PMP Enhancements for memory access and execution prevention on Machine mode

Nick Kossifidis, Joe Xie, Bill Huffman, Allen Baum, Greg Favor, Tariq Kurd, Fumio Arakawa, RISC-V TEE Task Group

This document is a copy of the primary version found here: https://docs.google.com/document/d/1Mh_aiHYxemL0umN3GTTw8vsbmzHZ_nxZXgjgOUzbvc8/edit?usp=sharing made on the 22nd January 2021.

Introduction

Being able to access the memory of a process running at a high privileged execution mode, such as the Supervisor or Machine mode, from a lower privileged mode such as the User mode, introduces an obvious attack vector since it allows for an attacker to perform privilege escalation and tamper with the code and/or data of that process. A less obvious attack vector exists when the reverse happens, in which case an attacker instead of tampering with code and/or data that belong to a high-privileged process, will tamper with the memory of an unprivileged / less-privileged process and trick the high-privileged process to use or execute it.

To prevent this attack vector, two mechanisms known as Supervisor Memory Access Prevention (SMAP) and Supervisor Memory Execution Prevention (SMEP) were introduced in recent systems. The first one prevents the OS from accessing the memory of an unprivileged process unless a specific code path is followed, and the second one prevents the OS from executing the memory of an unprivileged process at all times. RISC-V already includes support for SMAP, through the sstatus. SUM bit, and for SMEP by always denying execution of virtual memory pages marked with the U bit, with Supervisor mode (OS) privileges, as mandated on the Privilege spec.

Terms:

- **Ignored**: All accesses are permitted to the address range configured in the PMP entry
- **Enforced**: Only access types configured in the PMP entry are allowed; failures will cause an access exception
- **Denied**: No access is permitted to the address range configured in the PMP entry; failures will cause an access exception

Threat model

However, there are no such mechanisms present on Machine mode and with the current spec it’s not possible to mitigate such attacks using physical memory addressing and Physical Memory Protection. With the current spec if we want a PMP rule to be **enforced** only on non-Machine modes and denied on Machine mode, so that we can only allow access to a memory region by less-privileged modes, we don’t have the option. We can only have a locked rule that will be **enforced** on all modes, or a rule that will be **enforced** on non-Machine modes and be **ignored** by Machine mode. So on any physical memory region not protected with a Locked rule, Machine mode has unlimited access, including the ability to execute it.

Without being able to protect less-privileged modes from Machine mode, we can’t prevent the mentioned attack vector. This becomes even more important for RISC-V than on other architectures, since we also allow implementations where a hart will only have Machine and User modes available, where the whole OS will run on Machine mode instead of the non-existent Supervisor mode. In such implementations the attack surface is greatly increased, and the same kind of attacks performed on Supervisor mode and mitigated through SMAP/SMEP, can be performed on Machine mode without any available mitigations. On implementations with Supervisor mode present attacks are still possible against the Firmware and/or the Secure Monitor running on Machine mode.

Proposal

1) **Machine Security Configuration (mseccfg)** is a new Machine mode CSR, used for configuring various security mechanisms present on the hart, and only accessible to Machine mode. It is only available on harts with PMP support. It’s 64bits long and it’s address is 0x390 on RV64 and 0x390 (low 32bits), 0x391 (high 32bits) for RV32. All bits except bit0-bit2 are reserved for now and all mseccfg fields defined on this proposal are RW. The reset value of mseccfg is implementation-specific, otherwise if backwards compatibility is a requirement it should reset to 0 on hard reset.

2) On mseccfg we introduce a field on bit2 called **Rule Locking Bypass (mseccfg.RLB).** Note that it’s intended to be used as a debug mechanism, or as a temporary workaround during the boot process for simplifying sw, and optimizing the allocation of memory and PMP rules. Using this functionality under normal operation should be avoided since it weakens the protection of *M-mode-only* rules. Vendors who don’t need this functionality may hard-wire this field to 0.

   a) When *mseccfg.RLB* is 1 PMP rules with *pmpcfg.L* bit 1 can be removed and/or edited.

b) When *mseccfg.RLB* is 0 and pmpcfg.L is 1 in any entry (including disabled entries), then *mseccfg.RLB* is locked and any further modifications to *mseccfg.RLB* are ignored (WARL).

