Application-Processor Trusted Execution Environment (AP-TEE) for Confidential Computing on RISC-V platforms

Editor - Ravi Sahita, RISC-V AP-TEE Task Group

Version 1.0, 5/2022: This document is in development. Assume everything can change. See http://riscv.org/spec-state for details.

Table of Contents

Preamble

This document is in the Development state

Assume everything can change. This draft specification will change before being accepted as standard, so implementations made to this draft specification will likely not conform to the future standard.

Copyright and license information

Contributors

The proposed AP-TEE specifications (non-ratified, under discussion) have been contributed to directly or indirectly by (in alphabetical order):

Andrew Bresticker, Andy Dellow, Atish Patra, Atul Khare, Beeman Strong, Dingji Li, Dong Du, Dylan Reid, Guerney Hunt, Kailun Qin, Manuel Offenberg, Nick Kossifidis, Ravi Sahita (Editor <ravi@rivosinc.com>), Samuel Ortiz, Vedvyas Shanbhogue, Yann Loisel

Chapter 1. Introduction

This document describes a scalable Application-Processor Trusted Execution Environment (AP-TEE) interface proposal for RISC-V-based platforms. This AP-TEE interface specification enables application workloads that require confidentiality to reduce the Trusted Computing Base (TCB) to a minimal TCB, specifically, keeping the host OS/VMM and other software outside the TCB. The proposed specification supports an architecture that can be used for Application and Virtual Machine workloads, while minimizing changes to the RISC-V ISA and privilege modes.

> The following is a proposal being submitted by Rivos Inc. for the purposes of discussion and collaboration to eventually produce a RISC-V Standard. At this time, this is an initial proposal and is not an official draft of a RISC-V Specification.

Chapter 2. Notation

The key words "MUST", "MUST NOT", "SHOULD", and "SHOULD NOT", in this document are to be interpreted as described in RFC 2119.

MUST	This word, or the terms "REQUIRED" or "SHALL", means that the definition is an absolute requirement of the specification.
MUST NOT	This phrase, or the phrase "SHALL NOT", means that the definition is an absolute prohibition of the specification.
SHOULD	This word, or the adjective "RECOMMENDED", means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
SHOULD NOT	This phrase, or the phrase "NOT RECOMMENDED" means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

Chapter 3. Glossary

Hypervisor or Virtual Machine Monitor (VMM)	HS mode software that manages Virtual Machines by virtualizing hart, guest physical memory and IO resources. This document uses the term VMM and hypervisor interchangeably for this software entity.
VM	Virtual Machines hosted by a VMM
Host software	All software elements including type-1 or type 2 HS-mode VMM and OS; U mode user-space VMM tools; ordinary VMs hosted by the VMM that emulate devices. The hosting platform is typically a multi-tenant platform that hosts multiple mutually distrusting Tenants.
Tenant software	All software elements including VS-mode guest kernel software, and guest user-space software (in VU-mode) that are deployed by the workload owner (in a multi-tenant hosting environment).
Trusted Computing Base (TCB)Also, System/ Platform TCB	The hardware, software and firmware elements that are trusted by a relying party to protect the confidentiality and integrity of the relying parties' workload data and execution against a defined adversary model. In a system with separate processing elements within a package on a socket, the TCB boundary is the package. In a multi-socket system the TCB extends across the socket-to-socket interface, and is managed as one system TCB.
Application Processor (AP)	APs can support commodity operating systems, hypervisors/VMMs and applications software workloads. The AP subsystem may contain several processing units, on-chip caches, and other controllers for interfacing with memory, accelerators, and other fixed-function logic. Multiple APs may be used within a logical system.

AP-TEE	Application Processor- Trusted Execution Environment: An execution mode that provides HW-isolation for workload assets when in use (user/ supervisor code/ data) and provides HW-attestable confidentiality and integrity protection against specific attack vectors per a specified adversary and threat model. The term TEE and hardware-based TEE are also used as synonyms of AP-TEE in this document.
Confidential Computing	The protection of data in use by performing computation in a Hardware-based TEE.
TVM	TEE or Confidential VM - A VM instantiation of an AP-TEE workload
Confidential application or library	A user-mode application or library instantiation in a TVM. The user-mode application may be supported via a trusted runtime. The user-mode library may be hosted by a surrogate process runtime.
Attestation	The process by which a relying party can assess the security posture of the AP-TEE workload based on verifying a set of HW-rooted cryptographically-protected evidence.
TEE Security Manager (TSM)	HS-mode software module that acts as the trusted (in TCB) intermediary between the VMM and the TVM. This module extends the TCB chain on the AP-TEE platform.
RoT	Isolated HW/SW subsystem with an immutable ROM firmware and isolated compute and memory elements that forms the Trusted Compute Base of a TEE system. The RoT manages cryptographic keys and other security critical functions such as system lifecycle and debug authorization. The RoT provides trusted services to other software on the platform such as verified boot, key provisioning, and management, security lifecycle management, sealed storage, device management, crypto services, attestation etc. The RoT may be an integrated or discrete element [R7], and may take on the role of a Device Identification Composition Engine (DICE) as defined in [R2].

TEE-capable memory	Memory that provides access-control, confidentiality and integrity suitable per the threat model for use in the AP-TEE system. TEE-capable memory may also be used by untrusted software with appropriate TCB controls on the configuration.
SVN	Security Version Number - Meta-data about the TCB components that conveys the security posture of the TCB. The SVN is a monotonically increasing version number updated when security changes must be reflected in the attestation. The SVN is hence provided as part of the attestation information as part of the evidence of the TCB in use. The SVN is typically combined with other meta-data elements when evaluating the attestation information.
CDI	Compound Device Identifier - This value represents the hardware, software and firmware combination measured by the TCB elements transitively. A CDI is the output of a DICE [2] and is passed to the entity which is measured by the previous TCB layer. The CDI is a secret that may be certified to use for attestation protocols.
AIA	Advanced Interrupt Architecture
IMSIC	Incoming Message Signaled Interrupt Controller
MMIO	Memory Mapped I/O

Chapter 4. Architecture Overview and Threat Model

The AP-TEE extension supports a new class of hardware-attested trusted execution environment called TEE Virtual machines (TVM). The TVMs are supported by a hardware-rooted, attestable TCB and are run-time-isolated from the host OS/VMM and other platform software not in the TCB of the TVM. TVMs are protected from a broad set of software-based and hardware-based threats per the threat model described in [4_1_adversary_model]. The design enables the OS or VMM to maintain the role of resource manager even for the TVMs. The resources managed by the untrusted OS/VMM include memory, CPU, I/O resources and platform capabilities to execute the TVM workload.

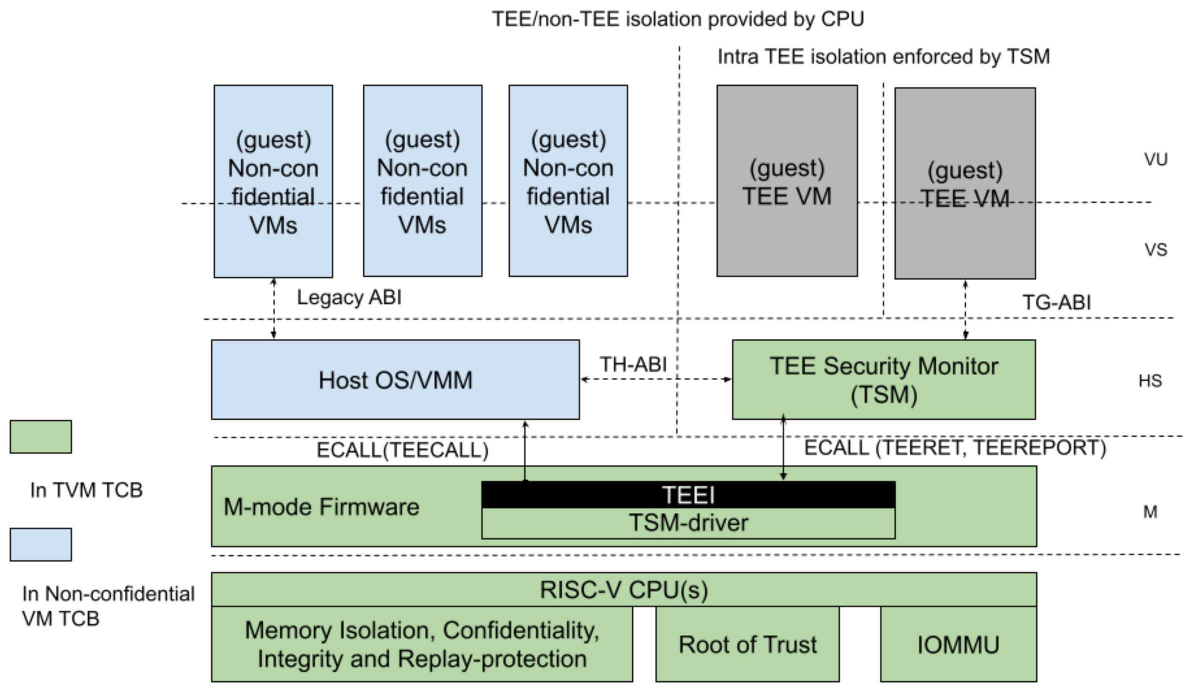
TEE/non-TEE isolation provided by CPU
Intra TEE isolation enforced by TSM
(guest) Non-con fidential VMs
(guest) Non-con fidential VMs
(guest) Non-con fidential VMs
(guest) TEE VM
(guest) TEE VM
VU
VS
Legacy ABI
TG-ABI
Host OS/VMM
TH-ABI
TEE Security Monitor (TSM)
HS
ECALL(TEECALL)
ECALL (TEERET, TEEREPORT)
In TVM TCB
M-mode Firmware
TEEI
TSM-driver
M
In Non-confidential VM TCB
RISC-V CPU(s)
Memory Isolation, Confidentiality, Integrity and Replay-protection
Root of Trust
IOMMU

Figure 1: TEE TCB for VM workloads

As shown in figure 1, the architecture comprises a HS-mode software module called the " **TEE Security Manager** " **(TSM)** that acts as the trusted intermediary between the VMM and the TVM. The TSM implements a set of TEE “flows” that are accessed via a **Trusted Execution Environment Interface (TEEI)** ABI hosted by a Trusted Security Manager Driver (**TSM Driver**) component operating in the M-mode of the CPU. The TSM itself operates in HS-mode (priv=01; V=0) of the CPU and enables the OS/VMM (also in HS-mode) to create TVMs, assign resources to TVMs, manage/execute and destroy a TVM - *this specification aims to describe the TEEI and TSM interfaces* . By using the Hypervisor extension of the RISC-V privileged specification [R0], this specification minimizes ISA changes to introduce a scalable architecture for hosting TEE workloads. More than one TVM may be hosted by the host OS/VMM. Each TVM may consist of the guest firmware, a guest OS and applications.

As shown in figure 1, the M-mode firmware is in the TCB of all AP-TEE workloads hosted on the platform. The TSM-driver (operating in M-mode) uses the hardware capabilities to provide:

- Isolation of memory associated with TEEs (including the TSM). We describe **TEE- capable memory** as memory that provides access-control, confidentiality and integrity suitable for use for AP-TEE components. The TEEI operations for memory management are described in detail below.

- Context switching of the hart state on TEE/Non-TEE transitions.
- A machine agnostic ABI as part of the TEEI, to allow lower privileged software to interact with the TSM-driver in an OS and platform agnostic manner.

The TSM-driver delegates parts of the TEE management functions to the TSM, specifically isolation across TEE-capable memory assigned to TVMs. The TSM is designed to be portable across AP-TEE class platforms and interact with the machine specific capabilities in the platform through the TEEI. The TSM provides an ABI to the OS/VMM which has two aspects: A **TH-ABI** that includes functions to manage the lifecycle of the TVM, such as creating, adding pages to a TVM, scheduling a TVM for execution, etc. in an OS/platform agnostic manner. The TSM also provides an ABI to the TVM contexts: A **TG-ABI** to enable the TVM workload to request attestation functions, memory management functions or paravirtualized IO functions.

In order to isolate the TVMs from the host OS/VMM and non-confidential VMs, the TSM state must be isolated first - this is achieved by enforcing isolation for memory assigned to the TSM - this is called the **TSM-memory-region.** The TSM-memory-region is expected to be a static region of memory that holds the TSM code and data. This region must be access-controlled from all software outside the TCB, and may be additionally protected against physical access via cryptographic mechanisms. Access to the TSM- memory-region and execution of code from the TSM-memory-region (the TSM flows) is predicated in hardware via an **AP-TEE mode bi t** maintained per hart. This mode is enabled per-hart via TEECALL and disabled via TEERET for operations described in the TEEI. Access to TEE-assigned memory is allowed for the hart when the AP-TEE mode is set. This per-hart AP-TEE mode bit is used by the processor to enforce access-control properties on instructions restricted for use by the TSM. This bit is cached in other micro-architectural states to enforce the isolation for TEE (TSM, TVM) resources (such as memory, IO, CSRs, TLB, paging structure caches etc). The implementation of this mode bit is not specified by this document, and may be implemented via an M-mode CSR for example.

The TSM functionality is explicitly limited to support the necessary security primitives to ensure that the OS/VMM and non-confidential VMs do not violate the security of the TVMs through the resource management actions of the OS/VMM. These security primitives require the TSM to enforce TVM virtual-hart state save and restore, as well as enforcing invariants for memory assigned to the TVM (including stage 2 translation). The host OS/VMM provides the typical VM resource management functionality for memory, IO etc.

Confidential VMs (under a VMM) are shown in figure 1 and Confidential applications (managed by an untrusted host OS) are shown in the architecture figure 2. As evident from the architecture, the difference between these two scenarios is the software TCB (owned by the tenant within the TVM) for the tenant workload - in the application TEE case, a minimal guest OS runtime may be used; whereas in the VM TEE case, an enlightened guest OS is in the TVM TCB. Other software models that map to the VU/VS modes of operation are also possible as TEE workloads. Importantly, the HW mechanisms needed for both cases are identical, and can be supported with appropriate extensions of the TG-ABI.

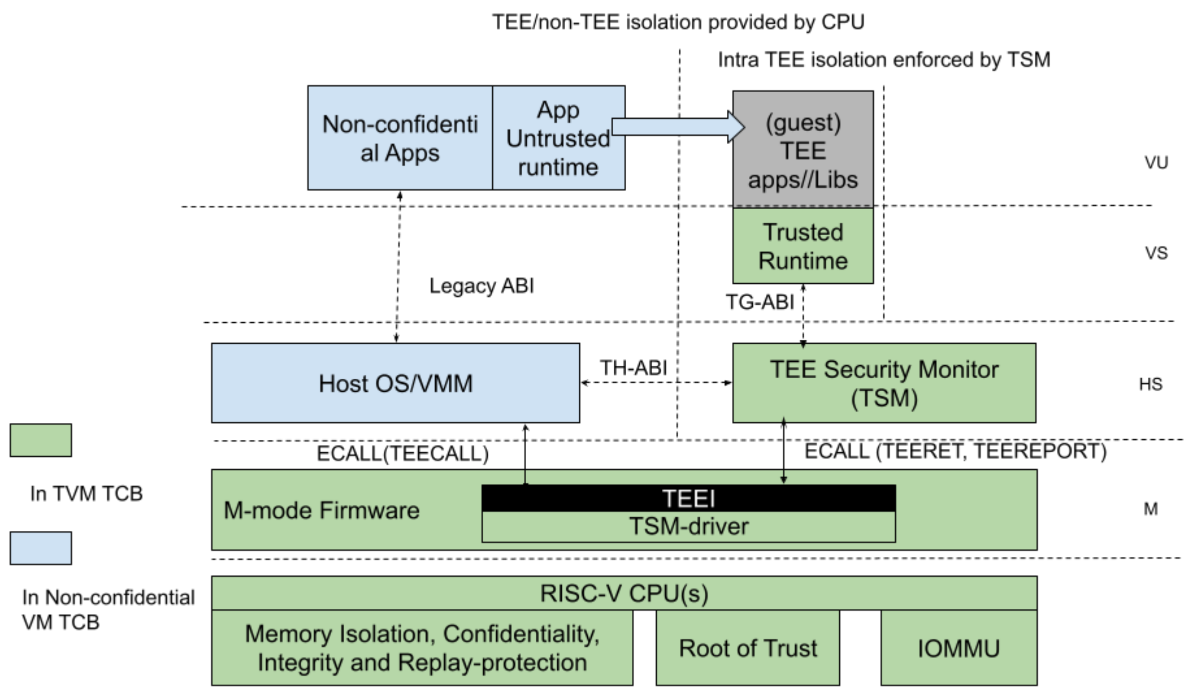
TEE/non-TEE isolation provided by CPU
Intra TEE isolation enforced by TSM
Non-confidential Apps
App Untrusted runtime
(guest) TEE apps//Libs
Trusted Runtime
VU
VS
Legacy ABI
TG-ABI
Host OS/VMM
TH-ABI
TEE Security Monitor (TSM)
HS
ECALL(TEECALL)
ECALL (TEERET, TEEREPORT)
In TVM TCB
M-mode Firmware
TEEI
TSM-driver
M
In Non-confidential VM TCB
RISC-V CPU(s)
Memory Isolation, Confidentiality, Integrity and Replay-protection
Root of Trust
IOMMU

Figure 2: TEE TCB for application workloads (hosted via a TVM)

The detailed architecture is described in the Section [5_reference_architecture_details]. Note that the architecture described above may have various implementations, however the goal of this specification is to propose a reference architecture and ratify the TEEI as a RISC-V non-ISA specification.

