих конечных точках каналов).

3.7.2.3. Корневая задача

Корневая задача - это ключевая задача в Безопасной ОС. Она функционирует в непрерывном цикле, обрабатывая входящие запросы и сообщения. Эти сообщения могут поступать от других задач Безопасного мира, служб ядра или из Обычного мира.

По сути, Корневая задача выступает центральным узлом для организации взаимодействий между Безопасной ОС и Обычным миром, а также для координации критически важных внутренних операций Безопасного мира.

3.7.3. Подсистема управления памятью

Подсистема управления памятью (MMS) - это критически важный компонент Безопасной ОС, отвечающий за интерфейс для выделения и управления памятью как в адресном пространстве ядра, так и в адресных пространствах пользовательских задач (Доверенных Приложений).

3.7.3.1. Безопасный аллокатор памяти

Для динамического выделения памяти ядро Безопасной ОС использует механизмы, адаптированных к различным потребностям:

1. Аллокаторы уровня страниц:
   Для управления крупными областями памяти и виртуальной памятью ядро использует аллокаторы страниц.

2. Аллокатор кучи ядра:
   Для небольших, часто выделяемых и освобождаемых блоков памяти в собственном адресном пространстве ядра используется аллокатор основанный на slab-алгоритме. Он подходит для управления структурами данных ядра, такими буферы сообщений IPC.

3.7.4. Файловая система

Безопасная ОС включает минималистичную файловую систему, в первую очередь предназначенную для поддержки загрузки исполняемых файлов Доверенных Приложений (ДП).

3.7.4.1. Линейная файловая система в ОЗУ

Реализована линейная файловая система в ОЗУ (RAM fs). Это означает очень простую, резидентную в памяти файловую систему, где файлы по сути являются непрерывными блоками данных, размещенными в ОЗУ и управляемыми Безопасной ОС.

Она не обладает сложной иерархией каталогов или возможностями постоянного хранения данных. Ее основное назначение - предоставить пространство имен и механизм, позволяющий ядру находить и получать доступ к двоичным файлам ДП.

Термин линейная подразумевает, что данные файла хранятся в едином непрерывном сегменте памяти, что упрощает доступ после того, как известны их местоположение и размер.

3.7.4.2. ELF-файлы

Файловая система Безопасной ОС способна хранить и предоставлять доступ к файлам формата ELF. Бинарные файлы ДП компилируются в формат ELF.

Когда требуется запустить новое ДП (например, в рамках запроса TEEC_OpenSession), подсистема управления задачами ядра взаимодействует с этой файловой системой в

ОЗУ для выполнения следующих действий:

1. Находит двоичный файл ELF, соответствующий запрошенному UUID Доверенного Приложения.

2. Считывает содержимое ELF-файла из RAM FS.

3. Парсит заголовки ELF для определения структуры ДП, включая его сегменты кода, данных, BSS и точку входа.

4. Загружает эти сегменты в виртуальное адресное пространство созданной задачи.

Такая интеграция позволяет Безопасной ОС динамически загружать и исполнять ДП, хранящиеся в виде ELF-файлов в ее контролируемой файловой системе в ОЗУ.

3.8. Capability-Based Модель Безопасности

3.8.1. Мандатный и Дискреционный контроль доступа

3.8.1.1. Мандатный контроль доступа

Термин мандатное управление доступом (Mandatory Access Control, MAC) описывает политику безопасности, в рамках которой права доступа субъектов (например, процессов или пользователей) к объектам (таким как файлы или области памяти) регламентируются централизованно, на основе предварительно определённых классификаций (меток), присвоенных этим субъектам и объектам. Субъектам предоставляется доступ к объектам только в том случае, если их метка безопасности удовлетворяет правилам политики для метки объекта, и это решение не зависит от воли владельца объекта. Такой подход обеспечивает соблюдение общесистемных политик контроля информационных потоков.

Основные преимущества MAC - это возможность обеспечить строгую, недискреционную (принудительную) безопасность. Такой подход оптимален для систем с высокими требованиями к надёжности, поскольку он предотвращает несанкционированную утечку информации и противостоит попыткам вмешательства со стороны программ пользовательского уровня. Политики безопасности централизованы, что гарантирует их единообразное применение в масштабах всей системы.

Тем не менее, у MAC-систем есть и недостатки. Они зачастую негибки, так как политики безопасности сложны в определении, реализации и адаптации к изменяющимся требованиям. Эта сложность может приводить к увеличению объёма доверенной вычислительной базы, особенно если логика принудительного применения политик реализуется на уровне ядра, что является существенным недостатком для микроядерных архитектур.

MAC-политики также могут вызывать дополнительные накладные расходы и снижать производительность из-за необходимости проверки меток при каждом обращении к ресурсам.

Важно отметить, что классические MAC-модели часто не предоставляют механизмов для ограничения прав доступа при передаче дескрипторов (или ссылок) на объекты между субъектами: если субъект уполномочен политикой MAC получить доступ к объекту, он может передать это право другому субъекту, как правило, без возможности дальнейшего гранулярного сужения прав в рамках самой MAC-политики.

