

RISC-V Security Verification using Perspec/Portable Stimulus

Junxia Wang, Siyan.Li, Leven.Li

Media Tek Building 1-B, No. 6 Park, Jiuxianqiao Road, Chaoyang District, Beijing China

Junxia.Wang@mediatek.com, Siyan.Li@mediatek.com, leven.li@mediatek.com

A T S Prasad, Kiran Kumar Palla, Yung Cheng Chen

Cadence Design Systems, 2655 Seely Ave, San Jose, CA 95134, USA

atprasad@cadence.com, kpalla@cadence.com, vicchen@cadence.com

Abstract - Modern SoC security features restrict accesses of shared memory and system resources only to the privileged agents in the system. RISC-V processor architecture enforces security with a Physical Memory Protection (PMP) specification. The complex PMP hardware poses a verification challenge. This paper describes how the Portable Test & Stimulus (PSS)^[2] and EDA tools, such as Perspec^[4], are used to efficiently verify security aspects of RISC-V based SoCs

I. INTRODUCTION

Modern SoC, designed for automotive, mobile or data center applications, typically has multiple processors, multi-level cache hierarchy, and multiple subsystems that share memory and system resources. Open access to shared memory and resources by all agents in the system leaves security holes in the SoC design. In RISC-V architecture based SoCs, this problem is addressed by Physical Memory Protection (PMP)^[3] hardware unit by limiting the physical addresses accessible by software running on a processor core. A PMP unit tackles the security aspect related to physical memory access privileges – read, write, execute permissions – in different execution modes of a processor core.

A RISC-V PMP unit is a programmable hardware block that allows multiple memory regions to be specified, each with its own privilege access policy per processor core. In a multi-cluster, multi-processor SoC context, verifying PMP is a complex challenge due to large space of concerning crosses of PMP regions, cores, access policies. The complexity is amplified when Physical Memory Attributes (PMA)^[3] - like shareability, cacheability, exclusiveness - are thrown into the verification mix. Another challenge is creating tests and test infrastructure to verify negative security scenarios. For example, forcing a privilege access violation to check the expected system response & behavior.

The SoC design used for this work is a typical system consisting of multi-core RISC-V 64-bit (RV64) CPU with PMP unit, few system memories and a cache sub-system.

This paper describes how the Portable Test & Stimulus (PSS)^[2] and EDA tools, such as Perspec, are used to efficiently verify security aspects of RISC-V based SoCs. PSS modeling of various SoC security test scenarios for verifying PMP features is described. PSS modeling for both positive and negative security tests is demonstrated.

II. OVERVIEW OF RISC-V PMP SPECIFICATION

In SoCs, it is desirable to limit the physical addresses accessible by software running on a hart (a.k.a hardware thread). Such limitation helps support secure processing and contain faults in the system. There are three kinds of privilege mode on RISC-V environment (Table 1). Code run in M-mode is often trusted. M-mode can be used to manage secure execution on RISC-V. U and S mode are intended for application and operating system usage respectively. An optional Physical Memory Protection (PMP) hardware unit provides per-hart machine-mode M-mode (Table 1) control registers to allow physical memory access privileges (read, write, execute) to be specified for each physical memory region. The programmed PMP values are checked for all accesses whose effective

privilege mode is S or U (Table 1). Accesses without proper permissions will trigger access fault exceptions.

Level (Mode)	Name	Abbreviation
0	User	U
1	Supervisor	S
2	Reserved	
3	Machine	M

Table 1: RISC-V-64 Privilege Level (Mode)

A PMP unit implementation may support 0, 16 or 64 entries. These entries divide the physical address space into different regions with varying access permissions (Table 4). It is achieved by configuring pmpcfg and pmpaddr registers for each region. Below description of PMP registers illustrates the register layout for 64 entries. However, the PMP hardware unit in our SoC design supports only 16 entries (physical memory regions).

PMP Registers

RISC-V specification describes two sets of Control and Status Registers (CSR) to implement PMP feature.

