Confidential VM Extension (CoVE) for Confidential Computing on RISC-V platforms

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

Version 0.1, 1/2023: 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 CoVE 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, Jiewen Yao, Kailun Qin, Manuel Offenberg, Nick Kossifidis, Rajnesh Kanwal, Ravi Sahita (Editor <ravi@rivosinc.com>), Samuel Ortiz, Vedvyas Shanbhogue, Yann Loisel

Chapter 1. Introduction

This document describes Confidential VM Extension (CoVE) interface proposal to provide a scalable Trusted Execution Environment(TEE) in virtualized workloads on RISC-V-based platforms. This CoVE 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.

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.
Confidential VM Extension (CoVE)	A set of non-ISA RISC-V extensions that enables confidential computing on RISC-V platforms.

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 CoVE, 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 confidential 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 confidential 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 CoVE platform.
RoT	Isolated HW/SW subsystem with an immutable ROM firmware and isolated compute and memory elements that form 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 CoVE 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 [R2] 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

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 Computing workloads that rely on HW-Attested Trusted Execution Environments, and strive to minimize the TCB footprint.

This specification describes the CoVE architecture which describes 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 Section 4.1. 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.

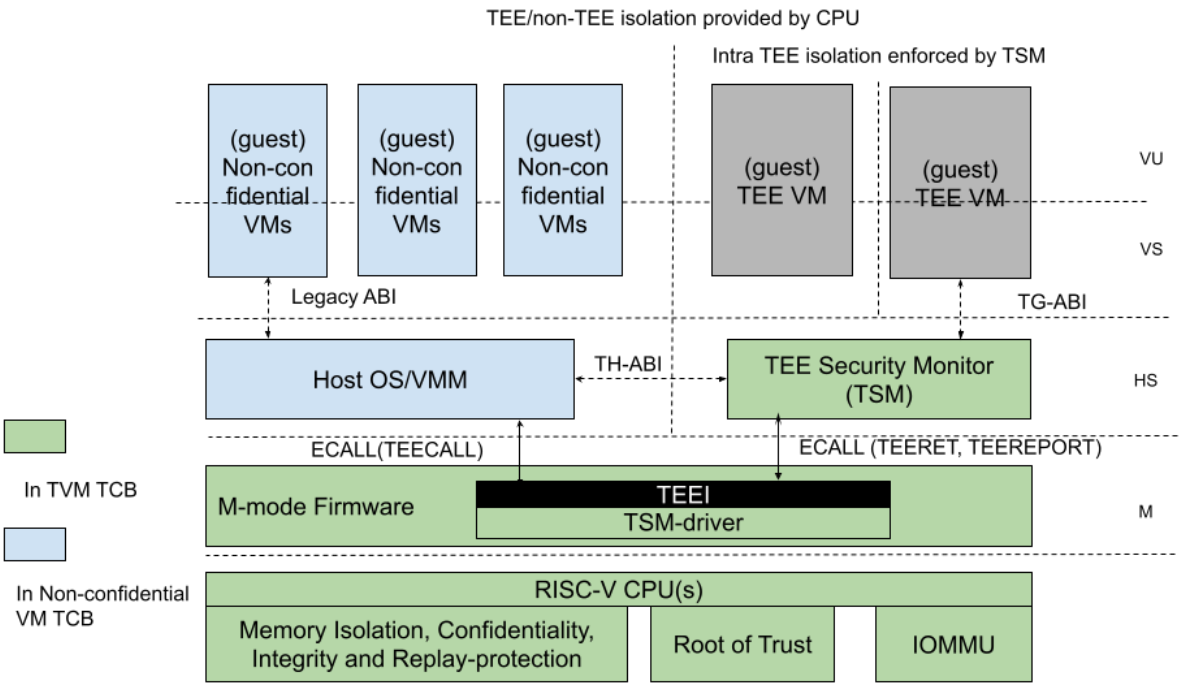

Figure 1: TEE TCB for VM workloads

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 TEE and non-TEE workloads. The TSM has a minimal possible HW-attested footprint. The TCB (which includes the TSM and HW) enforces strict confidentiality and integrity security properties for workloads. It also isolates confidential workloads from all other platform components (non-confidential and confidential). The responsibility of the TSM is to enforce the security objectives accorded to TEE workloads. The VMM continues to manage the security for non-confidential workloads, and the 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 transitioned to their confidential analogue via the TSM. Once the conversion process is complete, confidential memory may 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 accesses by the TVM. The non-confidential TVM-defined regions include those for shared-pages and MMIO.

The TSM implements a set of TEE "flows" that are accessed via a **Trusted Execution Environment Interface (TEEI)** ABI hosted by a TEE 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 CoVE 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 CoVE 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 set of host ABIs known as **COVH** 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 set of guest ABIs known as **COVG** that enablse 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 **Confidential mode bit** 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 Confidential mode is set. This per-hart Confidential 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 details of this mode bit is out of scope for this specification.

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 COVG.

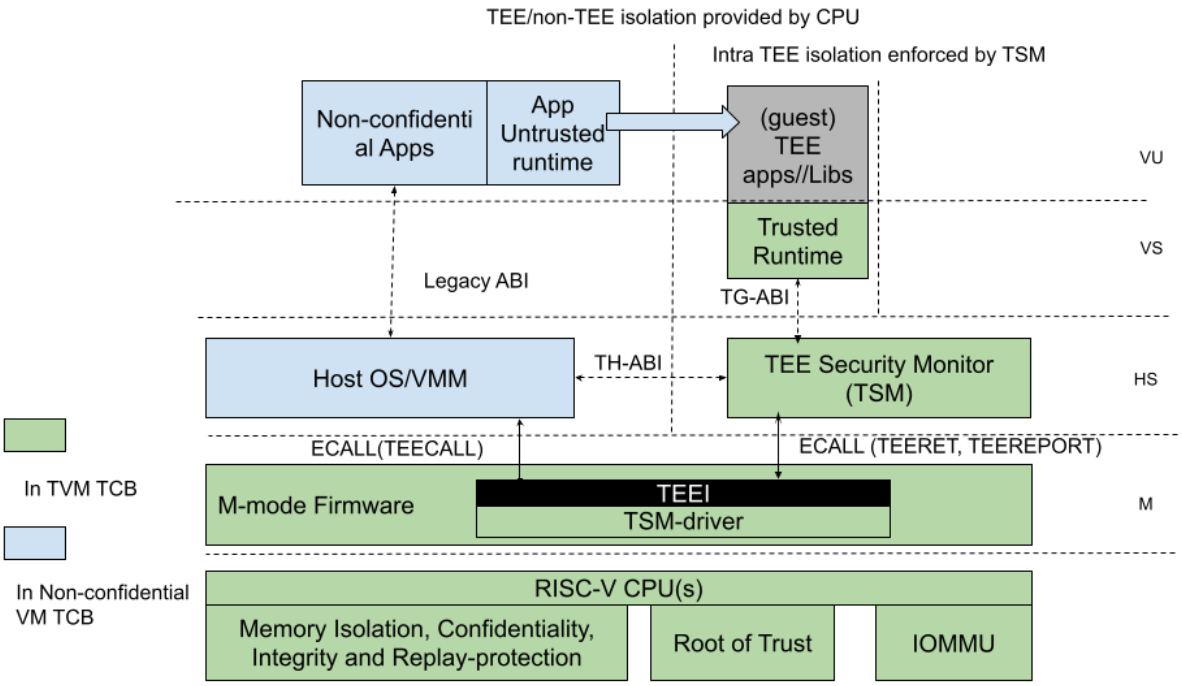

Figure 2: TEE TCB for application workloads (hosted via a TVM)

The detailed architecture is described in the Section Chapter 5. Note that the architecture described above may have various implementations, however the goal of this specification is to propose a reference architecture and ratify a normative COVE 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 G-stage 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.

4.4. TVM Security Requirements to address threat model

Category	Security Criteria	CoVE Requirement	Example methods to meet requirement	Description/Example	RVI HC/SIG/TG owner
Memory Footprint	Stolen/reserved memory	Implementation-specific	Minimize reserved memory	Recording meta data of secure memory	AP-TEE TG to specify
Memory Assignment	Ability to make memory confidential or non-confidential	Required	MMU, MPU, PMA/MTT extension	Confidential memory should be dynamically allocated/unallocated as required	"AP-TEE to specify priv. architecture"
TEE CPU State Protection	State Isolation	Required	Confidential qualifier and privilege levels M S HS U	Prevent untrusted code from arbitrarily accessing/modifying TEE CPU state	AP-TEE TG to specify
Memory Confidentiality	Memory isolation (read)	Required	cryptography and/or MMU, MPU, PMA/MTT extension	Prevent untrusted components from reading TEE memory	AP-TEE TG specify
Memory Confidentiality	Cipher text read prevention	Required	cryptography and/or MMU, MPU, PMA/MTT extension	Prevent untrusted code from accessing encrypted TEE memory	AP-TEE TG specify

Category	Security Criteria	CoVE Requirement	Example methods to meet requirement	Description/Example	RVI HC/SIG/TG owner
Memory Confidentiality	Per TEE encryption	Implementation-specific	cryptography and/or MMU, MPU, PMA/MTT extension	Each VM has one or more unique keys	AP-TEE TG to recommend
Memory Confidentiality	Memory encryption strength	Implementation-specific	cryptography	Encryption algorithm and key strength	AP-TEE TG to recommend
Memory Confidentiality	Number of encryption keys	Implementation-specific	cryptography	Number of TEE keys supported	AP-TEE TG to recommend
Memory Integrity	Memory integrity against SW attacks	Required	MMU, MPU, PMA/MTT extension	Prevent SW attacks such as remapping aliasing replay corruption etc.	AP-TEE TG to specify
Memory Integrity	Memory integrity against HW attacks	Implementation-specific	cryptography and/or MMU, MPU, PMA/MTT extension	Prevent HW attacks DRAM-bus attacks and physical attacks that replace TEE memory with tampered / old data	AP-TEE TG to recommend
Memory Integrity	Memory isolation (Write exec)	Required	cryptography and/or MMU, MPU, PMA/MTT extension	Prevent TEE from executing from normal memory; Enforce integrity of TEE data on writes	AP-TEE TG specify
Memory Integrity	Rowhammer attack prevention	Implementation-specific	cryptography and/or memory-specific extension	Prevent untrusted code from flipping bits of TEE memory	AP-TEE TG to recommend

Category	Security Criteria	CoVE Requirement	Example methods to meet requirement	Description/Example	RVI HC/SIG/TG owner
Shared Memory	TEE controls data shared with untrusted code	Required	cryptography and/or MMU, MPU, PMA/MTT extension	Prevent malicious code from exfiltrating information without TEE consent/opt-in	AP-TEE TG to specify
Shared Memory	TEE controls data shared with another TEE	Implementation-specific	cryptography and/or MMU, MPU, PMA/MTT extension	Ability to securely share memory with another TEE	AP-TEE TG to recommend
I/O Protection	DMA protection from untrusted devices	Required	DMA access-control e.g. IOPMP, IOMMU	Prevent untrusted peripheral devices from accessing TEE memory	AP-TEE TG to specify
I/O Protection	Trusted I/O from trusted devices	Implementation-specific	Device attestation, Link protection, IOMMU	Admission control to bind devices to TEEs	AP-TEE, IOMMU TG to specify
Secure IRQ	Trusted Interrupts	Required	Secure interrupt files, MMU, MPU, PMA/MTT extension	Prevent IRQ injections that violate priority or masking	AIA AP-TEE to specify
Secure Timetamp	Trusted timestamps	Required	Confindential mode qualifier for CSR accesses	Ensure TEE have consistent timestamp view	AP-TEE TG specify

Category	Security Criteria	CoVE Requirement	Example methods to meet requirement	Description/Example	RVI HC/SIG/TG owner
Debug & Profile	Trusted performance monitoring unit	Required	Confidential mode qualifier for perf. mon. counter controls	Ensure TEEs get correct PMU info; prevent data leakage due to PMU information (fingerprint attacks)	AP-TEE, Performance Mon. SIG to specify
Debug & Profile	Debug support	Required	Confidential mode qualifier for Sdtrig controls	Support debug trigger registers for TVM	AP-TEE, Debug TG to specify
Debug & Profile	Authenticated debug (Production device)	Required	Authorize debug via TEE RoT	Ensure hardware debug prob (e.g., JTAG SWD) is disabled in production	AP-TEE, Debug TG specify
Availability	TVM DoS Protection	Required	VMM retains ability to interrupt TVM	Prevent TVM from refusing to exit	AP-TEE TG specify
Availability	VMM DoS Protection	Implementation-specific	Not in scope for CoVE	Prevent untrusted code from refusing to run TEE	Not applicable
Side Channel	Protected address mapping (controlled side channel)	Required	Confidential mode qualifier, cryptography, MMU/MPU, MTT	Similar to memory remapping attacks	uSG SIG, AP-TEE to specify
Side Channel	Micro-architectural side channels (branch prediction	Required	uArch state flushing, entropy defenses	Prevent attacks such as meltdown/spectre (it is difficult to defend agains such attacks in advance)	uSC SIG, AP-TEE specify

Category	Security Criteria	CoVE Requirement	Example methods to meet requirement	Description/Example	RVI HC/SIG/TG owner
Side Channel	Control channels, single-step/zero-step attacks	Required	uArch state flushing, entropy defenses	Prevent interrupt/exception injection (combined with cache side channel to leak sensitive data)	uSC SIG , AP-TEE specify
Side Channel	Architectural cache side channel	Implementation-specific	uArch state flushing, entropy defenses	Prevent shared resource contention, e.g. attacks prime probe	uSG SIG, AP-TEE to specify
Side Channel	Architectural timing side channel	Implementation-specific	data independent operations, uArch state flushing	Leveraging data dependency timing channels	uSG SIG, AP-TEE to specify
Secure and measured boot	Establishes root of trust in support of attestation	Required	RoT unique trust chain for TEE TCB	Enforcing initial firmware authorization and versioning	Security Model TG
Attestation	Remote attestation	Required	HW RoT based PKI (trust assertions) via Internet	Prevent fake hardware and software TCB; Prevent malicious hardware debugging in production.	AP-TEE TG to specify
Attestation	Mutual attestation	Implementation-specific	S/U mode	Attestation to another TEE on the same platform	AP-TEE TG specify
Attestation	Remote mutual attestation	Required	Internet	Attestation to a relying party on a different platform	AP-TEE TG specify

Category	Security Criteria	CoVE Requirement	Example methods to meet requirement	Description/Example	RVI HC/SIG/TG owner
Attestation	Local attestation	Implementation-specific	Sealing	Verification of attestation by TCB	AP-TEE TG specify
Attestation	TCB versioning (and updates)	Required	Mutable firmware where TVM has to opt-in if TCB updates are allowed or not - HW TCB then enforces lower TCB elements are updatable (with apropos controls like SVN) only after that opt-in has been honored.	Allow TCB updates - Prevent TCB rollback	AP-TEE TG specify
Attestation	TCB composition -Single root of trust for msmt. for confidential compute	Required	How do we express the issue with TCB elements being composed of various elements? e.g. M-mode, ROT firmware. Perhaps we can only express the requirement of a single root of trust for measurement and reporting	Malicious components introduced in the TCB	AP-TEE TG specify