3.8.1.2. Дискреционный контроль доступа

Дискреционное управление доступом (Discretionary Access Control, DAC) - это политика безопасности, при которой права доступа к объектам определяются их владельцами. Владелец объекта сам решает, какие субъекты (например, пользователи или процессы) могут получить доступ к объекту и какие операции (чтение, запись, выполнение) им разрешены. Распространённые реализации включают списки контроля доступа (Access Control Lists, ACL) и биты разрешений.

Главные преимущества DAC - это гибкость и простота использования. Владельцы объектов могут динамически управлять разрешениями, что упрощает совместное использование ресурсов.

Несмотря на гибкость, DAC имеет существенные недостатки, особенно для защищённых сред. DAC обеспечивает более слабую защиту от злонамеренных действий скомпрометированных учётных записей пользователей, поскольку авторизованный субъект потенциально может передать свои права доступа неавторизованным субъектам.

Отсутствие централизованного контроля может привест к непоследовательному применению политик безопасности в системе. Строгое соблюдение принципа наименьших привилегий в рамках DAC затруднительно, так как субъекты могут сохранять более широкие разрешения, чем это действительно необходимо.

Кроме того, фундаментальным ограничением DAC является отсутствие механизмов для ограничения прав доступа самого владельца объекта: владелец всегда обладает полным контролем и не может быть ограничен в своих действиях над объектом средствами самого DAC.

Для доверенной среды исполнения (TEE) одного лишь DAC, как правило, недостаточно из-за этих уязвимостей.

3.8.1.3. сочетание гранулярности DAC, строгости MAC и устранение их недостатков

Модель безопасности на основе capability стремится объединить сильные стороны традиционных парадигм управления доступом, нивелируя при этом их индивидуальные недостатки и предлагая подход, отличный от чистых реализаций MAC или DAC. Capability в этом контексте сочетают как гранулярное, динамическое управление, присущее DAC, так и строгое, недискреционное применение политик, характерное для MAC.

Данный подход реализует принцип наименьших привилегий. Доверенное приложение (TA) получает дескрипторы (handles) - по сути, неподделываемые идентификаторы, инкапсулирующие мандаты, - только для тех конкретных объектов и ресурсов, которые ему действительно необходимы, и только с теми разрешениями, которые требуются для выполнения его функций, как это определено в его манифесте. Приложение не может обращаться к объектам (даже именовать их), если ему не был явно выдан соответствующий дескриптор. Эта особенность естественным образом устраняет недостаток DAC, связанный с невозможностью ограничить права владельца объекта: в данной системе само понятие владельца с неограниченными правами отсутствует. Любой субъект, включая создателя объекта, взаимодействует с этим объектом исключительно через дескрипторы с чётко определёнными разрешениями.

Модель обладает гибкостью и гранулярностью, присущими DAC, так как дескрипторы ссылаются на конкретные экземпляры объектов и могут динамически создаваться и делегироваться (например, корневой задачей или через объект-фабрики). Однако, в отличие от традиционного DAC, ядро Безопасной ОС (как часть TCB) централизованно управляет выдачей и проверкой этих мандатов, гарантируя невозможность их подделки или несанкционированного расширения прав со стороны доверенных приложений.

Более того, модель решает проблему MAC, касающуюся передачи прав: при делегировании дескриптора - можно создать его копию с урезанным набором прав. Таким образом, получатель такого ослабленного дескриптора будет обладать лишь явно предоставленными ему урезанными полномочиями, даже если исходный дескриптор у делегирующего субъекта был более полным. Это обеспечивает недискреционный контроль над доступом к ресурсам и их делегированием, аналогично MAC, но с дополнительной возможностью гранулярного управления правами при передаче.

С точки зрения размера TCB, система на основе capability может быть значительно компактнее, чем полномасштабная реализация MAC. Основная роль ядра в контроле доступа сводится к управлению и проверке дескрипторов и связанных с ними прав, а не к интерпретации сложной глобальной базы данных политик MAC. Проверка дескриптора при предъявлении является быстрой операцией, сводящейся к сравнению битов разрешений.

3.8.2. Дескрипторы как инкапсулированные мандаты

3.8.2.1. Обоснование проектного решения

В Безопасной ОС дескрипторы (handles) служат непрозрачными, неподделываемыми ссылками на системные ресурсы, управляемые ядром. Каждый такой дескриптор неразрывно связан с конкретным объектом и инкапсулирует набор разрешений, которые предопределяют допустимые операции над этим объектом со стороны владельца дескриптора.

На внутреннем уровне эти пользовательские дескрипторы отображаются на структуры, управляемые ядром. Эти внутренние структуры хранят состояние объекта, битовую маску предоставленных прав.

Такой механизм гарантирует, что любое взаимодействие с системными ресурсами опосредуется ядром, которое проверяет правомочность запрашиваемой операции на основе разрешений, связанных с предъявленным дескриптором.

В основе архитектуры безопасности, построенной на мандатах (полномочиях), лежат несколько ключевых принципов:

- Принцип наименьших привилегий:
  Модель безопасности по своей сути обеспечивает соблюдение этого принципа. Задачи и Доверенные Приложения получают строго минимальный набор дескрипторов с точно очерченными разрешениями, необходимыми исключительно для выполнения их целевых функций. Это существенно ограничивает потенциальный ущерб от скомпрометированного компонента, поскольку он лишен полномочий для доступа или модификации неавторизованных ресурсов.