- **PMP configuration registers:** 16 CSRs pmpcfg0-pmpcfg14 hold the PMP configurations, pmp0cfg0-pmp63cfg, for 64 physical memory regions. Odd numbered pmpcfg registers are invalid. See Figure 1

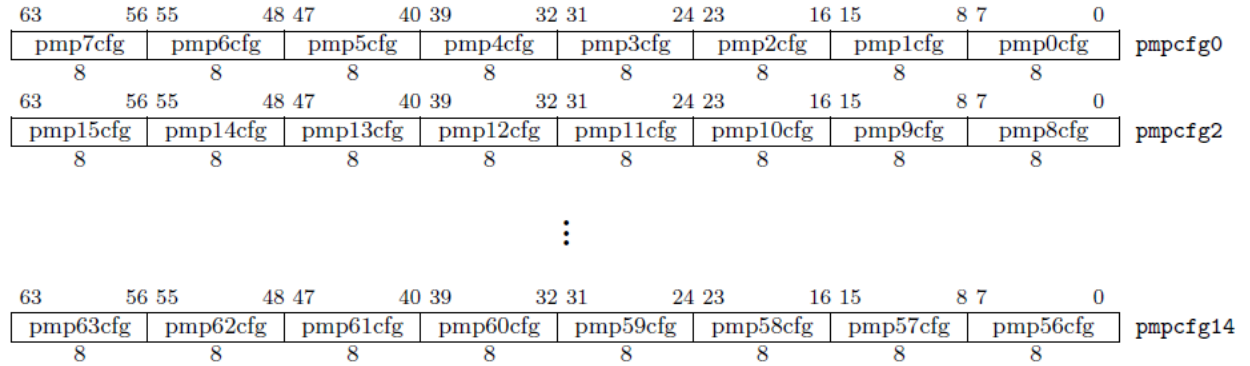


Figure 1: RV64 PMP Configuration CSR Layout

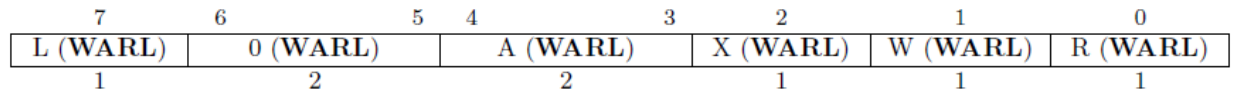


Figure 2: PMP Configuration CSR format

Figure 2 shows the layout of 8-bit PMP configuration register. The R, W, and X bits, when set, indicate that the PMP entry permits read, write and instruction execution, respectively. When one of these bits is clear, the corresponding access type is denied. The 'L' bit indicates that the PMP entry is locked. Any writes to the configuration and associated address registers are ignored. The 'A' field encodes the address-matching mode of the associated PMP address register. When A bit is 0, PMP functionality is disabled.

A	Name	Description
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region, ≥ 8 bytes

Figure 3: Encoding of A field in PMP Configuration registers

- **PMP address registers:** The PMP address registers are CSRs named pmpaddr0-pmpaddr63. Each PMP address register encodes bits 55:2 of a 56-bit physical address for RISC-V-64 bits cores.

Thus, with a combination of PMP configurations and address matching modes, the PMP unit enhances security of a RISC-V SoC by supporting granular control of permissions across multiple physical memory regions. Moreover, the permissions can be dynamically re-programmed by each hart to enforce its own security policy on the system memory and resources.

III. USING PERSPEC LIBRARY AND PSS SCENARIOS FOR SECURITY VERIFICATION

This section describes how Portable Stimulus & Test Standard (PSS) can be used to model PMP features and create test scenarios that verify PMP functionality. We describe the test development process in multiple incremental steps – starting from building blocks to full scenario specification to random scenario variations.

Test development using PSS follows below process:

1. Model compute subsystem (processor-memory)
2. Model PMP features
3. Develop security test scenarios using PMP features
4. Create test variations to cover concerning cross of PMP features

Step-1: Model compute subsystem (processor-memory)

The approach to modeling the compute subsystem is elaborately described in a previous work.^[1] We will repeat only the relevant highlights from the referenced paper.

Using the Perspec Coherency Library, the “modeling” process of the compute subsystem required no coding. We just needed to fill out the information related to the processor cores, the clusters, the memory types/sizes, the cache structure, etc., in the Perspec configuration tables. These tables were captured in an Excel/csv configuration file.

