Institut Polytechnique de Paris
IP Paris Graduate School

Internship report

for the Cybersecurity M2 research internship
in the Master 2 - Cybersecurity program

01.04.2021 - 30.09.2021

seL4 on SecCells

Operating system support for efficient compartmentalization

Florian Hofhammer

INSTITUT
POLYTECHNIQUE
DE PARIS

EPFL

hexhive

Florian Hofhammer

seL4 on SecCells

Internship coordinator:	**Olivier Levillain**, Maître de conférences SAMOVAR lab Télécom SudParis, Institut Polytechnique de Paris
Internship director:	**Mathias Payer**, Associate Professor HexHive lab École polytechnique fédérale de Lausanne
Internship supervisor:	**Atri Bhattacharyya**, PhD candidate HexHive lab École polytechnique fédérale de Lausanne
Host organization:	HexHive lab, École polytechnique fédérale de Lausanne BC 160 (Bâtiment BC) Station 14 CH-1015 Lausanne
Dates:	01.04.2021 - 30.09.2021

Acknowledgements

I would like to thank Prof. Mathias Payer for giving me the opportunity to conduct my research internship in his HexHive lab at EPFL, for what he already taught me about research during my project and of course for his valuable feedback and advice. I also wish to thank Atri Bhattacharyya for including me in his research project in which I carried out my work, for the intensive discussions we had and for his guidance throughout my internship.

Next, I would like to thank Prof. Olivier Levillain for creating the brand-new cyber security master's program at IP Paris that sparked and constantly increased my fascination for this area of research as well as his guidance and advice throughout my second year of master's that I conducted in said program.

Additionally, I thank Prof. Guénaël Renault and Prof. Silke Biermann for their advice and discussions which helped me grow my fascination for research and finally led up to my decision of pursuing a PhD.

Of course, I would also like to thank all the members of the HexHive lab for welcoming me so warmly to EPFL and all the scientific as well as non-scientific discussions we shared so far. I am very much looking forward to continuing to work with the people in the lab during my PhD!

Special thanks go to my family who supported me so much during the past five years of my bachelor's and master's studies, emotionally as well as physically, for example when I needed help moving to a new place (once again). Another big thank you goes to my S.O. Dominique who always had my back as a best friend and partner alike. Thanks for always keeping up with my outbursts about random things I learned that fascinate me!

Lausanne, September 20, 2021 Florian Hofhammer

Abstract

Abstract

Incorrect memory accesses pose some of the biggest security threats to software. The MITRE list of most dangerous software weaknesses lists out-of-bounds writes and reads in first and third place, respectively [39]. Most mitigations try to minimize exploitation impact by for example making code execution harder to achieve but they do not target the initial problem.

Both compartmentalization and capability-based systems aim to enforce the least-privilege principle by allowing very fine-grained resource accesses and thus prevent incorrect memory accesses from the start on. In our paper, we combine a novel microarchitecture called *Secure Cells* enabling simple and efficient compartmentalization with the seL4 microkernel providing a capability-based OS kernel.

We present the design of such a system and reason about its implications for performance and security and the system's potential. We also highlight difficulties encountered during implementation and what changes to the initial designs based thereon we identified.

Résumé

Les accès incorrects à la mémoire constituent l'une des plus grandes menaces pour la sécurité des logiciels. La liste MITRE des faiblesses logicielles les plus dangereuses place les écritures et les lectures hors limites en première et troisième position, respectivement [39]. La plupart des mesures d'atténuation tentent de minimiser l'impact de l'exploitation, par exemple en rendant l'exécution du code plus difficile, mais elles ne ciblent pas le problème initial.

La compartimentation et les systèmes basés sur les capacités visent tous deux à appliquer le principe du moindre privilège en autorisant des accès très fins aux ressources et ainsi à éviter les accès incorrects à la memoire. Dans notre article, nous combinons une nouvelle microarchitecture appelée *Secure Cells* permettant une compartimentation simple et efficace avec le micronoyau seL4 fournissant un noyau de système d'exploitation basé sur les capacités (*capabilities*).

Nous présentons la conception d'un tel système et raisonnons sur ses implications en termes de performance et de sécurité ainsi que sur le potentiel du système. Nous soulignons également les difficultés rencontrées lors de la mise en œuvre et les changements apportés aux conceptions initiales que nous avons identifiés.

Contents

List of Tables

List of Figures

List of Listings

Acronyms

ACL Access-Control List. 40

API Application Programming Interface. 13, 19, 26, 28, 34, 35, 37, 41, 42, 44

ASLR Address-Space Layout Randomization. 12

CAmkES Component Architecture for microkernel-based Embedded Systems. 43, 44

CapDL Capability Distribution Language. 43, 44

CFI Control Flow Integrity. 12, 18, 28

CIA Confidentiality, Integrity, Availability. 12, 19

CoW Copy-on-Write. 41

CPU Central Processing Unit. 7, 14, 16, 20, 22, 25, 27–29, 32, 34, 36, 38, 39, 41, 49

CSR Control and Status Register. 25, 27, 29

DoS Denial of Service. 12

FPGA Field-Programmable Gate Array. 45

GPR General Purpose Register. 20, 35

IPC Inter-Process Communication. 18, 19, 25–27, 45

ISA Instruction Set Architecture. 13, 20, 28, 47

LoC Lines of Code. 30, 37, 38

MMU Memory Management Unit. 14, 16, 18, 20, 23, 25, 33, 38

OS Operating System. 4, 14, 16, 18, 19, 26, 27, 35, 40, 42, 44, 46

PTE Page Table Entry. 14, 16, 41, 42

RPC Remote Procedure Call. 27

SecCells Secure Cells. 4, 7, 8, 13–20, 23–25, 27, 32, 34, 37–46, 48

SecDiv Security Division. 7–9, 15, 17, 18, 20–29, 31–38, 41, 42, 44, 45, 49

TCB Trusted Computing Base. 40

TCB Thread Control Block. 35

TLB Transaction Lookaside Buffer. 15, 16, 23, 42

1 Introduction

The memory protection and isolation of a computing device's processes' is fundamental for the whole system's security and for its ability to uphold the basic, so-called *CIA* principles: Confidentiality, Integrity, Availability. If an attacker can arbitrarily read data from the system's memory, the *confidentiality* of information processed by the device may be compromised. When being able to arbitrarily write to the system's memory, *integrity* of data cannot be guaranteed anymore, since an attacker can overwrite and manipulate information at her free will. In addition, an attacker might be able to influence a system's *availability* through manipulating executable memory and making the corresponding software crash by a so-called Denial of Service (DoS) attack.

For decades, there has been an ongoing arms race between attackers and defenders [38]. The idea of deviating control flow and therefore gaining control over execution and data in a process's address space by manipulating memory started to gain traction with Aleph One's article "Smashing The Stack For Fun And Profit" in 1996 [2]. Since then, numerous ideas have been presented to thwart attacks aiming to manipulate critical data and control flow information in memory, such as differentiating between executable and non-executable memory, randomizing memory addresses via Address-Space Layout Randomization (ASLR), defining fixed entrypoints for execution via Control Flow Integrity (CFI) in software as well as hardware [1, 35] and many more. Still, even nowadays memory reads and writes at non-intended locations are seen as two of the most dangerous software weaknesses as highlighted by their positions in the top three of the MITRE ranking [39].

The aforementioned exemplary measures try to reduce the impact of successful exploitation for code execution but they do not target the underlying problem of incorrect memory accesses. Additionally, current mitigations mainly aim to prevent arbitrary code execution but cannot protect against information leaks. Information leaks through ot-of-bounds memory reads are also harder to detect during testing because they usually do not alter the program's behavior whereas out-of-bounds writes often lead to unexpected behavior and crashes.

We therefore argue that there is a need for more fine-grained memory access control which allows to detect and prevent out-of-bounds accesses from the beginning on. Currently, there exist several approaches to tackle this problem. First, *capabilities* describe the idea of pointers to objects encoding resources being combined with unforgeable tokens so that only the entity holding the capability can access the object denoted by the capability's pointer. The access rights and memory addresses are verified by the kernel against the capability used by userspace software for making a certain request to the kernel for privileged operations. Capabilities can therefore provide fine-grained access control to objects passed between the kernel and userspace processes as well as to objects

shared between userspace processes directly. However, if a process misbehaves because of an attacker taking over control, she can still access all of the process's capabilities.

Second, *compartmentalization* allows to divide a process up into compartments that each have different access permissions to the process's virtual address space. Hence, different tasks inside of a process can execute with each their own set of privileges minimized to the memory access permissions necessary for completing the task. Compartmentalization is therefore a very effective measure for reducing the attack surface of a system and minimizing impact of successful exploitation by constraining an attacker to only restricted code and data access.

Capabilities can help to further refine access control inside of a compartment and to protect Application Programming Interface (API) functions and communication channels between compartments. In our project, we therefore aim to compensate for the shortcomings of capabilities and compartmentalization by combining these access control mechanisms into a single system to increase overall security. For this approach, we combine the compartmentalization features of SecCells, a novel architecture aiming to make compartmentalization a first-class citizen in modern systems, with the capability-based seL4 microkernel. This gives us very fine-grained access control to system resources and kernel API functions through capabilities while also allowing to divide userspace processes up into smaller, more restricted compartments. Since both SecCells and seL4 are designed for high performance, we expect negligible to no overhead when running seL4 on SecCells with compartmentalization features enabled in comparison to a seL4 system running on classic hardware.

Based on this approach, our core contributions consist of the implementation of SecCells as an extension to the RISC-V Instruction Set Architecture (ISA) in the QEMU system emulator [4] and the implementation of support for the new architecture in the seL4 microkernel. Additionally, we present `scthreads`, a userspace threading case study that makes use of the new compartmentalization features for efficient and strong memory isolation inside a userspace process.

2 Background

In this chapter, we present the two components that are used as a base for our implementation. From the hardware side, our system relies on SecCells, a novel approach for efficient userspace memory compartmentalization. Section 2.1 presents this idea, including its targeted security guarantees and microarchitectural contributions.

From a software perspective, we use the seL4 microkernel Operating System (OS) which was chosen for its simplicity as well as its extensive security guarantees. This system is presented in Section 2.2.

2.1 Secure Cells

Secure Cells (SecCells) introduces *range-based virtual memory*. Instead of translating virtual to physical addresses at page granularity via page table walks, contigous memory ranges are referenced by a base address and an upper memory address bound stored in a *Permission Table* per process. The idea of storing memory address translation information for contiguous memory regions as a single entry in a permission table instead of multiple entries in a hierarchy of page tables has architectural as well as security-related implications. In the following sections, we briefly introduce the SecCells design and then elaborate on the above-mentioned implications.

2.1.1 A new paradigm for memory access control

In modern Central Processing Unit (CPU) architectures such as Intel's `x86_64`, ARM's `ARMv8` or the RISC-V Foundation's `RISCV64`, there is usually one CPU register which holds a pointer into memory to the first-level page table [3, 19, 43]. The virtual address to be translated by the Memory Management Unit (MMU) into a physical memory address is split up into several indices where the bits of each of those shares denotes the offset into a page table. The first share is used as an index into the first-level page table which allows to retrieve a pointer to the second-level page table and so on until we reach a leaf Page Table Entry (PTE). Such a PTE holds access permissions, the base address of the physical memory page as well as other metadata.

This implies that not only can there only be a single set of permissions for a certain page at a time but also that contiguous memory regions have to be split up into pages to be referenced through this page table mechanism if they do not exactly cover the amount of memory covered by a page.

SecCells on the other hand does not make use of a multi-level table lookup procedure but stores the translation information in a single contiguous memory structure called

Permission Table. In SecCells, memory ranges are referred to as *cells* while entities accessing data are denoted as *Security Divisions (SecDivs)*.