4.1. Adversary Model

Unprivileged Software adversary - This includes software executing in U-mode managed by S/HS/M-mode system software. This adversary can access U-mode CSRs, process/task memory, CPU registers in the process context managed by system software.

System Software adversary - This includes system software executing in S/HS/VS modes. Such an adversary can access S/HS/VS privileged CSRs, assigned system memory, CPU registers and IO devices.

Startup Software adversary - This includes system software executing in early/boot phases of the system (in M-mode), including BIOS, memory configuration code, device option ROM/firmware that can access system memory, CPU registers, IO devices and IOMMU etc.

Simple Hardware adversary __ - This includes adversaries that can use hardware attacks such as bus interposers to snoop on memory/device interfaces, voltage/clock glitching, observe electromagnetic and other radiation, analyze power usage through instrumentation/tapping of power rails, etc. which may give the adversary the ability to tamper with data in memory.

Advanced Hardware adversary - This includes adversaries that can use advanced hardware attacks, with unlimited physical access to the devices, and use mechanisms to tamper-with/reverse-engineer the hardware TCB e.g., extract keys from hardware, using capabilities such as scanning electron microscopes, fib attacks etc.

Side/Covert Channel Adversary - This includes adversaries that may leverage any explicit/implicit shared state (architectural or micro-architectural) to leak information across privilege boundaries via inference of characteristics from the shared resources (e.g. caches, branch prediction state, internal micro-architectural buffers, queues). Some attacks may require use of high-precision timers to leak information. A combination of system software and hardware adversarial approaches may be utilized by this adversary.

4.2. Threat Model

T1: Loss of confidentiality of TVMs and TSM memory via in-scope adversaries that may read TSM/TVM memory via CPU accesses

T2: Tamper/content-injection to TVM and TSM memory from in-scope adversaries that may modify TSM/TVM memory via CPU side accesses

T3: Tamper of TVM/TSM memory from in-scope adversaries via software-induced row-hammer attacks on memory

T4: Malicious injection of content into TSM/TVM execution context using physical memory aliasing attacks via system firmware adversary

T5: Information leakage of workload data via CPU registers, CSRs via in-scope adversaries

T6: Incorrect execution of workload via runtime modification of CPU registers, CSRs, mode switches via in-scope adversaries

T7: Invalid code execution or data injection/replacement via second stagel paging remap attacks via system software adversary

T8: Malicious asynchronous interrupt injection or dropped leading to information leakage or incorrect execution of the TEE

T9: Malicious manipulation of time read from the virtualized time CSRs causing invalid execution of TVM workload

T10: Loss of Confidentiality via DMA access from devices under adversary control e.g. via manipulation of IOMMU programming

T11: Loss of Confidentiality from devices assigned to a TVM. Devices bound to a TVM must enforce similar properties as the TEE hosted on the platform.

T12: Content injection, exfiltration or replay (within and across TEE memory) via hardware approaches, including via exposed interface/links to other CPU sockets, memory and/or devices assigned to a TVM

T13: Downgrading TEE TCB elements (example TSM-driver, TSM) to older versions or loading Invalid TEE TCB elements on the platform to enable confidentiality, integrity attacks

T14: Leveraging transient execution side-channel attacks in TSM-driver, TSM, TVM, host OS/VMM or non-confidential workloads to leak confidential data e.g. via shared caches, branch predictor poisoning, page-faults.

T15: Leveraging architectural side-channel attacks due to shared cache and other shared resources e.g. via prime/probe, flush/reload approaches

T16: Malicious access to ciphertext with known plaintext to launch a dictionary attack on TVMs or TSM or trusted firmware to extract confidential data.

T17: Tamper of TVM state during migration of a TEE workload assets within the platform or from one platform to another.

T18: Forging of attestation evidence and sealed data associated with a TVM.

T19: Stale TLB translations (for U/HS mode or for VU/VS) created during TSM or TVM operations are used to execute malicious code in the TVM (or consume stale/invalid data)

T20: Isolation of performance monitoring and/or debug state for a TVM leading to information loss via performance monitoring events/counters and debug mode accessible information.

T21: A TVM causes a denial of service on the platform

This is not an exhaustive list and will be updated on a regular basis as attacks evolve._

4.3. Scope

This specification does not prescribe the scope of mitigation and focusses on the TEEI interface and use-of/impact-on the RISC-V ISA. It is recommended that implementations of this reference architecture address threats from system software adversaries. Implementations may choose to mitigate threats from additional adversaries. For all cases, denial of service by TVMs must be prevented. At the same time, denial of service by host software is considered out of scope. :imagesdir: ./images

Chapter 5. Reference Architecture Details

We describe the properties of the TSM, its instantiation, isolation and operational model for the TVM lifecycle. The description refers to Figure 1.

5.1. TSM initialization

The AP-TEE architecture requires a hardware Root-of-trust for supporting TCB measurement, reporting and storage [R8]. The Root-of-trust for Measurement (RTM) is defined as the TCB component that performs a measurement of an entity and protects it for subsequent reporting. The Root-of-trust for Reporting (RTR) is typically a HW RoT that reliably provides authenticity and non-repudiation services for the purposes of attesting to the origin, integrity and security version of platform TCB components. Each TCB layer should have associated security version numbers (SVN) to allow for TCB recovery in the event of security vulnerabilities discovered in a prior version of the TCB layer.

During platform initialization, HW elements form the RTM that measure the TSM-driver. The TSM-driver acts as the RTM for the TSM loaded on the platform. The TSM-driver initializes the TSM-memory-region for the TSM - this TSM-memory-region must be in TEE-capable memory. The TSM binary may be provided by the OS/VMM which may independently authenticate the binary before loading the binary into the TSM-memory-region via the TSM-driver. Alternatively, the firmware may pre-load the TSM binary via the TSM-driver. In both cases, the TSM binary loaded must be measured and may be authenticated (per cryptographic signature mechanisms) by the TSM-driver during the loading process, so that the TSM used is reflected in the attestation rooted in a HW RoT. The authentication process provides additional control to restrict TSM binaries that can be loaded on the platform based on policies such as version, vendor etc. In addition to the measurements, a security version number (SVN) of the TSM should be recorded by the TSM-driver into the firmware measurement registers accessible only to the TSM-driver and higher privilege components. The measurements and versions of the HW RoT, the TSM-driver and the TSM will subsequently be provided as evidence of a specific TSM being loaded on a specific platform.

During initialization, the TSM-driver will initialize a TSM-data region within the TSM-memory region. The TSM-data region may hold per-hart TSM state, memory assignment tracking structures and additional global data for TSM management. The TSM-data region is TEE-capable memory that is apriori access-control-restricted by the TSM-driver to allow only the TSM to access this memory. The per-hart TSM state is used to start TSM execution from a known-good state for security routines invoked by the OS/VMM. The per-hart TSM state should be stored in pages that form a TSM Hart Control Structure (THCS - See Appendix A) which is initialized as part of the TSM memory initialization. The THCS structure definition is part of the TEEI and may be extended by an implementation, with the minimum state shown in the structure. Isolating and establishing the execution state of the TSM is the responsibility of the TSM-driver. Saving and restoring of the execution state of the TSM (for interrupted routines) is performed by the TSM. The operating modes of the TSM are described in Section 5.2. Saving and restoring of the TVM execution state in the TVM virtual-harts (called the VHCS) is the responsibility of the TSM and is held in TEE-capable memory assigned to the TVM by the VMM.

5.2. TSM operation and properties

The TSM implements security routines that are invoked by the OS/VMM or by the TVMs, e.g. by the VMM to grant a TVM a TEE-capable memory page and setup second-stage mapping, activate a TVM virtual hart on a physical hart etc. The TSM security routines are invoked by the OS/VMM via an ECALL with the service call specified via registers. These service calls trap to the TSM-driver. The TSM-driver switches hart state to the TSM context by loading the hart's TSM execution state from the THCS.tssa and then returns via an MRET to the TSM. The TSM executes the security routine requested (where the TSM enforces the security properties) and may either return to the OS/VMM via an ECALL to the TSM-driver (TEERET with reason), or may use an SRET to return/enter into a TVM. On a subsequent TVM synchronous or asynchronous trap (due to ECALLs or any exception/interrupt) from a TVM, the TSM handles the cases delegated to it by the TSM-driver (via mideleg). The TSM saves the TVM state and invokes the TSM-driver via an ECALL (TEERET with reason) to initiate the return of execution control to the OS/VMM if required. The TSM-driver restores the context for the OS/VMM via the per-hart control sub-structure THCS.hssa (See Appendix A). This canonical flow is shown in figure 3.

Beyond the basic operation described above, the following different operational models of the TSM may be supported by an implementation:

- **Uninterruptible TSM** - In this model, the TSM security routines are executed in an uninterruptible manner for S-mode interrupts (M-mode interrupts are not inhibited). This implies that the TSM execution always starts from a fixed initial state of the TSM harts and completes the execution with either a TEERET to return control to the OS/VMM or via an SRET to enter into a TVM (where the execution may be interruptible again).
- **Interruptible TSM with no re-entrancy** - In this model, after the initial entry to the TSM with S-mode interrupts disabled, the TSM enables interrupts during execution of the TSM security routines. The TSM may install its interrupt handlers at this entry (or may be installed via the TEECALL flow as shown below). On an S-mode interrupt, the TSM hart context is saved by the TSM and keeps the interrupt pending. The TSM may then TEERET to the host OS/VMM with explicit information about the interruption provided via the pending interrupt to the OS/VMM. The TSM-driver supports a TEERESUME ECALL which enables the TSM to enforce that the resumption of the interrupted TSM security routine is initiated by the OS/VMM on the same hart. The TSM hart context restore is enforced by the TSM to allow for the resumed TSM security routine operation to complete. An example of an interruptible flow is the conversion of a large 2MB page to confidential memory, which may require a long latency encryption operation. Intermediate state of the operation must be saved and restored by the TSM for such flows.

This specification describes the operation of the TSM in this mode of operation.

- **Interruptible and re-entrant TSM** - In this model, similar to the previous case, the TSM security routines are executed in an interruptible manner, but are also allowed to be re-entrant. This requires support for trusted thread contexts managed by the TSM. A TSM security routine invoked by the OS/VMM is executed in the context of a specific TSM thread context (a stack structure may also be used). On an interruption of that routine using a TSM thread context, the TSM saves the TSM execution context for the TSM thread and returns control to the OS/VMM via a TEERET. The OS/VMM can handle the interrupt and may resume that TSM thread or may invoke another TSM security routine on a different (non-busy) thread context (and on a different hart). This model of TSM operation requires additional concurrency controls on internal data structures and per-TVM global data structures (such as the second stage page table structures).

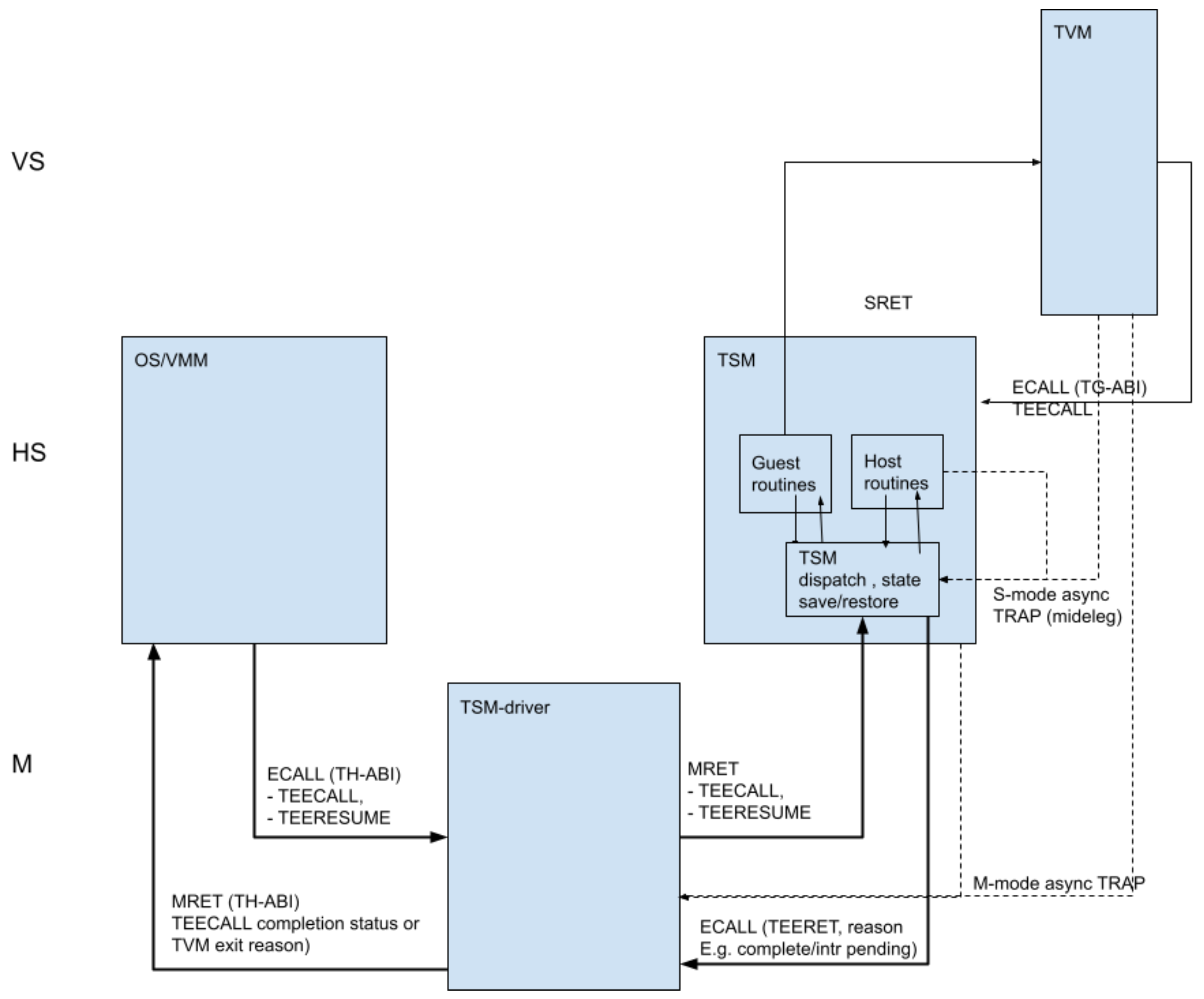

Figure 3: TSM operation - Interruptible and non-reentrant TSM model shown.

A TSM entry triggered by an ECALL (with AP-TEE service type) by the OS/VMM leads to the following context-switch to the TSM (performed by the TSM-driver):

The initial state of the TSM will be to start with a fixed reset value for the registers that are restored on resumed security operations.

ECALL (TEECALL / TEERESUME) pseudocode - implemented by the TSM-driver

- If trap is due to synchronous trap due to TEECALL/ TEERESUME then enable AP-TEE mode = 1 for the hart via M-mode CSR (implementation-specific)
- Locate the per-hart THCS (located within TSM-driver memory data region)
- Save operating VMM csr context into the THCS.hssa (Hart Supervisor State Area) fields : sstatus, stvec, scounteren, sscratch, satp (and other x state other than a0, a1 - see [9_appendix_a_thcs_and_vhcs]). Note that any v/f register state must be saved by the caller.
- Save THCS.hssa.pc as mepc+4 to ensure that a subsequent resumption happens from the pc past the TEECALL

- Establish the TSM operating context from the THCS.tssa (TSM Supervisor State Area) fields (See Appendix A)
- Set scause to indicate TEECALL
- Disable interrupts via sie=0.
    - For a preemptable TSM, interrupts do not stay disabled - the TSM may enable interrupts and so S/M-mode interrupts may occur while executing in the TSM. S-mode interrupts will cause the TSM to save state and TEERET.
- MRET to resume execution in TSM at THCS.tssa.stvec

ECALL (synchronous explicit TEERET) OR Asynchronous M-mode trap pseudocode - implemented by TSM-driver

- Locate the per-hart THCS (located within TSM-driver memory data region)
- If Asynchronous M-mode trap:
    - Handle M-mode trap
    - If required, pend an S-mode interrupt to the TSM and SRET
- *Implementation Note - The TSM-driver does not need to keep state of the TSM being interrupted as, on an interrupt the TSM can enforce:*
    - *If it was preemptible but not-reentrant that the next invocation on that hart is a TEERESUME with identical parameters as the interrupted security routine.*
    - *If the TSM was preemptible and re-entrant then the TSM would accept both TEERESUME and TEECALL as subsequent invocations (as long as TSM threads are available).*
- Restore the OS/VMM state saved on transition to the TSM: sstatus, stvec, scounteren, sscratch, satp and x registers (other than a0, a1). Note that any v/f register state must be restored by the caller.
- TSM-driver passes TSM/TVM-specified register contents to the OS/VMM to return status from TEERET (TSM sets a0, a1 registers always - other registers may be selected by the TVM)
- Clear AP-TEE-mode on hart (via implementation-specific M-mode CSR to block non-TEE mode accesses to TEE-assigned memory.)
- MRET to resumes execution in OS/VMM at mepc set to THCS.hssa.pc (THCS.hssa.pc adjusted to refer to opcode after the ECALL that triggered the TEECALL / TEERESUME)

The TSM is stateless across TEECALL invocations, however a security routine invoked in the TSM via a TEECALL may be interrupted and must be resumed via a TEERESUME i.e. *the TSM is preemptable but non-reentrant* . These properties are enforced by the TSM-driver, and other models described above may be implemented. The TSM does not perform any dynamic resource management, scheduling, or interrupt handling of its own. Hence the TSM is not expected to have a S-model interrupt file of its own, and so for issuing IPIs the TSM must invoke the TSM-driver and use the M-mode Interrupt file when the TSM issues IPIs.

