CODOMs: Protecting Software with Code-centric Memory Domains

Lluís Vilanova[*][†][§] Muli Ben-Yehuda[§] Nacho Navarro[*] Yoav Etsion[†][§] Mateo Valero[*]

[*]Barcelona Supercomputing Center (BSC) &
Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain
{vilanova,nacho,mateo}@ac.upc.edu

[†]Electrical Engineering [§]Computer Science
Technion — Israel Institute of Technology
Haifa, Israel
yetsion@tce.technion.ac.il

Abstract

Today's complex software systems are neither secure nor reliable. The rudimentary software protection primitives provided by current hardware forces systems to run many distrusting software components (e.g., procedures, libraries, plugins, modules) in the same protection domain, or otherwise suffer degraded performance from address space switches.

We present CODOMs (COde-centric memory DOMains), a novel architecture that can provide finer-grained isolation between software components with effectively zero run-time overhead, all at a fraction of the complexity of other approaches. An implementation of CODOMs in a cycle-accurate full-system x86 simulator demonstrates that with the right hardware support, finer-grained protection and run-time performance can peacefully coexist.

1. Introduction

The security and reliability of computing systems are ever growing concerns in today's networked computing world. Complex software systems contain a multitude of mutually distrustful or unreliable software components, which can span multiple granularities and purposes: individual functions, compilation units, code libraries, application plugins, or device drivers. Failing to properly isolate these components can have severe effects, malicious and negligent alike, such as privilege escalation, information leakage, denial of service, as well as data corruption caused by buffer overflow bugs. System security and resiliency thus require that individual software components be isolated in separate *domains*.

Existing architectures, however, lack efficient support for isolation at the software component level. Instead, they only support a few coarse-grained protection mechanisms, which are translated into two *Operating System* (OS) concepts: isolating user processes in separate address spaces, and running the OS kernel in a privileged processor mode. Such mechanisms, however, impose non-negligible runtime overheads and can only be managed by the OS. This makes them unsuitable for providing fine-grained isolation between software components that share a single address space.

In this paper we present the *COde-centric memory DOMains* architecture (CODOMs), which provides efficient support for protecting multiple, interacting software components that share an address space. CODOMs is based on the observation that the instruction pointer can serve as a capability enabling access to memory. Since software components are composed of both code and data, a component's data can only be accessed by its code. Therefore, only if the instruction pointer originates from a component's code, can the constituent instruction access the component's data.

The CODOMs design is driven by the following guidelines:
(1) Software is composed of multiple components, both trusted and untrusted, that share the same address space.
(2) Components interact to perform actions. The system must: (a) support cross-domain synchronous call/return with low overhead; (b) provide protected domain entry points; and (c) prevent callers and callees from tampering with each other.
(3) To avoid buffer copies, interacting domains must be allowed to securely reference each other's internal memory. This requires that domains be allowed to efficiently grant and revoke permissions to access their memory regions, as well as to be able to verify the validity of the pointers they are given.

CODOMs operates by associating every page with a tag, and multiple (not necessarily consecutive) pages can share the same tag. Code pages are also associated with a list of tags they can access (and how), as well as the ability to execute privileged instructions. Domains are thus *code-centric* and the instruction pointer itself determines which pages and privileged resources it can access. Moreover, control can switch between domains using simple call/return instruction at negligible run-time overheads. To facilitate fast cross-domain calls, CODOMs allows in-place sharing of data across domains using application-controlled *capability registers*. These facilitate secure sharing of arbitrary regions (base pointer and size). Moreover, CODOMs offers protection against capability tampering and can perform selective, user-level capability revocations more efficiently than previously proposed systems [30, 19, 5, 28]. Importantly, CODOMs puts special emphasis on low-overhead revocation for the common case of sharing in synchronous, cross-domain call/return scenarios.

We have evaluated CODOMs using a cycle-accurate x86 simulator, showing that switching domains incurs extremely low overheads. On a larger scale, we show that CODOMs is capable of isolating each module in the Linux kernel into its own domain with negligible runtime cost.

The main contributions of this paper include:
Code-centric domains / instruction pointer as a capability: A hardware-based isolation mechanism where multiple software components can share an address space.
Simple and efficient domain switching: The code-centric organization facilitates domain management and low-latency

978-1-4799-4394-4/14/$31.00 © 2014 IEEE

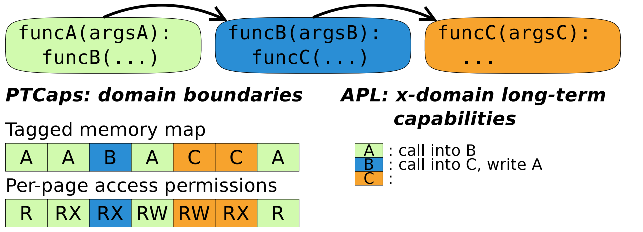


Figure 1: Example domain setup where components are isolated using CODOMs.

domain switching using simple call/return instructions.

Transient access grants: Application-controlled capability registers enable fine-grained, secure sharing of memory across domains without expensive OS intervention.

Efficient access grant revocation: CODOMs supports efficient selective capability revocation at user-level. Importantly, distinguishing between synchronous and asynchronous capabilities enables low-overhead revocation of the common (synchronous) cross-domain call/return case.

Mechanism complementarity: CODOMs mechanisms enable efficient adaptation to multiple system organizations and provide a path for systems to gradually harden their security.

2. CODOMs Concepts

The goals of the CODOMs architecture are to provide flexible hardware mechanisms to define domains and to support low-latency transfers of control and data across domains.

2.1. Code-Centric Isolation of Software Components

Code-centric isolation embraces the idea of software component as a unit, identifying code, data (both dynamic and static) and predefined interfaces for cross-component interaction. A domain is defined as an arbitrary collection of code and data pages, such that only the domain's code is allowed to freely access its code and data. This enables the instruction pointer to serve as a capability for accessing the domain, which can actually contain multiple components. Code-centric isolation also enforces domain interactions; if a domain is allowed to call into another, it can simply *call* a predefined entry point in the callee domain to transfer control.

Figure 1 illustrates a simplified example where routines from three software components (A, B and C) invoke each other. Specifically, routine *funcA* from domain A invokes routine *funcB* from domain B. The latter then invokes routine *funcC* (domain C). The figure also illustrates the two mechanisms that CODOMs uses to enforce domain isolation: *page table capabilities* (PTCaps) and *access protection lists* (APLs).

PTCaps extend page tables to include per-page tags such that all pages that compose a specific domain share a tag value. Besides the existing per-page permissions (R/W/X), the mechanism also adds two new permission bits to allow privileged operations and to store capabilities in memory (P and S bits, respectively; see § 4.1.1). In the figure, for example, all pages tagged with A compose domain A (data and code). This also demonstrates that domains can be composed of non-contiguous memory pages (i.e., a "sparse" memory region).

APLs maintain the cross-domain access and invocation permissions by associating each tag with a list of other tags and the operations it can perform on their constituent pages. For example, the APLs shown in Figure 1 indicate that code in domain A can call into the entry points of domain B and the code in domain B can call into domain C as well as write pages of domain A (if allowed by the per-page permissions).

When performing an access, CODOMs checks if: (1) the tag of the destination page is present in the APL for the tag of the currently executing instruction, and (2) the access is compatible with *both* the permission listed in the APL and the regular per-page protection bits. This provides a way to layer security at two levels: at domain (APL) and page granularity. For example, granting B write access to A does not allow it to write into the first memory page, since it is write-protected.

PTCaps and APLs are set up by the OS when the program (or a module thereof) is loaded. Furthermore, consistency across cores must be maintained through TLB shootdown operations [27]. Nevertheless, since the mechanisms are used for long-term access grants, these operations are infrequent.

