RISC-V Trusted Execution State Extension

Release 17 Nov 2020
commit:fa79e49a852f86575841beae321f1f65ab6e03b6

Mark Hill (mark.hill@huawei.com)

Jan 11, 2021

CONTENTS:

CHAPTER

ONE

RATIONALE AND SCOPE

Although Trusted Execution Environments (TEEs) can be built using the standard RISC-V Physical Memory Protection (PMP) scheme with machine mode as the trusted state, this approach has some limitations:

- Machine mode must manage the lower privilege tasks, all exceptions/interrupts and device drivers as well as managing secret data and providing trusted services. This violates the security *Principle of Least Privilege*, for example, key management code does not need access to OS state and the OS should not have direct access to secret state. Also, as the size of the trusted code increases the risk of security loopholes increases and there is a larger code base from which the gadgets used in Return/Jump Oriented Programming (ROP/JOP) attacks can be constructed. It also becomes less likely formal security proofs can be applied. In the *Security IC Platform Protection Profile (PP0084)* spec this is the threat mode defined as **T-AbuseFunc**.
- Transition between insecure and secure domains is inevitably heavyweight as it must be built on a syscall infrastructure based on the ecall instruction.

In addition, because in the standard PMP user executable code can always be run in m-mode it is not possible to enforce checks that m-mode only runs trusted/signed code. This makes the TEE susceptible to privilege escalation attacks where a software or physical attack can cause a flip to a higher privilege level and then run the attacker's code at that higher privilege level. This issue can be addressed by implementation and appropriate configuration of an enhanced PMP (ePMP) extension (which is both orthogonal and complementary to this extension) but is also fundamentally addressed in this extension by always forbidding execution in trusted state of code which is not explicitly tagged as trusted.

The purpose of this optional extension is to address these issues and provide stronger protection and quicker handling of secret data.

This extension does not provide protection against denial of service attacks by untrusted machine mode on the Trusted Execution State; any actor with machine level privileges has many ways to crash the core. To resolve this issue either the trustworthiness of the code running in Untrusted Execution needs to be raised, for example, using a software TEE, or the secure world must reside in a dedicated hardware enclave.

An additional aim is to minimize the additional state and mechanisms added to the architecture. Simpler solutions are more reliable, less likely to be prone to physical/software attacks, more aligned to the RISC-V philosophy and thus more likely to be adopted by the RISC-V community.

This extension is primarily aimed at protecting low to medium value assets, it is still anticipated that high value assets will be protected by a physically separate security domain.

CHAPTER

TWO

SPECIFICATION

A new execution state is defined, Trusted Execution State (TES), which is orthogonal to the privilege levels implemented in the core:

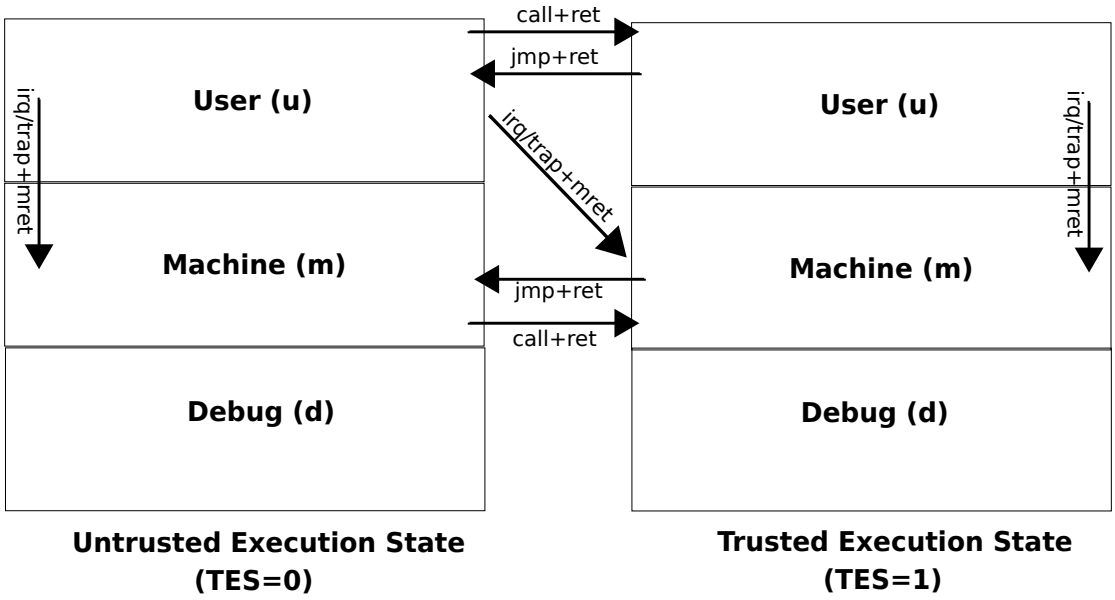

Fig. 1: **Trusted execution states and privilege levels**

When the core is operating in Trusted Execution State (TES=1) it has access to a set of trusted control registers and memory regions that can **only** be accessed when in this state; access from Untrusted Execution State (TES=0) is never permitted, even if running at a higher privilege level. For example, untrusted machine mode is never permitted to access trusted user state. Access to untrusted data from Trusted Execution State is permitted provided that the permission checks defined by the Base ISA are met. For example, trusted user mode only has the same access privileges to an untrusted data region as untrusted user mode. However, when in Trusted Execution State it is never permitted to execute code from untrusted regions of the address space because of the security risks of sharing code between untrusted and trusted execution.

The following specification describes how these rules are enforced from reset into a trusted state through all transitions from, and then back to, trusted state either at the same privilege level (horizontal transitions) or, via interrupts and exception handling, with a change in privilege level.

At reset Trusted Execution State is enabled (TES=1) and all instructions are fetched from a trusted memory region. Trusted memory regions are identified by a bit (T) in a new custom CSR array, `pmptectl`, which contains an 8-bit record per PMP entry (XLEN/8 entries in every XLEN-bit CSR). In the base specification only bit 0 of each record is defined, the remaining bits are read-only-zero (ROZ). Thus the T-bit for entry n is located at `pmptectlX[Y]` where `X=(8n/XLEN)` and `Y=(8n % XLEN)`.

Because trusted execution must start in a trusted memory region, reset hardware must initialize the PMP so that the reset PC matches an entry in the PMP with the T bit set. Additional PMP entries can then be programmed to create other trusted memory regions by setting the T-bit associated with that entry. Constraining the intial boot to

Fig. 2: **Configuration record for a pmptectl entry.**

a limited region of the address space also gives the additional security benefit of reducing the risk of an attack on the reset vector.

The T-bits can only be set or cleared when the processor is running with TES=1.

When TES=0 and the `pmptectl.T` for a PMP entry is set the fields `pmpcfg.A`, `pmpaddr` are `pmptectl.T` are m-mode read-only (MRO), all other fields, including any custom PMP fields are m-mode read-only zero (MROZ).

Note: *Rationale:* Granting read-only access when TES=0 allows untrusted m-mode to discover the PMP entries and memory regions that have been reserved for trusted use, hereby improving software maintainability. Other fields are hidden to make it harder for untrusted code to deduce the usage of the trusted regions. Having a shared PMP for trusted and untrusted code gives greater flexibility as it allows software architects to control the allocation of entries between trusted and untrusted based on the use model.

When TES=0 and the PMP entry's `pmptectl.T=0`: `pmpcfg`, `pmpaddr` plus any other custom fields may be read/written by code executing in m-mode (in accordance with the PMP lock behavior defined by the Base ISA). However, the `pmptectl` array can never be modified when TES=0.

When TES=1 all code executed must be from a trusted memory region, although it is permitted to make both trusted and untrusted data accesses (provided that the RW permission bits in the PMP configuration permit it). When TES=0 all RWX permissions are revoked for trusted regions and attempts to access them result in PMP faults.

Note: *Nomenclature:* This specification uses the pseudo function `pmp_lookup(addr)` to represent the lookup of address, `addr`, in the PMP. The function returns a structure containing all the CSR state associated with that PMP entry. For example, `pmp_lookup(pc).pmptectl.T`, refers to the `T` field of the `pmptectl` record associated with the PMP entry which matches the program counter, `pc`.

Throughout this specification the term *privilege level* refers to the standard RISC-V privilege levels which, in the base extension, are restricted to machine (m) and user (u) mode.

Note: *Implementation:* The architecture expects an implementation to signal the trust status of each memory transaction on the bus. The recommendation is that the core signals the trust status of the requesting instruction, `pmp_lookup(pc).pmptectl.T`, rather than that of the memory access, `pmp_lookup(data_addr).pmptectl.T` (for a load/store operation on `data_addr`). This allows peripherals to implement different behaviour for trusted and untrusted accesses to the same location, for example, to gate access to a trusted configuration register in a peripheral. To enhance security further implementations may also report `pmp_lookup(data_addr).pmptectl.T` and the privilege level at which the requesting instruction was executed. Further details of this signaling are implementation/bus protocol specific and outside the scope of this specification.

The PMP lookup scheme differs slightly from that in the Base ISA as matches for trusted regions are prioritised over untrusted ones. Searches for a trusted (`T=1`) and an untrusted (`T=0`) match proceed in parallel with entries considered disabled (`pmpcfg.A=OFF`) if `pmptectl.T` does not match `T`. If a trusted match is found then it is a trusted region, if not, and there is an untrusted match, is an untrusted region. Regardless of the current privilege or TES setting, if no match is found a PMP fault is generated.

If entry **n** has `pmpcfg.A=TOR` and `pmptectl.T=1` then entry **n-1** is read-only when running in Untrusted

Execution State (TES=0). This prevents untrusted code modifying the base address of a trusted region specified using the Bottom/Top of Range (BOR/TOR) scheme. This is similar to the Base ISA locking of entry **n-1** when entry **n** has `pmpcfg.A=TOR` and `pmpcfg.L`.

Note: *Rationale:* This scheme makes it impossible for any change to an untrusted entry to remove the trusted status of a region, it also has the property that PMP entries can be maintained in physical address order giving maximum opportunity to use a PMP entry as both a BOR for the next entry and a TOR for the current entry, helping to make efficient use of a limited resource.

Note: *Implementation:* For implementations of RISC-V which provide cache maintenance operations (CMOs) it must be guaranteed that any destructive CMOs that can be performed when in Untrusted Execution State (TES=0) cannot cause corruption of trusted state. As CMOs for RISC-V have not yet been standardised a precise definition of behavior cannot be given. Suggested behavior is all destructive/invalidating CMOs that operate on:

- Addresses when TES=0, should generate a PMP fault if `pmptectl.T` is set.
- Whole cache, way or set index, must be restricted to Trusted Execution State.

When caches are implemented, any code handing trusted regions of memory back for untrusted use should scrub the memory and then use CMOs to flush the caches so that no trusted data remains. This ensures that any untrusted invalidation cannot undo the scrubbing and reveal trusted data.

If implementing writeback caches in systems where the trust status of a transaction is reported on the bus (in order to support trust aware IOPMP/IOMMUS for example) a mechanism is needed to ensure the correct trust status for dirty lines is reported when they are evicted. If the above guidelines are enforced then this can either be achieved by looking up the address of the line in the PMP or by storing the T-bit along side the other cache line attributes. Further details of this signaling are implementation/bus protocol specific and outside the scope of this specification.

