SecureCells: A Secure Compartmentalized Architecture

Atri Bhattacharyya, Florian Hofhammer, Yuanlong Li,
Siddharth Gupta, Andres Sanchez, Babak Falsafi, Mathias Payer
EPFL, EcoCloud

***Abstract*—Modern programs are monolithic, combining code of varied provenance without isolation, all the while running on network-connected devices. A vulnerability in any component may compromise code and data of all other components. Compartmentalization separates programs into fault domains with limited policy-defined permissions, following the Principle of Least Privilege, preventing arbitrary interactions between components. Unfortunately, existing compartmentalization mechanisms target weak attacker models, incur high overheads, or overfit to specific use cases, precluding their general adoption. The need of the hour is a secure, performant, and flexible mechanism on which developers can reliably implement an arsenal of compartmentalized software.**

We present SecureCells, a novel architecture for intra-address space compartmentalization. SecureCells enforces per-Virtual Memory Area (VMA) permissions for secure and scalable access control, and introduces new userspace instructions for secure and fast compartment switching with hardware-enforced call gates and zero-copy permission transfers. SecureCells enables novel software mechanisms for call stack maintenance and register context isolation. In microbenchmarks, SecureCells switches compartments in only 8 cycles on a 5-stage in-order processor, reducing cost by an order of magnitude compared to state-of-the-art. Consequently, SecureCells helps secure high-performance software such as an in-memory key-value store with negligible overhead of less than 3%.

1. Introduction

Modern software systems are complex but monolithic, comprising multiple interacting subsystems, incorporating third-party code like libraries, plugins, or interpreted code, while interacting over untrusted interfaces including networks, shared memory, file systems, or user input. The lack of isolation between the components of a monolithic program allows vulnerabilities to have far-reaching consequences. An attacker who exploits one component can corrupt other parts — for example, a buggy Linux driver can compromise core kernel data structures. The traditional process abstraction for running monolithic software violates the Principle of Least Privilege [1] which requires components to only have access to the data necessary for their operation. Instead, all code running within a process' address space has equal permissions to all data and code regions allowing attackers to subvert pre-defined interfaces between components. For example, calls between components can jump to an arbitrary address bypassing checks on function call arguments.

Intra-address space compartmentalization allows developers to *isolate components* of a program within compartments, only granting each compartment permissions to access their own data. When compromised, a buggy component cannot access another component's data. Conversely, a component is guaranteed integrity of its private data against other corrupted compartments. Compartmentalization is a key defense mechanism that leverages the inherent modularity of code to fortify the cloud [2], [3], [4] and desktop [5] sandboxed environments, programs with third-party libraries [6]and underpins the design of security-focused microkernel operating systems [7], [8], [9], [10]. Compartmentalization constrains the negative effects of the myriad possible faults in software, including memory safety violations and logic errors, to compartment boundaries. For example, the Log4Shell exploit (CVE-2021-44228 [11]) which allowed attackers to exfiltrate secrets and inject arbitrary code in memory-safe programs can be mitigated by isolating the vulnerable Log4j framework in a separate compartment.

The compartmentalization mechanism enforcing the rules of access and communication between the program's components must be secure, performant and flexible. To be secure, the mechanism must enforce policy-dependent restrictions on memory accesses and inter-compartment calls in the face of powerful attackers. Particularly, the mechanism must prevent compromised compartments from escalating their memory access rights or from bypassing inter-compartment call gates. Developers for performance- and security-critical software such as operating systems constantly trade off the benefits of protection mechanisms against their overheads. The mechanism must implement low overhead checks and operations to support fine-grained compartmentalization for such programs. Faster compartment switching, for example, enables developers to refactor programs into smaller compartments with more frequent compartment switches, improving security while maintaining the same performance. Finally, a flexible mechanism which is able to support the wide variety of desired compartmentalization policies will bolster developer adoption.

Existing compartmentalization mechanisms lack one or more desirable features, often trading security for performance, or flexibility for backward compatibility or im-

plementation simplicity. Traditional, process-based isolation [12], [13], [14] only permits costly, microsecond-scale compartment switches. On the other end of the spectrum, protection-key [15] based mechanisms [16], [17], [18] are performant, with nanosecond-scale switches, but fail to deter attackers with code-injection capabilities. Mechanisms co-locating permissions with page-based virtual memory [17], [18], [19], [20], [21], [22] improve compatibility with existing page-tables but inherit the limited reach of modern Translation Lookaside Buffers (TLBs), incurring overheads for programs with large working sets. Finally, other mechanisms [21], [23] target simpler policies, such as protecting a single trusted compartment from an untrusted compartment.

SecureCells achieves the trifecta of secure, flexible, and high-performance compartmentalization by embedding compartmentalization into the architectural virtual memory abstraction. SecureCells proposes *i*) *TCB-maintained VMA-scale* access control, and *ii*) unprivileged (i.e., userspace) instructions implementing *securely-bounded* compartmentalization primitives, with *iii*) software implementing call gates, call stacks, and context isolation. Related efforts towards languages, compilers and libraries for compartmentalization can extend these benefits to developers by using SecureCells as the underlying isolation mechanism.

For the first pillar, access control, SecureCells introduces the first VMA-granular permissions table consolidating permissions for all compartments into a single data structure designed for efficient permission lookups. In contrast, previous mechanisms use per-compartment permission tables with either duplicate VMA bounds information [14], duplicate per-page permissions within a VMA [17], [18], or both [20], [13]. Deduplicating VMA bounds accelerates compartment switching, eliminating the need to re-load bounds for the target compartment. VMA-scale permission tracking requires smaller VMA-based permission lookaside buffers while also overcoming TLB-reach limits.

For the second pillar, SecureCells accelerates common compartmentalization operations with novel, low-cost unprivileged instructions. Particularly, SecureCells is the first mechanism to allow generic, unprivileged permission transfer from userspace. SecureCells maintains the integrity of permissions by bounding the semantics of untrusted userspace operations to known-safe parameters — the hardware checks the compartment switch instruction to enforce call gates, and permission transfer instructions to prevent privilege escalation.

SecureCells' final pillar leverages the flexibility of software for operations where possible without compromising security or performance (context isolation, call gates and call stack maintenance). This paper shows the first software mechanism for restoring register context following a compartment switch, necessary for isolating compartment contexts, without trusting any general-purpose registers.

In this paper, we:

- define SecureCells' key security and performance objectives and survey how related mechanisms meet these goals,
- propose SecureCells, a novel, secure, flexible, and performant mechanism which introduces compartments into the architectural virtual memory abstraction,
- apply SecureCells to typical application scenarios,
- present a hardware implementation of SecureCells based on the 5-stage in-order RISC-V RocketChip, and
- characterize SecureCells' performance for compartmentalizing micro- and macro-benchmarks.

2. Objectives For Architectural Isolation

Compartmentalization mechanisms are characterized by a specific set of objectives, which we introduce in this section. We demonstrate how SecureCells' objectives benefit compartmentalization using two representative programs described below and discuss how alternate goals lead to the differing designs of related mechanisms.

The characteristics of a compartmentalization mechanism determine its applicability. Primarily, the mechanism must be *secure* (**O1**) and enforce the restrictions on data access and communication despite arbitrarily compromised compartments. Second, the mechanism must be *performant* (**O2**), with low overhead for enforcing its checks and restrictions. A performant mechanism allows high-performance software to be compartmentalized without violating performance targets. Finally, a mechanism must be *flexible* (**O3**) in order to support the varying needs of software across security and performance criticality, and their corresponding isolation policies. A flexible mechanism does not make additional assumptions such as a hierarchical trust structure among compartments, and allows data to be shared in an arbitrary fashion. We concretize these objectives, based on insights from existing and candidate compartmentalized programs and related work, in the following subsections. We justify each objectives' importance using the two characteristic programs described below.

Use case: Browser. The first program, a browser (Figure 1), consists of a just-in-time (JIT) compilation engine (Engine), a sandboxed web application (WebApp) compiled and executed by the Engine, and a cryptographic library (CryptoLib) storing a secret key for encryption. The compartmentalization policy aims to isolate the Engine's data and CryptoLib's secret from the possibly malicious WebApp. Borrowing the threat model for browsers, we assume that the WebApp can exploit bugs in the Engine's compiler to generate and execute arbitrary code as the WebApp compartment, ultimately aiming to leak the browser's data or the cryptographic secret. The developer aims to prevent unauthorized inter-compartment data accesses by enforcing the per-compartment permissions shown in the figure. To maintain similar performance to the monolithic version, the developer desires minimal overhead from operations added due to compartmentalization: context switching and compartment switching.

Use case: Network Function Virtualization. The second program is a virtual network function pipeline (Figure 2) consisting of three stages progressively performing processing steps on a stream of packets. In particular, this

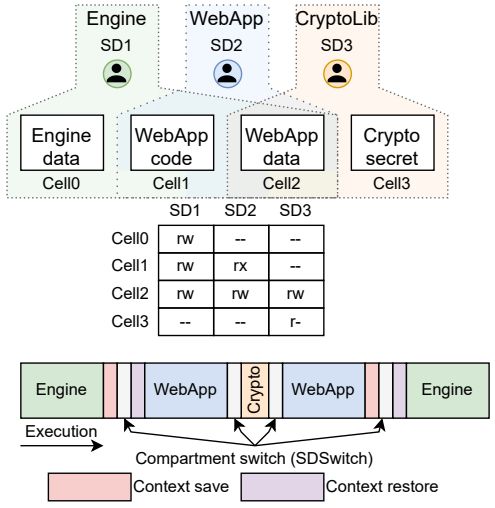

Figure 1. Browser compartmentalization with three compartments.

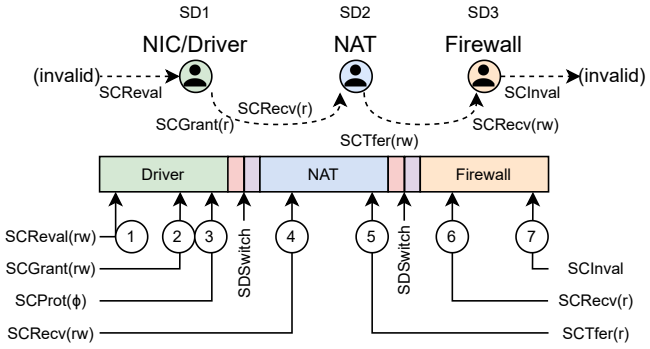

Figure 2. Permission transfers for a packet between SecureCells compartments. The figure shows the compartment executing on a core, the relevant SecureCells instructions.

pipeline has three compartmentalized stages, implementing the network card driver (Driver) which generates packets, a network address translation (NAT) stage which translates IP addresses in the header based on a translation table, and a firewall stage that implements checks on the packet headers based on a rule table. In this example, we omit further stages for simplicity. Middleboxes in datacenters and the internet [24] commonly contain virtual network functions sharing a buffer pool in uncompartmentalized dataflow pipelines. Translation and rule tables in the NAT and Firewall compartments must be isolated in private regions, protecting them from potential bugs in the Driver compartment that processes input from untrusted traffic from external sources. The programmer requires isolation of network stages for high reliability of the middlebox and low cost for passing packets between stages for line-rate packet processing, enabled by zero-copy packet flow through permission transfer.

2.1. Threat Model

Our threat model assumes an attacker who wants to compromise a compartmentalized program with multiple communicating compartments. We assume that the attacker has compromised one or more compartments, and gained the ability to both generate arbitrary code and execute it, but is restricted to the compromised compartments. The attacker wishes to compromise confidentiality, integrity, or gain code execution in other compartments. For example, the attacker might try to:

- gain permissions and directly access (load/store) another compartment's private memory,
- inject unsolicited code/data regions in another compartment's memory,
- execute unintended code in another compartment,
- create new compartments, or
- achieve any combination of the above.