The direct benefits of code-centric isolation are that it provides programmers and administrators simple mechanisms to express and enforce cross-domain permissions while incurring negligible runtime overheads when crossing domains. In contrast, more "traditional" capability systems need to explicitly manage capabilities, which work at a much more fine granularity; for example, domain A would require three different capabilities for its own pages (since they are not consecutive), plus one capability for every entry point in B. CODOMs thus provides higher performance with stronger software security and resiliency guarantees without affecting existing software design and synthesis methodologies.

2.2. Transient, Fine-Grained Capabilities

Software components communicate via procedure calls, whose arguments are passed either by value (using registers) or by reference (using pointers to memory). CODOMs preserves synchronous call/return semantics across domains using *capability registers*. These enable passing data by reference and thus avoiding costly memory copies or page remappings.

In the example, routine $funcA$ passes an argument to $funcB$, which forwards it to $funcC$. If that argument is a pointer to a memory buffer in domain A then $funcC$ will not be able to access it since its APL will not allow it. Pure isolation requires that buffers be reassigned their tag or copied across domains.

CODOMs therefore provides *capability registers* (CapRegs), an application-managed mechanism that temporarily grants access to a domain's memory. CapRegs are initialized by the user-level caller code to temporarily grant the callee access to an arbitrary memory range (at byte granularity). Once the callee is invoked, it is allowed to access said buffer until it returns control to the caller. Moreover, the callee is allowed to pass the CapRegs down the call chain to support call indirection. CODOMs provides instructions to "create" (i.e., initialize), copy, "weaken" (a controlled form of modification), activate/passivize (spill from/to memory), revoke, and verify pointers against CapRegs. CapRegs and their semantics are described in detail in § 4.1.3. For brevity, we will use the term capability when referring to both CapRegs and capabilities stored in memory. CODOMs only performs checks against active capabilities (CapRegs).

CODOMs ensures that regular code cannot forge capabilities by capping their authority to that of the instruction that created it. When an instruction creates a capability, the APL of the instruction's code page is copied into it. For example, suppose domain *B* in Figure 1 creates a capability spanning the last four pages. The capability will use *B*'s APL, allowing write access to all pages of domain *A* in the given range but not to the pages of domain *C*, since its APL precludes such access (thus capabilities are also "sparse"). § 4.1.3 describes how CODOMs maintains in-memory capability integrity.

Finally, CODOMs optimizes capability revocation by distinguishing between synchronous and asynchronous semantics:
Synchronous capabilities are used in cases where a caller only needs to grant temporary access rights to a callee (which may further delegate the grant). These are, by far, the most common type. Synchronous capabilities are implicitly revoked at no cost when the callee returns by ensuring it does not store them in memory; once it returns, it can no longer use them.
Asynchronous capabilities are used to grant accesses that outlive the callee. They are useful during asynchronous data transfers between domains (e.g., asynchronous disk read), or when two threads exchange capabilities through the memory. CODOMs provides efficient support to selectively revoke this type of capabilities (see § 4.1.5).

2.3. Usage Model

CODOMs proposes flexible mechanisms to support multiple use-cases where domains require different degrees of isolation, allowing systems to tune performance according to their needs.

The architecture is designed to allow the programmer and system administrator to define the desired security and fault isolation policies. For programmers, this burden can be as minimal as declaring that a compilation unit or a code library be used as an isolated domain and annotating its entry points — the routines that other domains can use to interact with it. For administrators, this enables enforcing system-wide policies such as using third-party modules in isolation so as to not compromise the system or affect its resiliency.

We envision immediate usefulness of the CODOMs architecture in structured scenarios where security-aware interfaces already exist (e.g., the user/kernel/hypervisor separation), or where some domains are a super-set of others (e.g., a plugin/application relationship). For example, CODOMs can subsume the prevalent kernel/user ring protection by implementing the kernel's system call interface as a privileged shared library running in a separate domain, allowing applications to securely invoke system calls without a processor trap.

CODOMs is also able to provide fine-grained component isolation in the spirit of capability-based addressing architectures [19, 5]. For example, using capabilities to traverse a linked list. Alternatively, the domain owning the linked list could provide an iterator interface that takes a callback to process each element (similar to `std::for_each` in C++). Nevertheless, such scenarios are beyond the scope of this paper, since they require complex capability register management through runtime, compiler and/or language extensions.

3. The Complexity and Performance of Memory Protection and Isolation

Memory protection and isolation has long been studied as an essential primitive for building secure and resilient systems. Proposed mechanisms attempt to balance simplicity of use with performance overheads. In this section we discuss previously proposed mechanisms while focusing on the three main axes of this tradeoff: (1) the complexity of defining and switching domains, (2) the performance of switching domains, and (3) the performance of sharing data between domains.

Existing protection mechanisms either provide complex usage semantics or incur substantial performance overheads when switching domains (or both). As a result, programmers typically push multiple semantic domains into a single physical one, thus compromising software security and resiliency.

Virtual memory address spaces are the most common mechanism for domain isolation [10, 22]. Managed by the OS, address spaces isolate arbitrary collections of memory pages by providing processes the illusion of using the memory of a virtually infinite standalone machine. The process abstraction, in turn, provides a convenient programming abstraction.

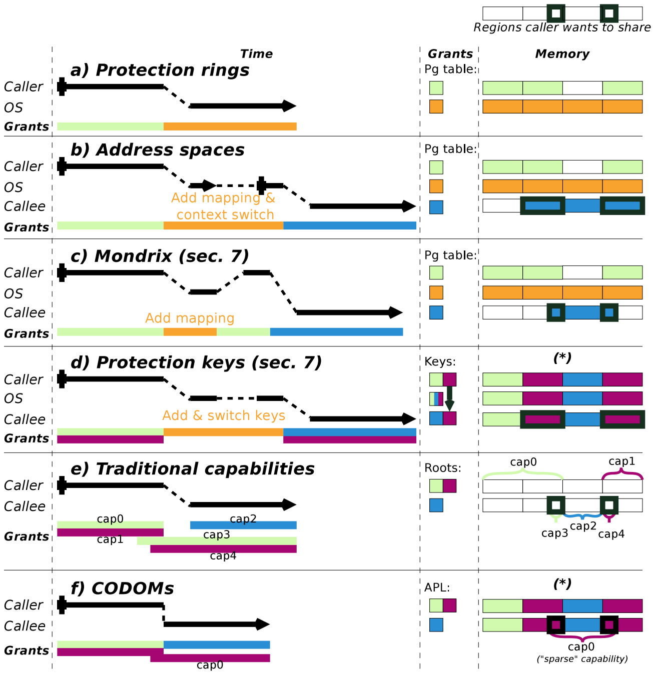
Protection rings (or privilege levels) is a common mechanism used to isolate executive code [23]. Rings are used by the OS to isolate itself from untrusted user processes, requiring the use of system calls for process/OS interaction.

Capability-based architectures lie on the other end of the spectrum [19]. Capabilities are communicable and unforgeable tokens that identify and authorize access to resources. Capabilities allow implementing fine-grained isolation that only grants software the minimum set of resources necessary to perform its task (principle of least authority).

3.1. Complexity of Defining and Switching Domains

Figure 2 illustrates notable domain isolation mechanisms.

Address spaces (Figure 2b) implicitly define domains for processes by encoding access permissions in the page table. Since the use of page tables (or similar forms of protection tables [15, 29]) is transparent to the application, address spaces provide a simple programming abstraction. However, as tables are privileged resources, managing them is costly and requires OS mediation. Furthermore, processes burden the programmer



(*) Caller and callee execute within the same page/protection table.

Figure 2: Illustration of domain crossings and data sharing for notable isolation mechanisms. The *Time* column illustrates the stages of a domain crossing, indicating the grants (regions of memory) available at each stage at the bottom; the *Grants* column shows the per-domain configuration of each mechanism; the *Memory* column shows the layout (and grants) from the point of view of each domain.

with managing concurrency and data movement/sharing.