2.1 Leaving trusted execution without changing privilege level

Transitioning from trusted to untrusted execution is either:

- By a jump instruction (`jal`, `jalr`, `c.j`, `c.jr`) to a target PC in an untrusted PMP region (`pmp_lookup(targetpc).pmptectl.T==0`) with a destination/link register which is `x0`/`zero` or not present in the encoding. This includes the pseudo ops `j` and `ret`.
- By an `mret` or a `tret` instruction (defined later) to an address in an untrusted PMP region.

They cause an immediate revocation of trust (TES $1 \rightarrow 0$) so the instruction at the target PC, and all subsequent instructions, are executed as untrusted, `TES' = TES AND pmp_lookup(pc').pmptectl.T`. For any oher instruction there is no revocation when the next PC is in an untrusted region, `pmp_lookup(targetpc).pmptectl.T==0`, and execution of the next PC will cause a PMP fault. This includes all other jump variants and sequential execution crossing a PMP boundary from a trusted to an untrusted region. In particular, attempting to call an untrusted function directly from trusted code will cause an exception.

When in Untrusted Execution State (TES=0) any attempt to execute code or access data which matches a trusted PMP entry will cause an exception and report an access fault with the same `mcause` as a conventional PMP fault. This includes sequential execution crossing a PMP boundary from an untrusted region to a trusted one.

2.2 Entering trusted execution without changing privilege level

Once in an Untrusted Execution State (TES=0) entry to Trusted Execution State (TES=1) is achieved by calling into the Trusted Execution State Vector (TESVEC) table. The table should be mapped to a PMP entry with the T bit set and should be executable by the current privilege mode. If the T bit is not set, or there is no executable permission for the current privilege mode, a PMP fault occurs and an exception is taken.

Entry into the TESVEC must be through a call instruction, a return (`ret`) or an `mret`. The call can either be a direct (`jal`) or indirect (`jalr`) jump-and-link instruction with `ra` as the link register. Entry into the TESVEC via any other instruction type will cause an instruction fault with the machine exception PC set to the address of the instruction and the mahcine trap value set to the address in the TESVEC to which the jump was attempted. Calls to the TESVEC must use `ra` as the link register to ensure that on entry to trusted code the `ra` register is guaranteed to contain the instruction immediately after the secure function call. Without this check an untrusted actor could set the `ra` register to an arbitrary address (including anywhere in trusted code) then jump (without a link or with an alternate link register) to the TESVEC. On exit from the secure function execution would then continue from the chosen location. This would not cause privilege escalation as Trusted Execution Status will be immediately revoked if the code is untrusted, but might permit ROP/JOP style attacks where an untrusted actor could cause a jump to a trusted code fragment/gadget when the secure function returns.

Additional security for the transitions in to Trusted Execution State can be achieved by requiring landing markers at all entry points. The marker instruction is `c.addi zero, 0x15` (opcode 0x0055). This is a standard RISC-V 16-bit instruction with no effect. If the target instruction is incorrect then an *Illegal instruction* fault is taken.

This feature is enabled using the *Entry Marker Enable*, `eme`, bit in the new *Trusted Machine Execution State Control Register*, `tmescr`.

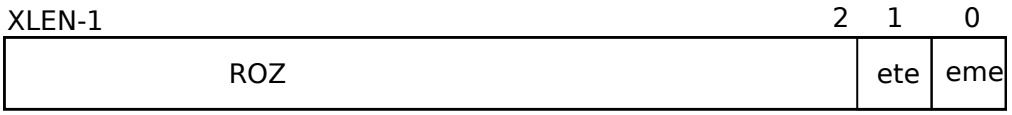

Fig. 3: **Trusted Machine Execution State (tmescr)**

This feature provides an additional level of security in the scenario that the contents of the TESVEC has been corrupted/compromised either by physical or software attack. The main disadvantage is that a secure function can no longer be a pure C function and would either need a hand assembled pre-amble or compiler support for the marker insertion.

The purpose of the `ete` field is described later in this section.

All TESVEC Records (TRs) are 8 bytes long and aligned to 8 bytes. The address of the first TR is programmed in the custom CSR `tmesvec`, the top of the TESVEC is set in `tmestop` and points to the address immediately above the TR in the TESVEC. If the (unsigned) value in `tmestop` is less than or equal to `tmesvec` then the TESVEC table is empty and no transition to Trusted Execution State is possible through this mechanism. Note that trusted execution can still be entered through trusted exception/interrupt handlers, see the sections on *Exceptions* and *Untrusted Interrupts* for details.

The `tmesvec` and `tmestop` CSRs are read-only when TES=0 and their bottom three bits are *Reserved* (writes ignored, read as zero). When a jump occurs to an address in the TESVEC the address is 8 byte aligned (bottom 3 bits of the jump address are ignored) and that TESVEC Record (TR) is fetched. The fetch must be treated as a trusted instruction fetch for the purposes of PMP lookup and memory access as the TESVEC must be placed in trusted memory to prevent modification by untrusted code.

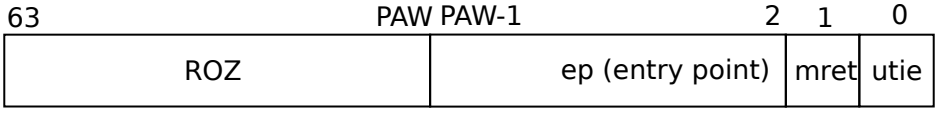

Fig. 4: **TESVEC Record (TR)**

Bits (PAW-1):2 of a TR specify bits (PAW-2):1 of an entry point into trusted code where PAW is the Physical Address Width (PAW=32 for a RV32IMC core). Bit 0 of a TR is used to control the masking of untrusted interrupts, bit 1 (`TR.mret`) is used to indicate that the entry point is only valid as the target of an `mret` instruction. When an `mret` to the TESVEC occurs the content of `mstatus.pll` is ignored and no change in privilege level occurs. Performing an `mret` to a TR which does not have this bit set or a call or return to an entry which does have this bit set causes an illegal instruction fault. More details of this mechanism can be found in the section on *Untrusted Interrupts*.

Note: *Usage:* Although gcc's default function alignment is 4 bytes it is possible to reduce this to 2 bytes as a compilation option, therefore functions used as TESVEC entries should append `__attribute__((aligned=4))` to the function declaration to guarantee suitable alignment of the entry point.

If the fetch of the TR is successful the core enters Trusted Execution State (TES=1) and two new custom CSRs, the *Trusted Execution State Entry Point Record* (`tmesepr`) and the *Trusted Execution State Entry Point Status* (`tmeseps`) are updated. These control registers are modifiable in trusted m-mode (PL=M,TES=1). Two custom user CSRs are also allocated, `tesepr` amd `teseps`, which provide u-mode read-only views of the `tmesepr` and `tmeseps` respectively.

Note: *Rationale:* The `teseps` and `tesepr` are writeable in m-mode so that they can be saved and restored during a context switch or to preserve User mode usage if a TESVEC call must be made during execution of an interrupt/exception handler.

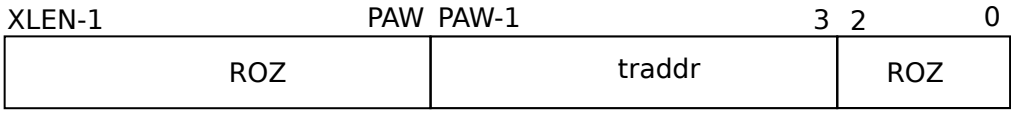

Fig. 5: **Trusted Execution State Entry Point Record (tesepr)**

The `tesepr.traddr` field is updated with bits (PAW-1):3 of the TR's address. The main reason for capturing the TR's address in `tesepr` is that it allows multiple TRs to share a trusted entry point (same value of traddr field) while still being able to differentiate which TR was used to enter it. See the section on *Untrusted OS with secure functions and tasks* for an illustration of when this is useful.

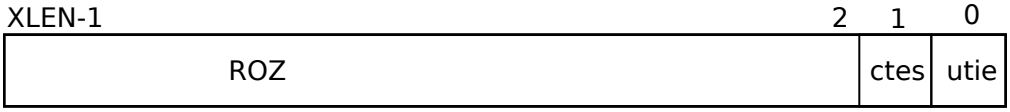

Fig. 6: **Trusted Execution State Entry Point Status (teseps)**

The `teseps.utie` field (*untrusted interrupt enable*) is updated to the value of `TR.utie`. The `teseps.utie` can be modified when in Trusted Execution Machine Mode (PL=M,TES=1) to enable/disable delivery of untrusted interrupts during trusted execution.

The `teseps.ctes` field (*caller TES*) is updated to the Trusted Execution State of the call into the TESVEC.

A trusted return instruction (`tret`) is defined which jumps to the contents of the `ra` register and continues execution from that point with `TES = teseps.ctes`.

This allows a secure routine that is returning to an untrusted caller to force revocation of trust (TES $1 \rightarrow 0$) and thus prevent a malicious attacker continuing to execute trusted code.

If `teseps.ctes` is set, continued trusted execution is permitted provided that the return address is within a trusted memory region, in summary, `TES' = TES AND pmp_lookup(ra).pmptectl.T AND teseps.ctes`. This behavior allows a trusted function entered by the TESVEC to be called by trusted or untrusted code and use a `tret` in both cases.

If a `tret` is executed when TES=0 an illegal instruction trap occurs.

Use of the instruction is optional, a standard `ret` can also be used as any jump to untrusted memory causes all subsequent instructions, including the jump target, to be executed as untrusted code (TES=0). However, for additional security, it is possible to enforce that all function returns from trusted to untrusted code use the `tret` instruction. This feature is enabled by setting *Enforce Tret Enable*, `tmescr.ete`, attempting to use a `ret` to exit trusted execution will then cause an illegal instruction fault.

If `ret` is used for returning from a TESVEC entry function some care is needed to ensure that an untrusted call to the TESVEC cannot be placed at a PMP boundary between untrusted and trusted code such that the next instruction (the return address for the call) would be to valid trusted code, which would allow that trusted code to be executed without going through the TESVEC. The recommended way to prevent this to place an unimplemented instruction (e.g. the pseudo assembler op `unimp`) at the start of any trusted code section. If a `tret` is used this situation cannot arise because `teseps.ctes` will be zero and execution will be forced to leave trusted execution regardless of `pmp_lookup(ra).pmptectl.T`.

Note: *Toolchain:* Support for `tret` is the one aspect of the specification that would benefit from compiler support. It is suggested that a new function attribute, `tes_entry`, is provided which generates a `tret` instead of a `ret` at all function return points. It should also clear `a1` and, potentially, `a0`, if they are not needed for returning a function result. Without this support it may be desirable to ensure that all trusted functions (in an RV32 machine) have a `int64_t` result type. This will ensure that no use of `a0` or `a1` as temporaries while in Trusted Execution State will be live in the register file on return from a trusted function. The attribute should also ensure that the function is aligned to a 4 byte boundary to avoid having to apply this attribute explicitly and could also be used to indicate when a function requires an Entry Marker if the `tmescr.eme` feature is enabled.

Trusted functions which return via a `tret` must only be called via the TESVEC. Calling then directly from trusted code could cause an unexpected exit from trusted execution when the `tret` instruction is executed. For trusted functions that need to be accessed from both trusted and untrusted code it is recommended that they do not use a `tret`. Depending on the performance/security trade off, they can either be placed directly in the TESVEC (if `tmescr.ete` is not set) and return via a conventional `ret`, or a trampoline function can be used in the TESVEC which calls the trusted function and then executes a `tret`

Calling via the TESVEC when already in a Trusted Execution State is permitted, so for example, a function pointer to a TESVEC entry can be passed around in a structure or as a function parameter and called either from trusted or untrusted code.