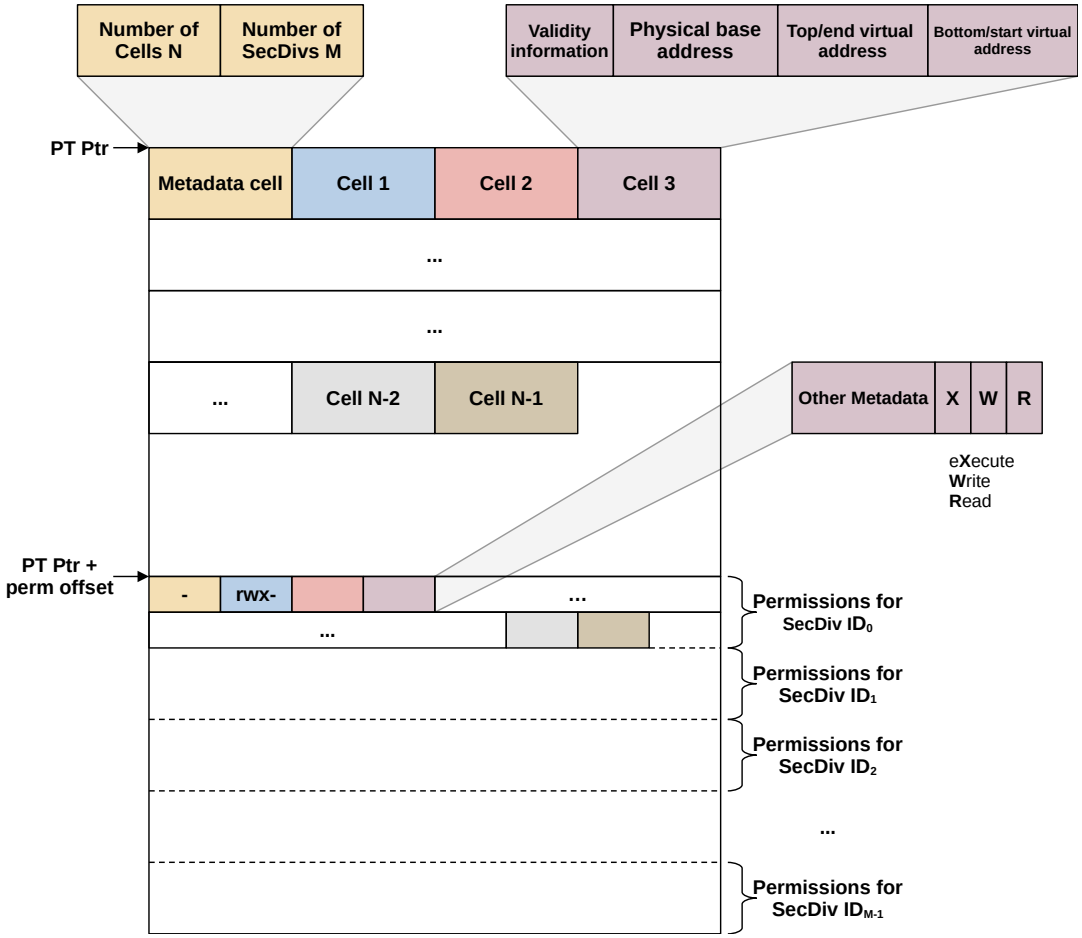

Figure 2.1: Permission table memory structure, permissions and corresponding cells color-coded

Figure 2.1 shows the structure of such a permission table. First, the virtual memory ranges are listed with their bounds and the corresponding base physical address and a validity flag such that any virtual address contained in a cell can be translated to the corresponding physical address. This range-based translation mechanism is further elaborated in Section 2.1.2. At a certain offset after this list of cell descriptors, the access permissions for all of the SecDivs to each of the cells are stored. This allows for multiple permissions for each cell in a single virtual address space. Section 2.1.3 provides the reasoning behind this design decision.

2.1.2 Increasing TLB reach

Transaction Lookaside Buffer (TLB) reach describes the amount of memory that can be accessed through cached translations stored in the TLB without having to do a

page table walk. Even with page table walks sped up through hardware entities in a processor's MMU, such translations can be costly since modern architectures mostly have a hierarchy of page tables with multiple layers which incurs several slow memory accesses. For example, Intel's `x86_64` architecture by default uses four levels of page tables for 4 KB pages with support for five levels of page tables introduced with the Ice Lake architecture [18, 19, 48]. Similarly, `ARMv8` and `RISCV64` also make use of four levels of page tables for 4 KB pages when targeting 48 bits of virtual addresses [3, 43].

Since memory in computing systems has been growing quickly over the past few decades, the TLB could not keep up with that growth. Therefore, more TLB misses are incurred which hurts a system's overall performance. This issue has already been encountered around the turn of the millennium with proposals on how to increase TLB reach ever since [27, 29, 37].

A measure to reduce the pressure on the TLB is to make use of modern architectures' large pages features for memory pages which allows for larger page sizes and therefore less PTEs, for example 2 MB and 1 GB pages on `x86_64`, `ARMv8` with a 4 KB granule for the smallest pages and `RISCV64` with the Sv48 memory model, the latter one also supporting 512 GB pages [3, 19, 43]. Still, there are only certain granularities for page sizes which means that for example for a 800 MB contiguous virtual memory region, either 400 PTEs covering 2 MB of memory each or a single PTE covering 1 GB of memory and wasting 200 MB in the process can be used. In such a case, custom range sizes as the SecCells architecture proposes can enable an OS to map exactly the needed amount of memory in a single translation structure entry.

This reduction of multiple PTEs into a single cell descriptor allows to have a higher TLB reach if the TLB size does not change. Therefore, less memory lookups for translations have to be performed, which greatly improves a system's performance. This main argument is also used to justify the non-constant lookup in case of TLB misses. In classic page tables, a translation can happen at constant time since the number of page table levels is fixed. In the SecCells permission table paradigm, the lookup time depends on the number of cells N since the cell descriptors are simply organized in a contiguous list. However, if the cells are disjoint the descriptors can be inserted into the permission table in a strictly totally ordered list which allows lookups with logarithmic complexity through binary search. Furthermore, the contiguous nature of the permission table increases the chance of all necessary information being present in CPU caches since it is never scattered across the memory space as the different page tables of each level of the hierarchy can be. This can help compensate the non-constant lookup time in a real-world implementation of the range-based virtual memory paradigm.

2.1.3 Enabling efficient compartmentalization

In modern systems, memory isolation is conducted at the granularity of processes through different virtual address spaces. In order to increase security and isolation, in-process memory compartmentalization can be leveraged which is natively supported and explicitly targeted by the SecCells architecture.

For example, when parsing and handling user-controlled data such as network packets

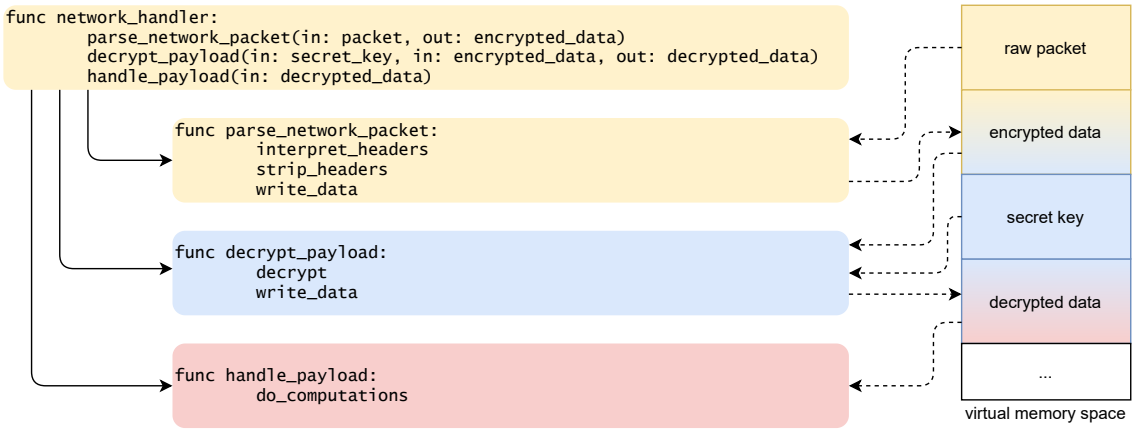

Figure 2.2: Network data handling pseudocode example for compartmentalization

as shown in the example in Figure 2.2, one might want to isolate parsing from decryption of encrypted payloads and handling of the payload. In the given example, one compartment parses the raw network packets and writes the encrypted payload to a buffer shared with another compartment. The second compartment is then the only compartment with access to secret key material to decrypt the payload and passes the decrypted information on to a third compartment which handles the decrypted payload. In this case, a bug in the network parsing cannot compromise the actual packet payload since the parsing compartment neither has access to the secret key material nor to the decrypted payload.

Table 2.1: Unprivileged instructions added by SecCells

Instruction	Operands	Description
SDSwitch	imm. offset, SecDiv ID	Direct switch
SDSwitch	reg. offset, SecDiv ID	Indirect switch
SDEntry	—	SecDiv entry point
SCInval	addr	Invalidate cell
SCReval	addr, perm	Revalidate cell
SCGrant	addr, perm, SecDiv ID	Grant perms
SCTfer	addr, perm, SecDiv ID	Transfer perms
SCProt	addr, perm	Drop own perms
SCCount	addr, perm	Count SecDivs with perm access

SecCells aims to facilitate use while keeping great overall system performance with hardware support for compartmentalization through SecDivs. Since each SecDiv has its own set of permissions for the cells as seen in Section 2.1.1, different responsibilities inside a process can be split up into SecDivs with each their own view on the virtual address space. The current SecDiv's ID is stored in a dedicated processor register such

that the MMU can check at every point in time that the currently executing SecDiv is actually permitted to access a requested memory range.

This register can only be modified by specific instructions for SecDiv switches in order to prevent an attacker from switching compartments by simply writing to the dedicated register. Those instructions exhibit a CFI-like behavior where they check for presence of another special instruction at the destination which marks a valid entry point into another SecDiv. Additionally, SecCells adds instructions to grant, transfer and drop permissions, quickly invalidate and revalidate cells via the valid flag in the cell descriptor and to retrieve the number of SecDivs having access to a certain cell. An overview over those instructions running solely in unprivileged user mode is also given in Table 2.1.

2.2 The seL4 microkernel

seL4 is an OS microkernel mainly developed in the Trustworthy Systems group at the University of New South Wales in Sidney, Australia (previously Data61/CSIRO [7, 13]) and governed by the seL4 Foundation [34, 40].

seL4 is an OS microkernel, meaning that it only provides the bare minimum of services such as hardware initialization, virtual memory and scheduling to userspace applications. Services typically built into an OS such as file systems and device drivers are fully delegated to userspace in a microkernel system. Instead of having multiple million lines of code running in the processor's privileged kernel mode as it is the case in monolithic OSs such as Linux or Windows, microkernels such as seL4 typically only run a few thousand lines of code in kernel mode while above-mentioned services like drivers are executed with usermode privileges [17].

This design approach reduces the amount of privileged code and thus the attack surface for malicious users. On the other hand, communication between such basic OS services cannot be conducted via accessing shared kernel memory buffers but has to be done via Inter-Process Communication (IPC) primitives, incurring a certain communication overhead. The seL4 developers claim nevertheless that their microkernel provides the fastest IPC implementation in any existing microkernel and that consequently, there is no security-performance trade-off [17].

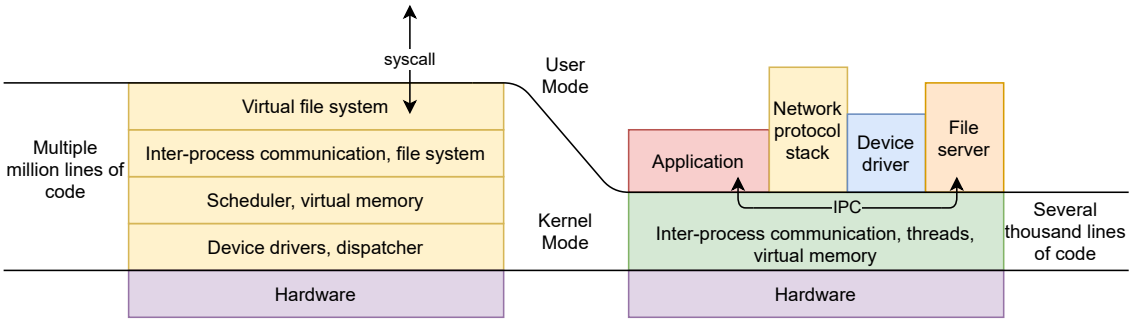

Figure 2.3: Monolithic kernel (left) vs. microkernel (right) (inspired by [17], Figure 2.1)

seL4 provides two further mechanisms for improving reliability and security of the kernel apart from the isolation of as many components as possible in independent userspace programs. For once, seL4 is formally verified, meaning that machine-verifiable proofs have been created that provide guarantees for functional correctness with regard to the kernel's specification as well as enforcement for the Confidentiality, Integrity, Availability (CIA) security principles. The proofs cover the kernel code written in the C programming language as well as the translation into binary code for certain architectures [17, 20, 21].

Additionally, seL4 is a *capability-based* system [17, 21]. A capability encodes an immutable reference to an object as well as access rights to this object. Objects can be of different types. For example, they can refer to plain virtual memory areas or API functions to be invoked. The OS ensures that only the entity holding a capability can operate on the encoded object. New capabilities can be derived from existing ones with a subset of the access permissions so that permissions can be passed on and diminished, if necessary. This mechanism allows very fine-grained access control and thus facilitates the implementation of the least privilege principle [17].

Combining SecCells with seL4 allows a developer to make use of the security guarantees of seL4 and its lightweight nature while improving privilege separation inside of a thread[1] with the help of SecCells' compartmentalization features. Therefore, some tasks separated into different threads and address spaces on an unmodified system may be grouped into a single thread on a system based on seL4 on SecCells without reducing memory isolation and hurting resulting security guarantees. Such a modification to a system can improve overall performance, since costly IPC calls requiring context switches from a userspace thread into the kernel and back out to another program running in userspace can be avoided.