Protection rings (Figure 2a) provide a hierarchical domain scheme. All rings (domains) share an address space, and code in one ring can access resources of all less privileged rings. The *totally-ordered* hierarchical structure of ring domains is less flexible than that of separate address spaces. For example, Linux kernel modules do not naturally fit this scheme: placing two completely independent modules in separate rings will unnecessarily give one full access to the other.

Capability systems (Figure 2e) grant access to memory resources through a set of “root” capabilities, and a domain is defined as the transitive closure of these roots. This requires using multiple capabilities to cover all the components that conform a single domain, and domain switches require switching all of them (e.g., in Figure 2e the caller has two “root” capabilities). Explicitly managing the roots burdens the programmer, so such systems have typically embedded high-level semantics into the domain-switching hardware. For example, protected procedures in the Plessey System 250 [19] define the notion of process in hardware, which handles the root capabilities. This design breaks backwards compatibility and imposes specific structure and isolation policies to the software.

CODOMs tries to take the best from each mechanism. A page-table system is used to aggregate domain resources, yet it allows multiple domains to share an address space, as well as invoke privileged operations. Importantly, this design makes code-centric domains compatible with existing software. Even though domains are managed by the OS they can be switched with simple control flow instructions, thereby simplifying their use and deployment. Finally, capabilities allow domains to share data at arbitrary granularity, thereby obviating expensive memory copies and OS-managed page table modifications.

3.2. Overhead of Switching Domains

Domain switching reconfigures some architectural resources and thus incurs in runtime overheads; at the very least, introducing a pipeline RAW dependency that affects ILP.

The overheads are exacerbated in mechanisms that require OS intervention to perform the switch, warranting an extra round-trip across privileged levels (Address spaces and protection keys). Protection rings and Mondrix embed additional cross-domain semantics in hardware, increasing these overheads. Traditional capability systems (Figure 2e) also incur extra overheads as they must switch the root capabilities (either through software or added hardware semantics). Finally, some address space isolation implementations need to flush the TLB on switches, on top of the OS intervention overhead.

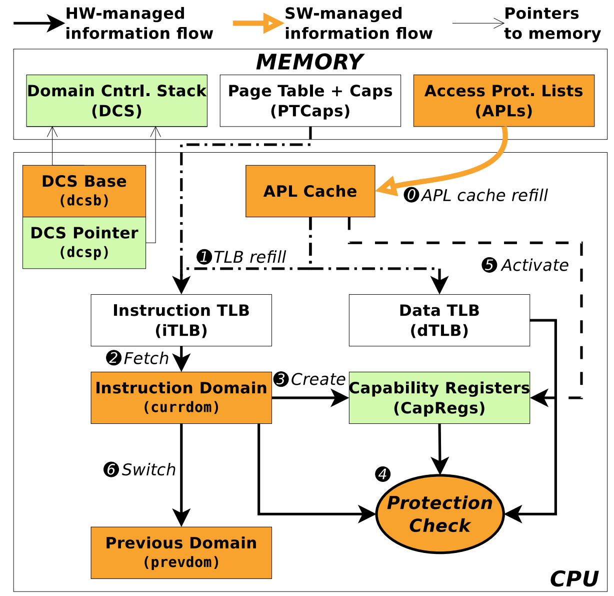
CODOMs circumvents all these overheads by having a code-centric approach to protection. It allows domain switches at user-level through regular function calls, without even paying the price of a RAW hazard during a domain switch.

3.3. Sharing Data Between Domains

Sharing data across domains is critical for effective domain interactions, yet the sharing facilities provided by different isolation mechanisms impose different runtime overheads.

Protection rings (Figure 2a) only allow low-privilege rings to easily share data with the high-privilege ones. Sharing in the opposite direction requires costly buffer copies or changing page table permissions. Address spaces (Figure 2b) limit sharing to page granularity and require costly OS intervention to modify page tables. Similarly, Mondrix (Figure 2c) requires OS-mediated table modifications to establish a shared memory region. Importantly, revoking or “downgrading” accesses on table-based systems imposes costly TLB shootdowns [27].

Capabilities (Figure 2e) can be viewed as a special form of routine arguments. In order to prevent forgery and tampering of capabilities, some systems (e.g., Plessey System 250 [19]) used typed segments to distinguish regular data from capabilities. Others (e.g., IBM System/38 [19]) allowed mixing data and capabilities by tagging memory at the word-level, thereby affecting memory and bandwidth utilization. In all cases, capabilities grant access to consecutive regions, and sharing non-consecutive regions requires setting up multiple capabilities (e.g., *cap3* and *cap4* in Figure 2e). To avoid this, data must be carefully laid out in memory, which is not always feasible for dynamic data structures. Finally, capability systems incur overheads when revoking capabilities. Since


HW-managed information flow
SW-managed information flow
Pointers to memory
MEMORY
Domain Cntrl. Stack (DCS)
Page Table + Caps (PTCaps)
Access Prot. Lists (APLs)
DCS Base (dcsb)
DCS Pointer (dcsp)
APL Cache
⓿APL cache refill
❶TLB refill
❺Activate
Instruction TLB (iTLB)
Data TLB (dTLB)
❷Fetch
Instruction Domain (currdom)
❸Create
Capability Registers (CapRegs)
❻Switch
❹
Protection Check
Previous Domain (prevdom)
CPU

Entry contents:

PTCaps	:	*..., T, P, S, Tag*	**currdom/prevdom**	:	*Tag, HwTag, HwAPL, P*
APLs	:	*Tag: [(Tag, Lvl), ...]*	**iTLB**	:	*..., P, HwTag*
APL cache	:	*Tag, HwTag, HwAPL*	**dTLB**	:	*..., S, HwTag*
CapRegs	:	*Base, Size, Tag, HwAPL, Lvl, Revoc. counter, Revoc. addr.*			

Figure 3: The CODOMs architecture. Added elements are colored, and extended elements are white. Light green elements are controlled by the application, and others by the TCB.

dellocated memory can be reused, a domain must revoke capabilities to unused memory regions to avoid leaking information. Various revocation methods have been proposed [19], many of which require OS intervention: (a) releasing entire segments and revoking all capabilities to said segment; (b) avoiding memory reuse until capabilities are garbage collected, which imposes expensive memory sweeps; or (c) using indirection tables that impose additional latency every time a capability is used. In all but the last case, it is not possible to revoke specific capabilities that allow a specific domain to access a specific buffer. Instead, all capabilities granting access to said buffer must be revoked.

The code-centric nature of CODOMs requires fewer, distinct capabilities since those are only needed for accesses beyond what is encoded in the APL. Furthermore, CODOMs ensures capability integrity without requiring memory tagging nor segmentation (§ 4). Finally, CODOMs optimizes revocations for the common synchronous (call/return) pattern.

4. CODOMs Implementation

CODOMs is designed to provide both security and performance, even in high ILP out-of-order processors. Figure 3 illustrates the key elements of the CODOMs architecture. As described below, some of the architectural elements must be managed by the *Trusted Computing Base* (TCB), which is the smallest subset of the OS required to maintain system integrity and enforce higher-level isolation policies.

4.1. Hardware Elements and Protection Primitives

4.1.1. Page Table Capabilities (PTCaps) are implemented as page table extensions, which include a *tag* (64 bits) and *tag presence* (T), *privileged capability* (P) and *capability storage* (S) bits. The latter are stored in unused bits of the page table entries, and tags are stored in a page physically contiguous to the page directory they extend. The T bit allows tags to be set for entire page table sub-trees, minimizing space and management overheads for the page table. The P bit indicates whether privileged operations are allowed in a code page, ensuring regular code cannot execute them. The S bit identifies pages that can be used to store capabilities.

