École Polytechnique Fédérale de Lausanne

Memory Model Adaptation
replacing the virtual memory model of a kernel

by Basil Ottinger

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

-
External Expert

Andres Sanchez
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 10, 2022

Acknowledgments

I thank Andres Sanchez for supervising my project, for his patience, support, and sound advice, and for the enlightening and interesting discussions we had. I thank Prof. Mathias Payer for providing me with this opportunity and supervising this project.

Lausanne, June 10, 2022 Basil Ottinger

Abstract

Compartmentalization is an important principle in today's software security policies. In recent years, there have been several proposals for compartmentalization mechanisms within a single virtual address space. This project takes up one such proposal and discusses the methods and procedure to replace a traditional virtual memory model based on paging by this proposed model.

Contents

Chapter 1

Introduction

1.1 Motivation

The principle of least privilege is an important design policy in software security. Each component of a system should operate with the least amount of privilege that is required to accomplish its task. Most importantly, this principle helps to limit the damage a malicious or malfunctioning component can inflict. It also minimizes the amount of components that might be responsible if a misuse or misbehavior is detected and thus reduces the number of components that need auditing.

A requirement to implement the principle of least privilege is compartmentalization, i.e. the ability to separate components from each other such that they have distinct privilege. To that end, we isolate different parts of a system at different granularity. Examples of isolation are virtual machines, containers (e.g. Docker), and system processes.

Naturally, the finer the granularity of separation, the better we can assign and track different privilege levels. Optimally, we can assign different levels of privilege to different parts of code running in the same virtual address space. For example, looking at an email client that is implemented as a single component, the entire code would need the privilege for network I/O. If the code were found to be vulnerable, the entire codebase would need to be audited. If, however, network, local file management, and GUI were separated, the fault or vulnerability would be confined to the network component as the other components lack the privilege to access network I/O.

Why does the granularity of system processes not suffice? Security is often expected to have practically zero overhead. Isolating components using processes requires an expensive context switch, involving saving the current context, updating the memory datastructure, restoring the context of the next process, and additional administrative processing by the kernel. These operations introduce a significant overhead.

A recent proposal, called SecureCells, presents a clean-slate memory model that introduces radical architectural changes to provide a mechanism for compartmentalization *within* the same address space. It achieves this by tracking a set of permissions for different compartments and allowing virtual memory mappings at an arbitrary resolution, even much smaller than the typical page size. The proposal even allows for dynamic granting of permissions between compartments providing a large degree of flexibility.

1.2 Goal

In the context of this semester project, we worked on adapting the memory model of an existing kernel (classical paging) to use a simplified version of the SecureCells model. While inspired by the topics of compartmentalization, we focus on explaining the methodology and process of replacing the memory model of a simple kernel.

To a reader with experience in this field, we recommend reading chapter 3 on design and chapter 4 on implementation. The chapter 2 provides some background and can be returned to in case something is unclear in the later chapters.

Chapter 2

Background

2.1 Unikernel

Today, virtualization and containerization is omnipresent in the cloud. Besides practical reasons, these features are often used to provide system-level isolation. Using a standard OS distribution, however, provides in many cases functionality far beyond what a given service needs. For example, a web-server might not need the audio driver or the print service that come with the distribution. This unnecessarily increases performance overhead and attack surface, increasing boot time, consuming CPU cycles, and providing more chances for exploitation.

The idea of a Unikernel is to build a kernel / OS tailored for a specific application such that it provides just the functionality required by that application. In other words, the application is packaged into an executable system image that provides exactly the functionality that is needed. Auditing and testing with such a minimal kernel is easier and more reliable, the kernel image is smaller, and there is overall less code that needs to be trusted. The term 'Library OS' refers to the framework which builds the Unikernel according to the application's needs and embeds the application.

