



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

RISC-V AP-TEE Task Group

Version 0.3, 1/2024: This document is in development. Assume everything can change. See
<http://riscv.org/spec-state> for details.

Table of Contents

| | |
|---|----|
| Preamble | 1 |
| Copyright and license information | 2 |
| Contributors | 3 |
| 1. Introduction | 4 |
| 2. Notation | 5 |
| 3. Glossary | 6 |
| 4. Architecture Overview and Threat Model | 10 |
| 4.1. Adversary Model | 13 |
| 4.2. Threat Model | 14 |
| 4.3. Scope | 15 |
| 4.4. TVM Security Requirements to address threat model | 16 |
| 5. Reference Architecture Details | 25 |
| 5.1. CoVE Memory Isolation | 25 |
| 5.1.1. Address Translation/Page Walk | 26 |
| 5.1.2. Management of isolation for Confidential Physical Memory | 27 |
| 5.1.3. Handling Implicit & Explicit Memory Accesses | 28 |
| 5.1.4. Cached translations/TLB management | 28 |
| 5.2. TSM initialization | 28 |
| 5.3. TSM operation and properties | 29 |
| 5.4. TSM and TVM Isolation | 33 |
| 5.5. TVM Execution | 34 |
| 5.6. Debug and Performance Monitoring | 34 |
| 6. TVM Attestation and Measurements | 36 |
| 6.1. Measurements | 36 |
| 6.1.1. TCB Elements | 36 |
| 6.1.2. TVM Measurements | 37 |
| Initial Measurements | 37 |
| Runtime Measurements | 38 |
| 6.2. Attestation | 38 |
| 6.2.1. Model | 38 |
| 6.2.2. Layered Evidence | 39 |
| 6.2.3. Evidence Format | 41 |
| Platform Token | 43 |
| EAT Profile Claim | 43 |
| Platform Public Key Claim | 44 |
| Platform Manufacturer Identifier Claim | 44 |
| Platform State Claim | 44 |
| Platform Software Components Claim | 45 |

| | |
|---|----|
| TSM Token | 46 |
| TSM Public Key Claim | 46 |
| TSM Software Components Claim | 46 |
| TVM Token | 47 |
| TVM Challenge Claim | 47 |
| TVM Identity Claim | 47 |
| TVM Public Key Claim | 48 |
| TVM Initial Measurements Claim | 48 |
| TVM Runtime Measurements Claim | 49 |
| 6.2.4. Evidence Generation | 49 |
| CBOR Attestation Certificate | 50 |
| X.509 Attestation Certificate | 51 |
| 7. TVM Lifecycle | 53 |
| 7.1. TVM build and initialization | 53 |
| 7.2. TVM execution | 54 |
| 7.3. TVM memory management | 55 |
| 7.3.1. Security requirements for TVM memory mappings | 55 |
| 7.3.2. Information tracked per physical page | 56 |
| 7.3.3. Page walk and Translation caching considerations | 57 |
| 7.3.4. Page conversion | 57 |
| 7.3.5. Global and per-TVM TLB management | 58 |
| 7.3.6. Page Mapping Page Assignment | 60 |
| 7.3.7. Measured page assignment into a TVM memory map | 60 |
| 7.4. TVM Interrupt Handling | 61 |
| 7.4.1. TVM timers | 61 |
| 7.4.2. TVM external interrupts | 61 |
| 7.4.3. Paravirtualized I/O | 64 |
| 7.5. TVM shutdown | 65 |
| 7.6. RAS interaction | 65 |
| 8. Confidential VM Extension (CoVE) SBI extension proposal | 66 |
| 8.1. TEEI - COVH runtime interface | 66 |
| 8.1.1. Operational model for the CoVE Host Extension | 66 |
| Platform TSM detection and capability enumeration | 67 |
| TVM creation | 67 |
| TVM memory management | 67 |
| Converting non-confidential memory to confidential memory | 67 |
| Defining confidential memory regions | 68 |
| Donating confidential pages for the TVM page-table pool | 68 |
| Mapping TVM code and data payload to confidential TVM-pages | 68 |
| VCPU shared state | 68 |
| VCPU creation | 70 |

| | |
|--|----|
| TVM execution | 70 |
| Mapping confidential demand-zero pages and non-confidential shared pages | 71 |
| Handling MMIO faults | 71 |
| Handling virtual instructions | 71 |
| Management of secure interrupts | 71 |
| TVM teardown | 71 |
| 8.1.2. Operational model for the CoVE Guest Extension | 72 |
| TVM-defined MMIO regions | 72 |
| TVM-defined Shared memory regions | 72 |
| 9. COVE Host Extension (EID #0x434F5648 "COVH") | 76 |
| 9.1. Listing of common enums | 76 |
| 9.2. Function: COVE Host Get TSM Info (FID #0) | 76 |
| 9.3. Function: COVE Host Convert Pages (FID #1) | 78 |
| 9.4. Function: COVE Host Reclaim Pages (FID #2) | 78 |
| 9.5. Function: COVE Host Initiate Global Fence (FID #3) | 78 |
| 9.6. Function: COVE Host Local Fence (FID #4) | 79 |
| 9.7. Function: COVE Host Create TVM (FID #5) | 79 |
| 9.8. Function: COVE Host Finalize TVM (FID #6) | 80 |
| 9.9. Function: COVE Host Destroy TVM (FID #7) | 82 |
| 9.10. Function: COVE Host Add TVM Memory Region (FID #8) | 82 |
| 9.11. Function: COVE Host Add TVM Page Table Pages (FID #9) | 83 |
| 9.12. Function: COVE Host Add TVM Measured Pages (FID #10) | 83 |
| 9.13. Function: COVE Host Add TVM Zero Pages (FID #11) | 84 |
| 9.14. Function: COVE Host Add TVM Shared Pages (FID #12) | 85 |
| 9.15. Function: COVE Host Create TVM VCPU (FID #13) | 85 |
| 9.16. Function: COVE Host Run TVM VCPU (FID #14) | 86 |
| 9.17. Function: COVE Host Initiate TVM Fence (FID #15) | 88 |
| 9.18. Function: COVE Host TVM Invalidate Pages (FID #16) | 89 |
| 9.19. Function: COVE Host TVM Validate Pages (FID #17) | 89 |
| 9.20. Function: COVE Host TVM Remove Pages (FID #18) | 90 |
| 10. COVE Interrupt Extension (EID #0x434F5649 "COVI") | 91 |
| 10.1. Function: COVE Interrupt Init TVM AIA (FID #0) | 91 |
| 10.2. Function: COVE Interrupt Set TVM AIA CPU IMSIC Addr (FID #1) | 92 |
| 10.3. Function: COVE Interrupt Convert AIA IMSIC (FID #2) | 93 |
| 10.4. Function: COVE Interrupt Reclaim TVM AIA IMSIC (FID #3) | 93 |
| 10.5. Function: COVE Interrupt Bind AIA IMSIC (FID #4) | 93 |
| 10.6. Function: COVE Interrupt Unbind AIA IMSIC Begin (FID #5) | 94 |
| 10.7. Function: COVE Interrupt Unbind AIA IMSIC End (FID #6) | 94 |
| 10.8. Function: COVE Interrupt Inject TVM CPU (FID #7) | 95 |
| 10.9. Function: COVE Interrupt Rebind AIA IMSIC Begin (FID #8) | 95 |
| 10.10. Function: COVE Interrupt Rebind AIA IMSIC Clone (FID #9) | 96 |

| | |
|--|-----|
| 10.11. Function: COVE Interrupt Rebind AIA IMSIC End (FID #10) | 96 |
| 11. COVE Guest Extension (EID #0x434F5647 "COVG") | 98 |
| 11.1. Function: COVE Guest Add MMIO Region (FID #0)..... | 98 |
| 11.2. Function: COVE Guest Remove MMIO Region (FID #1) | 98 |
| 11.3. Function: COVE Guest Share Memory Region (FID #2) | 99 |
| 11.4. Function: COVE Guest Unshare Memory Region (FID #3) | 99 |
| 11.5. Function: COVE Guest Allow External Interrupt (FID #4) | 100 |
| 11.6. Function: COVE Guest Deny External Interrupt (FID #5) | 101 |
| 11.7. Function: COVE Guest Get Attestation Capabilities (FID #6) | 101 |
| 11.8. Function: COVE Guest Extend Measurement (FID #7)..... | 103 |
| 11.9. Function: COVE Guest Get Evidence (FID #8) | 104 |
| 11.10. Function: COVE Guest Read Measurement (FID #9) | 105 |
| 12. Summary Listing of CoVE functions | 106 |
| 12.1. Summary of CoVE Host Extension (COVH)..... | 106 |
| 12.2. Summary of CoVE Interrupt Extension(COVI) | 110 |
| 12.3. Summary of CoVE Guest Extension (COVG)..... | 111 |
| 13. Appendix A: THCS and VHCS | 113 |
| 14. Appendix B: Interrupt Handling | 115 |
| Bibliography | 117 |

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

This specification is licensed under the Creative Commons Attribution 4.0 International License (CC-BY 4.0). The full license text is available at creativecommons.org/licenses/by/4.0/.

Copyright 2022 by RISC-V International.

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, Nicholas Wood, Nick Kossifidis, Rajnesh Kanwal, Ravi Sahita (Editor), Rob Bradford, Samuel Ortiz, Vedvyas Shanbhogue, Yann Loisel

Chapter 1. Introduction

This document describes the Confidential VM Extension (CoVE) interface for a scalable Trusted Execution Environment(TEE) for hardware virtual-machine-based 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. |

| | |
|-------------------------------------|--|
| RISC-V Supervisor Domains | <p>RISC-V privileged architecture [0] defines the S-mode for execution of supervisor software. S-mode software may optionally enable Hypervisor extension to host virtual machines. Typically, there is a single supervisor domain of execution with access to all physical memory.</p> <p>Supervisor Domains [R20] is a RISC-V privileged architecture extension to support physical address space (memory and devices) isolation for more than one supervisor domain. Supervisor domains enable the reduction of the supervisor Trusted Computing Base (TCB), with differential access to memory and other platform resources e.g. as used in this Confidential VM Extension (CoVE) spec.</p> |
| Confidential Computing | The protection of data in use by performing computation in a Hardware-based and Attestable Trusted Execution Environment. |
| Confidential VM Extension (CoVE) | <p>The set of non-ISA RISC-V ABI extensions defined in this specification that enables confidential computing on RISC-V platforms. In some deployment models, the CoVE ABI leverages the RISC-V ISA extensions specified in the RISC-V Supervisor Domains specification [R20]. CoVE is a Trusted Execution Environment ABI for Application Processors. A supervisor domain that provides HW-isolation for workload data 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.</p> |
| 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 [7], and may take on the role of a Device Identification Composition Engine (DICE) as defined in [R2]. |
| Confidential memory | Memory that is subject to access-control, confidentiality and integrity mechanisms per the threat model for use in the CoVE system. Confidential memory may also be used by non-TCB/ hosting software with appropriate TCB controls on the configuration, e.g a separate key used for TCB and non-TCB elements. |
| 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). A monolith Supervisor Domain exists with the host OS/VMM including device drivers and services forming the TCB. 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 software and hardware TCB.

This specification describes the CoVE architecture which enables a new class of hardware-attested trusted execution environment called TEE Virtual Machines (TVMs). The TVMs are supported by a hardware-rooted, attestable TCB and its execution state and memory 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 describes an isolated (Confidential) Supervisor Domain to enforce TCB and confidentiality properties, while using an isolated (Hosting) Supervisor Domain for the host domain, thus maintaining the OS/VMMs role as the resource manager (for both legacy VMs and TVMs). The resources managed by the hosting supervisor domain (OS/VMM) include memory, CPU, I/O resources and platform capabilities to host the TVM workload. The terms hosting supervisor domain and OS/VMM are used interchangeably in this specification.

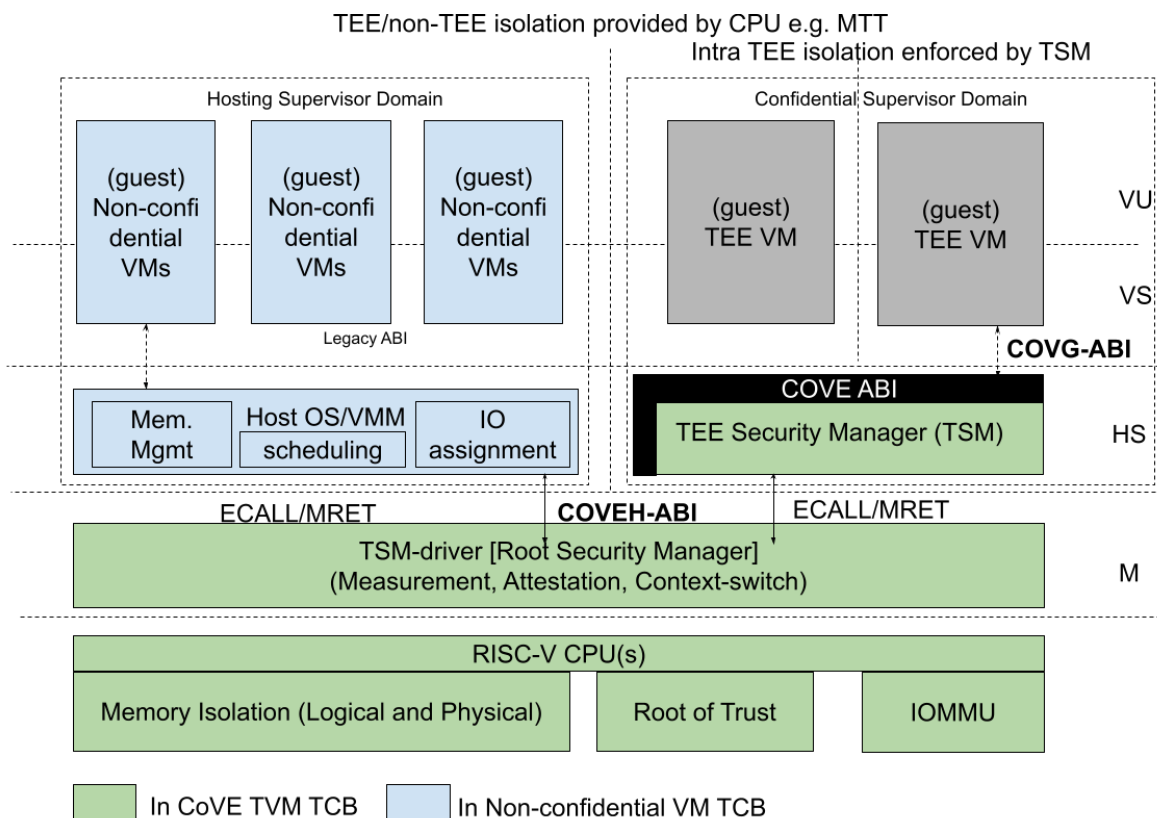


Figure 1: TCB Isolation for VM workloads using Supervisor Domains

As shown in figure 1, the Confidential Supervisor Domain is managed by software that operates in HS-mode and manages resources granted to it by the Hosting Supervisor Domain Manager (the OS/VMM). The Confidential Supervisor Domain Manager is called the "**TEE Security Manager**" or (**TSM**) - it acts as the trusted intermediary between TEE and non-TEE workloads on the same platform. The TSM should have a minimal HW-attested footprint. The TCB (which includes the TSM and HW) enforces strict confidentiality and integrity security properties for workloads in this supervisor domain. The Root Security Manager is an M-mode software module (called the "**TSM-driver**") which isolates the Confidential Supervisor Domain from all other Supervisor domains and other platform components (non-confidential and confidential). The responsibility of the TSM is to enforce the security objectives accorded to TEE workloads assigned to that supervisor domain. The VMM is expected to continue to manage the security for non-confidential workloads, and importantly the resource-assignment and scheduling management functions for all workloads (confidential and non-confidential).