This “modeling” process of our SoC compute subsystem was done in a couple of hours. Most of this time was spent tracking down the design information required to fill out the Perspec configuration tables.

Table 2 and Table 3 show an example of the processor and memory configuration tables.

- Table 2 - “processor_info” table: this table describes the processor subsystem of the design; the columns in this table represent the attributes of the design; some key attributes are:

- #tag: name of the processor cores; there are 4 of them in cluster R0: hart0 to hart3
- #kind: the kind/type of processor
- #cluster: name of the processor clusters; one cluster R0

- Table 3 - “memory_info” table: this table specifies the different memory blocks and their address ranges.

In this example, we have:

- #mem_block: Three different memory blocks: mem0, mem1, mem2
- #enabled: When TRUE, the memory block is enabled in the design

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	@table:processor_info													
2	@package: sml_pkg	@size_const: NUM_OF_CORES	@struct: sml_processor_info_s											
3	#tag	#kind	#cluster	#cluster_id	#core_id	#enabled	#scheme	#hdi_path	#coherency_level	#exclusive_able	#power_d_	#powers_	#barrier_	#clock_p_
4	hart0	RISCV	R0	0	0	TRUE	RISCV_C	NA	FULL	TRUE	FALSE	TRUE	TRUE	80ns
5	hart1	RISCV	R0	0	1	TRUE	RISCV_C	NA	FULL	TRUE	FALSE	FALSE	TRUE	80ns
6	hart2	RISCV	R0	0	2	TRUE	RISCV_C	NA	FULL	TRUE	FALSE	FALSE	TRUE	80ns
7	hart3	RISCV	R0	0	3	TRUE	RISCV_C	NA	FULL	TRUE	FALSE	FALSE	TRUE	80ns

Table 2: processor_info table

	A	B	C	D	E	F	G	H	I
1	@table: memory_info								
2	@package: sml_pkg	@size_const: NUM_OF_MEM_B...	@struct: sml_memory_info_s						
3	#mem_block	#base_addr	#end_addr	#alignment	#enabled	#backdoor_enabled	#exclusive_able	#atomics_supported	#is_l2_lim
4	mem0	0x90000000	0x97FFFFFF	1	TRUE	FALSE	TRUE	TRUE	TRUE
5	mem1	0x98000000	0x99FFFFFF	1	TRUE	FALSE	TRUE	TRUE	TRUE
6	mem2	0xA0000000	0xA1FFFFFF	1	TRUE	FALSE	TRUE	TRUE	TRUE

Table 3: memory_info table

Once above configuration tables were filled out, we were able to bring-up Perspec, create memory access tests using Perspec GUI (Graphical User Interface) and/or writing the PSS code directly. We were able to pipe clean the PSS based verification flow quickly using the generated tests before moving on to the next step.

Step-2: Model PMP features

The next step is to starting modeling security features in PSS that serve as building blocks for more complicated and full security test scenarios. Table 4 and Table 5 describe PMP and PMA configuration of the RISC-V SoC. These tables are then used to populate various attribute values, define constraints in the PSS model. For brevity, only key table columns are discussed.

Table 4 – Physical Memory Protection (PMP) table - describes PMP entries in CSV tables. Some important columns are:

- #region: Unique physical memory region identifier (integer)
- #start_pa: Start address of physical memory region of this PMP entry
- #size: Size of the region
- #region_type: Support address-matching modes (The 'A' field of PMP configuration register)
- #region_permissions: Read, Write, Execute permission and L-bit