Transitions into Trusted Execution State via the TESVEC and all transitions out of trusted execution state are horizontal and involve no change in privilege level. For example, a user mode untrusted task might call a trusted function with access to keys for signing/authentication purposes but which can only access CSRs and memory with user-mode permissions.

The mechanism for entering trusted execution via the TESVEC is summarized below.

2.3 General purpose register management

On transition from trusted to untrusted execution (TES $1 \rightarrow 0$) subsets of the general purpose register file are automatically cleared by hardware. Which general purpose registers are affected depends upon the type of jump performed. The purpose is to limit the risk of secret data leaking from Trusted Execution State while still supporting efficient inter-calling between the trusted and untrusted states. No automatic clearing of general purpose registers occurs when entering or returning from interrupt and exception handlers.

Note: *Scope:* The register selections are based on the standard ABI assignment as defined in *The RISC-V Instruction Set : User Manual*. An alternate Embedded ABI which, although compatible with RV32I/R64I implementations, is primarily for RV32E machines (with only 16 general purpose registers) is not supported and is out of the scope of this specification.

How the jump type affects which registers are cleared is summarised below.

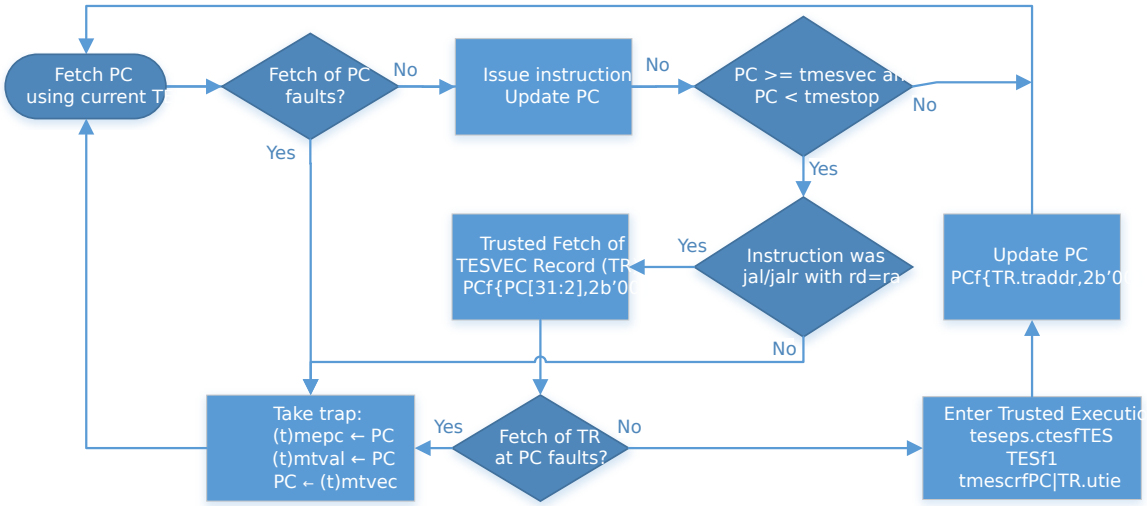

Fig. 7: **Entering Trusted Execution via the TESVEC**

Table 1: Summary of auto register clearing instructions on transition from trusted to untrusted (TES 1 → 0)

Pseudo	Instructions	General Purpose Registers Auto cleared
`j`	`jal zero, offset`	When `rs!=ra`, all ABI defined Temporary and Saved Registers are cleared.
	`c.j offset`	
	`jalr zero, rs, offset`	
	`c.jr rs`	
`ret`	`jalr zero, ra, 0`	All ABI defined Temporary and Function Argumment Registers, except for those used for Function Return (a0-a1), are cleared.
	`c.jr ra`	
`tret`	`tret`	
`mret`	`mret`	No clearing of registers occurs
Other		Fault: *Illegal TES transition*

The standard ABI categorises the registers as follows:

- *Function Arguments* : `a0-a7/x10-x17`
- *Return Registers* : `a0-a1/x10-x11`
- *Saved Registers* : `s0-s11/x8-x9,x18-x27`
- *Temporary Registers* : `t0-t6/x5-x7,x28-x31`

The ABI defined Saved Registers are not reset on a return to untrusted execution because an ABI compliant trusted function should restore the content of all Saved Registers to the (untrusted) values they had on entry from the untrusted caller.

When jumping from trusted to untrusted code (with `rs` != `ra`), Function Arguments are retained to supporting passing arguments to an untrusted function. See *Calling untrusted/insecure code* for further details of how to safely perform Untrusted Function Calls.

The TES extension maintains Untrusted and Trusted versions of the *Stack Pointer*. If an instruction references the *Stack Pointer* (`x2`) when TES=0 the *Untrusted Stack Pointer* is used and when TES=1 the *Trusted Stack Pointer* is used. In addition there is a new custom CSR defined, *Trusted view of Untrusted Stack Pointer* (`tusp`), with URW permissions when TES=1 and URO when TES=0. This allows trusted code to initialize or update the untrusted stack before transitioning to Untrusted Execution.

Note: *Usage:* This switching of the stack pointer does limit the number of parameters that can be passed between trusted/untrusted functions while still maintaining ABI compliance as all parameters must be passed using the eight Function Argument Registers. It is still possible for a trusted caller/callee to place/retrieve additional parameters on/from the stack during entry to a function (via the `tusp` CSR) but this requires a handcrafted call/entry sequence.

To prevent untrusted code from subverting trusted code by modifying the location of global/thread state prior to a secure function call, both Trusted and Untrusted versions of the *Global Pointer* (`x3/gp`) and *Thread Pointer* (`x4/tp`) are provided. Two additional CSRs are provided, `tugp` for the `gp` and `tutp` for the `tp`, to give trusted views of the untrusted versions of these registers, these CSRs have URW permissions when TES=1 and URO when TES=0.

2.4 Exceptions

At reset all exceptions default to indirecting through `tmtvec`, a new custom trusted version of the `mtvec` CSR. Exceptions can be delegated to untrusted handling using the trusted exception delegation register `tmedeleg`. This new custom register uses the same mapping of bits to exceptions as the `medeleg` register defined in the *RISC-V Instruction Manual, Volume II: Privilege Architecture*.

The `tmedeleg` remains m-mode read-only (MRO) when TES=0 so that untrusted code can check which exceptions are delegated to untrusted handling.

When operating with TES=1 all exceptions vector through `tmtvec` regardless of any `tmedeleg` settings. When TES=0 exceptions indirect through `mtvec` or `tmtvec` depending on the setting of `tmedeleg`.

When vectoring through `tmtvec` the details of the exception are captured in a new bank of registers which are trusted versions of the Base ISA registers used when exceptions vector through `mtvec`. The PC of the exception is stored in `tmepc`, the cause is stored in `tmcause` and the trap value is stored in `tmtval`. There is also a a new trusted variant of `mstatus` called `tmstatus`:

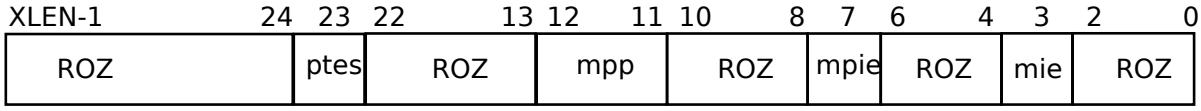

Fig. 8: **Trusted Machine Status (tmstatus)**

On an exception or interrupt which vectors through `tmtvec` the `tmstatus` is updated as follows:

Table 2: Summary `tmstatus` updates on an exception/imterrupt vecotred through `tmtvec`

New `tmstatus`	On exception set to . . .
`mpie`	Previous value of `tmstatus.mie`
`mie`	Zero
`mpp`	Previous privilege level
`ptes`	Previous trusted execution state

Once the exception status has been recorded, TES is set to 1 and execution continues by vectoring through `tmtvec`.

When vectoring delegated untrusted exceptions through `mtvec` the `mepc`, `mstatus` and `mcause` are updated as defined in the Base ISA.

If an `mret` occurs when in Trusted Execution State, TES is updated to `tmstatus.ptes` and is checked against `pmp_lookup(tmepc).pmptectl.T`. If they do not match an *Instruction Fault* of the type *Illegal TES Transition* is generated. The privilege level is set to `tmstatus.mpp`, the `tmstatus.mie` is set to `tmstatus.mpie`, `tmstatus.pie` is set to one, and `tmstatus.mpp` is set to u-mode (0).

If `tmtvec` does not point to trusted memory a PMP fault will occur and the core will follow the implementation defined behaviour for a fault which occurs in a handler.

A new XLEN-bits wide CSR scratch register, `tmscratch`, is also defined for trusted m-mode use (TES=1,PL=M). For example, to free up a general purpose register for use in a trusted handler.

Asynchronous exceptions should not be delivered while trusted interrupts are masked as an asynchronous exception caused by an untrusted operations could corrupt trusted operation. This may require the core to delay taking an asynchronous exception until interrupts are unmasked.

2.5 Trusted Interrupts

At reset all interrupts default to being trusted interrupts. When a trusted interrupt fires the processor transitions into Trusted Execution State (if TES was 0). Execution continues from an address based on `tmtvec` using the same direct/vector mode schemes that are defined for `mtvec`. If the vector entry does not map to trusted memory a PMP fault will occur and handled in the same way as any other fault which occurs in a handler.

When vectoring through `tmtvec` the details of the exception are captured in a bank of registers which are trusted versions of the Base ISA registers used when exception vector through `mtvec`. See the sections on *Exceptions* for more details of these. Trusted interrupts can be masked by clearing `tmstatus.mie` but can never be masked by setting `mstatus.mie`.

When a trusted interrupt handler completes and execution resumes via an `mret` the TES is set to `tmstatus.ptes` and is checked against `pmp_lookup(tmepc).pmptectl.T`. If they do not match an *Instruction Fault* of the type *Illegal TES Transition* is generated.

Therefore, if a trusted interrupt is taken when operating in Trusted Execution State the core will run the handler and return to the interrupted code without leaving Trusted Execution State.

2.6 Untrusted Interrupts

Trusted code can delegate interrupts for handling by untrusted handlers using the *Trusted Interrupt Delegation* registers, `tmidelegX`, where `X` depends on the number of interrupt sources. An interrupt can be delegated to untrusted handling by setting a bit in this register array, the `tmidelegX[Y]` bit controls the trust status of the interrupt with the id N where `N=X\*XLEN+Y`. In subsequent text this delegation bit is referred to as `tmideleg[N]`.

When operating in an Untrusted Execution State (TES=0) untrusted interrupts vector through `mtvec` and behave according to the *RISC-V Instruction Manual, Volume II: Privilege Architecture*. Both trusted and untrusted interrupts can be taken immediately provided that all masking/priority constraints defined in the Base ISA and interrupt controller are met.

If `mtvec` points to trusted memory a PMP fault will occur and follow the implementation defined behaviour for any fault which occurs in a handler.

However, when an untrusted interrupt occurs when in Trusted Execution State the behaviour depends upon `teseps.utie` (Untrusted Interrupt Enable). If this bit is clear, untrusted interrupts are masked and will not be delivered until untrusted execution resumes. If it is set, untrusted interrupts can pre-empt trusted execution (assuming all the standard ISA and interrupt controller conditions for the interrupt are met) but must first pass through a trusted pre-handler vectored off `tmtvec` and return to trusted execution via a trusted pre-handler. More details have how his could be done are given later this secton.