2.2 Virtual Memory

Most of today's systems provide some sort of virtual memory abstraction. We will assume familiarity with the concept and only briefly outline some key points: By adding a layer of indirection, virtual memory allows a system to appear as though it has more memory than it actually does. The indirection makes memory management easier, avoids fragmentation of physical memory, allows better multitasking, and aids in compartmentalization / fault isolation of processes.

The most common virtual memory model used in today's system is paging, usually imple-

mented using a (sparse) hierarchical pagetable. We briefly describe Sv48, the memory model that is one of the standard models for riscv64 and the one that our Unikernel uses.

Sv48 [1] is a 4-level paging scheme, very similar to 4-level paging in x86. Like the name suggests, it provides a 48 bit virtual address space. Pages have a size of 4 kB, so the page offset is 12 bits. Virtual page numbers (VPNs) use therefore 36 bits. Physical page numbers (PPNs) use 44 bits which allows for 2^{56} bytes of physical memory. The pagetable entries (PTEs) in the last layer of the hierarchical pagetable contain the 44 bits of the PPN along with 10 bits of flags for permissions, cache control, etc. The remaining bits are reserved for future use.

The RISC-V specification defines a number of Control and Status Registers (CSRs). One of those is the 'supervisor address translation and protection' register (SATP). It is consulted by the memory management unit (MMU) and contains the PPN of the base of the current pagetable used for translation and an address space identifier (ASID). It also contains 4 bits defining the mode of operation, i.e., the scheme used to translate addresses. Options include Sv39, Sv48, and Bare (no translation). Mode 14 and 15 are reserved for custom virtual memory schemes which we will use.

Chapter 3

Design

In section 3.1, we discuss the premises and components that we used. In section 3.2, we introduce the alternative memory model we work on implementing. In section 3.3, we consider what work needs to be done to replace the current model. Finally, in section 3.4, we consider how the interface to the virtual memory manager would need to be adapted to make use of the full potential of the alternative model.

3.1 Components

As a kernel to work on, we chose Nanos [4], a Unikernel project by NanoVMs [6] as it was the only Unikernel project that implemented RISCV at the time. We chose to use a Unikernel for two reasons: first, as a Unikernel, its very idea is to reduce the overhead of separating software components into isolated domains by reducing the size of a virtual machine. And second, it is a comparatively simple kernel and hence easier to get familiar with its codebase.

Concerning the platform, the authors of the alternative memory model already modified the qemu RISCV emulator to implement the functionality of the new memory model. We were provided the emulator to run and test our kernel on. Hardware implementation was therefore not part of this project.

3.2 Target Memory Architecture

We are working on changing a Unikernel, running with Sv48 (see section 2.2), to use an alternative memory model which we will describe in a simplified manner for the purposes of this discussion.

The full memory model is called SecureCells and aims to improve compartmentalization

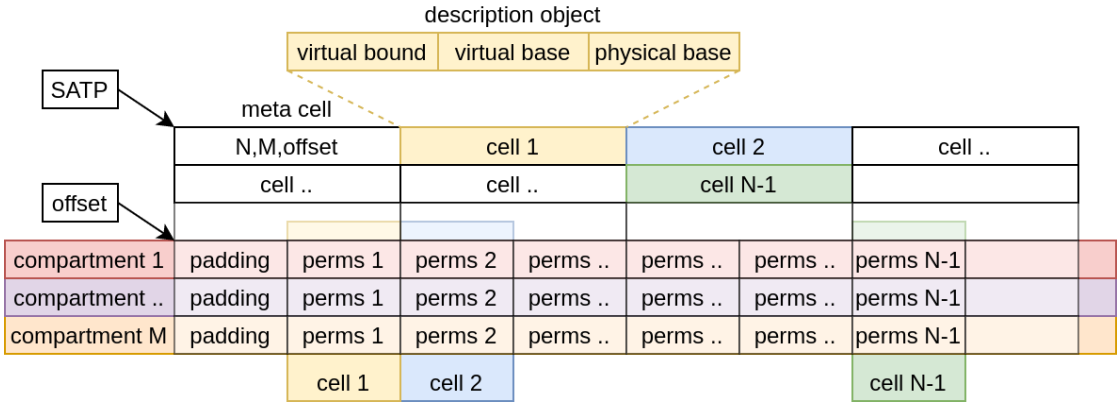

Figure 3.1: Permission table of the alternative simplified model.

performance by allowing fine-grained permission tracking within a single virtual address space and thereby alleviating the overhead of a context switch.