The policy used for compartmentalization is assumed to be sound, and the software implementations of the modules comprising compartments are assumed to be free of bugs that can be exploited via only their exposed communication interfaces. SecureCells' trusted computing base (TCB) consists of the hardware implementation and the supervisor. Exploitable bugs in the policy or TCB can lead to a compromise irrespective of the compartmentalization mechanism. While speculative side-channel attacks are outside the scope of our threat model, we discuss SecureCells' speculative resiliency in Appendix D.

2.2. Security Objectives

A secure mechanism must enforce restrictions on a compartmentalized program, as described below.

Obj. O1a. Mechanisms must implement *access control*, validating every memory access against the policy. For the browser in Figure 1, the table holds policy-defined permissions for each compartment and memory region. Mechanisms must, for example, prevent all accesses by the compromised WebApp from reading the Engine or CryptoLib's private regions as per the policy. Mechanisms must also prevent corruption of policy-defined permissions stored in memory or registers. Intel MPK-based protection [16], for example, loads permissions from a user-controlled register when executing a `wrpkru` instruction, allowing a compartment to corrupt its own permissions.

Obj. O1b. Inter-compartment communication consisting of cross-compartment calls demand *validity checks*. Relevant validity checks include checking that *i*) the entry point is valid, *ii*) the calling compartment is allowed to call the target compartment, *iii*) the return respects the call stack, and *iv*) the passed arguments are valid. Compartment switches from the WebApp to the Engine must use valid entry points which are followed by argument-validating code. Failure to enforce this constraint enables control-flow attacks such as return-oriented programming (ROP). Vanilla Intel MPK-based protection also lacks such entry-point checks to accompany `wrpkru` instructions.

Obj. O1c. *Context isolation* accompanying a cross-compartment call is essential for protecting mutually distrusting compartments. After a cross-compartment call, the

callee compartment (for example, the Engine) must be able to fetch its context without trusting the registers which are controlled by the caller (correspondingly, the WebApp), representing an attack vector. The WebApp, for example, could try to switch to the Engine with a malicious stack pointer register, attempting to corrupt the Engine by reading from the wrong stack. CODOM [25], for example, assumes a migrating thread model and is vulnerable to attacks through an invalid register state.

Obj. O1d. Mechanisms that allow untrusted compartments to modify or transfer their permissions must prevent privilege escalation through *TCB-imposed limitations* on these operations. Specifically, compartments should only be allowed to surrender access permissions or transfer existing permissions to other compartments. A stage in the network function pipeline, for example, should not be allowed to grant write permissions for a packet to the next stage if it has read-only permissions. Transferring permissions between compartments must also be mutual, requiring explicit actions from both compartments. One-directional permission transfers studied by Lipton *et. al.* [26] allow compartments to either steal other compartments' permissions (violating confidentiality) or inject illegal data or code into other compartments (violating integrity). Linux, which allows processes to specify their own permissions when `mmap`ing shared regions, violates this objective without syscall mechanisms like SECCOMP.

Obj. O1e. *Temporary exclusive access* to otherwise shared data regions enables compartments to use data regions safely, preventing exploitation of double-fetches. With exclusive access to a packet, the Firewall stage of the network function pipeline can safely validate and use addresses in the packet header in-place (without copying), with the assurance that another corrupt stage cannot concurrently modify the packet. XPC [20] recognizes this objective, allowing exclusive access to a single region tracked by the Relay Segment register.

Obj. O1f. *Auditability*, the ability to easily determine the global access permissions, facilitates auditing compliance to a compartmentalization policy by checking which compartments have access to which memory region. A browser might regularly audit its permissions to ensure that the WebApp has not escalated its privileges. An audit for a mechanism with a centralized permissions store, such as page tables, must only check this store simplifying audits. In contrast, an audit for CHERI [27] requires an expensive, full-memory scan since the set of memory regions accessible to a compartment is the transitive closure of capabilities held in its registers, along with capabilities held in any memory region accessible through these registers.

2.3. Performance Objectives

Low-overhead checks and operations allow performance-critical programs to be compartmentalized.

Obj. O2a. *Single-cycle access verification* in the common case is essential for core throughput. While most mechanisms meet this objective in the best (not common) case, page-table based isolation mechanisms suffer from the limited scalability of Translation Lookaside Buffers (TLBs) used to cache permissions. Programs with large datasets can incur high TLB miss rates, with correspondingly high verification latency in the common case due to page-table walks. UNIX process-based protection particularly suffers from this limitation since modern Address Space ID (ASID)-tagged TLBs will effectively contain duplicate entries for a shared page with separate permission for each compartment, effectively dividing an already capacity-limited structure among compartments [28]. This objective implicitly requires the mechanism to support a sufficiently large number of compartments and data regions. A mechanism with small limits, like Intel MPK which is restricted to 16 colors for data regions, will incur overheads from software workarounds required to virtualize the corresponding resource [16].

Obj. O2b. Cross-compartment calls are essential and frequent for communication between fine-grained compartments necessitating *fast compartment switches*. Fine-grained library isolation [6] requires compartment switches accompanying every function call to an untrusted library. A program isolating short-running functions, such as AES encryption using hardware AES-NI extensions, can incur a compartment switch every tens or hundreds of cycles [29]. Specialized hardware instructions accessible from userspace are essential for cheap compartment switches in tens of cycles. Even the fastest supervisor-mediated compartment switch still costs hundreds of cycles [27].

Obj. O2c. *Fast, zero-copy permission transfer* enables programs to efficiently move data between compartments. Data copying for passing large buffers during compartment calls can overwhelm high-performance programs, such as our example network function pipeline. Such applications typically pass packets by reference between unisolated stages profiting from zero-copy. Cheap permission transfers, within ten to hundred cycles, enable such applications to be compartmentalized with performance comparable to the monolithic versions. UNIX process-based permission transfers instead involve microsecond-scale system calls, precluding their use for practical compartmentalization.

2.4. Flexibility

A mechanism demands flexibility to be suitable to compartmentalization across a variety of application domains.

Obj. O3a. For flexibility, a mechanism must support *arbitrary sharing of data regions*, requiring independent per-compartment per-region permissions. A private region, for example, should be accessible by only a single compartment. Another shared region might allow read access to one compartment, write access to another, and execute permissions to a third. Mechanisms that target hierarchical security, for example, limit flexibility — the trusted compartment implicitly has permission to access an untrusted compartment's data — and exclude wide applicability. In contrast, even if the WebApp in the browser trusts the Engine, the Engine is denied execute permissions to the WebApp's code. A

mechanism must support, but not be exclusive to, specific trust models such as nested compartments.

Obj. O3b. To scale performance overheads with security objectives, we introduce a desirable property, *security-proportionality*. A security-proportional mechanism allows policies to trade-off overheads for security when unnecessary. Despite not trusting the WebApp, transitions from the WebApp to CryptoLib can elide context switching required for register isolation under a specific condition. Verification approaches [30], [31] can be used to prove that a small function in CryptoLib does not leak the key under the assumption that entry points are enforced, and that the function's code overwrites registers used to store the key before returning to the WebApp. By using the cheaper migrating thread model [32], a security-proportional mechanism can reduce overheads where acceptable. Process-based isolation, for example, is not security-proportional since every compartment switch incurs the same non-negotiable overheads (including context switching, page-table switching, or scheduling).

2.5. Alternate Visions for Compartmentalization

SecureCells envisions a future where the mechanism supports widespread application compartmentalization efforts, with consequently differing goals and designs compared to related mechanisms. First, some mechanisms only support *custom-tailored use cases* such as differentiating between single trusted-untrusted compartments [22], [21], [33], or a binary classification of data as (in)sensitive [23]. CODOMs [25] link code addresses to compartments, restricting code sharing that is abundant in modern programming. Specialization allows simpler hardware mechanisms, but do not support a broad spectrum of applications. Second, SecureCells does not aim to compartmentalize existing software with zero-modifications. While automated isolation techniques provide a crucial first step towards compartmentalized programs [34], [35], [36], security-critical software requires refactoring to fully realize the benefits of proper compartmentalization. Finally, related works target *compatibility with legacy hardware* or existing or upcoming software/hardware mechanisms and abstractions for isolation. Numerous mechanisms try to compartmentalize using process-based isolation implemented by the OS [13], [20], [12], retrofitting compartmentalization onto an abstraction originally designed for multiprogramming on unicore processors. Others leverage Intel MPK [22], [17], [21] or similar protection-key based mechanisms [18], synergizing with traditional page table-based virtual memory. Targeting immediate adoption, Hodor [22] and LOTRx86 [19] (ab)used existing processor features intended for other purposes to isolate compartments. HAKC [37] leverages state-of-the-art ARM extensions, PAC and MTE, to compartmentalize the Linux kernel, but requires a two-level clustering of closely-connected compartments to overcome MTE's compartment scaling limitations and still incurs a significant performance hit. With the sole exception of Mondrian [14], proposals assume current page-based virtual memory. Meanwhile, trends in applications and memory architectures have led to a resurgence in range-based translations and protections among academic proposals [38], [39], [40], [41], [42] and commercial processors including AMD's Zen lineup [43]. Table 1 summarizes the objectives satisfied by related mechanisms (justification in Appendix B). We discuss related mechanisms further in Section 5.

TABLE 1. COMPARISON OF COMPARTMENTALIZATION MECHANISMS BASED ON COMPLIANCE WITH THE OBJECTIVES DESCRIBED IN SECTION 2. LIMITED COMPLIANCE IS MARKED WITH "∼".

	O1						**O2**			**O3**	
	a	b	c	d	e	f	a	b	c	a	b
UNIX	✓	✓	✓	N/A		✓				✓	
Mondrian	✓	✓	✓	N/A		✓	✓			✓	
lwC	✓	✓	✓	N/A		✓				✓	
CODOM	✓	✓		N/A		✓		✓			✓
XPC	✓	✓	✓	✓	✓	✓		✓	∼	✓	
MPK						∼		✓	✓	✓	✓
ERIM	∼	∼				∼		✓	✓	✓	✓
Donky	∼	✓		N/A		✓		✓	✓	✓	
CHERI	✓	✓	✓						✓	✓	
SecureCells	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Complementary requirements. To satisfy application requirements, programs compartmentalized with SecureCells' mechanisms require complementary properties from other parts of the system including secure compartmentalization policies, a secure and performant supervisor interface, and formal verification of application-level properties aided by programming conventions. For example, supervisors might include a syscall for microsecond-scale compartment creation [13]. Safe calling conventions can provide formal guarantees against inadvertent information leakage from the stack [44]. These investigations are outside the scope of this paper.

SecureCells overview. SecureCells is a compartmentalization mechanism designed to satisfy the above objectives across a wide array of programs, providing flexibility and performance without compromising on security. SecureCells stores permissions in a centralized permissions table accessible only by the supervisor and hardware. A novel, range-based memory management unit (MMU) and lookaside buffer design (Section 3.1) allows single-cycle access control on the fast path satisfying objectives **O1a**, **O1f**, **O2a**, and **O3a**. SecureCells introduces fast, userspace instructions for common compartmentalization operations (see Table 2): switching compartments, transferring permissions and validating exclusive access for data regions (Section 3.2). These instructions satisfy requirements **O1b**, **O1e**, **O2b**, **O2c**. SecureCells delegates context isolation, call-stack maintenance, and argument validation to software. Section 3.3 outlines how software can securely and efficiently implement context isolation and call-stack maintenance. Software implementing these functions satisfy security (**O1b**, **O1c**) and flexibility (**O3a**, **O3b**) objectives.