#region	#start_pa	#size	#region_type	#region_permissions
0	0x0000_0000	2G	TOR	L,R,W,X
1	0x8000_0000	2M	TOR	L,R,W,X
2	0x9000_0000	2M	TOR, NA4, NAPOT	W,X
3	0x9000_0000	16M	TOR, NA4, NAPOT	W,X
4	0x9200_0000	2M	NAPOT	R
5	0x9200_0000	8M	TOR, NA4, NAPOT	R,W
6	0x9300_0000	2M	TOR, NA4, NAPOT	R,W,X
7	0x9400_0000	16M	TOR, NA4, NAPOT	L,R,W
8	0x9200_0000	16M	TOR, NA4, NAPOT	L,R
9	0x9500_0000	8M	TOR, NAPOT	L,R,W,X
10	0x9600_0000	2M	TOR, NAPOT	W,X
11	0x9620_0000	2M	TOR, NA4, NAPOT	X
12	0x9640_0000	2M	TOR, NA4, NAPOT	L,R,X
13	0x9660_0000	2M	TOR, NA4, NAPOT	L,X
14	0x9680_0000	2M	TOR, NA4, NAPOT	L,W
15	0x96A0_0000	2M	TOR, NA4, NAPOT	L,W,X

Table 4: Physical Memory Protection (PMP) table

Table 5 – Physical Memory Attributes (PMA) table – describes the Shareability, Cacheability, security attributes of a memory region. Virtual to physical address mapping (address translation) information is also captured in this table. Key table columns are:

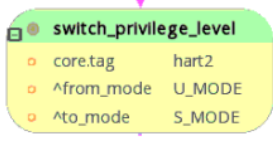

- #va: virtual address of the memory region
- #pa: physical address of the memory region
- #mem_block: memory region name. This example has 3 blocks: mem0, mem1, mem2
- #size: size of the memory region
- #shareability: specifies if the memory region is shareable or not

#va	#pa	#mem_block	#size	#shareability
0x9000_0000	0x9000_0000	mem0	128M	shareable
0x9800_0000	0x9800_0000	mem1	32M	shareable
0xA000_0000	0xA000_0000	mem2	32M	shareable

Table 5: Physical Memory Attributes (PMA) table

Once the PMP and PMA tables were filled out, PSS atomic actions were modeled. Here are few atomic actions that were implemented that serve as building blocks to create larger SoC level scenarios:

- Privilege Mode switching – M, S, U (Table 1) RISC-V privilege modes. This action generates code for privilege mode switching.
- Select a PMP region and randomize it's attributes. This action generates code to program a PMP region.
- Generic actions to write and read memory regions.
- Action to select and program specific RISC-V CPU registers.
- Actions to install trap handlers.

Action Name	UML Diagram	Description
switch_privilege_level		Switch hart execution mode between M, S, and U modes.
set_pmp_full_access		Permit S & U modes full R, W, X permissions to access all PMP entries

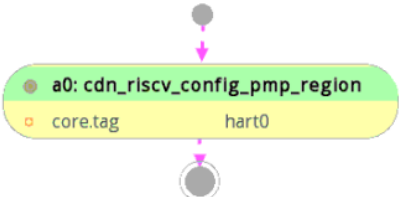
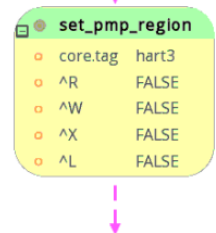

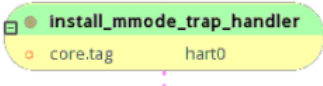
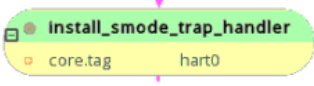
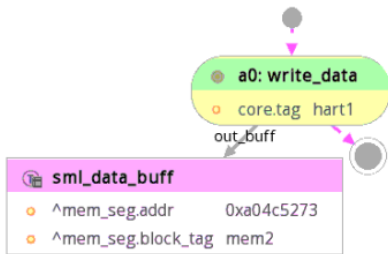
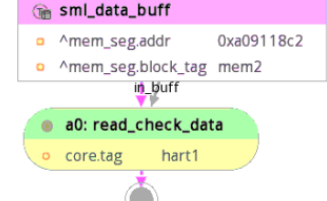
cdn_riscv_config_pmp_region		Configure (program) PMP registers based on the configuration specified in PMP table
set_pmp_region		Configure (re-program) specific permissions of a selected PMP region
cdn_ps riscv_reg_write		Program RISC-V system register with specific value
install_mmode_trap_handler		Override user M-mode (Table1) trap handler
install_smode_trap_handler		Override user S-mode (Table1) trap handler
write_data*		Write random data of specified size to a selected memory block
read_check_data*		Read and check (previously writer) data from a selected memory block

Table 6: Atomic actions required for complex scenarios

* Actions available in Perspec coherency library.