When the TSM is entered via the TSM-driver (as part of the ECALL [TEECALL] - MRET), the TSM starts with sstatus.sie set to 0 i.e. interrupts disabled. The sstatus.sie does not affect HS interrupts from being seen when mode = U/VS/VU. The OS/VMM sip and sie will be saved by the TSM in the

HSSA and will retain the state as it existed when the host OS/VMM invoked the TSM. The TSM may establish the execution context and re-enable interrupts (sstatus.sie set to 1).

If an M-mode interrupt occurs while the hart is operating in the TSM or any TVM, the control always goes to the TSM-driver handler, which can handle it, or if the event must be reported to the untrusted OS/VMM, they are pended as S-mode interrupts to the TSM which must save its execution context and return control to the OS/VMM via a TEERET..

If an S-mode interrupt occurs while the hart is operating in the TSM (HS-mode), it should pre-empt out and return to the OS/VMM using TEERET. The TSM may take certain actions on S-mode interrupts - for example, saving status of a host security routine, and/or change the status of TVMs. The TSM is however not expected to retire the S-mode interrupt but keep the event pending so they are taken when control returns to the OS/VMM via the TEERET.

If a S-mode interrupt occurs in U, VU or VS - external, timer, or software - then that causes the trap handler in TSM to be invoked. In response to trap delivery, the TSM saves the TVM virtual-hart state and returns to the OS/VMM via a TEERET ECALL. As part of return to the OS/VMM, the sstatus of OS/VMM is restored and when the OS starts executing the pending interrupt - external, timer, or software - may or may not be taken depending on the OS sstatus.sie. Under these circumstances the saving of the TVM state is the TSM responsibility.

When TVM is executing, hideleg will only delegate VS-mode external interrupt, VS-mode SW interrupt, and VS-mode timer interrupts to the TVM. S-mode SW/Timer/External interrupts are delegated to the TSM (with the behavior described above). *All other interrupts* , M-mode SW/Timer/External, bus error, high temp, RAS etc. are not delegated and delivered to M-mode/TSM-driver. Under these circumstances the saving of the state is the TSM-driver responsibility. Also since scrubbing the TVM state is the TSM responsibility, the TSM-driver may pend an S-mode interrupt to the TSM to allow cleanup on such events. See Appendix B for a table of interrupt causes and handling requirements.

Any NMIs experienced during TSM/TVM execution are always handled by the TSM-driver and must cause the TEEs to be destroyed (preventing any loss of confidential info via clearing of machine state). The TSM and therefore all TVMs are prevented from execution after that point.

5.3. TSM and TVM Isolation

TSM (and all TVMs) memory is granted by the host OS/VMM but is isolated (via access-control and/or confidentiality-protection) by the HW and TCB elements. The TSM, TVM and HW isolation methods used must be evident in the attestation evidence provided for the TVM since it identifies the hardware and the TSM-driver.

There are two facets of TVM and TSM memory isolation that are implementation-specific:

a) Isolation from host software access - The CPU may enforce a hardware-based access-control of TSM memory to prevent access from host software (VMM and host OS) V=0, HS-mode untrusted code. TEE and TVM address spaces are identified by an additional (implementation-defined) **AP-TEE mode qualifier** to maintain the isolation during access and in internal caches, e.g. Hart TLB lookup may be extended with the AP-TEE mode qualifier. TVM memory isolation must support sparse memory management models and architectural page-sizes of 4KB, 64K, 2MB, 1GB (and optionally 512GB). For example, The hardware may provide a memory ownership tracking table where there is an entry per physical page. The memory ownership tracking table may be a radix tree or a flat table. The memory ownership tracking table may allow memory ownership at multiple granularities such as 4K, 64K, 2M, 1G, etc. The memory ownership table may be enforced at the memory controller, or in a page table walker.

b) Isolation against physical/out-of-band access - The platform TCB may provide confidentiality, integrity and replay-protection. This may be achieved via a Memory Encryption Engine (MEE) to prevent TEE state being exposed in volatile memory during execution. The use of an MEE and the number of encryption domains supported is implementation-specific. For example, The hardware may use the **AP-TEE mode qualifier** during execution (and memory access) to cryptographically isolate memory associated with a TEE which may be encrypted and additionally cryptographically integrity-protected using a MAC on the memory contents. The MAC may be maintained at various granularity - e.g. cache block size or in multiples of cache blocks.

TVM isolation is the responsibility of the TSM via the second stage address translation table (hgatp). The TSM must track memory assignment of TVMs (by the untrusted VMM/OS) to ensure memory assignment is non-overlapping, along with additional security requirements. The following are the security requirements/invariants for enforcement of the memory access-control for memory assigned to the TVMs. These rules are enforced by the TSM and the HW:

1. Contents of a TVM page assigned (statically measured or lazy-initialized) to the TVM is bound to the Guest PA assigned to the TVM during TVM operation.
2. A TVM page can only be assigned to a single TVM, and mapped via a single GPA unless aliases are allowed in which case, such aliases must be tracked by the TSM). Aliases in the virtual address space are under the purview of the TVM OS.
3. 1st stage address translation - A TVM page mapping must be translated only via first stage translation structures which are contained in pages assigned to the same TVM.
4. 2nd stage address translation:
    a. A TVM page guest physical address mapping must be translated only via the TSM-managed second stage translation structures for that TVM.
    b. 2nd stage structures may not be shared between TVMs, and must not refer to any other TVMs pages.
    c. The OS/VMM has no access to TVM second stage paging structures
    d. The OS/VMM may install shared page mappings (via TSM oversight) to non-confidential pages that are not assigned to any TVM or the TSM - this is for example for untrusted IO.
    e. Circular mappings in the second stage paging structures are disallowed.
5. Access to shared memory pages must be explicitly signaled by the TVM via the GPA and enforced for memory ownership for the TVM by the HW.

5.4. TVM Execution

TVMs can access two classes of memory - “confidential memory” - which has confidentiality and access-control properties for memory exclusive to the TVM, and "non-confidential memory" which is memory accessible to the host OS/VMM and is used for untrusted operations (e.g. virt-io, grpc communication with/via the host). If the confidential memory is access-controlled only, the TSM and TSM-driver are the authority over the access-control enforcement. If the confidential memory is using memory encryption, the encryption keys used for confidential memory must be different from non-confidential memory.

All TVM memory is mapped in the second-stage page tables controlled by the TSM explicitly - the allocation of memory for the second stage paging structures pages used for the second stage mapping is also performed by the OS/VMM but the security properties of the second stage mapping are enforced by the TSM. By default any memory mapped to a TVM is confidential. A TVM may then explicitly request that confidential memory be converted to non-confidential memory regions using services provided by the TSM. More information about TVM Execution and the lifecycle of a TVM is described in the [7_tvm_lifecycle] section of this document.

5.5. Debug and Performance Monitoring

The following additional considerations are noted for debug and performance monitoring:

Debug mode considerations

In order to support probe-mode debugging of the TSM, the RoT must support an authorized debug of the platform. The authentication mechanism used for debug authorization is implementation-specific, but must support the security properties described in the Section 3.12 of the RISC-V Debug Support specification version 1.0.0-STABLE [R6]. The RoT may support multiple levels of debug authorization depending on access granted. For probe-based debugging of the hardware, the RoT performing debug authentication must ensure that separate attestation keys are used for TCB reporting when probe-debug is authorized vs when the platform is not under probe-debug mode.The probe-mode debug authorization process must invalidate sealed keys to disallow sealed data access when in probe-debug modes.

When a TVM is under self-hosted debugging - on a transition to TVM execution, the TSM-driver must set up the trigger CSRs for the TVM. For TVM debugging, the TSM-driver may inhibit M and S/HS modes in the triggers. On transitions back to the OS/VMM, the TSM-driver will save the trigger CSRs and associated debug states, thus not leaking any information to non-TEE workloads. TVM self-hosted debug may be enabled from TVM creation time or may be explicitly opted-into during execution of the TVM. The TSM may invoke the TSM-driver to set up a TVM-specific trigger CSR state (per the configuration of the TVM).

Performance Monitoring considerations

By default the TSM and all TVMs run with performance monitoring suppressed. If a TVM runs in this default mode (opted out of performance monitoring), on a transition to the TVM, the TSM-driver enforces this via inhibiting the counters (using mcountinhibit).

If the TVM has opted-in to performance monitoring, the TSM must invoke the TSM-driver to establish a TVM-specific performance monitoring controls (triggers, event selectors). For any counters that the TVM will use, the TSM will assign those to the TVM via the TSM-driver and inhibit counting in HS/M mode - with Sscofpmf and future RISC-V extensions these controls could be delegated to the TVM (VS mode) by the TSM. The TSM is free to use any counters that are not delegated. If the TSM is not using any counters and any of the TVMs opt-in to use hpm then the TSM may delegate the LCOFI interrupt (via hideleg[13]=1) for that TVM. The delegated TVM counters naturally inhibit counting in S/HS and M. The TSM-driver must save and clear counter/event selector values as control transitions to the VMM or a different TVM that is using hpm. On a transition back to the host OS/VMM, the TSM-driver must restore the saved hardware performance monitoring event triggers and counter enables.

The TVM may opt-in to use performance monitoring either at initialization or post-init. For TVMs that have performance monitoring enabled, the TSM-driver may implement a service for the TSM to allow dynamically saving and restoring performance monitoring controls when a TVM is executing - this can reduce the performance overhead for the TSM-driver to only perform the save/restore of the controls when required by the TVM.

Chapter 6. TVM Attestation

6.1. TCB Elements

Elements considered to be in the TCB for AP-TEE workloads are summarized below:

Hardware/firmware

- CPU: All hardware logic, including MMU, caches
- SOC: All hardware subsystems including memory confidentiality, integrity and replay-protection for volatile memory
- RoT for TCB measurement, evidence reporting, attestation, sealing
- IOMMU
- (optional) Devices may be included in the TCB if the devices support reporting evidence of their security posture.

Software/firmware

- TSM-driver that hosts a TEEI (with TH-ABI and TG-ABI security routines). Note that since the TSM-driver operates in M-mode, all M-mode firmware is included in the TCB for AP-TEE workloads.
- TEE Security Manager (TSM) and user-mode TSM components
- For confidential application/VM workloads, an AP-TEE-compatible Runtime/guest OS may be included for portability (but is not required).

6.2. Attestation

The TCB described above is reported to relying parties via an attestation mechanism and protocol.

Framework

The IETF RATS [x] describes the following reference model for attestation. In Remote Attestation, the Attester produces information about itself (Evidence) to enable a remote peer (the Relying Party) to decide whether to consider that Attester a trustworthy peer or not. The Verifier appraises evidence via appraisal policies and creates the Attestation Results to support Relying Parties in their decision process.

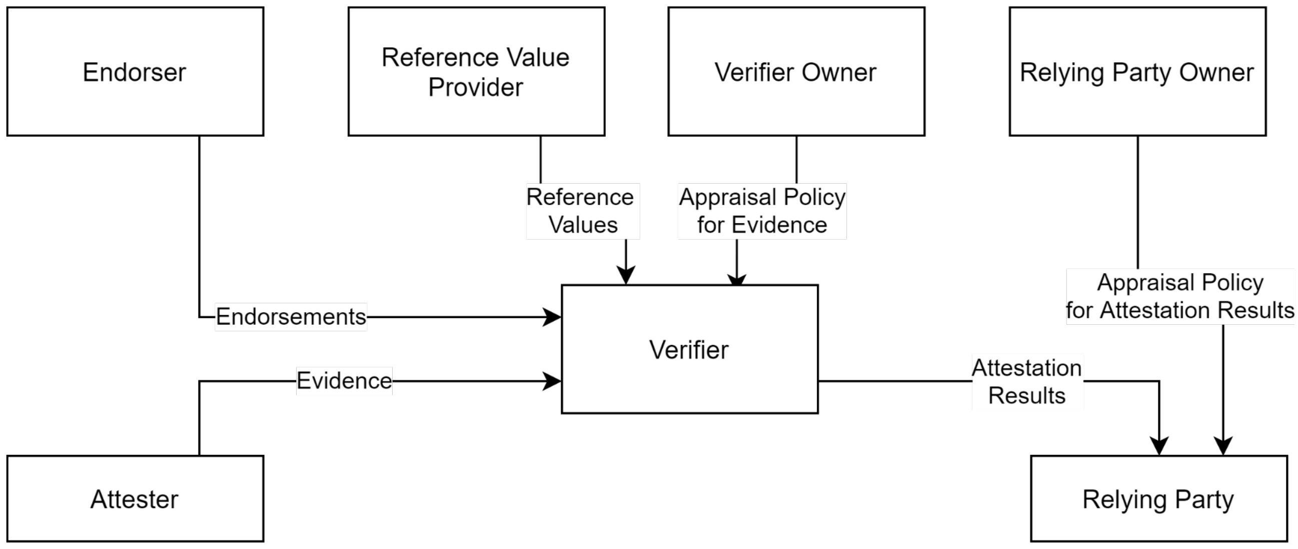

Figure 4: Remote Attestation Framework (IETF RATS)

This TEE proposal uses the layered attestation model [R1] where the RoT is the initial Attesting Environment. Claims are collected from or about each layer. The corresponding Claims can be structured in a nested fashion that reflects the nesting of the Attester's layers. The previous layer acts as the Attesting Environment for the next layer. Claims about a RoT typically are asserted by an Endorser.

The following are the key requirements for attestation mapped to this AP-TEE architecture:

In order for the TCB (described above) to be enforced by the architecture, the TSM driver measures the untrusted-host-supplied TSM binary and records its measurements, vendor and version into measurement registers which can be attested to via the HW RoT-rooted keys.

The TSM must then provide an implementation of a TEE-Guest ABI (TG-ABI) operation (teecall_tg_get_evidence) to enable a TVM to generate attestation evidence that a relying party can verify using the certificate chain.

The TCB extension and evidence collection for a TVM attestation is shown below:

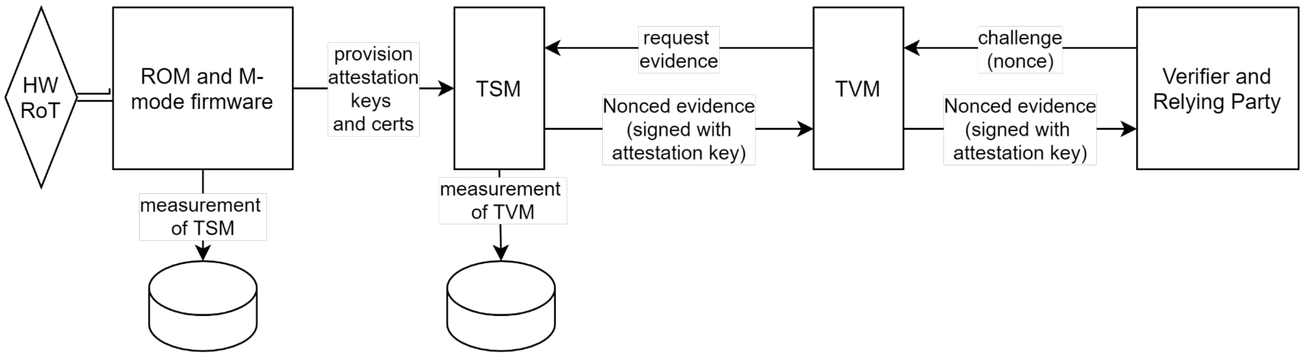
HW RoT
ROM and M-mode firmware
provision attestation keys and certs
TSM
request evidence
TVM
challenge (nonce)
Verifier and Relying Party
Nonced evidence (signed with attestation key)
Nonced evidence (signed with attestation key)
measurement of TSM
measurement of TVM

Figure 5: Layered Attestation architecture for TVMs

It is expected that an implementation will provide implementation-specific intrinsics to record measurements of the TSM into the firmware RoT for measurement to support the layered RTMs and attestation of AP-TEE workloads.