Category	Security Criteria	CoVE Requirement	Example methods to meet requirement	Description/Example	RVI HC/SIG/TG owner
Attestation	Dynamic vs Static Attestation interop (between platform TCB and TEE TCB) - enforce isolation of the entire trust chain	Required	TEE TCB should not be affected by other TCB reporting chains. TEE TCB is separately reportable and recoverable.	Malicious host tampers with TEE TCB or reporting chain	AP-TEE TG specify
Attestation	TCB transparency (and auditability)	Implementation-specific	Mutable firmware	TCB elements reviewable	AP-TEE TG recommend
Attestation	Sealing	Implementation-specific	HW Rot sealing keys per TVM	Binding of secrets to TEEs	AP-TEE TG specify
Operational Features	TVM Migration	Implementation-specific	Secure migration of TEEs	Malicious host tampers with TVM assets during migration	Hypervisor SIG, AP-TEE TG specify
Operational Features	TVM Nesting	Implementation-specific	Nested TEE Workloads	Malicious host tampers with nested VMM policies	Hypervisor SIG, AP-TEE TG specify
Operational Features	Memory introspection/ Scanners	Implementation-specific	Interoperability with security features for TVM workload	Unauthorised security TVM	Security HC to specify
Operational Features	QOS interoperability	Implementation-specific	Interoperability with QoS features for TVM workload	Malicious host uses QoS capabilities as a side-channel	QOS SIG to specify

Category	Security Criteria	CoVE Requirement	Example methods to meet requirement	Description/Example	RVI HC/SIG/TG owner
Operational Features	RAS interoperability	Implementation-specific	Interoperability with RAS features for TVM workload	Malicious host uses RAS capabilities as a side-channel or to cause integrity violations	RAS SIG to specify

Chapter 5. Reference Architecture Details

We describe the capabilities of the platform to support memory isolation requirements for confidentiality of workloads in TVMs. We then describe the properties of the TSM, its instantiation, isolation and operational model for the TVM life cycle. The description in this section refers to the reference architecture in Figure 1.

5.1. CoVE Memory Isolation

Memory isolation for TVMs are orchestrated by the TSM in two phases, the conversion of memory to Confidential memory and the assignment of confidential memory (and the enforcement of properties on its use) to TVMs. Thus, CoVE requires new Physical Memory Attributes (PMAs): **Confidential** and **Non-Confidential** (These are dynamic/programmable memory attributes [priv ISA]). A TVM needs to access both types of memory:

- Confidential memory - has Confidential PMA - used for TVM code, data
- Non-Confidential memory - has Non-Confidential PMA - used for communication between TVM and untrusted host entities

The COVH ABIs implemented by the TSM provides interfaces to the VMM to convert / donate memory to Confidential [Convert] and vice-versa [Reclaim]. TVM memory is by default assigned from Confidential memory regions. TVM may be assigned shared memory regions. Both properties are enforced by the TSM. Figure 3 below shows the abstract model of isolation:

- Hart with Confidential mode qualifier =1 is allowed to access Confidential and Non-Confidential memory
- Hart with Confidential mode qualifier =0 is allowed to access only Non-confidential Memory

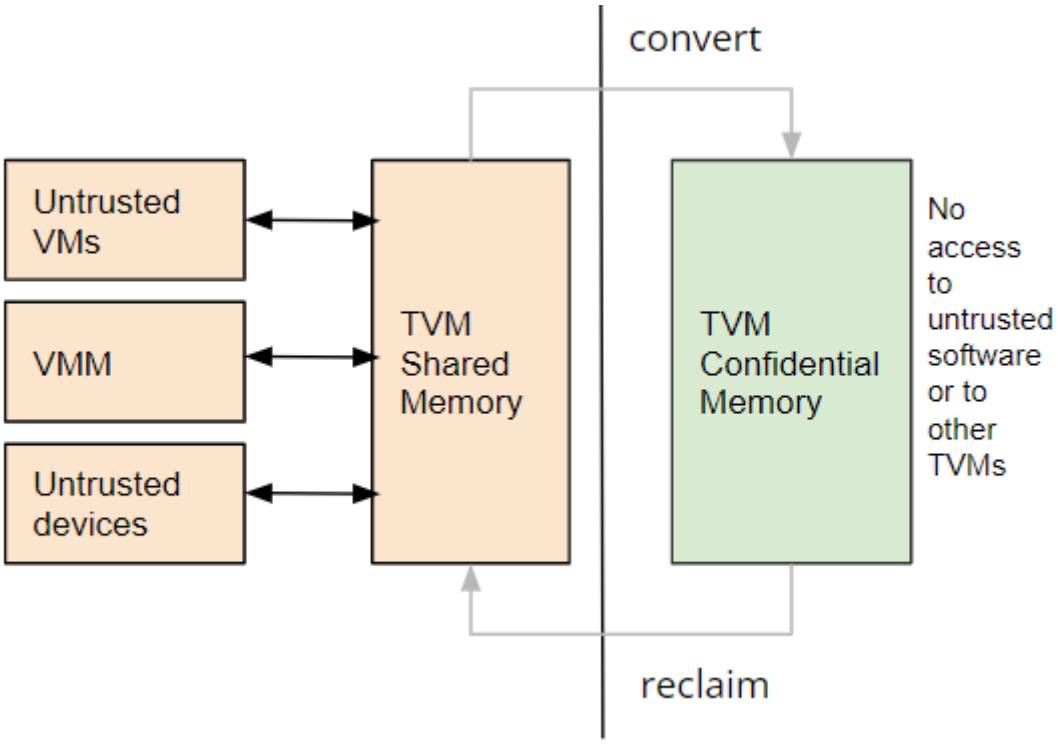

Figure 3: Confidential Memory conversion

Memory with Confidential PMA may be associated with a unique memory encryption key (and similar IO/fabric protection policies). Non-confidential memory is assigned by the VMM - the TSM and TSM-driver are expected to manage the Confidentiality PMA by programming a Memory Tracking Table (MTT). The desired security properties of memory tracking are discussed below - implementations are free to choose the format and structure of the memory tracking table/structures. The TSM manages finer-granular (page-based) allocation from Confidential memory regions (enforced by the memory tracking hardware) using the G stage page tables.

Four aspects of memory isolation are impacted due to this dynamic configurable confidential PMA:

5.1.1. Address Translation/Page Walk

The figure 4 below describes a reference model for memory tracking lookup where the physical address derived from the first and guest stage nested page walk is looked up to derive a confidential | non-confidential physical memory attribute. This lookup should be performed for the page sizes per the paging modes supported by the hart.

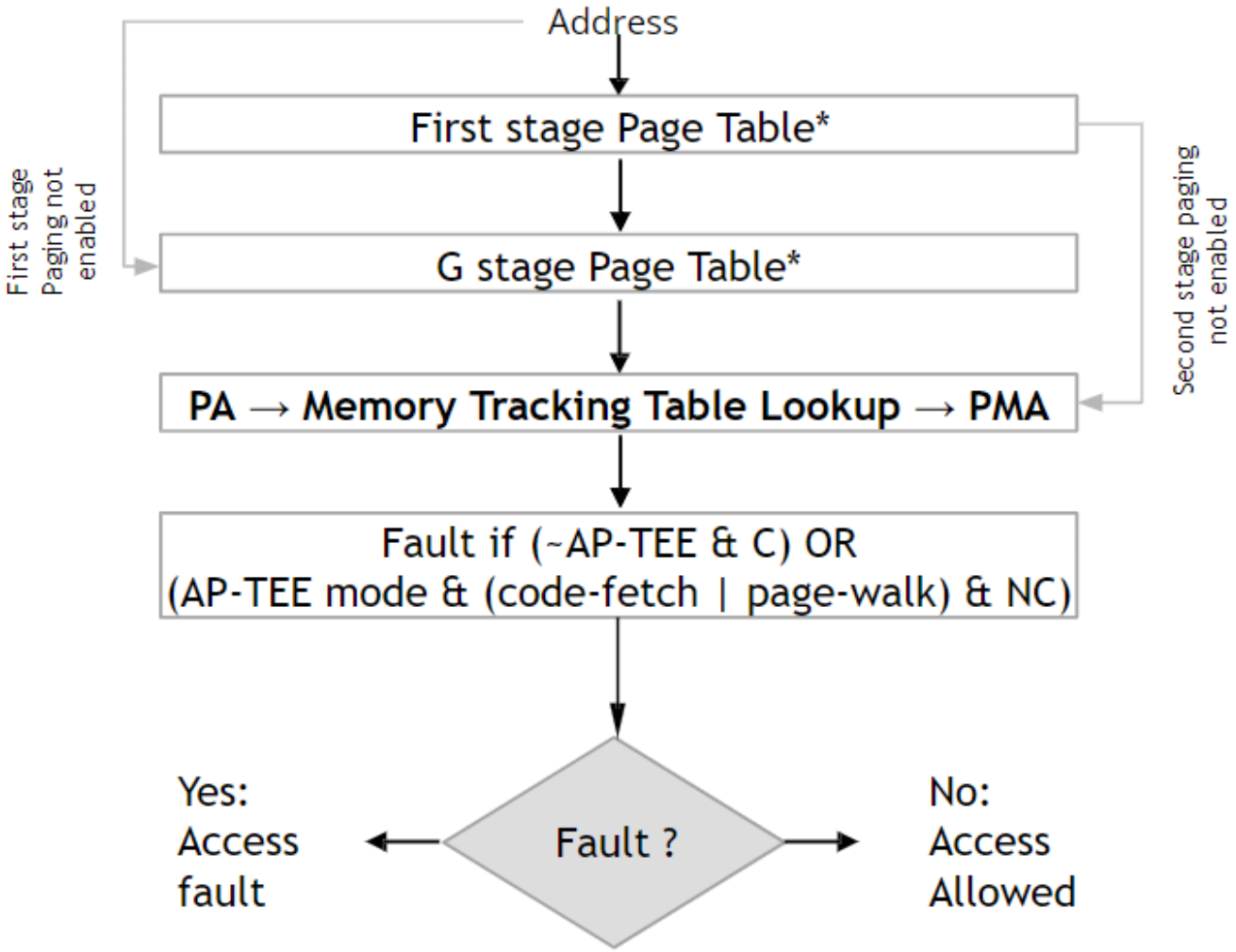

Figure 4: Memory Tracking for Confidential PMA

5.1.2. Management of Confidential | Non-Confidential dynamic Physical Memory Attributes

The SW TCB manages the assignment of the confidential PMA to memory regions, while the HW

TCB enforces the confidential PMA for TVM accesses. The region sizes at which the memory tracking enforces confidential properties may be multiples of the architectural page sizes supported by the hart MMU. Similarly, the IOMMU should support a similar memory tracking lookup to enable direct device access into TVM memory regions. For the CoVE reference architecture this TCB hence consists of the HW (MMU, IOMMU, Memory Controller) and the SW/FW elements - TSM-driver and the TSM. The TSM is responsible for enforcing isolation of confidential memory pages among TVMs (via G-stage translation) - pages assigned to the TVM may belong to C | NC PMA to allow for IO access. The TSM may manage additional attributes on TVM-assigned pages such as: TVM-owner, Page-sub-type, TLB versioning information, Locking semaphore and additional metadata etc. This extended memory tracking information managed by the TSM is referred to as the EMTT.

5.1.3. Handling Implicit, Explicit Accesses

For TVM accesses for instruction fetch and page walks (virtual address translation), a Confidential PMA is required to enforce the following security properties:

- TEE Instruction fetch - security property: TVM/TSM cannot fetch code from untrusted shared memory
- TEE Paging structure walk - security property: TVM/TSM cannot locate page tables in untrusted shared memory (for TVM, G-stage walks enforced to be in confidential memory by the TSM)
- TEE data fetch - security property: TVM/TSM allowed to relax data accesses to non-confidential memory (via MTT) to allow for IO accesses.

5.1.4. Cached translations/TLB management

During confidential memory conversion or reclamation, the TCB must enforce that stale translations cached in harts are not accessible to the untrusted host or another TVM context. During confidential memory assignment to a TVM (or during conversion of confidential memory to shared), the TCB must enforce that stale translations may not be held to memory yielded by a TVM (and used by the host for another TVM or VM or the host). These properties are implemented by the TSM in conjunction with the HW TCB via the proposed TEEI.

5.2. TSM initialization

The CoVE 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 Chapter 13) 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 the execution state of the TSM (for interrupted routines) is performed by the TSM. The operating modes of the TSM are described in Section 5.3. Saving and restoring 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.

5.3. 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 Chapter 13).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 G-stage page table structures).

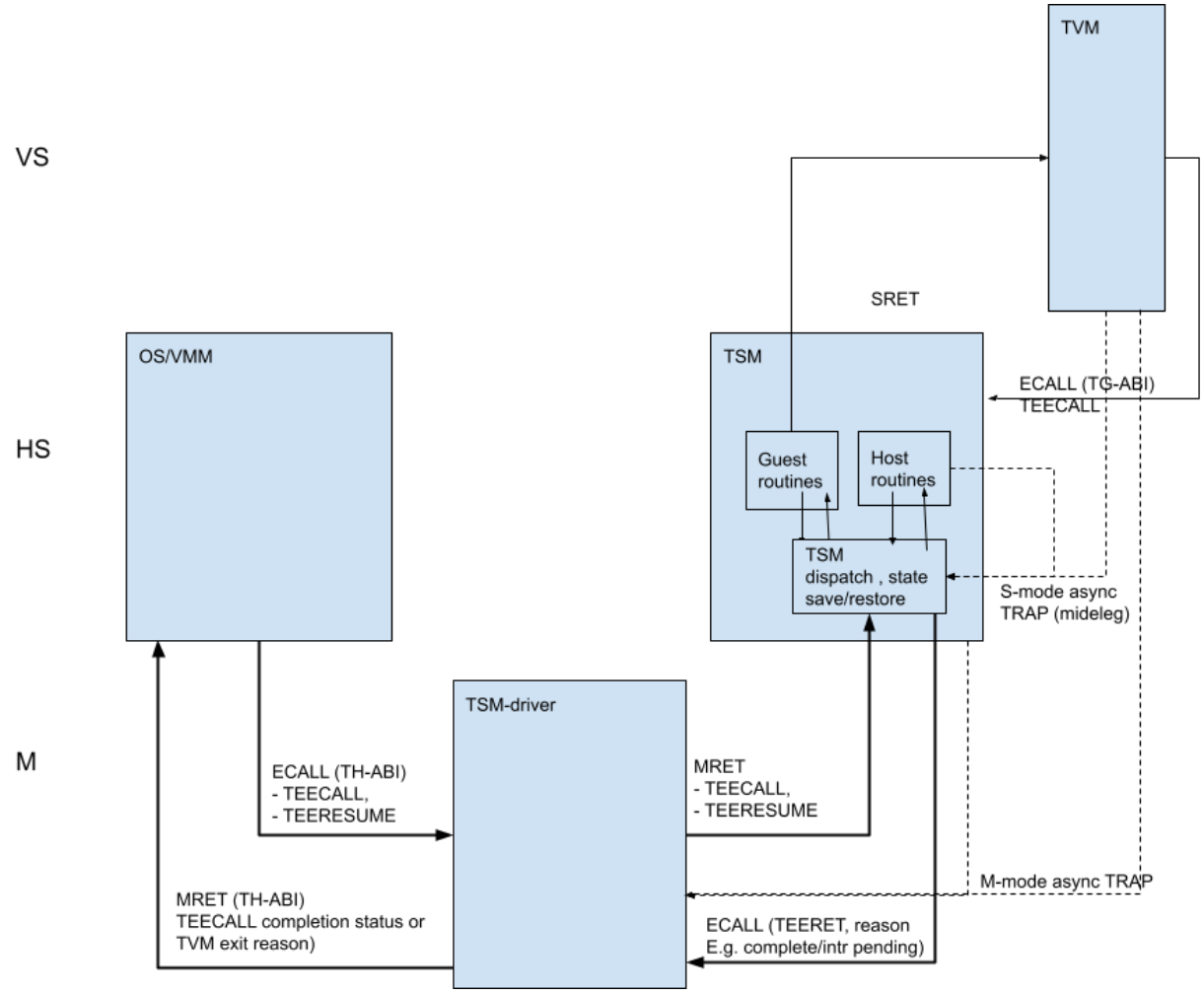

Figure 3: TSM operation - Interruptible and non-reentrant TSM model shown.