3) On mseccfg we introduce a field on bit1 called **Machine Mode Whitelist Policy (mseccfg.MMWP).** This is a sticky bit, meaning that once set it cannot be unset until a hard reset. When set it changes the default PMP policy for M-mode when accessing memory regions that don't have a matching PMP rule, to **denied** instead of **ignored**.

4) On *mseccfg* we introduce a field on bit0 called **Machine Mode Lockdown (mseccfg.MML)**. This is a sticky bit, meaning that once set it cannot be unset until a hard reset. When *mseccfg.MML* is set the system's behavior changes in the following way:

   a) The meaning of **pmpcfg.L** changes: Instead of marking a rule as locked and **enforced** on all modes, it now marks a rule as **M-mode-only** when set and **S/U-mode-only** when unset. The formerly reserved encoding of RW=01, and the encoding LRWX=1111, now encode a **Shared-Region**.

   An *M-mode-only* rule is **enforced** on Machine mode and **denied** on Supervisor or User modes. It also remains locked so that any further modifications to the configuration or address registers are ignored until a hard reset, unless *mseccfg.RLB* is set.

   An *S/U-mode-only* rule is **enforced** on Supervisor and User modes and **denied** on Machine mode.

   A *Shared-Region* rule is **enforced** on all modes, with restrictions depending on the *pmpcfg.L* and *pmpcfg.X* bits:

   - If *pmpcfg.L* is not set the region can be used for sharing data between M-mode and S/U-mode so it's not executable. M-mode has RW access to that region and S/U-mode has read access if *pmpcfg.X* is not set, or RW access if *pmpcfg.X* is set.

   - If *pmpcfg.L* is set the region can be used for sharing code between M-mode and S/U-mode so it's not writeable. Both M-mode and S/U-mode have execute access on the region and M-mode may also have read access if pmpcfg.X is set. The region remains locked so that any further modifications to the configuration or address registers are ignored until a hard reset, unless *mseccfg.RLB* is set.

   - The encoding pmpcfg.LRWX=1111 can be used for sharing data between M-mode and S/U mode, where both modes only have read-only access to the region. The region remains locked so that any further modifications to the configuration or address registers are ignored until a hard reset, unless *mseccfg.RLB* is set.

b) Adding a new **M-mode-only** or a **Shared-Region** rule with executable privileges is not possible and such *pmpcfg* writes are ignored, leaving *pmpcfg* unchanged. This restriction can be temporarily lifted e.g. during the boot process, by setting *mseccfg.RLB*.

c) Executing code with Machine mode privileges is only possible from memory regions with a matching **M-mode-only** rule or a **Shared-Region** rule with executable privileges. Executing code from a region without a matching rule or with a matching *S/U-mode-only* rule is **denied**.

d) If *mseccfg.MML* is not set, the combination of *pmpcfg.R=0*, *pmpcfg.W=1* remains reserved.