Attestation Evidence

Suitable evidence formats may be used by the Attester to present the evidence that the TVM is executing as a TEE. The evidence should attest to the above layered trust chain. The TSM must allow for attestation operation (certifying TVM measurements) to be executed in an interruptible manner. Once such evidence format is specified in the TCG DICE Attestation Architecture which describes evidence as X.509 Certificate with an extension for *TCB Info Evidence [R2].

The following key fields are present in that DiceTcbInfo (See OID in spec [2]). The fields are listed here with the usage described specific to the AP-TEE reference architecture.

Field	**Type**	**Description**
Vendor	UTF8String	The entity that created the TCB component.
Model	UTF8String	The product name associated with the TCB component.
Version	UTF8String	The revision string associated with the TCB component.
SVN	Integer	The security version number associated with the TCB component - the SVN makes parsing of the TCB simpler to differentiate updates that affect security from non-security related updates.
Layer	Integer	The DICE layer associated with this measurement of the TCB component.
Index	Integer	A value that enumerates measurement of assets within the TCB component and DICE layer.

FWIDs	List of FWID	A list of FWID values resulting from applying the hashAlg function over the object being measured (recommended components should cover: code, config, static data of a specific TCB binary component). FWIDs are computed by the DICE layer that is the Attesting Environment and certificate Issuer. Each FWID consists of:* HashAlg (OID) – an algorithm identifier for the hash algorithm used to produce a digest value.* Digest – a digest of the firmware, initialization values, or other settings of the TCB component.
Flags	Uint8	Enumerates potentially simultaneous operational states of the TCB component: (i) notConfigured, (ii) notSecure, (iii) recovery, (iv) debug. A value of 1 (TRUE) means the operational mode is active. A value of 0 (FALSE) means the operational state is not active. If the flags field is omitted, all flags are assumed to be 0 (FALSE).
VendorInfo	Octet String	Vendor supplied values that encode vendor, model, or device specific state
Type	Octet String	A machine readable description of the measurement

This extension defines attestation evidence about the DICE layer that is associated with the Subject key. The certificate Subject and SubjectPublicKey identify the entity to which the DiceTcbInfo extension applies. When this extension is used, the measurements in the evidence usually describe the software/firmware (and configuration) which will execute within the TCB. The AuthorityKeyIdentifier extension [2] MUST be supplied when the DiceTcbInfo extension is supplied. This allows the Verifier to locate the signer’s certificate. The DiceTcbInfo extension should be included with CRL entries that revoke the certificate that originally included the said DiceTcbInfo extension.

For TVM attestation, the following TCB Evidence Info will be sequenced using the above DiceTcbInfo structure. Multiple evidences may be provided via the **MultiDiceTcbInfo** extension:

- Cryptographic hash of the RoT FW binary and configuration, along with its SVN and other fields;
- Cryptographic hash of the TSM-driver binary and configuration, along with its SVN and other fields ;
- Cryptographic hash of the TSM binary and configuration, with its SVN and other fields;
- Cryptographic hash of the OSAM (described below) binary and configuration, with its SVN and other fields - this is applicable for remote attestation only;
    - If OSAM is a 3rd party - the certifying entity will need a separate evidence entry.
- Cryptographic hash of the TVM static binaries and configuration, along with its SVN and other fields.
- The TVM may additionally extend cryptographic measurements for other workload binaries and configuration loaded dynamically subsequent to boot via the TG-ABI.

The TVM TCB Evidence Info is managed by the TSM and is combined with the TSM's TCB Evidence info that is in turn managed by the TSM-driver. The TSM-driver provides a TEEI security routine to enable the TSM and transitively the TVM to generate an Attestation CDI (Composite Device Identifier) and key to participate in an Attestation certificate-based protocol for remote (and local) attestation.

We recommend at least the following CDIs to be supported for AP-TEE workloads:

1. Attestation CDI - This CDI is derived from the combination of the input values listed above and is expected to change across software updates or configuration changes of these components. This CDI is meant for remote attestation and is mandatory for AP-TEE implementations.
2. Versioned Sealing CDI - This CDI is also derived from the combination of the input values listed above seeded with a component security version number. This Versioned Sealing CDI allows for the sealing key to be bound to a version chain of the TCB components. This CDI is appropriate for sealing and is recommended for AP-TEE implementations.

For remote attestation of a TVM, an X.509 Attestation certificate (structure shown below) is provisioned or generated on-demand for the TVM via the TSM. This process requires the generation of a CDI certificate where the subject key pair is derived from the Attestation CDI value for any layer (e.g. TSM-driver). The authority key pair which signs the certificate (e.g. RoT) is derived from the UDS (for the RoT) or, after the initial hardware to software transition, from the Attestation CDI value for the current layer (e.g. TSM-driver). The DICE flow outputs the CDI values and the generated certificate; the private key associated with the certificate may be optionally passed along with the CDI values to avoid the need for re-derivation by the target layer. The UDS-derived public key is certified by an external authority during manufacturing to root the certificate chain in a HW RoT.

As a tangible example, the CDI private key for the TSM were used to sign a leaf certificate for an attestation key for the TVM, the certificate chain may look like this:

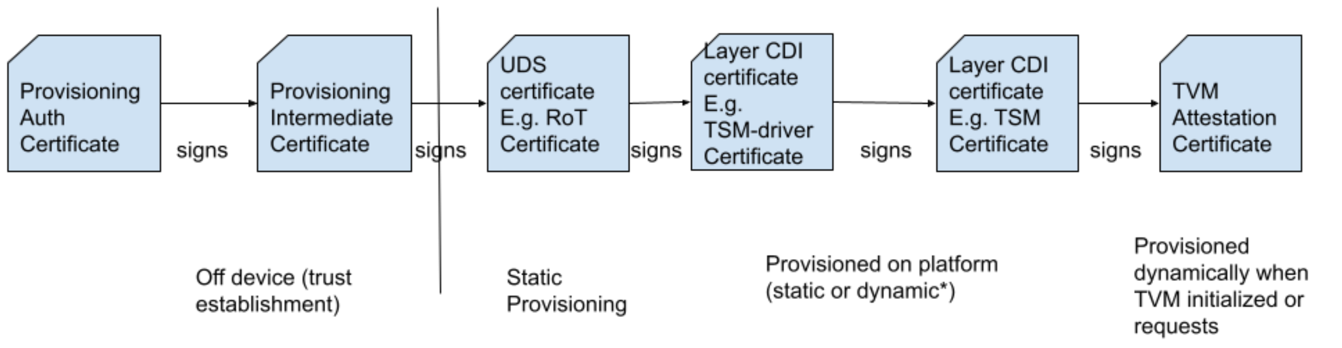
Provisioning Auth Certificate
signs
Provisioning Intermediate Certificate
signs
UDS certificate E.g. RoT Certificate
signs
Layer CDI certificate E.g. TSM-driver Certificate
signs
Layer CDI certificate E.g. TSM Certificate
signs
TVM Attestation Certificate
Off device (trust establishment)
Static Provisioning
Provisioned on platform (static or dynamic*)
Provisioned dynamically when TVM initialized or requests

Figure 6: Attestation Certificate generation

This attestation certificate can be used in a challenge/response protocol to a remote relying party which must verify the certificate chain for the attestation key used to sign the relying party challenge.

The Attestation key and certificate generation for TVMs may be performed with a U-mode TSM component called the Owner Signing Authority Module (OSAM) to enable a extension of the TCB to support interruptible signing operations. The OSAM may execute as part of the TSM or may be executed in the TSM U-mode to allow for the interruptibility models discussed in the TSM operation section of this document.

TVM Attestation:

X.509 CDI Certificates are used to enable Attestation certificates derived from the TSM CDI for each TVM hosted on the platform. All standard fields of a CDI certificate are described in the following table. This certificate can be generated given a CDI_Public key and the DICE input values.

Field	**Description**
signatureAlgorithm	id-ecdsa-with-SHA256 per RFC 5758 recommended. Other signatureAlgorithms may be used.
signatureValue	64 byte ECDSA signature, using UDS_Private or a previous CDI_Private as the signing key
version	v3
serialNumber	CDI_ID in ASN.1 INTEGER form
signature	id-ecdsa-with-SHA256 per RFC 5758
issuer	“<UDS_ID> or <CDI_ID>” UDS_ID, CD_ID are hex encoded lower case
validity	The validity values are populated as follows: notBefore can be any time known to be in the past, and notAfter is set to the standard value used to indicate no well-known expiry date, “99991231235959Z” per RFC 5280.
subject	“<CDI_ID>” where CDI_ID is hex encoded lower case

subjectPublicKeyInfo	When using ECDSA, per RFC 5480 (id-ecPublicKey)
issuerUniqueID	Not used
subjectUniqueID	Not used
extensions	Standard extensions are included as well as a custom TCG extension which holds information about the measurements used to derive CDI values. Both are described below.

CDI Standard Extensions

Extension	Critical	Description
authorityKeyIdentifier	non-critical	Contains only keyIdentifier set to UDS_ID or previous CDI_ID
subjectKeyIdentifier	non-critical	Set to CDI_ID
keyUsage	critical	Contains only keyCertSign. Other CDI certificates may be generated for other purposes for the TVM.
basicConstraints	critical	The cA field is set to TRUE. The pathLenConstraint field is normally not included, but may be included and set to zero if it is known that no additional DICE layers exist. For example, for TVMs, this field may be set to zero.

CDI Custom Extension Fields

Field	Value
extnID	OID from [2] for TcbEvidenceInfo
critical	TRUE
extnValue	A TcbEvidenceInfo (See above)

The TSM can issue an Attestation certificate to the TVM which includes the TVM TcbInfo, and can transfer that certificate to the TVM during initialization via a guest firmware mechanism (e.g. device tree or UEFI HOB). Alternately, the TSM can provide an interface to sign TVM TcBInfo and additional data (such as DRTM measurements done by the TVM) at runtime via the teecall_tg_gen_cert interface to generate additional TVM Attestation certificates.

ECALL (**teecall_tg_gen_cert**): invoked by TVM - this TEEI operation is serviced by the TSM.

Inputs/outputs

- Input: virtual address to 4KB buffer containing a CSR (Certificate Signing Request) and additional parameters (nonce)
- Input/output:virtual address to 4KB aligned buffer where TSM certificate will be returned

Validation

- Set result register to indicate failure
- Verify VA where TVM Attestation certificate will be returned is 4KB aligned and read/write else fault
- Verify TVM provided CSR <size TBD> is contained within a 4KB page and read accessible else fault

Setup

- Create TVM attestation structure in a temporary buffer in per-hart confidential memory
- Populate TVM TcbEvidenceInfo per the TVM measurements recorded by the TSM
- Copy additional data from CSR <TBD>

Process

- Compute attestation certificate (per certificate fields and extensions described above) using TSM as the DICE for TVM

Outputs

- Copy out attestation structure to TSM verified memory region
- Set result register to indicate success

Chapter 7. TVM Lifecycle

This section describes the TEEI operations for the lifecycle of a TVM including the OS/VMM interactions with the TSM.

7.1. TVM build and initialization

The host OS/VMM must be capable of hosting many TVMs on a AP-TEE-capable platform (limited only by the practical limits of the number of cpus and the amount of memory available on the system). To that end, the TVM should be able to use all of the system memory as TEE-capable memory, as long as the platform access-control mechanisms are applicable to all the available memory on the system. The TSM allows the OS/VMM to manage TEE-capable memory assignment by providing a two stage TEE memory management model. 1. Creation of confidential memory regions - this process converts memory pages from non-confidential to confidential memory (and in that process brings TEE-capable memory under TSM-managed memory tracking and encryption controls described earlier). 2. Allocation/Assignment of TEE-capable memory pages from the converted confidential memory regions for various purposes like creating TVM workloads etc.

The host OS/VMM may create a new TVM by allocating and initializing a TVM using the teecall_tvm_create function. As inputs to this TEECALL, the OS/VMM must assign the TVM with a unique identifier. For example, a physical address for a TVM global control structure may be used as a platform unique identifier or handle. An initial set of memory pages are granted to the TSM and tracked as TEE pages associated with that TVM from that point onwards until the TVM is destroyed via the teecall_tvm_destroy function.

A TVM context may be created and initialized by using the teecall_tvm_create_init function – this global init function allocates a set of pages for the TVM global control structure and resets the control fields that are immutable for the lifetime of the tvm e.g. configuration of which RISC-V CPU extensions the TVM is allowed to use, debug and pmon capabilities enabled etc.

The VMM may assign memory to the TVM via a sequence of teecall_tvm_page_map_add and teecall_tvm_page_add – the former grants memory pages that are to contain second-stage paging structures entries that translate a TVM guest physical address to the system physical address, while the latter is used to hold tvm data and is referenced by the hgatp leaf page table entries. For pages added to the TVM, the VMM must invoke teecall_tvm_msmt_extend which extends the static measurement hash of the TVM which will be used by the TSM to generate the attestation report (evidence) when requested by a challenger (relying party). Note that if the measurement steps are executed by the VMM in an incorrect order the final measurements will be different and flagged during attestation. In the initial set of measured TVM pages, the VMM would typically provide the guest firmware, boot loader and boot kernel as well as memory needed for the boot stack, heap and memory tracking structures. During teecall_tvm_page_add, the memory granted is tracked by the TSM to ensure that pages assigned to a TVM may not be assigned to a non-confidential VM or another TVM. Memory may be lazily added to the TVM subsequent to the TVM being executed using the teecall_tvm_page_add_post.

Lastly, the VMM can assign memory to the TVM to hold virtual hart state via teecall_tvm_vhart_add and teecall_tvm_vhart_init. Before the VMM can start executing the TVM virtual harts, the VMM must finalize the static measurement of the TVM via teecall_tvm_msmt_commit. The TSM prevents any TVM virtual harts from being entered until the TVM initialization is finalized.

7.2. TVM execution

The VMM uses teecall_tvm_enter to (re)activate a virtual hart for a specific TVM (identified by the unique identifier). This TEECALL traps into the TSM-driver which affects the context switch to the TSM – The TSM then manages the activation of the virtual hart on the physical hart selected by the VMM. During this activation the TCB trusted firmware can enforce that stale TLB entries that govern guest physical to system physical page access have been evicted across all hart TLBs. There may also be TLB flushes for the virtual-harts due to first stage translation changes (guest virtual to guest physical) performed by the TVM OS - these are initiated by the TVM OS to cause IPIs to the virtual-harts managed by the TVM OS (and verified by the TVM OS to ensure the IPIs are received by the TVM OS to invalidate the TLB lazily). This reference architecture requires use of AiA IMSIC [R9] to ensure these IPIs are delivered through the IMSIC associated with the guest TVM. Each TVM is allocated a guest interrupt file during TVM initialization.

During TVM execution, the HW enforces TSM-driven policies for memory isolation for confidential memory accessed by the TVM software – the following hardware enforcement is recommended to address the [4_architecture_overview_and_threat_model]:

- TVM instruction fetches and page walks (both VS/second-stage and G/first-stage) are implicitly enforced to be in confidential memory. This requires that the TVM supervisor code should not locate first stage page tables in non-confidential memory. The TSM enforces that second stage page tables are in confidential memory.
- TVM access to confidential or non-confidential memory is subject to VS-stage address translation (this is existing); G-stage address translation is enforced via the TSM-managed hgatp with the listed recommendations in Section [5_3_tsm_and_tvm_isolation].

For virtual-IO operations, the TVM code must first explicitly request TVM memory to become non-confidential (via teecall_tg_page_share) and then explicitly copy data from confidential to non-confidential memory, and lastly signal service requests to the host VMM (via teecall_tvm_host_req). When direct device assignment is supported (which is expected to require IOMMU changes for AP-TEE), trusted devices may DMA directly into TVM confidential memory.

TVM memory may be lazily granted to the TVM by the host VMM, the TVM kernel must explicitly accept such un-measured initialized pages via teecall_tg_page_accept, and update its internal memory database to indicate those guest physical page frames are trusted for mapping into VS-stage mappings. There are at least two scenarios here - first, late add of memory to enable TVM boot with the minimal measured state, and second, if some memory pages were converted to non-confidential by the TVM, and at a later point they are converted back to confidential, the TVM must accept those GPAs (else a VMM can sneak in an unmeasured page late for example).

During execution and typically during TVM initialization, the TVM code can extend the runtime measurement registers by invoking the teecall_tvm_drtm_extend – this allows the TVM to measure the next stage of kernel or application modules that are loaded in the TVM.

Also during execution, a remote relying party may challenge the TVM to provide attestation evidence that the TVM is executing as a HW-rooted TEE. The TVM code may in response request a

TSM-signed (hence HW-measurement rooted) attestation evidence via teecall_tg_get_evidence - this evidence structure contains signed hash of the TVM measurements (including the run-time and static measurements) and is replay-protected via a TVM (challenger) provided nonce as part of the signed evidence.