A TSM entry triggered by an ECALL (with CoVE extension 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 Confidential 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 Chapter 13). 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 Chapter 13)
- 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 preemptable 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 preemptable 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 Confidential mode on hart (via implementation-specific M-mode CSR to block non-TEE mode accesses to TEE-assigned memory.)
- MRET to resume 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. The TSM is not expected to issue IPIs itself; the TSM must track if appropriate IPIs are issued by the host OS/VMM to track that the required security checks are performed on each physical hart (or virtual hart context) as required by specific TEEI flows.

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 preempt 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 Chapter 14 for a table of interrupt causes and handling requirements.

The TSM may not need to program stimecmp on its own, though it may verify that time is not going back for a TVM. If the TSM needs to start a timer, it should context switch the stimecmp CSR and replace it with its timeout value if it’s later than the timer it wants to start. The TSM may still want to be aware of the value programmed into stimecmp to guard against step attacks on TVMs.

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.

5.4. 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) **Confidential mode qualifier** to maintain the isolation during access and in internal caches, e.g. Hart TLB lookup may be extended with the Confidential 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 **Confidential 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 G-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 security requirements/invariants for enforcement of the memory access-control for memory assigned to the TVMs is described in [TVM Memory management].

5.5. 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 G-stage paging structures pages used for the G-stage mapping is also performed by the OS/VMM but the security properties of the G-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 Chapter 7 section of this document.

5.6. 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).

The TVM may opt-in to use performance monitoring either at initialization or post-init.

If the TVM has opted-in to performance monitoring, the TSM may invoke the SBI PMU extension (via TSM-driver) or Supervisor counter delegation extension to establish a TVM-specific performance monitoring controls (counters, event selectors). However, the TVM must use SBI PMU extension unless TSM supports full trap & emulate support for the hpmcounter related ISA extensions. The TSM will assign a virtual counter to the TVM for the events requested to be monitored by the TVM in either approach. The TSM needs to manage a mapping between the virtual and physical counters as well. It must not delegate the LCOFI interrupt (via hideleg[13]=1) for the TVM and use the interrupt filtering mechanism defined in the Advanced Interrupt Architecture (AIA) to inject the LCOFI interrupt when the physical counter corresponding to the virtual counter overflows. The physical counters naturally inhibit counting in S/HS and M. The TSM 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 must restore the saved hardware performance monitoring event triggers and counter enables. If the TSM uses the SBI PMU extension instead of Supervisor counter delegation, the TSM-driver needs to perform the save/restore on behalf of the TSM. :imagesdir: ./images

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 [R1] 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.

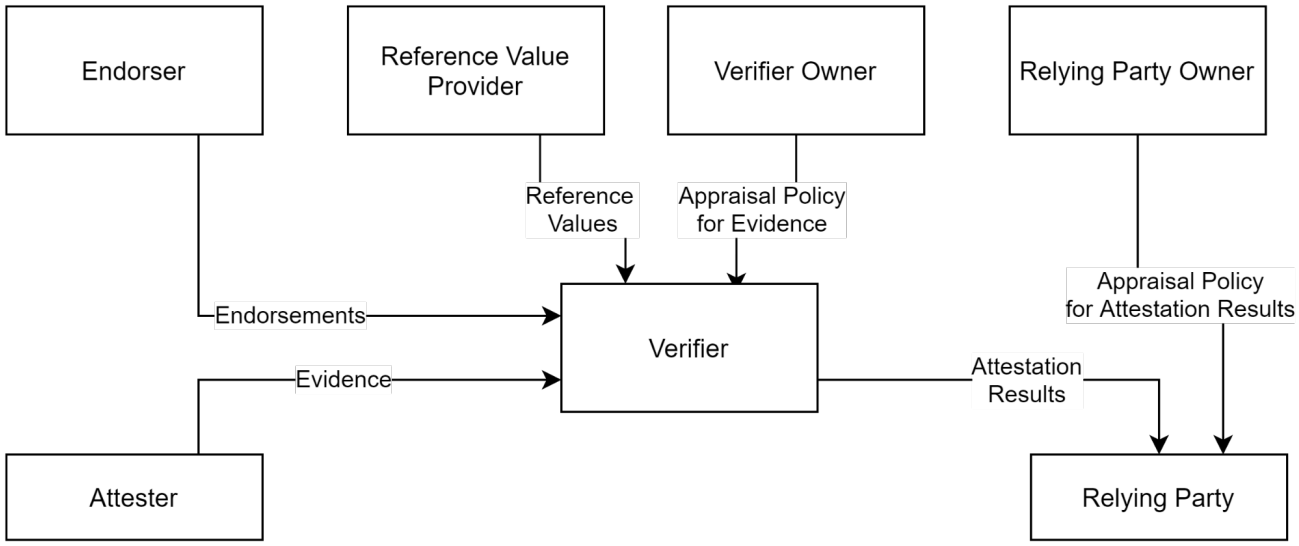

Figure 4: Remote Attestation Framework (IETF RATS)

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 (sbi_tee_guest_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:

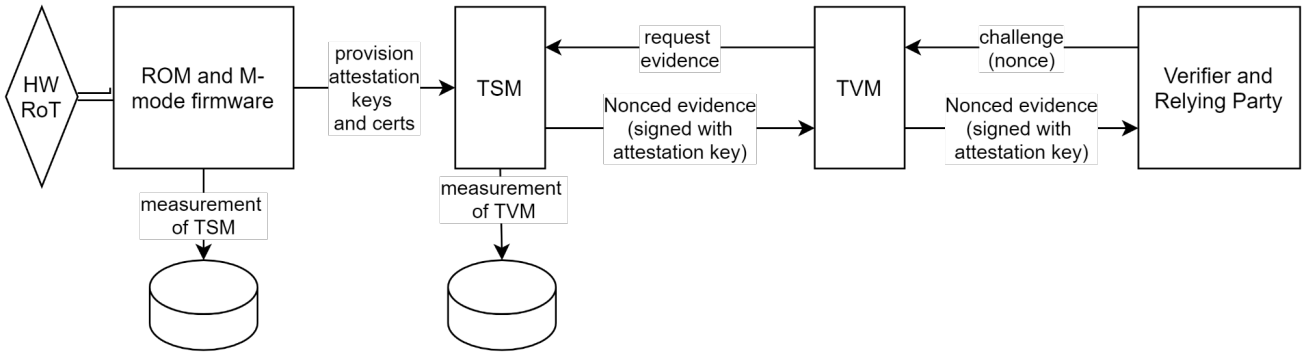

Figure 5: Layered Attestation architecture for TVMs

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 [R2]). 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	Bit String	A list of flags that 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 [R2] 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:

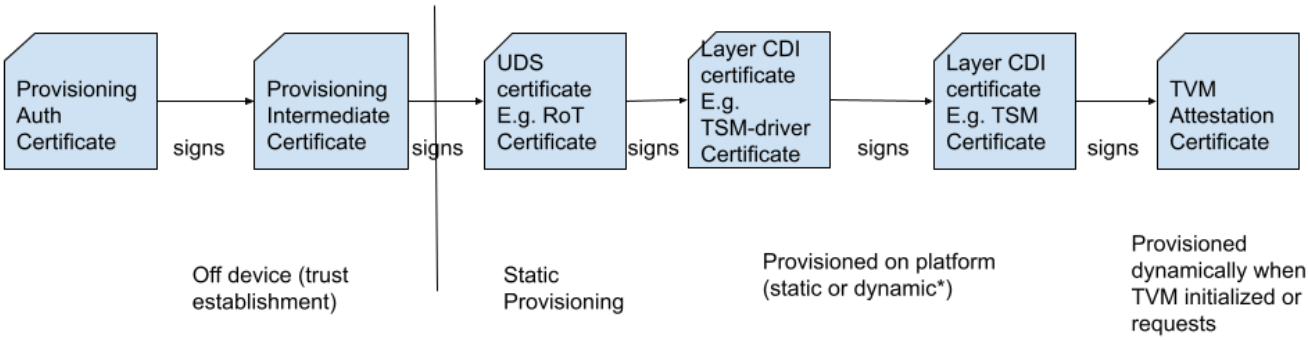

Figure 6: Attestation Certificate generation

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 an 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 [R2] 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 runtime measurements done by the TVM) at runtime via the sbi_tee_guest_get_evidence interface to generate additional TVM Attestation certificates.

sbi_tee_guest_get_evidence: 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 CoVE-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 `sbi_covh_create_tvm()` function. 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 `sbi_covh_destroy_tvm()` function.

A TVM context may be created and initialized by using the `sbi_covh_create_tvm()` 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 `sbi_covh_add_tvm_page_table_pages()` `sbi_covh_add_tvm_measured_pages()` and `sbi_covh_add_tvm_zero_pages()` - 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 two are 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 `sbi_covh_add_tvm_measured_pages()` which extends the static measurement hash of the TVM. The hash 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 `sbi_covh_add_tvm_measured_pages()` & `sbi_covh_add_tvm_zero_pages()`, 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. The pages may be lazily added to the TVM subsequent to the TVM execution using the `sbi_covh_add_tvm_zero_pages()`.

Lastly, the VMM can assign memory to the TVM to hold virtual hart state in

`sbi_covh_create_tvm_vcpu()` TEECALL. Before the VMM can start executing the TVM virtual harts, the VMM must finalize the static measurement of the TVM via `sbi_covh_finalize_tvm()`. The TSM prevents any TVM virtual harts from being entered until the TVM initialization is finalized.

7.2. TVM execution

The VMM uses `sbi_covh_run_tvm_vcpu()` 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 calling physical hart. 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 VS-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 threat model described in Chapter 4:

- TVM instruction fetches and page walks (both VS/second-stage and G/VS-stage) are implicitly enforced to be in confidential memory. This requires that the TVM supervisor code should not locate VS-stage page tables in non-confidential memory. The TSM enforces that G-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 5.4.

For virtual-IO operations, the TVM code must register virtual-IO memory regions for trap and emulation by the host using `sbi_covg_add_mmio_region()`. Any read/write by the TVM from/to this memory region will result in a guest page-fault into TSM and TSM will forward the fault to the host. TSM will also communicate additional information such as faulting instruction, faulting address and the GPR value (in case of store instruction) to the host. When direct device assignment is supported (which is expected to require IOMMU changes for CoVE), trusted devices may DMA directly into TVM confidential memory.

TVM memory may be lazily granted to the TVM by the host VMM, however confidential memory may be only lazily added via `sbi_covh_add_tvm_zero_pages()` after the TVM measurement has been finalized. The TVM manages its internal memory database to indicate which guest physical page frames are confidential for mapping into VS-stage mappings. There are at least two use scenarios for this ABI - first, late addition 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 via `sbi_covg_share_memory_region()`, and at a later point they are converted back to confidential, the VMM may add zero pages for those mappings.

During execution and typically during TVM initialization, the TVM code can extend the runtime

measurement registers by invoking the `sbi_covg_measurement_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 `sbi_covg_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 G-stage page-table mapped confidential TVM memory mapping to be invalidated (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 new mappings may be activated, the TSM tracks that the VMM has invoked `sbi_covg_local_fence()` and caused invalidation of the TLB on all virtual harts of the TVM. The VMM achieves this via inter-processor interrupts to all the vcpus for the TVM. The local fence is enforced by the TSM by executing HFENCE.GVMA for the TVM VMID. This sequence is described in more detail in Section 7.3.

7.3. TVM memory management

The RISC-V architecture supports page types of 4KB, 2MB, 1GB and 512GB. The untrusted OS/VMM may assign memory to the TVM at any architecture-supported page size. The TSM configures the memory tracking table (MTT) via the TSM-driver to track the assignment of memory pages to trusted execution contexts (i.e. TVMs).

Memory access-control is enforced at two levels:

- Isolation of memory assigned to TEEs - this includes memory assigned to the TSM as well as any TVMs - this tracking is configured by the firmware TCB (TSM-driver) via the Memory Tracking Table structure and is enforced by the CPU MMU. The MTT tracks the Confidential | Non-confidential state for a software-accessible physical address.
- Isolation of memory between TVMs - memory tracking is augmented by the TSM via the G-stage translation structures to maintain compatibility with OS/VMM memory management, and is also enforced by the CPU MMU. The correct operation of this access-control level is dependent on trusted enforcement of item 1 above.

7.3.1. Security requirements for TVM memory mappings

The following are the security requirements/invariants for enforcement of memory access-control for memory assigned to the TVMs. These rules are enforced by the TSM and the CPU MMU:

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. VS-stage address translation - A TVM page mapping must be translated only via VS-stage translation structures which are contained in pages assigned to the same TVM.
4. G-stage address translation:
    a. A TVM page guest physical address mapping must be translated only via the TSM-managed G-stage translation structures for that TVM.
    b. G-stage structures must not be shared between TVMs, and must not refer to any other TVMs pages.
    c. The OS/VMM has no access to TVM G-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 G-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.

7.3.2. Information tracked per physical page

The Extended Memory Tracking Table (EMTT) information managed by the TSM is used to track additional fields of metadata associated with physical addresses. The page size is implicit in the MTT and EMTT lookup - 4KB, 2MB, 1GB, 512GB. Actual page sizes supported are implementation-specified.