In this scheme, compute resources like memory start off as traditional untrusted resources owned by the non-confidential/hosting supervisor domain, and are expected to be donated/transitioned to the confidential supervisor domain via ABI supported by the TSM. Once the conversion process is complete, confidential memory may be assigned to one or more TVMs by the TSM. A converted confidential resource may be freely assigned to another TVM within the same supervisor domain when it is no longer in use. However, an unused confidential resource must be explicitly reclaimed for use in the non-confidential supervisor domain (such memory conversions are tracked and enforced via the TSM in the owning supervisor domain to enforce isolation properties). The hosting supervisor domain may use the reclaimed memory for itself or for non-confidential VMs.

Each TVMs 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 ABI that are accessed by the OS/VMM in the Hosting Supervisor Domain Manager via a **Trusted Execution Environment Interface (TEEI)**. This ABI is invoked via a **TSM-Driver** that operates in the M-mode privilege level on the hart. The TSM itself operates in HS-mode (priv=01; V=0) of the hart 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 [0], this specification minimizes ISA changes to introduce a scalable architecture for hosting TEE workloads. More than one confidential supervisor domains may be hosted by the TSM-driver. Similarly, more than one TVMs may be hosted by the host OS/VMM via confidential supervisor domains. Each TVM may consist of the guest firmware, a guest OS and applications. The software components included in the TVM are implementation specific.

As shown in figure 1, the M-mode firmware TSM-driver is in the TCB of all Supervisor domains and hence in the TCB for 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 **Confidential memory** as memory that is subject to access-control, confidentiality and integrity suitable per threat model for CoVE components. For Supervisor Domain memory isolation, the Smmmtt extension may be used for deployment scenarios where the TSM is at the same privilege level as

the VMM. The TEEI operations for memory management are described in detail below.

- Context switching of the hart supervisor domain id and MTT on TEECALL/TEERET synchronous transitions or asynchronous transitions (due to interrupts).
- 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 confidential memory assigned to TVMs. The TSM is designed to be portable across RISC-V platforms (that support CoVE) 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 enables the TVM workload to request attestation functions, memory management functions or paravirtualized IO.

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 supervisor domain that the TSM occupies - 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 (e.g. using Smmtt), 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 (for the TSM ABIs) is enforced in hardware via the maintenance of the execution context (ASID, VMID and SDID) maintained per hart. This context is enabled per-hart via the TEECALL interface to context switch into the confidential supervisor domain context via the TSM-driver and disabled via the TEERET interface to context restore to the hosting supervisor domain. Access to TEE-assigned memory is allowed for the hart when the access is permitted as per the active permissions enforced by the MMU for the supervisor domain active on the hart (enforced through Sv and Smmtt for CoVE). This per-hart execution context is used by the processor to enforce access-control properties on memory accessed by TEE workloads managed by the TSM. The details of the supervisor domain access protection is specified in the Smmtt specification [\[R20\]](#).

TSM functionality should be explicitly limited to support only the 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 G-stage 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 runtime may be used; whereas in the VM TEE case, an enlightened guest OS is expected in the TVM TCB. Other SW 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 the CoVE ABI.

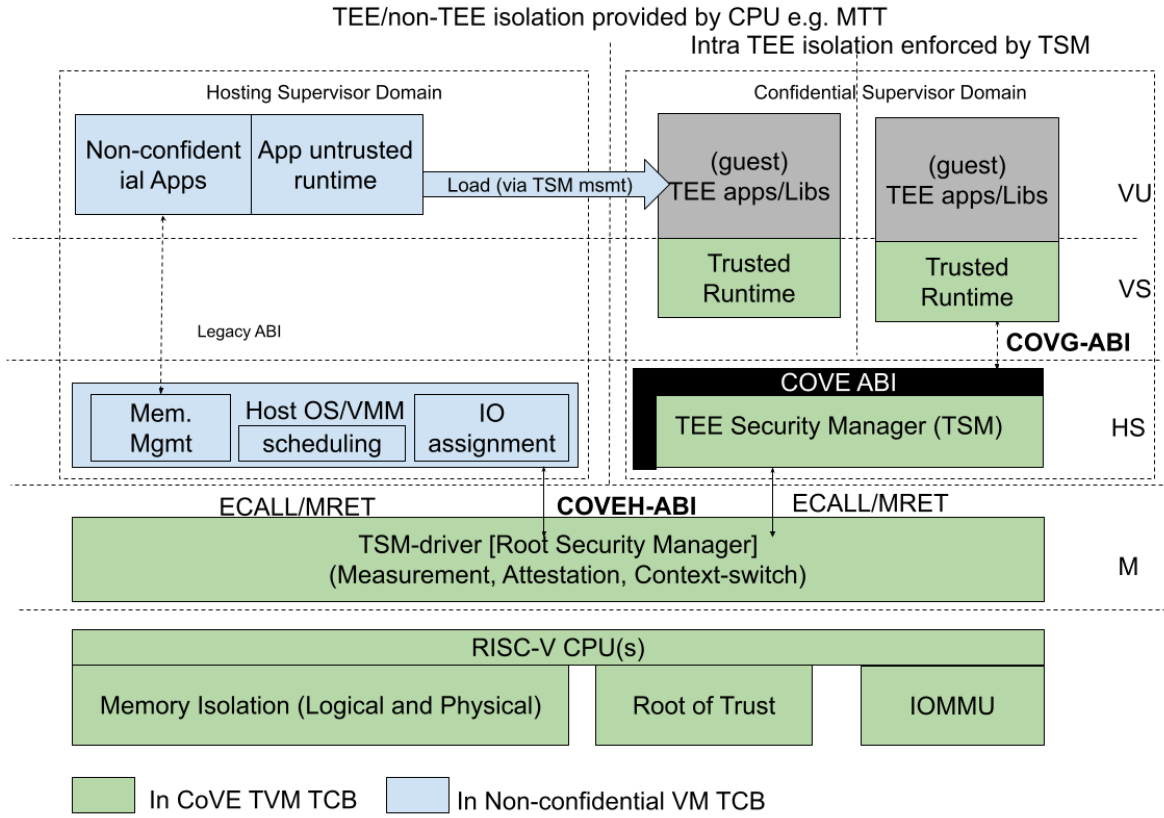


Figure 2: TEE TCB for application workloads (hosted via a TVM)

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 ABI for Confidential VMs 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. With user space I/O an Unprivileged software adversary may also have ability to submit requests to I/O devices made available by system software for U-mode access.

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, IOMMU(s) 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, IOMMU(s), IO devices and platform configuration registers (e.g., address range decoders, SoC fabric configuration, etc.).

Non-invasive Hardware adversary - This includes adversaries that can use non-invasive (requiring no physical change to the target hardware) attacks such as bus interposers to snoop on memory and/or device interfaces, voltage and/or clock glitching, observe electromagnetic and other radiation, analyze power usage through instrumentation/tapping of power rails, etc. which may then give the adversary the ability to tamper with data in use.

Invasive Hardware adversary - This includes adversaries that can use invasive 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 TCB components 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 non-TCB 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 threat model is not an exhaustive list and will be updated (via the RISC-V Security Model specification [\[R21\]](#)) on a regular basis as attacks evolve.

4.3. Scope

This specification describes the threats that a system implementing CoVE should address, however, it does not prescribe the scope of mitigations; instead it focusses on mitigations enabled via the COVH/G interface and the use of the RISC-V ISA (and extensions such as Smmmtt). This specification also provides recommendations that implementations of this reference CoVE architecture must address per their chosen scope of adversaries from the list of adversaries discussed above, and what performance/security trade-offs they accept. For threats from any adversaries, implementations may choose to mitigate threats using additional platform capabilities as needed. For all scenarios though, denial of service by TVMs must be prevented. At the same time, denial of service by non-TCB software (e.g. in a hosting supervisor domain) 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 normative or non-normative Reference |
|--------------------------|---|------------------|---|---|--|
| Memory Assignment | Ability to make memory confidential or non-confidential | Required | MMU, MPU, MTT | Confidential memory should be dynamically allocated/unallocated as required | RISC-V Priv. ISA, Supervisor Domains (Sdid, Smmmtt) |
| TEE CPU State Protection | State Isolation | Required | Priv. levels (M,S,HS, U) and Execution context (ASID, VMID, SDID) | Prevent non-TCB components from arbitrarily accessing/modifying TEE CPU state | Priv ISA w/ virtual memory system, Supervisor Domains (Sdid, Smmmtt) |
| Memory Confidentiality | Memory isolation (read) | Required | cryptography and/or MMU, MPU, MTT extension | Prevent non-TCB components from reading TEE memory | Priv ISA w/ virtual memory system, Supervisor Domains (Smmmtt) |
| Memory Confidentiality | Cipher text read prevention | Required | cryptography and/or MMU, MPU, MTT extension | Prevent non-TCB components from accessing encrypted TEE memory | Svpams (Supervisor Domains) |

| | | | | | |
|------------------------|-------------------------------------|-------------------------|---|---|-----------------------------|
| Memory Confidentiality | TEE encryption | Implementation-specific | Cryptography and/or MMU, MPU, MTT extension | TEE memory confidentiality against logical attacks via MMU; additionally address physical attacks via cryptography. If cryptography used, TEE should have a unique encryption key; each TEE VM may also have one or more unique keys. Also see related requirements around ciphertext disclosure and memory integrity | Svpams (Supervisor Domains) |
| Memory Confidentiality | Memory encryption strength | Implementation-specific | cryptography | Encryption algorithm and key strength | Security Model |
| Memory Confidentiality | Number of encryption keys | Implementation-specific | cryptography | Number of TEE keys supported | Security Model |
| Memory Integrity | Memory integrity against SW attacks | Required | MMU, MPU, MTT | Prevent SW attacks such as remapping aliasing replay corruption etc. | CoVE ABI |

| | | | | | |
|------------------|---|-------------------------|---|---|---|
| Memory Integrity | Memory integrity against HW attacks | Implementation specific | cryptography and/or MMU, MPU, MTT extension | Prevent HW attacks DRAM-bus attacks and physical attacks that replace TEE memory with tampered / old data | Security Model |
| Memory Integrity | Memory isolation (Write exec) | Required | cryptography and/or MMU, MPU, MTT | Prevent TEE from executing from normal memory; Enforce integrity of TEE data on writes | Supervisor Domains (Sdid, Smmmtt) |
| Memory Integrity | Rowhammer attack prevention | Implementation specific | cryptography and/or memory-specific extension | Prevent non-TCB components from flipping bits of TEE memory | Security Model |
| Shared Memory | TEE controls data shared with non-TCB components | Required | cryptography and/or MMU, MPU, MTT | Prevent non-TCB code from exfiltrating information without TEE consent/opt-in | Supervisor Domains (Sdid, Smmmtt) |
| Shared Memory | TEE controls data shared with another TEE | Implementation specific | cryptography and/or MMU, MPU, MTT | Ability to securely share memory with another TEE | Supervisor Domains (Sdid, Smmmtt, Svpams) |
| I/O Protection | DMA protection from non-TCB-admitted devices | Required | DMA access-control e.g. IOPMP, IOMTT, IOMMU | Prevent non-TCB peripheral devices from accessing TEE memory | See CoVE-IO [R22] , IOMMU, Supervisor Domains (IOMTT) |
| I/O Protection | Trusted I/O from devices admitted into the TCB of a TVM | Implementation specific | Device attestation, Link protection, IOMMU | Admission control to bind devices to TEEs | See CoVE-IO [R22] , IOMMU, Supervisor Domains (IOMTT) |

| | | | | | |
|-----------------|--|-------------------------|--|---|------------------------------------|
| Interrupts | Trusted (no spoofing/tampering/dropped) Interrupts | Required | Secure interrupt files, MMU, MPU, MTT | Prevent IRQ injections that violate priority or masking | Supervisor Domains (Smsdia) w/ AIA |
| Secure Timetamp | Trusted timestamps | Required | Confidential supervisor domain qualifier for CSR accesses | Ensure TEE have consistent timestamp view | Supervisor Domains (Sdid) |
| Debug & Profile | Trusted performance monitoring unit data | Required | Confidential supervisor domain context switch of perf. mon. counters | Ensure TEEs get correct PMU info; prevent data leakage due to PMU information (fingerprint attacks) | Supervisor Domains (Secure Debug) |
| Debug & Profile | Secure External Debug support | Required | Confidential supervisor domain qualifier for External debug controls | Support debug trigger registers for TVM | Supervisor Domains (Secure Debug) |
| Debug & Profile | Authenticated debug (Production device) | Required | Authorize debug via TEE RoT | Ensure hardware debug prob (e.g., JTAG SWD) is disabled in production | Supervisor Domains (Secure Debug) |
| Availability | TVM DoS Protection | Required | VMM retains ability to interrupt TVM | Prevent TVM from refusing to exit | Supervisor Domains |
| Availability | VMM DoS Protection | Implementation-specific | Not in scope for CoVE | Prevent non-TCB hosting components from denying service to a TVM | Not in scope |

| | | | | | |
|--------------|--|----------|------------------------------------|--|------------------------------------|
| Side Channel | Address mapping caches (controlled side channel) | Required | Supervisor domain Id, MMU/MPU, MTT | HW/SW TCB should use tagging/ partitioning/ flushing techniques to address those types of side channels due to temporal/spatial shared resources | Supervisor Domains, Security Model |
|--------------|--|----------|------------------------------------|--|------------------------------------|

| | | | | | |
|--------------|--|-------------------------|--|---|--|
| Side Channel | Transient-execution attack (TEA) side channels | Implementation-specific | <p>* Bounds check bypass TEA and variants - should be addressed by TVM software using apropos synchronization. TCB SW should use synchronization to isolate TCB code from non-TCB code. *</p> <p>Branch target injection TEA and variants - should be addressed by TCB SW via flushing across privilege boundaries to remove untrusted state injected by non-TCB software *</p> <p>Speculative store bypass TEA and variants - should be addressed by TCB HW via synchronization/barriers to prevent speculative execution of memory reads which may allow unauthorized disclosure of information.</p> | Implementations should mitigate attacks such as these spectre variants (In practice, it is difficult to defend against such attacks in advance) | Supervisor Domain Id, Addtl. Recommendations in Security Model |
|--------------|--|-------------------------|--|---|--|

| | | | | | |
|--------------------------|---|-------------------------|--|---|--------------------------|
| Side Channel | Control channels, single-step/zero-step attacks | Required | leverage HW/SW TCB mechanisms to enforce restrictions on single-stepping or zero-stepping via use of state flushing/barriers, entropy defenses and detection mechanisms. | Prevent interrupt/exception injection (combined with cache side channel to leak sensitive data) | Security Model |
| Side Channel | Architectural cache side channel | Implementation-specific | cache partitioning-based defenses | Prevent shared resource contention, e.g. attacks such as prime probe | Security Model |
| Side Channel | Architectural timing side channel | Implementation-specific | data independent execution latency (DIEL) operations, uArch state flushing | Leveraging data dependency timing channels | Security Model |
| 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 | CoVE ABI, Security Model |
| Attestation | Remote attestation | Required | HW-RoT-rooted PKI (trust assertions) via Internet | Prevent fake hardware and software TCB; Prevent non-TCB hardware debugging in production. | CoVE ABI, Security Model |
| Attestation | Mutual attestation | Implementation-specific | S/U mode | Attestation to another TEE on the same platform | CoVE ABI, Security Model |

| | | | | | |
|-------------|--|-------------------------|--|---|---------------------------------|
| Attestation | Remote mutual attestation | Required | Internet | Attestation to a relying party on a different platform. Requires provisioning of the TEEs to act as delegated relying parties | CoVE ABI, Security Model |
| Attestation | Local attestation | Implementation-specific | Sealing | Verification of attestation by TCB | Future CoVE ABI, Security Model |
| Attestation | TCB versioning (and updates) | Required | Mutable firmware where TVM has to opt-in at startup if TCB updates are allowed while the TVM is executing - HW TCB then enforces lower TCB elements are updatable (with apropos controls like security version enforced) to enforce the opt-in policy. | Allow TCB updates - Prevent TCB rollback | CoVE ABI, Security Model |
| Attestation | TCB composition for confidential computing | Required | Single root of trust for measurement and reporting | Malicious components introduced in the TCB | CoVE ABI, Security Model |

| | | | | | |
|----------------------|--|-------------------------|---|---|--------------------------|
| 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 | CoVE ABI, Security Model |
| Attestation | TCB transparency (and auditability) | Implementation-specific | Mutable firmware | TCB elements reviewable | CoVE ABI, Security Model |
| Attestation | Sealing | Implementation-specific | HW Rot sealing keys per TVM | Binding of secrets to TEEs | CoVE ABI, Security Model |
| Operational Features | TVM Migration | Implementation-specific | Secure migration of TEEs | Malicious host tampers with TVM assets during migration | Future CoVE ABI |
| Operational Features | TVM Nesting | Implementation-specific | Nested TEE Workloads | Malicious host tampers with nested VMM policies | Future CoVE ABI |
| Operational Features | Memory introspection | Implementation-specific | Interoperability with security features for TVM workload | Unauthorised security TVM | Future CoVE ABI |
| Operational Features | QoS interoperability | Implementation-specific | Interoperability with QoS features for TVM workload | Malicious host uses QoS capabilities as a side-channel | Security Model |
| 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 | Security Model |