The TSM enforces specific security checkpoints during TVM execution – it tracks when TLB flushes are required by the VMM to ensure stale TLB entries are not utilized by the TVM. To enforce this property, the TSM requires second-stage-page-table-mapped confidential TVM memory to be blocked (effectively ensuring new TLB entries cannot be created) before the pages mapped by the mapping can be relocated, fragmented (for page promotion or demotion) or reclaimed back by the VMM. Then, before the the new mappings may be activated, the TSM enforces that the VMM invokes teecall_tvm_fence and causes invalidation of the TLB on all virtual harts of the TVM via interprocessor interrupts which causes the TSM to execute HFENCE.GVMA for the TVM VMID.

7.3. TVM shutdown

The VMM may stop a TVM virtual hart at any point (same as legacy operation for the VMM but in this case via the TSM). If the TVM being shutdown is executing, the VMM stops TVM execution by issuing an asynchronous interrupt that yields the virtual hart and taking control back into the VMM (without any TVM state leakage as that is context saved by the TSM on the trap due to the interrupt). Once the TVM virtual harts are stopped, the VMM must issue a teecall_tvm_shutdown that can verify that no TVM harts are executing and then may reclaim all memory granted to the TVM via teecall_tvm_page_reclaim which will verify the TSM hgatp mapping and tracking for the page and restore it as a VMM-available page to grant to another TVM or non-confidential VM. The VMM should revoke the TVM global control page as the last page via teecall_tvm_destroy.

Chapter 8. Trusted Execution Environment (TEE) SBI extension proposal

This section describes the normative Trusted Execution Environment (TEE) SBI extension proposal. The proposal introduces three new extensions that will be described later:

- TEE Host Extension (EXT_TEE_HOST),
- TEE Interrupt Extension (EXT_TEE_INTERRUPT), and
- TEE Guest Extension (EXT_TEE_GUEST).

8.1. TEEI - TH-ABI runtime interface

ECALL invocation from VS (guest OS) causes traps that are handled by the TSM module (enforced via mdeleg configuration). The TSM then may provide intrinsics via the TG-ABI (TEE-Guest ABI) to the TVM to provide attestation and other trusted services. The TSM may allow the TEE (application or VM) to request host (untrusted) services via the TH-ABI (TEE host-ABI).

8.1.1. Background

Virtualization platforms are typically comprised of several components including platform firmware, host OS, VMM, and the actual payloads that run on them (typically in a VM). This model is well established, but the downside is that most platform components are in the TCB. This aspect is ill-suited for confidential compute workloads that rely on Trusted Execution Environments, and strive to minimize the TCB footprint.

We seek to alleviate this concern by introducing the notion of the a trusted broker for TEEs known as the TEE Security Manger (TSM). The idea is that the TSM has a minimal possible footprint, and acts as a trusted intermediary between the the untrusted platform components, and the TEE workloads that run on it. The TCB (which includes the TSM and HW) enforces strict confidentiality and integrity security properties for code execution and data integrity for confidential workloads. It also isolates confidential workloads from all other platform components (non-confidential and confidential). The exact hardware and software mechanisms used to guarantee the isolation are platform dependent, but confidential workloads have execution, data integrity, and confidentiality protections from any other platform component. The responsibility of the TSM is to enforce the guarantees accorded to TEE workloads. Specifically, security invariants for confidential workloads are enforced by the TSM. The VMM manages security for non-confidential workloads, and resource and scheduling management functions for all workloads (confidential and non-confidential).

In this scheme, compute resources like memory start off as traditional untrusted resources in the non-confidential world, and are transtioned to their confidential analogue via the TSM. Once the conversion process is complete, it can be assigned to a TVM via the TSM. A converted confidential-resource can be freely assigned to another TVM when it’s no longer in use. However, an unused confidential-resource must be explicitly reclaimed for use in the non-confidential world (this is tracked and enforced by the TSM).

The TEE address space can be comprised of confidential and non-confidential regions. The former includes both measured pages (that are part of the initial TVM payload), and confidential zero-pages that can be mapped-in on demand by the VMM following runtime access by the TVM. The non-confidential TVM-defined regions include those for shared-pages and MMIO. Without direct-IO support to TVMs, MMIO regions are non-confidential. When direct IO is supported - MMIO page mappings will also be allowed to be in the confidential memory region.

8.1.2. Operational model for the TEE Host Extension

Executing confidential workloads in a TEE requires a sequence of one or more of the steps detailed below. We'll assume that these steps are performed by an untrusted entity like the OS/VMM (host) in conjunction with the TSM.

1. Platform TSM detection and capability enumeration
2. Conversion of non-confidential memory to confidential memory
3. Trusted VM (TVM) creation
4. Donating confidential memory to the TSM for TVM page management
5. Defining TVM confidential memory regions
6. Mapping TVM code and data payload to confidential-memory regions
7. Creating TVM VCPUs
8. Finalizing TVM creation
9. Scheduling TVM execution
10. Management of TVM secure interrupts
11. Handling and servicing TVM faults and exits
12. Mapping TVM demand-zero confidential memory regions
13. Mapping TVM non-confidential shared pages on demand
14. Processing TVM-access to MMIO regions
15. Tearing down TVMs
16. Reassignment of confidential memory for other TVMs
17. Reclaiming confidential memory for non-confidential VMs

Platform TSM detection and capability enumeration

Platform support for the TSM can be detected by probing for the EXT_TEE_HOST extension, and then calling `sbi_tee_host_get_tsm_info()` to get information about the current status of the TSM. The TSM must be be in `TSM_READY` in order to process further ECALLs.

TVM creation

TVMs are created using the sbi_tee_host_create_tvm(). This creates a TVM with state set to `TVM_INITIALIZING`. The host must assign confidential memory for page tables, payload mapping, and VCPUs before it can be transitioned into a `TVM_RUNNABLE` state.

TVM memory management

The host is responsible for the following memory management functions:

1. Converting non-confidential memory to confidential memory
2. Donating confidential memory for the TVM page-table pool
3. Defining confidential memory regions
4. Mapping TVM code and data payload to confidential TVM-pages
5. Mapping zero-page confidential pages to the TVM regions
6. Mapping non-confidential pages TVM-defined regions for shared-pages / MMIO

Converting non-confidential memory to confidential memory

Platform memory is non-confidential by default, and must be converted to confidential memory before use with TVMs. The conversion process is initiated by designating the host physical pages that to be converted, and then issuing fence operations to ensure that all outstanding TLB entries to the non-confidential memory are flushed across all CPUs/harts on the platform. This ensures that there's no overlapping mapping between the confidential and non-confidential memory regions on the platform.

This requires the host to make three separate ECALLs to the TSM:

1. sbi_tee_host_convert_pages()
2. sbi_tee_host_global_fence()
3. sbi_tee_host_local_fence()

The memory conversion process is complete when sbi_tee_host_local_fence() is successfully completed on the CPU/hart on the platform.

Converted memory can be assigned to TVMs, but cannot be repurposed for non-confidential operations unless it's reclaimed. If the host assigns converted memory to non-confidential VMs, or uses it for page-table mappings, access to the converted memory from inside the non-confidential VM will cause an access fault.

Defining confidential memory regions

The host can declare the TVM physical address ranges for mapping of confidential memory. There can be multiples ranges, but no two regions can overlap. The region can be sparsely mapped; however, any sparsely mapped confidential page that's demand-paged following an access fault by the TVM can only be a demand-zero page.

All ranges must be defined by calling `sbi_tee_host_finalize_tvm()`.

Donating confidential pages for the TVM page-table pool

The host must ensure that the TSM has sufficient confidential memory for mapping and managing TVM page-tables for the code and data payloads by calling `sbi_tee_host_add_tvm_page_table_pages()`.

Mapping TVM code and data payload to confidential TVM-pages

The host can create a confidential page region by calling `sbi_tee_host_add_tvm_memory_region()` with `CONFIDENTIAL_REGION`. The region can be sparsely populated, and since the host cannot directly access confidential memory, it must copy the TVM code and data payload from non-confidential memory to confidential memory by calling `sbi_tee_host_add_tvm_measured_pages()`. This operation requires the host to convert a sufficient number of non-confidential pages to confidential (by calling `sbi_tee_host_convert_pages()`, or by using converted page that aren't currently assigned to a TVM. The TSM copies the payload for the TVM from non-confidential pages to confidential pages, and extends the corresponding measurements for the TVM.

VCPU shared state enumeration

The TSM communicates additional information about TVM exits from `sbi_tee_host_run_tvm_vcpu()` using a non-confidential shared memory region that's configured on a per-VCPU basis by the host. The host can also use this shared memory region to control and configure TVM parameters like the initial-entry point (SEPC), initial parameter, etc., and to respond to TVM exits.

The layout of the shared-memory region can vary by TSM version. The host can determine the size and offset of the regions enumerated in `vcpu_register_set_id` by calling `sbi_tee_host_get_tvm_vcpu_num_register_sets()` to get the number of enumerated sets, and then `sbi_tee_host_get_tvm_vcpu_num_register_sets()` to determine the offset.

VCPU creation

The host must register CPUs/harts with the TSM before they can be used for TVM execution by calling `sbi_tee_host_create_tvm_vcpu()`. The host must also configure the the the non-confidential shared memory that's set-up by the host while creating the VCPU. The shared memory is used both the host and the TSM for when processing TVM exits from `sbi_tee_run_vcpu()`.

TVM execution

Following assignment of memory and VCPU resources, the host can transition the guest into a `TVM_RUNNABLE` state by calling `sbi_tee_host_finalize_tvm()`. Note that some TEE calls are no longer permissible after this transition.

The host can use the aforementioned shared-memory to set up TVM execution parameters like the entrypoint (`ENTRY_PC`) / boot argument (`ENTRY_ARG`), then `sbi_tee_host_finalize_tvm()`, followed by sbi_tee_host_run_tvm_vcpu()` to begin execution. TVM execution continues until there an event like an interrupt, or fault that cannot be serviced by the TSM. Some interrupts and exceptions are resumable, and the host can determine reason specific reason by examining the `scause` field in the `tvm_vcpu_supervisor_csrs` previously setup by the call to `sbi_tee_host_create_tvm_vcpu()`. The host can then examine the shared-memory region if needed to determine further course of action. This may involve servicing exits caused by TVM-ECALLs that require host action(like adding of MMIO and shared-memory regions), TVM page-faults, virtual instructions, etc.

Mapping confidential demand-zero pages and non-confidential shared pages

The host can handle TVM page-faults by determining whether it was caused by access to a confidential or non-confidential region. In the former case, it can call use `sbi_tee_host_add_tvm_zero_pages()` to populate the region with a previously converted confidential page. The TSM verifies that the confidential page isn't currently in use, and zeroes it out before assigning it the TVM. Demand-zero pages have no bearing on the TVM measurement, and can be added at any point of time.

The host can process non-confidential pages by calling `sbi_tee_host_add_shared_pages()`. Non-confidential shared memory regions are defined by the TVM using the EXT_TEE_GUEST extension.

Handling MMIO faults

TVMs can define MMIO regions using the EXT_TEE_GUEST extension, and a rutime access to such a region causes a resumable exit from the TVM. The host can examine the exit code and `scause`, update the per-VCPU shared-memory region as appropriate, and resume TVM execution. This may involve instruction decoding using the information from the shared-memory region.

Handling virtual instructions

The host can handle exits caused by virtual instruction by examining and decoding the contents of the shared-memory region.

Management of secure interrupts

The host can use the Tee Interrupt Extension (EXT_TEE_INTERRUPT) to manage secure TVM interrupts on platforms with AIA-support.

TVM teardown

The host can teardown a TVM by calling `sbi_tee_host_destroy_tvm()`. This automatically releases all confidential memory assigned to the TVM, and it can be repurposed for use with other TVMs. However, reclaiming the memory for use by non-confidential workloads requires an explicit call to `sbi_tee_host_reclaim_pages()`.

8.1.3. Operational model for the TEE Guest Extension

This interface is used by TVMs to communicate with TSM. Presently, this extension only allows guests to define memory regions for shared-pages and MMIO regions.

TVM-defined memory regions

TVMs can determine the physical address location for mapping of non-confidential regions at runtime, and communicate the decision host about TVM-established shared-pages and MMIO regions by calling `sbi_tee_guest_add_memory_region()`. This results in an exit to the host, and it can retrieve the information by checking the exit code from the TVM and examining the shared-memory region for the TVM VCPU. The expectation is that the host will service a subsequent page-fault that results from a TVM-access to the non-confidential region.

TVM-driven confidential/non-confidential memory conversion

TVMs can choose to yield access to confidential memory at runtime and request shared (non-confidential) memory. The TVM must communicate it's request to the host to convert confidential to non-confidential and vice-versa explicitly via the `sbi_tee_guest_(un)share_memory_region`. This request results in an exit to the TSM which enforces the security properties on the mapping and exits to the VMM host to enforce TLB invalidation. The expectation is that the host will service these requests and handle subsequent page-faults to allow the TVM to access required confidential or non-confidential memory per the TVMs request.

Chapter 9. TEE Host Extension (EID #0x54454548)

9.1. Listing of common enums

The following enums are referenced by several functions described below.

```
enum tsm_page_type {
    /* 4KiB */
    PAGE_4K = 0,
    /* 2 MiB */
    PAGE_2MB = 1,
    /* 1 GiB */
    PAGE_1GB = 2,
    /* 512 GiB */
    PAGE_512GB = 3,
}
```

```
enum tvm_state {
    /* The TVM has been created, but isn't yet ready to run */
    TVM_INITIALIZING = 0,
    /* The TVM is in a runnable state */
    TVM_RUNNABLE = 1,
};
```

```
enum vcpu_register_set_id {
    /* General purpose registers */
    GPRS = 0,
    /* Supervisor CSRs */
    SUPERVISOR_CSRS = 1,
    /* Hypervisor (HS-level) CSRs */
    HYPERVISOR_CSRS = 2,
};
```

```
/*
 * General purpose registers for he TVM VCPU.
 * Corresponds to `GPRS` in `vcpu_register_set_id`.
 */
struct tvm_vcpu_supervisor_gprs {
    /*
     * Indexed VCPU GPRs from X0 - X31.
     *
     * The TSM will always read or write the minimum number of registers in this set
to
```

```
     * complete the requested action, in order to avoid leaking information from the
TVM.
     *
     * The TSM will write to these registers upon return from `TvmCpuRun` when:
     * 1) The VCPU takes a store guest page fault in an emulated MMIO region.
     * 2) The VCPU makes an ECALL that is to be forwarded to the host.
     *
     * The TSM will read from these registers when:
     * 1) The VCPU takes a load guest page fault in an emulated MMIO region.
     * 2) The host calls `sbi_tee_host_finalize_tvm()`, latching the entry point
argument
     * (stored in 'A1') for the boot VCPU.
     *
     */
    unsigned long gprs[32];
};
```

```
/*
 * Hypervisor [HS-level] CSRs.
 * Corresponds to `HYPERVISOR_CSRS` in `vcpu_register_set_id`.
 */
struct tvm_vcpu_hypervisor_csrs {
    /*
     *
     * HTVAL value for guest page faults taken by the TVM vCPU. Written by the TSM
upon return
     * `sbi_tee_host_run_tvm_vcpu()`.
     *
     */
    unsigned long htval;
    /*
     *
     * HTINST value for guest page faults or virtual instruction exceptions taken by
the TVM vCPU.
     *
     * The TSM will only write `htinst` in the following cases:
     *
     * MMIO load page faults. The value written to the register in `gprs`
corresponding to the
     * 'rd' register in the instruction will be used to complete the load upon the
next call to
     * `sbi_tee_host_run_tvm_vcpu()` for this vCPU.
     *
     * MMIO store page faults. The TSM will write the value to be stored by the vCPU
to the
     * register in `gprs` corresponding to the 'rs2' register in the instruction upon
return
     * from `sbi_tee_host_run_tvm_vcpu()`.
     *
     */
    unsigned long htinst;
};
```

```
/*
 * Supervisor-level CSRs.
 * Corresponds to `SUPERVISOR_CSRS` in `vcpu_register_set_id`.
 */
struct tvm_vcpu_supervisor_csrs {
    /*
     * Initial SEPC value (entry point) of a TVM vCPU. Latched for the TVM's boot VCPU
when
     * sbi_tee_host_finalize_tvm() is called; ignored for all other VCPUs.
     */
    unsigned long sepc;
    /*
     * SCAUSE value for the trap taken by the TVM vCPU. Written by the TSM upon return
from
     * `sbi_tee_host_run_tvm_vcpu()`
     */
    unsigned long scause;
    /*
     * STVAL value for guest page faults or virtual instruction exceptions taken by
the TVM VCPU.
     * Written by the TSM upon return from sbi_tee_host_run_tvm_vcpu()
     *
     * Note that guest virtual addresses are not exposed by the TSM, so only the 2
LSBs will
     * ever be non-zero for guest page fault exceptions.
     */
    unsigned long stval;
};
```

```
struct tvm_vcpu_register_set_location {
    /*
     * A value of enum type `vcpu_register_set_id`.
     */
    uint16_t id;
    /*
     * The offset of the register set from the start of the VCPU's shared-memory state
area.
     */
    uint16_t offset;
};
```

9.2. Function: TEE Host Get TSM Info (FID #0)

```
struct sbiret sbi_tee_host_get_tsm_info(unsigned long tsm_info_address,
                                        unsigned long tsm_info_len);
```

Writes up to `tsm_info_len` bytes of information at the physical memory address specified by `tsm_info_address`. `tsm_info_len` should be the size of the the `tsm_info` struct below. The information returned by the call can be used to determine the current state of the TSM, and configure parameters for other TVM-related calls.