Step-3: Develop security test scenarios targeting PMP features

This section describes, using an example, how to create a PSS test scenario to target PMP features, using PSS atomic actions developed in the previous step.

Creating a legal test scenario: pmp_write_data_in_smode

- Do same set of operations concurrently on each of the selected harts:
 - Enable address translations in S(supervisor)-mode by programming SATP (Supervisor Address Translation and Protection) register
 - Configure PMP entries as defined in PMP table (Table 4)
 - Switch privilege level from M to S mode (Table 1)
 - Perform write access in S-mode

Figure 4 shows PSS model for the test scenario describe a bove

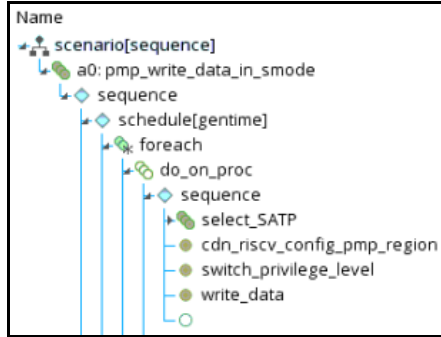


Figure 4: PSS Scenario for pmp_write_data_in_smode

Figure 5 shows UML diagram of the PSS implementation of pmp_write_data_in_smode scenario described above. Figure 6 shows the 2nd solution generated by PSS tool for different assignment of cores and memory blocks

