Caplification: Bridging Capability-Aware and Capability-Oblivious Software

Jason Zhijingcheng Yu
National University of Singapore
Singapore
yu.zhi@comp.nus.edu.sg

Mingkai Li*
Columbia University
New York, NY, USA
mingkai.li@columbia.edu

Aditya Badole
National University of Singapore
Singapore
adityabadole@outlook.com

Trevor E. Carlson
National University of Singapore
Singapore
tcarlson@comp.nus.edu.sg

Michael Swift
University of Wisconsin-Madison
Wisconsin, MI, USA
swift@cs.wisc.edu

Prateek Saxena
National University of Singapore
Singapore
prateeks@comp.nus.edu.sg

Abstract

Hardware capabilities offer an alternative to how access control is commonly implemented in processors today, i.e., through enforcement of permission checks on virtual memory at the time of address translation. Despite conceptual strengths of capability hardware, it is challenging for existing capability-oblivious software stacks to be compatible with capability hardware and to interoperate with capability-aware software, hindering faster adoption. Prior attempts to achieve this sacrifice the inherent advantages of capabilities. They require trusting a software central authority (e.g., the OS kernel) for capability-based isolation and limit the scope of capability-based memory sharing to individual virtual address spaces. This paper proposes the idea of *caplification*, a novel mechanism to enable seamless co-existence of capability-aware and capability-oblivious software stacks. We concretely implement our proposed idea on a modern RISC-V capability hardware and show how it enables running a commodity unmodified (or capability-oblivious) Linux OS. Our design retains the full advantages provided by hardware capabilities, such as creating fine-grained hardware-isolated memory compartments both in user- and kernel-space. We evaluated our prototype system both on QEMU emulation and on hardware RTL simulation. We find that the performance of our system is comparable to prior baseline designs, while offering cost improvements in scenarios of secure data sharing.

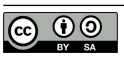
CCS Concepts

• **Security and privacy → Operating systems security**; • **Computer systems organization → Architectures**.

Keywords

Capability-based security; Memory isolation; Virtual memory; Operating systems

*Work done during a research internship at National University of Singapore.



SACMAT '25, Stony Brook, NY, USA

ACM ISBN 979-8-4007-1503-7/2025/07
https://doi.org/10.1145/3734436.3734449

ACM Reference Format:
Jason Zhijingcheng Yu, Mingkai Li, Aditya Badole, Trevor E. Carlson, Michael Swift, and Prateek Saxena. 2025. Caplification: Bridging Capability-Aware and Capability-Oblivious Software. In *Proceedings of the 30th ACM Symposium on Access Control Models and Technologies (SACMAT '25), July 8–10, 2025, Stony Brook, NY, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3734436.3734449

1 Introduction

Isolation plays an important role in preventing undesired interference between software components. Modern systems have predominantly relied on virtual memory access control (VMAC) for isolation (Figure 1 (a)), where a trusted central authority (e.g., the OS kernel) configures a global isolation policy (e.g., via page tables). The central authority is entrusted to effect all permission changes on behalf of all security principals running in isolated domains.

Hardware capability-based security [14, 28], in contrast, uses *capabilities* to implicitly define isolation policies. A capability is an unforgeable token that binds together a memory location and the permissions granted for accessing it. Hardware enforces isolation policies based entirely on the capability presented for each memory access, regardless of which software principal presents it. Prior work has discussed the benefits of capability-based security over VMAC-based mechanisms [16, 28, 36, 57, 58, 62, 65, 68]. Capabilities reduce the trusted computing base (TCB) by removing trust in a central authority, prevent bugs such as the confused deputy [9] by avoiding ambient authority, and facilitate resource sharing thanks to their context-independent nature. More recent work [68] has extended the base capability model with support for exclusive ownership and revocable delegation, further improving the expressiveness of fine-grained memory safety and flexible isolation.

However, one important challenge to the practicality of hardware capabilities is in providing *compatibility and interoperability between capability-based models and capability-oblivious software stacks.*

Prior capability-based designs either do not directly support capability-oblivious software [68] (Figure 1 (b)) or have important limitations. Most follow the strategy of stacking capabilities on top of virtual memory [7, 15, 17, 26, 61, 63] (Figure 1 (c)). Capabilities in such designs reference virtual addresses rather than physical ones. A prominent example is CHERI [61].

Such a strategy sacrifices two important properties of capabilities. Firstly, capabilities are no longer context-independent. Rather,

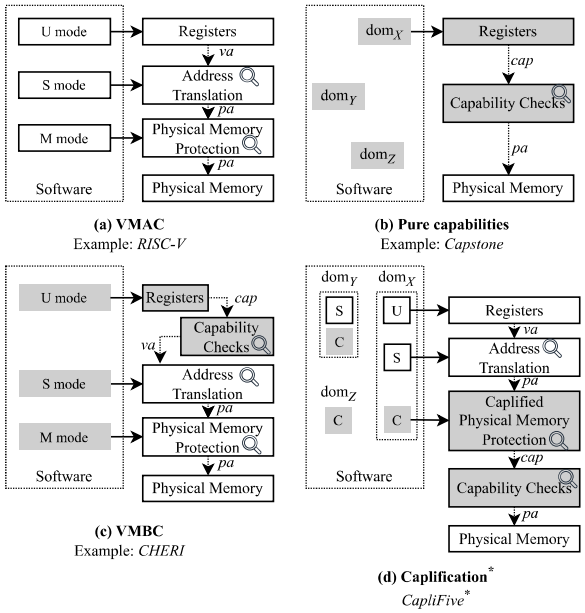


Figure 1: Comparison of hardware isolation mechanisms, using RISC-V as an example. Solid arrows represent writable permissions. Dotted arrows represent the pathways for software to access physical memory. Shaded regions represent structures that contain capabilities or capability-aware software components. The asterisked mechanism (d) is our work. Labels *va*, *pa*, and *cap* represent virtual addresses, physical addresses, and capabilities, respectively. The magnifying glass icon (🔍) indicates locations of checks on the memory access.

the memory resources a capability references now depend on the virtual address space it is used in, and consequently software can no longer share memory by directly passing capabilities. Secondly, it re-introduces ambient authority in the OS kernel, which is still in the TCB even for capability-based isolation. A malicious or buggy kernel can compromise capability-based isolation by corrupting page mappings or permissions. This limits existing designs to be applicable largely to in-process isolation on a trusted OS.

Our work. We introduce the concept of *caplification* designed to address this problem. Caplification allows existing software that is not capability-aware, i.e., is capability-oblivious, to work seamlessly on a hardware platform with hardware capabilities. The key idea is simple: It turns capability-oblivious *virtual memory* accesses into capability-based accesses by adapting existing memory protection data structures (e.g., page tables) to be capability-based (Figure 1 (d)). We refer to a data structure thus adapted as *caplified* (short for *capabilitified*). A traditional capability-oblivious system allows privileged software to configure the address translation data structures with arbitrary physical addresses and access permissions (Figure 1 (a)). Instead, in a caplified address translation system, the virtual address maps to a physical capability. This way the software uses regular virtual memory accesses, but these are translated into hardware capabilities under the hood. Since *all memory accesses are ultimately capability-based,* caplification retains the full benefits of capabilities, including context independence and eliminating the need for trusted central authority in software. Caplification thus provides a consistent way for software following the two programming models—capability-oblivious and capability-based—to interact seamlessly, enabling incremental migration of existing capability-oblivious software to a capability-based model. As caplification retains all the advantages of hardware capabilities, it enables capabilities as a fundamental isolation primitive in various software contexts, e.g., both user-space and kernel-space.

We demonstrate the concept of caplification on a concrete hardware and software co-design called CapliFive. CapliFive caplifies the PMP (Physical Memory Protection) data structure on RISC-V [59, 60] to fuse Capstone [68], a recent capability-based design, with the capability-oblivious model of RISC-V. We present both functional emulation on QEMU [5] and hardware RTL implementations of CapliFive, and show that it:

(1) supports unmodified **capability-oblivious Linux stack**;

(2) runs capability-aware fault domains **in both user- and kernel-space**, on an untrusted capability-oblivious Linux kernel;

(3) allows capability-oblivious software to share virtual memory pages with capability-based software, while utilizing hardware capability-based isolation underneath for both.

