Recursive Privilege Architecture: A Unified Model for Extensible Isolation

Yongkang Liu

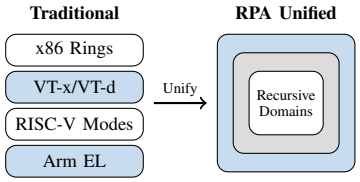

Fig. 1. Privilege fragmentation vs. RPA unification.

***Abstract*—Modern processor privilege architectures face a fundamental limitation: privilege levels are fixed by hardware. Each new requirement—virtualization, trusted execution, memory protection—demands separate hardware extensions. We propose the Recursive Privilege Architecture (RPA), which transforms privilege configuration from hardware-fixed to dynamically configurable through two symmetric primitives (`descend` and `ascend`). RPA flexibly supports systems from microcontrollers to high-performance CPUs. Key features include: page table stacking for nested virtualization, INHERIT mode for lightweight isolation with near-function-call overhead, and security domains separating management rights from access rights. A Python executable specification with 91 test cases validates the design.**

***Index Terms*—Privilege architecture, virtualization, confidential computing, memory protection, processor design**

I. Introduction

MODERN processors implement privilege separation through fixed hardware mechanisms. x86 defines four privilege rings, ARM provides four execution levels, and RISC-V specifies three modes. These share a fundamental limitation: fixed privilege levels cannot adapt to diverse isolation requirements. High-end servers need nested virtualization, while embedded systems need lightweight isolation but cannot afford complete privilege architecture overhead.

This limitation has led to fragmentation. Each requirement forces specific extensions—VT-x for virtualization, SGX/TDX for trusted execution, MPK for memory protection—resulting in complexity and redundancy.

We propose the Recursive Privilege Architecture (RPA), addressing this through three contributions: (1) **Recursive Privilege Model**—each domain views itself as Domain 0, children as Domain 1, 2, 3, ..., and parent as Domain -1, enabling unlimited nesting depth; (2) **Unified Primitives**—two symmetric primitives replace heterogeneous mechanisms; (3) **ISA-Independent Design**—RPA requires only two new instructions.

Manuscript received May 2024.

The author is with HappyFull Technologies, Shanghai, China (e-mail: yongkang.liu0814@gmail.com). ORCID: 0009-0006-3009-0037.

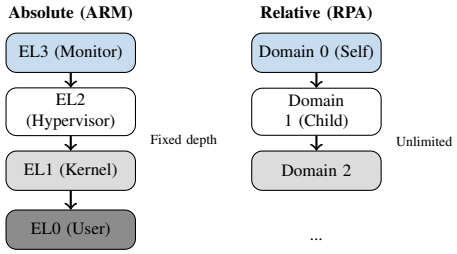

Fig. 2. ARM fixed privilege levels vs. RPA relative perspective.

We validate RPA through a Python executable specification with 91 test cases [1].

II. Background and Motivation

Traditional Architectures. x86 implements four privilege rings via CPL. Ring 0 (kernel) has full access; Ring 3 (user) is restricted. Transitions via syscall/sysret cost 100–500 cycles [2]. ARM provides four execution levels (EL0–EL3) [3]. World switches via SMC cost 50–200 cycles [4]. RISC-V defines three privilege modes; virtualization requires H-extension.

Hardware Isolation Extensions. Intel MPK provides 16 protection domains with 1–10 cycle switches [5], but no kernel protection. Intel SGX/TDX and ARM TrustZone lack nesting support. CHERI introduces capabilities for memory safety [6], orthogonal to RPA's focus.

Scalability Challenges. Nested virtualization requires software emulation with 10–30% overhead per level [7]. Existing architectures lack scalability for both high-end and low-end systems.