Based on this goal, the following chapters provide an overview of design and implementation of such a system combining the SecCells architecture with the seL4 microkernel.

[1] In seL4 terminology, there is no notion of processes. Any schedulable task is called a *thread*.

3 System design

In this chapter, we present the ideas and steps to tackle the implementation of seL4 on SecCells. The chapter is divided up into three parts. In the first section, we provide an overview about the general structure of the project and the targeted components. The second section shows the anticipated changes to the memory subsystem to account for the range-based virtual memory. As a seL4 thread does not necessarily need to make use of SecCells' compartmentalization features since any code can just run in a single SecDiv, we can draw a line between changes to virtual memory mainly affecting the kernel space presented in Section 3.2 and seL4 userspace compartmentalization introduced in Section 3.3.

3.1 General project structure

In order to get a full-blown system up and running with SecCells' virtual memory and compartmentalization features, there are three main components to consider.

Looking at the system stack from the bottom up, the first component is the hardware, more specifically the CPU including a MMU. SecCells is designed to be architecture-agnostic and its microarchitectural extensions can be applied to processors implementing any ISA. Our SecCells implementation is based off of the RISCV64GC ISA which describes a RISC-V architecture with 64 bit wide General Purpose Registers (GPRs) and the GC ISA extensions for integer multiplication and division, floating point support, a compressed instruction format and other basic functionalities for a general-purpose ISA [42]. For prototyping and functionality tests, we implement the SecCells modifications in QEMU, a virtualization engine and emulator supporting several ISAs, including RISC-V [4, 30]. Both the RISC-V architecture as well as the QEMU emulator were chosen for their open-source nature and their resulting availability and extensibility for research purposes.

The second component is the code running in the CPU's privileged mode. This includes the seL4 kernel itself as well as code needed for booting up the system such as a bootloader. In our system, OpenSBI [45] is used for basic hardware initialization and followed up by seL4's elfloader which loads kernel and userspace applications from an image into memory before handing over control to the kernel [36].

An userspace application running on top of the seL4 kernel and thus in the CPU's unprivileged mode forms the third and last component. On seL4, the kernel hands over control to the so-called *rootserver* after initialization. This rootserver is a userspace application that gets access to the system resources provided by the kernel and can then

setup other threads and dispatch the resources as required by passing on the corresponding capabilities [5].

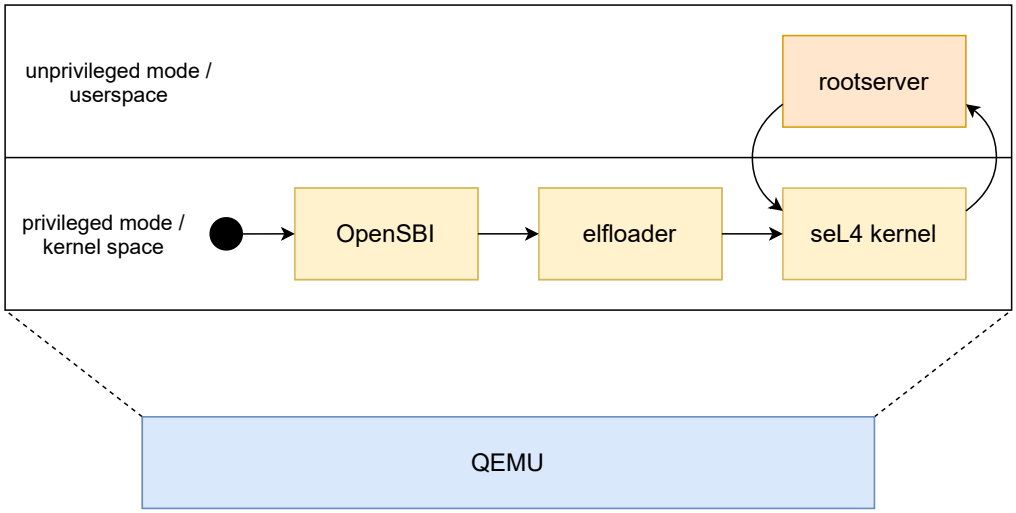

Figure 3.1: The components of a basic seL4 system stack, arrows denoting possible control flow transfers

3.2 Range-based virtual memory

Virtual memory plays an important role in achieving certain security properties through isolation of virtual address spaces. The seL4 developers themselves claim that "[u]sing seL4 without a full memory-management unit (MMU) makes little sense, as its resource management is fundamentally based on virtual memory" [5].

The memory structure shown in Figure 3.2 is an implementation-dependent extension of the general design as seen in Figure 2.1.

First, the base pointer to the permission table is stored in the `SATP` register which in classic RISC-V implementations holds a pointer to the first-level pagetable.

Second, the size of a cell descriptor is set to 16 bytes which allows for enough space to store physical and virtual page numbers. Note that even though we do not rely on paging mechanisms, we still use the term page number because we set the granularity for ranges to multiples of 4 KB and we can thus reduce the necessary number of bits for storing bottom and top addresses by 12 each[2]. This is an arbitrary implementation choice in our current system. Range boundaries can be tuned for finer granularity by making use of the currently unused 11 bits per cell descriptor or by increasing the size of a cell descriptor and thus being able to use more bits for storing addresses.

Third, we store the additional number of 64 byte cache lines T used for storing the permissions a single SecDiv has on the different cells in the metadata cell. This number defines the capacity of our permission table for storing cell descriptors and permissions

[2] $4096 = 2^{12}$

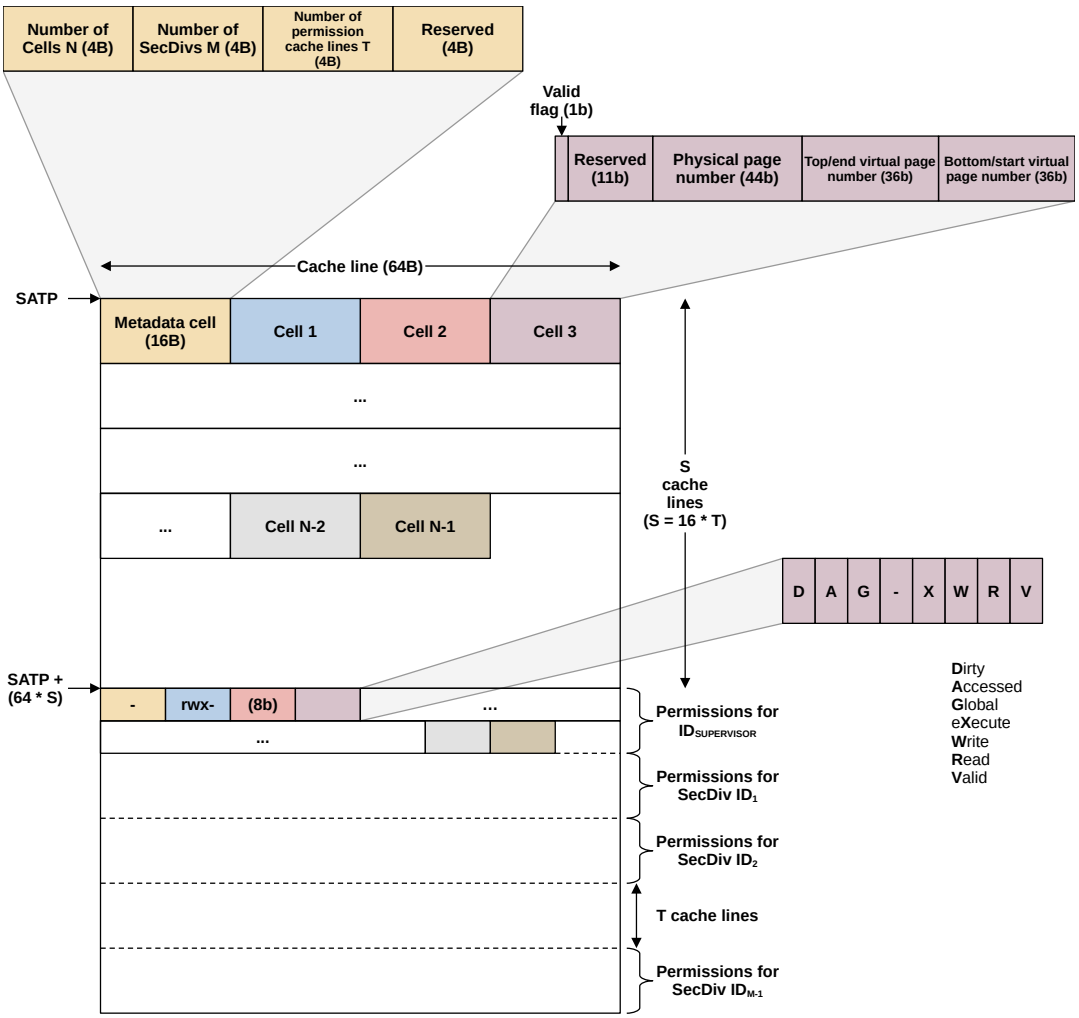

Figure 3.2: Permission table memory structure in our RISC-V system, permissions and corresponding cells color-coded

and allows to easily address a set of permissions to a cell, since it fixes the offset from the start of the cell list pointed to by `SATP` to the start of the permission list. The granularity of 64 bytes and the notion of cache lines were chosen because this is a common cache line size in CPU data caches [19].

With the above definitions, the address for the permissions a SecDiv j has on cell i can be calculated as follows:

$$\begin{aligned} \text{addr} &= \text{SATP} + (64 * S) + (64 * T * j) + i \\ &= \text{SATP} + (64 * 16 * T) + (64 * T * j) + i \end{aligned}$$

Note that there is no bit in the permissions denoting whether they apply to accesses from kernel space or from userspace. In our system, we denote SecDiv 0 as the *supervisor SecDiv* and reserve all other IDs for userspace compartments. By default, the kernel

initializes only a single compartment for the rootserver which in turn can then setup more SecDivs if necessary. The MMU then only takes the SecDiv ID into consideration when checking access permissions.

The cell descriptors and permissions are simply stored in contiguous lists in a coherent range of memory. How to access the cell descriptors is up to the system designer. Even though SecCells proposes storing the cell descriptors in a totally ordered way, one could simply perform lookups by walking the list of descriptors linearly. In this case, worst-case lookup complexity is $O(N)$ no matter whether the list is ordered or not. In the latter case, insertion can be performed by simply appending to the list. In this case, complexity is $O(M)$ since a set of performances has to be appended to each of the M SecDiv's permission lists as well. Deletion complexity is $O(N * M)$, as for the descriptor list as well as any of the M permission lists it may be necessary to shift up to N cell descriptors and permission sets around if the first cell is deleted. In case of totally ordered lists, lookups can be performed via binary search and the worst-case complexity therefore drops down to $O(\log(N))$. On the other hand, insertion complexity increases, since it is not possible to simply append to the lists but the correct position in the list has to be found and up to N other descriptors and permission sets for each of the M SecDivs have to be shifted back to free up space for the ones to insert. Table 3.1 compares the complexities side-by-side.

Table 3.1: Worst-case execution time in Big O notation for unordered and ordered cell descriptors, M number of SecDivs and N number of cells as in Figures 2.1 and 3.2

Operation	Unordered	Ordered
Lookup	$O(N)$	$O(\log(N))$
Insertion	$O(M)$	$O(\log(N) + (N * M))$
Deletion	$O(N * M)$	$O(N * M)$

We argue that even though insertion complexity greatly increases, the totally ordered list should be preferred over the non-ordered list since lookup complexity decreases and lookups are performed much more frequently. More specifically, a lookup has to be performed for every memory access, be it data access triggered by the code or instruction loading for running the code itself. Depending on the implementation of TLB-like caching mechanisms, the decreased lookup complexity therefore outweighs the increased insertion complexity.

Our modifications to the system stack as shown in Figure 3.1 include implementing SecCells-based virtual memory in QEMU's emulated MMU as well as the seL4 microkernel. There is no need to modify OpenSBI since it only runs in physical memory and does not enable virtual memory translation. The elfloader can be neglected as well even though it enables virtual memory. However, the kernel does not use the mappings set up by the elfloader but maps any necessary virtual memory to physical memory again on its own. Since our design for the RISC-V software MMU in QEMU can support both

the original page-based as well as the SecCells range-based virtual memory modes at runtime, the elfloader can still use paging for its virtual memory requirements whereas the kernel sets up any necessary mappings in SecCells-style virtual memory on its own. Nevertheless, we still modify the elfloader to have the full system stack make use of SecCells-based virtual memory to ease later hardware implementation.

We make use of the totally ordered cell descriptors for faster lookups through binary search. Since the memory ranges encoded by the cells are fully disjoint, we can establish a strict total order on the cell descriptors with respect to the base virtual address. The complexity for inserting and deleting cell descriptors and the corresponding permission sets is shifted to the kernel while the lookups are performed in hardware and thus in our case implemented in QEMU.