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 is orchestrated by the TSM-driver and the TSM in two phases: the conversion of memory to confidential memory and the assignment of confidential memory (alongwith the enforcement of properties on use) to TVMs. To enforce isolation across Host and Confidential supervisor domains, CoVE requires isolation of physical memory (that supports paging when enabled). There are two deployment models possible here. CoVE ABI is equally applicable for both modes - this specification focusses on the second deployment model (b) where a peer supervisor domain is used to host confidential workloads.

1. When the TSM is the only root HS mode component on the platform, the G-stage page table can be used to enforce isolation between confidential TVMs and ordinary VMs. In this model the host VMM must execute in the de-privileged VS mode and the TSM must provide nested virtualization of the H-extension controls. This model may be suitable for client/embedded systems.
2. The TSM operates in S/HS mode as a peer supervisor domain manager to the hosting supervisor domain which operates in S/HS mode as well. This model uses the MTT along with G-stage PT for confidential TVM isolation (where 1st stage PT is used by the Guest OS normally). The MTT is used to assign physical memory to the Confidential supervisor domain called **Confidential** memory and memory accessible to the hosting supervisor domain called **Non-Confidential**. MTT allows dynamic programming of the per-domain access permissions. A TVM and/or TSM needs to access both types of memory:
 - Confidential memory - used for TVM code, data
 - Non-confidential memory - used for communication between TVM and the non-TCB host software and/or non-TCB IO devices.

The TSM COVH ABI provides interfaces to the OS/VMM to convert / donate memory from the hosting supervisor domain to the confidential supervisor domain. Similarly, a separate ABI intrinsic is used to reclaim memory back from the confidential supervisor domain to the hosting supervisor domain. Once physical memory is converted to confidential - it is accessible only to the confidential supervisor domain. By default, TVM memory is assigned by the TSM (which operates in the confidential supervisor domain context) from confidential physical memory regions. Note that a TVM may be assigned non-confidential (shared) memory regions as well explicitly under the TVMs control. The TSM manages the type and accessibility of all memory assigned to the TVM, to mitigate attacks from non-TCB software. The TSM enforces isolation between TVMs by using the G-stage page table.

- Hart operating with the confidential supervisor domain context has MTT permissions to access Confidential and Non-confidential memory

- Hart not operating in a Confidential supervisor domain has access permissions only for Non-confidential memory

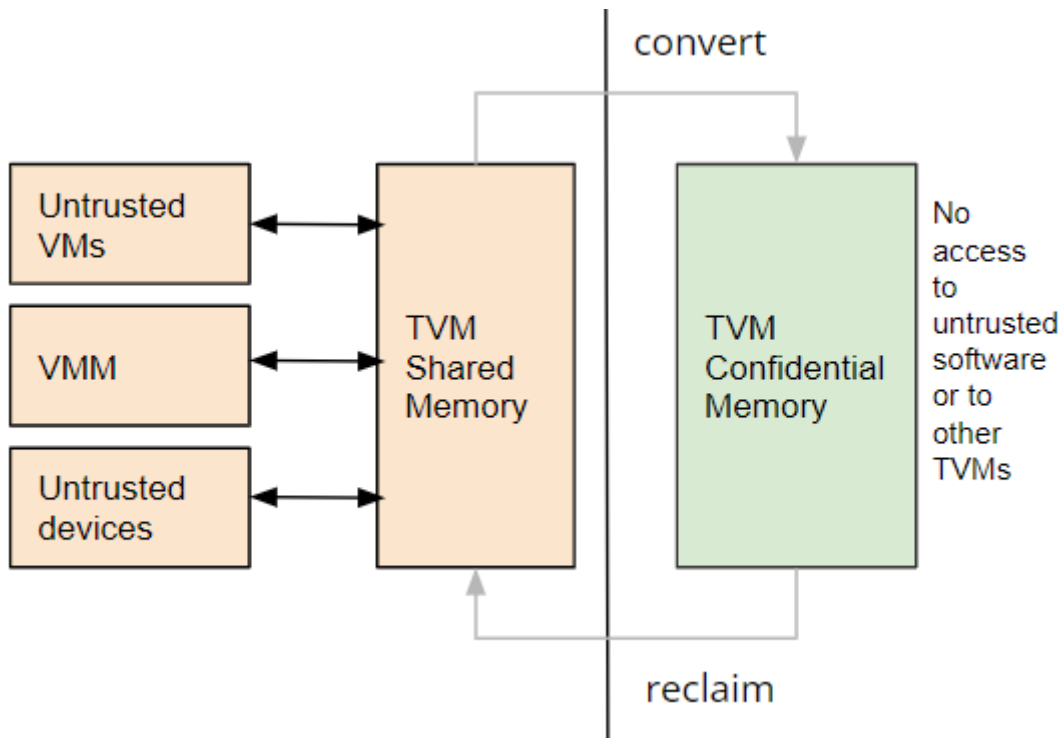


Figure 3: Confidential memory conversion



To also provide physical memory protection, memory accessible to the confidential supervisor domain via MTT may also be associated with a unique memory encryption key. Additionally each TVM may also be associated with a unique memory encryption key. These additional protection aspects are platform and implementation dependent.

Confidential and non-confidential memory are both always assigned by the VMM - the TSM and TSM-driver are expected to manage the isolation for confidential memory by programming the Memory Tracking Table (MTT). The desired security properties of memory tracking are discussed below. The TSM manages finer-granular (page-based) allocation from confidential memory regions (enforced by the memory tracking hardware) using the G-stage page table.

Four aspects of memory isolation are impacted due to this dynamic configurable property of the MTT:

5.1.1. Address Translation/Page Walk

The figure 2 below describes a reference model for memory tracking lookup where the physical address derived from the two-stage address translation and protection mechanism is looked up via the MTT configured for the active supervisor domain to get the access permissions for the physical address. This lookup should be performed for each implicit and explicit memory access and per the paging sizes/modes supported by the hart.

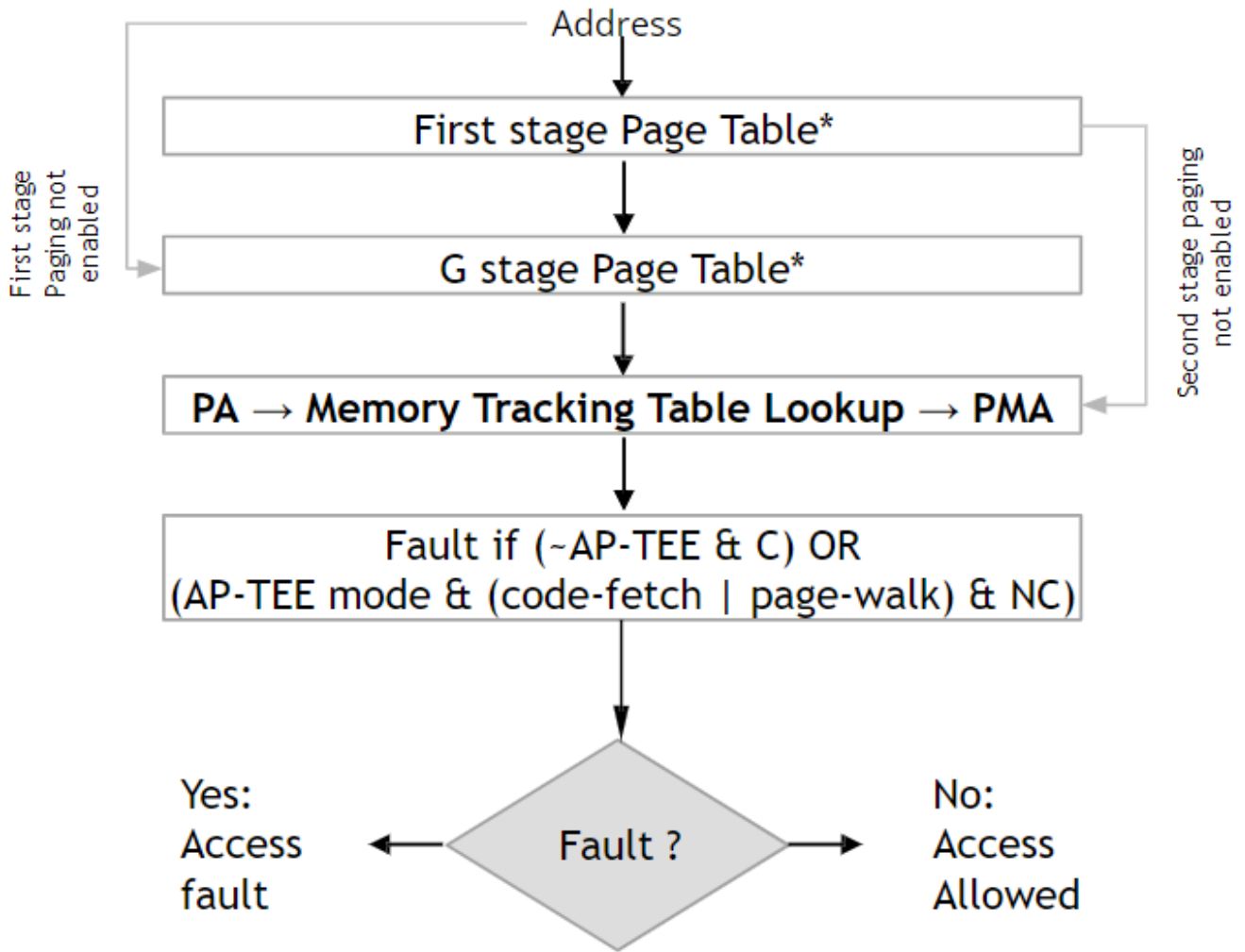


Figure 4: Memory Tracking for Supervisor Domains

5.1.2. Management of isolation for Confidential Physical Memory

The SW TCB (TSM) manages the assignment of physical memory to the Confidential supervisor domain, while the HW TCB (hart MMU including virtual memory system, MTT Extensions) enforces the access-control for confidential memory against other supervisor domains. The region sizes at which the memory tracking enforces isolation may be multiples of the architectural page sizes supported by the hart MMU. The IOMMU is expected to support a similar memory tracking lookup to enable a device/function trusted by the TVM to directly access TVM confidential memory regions. For the CoVE reference architecture this TCB consists of the HW (e.g. MMU, IOMMU, Memory Controller) and the SW/FW elements - TSM-driver and the TSM. The TSM-driver is responsible for enforcing isolation of confidential memory regions (consisting of multiple pages via MTT) and the TSM is responsible for enforcing isolation of confidential memory pages among TVMs (via G-stage translation) - pages assigned to the TVM may be exclusively accessible to the confidential supervisor domain or may be shared with the hosting supervisor domain (e.g. to allow for paravirtualized 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 software is referred to as the Extended Memory Tracking Table (EMTT).

5.1.3. Handling Implicit & Explicit Memory Accesses

For TVM accesses for instruction fetch and page walks, isolated/confidential memory is required to enforce the following security properties:

- TEE Instruction fetch - security property: TVM/TSM must not fetch code from untrusted/shared memory - enforced by the hart
- TEE Paging structure walk - security property: TVM and TSM must not locate page tables in untrusted shared memory.
- TEE data fetch - security property: The TVM via the TSM may be 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 HW TCB and SW TCB (TSM) must enforce via memory-management fences that stale data is not accessible to the TVM (or the hosting OS/VMM). 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 (e.g. MTT cache invalidations) via the proposed COVH interface.

5.2. TSM initialization

The CoVE architecture requires a hardware Root-of-trust for supporting TCB measurement, reporting and storage [8]. The Root-of-trust for Measurement (RTM) is defined as the TCB component that performs a measurement of an entity and cryptographically signs it as attestation evidence subsequently reported to a relying party. 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/FW 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 confidential 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 platform firmware may pre-load the RoT-authenticated 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 confidential 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 confidential memory in TSM Hart Control Structures (THCS - See [Chapter 13](#)) which is initialized as part of the TSM memory initialization. The THCS structure definition is part of the COVH ABI 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 confidential memory assigned to the TVM by the VMM.

5.3. TSM operation and properties

The TSM implements COVH APIs that are invoked by the OS/VMM or by the TVMs, e.g. by the VMM to grant a TVM a confidential 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 and medeleg). 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. 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).