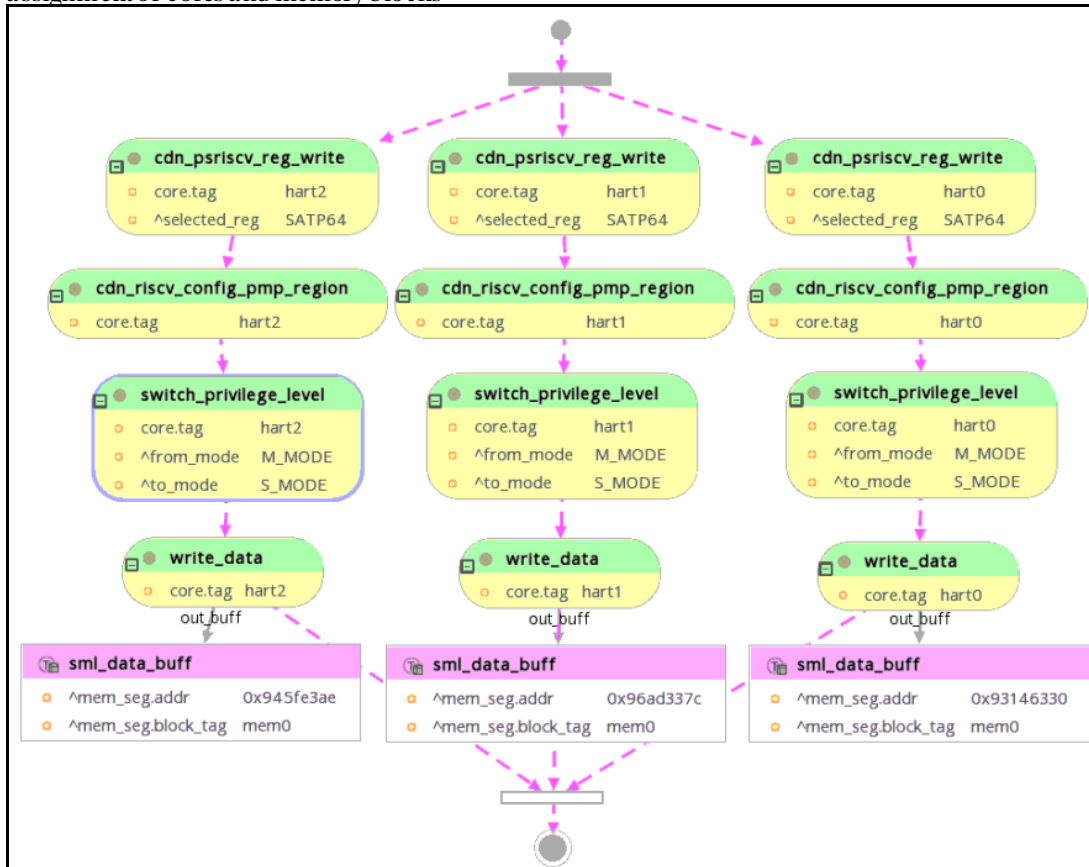


Figure 5: UML diagram of pmp_write_data_in_smode test scenario

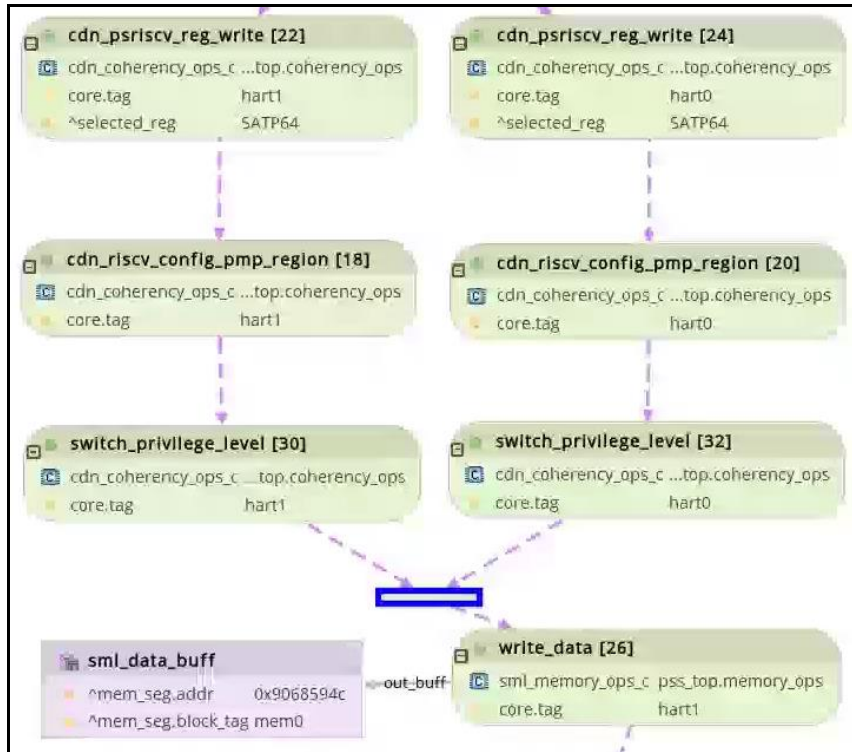


Figure 6: UML diagram of pmp_write_data_in_smodetest scenario

Few observations on above scenario:

- A legal test is automatically generated due to PSS model constraints. In this case, only the addresses that S-mode has permission to write are generated. The generated addresses correspond to PMP table (Table 4) regions 6, 7, and 15 – all of which permit write accesses.
- All the harts are trying to trigger PMP checks concurrently during write accesses.
- Another solution of the same PSS scenario (in Figure 4) would produce a slightly different scenario:
 - The number of harts participating in the scenario is random
 - The addresses generated for write accesses in S-mode is also random, which means they target different PMP regions with different permissions.

We were able to create the first test scenario described above in a few minutes, using atomic actions developed in Step-1 and native PSS operators (like sequence, parallel etc.) provided by Perspec tool.

Creating a negative test scenario: pmp_region_gaps_with_exception

- On a selected hart, do the following:
 - Install trap handlers to handle exception due to PMP violations
 - Enable address translations in S-mode
 - Configure all PMP region based on the PMP table (Table 4)
 - Switch privilege level from M to S mode (Table 1)
 - Do a write access to a PMP region that has no write permission.