Returns the number of bytes written to `tsm_info_address` on success.

```
enum tsm_state {
    /* TSM has not been loaded on this platform. */
    TSM_NOT_LOADED = 0,
    /* TSM has been loaded, but has not yet been initialized. */
    TSM_LOADED = 1,
    /* TSM has been loaded & initialized, and is ready to accept ECALLs.*/
    TSM_READY = 2
};

struct tsm_info {
    /*
     * The current state of the TSM (see tsm_state enum above). If the state is not
TSM_READY,
     * the remaining fields are invalid and will be initialized to 0.
     */
    uint32_t tsm_state;
    /* Version number of the running TSM. */
    uint32_t tsm_version;
    /*
     * The number of 4KiB pages which must be donated to the TSM for storing TVM
     * state in sbi_tee_host_create_tvm_vcpu().
     */
    unsigned long tvm_state_pages;
    /* The maximum number of VCPUs a TVM can support. */
    unsigned long tvm_max_vcpus;
    /*
     * The number of 4kB pages which must be donated to the TSM when
     * creating a new VCPU.
     */
    unsigned long tvm_vcpu_state_pages;
};
```

The possible error codes returned in `sbiret.error` are shown below.

Table 1. TEE Host Get TSM Info

Error code	Description
SBI_SUCCESS	The operation completed successfully.

Error code	Description
SBI_ERR_INVALID_ADDRESS	`tsm_info_address` was invalid.
SBI_ERR_INVALID_PARAM	tsm_info_len was insufficient.
SBI_ERR_FAILED	The operation failed for unknown reasons.

A list of possible TSM states and the associated semantics appears below (TBD: States for TSM update).

Table 2. TSM States

TSM State	Meaning
TSM_NOT_LOADED	TSM has not been loaded on this platform.
TSM_LOADED	TSM has been loaded, but has not yet been initialized.
TSM_READY	TSM has been loaded & initialized, and is ready to accept ECALLs.

9.3. Function: TEE Host Convert Pages (FID #1)

```
struct sbiret sbi_tee_host_convert_pages(unsigned long base_page_address,
                                         unsigned long num_pages);
```

Begins the process of converting `num_pages` of non-confidential memory starting at `base_page_address` to confidential-memory. On success, pages can be assigned to TVMs only following subsequent calls to `sbi_tee_host_global_fence()` and `sbi_tee_host_local_fence()` that complete the conversion process. The implied page size is 4KiB.

The `base_page_address` must be page-aligned.

The possible error codes returned in `sbiret.error` are shown below.

Table 3. TEE Host Convert Pages

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_ADDRESS	`base_page_address` was invalid.
SBI_ERR_INVALID_PARAM	`num_pages` was invalid.
SBI_ERR_FAILED	The operation failed for unknown reasons.

9.4. Function: TEE Host Reclaim Pages (FID #2)

```
struct sbiret sbi_tee_host_reclaim_pages(unsigned long base_page_address,
                                         unsigned long num_pages);
```

Reclaims `num_pages` of confidential memory starting at `base_page_address`. The pages must not be currently assigned to an active TVM. The implied page size is 4KiB.

The possible error codes returned in `sbiret.error` are shown below.

Table 4. TEE Host Reclaim Pages

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_ADDRESS	`base_page_address` was invalid.
SBI_ERR_INVALID_PARAM	`num_pages` was invalid.
SBI_ERR_FAILED	The operation failed for unknown reasons.

9.5. Function: TEE Host Initiate Global Fence (FID #3)

```
struct sbiret sbi_tee_host_global_fence(void);
```

Initiates a TLB invalidation sequence for all pages marked for conversion via calls to `sbi_tee_host_convert_pages()`. The TLB invalidation sequence is completed when `sbi_tee_host_local_fence()` has been invoked on all other CPUs. An error is returned if a TLB invalidation sequence is already in progress.

The possible error codes returned in `sbiret.error` are shown below.

Table 5. TEE Host Initiate Fence

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_ALREADY_STARTED	A fence operation is already in progress.
SBI_ERR_FAILED	The operation failed for unknown reasons.

9.6. Function: TEE Host Local Fence (FID #4)

```
struct sbiret sbi_tee_host_local_fence(void);
```

Invalidates TLB entries for all pages pending conversion by an in-progress TLB invalidation operation on the local CPU.

The possible error codes returned in `sbiret.error` are shown below.

Table 6. TEE Host Local Fence

Error code	Description
SBI_SUCCESS	The operation completed successfully.

Error code	Description
SBI_ERR_FAILED	The operation failed for unknown reasons.

9.7. Function: TEE Host Create TVM (FID #5)

```
struct sbiret sbi_tee_host_create_tvm(unsigned long tvm_create_params_addr,
                                       unsigned long tvm_create_params_len);
```

Creates a confidential TVM using the specified parameters. The `tvm_create_params_addr` is the physical address of the buffer containing the `tvm_create_params` structure described below, and `tvm_create_params_len` is the size of the structure in bytes.

Callers of this API should first invoke `sbi_tee_host_get_tsm_info()` to obtain information about the parameters that should be used to populate `tvm_create_params`.

```
struct tvm_create_params {
    /*
     * The base physical address of the 16KiB confidential memory region
     * that should be used for the TVMs page directory. Must be 16KiB-aligned.
     */
    unsigned long tvm_page_directory_addr;
    /*
     * The base physical address of the confidential memory region to be used
     * to hold the TVMs state. Must be page-aligned and the number of
     * pages must be at least the value returned in tsm_info.vm_state_pages
     * returned by the call to sbi_tee_host_get_tsm_info().
     */
    unsigned long tvm_state_addr;
    /*
     * The vcpuid for the VCPU that will be designated as the boot VCPU.
     * The host must add create a VCPU with this vcpuid by calling
`sbi_tee_host_create_tvm_vcpu`
     * before calling `sbi_tee_host_finalize_tvm().
     */
    unsigned long tvm_boot_vcpuid;
};
```

Returns the **`tvm_guest_id`** in sbiret.value on success. The **`tvm_guest_id`** can be used to uniquely reference the TVM in invocations of the other functions that appear below. On success, the TVM will be in the "TVM_INITIALIZING" state, until a subsequent call to `sbi_tee_host_finalize_tvm()` to transition to it a `TVM_RUNNABLE` state.

The list of possible TVM states appears below.

Table 7. TEE TVM States

State	Description

| TVM_INITIALZING | The TVM has been created, but isn't yet ready to run. | TVM_RUNNABLE | The TVM is in a runnable state, and can be executed by | calling `sbi_tee_host_run_tvm_vcpu()`.

The possible error codes returned in `sbiret.error` are shown below.

Table 8. TEE Host Create TVM Errors

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_ADDRESS	`tvm_create_params_addr` was invalid.
SBI_ERR_INVALID_PARAM	`tvm_create_params_len` was invalid.
SBI_ERR_FAILED	The operation failed for unknown reasons.

9.8. Function: TEE Host Finalize TVM (FID #6)

```
struct sbiret sbi_tee_host_finalize_tvm(unsigned long tvm_guest_id);
```

Transitions the TVM specified by `tvm_guest_id` from the "TVM_INITIALIZING" state to a "TVM_RUNNABLE" state. The host must finalize TVM shared-memory execution parameters like the entry point (`ENTRY_PC`) and boot argument (`ENTRY_ARG`) on the boot VCPU configured by `sbi_tee_host_create_tvm()` before making this call.

The possible error codes returned in `sbiret.error` are shown below.

Table 9. TEE Host Finalize TVM Errors

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_PARAM	`tvm_guest_id` was invalid, or the TVM wasn't in the `TVM_INITIALIZING` state.
SBI_ERR_FAILED	The operation failed for unknown reasons.

9.9. Function: TEE Host Destroy TVM (FID #7)

```
struct sbiret sbi_tee_host_destroy_tvm(unsigned long tvm_guest_id);
```

Destroys a confidential TVM previously created using **`sbi_tee_host_create_tvm()`**.

Confidential TVM memory is automatically released following successful destruction, and it can be assigned to other TVMs. Repurposing confidential memory for use by non-confidential TVMs requires an explicit call to **`sbi_tee_host_reclaim_pages()`** (described below).

The possible error codes returned in `sbiret.error` are shown below.

Table 10. TEE Host Destroy TVM Errors

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_PARAM	`tvm_guest_id` was invalid.
SBI_ERR_FAILED	The operation failed for unknown reasons.

9.10. Function: TEE Host Add TVM Memory Region (FID #8)

```
struct sbiret sbi_tee_host_add_tvm_memory_region(unsigned long tvm_guest_id,
                                                 unsigned long tvm_gpa_addr,
                                                 unsigned long region_len);
```

Marks the range of TVM physical address space starting at `tvm_gpa_addr` as reserved for the mapping of confidential memory. The memory region length is specified by `region_len`.

Both `tvm_gpa_addr` and `region_len` must be 4kB-aligned, and the region must not overlap with a previously defined region. This call must not be made after calling `sbi_tee_host_finalize_tvm()`.

```
enum tvm_memory_region_type {
    /*
     * Reserved for mapping confidential pages. The region is initially unpopulated,
and pages
     * of confidential memory can be inserted by calling
`sbi_tee_host_add_tvm_zero_pages()` and
     * `sbi_tee_host_add_tvm_measured_pages().
     */
    CONFIDENTIAL_REGION = 0,
    /*
     * The region is initially unpopulated, and pages of shared memory may be inserted
by calling
     * `sbi_tee_host_add_tvm_shared_pages()`. Attempts by a TVM VCPU to access an
unpopulated region
     * will cause a `SHARED_PAGE_FAULT` exit from `sbi_tee_host_run_tvm_vcpu()`.
     */
    SHARED_MEMORY_REGION = 1,
    /*
     * The region is unpopulated; attempts by a TVM VCPU to access this region will
cause a
     * `MMIO_PAGE_FAULT` exit from `sbi_tee_host_run_tvm_vcpu()`.
     */
    EMULATED_MMIO_REGION = 2,
};
```

The possible error codes returned in `sbiret.error` are shown below.

Table 11. TEE Host Add TVM Memory Region

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_ADDRESS	`tvm_gpa_addr` was invalid.
SBI_ERR_INVALID_PARAM	`tvm_guest_id` or `region_len` were invalid, or the TVM wasn't in the correct state.
SBI_ERR_FAILED	The operation failed for unknown reasons.

9.11. Function: TEE Host Add TVM Page Table Pages (FID #9)

```
struct sbiret sbi_tee_host_add_tvm_page_table_pages(unsigned long tvm_guest_id,
                                                    unsigned long base_page_address,
                                                    unsigned long num_pages);
```

Adds `num_pages` confidential memory starting at `base_page_address` to the TVM's page-table page-pool. The implied page size is 4KiB.

Page table pages may be added at any time, and a typical usecase is in response to a TVM page fault.

The possible error codes returned in `sbiret.error` are shown below.

Table 12. TEE Host Add TVM Page Table Pages

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_ADDRESS	`base_page_address` was invalid.
SBI_ERR_OUT_OF_PTPAGES	The operation could not complete due to insufficient page table pages.
SBI_ERR_INVALID_PARAM	`tvm_guest_id` or `num_pages` were invalid, or `tsm_page_type` is invalid.
SBI_ERR_NOT_SUPPORTED	The `tsm_page_type` isn't supported by the TSM.
SBI_ERR_FAILED	The operation failed for unknown reasons.

9.12. Function: TEE Host Add TVM Measured Pages (FID #10)

```
struct sbiret sbi_tee_host_add_tvm_measured_pages(unsigned long tvm_guest_id,
                                                  unsigned long source_address,
                                                  unsigned long dest_address,
                                                  unsigned long tsm_page_type,
                                                  unsigned long num_pages,
                                                  unsigned long tvm_guest_gpa);
```

Copies num_pages pages from non-confidential memory at `source_address` to confidential memory at `dest_addr`, then measures and maps the pages at `dest_addr` at the TVM physical address space at `tvm_guest_gpa. The mapping must lie within a region of confidential memory created with `sbi_tee_host_add_tvm_memory_region()`. The tsm_page_type parameter must be a legal value for enum type `tsm_page_type`.

This call must not be made after calling `sbi_tee_host_finalize_tvm()`.

The possible error codes returned in `sbiret.error` are shown below.

Table 13. TEE Host Add TVM Measured Pages

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_ADDRESS	`source_address` was invalid, or `dest_address` wasn't in a confidential memory region.
SBI_ERR_INVALID_PARAM	`tvm_guest_id`, `tsm_page_type`, or `num_pages` were invalid, or the TVM wasn't in the the `TVM_INITIALIZING` state.
SBI_ERR_FAILED	The operation failed for unknown reasons.

9.13. Function: TEE Host Add TVM Zero Pages (FID #11)

```
struct sbiret sbi_tee_host_add_tvm_zero_pages(unsigned long tvm_guest_id,
                                              unsigned long base_page_address,
                                              unsigned long tsm_page_type,
                                              unsigned long num_pages,
                                              unsigned long tvm_base_page_address);
```

Maps num_pages zero-filled pages of confidential memory starting at `base_page_address` into the TVM's physical address space starting at `tvm_base_page_address`. The `tvm_base_page_address` must lie within a region of confidential memory created with `sbi_tee_host_add_tvm_memory_region()`. The `tsm_page_type` parameter must be a legal value for the `tsm_page_type` enum. Zero pages for non-present TVM-specified GPA ranges may be added only post TVM finalization, and are typically demand faulted on TVM access.

The possible error codes returned in `sbiret.error` are shown below.

Table 14. TEE Host Add TVM Zero Pages Errors

Error code	Description
SBI_SUCCESS	The operation completed successfully.

Error code	Description
SBI_ERR_INVALID_ADDRESS	`base_page_address` or `tvm_base_page_address` were invalid.
SBI_ERR_INVALID_PARAM	`tvm_guest_id`, `tsm_page_type`, or `num_pages` were invalid.
SBI_ERR_FAILED	The operation failed for unknown reasons.

9.14. Function: TEE Host Add TVM Shared Pages (FID #12)

```
struct sbiret sbi_tee_host_add_tvm_shared_pages(unsigned long tvm_guest_id,
                                                unsigned long base_page_address,
                                                unsigned long tsm_page_type,
                                                unsigned long num_pages,
                                                unsigned long tvm_base_page_address);
```

Maps num_pages of non-confidential memory starting at `base_page_address` into the TVM's physical address space starting at `tvm_base_page_address`. The `tvm_base_page_address` must lie within a region of non-confidential memory previously defined by the TVM via the guest interface to the TSM. The `tsm_page_type` parameter must be a legal value for the `tsm_page_type` enum.

Shared pages can be added only after the TVM begins execution, and calls the TSM to define the location of shared-memory regions. They are typically demand faulted on TVM access.

The possible error codes returned in `sbiret.error` are shown below.

Table 15. TEE TEE Host Add TVM Shared Pages

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_ADDRESS	`base_page_address` or `tvm_base_page_address` were invalid.
SBI_ERR_INVALID_PARAM	`tvm_guest_id`, `tsm_page_type`, or `num_pages` were invalid.
SBI_ERR_FAILED	The operation failed for unknown reasons.

9.15. Function: TEE Host Get TVM VCPU Num Register Sets (FID #13)

```
struct sbiret sbi_tee_host_get_tvm_vcpu_num_register_sets(unsigned long tvm_guest_id);
```

Returns the number of register sets in the VCPU shared-memory state area for vCPUs of `guest_id` in sbiret.value on success. The host can use this to the number of enumerate individual register sets in the vCPU shared-memory state area (also enumerated by the `vcpu_register_set_id` enum). The offsets for the state can vary across TSM versions, and they can be determined by calling `sbi_tee_host_get_tvm_vcpu_register_set()`.

Note that the VCPU layout is likely to be common across all TVMs, in which case the host can enumerate it once. The interface is intended to provide future extensibility to accommodate heterogeneous TVMs that may choose to "opt-in" or "opt-out" of specific platform extensions.

The possible error codes returned in `sbiret.error` are shown below.

Table 16. TEE Host Get TVM VCPU Num Register Sets

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_PARAM	`tvm_guest_id` was invalid.
SBI_ERR_FAILED	The operation failed for unknown reasons.