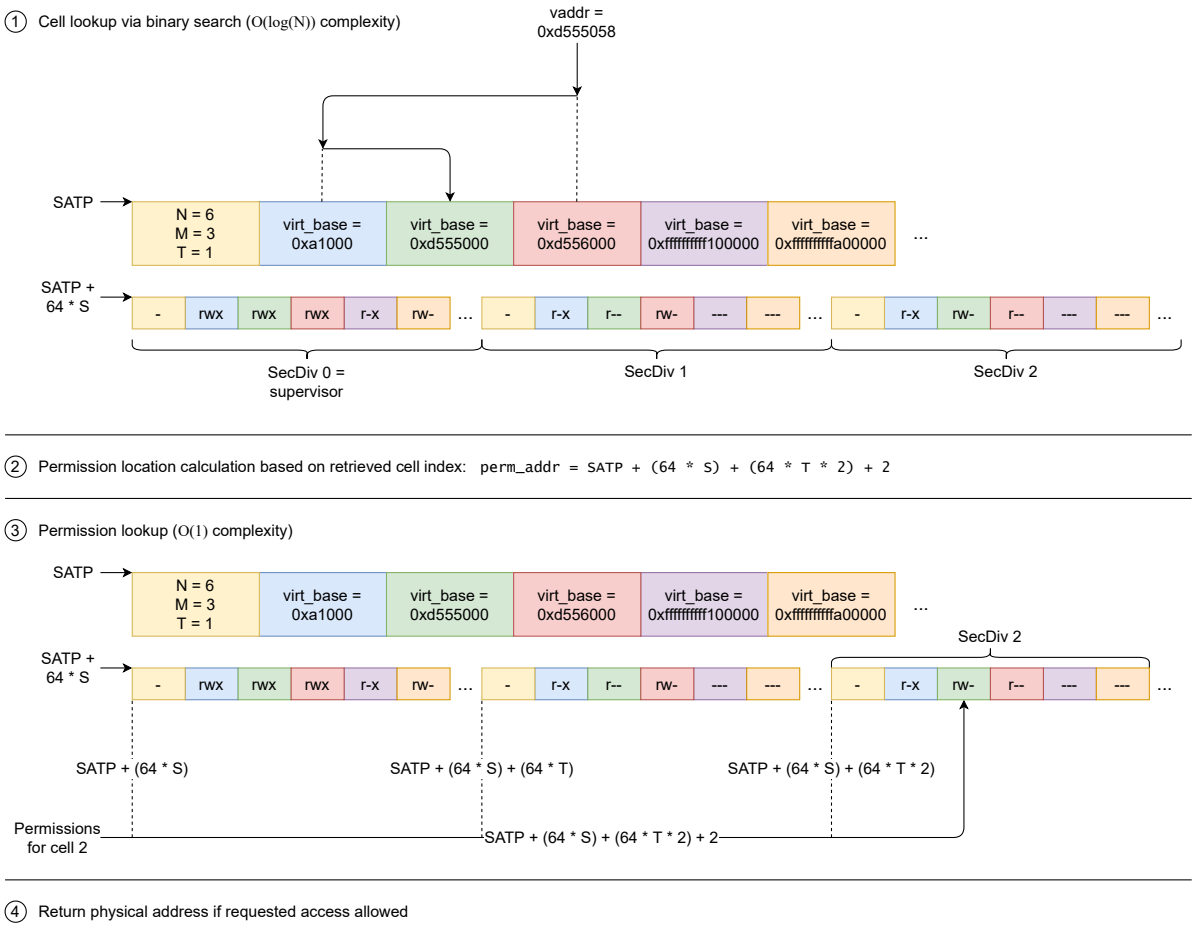

Figure 3.3: Exemplary steps for virtual address translation when SecDiv 2 requests access to address `0xd555058`

3.3 Userspace compartmentalization

The main objective of the SecCells architecture is to enable efficient compartmentalization in userspace. The range-based virtual memory presented above is a tool to support this goal together with the notion of SecDivs. The following sections present the implications of compartmentalization as well as possible use cases for this technology.

3.3.1 Compartmentalized memory layout

Figure 3.3 showing the address translation procedure also implicitly gives an example for compartmentalization. There are two userspace SecDivs with different permissions on the cells starting at `0xd555000` and `0xd556000`, respectively. Note that in the actual implementation, the cells also store an end address and the physical address. Additionally, the cells only store the page numbers at a granularity of 4 KB which in this case would lead to base addresses of `0xd555` and `0xd556` stored in the corresponding cell descriptors. The referenced figure is thus only showing a simplified example. In the example, SecDiv 1 can access the memory range `0xd555000-0xd555fff` with read-write permissions while SecDiv 2 can access the same memory read-only. For the cell starting at `0xd556000`, the permissions are reversed. Such an example shows two SecDivs communication via shared buffers where they are mutually distrustful and do not want the respective other SecDiv to be able to modify the data in the shared buffer.

The different permissions for SecDivs on cells allow to have different views on the same virtual address space depending on the currently executing SecDiv. In the example shown in Figures 3.3 and 3.4, SecDivs 1 and 2 cannot access kernel code and data at all while sharing the same executable code section for the userspace code. However, their view on the shared data buffers differs as well.

SecCells by this mechanisms manages to provide each SecDiv with a different view on memory even though they execute in the same virtual address space. Therefore, memory accesses can be isolated depending on which SecDiv is currently active. The MMU determines which SecDiv is trying to access memory and thus causing an address translation and access check by reading the value from the `USID` register newly introduced to the RISC-V CPU. This processor register implemented as a Control and Status Register (CSR) can only be modified via specific instructions for switching SecDivs as seen in Table 2.1 and further explained in Section 3.3.2.

The isolated view on memory can provide more security in a single process as shown exemplary in Figure 2.2 as well as allow for the grouping of previously separated processes into a single process. The latter may in some cases be necessary to achieve good isolation between security-critical components but adds overhead for IPC and context switching. SecCells reduces that cost to the cost of SecDiv switching which we argue is minimal since no detour through the kernel has to be taken and IPC can be reduced to writing to and reading from shared buffers instead of copying data to and from the kernel in addition to the context switches. For example, modern web browsers such as Google Chrome or Mozilla Firefox make heavy use of multi-process architectures to isolate websites from each other [10, 31, 32]. Since the base process for handling each website is the same,

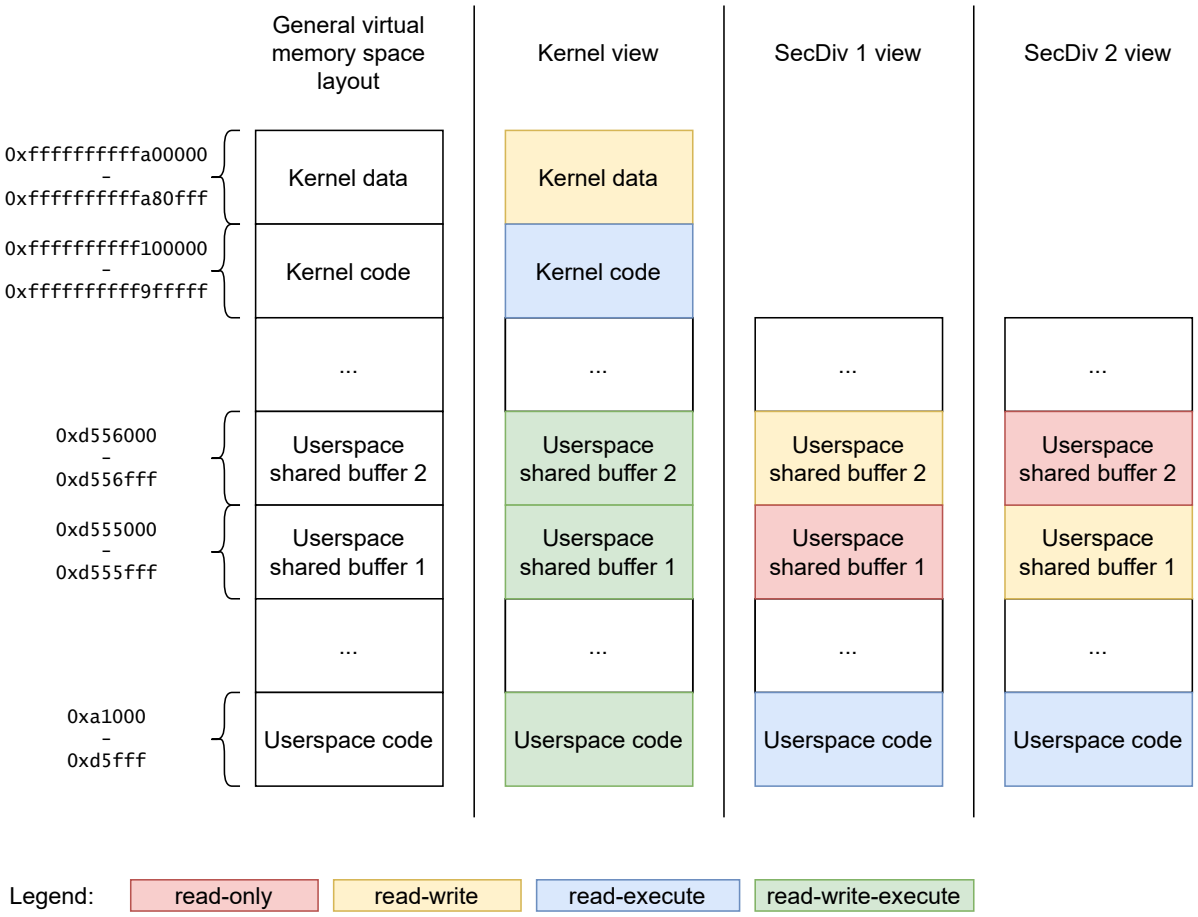

Figure 3.4: Example for different views of SecDivs on the same virtual memory space (c.f. Figure 3.3)

code could be shared while websites are isolated from each other via SecDivs inside a single process.

Cells have to be mapped and unmapped in privileged mode and therefore by the kernel. Since the size of a memory range described through a cell is not fixed like a page size, only a single syscall or API call into the kernel is necessary in cases where multiple pages would have to be mapped via independent syscalls in a classic pagetable-based environment.

3.3.2 Managing SecDiv switching and permissions

Context switching is a costly operation. It requires saving and restoring of context information and virtual address space switching with such a privileged operation being executed by an OS's kernel. This observation can be extended to IPC. An IPC call requires at least a context switch into the kernel, copying of arguments from one userspace task to another and a context switch back from the privileged mode into the target usermode application. The necessity to copy data can be avoided by setting up

shared memory via which the processes can communicate directly. However, setting up shared memory also has to be done by the OS kernel since it requires modifying the memory mappings such that a shared buffer in different virtual address spaces points to the same physical memory location. Even though seL4 provides a high-performance IPC mechanism [5, 14, 17, 20, 26], such overhead cannot be completely avoided in IPC-based systems.

In our system, we aim to reduce the frequency of IPC calls by grouping tasks into a single process and isolating them into SecDivs with different memory views as described in Section 3.3.1 where possible. Compartments can then communicate via shared buffers in the same virtual address space. To efficiently make use of this mechanism, it is of high importance to switch between SecDivs and manage access permissions with lower overhead than IPC incurs. We achieve minimal performance cost by avoiding the cost of context switches and communication through the kernel. Any operation as described in Table 2.1 is performed in userspace via new unprivileged processor instructions listed in Table 3.2. For the actual binary encodings for the RISC-V assembly instructions we refer to Appendix A.

Table 3.2: Unprivileged RISC-V instructions added by SecCells

Instruction	RISC-V equivalent	Comment
SDSwitch (direct)	`jals`	jump-and-link-secure
SDSwitch (indirect)	`jalrs`	jump-and-link-secure-register
SDEntry	`entry`	Executed as no-op
SCInval	`inval`	
SCReval	`reval`	
SCGrant	`grant`	
SCTfer	`tfer`	
SCProt	`prot`	
SCCount	`count`	

In addition to the new instructions, we add two CSRs. The `USID` register already briefly mentioned in Section 3.3.1 contains the ID of the currently active SecDiv. This register can only be updated via privileged writes in the CPU's supervisor mode or the `jals` and `jalrs` unprivileged instructions in user mode. Additionally, we introduce the `URID` CSR which is subject to the same restrictions. This register holds the previously active SecDiv's ID. The rationale behind this register is that a SecDiv can know which compartment was previously active. In an Remote Procedure Call (RPC)-like setting, this allows to easily return to a caller in another SecDiv by issuing one of the SecDiv switch instructions with the corresponding SecDiv ID as an argument. The switching instructions copy the current `USID` value into `URID` and then update `USID` with the requested target SecDiv's ID.