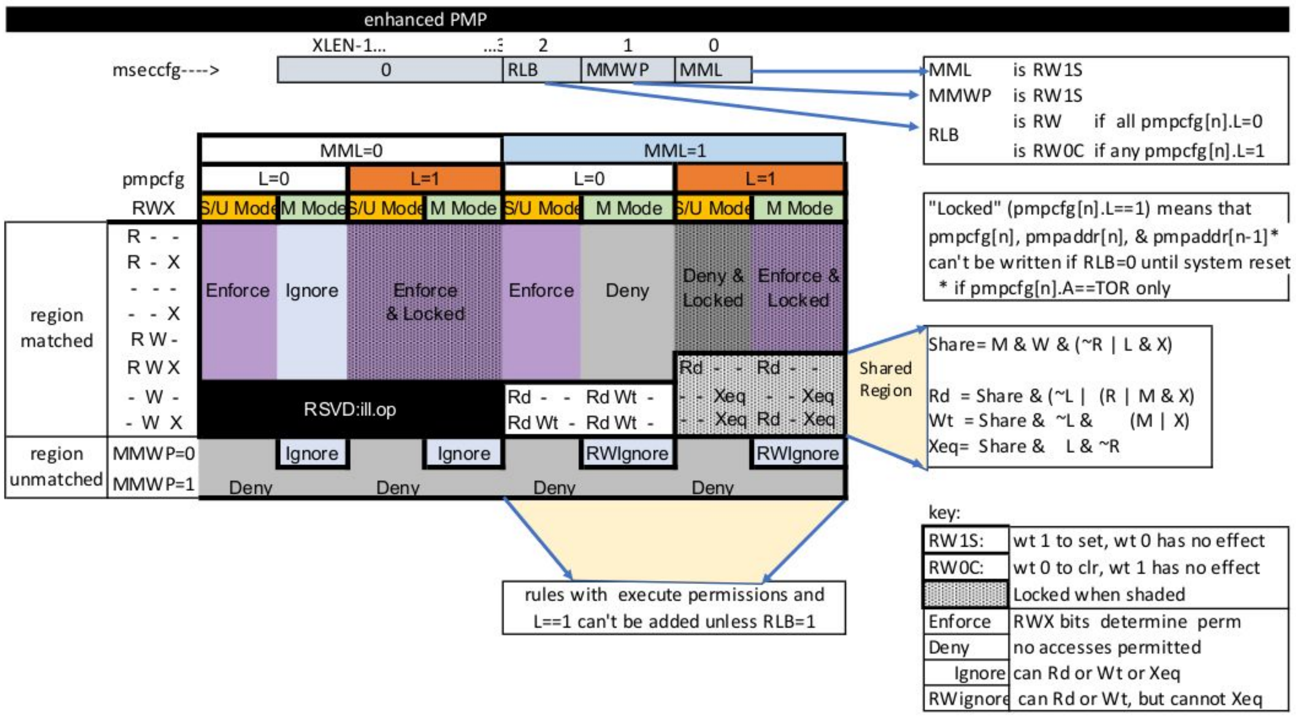
Truth table when *mseccfg.MML* is set

Bits on *pmpcfg* register				Result	
L	R	W	X	M Mode	S/U Mode
0	0	0	0	Inaccessible region (Access Exception)	
0	0	0	1	Access Exception	Execute-only region
0	0	1	0	Shared data region: Read/write on M mode, read-only on S/U mode	
0	0	1	1	Shared data region: Read/write for both M and S/U mode	
0	1	0	0	Access Exception	Read-only region
0	1	0	1	Access Exception	Read/Execute region
0	1	1	0	Access Exception	Read/Write region
0	1	1	1	Access Exception	Read/Write/Execute region
1	0	0	0	Locked inaccessible region* (Access Exception)	
1	0	0	1	Locked Execute-only region*	Access Exception
1	0	1	0	Locked Shared code region: Execute only on both M and S/U mode.*	

1	0	1	1	Locked Shared code region: Execute only on S/U mode, read/execute on M mode.*	
1	1	0	0	Locked Read-only region*	Access Exception
1	1	0	1	Locked Read/Execute region*	Access Exception
1	1	1	0	Locked Read/Write region*	Access Exception
1	1	1	1	Locked Shared data region: Read only on both M and S/U mode.*	

*: Locked entries cannot be removed or modified until a hard reset, unless *mseccfg.RLB* is set.