- Гранулированный контроль доступа:

Модель предоставляет точный и детализированный контроль над доступом к ресурсам. Каждый дескриптор определяет не только, к какому объекту разрешен доступ, но и какие конкретно операции (методы) могут быть над ним выполнены. Это позволяет определять и применять детальные политики безопасности для каждого отдельного объекта.

- Упрощенное определение политик:
  В сравнении с традиционными системами мандатного контроля доступа, часто требующими наличия в ядре базы данных политик и сложных парсеров правил модель на основе capability предлагает более легковесный механизм. Проверка дескриптора включает прямой поиск и проверку битов разрешений, что снижает объём доверенной кодовой базы и минимизирует накладные расходы, связанные с решениями о контроле доступа.

3.8.2.2. Объекты

Объекты являются сущностями, управляемыми и защищаемыми ядром Безопасной ОС. Они представляют все системные ресурсы, включая, задачи (процессы), потоки, сервисы, каналы связи (pipes), объекты виртуальной памяти (VMO), примитивы синхронизации. Каждый объект инкапсулирует свое состояние и логику для операций над ним.

Жизненный цикл каждого объекта (выделение, освобождение, изменение состояния) строго контролируется ядром совместно с его соответствующими менеджерами ресурсов. Ядро использует подсчета ссылок, для отслеживания активных дескрипторов объектов, гарантируя, что ресурсы освобождаются только тогда, когда они больше не используются.

3.8.2.3. Дескрипторы объектов

Дескриптор объекта - это уникальный в контексте задачи идентификатор, служащий маркером, который представляет определенное полномочие: право доступа к базовому объекту ядра с предопределенным набором разрешений.

Это абстракция, используемая клиентскими задачами для обращения к ресурсам, управляемым ядром, без прямого доступа к его структурам данных.

Дескрипторы обладают важными свойствами безопасности:

- Неподделываемость:
  Дескрипторы генерируются и управляются исключительно ядром. Задачи не могут самостоятельно создавать или произвольно изменять дескрипторы. Это предотвращает попытки задач подделать или угадать значения дескрипторов для получения несанкционированного доступа.

- Контролируемое дублирование:
  Хотя ядро может дублировать дескрипторы (например, для совместного доступа или создания дескриптора с урезанными правами), задачи не могут произвольно копировать имеющиеся у них дескрипторы. Создание копии дескриптора - это операция требующаяя соответствующий доступ у объекта фабрики, контролируемый ядром аналогично остальным доступам.

- Контекстная специфичность:
  Значение дескриптора, действительно только в контексте владеющей им задачи.

Сопоставление значения дескриптора с реальным объектом ядра и его разрешениями осуществляется через ядерную таблицу дескрипторов.

3.8.2.4. Объект-фабрика

Безопасная ОС использует концепцию объекта-фабрики для управления созданием новых объектов ядра. Ядром назначается специальный, единственный в своем роде, объект-фабрика.

Чтобы создать новый объект, такой как задача, канал связи или VMO, ДП должно обладать дескриптором на объект-фабрику и вызвать у него метод создания.

Привилегия на создание новых объектов предоставляется не всем. Манифест задачи определяет её начальный набор дескрипторов, который может включать дескриптор на объект-фабрику вместе с разрешениями, указывающими, какие типы объектов эта задача может запрашивать у фабрики. Это гарантирует, что ДП могут создавать только те объекты, которые соответствуют их назначению и политике безопасности.

Сам объект-фабрика создаются ядром на этапе его инициализации. Фабрика существует все время работы системы.

3.8.2.5. Методы объектов

Операции над объектами, такие как чтение из канала или запись в него, отображение VMO в адресное пространство или передача сигнала примитиву синхронизации, предоставляются ДП в виде специфических методов объекта. Эти методы вызываются через системные вызовы, при которых ДП передает дескриптор целевого объекта и параметры для метода.

При каждой попытке вызова метода объекта через дескриптор ядро в обязательном порядке выполняет проверку полномочий. Оно удостоверяется, что предъявленный дескриптор действителен, ссылается на реально существующий объект ожидаемого типа, и что разрешения, инкапсулированные в дескрипторе, явно предоставляют право на выполнение запрошенной операции.

Набор методов, применимых к каждому типу объектов, предопределен и реализован в ядре. ДП не могут определять или присоединять новые, пользовательские методы к существующим типам объектов.

3.8.3. Контроль доступа на основе полномочий

Безопасная ОС реализует строгую политику безопасности, основанную на полномочиях. Эта политика предписывает, что задача может выполнить операцию над объектом только в том случае, если она обладает действительным дескриптором (полномочием) для этого объекта, и разрешения, связанные с этим дескриптором, явно авторизуют предполагаемую операцию. Никакие неявные права не предоставляются. Весь доступ опосредуется через явные полномочия.

3.8.3.1. Разрешения

Поскольку системные вызовы служат механизмом вызова методов объекта, набор допустимых операций для любого типа объектов фиксирован и известен ядру.