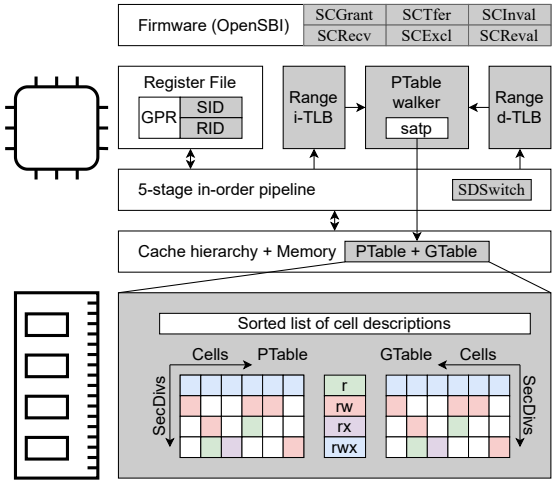

Figure 3. SecureCells' architecture, highlighting modified hw/sw in gray.

3. SecureCells

SecureCells proposes hardware-software co-design to satisfy the manifold objectives for efficient and secure compartmentalization. The key insight that *compartmentalization operations from untrusted userspace are secure with TCB-maintained permission checks* allows SecureCells to implement compartment switch and permission transfer through trusted hardware-checked userspace instructions which are hundred to thousand times faster compared to traditional supervisor calls. Pragmatically, SecureCells retains software for operations such as context switching which, while common, would not benefit significantly from hardware support. Software implementations of such operations achieve higher flexibility and resilience to implementation errors at negligible or low additional performance cost compared to a hardware implementation. For example, both hardware and software context switching can saturate the L1 data cache bandwidth, achieving similar performance. The second insight is that VMA-based permission tracking eliminates permission duplication inherent in page-table entries for pages within a VMA. SecureCells leverages this insight, eliminating overheads for permission storage (compared to equivalent page tables) and allowing hundred-times smaller core-side lookaside buffers.

SecureCells protects compartments, application-defined mutually untrusting parts of a program, by controlling their access to memory regions. Each compartment is allocated a Security Division (SD) with individual permissions to each VMA-granularity data region (cell). The Browser (Figure 1), for example, has three compartments (Engine, WebApp and CryptoLib) allocated SD_1-SD_3, and four cells. SecureCells augments each core with a read-only register ($SDID$) tracking the currently executing compartment. Along with a table for storing permissions (PTable) and a modified MMU for enforcing the permissions, SecureCells implements single-cycle access control. The WebApp SD has executable permissions to one code cell and read-write permission to one data cell. Userspace instructions (see Table 2) enable secure compartment switching and permission surrender/transfer. Another per-core read-only register (RID) tracks the caller after a compartment switch, allowing the callee to securely identify its caller. During permissions transfer, a granted permissions table (GTable) tracks outstanding permissions. Further, the design implements context isolation and secure call stacks in software leveraging the above hardware primitives. Figure 3 summarizes SecureCells's architecture.

3.1. Access control

The Permissions Table (PTable) stores per-cell, per-SD permissions for the compartmentalized program. For each cell, the permissions for each SD are independent and define the degree of sharing for that cell. Compartment-private data stores allow only one SD non-null permissions in this table. Shared cells can be readable, writable, or executable by more than one SD. For a JIT compiler, the cell holding generated data will be writable by the compiler SD, while being executable by the sandboxed code's SD. The PTable is stored in privileged supervisor memory, restricting accesses (including stores) from userspace. Figure 1 shows the permissions for the three browser compartments, assigned to separate SDs, to the four data regions, similarly assigned to cells. SecureCells' current PTable design supports a large number of SDs and cells (2^{29} and 2^{32} respectively), vastly exceeding application requirements. A finely-compartmentalized modern browser, for example, will only require a few hundred compartments, isolating each loaded shared library (around 100 on the author's Firefox installation) and per-tab rendering compartments [45].

SecureCells replaces the core's MMU with a PTable walker and range-based lookaside buffers for permissions and translations. The MMU checks access permissions based on the accessed address and the executing SD identified by the core's $SDID$ register. The lookaside buffers track a small number of frequently accessed cells, along with $SDID$-tagged permissions. In the common case, access control verifies permissions from entries in this buffer. When accesses miss in this buffer, the PTable walker reads the required permission from the PTable. The walker first performs a (fast) binary search in a sorted list of permissions to find the correct cell containing the accessed address, then reads the correct permission from the PTable for that cell. The location of the permission in the PTable is found through very simple arithmetic.

SecureCells' PTable layout and MMU design has three key advantages: fast PTable walks, scalability to large data working sets, and low silicon cost. The PTable layout aids fast permission lookups by sorting the cell descriptors, allowing a binary search for the cell descriptor containing an address, and the contiguous layout of the permissions for a particular SD, which improves spatial locality for PTable walks. Range-based lookaside buffers also enable scalability for programs with large datasets, since permissions should be verified against TLB entries in the common case. With growing dataset sizes, traditional processors require larger

TABLE 2. OVERVIEW OF SECURECELLS' USERSPACE INSTRUCTIONS.

Instruction	Purpose
SDSwitch	Switch to another SD
SCProt	Change current SD's permission to cell
SCInval	Mark a cell invalid
SCReval	Revalidate an invalid cell
SCGrant	Grant cell permissions to another SD
SCRecv	Accept granted cell permissions
SCTfer	Grant and drop cell permissions
SCExcl	Check for exclusive access to a cell

TLBs in order to track additional page translations and permissions. Importantly, all permissions for pages within a VMA are the same, leading to duplication in TLB entries' permissions. However, the growth of program datasets has exceeded the TLB reach of modern processors, leading to attempts at range-based translations (explicitly managed by the supervisor [39], or implicitly through coalescing [40]). In contrast, as dataset sizes grow, the cell count remains constant and the size of cells increases. Previous work in range-based translation caching [39], [41] have also demonstrated that processors require hundred-times smaller range-based lookaside buffers than in traditional systems, drastically reducing silicon cost. Research proposals [41], [46] have also tackled external fragmentation from range-based translations by introducing a system-wide page table after the last-level cache.

3.2. Userspace Instructions

SecureCells introduces 8 new serializing userspace instructions for accelerating common compartmentalization operations (Table 2). These instructions, formally defined in Appendix A, implement speculation-free compartment switching with checked entry points, permission surrender and transfer for zero-copy dataflow.

The SDSwitch instruction targets secure, low-overhead compartment switching within userspace. SDSwitch resembles function call instructions, with direct and indirect variants, additionally switching the core's $SDID$ register and saving the caller's $SDID$ to RID. Since the $SDID$ register is not writable from userspace, inter-compartment calls must use SDSwitch. The cost of executing an SDSwitch is essentially the cost of pipeline serialization, plus the negligible cost of updating core registers, making it extremely cheap. For an in-order 5-stage pipeline, an SDSwitch instruction completes in around 8 cycles. For an out-of-order processor, pipeline serialization is an essential cost incurred by all related mechanisms to prevent Spectre-like [47] speculative execution attacks, typically requiring 50-100 cycles. For example, serialization dominates ERIM's MPK-based 99-cycle switch latency. Compared to supervisor-controlled compartment switching, SDSwitch eliminates the cost of serialization on supervisor entry, context switches on entry and exit, syscall dispatch, scheduling, and accounting costs.

SecureCells requires SDSwitch instructions for both forward and backward edges on cross-compartment calls. We show how software can implement cheap, secure call stacks in Section 3.3. Programs are also allowed more flexibility, and can implement both remote procedure call (RPC)-like call-and-return (as in Figure 1) and circular function call graphs with one-way switches (as in Figure 2).

SDSwitch instructions impose an additional restriction over function calls in order to enforce call gates — the instruction at the target address must be an executable SDEntry instruction for the target SD. This requirement limits the valid entry points for a compartment to the executable SDEntry instructions in its code, and is conceptually similar to Intel's CET [48]. A compartment can mark valid entry points with SDEntry instructions, and implement call gates directly afterwards. Note that while our attacker can inject arbitrary code into a compromised SD, it cannot write code into any other SD, protecting uncompromised SDs from attack via code injection containing unintended entry points. The only remaining way for an attacker to propagate between compartments is by using valid interfaces. Proper input validation, which is always crucial for compartmentalized programs, protects against this attack vector. In contrast, Intel MPK-based methods allow an attacker to inject and execute a wrpkru instruction into a compromised compartment to elevate its privileges to access all memory. Further, the core executing SDSwitch updates the RID register with the caller's $SDID$, allowing the callee to identify and validate the caller.

The SCProt instruction allows a SD to update its permissions to a cell, with the restriction that the new permissions are a strict subset of existing permissions. Essentially, SCProt allows a SD to surrender permissions when no longer needed. This instruction supports a common paradigm in secure software where a program drops permissions as soon as possible.

An SCGrant-SCRecv instruction pair, executed by separate compartments, allows permissions for a cell to be transmitted between them. When the granting SD executes SCGrant, the targeted $SDID$ and permissions are stored in the GTable. A SD is only allowed to grant permissions it already has. Only the targeted SD can later accept these permissions by executing an SCRecv, specifying the SD it expects to receive permissions from. Mutual involvement in permission transfers prevents SDs from "stealing" from or "injecting" into other SDs' permissions, ensuring confidentiality and integrity respectively. Note how this prevents malicious code injection in particular, including where an attacker might try to inject new entry points (SDEntry instructions). Recognizing a common software pattern where a SD hands over its permissions to the next stage and drops its own permissions, SecureCells introduces the SCTfer instruction. Unlike SCGrant, a SD executing SCTfer also drops its own permissions to the cell involved. The semantics of SCTfer are identical to consecutive SCGrant and SCProt instructions, but SCTfer deduplicates permission checks. All data transfer instructions (SCProt, SCGrant, SCTfer, SCRecv) also flush the relevant entry from the MMU's lookaside buffer. The network function is dependent on these instructions to progressively transfer permissions to

SecureCells Program State

A set of M SDs (S), incl. SD_{sup} for the supervisor
A set of N cells (C) each of which is valid or invalid
Per-core register SID
Per-core register RID
PTable $PT : S \times C \mapsto \mathcal{P}(\{ r, w, x \})$
GTable $GT : S \times C \mapsto S \times \mathcal{P}(\{ r, w, x \})$

Instruction 1 SDSwitch($addr$, SD_{tgt})
Switch to SD_{tgt} at instruction pointer $addr$

$c_i \leftarrow cell(addr)$
assert $valid(c_i)$
assert instruction at $addr$ is $SDEntry$
assert $x \in PT(SD_{tgt}, c_i)$
instruction pointer $\leftarrow addr$
$RID \leftarrow SID$
$SID \leftarrow SD_{tgt}$

Instruction 2 SCProt($addr$, $perm$)
Restrict rights to $addr$ to $perm$

$c_i \leftarrow cell(addr)$
assert $valid(c_i)$
$p_{i,cur} \leftarrow PT(SD_{cur}, c_i)$
assert $perm \subseteq p_{i,cur}$
$PT(SD_{cur}, c_i) \leftarrow perm$

Instruction 3 SCGrant($addr$, SD_{tgt}, $perm$)
Grant SD_{tgt} $perm$ rights to $addr$