4.1.2. Access Protection Levels specify four totally ordered values: *None*, *Use*, *Read* (including execute) and *Write*. *Use* is overloaded based on the target page:

- *Capability storage* pages: allows load/store of capabilities from/to memory and disallows regular loads/stores.
- *Code pages*: enables domain entry points. Permits calling into an address aligned to a system-configurable value. If accessed through a capability with size zero indicates an unaligned address and enables using arbitrary return addresses and function pointers across domains.

4.1.3. Capability Registers (CapRegs) store the per-core active capabilities. Capabilities occupy 256 bits (32 B) when stored in memory, and include a *base address* and *size* (2×48 bits), a 2-bit *access protection level*, a *revocation counter address* (48 bits), a *revocation counter value* (46 bits) and a *tag* (64 bits). CapRegs are managed by the following operations:

Create initializes a CapReg. The application must provide all the fields but the tag. The tag is set by the hardware (based on the PC) to prevent forgery, and the tag's APL is cached into the CapReg (see § 4.2). If a revocation counter is not provided, the capability is synchronous.

Modify is only allowed to weaken the access protection level and to shrink the address range of the capability.

Spill is only permitted at 32 B-aligned addresses and if the the target page is marked as *capability storage* and is accessible with (at least) a *Use* level. Synchronous capabilities can only be spilled into the DCS, and capability push/pop instructions are also provided to interact with the DCS (§ 4.1.4). Since a *Use* level ensures code cannot directly read or modify the memory contents, this ensures the integrity and unforgeability of passive capabilities.

Probe checks if dereferencing a pointer fails using a CapReg.

Usage only works with active capabilities. Two usage models are supported: *Implicit use* validates memory accesses against all active capabilities. This simplifies compiler support and enables transparently adding capability use to existing code[1]. *Explicit use* is provided trough separate instructions that identify which CapReg to use. Compilers can use this to minimize the number of CapReg checks, improving energy efficiency.

[1] An approach similar to the *sidecar registers* in Mondrix [30] could also be used to automatically bound the number of implicit capability checks.

4.1.4. Domain Control Stack (DCS) provides memory that can be used to spill capabilities. The DCS is a private per-thread memory structure with *capability storage* pages. Since it is private, all capabilities (synchronous and asynchronous) can be spilled into it without breaking synchronous capability revocation. The DCS is bounded by the `dcsb` and `dcsp` registers (Figure 3), and code is implicitly granted *Use* access to that range. Unprivileged code modifies the `dcsp` register only indirectly using capability push/pop instructions. The `dcsb` register controls DCS frames. It can only be modified by the TCB, and the hardware ensures pop operations never cross DCS frames. § 5.4 further describes the DCS operation.

4.1.5. Asynchronous Capability Revocation is based on the revocation fields set when a capability was created. Setting these fields is a privileged operation, since the hardware uses them to access memory. The *revocation counter address* points to a "revocation counter" stored in memory. A capability is considered valid as long as its counter value matches that stored in the revocation counter. When a revocation instruction is executed, CODOMs first verifies that the instruction's tag matches that stored on the capability. This ensures that only the domain that created a capability can revoke it. CODOMs then increments the revocation counter, thereby invalidating all capabilities that use the same counter address by setting their protection level to *None*. Selective revocation is possible by associating different capabilities to different counters.

Passive (stored in memory) asynchronous capabilities are lazily invalidated when loaded from memory if their counter value does not match the one in memory.

Active asynchronous capabilities that share a revocation counter are immediately invalidated. One possible implementation uses a central directory [27] to track the active asynchronous capabilities. In this case, the revoking core signals the directory, which in turn invalidates all capabilities that use the same revocation counter. Since most capabilities are synchronous (and not asynchronous), this operation is infrequent.

A revocation counter can be reused $2^{46} - 1$ times until it overflows (raising an exception), and 2^{48} different counters can exist in the system. When a counter is reused after an overflow, the system must ensure all capabilities that use the overflowed counter and are stored in capability storage pages of the current address space are invalidated. Nevertheless, the magnitude of the revocation counter ($2^{46} - 1$) makes such events extremely infrequent. More importantly, the system does not have to track the data capabilities grant access to.

4.2. Implementing Access Protection Lists in Hardware

The APLs of the recently used tags are cached in the per-CPU *APL cache* (see Figure 3). This cache is managed by the OS and allows multiplexing the unbounded spaces of tags and APLs (⓿ in Figure 3). The cache maps PTCaps tags to its hardware version *HwTag* and a portion of the corresponding APL (*HwAPL*). HwAPLs contain the 2-bit protection levels corresponding to cached tags (using *None* if not present).

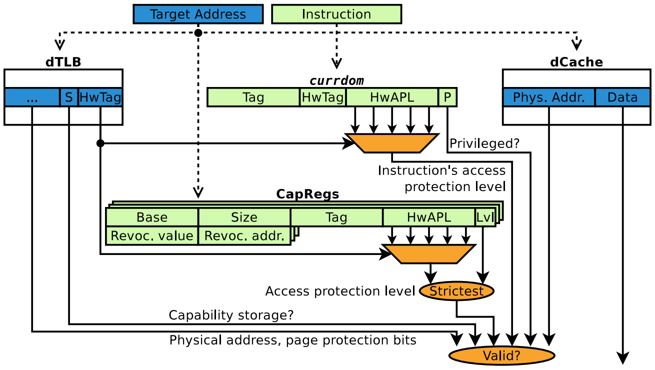


Figure 4: Access protection check logic on a memory access.

On a TLB miss, CODOMs caches the tag's APL entry in the TLB (❶ in Figure 3). The iTLB is extended with the *Tag*, *HwTag* and *HwAPL* from the APL cache entry and the *privileged capability* bit from the page table entry. The dTLB is extended with the *HwTag* and the *capability storage* bit. This information can be stored on a separate structure to optimize energy and delay, since it is only needed after a TLB hit.

When an instruction is fetched, the relevant information from the iTLB is stored in the `currdom` register (❷ in Figure 3), which encodes the information of the *current domain*. Since many instruction sequences reside in the same page, this can be optimized to reduce the number of iTLB lookups and the amount of storage required to pass that information. Whenever the contents of the `currdom` register are modified, its previous value is copied to the `prevdom` register, providing the identity of the previously executing domain (❻ in Figure 3). When a capability is created, the *Tag* and *HwAPL* of the `currdom` register are copied into the CapReg (❸ in Figure 3). The *HwAPL* in an active capability is never stored into memory, and is instead restored from the APL cache when it is activated (loaded from memory; ❺ Figure 3).

When an APL cache entry is modified, the *HwTag* and *HwAPL* values in the CapRegs and TLB entries of that CPU need to be reloaded. No TLB shootdown-like operations are required, since there is one independent APL cache per CPU, and its information does not leak into passive capabilities.

Figure 4 depicts CODOMs access checks (❹ in Figure 3). The *HwTag* in the dTLB is used to index the *HwAPL* of the `currdom` register and retrieve the protection level for the target address. Conversely, the *HwTag* in the `currdom` register is used to check control flow instructions. For example, an APL cache with 32 entries requires a 5-bit *HwTag* and a 64-bit *HwAPL* (32 entries of 2 bits). The same applies to capabilities, except that weakened capabilities are implemented by using the stricter of the capability and the *HwAPL* access protection level. The protection checks are performed in parallel to the actual cache access to hide their latency.

4.3. Domain Switches and Out-of-Order Execution

4.3.1. Protection Domain Checks and Switches on out-of-order processors are efficiently implemented through the `currdom` register. The information provided by the iTLB is sent to the rename stage. The `currdom` register is renamed every time a domain switch occurs (when its contents change). Since the register is not changed beyond that stage, the rename stage itself can set the new value and mark the register as ready. This eliminates RAW hazards during domain switches and allows maintaining instructions from different domains in-flight. Experimental measurements show that 6 physical `currdom` registers are sufficient to eliminate all RAW hazards.