All three points are difficult to achieve with prior designs. For example, the strategy proposed in [61] cannot achieve (2) and (3), while it requires extensive software changes to the Linux kernel to achieve (1). Section 3 discusses prior approaches in more detail and explains how they fail to achieve the above goals. CapliFive shows for the first time how to run a commodity OS like Linux on capability-based architectures, without intrusive changes to the OS and without sacrificing the full benefits of hardware capabilities.

Contribution. We propose caplification, a novel way to fuse hardware capabilities into a capability-oblivious software stack. Our prototype design, CapliFive, and its implementation are both publicly available [39, 40, 53, 67].

2 Background

Virtual memory access control (VMAC). The predominant way modern hardware enforces security protection on memory is through access control at the point of address translation, i.e., when virtual memory is translated to physical memory via page tables. It relies on ambient authority to control which software can modify the page table. Software runs at one of a fixed number of privilege levels. For example, RISC-V [60] specifies three privilege levels: U-mode, S-mode, and M-mode. An OS kernel, typically running in S-mode, is allowed to configure the page tables of processes, typically running in U-mode, merely by virtue of running at a high privilege level.

Hardware capabilities. In contrast to VMAC, hardware capabilities are physical memory pointers which carry with them the security access permissions directly. It is sufficient and necessary for a software running on a capability-based hardware architecture to present a valid capability to access the corresponding memory. The validity of the access is checked by the hardware directly, without the need to consult privileged software at the time of access. Granting of permissions to a memory location is implicit when a software passes the associated capability; no specific data structures

need to be updated explicitly. This is in contrast to how isolation policies modifications are effected in VMAC.

Advantages of capabilities. Prior work has demonstrated advantages of capability-based isolation over VMAC [28, 32]. Capabilities avoid ambient authorities [28] since isolation policy changes are implicitly handled with capability passing, rather than explicitly as in VMAC. Capabilities provide finer-grained memory protection than virtual pages, making them also suitable for ensuring memory safety [15, 62], not just for isolation. Capabilities also have context-independent semantics that does not change with its environment, facilitating data exchange across the whole software system.

For example, consider an isolated software component *A* wishing to share a memory region with another isolated software component *B*. For VMAC-isolated components to share memory, the OS kernel needs to set up page mappings in their respective virtual address spaces to shared physical memory frames. This both adds to the performance overhead and brings the OS into the TCB. VMAC requires that *A* and *B* run in the unprivileged user-space. In contrast, in a capability-based system, *A* merely needs to delegate a capability to *B*, which is a cheap hardware-supported operation. In addition, capabilities work in any context: They work when *A* and *B* are kernel modules as well as when they are user applications.

New capability hardware. Existing work highlights some perils with hardware capability designs [28], such as capability leaks. Recent hardware capability designs, such as Capstone [68], is designed to address these and improve expressiveness over the classical idea of capabilities. In the example above, such new designs allow *A* to delegate its memory region to *B* in such a way that (1) guarantees *B* exclusive access to the memory region, and (2) enables *A* to regain its own access to the memory region and revoke *B*'s access. In a setup where *A* and *B* distrust each other, such sharing primitives avoid security issues such as time-of-check-to-time-of-use (TOCTTOU) attacks [13]. They also avoid extra data copies which are necessary with classical hardware capabilities.

3 Problem

Hardware capabilities present a different programming model from the traditional VMAC mechanism. Capability-oblivious software which uses VMAC only provides a memory address when accessing memory, whereas in capability-based designs, software needs to explicitly present a capability. This has been a roadblock to the adoption of hardware capabilities. Compatibility and interoperability between capabilities and capability-oblivious software are thus crucial goals. We summarize the desired goals as follows:

(G1) supporting unmodified capability-oblivious software stacks;
(G2) supporting isolated capability-aware fault domains in both user- and kernel-space without trusting the privileged software;
(G3) enabling capability-aware and capability-oblivious software to share memory using capabilities as the underlying mechanism.

Issues with prior work. Prior hardware-capability-based designs, summarized in Table 1, are *not designed to* achieve the above goals together. Most existing efforts to retrofit capabilities into capability-oblivious software systems place capabilities on top of virtual memory [62]. Specifically, bound and permission checks on addresses are performed before address translation and page-based permission checks. We refer to this strategy as *virtual-memory-backed capabilities* (VMBC), illustrated in Figure 1 (c). The privileged software such as the OS kernel has complete control over the address translation process. As such, the OS kernel needs to be trusted to enforce proper isolation. When software components intend to share memory through capabilities, they must also rely on the OS kernel to map the shared virtual pages to the same physical locations.

Many such VMBC-based designs have been proposed, but they fail to achieve the above goals G1-G3. CHERI [62] adopts VMBC and supports a hybrid mode that allows mixing capability-aware and capability-oblivious code. It fails to achieve goals G2 or G3, as evidenced by the existing software stacks built on it. For example, CheriBSD [41, 62] is a FreeBSD variant adapted to use CHERI capabilities for memory safety. It requires full trust in the kernel, and supports capability-based isolation only for in-process isolation. CheriOS [16] is a single-address-space OS on CHERI which uses capabilities for isolating processes. It relies on a trusted nanokernel for isolation through heavy use of software-enforced (rather than hardware) capabilities. Similarly, CHERI-TrEE [56] builds TEE primitives using CHERI capabilities. CHERI-TrEE relies on a trusted monitor for critical tasks such as memory allocation and revocation. Both CheriOS and CHERI-TrEE require software stacks that are built from scratch and capability-aware. It is unclear how either CheriOS or CHERI-TrEE supports existing capability-oblivious software stacks. CAP-VM [44] uses the intravisor, a trusted in-process component, to isolate multiple compartments within the same virtual address space with capabilities. It uses the CHERI hybrid mode [61] to support unmodified applications which are capability-oblivious. But to run capability-based software, the code has to run inside a process with a dedicated virtual address-based capability, for which the OS has to be trusted, and has no direct way of passing capabilities to other capability-aware software domains. ORC [45] replaces the CHERI hybrid mode design in CAP-VM with the pure capability mode (thus requiring applications to be capability-aware) to sacrifice compatibility for finer-grained sharing and more efficient memory deduplication. Similar to CAP-VM, ORC relies on a trusted intravisor for all granting and revoking all capabilities, and imposes tight constraints on the use of capabilities (e.g., they cannot be stored in memory).

Apart from VMBC, other prior designs also fail to achieve G1–G3. For example, CODOMs [57, 58] is a design that does not adopt VMBC. Instead, it exposes two ways of addressing memory directly, regular virtual memory addresses and hardware capabilities. It has a fixed set of registers which translate into hardware capabilities, which is different from our goal of making arbitrary virtual address capability-based. The full benefits of capability-based memory addresses, for instance providing memory safety, cannot be leveraged using such a design. In addition, the design has the drawback of relying on a software TCB, including the OS kernel.

4 Caplification of a Software Stack

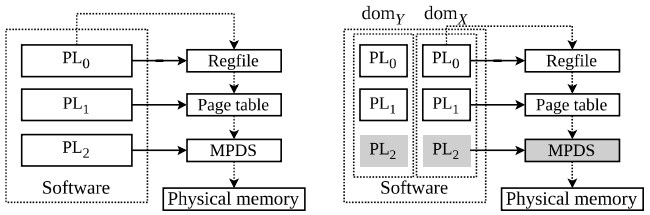
We propose *caplification*, a conceptually simple but novel way to achieve all three goals G1–G3 simultaneously.

4.1 Core Idea of Caplification

A conceptual isolation model. Caplification enables a novel isolation model illustrated in Figure 1 (d). The software is divided into

Table 1: Summary of existing hardware-capability-based system designs compared with caplification (our work).

Design	G1	G2	G3
CheriBSD [41, 61]	✓	✗	✓
CheriOS [16]	✗	✗	✗
CHERI-TrEE [56]	✗	✗	✗
CAP-VM [44]	✓	✗	✗
ORC [45]	✗	✗	✗
CODOM [57]	✓	✗	✓
Caplification (our work)	✓	✓	✓



(a) An abstract VMAC system before caplification

(b) An abstract VMAC system after caplification

Figure 2: A VMAC-based system before and after caplification. Solid arrows indicate which software components control which architectural structures. Dotted arrows indicate the pathway of a memory access from PL_0. Shaded areas represent capability-aware software components or structures.

isolated fault *domains*, which are the basic units of capability-based isolation. Domains follow no privilege hierarchy but each has its internal privilege hierarchy mirroring that in a VMAC system, which we refer to as the *internal structure* of the domain. The internal structure allows a capability-oblivious software stack (e.g., an unmodified Linux stack) to run inside a domain. Each domain fully controls its own internal structure and communicates with other domains about their capability configuration only when needed, such as when sharing memory.