Instead of a hierarchical pagetable, the simplified memory model manages a datastructure we call *permission table* depicted in figure 3.1. The first half of the table contains a meta cell followed by a simple *sorted* list of description objects. The meta cell contains the number cells/mappings (N), the number of compartments (M), and the offset for the permission section. The description objects contain a virtual address range and physical address representing the base of the target physical memory. Since the list is sorted, a translation can be found efficiently by performing a binary search. At the offset in the same table, we have a set of permissions. There is a permission section for each compartment containing permission flags for each mapped range. Depending on which piece of code is executing, determined by a special register, a different replica of permissions is consulted.

A new mapping is created by inserting a new description object at the right place in the list. There might be concern that the memory requirements for the permission table might become infeasible to map the enormous virtual address space Sparse hierarchical pagetables were created to alleviate this problem after all. By having each description object map a contiguous region much larger than the typical page, this problem is overcome at the cost of potential physical memory fragmentation. The problem of fragmentation is addressed by the authors of the memory model by introducing an additional layer of indirection and we do not concern ourselves with it here.

3.3 Steps

In general, the virtual memory manager (VMM) provides the following functionality: a piece of code setting up new memory of a process first requests physical memory from the physical memory manager. It then passes the physical memory address together with the desired virtual

address range to the VMM. On initialization, the VMM is provided with memory to hold the translation datastructure and is responsible for writing the mapping into that datastructure used by the memory management unit (MMU) for translation.

Good software is built with modularity and maintainability in mind. Different pieces of code should be decoupled and only interact through a well-defined interface. To re-implement a certain functionality, such as virtual memory, the module can simply be replaced, as long as the interface it provides to the rest of the codebase is maintained. The basic interface consists of the following two functions along with some functions to query the state of certain entries.

```
map(vaddr, paddr, length, pageflags);
unmap(vaddr, length);
```

As our alternative memory model has strictly more functionality than Sv48, we can implement the complete interface with its datastructure. Trying to *replace* Sv48's functionality means that we will only have one compartment and hence only one set of permissions. In section 3.4, we will discuss how the interface might look like or how it would need to change to make use of the full functionality.

What remains to be done by the kernel is to initialize the MMU. When programming on a higher layer of abstraction, replacing the implementation while maintaining the interface would be enough. Since we are operating at a level where hardware is involved, there is some additional work. The VMM is only in charge of setting up and populating the datastructure (page or permission table) which is then referenced by the MMU to do the translation. But the MMU must be initialized in such a way that it translates addresses according to the alternative memory model rather than Sv48. Finally, the MMU implementation, i.e. the hardware or emulator would would need to be adapted as well but this is beyond the scope of this project and we are provided with an emulator.

In summary, there are two parts that need adaption: the implementation of the VMM and the MMU initialization.

3.4 Adaptation Considerations

While replacing paging by a memory model with strictly more functionality, some questions rise concerning the interface. The difference stems from the fact that the new memory model keeps track of not one, but M sets of permissions (one set per compartment). The original signature (see section 3.3) only takes one set of pageflags as a parameter for creating a new mapping. For our work, we maintained the same prototype and only modified the permissions for a single compartment that we keep track of. To make use of the full potential of the alternative model, the map function would either need to receive an array of permission flags, indexed by the

compartment id:

```
map(vaddr, paddr, length, cellflags[]);
```

Alternatively, the map function could dynamically check what the current value of the compartment id register is and set minimal permissions for all other compartments. It might also make sense to provide an interface with both versions: one for the loader setting up the execution environment initially and the other to be called called by the executing code if more memory is required at runtime.