Those changes only require minor changes to the seL4 kernel such as setting `USID` to

the supervisor ID 0 upon kernel entry and restoring it when returning to userspace. Additionally, the kernel has to provide an API function which can be called from userspace to create new SecDivs since their number is stored in the permission table's metadata cell that can only be modified in privileged mode. The modifications to QEMU include the addition of the instruction encodings and emulation of the desired behavior for those CPU instructions as well as the new registers. Since we rely on the GNU RISC-V toolchain for compiling the seL4 kernel and userspace code, we also add support for the new instructions to the GNU binutils such that the assembler (`as`) and debugger (`gdb`) allow us to emit binary code for the instructions and also debug code using those instructions.

The semantics for the `jals` and `jalrs` functions are similar to those of the RISC-V base ISA's `jal` and `jalr` with the main difference being that they update `USID` and `URID`. Additionally, the CPU checks that the instruction at the jump destination is an `entry` instruction which itself is executed as a no-op. This allows for basic CFI functionality guaranteeing that SecDiv switches can only occur at code locations defined by the programmer.

The `inval` and `reval` instructions respectively unset and set the valid bit in a cell descriptor. In addition, access permissions for all SecDivs are blanked at invalidation and a SecDiv revalidating a cell can define the permissions to set for the targeted cell. Invalidation is only allowed if the requesting SecDiv has exclusive access to the targeted cell to prevent a malicious compartment from retracting other compartments' permissions and elevate its own privileges. This mechanism together with `grant`, `tfer` and `prot` allows for easy implementation of pipeline mechanisms where each SecDiv in the pipeline operates on a subset of the previous SecDiv's permissions and where the last compartment in the line can invalidate a cell such that the first SecDiv can revalidate it with its original privileges and restart the pipeline. The latter three instructions grant a subset of a SecDiv's own permissions to another SecDiv, transfer them by granting permissions and dropping the own permissions, or just drop the own permissions, respectively. Finally, `count` returns the number of SecDivs currently being able to access a given address with certain permissions. This functionality enables a SecDiv to make sure that it has exclusive access to certain memory regions and there is thus no risk of leaking data to other SecDivs via such address ranges.

3.3.3 Handling invalid arguments

The instructions described in Section 3.3.2 introduce many possibilities for a programmer to provide incorrect arguments to an instruction. In such cases, we do not want the CPU to fail silently and continue to execute the next instruction but the processor should throw descriptive exceptions which may be handled in software. For example, we require the SecDiv ID on which to operate in certain operations such as granting permissions to be neither the supervisor ID 0 nor higher than the current maximum SecDiv ID bounded by the parameter M in the metadata cell at the beginning of the permission table (c.f. Figures 2.1 and 3.2).

For this reason, we introduce the exceptions listed in Table 3.3 for errors occurring

during the execution of the unprivileged instructions enumerated in Table 3.2. For more details on the exceptions, we refer to Table A.1.

Table 3.3: Exceptions and their meaning added to a RISC-V CPU

Exception	Description
Illegal address	Provided address is not mapped
Illegal permissions	Too little source permissions, already existing target permissions
Invalid cell state	Valid flag value differs from expected value
Invalid SecDiv ID	Target SecDiv ID is 0 or too high
Illegal target	Instruction at SecDiv switch destination is not `entry`

In order to handle the new exceptions in the seL4 kernel after adding support for them to QEMU, we also need to modify the OpenSBI firmware. More specifically, traps through exceptions in RISC-V are by default handled in machine mode only while the kernel is running in supervisor mode. However, the firmware starting up in machine mode can set up the corresponding mask in the `MEDELEG` CSR that fixes which exceptions to delegate to supervisor mode [43].

4 Implementation and design process

QEMU is a big open source project comprised of 604 078 Lines of Code (LoC) in 8 336 files[3]. seL4 weighing in at 195 334 LoC in 926 files[4] has an arguably smaller code base but is a complex project nevertheless. Still, the design described in Chapters 2 and 3 is pretty straightforward to implement when being familiar with the two projects' respective structure. In this chapter, we therefore do not elaborate on inevitable programming errors and resulting debugging sessions, but describe the process that lead up to the previously presented design, starting from a permission table with no particular order of the cell descriptors and no metadata cell at the start of the table.

In addition, we present a case study in Section 4.3. Our `scthreads` model presented in this section provides an easy to use interface for software developers to simplify usage of strong isolation through compartmentalization. It therefore promotes security by abstracting away low-level compartment management and lowering the bar for developers to secure their systems through compartmentalization.

4.1 Metadata cell introduction

In the original design, the permission table did not contain a metadata cell containing information such as the number of cells described by the corresponding descriptors in the table. Instead, the non-ordered list of cell descriptors was bounded by a descriptor containing a bitstring that is not obtainable through valid cell encodings. Therefore, the number of cells N could be retrieved by hardware and software through iterating over the list of cell descriptors and incrementing a counter until the marker bitstring is encountered.

The values for S and T (c.f. Figure 3.2) were then calculated based on the following formulas:

$$S = \min\left(2^n \mid n \in \mathbb{N} \wedge 2^n \geq 16 * N\right) / 64$$
$$T = \min\left(2^n \mid n \in \mathbb{N} \wedge 2^n \geq N \wedge 2^n \geq 64\right) / 64$$

Both S and T therefore encode memory ranges that have a size that is a power of 2. Even though their calculation is similar, they are independent from each other due to

[3]The numbers for QEMU were retrieved from the source tree at the git tag v6.0.0 via `git ls-files | xargs wc -l` and `git ls-files | wc -l`, respectively.

[4]The numbers for seL4 were retrieved from the source tree at git revision f2c96c3246ebbc1cb29b48471e12e86a0b22891c via `git ls-files | xargs wc -l` and `git ls-files | wc -l`, respectively.

different constraints in their calculation at least for small N.

4.1.1 Performance considerations

The original approach allows to retrieve the values for N, S and T either directly by counting or indirectly by calculation from the permission table. Since those values are sufficient for address translation and access control as seen in Section 3.2, the desired functionality could be achieved.

However, this approach has serious performance implications. Whenever the number N of cells is needed, the whole cell descriptor list has to be walked to count the descriptors until the list end marker is reached. Even on address translation the full list has to be scanned even if the desired virtual address is already contained in the first cell in the list since N is needed to calculate S and T and therefore to retrieve the permissions the requesting SecDiv has. As a result, not only the worst-case execution time as listed in Table 3.1 and explained in Section 3.2 is $O(N)$ but also the best-case execution time.

Storing N, S and T in a metadata cell at the start of the permission table influences performance positively in multiple ways. For once, the best-case execution time for address translation drops down from $O(N)$ to $O(1)$ in the case where the desired address is located in the list's first cell since S and T are directly available without counting the cell descriptors first. Similar effects can also be observed for other operations. For example, the worst-case execution time for insertion of a new cell is reduced from $O(N+M)$ due to the need of counting the cells and then inserting permissions for every SecDiv down to $O(M)$ since the descriptor for the new cell can be directly appended to the end of the list without counting the current number of cell descriptors first.

Second, storing S and T explicitly removes the need of recalculating them based on N every time they are needed, for example during every address translation operation as outlined above.

In a second iteration, we redefine S and T the following way:

$$T = \min\left(n \mid n \in \mathbb{N} \wedge 64 * n \geq N\right)$$
$$S = 16 * T$$

With this new definition, we get more space flexibility since we do not rely on memory alignments based on powers of 2 anymore as well as implicit performance improvements. First, the calculation of T is simplified through simpler constraints. Second, S being directly derived from T based on the ratio of cell descriptor size (16 bytes) to permission set size (1 byte) implies that the space allocated for a SecDiv's permission sets can hold the exact same number of permissions as the cell descriptor list can hold descriptors. Therefore, the two parts of the permission table have the same capacity and have to be resized at the same point in time. With S and T being more independent, the two components – cell descriptor list and permission set lists – may need to be resized at different points in time. As such resizing includes many memory accesses to move around data, grouping these operations may reduce the frequency of resize operations.

Additionally, a kernel developer can freely implement any desired policies and thresholds when and how often to increase or reduce T and thus require moving data in the permission table around. The granularity of 64 bytes for T instead of powers of 2 allows for more flexibility for such policies.

4.1.2 Security considerations

While the values for N, S and T are explicitly known either directly or derived from the other values, the number of SecDivs M generally is only implicitly known. By that, we mean that the CPU does not have a way to determine the number of compartments but that M is known to a developer who has to explicitly create and reference SecDivs in her code. Thus, this number generally does not have to be stored in the permission table or retrieved via counting with a similar system as the cell descriptor list end marker.

However, this poses a security risk in case an attacker can gain control over a system based on this design. Instructions such as SCGrant/`grant` and SCTfer/`tfer` can be executed from userspace but modify the system's permission table which cannot be manipulated otherwise by userspace. This is possible since the CPU accesses this part of memory directly, bypassing usual access control. Therefore, an attacker controlling the SecDiv ID parameter to such instructions can set it so that the CPU calculates an address outside of the memory region reserved for the permission table. This then leads to a privileged write primitive controllable from userspace.

When introducing a metadata cell, it can hold the number M of SecDivs which allows the CPU to quickly check whether an ID provided to instructions modifying the permission table is out of bounds and to thus cause a CPU exception to be thrown. Having meta information such as the number of compartments easily accessible in a metadata cell can therefore mitigate certain security issues arising from malicious usage of the newly introduced SecCells-specific instructions.

4.2 Strict total order on cell descriptors

As already mentioned in Section 4.1, the cell descriptors in the permission table originally had no order. Therefore, whenever a certain cell descriptor had to be found, the list had to be walked sequentially, resulting in the worst-case complexities listed in Table 3.1.

4.2.1 Performance improvements

The motivation behind the enforcement of strictly ordered cell descriptors with regard to the base virtual address of the corresponding cells is to reduce search complexity for the average as well as the worst case. We argue that searching for a certain cell descriptor is the most prevalent operation and therefore the operation that results in the biggest overall performance improvements if well optimized. Looking up the information stored in the list for a certain cell occurs on every memory translation, hence at a much higher frequency than insertion or deletion of cell descriptors. Thus, we accept higher

worst-case complexity for operations such as insertion in order to reduce the average-case complexity for lookups from $O(N/2)$ to $O(\log(N))$ and worst-case complexity from $O(N)$ to $O(\log(N))$ as well.

4.2.2 Disjoint and unique cells

The introduction of a strict total order on the cell descriptors with regard to their base virtual address has implications on the format of cells and their relations. In the original design, cells could be partially or fully overlapping as shown exemplary in Figure 4.1. However, it had to be assured that no two overlapping cells are accessible to the same SecDiv since this would allow different permissions on the a virtual address in the overlapping memory region. Such a conflict prevents the MMU from correctly checking access permissions when translating virtual into physical addresses. Hence, overlapping cells are only allowed to be accessible by different SecDivs.

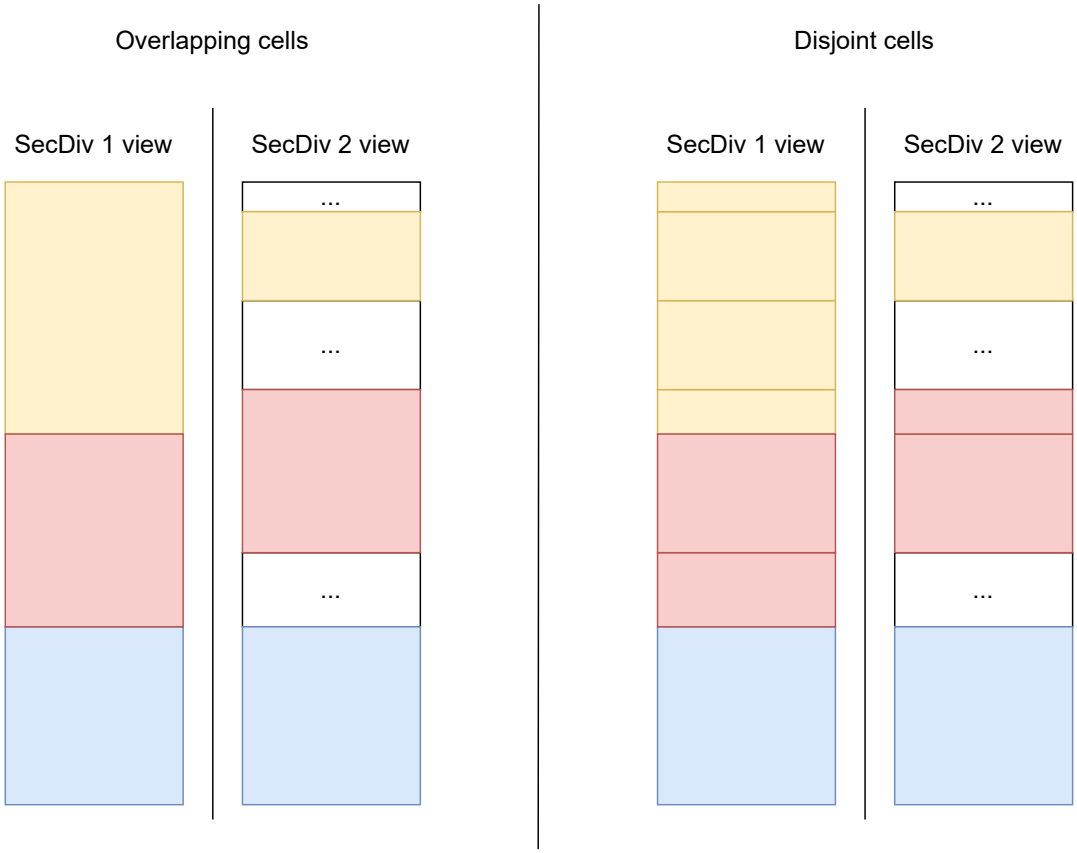

Figure 4.1: Example for overlapping (left) vs. disjoint (right) cells

When looking up the translations for a certain address via binary search on an ordered cell descriptor list, the worst-case complexity would still be $O(N)$ just like for linear search since all cells could be overlapping and including the requested address. Binary search could therefore only find one cell that includes the address but would still have to

look left and right from the found cell descriptor to check whether cells are overlapping.