Memory Type	**Confidential or Non-confidential (enforced via MTT)**
Page-Type	Reserved - page that may not be assigned to any TEE entity If the Memory type is Confidential, the following page types may be used: * Unassigned - page not assigned to any TEE (TSM or TVM) * TVM - page assigned to a TVM (mapped via HGAT). * TSM - page used by the TSM (for MTT and other control structures)
Page Owner	If the Memory Type is Confidential and Page-Type is TVM, this value holds the identifier (e.g. PPN) for the TVM control page (4KB TEE- TSM-TVM page); else it is 0.

Page sub-type	Following types apply If Memory Type is Confidential and Page-Type is TVM: * HGATP - pages used for HGATP structures * Data - pages used for TVM content Following types apply If Memory Type is Confidential and Page-Type is TSM: * MTT - pages used for MTT structures * TVMC - pages used for TVM control structure(s) for global control * VHCS - pages used for TVM VHCS (virtual hart control structures)
Page TLB version	TLB version in which the page mapping was invalidated to allow for VMM memory management. If the page is Unassigned, the TLB version is per the global TLB mgmt. If the page is assigned to a TVM, it is versioned per the TVM-local TLB mgmt.
Additional meta-data	Locking state e.g.

7.3.3. Page walk and Translation caching considerations

Any caching of the address translation information when the memory tracking for confidential memory is enabled must cache whether the address translation is for a TEE context or not. A miss in the cached MTT information is expected to cause a lookup of the MTT structure using the PA and the resolved page size for TEE ownership evaluation - which results in the TEE ownership information that is cached.

The MTT lookups are performed using the physical address, and must be enforced for all modes of operation i.e., with paging disabled, one-level paging and guest-stage paging.

Any MTT cached information may be flushed as part of HFENCE.GVMA. The TSM and VMM may both issue this operation. TSM issues this fence when memory ownership is transferred between TEE and non-TEE ownership via sbi_covh_convert_pages.

7.3.4. Page conversion

Post measured boot, the system memory map must be available to the TSM on load (accessed as part of initialization of the TSM). This memory map structure may be placed in the memory that is accessible only to the HW and SW TCB. VMM chosen memory regions must be a strict subset of this set of memory regions. Memory regions used for the TSM are marked as reserved by the TSM-driver in this memory map - the TSM uses its memory space to host an Extended MTT (EMTT).

The operations used by the host for page conversion are:

- sbi_covh_convert_pages: 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_covh_local_fence()`) to remove any cached mappings to the memory regions invalidated for conversion via the `sbi_covh_convert_pages()`.
- sbi_covh_local_fence: This operation completes the TLB version tracking of pages in the region

being converted 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_covh_reclaim_pages: VMM may unassign memory for TVMs by destroying them. All confidential-unassigned memory may be reclaimed back as nonconfidential using this interface.

Conversion Operation: TSM uses the EMTT which maps each assignable (non-reserved) PA to page_owner, type, sub-type and other fields such as page_tlb_version. Page conversion involves the following steps by the TSM:

- Verify page(s) donated by the VMM is/are Non-Confidential page(s)
- Initiates a new TLB version tracking cycle via `sbi_covh_convert_pages()` - invalidates MTT entries (synchronized) for the requested page(s) and size as pages being converted to confidential (i.e. "in transition")
- TSM enforces a TLB versioning scheme (described below) and using that enforces that the VMM performs the invalidation of the hart TLBs (via IPIs) to remove any cached mappings - VMM performs a local fence operation on each hart via the `sbi_covh_local_fence()`.
- At the last fence operation, TSM verifies that TLB fence was completed for all harts for the batch of pages selected for conversion, and marks those mappings as usable as confidential memory.
- At this point non-TEE mode software cannot create new TLB entries to donated pages - since non-TEE mode accesses to MTT-tracked Confidential pages will fault (including implicit accesses)

7.3.5. Global and per-TVM TLB management

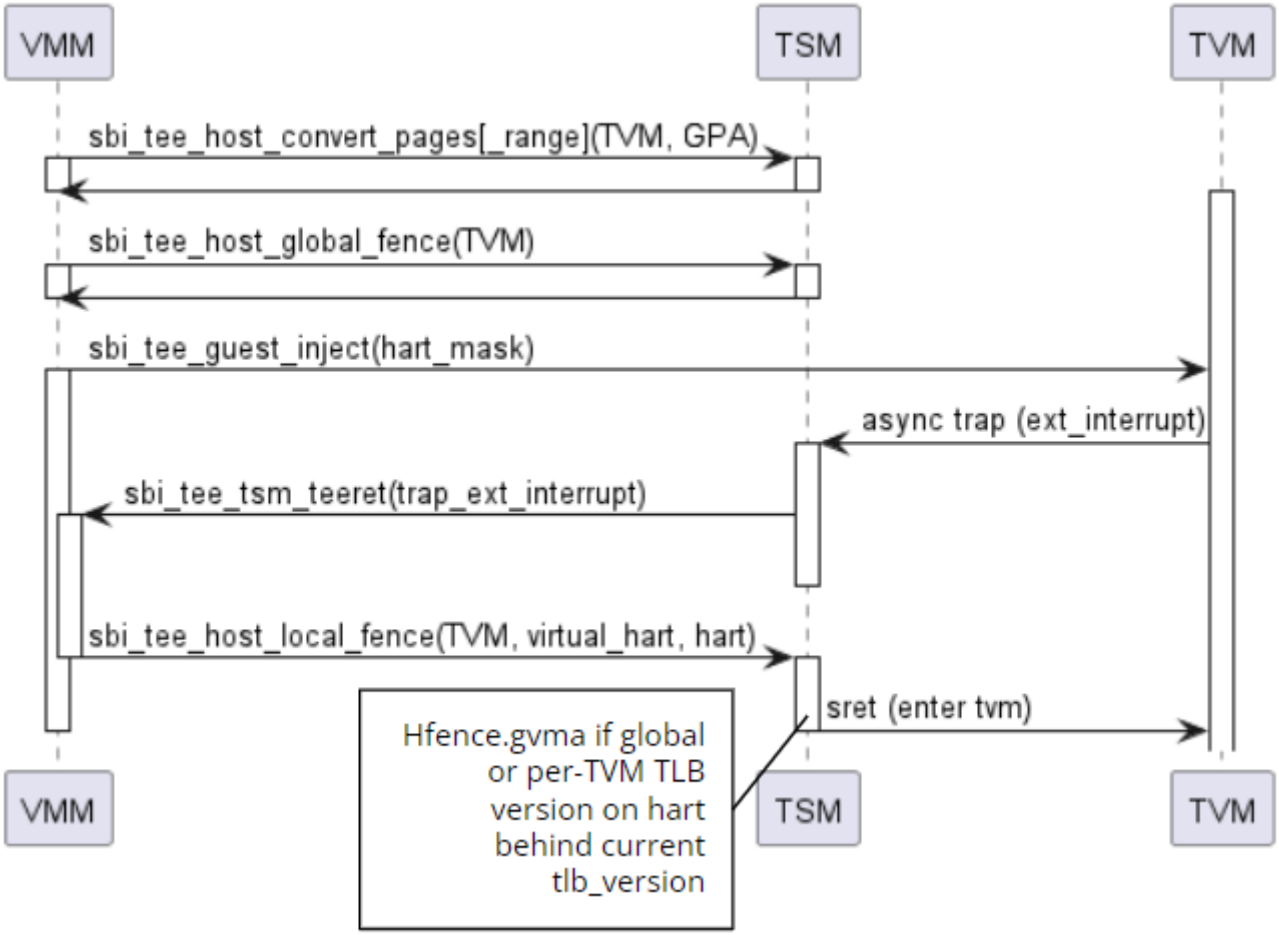

Figure 6: TLB management for memory conversion

The TSM tracks global TLB version for memory conversions and via the per-TVM and per-vcpu control structures tracks TVM-scoped TLB versions. The TSM also maintains reference counts for the number of harts that were activated during a TLB version. A similar TLB version is managed associated with the physical address in the EMTT.

If the VMM initiates memory conversion to confidential, or any change to an assigned confidential and present GPA mapping for a TVM (e.g. remove, relocate, promote etc.) - then it must execute the following sequence (enforced by TSM) to affect that change:

- Invalidate the mapping it wants to modify (page or range of pages). This step prevents new cached mappings from being populated in the TLB
- In the PA metadata maintained by the TSM (EMTT), captures into the per-page metadata, the TLB version at which the conversion was initiated or the mapping was invalidated
- Initiate global or per-TVM fence/increment the TLB version for the platform or the TVM (this operation needs to be performed only on any one hart).
- Issue an IPI to each hart (for global operations like conversion), or the TVM virtual-harts executing to trap to the TSM — this step enables the TSM to perform a local fence (via Hfence.GVMA), thus preventing pre-existing (stale) mappings from being utilized. The page meta-data is updated to complete the TLB tracking.
- TVM exit/trap allows the TSM to keep track that all active harts (for global conversion) or the TVM virtual-harts (for per-TVM scope invalidation) have been invalidated and updated to the

new TLB version - the TVM exit is reported to the VMM.

- Migration of a virtual-hart to a different hart is checked by the TSM to compares the TVM TLB version with the hart TLB version and is fenced by the TSM during vcpu run.
- -----No active/usable translations for converted memory or for TVM G-stage mappings exist at this point -----
- Invoke the specific mapping change operation (remove, relocate, promote, migrate etc.)
- Checks that the affected mapping(s) are invalidated in the MTT and/or g-stage mapping and validate the mapping
- Subsequent page walks may create cached mappings from this point onwards.

7.3.6. Page Mapping Page Assignment

The VMM uses this operation to add a hgatp structure page to be used for mapping a guest physical address (GPA) to a physical address (PA). The inputs to this operation are the TVM identifier and the physical address(es) for the new page(s) to be used for the hgatp structure entries

Page Mapping Assignment Operation:

- Verify that the TVM has been created successfully
- Verify that the PPN(s) for the new page(s) to be used for TVM hgatp is/are Unassigned-Confidential per the MTT
- For the GPA to be mapped, perform a TVM-hgatp walk to locate the non-leaf entry that should refer to the new page being added (to hold the next level of the mapping for the GPA). If the mapping already exists, the operation is aborted.
- Initialize the new hgatp page to zero (no hgatp page table entries are valid)
- Update the parent hgatp entry to refer to the new hgatp page (mark non-lead as valid)
- Update the hgatp page EMTT entry with the TVM owner-id and page-type

7.3.7. Measured page assignment into a TVM memory map

VMM uses the sbi_covh_add_tvm_zero/measured_pages interfaces to add a 4KB/2MB/1GB page to the TVM. The page assigned to the TVM is identified by its PA. A source page (also PA) may be provided to initialize the page contents. In this case, the TVM initialization must not have been committed by the VMM, and the contents of the page and the GPA selected by the VMM are measured into the TVM (static) measurement. If the contents of the page are not specified, which is allowed post-finalization of the TVM, the TSM zero's the page during initialization. The guest physical address (GPA) to the selected page physical address (PA) is specified in the add operation by the VMM. The TSM verifies that a free guest page mapping must exist for this operation to succeed. Effectively, this operation sets up the properties of the HGATP L0 leaf entry for the PA.

The inputs to this operation are: TVM identifier, physical address for the new page to be assigned to the TVM, source physical address for the source of the page contents to be loaded for the TVM (and measured by the TSM), and the GPA and page size to be used for the guest mapping to be added.

Page Assignment operation:

- Verify that the TVM has been created successfully
- If the source page is provided, this operation can only be performed if the TVM measurement has not been finalized.
- Verify that the PFN for the new page to be used for TVM is free in the MTT
- For the GPA to be mapped, perform a TVM-hgatp walk to locate the leaf entry that should refer to the new page being added. If the mapping does not exist OR exists but is not in the unmapped state, the operation is aborted.
- Initialize the new TVM page with contents from source page OR zero if no source page is provided (for lazy addition of memory to TVM). Note that the TVM initialization of memory will be with Confidential-mode asserted and via the TSMs paging structure of the PA assigned to the TVM - hence the memory will be treated as confidential.
- The measurement of the TVM is extended with the GPA used to map to the page.
- Update the TVM page MTT entry with the TVM owner PPN and page type as TEE-TVM
- Update the leaf hgatp page table entry to refer to the new page (mark leaf as valid) to allow TLB mappings to be created when the TVM vcpu is executing subsequently.

7.4. TVM Interrupt Handling

While OS/VMMs traditionally have unfettered access to the virtualized timer and interrupt state of legacy VMs, TVMs must be protected from malicious injection or filtering of interrupts or modification of timers which could lead to incorrect execution of or information leakage from the TVM. As such, a combination of hardware isolation features and COVH support are necessary to guard access to this state while still ultimately giving the OS/VMM control over resource management.

7.4.1. TVM timers

The Sstc ISA extension allows for configuration and delivery of timer interrupts directly at VS level without the involvement of HS-level software. While this feature can mostly be used as-is to provide isolated timer support for TVMs, the TSM must still ensure that VS-level timer state cannot be modified by the OS/VMM. In particular: The TSM should ensure that VS-level timer interrupts intended for a TVM are delivered to the TVM without OS/VMM involvement while the TVM is running. This is done by delegating (hideleg[6] = 1) and enabling (hie.VSTIE = 1) VS-level timers at VS level.

While the OS/VMM should still be able to read a TVM's vstimecmp (for scheduling purposes), it must not be able to overwrite it. To support this the TSM and TSM-driver should leave the vstimecmp CSR intact when context-switching back to the OS/VMM, but should always restore the vstimecmp CSR from saved state when resuming.

7.4.2. TVM external interrupts

Hardware-accelerated interrupt-controller virtualization is possible for TVMs on platform supporting the Advanced Interrupt Architecture [AIA] and an implementation-defined method of isolating IMSIC guest interrupt files between the non-TEE and TEE worlds (either using an MTT as

described above, or via other means). This enables delivery of MSIs from TVM-assigned devices and inter-processor interrupts without OS/VMM interference for TVM virtual harts.

The AIA supports two mechanisms for tracking of interrupts at VS-level: IMSIC guest interrupt files, of which there are a fixed number per physical hart. These allow delivery of external interrupts directly to VS-level as a Virtual Supervisor External Interrupt. Guest interrupt files occupy a single 4kB page of physical address space.

Memory-resident interrupt files (MRIFs), which track pending and enabled interrupts in a 4kB page of DRAM. While the RISC-V IOMMU supports automatically updating an MRIF's pending bits and delivering a notice interrupt to the host when an MSI is targeted at an MRIF, the hypervisor is still responsible for injection of the VSIE to the guest. IPI emulation must be provided by the hypervisor. MRIFs are only constrained by the amount of available DRAM, however.