$c_i \leftarrow cell(addr)$
assert $valid(c_i) \wedge perm \neq \phi$
$p_{i,cur} \leftarrow PT(SD_{cur}, c_i)$
assert $perm \subseteq p_{i,cur}$
$GT(SD_{cur}, c_i) \leftarrow (SD_{tgt}, p_{tgt})$

Instruction 4 SCRecv($addr$, SD_{src}, $perm$)
Accept $perm$ rights to $addr$ from SD_{src}

$c_i \leftarrow cell(addr)$
assert $valid(c_i) \wedge perm \neq \phi$
$(SD_{tgt}, gp_{tgt}) \leftarrow GT(SD_{src}, c_i)$
$p_{i,cur} \leftarrow PT(SD_{cur}, c_i)$
assert $SD_{cur} = SD_{tgt} \wedge perm \subseteq gp_{tgt}$
if $perm = gp_{tgt}$ **then**
    $GT(SD_{src}, c_i) \leftarrow (SD_{inv}, \phi)$
else
    $GT(SD_{src}, c_i) \leftarrow (SD_{tgt}, gp_{tgt} - perm)$
end if
$PT(SD_{cur}, c_i) \leftarrow perm \cup p_{i,cur}$

Instruction 5 SCTfer ($addr$, SD_{tgt}, $perm$)
Transfer all $perm$ rights for $addr$ to SD_{tgt}

SCGrant($addr$, SD_{tgt}, $perm$)
SCProtect ($addr$, ϕ)

Instruction 6 SCReval($addr$, $perm$)
Re-validate address $addr$ with $perm$ rights

$c_i \leftarrow cell(addr)$
assert $invalid(c_i) \wedge perm \neq \phi$
Validate c_i
$PT(SD_{cur}, c_i) \leftarrow perm$

Instruction 7 SCInval($addr$)
Invalidate $addr$ cell

$c_i \leftarrow cell(addr)$
assert $valid(c_i)$
for all $SD_j \in S - \{ SD_{sup}, SD_{cur} \}$ **do**
    $p_{i,j} \leftarrow PT(SD_j, c_i)$
    $(SD_{tgt}, gp_{tgt}) \leftarrow GT(SD_j, c_i)$
    assert $p_{i,j} = \phi \wedge gp_{tgt} = \phi \wedge SD_{tgt} = SD_{inv}$
end for
$PT(SD_{src}, c_i) \leftarrow \phi$
$GT(SD_{cur}, c_i) \leftarrow (SD_{inv}, \phi)$
Invalidate c_i

Instruction 8 SCExcl($addr$, $perm$)
Verify exclusive $perm$ rights to $addr$

$c_i \leftarrow cell(addr)$
assert $valid(c_i) \wedge perm \neq \phi$
$p_{i,cur} \leftarrow PT(SD_{cur}, c_i)$
assert $perm \subseteq p_{i,cur}$
$(SD_{tgt}, gp_{tgt}) \leftarrow GT(SD_{cur}, c_i)$
if $perm \cap gp_{tgt} \neq \phi$ **then**
    return $False$
end if
$excl \leftarrow True$
for all $SD_j \in S - \{ SD_{sup}, SD_{cur} \}$ **do**
    $p_{i,j} \leftarrow PT(SD_j, c_i)$
    $(SD_{tgt}, gp_{tgt}) \leftarrow GT(SD_j, c_i)$
    if $perm \cap p_{i,j} \neq \phi \vee perm \cap gp_{tgt} \neq \phi$ **then**
        $excl \leftarrow False$
    end if
end for
return $excl$

Figure 4. SecureCells' state and userspace instructions.

a packet between stages, as illustrated in Figure 2. However, a SD can only have a single outstanding grant for a particular cell. If a SD grants a second set of permissions to a cell before the first set of permissions to the same cell is accepted, the first grant will be overwritten in the GTable.

The `SCInval-SCReval` instruction pair allows dataflow pipelines to optimize the end and beginning of dataflow pipelines such as the aforementioned network function pipeline. The pipeline stages progressively drop permissions to the cell holding a packet, and finally wish to drop all permissions after the final stage. However, dataflow pipelines reuse the cells to hold packets, implying that the Driver SD must find a way to regain write permission to the cell to write a new packet's contents to it. While this use case seems to require an illegal privilege escalation *prima facie*, the fact that the end of the pipeline "discarded" the cell holding the packet implies that its contents are trash, and allowing another SD escalated permissions to the cell is secure. To support such usage, SecureCells introduces the concept of validity for a cell. The `SCInval` allows a compartment to explicitly state that a cell holds trash and is available for reuse. On executing this instruction, this cell becomes unavailable for memory accesses and cannot be used by any instruction apart from `SCReval`. The `SCReval` allows any compartment to re-validate and use an invalid cell with arbitrary permissions. SecureCells imposes a key restriction in order to secure cell reuses. A SD can only invalidate a cell when it has exclusive access to it, requiring all other sharers to explicitly drop their permissions to this cell. This restriction ensures that a malicious SD cannot indirectly elevate its privilege to a shared cell by using an `SCInval-SCReval` sequence.

Exclusive access to a data region is critical to security and performance, and SecureCells introduces the `SCExcl` instruction for this purpose. Apart from enabling invalidation of a cell, exclusive access is also important for safety in concurrent programming. Concurrent access to data regions enables double fetch vulnerabilities (such as time-of-check-to-time-of-use or TOCTTOU). The `SCExcl` instruction allows a SD to check whether it has exclusive access to a cell. With exclusive access, a SD can skip making private copies of data for double fetches, improving performance. Conversely, when the policy dictates that a SD should have exclusive access to a cell, that SD can verify compliance with the policy using `SCExcl`.

3.3. Software Mechanisms

SecureCells delegates certain operations to software: argument validation for call gates, maintaining call stacks, and context switching for register context isolation. Of these, argument validation is arbitrarily variable based on the compartmentalization policy and best left for software checks in hardware-enforced call gates. SDs can determine their caller by reading the RID register, and find arguments in register or memory, and implement software checks as necessary. Software maintained call stacks for inter-compartment calls allow flexibility of calling models, simplifies hardware and

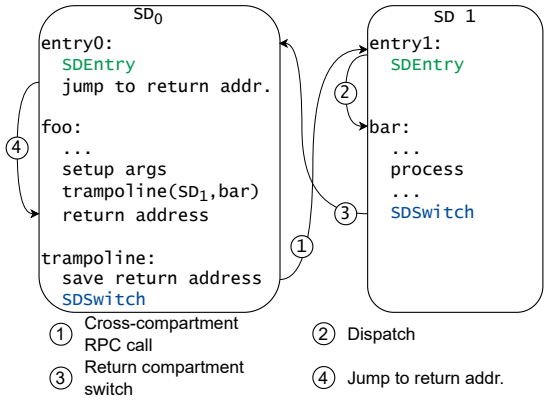

Figure 5. Cross-compartment procedure call in SecureCells.

remains secure. The software can securely restore with the same performance as hardware, making a hardware implementation unnecessary. In-software operations also improve SecureCells' security-proportionality as these operations can be skipped for lower overheads when safe to do so.

Both forward and return edges on RPC-like cross-compartment calls use `SDSwitch` instructions, as illustrated in Figure 5 where function `foo` makes a cross-compartment call to `bar`. Arguments are passed in registers. In this example, the caller uses a trampoline to hide its return address before switching to SD_1 (Step 1) and uses this address on the return path (Step 4). SD_0 is able to hide its calling address from SD_1, just leaking the address of the generic trampoline. Further, following the return switch to its entry point, SD_0 can read RID to verify that the return is indeed from the called SD, not any other. On the other side, the callee (SD_1) can store its caller and switch back to the caller's entry point on the backward edge. If SD_1 contains nested calls to other compartments, it merely needs to remember its caller somewhere in its memory. The dispatch (Step 2) secures the forward edge to `bar` with call gates. While this example is secure, the flexibility of software allows other calling patterns.

Context isolation requires the caller to save non-argument/return registers to a state store before a `SDSwitch` and restore the same state on the return edge. The second step (context restore) is challenging since it requires the SD to find its state store without trusting any register state, since the register state prior to the `SDSwitch` persists. We propose an array of per-SD private cells as state stores, indexed by $SDID$. The base of this array is easily constructed with instruction pointer-relative instructions following an entry point. Simple arithmetic involving the readable $SDID$ register allows a SD to locate its state store, and consequently restore the register state. The latency of in-software context saving to memory is limited by the core's bandwidth to the L1 cache, the same as for any potential hardware implementation. Therefore, delegating this operation to software has no performance impact. Context switching also switches the stack pointer between per-compartment private stacks.

3.4. Implementation

Our implementation of SecureCells augments and modifies the RocketChip [49] core and firmware. An overview of the implementation is shown in Figure 3, with additions to the existing processor highlighted in grey. RISC-V provides the ideal, open platform for implementing fully-functional prototypes of experimental architectures. SecureCells permits a range of implementations for single and multi-core processors containing in-order and/or out-of-order cores depending on the application's requirements: from firmware implementations on low-power embedded processors through hardware or microcode implementations on mobile, desktop, and server processors. We discuss the trade-offs in detail in Appendix E. To match the RocketChip's simple, in-order pipeline, we implement access control and compartment switching in hardware within the pipeline and emulate the remaining instructions in firmware.

SecureCells provides an alternate virtual memory mode, replacing SV-39 and SV-48. We replace the core's MMU with a range-based TLB and a PTable walker (replacing the traditional page-table walker). We design the layout of the two-dimensional tables (PTable and GTable) in memory to accelerate cell lookups and maximize spatial locality within the cache hierarchy when accessing permissions. We add $SDID$ and RID to the core's Control-Status Registers (CSRs), and implement `SDSwitch` in the core pipeline. The remaining instructions are implemented through hardware-assisted firmware by modifying OpenSBI [50].

The unified PTable-GTable in memory starts with a sorted list of cell descriptions, followed by the permissions held in the PTable, and then the mappings for the GTable. Each cell is described by the base and bound virtual addresses, the corresponding physical address base, and a single bit denoting validity. The sorted list of cell descriptions allows the PTable walker to perform a binary search when looking for the cell which contains a particular address, greatly accelerating lookups. The row-major layout of the PTable groups permissions for the same SD in contiguous cache lines, resulting in intra-cache line spatial locality for permission lookups, and synergizing well with next-line prefetchers. As a result, most MMU permission lookups are likely to be served by the L1 cache. The unified PTable-GTable together occupies $\sim 160kB$ to track permissions to 1024 cells with 64 SDs, equal to the memory used by leaf page-table entries to map 80MB of data.

The range-based lookaside buffer holds a few cell descriptions and the corresponding permissions tagged by $SDID$. The implementation of these structures is inspired by recent forays into range-based translation caches [41], [39], [38], primarily aimed at tackling the limited reach of modern page-based translation lookaside buffers (TLBs). Midgard [41] has shown that such lookaside buffers can sufficiently cover the working set of large applications with a few ($\sim$ 16) entries.

SecureCells' userspace instructions are implemented through hardware-software co-design. The `SDSwitch` instruction is implemented purely in hardware, and the remaining permission-modifying instructions are emulated through firmware. Additional hardware helpers, designed to aid operations trivially achieved in hardware but costly in software, simplify and accelerate the emulation. One notable operation is the lookup of the cell's index in the PTable, which is common for all added instructions. While a binary search in software is expensive, the MMU already holds this information. We add an instruction, only accessible in RISC-V's machine mode and similar to the `AT` instruction in ARMv8-A ISA [51], to directly query the MMU. We envision that higher performance processors with microcode sequencers can implement these instructions in microcode, and leave the investigation of the requirements of such an implementation to future work.