Untrusted interrupts can be masked regardless of the current Trusted Execution State by clearing `mstatus.mie`, in addition, untrusted interrupts will always be masked during trusted execution if `tmstatus.mie` is clear.

A summary of how `teseps.utie` and `tmideleg[N]` effect an interrupts handling are summarised below.

Table 3: Summary of interrupt handling options

tmideleg[N]	utie	TES	Interrupt handling
0	0 or 1	0 or 1	Vector off **tmtvec** immediately if `tmstatus.mie` set
1	0	0	Vector off **mtvec** immediately if `mstatus.mie` set
1	0	1	Vector off **mtvec** delayed until TES $1 \rightarrow 0$
1	1	0	Vector off **mtvec** immediately if `mstatus.mie` set
1	1	1	Vector off **tmtvec** immediately to a trusted pre-handler if `mstatus.mie` and `tmstatus.mie` set

In a system with untrusted interrupts the approach chosen to handle them is a trade-off between complexity, security and latency and may vary depending upon the interrupt source and expected duration of the trusted operation, suggested approaches are:

- Set `tmideleg[N]=1` and `teseps.utie=0` so that handling of the untrusted interrupt N is delayed until the processor returns to Untrusted Execution (TES=0). This is the safest and simplest solution but might cause unacceptably high interrupt latencies for untrusted interrupts. This could be mitigated by limiting the length of time spent in trusted execution, for example, by breaking down long secure operations into sequences of shorter ones. This approach could be further refined by the secure operations polling the pending interrupts and yielding/returning to untrusted state if an untrusted interrupt with hard real time constraints arrives.

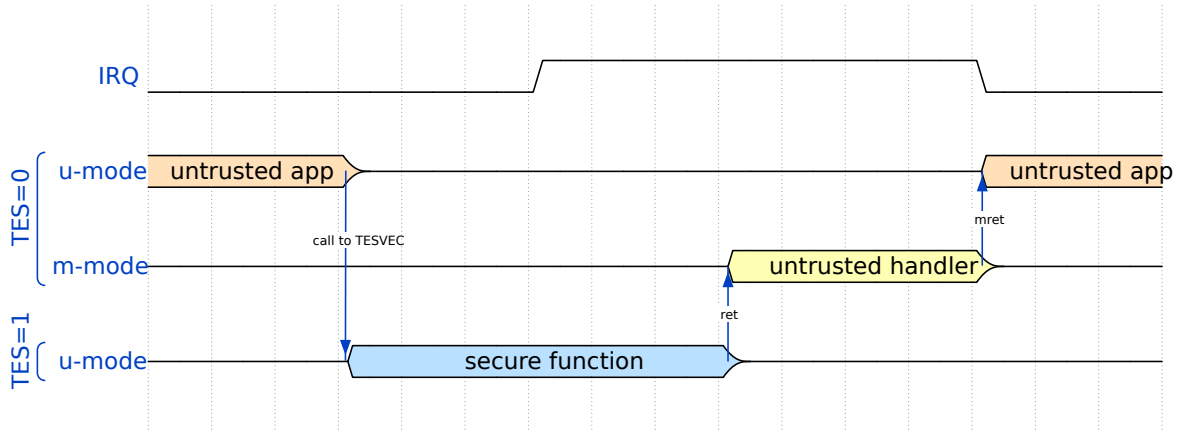

Fig. 9: **Delayed untrusted interrupt (teseps.utie=0, tmideleg[N]=1)**

- Set `tmideleg[N]=0`, with `teseps.utie` either 0 or 1, so that interrupt N is always handled by a trusted interrupt handler (vectored of `tmtvec`). This is the best solution for very latency sensitive interrupts provided that the handling is simple enough that its trustworthiness can be assured. Note that in this configuration if the interrupt occurs while TES=0 the interrupt will still be processed by the trusted handler. The main disadvantage is that it adds to the amount of code running in the trusted domain.
- Set `tmideleg[N]=1` and `teseps.utie=1` so that if the interrupt occurs when in an Untrusted Execution State it will be handled by an untrusted handler vectored off `mtvec` unless it is masked by `mstatus.mie`, however, if the core is in a Trusted Execution State the interrupt will vector off `tmtvec`, unless it is masked by either `mstatus.mie` or `tmstatus.mie`, and will be processed by a trusted pre-handler. The pre-handler:

  1. Saves the general purpose register state (`x1-x31`).
  2. Saves the `mstatus`, sets the *previous privilege* (`mstatus.mpp`) to machine mode and clears *previous interrupt enable* (`mstatus.mpie`), so that when the untrusted handler performs an `mret` interrupts will still be masked and the core will remain in machine mode.
  3. Set `mepc` to the address of a TESVEC record assigned to performing returns from untrusted handlers. This entry will have `TR.mret` set to mark it as a valid target of an `mret` instruction.

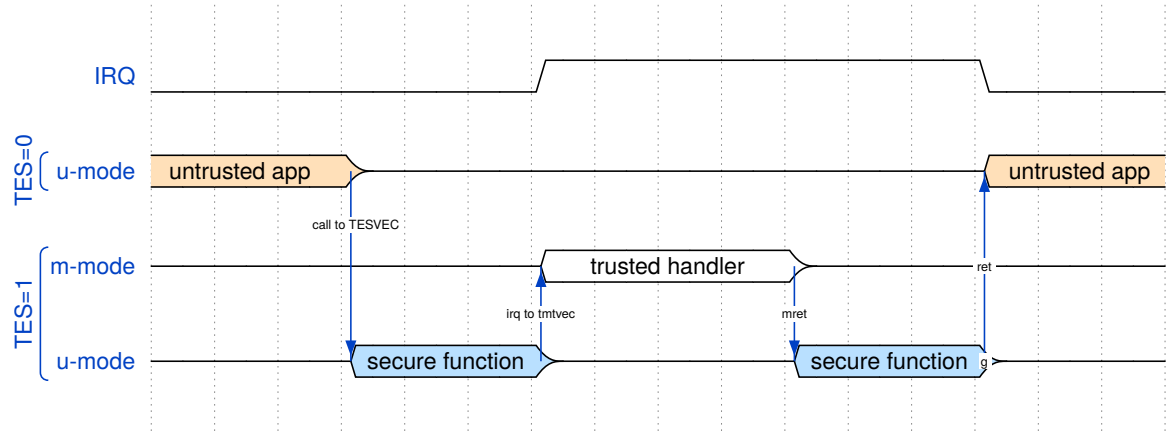

Fig. 10: **Low Latency Trusted Interrupt (teseps.utie=X, tmideleg[N]=0)**

4. Clear the Function Argument Registers, note that the Temporary and Saved Registers will be cleared automatically by the jump in the next stage.
5. Jumps to the untrusted handler. This untrusted handler can behave exactly as it would if the interrupt was taken while executing in an Untrusted Execution State, however, optimisations are possible if the pre-handler is known to have run as the machine context prior to the interrupt will already have been saved. For example, the handler could be setup with two entry points, the main untrusted entry point which spills to the stack the any untrusted state that needs to be preserved (and sets a register to indicate that a restore is needed) and a (later) trusted entry point (which will therefore leave the restore indicating register clear). At the end of the handler the restore sequence is skipped if the trusted entry point was used.

When the untrusted handler performs an `mret` the routine for performing returns from untrusted handlers will be entered. This post-handler routine will restore the trusted state and resume execution of the trusted code by performing another `mret` which will restore the machine state saved in `tmepc` and `tmstatus` when the trusted pre-handler was entered.

This option is illustrated below.

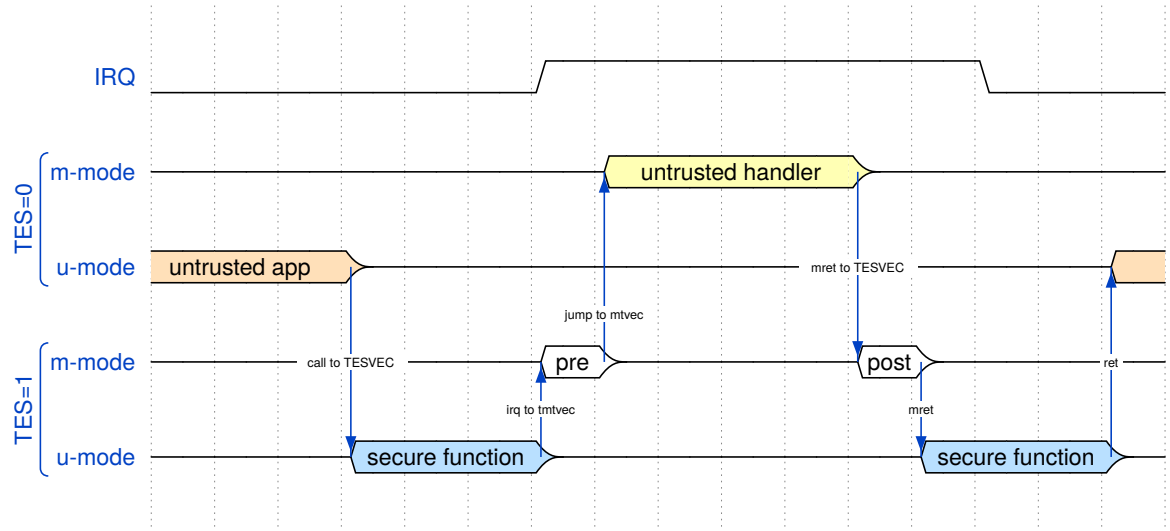

Fig. 11: **Untrusted handler with trusted pre and post handler (teseps.utie=1, tmideleg[N]=1)**

Note that when an `mret` to the TESVEC occurs the content of `mstatus.mpp` is ignored and no change in privilege level occurs.

An alternate implementation which avoids executing an `mret` to the TESVEC is to place the body of the untrusted handler in a separate function. This function can then be called directly by the top-level untrusted interrupt handler

or, if the interrupt occurs during trusted execution, use the jump and return via the TESVEC scheme described in *Calling untrusted/insecure code*.

2.7 Core Level Interrupt Controller (CLIC) support

Note: The CLIC (Core Level Interrupt Controller) specification is currently only in draft form and changes are ongoing. This section is therefore subject to change.

The Core-Local Interrupt Controller (CLIC) is designed to provide low-latency, vectored, pre-emptive interrupts for RISC-V systems. The latest specification for the CLIC can be found here: https://github.com/riscv/riscv-fast-interrupt/blob/master/clic.adoc. It is important to read and understand this specification before reading the rest of this section as knowledge of the CLIC specification is assumed.

To reduce design complexity and verification space if both Trusted Execution State and the CLIC are implemented: the CLIC is permanently activated, the basic interrupt scheme defined in the Base ISA is not available and the `mip`, `mie`, `mideleg` and `tmideleg` registers are tied to zero.