4.3.2. Capability Registers may also generate RAW hazards when modified. CODOMs alleviates this by providing a 2-wide register window for active capabilities. Register `capX` is used by the current instruction sequence, and `capXn` for the “next” window. Writing into `capX` also writes into `capXn`. The windows are swapped on protection domain switches, thus allowing software to eliminate RAW hazards on `capX`.

5. System Software: Gradual Security Hardening

CODOMs primitives support multiple isolation granularities, ranging from simple user/kernel isolation to fine-grained isolation, with a performance profile tuned for each specific case. The degree of software component isolation depends on whether code can be modified to fully exploit CODOMs and the trust relationship between each pair of domains.

5.1. Domain Management

PTCaps and APLs are managed by the TCB. As domains are identified by their tag, domain creation takes an unused tag and constructs an APL for it. The APL initially contains a single *Write* grant to its own tag. Long-term cross-domain grants add entries into other APLs, while short-term grants can be directly handled through capabilities. By managing the PTCaps, the TCB can accurately control which pages store capabilities and which ones have access to privileged instructions.

5.2. Domain Boundaries

System policies dictate the desired domain boundaries: which software components are grouped into a single domain, and what type of isolation is required between domains. To that end, dynamically-loaded components can serve as the base isolation unit; current systems make extensive use of dynamic loading and linking, and applications (including the OS kernel) comprise a patchwork of loadable modules. Domains can consist of groups of tightly-coupled dynamically linked components (e.g., a plugin and its utility libraries) and interact by invoking each other’s routines.

A simple approach is to create a new domain for each process, which can then invoke the TCB to create more domains. The OS could also provide mechanisms to declare components that are always isolated from user code. For example, the user-level binary loader can be part of the TCB and can use cues encoded as binary format extensions, file system permissions or mandatory access control systems like *AppArmor* [1]. This applies both to domain relationships and their entry points.

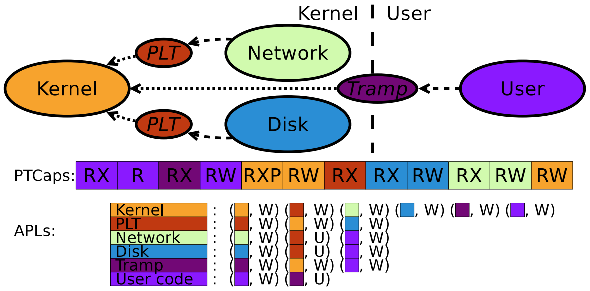


Figure 5: Example coarse-grained domain setup.

Existing loaders identify the public entry points of a dynamic module through its *Procedure Linkage Table* (PLT [18]). The PLT is an array of function trampolines (generated by the loader) that interfaces the entry points with the code that invokes them. The loader can harness this to enforce these entry points by giving a caller only *Use* access to the PLT (which, in turn, is given full access to the callee).

5.3. Example: Coarse-Grained Isolation

The code-centric approach enables coarse-grained isolation with minimal or no changes in the code. For example, Figure 5 shows a *User* that is isolated from the kernel, and *Network* and *Disk* subsystems that are isolated from each other.

CODOMs subsumes the implementation of privilege levels by: (1) giving *User* only *Use* access to call *Tramp*, a system call trampoline that then jumps into *Kernel* (similar to the *vdso* in Linux, or the *KIP* in L4 [16]); (2) setting the privileged capability bit on the *Kernel* code pages; and (3) giving all kernel-level domains full access to *User* (a simple return can resume execution at the user). Unlike privilege levels, CODOMs can also encode hierarchical domain relationships that are not totally ordered. For example, the *Network* and *Disk* subsystems in the example can be isolated from each other. Since they are dynamically loaded as Linux kernel modules, a PLT can be used to control their access to *Kernel*, provided that it uses capabilities when passing data to them. Given the flexibility of CODOMs, a backwards-compatible alternative that ensures subsystem isolation exists: give subsystems full access to *Kernel*, but not among themselves.

The same technique can also be applied at user-level. For example, an application can be granted only *Use* access to a an encryption component, which in turn has full access to the application. This allows the use of encryption without exposing the encryption keys to the application.

5.4. Example: Fine-Grained Isolation

In more fine-grained scenarios where relationships are not hierarchical, further actions are required: (1) both parties have to use capabilities to pass references to data, and the pointers should be verified against them; (2) since the same stack is used across domains, capabilities are used to grant access to portions of them; (3) some of the architectural state in the caller or callee might need to be hidden to the other party; and (4) caller and callee might need to use different DCS frames. As policies are software-provided (rather than being hardwired), other organizations are also possible. Furthermore, their requirements depend on the domain trust relationships.

We have developed a proof-of-concept compiler and linker support that leverages existing ABIs to handle these cases. The developer can tag functions and data to place them on specific domains. This information is embedded into the ELF binaries, together with domain trust relationships. Additionally, an interface compiler generates caller/callee stubs for a list of domain entry points. The routines are specialized according to the trust relationship between domains (up to the extreme of empty stubs if domains fully trust each other). Even though it could be generated by the linker, for simplicity the interface compiler also generates the PLT code (the "gate"). Like in other systems, the lack of advanced compiler support forces the programmer to explicitly manage pointer verification and capabilities for data beyond stack arguments.

Figure 6 shows the organization of two domains using the aforementioned tools. The stack is placed on a separate domain and is only accessible through a synchronous capability ($\texttt{CapReg}_0$). This ensures other threads will not be able to tamper with the return address after a cross-domain call. The gate code has the *privileged capability* bit (to manage DCS frames) and *Write* access to all domains. Since user code cannot access previous DCS frames, those can be safely used to store cross-domain information. The steps required to perform a full call/return in a completely isolated scenario (upper bound overheads for hybrid software/hardware isolation) are:

1) Caller: The caller stub starts by pushing into the stack and zeroing all registers not used as arguments, concealing from the callee all unnecessary information. The same applies to capabilities (pushed to the DCS). The stub then pushes any procedure arguments into the stack (if necessary), and grants them access by deriving $\texttt{CapReg}_1$ from $\texttt{CapReg}_0$, and adjusts $\texttt{CapReg}_0$ to forbid access to previous frames. Finally, it calls into the PLT, which executes the gate code.

2) Gate (call path): The gate code saves the regular stack pointers and `dcsb` register into the DCS and creates a new DCS frame by adjusting the `dcsb` and `dcsp` registers. It then injects itself in the callee's return path by saving the caller's return address to the DCS, replacing it with a pointer to its own return path address, and creating a capability for the callee to return into that address ($\texttt{CapReg}_2$ at the top of Figure 6; note the use of size zero to avoid alignment checks). Finally, the gate jumps into the callee stub. Note that this jump already exists in the PLT of dynamically loaded objects.

3) Callee: The callee stub conceals its state from the caller by zeroing all registers and capabilities that are not results, and then returns to the injected gate (thanks to $\texttt{CapReg}_2$).

4) Gate (return path): The gate restores the state it saved, unrolls the DCS frame, and jumps back to the caller's return address. Unrolling the DCS frame ensures all synchronous capabilities stored in it are implicitly revoked.

Processor speed	2.4 GHz
Processor width	4 (insts in fetch; μops for the rest)
Register file	160 (int), 144 (float)
Load/Store/Inst. queue entries	48/32/36
ROB entries	128
i/d TLB	64 entries, 4-way
i/d Cache	32 KB, 8-way, 4 cycles
L2 cache	256 KB, 8-way, 7 cycles
L3 cache	6 MB, 12-way, 30 cycles
RAM latency	65 ns
Capability registers	8
APL cache entries	32
`currdom`/`prevdom` registers	6

Table 1: Simulator configuration

5.5. Miscellaneous

5.5.1. Shared Libraries can be mapped to different domains by mapping their physical pages into multiple virtual addresses, one for each domain that uses them. All copies point to the same physical memory, which eliminates memory and cache performance overheads but increases TLB pressure (much like Mondrix [30] and Koldinger et al. [15]).