While it is possible to support execution of a TVM virtual hart using either a guest interrupt file or an MRIF, the architecture describes below constraints for the TVM virtual harts to only use guest interrupt files while they are actively executing in order to simplify the duties of the TSM. Inactive (swapped out) TVM virtual harts may use an MRIF, however, and an MRIF is required when migrating a TVM virtual hart between physical harts. In either case the page of physical memory corresponding to a guest interrupt file or MRIF for a TVM virtual hart must be considered confidential to the TVM and must be inaccessible to the OS/VMM. The implementation must additionally provide a mechanism for isolating guest interrupt file CSR state from the OS/VMM.

Two fundamental operations must be supported by the TSM in order to enable the use of the IMSIC or MRIFs for TVM virtual harts:

Binding a TVM virtual hart to an IMSIC guest interrupt file on a physical CPU, migrating any interrupt state from the virtual hart's MRIF.

Unbinding a TVM virtual hart from an IMSIC guest interrupt file and migrating interrupt state to an MRIF.

If MRIFs are not supported by the hardware then TSM must additionally support one more operation to allow TVM virtual hart migration from one physical hart to another:

Rebinding a TVM virtual hart to an IMSIC guest interrupt file on a physical CPU, migrating any interrupt state from the virtual hart's previous IMSIC guest interrupt file.

Additionally, the TSM must provide a way for the OS/VMM to query if an inactive virtual hart has external interrupts pending. The COVH calls to support these operations are described below:

tvm_vhart_aia_init

Initializes the AIA state for a virtual hart. Must be called after the virtual hart has been added but before the TVM is run for the first time.

The OS/VMM supplies: The guest physical address of the IMSIC for the virtual hart The supervisor physical address of a page of confidential memory that is to be used as an MRIF for the virtual hart. The page is available to be reclaimed upon destruction of the virtual hart. An MSI address + data pair that is to be signaled when an MSI is delivered to a virtual hart's MRIF.

tvm_vhart_imsic_bind

Binds a virtual hart to a guest interrupt file on the current physical hart. The guest interrupt file number is supplied by the OS/VMM.

The TSM is then responsible for: Converting the guest interrupt file page to confidential memory. Updating IOMMU MSI page tables with the address of the interrupt file. Migrating MRIF state (if any) to the guest interrupt file. Mapping the guest interrupt file at the previously-specified address in the TVM's guest physical address space.

Upon success the virtual hart is considered "bound" to the current physical hart and is eligible to be run. Attempts to run the virtual hart on a different physical hart or to run an "unbound" virtual hart shall return an error.

Note that depending on the implementation's mechanism for isolating guest interrupt files, a coordinated TLB invalidation of the guest interrupt file using the invalidate + fence procedure described in Section 7.3 may be required when converting the interrupt file to confidential memory.

tvm_vhart_imsic_unbind

Unbinds the virtual hart from its guest interrupt file, migrating it to an MRIF. Must be called from the same physical hart to which the virtual hart is currently bound.

The OS/VMM is responsible for coordinating a TLB invalidation of the address of the guest interrupt file in the TVM's guest physical address space using the invalidate + fence procedure described in Section 7.3.

The TSM is then responsible for: Verifying that TLB invalidation of the guest interrupt file is complete. Updating IOMMU MSI page tables. Copying interrupt state from the guest interrupt file to the virtual hart's MRIF. Converting the guest interrupt file back to a non-confidential state.

Upon success the virtual hart is considered "unbound" and the guest interrupt file it was using is available for OS/VMM use.

While a TVM virtual hart is unbound, MSIs directed at the virtual hart shall trigger the notice interrupt registered in tvm_vhart_aia_init. Attempts by other TVM virtual harts to write the virtual hart's IMSIC in the guest physical address space (e.g. for the purposes of generating an IPI) shall generate a guest page fault exit on the virtual hart which initiated the write.

tvm_vhart_imsic_rebind

Rebinds a virtual hart to a guest interrupt file on the current physical hart. The guest interrupt file number is supplied by the OS/VMM. State of the previous guest interrupt file is copied over to the new file at the end of the operation.

This is an optional interface that must be supported in case of missing MRIF support. Given the complexity introduced due to missing MRIF the interface is divided into three ABI calls to migrate a virtual hart:

- tvm_vhart_imsic_rebind_begin(): Attaches the hart to the new interrupt file and updates

IOMMU MSI page tables with the address of the new interrupt file. The previous interrupt file is no more in use after this call and all the interrupts are forwarded to the new interrupt file.

- tvm_vhart_imsic_rebind_clone(): This must be called from the previous physical hart to create a copy of the previous interrupt file state.
- tvm_vhart_imsic_rebind_end(): Must be run on the new hart. This call copies over the saved interrupt state to new interrupt file.

Upon success, the virtual hart is considered "bound" to the current physical hart and is eligible to be run. Attempts to run the virtual hart on a different physical hart or to run a "rebinding" virtual hart shall return an error. The previous interrupt file is now free to be used by another virtual hart.

Note that depending on the implementation's mechanism for isolating guest interrupt files, a coordinated TLB invalidation of the guest interrupt file using the invalidate + fence procedure described in Section 7.3 may be required when converting the interrupt file to confidential memory.

tvm_vhart_external_interrupt_pending

Returns if the virtual hart has an external interrupt pending. For virtual harts using guest interrupt files, it is expected that the OS/VMM will use the hgeip CSR and Supervisor Guest External Interrupts to determine if the virtual hart has an interrupt pending. For virtual harts using MRIFs, the OS/VMM may need this call to disambiguate the cause of a notice interrupt from the IOMMU. In either case the TSM should inspect the interrupt state of the specified virtual hart and return whether or not it has an external interrupt pending.

7.4.3. Paravirtualized I/O

It is expected that the OS/VMM will need to provide paravirtualized I/O support to TVMs, which naturally requires that the OS/VMM be able to inject VSEI to TVM virtual harts. The OS/VMM must not be allowed to arbitrarily inject such interrupts, however, so the TSM must provide a mechanism whereby only allow-listed interrupts may be triggered.

sbi_covg_allow_external_interrupt

Registers an interrupt ID that the OS/VMM is allowed to trigger. Passing an interrupt ID of -1 allows the injection of all external interrupts. TVM vCPUs are started with all external interrupts completely denied by default. Generates a TVM exit to notify the OS/VMM of the interrupt vector.

sbi_covi_inject_tvm_cpu

Injects a previously allow-listed interrupt into a TVM. The TSM updates the interrupt state of the targeted virtual hart. The TSM may also enforce rate-limiting on the injection of interrupts in order to prevent single-step attacks by the OS/VMM.

7.5. 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 sbi_covh_destroy_tvm that can verify that no TVM harts are executing and unassigns all memory assigned to the TVM.

The VMM may choose grant the confidential memory to another TVM or may reclaim all memory granted to the TVM via sbi_covh_reclaim_pages which will verify the TSM hgatp mapping and tracking for the page and restore it as a VMM-available page to grant to a non-confidential VM.

Reclaim TSM operation:

- Verifies that the PAs referenced are either Non-confidential (No-operation) or Confidential-Unassigned state
- TSM takes exclusive lock over the MTT tracker entry for the PA
- TSM scrubs page contents
- TSM updates MTT tracker entry (synchronized) for the page as Non-confidential and returns the PA as an Non-Conf page to the VMM
- VMM translations to the PA (via 1st or G stage mappings) may be created now

7.6. RAS interaction

The TSM performs minimal fail-safe tasks when handling RAS events. RAS-induced access violations on a TVM lead to TSM-enforced TVM shutdown and are reported to the OS/VMM for further analysis (without allowing any TVM access). Similarly, RAS-interrupts (both high and low priority) are forwarded by the TSM to the OS/VMM for handling.

Chapter 8. Confidential VM Extension (CoVE) SBI extension proposal

This section describes the normative Confidential VM Extension(CoVE) SBI extension proposal. The proposal introduces three new extensions that will be described later:

- CoVE Host Extension (EXT_COVH)
- CoVE Interrupt Extension (EXT_COVI)
- CoVE Guest Extension (EXT_COVG)

8.1. TEEI - COVH runtime interface

ECALL invocation from VS (guest OS) causes traps that are handled by the TSM module (enforced via `medeleg` configuration). The TSM then may provide intrinsics via the COVG (CoVE-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 COVH (CoVE host-ABI).

8.1.1. Operational model for the CoVE Host Extension

Executing confidential workloads in a CoVE 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_COVH extension, and then calling `sbi_covh_get_tsm_info()` to get information about the current status of the TSM. The TSM must be in `TSM_READY` in order to process further ECALLs.

TVM creation

TVMs are created using the sbi_covh_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 are 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_covh_convert_pages()`
2. `sbi_covh_global_fence()`
3. `sbi_covh_local_fence()`

The memory conversion process is complete when sbi_covh_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 confidential memory. There can be multiple 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_covh_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_covh_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_covh_add_tvm_memory_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_covh_add_tvm_measured_pages()`. This operation requires the host to convert a sufficient number of non-confidential pages to confidential (by calling `sbi_covh_convert_pages()`, or by using converted pages 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

Host needs access to some of the TVM CSRS and GPRs to handle TVM exits. For example, the host needs `htval` to determine the fault address, `a0`-`a7` GPRs are needed to handle forwarded ECALLs and so on. For this purpose, the host and TSM use NACL Extension based shared memory interface [R10], from now on called NACL shared memory to avoid confusion with shared memory pages between TVM and the host.

The NACL shared memory interface is between TSM and the host and TSM is responsible for writing any trap-related CSRs and GPRs needed by the host to handle the exception. TSM is also responsible for reading the returned result and forwarding it to the TVM. Further details about which CSRs and GPRS are used by the TSM and the host can be found in Table 1. The layout of NACL shared memory is shown below as `struct nacl_shmem` and `scratch` space layout for TSM is shown as `struct tsm_shmem_scratch`.

```
struct nacl_shmem {
    /* Scratch space. The layout of this scratch space is defined by the particular
function being
     * invoked.
     *
     * For the `sbi_covh_run_tvm_vcpu()` function in the COVH extension, the layout of
this
     * scratch space matches the `tsm_shmem_scratch` struct given below.
     */
    uint64_t scratch[256];
```

```
    uint64_t _reserved[240];
    /* Bitmap indicating which CSRs in `csrs` the host wishes to sync.
     *
     * Currently unused in the CoVE extensions and will not be read or written by the
TSM.
     */
    uint64_t dirty_bitmap[16];
    /* Hypervisor and virtual-supervisor CSRs. The 12-bit CSR number is transformed
into a 10-bit
     * index by extracting bits `{csr[11:10], csr[7:0]}` since `csr[9:8]` is always
2'b10 for HS
     * and VS CSRs.
     *
     * These CSRs may be updated by `sbi_covh_run_tvm_vcpu()` in the COVH extension.
See
     * the documentation of `sbi_covh_run_tvm_vcpu()` for more details.
     */
    uint64_t csrs[1024];
};

struct tsm_shmem_scratch {
    /* General purpose registers for a TVM guest.
     *
     * The TSM will always read or write the minimum number of registers in this set
to complete
     * the requested action. To avoid leaking information from the TVM, the TSM must
follow the
     * given rules.
     *
     * The TSM will write to these registers upon return from
`sbi_covh_run_tvm_vcpu()` when:
     *  - The vCPU takes a store guest page fault in an emulated MMIO region.
     *  - The vCPU makes an ECALL that is to be forwarded to the host.
     *
     * The TSM will read from these registers when:
     *  - The vCPU takes a load guest page fault in an emulated MMIO region.
     */
    uint64_t guest_gprs[32];
    uint64_t _reserved[224];
};
```

The below table describes the list of CSRs and GPRs that the TSM and the host are supposed to use from NACL shared memory. It also describes the operation allowed for each entity in terms of `R` (read) and `W` (write) permissions. Note that the TSM and the host can read/write to any of the fields without any faults but the permissions depict the expected use case. For write only CSRs or GPRs TSM is supposed to ignore any modifications by the host. TSM is only supposed to take modifications from CSRs or GPRs with read permission such as `a0` and `a1` GPRs.

Table 1. TSM NACL CSRs and GPRs

CSRs	TSM	Host	Purpose
htinst	W	R	TSM writes the faulting instruction into htinst to allow the host to emulate the MMIO.
htval	W	R	In case of a guest page-fault, TSM writes the guest's physical address that faulted into htval CSR.
htimedelta	W	R	TSM writes the guest htimedelta in this CSR. This is to allow the host to schedule an internal software timer for the guest to keep the timer interrupt ticking.
vstimecmp	W	R	TSM writes the guest's vstimecmp to allow the host to schedule an internal software timer for the guest.
vsie	W	R	TSM writes the guest's vsie to allow the host to check which interrupts are enabled. This is useful in waking up a guest's vcpu when it's sleeping due to a `WFI` instruction.
GPRs			
a0	RW	RW	Used for both passing argument and returning the result for ECALLs forwarded to the host.
a1	RW	RW	Used for both passing argument and returning the result for ECALLs forwarded to the host.
a2	W	R	Used for passing an argument for ECALLs forwarded to the host.
a3	W	R	Used for passing an argument for ECALLs forwarded to the host.
a4	W	R	Used for passing an argument for ECALLs forwarded to the host.
a5	W	R	Used for passing an argument for ECALLs forwarded to the host.
a6	W	R	Used for passing an argument for ECALLs forwarded to the host.
a7	W	R	Used for passing an argument for ECALLs forwarded to the host.
x0-x31	RW	RW	Any of the GPR used in load/store instruction trapped for MMIO emulation.

It's recommended that the TSM should transform the load or store instruction to/from `a0` before writing to the htinst CSR. So that `a0` will be the only GPR used for MMIO emulation reducing the GPRs accessible to the host.

VCPU creation

The host must register CPUs/harts with the TSM before they can be used for TVM execution by calling `sbi_covh_create_tvm_vcpu()`. The NACL shared memory interface is used between the host and the TSM for processing TVM exits from `sbi_covh_run_tvm_vcpu()`.

TVM execution

Following the assignment of memory and VCPU resources, the host can transition the guest into a `TVM_RUNNABLE` state by calling `sbi_covh_finalize_tvm()`. The host must set up TVM Boot vCPU execution parameters like the entrypoint (`ENTRY_PC`) and boot argument (`ENTRY_ARG`) using

arguments to `sbi_covh_finalize_tvm()`. Note that some TEE calls are no longer permissible after this transition.

The host can then call sbi_covh_run_tvm_vcpu()` to begin execution. The host must boot vCPU `0` first otherwise `sbi_covh_run_tvm_vcpu()` call will fail. TVM execution continues until there is 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 specific reason by examining the `scause` CSR. The host can then examine the NACL shared memory if needed to determine further course of action. This may involve servicing exits caused by TVM-ECALLs that require host action (like adding MMIO region or share memory with the host), 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 use `sbi_covh_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 to the TVM. Demand-zero pages have no bearing on the TVM measurement, and can be added at any point in time.