When enabling both the Trusted Execution State and the CLIC extensions some additional features are needed:

- The bottom 6 bits of `tmtvec` and `mtvec` must be hardwired to `000011`. This forces the CLIC to be enabled and, as per the CLIC specification, requires that the unvectored trap handler is always aligned to 64 bytes.
- A new CSR, the *Trusted Machine Trap Vector Table* (`tmtvt`), is defined through which trusted interrupts are vectored instead of going via `mtvt`. Selection between use of `tmtvec` and `tmtvt` is the same as that for `mtvec` and `mtvt` in the CLIC specification.
- A trusted variant of `mintthresh` (`tmintthresh`) is defined which can be used by trusted handlers to temporarily raise the threshold level for taking interrupts.
- The `tmcause` register is updated in line with the `mcause` register changes defined in the CLIC specification, with the addition that aliases of `ptes`, (and for the Zmulti extension `ptdid`) are also added to reduce context save/restore code:

```
tmcause
Bits   Field         Description
XLEN-1 Interrupt     Interrupt=1, Exception=0
   30  minhv         Hardware vectoring in progress when set
29:28  mpp[1:0]      Previous privilege mode, alias of tmstatus.mpp
   27  mpie          Previous interrupt enable, alias of tmstatus.mpie
26:25  (reserved)
   24  ptes          Previous trusted execution state, tmstatus.ptes alias
23:16  mpil[7:0]     Previous interrupt level
   15  (reserved)
14:12  ptdid         Previous trusted domain id, tmstatus.ptdid alias if␣
↪Zmultit
 11:0  exccode[11:0] Exception/interrupt code
```

- There are no trusted variants of the optional `xscratch` registers defined in the CLIC specifcation.
- By default, all CLIC memory-mapped registers are read-only when in Untrusted Execution State, when in Trusted Execution State the access permissions are those defined in the CLIC specification.
- A new set of custom memory-mapped registers, `clicinttidlg`, are provided which can be used to delegate individual interrupts to untrusted handling. The array is read-only during untrusted execution. The array is mapped at the offset range `0x0A00-0x0BFF` in the m-mode CLIC memory map and contains 1-bit per interrupt (a maximum of 4096 interrupts). The per-interrupt registers (`clicintip`, `clicintie`, `clicintattr` and `clicintctl`) for any interrupt which has been delegated become modifiable in Untrusted Execution State.

- When selecting the next horizontal interrupt to service using `mnxti`, only interrupts mapped to the current Untrusted/Trusted Execution State are considered. When in Trusted Execution State the trap handler table entry returned by a read is an offset from `tmtvt` not `mtvt`, and writes/sets update `mintstatus.mil` and `tmcause.exccode`.
- A custom memory-mapped register, *Untrusted Interrupt Level Shiftt*, `clicutils`, is defined at address 0x0800 in the m-mode CLIC memory map which can be used to reduce the level of all untrusted interrupts. The register is read-only during untrusted execution. The default value is zero which applies no level reduction to untrusted interrupts. The register can be set to any value from 0 to 7 to indicate by how much untrusted levels are shifted. The level of trusted interrupt N uses the standard scheme defined in the CLIC specification: the top `nlbits` of `clicintctl[N]` specify the top bits of the 8-bit level, the lower bits are all set to 1. Here is an example:

```
Trusted Interrupt N Level Calculation:
   CLICINTCTLBITS = 6
   cliccfg.nlbits = 3
   nlbits         = min(CLICINTCTLBITS,cliccfg.nlbits) = 3
   clicintctl[N]  = LLLppp..
   level          = LLL11111
   prority        = ppp11111
```

  For an untrusted interrupt N, the top `clicutils` bits of the the level are set to 0, the next most significant bits are the top `nlbits-min(nlbits,clicutils)` bits of `clicintctl[N]` and the lower bits are set to 1. For example:

```
Untrusted Interrupt N Level calculation:
   CLICINTCTLBITS = 8
   cliccfg.nlbits = 5
   nlbits         = min(CLICINTCTLBITS,cliccfg.nlbits) = 5
   clicutils      = 3
   level bits     = nlbits-min(nlbits,clicutils) = 2
   clicintctl[N]  = LLpppppp
   level          = 000LL111
   prority        = pppppp11

Untrusted Interrupt N Level calculation where clicutils>nlbits:
   CLICINTCTLBITS = 6
   cliccfg.nlbits = 4
   nlbits         = min(CLICINTCTLBITS,cliccfg.nlbits) = 4
   clicutils      = 5
   level bits     = nlbits-min(nlbits,clicutils) = 0
   clicintctl[N]  = pppppp..
   level          = 00000111
   prority        = pppppp11
```

Rules for when an interrupt can be taken are summarised below:

```
Interrupt N type:      Taken if:
   Trusted                 tmstatus.mie AND clicintie[N] AND
                           level > max(mintstatus.mil,tmintthresh)

   Untrusted               tmstatus.mie AND clicintie[N] AND mstatus.mie AND
                           level > max(mintstatus.mil,tmintthresh) AND
                           level > max(mintstatus.mil,mintthresh) AND
                           (TES==0 OR teseps.utie)
```

Note that:

- If trusted and untrusted interrupts have the same level the trusted interrupt are always selected over the untrusted ones. Therefore, the highest level of trusted interrupt can never be blocked by untrusted execution regardless of the setting of `mstatus.mie`, `mintthresh` or `clicutils`.
- For an untrusted interrupt to be activated during trusted execution `teseps.utie` must be set and the

interrupt handling will commence with entry to a trusted pre-handler vectored through `tmtvec` or `tmtvt`.

- When an `mret` is executed in untrusted mode the new CLIC interrupt level is `min(mcause.mpil, 255>>clicutils)`. This prevents an untrusted handler raising the interrupt level above the maximum permissible level for an untrusted interrupt.
- Horizontal TES transitions due to calls/return through the TESVEC have no affect on the current interrupt level so it is possible for untrusted execution to temporarily run at a higher level than `255>>clicutils` if a trusted handler makes a call to an untrusted function.

Table 4: Summary of custom m-mode CLIC memory map

Address Offset	Register	Setting	Permissions		Description
			TES=0	TES=1	
0x0800	clicutils	0 to 7	RO	RW	Untrusted Trusted Interrupt Level Shift
0x0A00-0x0BFF	clicinttidlg[i]	1 bit/interrupt	RO	RW	Trusted Interrupt Delegation

2.8 Summary of State Transitions

Trusted state transitons associated with instruction exection (as opposed to traps and interrupts) is summarized in the table below.

Table 5: Summary of instruction driven Trusted State Transitions

Current trust state		PMPTECTL.T of next PC	Next trust state	
TES	Insn		TES	Auto-clear
X	call/ret/mret TESVEC	1	1	-
		0	Instruction access fault	
	other TESVEC	X	Instruction access fault	
0	any legal	0	0	-
		1	Instruction access fault	
1	tret	ctes	ctes	Yes
		!ctes	Instruction access fault	
	mret	ptes	ptes	No
		!ptes	Instruction access fault	
	j/ret	0	0	Yes
	other		Instruction access fault	
	not tret/mret	1	1	-

2.9 Trap Vector Locking

An additional security feature is the ability to lock the trap vectors so that they cannot me modified until the next reset is performed.

Two lock registers are provided:

- `mtvlock` is used to lock the untrusted trap vector registers, `mtvec`, and if the CLIC is implemented, `mtvt`.
- `tmtvlock` is used to lock the trusted trap vector registers, `tmtvec`, and if the CLIC is implemented, `tmtvt`.

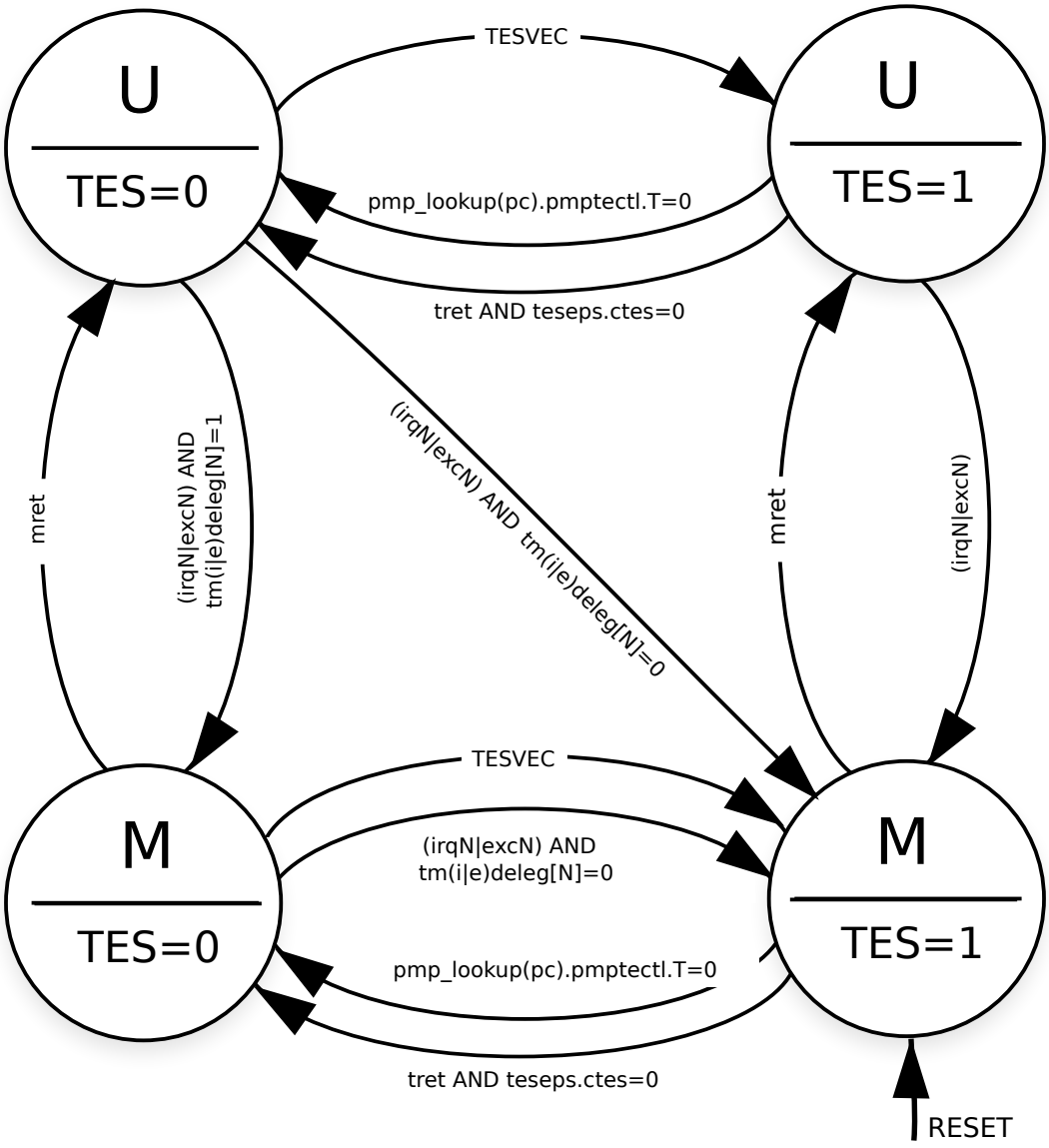

Fig. 12: **Trusted Execution State Transiton Summary**

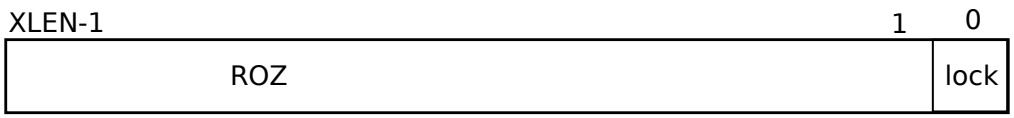

Fig. 13: **Trap Vector Lock Register Format (mtvlock and tmtvlock)**

2.10 Debug

2.10.1 Trusted Self-hosted Debug

A trusted debug monitor running in trusted m-mode and monitoring trusted and untrusted u-mode tasks can set soft breakpoints using the `ebreak` instruction and hard breakpoints using triggers. To enable trusted self-hosted debug mode `tmedeleg[3]` must be cleared so that both trusted and untrusted debug exceptions are vectored through `tmtvec`. When execution of trusted user code resumes (via an `mret`) Trusted Execution State is retained. However if re-entering untrusted user code the automatic trust revocation mechanism is activated and Untrusted Execution State (TES=0) is entered.