Каждый отдельный дескриптор несет собственный, привязанный к нему, набор битов разрешений. Эти биты точно определяют, какие методы владелец именно этого дескриптора может вызывать у связанного объекта. Это позволяет осуществлять гранулированный контроль, при котором разные дескрипторы на один и тот же объект могут обладать различными уровнями привилегий.

Когда дескриптор передается или разделяется между задачами, разрешения, связанные с новым дескриптором, предоставленным получателю, никогда не могут быть шире, чем у исходного разделяемого дескриптора. При распространении разрешения могут быть только сохранены или ограничены (сужены), но не расширены. Это соответствует принципу наименьших привилегий и предотвращает эскалацию привилегий через делегирование.

3.8.3.2. Манифесты задач

Каждое Доверенное Приложение связано с манифестом - декларативной структурой метаданных. Манифест, определяет начальный набор полномочий (дескрипторов), которые ДП получит при своем создании, а также точные разрешения, предоставленные для каждого из этих начальных дескрипторов.

Во время инстанцирования ДП ядро анализирует его манифест. На основе этой спецификации ядро заполняет начальную таблицу дескрипторов ДП указанными дескрипторами и соответствующими им разрешениями. Это устанавливает базовый набор привилегий ДП.

3.8.3.3. Корневая задача

Основная обязанность Корневой Задачи - инстанцирование и инициализация ДП. При запуске нового ДП Корневая Задача отвечает за обеспечение того, чтобы ДП получило свой начальный набор дескрипторов с корректными разрешениями, как указано в манифесте

Так же корневая задача отвечает за коммуникацию между мирами. При получении сообщения, ядро передает его через канал корневой задаче, которая ищет или создает соответствующее ДП и перенаправляет сообщение доверенному приложению.

3.8.3.4. Вызов метода

Вызов метода объекта Доверенным Приложением происходит следующим образом:

1. ДП инициирует системный вызов, предоставляя дескриптор целевого объекта, идентификатор вызываемого метода и любые необходимые параметры.

2. Ядро получает системный вызов и переходит в привилегированный режим выполнения. Сначала оно проверяет предоставленный дескриптор путем поиска его в таблице дескрипторов вызывающего ДП.

3. Если дескриптор действителен, ядро проверяет биты разрешений, связанные с ним, чтобы убедиться, что ДП авторизовано для вызова запрошенного метода у целевого объекта.

4. Если и дескриптор, и разрешения действительны, ядро передает управление соответствующей внутренней функции, реализующей метод объекта, передавая ей

проверенные параметры. В противном случае ДП возвращается код ошибки.

Параметры для методов объектов передаются как часть системного вызова. Это могут быть простые значения, указатели на буферы данных в адресном пространстве ДП или другие дескрипторы. Ядро отвечает за безопасное копирование данных между адресным пространством ДП и пространством ядра (используя семантику copy_from_user/copy_to_user) и за проверку любых дескрипторов, передаваемых в качестве параметров.

3.8.3.5. Аспекты производительности

Основой контроля доступа на основе полномочий являются проверка дескрипторов и прав доступа при каждом вызове метода.

Поиск дескриптора в таблице дескрипторов ДП производится прямым индексированием по идентификатору дескриптора за время O(1).

Проверка прав доступа производится единственной инструкцией битового AND между битовой маской разрешений дескриптора и битовой маской полномочия, соответствующего вызывающему методу.

Таким образом операции остаются легковесными и не вносят значительных задержек в критические пути выполнения, тем самым обеспечивая баланс между надежной безопасностью и производительностью системы.

3.9. Системные вызовы

3.9.1. Защищённые точки входа

3.9.1.1. Общие сведения о системных вызовах ОС

Системный вызов позволяет коду пользовательского режима (в данном случае - Доверенным Приложениям) запрашивать функциональность ядра контролируемым и безопасным образом.

В отличие от традиционных операционных систем, системные вызовы в данной ОС оперируют исключительно дескрипторами (handles). Эти дескрипторы являются ссылками на защищённые объекты, управляемые ядром.

Интерфейс системных вызовов представляет собой единственный шлюз, через который Доверенные Приложения взаимодействуют с объектами и сервисами, управляемыми ядром Защищённой ОС. Каждый такой системный вызов проходит верификацию, использующую таблицу системных вызовов, индексируемую уникальным номером вызова. Этот механизм входа гарантирует, что все запросы к сервисам ядра проходят проверку перед исполнением.

3.9.1.2. Жизненный цикл системного вызова

Каждый системный вызов следует каноническому жизненному циклу объектов Защищённой ОС, обеспечивая единообразную и безопасную обработку:

1. Инициация вызова:
   ДП инициирует системный вызов, передавая один или несколько дескрипторов

объектов и любые требуемые операционные параметры (например, буферы данных, флаги).

2. Разрешение дескриптора и проверка полномочий:
   При входе в ядро Защищённая ОС осуществляет поиск предоставленных дескрипторов в таблице дескрипторов, принадлежащей вызывающему ДП. Затем ядро проверяет, предоставляют ли полномочия (capability), связанные с дескриптором, право на выполнение запрошенной операции над целевым объектом.