TABLE 3. HW CONFIGURATION OF THE SECURECELLS PROTOTYPE.

Component	Configuration	
	Baseline	SecureCells
Core	1 × Rocket, 6-stage, in-order	
L1 - D/I TLB	32-entry, fully-assoc.	16(D)/8(I)-entry, fully-assoc.
L2 TLB	1024-entry, 4-way assoc.	32-entry, fully-assoc.
L1 D/I-cache	32KB, 8-way associative	
L2 cache	16MB, 16-way associative	
Main memory	DDR3, 800MHz, 1GB	

TABLE 4. FPGA RESOURCE UTILIZATION FOR SECURECELLS' MMU

	Traditional			SecureCells		
	LUTs	FFs	SRAM	LUTs	FFs	SRAM
L1 ITLB	1915	1886	0	1529	869	0
L1 DTLB	2613	2048	0	1272	1903	0
L2 TLB + PTW	5000	3428	18KiB	3826	4596	0

4. Evaluation

In this section, we evaluate key metrics for SecureCells' security and performance. First, we show how SecureCells provides security for the Browser described in Section 2. Second, we measure the latency of the SecureCells' userspace instructions in microbenchmarks, particularly comparing compartment switching latency to related work. We finally measure SecureCells' performance for two representative workloads highlighting the effect of range-based access control and using userspace instructions for compartment switching and permissions transfer.

Testbench. We ran the security evaluation on a QEMU implementation of SecureCells, which faithfully models its functional behavior, and the performance experiments on our hardware implementation of SecureCells, which uses cycle-accurate Register-Transfer Level (RTL) simulation to accurately measure its timing behavior. The core configuration, described in Table 3, resembles ARM's Cortex-A75. Our baseline is an identical core using a traditional page-based MMU and TLBs instead of SecureCells. Table 4 shows the FPGA resource utilization for both the baseline and SecureCells MMUs. SecureCells' PTable walker contains simpler logic than the baseline, as evidenced by the fewer

LUTs required in the design. Additionally, the much smaller range TLB eliminates the 18KiB block SRAM required to store $1,024$ entries in the baseline L2 TLB. We run our benchmarks on a seL4 kernel ported to use SecureCells' memory protections. To evaluate realistic workloads on the seL4 kernel, we faithfully ported core functionality of benchmarks, carefully limiting system calls.

4.1. Security Evaluation

To evaluate SecureCells' security claims, we test that a properly compartmentalized SecureCells program prevents common attack vectors for monolithic software. We also include an in-depth analysis of SecureCells' instruction semantics and checks in Appendix A.

Access Control. We evaluate SecureCells' access control on a mock Browser, modeling the example described in Section 2. The Browser contains a simple compiler Engine that generates code for sandboxed WebApp applications. The WebApp can allocate arrays, and read/write elements in the array through get/set instructions. We emulate a buggy Engine that generates vulnerable WebApp code lacking bounds checks on array accesses, allowing the WebApp arbitrary reads and writes. With the monolithic Browser, an attacker WebApp could leak/modify the Engine's data as well as that of a second sandboxed WebApp. When compartmentalized with SecureCells with the permissions shown in Figure 1, illegal accesses by the attacker WebApp outside its data cell instead raise load/store access faults. SecureCells' access control also prevents arbitrary code injection by the WebApp by preventing the WebApp from writing to either its or the Engine's code regions.

Context Isolation and Call Gates. When uncompartmentalized, the WebApp can modify the Engine's stack enabling control- and data-flow attacks like ROP [52]. Using SecureCells for separation, inter-compartment calls between the Engine and the WebApp are protected through call gates implementing context isolation (Section 3.3) including stack switching. SecureCells successfully prevents the WebApp from accessing the Engine's stack.

4.2. Performance Microbenchmarks

First, we create microbenchmarks to measure the latency of each userspace instruction introduced by SecureCells, of which `SDSwitch` is directly implemented in hardware, and the other instructions are emulated in firmware.

In Table 5, we compare SecureCells' compartment switching cost with that of related mechanisms, particularly for a round-trip cross-compartment call. SecureCells' userspace `SDSwitch` enables 8-cycle compartment switches, with optional software context saving costs, which is more than $5\times$ faster than XPC's switch. `SDSwitch`'s latency consists of pipeline serialization (5 cycles), an instruction permission check (2 cycles) and a single cycle for the targeted `SDEntry` instruction. Of course, both XPC and SecureCells would incur higher pipeline serialization costs on an out-of-order core, putting SecureCells on par with, or better than, the MPK-based ERIM. Note that ERIM requires stringent code integrity and control-flow integrity guarantees while SecureCells does not impose any code requirements for its compartmentalization guarantees.

TABLE 5. COMPARTMENT SWITCHING COST OF VARIOUS COMPARTMENTALIZATION MECHANISMS.

	Round-trip Cycles			CPU	
	Switch	Context Saving	Total	OoO[1]	Model
lwC	2×6000		12000	✓	SkyLake
seL4	2×514		1027		RocketChip
CHERI	254[2]	129[3]	425		CHERI
ERIM	2×99	Opt[5]	198	✓	Xeon
XPC	82[4]	Opt[5]	82		RocketChip
SecureCells	2×8	Opt[5]	16		RocketChip

[1] Out-of-order CPUs incur higher pipeline serialization costs
[2] In-kernel time
[3] Userspace time (caller, libcheri)
[4] XPC call + return + TLB miss
[5] Optional, software-implemented context switch

TABLE 6. CYCLES FOR EMULATING SECURECELLS INSTRUCTIONS.

Instruction	Trap entry	Dispatch	Emulation	Total Cycles
`SCProt`		32	33	144
`SCInval`		35	68	182
`SCReval`		39	44	162
`SCRecv`	79	54	69	202
`SCGrant`		52	63	194
`SCTfer`		61	62	202
`SCExcl`		57	67	203

All instructions other than `SDSwitch` and `SDEntry` are emulated by the firmware, and therefore incur the costs of context saving, firmware entry and exit handlers, and dispatch to the correct emulation function. Table 6 shows the latency of each instruction, breaking down the cycles spent on each of the above overheads. A microcode implementation of SecureCells would allow the core to use internal registers for storage, eliminating the context switch, and directly lookup the microcode ROM to find the emulation microcode, eliminating dispatch. Consequently, a microcode-based implementation would reduce SecureCells' cost to that of the core emulation code only.

4.3. Compartment Switching and Access Control

To evaluate SecureCells' practical performance, we create a simplified benchmark representative of a popular server workload, `memcached`, accurately modelling the workload's memory access patterns across varying dataset sizes. Our benchmark implements the core hashtable-based storage and the common query path loaded by an in-process load generator function and omits system call-dependent features (networking, dynamic resizing), and the global LRU list. The benchmark isolates the data store from the vulnerable external interface — attackers might send malformed requests to trick the interface into directly accessing the data

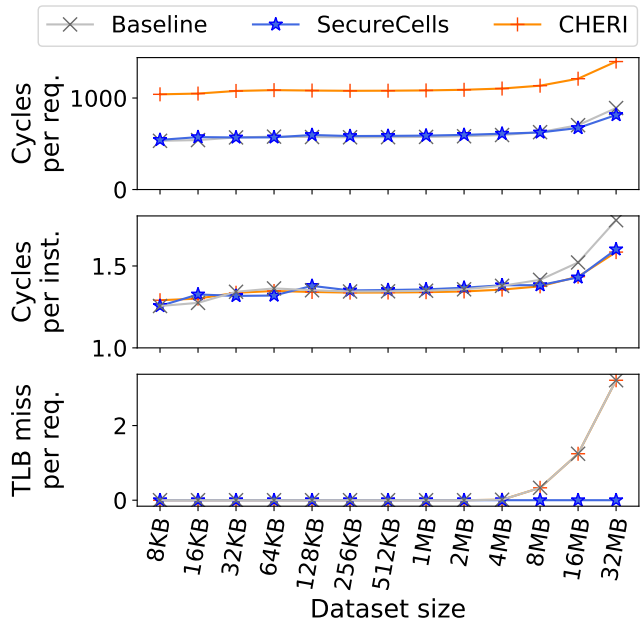

Figure 6. Comparison of cycles-per-request, cycles-per-instruction (CPI), and TLB miss rate while executing compartmentalized `memcached` benchmark on SecureCells, compared to the uncompartmentalized version on RocketChip (lower is better).

store — by assigning them to separate SDs. The interface deserializes incoming requests, queries the data store by switching compartments using `SDSwitch`, and serializes the outgoing response. For simplicity, this benchmark utilizes the migrating thread model.

Compartmentalizing the server allows us to measure the overheads of frequent compartment switches, while varying the program's dataset size allows us to compare SecureCells' scalability. We scale the dataset size by sweeping the number of fixed-size (64B) entries stored in the data store, all of which are accessed randomly by the load generator. We compare the compartmentalized server running on SecureCells' implementation to an uncompartmentalized server running on an unmodified RocketChip core by measuring the average count of instructions retired and cycles used to process each request. To compare against another emerging compartmentalization architecture, we also conservatively model CHERI's performance on this benchmark, adding the costs of supervisor-mediated compartment switches with hardware support, as reported in the paper [27]. We model each compartment switch as 191 instructions requiring 254 cycles, excluding the costs of context switching and ignoring other microarchitectural overheads. In Figure 6, we plot the average per-request cycle count and the cycles-per-instruction (CPI) for the server.

SecureCells implements fast compartment switching, and the cost of switching to and from the data store compartment for each request (16 cycles) is minuscule ($< 3\%$) compared to the request processing time (minimum 532 cycles). Consequently, SecureCells' performance closely

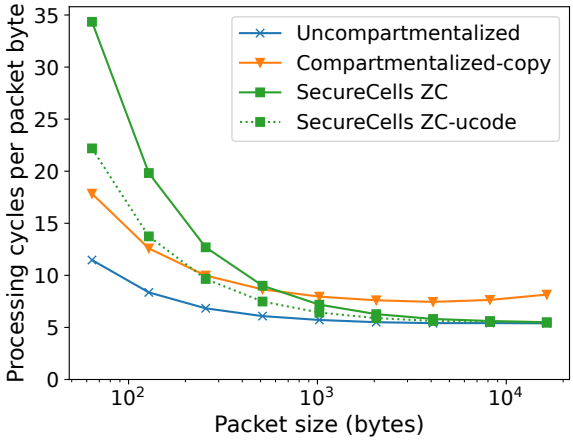

Figure 7. Packet processing cycles-per-byte comparison.

tracks that of the baseline even for small dataset sizes. In contrast, CHERI's compartment switching overwhelms the request processing time, only approaching the baseline's performance for large dataset sizes. While CHERI's performance for compartmentalization compares favorably to that of traditional OS-based isolation techniques, it offers unacceptable overheads for finer, function-granularity compartmentalization (up to 95.5%).

The CPI graph highlights the baseline system's limited TLB reach. As the dataset exceeds the TLB reach of 4MB, the baseline starts to encounter TLB misses on accesses to the data store. Consequently, the baseline CPI starts to degrade compared to SecureCells, and only worsens as the dataset increases past the CPU's last-level cache capacity. In contrast, SecureCells' range-based lookaside buffer comfortably scales to large datasets, allowing the `memcached` server to serve requests 9.3% faster for a 32MB dataset.