2.10.2 Untrusted Self-hosted Debug

An untrusted debug monitor running in untrusted m-mode and monitoring untrusted u-mode tasks can set soft breakpoints using the ebreak instruction and hardware breakpoints using triggers.To enable untrusted self-hosted debug mode `tmedeleg[3]` must be set so that untrusted debug exceptions are vectored through `mtvec`. If an ebreak occurs when running trusted code this will be vectored through `tmtvec`. However, to avoid an untrusted debug monitor making inferences about architectural state updates during trusted execution, all triggers are ignored in trusted execution state when `tmedeleg[3]` is set.

Self-hosted debug exception handling is summarized in the table below.

Table 6: Summary of Self-Hosted Debug Exception handling scenarios

tmedeleg[3]	TES on entry	Exception Type	Vector	TES at mret
0	0	Trigger or ebreak	tmtvec	$1 \rightarrow 0$
	1		tmtvec	1
1	0		mtvec	0
	1	ebreak	tmtvec	1
	1	Trigger	Ignored, no debug exception taken	

2.10.3 Trusted External Debug

Whether the external debug is trusted or untrusted is controlled by a mode pin at the boundary of the core. An external debugger operating in Trusted External Debug Mode (TEDM asserted) can take full control of the core out of reset. At this point the core is operating in a Trusted Execution State so the SOC must therefore provide mechanisms (e.g. fuses, or keys) to ensure that this mode pin can be locked down on production parts.

Trusted External Debug mode operates in Trusted Execution State (TES=1), memory access commands and accesses initiated by instructions executed from the PBUF are always performed in Trusted Execution State. However, when single stepping the current Trusted Execution State of the machine is used. It is therefore possible to single step through trusted and untrusted code, including stepping through trusted state transitions via the TESVEC.

This high level of privilege makes it essential that extensive measures are used to ensure TEDM cannot be enabled by a malicious attack.

The PBUF code can never cause a change to the Trusted Execution State (TES) of the core because all jumps are illegal, interrupts are masked and exceptions are reported to the debugger and do not redirect the PC to `mtvec` or `tmtvec`.

2.10.4 Untrusted External Debug

Untrusted external debug mode (TEDM de-asserted) allows debug operation at a reduced level of trustworthiness. However, access to this mode should still be carefully controlled as the high level control a debugger has significantly increases the risk that secret information might be inferred through side channel observations. In particular production parts should provide mechanisms (e.g. fuses or keys) to lock down any form of external debug. When operating in untrusted external debug mode it is impossible to get direct visibility of the architectural state of the core when it is running in Trusted Execution State. For example:

- If the debugger requests to halt the core straight out of reset, or at any other time when the core is in Trusted Execution State, halting will not occur until the core exits Trusted Execution State.
- Single stepping through untrusted code is supported but if execution steps in to Trusted Execution State the core will not halt until the core leaves Trusted Execution State.
- Triggers are disabled when TEDM=0 and the core is in Trusted Execution State.
- If the external debug unit has support for *System Bus Access* all memory accesses must be validated by the PMP and refused if they match a trusted PMP entry.

If an `ebreak` instruction is encountered while running in Trusted Execution State the request to transfer control to external debug is ignored and a trusted exception will be taken by vectoring off `tmtvec` instead.

Table 7: Summary of External Debug handling scenarios

TEDM	Current TES	Exception Type	Behaviour
0	0	ebreak	Enter untrusted debug mode
		Trigger	
		Single step completion	
	1	ebreak	Trusted exception vectored off tmtvec
		Trigger	Ignored, no entry to debug mode made
		Single step completion	
1	X	Any	Enter trusted debug mode

Note: *Implementation:* dynamic changes in the TEDM pin should be supported to allow locking/unlocking mechanisms to be implemented at the SOC level which can be controlled via the debugger. In particular, if TEDM is asserted/de-asserted while the CPU is halted subsequent debug operations should be performed with the new permissions, and if trusted execution is resumed with TEDM de-asserted it should not halt until the core leaves trusted execution.

CHAPTER

THREE

SECURITY ASSESSMENT

Because the trust status of every instruction is checked against the PMP, and immediate revocation of trust occurs if this trust is violated (TES $1 \rightarrow 0$), this architecture provides a high level of robustness against software and physical attacks attempting to escalate the privilege level. Sustained privileged execution of untrusted code would require co-ordinated attacks on the PMP and the TESVEC (or the IFU TES status) and security checks in the SOC (for example, via a Trusted Address Space Controller).

Note: *Implementation:* A physical attack might flip the trusted status of individual instructions as they pass down the pipe but this would be transient and flushed to an untrusted status during the PMP T-bit checks on subsequent instructions. An alternate microarchitecture is to have a single TES FSM (with multiple redundant states) and create guard periods which flush the pipe at all TES transitions so that trusted and untrusted instructions are never present in the pipe at the same time. This increases the cost of transitioning between states but may give better resilience. With either implementation, fundamentally, this architecture is believed to be more secure than competitor trust ISA architectures. A design may also want to implement additional protection/redunandancy within the PMP to reduce the risk of physical attack to that.

Note that trusted code (T bit set in the PMP entry) can never be read or modified by untrusted code regardless of the RWX access permissions for the PMP entry.

The TESVEC maintains Control Flow Integrity (CFI) of transitions into Trusted Execution by limiting these to a small set of valid entry points in to trusted code. The only other ways to transition to trusted execution are reset or via `tmtvec` when an exception/interrupt occurs.

The Trusted Execution State mechanism is compatible with memory fix/flash patch features provided that the following guidelines are followed:

- The flash patch/memory fix control registers must either be mapped to a trusted region or provide a lock mechanism so that they cannot be reprogrammed by untrusted code.
- Any patches to trusted code must redirect execution to a region of memory which is trusted, any patches to untrusted code must redirect execution to a region of memory which is untrusted. If these rules are not followed the standard TES access checks will ensure a PMP fault is generated
- Trusted External Debug Mode (TEDM) must be fused off or key protected in production parts. If TEDM is enabled the debugger has access to trusted execution state out of reset and can bypass secure boot and reprogram the flash patch. However all security is compromised in this state, not just the flash patch mechanism, hence the importance of protecting access to this feature.

When in Untrusted External Debug Mode the external debugger is prevented from halting the core until it leaves trusted execution for the first time, there is therefore no opportunity to bypass the secure boot process or reprogram the flash patch controller provided the guidelines above are followed.

CHAPTER

FOUR

USE CASES/MODELS

4.1 Lightweight secure function calls

This extension provides a very lightweight mechanism to implement Secure Function Calls. Trusted functions are placed in trusted memory and the TESVEC table is initialized with a table of entry points for all the Secure Functions that have been implemented. Secure functions can be written entirely in C and use the standard calling convention defined in the ABI with no special pre-amble or post-amble; provided that all parameters can be passed in the eight registers assigned to Function Arguments. Trusted code can call these routines directly without the overhead of indirecting through the TESVEC table.

When calling a secure function from untrusted code the calling convention is the same as the ABI for conventional functions. The only difference is that at link time the target of the call resolves to an address in the TESVEC table rather than the address of the secure function. The code below gives an example of how this could be implemented:

In this setup both untrusted and untrusted code appear to call the secure function through the same label, however, the header file prefixes all entry points with **to** when compiling untrusted code so the function is actually called via the TESVEC. If the secure function was called from trusted code then trusted execution continues when the secure function returns, however, if the call was from untrusted code the return address will be in untrusted memory and Untrusted Execution State is entered (TES=0).

On return from a secure function all ABI defined Temporary and Function Argument registers, except the Function Return registers, are reset to zero. The ABI defined Saved Registers are not reset because the secure function should return them to the value they had on entry prior to returning to the untrusted code.

For enhanced security a secure function can return to its caller using a `tret`. The use of `tret` is a software decision, it is always permitted to use `ret` to return to untrusted code (unless `tmescr.ete` has been set to enforce use of a `tret` in this situation). Which method is chosen is a trade-off between enhanced security and performance/software complexity. However, if a `tret` is used in a TESVEC entry point function it is no longer possible for that function to be called directly from trusted code, either all calls must be made through the TESVEC or a trampoline function must be used in the TESVEC which then calls the shared code.

Note that the shadowing mechanism ensures that untrusted code cannot modify/corrupt the Global Pointer (`gp`) and Thread Pointer (`tp`) values used by trusted code and therefore prevents secure functions making unintended data accesses/jumps based on the content of these registers.

If there are large numbers of arguments to the function it is possible that they could exceed the 8 registers available for argument passing. The caller will then use the stack to pass these additional parameters, however, due to stack shadowing, if the caller is untrusted these parameters are not directly available on the trusted stack. A number of solutions to this are possible:

- Modify the API so that some of the arguments are passed in a structure so that none get spilled to the stack.
- Use `asm` inserts in the secure function to read variables from the untrusted stack via the `tusp` CSR.
- Insert a trampoline function that copies the arguments passed on the untrusted stack to the trusted stack.

Some additional care needs to be taken with passing parameters on the stack if both trusted and untrusted calls through the TESVEC are made.

```c
#include "stdlib.h"
#include "securelib.h"

int main()
{
   sc_one();
   sc_two();

   return sc_three(3);
}
```

Untrusted code

```c
#ifndef SECURE_CODE
    #define sc_one   tosc_one
    #define sc_two   tosc_two
    #define sc_three tosc_three
#endif

void sc_one(void);
void sc_two(void);
int  sc_three(int);
```

Secure function header (securelib.h)

```c
#define SECURE_CODE
#include "securelib.h"

void sc_one(void)
{
}
void sc_two(void)
{
}
int  sc_three(int a)
{
    return a;
}

__attribute__((tes_entry)) void sc_wrap_two(void)
{
    sc_two();
}

#define TESVEC_RECORD(tesentry,fn) \
    ".align 3\n" \
    ".global to" #fn "\n" \
    "to" #fn ":\n"\
    "    .word " #tesentry "\n"

asm (
".align 3\n"
".global tesvec_table\n"
"tesvec:\n"
    TESVEC_RECORD(sc_one,      sc_one)
    TESVEC_RECORD(sc_wrap_two, sc_two)
    TESVEC_RECORD(sc_three,    sc_three)
"testop:\n"
);
```

Secure code

Fig. 1: **Example of secure function calling (with and without using trampolines)**

4.2 Calling untrusted/insecure code

Calling untrusted code is not as efficient as making trusted calls as additional steps are needed to protect general purpose register contents and to control the re-entry point to the trusted code. The recommended procedure is as follows:

1. The trusted code calls a trusted trampoline function with an argument list that matches that of the target untrusted function.
2. The trampoline function saves the ABI defined Saved Registers and the Return Address (`ra`) to the trusted stack and, ideally, clears the unused ABI defined Function Registers as these registers will not be cleared by the hardware mechanism.
3. The trampoline labels the point of return from the untrusted call and saves it in a TESVEC entry. It then sets `ra` to point to that TESVEC entry.
4. The trampoline jumps (with no link register) to the untrusted function so that the `ra` set in the previous step is preserved. The transition to untrusted execution will automatically clear the Temporary and Saved Registers so no code is required to clear these.
5. On return from the untrusted code the trampoline restores the Saved Registers and the Return Address and returns to the Trusted Caller.