5.5.2. Ambient Authority is a concept on which the POSIX interface relies, as some operations require the OS to be aware of the caller's identity. For example, memory allocations must be assigned to the requesting domain. This is handled by inspecting the `prevdom` register or, alternatively, providing a software capability-based interface (e.g., file descriptors).

5.5.3. Function pointers can be passed using a capability. If the caller is considered potentially malicious, the address must be verified against the provided capability. In order to execute the callee outside the caller's DCS frame, the TCB can provide a trampoline that creates a new frame and then calls the function (similar to the gate code above).

6. Evaluation

We have evaluated CODOMs using the cycle-accurate GEM5 simulator [4] running in full-system out-of-order mode with version 2.6.27.62 of the Linux kernel. Our simulation parameters, listed in Table 1, mimic an Intel Nehalem processor.

6.1. Micro-Benchmarks

We compare the performance of domain switching in CODOMs and other mechanisms (see Figure 2 and §§ 3 and 7) by calling a procedure on a different domain for 10K times. Every benchmark tests a combination of mechanism, number of procedure arguments, and randomly generated caller and callee workloads (see Table 2). We compare these against a regular call/return, taking the second of two repetitions.

6.1.1. Alternative Isolation Mechanisms are evaluated using micro-benchmarks that measure the following:

Syscall: The overhead of using an empty system call.

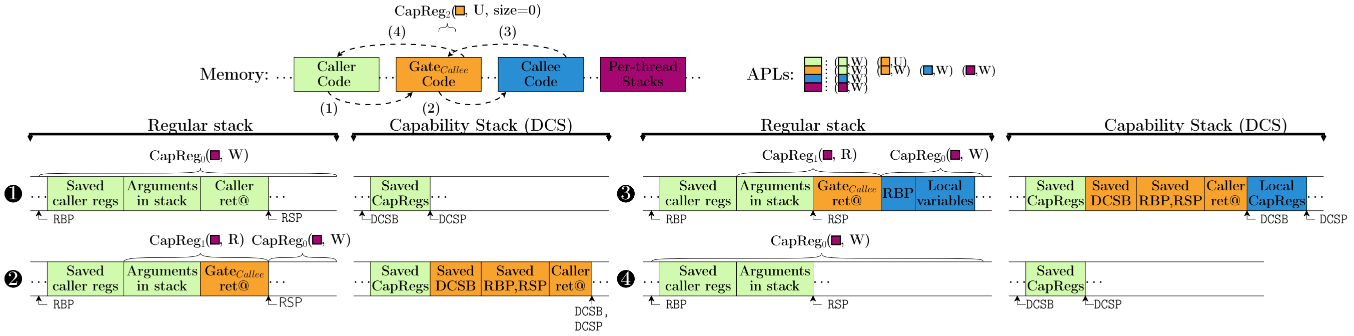


Figure 6: Example sequence of two fully-isolated domains: caller $\xrightarrow{1}$ gate $\xrightarrow{2}$ callee $\xrightarrow{3}$ gate $\xrightarrow{4}$ caller. Colors in stack/DCS show the domain that wrote into that memory.

Parameter	Values
Number of arguments	0, 5, 10
Caller/callee workload insts.	0, 25, 50, 100, 1000
Workload distribution	60% integer / 20% read / 20% write

Table 2: Micro-benchmark execution parameters.

Address Spaces: The cost of switching address spaces by communicating data using a POSIX pipe.

NaCl [31]: The callee switches segments (`cs`, `ds` `es` and `gs`) at user-level before and after performing its workload, imitating a naively optimistic implementation. The technique has also been applied at the kernel level [21].

Memory Keys (kernel) [15, 12, 14]: An approximation of a key-based memory protection switch, used to isolate kernel components. A system call implements the callee, switching keys before and after the call. Optimistically assumes that the cost of switching keys is equivalent to an instruction barrier.

Memory Keys (user) [15, 12, 14]: Like the previous one, but used to isolate user components. Before and after its workload, the callee invokes a system call that switches the keys.

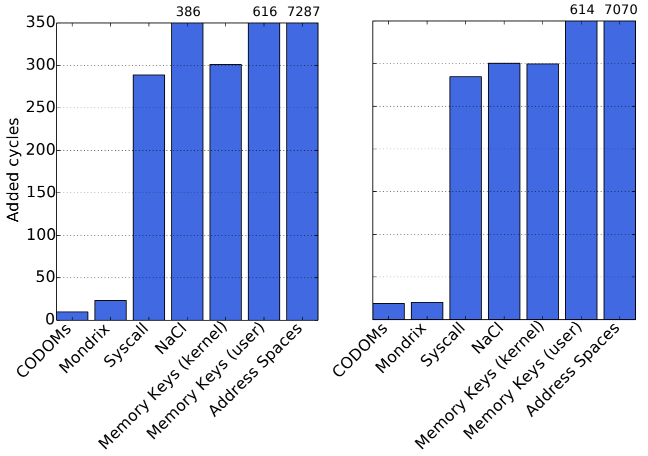
Mondrix [30]: Implicitly switching domains using call/return instructions. The benchmark optimistically approximates the cost of a domain switch using an instruction barrier, and it does not model the OS intervention and TLB-shootdown-like costs associated to grants and revocations (see § 7).

Figure 7 depicts the mechanisms overhead. The figure show that *CODOMs* and *Mondrix* provide the lowest overheads. Nevertheless, experiments on a real machine (not presented for brevity) show that the overheads of our coarse *Mondrix* approximation are over an order of magnitude higher than CODOMs. The figure shows the *None (leak GPR)* setting for CODOMs (Figure 8), slightly more secure than Mondrix.

Other mechanisms incur substantially higher overheads. All but *CODOMs*, *Mondrix* and *NaCl* require intermediate system calls (i.e., switching domains is privileged). In addition, all mechanisms but *CODOMs* hinder pipeline throughput by introducing RAW hazards on a domain switch. Thus, CODOMs is the only system to eliminate both sources of overheads.

6.1.2. Domain Trust Relationship defines the overhead of gate and stub codes (§ 5.4). We evaluate the following settings:

All: Both domains trust each other, and the callee can access the caller (to avoid using a return capability); no state is concealed and *gate* code is equivalent to a resolved PLT entry.



(a) Avg. of all workloads (b) Avg. of empty-body workloads

Figure 7: Domain switch overhead for the evaluated systems. The overhead is depicted as additional cycles over a procedure call/return. *Mondrix* and *Memory Keys* are optimistically approximated by an instruction barrier. Our *Mondrix* approximation does not simulate the added costs of grants and revocations. *CODOMs* uses setting "*None (leak GPR)*" in Figure 8.

Caller: The callee trusts the caller, but the caller conceals its state from the callee (e.g., kernel→module).

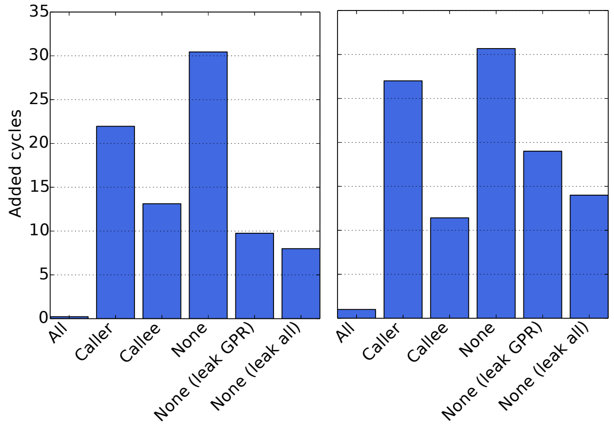
Callee: The opposite trust relationship (e.g., module→kernel).

None: Domains do not trust each other (≈ *Caller* + *Callee*).

None (leak GPR): Similar to *None*, but general-purpose registers are not concealed. Slightly more secure than Mondrix (which provides read access to the whole stack).