4.4. Compartmentalized pipeline

To illustrate SecureCells' zero-copy permission transfer performance, we implement the virtual network function pipeline presented in Figure 2. The Driver stage generates a "packet" by writing a UDP/IP packet of varying length into a packet buffer, whereas the Firewall and NAT read and modify the IP and UDP headers respectively, but ignore the packet's payload.

Representing the ideal performance target, we include the "uncompartmentalized" configuration that passes the packet by reference, incurring no overheads for data transfer. The second configuration, "compartmentalized-copy", compartmentalizes pipeline stages and uses shared buffers to transfer packets by copy. The third, zero-copy "SecureCells ZC" configuration isolates stages, and uses userspace instructions to transfer access permissions to packets, each of which occupies a different cell. Finally, the "SecureCells ZC-μcode" configuration models the possible performance of a microcode implementation of SecureCells' dataflow instructions by mitigating trapping overheads to the firmware

and dispatch. This model is conservative, ignoring possible optimizations from parallelizing checks in microcode.

In Figure 7, we plot the average number of cycles required by the benchmark to process a byte of a packet as the packet size grows. Fixed costs, such as a function call, compartment switch or permission transfer, have diminishing impacts as the packet size grows. The costs for generating and copying the packet, however, grows linearly with packet size, and add a constant vertical offset in the graph. The "compartmentalized-copy" configuration incurs additional costs over the uncompartmentalized baseline due to compartment switches (4.4% for small packets) and packet copy (51.1%). The "SecureCells ZC" configuration trades-off linearly-growing packet copying costs with fixed-cost permission transfers and (in)validations. While the $\sim$ 250-cycle average latency of SecureCells' permission-modifying instructions causes a massive 199% overhead for the smallest packets, this fixed cost quickly gets amortized for larger packets. Indeed, this configuration overtakes the "compartmentalized-copy" configuration for 600B packets and above, and approaches the performance of the uncompartmentalized configuration (2.0% overhead) for 16kB packets. Finally, the "SecureCells ZC-μcode" configuration highlights SecureCells' performance potential, with (average) 69-cycle operations for transferring permissions which lowers the break-even threshold to 200B packets.

5. Related Work

A variety of compartmentalization techniques exist, both in software and leveraging hardware, targeting differing goals and with consequently different designs.

Attacks often target specific, sensitive data for leakage or corruption (e.g., keys or flags). Consequently, various proposals such as IMIX [23], ERIM [17], and MemSentry [21] introduced mechanisms to specifically protect such data from untrusted or unsafe code. However, these mechanisms fail to apply to more generic scenarios, with more than two compartments, per-compartment sensitive or private data, and non-hierarchical trust models. COde-centric memory DOMains [25] proposed an architecture where the instruction pointer identifies the running compartment, in a bid to isolate untrusted libraries. However, this proposal is unable to support the extensive code sharing in modern programs, including shared libraries like `libc`.

Compatibility with existing systems brings immediate security benefits. By mapping the same physical pages across separate per-compartment page tables with different permissions, the existing virtual memory implementation can mimic intra-address space compartmentalization. Typically, such mechanisms require costly supervisor intervention to switch compartments limiting the temporal granularity of compartmentalization. SMV [28] introduced an API for creating intra-address space memory views, but relied on the supervisor for compartment transitions. LightWeight Contexts (lwC) [13] proposed a new OS abstraction enabling a fast-path in the supervisor for compartment switching, essentially eliminating overheads from unnecessary tasks such as scheduling. lwC successfully reduces the cost of a compartment switch from 4 to 2μs, but remains an order of magnitude away from nanosecond-scale switching. Hodor [22] uses the VMFUNC instruction, introduced for virtual machines, to instead switch page tables in a few hundred cycles, eliminating supervisor overheads but consequently inherits the additional costs of two-dimensional page table walks. LOTRx86 [19] repurposed unused x86 rings to introduce a privileged userspace for storing sensitive data. XPC [20] prioritized software compatibility, choosing to accelerate the remote-procedure call (RPC) interface used for process-based compartmentalization with new userspace instructions. To achieve this goal, XPC cores track a complicated system of metadata across the cores and memory, storing a list of compartments, entry points, valid caller-callee pairs, and a caller stack. XPC is secure, performant, and can allow exclusive access to a single data memory range at almost zero cost. However, XPC requires additional caches for dedicated storage of its metadata, does not allow permissions to be transferred, and requires hardware to implement features cheaply implementable in software (e.g., call stacks), and cannot support non-RPC like compartment switches. With page table-based virtual memory, such proposals all inevitably suffer from the scalability limitations of modern TLBs [40], [39], [38].

Existing architectures have introduced features for intra-address space isolation, e.g., Intel's MPK and ARM's MTE extensions, with fast compartment switching (< 100 cycles) in the common case. These extensions enforce additional permissions, but are insecure under stronger threat models due to designs which prioritize compatibility with existing processors. MPK, for example, is defeated by arbitrary code injection. ERIM [17] requires complicated code scanning to prevent code injection, and Donky [18] requires hardware modifications to introduce an additional trusted privilege level within userspace. Since neither ERIM nor Donky validates code accesses, an attacker targeting cross-compartment code injection need not make the malicious code executable for the target before tricking the target into executing this code. Memory keys also architecturally limit the number of memory regions for which permissions can be efficiently tracked, leaving no room for future microarchitectural advances to improve code performance. These systems also inherit the TLB-reach issues of modern TLBs.

Range-based permission tracking tackling the TLB-reach issue appeared in Mondrian Memory Protection (MMP) [14]. MMP proposed a virtual memory architecture tracking segment-based permissions for compartments within an address space, simulating zero-copy for networking through redundant mappings for packet buffers with different, static permissions. MMP only implements access permission checks in hardware, delegating other operations, including compartment switches, to the supervisor, precluding high-performance applications. MMP also uses different permissions tables for each compartment, reading duplicated range boundaries on each switch.

CHERI refers to hardware-enforced memory capabili-

ties [53], and an eponymous compartmentalization mechanism reusing the same capabilities [27]. The original proposal for memory capabilities offers a practical mechanism to mitigate spatial safety bugs, restricting the ability of pointers to access memory beyond bounds. We recognize that CHERI's capabilities can prevent memory corruption within a compartment, motivating integration with SecureCells to together improve security. CHERI compartmentalization encapsulates capabilities to a compartment's code and data, relying on costly supervisor-mediated compartment switches. CHERI lacks auditability since capabilities are spread throughout memory, and a bug resulting in a capability being leaked cannot be cheaply detected and fixed. CHERI's switching costs are not security-proportional, lacking the ability to skip context switching costs when acceptable. Finally, CHERI's permissions are built on traditional page-based translations, and inherit TLB limitations. Nonetheless, CHERI allows more granular per-object capabilities as compared to SecureCells' per-VMA permissions.

Along with mechanisms, policy research is equally important. Researchers have attempted to formalize a compartment program's guarantees [54], determine the scope of access following permission transfers under the take-grant model [26], automatically infer isolation policies from programs [34], [35], [36], provide hints to programmers on isolation boundaries based on automated analysis [55], and reason about what guarantees remain when one or more compartments are compromised [56].

6. Discussion

Legacy program/OS support. SecureCells is compatible with existing pre-emptive operating systems which already separate architecture-specific memory management code. SecureCells also supports page-based memory management (demand paging, swapping) when integrated with upcoming intermediate-address space memory architectures [46], [41]. Since SecureCells preserves the VMA-based view of virtual memory, an OS can present a legacy userspace environment for existing monolithic applications by allocating a single compartment in the PTable. Legacy applications will also benefit from SecureCells' improved TLB-reach with range-based address translations.

Adopting SecureCells. SecureCells faces the daunting task of changes across the software and hardware stack. Nonetheless, library and compiler support for software development can greatly aid developer adoption. We developed a prototype library (`scthreads`) to support compartments with isolated contexts, and envision that most software can be ported through compilation with a SecureCells-compatible C/C++ library. We compartmentalized the example Browser (∼1kLoC), initially developed and tested on an `x86` machine, in approximately two additional days. Software such as browsers desiring the full benefits of compartmentalization will still require rewriting (to refactor monolithic code into compartments). SecureCells' userspace instructions map to common compartmentalized applications' operations, evidenced by strong parallels between SecureCells' instructions and APIs in related mechanisms or language-level operations in compartmentalization frameworks (Table 7). This mapping will simplify porting existing compartmentalized applications, such as Nginx-lwC [13], to run on SecureCells by replacing existing operations with the SecureCells equivalent (e.g., substitute `SDSwitch` in place of `lwSwitch`). Existing software compartmentalization libraries and compilers [28] can also use SecureCells as a backing mechanism. For example, consider a SecureCells backend for the LitterBox sandbox, used by the compartmentalizing compiler Enclosures [6] to isolate untrusted Go libraries, improving performance and security over the existing Intel VT-x and MPK backends respectively. Enclosure switching (`Prolog` and `Epilog`) map to `SDSwitch` instructions whereas data movement (`transfer`) maps to a `SCTfer-SCRecv` pair.

TABLE 7. MAPPING SECURECELLS INSTRUCTIONS TO RELATED MECHANISMS, LIBRARIES AND LANGUAGE FEATURES.

Instruction	Analogous API
`SDSwitch`	`dcall` [3], `CCall/CReturn` [4], `Prolog/Epilog` [5], `lwSwitch` [6]
`SCProt`	`mprotect`[1], `mpk_mprotect` [2], `dk_mprotect` [3], `CAndPerm` [4]
`SCInval`	`munmap`[1], `mpk_free` [2], `dk_munmap` [3]
`SCReval`	`mmap`[1], `mpk_mmap` [2], `dk_mmap` [3]
`SCGrant`	`mmap(MAP_PRIVATE)` [1],
`SCRecv`	`dk_domain_assign_key` [3], `Transfer` [5],
`SCTfer`	`lwOverlay` [6]

[1] Linux processes
[2] `libmpk` [16]
[3] Donky [18]
[4] CHERI [27], [53]
[5] Enclosures [6]
[6] lwC [13]

System call semantics with SecureCells. Recent work [57] has demonstrated that the Linux system call interface can be used to compromise userspace compartmentalization. Modifications of the syscall interface, such as those proposed in Jenny [58], are orthogonal to the compartmentalization mechanism and can be applied to SecureCells. We leave a systematic evaluation of kernel performance and system call semantics with SecureCells to future work.

Advantages for microkernels and system calls. Fast compartmentalization is the key objective for practical microkernel operating systems. By running the OS kernel and drivers in SDs, SecureCells improves over a modern microkernel's switching time by two orders of magnitude. Similarly, userspace programs can benefit from significantly faster system calls if the kernel is assigned a compartment within each program's address space. Essentially, the costly system call entrances can be replaced by cheaper `SDSwitch` instructions into the kernel.

Speculative-execution attacks (SEA). SecureCells does not mitigate existing SEA, but takes care not to introduce vulnerabilities (see Appendix D). Nonetheless, SecureCells' access control limits the leakage scope of Spectre attacks to a compartment's accessible cells, weakening SEA.

7. Conclusion

Compartmentalization requires labor-intensive code restructuring, deterring developers from adopting piecemeal solutions which provide partial protection or which cripple performance. This paper introduces SecureCells, a secure, flexible and performance-focused compartmentalization architecture to underpin future software compartmentalization efforts. Further work is required, for scaling our FPGA prototype to an out-of-order, multicore processor, investigating implementations of higher-level abstractions on SecureCells' mechanisms, developing software conventions to develop correctly compartmentalized programs for SecureCells, and to improve OS support for the architecture.