In the internal structure of a domain, the highest privilege level (*C* in Figure 1 (d)) is capability-aware while the lower privilege levels are capability-oblivious. When software from a lower privilege level accesses memory, it provides a virtual address which is translated into a physical address in a way identical to VMAC, but the physical address is subsequently mapped to a capability provided by the same domain. Thus, all memory accesses, including ostensibly capability-oblivious ones, are ultimately based on capabilities. Capability-oblivious software, no matter the privilege level, is bounded by the capabilities its domain holds. As a result, capabilities can thus be used for both user-space and kernel-space isolation. Domains may also share memory simply by passing (physical) capabilities, without involving privileged software.

Caplifying memory protection data structures. The core idea behind caplification is to map physical addresses to hardware capabilities through adjustments to an architectural data structure relevant to memory protection. Caplification replaces addresses and permissions in such a data structure with hardware capabilities, which are *unforgeable in software*. This ensures that capabilities are the fundamental isolation mechanism on the system and allows each individual isolated component to manage the data structure itself, instead of relying on a trusted central authority. In contrast, VMAC-based architectures allow privileged software to write arbitrary values to such data structures. For example, the OS kernel can set arbitrary frame numbers and permissions in page table entries.

Caplification illustrated. To illustrate the idea of caplification in more detail, consider a typical VMAC-based architecture with three privilege levels, PL_0 through PL_2, as illustrated in Figure 2 (a). PL_0 is the lowest privilege level and typically runs user application code, while PL_2 is the highest privilege level that usually runs firmware. PL_1 typically runs the OS kernel, which controls the page table. PL_2 has access to a memory protection data structure (MPDS) that all memory accesses from the two lower privilege levels are checked against. Examples of MPDS include the PMP in RISC-V [59, 60] and GDT in x86 [1, 23], and MPU in ARM [2].

Caplifying such an architecture results in the architecture illustrated in Figure 2 (b). The direct effect of caplification is that the MPDS becomes capability-based. Caplification replaces any entry in the MPDS that originally represents a pair of memory region bounds and its access permissions with a capability, which, in addition to also including such information, is *unforgeable*. All creation and manipulation of capabilities follow strict rules enforced by the hardware, which ensures the authority they convey cannot be expanded (i.e., *monotonic* [20, 62]). Consequently, the highest privilege level PL_2 becomes capability-aware and handles capabilities directly. This includes changing capability bounds and permissions, as well as setting the caplified data structure to them. PL_2 has access to instructions that write capabilities into the MPDS, which amounts to sharing those capabilities with the lower privilege levels PL_0 and PL_1, which use them implicitly.

Memory accesses. All memory accesses become capability-based after caplification. Accesses from PL_0 and PL_1 use capabilities implicitly through the caplified MPDS. In contrast, PL_2 software uses capabilities explicitly for memory accesses. We deem it practical to impose a capability-based programming model on PL_2, as PL_2 typically does not run significant amounts of code, and the cost of adapting PL_2 software to use capabilities explicitly for accessing memory is thus much lower than that of adapting PL_0 and PL_1 software, Moreover, making PL_2 use capabilities explicitly has the added benefit of enabling a pure capability environment consisting entirely of PL_2 software, suitable for software migrated or developed anew with a capability-aware toolchain.

Domains with full privilege hierarchies. When all memory accesses become capability-based, either implicitly or explicitly, the whole software stack can be isolated using capabilities into isolated units or *domains*. Systems using caplification are special in that each domain has a full privilege hierarchy inside it, with its own software at each level. This is called the *internal structure* of the domain. The memory accesses each domain can perform are defined by the set of capabilities the domain holds, regardless of the privilege level the domain accesses memory from. Domains are not fixed but can instead be created and destroyed during run-time, and there is no limit to the number of domains. The threat model (Section 4.2) we consider assumes that the domains are mutually-distrusting. The interaction between domains in caplification follows directly from that in capability-based architecture designs. The example in Figure 2 (b) shows two domains, dom_X and dom_Y.

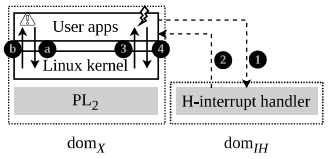


Figure 3: Interrupt and exception handling in CapliFive. Dashed arrows represent domain switching. Shaded components are capability-aware.

4.2 Threat model

Our trust assumptions made are standard and common in prior work on capability-based architectures [62, 68]. By default, the hardware does not assume any trust between domains concerning data confidentiality or integrity. Rather, it expects that software to explicitly pass capabilities with appropriate permissions where it wants to trust software external to its domain. When discussing the security of any domain, we assume that the software running within a given domain trusts other software in its domain—otherwise, it can opt to place untrusted software in a separate domain. Trusted software is assumed to bug-free. A domain need not trust any other domain; it can assume that all the other domains are arbitrarily faulty and malicious. Within each domain, we consider the firmware and the OS kernel as trusted. We assume the hardware implementation is trusted and bug-free. Microarchitectural side-channel attacks are out of scope as they correspond to hardware design and implementation details below the ISA. Following prior work, availability, or safeguarding against denial-of-service attacks, is considered with a single domain assumed trusted.

4.3 Key Technical Challenges

Domain switching. A hardware thread can switch between different domains. Synchronous domain switching happens when a domain voluntarily calls into a different domain. Within each domain, since the privilege hierarchy is associated with its own trust model (e.g., the OS kernel in PL_1 should be able to control all communication from a process in PL_0), we only allow PL_2 to perform synchronous domain switching. Asynchronous domain switching is performed without intervention from a domain itself when an interrupt occurs.

Exception and interrupt handling. Exceptions and interrupts need to be treated carefully among mutually-distrusting domains. This has been a challenge on VMAC-based architectures, which face a dilemma regarding whether privileged software should handle asynchronous exceptions or interrupts. For example, Ahoi attacks show how host-injected exceptions can trick a confidential virtual machine otherwise protected by a TEE to leak secrets or execute attacker-controlled code [46, 47]. With caplification, we strictly distinguish exceptions and interrupts as different types of events. Exceptions are events that directly result from the execution of an instruction in a domain, while hardware interrupts (H-interrupts) are externally generated events that target a hardware thread as a whole. We let each domain handle its own exception directly, without involving other domains on the system. This can be done safely, as caplification has confined resources a domain can access by the capabilities it holds and localized the effects of privileged states to the domain itself. H-interrupts, on the other hand, are in general not specific to any domain and do not reflect the domain state. Handling them safely thus requires switching to an *interrupt handling domain*, which we refer to as dom_{IH}. As a way to delegate interrupt handling to capability-oblivious software in another domain, dom_{IH} can post a virtual interrupt (V-interrupt) to it. The receiving domain handles the V-interrupt in the same way as a capability-oblivious software stack handles an interrupt. In particular, the resumed domain can unilaterally mask the V-interrupt delivery. Such control prevents an untrusted party from injecting asynchronous interrupts at chosen times to lead the domain into an inconsistent state, as demonstrated in several attacks [11, 46, 47].

Figure 3 illustrates examples of interrupt and exception handling in a caplified system. If, during the execution of dom_X, the interrupt controller raises an interrupt to the CPU, the CPU performs a domain switch from dom_X to dom_{IH} (1), which receives and handles the H-interrupt and then posts a corresponding V-interrupt to dom_X when resuming the latter's execution (2). The domain dom_X then handles the V-interrupt within itself as in a VMAC-based system, involving a trap into the kernel interrupt handler (3) followed by a return (4). In another example, when an exception occurs during the execution of dom_Y, it is handled entirely within dom_Y itself, trapping into the kernel exception handler (a) which then returns to the excepted process after handling (b).