3. Атомарное выполнение операции:
   Если проверка полномочий проходит успешно, ядро выполняет запрошенную операцию. Операции спроектированы как атомарные с точки зрения ТА для поддержания целостности системы.

4. Определение результата и прав владения:
   Результат операции, включая любые вновь созданные дескрипторы объектов или данные, подготавливается к возврату.

5. Возврат в ДП:
   Результат операции (успех или код ошибки) возвращается ДП. Обо всех ошибках, возникших на этапах разрешения дескриптора, проверки полномочий или выполнения операции, также сообщается вызывающей стороне.

3.9.1.3. Формат передачи аргументов

Аргументы защищённых системных вызовов передаются в соответствии со стандартным соглашением о вызовах RISC-V. Это включает использование регистров общего назначения для номеров системных вызовов, идентификаторов дескрипторов, длин данных и специфичных для полномочий флагов. Также могут передаваться указатели на буферы данных в пространстве пользователя (в пределах адресного пространства ДП).

Для передачи данных между ДП и ядром или при работе с областями разделяемой памяти Защищённая ОС использует выделенные безопасные примитивы копирования (copy_from_user и copy_to_user) для предотвращения прямого доступа ядра к потенциально вредоносной или некорректно сформированной памяти ДП.

3.9.1.4. Проверка разрешений дескриптора

Ключевым этапом жизненного цикла системного вызова является проверка разрешений (полномочий) дескриптора. При каждом вызове системного вызова ядро проверяет, обладает ли вызывающее ДП необходимыми полномочиями для выполнения операции над указанным объектом.

Этот процесс полностью опирается на базовую объектную модель на основе полномочий:

- Дескрипторы - это 32-битные значения, которые ядро преобразует во внутренние указатели на объекты ядра путем поиска в таблице объектов (дескрипторов), принадлежащей конкретному ДП.

- Каждая запись в таблице объектов связывает дескриптор с объектом ядра и соответствующей маской разрешений (полномочий).

- Любой доступ к объекту, инициированный системным вызовом, вызывает проверку типа объекта и разрешений, предоставленных дескриптором.

Если поиск дескриптора завершается неудачей (например, дескриптор недействителен или отсутствует в таблице ДП) или если маска разрешений, связанная с дескриптором, не содержит необходимых флагов полномочий, системный вызов немедленно прерывается, и ДП возвращается код ошибки.

Такая проверка полномочий для каждого объекта и каждого действия системного вызова обеспечивает надёжное применение модели безопасности и гарантирует, что:

- Доступ к объектам никогда не бывает неявным. Он должен быть авторизован через дескриптор, явно указанный и разрешённый в манифесте задачи или правомерно полученный иным способом.

- Отсутствуют сторонние полномочия. ДП не могут выполнять операции, на которые у них нет явных мандатов.

- Предотвращаются утечки объектов или полномочий между границами ДП, поскольку дескрипторы специфичны для контекста, а полномочия на передачу строго контролируются.

- Происхождение и путь доступа к каждому ресурсу остаются отслеживаемыми через систему дескрипторов.

3.9.2. Спецификация системных вызовов

3.9.2.1. SYS_LOG

Интерфейс журналирования, предназначенный для вывода отладочной информации из защищённого мира.

- Параметры:
  - data:
    Указатель в пользовательском пространстве Доверенного Приложения на строку для журналирования.
  - size:
    Длина строки.

- Поведение:
  Копирует сообщение для журнала из пользовательского пространства ДП во внутренний буфер ядра. Выводит содержимое буфера на консоль, используя механизм ядра printk. Контролирует максимальный размер сообщения для предотвращения переполнения буфера.

- Возвращаемое значение:
  0 в случае успеха или код ошибки (-EINVAL при превышении допустимого размера, -EFAULT при недействительном указателе в пользовательском пространстве).

3.9.2.2. SYS_VMO_CREATE

Запрашивает через дескриптор объекта-фабрики создание объекта виртуальной памяти

(VMO) заданного размера (в страницах).

Дескриптор нового VMO сохраняется в области памяти, доступной для записи из пользовательского режима.

- Параметры:
  - handle:
    Дескриптор объекта-фабрики.
  - count:
    Требуемый размер нового VMO в страницах.
  - out:
    Указатель в пользовательском пространстве для сохранения дескриптора созданного VMO.

- Требуемые полномочия:
  Дескриптор handle фабрики должен обладать полномочием FACTORY_CREATE_VMO_CAP.

- Поведение:
  Системный вызов получает объект-фабрику по предоставленному дескриптору и проверяет наличие необходимого полномочия. Вызывает функцию vm_object_allocate() для создания нового VMO указанным размером count страниц. В таблице дескрипторов вызывающей задачи выделяется новый дескриптор для созданного VMO с полномочиями OBJECT_CAP_TRANSFER | VMO_CAP_FULL. Копирует новый дескриптор VMO по адресу, указанному в out.

- Возвращаемое значение:
  0 в случае успеха или код ошибки (-EINVAL, -ENOMEM, -EFAULT).