9.16. Function: TEE Host Get TVM VCPU Register Set (FID #14)

```
struct sbiret sbi_tee_host_get_tvm_vcpu_register_set(unsigned long tvm_guest_id,
                                                      unsigned long
vcpu_register_set_id);
```

The host can use this this interface to discover the shared-memory offset of the VCPU state correspomding to the enum values in `vcpu_register_set_id` for `tvm_guest_id`. The `vcpu_register_set_id` parameter must be a legal value for the `vcpu_register_set_id` enum.

Returns a 32-bit value with the same layout as the `tvm_vcpu_register_set_location` structure in sbiret.value on success.

Note that the VCPU layout is likely to be common across all TVMs, in which case the host can enumerate it once. The interface is intended to provide future extensibility to accommodate heterogeneous TVMs that may choose to "opt-in" or "opt-out" of specific platform extensions.

The possible error codes returned in `sbiret.error` are shown below.

Table 17. TEE Host Get TVM VCPU Register Set

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_PARAM	`tvm_guest_id` or `vcpu_register_set_id` was invalid.
SBI_ERR_FAILED	The operation failed for unknown reasons.

9.17. Function: TEE Host Create TVM VCPU (FID #15)

```
struct sbiret sbi_tee_host_create_tvm_vcpu(unsigned long tvm_guest_id,
                                           unsigned long tvm_vcpu_id,
                                           unsigned long tvm_state_page_addr,
                                           unsigned long tvm_vcpu_shared_page_addr);
```

Adds a VCPU with ID `vcpu_id` to the TVM specified by `tvm_guest_id`. `tvm_state_page_addr` must be page-aligned and point to a confidential memory region used to hold the TVM's vCPU state, and must be `tsm_info::tvm_state_pages` pages in length.`tvm_vcpu_shared_page_addr` must be page-aligned and point to a sufficient number of non-confidential pages to hold a structure with the maximum offset enumerated by `sbi_tee_host_get_tvm_vcpu_register_set`. These pages are "pinned" in the non-confidential state (i.e. cannot be converted to confidential) until the TVM is destroyed.This call must not be made after calling `sbi_tee_host_finalize_tvm()`. The host must configure a boot VCPU by adding a `tvm_vcpu_id` with a value that specified for `tvm_boot_vcpuid` in the `tvm_create_params` structure that was used with sbi_tee_tvm_create().

The possible error codes returned in `sbiret.error` are shown below.

Table 18. TEE Host Create TVM VCPU Errors

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_PARAM	`tvm_guest_id` or `tvm_vcpu_id` were invalid, or the TVM wasn't in `TVM_INITIALIZING` state.
SBI_ERR_FAILED	The operation failed for unknown reasons.

9.18. Function: TEE Host Run TVM VCPU (FID #16)

```
struct sbiret sbi_tee_host_run_tvm_vcpu(unsigned long tvm_guest_id,
                                        unsigned long tvm_vcpu_id);
```

Runs the VCPU specified by `tvm_vcpu_id` in the TVM specified by `tvm_guest_id`. The `tvm_guest_id` must be in a "runnable" state (requires a prior call to `sbi_tee_host_finalize_tvm()`). The function does not return unless the TVM exits with a trap that cannot be handled by the TSM.

Returns 0 on success in sbiret.value if the TVM exited with a resumable VCPU interrupt or exception, and non-zero otherwise. In the latter case, attempts to call `sbi_tee_host_run_tvm_vcpu()` with the same `tvm_vcpu_id` will fail.

The possible error codes returned in `sbiret.error` are shown below.

Table 19. TEE Host Run TVM VCPU Errors

Error code	Description
SBI_ERR_SUCCESS	The TVM exited, and sbiret.value contains 0 if the

Error code	Description
interrupt or exception is resumable. The host can	examine `scause` to determine details.
SBI_ERR_INVALID_PARAM	`tvm_guest_id` or `tvm_vcpu_id` were invalid, or the TVM wasn't in `TVM_RUNNABLE` state.
SBI_ERR_FAILED	The operation failed for unknown reasons.

The TSM updates the `scause` field in the `tvm_vcpu_supervisor_csrs` region in the shared-memory for the VCPU that was previously configured by the host. The host should use the `scause` field to determine whether the exit was caused by an interrupt or exception, and then use the additional information to in the shared-memory region to determine further course of action (if sbiret.value is 0).

The TSM sets the most significant bit in `scause` to indicate that that the exit was caused by an interrupt, and if this bit is clear, the implication is that the the exit was caused by an exception. The remaining bits specific information about the interrupt or exception, and the specific reason can be determined using the enumeration detailed below.

```
enum tvm_interrupt_exit {
    /* Refer to the privileged spec for details. */
    USER_SOFT = 0,
    SUPERVISOR_SOFT = 1,
    VIRTUAL_SUPERVISOR_SOFT = 2,
    MACHINE_SOFT = 3,
    USER_TIMER = 4,
    SUPERVISOR_TIMER = 5,
    VIRTUAL_SUPERVISOR_TIMER = 6,
    MACHINE_TIMER = 7,
    USER_EXTERNAL = 8,
    SUPERVISOR_EXTERNAL = 9,
    VIRTUAL_SUPERVISOR_EXTERNAL = 10,
    MACHINE_EXTERNAL = 11,
    SUPERVISOR_GUEST_EXTERNAl = 12,
};
```

```
enum Exception {
    /* Refer to the privileged spec for details. */
    INSTRUCTION_MISALIGNED = 0,
    INSTRUCTION_FAULT = 1,
    ILLEGAL_INSTRUCTION = 2,
    BREAKPOINT = 3,
    LOAD_MISALIGNED = 4,
    LOAD_FAULT = 5,
    STORE_MISALIGNED = 6,
    STORE_FAULT = 7,
    USER_ENVCALL = 8,
    SUPERVISOR_ENVCALL = 9,
    /*
     * The TVM made an ECALL request directed at the host.
     * The host should examine GPRs A0-A7 in the `tvm_vcpu_supervisor_gprs`
     * area of the VCPU shared-memory region to process the ECALL.
    */
    VIRTUAL_SUPERVISOR_ENV_CALL = 10,
    /* Refer to the privileged spec for details. */
    MACHINE_ENVCALL = 11,
    INSTRUCTION_PAGE_FAULT = 12,
    LOAD_PAGE_FAULT = 13,
    STORE_PAGE_FAULT = 15,
    GUEST_INSTRUCTION_PAGE_FAULT = 20,
    /*
     * The TVM encountered a load fault in a confidential, MMIO, or shared-memory
     * region. The host should determine the fault address by retrieving the
     * `htval` from `tvm_vcpu_hypervisor_csrs` and `stval` from
`tvm_vcpu_supervisor_csrs`
     * and combining them as follows: "(htval << 2) | (stval & 0x3)". The fault
address
     * can then be used to determine the type of memory region, and making the
appropriate
     * call (example: sbi_tee_host_add_tvm_zero_pages() to add a demand-zero
confidential
     * page if applicable), and then calling sbi_tee_host_run_tvm_vcpu to resume
execution at
     * the following instruction.
    */
    GUEST_LOAD_PAGE_FAULT = 21,
    /*
     * The TVM executed an instruction that caused an exit. The host should decode the
     * instruction by examining `stval` from `tvm_vcpu_supervisor_csrs`, and determine
     * the further course of action, and calling then calling
sbi_tee_host_run_tvm_vcpu
     * if appropriate to resume execution at the following instruction.
    */
    VIRTUAL_INSTRUCTION = 22,
    /*
     * The TVM encountered a store fault in a confidential, MMIO, or shared-memory
     * region. The host should determine the fault address by retrieving the
     * `htval` from `tvm_vcpu_hypervisor_csrs` and `stval` from
`tvm_vcpu_supervisor_csrs`
     * and combining them as follows: "(htval << 2) | (stval & 0x3)". The fault
address
     * can then be used to determine the type of memory region, and making the
appropriate
     * call (example: sbi_tee_host_add_tvm_zero_pages() to add a demand-zero
confidential
     * page if applicable), and then calling sbi_tee_host_run_tvm_vcpu to resume
execution at
     * the following instruction.
     */
    GUEST_STORE_PAGE_FAULT = 23,
};
```

Chapter 10. TEE Interrupt Extension (EID #0x54414949)

The TEE Interrupt extension supplements the TEE Host extension with hardware-assisted interrupt virtualization using the RISC-V Advanced Interrupt Architecture (AIA) on platforms which support it.

10.1. Function: TEE Interrupt Init TVM AIA (FID #0)

```
struct sbiret sbi_tee_interrupt_init_tvm_aia(unsigned long tvm_guest_id,
                                             unsigned long tvm_aia_params_addr,
                                             unsigned long tvm_aia_params_len);
```

Configures AIA virtualization for the TVM identified by `tvm_guest_id` based on the parameters in the `tvm_aia_params` structure at the non-confidential physical address at `tvm_aia_params_addr`. The `tvm_aia_params_len` is the byte-length of the `tvm_aia_params` structure.

This cannot be called after `sbi_tee_host_finalize_tvm()`.

The format and semantics of the `tvm_aia_params_addr` structure appears below.

```
struct tvm_aia_params {
    /*
     * The base address of the virtualized IMSIC in TVM physical address space.
     *
     * IMSIC addresses follow the below pattern:
     *
     * XLEN-1 >=24 12 0 | | | |
     *
     * |xxxxxx|Group Index|xxxxxxxxxxx|Hart Index|Guest Index| 0 |
     *
     * The base address is the address of the IMSIC with group ID, hart ID, and guest
ID of 0.
     */
    unsigned long imsic_base_addr;
    /* The number of group index bits in an IMSIC address. */
    uint32_t group_index_bits;
    /* The location of the group index in an IMSIC address. Must be >= 24. */
    uint32_t group_index_shift;
    /* The number of hart index bits in an IMSIC address. */
    uint32_t hart_index_bits;
    /* The number of guest index bits in an IMSIC address. Must be >=
log2(guests_per_hart + 1). */
    uint32_t guest_index_bits;
    /*
     * The number of guest interrupt files to be implemented per VCPU. Implementations
may reject
```

```
     * configurations with guests_per_hart > 0 if nested IMSIC virtualization is not
supported.
     */
    uint32_t guests_per_hart;
};
```

The possible error codes returned in `sbiret.error` are shown below.

Table 20. TEE Interrupt Init TVM AIA

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_ADDRESS	`tvm_aia_params_addr` was invalid.
SBI_ERR_INVALID_PARAM	`tvm_guest_id` or `tvm_aia_params_addr` were invalid, or the TVM wasn't in the `TVM_INITIALIZING` state.
SBI_ERR_FAILED	The operation failed for unknown reasons.

10.2. Function: TEE Interrupt Set TVM AIA CPU IMSIC Addr (FID #1)

```
struct sbiret sbi_tee_interrupt_set_tvm_aia_cpu_imsic_addr(unsigned long tvm_guest_id,
                                                           unsigned long tvm_vcpu_id,
                                                           unsigned long
tvm_vcpu_imsic_gpa);
```

Sets the guest physical address of the specified VCPU's virtualized IMSIC to `tvm_vcpu_imsic_gpa`. The `tvm_vcpu_imsic_gpa` must be valid for the AIA configuration that was set by `sbi_tee_interrupt_init_tvm_aia()`. No two VCPUs may share the same `tvm_vcpu_imsic_gpa`.

This can be called only after `sbi_tee_interrupt_init_tvm_aia()` and before `sbi_tee_host_finalize_tvm()`. All VCPUs in an AIA-enabled TVM must have their IMSIC configuration set prior to calling `sbi_tee_host_finalize_tvm()`.

The possible error codes returned in `sbiret.error` are shown below.

Table 21. TEE Interrupt Set TVM AIA CPU IMSIC Addr

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_ADDRESS	`tvm_vcpu_imsic_gpa` was invalid.
SBI_ERR_INVALID_PARAM	`tvm_guest_id` or `tvm_vcpu_id` were invalid, or the TVM wasn't in the `TVM_INITIALIZING` state.
SBI_ERR_FAILED	The operation failed for unknown reasons.

10.3. Function: TEE Interrupt Convert AIA IMSIC (FID #2)

```
struct sbiret sbi_tee_interrupt_convert_aia_imsic(unsigned long imsic_page_addr);
```

Starts the process of converting the non-confidential guest interrupt file at `imsic_page_addr` for use with a TVM. This must be followed by calls to `sbi_tee_host_global_fence()` and `sbi_tee_host_local_fence()` before the interrupt file can be assigned to a TVM.

The possible error codes returned in `sbiret.error` are shown below.

Table 22. TEE Interrupt Convert AIA IMSIC

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_ADDRESS	`imsic_page_addr` was invalid.
SBI_ERR_FAILED	The operation failed for unknown reasons.

10.4. Function: TEE Interrupt Reclaim TVM AIA IMSIC (FID #3)

```
struct sbiret sbi_tee_interrupt_reclaim_tvm_aia_imsic(unsigned long imsic_page_addr);
```

Reclaims the confidential TVM interrupt file at `imsic_page_addr`. The interrupt file must not currently be assigned to a TVM.

The possible error codes returned in `sbiret.error` are shown below.

Table 23. TEE Interrupt Reclaim TVM AIA IMSIC

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_ADDRESS	`imsic_page_addr` was invalid.
SBI_ERR_INVALID_PARAM	The memory is still assigned to a TVM.
SBI_ERR_FAILED	The operation failed for unknown reasons.

Chapter 11. TEE Guest Extension (EID 0x54454547)

The TEE Guest extension supplements the TEE Host extension, and TVMs to communicate with TSM. A typical usecase for this extension is to relay information to the host. TEE-Guest calls cause a trap to the TSM which may exit to the host with scause set to ECALL, a6 set to FID, a0-a5 set to ECALL args.

11.1. Function: TEE Guest Add Memory Region (FID #0)

```
struct sbiret sbi_tee_guest_add_memory_region(unsigned long tvm_memory_region_type,
                                              unsigned long tvm_gpa_addr,
                                              unsigned long region_len);
```

Marks the range of TVM physical address space starting at `tvm_gpa_addr` as reserved for the mapping of non-confidential memory. The type of memory is specified by `tvm_memory_region_type` and the length is specified by by `region_len`. `tvm_memory_region_type` must be of type `SHARED_MEMORY_REGION` or `EMULATED_MMIO_REGION`.

Both `tvm_gpa_addr` and `region_len` must be 4kB-aligned, and the region must not overlap with a previously defined region. This call will result in an exit to the host on success.

Table 24. TEE TEE Guest Add Memory Region

Error code	Description
SBI_SUCCESS	The operation completed successfully.
This implies an exit to the host, and a subsequent resume of execution.	SBI_ERR_INVALID_ADDRESS
`tvm_gpa_addr` was invalid.	SBI_ERR_INVALID_PARAM
`tvm_memory_region_type` or `region_len` were invalid	SBI_ERR_FAILED

11.2. Function: TEE Guest Share Memory Region (FID #1)

```
struct sbiret sbi_tee_guest_share_memory_region(unsigned long tvm_gpa_addr,
                                                unsigned long region_len);
```

Initiates the conversion of TVM physical address space starting at `tvm_gpa_addr` from confidential to non-confidential/shared memory. The requested range must lie within an existing region of confidential address space, and may or may not be populated. If the region of address space is populated, the TSM invalidates the pages and marks the region as pending conversion to shared. The host must complete a TVM TLB invalidation sequence, initiated by tee_host_tvm_initiate_fence(), in order to complete the conversion. The calling TVM vCPU is considered blocked until the conversion is completed; attempts to run it with tee_host_tvm_run() will fail. Any guest page faults taken by other TVM vCPUs in this region prior to completion of the conversion are considered fatal. The host may not insert any pages in the region prior to completion of the conversion. Upon completion, the host may reclaim the confidential pages that were previously mapped in the region using tee_host_tsm_reclaim_pages() and may insert shared pages into the region using tee_host_tvm_add_shared_pages(). If the range of address space is completely unpopulated, the region is immediately converted to shared and the host may insert shared pages.

Both `tvm_gpa_addr` and `region_len` must be 4kB-aligned.

Table 25. TEE Guest Share Memory Region

Error code	Description
SBI_SUCCESS	The operation completed successfully.
This implies an exit to the host, and a subsequent resume of execution.	SBI_ERR_INVALID_ADDRESS
`tvm_gpa_addr` was invalid.	SBI_ERR_INVALID_PARAM
`region_len` was invalid, or the entire range does not map to a `CONFIDENTIAL_REGION`	SBI_ERR_FAILED

11.3. Function: TEE Guest Unshare Memory Region (FID #2)

```
struct sbiret sbi_tee_guest_unshare_memory_region(unsigned long tvm_gpa_addr,
                                                  unsigned long region_len);
```

Initiates the conversion of TVM physical address space starting at `tvm_gpa_addr` from shared to confidential. The requested range must lie within an existing region of non-confidential address space, and may or may not be populated. If the region of address space is populated, the TSM invalidates the pages and marks the region as pending conversion to confidential. The host must complete a TVM TLB invalidation sequence, initiated by tee_host_tvm_initiate_fence(), in order to complete the conversion. The calling TVM vCPU is considered blocked until the conversion is completed; attempts to run it with tee_host_tvm_run() will fail. Any guest page faults taken by other TVM vCPUs in this region prior to completion of the conversion are considered fatal. The host may not insert any pages in the region prior to completion of the conversion. Upon completion, the host may (if required) convert host memory pages using tee_host_convert_pages() and may insert un-assigned confidential pages into the region using sbi_tee_host_add_tvm_zero_pages(). If the range of address space is unpopulated, the host may insert zero pages on faults during TVM access.