III. RPA Architecture

A. Recursive Privilege Model

RPA's fundamental innovation is the *relative perspective*: each domain views itself as Domain 0, children as Domain 1, 2, 3, ..., and parent as Domain -1. This contrasts with traditional architectures where privilege levels are absolute. RPA organizes domains as a tree: Domain 0 is the Root of Trust; Domain $N + 1$ is a child of Domain N, created by the parent's `descend` operation. Hierarchy depth is bounded by memory for domain control blocks, not by hardware-fixed limits.

B. Core Primitives

RPA defines two symmetric primitives: **`descend`(control_block_addr)** enters a child domain; **`ascend`(service_type)** returns to the parent. These replace traditional mechanisms: system calls (user calls `ascend`), VM entry/exit (guest uses `ascend`, hypervisor uses `descend`), TEE switches (normal world calls `descend`), and exception handling (triggers implicit `ascend`). The symmetry simplifies hardware design: both primitives share the same context-switch infrastructure, differing only in direction.

C. DomainControlBlock

The DomainControlBlock (128 bytes, aligned to 8-unit boundaries) resides in ordinary memory. Key fields include: `ctrlblock_size` (parent-set), `exception_vector` (child-set), `ipa_regions` (parent-set IPA boundary), `domain_id` (system-set), `pagetable` (child-set), `security_domain` (system-set), and `ctx_save_area` (ISA-specific context). **Security Principle:** The parent creates and manages the child's control block; the child cannot access its own control block through direct memory access. A parent can create multiple child control blocks to implement multiple VMs or processes.

D. Page Table Stacking

RPA supports page table stacking: each domain can have its own page table translating to the parent's address space. For Domain $N+1$ accessing VA: (1) translate through PT_{N+1} to IPA_N; (2) check against `ipa_regions`; (3) continue through PT_N; (4) repeat until physical memory.

INHERIT Mode: When pagetable = 0, the child shares the parent's address space with only register context save/restore, enabling near-zero-overhead exception interception. This mode is suitable for: (1) small processors without address mapping capabilities; (2) simple protection regions within ordinary programs.

E. Security Domains

Security domains distinguish *management rights* (parent configures child's IPA regions, page tables) from *access rights* (parent cannot read child's private memory). End users can create a security domain to enroll system components; members outside the domain cannot access data within. Attributes include: `isolated` (independent domain) and `confidential` (inaccessible outside the domain; memory encryption is the underlying mechanism).

IV. System Integration

ISA Integration. RPA is ISA-agnostic. The `ctx_save_area` field stores ISA-specific register state. Primitives can be mapped to: (1) dedicated instruction encodings, (2) existing privileged instructions with immediates, or (3) coprocessor interface.

Exception Handling. When an exception occurs in a child domain, hardware performs an implicit `ascend`, transferring control to the parent's exception handler. The child sets `exception_vector` to receive its own exceptions.

Memory Management. `pagetable = 0` shares parent's address space (INHERIT mode); `pagetable != 0` uses independent page table. The parent constrains child's accessible range via `ipa_regions`.

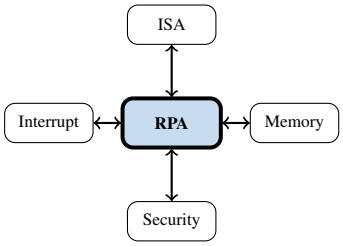

Fig. 3. RPA integration with system modules.

TABLE I
Privilege Architecture Comparison

Feature	x86	ARM	RISC-V	RPA
Levels	Fixed (4)	Fixed (4)	Fixed (3)	Config.
Nesting	VT-x	No	H-ext	Native
Confidential	SGX/TDX	TrustZone	Custom	SecDom
Transition	100-500c	50-200c	10-100c	1-10c†
ISA change	No	No	No	Minimal

†INHERIT mode. Overhead estimates from published data [2], [8].

V. Comparison

Table I compares RPA with existing architectures.

Extensibility. x86, ARM, and RISC-V have fixed privilege levels. RPA supports unlimited nesting bounded by memory, not hardware.