Visual representation of the proposal


enhanced PMP
XLEN-1...
...3
2
1
0
mseccfg---->
0
RLB
MMWP
MML
MML is RW1S
MMWP is RW1S
RLB is RW if all pmpcfg[n].L=0
is RW0C if any pmpcfg[n].L=1
MML=0
MML=1
pmpcfg
L=0
L=1
RWX
S/U Mode
M Mode
R - -
R - X
- - -
- - X
R W -
R W X
- W -
- W X
region matched
region unmatched
Enforce
Ignore
Enforce & Locked
Deny
Deny & Locked
Enforce & Locked
RSVD:ill.op
Rd - -
Rd Wt -
Rd Wt -
Rd Wt -
Rd - -
- - Xeq
- - Xeq
Rd - Xeq
Shared Region
MMWP=0
MMWP=1
RWIgnore
"Locked" (pmpcfg[n].L==1) means that pmpcfg[n], pmpaddr[n], & pmpaddr[n-1]* can't be written if RLB=0 until system reset
* if pmpcfg[n].A==TOR only
Share= M & W & (~R | L & X)
Rd = Share & (~L | (R | M & X)
Wt = Share & ~L & (M | X)
Xeq= Share & L & ~R
rules with execute permissions and L==1 can't be added unless RLB=1
key:
RW1S: wt 1 to set, wt 0 has no effect
RW0C: wt 0 to clr, wt 1 has no effect
Locked when shaded
Enforce RWX bits determine perm
Deny no accesses permitted
Ignore can Rd or Wt or Xeq
RWignore can Rd or Wt, but cannot Xeq

Rationale (WiP)

1. Since a CSR for security and / or global PMP behavior settings is not available with the current spec, we needed to define a new one. This new CSR will allow us to add further security configuration options in the future and also allow developers to verify the existence of the new mechanisms defined on this proposal.

2. There are use cases where developers want to enforce PMP rules on M-mode during the boot process, that are also able to modify, merge, and / or remove later on. Since a rule that is enforced on M-mode needs to also be locked (or else M-mode software can remove it at any time), the only way for developers to approach this is to keep adding PMP rules to the chain and rely on rule priority. This is a waste of PMP rules and since it's only needed during boot, RLB is a simple workaround that can be used temporarily and then disabled and locked down.

RLB is also there to allow the registration of the newly-defined Shared code regions during the boot process. When MML is set though, according to 4b it's not possible to add a *Shared-Region* rule with executable privileges. So RLB can be set temporarily during the boot process to register such regions. Note that it's still possible to register executable *Shared-Region* rules using initial register settings (that include MML being set and the rule being set on PMP registers) on hardware reset, without using RLB.

Be aware that RLB introduces a security vulnerability if left set after the boot process is over and in general it should be used with caution, even when used temporarily. Having editable PMP rules on M-mode is a false sense of security since it only takes a few instructions to lift any PMP restrictions this way. It doesn't make sense to have a security control in place and leave it unprotected. RLB is only meant as a way to optimize the allocation of PMP rules, and allow the bootrom/firmware to register executable *Shared-Region* rules. If developers / vendors have no use for such functionality, they should never set RLB and if possible hard-wire it to 0. In any case **RLB should be disabled and locked as soon as possible**. Note that if RLB is not used and left unset, it'll get locked as soon as a PMP rule with the L bit set on pmpcfg is added.

3. With the current spec M-mode can access any memory region unless restricted by a PMP rule with the L bit set on pmpcfg. There are cases where this approach is overly permissive, and although it's possible to restrict M-mode by adding PMP rules during the boot phase, this can also be seen as a waste of PMP rules. Having the option to block anything by default, and use PMP as a whitelist for M-mode is considered a safer approach. This functionality can be used during the boot process or upon hardware reset, using initial register settings.

4a. The current dual meaning of the *pmpcfg.L* bit that marks a rule as Locked and **enforced** on all modes is neither flexible nor clean. With the introduction of *Machine Mode Lock-down* the *pmpcfg.L* bit distinguishes between rules that are **enforced** only on M-mode (*M-mode-only*) or only on S/U-modes (*S/U-mode-only*). The rule locking becomes part of the definition of an *M-mode-only* rule, since when a rule is added on M mode, if not locked, can be modified or removed in a few instructions. On the other hand, S/U modes can't modify PMP rules anyway so locking them doesn't make sense.

This separation between *M-mode-only* and *S/U-mode-only* rules also allows us to distinguish which regions are to be used by processes in Machine mode (L=1) and which by Supervisor or User mode processes (L=0), in the same way the U bit on the Virtual Memory's PTEs marks which Virtual Memory pages are to be used by User mode applications (U=1) and which by the Supervisor / OS (U=0). With this distinction in place we are able to implement memory access and execution prevention on M-mode for any physical memory region that is not *M-mode-only*.