Both `tvm_gpa_addr` and `region_len` must be 4kB-aligned.

Table 26. TEE Guest Share Memory Region

Error code	**Description**
SBI_SUCCESS	The operation completed successfully.
This implies an exit to the host, and a subsequent resume of execution.	SBI_ERR_INVALID_ADDRESS
`tvm_gpa_addr` was invalid.	SBI_ERR_INVALID_PARAM
`region_len` was invalid, or the entire range doesn’t span a `SHARED_MEMORY_REGION`	SBI_ERR_FAILED

Chapter 12. Summary Listing of TEEI

Summary of TSM load and initialization operations	
sbi_tee_host_get_tsm_info	Used by the OS/VMM to discover if a TSM is loaded and initialized else returns an error. If a TSM is loaded and initialized, this operation is used to enumerate TSM information such as: TEE-capable memory regions, Size of static memory to allocate per TVM, Size of memory to allocate per TVM Virtual Hart and so on.
sbi_tee_host_tsm_load	Used by the OS/VMM to load a TSM binary image into TSM-memory region. Pages used for TSM will be declared as part of this function to load the TSM. Loading and updates to the TSM should be done via the TSM-driver. This interface is TBD.
sbi_tee_host_tsm_init_global	Perform a global state initialization of the TSM after a load or update. This operation and the following should succeed before the TSM is considered ready to service other TVM operations. This interface is TBD.
sbi_tee_host_tsm_init_local	Perform a local (per-hart) initialization of TSM after the global init has been performed.This operation and the above should succeed before the TSM is considered ready to service other TVM operations.
sbi_tee_host_tsm_update	Update TSM binary and/or configuration. Ideally this operation should be performed without shutting down the TVMs, however all TVMs have to be paused before an update can be issued. The TSM update process description is TBD.
sbi_tee_host_tsm_shutdown	Shuts down the TSM. All TVMs must be shutdown and all TVM memory must be reclaimed before this operation can succeed.
Summary of TVM global operations	
sbi_tee_host_create_tvm	TVM creation (static) process where a set of TEE pages are assigned for a TVM to hold a TVM’s global state. This routine also configures the global configuration that applies to the TVM and affects all TVM hart settings. For example, features enabled for this TVM, perfmon enabled, debug enabled etc.

Summary of TSM load and initialization operations	
sbi_tee_host_destroy_tvm	TVM shutdown verifies VMM has stopped all virtual hart execution for the TVM. The TVM virtual hart may not be entered after this point. The VMM may start reclaiming TVM memory after this point.
Summary of TVM Global memory management operations	
sbi_tee_host_convert_pages	Begins the process of converting memory to be used as confidential memory. The region consists of one or more contiguous 4KB memory naturally aligned regions.
sbi_tee_host_global_fence	This operation initiates TLB version tracking of pages in the region being converted to confidential. The TSM enforces that the VMM performs invalidation of all harts (via IPIs and subsequent sbi_tee_host_local_fence) to remove any cached mappings to the memory regions blocked for conversion via the sbi_tee_host_convert_pages.
sbi_tee_host_local_fence	This operation completes the TLB version tracking of pages in the region being connverted to confidential. The TSM tracks that all available physical harts have executed this operation before it considers the TLB version updated. The last local fence completes the conversion of a memory region from non-confidential to confidential for a set of TVM pages.
sbi_tee_host_reclaim_pages	VMM may unassign memory for TVMs by destroying them. All confidential-unassigned memory may be reclaimed back as non-confidential using this interface.
Summary of TVM memory management operations	
sbi_tee_guest_unshare_memory_region	Convert a memory region from non-confidential to confidential for a set of TVM pages.This operation initiates TSM tracking of these pages and also changes the encryption properties of these pages. These pages can then be selected by the VMM to allocate for TVM control structure pages, second stage page table pages, and TVM pages.

Summary of TSM load and initialization operations	
sbi_tee_host_add_tvm_page_table_pages	Add one or more page mappings to the second stage translation structure for a TVM. The pages to be used for the second stage page table structures must have been converted (and tracked) by the TSM as TEE pages; otherwise this operation will not succeed.
sbi_tee_host_add_tvm_measured_pages	Add a page for an existing mapping for a TVM page - this add_pre must be performed before finalization of the TVM measurement via teecall_tvm_msmt_commit. For this operation, the VMM must provide the page contents that get copied into confidential memory pages for the TVM (and get tracked, encrypted etc). The contents of these pages are also measured via the teecall_tvm_msmt_extend, including the GPA at which the page is mapped. After the TVM msmt is finalized via teecall_tvm_msmt_commit, no more pre-add pages are allowed by the TSM for that TVM.
sbi_tee_host_add_tvm_zero_pages	Add a zero page for an existing mapping for a TVM page (post initialization). This operation adds a zero page into a mapping and keeps the mapping as pending (i.e. access from the TVM will fault until the TVM accepts that GPA
sbi_tee_guest_global_fence	Blocks a set of page mappings for an existing mapping for a TVM page. This operation prevents new TLB mappings from being created for a particular TVM page mapping. Note that stale TLB mappings may exist and those are invalidated by the TSM. The TSM enforces that mappings are blocked and flushed by the VMM before allowing any page relocation and/or page fragmentation operations.
sbi_tee_guest_local_fence	Issue a TVM TLB invalidation (for the relevant harts) after a set of changes to the TVM mappings for confidential pages. The TSM enforces a hfence.gvma for the affected TVM vmid/asid to enforce stale tlb mappings are flushed. For implementations using memory tracking, this operation should also invalidate additional caching structures for page meta-data.

Summary of TSM load and initialization operations	
sbi_tee_guest_page_relocate	Relocate a page for an existing mapping for a TVM page. This operation allows the VMM to reassign a new SPA for an existing TVM page mapping. The page mapping must be blocked and fenced before the page mapping can be relocated. This interface specification is TBD.
sbi_tee_guest_page_promote	Promote a set of small page mappings (existing mappings) for a set of TVM pages to a large page mapping. The affected mappings must be blocked before the promote operation can succeed. The VMM may reclaim the freed second stage page table page if the operation succeeds.
sbi_tee_guest_page_demote	Demote a large page mapping for an existing mapping to a set of TVM pages and corresponding small page mappings. The affected mapping must be blocked before the operation can succeed. The VMM must provide a free TEE-capable page to the TSM to use as a new second stage page table in the fragmented mapping.
Summary of TVM virtual hart management operations	
sbi_tee_host_create_tvm_vcpu	This operation allows the VMM to assign TEE pages for a virtual hart context structure (VHCS) for a specific TVM. This routine also initializes the hart-specific fields of this structure.Note that a virtual hart context structure may consist of more than 1 4KB page. The number of pages are enumerated via the tsm_info call.
Summary of TVM measurement operations	

Summary of TSM load and initialization operations	
sbi_tee_host_add_tvm_measured_pages	This operation is used to extend the static measurement for a TVM for added page contents.The operation performs a SHA384 hash extend to the measurement register managed by the TSM on a 256 byte block of the page. The page must be added to a valid GPA mapping via the add_pre_init operation. The GPA of the page mapped is part of the measurement operation.The measurement process is a state machine that must be faithfully reproduced by the VMM otherwise the attestation evidence verification by the relying party will fail and the TVM will not be considered trustworthy.
sbi_tee_host_finalize_tvm	This operation enables a VMM to finalize the measurement of a TVM (static). The TSM enforces that a TVM virtual harts cannot be entered unless the TVM measurement is committed via this operation.
TVM runtime operations	
sbi_tee_run_vcpu	Enter or resume a TVM virtual hart (on any physical hart). A resume operation is performed via a flag passed to this operation. This operation activates a virtual-hart on a physical hart, and may be performed only on a TVM virtual hart structure that is assigned to the TVM and one that is not already active. The TSM verifies if the operation is performed in the right state for that virtual hart.
TSM runtime operations	
teecall_ **tsm** _teeret	This operation is used by a TSM to return control to the OS/VMM via the TSM-driver TEERET flow.This operation may be used by the TSM in various scenarios - in response to a sbi_tee_guest_* operation for requests to the VMM, or due to an S-mode interrupt that the TSM must report to the OS/VMM. It is also used to communicate faults in the second stage page table for a TVM etc.

12.1. TEEI - TG-ABI runtime interface

sbi_tee_guest_drtm_extend	This intrinsic is used by a TVM component to act as a dynamic root of trust of measurement (DRTM) for the TVM to extend runtime measurements. These measurements are managed by the TSM in the TVM global structure (To be specified TBD). These measurements are used in the TcbEvidenceInfo when the TVM attestation certificate is generated via teecall_tg_get_evidence. This interface specification is TBD.
sbi_tee_guest_get_evidence	This intrinsic is used by a TVM to get attestation evidence to report to a (remote) relying party.It is supported by the TSM to provide HW-key-signed measurements of the TVM and the TSM. The attestation key used to sign the evidence is provisioned into the TVM by the TSM. The TSM certificate is provisioned by the FW TCB (TSM-driver and HW RoT). This interface specification is TBD.
sbi_tee_guest_share_memory_region	This intrinsic is used by the TVM to request the conversion of the specified GPA to non-confidential (from confidential).The GPA must be mapped to the TVM in a present state, and must be scrubbed by the TVM before it is yielded. The TSM enforces that the page is not-present in the second stage page table and not tracked as a TEE page. The VMM owns the process of reclaiming the page.
sbi_tee_guest_host_invoke	This intrinsic is supported by the TSM to provide the TVM the ability to request host services e.g. para-virt IO.The TVM indicates to the TSM during this operation which x/v/f registers should be passed to the OS/VMM without clearing. The specification of this interface is TBD.
sbi_tee_guest_enable_debug	This intrinsic is supported by the TSM to enable the TVM to request for debugging to be enabled for the TVM (TSM invokes TSM-driver to enable debugging if the TVM was created with debug opt-in; TSM enforces state save and restore of debug state for TVM hart). The specification of this interface is TBD.

sbi_tee_guest_enable_perfmon	This intrinsic is supported by the TSM to enable the TVM to request performance monitoring (where the TSM enforces state save and restore of the performance monitoring inhibit and trigger controls). The specification of this interface is TBD.

Chapter 13. Appendix A: THCS and VHCS

The TSM Hart Control Structure (THCS) is divided into two sections - the Hart Supervisor State Area (HSSA) and the TSM Supervisor State Area (TSSA). This structure is specified as part of the TEEI as the recommended minimum that the TSM-driver should support to isolate TSM state.

VMM-managed hart f/v registers* are expected to be saved/restored by the VMM before a TEECALL, and restored (similar to v/f register management performed by the VMM for ordinary guest VMs). The TSM-driver saves OS/VMM S/HS-mode CSRs and x registers on ECALLs into the HSSA on a TEECALL (per the RISC-V SBI [5] convention). The TSM-driver initializesTSM S/HS-mode CSRs from the TSSA on entry into the TSM (via TEECALL). Per-Hart TSM f/v registers* state is managed (saved/restored) by the TSM in reserved memory for the TSM (hence not shown below).

HSSA (***TBD - specify initial values of TSSA state*** **)**	
CSR	**Description**
sstatus	Saved/Restored by TSM-driver
stvec	Saved/Restored by TSM-driver
sip	Saved/Restored by TSM-driver
sie	Saved/Restored by TSM-driver
scounteren	Saved/Restored by TSM-driver
sscratch	Saved/Restored by TSM-driver
satp	Saved/Restored by TSM-driver
senvcfg	Saved/Restored by TSM-driver
scontext	Saved/Restored by TSM-driver
mepc	Saved/Restored by TSM-driver. Value of the mepc saved during TEECALL in order to restore during TEERET flow
TSSA	
CSR	Description
sstatus	Initialized/Restored by TSM-driver
stvec	Initialized/Restored by TSM-driver
sip	Initialized/Restored by TSM-driver
sie	Initialized/Restored by TSM-driver
scounteren	Initialized/Restored by TSM-driver
sscratch	Initialized/Restored by TSM-driver
satp	Initialized/Restored by TSM-driver
senvcfg	Initialized/Restored by TSM-driver
scontext	Initialized/Restored by TSM-driver

mepc	Initialized/Saved/Restored by TSM-driver to specify TSM entrypoint during TEECALL/TEERESUME
interrupted	Set/Cleared by TSM-driver. Boolean flag

TVM per-hart state x/v/f is saved/restored by the TSM (prior to SRET and post delegated-trap into the TSM from the TVM) and uses the dynamic memory assigned to the TEE VM. The control structure for the TVM virtual hart is shown as the VHCS below. These guest control CSRs are restored by the TSM when a TVM virtual hart is being entered and is configured on the required state of that TVM.

Virtual Hart Control Structure (VHCS)

CSR	Description
hstatus	Initialized by TSM
hedeleg	Initialized by TSM to enforce events that are to always be handled by the TSM (default all)
hideleg	Initialized by TSM to enforce events that are to always be handled by the TSM (default all)
hvip	Initialized (cleared) by the TSM
hip	Initialized (cleared) by the TSM
hie	Initialized by TSM to enforce events that are to always be handled by the TSM (default all)
hgeip	Initialized (cleared) by the TSM
hgeie	Initialized (cleared) by the TSM
henvcfg	Initialized by TSM
hvenvcfg	Initialized by TSM
hcounteren	Initialized by TSM per TVM configuration
htimedelta	Initialized by TSM per TVM configuration
htimedeltah	Initialized by TSM per TVM configuration
hgatp	TVM enforces page remap protection via this second stage translation. Hart register is programmed by TSM to activate at TVM entry via SRET
The values htval and htinst are cleared by TSM on TEECALL and masked (to clear page offset) by the TSM on a TEERET when reporting a guest page fault_The vs* and x/v/f registers are not listed here but are maintained by the TSM per virtual hart for TVMs._	

Chapter 14. Appendix B: Interrupt Handling

The following table describes the interrupt handling delegation for an interruptible and non-preemptable TSM.

Interrupt	Exception Code	Description	If AP-TEE and mode/Handled by;
1	0	Reserved	*/M(TSM-driver)
1	1	Supervisor software interrupt	VU(TVM)/VS (TVM); VU(TVM)/VS(TVM); U(TSM)/HS(TSM); HS(TSM)/ M(TSM-driver)
1	2	Reserved	*/M(TSM-driver)
1	3	Machine software interrupt	"
1	4	Reserved	"
1	5	Supervisor timer interrupt	VU(TVM)/VS (TVM); VU (TVM)/VS(TVM); U(TSM)/HS(TSM); HS(TSM)/M(TSM-driver)
1	6	Reserved	*/M(TSM-driver)
1	7	Machine timer interrupt	"
1	8	Reserved	"
1	9	Supervisor external interrupt	VU(TVM)/VS (TVM); VU(TVM)/VS(TVM); U(TSM)/HS(TSM); HS(TSM)/M(TSM-driver)
1	10	Reserved	*/M(TSM-driver)
1	11	Machine external interrupt	"
1	12–15	Reserved	"
1	≥16	Designated for platform use	"

0	0	Instruction address misaligned	VU(TVM)/ VS(TVM); VU(TVM)/VS(TVM); U(TSM)/HS(TSM); HS(TSM)/M(TSM-driver)
0	1	Instruction access fault	"
0	2	Illegal instruction	"
0	3	Breakpoint	"
0	4	Load address misaligned	"
0	5	Load access fault	"
0	6	Store/AMO address misaligned	"
0	7	Store/AMO access fault	"
0	8	Environment call from U-mode	VU(TVM)/VS (TVM); U(TSM)/HS(TSM)
0	9	Environment call from S-mode	VS(TVM)/HS (TSM); HS(TSM)/M(TSM-driver)
0	10	Reserved	*/M(TSM-driver)
0	11	Environment call from M-mode	*/M(TSM-driver)
0	12	Instruction page fault	VU (TVM) / VS (TVM); VS (TVM) / HS (TSM); U (TSM) / HS (TSM); HS (TSM) / M (TSM-driver)
0	13	Load page fault	"
0	14	Reserved	*/M(TSM-driver)
0	15	Store/AMO page fault	VU(TVM)/VS(TVM); VS(TVM)/HS(TSM); U(TSM)/HS(TSM); HS(TSM)/M(TSM-driver)
0	16–23	Reserved	*/M(TSM-driver)
0	24–31	Designated for custom use	Per custom use
0	32–47	Reserved	*/M(TSM-driver)
0	48–63	Designated for custom use	Per custom use
0	≥64	Reserved	*/M(TSM-driver)