The same argument holds true for certain CPU instructions such as `count`. With overlapping cells, every cell has to be checked for overlaps containing the address provided as a parameter to `count`. Permissions for any SecDiv have to be checked in order to reliably return the number of compartments having access to the given address. With non-overlapping cells, only the permissions for a single cell containing the provided virtual address have to be analyzed. Hence, complexity greatly decreases for such cases.

In Figure 4.1, we provide an example for how to transition from an overlapping cell model to non-overlapping cells. In the case of overlapping cells being allowed, the blue memory regions can be mapped as a single cell with only the permissions differing for the affected SecDivs. The red and yellow memory regions however have to be mapped as distinct cells since they differ in base and top addresses encoded in the cell descriptor (c.f. Figure 3.2). If cells are non-overlapping, the red and yellow memory regions have to be split up into smaller cells. Parts of those can be mapped as a single cell into both compartments. The disjoint nature allows for efficient implementation of a binary search algorithm and the SecCells-specific instructions referring to cells.

Additionally, the original design incorporated a *deleted* flag in the cell descriptor. When unmapping a cell, the descriptor was not directly removed from the permission table but the deleted flag was set. Since deletion has a high worst-case complexity as noted in Table 3.1, this mechanism reduced the complexity of memory unmapping and shifted responsibility for scheduling the actual deletion of all cells marked as deleted via a kernel API call to the developer. Since any operations on the permission table requiring cell lookup relied on linear search, cell descriptors having the deleted flag set could simply be skipped. In this model, even with non-overlapping cells multiple cell descriptors encoding the same virtual address where only one does not have the deleted bit set could exist in the table at the same time. This again imposes problems onto the binary search approach, as its dependence on unique addresses for quick lookup is not necessarily satisfied.

We therefore decide to remove the deleted bit from the cell descriptor and compact the permission table upon each cell unmapping. This results in the cell descriptors encoding only disjoint, unique cells such that binary search can be performed easily and operations on the permission table are therefore simplified. As a side-effect, we gain one more free bit in the cell descriptor. Even though it is currently not used, it could be used for implementation-specific decisions such as increasing the number of bits available for virtual address to reduce granularity of cells from 4 KB to lower values.

The modifications to the original design described above due to problems encountered during implementation of the system lead up to the refined design for SecCells outlined in Chapters 2 and 3. The implementation phase of the project therefore incurred significant changes that influence not only the implementation of seL4 on a RISC-V based SecCells architecture but also the overall generic design.

4.3 Case study: `scthreads` - mutually distrustful userspace threads

Under the assumption of an attacker with arbitrary code execution capabilities in userspace, simply isolating memory accesses through SecDivs does not provide meaningful security guarantees if the attacker can divert control flow so that she can arbitrarily switch between compartments. To mitigate such issues and have developer-defined entry points for SecDivs, the `entry` instruction described in Sections 2.1 and 3.3.2 can be leveraged.

We propose a userspace threading model called *scthreads* that aims to mitigate an attacker fully controlling the execution in the address space. In the following, we differentiate between *user threads* and *kernel threads*. When referring to the former, we mean our threading model designed to execute fully in userspace without interaction with the kernel. The latter describe threads provided by seL4 which can either execute in the same address space similar to what is known as threads in OSs such as Linux or in different virtual address spaces equivalent to what is referred to as processes in other OSs. Threads as provided by seL4 have to be created via a combination of kernel API calls to set up address space, capabilities and the Thread Control Block (TCB). They are also subject to task scheduling done by the kernel independently from each other as long as they do not implement explicit synchronization mechanisms.

Our `scthreads` model in contrast relies on *cooperative scheduling*, meaning that another thread is only allowed to execute once the previously executing thread explicitly hands over control. Every user thread executes in its own SecDiv. For switching between compartments, we define a single entrypoint that is preceded by a routine to save the currently executing thread's context and followed by another routine to restore the target thread's context. A thread context in our model includes GPR as well as an isolated stack region.

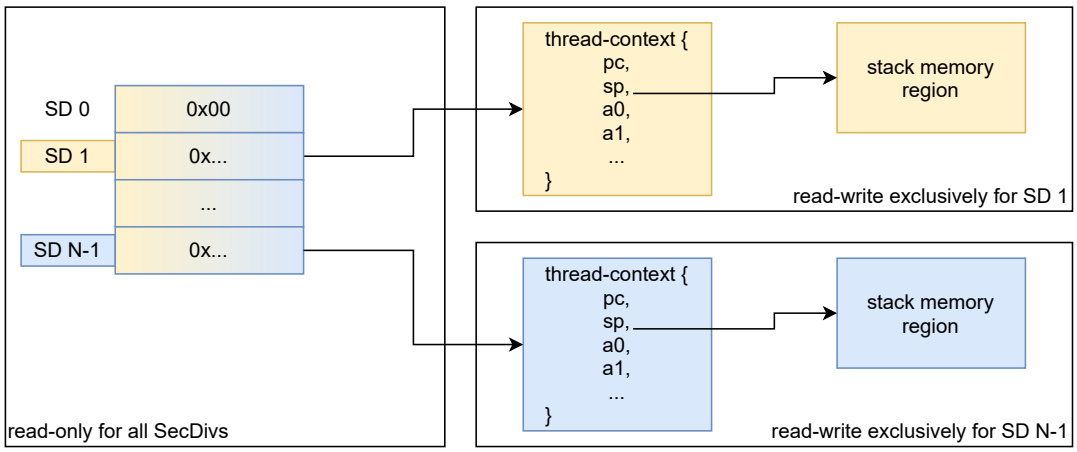

Figure 4.2: Isolated storing of thread contexts enforced by SecDivs

In order to store the contexts safely such that a malicious thread cannot tamper with

another thread's context and therefore bring it under its control the next time the attack target is scheduled, we require the number of threads to be known at the start of the execution such that memory can be set up and protected by an initial thread. For each thread, a memory region to store its register contents as well as space for a stack are mapped. The stack pointer in the former is set to point to the latter and another pointer is set up in a global table indexed by the SecDiv IDs to refer to the corresponding context. Once the necessary memory structures are set up, the initial thread transfers privileges on the context memory structures to the corresponding SecDivs, grants read-only privileges on the shared pointer table to all SecDivs and lowers its own privileges on the same table. Therefore, after the setup any SecDiv is only able to access its own context and cannot modify the context pointer table as shown in Figure 4.2. As a result, it is not possible for a thread to tamper with another thread's context.

In consequence, a malicious thread can only tamper with data accessible to its restricted view on the virtual memory space. As soon as a user thread context switch occurs, execution continues in the new thread at the supposed location due to the context being protected from manipulation. For this security property, it is however necessary to enforce entry into another thread via a unique save-switch-restore routine as described above. When a thread issues a function call to such a routine, the address of the next CPU instruction to execute after returning from the call is saved into RISC-V's `ra` register. Upon a return instruction, execution continues at the location pointed to by `ra` [42]. Therefore, by saving the `ra` register's contents in the context and restoring it when entering a thread, it is possible to simply issue a return instruction at the end of the context restore subroutine. Through this mechanism, execution continues transparently for the scheduled thread exactly where it left off.

5 Evaluation and contributions

In order to assess the functionality of our system, we set up a project based on the adapted seL4 kernel. Since the kernel passes capabilities to physical memory directly to the rootserver which can then request memory mappings from the kernel via API calls, a userspace application allows us to test all the added functionalities.

For once, the modifications made to seL4 for mapping and unmapping cells can be tested by requesting those operations and accessing the memory accordingly. Second, the additional instructions introduced by SecCells can be included in our C code and assembled thanks to our modifications to the GNU binutils.

Listing 5.1: C preprocessor macro for indirect SecDiv switching via a register

```
#define jalrs(ret_reg, dest_reg, sd_reg)        \
  do {                                          \
    asm volatile (                              \
    "jalrs␣%[ret],␣%[dest],␣%[sd]"              \
    : [ret] "=r" (ret_reg)                      \
    : [dest] "r" (dest_reg), [sd] "r" (sd_reg)  \
    );                                          \
  } while (0)
```

Listing 5.2: C preprocessor macro for granting permissions to another SecDiv

```
#define grant(addr_reg, sd_reg, perms_imm) \
  do {                                     \
    asm volatile (                         \
    "grant␣%[addr],␣%[sd],␣%[perms]"       \
    :                                      \
    : [addr] "r" (addr_reg),               \
      [sd] "r" (sd_reg),                   \
      [perms] "i" (perms_imm)              \
    );                                     \
  } while (0)
```

Wrapping certain instructions into preprocessor macros as exemplary shown in Listings 5.1 and 5.2 allows to develop easily understandable C code such as the example for SecDiv switching presented in Listing 5.3.

In addition to the modifications of the virtual hardware and software stack as listed in Table 5.1, we implemented a `scthreads` prototype in roughly 250 LoC and another 750

Listing 5.3: C code excerpt for switching to another SecDiv

```
  /* Grant read-execute permissions on code to other SecDiv */
  grant(&&target_entrypoint, target_usid, RT_R | RT_X);
  /* Switch SecDiv */
  jals(target_usid, target_entrypoint);

target_entrypoint:
  entry();
```

LoC of functionality tests. Those tests currently present the expected behavior for both the emulated hardware implementation of address translation in the MMU and CPU instructions via QEMU as well as the seL4 kernel executing on the emulated RISC-V based SecCells hardware. It has to be noted that our tests currently have the limitation of not fully leveraging the test framework provided by seL4 since this requires extensive modifications to multiple userspace libraries. Such extensions are currently out of scope for this thesis.

Table 5.1: Code contribution, changes retrieved with `git diff --stat`

Component	Base revision	LoC added	LoC removed
QEMU	v6.0.0	1 599	124
seL4	f2c96c3246ebbc1cb29b48471e12e86a0b22891c	1 386	44
elfloader	c09dea9af4908ed0ef05d085ae85eda19dfae5dd	143	1
OpenSBI	v0.9	4	0
seL4_libs	600fe15be5d8d28b68a1f85fa4e5e8d131fde1bc	26	1
GNU binutils	c3eb4078520dad8234ffd7fbf893ac0da23ad3c8	68	0
Total:		3 226	170

The RISC-V architecture provides a basic set of performance counters that can be complemented by platform-specific implementations of additional counters. The standard hardware performance counters include cycle count, real-time clock and retired instructions [42]. To assess performance of a system, we want to either measure elapsed time or cycles. The number of retired instructions on its own does not provide meaningful insights because different instructions can differ in the number of cycles required to retire them. As QEMU is not a cycle-accurate simulator, the system assumes that each instruction takes exactly one cycle and returns the same value for both the cycle and instruction counter. The virtual clock implemented by QEMU relies on the host time. Thus, the retrieved values depend on the time necessary for executing the emulated instructions and therefore on their implementation as well as CPU scheduling on the host and other side effects.

In summary, there are no reliable performance counters in QEMU and we can therefore only reason about performance but not conduct meaningful measurements. Evaluation

of our system's performance thus has to be left to future work to be done on a hardware implementation of a CPU implementing the SecCells architecture.

6 Related work

The combination of seL4's security guarantees through capabilities and formal verification and SecCells microarchitectural defense mechanisms allow for building strong and secure systems. As this general goal is targeted by research as well as industry, we present other work related to the domains of seL4 and SecCells in this chapter.

6.1 Capability-based systems

Capabilities provide a means for fine-grained access control. In contrast to classic Access-Control Lists (ACLs), access to a requested resource is not granted based on the requesting party exhibiting certain characteristics but based on the ability of presenting a tamper-proof capability token.