Nevertheless, SecureCells enables practical, effective, and efficient compartmentalization by tackling the core architectural requirements for a mechanism. SecureCells strictly enforces access controls and protects permissions from corruption, while supporting secure 8-cycle compartment switching. SecureCells constrains inter-compartment control flow to respect call gates, protecting these interfaces from fault propagation. SecureCells is also an enabler for data processing pipelines with userspace zero-copy data transfers. SecureCells remains flexible, eschewing policy-specific specializations. We have published the SecureCells prototype, benchmarks and supporting infrastructure at https://www.hexhive.epfl.ch/securecells.

8. Acknowledgements

We thank the anonymous reviewers for their insightful feedback and guidance through the major revision process. This project was partially supported by ONR IOT-D grant 13000660-052, FNS grant 200020B_188696, Fondation Botnar, an IBM PhD Fellowship, a Qualcomm Innovation Fellowship, and a gift from Intel Corporation. Any findings are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975. [Online]. Available: https://doi.org/10.1109/PROC.1975.9939

[2] Z. Bloom, "Cloud computing without containers," https://blog.cloudflare.com/cloud-computing-without-containers/, 2018.

[3] S. Shillaker and P. R. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, A. Gavrilovska and E. Zadok, Eds. USENIX Association, 2020, pp. 419–433. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shillaker

[4] L. Miller, P. Mérindol, A. Gallais, and C. Pelsser, "Towards secure and leak-free workflows using microservice isolation," in *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, 2021, pp. 1–5.

[5] "Project fission," 2021, accessed: Nov 2021. [Online]. Available: https://wiki.mozilla.org/Project_Fission

[6] A. Ghosn, M. Kogias, M. Payer, J. R. Larus, and E. Bugnion, "Enclosure: language-based restriction of untrusted libraries," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds. ACM, 2021, pp. 255–267. [Online]. Available: https://doi.org/10.1145/3445814.3446728

[7] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser, "CHORUS distributed operating system," *Comput. Syst.*, vol. 1, no. 4, pp. 305–370, 1988.

[8] D. Hildebrand, "An architectural overview of QNX," in *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures, Seattle, WA, USA, 27-28 April 1992s*. USENIX, 1992, pp. 113–126.

[9] R. Levin, E. S. Cohen, W. M. Corwin, F. J. Pollack, and W. A. Wulf, "Policy/mechanism separation in HYDRA," in *Proceedings of the Fifth Symposium on Operating System Principles, SOSP 1975, The University of Texas at Austin, Austin, Texas, USA, November 19-21, 1975*. ACM, 1975, pp. 132–140. [Online]. Available: https://doi.org/10.1145/800213.806531

[10] D. B. Golub, R. W. Dean, A. Forin, and R. F. Rashid, "UNIX as an application program," in *Proceedings of the Usenix Summer 1990 Technical Conference, Anaheim, California, USA, June 1990*. USENIX Association, 1990, pp. 87–95.

[11] N. V. D. (NVD), "Cve-2021-44228," https://www.cve.org/CVERecord?id=CVE-2021-44228, 2021.

[12] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. ACM, 2009, pp. 207–220. [Online]. Available: https://doi.org/10.1145/1629575.1629596

[13] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel, "Light-weight contexts: An os abstraction for safety and performance," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 2016, pp. 49–64. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton

[14] E. Witchel, J. Cates, and K. Asanovic, "Mondrian memory protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), San Jose, California, USA, October 5-9, 2002*. ACM Press, 2002, pp. 304–316. [Online]. Available: https://doi.org/10.1145/605397.605429

[15] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volumes 1-4*, vol. 2, no. 11, 2011.

[16] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel mpk)," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 2019, pp. 241–254. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/park-soyeon

[17] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "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*. USENIX Association, 2019, pp. 1221–1238. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[18] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain keys - efficient in-process isolation for RISC-V and x86," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. USENIX Association, 2020, pp. 1677–1694. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel

[19] H. Lee, C. Song, and B. B. Kang, "Lord of the x86 rings: A portable user mode privilege separation architecture on x86," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. ACM, 2018, pp. 1441–1454. [Online]. Available: https://doi.org/10.1145/3243734.3243748

[20] D. Du, Z. Hua, Y. Xia, B. Zang, and H. Chen, "XPC: architectural support for secure and efficient cross process call," in *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, 2019, pp. 671–684. [Online]. Available: https://doi.org/10.1145/3307650.3322218

[21] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, G. Alonso, R. Bianchini, and M. Vukolic, Eds. ACM, 2017, pp. 437–452. [Online]. Available: https://doi.org/10.1145/3064176.3064217

[22] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 2019, pp. 489–504. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[23] T. Frassetto, P. Jauernig, C. Liebchen, and A. Sadeghi, "IMIX: in-process memory isolation extension," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. USENIX Association, 2018, pp. 83–97. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/frassetto

[24] J. Martins, M. Ahmed, C. Raiciu, V. A. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, R. Mahajan and I. Stoica, Eds. USENIX Association, 2014, pp. 459–473. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins

[25] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero, "Codoms: Protecting software with code-centric memory domains," in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 2014, pp. 469–480. [Online]. Available: https://doi.org/10.1109/ISCA.2014.6853202

[26] R. J. Lipton and L. Snyder, "A linear time algorithm for deciding subject security," *J. ACM*, vol. 24, no. 3, pp. 455–464, 1977. [Online]. Available: https://doi.org/10.1145/322017.322025

[27] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. M. Norton, M. Roe, S. D. Son, and M. Vadera, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 20–37. [Online]. Available: https://doi.org/10.1109/SP.2015.9

[28] T. C. Hsu, K. J. Hoffman, P. Eugster, and M. Payer, "Enforcing least privilege memory views for multithreaded applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 393–405. [Online]. Available: https://doi.org/10.1145/2976749.2978327

[29] E. G. AbdAllah, Y. R. Kuang, and C. Huang, "Advanced encryption standard new instructions (aes-ni) analysis: Security, performance, and power consumption," in *Proceedings of the 12th International Conference on Computer and Automation Engineering*, 2020.

[30] M. Kolosick, S. Narayan, E. Johnson, C. Watt, M. LeMay, D. Garg, R. Jhala, and D. Stefan, "Isolation without taxation: near-zero-cost transitions for webassembly and SFI," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–30, 2022. [Online]. Available: https://doi.org/10.1145/3498688

[31] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: Fine-grained execution units with private memory," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016, pp. 56–71. [Online]. Available: https://doi.org/10.1109/SP.2016.12

[32] B. Ford and J. Lepreau, "Evolving mach 3.0 to A migrating thread model," in *USENIX Winter 1994 Technical Conference, San Francisco, California, USA, January 17-21, 1994, Conference Proceedings*. USENIX Association, 1994, pp. 97–114. [Online]. Available: https://www.usenix.org/conference/usenix-winter-1994-technical-conference/evolving-mach-30-migrating-thread-model

[33] D. Kilpatrick, "Privman: A library for partitioning applications," in *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*. USENIX, 2003, pp. 273–284. [Online]. Available: http://www.usenix.org/events/usenix03/tech/freenix03/kilpatrick.html

[34] N. Roessler, L. Atayde, I. Palmer, D. P. McKee, J. Pandey, V. P. Kemerlis, M. Payer, A. Bates, J. M. Smith, A. DeHon, and N. Dautenhahn, "μscope: A methodology for analyzing least-privilege compartmentalization in large software artifacts," in *RAID '21: 24th International Symposium on Research in Attacks, Intrusions and Defenses, San Sebastian, Spain, October 6-8, 2021*, L. Bilge and T. Dumitras, Eds. ACM, 2021, pp. 296–311. [Online]. Available: https://doi.org/10.1145/3471621.3471839

[35] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, and M. Franz, "Pkru-safe: automatically locking down the heap between safe and unsafe languages," in *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, Y. Bromberg, A. Kermarrec, and C. Kozyrakis, Eds. ACM, 2022, pp. 132–148. [Online]. Available: https://doi.org/10.1145/3492321.3519582

[36] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, "Towards fine-grained, automated application compartmentalization," in *Proceedings of the 9th Workshop on Programming Languages and Operating Systems, Shanghai, China, October 28, 2017*, J. Lawall, Ed. ACM, 2017, pp. 43–50. [Online]. Available: https://doi.org/10.1145/3144555.3144563

[37] D. McKee, Y. Giannaris, C. O. Perez, H. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing kernel hacks with hakc," in *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, vol. 22, 2022, pp. 1–17.

[38] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*. ACM, 2013, pp. 237–248. [Online]. Available: https://doi.org/10.1145/2485922.2485943

[39] Z. Yan, D. Lustig, D. W. Nellans, and A. Bhattacharjee, "Translation ranger: operating system support for contiguity-aware tlbs," in *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, 2019, pp. 698–710. [Online]. Available: https://doi.org/10.1145/3307650.3322223

[40] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012*. IEEE Computer Society, 2012, pp. 258–269. [Online]. Available: https://doi.org/10.1109/MICRO.2012.32

[41] S. Gupta, A. Bhattacharyya, Y. Oh, A. Bhattacharjee, B. Falsafi, and M. Payer, "Rebooting virtual memory with midgard," in *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 2021, pp. 512–525. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00047

[42] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, "Redundant memory mappings for fast access to large memories," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*. ACM, 2015, pp. 66–78. [Online]. Available: https://doi.org/10.1145/2749469.2749471

[43] A. Bhattacharjee, "Preserving the virtual memory abstraction," https://www.sigarch.org/preserving-the-virtual-memory-abstraction/, May 2017.

[44] L. Skorstengaard, D. Devriese, and L. Birkedal, "Reasoning about a machine with local capabilities: Provably safe stack and return pointer management," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 1, pp. 5:1–5:53, 2020. [Online]. Available: https://doi.org/10.1145/3363519

[45] A. Barth, C. Jackson, C. Reis, T. Team *et al.*, "The security architecture of the chromium browser," in *Technical report*. Stanford University, 2008.

[46] L. Zhang, E. Speight, R. Rajamony, and J. Lin, "Enigma: architectural and operating system support for reducing the impact of address translation," in *Proceedings of the 24th International Conference on Supercomputing, 2010, Tsukuba, Ibaraki, Japan, June 2-4, 2010*. ACM, 2010, pp. 159–168. [Online]. Available: https://doi.org/10.1145/1810085.1810109

[47] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 1–19. [Online]. Available: https://doi.org/10.1109/SP.2019.00002

[48] Intel Corporation, "Control-flow enforcement technology specification," https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, May 2019, accessed: 2020-03.

[49] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[50] riscv-software src, "Risc-v open source supervisor binary interface," 2022. [Online]. Available: https://github.com/riscv-software-src/opensbi

[51] A. L. (or its affiliates), "Arm armv8-a architecture registers," 2022. [Online]. Available: https://developer.arm.com/documentation/ddi0595/2021-12/AArch64-Instructions/AT-S12E0R--Address-Translate-Stages-1-and-2-EL0-Read

[52] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. ACM, 2007, pp. 552–561. [Online]. Available: https://doi.org/10.1145/1315245.1315313

[53] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. M. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 2014, pp. 457–468. [Online]. Available: https://doi.org/10.1109/ISCA.2014.6853201

[54] Y. Juglaret, C. Hritcu, A. A. de Amorim, B. Eng, and B. C. Pierce, "Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation," in *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 2016, pp. 45–60. [Online]. Available: https://doi.org/10.1109/CSF.2016.11

[55] K. Gudka, R. N. M. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson, "Clean application compartmentalization with SOAAP," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, I. Ray, N. Li, and C. Kruegel, Eds. ACM, 2015, pp. 1016–1031. [Online]. Available: https://doi.org/10.1145/2810103.2813611

[56] C. Abate, A. A. de Amorim, R. Blanco, A. N. Evans, G. Fachini, C. Hritcu, T. Laurent, B. C. Pierce, M. Stronati, and A. Tolmach, "When good components go bad: Formally secure compilation despite dynamic compromise," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. ACM, 2018, pp. 1351–1368. [Online]. Available: https://doi.org/10.1145/3243734.3243745

[57] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "PKU pitfalls: Attacks on pku-based memory isolation systems," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. USENIX Association, 2020, pp. 1409–1426. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/connor

[58] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, "Jenny: Securing syscalls for pku-based memory isolation systems," in *USENIX Security Symposium*, 2022.