Figure 7 shows PSS scenario description and Figure 8 shows the UML diagram for the test scenario **pmp_write_data_with_exception**

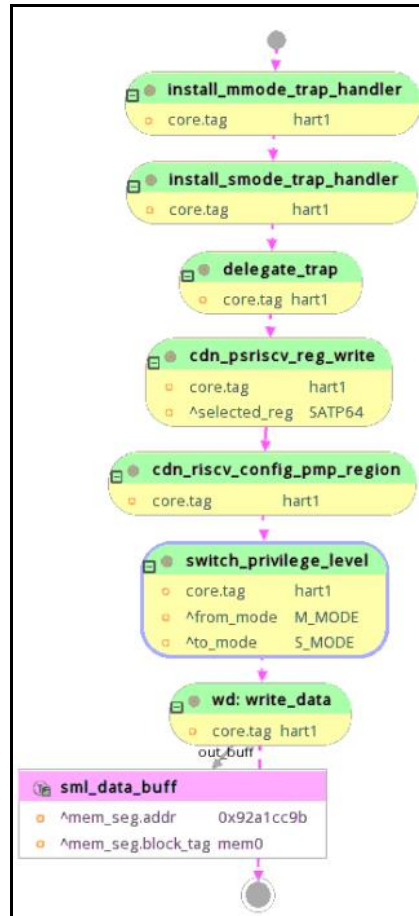
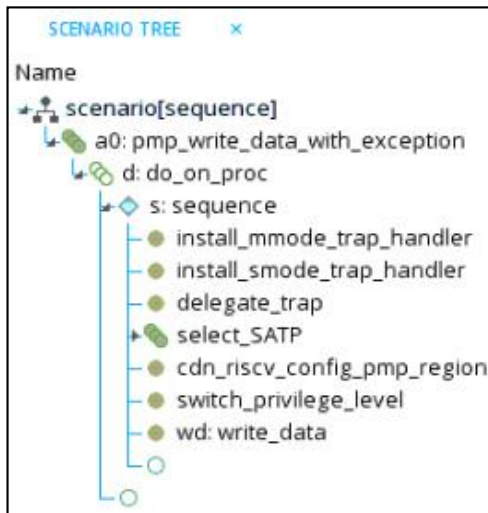


Figure 7: pmp_write_data_with_exception test scenario

Figure 8: UML diagram of pmp_write_data_with_exception test scenario

For negative tests, an exception is expected and that will be handled gracefully by the trap handler. The test continues after the exception and executes till the end. The test is deemed a 'pass' only when the expected exception happens after an operation (like memory access).

In this section, we have demonstrated how easily positive as well as negative test scenarios are created by mixing PMP atomic actions, Perspec library actions and Perspec native operators. The next section describes several other test scenarios developed in the same way.

Step-4: Create test variations to cover concerning cross of PMP features

An exhaustive test plan for verifying PMP functionality in the SoC requires multiple tests targeting various corner cases and negative conditions. The remainder of this section briefly describes several other scenarios we were able to create in a relatively short amount of time.

pmp_region_overlap: Validate PMP privilege accesses with overlapping memory address regions

pmp_change_lock: Enable PMP lock bit to force privilege access checks even in machine mode