The *Genode OS framework* [8] implements this approach into its system. One of the main features of Genode is hierarchical sandboxing, which allows a component to spawn isolated child components. Several such steps in a row lead to a recursive tree-like hierarchy of components. The isolation of such parts of a Genode-based system to a large part relies on the secure resource management based on capabilites. Even though the Genode project provides its own microkernel as a root of the component tree, it is generally considered an OS framework being able to run on different kernels and abstracting their behavior away such that software developed for Genode can run no matter which underlying kernel is used. For example, the Genode OS framework supports Linux as a classic monolithic kernel as its base but also seL4 as a capability-based microkernel [6, 8].

Google's *Fuchsia OS* [9] with the underlying Zircon kernel is another example for a capability-based OS. On Fuchsia, software is heavily sandboxed using strictly isolated containers. The only possibility for software to interact with other components on the same system or the outside world is via capabilities explicitly provided to a component.

CHERI (Capability Hardware Enhanced RISC Instructions) [44, 47] differs from the previously presented approaches in that it is not an OS implementing capabilities but a set of architectural modifications, enabling capabilities on the hardware level. CHERI capabilities can encode any pointer into memory and access permissions related to this address. Therefore, the system's capabilities not only apply to predefined objects encoding system resources but to anything that can be referenced via a pointer. In addition, the Trusted Computing Base (TCB) is minimized down to the hardware level. On other systems, the hardware as well as the OS kernel have to be trusted in order to achieve the promised security guarantees whereas CHERI provides tamper-proof capabilities already on the hardware level.

6.2 Range-based virtual memory

Gandhi et al.'s paper "Range Translations for Fast Virtual Memory" [11] gets rid of classic pagetable-based virtual memory and proposes to do virtual to physical address translation at range granularity, where ranges can have arbitrary size. The cell-based virtual memory in SecCells is similar to this approach, there are however important differences. First, ranges and cells in both cases have a granularity of 4 KB. However, as described in Chapter 3, this is just an implementation detail in our current system and not fixed by design, whereas the range-based system by Gandhi et al. incorporates this granularity into the design. Second, their range-based virtual memory does not account for multiple permissions for a single address range as SecCells does. Memory protection information is stored together with translation information similar to classic PTEs. Furthermore, the range descriptions used for address translation are stored in a B-Tree with base and top address as keys, each node in the tree storing translation information for four ranges and pointing to five child nodes. This structure is not only way more complex than the ordered list of cell descriptors we described in earlier chapters but also requires more memory due to the additional pointers to other nodes.

The recently published *Midgard* paper [12] introduces a new address translation layer in between the virtual memory as seen by a userspace process and the physical memory as seen by the CPU. Midgard aims to make memory translations into physical addresses less frequent by providing a single system-wide, range-based memory space. The addresses found in this memory space are used for indexing the CPU's cache hierarchy. This method defers the actual translation into physical addresses until a cache miss occurs which greatly reduces translation frequency. Gupta et al. use a B-Tree approach similar to the one proposed by Gandhi et al. for the front-side translation from processes' virtual address spaces to the system-wide Midgard address space and a radix pagetable for the backend translation into actual physical addresses where necessary.

6.3 Compartmentalization and memory protection

Mondrian Memory Protection (MMP) [46] introduces multiple protection domains per virtual address space. The authors propose different methods to store protection information, one of which is a permission table based on memory segments similar to the SecCells proposition. However, MMP requires a privileged supervisor domain to provide an API for permission modification, whereas in SecCells each compartment can grant permissions it owns to other SecDivs without requiring to trap into a more privileged supervisor such as the kernel.

Light-weight Contexts (lwCs) [24] provide a threading model similar to the one proposed by `scthreads`. Threads in userspace cooperatively switch from one to another and have their own independent system context. In the lwC design, memory isolation stems from fully independent memory mappings per context and therefore different pagetables. Thus, context switching and initialization incur significant overhead due to the creation of Copy-on-Write (CoW) mappings for resources upon lwC creation and the

TLB invalidations necessary upon lwC switching.

Intel *Memory Protection Keys (MPK)* [19] is a hardware extension that allows for 16 different protection domains in one virtual address space by tagging PTEs with a 4-bit domain ID. Park et al. try to overcome this restriction with their implementation of *libmpk* [28]. This library provides an abstraction layer on top of the hardware and an OS kernel supporting the MPK technology. The library provides virtualization of protection keys and thus an unlimited number of protection domains as well as an API for secure memory management based on the virtual protection keys. *ERIM* [41] also builds on top of Intel MPK. Since MPK provides a means to implement protection domains inside a virtual address space but is not an implementation of a full compartmentalization system on its own, ERIM leverages protection keys to provide efficient isolation within a userspace process. The system enforces controlled domain switching through specifically provided call gates as well as binary inspection and rewriting to prevent the occurrence of instructions modifying MPK information and therefore bypassing the restrictions imposed by the call gates.

Donky [33] proposes a RISC-V implementation for a protection key system similar to Intel MPK. However, Donky provides 1024 different protection domains, as there are more bits for protection keys available in the PTEs. Additionally, the authors present a compartmentalization system building on top of their protection keys. Compartment switches can only be effectuated through a domain monitor running in userspace. The monitor ensures that it is the only possible entry point to other compartments through hardware call gates. An important difference to previous systems such as ERIM as well as the SecCells approach is that a compartment can have multiple protection keys associated. In Donky, a memory address can thus have only one key assigned and compartments can have multiple keys associated with them. In SecCells, the principle is inverted such that a SecDiv has its specific identifier and cells can have multiple identifiers associated with them through the permission table.

7 Future work

Throughout this project, we implemented a fully functional version of the seL4 microkernel able to boot and run on a RISC-V system with modifications to support the SecCells design. Nevertheless, several engineering as well as research angles remain open for future work. We present these ideas and plans in the following sections.

7.1 seL4 formal verification

One of seL4's main selling points is the formal verification process that the kernel undergoes and provides guarantees for functional correctness and certain security aspects [17, 20, 21]. The extensive formal verification and the corresponding machine-checked proofs apply to the kernel's source code as well as the compiled binary executable for certain architectures, including RISC-V [15–17].

Our modifications to the seL4 kernel break the proofs established by the Trustworthy Systems group. Therefore, formalizing the interaction between seL4 and SecCells and extending the proofs to this new architecture currently remain an open project and could help widespread adoption. Across existing architectures, architecture-specific proofs for differing implementations of the page-based virtual memory model are required, the base paradigm however remains the same. We therefore mainly anticipate major adaptations being necessary for switching the virtual memory paradigm from a page-based to a range-based model.

7.2 CapDL and CAmkES support

The goal of the Capability Distribution Language (CapDL) is on the one hand to specify the distribution of capabilities in a seL4-based system to allow reasoning about security guarantees based on the specification. On the other hand, it aims to allow developers to model a capability distribution that can then be automatically set up by a bootstrapping process [23]. Thus, the usage of CapDL is not mandatory in a seL4 system but can aid in the development process for verification of specifications and automatic system setup based on the previously written CapDL specifications. Since the SecCells on seL4 project introduces new capabilities for memory ranges instead of single pages and the system behavior for memory management was altered, support for our modifications in CapDL-based systems is currently still missing.

The Component Architecture for microkernel-based Embedded Systems (CAmkES) project provides a framework for specifying a full component-based system on top of

a microkernel such as seL4 [22, 25]. The framework allows to specify the components and their interfaces with each other and therefore aims to simplify the creation of systems with multiple interleaved components for the embedded system domain. Since the framework's runtime interacts with the underlying OS, it has to be aware of the kernel's API to correctly setup and manage the modeled components. Similar to CapDL, our modifications to the capabilities and API endpoints provided by the seL4 kernel require modifications to the framework that have not been implemented yet.

Adding support for SecCells to CapDL and CAmkES is not functionally required but allows for simplified system development and therefore can support adoption of the architecture in seL4-based environments.

7.3 `scthreads` stabilization

Currently, the `scthreads` implementation of the design idea presented in Section 4.3 is functional but still in a prototype state. Also, the API has not been formalized yet.

Therefore, we identify formalizing the interface provided by `scthreads` and implementing a stable library based on those definitions as future projects. Also, the code is currently strongly entangled with the memory management API provided by the seL4 kernel due to the required setup of memory regions for safely storing the threads' contexts. Adding an abstraction layer can facilitate porting the library to other OSs running on a SecCells-enabled architecture later on.

7.4 Overlapping cells in SecCells

As mentioned in Section 4.2.2, cells in the original SecCells design could be overlapping. This approach showed to introduce architecture-independent difficulties to the implementation of memory management in software as well as the SecCells-specific instructions in hardware due to the increased complexity of the permission table. Therefore, we do not try to achieve implementations for overlapping cells on RISC-V or other SecCells-enabled architectures.

Nevertheless, overlapping cells simplify compartmentalization of userspace applications, for example by carving out parts of a big buffer in memory where the parts are accessible to one SecDiv and another compartment controls the full buffer (c.f. yellow memory regions in Figure 4.1). Therefore, an open question remains how to design and implement a library that presents cell-based memory to userspace and allows overlapping cells but splits them up where necessary to interact with the kernel and the hardware based on disjoint cells as exemplary shown in Figure 4.1.

7.5 Performance evaluation on hardware

As briefly outlined in Chapter 5, QEMU does not allow for proper performance evaluation of an emulated system. Even though we provided detailed arguments for the perfor-

mance improvements possible through SecCells, we therefore currently cannot provide numbers and measurements of actual performance evaluations.

At the time of writing, an Field-Programmable Gate Array (FPGA) implementation of a RISC-V softcore adapted to the SecCells architecture is underway but has not been finished yet. Experiments for measuring the performance impact of using isolation based on SecDivs in a single address space and having them communicate via shared buffers instead of software running in multiple address spaces and communicating via IPC mechanisms. Consequently, support for our claims and arguments through measurement data is left for future work.

8 Conclusion

In summary, our project proves the feasibility of combining a security-focused microkernel with new microarchitectural features to improve compartmentalization simplicity and performance. Compartmentalization is a declared goal of the SecCells architecture and it complements well the fine-grained memory access control provided by seL4 through capabilities. In our system, compartmentalization is a first-class citizen extending the memory isolation provided by established measures such as virtual memory addressing for processes to a more fine-granular isolation inside a single virtual address space.

Our work consequently pushes the state of the art in compartmentalization by showing that isolation of compartments in a single virtual address space via microarchitectural extensions is possible with a reasonable implementation effort in both hardware and software. We therefore provide a basis for future research with our implementation for seL4 on SecCells and pave the way for more widespread adoption of hardware-supported security in high-criticality systems where security-focused OSs such as seL4 are already deployed to improve overall reliability through attack resilience.

A RISC-V instructions and exceptions

The encodings presented below are formatted as in the RISC-V ISA manual [42, 43].

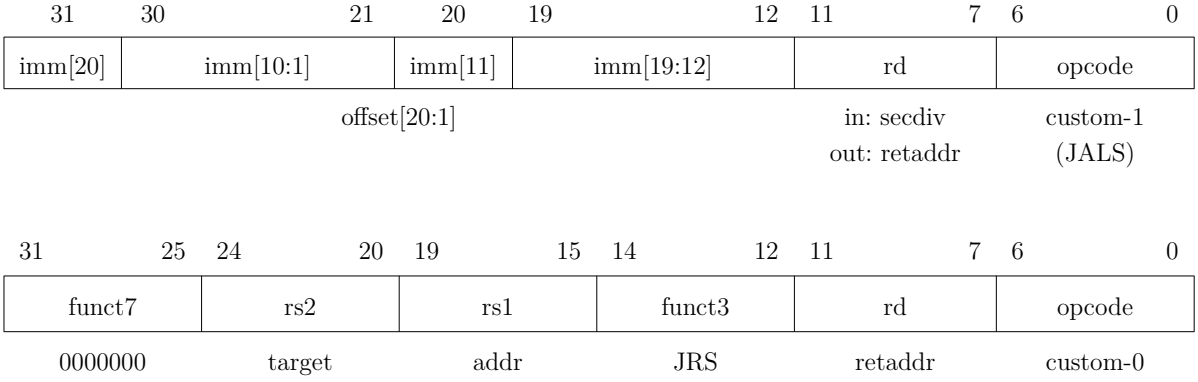

Figure A.1: Encodings for the SDSwitch instructions in RISC-V

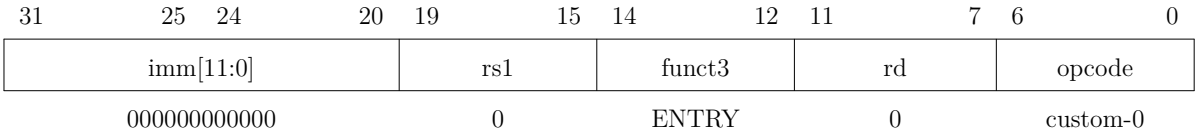

Figure A.2: Encodings for the SDEntry instruction in RISC-V

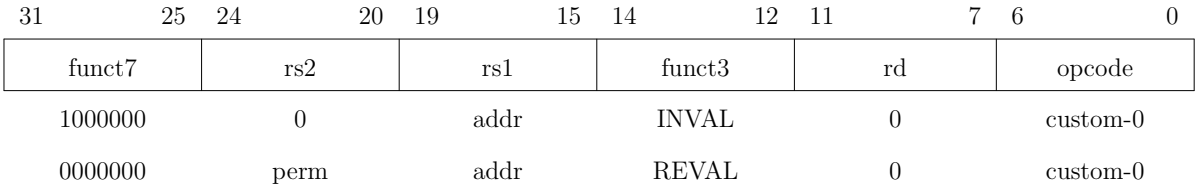

Figure A.3: Encodings for the SCInval and SCReval instructions in RISC-V

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
0000000	perm	addr	PROT	0	custom-0