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

Handling MMIO faults

TVMs can define MMIO regions using the EXT_COVG extension, and a runtime access to such a region causes a resumable exit from the TVM. The host can examine the exit code from `scause` CSR, and when the exception is a guest load/store page fault, the host will check if the fault address belongs to any of the registered MMIO emulation regions. The fault address information comes from `stval` and `htval` CSRs. After emulation, the host updates the NACL shared memory region as appropriate and resumes TVM execution. This process also involves instruction decoding using the `htinst` CSR from the NACL shared memory region.

Handling virtual instructions

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

Management of secure interrupts

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

TVM teardown

The host can teardown a TVM by calling `sbi_covh_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_covh_reclaim_pages()`.

8.1.2. Operational model for the CoVE Guest Extension

This interface is used by TVMs to communicate with TSM. Presently, this extension allows guests to define memory regions for MMIO emulation by host, share pages with the host and control interrupt injection by host.

TVM-defined MMIO regions

TVM can register the physical address location as a non-confidential MMIO region at runtime to be emulated by the host. This is done by calling `sbi_covg_add_mmio_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 NACL shared memory region. The expectation is that the host will service a subsequent page-fault that results from a TVM-access to the non-confidential region.

TVM-defined Shared memory regions

TVMs can choose to yield access to confidential memory at runtime and request shared (non-confidential) memory. The TVM must communicate its request to the host to convert confidential to non-confidential and vice-versa explicitly via the `sbi_covg_share_memory_region()` and `sbi_covg_unshare_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. If the region of address space is populated, the host must first invalidate and remove the confidential pages. This requires the host to make three separate ECALLs to the TSM:

1. `sbi_covh_tvm_invalidate_pages()`
2. `tee_host_tvm_initiate_fence()`
3. `sbi_covh_tvm_remove_pages()`

Upon completion, the host may reclaim the confidential pages that were previously mapped in the region using `tee_host_tsm_reclaim_pages()`. The host must then continue the TVM execution and insert shared pages into the region using `tee_host_tvm_add_shared_pages()` on the page-fault when TVM tries to access the region. If the region of address space is unpopulated, the page removal ECALLs are not needed and the host can insert shared pages into the region on the next page-fault.

The calling TVM vCPU is considered blocked until the assignment-change is completed. Attempts to run it with `sbi_covh_run_tvm_vcpu()` will fail. Any guest page faults taken by other TVM vCPUs in the invalidated pages continue to be reported to the host.

Both sharing and unsharing operations are destructive, i.e. the contents of memory in the range to be converted are lost.

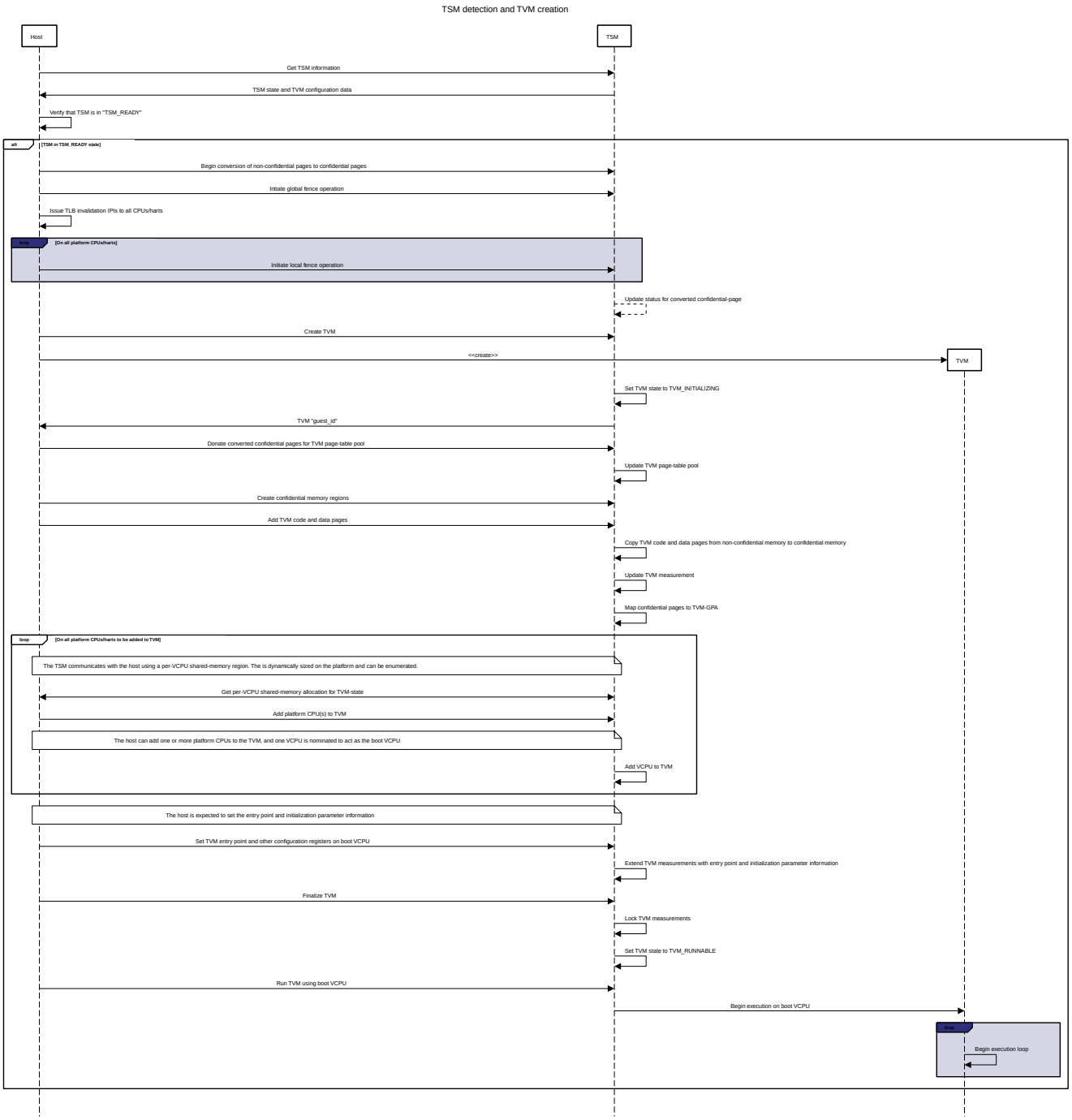

Figure 7: TSM Detection and TVM creation

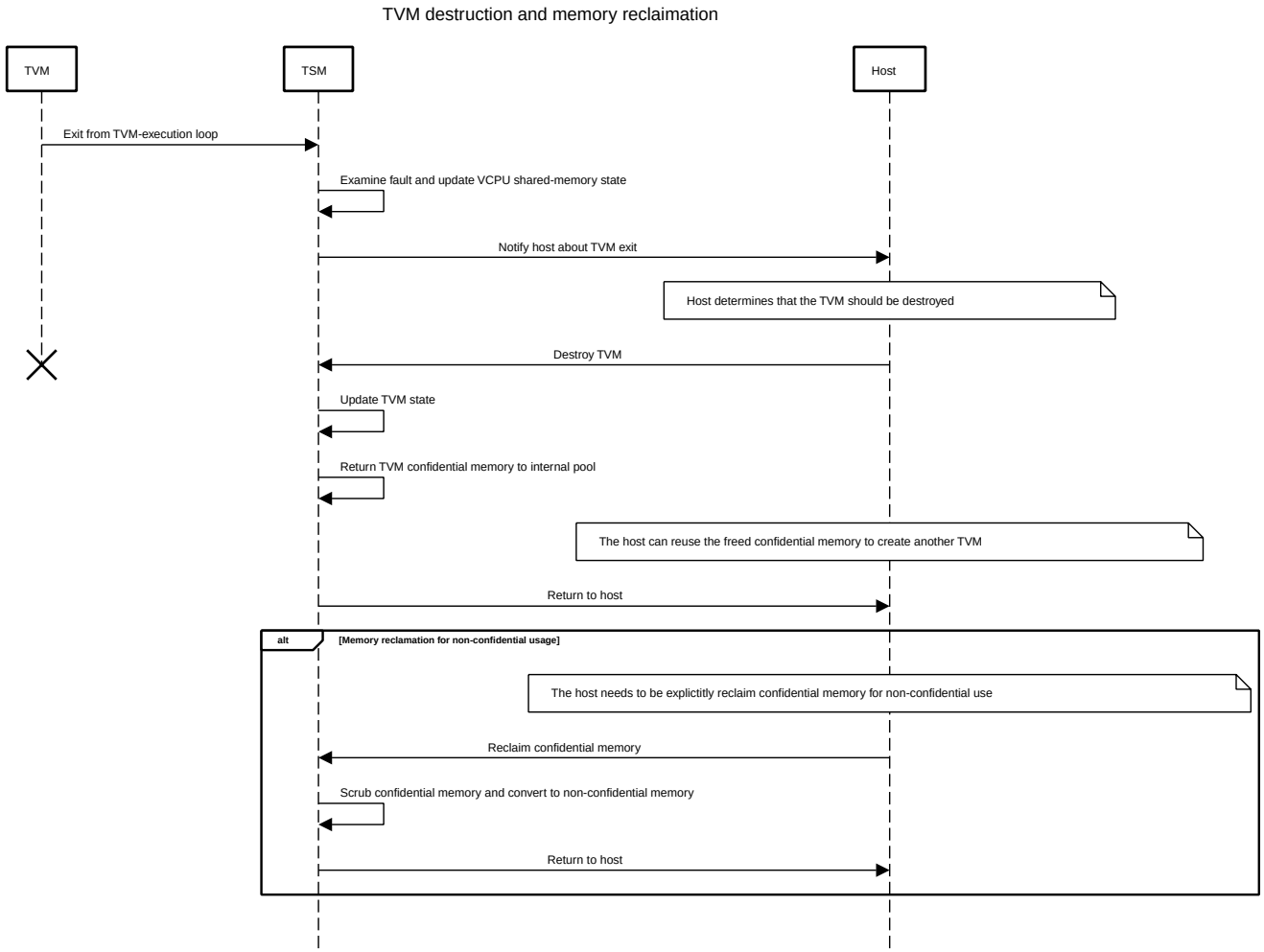

Figure 8: TVM destruction and Memory reclamation

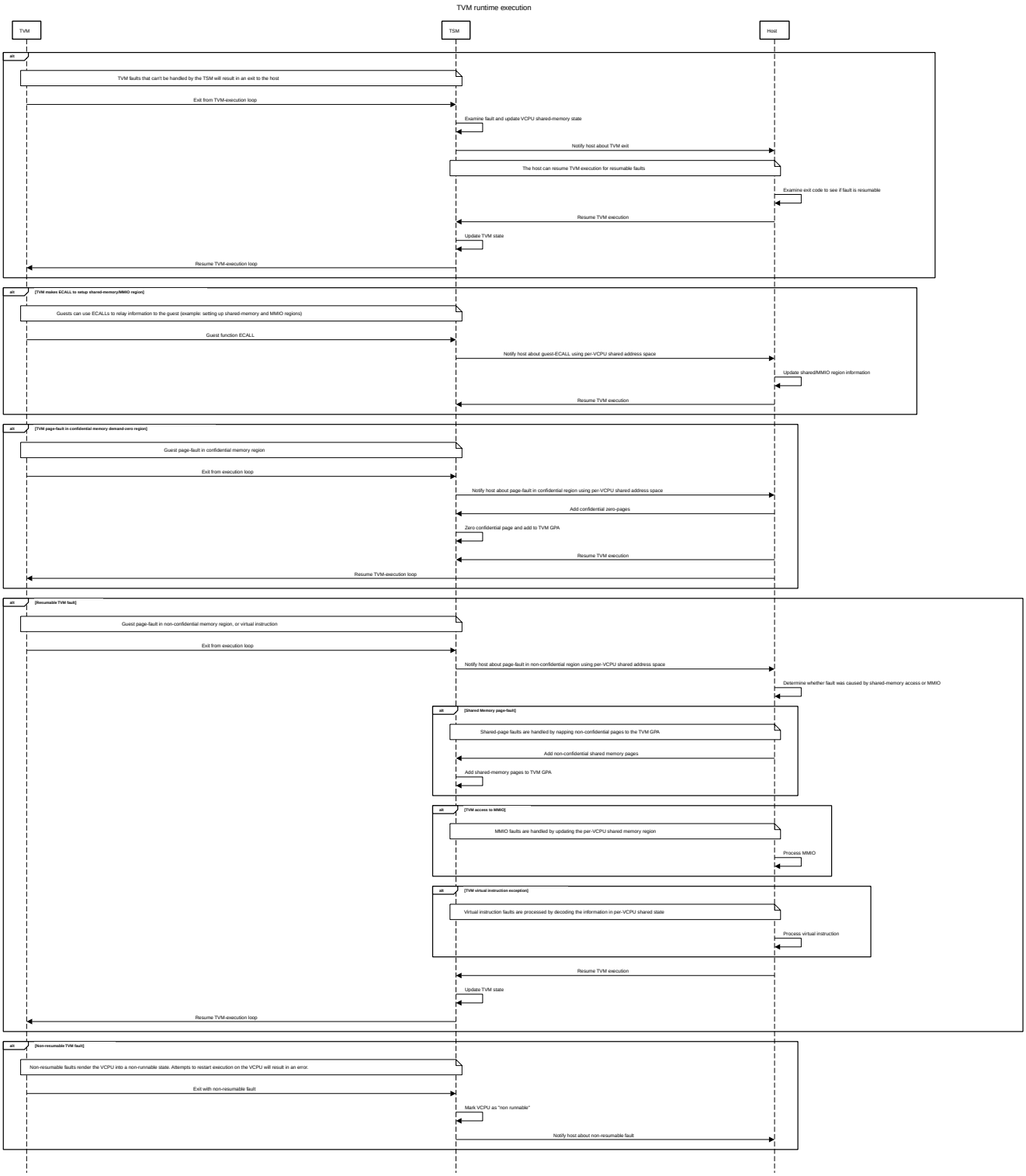

Figure 9: TVM runtime execution

Chapter 9. COVE Host Extension (EID #0x434F5648 "COVH")

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,
};
```

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

```
struct sbiret sbi_covh_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 `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_covh_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 2. COVE Host Get TSM Info

Error code	Description
SBI_SUCCESS	The operation completed successfully.
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 3. 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: COVE Host Convert Pages (FID #1)

```
struct sbiret sbi_covh_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_covh_global_fence()` and `sbi_covh_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 4. COVE 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: COVE Host Reclaim Pages (FID #2)

```
struct sbiret sbi_covh_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 5. COVE 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: COVE Host Initiate Global Fence (FID #3)

```
struct sbiret sbi_covh_global_fence(void);
```

Initiates a TLB invalidation sequence for all pages marked for conversion via calls to `sbi_covh_convert_pages()`. The TLB invalidation sequence is completed when `sbi_covh_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 6. COVE 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: COVE Host Local Fence (FID #4)

```
struct sbiret sbi_covh_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 7. COVE Host Local Fence

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_FAILED	The operation failed for unknown reasons.

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

```
struct sbiret sbi_covh_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.