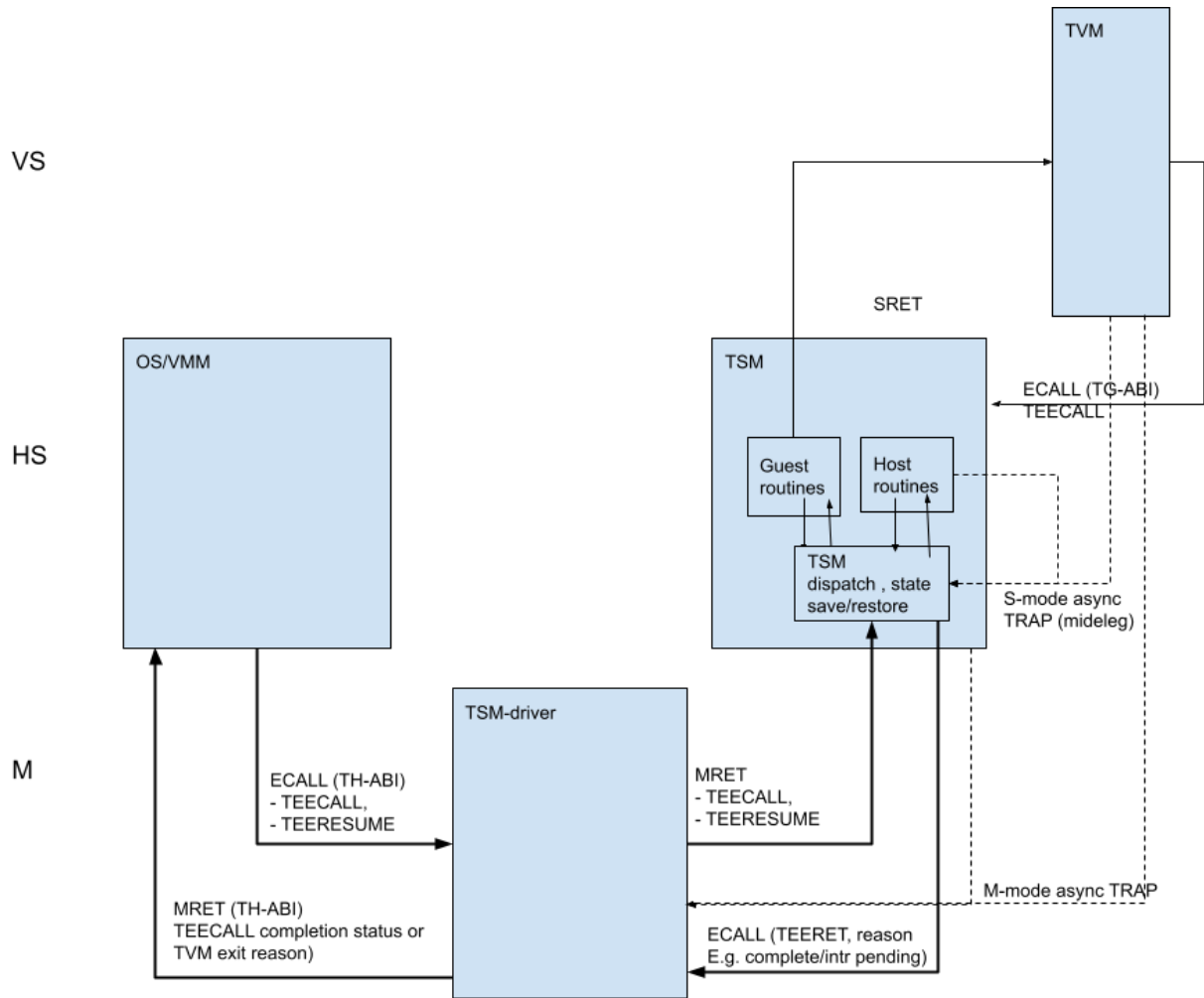


Figure 5: TSM operation - Interruptible and non-reentrant TSM model shown.

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 a synchronous trap due to TEECALL/ TEERESUME then activate confidential supervisor domain for the hart via M-mode `mttp` CSR (See Supervisor Domains specification [\[R20\]](#) for CSR definition)
- 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)
- Enable hosting supervisor domain on hart (via Supervisor Domains [\[R20\]](#) M-mode CSR `mttp` to

disable non-TCB accesses to confidential 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 COVH/G 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 - For the deployment model (a), the CPU must enforce hardware-based access-control of TSM memory via the G-stage page tables to prevent the guest VMM from accessing TSM memory. For the deployment model (b). The CPU must also similarly enforce access-control of TSM memory to prevent access from host supervisor domain components (VMM and host OS that operate in V=0, HS-mode) software. Since in this deployment model (b), other supervisor domains have access to 1st and G-stage paging hardware, the root security manager (TSM-driver) must use MTT to isolate supervisor domain memory. In this deployment model, TEE and TVM address spaces are identified by supervisor domain identifiers (Smsdid) to maintain the isolation during access and in internal address translation caches, e.g. Hart TLB lookup may be extended with the SDID in addition to the ASID, VMID for workloads in the Confidential supervisor domain. TVM memory isolation must support sparse memory management models and architectural page-sizes of 4KB, 64K, 2MB, 1GB (and optionally 512GB). The hardware may implement the MTT as specified in the Smmmtt privileged ISA extension, or other approaches may be used such as a flat table. The memory tracking 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 Supervisor Domain Identifier 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 (hgap). 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

As described above, TVMs can access both classes of memory - isolated 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. virtio, gRPC communication with 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 (instead or in addition), 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 [6]. 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. Note that the external debug opt-in control for the hosting supervisor domain must be independent from the confidential supervisor domain. Similarly, external debug controls should be independently managed by the RoT to allow for root security manager (TSM-driver) debug.

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-initialization of the TVM.

If the TVM has opted-in to performance monitoring, the TSM may invoke the SBI PMU extension (via TSM-driver) or use M-mode counter delegation (`Smcdeleg`) and Supervisor counter configuration (`Ssccfg`) extensions to establish TVM-specific controls and configuration that allows performance monitoring in a TVM. 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.

Chapter 6. TVM Attestation and Measurements

The CoVE TVM attestation framework allows for CoVE workload owners to assert the trustworthiness of the hardware and software environment their workload is running in.

Attestation relies on the ability for the SoC to generate a cryptographic evidence for a workload executing in a CoVE TVM. The workload executing in a TVM may request this cryptographic evidence to relay to a remote relying party which can then verify that the evidence is valid (per some appraisal policy), and thus attest to the trustworthiness of the TVM. The relying party can then accept to release secrets or attestation result tokens back to the trusted workload.

This section describes the CoVE attestation evidence content, format and generation interface.

6.1. Measurements

A CoVE workload measurement is a cryptographic hash of at least one of the TCB elements for the workload. Measurements can only be extended with additional data, by applying a cryptographic hash algorithm (H_{alg}) to the current measurement value ($M_{current}$):

$$M_{extended} = H_{alg}(M_{current} || NewData)$$

Measurements must be stored in integrity protected measurement registers, either in the Hardware RoT of Measurement (RoTM) or in the TSM when it acts as an extended RoTM.

In order to verify a workload environment trustworthiness, the remote attestation service must be able to compare the CoVE TCB measurements with a set of reference values. All components of what constitutes the workload Trusted Computing Base (TCB) must be measured and included in the attestation evidence.

6.1.1. TCB Elements

Elements considered to be in the TCB for CoVE workloads are both hardware and software components.

The overall workload TCB is a composition of 3 independent TCBs:

1. The Platform TCB is made of all hardware and software TCB elements for the CoVE host platform. They are not confidential compute specific and measurements for those elements could be shared with e.g. a measured boot attestation.
2. The TSM TCB elements are the confidential compute specific but TVM agnostic elements for the overall TCB.
3. The TVM TCB is composed of the TVM specific elements.

The TCB elements for each of them is summarized in the following table:

Table 1. COVE Workload TCB Elements

| TCB | Hardware Elements | Software Elements |
|----------|---|--|
| Platform | HW RoT for boot, measurement and storage | All M-mode firmwares, including the TSM-driver |
| | All CPU hardware logic, including MMU and caches | |
| | All SoC subsystems, including memory confidentiality, integrity and replay-protection for volatile memory | |
| | IOMMU and translation agents | |
| TSM | N/A | TEE Security Manager (TSM) and its U-mode components |
| TVM | Directly assigned, TEE-IO compliant devices | All TVM measured pages |

6.1.2. TVM Measurements

At a high level, a TVM measurement is separated into a set of initial measurements and a runtime one.

Initial Measurements

TVM initial measurements are generated from the CoVE workload TCB elements involved in the TVM construction. Any TCB element that directly or indirectly supports a TVM must be measured into the TVM initial measurement registers. Once a TVM is finalized, i.e. after the `sbi_tee_host_finalize_tvm()` TH-ABI is called, the TVM initial measurements must no longer be extended.

Each TVM's initial measurements are stored in dedicated measurement registers and a CoVE implementation must provide at least 1 and at most 8 of them. The initial measurement registers cover platform, TSM & TVM specific TCB element measurements.

Since they hold TCB elements measurements bound to the platform lifetime, the platform/TSM measurement registers may be stored in the hardware Root of Trust for Measurement (RTM). The TVM measurements must be maintained in TSM confidential memory with TVM bound life cycle, effectively making the TSM an extended RTM for the TVM.

The initial TCB measurement mechanism and storage layout is implementation specific. One such possible layout, based on a 6 initial measurement registers CoVE implementation may look like the following table:

Table 2. Initial Measurements Layout Example (6 registers)

| Register | Usage | TCB Elements | Storage |
|----------|-------------------|------------------------------|---------|
| 0 | Platform Firmware | RoT Firmware | RoT |
| | | All SoC Subsystems Firmwares | |
| | | All Regular M-mode Firmwares | |
| | | TSM-Driver | |

| Register | Usage | TCB Elements | Storage |
|----------|--------------------------|---------------------------------------|---------|
| 1 | Platform Configuration | All Firmware Manifests | RoT |
| 2 | TSM Software | TSM Binary | RoT |
| 3 | TSM Configuration | TSM Manifest | RoT |
| 4 | TVM Code and Static Data | TVM Measured Pages | TSM |
| 5 | TVM Configuration | TVM Entry Point and Initial Arguments | TSM |

Runtime Measurements

After the TVM is finalized, initial measurements are immutable and can no longer be modified. However, a TVM guest may want to measure some of its software components (Guest firmware, kernel OS, additional configuration parameters, etc) while it is booting or running.

The TSM optionally provides an interface for TVM guests to extend their measurement into runtime measurement registers. When supporting that feature, the TSM must store runtime measurements separately from the TVM initial ones. The TSM can use up to 18 runtime measurement registers for that purpose.

The TVM measurement extension interface is exposed through the optional TG-ABI `sbi_covg_extend_measurement()` FID.

6.2. Attestation

All above described TCB elements measurements are added to an attestation evidence and then reported to relying parties. The attestation mechanism and protocol that take place between the attester (i.e. the TVM) and the remote attestation service are out of this document scope.

In this section we describe the high level attestation model for CoVE, together with the attestation evidence content, format and generation process.

6.2.1. Model

The CoVE attestation model follows the IETF RATS [1] Remote Attestation architecture. CoVE implementations perform the RATS Attester role, and each CoVE TCB component participates to the generation of a layered Attestation Evidence composed of TCB specific Claims. Moreover, the generated CoVE Evidence freshness is established through the inclusion on a cryptographic nonce in the TVM Claims Set. This is enforced by the `sbi_covg_get_evidence` intrinsic signature.

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 authenticates the Evidence with Endorser provided trust anchors (Endorsements), compares it against Reference Values and appraises it via appraisal policies. It eventually creates Attestation Results to support Relying Parties in their decision process.

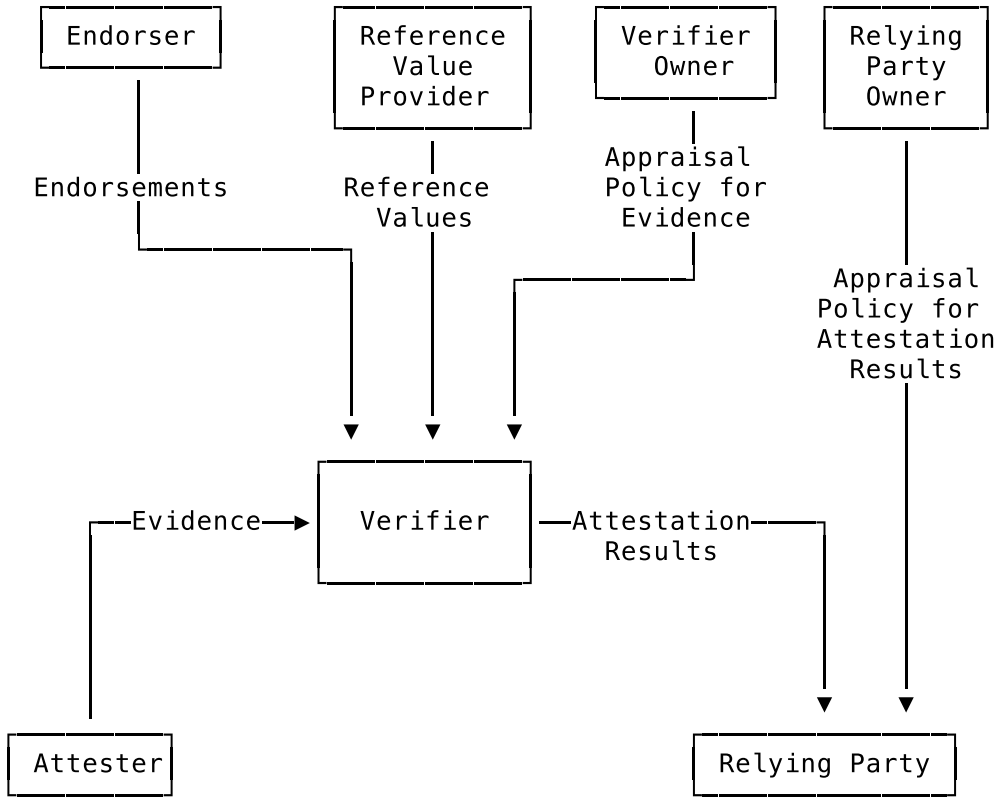


Figure 6: Remote Attestation Framework (IETF RATS)

6.2.2. Layered Evidence

The CoVE Attestation Evidence is a collection of Claims representing the overall workload TCB and state, which includes those of the platform, TSM and TVM. It uses the DICE[2] layered attestation model where each TCB layer collects Claims about the layers it loads, acting as the Attesting Environment for them. As the root TCB layer, the hardware RoT is the initial CoVE Attesting Environment and its own Claims are asserted by an Endorser.

In the DICE model, each TCB layer computes the Compound Device Identifier (CDI) for the layer it loads. CDIs are secrets that must not leave the CoVE TCB. The root TCB layer, i.e. the hardware RoT, is factory provisioned with a Unique Device Secret (UDS) that serves as the original secret from which all other CDIs are directly or indirectly derived. As such, the TVM CDI is rooted into the provisioned and platform manufacturer endorsed UDS. The CDI for the current TCB layer is a cryptographic derivation of a combination of the Attesting Environment CDI (i.e. the loading TCB layer) and the current TCB layer measurements:

$$CDI_0 = KDF(UDS_{Len}, UDS \parallel H_{alg}(Meas(TCB_0)))$$

$$CDI_N = KDF(CDI_{Len}, CDI_{N-1} \parallel H_{alg}(Meas(TCB_N)))$$

Asymmetric key pairs can be derived from a CDI in order to generate the attestation key for a DICE layer. This key can then be used to attest of the next TCB layer.

$$(UDS_PrivKey, UDS_PubKey) = AsymKDF(UDS)$$

$$(CDI_PrivKey_N, CDI_PubKey_N) = AsymKDF(CDI_N)$$

A fixed length TCB layer identifier called **CDI_ID** can be derived from its attestation key:

$$UDS_ID = KeyDerivationFunction(ID_{Len}, UDS_PubKey)$$

$$CDI_ID_N = KeyDerivationFunction(ID_{Len}, CDI_PubKey_N)$$

When loading the next TCB layer, the attesting layer provides it with both its CDI and its attestation certificate.