3.9.2.3. SYS_CHANNEL_CREATE

Запрашивает у объекта-фабрики создание пары двунаправленных конечных точек канала. Системный вызов возвращает два новых дескриптора, каждый из которых ссылается на одну из конечных точек только что созданного, связанного между собой, канала.

- Параметры:
  - factory:
    Дескриптор объекта-фабрики.
  - rx:
    Указатель в пользовательском пространстве для сохранения дескриптора одной из конечных точек канала.
  - tx:
    У казатель в пользовательском пространстве для сохранения дескриптора другой конечной точки канала.

- Требуемые полномочия:
  Дескриптор factory должен обладать полномочием FACTORY_CREATE_CHANNEL_CAP.

- Поведение:
  Получает объект-фабрику и проверяет его полномочия. Вызывает channel_create()

для установления новой пары конечных точек канала. В таблице дескрипторов вызывающей задачи выделяются два новых дескриптора для этих конечных точек, каждому предоставляются полномочия OBJECT_CAP_TRANSFER | OBJECT_CAP_WAIT | CHANNEL_CAP_FULL. Копирует эти новые дескрипторы по адресам, указанным в rx и tx.

- Возвращаемое значение:
  0 в случае успеха или код ошибки (-EINVAL, -ENOMEM, -EFAULT).

3.9.2.4. SYS_CHANNEL_READ

Предпринимает попытку извлечь ожидающее сообщение (данные или, дескрипторы) из конечной точки канала, указанной дескриптором.

- Параметры:
  - h:
    Дескриптор конечной точки канала, из которой производится чтение.
  - data:
    Указатель на буфер в пользовательском пространстве для сохранения полученных данных сообщения.
  - size:
    Размер буфера data.
  - handles:
    Указатель на массив в пользовательском пространстве для сохранения полученных дескрипторов.
  - handles_count:
    Указатель в пользовательском пространстве на переменную. До вызоыва содержит ёмкость массива handles, После возврата - фактическое количество полученных дескрипторов.

- Требуемые полномочия:
  Дескриптор h конечной точки канала должен обладать полномочием CHANNEL_CAP_RECV.

- Поведение:
  Получает объект конечной точки канала и проверяет полномочие на приём. Если handles_count предоставлен, копирует значение ёмкости из пользовательского пространства. Вызывает channel_receive() для извлечения сообщения из очереди ядра, включая любые прикреплённые дескрипторы. Копирует полученные данные сообщения в пользовательский буфер data, но не более size байт. Если handles и handles_count были предоставлены, копирует полученные дескрипторы в пользовательский массив handles. После копирования освобождает данные сообщения, хранившиеся на стороне ядра.

- Возвращаемое значение:
  0 в случае успеха или код ошибки (-EINVAL, -EFAULT или ошибки от channel_receive).

3.9.2.5. SYS_CHANNEL_WRITE

Помещает в очередь сообщение (данные и, опционально, дескрипторы) для отправки через конечную точку канала, указанную дескриптором.

- Параметры:
  - h:
    Дескриптор конечной точки канала, в которую производится запись.
  - data:
    Указатель на данные сообщения в пользовательском пространстве для отправки.
  - size:
    Размер данных сообщения.
  - handles:
    Указатель на массив дескрипторов в пользовательском пространстве для передачи вместе с сообщением.
  - handles_count:
    Количество дескрипторов в массиве handles.

- Требуемые полномочия:
  Дескриптор h конечной точки канала должен обладать полномочием CHANNEL_CAP_SEND. Передаваемые дескрипторы (в массиве handles) проверяются на наличие полномочия OBJECT_CAP_TRANSFER.

- Поведение:
  Если handles_count не равен нулю, копирует дескрипторы из пользовательского массива handles во внутренний буфер ядра. Выделяет память в ядре и копирует данные сообщения data из пользовательского пространства. Получает объект конечной точки канала и проверяет полномочие на отправку. Вызывает channel_send() для помещения в очередь копии данных и дескрипторов, находящихся в ядре. После помещения в очередь освобождает находящуюся в ядре копию данных сообщения.

- Возвращаемое значение:
  0 в случае успеха или код ошибки (-EINVAL, -ENOMEM, -EFAULT).

3.9.2.6. SYS_TASK_CREATE

Запрашивает у объекта-фабрики создание нового, пустого объекта задачи (заготовки процесса) с заданным именем. Дескриптор новой задачи сохраняется в памяти, доступной для записи из пользовательского режима.

- Параметры:
  - factory:
    Дескриптор объекта-фабрики.
  - name:
    Указатель в пользовательском пространстве на имя новой задачи.
  - size:
    Длина имени задачи.
  - out:
    Указатель в пользовательском пространстве для сохранения дескриптора созданной задачи.

- Требуемые полномочия:
  Дескриптор factory должен обладать полномочием FACTORY_CREATE_TASK_CAP.

- Поведение:

Копирует имя задачи name из пользовательского пространства. Получает объект-фабрику и проверяет его полномочия. Вызывает task_create_empty() для создания структуры новой задачи и её виртуального адресного пространства. Задача изначально находится в состоянии TASK_CREATED. В таблице дескрипторов вызывающей задачи выделяется новый дескриптор для созданной задачи с полномочиями TASK_GET_SPACE_CAP | OBJECT_CAP_TRANSFER. Копирует новый дескриптор задачи по адресу, указанному в out.