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
perm[11:5]	target	addr	GRANT	perm[4:0]	custom-0
perm[11:5]	target	addr	TFER	perm[4:0]	custom-0

Figure A.4: Encodings for the SCGrant, SCTfer and SCProt instructions in RISC-V

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
0000000	perm	addr	COUNT	count	custom-0

Figure A.5: Encodings for the SCCount instruction in RISC-V

31 30 25	24 21 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[20] imm[10:1] imm[11]		imm[19:12]		rd	opcode	J-type

RV64 Xsecc Extension

offset[20:1]				rs/rd	0101011	JALS
0000000	rs2	rs1	001	rd	0001011	JALRS
000000000000		00000	010	00000	0001011	ENTRY
1000000	00000	rs1	011	00000	0001011	INVAL
0000000	rs2	rs1	011	00000	0001011	REVAL
0000000	rs2	rs1	100	00000	0001011	PROT
perm[11:5]	rs2	rs1	101	perm[4:0]	0001011	GRANT
perm[11:5]	rs2	rs1	110	perm[4:0]	0001011	TFER
0000000	rs2	rs1	111	rd	0001011	COUNT

Figure A.6: Binary encodings for all SecCells instructions in RISC-V

Table A.1: Exceptions thrown by a RISC-V CPU for newly introduced instructions

Instruction	Cause	`SCAUSE` register	`STVAL` register
`jals/jalrs`	Target ID 0 or too high	`INV_SDID = 0x1a`	Target ID
	Target instruction not `entry`	`ILL_TGT = 0x1c`	Target address
`inval`	Illegal address	`ILL_ADDR = 0x18`	Requested address
	Cell already invalid	`INV_CELL_STATE = 0x1b`	0
`reval`	Illegal address	`ILL_ADDR = 0x18`	Requested address
	Cell already valid	`INV_CELL_STATE = 0x1b`	1
	Illegal permissions	`ILL_PERM = 0x19`	`(type « 8) | perms` where `type` = 0 if perm not in RWX 1 if empty perms given
`grant/tfer`	Illegal address	`ILL_ADDR = 0x18`	Requested address
	Target ID 0 or too high	`INV_SDID = 0x1a`	Target ID
	Cell invalid	`INV_CELL_STATE = 0x1b`	0
	Illegal permissions	`ILL_PERM = 0x19`	`(type « 8) | perms` where `type` = 0 if perm not in RWX 1 if empty perms given 2 if current SecDiv has insufficient perms 3 if target SecDiv already has the perms
`prot`	Illegal address	`ILL_ADDR = 0x18`	Requested address
	Cell invalid	`INV_CELL_STATE = 0x1b`	0
	Illegal permissions	`ILL_PERM = 0x19`	`(type « 8) | perms` where `type` = 0 if perm not in RWX 2 if current SecDiv has insufficient perms
`count`	Illegal address	`ILL_ADDR = 0x18`	Requested address
	Cell invalid	`INV_CELL_STATE = 0x1b`	0
	Illegal permissions	`ILL_PERM = 0x19`	`(type « 8) | perms` where `type` = 0 if perm not in RWX 1 if empty perms given

Bibliography

[1] Martín Abadi et al. "Control-Flow Integrity". In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS '05. Alexandria, VA, USA: Association for Computing Machinery, 2005, pp. 340–353. ISBN: 1595932267. DOI: 10.1145/1102120.1102165.

[2] Aleph One. "Smashing The Stack For Fun And Profit". In: *Phrack* 7 (49 1996). URL: http://phrack.org/issues/49/14.html#article (visited on 07/09/2021).

[3] ARM, ed. *ARM Cortex-A Series - Programmer's Guide for ARMv8-A*. Version 1.0. 24th Mar. 2015.

[4] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 41.

[5] Peter Chubb, Gerwin Klein and Kent McLeod. *Frequently Asked Questions on seL4*. Sept. 2021. URL: https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html (visited on 14/09/2021).

[6] Scott Constable, Arash Sahebolamri and Steve Chapin. "Extending seL4 Integrity to the Genode OS Framework". In: (2017).

[7] CSIRO. *Statement regarding D61 changes*. 28th May 2021. URL: https://www.csiro.au/en/news/News-releases/2021/D61-statement (visited on 13/09/2021).

[8] Norman Feske. *Genode Operating System Framework Foundations 21.05*. May 2021. URL: https://genode.org/documentation/genode-foundations-21-05.pdf.

[9] Fuchsia Developers. *Fuchsia Documentation*. Google. Sept. 2021. URL: https://fuchsia.dev/fuchsia-src/ (visited on 18/09/2021).

[10] Anny Gakhokidze. *Introducing Firefox's new Site Isolation Security Architecture*. Mozilla Corporation. 18th May 2021. URL: https://hacks.mozilla.org/2021/05/introducing-firefox-new-site-isolation-security-architecture/ (visited on 16/09/2021).

[11] Jayneel Gandhi et al. "Range Translations for Fast Virtual Memory". In: *IEEE Micro* 36.3 (May 2016), pp. 118–126. DOI: 10.1109/mm.2016.10.

[12] Siddharth Gupta et al. "Rebooting Virtual Memory with Midgard". In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, June 2021. DOI: 10.1109/isca52012.2021.00047.

[13] Gernot Heiser. *"Trustworthy Systems Research is Done" – Are You Kidding, CSIRO?* 31st May 2021. URL: https://microkerneldude.wordpress.com/2021/05/31/trustworthy-systems-research-is-done-are-you-kidding-csiro/ (visited on 13/09/2021).

[14] Gernot Heiser. *How to (and how not to) use seL4 IPC.* 7th Mar. 2019. URL: https://microkerneldude.wordpress.com/2019/03/07/how-to-and-how-not-to-use-sel4-ipc/ (visited on 13/09/2021).

[15] Gernot Heiser. *seL4 Integrity Enforcement Proved for RISC-V.* 4th Aug. 2021. URL: https://microkerneldude.wordpress.com/2021/08/04/sel4-integrity-enforcement-proved-for-risc-v/ (visited on 19/09/2021).

[16] Gernot Heiser. *seL4 on RISC-V Verified to Binary Code.* 5th May 2021. URL: https://microkerneldude.wordpress.com/2021/05/05/sel4-on-risc-v-verified-to-binary-code/ (visited on 19/09/2021).

[17] Gernot Heiser. *The seL4 Microkernel - An Introduction.* Version 1.2. The seL4 Foundation, 10th June 2020. URL: https://sel4.systems/About/seL4-whitepaper.pdf (visited on 01/04/2021).

[18] Intel Corporation. *5-Level Paging and 5-Level EPT.* Tech. rep. Version 1.1. May 2017. URL: https://software.intel.com/content/www/us/en/develop/download/5-level-paging-and-5-level-ept-white-paper.html (visited on 09/09/2021).

[19] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4.* June 2021. URL: https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html (visited on 09/09/2021).

[20] Gerwin Klein et al. "Comprehensive formal verification of an OS microkernel". In: *ACM Transactions on Computer Systems* 32.1 (Feb. 2014), pp. 1–70. DOI: 10.1145/2560537.

[21] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09.* ACM Press, 2009. DOI: 10.1145/1629575.1629596.

[22] Ihor Kuz et al. "CAmkES: A component model for secure microkernel-based embedded systems". In: *Journal of Systems and Software* 80.5 (May 2007), pp. 687–699. DOI: 10.1016/j.jss.2006.08.039.

[23] Ihor Kuz et al. "capDL: A Language for Describing Capability-Based Systems". In: *Proceedings of the first ACM asia-pacific workshop on Workshop on systems - APSys '10.* ACM Press, 2010. DOI: 10.1145/1851276.1851284.

[24] James Litton et al. “Light-Weight Contexts: An OS Abstraction for Safety and Performance”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 49–64. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton.

[25] Kent McLeod and Gerwin Klein. *CAmkES*. Nov. 2020. URL: https://docs.sel4.systems/projects/camkes/ (visited on 19/09/2021).

[26] Zeyu Mi et al. “SkyBridge: Fast and Secure Inter-Process Communication for Microkernels”. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, Mar. 2019. DOI: 10.1145/3302424.3303946.

[27] Cheol Ho Park and Daeyeon Park. “Increasing TLB reach with multiple pages size subblocks”. In: *Conference Proceedings of the IEEE International Performance, Computing, and Communications Conference (Cat. No.02CH37326)*. IEEE, 2002. DOI: 10.1109/ipccc.2002.995143.

[28] Soyeon Park et al. “libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)”. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 241–254. ISBN: 978-1-939133-03-8.

[29] Binh Pham et al. “Increasing TLB reach by exploiting clustering in page translations”. In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Feb. 2014. DOI: 10.1109/hpca.2014.6835964.

[30] QEMU Developers. *QEMU – A generic and open source machine emulator and virtualizer*. Software Freedom Conservancy, Inc. 2021. URL: https://www.qemu.org (visited on 13/09/2021).

[31] Charles Reis and Alexander Moshchuk. *Protecting more with Site Isolation*. Google. 20th July 2021. URL: https://security.googleblog.com/2021/07/protecting-more-with-site-isolation.html (visited on 16/09/2021).

[32] Charles Reis, Alexander Moshchuk and Nasko Oskov. “Site Isolation: Process Separation for Web Sites within the Browser”. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1661–1678. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/reis.

[33] David Schrammel et al. “Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1677–1694. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel.

[34] seL4 Project. *The seL4 Foundation*. LF Projects, LLC. 2020. URL: http://sel4.systems/Foundation/ (visited on 10/09/2021).

[35] Vedvyas Shanbhogue, Deepak Gupta and Ravi Sahita. "Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity". In: *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, June 2019. DOI: 10.1145/3337167.3337175.

[36] Simon Shields, Gerwin Klein and Axel Heider. *Elfloader*. seL4 Foundation. Sept. 2021. URL: https://docs.sel4.systems/projects/elfloader/ (visited on 13/09/2021).

[37] Mark Swanson, Leigh Stoller and John Carter. "Increasing TLB reach using superpages backed by shadow memory". In: *ACM SIGARCH Computer Architecture News* 26.3 (June 1998), pp. 204–213. DOI: 10.1145/279361.279388.

[38] László Szekeres et al. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy*. IEEE, May 2013. DOI: 10.1109/sp.2013.13.

[39] The MITRE Corporation. *2021 CWE Top 25 Most Dangerous Software Weaknesses*. 26th July 2021. URL: http://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html (visited on 19/09/2021).

[40] Trustworthy Systems. *The seL4 microkernel*. University of New South Wales. 2021. URL: https://trustworthy.systems/projects/seL4/ (visited on 10/09/2021).

[41] Anjo Vahldiek-Oberwagner et al. "ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1221–1238. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner.

[42] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Version 20191213. RISC-V Foundation. Dec. 2019.

[43] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Version 20190608-Priv-MSU-Ratified. RISC-V Foundation. June 2019.

[44] Robert N. M. Watson et al. *An Introduction to CHERI*. Tech. rep. University of Cambridge Computer Laboratory, July 2020.

[45] Western Digital Corporation et al. *RISC-V Open Source Supervisor Binary Interface (OpenSBI)*. Version 0.9. RISC-V International, 2021. URL: https://github.com/riscv-software-src/opensbi (visited on 13/09/2021).

[46] Emmett Witchel, Josh Cates and Krste Asanović. "Mondrian memory protection". In: *ACM SIGPLAN Notices* 37.10 (Oct. 2002), pp. 304–316. DOI: 10.1145/605432.605429.

[47] Jonathan Woodruff et al. "The CHERI capability model: revisiting RISC in an age of risk". In: *ACM SIGARCH Computer Architecture News* 42.3 (14th June 2014), pp. 457–468. DOI: 10.1145/2678373.2665740.

[48] Huaisheng Ye. *Introduction to 5-Level Paging in 3rd Gen Intel Xeon Scalable Processors with Linux*. Tech. rep. Lenovo Press, 23rd May 2021. URL: https://lenovopress.com/lp1468-introduction-to-5-level-paging (visited on 09/09/2021).

Declaration of Academic Honesty

Hereby, I declare that I have composed the presented paper independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such.

This paper has neither been previously submitted to another authority nor has it been published yet.

Palaiseau, September 20, 2021

(Florian Hofhammer)