To minimise the risk of security holes in the trampoline construction it can be generated using a define. An example of a suitable define is provided below. The define has three fixed arguments: the return type (`RT`), the function name (`FN`) and number of function arguments (`NARGS`) and a variadic which is the arguments to the function. It generates a function called `ut_fn_FN` with the same arguments as the wrapped function plus one additional argument which is a pointer to the TESVEC record where the return address for trusted execution is placed:

```
#define UT_FN_WRAPPER(RT,FN,NARGS,...) \
RT ut_fn_ ##FN(__VA_ARGS__, tesvec_record_t *tr) { \
    asm ( \
        "la ra, 1f\n\t" \
        "sw ra, 0(%[TR])\n\t" \
        "sw zero, 4 (%[TR])\n\t" \
        "mv ra, %[TR]\n\t" \
        ".irpc  arg,01234567\n\t" \
        ".if \\arg>=" #NARGS "\n\t"\
        "    mv a\\arg, zero\n\t" \
        ".endif\n\t" \
        ".endr\n\t" \
        "la t0," #FN "\n\t" \
        "jr t0\n\t" \
        "1:\n\t" \
    : : [TR] "r" (tr) : "ra", "s0", "s1", "s2", "s3", "s4", "s5", "s6", "s7", "s8",
↪ "s9", "s10", "s11" ); \
}
```

By listing all the Saved Registers as being clobbered, code to save and restore these will automatically be generated by the compiler. The `.irpc` loop zeros all Argument Registers that are not used to pass arguments to the untrusted function.

4.3 Trusted OS with untrusted (sandboxed) and trusted tasks

A trusted OS runs in Trusted Execution State (TES=1) and can start trusted or untrusted tasks running in any supported privilege mode. When running a trusted OS all exceptions are handled by trusted exception handlers (`tmedeleg[N]=0`). The OS start tasks running either in untrusted memory (for an untrusted task) or in trusted memory for a trusted one. When a task performs a system call (by executing an ecall instruction) the OS will be re-entered through `tmtvec` with TES=1. The system call is processed and an `mret` is performed to continue execution of the untrusted or trusted task.

4.4 Untrusted OS with secure functions and tasks

After the boot code completes and the processor enters Untrusted Execution State (TES=0) the setup and execution of an untrusted OS can proceed largely unaware of the presence of the Trusted Execution State. The constraints are:

- PMP entries reserved for secure use will be locked and read-only to an untrusted OS
- Only interrupts and exceptions which have been delegated using `tmideleg` and `tmedeleg` respectively can be handled. Typically it is expected that all exceptions (which occur while running untrusted code) are handled by the untrusted OS.
- Memory subsystems and peripherals with trust protection will be inaccessible. Attempts to access them will either result in a PMP fault exception or, if the region is not mapped to the PMP, will result in a bus fault.
- To run secure tasks the trusted environment will need to provide trusted services to the untrusted OS for initialising, starting and terminating secure tasks. In this use case untrusted interrupts are possible and could occur during execution of a either secure function or task. Refer to the Untrusted Interrupts section for a description of the mechanisms by which untrusted interrupts can be handled.

There is an additional complication that due to context switching during untrusted execution the secure function/task that was executing when an untrusted interrupt occurs may not be the same function/task that continues when the handler returns. This requires the trusted environment to maintain an array of multiple secure contexts, each with a dedicated entry in the TESVEC for resuming that context. This is illustrated in the figure below.

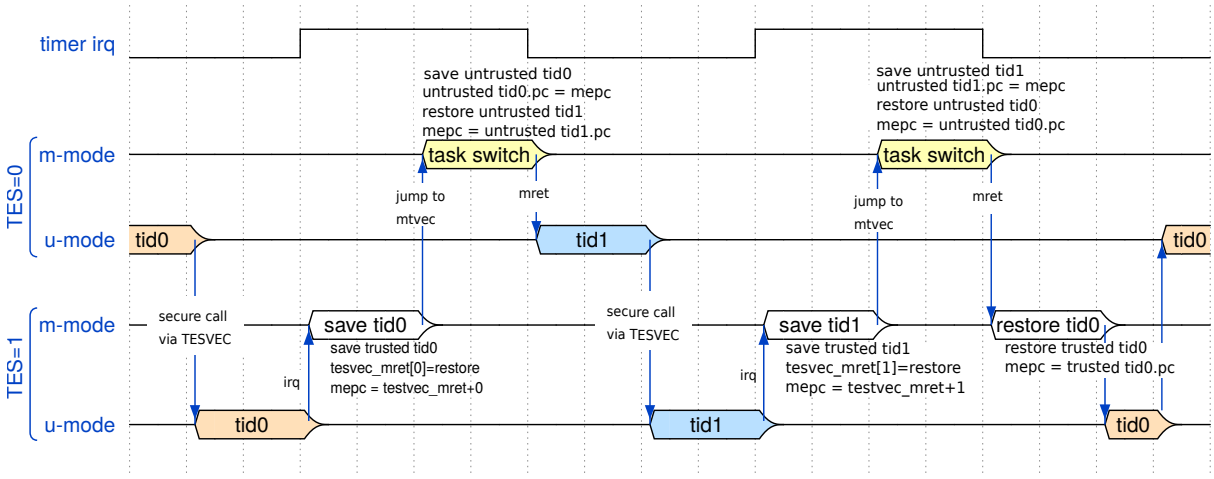

Fig. 2: **Switching between trusted contexts using an untrusted scheduler**

When programming a TESVEC record to restart a specific context, a shared routine can be used as the context can be identified using `tesepr.traddr`. The figure below illustrates how TESVEC could be used to manage restoring multiple secure contexts.

```c
#include <stdint.h>

#define TR_UTIE_MASK 1
#define TR_MRET_MASK 2

typedef uint64_t tesvec_record_t;

void sc_init_context(uint32_t id);
void sc_restore_context(uint32_t id);

// TESVEC record base for restoring secure contexts
tesvec_record_t *tesvec_mret;

tesvec_record_t* tesvec_alloc(uint32_t n, void (*entrypoint)(void))
{
    // alloc n additional TESVEC entries which all vector to entrypoint
    tesvec_record_t *testop, *base;
    asm volatile ( "csrr tmestop, %[testop]" : [testop] "=r"(testop) );
    base = testop;

    for(int i=0; i<n; i++)
        *(testop++) = TR_MRET_MASK | (tesvec_record_t) (uintptr_t) entrypoint;

    asm volatile ( "csrw tmestop, %[testop]" : : [testop] "r"(testop) );
    return base;
}

void tesvec_restore_ep(void)  __attribute__((aligned(4)));
void tesvec_restore_ep(void)
{
    // calculate context id from tesvec_mret and tescr.traddr
    tesvec_record_t *tescr;
    asm volatile ( "csrr tmescr,  %[tescr]"  : [tescr]  "=r"(tescr) );

    uint32_t id = (tescr-tesvec_mret)/sizeof(tesvec_record_t);

    sc_restore_context(id);
}

void sc_init_contexts(uint32_t n)
{
    // allocate tesvec entries to restoring secure contexts
    tesvec_mret = tesvec_alloc(n,tesvec_restore_ep);
    for(int i=0; i<n; i++)
        sc_init_context(i);
}
```

Fig. 3: **Using to TESVEC to manage restart of multiple secure contexts**

CHAPTER

FIVE

ISA SUMMARY AND ENCODINGS

5.1 Instruction Summary

One new instruction is added `tret`, return from trusted state, which takes no explicit operands. It performs an indirect jump to the address in `x1`/`ra` and, if `teseps.ctes` is clear, switches to Untrusted Execution State (TES=0). If `teseps.ctes` is set, continued trusted execution is permitted provided that the return address is within a trusted memory region, in summary, `TES = pmp_lookup(ra).pmptectl.T AND teseps.ctes`.

If executed when TES=0 it causes an illegal instruction trap.

Note: The opcode has not yet been assigned.

5.2 CSR Summary

The table below summarises the custom CSRs added to the ISA to support Trusted Execution State in a baseline configuration with M-mode and, optionally, U-mode. ROZ stands for read only zero, the difference between MROZ and UROZ is that UROZ will aleays retun zero regardless of privliege level but MROZ will trap if an access is attmpted from a lower privilege level. The `tesepr` amd `teseps` are u-mode accessible views of the `tmesepr` and `tmeseps`, that is they share the same state bits.

Table 1: Custom TES CSRs for baseline configuration with CLIC)

Addr	Permission		Name	Reset	Description
	TES=1	TES=0		Value	
0x7E0	MRW	MROZ	*tmescr*	0	Trusted Machine Execution State Control Register
0x7E1	MRW	MRO	*tmesvec*		Trusted Machine Execution State Vector Base
0x7E2	MRW	MRO	*tmestop*		Trusted Machine Execution State Vector Top
0x7E3	MRW	MRO	*tmedeleg*		Trusted Machine Exception Delegation
0x7E4	MRW	MROZ	*tmtvec*	0 or 3 if CLIC	Trusted Machine Trap Vector Base Address Register
0x7E5	MRW	MROZ	*tmtvt*	0	Trusted Machine Trap Vector Table (CLIC only)
0x7E6	MRW	MROZ	*tmintthresh*	0	Trusted Machine Interrupt Threshold (CLIC only)
0x7E7	MRW	MROZ	*tmstatus*	mie=0, other=X	Trusted Machine Status
0x7E8	MRW	MROZ	*tmepc*	X	Trusted Machine Exception Program Counter
0x7E9	MRW	MROZ	*tmcause*	0	Trusted Machine Trap Cause
0x7EB	MRW	MROZ	*tmtval*	X	Trusted Machine Trap Value
0x7EC	MRW	MROZ	*tmscratch*	X	Trusted Machine Scratch Register
0x7ED	MRW	MROZ	*tmesepr*	X	Trusted Machine Execution State Entry Point Record
0x7EE	MRW	MROZ	*tmeseps*	0	Trusted Machine Execution State Entry Point Status
0x7EF	MRW	MRW	*mtvlock*	0	Machine Trap Vector Lock
0x7F0	MRW	MROZ	*tmtvlock*		Trusted Machine Trap Vector Lock
0x7F8	MRW	Var	*pmptectl0-7*	impdef	PMP Trusted Execution Control
0x800	URW	UROZ	*tusp*	X	Trusted view of Untrusted Stack Pointer
0x801	URW	UROZ	*tugp*	X	Trusted view of Untrusted Global Pointer
0x802	URW	UROZ	*tutp*	X	Trusted view of Untrusted Thread Pointer
0xCC0	URO	UROZ	*tesepr*	X	Trusted Execution State Entry Point Record
0xCC1	URO	UROZ	*teseps*	0	Trusted Execution State Entry Point Status
TBA	MRW	MROZ	*tmidelegX*		Trusted Machine Interrupt Delegation (if no CLIC)

To allow untrusted execution to discover which PMP entries and memory regions have been reserved for trusted use some fields of the PMP fields assigned for trusted use are readable (MRO) when TES=0.

Table 2: Access permissions for PMP fields

pmpcfg.L	pmptectl.T	PMP Permissions		
		TES=1	TES=0	
			pmpcfg.A pmpaddr pmptectl.T	PMP other
0	0	MRW		
1	0	MRO		
0	1	MRW	MRO	MROZ
1	1	MRO		MROZ

Access permissions to all other CSRs are unaffected by the current Trusted Execution State.