Localizing privileged states. Some registers have system-wide impact on software execution. For example, the interrupt enabling bit affects whether the whole system can receive and handle hardware interrupts. We say that those registers are the privileged states. Since the full privilege hierarchy is now confined to domains, which are not allowed to interfere with each other, it is necessary to localize privileged states. The localization follows two strategies, depending on the semantics of the specific state. For states that control code behaviours, we simply make them domain-local, i.e., a domain has its own copy of such states that is only effective within the domain itself. States with system-wide effects are virtualized so that their effects are confined within the domain. Such examples include states related to interrupt handling which are adapted to work with virtualized interrupts. As with the first class of states, we also give those states their domain-local copies.

5 CapliFive: Supporting Caplification on Capstone

We demonstrate CapliFive, a concrete hardware and software co-design of caplification on RISC-V using Capstone ISA [68].

Capstone. Capstone is a recent capability-based architecture design with improved expressiveness of capabilities, which enables revocable capability delegation and exclusive ownership through additional capability types and operations. In particular, *linear capabilities* never overlap with other existing capabilities, and *revocation capabilities* are snapshots of capabilities which can later be used to revoke all capabilities derived from them and at the same time restore the original snapshot capabilities. Such features of Capstone are particularly useful as building blocks for isolation, and we choose it as the base capability design to build on for this reason.

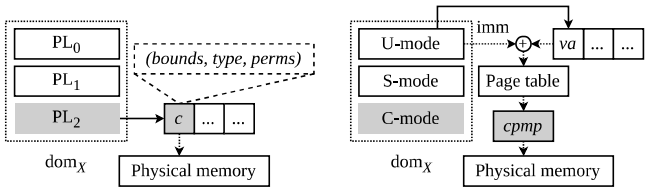


Figure 4: Memory access from different parts of a CapliFive domain internal structure. The shaded areas are capability-aware software components or data structures.

RISC-V. RISC-V [60] is an open ISA design that adopts VMAC. A typical RISC-V system consists of three privilege levels: the U-, S-, and M-modes, from the least to the most privileged. We choose RISC-V for its open-source software and hardware support.

5.1 Instruction Set Architecture

Central in CapliFive is an ISA adapted from Capstone-RISC-V [54]. We have released the specification publicly [53].

Capabilities. CapliFive adopts the 128-bit capability format in Capstone. Each capability records its type, bounds, a cursor address, permissions, and an ID of a node in the *revocation tree*, a hardware-maintained data structure that tracks the capability derivation relationships and validity. Every 128-bit-aligned memory location is tagged with a single tag bit to indicate whether it contains a capability (when the bit is set) or an integer. More details are available in the Capstone work [68].

`cpmps`: caplified PMP. PMP (Physical Memory Protection) is a set of registers (`pmpcfgs` and `pmpaddrs`) in RISC-V [60] that define a list of physical address ranges and their respective access permissions. All those registers are only accessible in M-mode. A RISC-V implementation may provide 0, 16, or 64 PMP entries. For a memory access by S/U-modes, the CPU looks up the physical address in the PMP entry list and enforces the access permissions specified in the first matching entry. We caplify PMP into a list of n registers, $\texttt{cpmp}_0$ – $\texttt{cpmp}_{n-1}$, each containing a capability. Each memory access, including the physical address and access type, is checked against all `cpmps` and is permitted when at least one of them permits it.

C-mode: capability-aware M-mode. The original M-mode in RISC-V becomes capability-aware in CapliFive. For clarity, we refer to M-mode, thus adapted, as *C-mode* (*C* for *Capability*). Most C-mode capability-aware registers and instructions derive from Capstone. C-mode directly manipulates capabilities and uses them explicitly for memory accesses (Figure 4 (a)). Specifically, it can hold capabilities in general-purpose registers `x1` – `x31` and use them in capability instructions. For memory accesses, the original load/store instructions, e.g., `LD` and `SD`, require capabilities as the base address operand in C-mode. C-mode can use the added instructions `LDC` and `STC` to load and store capabilities in memory. In addition, C-mode gains access to CSRs (control and status registers) `cepc`, `ctvec`, and `cscratch`. They replace the original `mepc`, `mtvec`, and `mscratch` to accept capability values. C-mode software can actively perform synchronous domain switches using `CALL` and `RETURN` instructions. The `CALL` instruction calls into other domains. It expects in its operand a *sealed capability* – a capability that represents a domain execution context. A corresponding `RETURN` returns from such calls.

Exposing memory to S/U-modes. C-mode sets up physical memory protection for U-mode and S-mode through `cpmps`. C-mode uses the `CCSRRW` (capability control and status register read and write) instruction to exchange capabilities between general-purpose registers and `cpmps`. Once set, the capabilities in `cpmps` are implicitly used for memory accesses from S/U-modes (Figure 4 (b)).

System reset and compatible mode. Upon system reset, CapliFive does not immediately enter C-mode. Instead, it starts in the *compatible mode* (not to be confused with the modes in privilege levels) whose behaviour is identical to that of a RISC-V system. This allows reusing existing M-mode firmware to initialize the CPU and the I/O devices. After initialization, the M-mode software can enter the *capability mode* through a `CAPENTER` instruction, which immediately replaces M-mode with C-mode. `CAPENTER` also creates the genesis capabilities, from which all later capabilities that exist on the system are derived. This operation is one-way – the system cannot switch back to the compatible mode without a full reset.

Interrupt and exception handling. Following the design of Capstone, CapliFive provides a register `cih` that stores the sealed capability to dom_{IH}, the H-interrupt handler domain. An H-interrupt amounts to an asynchronous domain switching to dom_{IH}. CapliFive allows the H-interrupt handler domain to post V-interrupts by extending the `RETURN` instruction with an additional operand for specifying the bit map of the V-interrupts to post. The in-domain exception and V-interrupt handling mechanism mirrors the exception and interrupt handling mechanism in RISC-V [60]. The only differences are the changes for localizing privileged states. To S-mode and U-mode software, such adjustments are transparent.

Domain switching. For both types of domain switching, CapliFive performs the necessary save and restore operations to protect domain data, including the CSRs that are local to each domain. Missing out a CSR can lead to serious security and functionality consequences. For example, failing to save or restore the C-mode trap vector `ctvec` prevents the involved domains from delivering their exceptions and V-interrupts to the desired locations.

Synchronous and asynchronous domain switching requires saving and restoring different parts of the context. Since domains control when synchronous domain switching takes place, they can perform the necessary context save/restore by themselves in software. The exceptions are some CSRs that can affect C-mode execution right after a synchronous switching, such as `mip` and `mideleg`. The hardware needs to swap those CSRs atomically. For asynchronous domain switching, as the original domain cannot control when it takes place, the hardware needs to perform all the necessary context saves and restores for the domain.

5.2 Software Stack

The various software components in a CapliFive system interact with one another, both across domains and within the same domain, through well-defined interfaces, as briefly described below.

Supervisor Binary Interface (SBI). CapliFive Supervisor Binary Interface (SBI) is the interface between the capability-aware C-mode software and the legacy software in the S/U modes. It extends the

Table 2: SBI exposed to capability-oblivious software.

Function	Semantics
`dom-create`	creates a new domain
`dom-call`	synchronously invokes a domain
`dom-schedule`	submits a domain to the scheduler as a new thread
`region-create`	creates a new memory region
`region-share`	shares a memory region with a domain
`region-query`	queries information about a memory region
`region-count`	counts the memory regions held by the domain

Table 3: Sharing modes for cross-domain arguments.

Sharing mode	Capability type	Post-call revocation
default	non-linear	Yes
borrowed	linear	Yes
shared	non-linear	No
transferred	linear	No

RISC-V SBI [42] with support for capability-based isolation. Table 2 summarizes the functions available through CapliFive SBI.

Domain Binary Interface (DBI). The DBI defines the interface between C-modes of domains. This includes, for example, functions to request an entry into S-mode and to share a memory region in the form of a capability.

Interrupt Handler Interface (IHI). Each domain (except for the H-interrupt handler domain itself) can interact with the H-interrupt handler domain through a synchronous call. The interrupt handler interface defines the format of such communications.

Capability-based inter-domain communication. CapliFive allows applications to take advantage of the expressiveness of capabilities for inter-domain communication. This is achieved by using the appropriate capability type. Conceptually, each argument has a pair of attributes. The first attribute is a *sharing mode* which captures the desired change in ownership and exclusiveness of the argument. It translates into the capability type and whether revocation should be performed after the call returns, as shown in Table 3. The second attribute is the *sharing permission*, which dictates the restrictions on the access permissions the callee has to the argument. The available values for the sharing permission include *in* (read-only), *out* (write-only), *inout* (read-write), *exe* (execute-only), and *full* (read-write-execute).