As the first step in the CoVE DICE chain, the hardware RoT is responsible for:

1. Generating the Platform CDIs. This is the output of a KDF that takes the UDS and a combination of an extended cryptographic hash of all the Platform TCB software components.
2. Generating the Platform attestation certificate. This is a nested EAT which Claims are described in [Section 6.2.3.1](#), and include all the Platform CDI inputs. It is signed by the RoT attestation key, which is derived from the UDS itself.
3. Passing both the Platform CDI and certificate to the TSM-driver

The next step in the DICE chain is the TSM-driver. It is responsible for generating the TSM CDI and attestation certificate and follow similar steps as the above described ones to do so. The generated TSM attestation certificate is composed of both the hardware RoT generated Platform Token and the TSM-driver created TSM Token (See [Section 6.2.3.2](#)). It is signed by the Platform CDI-derived attestation key.

As the following step in the DICE chain, the TSM generates and provision any TVM it creates with its CDI. TVM CDIs are derived from the TSM CDI and the TVM specific measurements. However, unlike the RoT and the TSM-driver, the TSM does not pass attestation certificates to its TVMs. For evidence freshness establishment reasons, TVMs asynchronously request their attestation certificates from the TSM, in the form of a TSM-signed Attestation Evidence.

TVMs are the CoVE DICE chain leaves and they can obtain an Attestation Evidence from the TSM by calling the CoVE guest-ABI `sbi_covg_get_evidence()` FID. The TVM provides a challenge value to the TSM through this call, and that value must be included in the generated Evidence. This value allows relying parties to establish the Attestation Evidence freshness.

The CoVE Evidence is composed of separated but cryptographically bound attestation tokens for each of the above-describe TCB layers (Platform, TSM and TVM). As described above, each TCB layer uses its DICE-derived attestation key to signs the next layer, creating a HW RoT-rooted signature chain. A Relying Party can then verify and authenticate the Evidence with platform owner or manufacturer provided Endorsements, like e.g. a Trust Anchor.

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

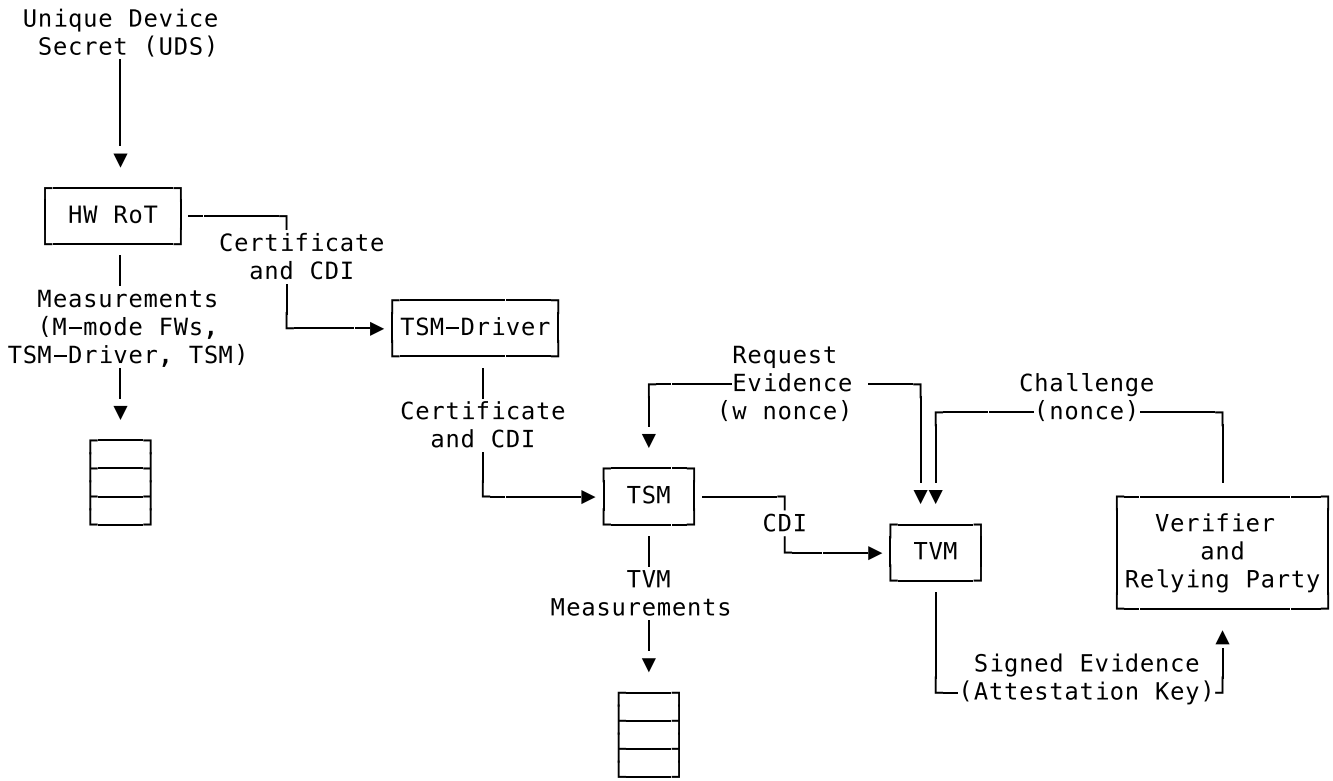


Figure 7: Layered Attestation architecture for TVMs

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 CoVE layered RTMs attestation of CoVE workloads.

6.2.3. Evidence Format

The CoVE Attestation Evidence uses the IETF Entity Attestation Token ([11]), formatted as an untagged, unprotected Concise Binary Object Representation ([12]) Web Token ([14]). A CoVE EAT profile is proposed to narrow the EAT specification for the CoVE use case to enable interoperability.

The UCCS is composed of one EAT submodule Claims-Set map where the map values are attestation tokens for the TVM, TSM and Platform Claims.

The TVM EAT is a CWT tagged CBOR formatted token, wrapped with a COSE_Sign1 [15] envelope. It is signed by the TSM attestation key and must contain a TVM provided challenge, e.g. a Relying Party provided nonce for establishing Evidence freshness.

The TSM EAT is a CWT tagged CBOR formatted token, wrapped with a COSE_Sign1 [15] envelope. It is signed by the Platform attestation key and must include the DICE derived public key for the TSM.

The Platform EAT is a CWT tagged CBOR formatted token, wrapped with a COSE_Sign1 [15] envelope. It is signed by the RoT attestation key and must include the DICE derived public key for the Platform.

The CoVE layered Evidence structure is represented by the above described composition of cryptographically chained EAT tokens. Verifier can then attest of a CoVE workload trustworthiness by independently inspecting each token, while being able to verify that the TCB represented by one token was used to generate the next one.

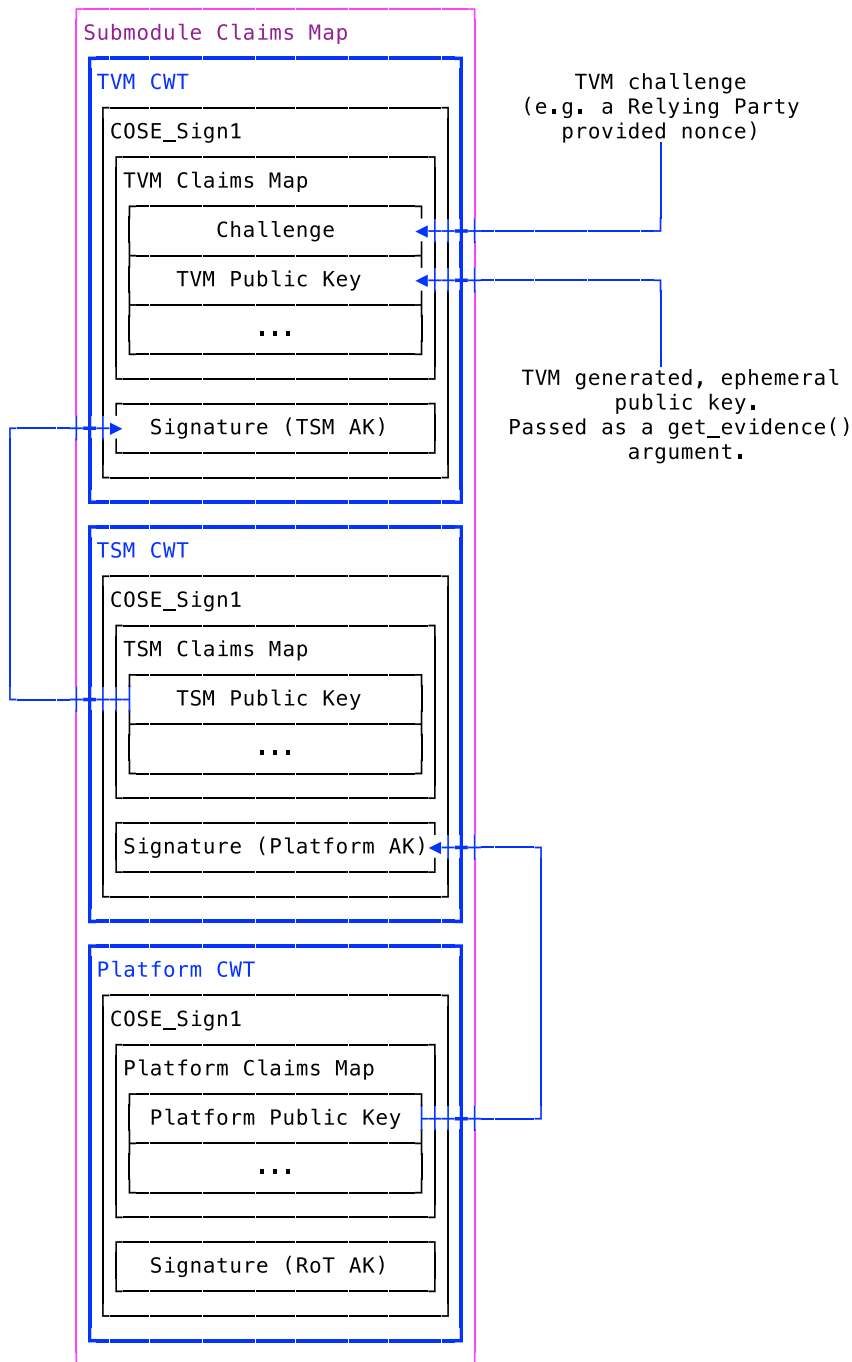


Figure 8: CoVE Attestation Evidence Format

The CoVE Attestation Evidence is defined by the following Concise Data Definition Language (CDDL):

Figure 9: CoVE Attestation Evidence CDDL

```
submodule-label = 266 ; EAT Submodule

protected-cove-token-header-map = {
  alg => int    ; See https://www.iana.org/assignments/cose/cose.xhtml
  ? kid => bstr ; Optional, only needed for the platform token
}

unprotected-cove-token-header-map = {
  * int / tstr => any
}
```

```

}

signed-cove-token = #6.18(COSE-Sign1-cove-token)
COSE-Sign1-cove-token = [
  protected: bstr .cbor protected-cove-token-header-map
  unprotected: unprotected-cove-token-header-map
  payload: bstr .cbor platform-token / tsm-token / tvn-token
  signature: bstr
]

; One EAT Submodule
; Each token is a nested, signed CWT
riscv-cove-token = {
  submodule-label => {
    "platform" => signed-cove-token
    "tsm" => signed-cove-token
    "tvm" => signed-cove-token
  }
}

```

Platform Token

The Platform Token is a nested EAT token in the Evidence and contains a map of Claims. The map is wrapped in a **COSE_Sign1** envelope and composed of the following claims:

Figure 10: Platform Claims Map

```

platform-token = #6.61(platform-token-map)

platform-token-map = {
  riscv-cove-eat-profile ; EAT Profile
  platform-public-key    ; Platform public key
  platform-manufacturer-id ; Platform manufacturer identifier
  platform-state         ; Platform operational state
  platform-sw-components ; Platform SW components
}

```

All above described Claims must be present in the Platform Token.

EAT Profile Claim

The platform EAT profile claim describes the EAT profile that the CoVE platform implements. The profile should include a description of all three tokens (platform, TSM and TVM) as they are bound together.

Figure 11: EAT Profile Claim

```

riscv-cove-eat-profile-label = 265 ; EAT profile
riscv-cove-eat-profile-doc = "https://riscv.org/TBD"

```



```
riscv-cove-eat-profile = (  
    riscv-cove-eat-profile-label => riscv-cove-eat-profile-doc  
)
```

Platform Public Key Claim

The platform public key claim identifies the platform public key that signs the TSM token. The value of the claim is a COSE_Key [15] structure encoded as a CBOR bytes string.

Figure 12: Platform Public Key Claim

```
platform-public-key-label = TBD  
riscv-cove-public-key-type = bytes .cbor COSE_Key  
  
platform-public-key = (  
    platform-public-key-label => riscv-cove-public-key-type  
)
```

Platform Manufacturer Identifier Claim

The platform manufacturer identifier claim uniquely identifies the manufacturer for the CoVE implementation providing the Attestation Evidence. Verification services can use the value of this claim to obtain the manufacturer provided Endorsements for the platform implementation.

Figure 13: Platform Manufacturer Identifier Claim

```
platform-manufacturer-id-label = TBD  
platform-manufacturer-id-type = bytes .size 64  
  
platform-manufacturer-id = (  
    platform-manufacturer-id-label => platform-manufacturer-id-type  
)
```

Platform State Claim

The platform state claim describes the operational state of the platform. The values for this claim can be:

- **NotConfigured** - The platform requires additional information to operate
- **Secured** - This is the default state under regular operation conditions
- **Debug** - The platform can be debugged
- **Recovery** - The platform is recovering from a failure

Figure 14: Platform State Claim

```
platform-state-label = TBD  
platform-state-not-configured = 1  
platform-state-secured = 2
```

```

platform-state-debug = 3
platform-state-recovery = 4
platform-state-type =
    platform-state-not-configured /
    platform-state-secured /
    platform-state-debug /
    platform-state-recovery

platform-state = (
    platform-state-label => platform-state-type
)

```

Platform Software Components Claim

The platform software components claim lists all software and firmware components that compose the CoVE platform TCB.

Each component in the Claim is a map of the following values:

- **Component type:** This is a human-readable string that represents the measured component.
- **Component measurement:** This is the hash value of the component as it was loaded in memory.
- **Component SVN:** This is the component Secure Version Number. The semantics of that value is defined by the component creator or vendor. The SVN can be generated from multiple SVNs.
- **Component manifest:** This is the hash value of the component manifest that was used when loading and verifying the associated component. This field is *optional*. The format of the manifest for the component is out of scope of this specification.
- **Component signer:** This is the hash value of a signing authority for the component.
- **Hash algorithm identifier:** This describes which algorithm was used to generate the component measurement. It is recommended to use one of the IANA defined [\[HashAlgorithmNames\]](#).

Figure 15: Platform Software Components Claim

```

platform-sw-components-label = TBD
riscv-cove-sw-component = {
    1 => text                ; Component type
    2 => riscv-cove-hash-type ; Component measurement value
    3 => text                ; Component Secure Version Number (SVN)
    ? 4 => riscv-cove-hash-type ; Component manifest hash
    5 => riscv-cove-hash-type ; Component signer public key hash value
    6 => text                ; Hash algorithm identifier
}

platform-sw-components = (
    platform-sw-components-label => [ + riscv-cove-sw-component ]
)

```

TSM Token

The TSM Token is a nested EAT token in the Evidence and contains a map of Claims. The map is wrapped in a `COSE_Sign1` envelope and composed of the following claims:

Figure 16: TSM Claims Map

```
tsm-token = #6.61(tsm-token-map)

tsm-token-map = {
  tsm-public-key    ; TSM public key
  tsm-sw-components ; TSM SW components
}
```

All above described Claims must be present in the TSM Token.