An attacker that manages to tamper with a memory region used by S/U mode, even after successfully tricking a process running on M-mode to use or execute that region, will fail to perform a successful attack since that region will be *S/U-mode-only* hence any access when on M-mode will trigger an access exception.

In order to support zero-copy transfers between M-mode and S/U-mode we need to either allow shared memory regions, or introduce a mechanism similar to the *sstatus.SUM* bit to temporary allow the high-privileged mode (in this case M-mode) to be able to perform loads and stores on the region of a less-privileged process (in this case S/U-mode). In our case after discussion within the group it seemed a better idea to follow the first approach and have this functionality encoded on a per-rule basis to avoid the risk of leaving a temporary, global bypass active when exiting M-mode, hence rendering memory access prevention useless.

Although it's possible to use m*status.MPRV* on M-mode to read/write data on an *S/U-mode-only* region using general purpose registers for copying, this will happen with S/U-mode permissions, honoring any MMU restrictions put in place by S-mode. Of course it's still possible for M-mode to tamper with the page tables and / or add *S/U-mode-only* rules and bypass the protections put in place by S-mode but if an attacker has managed to compromise M-mode to such extent, no security guarantees are possible in any way. Also note that the threat model we present here assumes buggy software on M-mode, not compromised software. We considered disabling *mstatus.MPRV* but it seemed too much and out of scope.

Shared-region rules can be used both for zero-copy data transfers and for sharing code segments. The latter may be used for example to allow S/U-mode to execute code by the vendor, that makes use of some vendor-specific ISA extension, without having to go through the firmware with an ecall. This is similar to the vDSO approach followed on Linux, that allows userspace code to execute kernel code without having to perform a system call.

To make sure that shared data regions can't be executed and shared code regions can't be modified, the encoding changes the meaning of the X bit. In case of shared data regions, with the exception of the LRWX=1111 encoding, the X bit marks the capability of S/U-mode to write to that region, so it's not possible to encode an executable shared data region. In case of shared code regions, the X bit marks the capability of S/U-mode to read from that region, and since RW=01 is used for encoding the shared region, it's not possible to encode a shared writable code region. Note that the capabilities marked by the X bit are always available on M-mode but M-mode still can't execute a shared data region, nor write a shared code region. A *Shared-region* rule for a shared code region must be added after M-mode has written the code segment there.

Using the LRWX=1111 encoding for a locked shared read-only data region was decided later on, its initial meaning was an M-mode-only read/write/execute region. The reason for that change was that the already defined shared data regions were not locked, so r/w access to M-mode couldn't be restricted. In the same way we have execute-only shared code regions for both modes, it was decided to also be able to allow a least-privileged shared data region for both modes. This approach allows for example to share the .text section of an ELF with a shared code region and the .rodata section with a locked shared data region, without allowing M-mode to modify .rodata. We also decided that having a locked read/write/execute region on M-mode doesn't make much sense and could be dangerous, since M-mode won't be able to add further restrictions there (as in the case of S/U-mode where S-mode can further limit access to an LWRX=0111 region through the MMU), leaving the possibility of modifying an executable region on M-mode open.

Note that for encoding Shared-region rules initially we used one of the two reserved bits on pmpcfg (bit 5) but in order to avoid allocating an extra bit, since those bits are a very limited resource, it was decided to use the reserved R=0,W=1 combination.

4b. The idea with this restriction is that after the Firmware or the OS running on M-mode is initialized and MML is set, no new code regions are expected to be added since nothing else is expected to run on M-mode (everything else will run on S/U mode). Since we want to limit the attack surface of the system as much as possible, it makes sense to disallow any new code regions which may include malicious code, to be added/executed on M-mode.

4c. In case *mseccfg.MMWP* is not set, M-mode can still access and execute any region not covered by a PMP rule. Since we try to prevent M-mode from executing malicious code and since an attacker may manage to place code on some region not covered by PMP (e.g. a directly-addressable flash memory), we need to ensure that M-mode can only execute the code segments initialized during firmware / OS initialization.

4d. We are only using the encoding RW=01 together with MML, if MML is not set the encoding remains usable for future use.