Threading. CapliFive supports multithreading and multiprocessing. It multiplexes all software threads on a single hardware thread (i.e., a *hart* in the RISC-V terminology). By default, the first domain to run after system resets (dom_0) is special in this respect as it plays a direct role in scheduling threads. Note that this does not violate our threat model as dom_0 is still unable to break data confidentiality or integrity. Specifically, dom_0 has the permission to configure the hardware timer interrupts through the memory-mapped CLINT timer registers (`mtimecmp` and `mtime` as specified in the RISC-V ISA [60]). The domain dom_0 can use the interrupt handler interface to request the H-interrupt handler domain to switch to a different thread with a specified domain dom_X. During the execution of a thread other than the one dom_0 runs on, whenever an H-interrupt occurs (including timer interrupts), the H-interrupt handler domain takes control and resumes the execution of dom_0. The H-interrupt handler domain can choose to delegate the H-interrupt to dom_0 as a V-interrupt. The in-domain interrupt handler in dom_0 can henceforth perform scheduling of userspace threads along with the domains they invoke, but without being aware of CapliFive domains. When the userspace thread that invoked the other thread is scheduled to run again, it can choose to resume the execution of the thread that dom_X belongs to.

6 Implementation

Our prototype implementation of CapliFive consists of two parts: the hardware architecture and the software stack. We present two implementations of the hardware architecture: CapliFive-QEMU and CapliFive-RTL. CapliFive-QEMU extends QEMU [5] with CapliFive features. Its goal is to explore the expressiveness of CapliFive. Since it only emulates the hardware behaviour at the ISA level, it is unsuitable for cycle-level performance evaluation. CapliFive-RTL instead focuses on the register-transfer level (RTL), which models the cycle-level behaviours of the hardware. CapliFive-RTL modifies CVA6 [30], an open-source RISC-V core. We have released all source code publicly [39, 40, 67].

6.1 CapliFive-QEMU

The CapliFive-QEMU implementation is straightforward. We make use of helper functions to implement most CapliFive operations, instead of implementing them as inline code in the TCG IR.

6.2 Software Stack

We created a compiler, CapliFive-CC, for developing C-mode pure-capability software. We used it mainly to support the prototype software implementation for CapliFive. CapliFive-CC takes as input C code with CapliFive-specific annotations and intrinsics. It implements all pointers as capabilities, and allows specifying the capability type through annotations. CapliFive-CC also supports in-domain and cross-domain abstract binary interfaces (ABIs) and recognizes intrinsics that expose CapliFive-specific operations.

We use CapliFive-CC to develop the C-mode software, CapliFive-SBI. It provides the extended supervisor binary interface to expose CapliFive-specific isolation operations to lower privilege levels (Section 5.2). We modified OpenSBI [43] to perform a `CAPENTER` in the last stage of the booting process. CapliFive-CC takes over control, performs its own initialization, including setting up C-mode-specific CSRs and the H-interrupt handler domain, before handing over control to S-mode. S-mode runs an unmodified Linux kernel with a kernel module (`modcaplifive`), which exposes a U-mode interface through IOCTLs. U-mode provides `libcaplifive` which includes wrappers of the IOCTL interface.

6.3 CapliFive-RTL

We implement CapliFive-RTL as a basis for our performance evaluation in Section 8. For simplicity, our implementations leave out the handling of linearity, i.e., they do not prevent duplicating linear capabilities. Nor do our implementations include reference counting to recycle allocated nodes.

Overview. We build CapliFive-RTL on CVA6 [70], a 6-stage in-order RISC-V CPU in SystemVerilog. Figure 5 presents an overview of the CapliFive-RTL design, with added and modified components highlighted. We consider a setup with a single CPU core with an ISA based on RV64IMAC [54]. Most changes to CVA6 concentrate on the

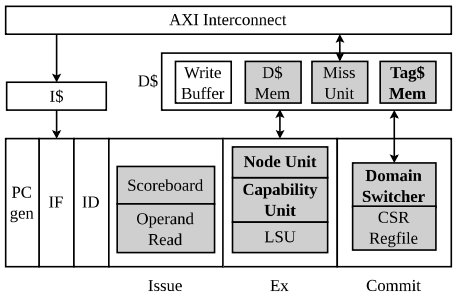


Figure 5: A simplified overview of CapliFive-RTL. The shaded components are added (bold) or modified (regular) over the original CVA6. Arrows represent data flows.

issue, execution, and commit stages, where maintenance, manipulation, and enforcement of capabilities take place. CapliFive-RTL also extends the data cache (D$) to facilitate tagged memory accesses.

Extension to data paths. We extend all general-purpose registers (GPRs, *Operand read* in Figure 5) and some CSRs (*CSR Regfile*) to optionally hold capabilities, with tag bits indicating whether they hold capabilities or integers.

Extension to data cache. CapliFive-RTL extends the data cache to maintain tags associated with 16-byte-aligned memory locations. Each cache line in the data cache memory is extended to cache tags. For each cache miss, the miss unit requests the cache line from the next level of the memory subsystem, including both data and tags. The tags are maintained in a separate physical memory region not directly accessible to software. CapliFive-RTL caches them in a blocking write-back write-allocate *tag cache*.

Node unit. The node unit maintains the revocation tree (Section 5.1). It accepts three types of requests from other components: *queries, allocations,* and *mutations*. Queries look up the validity of specified nodes. Most queries are asynchronous and only synchronized in the commit stage. Allocations speculatively allocate new revocation nodes. Some instructions need IDs of newly allocated revocation nodes. Though those instructions may require further complex changes to the revocation tree, the node IDs can be decided cheaply in advance. Mutations make actual (architectural) changes to the revocation tree. The node unit maintains pending mutations in a mutation buffer and processes them in a state machine using spare memory bandwidth. Similarly to tags, revocation nodes reside in a physical memory region that software cannot directly access. A dedicated writeback write-allocate *node cache* caches the nodes.

Capability unit. We add a fixed-latency capability unit to perform simple manipulation on capabilities, e.g., shrinking and tightening. The capability unit also performs compression and decompression of capabilities stored in registers. For operations that require *valid* capability operands, the capability unit sends corresponding node queries to the node unit. The commit stage waits until the associated query is resolved before retiring an instruction or generating an exception if the query results do not meet the requirements (i.e., the capability is invalid). For instructions that might change the validity of any capability, the commit stage sends the mutation requests to the node unit after retiring the instructions.

Load/store unit. The extended load/store unit (LSU) supports loading and storing capabilities as well as using capabilities for memory access. In the former case, the off-band tag bit also needs to be loaded or stored to indicate whether the data is a capability. This is handled mostly transparently by a modified data cache, as discussed earlier. In the latter, the LSU performs bounds and permissions checks, and, similarly to the capability unit, sends necessary capability validity queries to the node unit.

Domain switcher. The domain switcher performs both synchronous and asynchronous domain switching. The commit stage activates the domain switcher after retiring an instruction that requires domain switching or is tagged with a hardware interrupt. Domain switching involves swapping a number of registers with memory contents. We implement this as a sequence of load and store requests to the LSU. The other pipeline stages, including existing load requests and uncommitted store requests in the LSU, are flushed during domain switching. The domain switcher redirects the frontend to the new PC capability cursor at the end of domain switching.

7 Case Studies

Our prototype systems runs Linux as a single domain on RISC-V. In addition, we present two case studies to showcase compatibility and interoperability between capability-aware and capability-oblivious software. They also highlight how capability-based isolation can directly benefit capability-oblivious software.

Kernel-space isolation: Linux driver (`driver`). CapliFive can be used to compartmentalize kernel components. Our case study builds on a bio-based null block device driver [37], a Linux kernel module which simulates a block device and interacts with the other parts of the kernel through the bio interface. We split the driver into two components running in separate domains, dom_A and dom_B.