- Возвращаемое значение:
  0 в случае успеха или код ошибки (-EINVAL, -ENOMEM, -EFAULT).

3.9.2.7. SYS_VM_MAP_VMO

Отображает существующий объект виртуальной памяти, в целевое виртуальное адресное пространство задачи (VM space), по указанному базовому адресу с заданными правами доступа.

- Параметры:
  - vmspace:
    Дескриптор целевого объекта VM space.
  - vmo:
    Дескриптор VMO, который необходимо отобразить.
  - prot:
    Флаги защиты для отображения (VMA_READ, VMA_WRITE, VMA_EXECUTE).
  - base:
    Базовый виртуальный адрес для отображения в целевом VM space.

- Требуемые полномочия:
  Дескриптор vmspace должен обладать полномочием VMSPACE_VMO_MAP_CAP.
  Дескриптор vmo должен обладать полномочиями, соответствующими запрошенным флагам prot (например, VMO_CAP_READ для VMA_READ).

- Поведение:
  Получает объект VM space и проверяет его полномочие на отображение. Получает VMO и проверяет его полномочия на соответствие запрошенным правам доступа. Вызывает vm_space_map_vmo() для выполнения операции отображения.

- Возвращаемое значение:
  Фактический базовый виртуальный адрес, по которому был отображён VMO, в случае успеха, или код ошибки, приведённый к типу unsigned long (например, (unsigned long)NULL или другие индикаторы ошибки).

3.9.2.8. SYS_VM_FREE

Отменяет отображение диапазона виртуальных адресов из VM space задачи, указанного дескриптором.

- Параметры:
  - vmspace:
    Дескриптор объекта VM space, из которого отменяется отображение.
  - base:
    Базовый виртуальный адрес диапазона для отмены отображения.

    - size:
      Размер диапазона для отмены отображения.

- Требуемые полномочия:
  Дескриптор vmspace должен обладать полномочием VMSPACE_ALLOCATE_CAP

- Поведение:
  Получает объект VM space и проверяет его полномочия. Вызывает vm_free() для отмены отображения указанного диапазона виртуальных адресов.

- Возвращаемое значение:
  0 в случае успеха, или -EINVAL, если дескриптор vmspace недействителен или не имеет полномочий.

3.9.2.9. SYS_TASK_SPAWN

Запускает ранее созданную задачу. Это включает инициализацию её главного потока с указанной точкой входа и перевод состояния задачи из TASK_CREATED в TASK_SPAWNED, делая её доступной для планировщика.

- Параметры:

    - t:
      Дескриптор объекта задачи для запуска.
    - ep:
      Адрес (виртуальный) точки входа для главного потока задачи t.

- Поведение:
  Получает целевой объект задачи new по дескриптору t. Проверяет, что задача находится в состоянии TASK_CREATED. Изменяет права доступа к странице дескрипторов задачи на чтение для пользователя (VMA_READ | VMA_USER). Вызывает task_create_initial_thread() для настройки главного потока с точкой входа ep и политикой планирования Round-Robin. Добавляет новый поток в список потоков задачи. Вызывает task_run() для перевода состояния задачи в TASK_SPAWNED (делая её готовой к выполнению).

- Возвращаемое значение:
  0 в случае успеха или код ошибки (-EINVAL, если задача не в состоянии TASK_CREATED, или -ENOMEM).

3.9.2.10. SYS_OBJECT_CLOSE

Освобождает указанный дескриптор в таблице объектов вызывающей стороны. Это уменьшает счётчик ссылок на соответствующий объект ядра. Если счётчик ссылок достигает нуля, объект удаляется.

- Параметры:

    - handle:
      Дескриптор, который должен быть закрыт.

- Поведение:
  Вызывает object_table_remove() для удаления указанного handle из таблицы дескрипторов текущей задачи и управления жизненным циклом нижележащего

объекта.

- Возвращаемое значение:
  0 в случае успеха или код ошибки, если дескриптор недействителен.

3.9.2.11. SYS_OBJECT_WAIT_MANY

Ожидает на одном или нескольких объектах ядра, указанных массивом записей ожидания, до тех пор, пока один или несколько указанных сигналов не будут установлены на соответствующих объектах.

- Параметры:
  - u_entries:
    Указатель в пользовательском пространстве на массив структур struct wait_entry. Каждая запись содержит handle объекта и сигналы, которые необходимо ожидать на этом объекте.
  - count:
    Количество записей в массиве u_entries.
- Требуемые полномочия:
  Каждый дескриптор в u_entries должен обладать полномочием OBJECT_CAP_WAIT.
- Поведение:
  Копирует массив wait_entry из пользовательского пространства. Для каждой записи получает объект ядра по дескриптору и проверяет полномочие OBJECT_CAP_WAIT. Инициализирует объект completion и object_waiter для каждого ожидаемого объекта, связывая их с общим объектом completion. Блокирует вызывающий поток, ожидая на объекте completion (completion_wait()). После разблокировки (когда произошёл сигнал) получает установленные сигналы для каждого объекта с помощью object_waiter_finish(). Копирует обновлённый массив wait_entry (с установленными сигналами) обратно в пользовательское пространство.
- Возвращаемое значение:
  0 в случае успеха или код ошибки (-EINVAL при неверном количестве, недействительных дескрипторах или полномочиях, -EFAULT).