Unification. Traditional architectures use different mechanisms for syscalls, VM entry/exit, and SMC. RPA unifies these into `descend`/`ascend`, enabling composition (e.g., TEE inside VM).

Confidential Computing. Intel SGX/TDX and ARM CCA lack nesting. RPA security domains support nesting with management/access rights separation.

Limitations. RPA introduces hardware complexity for multi-level translation. Each domain requires 128 bytes. RPA provides privilege separation, not memory safety (orthogonal to CHERI).

VI. Evaluation

Executable Specification. We implemented RPA as a Python-based executable specification (∼1,500 lines), inspired by CHERI and RISC-V formal specification approaches [9], [10]. The artifact is available [1].

Functional Verification. We developed 91 test cases covering domain operations (22), page table translation (18), IPA boundary checking (15), security groups (12), interrupt handling (10), memory encryption (7), and exception propagation (6). All tests pass, validating: (1) recursive privilege model is semantically viable; (2) two primitives suffice for all transitions; (3) INHERIT mode behaves correctly.

TABLE II
THEORETICAL TRANSITION OVERHEAD

Operation	INHERIT	Indep.	Trad.
Reg save/restore	5–10c	50–100c	100–300c
Page table switch	0	50–100c	50–100c
TLB flush	0	100–500c	100–500c
Total	**5–10c**	**200–700c**	**250–900c**

Traditional overhead from [2], [8].

Performance. Table II shows INHERIT mode reduces transition to 5–10 cycles, comparable to function calls. With independent page tables, overhead resembles traditional system calls, but nesting efficiency improves.

VII. DISCUSSION

Hardware Considerations. DomainBlocks (128 bytes each) can reside in dedicated SRAM, tagged memory, or ordinary memory. For 16 concurrent domains, 2 KB suffices. TLB entries must include domain ID to avoid flushing on domain switch.

Security. Children cannot modify their own DomainBlocks through memory protection or hash verification. IPA boundary checking prevents memory constraint escapes. RPA does not address side-channel attacks—orthogonal to privilege separation.

Relationship with CHERI. CHERI provides memory safety through capabilities; RPA provides extensible privilege hierarchy. They are complementary: CHERI capabilities could operate within each RPA domain.

Future Work. Following CHERI's precedent, we validate through specification and functional tests. Future work includes: formal Sail specification, gem5 simulation, FPGA prototype, and OS integration.

VIII. CONCLUSION

We presented the Recursive Privilege Architecture (RPA), a unified model for extensible isolation transforming privilege configuration from hardware-fixed to memory-configurable. Key innovations: (1) recursive privilege model with unlimited nesting depth; (2) two symmetric primitives replacing heterogeneous mechanisms; (3) ISA-independent design with only two new instructions; (4) INHERIT mode reducing transition overhead to function-call levels. A Python executable specification with 91 test cases validates the design.

REFERENCES

[1] RPA Project, "RPA Python Simulator: Executable Specification of Recursive Privilege Architecture," https://github.com/arthurdev000/rpa-pysim, 2024, version 1.0.

[2] L. Soares and M. Stumm, "FlexSC: Flexible system call scheduling with exception-less system calls," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010, pp. 1–8.

[3] *ARM Architecture Reference Manual ARMv8, for ARMv8-A Architecture Profile*, ARM Limited, 2023, document DDI0487.

[4] *ARM TrustZone Technology*, ARM Limited, 2023, ARM Developer Documentation.

[5] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, 2023, order Number: 325384-079US.

[6] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," *IEEE Symposium on Security and Privacy*, pp. 20–37, 2015.

[7] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010, pp. 423–436.

[8] "Performance Best Practices for VMware vSphere 7.0," Technical White Paper, 2022.

[9] CHERI Project, "CHERI Sail Model," https://github.com/CTSRD-CHERI/sail-cheri-riscv, 2023, accessed: 2024.

[10] RISC-V International, "RISC-V Sail Model," https://github.com/riscv/sail-riscv, 2023, accessed: 2024.