The main part of the driver runs in S-mode of dom_A alongside the Linux kernel, while some isolated functions are moved to S-mode of dom_B. The component in dom_A is responsible for interacting with the kernel, and sharing data with the component in dom_B. Depending on the sharing semantics, CapliFive uses different types of capabilities. The data structure representing the null block device (`struct nullb_device`) is shared synchronously between two components through a linear capability (`nullb_dev_buf`). Linear capabilities ensure that the physical memory is accessible to only one component at any given time, providing temporal isolation [69]. Arguments and return values are also passed synchronously between two components through `arg_buf` and `rv_buf` capabilities. Additional data specific to a function call is shared asynchronously through a non-linear capability (`metadata_buf`). This case study highlights *benefits of fine-grained memory isolation in the kernel* possible with CapliFive, a salient advantage of capability hardware.

User-space isolation: web server (`webserver`). This case study shows that *CapliFive can be used for user-space isolation and allows capability-oblivious user programs to interact with capability-aware software.* Consider a web server setup that consists of three components isolated in three domains dom_A, dom_B, and dom_C. The frontend in dom_A is a U-mode process. It receives requests from user and forwards the request to dom_B directly without kernel intervention through shared capabilities. The middleware in dom_B handles these requests and generates responses accordingly. It can

then execute external scripts for dynamic content generation if needed, which run in the backend in C-mode of dom_C. Communications between the web server and external scripts are only between dom_B and dom_C through shared capabilities.

We assume that the server script in dom_C is potentially buggy or backdoored. The attacker may attempt to use the server script to leak or tamper with dom_B and dom_A. They may also target specific connections, e.g., to leak private data exchanged in one connection to another. To provide temporal isolation, dom_B creates dom_C without provisioning writable capabilities. For every connection when dom_B needs to invoke dom_C for request handling, it passes writable capabilities to zeroed memory regions or uninitialized capabilities. Requests and external scripts are handed from the user to the component in dom_A, which forwards them directly to dom_B through linear capabilities (`socket_buf` and `script_buf`). When handling the request, dom_B may request files from dom_A through a linear capability (`file_buf`). Domains dom_B and dom_C can write the response to a linear capability (`response_buf`), which will be de-linearized once the response is complete. Domain dom_A can then read the content of the response after a capability revocation, which it in turn forwards to the user. The three domains share all metadata asynchronously through a non-linear capability (`metadata_buf`).

8 Performance Evaluation

We evaluate the performance of our prototype CapliFive implementation (Section 6), focusing on the following questions:
EQ1. What is the overhead of individual operations in CapliFive?
EQ2. How does the overall performance of applications in CapliFive compare with that of the VMAC baseline?
Comparison baselines. We compare against two baselines:

PMP. We compare the caplified PMPs in CapliFive with the original physical memory protection (PMP) design in RISC-V, a feature on RISC-V [60] to enforce memory isolation. This baseline design follows Keystone [27] and relies on trusted M-mode software to configure PMPs to provide isolation. Application software traps into M-mode for domain switching.

No-isolation. We also compare with running all software components on RISC-V with no isolation in M-mode.

8.1 Microbenchmarks

We run a collection of microbenchmarks on CapliFive-RTL simulated with Verilator [51], a cycle-accurate RTL simulator. We use a single-core setup with 16 `cpmp` entries, a 2KB 4-way set-associative tag cache with 8-byte cache lines, a 4KB 4-way set-associative node cache with 16-byte cache lines. Both baselines are evaluated on unmodified CVA6 with 16 PMP entries, also simulated with Verilator. We measure the costs of the different operations below.
Sequential loads/stores. CapliFive loads or stores data at 8-byte granularity using a capability sequentially. The PMP baseline performs sequential loads or stores with `LD` in S-mode on virtual addresses, and the No-isolation baseline does the same in M-mode on physical addresses. We also measure the cost of loading or storing pointers. Those pointers correspond to capabilities in CapliFive and 8-byte integers in both baselines. We report the cycles needed for 128 such operations after the cache is warm, so no TLB misses or page faults occur.
Traps and returns. For both CapliFive and PMP, we measure the cost of vertical traps, including the context saving time for both U-to-S and S-to-M/C traps. We also measure the cost of returning from those traps. For the No-isolation baseline, we report the cost of plain function calls and returns.
Synchronous domain switching. We consider switching between domains in various privilege levels. CapliFive supports isolation in C-mode, and a domain switch from C-mode of one domain to that of another constitutes a `CALL` instruction. Depending on the application-specific security requirements, the software may need to save and scrub (i.e., zero) part or all of the GPRs. We also measure the worst-case cost of such extra operations. Domain switching from or to a lower privilege level involves the cost of switching between privilege levels (traps and returns). PMP involves first trapping into M-mode, followed by an adjustment to the PMP entry permissions, and a return to the target privilege level with `MRET`. We account for the cost of traps and returns from trap separately from the domain switching operation in M/C-mode itself for both CapliFive and PMP. For No-isolation, we evaluate the cost of an empty function call in M-mode.
Interrupt delivery. We count the cycles between when an interrupt is taken and when the interrupt handler can start handling the interrupt. We consider the scenario where the interrupt handler is a different domain from the currently executing one. For CapliFive, this is an asynchronous domain switch into the H-interrupt handler domain. PMP first delivers the interrupt to the M-mode security monitor, which then switches to the H-interrupt handler domain. The process is thus the same as that of a synchronous domain switch, except that the trap into M-mode is now asynchronous due to an interrupt. For No-isolation, we consider a trap from S-mode into M-mode due to an interrupt, including the cost of saving the context based on the Linux implementation.
Inter-domain communication. We evaluate two types of inter-domain communication. Sharing data between CapliFive domains involves creating a revocation capability, an adjustment to the capability permissions, followed by a revocation after the use by other domains is complete. Transferring data involves simply passing a capability. PMP involves copying the data in and out through a shared buffer for either data sharing and transfer. No-isolation does not require any operation for data sharing and transfer. We evaluate the cost of sharing or transferring 1 KB of data.
Results. Table 4 summarizes the results. In both CapliFive and the baselines, each 8-byte load or store in the sequential load/store benchmarks takes around 4 cycles. The gap in pointer loads/stores between CapliFive and the baselines is accounted for by its doubled size of the pointer (capabilities are 16 bytes long). Domain switching in both CapliFive and PMP costs more compared to a function call (No-isolation). The cost comes mostly from swapping of contexts, including some of the S-mode CSRs for PMP and C-mode CSRs for CapliFive. Noteworthy is CapliFive's support for isolation and domain switching in C-mode. Pure C-mode domains can decide (potentially with the help of a compiler) which part of the context to save and scrub, depending on which part is live and which part it considers as sensitive. The resulting cost can be as low as 88 cycles, which, though still 8.4x higher than a plain function call (18 cycles), is 5.4x lower than S-mode domain switching in PMP (742 cycles when considering the cost traps and returning from traps).

Table 4: Microbenchmark results (measured in cycles).

Microbenchmark	CapliFive	PMP	No-isolation
Sequential loads	511	511	511
Sequential stores	471	445	440
Sequential ptr. loads	766	511	511
Sequential ptr. stores	846	445	440
U-to-S trap	99	93	18
S-to-M/C trap	113	113	18
S-to-U return	67	67	14
M/C-to-S return	158	158	14
Sync. dom. switch (C)	88–308	N/A	18
Sync. dom. switch (S/U)	614	471	18
Interrupt delivery	951	731	119
Data sharing (1KB)	25	1042	0
Data transfer (1KB)	4	1042	0

Data sharing and transfer incur a small fixed overhead in capability manipulation in CapliFive, but have a cost that increases with the amount of data involved in PMP.

The cost of most operations in CapliFive is similar to that in PMP, except for data sharing where CapliFive significantly reduces the cost and interrupt delivery where CapliFive is significantly more costly. Both have higher cost than No-isolation. CapliFive supports the cheaper C/M-mode isolation while PMP does not.

8.2 Macrobenchmarks

We profile our two case studies presented in Section 7, which run on CapliFive-QEMU to obtain the occurrences of individual operations and events during execution. We approximate the overall performance by combining them with the microbenchmark results. We focus on the cost of interrupt delivery, cross-domain data sharing, and context switches (including the associated traps and returns from traps). Based on the micro-benchmarking results, these aspects are where we expect to see most performance differences.