3.9.2.12. SYS_OBJECT_COPY

Дублирует существующий дескриптор в рамках той же задачи, назначая новому дескриптору потенциально ограниченный набор полномочий. Используется для создания дескрипторов с урезанными правами.

- Параметры:
  - handle:
    Существующий дескриптор для дублирования.
  - caps:
    Требуемая маска полномочий для нового дескриптора. Эта маска должна быть подмножеством или равна полномочиям исходного дескриптора handle.
  - out:
    Указатель в пользовательском пространстве для сохранения нового

(дублированного) дескриптора.

- Требуемые полномочия:
  Исходный дескриптор handle должен обладать всеми полномочиями, указанными в параметре caps для копии.

- Поведение:
  Получает нижележащий объект, связанный с исходным дескриптором handle, и проверяет, что его существующие полномочия покрывают запрошенные caps. Вызывает object_table_alloc_handle() для создания новой записи дескриптора в таблице дескрипторов текущей задачи для того же объекта, но с указанными caps. Копирует значение нового дескриптора по адресу, указанному в out.

- Возвращаемое значение:
  0 в случае успеха или код ошибки (-EINVAL, если исходный дескриптор недействителен или запрошенные caps не разрешены, -EFAULT).

3.10. Фреймворк доверенных приложений

3.10.1. Стандартная библиотека для доверенных приложений

3.10.1.1. Обзор стандартной библиотеки

Платформа доверенных приложений предоставляет легкую, безопасную среду выполнения и интерфейс разработки для написания доверенных приложений пользовательского режима, которые выполняются в защищенной операционной системе.

Платформа определяет стандартную среду выполнения C, дополненную системными возможностями, доступными через модель полномочий на основе дескрипторов.

Основная роль платформы заключается в содействии безопасной разработке программного обеспечения, гарантируя, что приложения будут соответствовать системным политикам безопасности и эффективно работать в условиях ограничений производительности надежной среды выполнения.

3.10.2. Спецификация операций с дескрипторами

3.10.2.1. Функции для работы с каналами

- channel_read:
  Считывает сообщение из конечной точки канала, указанной в дескрипторе. Эта функция является обёрткой для системного вызова SYS_CHANNEL_READ.

- channel_write:
  Отправляет сообщение в конечную точку канала, заданную дескриптором. Эта функция является обёрткой для системного вызова SYS_CHANNEL_WRITE.

- channel_from_handle:
  Выполняет безопасное приведение общего дескриптора к типу дескриптора канала на уровне библиотеки. Не вызывает системный вызов.

3.10.2.2. Функции фабрики

- factory_init:
  Функция извлекает дескриптор уже существующего объекта-фабрики, делая его доступным для ДП.

- factory_create_vmo:
  Вызывает системный вызов SYS_VMO_CREATE, используя дескриптор фабрики, чтобы запросить у ядра создание нового объекта виртуальной памяти заданного размера. Возвращает дескриптор созданного объекта.

- factory_channel_create:
  Оборачивает системный вызов SYS_CHANNEL_CREATE.

- factory_task_create:
  Создаёт новую, неинициализированную структуру задачи в ядре, обращаясь к системному вызову SYS_TASK_CREATE через дескриптор фабрики.

3.10.2.3. Функции для работы с объектами

- object_copy:
  Дублирует дескриптор, вызывая системный вызов SYS_OBJECT_COPY.

- object_close:
  Освобождает дескриптор через вызов SYS_OBJECT_CLOSE.

3.10.2.4. Функции для работы с задачами

- task_spawn:
  Запускает ранее созданную задачу, идентифицированную по её дескриптору. Функция оборачивает системный вызов SYS_TASK_SPAWN.

- task_share_handle:
  Предоставляет высокоуровневый механизм для безопасной передачи дескриптора другой задаче. Является обёрткой, которая использует функцию channel_write для отправки дескриптора через предварительно установленный канал связи.

3.10.2.5. Функции управления памятью

- vm_init:
  Получает дескриптор объекта виртуального адресного пространства, принадлежащего вызывающей задаче.

- vm_map_vmo:
  Отображает объект виртуальной памяти в адресное пространство задачи. Функция оборачивает системный вызов SYS_VM_MAP_VMO.

- vm_free:
  Отменяет отображение региона виртуальной памяти из адресного пространства задачи. Вызывает SYS_VM_FREE.

3.10.3. Спецификация стандартной библиотеки ввода-вывода

3.10.3.1. Функции, форматированного вывода

- printf:
  Стандартная функция для форматированного вывода строк. Вызывает системный вызов SYS_LOG.

- sprintf:
  Стандартная функция для форматированного вывода строк в буфер.

- vprintf:
  Стандартная функция для форматированного вывода строк в буфер с переменным числом аргументов.

3.10.3.2. Функция журналирования

- tee_log:
  Механизм безопасного журналирования для ДП. Функция вызывает системный вызов SYS_LOG.

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