CHAPTER

SIX

TRUSTED EXECUTION STATE SUB-EXTENSIONS

6.1 Extension ztesmultit: Multi-T support

The Trusted Execution State specification supports an optional extension that allows multiple trusted domains to co-exist with isolation of memory and peripherals between the domains.

6.1.1 Multi-T Rationale

The primary motivation for this extension is to provide trusted applications with low-latency, efficient access to highly-secure services, for example, a hardware cryptographic IP and its associated drivers. The secure service runs within a different trusted domain to that of the client trusted software. The aim is to provide this separation through functional calls without the need to transition via a higher privilege level involving environment calls and PMP re-programmings or activations.

The functions that provide the trusted service are assumed to be highly reliable and, in particular, are trusted to transfer control back to the correct client return address with all Saved Registers correctly restored and the stack uncorrupted.

6.1.2 Multi-T Specification

Eight separate trusted domains are supported with regions of the address space assigned to a domain using an additional field in the `pmptectl` of a PMP entry called the *trusted domain id* (`tdid`). There is also an additional bit to indicate if the content of the region can be shared (`sh`) with other domains.

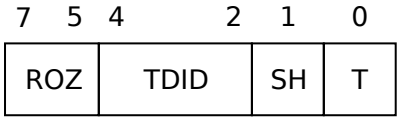

Fig. 1: **Multi-T PMPTECTL Entry**

Trusted domains operate in isolation from each other with no precedence between them.

A trusted domain can be configured to generate a PMP fault if it attempts execution of any code which has been shared by another trusted domain (by the setting of `pmptectl.sh` in that region). This opt-out is enabled for a specific domain (N) by setting bit N in a new field in `tmescr`, *No shared code* (`noshx`). The purpose is to minimise how much code it is legal to execute when executing in that domain, further reducing the risk of code reuse attacks. This option allows software architects to trade off code sharing and security on a domain by domain basis.

Domain access permissions aresummarized below.

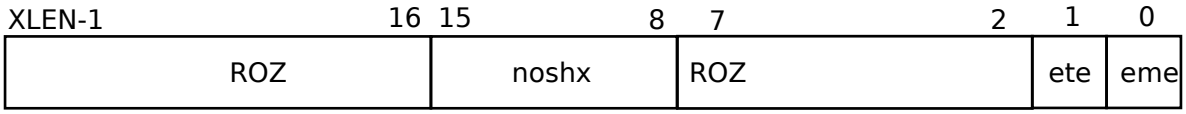

Fig. 2: **Extended Trusted Machine Execution State (tmescr)**

Table 1: Domain access permissions controlled by **pmptectl** record

pmptectl			Domain access permissions when hart . . .		
t	sh	tdid	TES=0	TES=1, TDID=N	TES=1, TDID!=N
0	0	N	RWX	RW	RW
0	1	N	RX	RWX	RW, X if !noshx[hart.TDID]
1	0	N	-	RWX	-
1	1	N	-	RWX	RW, X if !noshx[hart.TDID]

Sharing of both code and data between trusted domains and between trusted domains and untrusted domains is possible, for example, a single run time library image can be shared and doesn't need to be replicated in each domain. When a region is shared with the untrusted domain (`pmptectl.t=0,pmptectl.sh=1`) no write permissions are granted to the untrusted domain.

Reset hardware must initialize the PMP to hold a trusted entry which matches the reset address and has a `pmptectl.tdid=0`. At reset, execution always starts in Trusted Domain zero, (`TDID=0`). During PMP lookup any entry which has either the `pmptectl.t` or `pmptectl.sh` bit set is owned by a trusted domain and matching of that entry is prioritised above any entry owned by the untrusted domain. Note that if the `pmptectl.tdid` and `pmptectl.sh` of all PMP entries are zero then the behaviour matches that of a core without the Multi-T Extension and the feature is effectively disabled.

The content of the `pmptectl` can only be modified when operating in trusted m-mode. Domain access checks are in addition to the RWX permission checks for the region and are applied regardless of privilege level. If domain and RWX faults occur on the same PMP access, the domain access faults are reported as the higher priority fault. If a domain violation occurs a PMP fault is generated.

Domain access controls can be modified by any process running in trusted m-mode, provided that the entry has not been locked.

Entry to a different trusted domain is by calling a trusted entry point allocated in the Trusted Execution State Vector (TESVEC) table. The current trusted domain id, TDID, is updated to the `pmptectl.tdid` of the PMP entry for the entry point specified in the TESVEC Record, `TDID=pmp_lookup(TR.ep<<2).pmptectl.tdid`.

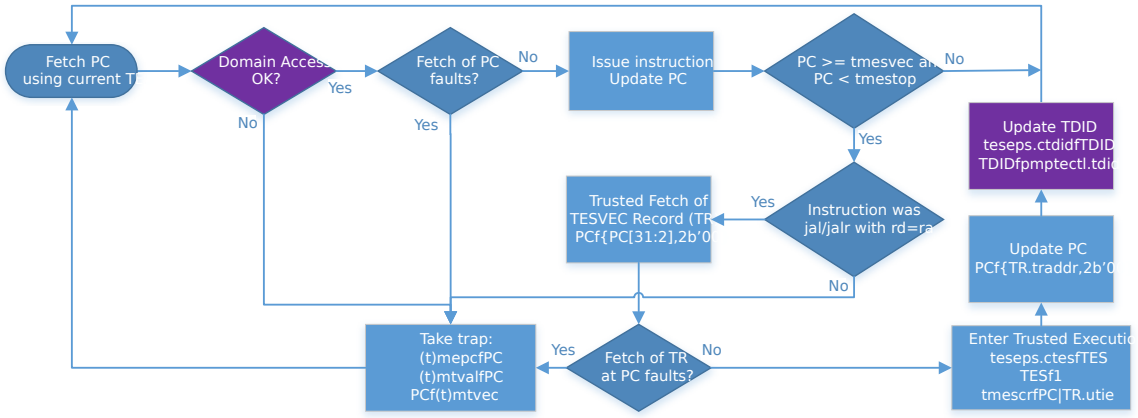

Fig. 3: **Updated TES entry flow for multi-t extension**

In addition to saving the *Caller TES* in `teseps.ctes`, as defined in the Base Specification, an additional field is

defined in `teseps` which is updated with the *Caller TDID*, `teseps.ctdid`. If the TESVEC was called from untrusted code then both `teseps.ctes` and `teseps.ctdid` are set to zero.

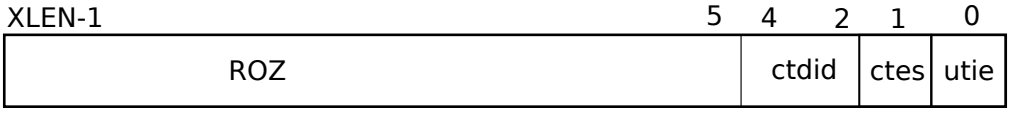

Fig. 4: **Multi-T Trusted Execution State Entry Point Status (teseps)**

A domain access PMP fault is also generated if code in a shared region (`pmptectl.sh=1`) attempts a call through the TESVEC. This eliminates the risk that execution could continue in the domain bound to the TESVEC entry when the trusted function returns. A domain fault is also generated if TESVEC entry is in a shared region. The purpose of the sharing mechanism to avoid duplication of common library code, it is not intended for use in transitioning between domains.

Execution continues in that trusted domain only while executing from regions allocated to that trusted domain or shared by other trusted domains. During normal execution the only way to leave this trusted domain is by executing a return, via a `ret` or `tret` instruction, to a region owned by the trusted domain specified in `teseps.ctdid`. This causes the current trusted domain id to be updated (`TDID=teseps.ctdid`). If a `tret` is used then the current trusted domain is set to `teseps.ctdid`. If this does not match the trusted domain of the return address a Domain Fault will be generated.

If the last TESVEC call was from untrusted code, `teseps.ctes` will be zero and execution of a `tret` will force execution to continue in an *Untrusted Execution State* (TES=0).

The rules for the automatic clearing of registers when transitioning between trusted domains via a `ret` or `tret` are the same as for transitions from TES $1 \rightarrow 0$.

Any other form of jumping or branching to an unshared region owned by any other domain results in a PMP fault.

Any exception or interrupt that is vectored through either `tmtvec` (or `tmtvt` if using the CLIC's vector mode) causes the trusted domain id of the first instruction of the interrupt handler to be saved to a new field of the `tmstatus`, *Previous TDID* (`ptdid`). This is in addition to the current TES being saved to `tmstatus.ptes`. If the core was in an Untrusted Execution State then the `ptdid` is set to zero. The TES is then set 1 and the trusted domain id is set to the domain associated with the trusted handler, `TDID=pmp_lookup(irq_handler).pmptectl.tdid`. When an `mret` occurs the TES is set to the value of `tmstatus.ptes` and the trusted domain is set to `tmstatus.ptdid`. When vectoring through the CLIC's `tmtvt` it is possible to associate a handler with a specific trusted domains just by mapping ot to a PMP region wih the appropriate `pmptectl.tdid` setting.

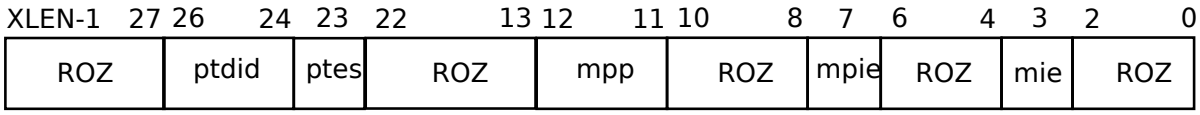

Fig. 5: **Extended Trusted Machine Status (tmstatus)**

The effect of domains on the trusted state of the machine is summarized in the table below.

Table 2: Summary of Domain Related Transitions

Current trust state			PMPTECTL of next PC			Next trust state		
TES	**TDID**	**Insn**	**T**	**SH**	**TDID**	**TES**	**TDID**	**Auto-clear**
X	-	call/ret/mret TESVEC	1	0	N	1	N	-
				1	X	Instruction access fault		
			0	X				
		other TESVEC	X					
0	-	any legal	0	X	X	0	-	-
			1			Instruction access fault		
1	N	tret	ctes	0	ctdid	ctes	ctdid	Yes
					!ctdid	Instruction access fault		
			!ctes		X			
		mret	ptes		ptdid	ptes	ptdid	No
					!ptdid	Instruction access fault		
			!ptes		X			
		j/ret	0		X	0	-	Yes
		other				Instruction access fault		
		ret	1		!N AND ctdid	1	ctdid	Yes
					!N AND !ctdid	Instruction access fault		
		not tret/mret			N	1	N	-
		not any ret			!N	Instruction access fault		
		any legal	X	1	N OR !noshx[N]	1	N	-
					!N AND noshx[N]	Instrcution access fault		

Note: *Implementation:* When signaling the trust status of a memory transaction implementations are expected to provide an additional sideband signal which gives the trusted domain id of the transaction. This allows additional hardware security checks to be applied when accesses are made to secure IPs. Further details of this signaling are product/bus protocol specific and outside the scope of this specification.

If a trusted interrupt or exception handler has to access a service in a different trusted domain it should save the current `teseps` and `tesepr` prior to calling the TESVEC and then restore them before restarting the thread. The same is true if the handler is performing a switch to a different thread.