Results. Figure 6 summarizes the results. The raw profiling results are given in Appendix A. The overhead of data sharing is 86% to 98% lower in CapliFive compared to PMP. The reduction is especially evident in `webserver`, which shares or transfers a larger amount of data in fewer passes compared to the other two case studies. The cost of context switching in CapliFive is higher or lower than in PMP, depending on the application scenario. As PMP does not support C/M-mode isolation, it needs to resort to domain switches from S-mode, which is more costly. For `webserver`, the context switching overhead is almost the same between CapliFive and PMP (2% higher in CapliFive). In contrast, for `driver`, which only performs S-to-S domain switches, the domain switch cost in CapliFive is 19% higher than in PMP. Interrupt delivery in CapliFive is 30% more costly. However, this performance impact is minimal in our case studies as interrupts are rare. Overall, the total of the three types of overhead in CapliFive is reduced by 9% in `webserver` and increased by 12% in `driver`, as compared to PMP.

Compared to PMP, CapliFive has comparable performance and reduces the overhead of cross-domain data sharing (by up to 98%).

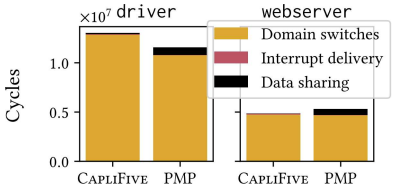


Figure 6: Summary of macrobenchmark results.

9 Related Work

We have discussed the most relevant prior work in Section 3. We discuss the broader context motivating our work in this section.

Capability-based security. Early work on capability-based systems [14, 21, 24, 28, 32] before the 1980s had the ambitious goal of closing the semantic gap between high-level programming languages and hardware architectures. Recent work on hardware capabilities has focused on their potential as a memory safety mechanism, [15, 18, 25, 26, 32, 50, 61, 63]. Beyond hardware capabilities, prior work has also explored software-based capabilities as an isolation primitive [4, 6, 12, 20, 64]. Such designs require a trusted software monitor, e.g., a microkernel, to interpret capabilities and translate them into hardware isolation primitives such as VMAC.

VMAC security extensions. Outside capability-based designs, prior work has devised extensions to VMAC to accommodate more isolation needs without breaking compatibility with existing software. Designs based on memory protection keys (MPKs), memory domains, or page groups [8, 19, 48, 49, 52, 55, 66] associate permissions with whole virtual page groups. A common trusted software monitor either directly controls such permissions or monitors how the isolated components adjust the permissions. Extended page table (EPT) designs as found in hardware-assisted virtualization extensions (e.g., Intel VMX [23], AMD SVM [1], and RISC-V H extension [60]) extends the numbers of privilege levels and address translations by one, which leaves application needs such as nested isolation of the picture [38]. Isolation mechanisms based on the EPT, with or without VMFUNC, rely on a trusted hypervisor and lack support for inter-domain sharing [19, 22, 29, 31, 33, 34, 35]. Trusted execution environment (TEE) extensions either rely on a trusted software monitor to partition the physical memory into multiple VMAC-based compartments [1, 2, 3], or enforce ad-hoc restrictions on virtual memory [10, 23].

10 Conclusion

This paper presents CapliFive, a novel system design that uses the strategy of caplification to maintain compatibility and interoperability between hardware capabilities and capability-oblivious software stacks without sacrificing the unique advantages of capability-based isolation. CapliFive is capable of bringing the benefits of capability-based isolation to capability-oblivious software and providing a consistent isolation primitive across all levels of existing capability-oblivious software stacks.

Acknowledgements

We thank NUS KISP Lab members for their feedback. This research is supported by a Singapore Ministry of Education (MOE) Tier 2 grant MOE-T2EP20124-0007.

References

[1] [n. d.] AMD64 Architecture Programmer's Manual, Volumes 1-5, 40332, 24592, 24593, 24594, 26568, 26569. ().

[2] 2023. Arm Architecture Reference Manual for A-profile architecture. (2023).

[3] [n. d.] Arm CCA Security Model. ().

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, Big Sky Montana USA, (Oct. 2009), 29–44. ISBN: 978-1-60558-752-3. doi:10.1145/1629575.1629579.

[5] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. USENIX Association, Anaheim, CA, (Apr. 2005).

[6] Borna Blazevic, Michael Peter, Mohammad Hamad, and Sebastian Steinhorst. [n. d.] TEEVseL4: Trusted Execution Environment for Virtualized seL4-based Systems.

[7] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. 1994. Hardware support for fast capability-based addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (Asplos Vi). Association for Computing Machinery, New York, NY, USA, 319–327. ISBN: 0-89791-660-3. doi:10.1145/195473.195579.

[8] Yuan Chen, Jiaqi Li, Guorui Xu, Yajin Zhou, Zhi Wang, Cong Wang, and Kui Ren. 2020. Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX. (Oct. 2020). Retrieved July 1, 2023 from arXiv: 2010.12400 [cs].

[9] Tyler Close. 2009. ACLs don't. Pre-published.

[10] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 86.

[11] Jinhua Cui, Jason Zhijingcheng Yu, Shweta Shinde, Prateek Saxena, and Zhiping Cai. 2021. SmashEx: Smashing SGX Enclaves Using Exceptions. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Virtual Event Republic of Korea, (Nov. 2021), 779–793. ISBN: 978-1-4503-8454-4. doi:10.1145/3460120.3484821.

[12] Everton de Matos and Markku Ahvenjärvi. 2022. seL4 Microkernel for virtualization use-cases: Potential directions towards a standard VMM. *Electronics*, 11, 24, (Dec. 2022), 4201. arXiv: 2210.04328 [cs]. doi:10.3390/electronics11244201.

[13] Drew Dean and Alan J. Hu. 2004. Fixing races for fun and profit: How to use access(2). In *13th USENIX Security Symposium (USENIX Security 04)*. USENIX Association, San Diego, CA, (Aug. 2004).

[14] Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9, 3, (Mar. 1966), 143–155. doi:10.1145/365230.365252.

[15] Joe Devietti, Colin Blundell, Milo M K Martin, and Steve Zdancewic. [n. d.] HardBound: Architectural Support for Spatial Safety of the C Programming Language.

[16] Lawrence G Esswood. [n. d.] CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor.

[17] Marco Fillo, Stephen W Keckler, William J Dally, Nicholas P Carter, Andrew Chang, Yevgeny Gurevich, and Whay S Lee. [n. d.] The M Machine Multicomputer.

[18] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021. Efficient and provable local capability revocation using uninitialized capabilities. *Proceedings of the ACM on Programming Languages*, 5, POPL, (Jan. 2021), 1–30. doi:10.1145/3434287.

[19] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, (July 2019), 489–504. ISBN: 978-1-939133-03-8.

[20] Gernot Heiser. 2020. The seL4 Microkernel – An Introduction, (June 2020).

[21] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. 1981. IBM system/38 support for capability-based addressing. In *Proceedings of the 8th Annual Symposium on Computer Architecture* (ISCA '81). IEEE Computer Society Press, Washington, DC, USA, 341–348.

[22] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. 2018. EPTI: Efficient defence against meltdown attack for unpatched VMs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, (July 2018), 255–266. ISBN: ISBN 978-1-939133-01-4.

[23] 2023. Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. (2023).

[24] 1981. *Introduction to the iAPX 432 Architecture*.

[25] Dejice Jacob and Jeremy Singer. 2022. Capability Boehm: challenges and opportunities for garbage collection with capability hardware. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, Virtual Switzerland, (Feb. 2022), 81–87. ISBN: 978-1-4503-9251-8. doi:10.1145/3516807.3516823.

[26] Yonghae Kim, Anurag Kar, Jaewon Lee, Jaekyu Lee, and Hyesoon Kim. 2023. RV-CURE: A RISC-V Capability Architecture for Full Memory Safety. doi:10.48550/ARXIV.2308.02945.

[27] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*. ACM, Heraklion Greece, (Apr. 2020), 1–16. ISBN: 978-1-4503-6882-7. doi:10.1145/3342195.3387532.

[28] Henry M. Levy. 1984. *Capability-Based Computer Systems*. Digital Press, Bedford, Mass. ISBN: 978-0-932376-22-0.

[29] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, Denver Colorado USA, (Oct. 2015), 1607–1619. ISBN: 978-1-4503-3832-5. doi:10.1145/2810103.2813690.