None (leak all): Similar to *None*, but no register is concealed (e.g., guard against dangling pointers and stack smashing).

Figure 8 depicts the overhead of switches for the different trust relationships. The figure shows that *All* delivers the best performance by avoiding RAW hazards, and only incurs in the overhead of the jump in the gate code. In contrast, *None* shows the highest overhead since it implements all the operations described in § 5.4. Still, its overhead is lower than other mechanisms. The rest show intermediate overheads whose main factors are the DCS frame management, the gate's



(a) Avg. of all workloads (b) Avg. of empty-body workloads

Figure 8: Cycles added to domain switch operations over a procedure call/return, for different domain trust relationships.

Benchmark	CODOMs	Code	Total
CODOMs: All	0.45	0.09	0.54
CODOMs: Callee	0.47	2.41	2.88
CODOMs: Caller	0.49	6.69	7.18
CODOMs: None	0.50	7.57	8.07
CODOMs: None (leak GPR)	0.48	3.54	4.02
CODOMs: None (leak all)	0.47	3.01	3.48
Address Spaces	-	-	1280.83
NaCl	-	-	40.75
Syscall	-	-	29.42

Table 3: Average energy overheads (%), relative to a procedure call/return. Includes setups in Figure 8 and x86 mechanisms.

return address injection, and the safeguard of the caller's stack pointers (this last not present in *Callee*). These results show that separating mechanisms from policies allows tuning the performance to the desired isolation properties.

6.2. Hardware Overheads

We have modeled CODOMs using McPAT [20] (32nm process), which estimates CODOMs incurs a 1.89% per-core area overhead. Table 3 shows CODOMs energy overheads compared to other x86 mechanisms. The CODOMs overheads are decomposed into the hardware structures and the execution of the additional policy-specific code. The table shows that CODOMs energy overheads are practically negligible.

6.3. Macro-Benchmarks

We have evaluated the system-wide impact of CODOMs by considering all Linux kernel modules as separate domains. The overall system overhead was measured for two macro-benchmarks: a parallel Linux kernel compilation, and *netperf* using the TCP bulk transfer test.

Since detailed full-system simulation is too slow, we have extrapolated the macro-benchmarks' timing by injecting the domain switch overheads (obtained by the micro-benchmarks) to the runtime of a native execution. The number of domain switches was measured using a modified version of QEMU [2], considering every module as a separate domain by inspecting their load addresses.

	Domains	Switches	Instructions	
			→	←
Compile	kernel / ext2	6403400	1029	16
	kernel / scsi	1777834	200	19
	kernel / libata	1638960	360	26
	kernel / cfq-iosched	1187154	390	31
	kernel / unix	149170	234	13
	scsi / scsi-sd	105444	21	48
	libata / scsi	63270	111	13
	Others	114327	-	-
	Total	11439559	-	-
netperf	kernel / e1000	22098737	261	41
	Others	14048	-	-
	Total	22112785	-	-

Table 4: Number of domain switches (calls & returns) during the benchmarks' execution. The two rightmost columns show the arithmetic mean of instructions executed in a domain before switching into the other.

Isolation model	Compile (%)	*netperf* (%)
None	0.10 ± 0.01	0.15 ± 0.02
None (leak-GPR)	0.03 ± 0.01	0.05 ± 0.02

Table 5: Runtime overheads incurred when considering each kernel module a separate domain.

Table 4 shows that modules typically perform short bursts of operations. This demonstrates the adequacy of providing unsupervised domain switching and access grant primitives with a low impact on ILP. Furthermore, more than 99.6% of the domain switches involve no more than 8 domains.

Table 5 depicts the system slowdown. In all cases, the overheads are effectively negligible (less than 1%), although replacing system calls with CODOMs would actually improve system performance. Even though our Mondrix approximation provides similar raw performance, it still requires OS intervention and operations similar to a TLB shootdown (GLB and PLB tables) for grants and revocations.

Figure 9 shows the memory access distribution of the domains, according to the owner of that memory. Dynamically allocated memory is owned by the domain requesting the allocation, or the one creating an allocation pool. Most of the non-stack accesses go to "remote" memory (of some other domain) in the call chain ("*Synch.* *" accesses), suggesting these can be handled with synchronous capabilities: the owner is (indirectly) calling into the domain that uses that memory.

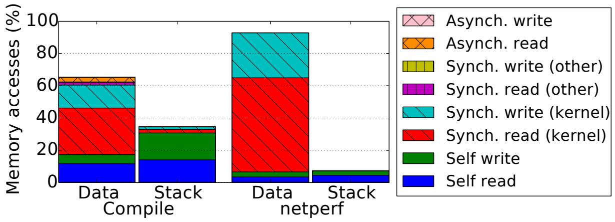


Figure 9: Domain memory access distribution, according to the owner of the accessed memory.

Mechanism		Number of hardware domains	Domain switch		Grant / revocation				Hardware Costs
			Runtime overhead	User-level	Mechanism	User-level	Granularity	Sparse	
Common	Address spaces	∞	↑↑	✗	PT	✗	Page	✗	-
	Privilege levels	↓↓	↑	✗	-	-	-	-	-
Virtual Memory-based	NaCl [31]	↓⋆	↓	✓	PT	✗	Page	✗	-
	Nooks [26]	∞	↑↑	✗	PT	✗	Page	✗	-
	Fides [25]	∞	↑↑↑	✗	PT	✗	Page	✗	-
	Small spaces [21]	↓⋆	↓	✗	PT	✗	Page	✗	-
	Memory keys	→⋆	↓	✗	PT	✗	Page/key	✓	↓
	PLB [15]	↑⋆	↓♣	✗	PT	✗	Page	✗	↑
	Mondrix [30]	∞	↓♣	✓	PT†	✗†	Range	✗	↑↑
Encryption	SP [17]	↓↓	↓	✗	PT	✗	Page	✗	↑
	Bastion [7]	↑	↑↑↑	✗	PT	✗	Page	✗	↑↑
Capabilities	Guarded ptrs. [5]	∞	↓	✓	Reg.	✗	Range	✗	↑↑
	CHERI [28]	∞	↓	✓	Reg.	✗	Range	✗	↑↑
CODOMs		→⋆	↓↓↓	✓	Reg.	✓‡	Hybrid	✓	→

↓↓↓ / ↓↓ / ↓ / → / ↑ / ↑↑ / ↑↑↑ / ∞: Extremely low / Very low / Low / Medium / High / Very high / Extremely high / Unlimited

⋆ Software multiplexing can be used to provide more domains

♣ Uses tagged associative TLB-like structures

† HW-support for transient read-only access permissions to stack area

‡ "Asynchronous" capabilities (§ 4.1.5) may require supervision

Table 6: Summary of some hardware-assisted domain isolation mechanisms (*PT*=Page table; *Reg*=Register).

Of these, most accesses point to memory owned by the main kernel ("*Synch. * (kernel)*"). This is primarily due to structures allocated at its generic layers, later accessed by the device- or protocol-specific modules. A rewrite of the system could minimize these accesses, as could having the core *Kernel* accessible from all domains, while retaining inter-module isolation. Still, "remote" accesses are an intrinsic property of fine-grained isolation.

Most "remote" accesses can be handled with synchronous capabilities ("*Async., **" is small), showing the convenience of distinguishing between capability types in CODOMs.

7. Related Work

The security and reliability issues that plague the software world have revitalized interest in more efficient and finer-grained memory isolation mechanisms.

Table 6 summarizes proposed domain isolation mechanisms, starting with *Common* primitives such as virtual address spaces and privilege levels. While effective, these primitives are tuned for application-level isolation. They incur high overheads when switching protection domains, require costly OS intervention [26, 25] and are limited to page granularity.