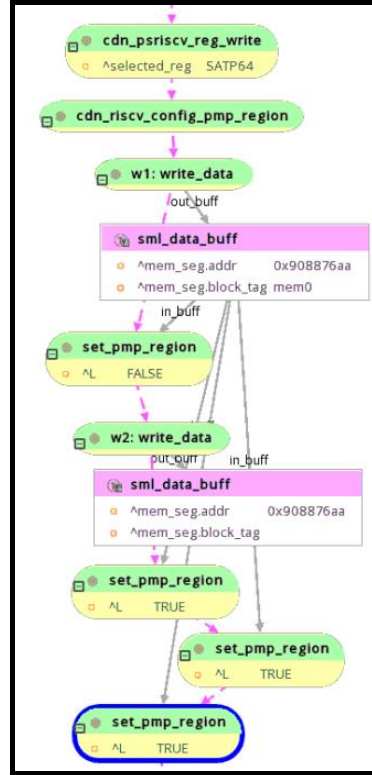


Figure 9 (left): UML diagram of pmp_region_overlap test scenario

Figure 10 (right): UML diagram of pmp_change_lock test scenario

For C code snippets, PSS can randomize different values defined in PMP Table 4 (ie: region type, permissions, etc) based on different PMP entries to hit verification holes in comparison with direct C test

```
setPMP(0, ((uint64_t)0x1fffffff), 0x8f); // Region 0, Start: 0x0, Size: 0x80000000, A:TOR,X:1,W:1,R:1
setPMP(1, ((uint64_t)0x2007ffff), 0x8f); // Region 1, Start: 0x80000000, Size: 0x200000, A:TOR,X:1,W:1,R:1
setPMP(2, ((uint64_t)0x2403ffff), 0x1e); // Region 2, Start: 0x90000000, Size: 0x200000, A:NAPOT,X:1,W:1,R:0
setPMP(3, ((uint64_t)0x241fffff), 0x1e); // Region 3, Start: 0x90000000, Size: 0x1000000, A:NAPOT,X:1,W:1,R:0
setPMP(4, ((uint64_t)0x2483ffff), 0x19); // Region 4, Start: 0x92000000, Size: 0x200000, A:NAPOT,X:0,W:0,R:1
setPMP(5, ((uint64_t)0x249fffff), 0xb); // Region 5, Start: 0x92000000, Size: 0x800000, A:TOR,X:0,W:1,R:1
setPMP(6, ((uint64_t)0x24cfffff), 0xf); // Region 6, Start: 0x93000000, Size: 0x200000, A:TOR,X:1,W:1,R:1
setPMP(7, ((uint64_t)0x251fffff), 0x9b); // Region 7, Start: 0x94000000, Size: 0x1000000, A:NAPOT,X:0,W:1,R:1
setPMP(8, ((uint64_t)0x24bfffff), 0x89); // Region 8, Start: 0x92000000, Size: 0x1000000, A:TOR,X:0,W:0,R:1
setPMP(9, ((uint64_t)0x255fffff), 0x8f); // Region 9, Start: 0x95000000, Size: 0x800000, A:TOR,X:1,W:1,R:1
setPMP(10, ((uint64_t)0x2587ffff), 0xe); // Region 10, Start: 0x96000000, Size: 0x200000, A:TOR,X:1,W:1,R:0
setPMP(11, ((uint64_t)0x258bffff), 0x1c); // Region 11, Start: 0x96200000, Size: 0x200000, A:NAPOT,X:1,W:0,R:0
setPMP(12, ((uint64_t)0x2593ffff), 0x9d); // Region 12, Start: 0x96400000, Size: 0x200000, A:NAPOT,X:1,W:0,R:1
setPMP(13, ((uint64_t)0x259fffff), 0x8c); // Region 13, Start: 0x96600000, Size: 0x200000, A:TOR,X:1,W:0,R:0
setPMP(14, ((uint64_t)0x25a3ffff), 0x9a); // Region 14, Start: 0x96800000, Size: 0x200000, A:NAPOT,X:0,W:1,R:0
setPMP(15, ((uint64_t)0x25abffff), 0x9e); // Region 15, Start: 0x96a00000, Size: 0x200000, A:NAPOT,X:1,W:1,R:0
```

Figure 11: C functions of setPMP regions

All above test scenarios have been successfully modeled and validated in just a few weeks once the PSS model described in Step-3 is ready. We also successfully simulated the Perspec generated code on a RISC-V reference platform.

III. CONCLUSION

This paper shows how PSS and EDA tools, such as Perspec, enable efficient verification of complex SoC level security scenarios. The main technical contributions are:

- Out-of-box atomic actions and scenarios are ready to be used flexibly
- Create large number of tests, covering all crosses, in a relatively short amount of time.

- Model both positive and negative security test scenarios with PSS.
- RISC-V security verification approach focusing on PMP features.

REFERENCES

- [1] Coherency Verification & Deadlock Detection Using Perspec/Portable Stimulus: <https://dvcon-proceedings.org/document/coherency-verification-deadlock-detection-using-perspec-portable-stimulus/>
- [2] Portable Test and Stimulus Standard Version 2.0: <https://www.accellera.org/downloads/standards/portable-stimulus>
- [3] RISC-V Privileged Spec: <https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf>
- [4] Perspec System Verifier: https://www.cadence.com/zh_TW/home/tools/system-design-and-verification/software-driven-verification/perspec-system-verifier.html