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 virtual hart settings. For example, features enabled for this TVM, perfmon enabled, debug enabled etc.

Callers of this API should first invoke `sbi_covh_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 TVM's 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 TVM's 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_covh_get_tsm_info().
     */
    unsigned long tvm_state_addr;
};
```

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_covh_finalize_tvm()` is made to transition the TVM to a `TVM_RUNNABLE` state.

The list of possible TVM states appears below.

Table 8. COVE TVM States

State	Description
TVM_INITIALIZING	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

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

Table 9. COVE 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: COVE Host Finalize TVM (FID #6)

```
struct sbiret sbi_covh_finalize_tvm(unsigned long tvm_guest_id,
                                    unsigned long entry_sepc,
```

```
                                        unsigned long entry_arg);
```

Transitions the TVM specified by `tvm_guest_id` from the `TVM_INITIALIZING` state to a `TVM_RUNNABLE` state. Also, sets the entry point (`ENTRY_PC`) using `entry_sepc` and boot argument (`ENTRY_ARG`) using `entry_arg` for the boot VCPU. Both `entry_sepc` and `entry_arg` are included in the measurement of the TVM. 'entry_sepc' is the address in TVM binary to start the boot VCPU from and `entry_arg` is the address of guest fdt and is passed as an argument to the boot VCPU in `a1` GPR.

The TSM enforces that a TVM virtual harts cannot be entered unless the TVM measurement is committed via this operation. No additional measured pages may be added after this operation is successfully completed.

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

Table 10. COVE 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: COVE Host Destroy TVM (FID #7)

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

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

Confidential TVM memory is automatically un-assigned 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_covh_reclaim_pages()`** (described below).

TVM destroy verifies that the VMM has stopped all virtual harts execution for the TVM otherwise this call will fail. The TVM virtual hart may not be entered after this point. The VMM may start reclaiming TVM memory after this call succeeds.

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

Table 11. COVE 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: COVE Host Add TVM Memory Region (FID #8)

```
struct sbiret sbi_covh_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_covh_finalize_tvm()`.

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

Table 12. COVE 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: COVE Host Add TVM Page Table Pages (FID #9)

```
struct sbiret sbi_covh_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 use case is in response to a TVM page fault.

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

Table 13. COVE Host Add TVM Page Table Pages

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_ADDRESS	`base_page_address` was invalid.

Error code	Description
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: COVE Host Add TVM Measured Pages (FID #10)

```
struct sbiret sbi_covh_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_address`, then measures and maps the pages at `dest_address` at the TVM physical address space at `tvm_guest_gpa`. The mapping must lie within a region of confidential memory created with `sbi_covh_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_covh_finalize_tvm()`.

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 4KB page. The page must be added to a valid GPA mapping. 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 by the relying party.

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

Table 14. COVE 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.

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

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

```
struct sbiret sbi_covh_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_covh_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.

This call may be made only after calling `sbi_covh_finalize_tvm()`.

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

Table 15. COVE Host Add TVM Zero Pages Errors

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.

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

```
struct sbiret sbi_covh_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 16. COVE 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: COVE Host Create TVM VCPU (FID #13)

```
struct sbiret sbi_covh_create_tvm_vcpu(unsigned long tvm_guest_id,
                                       unsigned long tvm_vcpu_id,
                                       unsigned long tvm_state_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. This call must not be made after calling `sbi_covh_finalize_tvm()`.

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

Table 17. COVE 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.

9.16. Function: COVE Host Run TVM VCPU (FID #14)

```
struct sbiret sbi_covh_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_covh_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_covh_run_tvm_vcpu()` with the same `tvm_vcpu_id` will fail.

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

Table 18. COVE Host Run TVM VCPU Errors

Error code	Description
SBI_ERR_SUCCESS	The TVM exited, and sbiret.value contains 0 if the 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 hosts `scause` CSR. 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 in the NACL 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 the exit was caused by an interrupt, and if this bit is clear, the implication is that the exit was caused by an exception. The remaining bits are 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 NACL shared memory area 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` and `stval`
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_covh_add_tvm_zero_pages() to add a demand-zero confidential page
if
     * applicable), and then calling sbi_covh_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 `htinst` CSR and determine the further course of action, and then
calling
     * sbi_covh_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` and `stval`
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_covh_add_tvm_zero_pages() to add a demand-zero confidential page
if
     * applicable), and then calling `sbi_covh_run_tvm_vcpu()` to resume execution at
the following
     * instruction.
     */
    GUEST_STORE_PAGE_FAULT = 23,
};
```

9.17. Function: COVE Host Initiate TVM Fence (FID #15)

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

Initiates a TLB invalidation sequence for all pages that have been invalidated in the given TVM's address space since the previous call to `sbi_covh_tvm_fence()`. The TLB invalidation sequence is completed when all vCPUs in the TVM that were running prior to the call to `sbi_covh_tvm_fence()` have taken a trap into the TSM, which the host can cause by sending an IPI to the physical CPUs on which the TVM's vCPUs are running. Note that the physical CPUs don't have to necessarily perform anything on those IPIs. An error is returned if a TLB invalidation sequence is already in progress for the TVM.

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

Table 19. COVE Host Initiate TVM 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.

9.18. Function: COVE Host TVM Invalidate Pages (FID #16)

```
struct sbiret sbi_covh_tvm_invalidate_pages(unsigned long tvm_guest_id,
                                            unsigned long gpa,
```

```
                                                unsigned long length);
```

Invalidates the pages in the specified range of guest physical address space and thus marks the pages as blocked from any further TVM accesses.

For each page in the range, the TSM must verify that:

- The page is currently marked present in the TVM's page table.
- The page is either mapped and uniquely owned by the TVM, or shared and owned by the host.

After verifying these pre-conditions are met, the TSM then invalidates the pages. The host must complete a TVM TLB invalidation sequence, initiated by `sbi_covh_tvm_fence()`, in order to complete the invalidation.

Guest page faults taken by the TVM on invalidated pages continue to be reported to the host. The pages remain invalid until the mappings are validated (marked present), removed, or become part of a huge page by promotion/demotion operation.

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

Table 20. COVE Host TVM Invalidate Pages

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

9.19. Function: COVE Host TVM Validate Pages (FID #17)

```
struct sbiret sbi_covh_tvm_validate_pages(unsigned long tvm_guest_id,
                                          unsigned long gpa,
                                          unsigned long length);
```

Marks the invalidated pages in the specified range of guest physical address space as present.

For each page in the range, the TSM must verify that the page was previously invalidated using `sbi_covh_tvm_invalidate_pages()`. After verifying the TSM will mark the pages as present and restore the pages to their previous state.

This ECALL may be used to revert an in-progress page removal or huge page promotion/demotion sequence.

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

Table 21. COVE Host TVM Validate Pages

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

9.20. Function: COVE Host TVM Remove Pages (FID #18)

```
struct sbiret sbi_covh_tvm_remove_pages(unsigned long tvm_guest_id,
                                        unsigned long gpa,
                                        unsigned long length);
```

Removes mappings for invalidated pages in the specified range of guest physical address space. The range to be unmapped must already have been invalidated and fenced, and must lie within a removable region of the guest's physical address space. The TSM zeros out all PTEs within the specified range and returns the ownership of the pages to the host if previously owned by the TVM.

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

Table 22. COVE Host TVM Remove Pages

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

Chapter 10. COVE Interrupt Extension (EID #0x434F5649 "COVI")

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

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

```
struct sbiret sbi_covi_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_covh_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 23. COVE 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: COVE Interrupt Set TVM AIA CPU IMSIC Addr (FID #1)

```
struct sbiret sbi_covi_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_covi_init_tvm_aia()`. No two VCPUs may share the same `tvm_vcpu_imsic_gpa`.

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

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

Table 24. COVE 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: COVE Interrupt Convert AIA IMSIC (FID #2)

```
struct sbiret sbi_covi_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_covh_global_fence()` and `sbi_covh_local_fence()` before the interrupt file can be assigned to a TVM.

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

Table 25. COVE 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: COVE Interrupt Reclaim TVM AIA IMSIC (FID #3)

```
struct sbiret sbi_covi_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 26. COVE 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.

10.5. Function: COVE Interrupt Bind AIA IMSIC (FID #4)

```
struct sbiret sbi_covi_bind_aia_imsic(unsigned long tvm_guest_id,
                                      unsigned long tvm_vcpu_id,
                                      unsigned long imsic_mask);
```

Binds a TVM vCPU to the current physical CPU using the confidential guest interrupt files specified in `imsic_mask`, restoring interrupt state from the vCPU's software interrupt file if necessary. Note that `imsic_mask` is in the same format as the `hgeie` and `hgeip` CSRs, that is bit N corresponds to guest interrupt file N-1 and bit `0` is always `0`. The number of bits set in `imsic_mask` must be equal to the number of interrupt files in the vCPU's virtualized IMSIC (i.e. 1 + `guests_per_hart`). The vCPU must currently be unbound. Upon completion, the vCPU is eligible to be run on this CPU with `sbi_covh_run_tvm_vcpu()`.

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

Table 27. COVE Interrupt Bind AIA IMSIC

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_PARAM	`tvm_guest_id` or `tvm_vcpu_id` or `imsic_mask` were invalid.
SBI_ERR_FAILED	The operation failed for unknown reasons.

10.6. Function: COVE Interrupt Unbind AIA IMSIC Begin (FID #5)

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

Begins the unbinding process for the specified vCPU from its guest interrupt files. The translations for the vCPU's virtualized IMSIC are invalidated, and a TLB flush sequence for the TVM must be completed before calling `sbi_covi_unbind_aia_imsic_end()` to complete the unbinding process. Must be called on the physical CPU to which the vCPU is bound.

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

Table 28. COVE Interrupt Unbind AIA IMSIC Begin

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

10.7. Function: COVE Interrupt Unbind AIA IMSIC End (FID #6)

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

Completes the unbinding process for the specified vCPU from its guest interrupt files after a TLB flush sequence for the TVM has been completed. The interrupt state is saved to the vCPU's software interrupt file and the guest interrupt files are free to be reclaimed via `sbi_covi_reclaim_tvm_aia_imsic()` or bound to another vCPU via `sbi_covi_unbind_aia_imsic_begin()`. Must be called on the physical CPU to which the vCPU is bound. Upon success, the vCPU is free to be bound to another physical CPU.

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

Table 29. COVE Interrupt Unbind AIA IMSIC End

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

10.8. Function: COVE Interrupt Inject TVM CPU (FID #7)

```
struct sbiret sbi_covi_inject_tvm_cpu(unsigned long tvm_guest_id,
                                      unsigned long tvm_vcpu_id
                                      unsigned long interrupt_id);
```

Injects an external interrupt with the given `interrupt_id` into the specified vCPU. If the vCPU is presently bound to an IMSIC guest interrupt file, the interrupt is immediately injected by writing to the interrupt file. If it is not bound, the interrupt is recorded in the software and will be injected once the vCPU becomes bound. The specified interrupt ID must be valid and must have been allowed by the guest with `sbi_covg_allow_external_interrupt()`.

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

Table 30. COVE Interrupt Inject TVM CPU

Error code	Description
SBI_SUCCESS	The operation completed successfully.
SBI_ERR_INVALID_PARAM	`tvm_guest_id` or `tvm_vcpu_id` or `interrupt_id` were invalid.
SBI_ERR_FAILED	The operation failed for unknown reasons.

10.9. Function: COVE Interrupt Rebind AIA IMSIC Begin (FID #8)

```
struct sbiret sbi_covi_rebind_aia_imsic_begin(unsigned long tvm_guest_id,
                                              unsigned long tvm_vcpu_id,
                                              unsigned long imsic_mask);
```

Begins the rebinding process for the specified vCPU to the current physical CPU and the specified confidential guest interrupt file. The host must complete a TLB invalidation sequence for the TVM before cloning the old interrupt file state using `sbi_covi_rebind_aia_imsic_clone()`. Once cloned, the old file will be restored to the new guest interrupt file on `sbi_covi_rebind_aia_imsic_end()` invocation.

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

Table 31. COVE Interrupt Rebind AIA IMSIC Begin

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

10.10. Function: COVE Interrupt Rebind AIA IMSIC Clone (FID #9)

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

TSM clones the old guest interrupt file of the specified VCPU. The cloned copy is maintained in VCPU specific structure visible to TSM only. The host must make sure to invoke this from the old physical CPU. The guest interrupt file after this is free to be reclaimed or bound to another VCPU.

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

Table 32. COVE Interrupt Rebind AIA IMSIC Clone

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

10.11. Function: COVE Interrupt Rebind AIA IMSIC End (FID #10)

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

Completes the rebinding process for the specified vCPU from this physical CPU and its guest interrupt files. Must be called from the same physical CPU as `sbi_covi_rebind_aia_imsic_begin()`.

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

Table 33. COVE Interrupt Rebind AIA IMSIC End

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

Chapter 11. COVE Guest Extension (EID #0x434F5647 "COVG")

The COVE Guest extension supplements the COVE Host extension, and allows TVMs to communicate with TSM. A typical use case for this extension is to relay information to the host. COVE-Guest calls cause a trap to the TSM. TSM should do any processing required and then must forward the ECALL to the host with `scause` set to ECALL, `a7` set to EID, `a6` set to FID, `a0`-`a5` set to ECALL args.

11.1. Function: COVE Guest Add MMIO Region (FID #0)

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

Marks the specified range of TVM physical address space starting at `tvm_gpa_addr` as used for emulated MMIO. Upon return, all accesses by the TVM within the range are trapped and may be emulated by the host.

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 34. COVE Guest Add MMIO Region

Error code	Description
SBI_SUCCESS	The operation was 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_FAILED	The operation failed for unknown reasons.

11.2. Function: COVE Guest Remove MMIO Region (FID #1)

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

Removes the specified range of TVM physical address space starting at `tvm_gpa_addr` from the emulated MMIO regions. Upon return, all accesses by the TVM within the range will result in a page fault.

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 35. COVE Guest Remove MMIO Region

Error code	Description
SBI_SUCCESS	The operation was 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_FAILED	The operation failed for unknown reasons.

11.3. Function: COVE Guest Share Memory Region (FID #2)

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

Initiates the assignment-change 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. This ECALL results in an exit to the TSM which enforces the security properties on the mapping and exits to the VMM host. The host then removes any confidential pages already populated in the region and inserts non-confidential pages on page-faults.

The calling TVM vCPU is considered blocked until the assignment-change is completed. attempts to run it with `sbi_covh_run_tvm_vcpu()` will fail. Any guest page faults taken by other TVM vCPUs in the invalidated pages continue to be reported to the host.

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

The possible error codes returned in sbiret.error are:

Table 36. COVE 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	The operation failed for unknown reasons.

11.4. Function: COVE Guest Unshare Memory Region (FID #3)