Software Fault Isolation (SFI) techniques provide fine-grained isolation by enforcing policies in software. They require non-trivial additions to the TCB, increasing the attack surface. Some rely on specific languages and trusted toolchains (e.g., Singularity [13] or BGI [6]), or rely on a trusted VM (e.g., SPIN [3] or Java [11]). Others verify binaries at load-time using *proof-carrying code* [24], or rely on dynamic binary translation [8, 9]. Imposing languages and tools makes it harder for third-parties to develop and distribute their software. Furthermore, SFI has overheads on safe purely computational code [24]. Systems like NaCl [31] mix SFI with existing mechanisms like segmentation [31, 9] or paging [8]. Nevertheless, this hybridization only serves as a measure to partially alleviate the overheads of SFI.

Some systems build on top of existing virtual memory primitives. *Nooks* [26] and *Fides* [25] are susceptible to high domain switching overheads, while *Small Spaces* [21] relies on scarce resources like segment registers. Still, these mechanisms are privileged and thus require costly OS intervention.

PA-RISC, Itanium and POWER 6 use key-based memory protection [15, 12, 14] (Figure 2d). Page table entries contain a tag identifier, and a small key set describes the tags that can be accessed at any given time. The key set is a privileged resource, requiring costly OS intervention, and multiplexing the tags requires expensive TLB shootdown operations.

Koldinger et al. [15] decouple protection and translation by running all processes on a single address space, instead adding a *Protection Lookaside Buffer* (PLB); a separate TLB-like structure that maps virtual page addresses and protection domains to access rights. The PLB is a privileged resource, requiring expensive OS intervention.

Mondrix [30] builds on the PLB concepts (Figure 2c), providing protection at arbitrary granularities. Unsupervised domain switches are supported by adding a table that controls the ability to switch domains at call/return boundaries (separately cached in hardware, the GLB). Both *PLB* and *Mondrix* use tables that are managed by the OS, require expensive highly-associative hardware caches, and revoking grants requires expensive operations akin to TLB shootdowns [27]. *Mondrix* alleviates some of these costs by adding domain switch semantics that control read-only grants to the stack.

Bastion [7] protects components in untrusted software stacks using extended hardware virtualization. The hypervisor adds a domain tag to pages of protected components. Whenever the processor interacts with external memory, tagged pages are encrypted and their contents verified. The use of a hypervisor incurs in high overheads, while encryption and hashing (interspersed with code and data) can delay memory accesses and induce poor memory bandwidth utilization.

Several newer capability architectures have been proposed, such as Carter et al. [5] and CHERI [28]. The main advantage of Carter et al. [5] lies in its integration of capabilities as protected pointers, while CHERI [28] provides a more classical approach that avoids high-level hardware constructs. Still, both suffer from most of the problems on previous systems: capabilities are not sparse, do not support efficient revocation, use word-level memory tagging, and (after switching) a domain must explicitly configure the set of capabilities that define its "root" grants.

8. Conclusions and Future Work

This paper presents CODOMs, an architecture that enables software components to be isolated using separate protection domains that coexist on the same address space. These isolation domains can efficiently interact through regular procedure calls. CODOMs draws from previous works on memory protection keys and capability architectures to hybridize their ideas in a novel architecture.

CODOMs provides the novel concept of code-centric protection domains, which simplifies the management of domains, provides very low overheads (even in OoO processors), and enables transparent integration. CODOMs also provides transient capabilities that do not require expensive memory tagging. In addition, CODOMs capabilities support efficient access grant and selective revocation operations.

The composable nature of CODOMs primitives avoids hardwiring semantics into the hardware, allowing systems to tune performance according to their needs, and enabling software developers to gradually harden system security and resiliency.

Our evaluation shows that CODOMs incurs low latency protection domain switches and provides efficient access grants and revocations. Furthermore, it can maintain pipeline throughput even in out-of-order processors. Some of the contributed techniques can also be applied to other systems.

CODOMs is a first step towards restructuring the hardware/software interfaces to support modular, exokernel-like hypervisors and OSes. To this end, we plan to extend CODOMs to support device I/O protection in order to allow arbitrary protection domains to safely and directly operate with devices.

Acknowledgements

We would like to thank Lluc Álvarez, Javier Cabezas, Ana Jokanović, Marc Jordà, Carlos Villavieja, our shepherd Mohit Tiwari and the anonymous reviewers for their help and comments on this paper. This work has received funding from: the European Commission through TERAFLUX (FP7-249013) and RoMoL (GA-321253); the Spanish Government through Programa Severo Ochoa (SEV-2011-0067); the Spanish Ministry of Science and Technology through TIN2007-60625 and TIN2012-34557; the Israel Science Foundation (grant 769/12 and equipment grant 1719/12); and the Ministry of Science and Technology, Israel. Yoav Etsion was supported by the Center for Computer Engineering at the Technion.

References

[1] M. Bauer, "Paranoid Penguin: An introduction to Novell AppArmor," *Linux Journal*, 2006.

[2] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX ATC*, 2005.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility safety and performance in the SPIN operating system," in *SOSP*, 1995.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, August 2011.

[5] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," in *ASPLOS*, October 1994.

[6] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *SOSP*, 2009.

[7] D. Champagne and R. B. Lee, "Scalable architectural support for trusted software," in *HPCA*, 2010.

[8] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: software guards for system address spaces," in *OSDI*, 2006.

[9] B. Ford and R. Cox, "Vx32: lightweight user-level sandboxing on the x86," in *USENIX ATC*, 2008.

[10] J. Fotheringham, "Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store," *CACM*, October 1961.

[11] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Addison-Wesley, 1996.

[12] C. Gray, M. Chapman, P. Chubb, D. Mosberger-Tang, and G. Heiser, "Itanium — a system implementor's tale," in *USENIX ATC*, April 2005.

[13] G. C. Hunt and J. R. Larus, "Singularity: rethinking the software stack," *SIGOPS Operating Systems Review*, 2007.

[14] *Power ISA* ^{TM}, Version 2.06 revision B ed., IBM, July 2010.

[15] E. J. Koldinger, J. S. Chase, and S. J. Eggers, "Architectural support for single address space operating systems," in *ASPLOS*, 1992.

[16] L4Ka Team, *L4 eXperimental Kernel Reference Manual, Version X.2*, System Architecture Group, Department of Computer Science, Universität Karlsruhe, May 2003.

[17] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, "Architecture for protecting critical secrets in microprocessors," in *ISCA*, 2005.

[18] J. R. Levine, *Linkers and Loaders*. Morgan Kaufmann, 1999.

[19] H. M. Levy, *Capability-Based Computer Systems*. Digital Press, 1984.

[20] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.

[21] J. Liedtke, "Improved address-space switching on Pentium processors by transparently multiplexing user address spaces," German National Research Center for Information Technology, Tech. Rep., 1995.

[22] W. Lonergan and P. King, "Design of the B5000 system," *DATAMATION*, May 1961.

[23] M. D. Schroeder and J. H. Saltzer, "A hardware architecture for implementing protection rings," *CACM*, March 1972.

[24] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary CPU architectures," in *USENIX Security*, 2010.

[25] R. Strackx and F. Piessens, "Fides: selectively hardening software application components against kernel-level or process-level malware," in *CCS*, 2012.

[26] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *SOSP*, 2003.

[27] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. Unsal, "DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory," in *PACT*, September 2011.

[28] R. N. Watson, P. G. Neumann, J. Woodruff, J. Anderson, R. Anderson, N. Dave, B. Laurie, S. W. Moore, S. J. Murdoch, P. Paeps, M. Roe, and H. Saidi, "CHERI: a research platform deconflating hardware virtualization and protection," in *RESoLVE*, March 2012.

[29] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," in *ASPLOS*, October 2002.

[30] E. Witchel, J. Rhee, and K. Asanović, "Mondrix: memory isolation for Linux using mondriaan memory protection," in *SOSP*, 2005.

[31] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: a sandbox for portable, untrusted x86 native code," *CACM*, January 2010.