[59] Q. Li, D. Hartley, and B. Rosenberg, "An introduction to access control on qualcomm snapdragon platforms," https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/whitepaper_0.pdf, 2020.

[60] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "PACMAN: attacking ARM pointer authentication with speculative execution," in *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, V. Salapura, M. Zahran, F. Chong, and L. Tang, Eds. ACM, 2022, pp. 685–698. [Online]. Available: https://doi.org/10.1145/3470496.3527429

[61] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium*, 2018.

[62] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: preventing speculative execution attacks at their source," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, pp. 572–586. [Online]. Available: https://doi.org/10.1145/3352460.3358306

Appendix A.
Formal Description and Analysis of SecureCells' Userspace Instructions

We define the semantics of SecureCells' unprivileged instructions in Figure 4 and discuss their corresponding security checks below.

`SDSwitch`. This instruction checks that the jump target is valid, and holds an `SDEntry` instruction executable by the target SD. With the precondition that the caller SD does not have writable permission to any cell executable by the target SD, `SDSwitch` guarantees compartment entry at previously defined entry points (helping implement call gates).

`SCProt`. This instruction checks that the target cell is accessible by the SD, and the new permissions are a subset of the existing permissions. After this instruction, the SD is assured to have no more permissions than before.

SCGrant, SCRecv and SCTfer. SCGrant checks that the granting SD has permissions to the cell, and that the granted permissions are a subset of its existing permissions. SCRecv, in turn, checks that the SD is receiving permissions for a valid cell, that the permissions were previously granted by the specific SD that the receiving SD expects, and that the received permissions are a subset of the permissions granted. SCTfer includes the checks of both SCGrant and SCProt. The granting and receiving SDs must cooperate in order to transfer permissions, and together finish with the same or fewer permissions than they began with.

A correct compartment is defined to not grant or receive any permissions or invalidate cells that it is not required to grant as per a correct compartmentalization policy. Considering a set of compromised attacker SDs and their permissions to cells and assuming that uncompromised compartments are correct, SecureCells guarantees that the attackers can neither gain any new permissions through any sequence of permission transfer instructions nor elevate the permissions of any uncompromised compartment. Using SCGrant and SCRecv instructions, the compromised compartments can transfer permissions between themselves but those grants cannot include permissions which none of the attackers had initially. The only way for the attackers to gain permissions is from an uncompromised SD either granting permissions to a cell or from invalidating a private cell which one of the attackers can validate with SCReval. The only way for the attackers to inject permissions is to have an uncompromised SD receive them. By definition, uncompromised compartments will do neither of the above. Once again, we stress on the importance of a correct compartmentalization policy. No mechanism, including SecureCells, can protect against an insecure policy where compartments transfer permissions from/to untrusted compartments without proper validation.

SCInval. This instruction allows a SD to invalidate a cell to which it has exclusive access, and to which no outstanding permission grants exist. The first condition can be true for a private region, or for one which other SDs have willingly dropped permissions. Consequently, no other SD will unwittingly lose permissions to the invalidated cell as a consequence of SCInval. The second condition provides the assurance that no compartment can regain permissions to the cell without executing SCReval.

SCReval. This instruction checks that the address corresponds to an existing cell and that it is currently invalid. Due to the initial invalidity of the cell, no SDs could have access to the cell to be revalidated.

SCExcl. This instruction does not modify any permissions, only allowing a SD to check if it has exclusive access to a cell to which it already has access to.

Appendix B. Justification for Table 1

Obj. O1a. MPK, ERIM and Donky do not check permissions for instruction fetches, simplifying code injection. Under our threat model, an attacker can inject wrpkru instructions to corrupt permissions.

Obj. O1b. Through code injection, call gates in MPK and ERIM can be bypassed.

Obj. O1c. CODOM requires migrating threads without context isolation. MPK, ERIM and Donky rely on call gates if context isolation is desired. However, MPK and ERIM cannot enforce call gates under our threat model. Donky gives no mechanism for a compartment to restore its state without trusting general-purpose registers. Further, Donky cannot adopt a SecureCells-like software approach because a compartment has no way to identify itself.

Obj. O1d. CHERI allows one compartment to unilaterally send a capability to another compartment, unchecked by the TCB and unacknowledged by the receiver.

Obj. O1e. No mechanism except XPC considers the challenge of exclusive access.

Obj. O1f. A compartment in MPK and ERIM cannot check the value of the pkru register for another compartment, hindering audits. Cross-core pkru reads are not possible. CHERI requires an expensive full memory scan for capabilities to perform an audit.

Obj. O2a. Page-table based translation and permission checking encounter TLB-reach limits leading to multi-cycle common case access verification for many widely-used programs including memcached. The mechanisms relying on such page tables for either translation or permission checking fail this requirement.

Obj. O2b. Supervisor-mediated cross-compartment calls in UNIX-like OSs, Mondrian, lwC and CHERI require 100s or 1000s of cycles to complete.

Obj. O2c. Supervisor-mediated permission transfers are slow (UNIX, MMP, lwC). MMP proposes the use of redundant mappings with different permissions to implement a form of zero-copy transfer which is not generic. CODOM does not really support permission transfers. XPC restricts permission transfer to a single relay segment.

Obj. O3a. CODOM identifies the executing compartment by the instruction pointer, limiting the flexibility to share code/data regions between compartments.

Obj. O3b. UNIX, MMP, lwC, XPC and CHERI cannot eliminate context switching when a permissive policy allows migrating threading between compartments.

Appendix C. Existing mechanisms with SecureCells

Many existing performance or security mechanisms can be integrated with SecureCells, either unmodified or with modifications described in this section.

Physical Memory Protections. SecureCells enforces permissions on the virtual address space, and is therefore trivially compatible with physical memory protection schemes including RISC-V's Physical Memory Protection (PMP) mechanism, processor reserved memory for Intel's SGX and vendor-specific protections like Qualcomm's XPU [59]. These mechanisms will apply to the physical

address output by SecureCells' MMU after PTable access control checks.

Pointer authentication and capabilities. ARM's pointer authentication code (PAC) feature and CHERI's capabilities improve memory safety by protecting pointers from illegal modifications (overwriting when stored in memory and out-of-bound increment respectively). Both mechanisms are orthogonal to, and can integrate with SecureCells, which checks accessess against PTable permissions when the pointers protected by these mechanisms are finally dereferenced, providing another layer of protection against attacks like PACMAN [60].

Hardware and Software Control Flow Integrity. Hardware (e.g., Intel CET) and software (e.g., LLVM-CFI) control-flow protections can integrate with SecureCells, improving intra-compartment control-flow protection to complement SecureCells' inter-compartment call gates (`SDEntry`). CET can continue to check indirect call targets for `endbr` instructions. LLVM's and other fine-grained CFI pointer checks are implemented in software, orthogonal to hardware control flow checks.

Page-based mechanisms. By itself, SecureCells restricts popular mechanisms (e.g., guard pages, swapping) operating on pages and page tables since translations and protections are tracked at cell granularity. However, SecureCells can be integrated with upcoming intermediate address-space systems like Midgard re-enabling programmers to implement these crucial features. Midgard couples SecureCells-like range-based translation at the core with a second level of page-granularity translations at the backside of the last-level cache. Guard pages and swapping can both be implemented by unmapping the requisite pages in the backside translation.

Appendix D. Speculative Side-Channel Attacks

We consider the threat of speculative side-channel attacks like Spectre [47] in SecureCells' design, despite omitting such attacks from our attacker model. SecureCells introduces additional mechanisms for changing an executing thread's permissions, through userspace compartment switching and permission transfers. Fault-based attacks like Meltdown [61] must be prevented in implementations by preventing faulting loads from accessing memory or forwarding their data to subsequent instructions [62].

SecureCells specifies that userspace instructions are serializing, precluding speculative permission changes. An attacker cannot, for example, speculatively switch to a victim SD using an `SDSwitch` following a long-latency branch and read the victim's private data using the victim's permissions. SecureCells' permission transfer instructions are atomic, preventing visibility or exploitation of any intermediate permission state. An attacker SD cannot, for example, drop permissions for a cell using `SCProt` while transferring the same permissions using `SCTfer` in parallel. Our firmware (and future microcode) implementation use load-linked store-conditional atomic operations commonly available across architectures to ensure atomicity. SecureCells allows the pipeline to speculate as usual within a compartment's execution, and speculative accesses are also subject to access control by the MMU and cannot illegally access any cell. Access control, therefore, also limits the leakage potential of existing Spectre gadgets. Whereas a Spectre gadget on a traditional processor can address and access any user memory in the process' address space, the same Spectre gadget can only access memory within the compartment's cells. SecureCells also limits the code (speculatively) executable within a compartment, further restricting the availability of Spectre gadgets.

Appendix E. SecureCells Implementation Trade-Offs

SecureCells permits a range of implementations scaling from simple microcontrollers with firmware emulation for added userspace instructions to server grade processors with microcode or hardware implementations. In this section, we describe the trade-offs and justify our implementation in Section 3.4.

Firmware. On the simplest side of the spectrum, instructions can be emulated by firmware using trap-and-emulate. Firmware is programmable code which runs in a privileged execution mode and uses native ISA instructions. SecureCells' instructions will trap into firmware, and be dispatched to the emulation code. Firmware implementations are cheap, requiring no additional hardware, but slower than alternate implementations. For the simple RISC-V RocketChip microcontroller, we choose firmware emulation for permission transfer instructions. Note that the firmware can also forward traps to be emulated by either the supervisor or even a privileged userspace library. However, the additional security risk of emulation by less trusted software risk and the overhead of forwarding traps makes such implementations less attractive.

Hardware. Alternatively, instructions can be implemented in hardware with finite-state machine circuits. While this design option implies better performance, designing complex hardware comes with silicon and power costs and substantial complexity. Hardware bug fixes incur the significant cost of the tape-out process. Server and desktop processors generally include beefy cores with large silicon area, where hardware implementations may match the processor's targeted performance. We implement the crucial `SDSwitch` instruction in hardware to reap the performance advantage, and because of the simplicity of its design.

Microcode. A third option, microcode, is programmable code provided by the processor manufacturer, built from low level operations including ones not available through the ISA interface. When a instruction implemented in microcode is encountered, a microcode sequencer fetches microcode from an on-chip RAM and executes them in the pipeline. Microcode eliminates the cost of trapping and dispatch encountered in firmware emulation (77% of the latency of

emulating `SCProt`), and can also leverage hardware-specific optimizations. Microcode is popular for implementing complicated instructions with high performance like SGX's `EENTER`/`EEXIT` instructions. Microcode also has the advantage of being programmable, and have been leveraged to fix processor errata and bugs. While the simple RocketChip lacks a microcode sequencer, we envision microcode to be ideal for implementing SecureCells' permission transfer instructions for high-performance processors.