```
struct sbiret sbi_covg_unshare_memory_region(unsigned long tvm_gpa_addr,
```

```
                                                unsigned long region_len);
```

Initiates the assignment-change 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. This ECALL results in an exit to the TSM which enforces the security properties on the mapping and exits to the VMM host. The host then removes any non-confidential pages already populated in the region and inserts confidential pages on page-faults.

The calling TVM vCPU is considered blocked until the assignment-change is completed. Attempts to run it with `sbi_covh_run_tvm_vcpu()` will fail. Any guest page faults taken by other TVM vCPUs in the invalidated pages continue to be reported to the host.

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

Table 37. COVE Guest Unshare 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	The operation failed for unknown reasons.

11.5. Function: COVE Guest Allow External Interrupt (FID #4)

```
struct sbiret sbi_covg_allow_external_interrupt(unsigned long interrupt_id);
```

Allows injection of the specified external interrupt ID into the calling TVM vCPU. Passing an `interrupt_id` of -1 allows the injection of all external interrupts. TVM vCPUs are started with all external interrupts completely denied by default.

The possible error codes returned in sbiret.error are:

Table 38. COVE Guest Allow External Interrupt

Error code	Description
SBI_SUCCESS	The operation was completed successfully. This implies an exit to the host and a subsequent resume of execution.
SBI_ERR_INVALID_PARAM	`interrupt_id` was invalid.
SBI_ERR_FAILED	The operation failed for unknown reasons.

11.6. Function: COVE Guest Deny External Interrupt (FID #5)

```
struct sbiret sbi_covg_deny_external_interrupt(unsigned long interrupt_id);
```

Denies injection of the specified external interrupt ID into the calling TVM vCPU. Passing an `interrupt_id` of -1 denies injection of all external interrupts.

The possible error codes returned in sbiret.error are:

Table 39. COVE Guest Deny External Interrupt

Error code	Description
SBI_SUCCESS	The operation was completed successfully. This implies an exit to the host and a subsequent resume of execution.
SBI_ERR_INVALID_PARAM	`interrupt_id` was invalid.
SBI_ERR_FAILED	The operation failed for unknown reasons.

11.7. Function: COVE Guest Get Attestation Capabilities (FID #6)

```
struct sbiret sbi_covg_get_attcaps(unsigned long tvm_gpa_cap_addr,
                                   unsigned long caps_size);
```

This intrinsic is used by a TVM component to get the SBI implementation attestation capabilities. The attestation capabilities let the CoVE implementations expose which hash algorithm is being used for measurements, which evidence formats are supported. The attestation capabilities structure also contains a map of all measurement registers the TVM can extend.

Both `tvm_cap_addr` and `caps_size` must be 4kB-aligned.

```
enum HashAlgorithm {
    /* SHA-384 */
    Sha384,
    /* SHA-512 */
    Sha512
};

struct AttestationCapabilities {
    /* The TCB Secure Version Number. */
    uint64_t tcb_svn;
    /* The supported hash algorithm */
    enum HashAlgorithm hash_algorithm;
```

```
    /* The supported evidence formats. This is a bitmap */
    uint32_t evidence_formats;
    /* Number of static measurement registers */
    uint_8 static_measurements;
    /* Number of runtime measurement registers */
    uint_8 runtime_measurements;
    /* Array of all measurement register descriptors */
    MeasurementRegisterDescriptor[MAX_MEASUREMENT_REGISTERS] msmt_regs;
};
```

Table 40. COVE Guest Get Attestation Capabilities

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_caps_addr` was invalid.
SBI_ERR_INVALID_PARAM	`caps_len` was invalid, or the entire range doesn't span a `CONFIDENTIAL_MEMORY_REGION`
SBI_ERR_FAILED	The operation failed for unknown reasons.

11.8. Function: COVE Guest Measurement Extend (FID #7)

```
struct sbiret sbi_covg_measurement_extend(unsigned long tvm_gpa_buf_address,
                                          unsigned long buffer_len,
                                          Unsigned long msmt_index);
```

This intrinsic is used by a TVM component to build the chain of trust of measurement for the TVM to extend runtime measurements beyond the static measurements performed by the TSM. The measurements for each TVM always contain the same chain of TCB elements rooted in the HW RoT.

The TVM static measurements are managed by the TSM in the TVM global structure. These measurements are used in the TcbEvidenceInfo when the TVM attestation certificate is generated via sbi_covg_get_evidence.

Both `tvm_gpa_buf_addr` and `region_len` must be 4kB-aligned. msmt_index must be a valid index per the attestation capabilities reported via `sbi_covg_get_attcaps`.

Table 41. COVE Guest Measurement Extend

Error code	Description
SBI_SUCCESS	The operation completed successfully. This implies an exit to the host, and a subsequent resume of execution.

Error code	Description
SBI_ERR_INVALID_ADDRESS	`tvm_gpa_buf_addr` was invalid.
SBI_ERR_INVALID_PARAM	`region_len` was invalid, or the entire range doesn't span a `CONFIDENTIAL_MEMORY_REGION`
SBI_ERR_FAILED	The operation failed for unknown reasons.

11.9. Function: COVE Guest Get Evidence (FID #8)

```
struct sbiret sbi_covg_get_evidence(uint64_t cert_request_addr,
                                    uint64_t cert_request_size,
                                    uint64_t request_data_addr,
                                    enum EvidenceFormat evidence_format,
                                    uint64_t cert_addr_out,
                                    uint64_t cert_size);
```

If the `sbi_covg_get_attcaps` enumerates attestation services provided by the TSM, then this intrinsic is used by a TVM to get attestation evidence to report to a (remote) relying party. This may take the form of a request for an attestation certificate or a TSM-signed TVM measurement (using an attestation certificate specific to the TVM).

Get attestation evidence from a Certificate Signing Request (CSR) per datatracker.ietf.org/doc/html/rfc2986. The caller passes the CSR and its length through the first 2 arguments. The third argument is the address where the caller places a data blob that will be included in the generated certificate. Typically, this is a cryptographic nonce. The fourth argument is the evidence format: DiceTcbInfo (0), DiceMultiTcbInfo (1) or OpenDice (2). The fifth argument is the address where the generated certificate will be placed. The evidence is formatted an x.509 DiceTcbInfo certificate extension

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).

Both `cert_request_addr`, `request_data_addr` and `cert_addr_out` must be 4kB-aligned.

Table 42. COVE Guest Get Evidence

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	One of the addresses provided was invalid.
SBI_ERR_INVALID_PARAM	`cert_size` or `cert_request_size` was invalid, or the entire range doesn't span a `CONFIDENTIAL_MEMORY_REGION`
SBI_ERR_FAILED	The operation failed for unknown reasons.

Chapter 12. Summary Listing of CoVE functions

12.1. Summary of CoVE Host Extension (COVH)

sbi_covh_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_covh_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_covh_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.
sbi_covh_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_covh_local_fence) to remove any cached mappings to the memory regions that were previously selected for conversion via the sbi_covh_convert_pages.
sbi_covh_local_fence	This operation completes the TLB version tracking of pages in the region being converted 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_covh_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.

sbi_covh_finalize_tvm	This operation enables the VMM to finalize the measurement of a TVM (static). The TSM enforces that the TVM virtual harts cannot be entered unless the TVM measurement is committed via this operation.
sbi_covh_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.
sbi_covh_add_tvm_memory_region	Adds a memory region to the TVM at the specified range of guest physical address space. The memory range is confidential to the guest and may only be populated with confidential pages.
sbi_covh_add_tvm_page_table_pages	Add one or more page mappings to the G-stage translation structure for a TVM. The pages to be used for the G-stage page table structures must have been converted (and tracked) by the TSM as TEE pages; otherwise this operation will not succeed.

sbi_covh_add_tvm_measured_pages	Copies the given number of pages from non-confidential memory at `source_address` to confidential memory at `dest_address`, then measures and maps the pages at `dest_address` in the TVM physical address space at `tvm_guest_gpa`. The mapping must lie within a region of confidential memory created with `sbi_covh_add_tvm_memory_region()`. This call must not be made after calling `sbi_covh_finalize_tvm()`. 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 the whole page. The GPA at which the page is mapped is also part of the measurement operation. The measurement process is a state machine, which means that the order in which measured pages are added to the TVM also affects the attestation evidence. The VMM must faithfully reproduce the state machine for the measurement process otherwise the attestation evidence verification by the relying party will fail and the TVM will not be considered trustworthy.
sbi_covh_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_covh_add_tvm_shared_pages	Maps the given number of pages of non-confidential memory into the TVM's physical address space. The guest physical address must lie within a region of non-confidential memory previously defined by the TVM via the guest interface to the TSM.
sbi_covh_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 one 4KB page. The number of pages are enumerated via the tsm_info call.

sbi_covh_run_tvm_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.
sbi_covh_tvm_fence	Initiates a TLB invalidation sequence for all pages that have been invalidated in the given TVM's address space since the previous call to `sbi_covh_tvm_fence()`. The TLB invalidation sequence is completed when all vCPUs in the TVM that were running before the call to `sbi_covh_tvm_fence()` have taken a trap into the TSM, which the host can cause by sending an IPI to the physical CPUs on which the TVM's vCPUs are running.
sbi_covh_tvm_invalidate_pages	Invalidates the pages in the specified range of guest physical address space and thus marks the pages as blocked from any further TVM accesses. Guest page faults taken by the TVM on invalidated pages continue to be reported to the host. The page remains invalid until the mapping is validated (marked present), removed, or becomes part of a huge page by promotion/demotion operation.
sbi_covh_tvm_validate_pages	Marks the invalidated pages in the specified range of guest physical address space as present. This ECALL may also be used to revert an in-progress page removal or huge page promotion/demotion sequence.
sbi_covh_tvm_remove_pages	Removes mappings for invalidated pages in the specified range of guest physical address space. The range to be unmapped must already have been invalidated and fenced, and must lie within a removable region of the guest's physical address space.
sbi_covh_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 invalid and fenced before the page mapping can be relocated. This interface specification is TBD.

sbi_covh_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 invalidated before the promote operation can succeed. The VMM may reclaim the freed G-stage page table page if the operation succeeds. This interface specification is TBD.
sbi_covh_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 invalidated before the operation can succeed. The VMM must provide a free TEE-capable page to the TSM to use as a new G-stage page table in the fragmented mapping. This interface specification is TBD.

12.2. Summary of CoVE Interrupt Extension(COVI)

sbi_covi_init_tvm_aia	This intrinsic is supported by the TSM to configure AIA virtualization for the TVM
sbi_covi_set_tvm_aia_cpu_imsic_addr	Set TVM CPU AIA address
sbi_covi_convert_tvm_aia_imsic	Convert TVM GPA AIA address to confidential
sbi_covi_reclaim_tvm_aia_imsic	Reclaim TVM GPA AIA address from confidential
sbi_covi_bind_aia_imsic	Binds a TVM vCPU to the current physical CPU using the confidential guest interrupt file.
sbi_covi_unbind_aia_imsic_begin	Begins the unbind process for the specified vCPU from its guest interrupt file.
sbi_covi_unbind_aia_imsic_end	Completes the unbind process for the specified vCPU from its guest interrupt files after a TLB flush sequence for the TVM has been completed.
sbi_covi_inject_tvm_cpu	Injects an external interrupt with the given interrupt_id into the specified vCPU.
sbi_covi_rebind_aia_imsic_begin	Begins the rebinding process for the specified vCPU to the current physical CPU and the specified confidential guest interrupt file. The host must complete a TLB invalidation sequence for the TVM before cloning old interrupt file state using `sbi_covi_rebind_aia_imsic_clone()`.
sbi_covi_rebind_aia_imsic_clone	Clones the old guest interrupt file of the specified vCPU. Caller must make sure to invoke this from old physical CPU. The guest interrupt file after this is free to be reclaimed or bound to another vCPU.

sbi_covi_rebind_aia_imsic_end	Completes the rebind process for the specified vCPU from this physical CPU and its guest interrupt files. Must be called from the same physical CPU as `sbi_covi_rebind_aia_imsic_begin()`.

12.3. Summary of CoVE Guest Extension (COVG)

sbi_covg_add_mmio_region	Marks the specified range of TVM physical address space starting at `tvm_gpa_addr` as used for emulated MMIO. Upon return, all accesses by the TVM within the range are trapped and may be emulated by the host.
sbi_covg_remove_mmio_region	Removes the specified range of TVM physical address space starting at `tvm_gpa_addr` from the emulated MMIO regions. Upon return, all accesses by the TVM within the range will result in a page fault.
sbi_covg_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 G-stage page table and not tracked as a TEE page. The VMM owns the process of reclaiming the page.
sbi_covg_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, G-stage page table pages, and TVM pages.
sbi_covg_allow_external_interrupt	Allows injection of the specified external interrupt ID into the calling TVM vCPU. Passing an `interrupt_id` of -1 allows injection of all external interrupts. TVM vCPUs are started with injection of external interrupts completely disabled by default.
sbi_covg_deny_external_interrupt	Denies injection of the specified external interrupt ID into the calling TVM vCPU. Passing an `interrupt_id` of -1 denies injection of all external interrupts.

sbi_covg_get_attcaps	This intrinsic is used by a TVM to get attestation capabilities supported by the TSM. the capabilities enumerated are then used to extend measurements and/or get evidence to support attestation.
sbi_covg_measurement_extend	This intrinsic is used by a TVM component to build the chain of trust of measurement 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 sbi_covg_get_evidence. This interface specification is TBD.
sbi_covg_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_covg_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_covg_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 [R5] convention). The TSM-driver initializes TSM 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 G-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 CoVE 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)

Bibliography

- [R0] RISC-V Privileged specification github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf
- [R1] IETF RFC 9334 Remote ATtestation procedureS (RATS) Architecture datatracker.ietf.org/doc/rfc9334/
- [R2] TCG DICE Attestation Architecture, Version 1.00 Revision 0.23 trustedcomputinggroup.org/resource/dice-attestation-architecture/
- [R3] DMTF DSP0274 Security Protocol and Data Model (SPDM) Specification, Version 1.2.1 www.dmtf.org/dsp/DSP0274
- [R4] TCG Reference Integrity Manifest (RIM) Information Model trustedcomputinggroup.org/resource/tcg-reference-integrity-manifest-rim-information-model/
- [R5] RISC-V Supervisor Binary Interface riscv-non-isa/riscv-sbi-doc
- [R6] RISC-V Debug Specification Standard github.com/riscv/riscv-debug-spec/blob/master/riscv-debug-stable.pdf
- [R7] RISC-V Zero-Trust Platform Security Model docs.google.com/document/d/1TRHhsGiB5W4K8M7I4e-f40mOPErTb9sv/edit#heading=h.gjdgxs
- [R8] Trusted Computing Group (TCG) Glossary, Version 1.1 Revision 1.0 trustedcomputinggroup.org/resource/tcg-glossary/
- [R9] The RISC-V Advanced Interrupt Architecture Document v0.2.1-draft https://github.com/riscv/riscv-aia/releases
- [R10] The RISC-V Nested Acceleration ("NACL") extension TODO Add spec link once available[TODO]