TSM Public Key Claim

The TSM public key claim identifies the platform public key that signs the TVM token. The value of the claim is a `COSE_Key` [15] structure encoded as a CBOR bytes string.

Figure 17: TSM Public Key Claim

```
tsm-public-key-label = TBD

tsm-public-key = (
  tsm-public-key-label => riscv-cove-public-key-type
)
```

TSM Software Components Claim

The TSM software components claim lists all software components that compose the CoVE TSM TCB.

The TSM software components that influence the TSM TCB are the TSM-Driver and the TSM.

Figure 18: TSM Software Components Claim

```
tsm-sw-components-label = TBD

tsm-sw-components-type = [
  tsm-driver
  tsm
]

tsm-driver = (riscv-cove-sw-component)
tsm = (riscv-cove-sw-component)

tsm-sw-components = (
  tsm-sw-components-label => tsm-sw-components-type
)
```

)

TVM Token

The TVM Token is a nested EAT token in the Evidence and contains a map of Claims. The map is wrapped in a `COSE_Sign1` envelope and composed of the following claims:

Figure 19: TVM Claims Map

```
tvm-token = #6.61(tvm-token-map)

tvm-token-map = {
    tvm-challenge          ; A TVM guest provided challenge
    ? tvm-identity         ; TVM identity
    tvm-public-key         ; TVM public key
    tvm-initial-measurements ; TVM initial measurements
    ? tvm-runtime-measurements ; TVM runtime measurements
}
```

The TVM runtime measurements Claim is optional, all other above described Claims must be present in the TVM Token.

TVM Challenge Claim

The TVM challenge claim is a `sbi_covg_get_evidence()` caller provided value. The semantics of this Claim is TVM implementation specific, but it is generally used for demonstrating Evidence freshness to a Relying Party.

Figure 20: TVM Challenge Claim

```
tvm-challenge-label = 10 ; EAT nonce
tvm-challenge-type = bytes .size 64

tvm-challenge = (
    tvm-challenge-label => tvm-challenge-type
)
```

TVM Identity Claim

The TVM identity claim value is a `sbi_tee_host_finalize_tvm()` provided argument. It is an optional claim and is not included in the TVM token when the TVM identity argument is set to 0.

It is used by the host TVM creator (e.g. the host VMM) to bind a TVM to an identity or more generically a specific piece of data (e.g. an Attestation Service public key, a configuration blob, etc) through its hash value.

TVM identity allows for untrusted hosts to provide a TVM with unmeasured but attestable pieces of data. A Relying Party can then verify the TVM measurements separately from the host provided TVM identity.

Figure 21: TVM Identity Claim

```
tvm-identity-label = TBD
tvm-identity-type = bytes .size 64

tvm-identity = (
  tvm-identity-label => tvm-identity-type
)
```

TVM Public Key Claim

The TVM public key claim value is a `sbi_covg_get_evidence()` caller provided value. In other words, the TVM guest provides its own, generally ephemeral public key to the TSM to be included into the Evidence. A Relying Party will use that public key to encrypt secrets that are released to the trusted TVM.

The value of the TVM public key claim is a `COSE_Key` [15] structure encoded as a CBOR bytes string.

Figure 22: TVM Public Key Claim

```
tvm-public-key-label = TBD

tvm-public-key = (
  tvm-public-key-label => riscv-cove-public-key-type
)
```

TVM Initial Measurements Claim

The TVM initial measurements claim value is the list of all initial measurements for the TVM. The list must contain at most 8 entries.

Each measurement in the list is a map of the following values:

- **Measurement register index:** This describes the measurement register index used by the TSM to store the measurement value. This can be mapped to well known measurement register indexes like e.g. the TCG[17] defined ones.
- **Measurement value:** This is the measurement value.
- **Hash algorithm identifier:** This describes which algorithm was used to generate the component measurement. It is recommended to use one of the IANA registered hash algorithm name[16].

Figure 23: TVM Initial Measurements Claim

```
tvm-initial-measurements-label = TBD
riscv-cove-measurement = {
  1 => uint           ; Measurement register index
  2 => riscv-cove-hash-type ; Measurement value
  3 => text           ; Hash algorithm identifier
}
```

```
tvm-initial-measurements = (  
    tvm-initial-measurements-label => [ 1*8 riscv-cove-measurement ]  
)
```

TVM Runtime Measurements Claim

The TVM runtime measurements claim value is the list of all runtime measurements for the TVM. The list must contain at most 18 entries.

By calling into the `sbi_covg_extend_measurement()` SBI FID, a TVM guest can extend TVM measurements after the TVM is finalized. The extended measurement values are stored into a set of runtime measurement registers.

Figure 24: TVM Runtime Measurements Claim

```
tvm-runtime-measurements-label = TBD  
  
tvm-runtime-measurements = (  
    tvm-runtime-measurements-label => [ 1*18 riscv-cove-measurement ]  
)
```

6.2.4. Evidence Generation

TVM guest Attesters can request from the TSM to generate an Evidence that attest to their own layered TCB layers, by calling into the TG-ABI `sbi_covg_get_evidence()` FID.

The `sbi_covg_get_evidence()` returns an attestation certificate that includes the UCCS EAT formatted CoVE Attestation Evidence described in the previous sections of this document.

The attestation key and certificate generation for the TVM may be performed by the TSM directly or with a U-mode TSM component, to allow for the interruptibility models discussed in the TSM operation section of this document.

The CoVE attestation certificate can either be X.509[18] or CBOR formatted, depending on the `format` argument passed by the Attester to `sbi_covg_get_evidence()`. CoVE implementations must support at least one certificate format, and describe all supported formats through the `AttestationCapabilities` structure returned by the `sbi_covg_get_attcaps()` SBI call.

The CoVE attestation certificate issuer is the TSM and is represented by the TSM `CDI_ID` lowercase hexadecimal encoded string.

The CoVE attestation certificate subject is the TVM and is represented by the TVM `CDI_ID` lowercase hexadecimal encoded string.

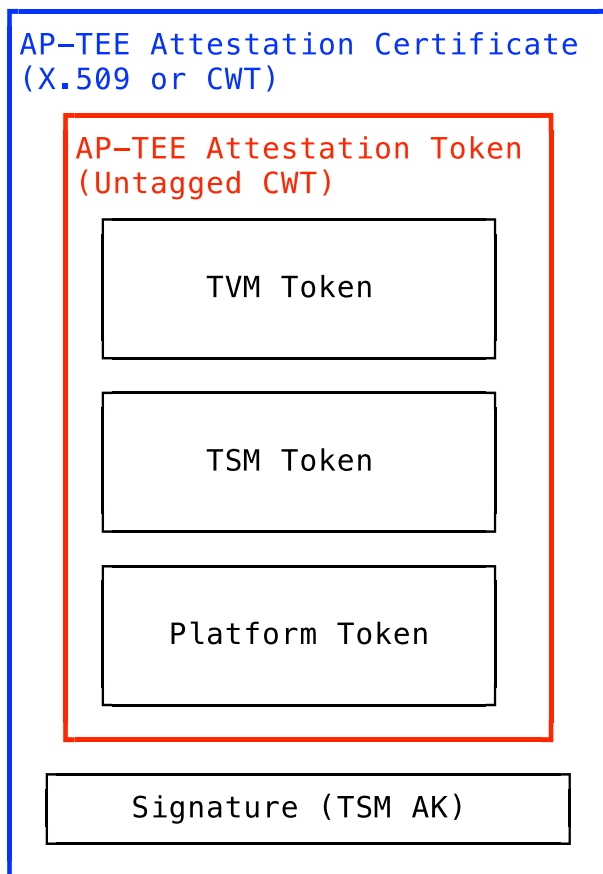


Figure 25: CoVE Attestation Evidence Certificate

CBOR Attestation Certificate

CBOR formatted CoVE Attestation certificates are COSE_Sign1 protected CWTs, signed with the TSM attestation key. The algorithm used to sign the certificate is described by the COSE_Sign1 envelope. It is recommended to use an EdDSA scheme with SHA-512, e.g. Ed25519.

The CBOR certificate COSE_Sign1 payload is a CWT which claim set is composed of the CoVE evidence token and 2 additional claims:

- **Issuer** is the certificate issuer representation, i.e. the TSM **CDI_ID**.
- **Subject** is the certificate subject representation, i.e. the TVM **CDI_ID**.

Figure 26: CoVE CBOR Attestation Certificate

```

; COSE_Sign1 wrapped certificate payload
riscv-cove-certificate = #6.18(riscv-cove-certificate-payload)

; The payload is a CWT
riscv-cove-certificate-payload = #6.61(riscv-cove-claims)

issuer-label = 1 ; CWT iss
subject-label = 2 ; CWT sub
evidence-label = TBD

riscv-cove-claims = (

```

```

    issuer-label => text                ; Certificate issuer
    subject-label => text                ; Certificate subject
    evidence-label => riscv-cove-token ; Evidence token
)

```

X.509 Attestation Certificate

X.509 formatted CoVE Attestation certificates are leaf certificates that follow the DICE[2] X.509 certificate extension format. They are signed by the TSM attestation key and contain a DICE defined custom extension for carrying the attestation evidence as an unprotected CWT Claim Set[14].

The X.509 custom extension value is generated by wrapping the CoVE attestation evidence into a tagged UCCS.

The TSM generated certificate fields, standard and custom extensions, along with the UCCS formatted evidence are described below:

Table 3. COVE X.509 Attestation Certificate Fields

| Field | Description |
|----------------------|--|
| signatureAlgorithm | <code>id-ecdsa-with-SHA512</code> [19] is recommended. Other signature algorithms may be used. |
| signatureValue | 64 bytes ECDSA signature, using the TSM <code>CDI_PriKey</code> as the signing key |
| version | v3 |
| serialNumber | The TSM <code>CDI_ID</code> in ASN.1 INTEGER form |
| signature | <code>id-ecdsa-with-SHA512</code> [19] is recommended. Other signature algorithms may be used. |
| issuer | The TSM <code>CDI_ID</code> |
| validity | The validity values are populated as follows: <code>notBefore</code> can be any time known to be in the past, and <code>notAfter</code> is set to the standard value used to indicate no well-known expiry date, “99991231235959Z” per RFC 5280. |
| subject | The TVM <code>CDI_ID</code> |
| subjectPublicKeyInfo | When using EcDSA, per RFC 5480 (<code>id-ecPublicKey</code>) |
| issuerUniqueID | Not used |
| subjectUniqueID | Not used |
| extensions | Standard extensions are included as well as a custom TCG DICE extension which holds the CoVE attestation evidence. Both are described below. |

Table 4. COVE X.509 Attestation Certificate 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 is set to 0. |

Table 5. COVE X.509 Attestation Certificate Custom Extension Fields

| Field | Value |
|-----------|--|
| extnID | OID from [2] for UccsEvidence |
| critical | TRUE |
| extnValue | The CoVE UCCS X.509 extension (See CDDL below) |

Figure 27: CoVE UCCS X.509 extension

```
riscv-cove-x509-ext = #6.601(riscv-cove-token) ; Unprotected CBOR Web Token
```

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 confidential 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 the assignment of confidential memory 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 confidential memory under TSM-managed memory tracking and encryption controls described earlier).
2. Allocation/Assignment of confidential 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 hcatp leaf page table entries. For pages added to the TVM, the VMM must invoke `sbi_covh_add_tvm_measured_pages()` which extends the initial 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 initial 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 [9] 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 h gatp 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 runtime and initial 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 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 access permissions for confidential supervisor domains and hosting supervisor domains for all software-accessible physical memory addresses.
- 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 (initially 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 access 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 access evaluation - which results in the TEE access 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 access is transferred between TEE and non-TEE domains 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.

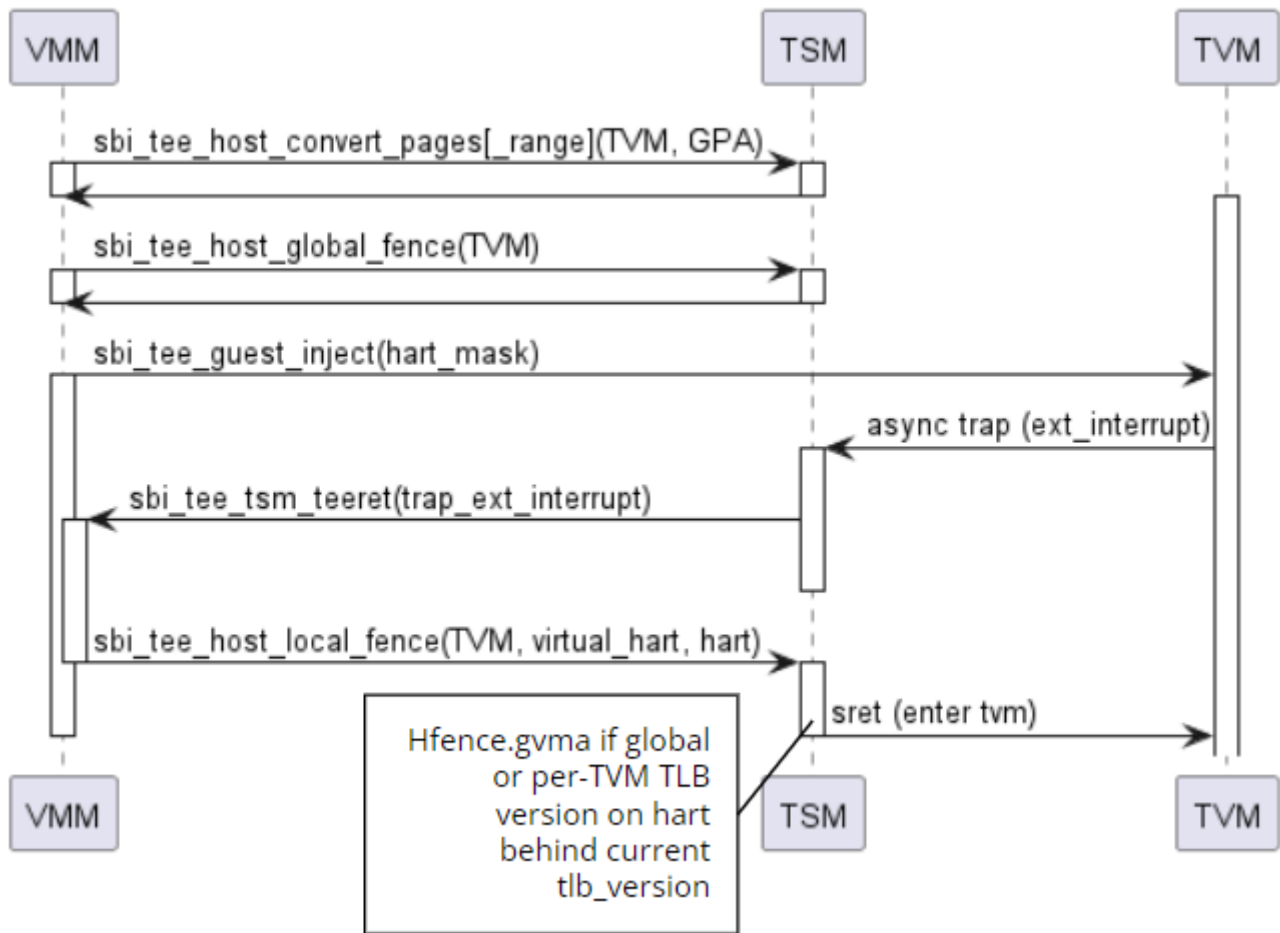
- `sbi_covh_reclaim_pages`: VMM may unassign memory for TVMs by destroying them. All confidential-unassigned memory may be reclaimed back as nonconfidential using this interface.