[30] Valentin Martinoli, Yannick Teglia, Abdellah Bouagoun, and Régis Leveugle. 2022. CVA6's Data cache: Structure and Behavior. (July 2022). Retrieved Apr. 4, 2024 from arXiv: 2202.03749 [cs].

[31] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, Dresden Germany, (Mar. 2019), 1–15. ISBN: 978-1-4503-6281-8. doi:10.1145/3302424.3303946.

[32] Mark S Miller, Ka-Ping Yee, and Jonathan Shapiro. [n. d.] Capability Myths Demolished.

[33] Vikram Narayanan and Anton Burtsev. 2023. The Opportunities and Limitations of Extended Page Table Switching for Fine-Grained Isolation. *IEEE Security & Privacy*, 21, 3, (May 2023), 16–26. doi:10.1109/MSEC.2023.3251385.

[34] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2020. Lightweight kernel isolation with virtualization and VM functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, Lausanne Switzerland, (Mar. 2020), 157–171. ISBN: 978-1-4503-7554-2. doi:10.1145/3381052.3381328.

[35] Vikram Narayanan et al. 2019. LXDs: Towards Isolation of Kernel Subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, (July 2019), 269–284. ISBN: 978-1-939133-03-8.

[36] Robert M Norton. 2016. Hardware Support for Compartmentalisation. Tech. rep. (May 2016).

[37] [n. d.] Null block device driver. https://docs.kernel.org/block/null_blk.html. (). Retrieved Jan. 24, 2024 from.

[38] Wing-Chi Poon and Aloysius K Mok. [n. d.] Bounding the Running Time of Interrupt and Exception Forwarding in Recursive Virtualization for the x86 Architecture.

[39] [SW], Project-Starch/Caplifive-Buildroot Feb. 6, 2025. Project STARCH. URL: https://github.com/project-starch/caplifive-buildrootRetrieved May 10, 2025 from.

[40] [SW], Project-Starch/Caplifive-Qemu Jan. 21, 2025. Project STARCH. URL: https://github.com/project-starch/caplifive-qemuRetrieved May 10, 2025 from.

[41] Alexander Richardson. 2019. Secure Linking in the CheriBSD Operating System. (Jan. 2019).

[42] 2023. RISC-V Supervisor Binary Interface Specification.

[43] 2024. Riscv-software-src/opensbi. RISC-V Software. (Apr. 2024). Retrieved Apr. 30, 2024 from.

[44] Vasily A Sartakov, Lluís Vilanova, David Eyers, Takahiro Shinagawa, and Peter Pietzuch. [n. d.] CAP-VMs: Capability-Based Isolation and Sharing in the Cloud.

[45] Vasily A. Sartakov, Lluís Vilanova, Munir Geden, David Eyers, Takahiro Shinagawa, and Peter Pietzuch. 2023. ORC: Increasing Cloud Memory Density via Object Reuse with Capabilities. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, (July 2023), 573–587. ISBN: 978-1-939133-34-2.

[46] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. 2024. WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP. (Apr. 2024). Retrieved Apr. 30, 2024 from arXiv: 2404.03526 [cs].

[47] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. 2024. Heckler: Breaking Confidential VMs with Malicious Interrupts. (Apr. 2024). Retrieved Apr. 30, 2024 from arXiv: 2404.03387 [cs].

[48] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, (Aug. 2022), 936–952. ISBN: 978-1-939133-31-1.

[49] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, (Aug. 2020), 1677–1694. ISBN: 978-1-939133-17-5.

[50] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proceedings of the ACM on Programming Languages*, 3, POPL, (Jan. 2019), 1–28. doi:10.1145/3290332.

[51] [SW] Wilson Snyder, Paul Wasson, Duane Galbi, and et al, Verilator. URL: https://github.com/verilator/verilator.

[52] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-unikernel isolation with Intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, Lausanne Switzerland, (Mar. 2020), 143–156. ISBN: 978-1-4503-7554-2. doi:10.1145/3381052.3381326.

[53] [n. d.] The Capstone-RISC-V Instruction Set Reference. Retrieved May 10, 2025 from https://capstone.kisp-lab.org/specs-caplifive/.

[54] [n. d.] The Capstone-RISC-V Instruction Set Reference. https://capstone.kisp-lab.org/specs/. (). Retrieved Jan. 22, 2024 from.

[55] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. Nadia Heninger and Patrick Traynor, (Eds.) USENIX Association, 1221–1238.

[56] Thomas Van Strydonck, Job Noorman, Jennifer Jackson, Leonardo Alves Dias, Robin Vanderstraeten, David Oswald, Frank Piessens, and Dominique Devriese. 2023. CHERI-TrEE: Flexible enclaves on capability machines. In *Proceedings of the 8th IEEE European Symposium on Security and Privacy*.

[57] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: protecting software with code-centric memory domains. *ACM SIGARCH Computer Architecture News*, 42, 3, (Oct. 2014), 469–480. doi:10.1145/2678373.2665741.

[58] Lluís Vilanova, Marc Jordà, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2017. Direct Inter-Process Communication (dIPC): Repurposing the CODOMs Architecture to Accelerate IPC. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, Belgrade Serbia, (Apr. 2017), 16–31. ISBN: 978-1-4503-4938-3. doi:10.1145/3064176.3064197.

[59] Andrew Waterman, Krste Asanovic, and CS Division. [n. d.] The RISC-V Instruction Set Manual (Volume I: Unprivileged ISA). ().

[60] Andrew Waterman, Krste Asanovic, John Hauser, and CS Division. [n. d.] The RISC-V Instruction Set Manual (Volume II: Privileged Architecture). ().

[61] Robert N. M. Watson et al. [n. d.] Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9). Tech. rep. Computer Laboratory, University of Cambridge, 523 pages. doi:10.48456/TR-987.

[62] Robert N.M. Watson et al. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. IEEE, San Jose, CA, (May 2015), 20–37. ISBN: 978-1-4673-6949-7. doi:10.1109/SP.2015.9.

[63] Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, San Jose California, (Oct. 2002), 304–316. ISBN: 978-1-58113-574-9. doi:10.1145/605397.605429.

[64] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. 1974. HYDRA: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17, 6, (June 1974), 337–345. doi:10.1145/355616.364017.

[65] Hongyan Xia et al. 2018. CheriRTOS: A Capability Model for Embedded Devices. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, Orlando, FL, USA, (Oct. 2018), 92–99. ISBN: 978-1-5386-8477-1. doi:10.1109/ICCD.2018.00023.

[66] Yuanchao Xu, ChenCheng Ye, Yan Solihin, and Xipeng Shen. 2020. Hardware-Based Domain Virtualization for Intra-Process Isolation of Persistent Memory Objects. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, (May 2020), 680–692. ISBN: 978-1-72814-661-4. doi:10.1109/ISCA45697.2020.00062.

[67] [SW] Jason Yu, Jasonyu1996/Capstone-c Dec. 2, 2024. URL: https://github.com/jasonyu1996/capstone-cRetrieved May 10, 2025 from.

[68] Jason Zhijingcheng Yu, Conrad Watt, Aditya Badole, Trevor E. Carlson, and Prateek Saxena. 2023. Capstone: A Capability-based Foundation for Trustless Secure Memory Access. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, (Aug. 2023), 787–804. ISBN: 978-1-939133-37-3.

[69] Jason Zhijingcheng Yu, Conrad Watt, Aditya Badole, Trevor E. Carlson, and Prateek Saxena. 2023. Capstone: A Capability-based Foundation for Trustless Secure Memory Access (Extended Version). doi:10.48550/ARXIV.2302.13863.

[70] F. Zaruba and L. Benini. 2019. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-GHz 64-Bit RISC-V core in 22-nm FDSOI technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27, 11, (Nov. 2019), 2629–2640. doi:10.1109/TVLSI.2019.2926114.

A Profiling Results

Table 5 summarizes the profiling results.

Table 5: Summary of profiling results. Each data sharing or transfer result consists of the number of occurrences and the total amount of data shared or transferred.

	`driver`	`webserver`
Domain switching (U)	0	2340
Domain switching (S)	14520	2142
Domain switching (C)	0	1292
Interrupt delivery	61	75
Memory access fault	0	13
Asynchronous sharing	7260, 58080	6406, 52972
Sync. immutable sharing	7260, 745360	630, 245134
Sync. immutable transfer	0, 0	430, 87590
Sync. mutable transfer	0, 0	430, 71318