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-TCB/hosting supervisor domain software cannot create new TLB entries to donated pages - since host software accesses to confidential memory pages will fault (including implicit accesses)

7.3.5. Global and per-TVM TLB management

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 hgap 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 hgap 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 hgap is/are Unassigned-Confidential per the MTT
- For the GPA to be mapped, perform a TVM-hgap 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 hgap page to zero (no hgap page table entries are valid)
- Update the parent hgap entry to refer to the new hgap page (mark non-leaf as valid)
- Update the hgap 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 (initial) 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 performed by the TSM in the context of the confidential supervisor domain 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 the 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 hgap 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. This specification introduces three new extensions:

- 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. These steps are performed by the non-TCB hosting 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 [10], 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 6. 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 documentation of `sbi_covh_run_tvm_vcpu()` for 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 6. 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. |

| CSRs | TSM | Host | Purpose |
|-------------|-----|------|--|
| 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 a 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 is 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.

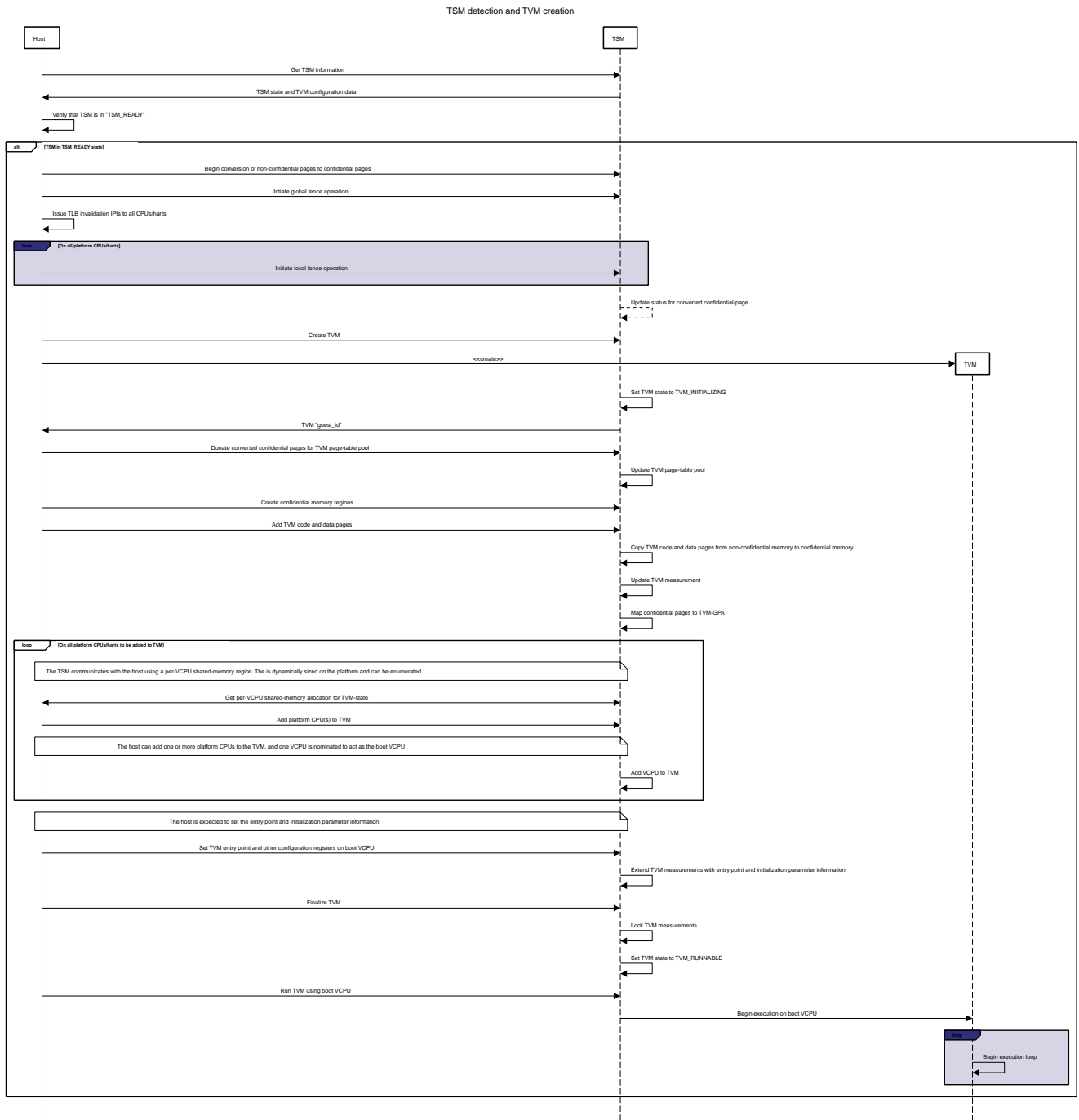


Figure 30: TSM Detection and TVM creation

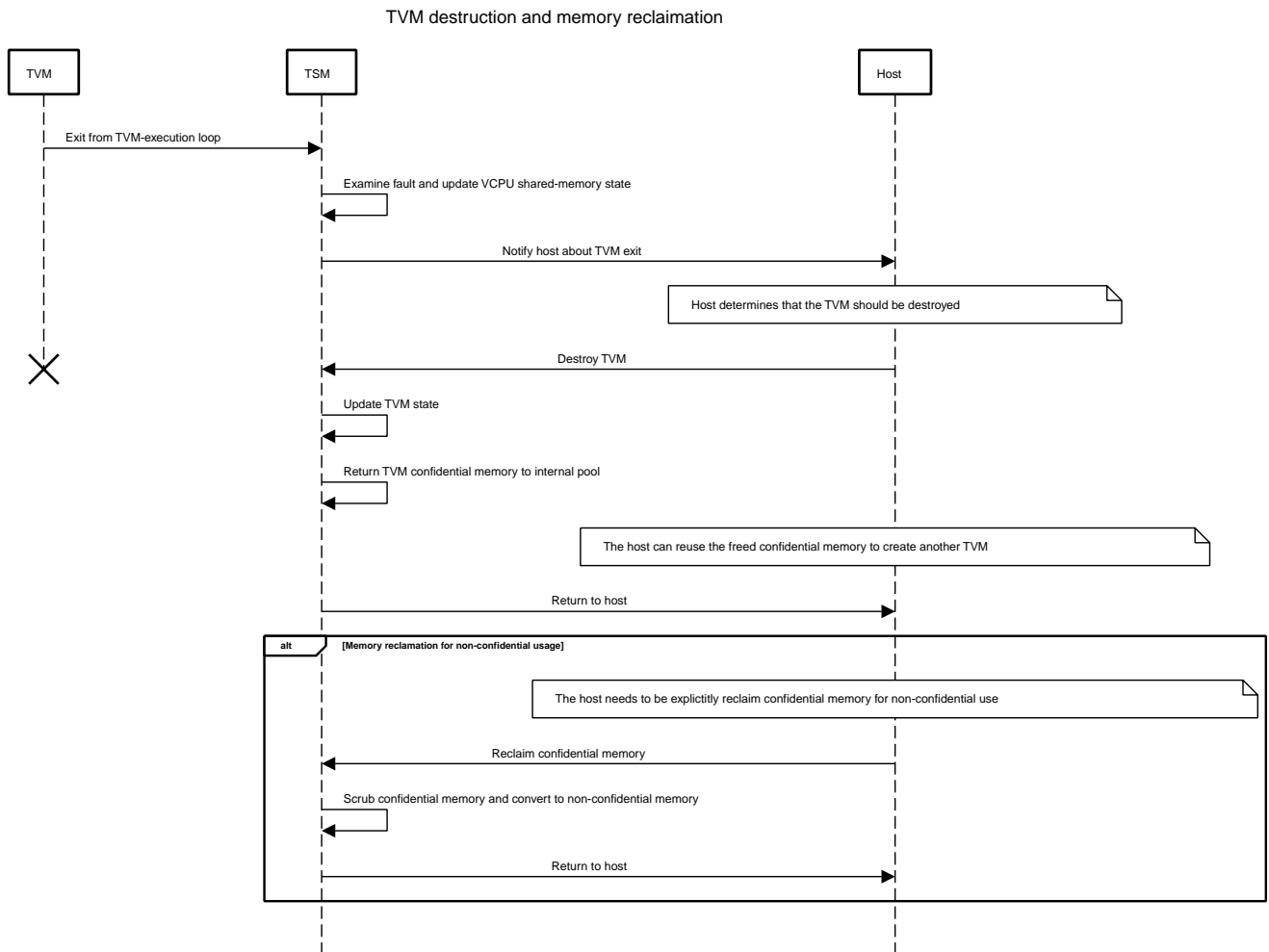


Figure 31: TVM destruction and Memory reclamation



Figure 32: 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 7. COVE Host Get TSM Info

| Error code | Description |
|-------------------------|---|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_ADDRESS | <code>tsm_info_address</code> was invalid. |
| SBI_ERR_INVALID_PARAM | <code>tsm_info_len</code> 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 8. 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 9. COVE Host Convert Pages

| Error code | Description |
|-------------------------|---|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_ADDRESS | <code>base_page_address</code> was invalid. |
| SBI_ERR_INVALID_PARAM | <code>num_pages</code> 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 10. COVE Host Reclaim Pages

| Error code | Description |
|-------------------------|---|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_ADDRESS | <code>base_page_address</code> was invalid. |
| SBI_ERR_INVALID_PARAM | <code>num_pages</code> 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 11. 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 12. 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 13. 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 14. COVE Host Create TVM Errors

| Error code | Description |
|-------------------------|--|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_ADDRESS | <code>tvm_create_params_addr</code> was invalid. |
| SBI_ERR_INVALID_PARAM | <code>tvm_create_params_len</code> 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,  
unsigned long tvm_identity_addr);
```

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.

`tvm_identity_addr` points to a 64 bytes buffer containing a host-defined TVM identity. This piece of data can be used to bind TVMs to a host-defined identity (e.g. an attestation service public key, a guest configuration file hash, an attestation policy description, etc). Although this piece of data is included in the TVM attestation certificate as a dedicated TVM claim (`tvm-identity`), it is **not** included in the TVM measurements. That allows for the host to optionally personalize cryptographically identical TVMs through an attestable and verifiable identity.

The semantics of this piece of data is defined by the host and can be ignored by both the guest and the attestation services. However, when being used, the TVM identity can be leveraged as follows:

1. The host passes some information to the guest through e.g. some out-of-band VM orchestration mechanisms. This could be e.g. the hash value for a policy file the guest is expected to apply at runtime.
2. The guest compares the passed host data with the `tvm-identity` attestation certificate claim and can decide to use it or not depending on this local verification process.
3. When requesting a confidential resource, the relying party can check that the host provided identity data is trustworthy and that the guest measurements are for a TCB that may have used it.
4. The relying party can choose to release the resource to the guest based on this verifiable TVM identity.

Giving TVMs an identity is optional and the TSM must not include a TVM identity claim in the TVM attestation token when `tvm_identity_addr` is set to 0. When a TVM identity is provided, the `tvm_identity_addr` must be different than 0 and 64B-aligned.

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 15. COVE Host Finalize TVM Errors

| Error code | Description |
|-----------------------|--|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> or <code>tvm_identity_addr</code> was invalid, or the TVM wasn't in the <code>TVM_INITIALIZING</code> 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 16. COVE Host Destroy TVM Errors

| Error code | Description |
|-----------------------|---|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> 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 17. COVE Host Add TVM Memory Region

| Error code | Description |
|-------------------------|--|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_ADDRESS | <code>tvm_gpa_addr</code> was invalid. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> or <code>region_len</code> were invalid, or the TVM wasn't in the correct state. |

| Error code | Description |
|----------------|---|
| 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 tvn_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 18. COVE Host Add TVM Page Table Pages

| Error code | Description |
|-------------------------|---|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_ADDRESS | <code>base_page_address</code> was invalid. |
| SBI_ERR_OUT_OF_PTPAGES | The operation could not complete due to insufficient page table pages. |
| SBI_ERR_INVALID_PARAM | <code>tvn_guest_id</code> or <code>num_pages</code> were invalid, or <code>tsm_page_type</code> is invalid. |
| SBI_ERR_NOT_SUPPORTED | The <code>tsm_page_type</code> 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 tvn_guest_id,
                                              unsigned long source_address,
                                              unsigned long dest_address,
                                              unsigned long tsm_page_type,
                                              unsigned long num_pages,
                                              unsigned long tvn_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 initial 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 19. COVE Host Add TVM Measured Pages

| Error code | Description |
|-------------------------|--|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_ADDRESS | <code>source_address</code> was invalid, or <code>dest_address</code> wasn't in a confidential memory region. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> , <code>tsm_page_type</code> , or <code>num_pages</code> were invalid, or the TVM wasn't in the <code>TVM_INITIALIZING</code> 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 20. COVE Host Add TVM Zero Pages Errors

| Error code | Description |
|-------------------------|--|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_ADDRESS | <code>base_page_address</code> or <code>tvm_base_page_address</code> were invalid. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> , <code>tsm_page_type</code> , or <code>num_pages</code> 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 tvn_guest_id,  
                                           unsigned long base_page_address,  
                                           unsigned long tsm_page_type,  
                                           unsigned long num_pages,  
                                           unsigned long tvn_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 21. COVE Host Add TVM Shared Pages

| Error code | Description |
|-------------------------|--|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_ADDRESS | <code>base_page_address</code> or <code>tvm_base_page_address</code> were invalid. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> , <code>tsm_page_type</code> , or <code>num_pages</code> 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 tvn_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 22. COVE Host Create TVM VCPU Errors

| Error code | Description |
|-----------------------|---|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> or <code>tvm_vcpu_id</code> were invalid, or the TVM wasn't in <code>TVM_INITIALIZING</code> 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 23. COVE Host Run TVM VCPU Errors

| Error code | Description |
|-----------------------|---|
| SBI_ERR_SUCCESS | The TVM exited, and <code>sbiret.value</code> contains 0 if the interrupt or exception is resumable. The host can examine <code>scause</code> to determine details. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> or <code>tvm_vcpu_id</code> were invalid, or the TVM wasn't in <code>TVM_RUNNABLE</code> 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. i
     * 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 tvml_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 24. COVE Host Initiate TVM Fence

| Error code | Description |
|-------------|---------------------------------------|
| SBI_SUCCESS | The operation completed successfully. |

| Error code | Description |
|-------------------------|---|
| 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 25. COVE Host TVM Invalidate Pages

| Error code | Description |
|-------------------------|--|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> or <code>length</code> were invalid. |
| SBI_ERR_INVALID_ADDRESS | <code>gpa</code> 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 26. COVE Host TVM Validate Pages

| Error code | Description |
|-------------------------|--|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> or <code>length</code> were invalid. |
| SBI_ERR_INVALID_ADDRESS | <code>gpa</code> 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 tvn_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 27. COVE Host TVM Remove Pages

| Error code | Description |
|-------------------------|--|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> or <code>length</code> were invalid. |
| SBI_ERR_INVALID_ADDRESS | <code>gpa</code> 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 tvmaia_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 28. COVE Interrupt Init TVM AIA

| Error code | Description |
|-------------------------|--|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_ADDRESS | <code>tvm_aia_params_addr</code> was invalid. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> or <code>tvm_aia_params_addr</code> was invalid, or the TVM wasn't in the <code>TVM_INITIALIZING</code> 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 tvn_guest_id,
                                                  unsigned long tvn_vcpu_id,
                                                  unsigned long tvn_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 29. COVE Interrupt Set TVM AIA CPU IMSIC Addr

| Error code | Description |
|-------------------------|---|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_ADDRESS | <code>tvm_vcpu_imsic_gpa</code> was invalid. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> or <code>tvm_vcpu_id</code> were invalid, or the TVM wasn't in the <code>TVM_INITIALIZING</code> 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 30. COVE Interrupt Convert AIA IMSIC

| Error code | Description |
|-------------------------|---|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_ADDRESS | <code>imsic_page_addr</code> 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 31. COVE Interrupt Reclaim TVM AIA IMSIC

| Error code | Description |
|-------------------------|---|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_ADDRESS | <code>imsic_page_addr</code> 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 tvml_guest_id,  
                                     unsigned long tvml_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 32. COVE Interrupt Bind AIA IMSIC

| Error code | Description |
|-----------------------|--|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> or <code>tvm_vcpu_id</code> or <code>imsic_mask</code> 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 tvml_guest_id,
                                              unsigned long tvml_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 33. COVE Interrupt Unbind AIA IMSIC Begin

| Error code | Description |
|-----------------------|---|
| SBI_SUCCESS | The operation was completed successfully. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> or <code>tvm_vcpu_id</code> 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 tvml_guest_id,
                                              unsigned long tvml_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 34. COVE Interrupt Unbind AIA IMSIC End

| Error code | Description |
|-----------------------|---|
| SBI_SUCCESS | The operation was completed successfully. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> or <code>tvm_vcpu_id</code> 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 tvn_guest_id,
                                     unsigned long tvn_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 35. COVE Interrupt Inject TVM CPU

| Error code | Description |
|-----------------------|--|
| SBI_SUCCESS | The operation completed successfully. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> or <code>tvm_vcpu_id</code> or <code>interrupt_id</code> 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 tvn_guest_id,
                                              unsigned long tvn_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 36. COVE Interrupt Rebind AIA IMSIC Begin

| Error code | Description |
|-----------------------|--|
| SBI_SUCCESS | The operation was completed successfully. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> or <code>tvm_vcpu_id</code> or <code>imsic_mask</code> 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 tvml_guest_id,  
                                              unsigned long tvml_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 37. COVE Interrupt Rebind AIA IMSIC Clone

| Error code | Description |
|-----------------------|---|
| SBI_SUCCESS | The operation was completed successfully. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> or <code>tvm_vcpu_id</code> 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 tvml_guest_id,  
                                             unsigned long tvml_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 38. COVE Interrupt Rebind AIA IMSIC End

| Error code | Description |
|-----------------------|---|
| SBI_SUCCESS | The operation was completed successfully. |
| SBI_ERR_INVALID_PARAM | <code>tvm_guest_id</code> or <code>tvm_vcpu_id</code> 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 tvmm_gpa_addr,
                                       unsigned long region_len);
```

Marks the specified range of TVM physical address space starting at `tvmm_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 `tvmm_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 39. 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 | <code>tvmm_gpa_addr</code> 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 tvmm_gpa_addr,
                                           unsigned long region_len);
```

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

Both `tvmm_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 40. 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 | <code>tvm_gpa_addr</code> 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 41. 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 | <code>tvm_gpa_addr</code> was invalid. |
| SBI_ERR_INVALID_PARAM | <code>region_len</code> 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 42. 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 | <code>tvm_gpa_addr</code> was invalid. |
| SBI_ERR_INVALID_PARAM | <code>region_len</code> was invalid, or the entire range doesn't span a <code>SHARED_MEMORY_REGION</code> |
| 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 43. 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 | <code>interrupt_id</code> 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 44. 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 | <code>interrupt_id</code> 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 tvmm_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 attestation certificate formats are supported, and the number of dedicated measurement registers for the TVM initial and runtime measurements.

The attestation capabilities structure also contains a map of all TVM measurement registers, both initial and runtime ones. Only runtime ones can be extended by the TVM guest at runtime.

Both `tvm_cap_addr` and `caps_size` must be page aligned.

```
enum HashAlgorithm {  
    /* SHA-384 */  
    Sha_384,  
    /* SHA-512 */  
    Sha_512,  
    /* SHA3-384 */  
    Sha3_384,  
    /* SHA3-512 */  
    Sha3_512,  
    ...  
};
```

```

    Sha3_512,
};

// CBOR formatted attestation certificate
#define ATTESTATION_CERTIFICATE_CBOR (1 << 0)

// X.509 formatted attestation certificate,
// with a TCG DICE compliant extension (UCCS).
#define ATTESTATION_CERTIFICATE_X509 (1 << 1)

#define MAX_INITIAL_MEASUREMENT_REGISTERS 8
#define MAX_RUNTIME_MEASUREMENT_REGISTERS 18
#define MAX_MEASUREMENT_REGISTERS (MAX_INITIAL_MEASUREMENT_REGISTERS \
    + MAX_RUNTIME_MEASUREMENT_REGISTERS)

struct AttestationCapabilities {
    /* The TCB Secure Version Number. */
    uint64_t tcb_svn;

    /* The supported hash algorithm */
    enum HashAlgorithm hash_algorithm;

    /*
     * The supported attestation certificate formats.
     * This is a bitmap of ATTESTATION_CERTIFICATE_* flags.
     */
    uint32_t certificate_formats;

    /* Number of initial measurement registers */
    uint_8 initial_measurements;

    /* Number of runtime measurement registers */
    uint_8 runtime_measurements;

    /* Array of all measurement register descriptors */
    MeasurementRegisterDescriptor[MAX_MEASUREMENT_REGISTERS] msmt_regs;
};

enum MeasurementType {
    /* Initial measurement */
    Initial,

    /* Runtime measurement */
    Runtime,
}

#define UNMAPPED_TCG_PCR 0xff

struct MeasurementRegisterDescriptor {
    /*
     * The hash function algorithm used for that register.

```

```

    * This must match the AttestationCapabilities 'hash_algorithm' field
    * value.
    */
    enum HashAlgorithm hash_algorithm;

    /* Initial or runtime measurement register */
    enum MeasurementType measurement_type;

    /*
    * This is the TCG PCR index this measurement maps to, such as [0~16,23]
    * as defined in TCG PC Client Specific Platform Firmware Profile Spec.
    * Implementations not mapping their measurement registers to TCG
    * PCR indexes must use UNMAPPED_TCG_PCR for this value.
    */
    uint8_t tcg_pcr_index;
};

```

Table 45. 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 | <code>tvm_caps_addr</code> was invalid. |
| SBI_ERR_INVALID_PARAM | <code>caps_len</code> was invalid, or the entire range doesn't span a <code>CONFIDENTIAL_MEMORY_REGION</code> |
| SBI_ERR_FAILED | The operation failed for unknown reasons. |

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

```

struct sbiret sbi_covg_extend_measurement(unsigned long msmt_buf_addr,
                                          unsigned long msmt_buf_len,
                                          unsigned long msmt_index);

```

This intrinsic is used by a TVM component to extend the TVM runtime set of measurements with one additional data blob. The hash function algorithm used to generate the measurement data must match the `sbi_covg_get_attcaps` reported one.

TVMs can call this function at any time after being finalized. The extended runtime measurement register value will be included in all following attestation certificates generated via `sbi_covg_get_evidence` calls.

`msmt_buf_addr` must be page aligned and must point to a digest generated by the hash function algorithm reported via `sbi_covg_get_attcaps`. `msmt_buf_len` must be equal to the hash function output length, which is a characteristic of the selected hash function algorithm. `msmt_index` must be

a valid runtime measurement register index, per the attestation capabilities reported via `sbi_covg_get_attcaps`.

Table 46. COVE Guest Runtime Measurement Extension

| 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 | <code>msmt_buf_addr</code> was invalid. |
| SBI_ERR_INVALID_PARAM | The <code>msmt_index</code> value is invalid. |
| SBI_ERR_FAILED | The operation failed for unknown reasons. |

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

```
struct sbiret sbi_covg_get_evidence(unsigned long pub_key_addr,
                                   unsigned long pub_key_size,
                                   unsigned long challenge_data_addr,
                                   unsigned long cert_format,
                                   unsigned long cert_addr_out,
                                   unsigned long 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 an attestation evidence to report to a remote relying party.

This intrinsic returns an attestation certificate at the address passed as its fifth argument (`cert_addr_out`). The certificate is signed by the TSM attestation key, and includes the TVM attestation evidence. The TSM attestation key is also included in the reported TSM token.

The caller passes the TVM public key address as the first argument (`pub_key_addr`). This key will be included in the generated certificate and represents the TSM-certified TVM identity.

The third argument (`challenge_data_addr`) points to the attestation challenge blob, typically a relying party generated nonce used for demonstrating the attestation evidence freshness.

The fourth argument (`cert_format`) is the caller's selected attestation certificate format. This must be one of the supported `ATTESTATION_CERTIFICATE_*` flag, per the attestation capabilities reported via `sbi_covg_get_attcaps`.

All addresses (`pub_key_addr`, `challenge_data_addr` and `cert_addr_out`) must be page aligned, and both `pub_key_addr` and `challenge_data_addr` must point to confidential memory.

Table 47. 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 | <code>pub_key_size</code> , <code>cert_size</code> or <code>cert_format</code> was invalid, or the entire range doesn't span a <code>CONFIDENTIAL_MEMORY_REGION</code> |
| SBI_ERR_BUSY | The attestation certificate could not be generated due to some resources being busy. The request may be retried. |
| SBI_ERR_FAILED | The operation failed for unknown reasons. |

11.10. Function: COVE Guest Read Measurement (FID #9)

```
struct sbiret sbi_covg_read_measurement(unsigned long msmt_buf_addr_out,
                                       unsigned long msmt_buf_size,
                                       unsigned long msmt_index);
```

This intrinsic returns the TVM measurement register value for the `msmt_index` measurement register. TVMs can read both initial and runtime measurement register values back.

`sbi_covg_read_measurement` returns the register value at `msmt_buf_addr_out` and `msmt_buf_size` must be large enough to accommodate for the hash function algorithm output length, as reported by `sbi_covg_get_attcaps`.

`msm_index` must be one of the `sbi_covg_get_attcaps` reported measurement register indexes.

`msmt_buf_addr_out` must be page aligned.

Table 48. COVE Guest Read Measurement

| 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 | <code>msmt_buf_addr_out</code> was invalid. |
| SBI_ERR_INVALID_PARAM | <code>msmt_buf_size</code> was invalid, or the entire range doesn't span a <code>CONFIDENTIAL_MEMORY_REGION</code> |
| 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: Confidential 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 (initial). 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 | <p>Copies the given number of pages from non-confidential memory at <code>source_address</code> to confidential memory at <code>dest_address</code>, then measures and maps the pages at <code>dest_address</code> in the TVM physical address space at <code>tvm_guest_gpa</code>. The mapping must lie within a region of confidential memory created with <code>sbi_covh_add_tvm_memory_region()</code>. This call must not be made after calling <code>sbi_covh_finalize_tvm()</code>.</p> <p>This operation is used to extend the initial 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.</p> |
| sbi_covh_add_tvm_zero_pages | <p>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).</p> |
| sbi_covh_add_tvm_shared_pages | <p>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.</p> |
| sbi_covh_create_tvm_vcpu | <p>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 <code>tsm_info</code> call.</p> |

| | |
|---|--|
| 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 for version 2 of the ABI. |
| 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 Confidential memory page to the TSM to use as a new G-stage page table in the fragmented mapping. This interface specification is TBD for version 2 of the ABI. |

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 <code>sbi_covi_rebind_aia_imsic_clone()</code> . |

| | |
|---|--|
| 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_extend_measurement | This intrinsic is used by a TVM component to extend the TVM runtime set of measurement with one additional data blob. The hash function algorithm used to generate the measurement data must match the sbi_covg_get_attcaps reported one. |
| sbi_covg_get_evidence | This intrinsic is used by a TVM to get an attestation evidence to report to a remote relying party. It returns an attestation certificate signed by the TSM attestation key, and includes the TVM attestation evidence. The TSM attestation key is also included in the reported TSM token. |
| sbi_covg_read_measurement | This intrinsic returns a the TVM measurement register value for the msmt_index measurement register. TVMs can read both initial and runtime measurement register values back. |
| 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 [5] 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

- [0] RISC-V Privileged specification github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf
- [1] IETF RFC 9334 Remote ATtestation procedureS (RATS) Architecture datatracker.ietf.org/doc/rfc9334/
- [2] TCG DICE Attestation Architecture, Version 1.00 Revision 0.23 trustedcomputinggroup.org/resource/dice-attestation-architecture/
- [3] DMTF DSP0274 Security Protocol and Data Model (SPDM) Specification, Version 1.2.1 www.dmtf.org/dsp/DSP0274
- [4] TCG Reference Integrity Manifest (RIM) Information Model trustedcomputinggroup.org/resource/tcg-reference-integrity-manifest-rim-information-model/
- [5] RISC-V Supervisor Binary Interface riscv-non-isa/riscv-sbi-doc
- [6] RISC-V Debug Specification Standard github.com/riscv/riscv-debug-spec/blob/master/riscv-debug-stable.pdf
- [7] RISC-V Zero-Trust Platform Security Model docs.google.com/document/d/1TRHhsGiB5W4K8M7I4e-f40mOPeRtB9sv/edit#heading=h.gjdgxs
- [8] Trusted Computing Group (TCG) Glossary, Version 1.1 Revision 1.0 trustedcomputinggroup.org/resource/tcg-glossary/
- [9] The RISC-V Advanced Interrupt Architecture Document v0.2.1-draft <https://github.com/riscv/riscv-aia/releases>
- [10] The RISC-V Nested Acceleration ("NACL") extension[github.com/riscv-non-isa/riscv-sbi-doc/releases/download/v2.0-rc8/riscv-sbi.pdf]
- [11] IETF Entity Attestation Token (EAT) <https://github.com/ietf-rats-wg/eat>
- [12] Concise Binary Object Representation <https://datatracker.ietf.org/doc/rfc8949/>
- [13] CBOR Web Token <https://datatracker.ietf.org/doc/rfc8392/>
- [14] Unprotected CWT Claims Sets <https://datatracker.ietf.org/doc/draft-ietf-rats-uccs/>
- [15] CBOR Object Signing and Encryption <https://datatracker.ietf.org/doc/rfc9052/>
- [16] IANA Hash Function Textual Names <https://www.iana.org/assignments/hash-function-text-names/hash-function-text-names.xhtml>
- [17] TCG PC Client Platform Profile trustedcomputinggroup.org/resource/pc-client-specific-platform-firmware-profile-specification/
- [18] X.509 Certificate Profile <https://www.rfc-editor.org/rfc/rfc5280>
- [19] X.509 Algorithms for DSA and ECDSA <https://datatracker.ietf.org/doc/rfc5758/>
- [20] RISC-V Supervisor Domain Access Protection[github.com/riscv/riscv-smmmt/releases/download/v1.0.4/smmmt-spec.pdf]
- [21] RISC-V Platform Security Model[github.com/riscv-non-isa/riscv-security-model/releases/download/0.1/riscv-platform-security-model.pdf]
- [22] RISC-V CoVE-IO[github.com/riscv-non-isa/riscv-ap-tee-io/releases/download/v0.1.0/riscv-