Chapter 4

Implementation

In section 4.1, we briefly describe the toolbox for studying, running, and debugging the project. In section 4.2, we describe the VMM implementation of the original Unikernel before any modification. Finally, in section 4.3, we outline how the new memory model can be fitted within the bounds of the old model.

4.1 Toolbox

There are two components needed: resources for understanding the codebase and software for running and debugging the code. For familiarizing with the codebase, clean, well-understandable code and good documentation or tutorials (blog posts, etc.) are useful. In our case, the code was understandable but documentation was sparse. Blog posts such as [5] were very enlightening. Also of great value is a lively and active developer community. In this, we were fortunate and the developers of Nanos were fast to respond and provided us with help in setting up the toolchain.

For running and debugging the code, we need a way to know what's going on. Unit testing for datastructures is straight forward. For runtime testing, the simplest way are early debug print statements. Tricky bugs, however, require a deeper view. Since we are running on qemu, one way is to use qemu's logging capabilities to see the machine state. Another option is to get a debugger like gdb running and connect to the emulator.

Since we are working on a RISCV architecture, we need the toolchain and binutils for RISCV. We also need a gdb version compiled for RISCV.

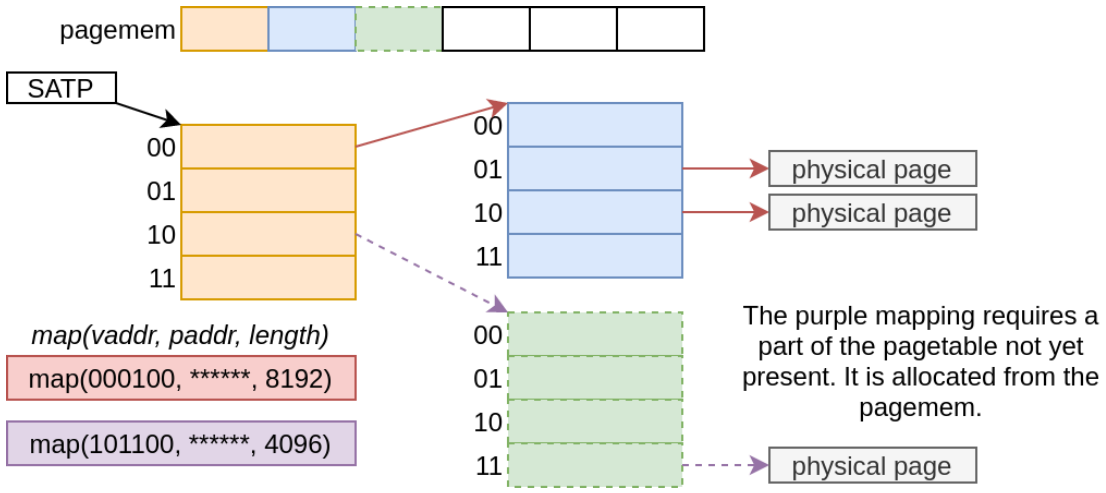

Figure 4.1: pagemem allocation of new table page.

4.2 Virtual Memory with Sv48

Using Sv48, the Unikernel initializes the VMM by providing a physical memory region to the VMM to manage. We call that region `pagemem`. The VMM uses it to allocate memory for holding the pagetable. With Sv48, immediately on initialization, the VMM allocates the root pagetable page and returns the table-base. After that, whenever a new mapping requires a part of the pagetable that has not been allocated yet, the VMM allocates a new table page from the `pagemem`. We provide a small example of a two layer paging system with 4 entries per page in figure 4.1.

The `pagemem` memory region is a contiguous range of physical memory. Normally, in a typical multi-process OS, allocated table-pages would be used in different pagetables depending on different values of the SATP register. In our case, using the Unikernel, the SATP is assigned the address of the root pagetable only once when the VMM is initialized and remains unchanged throughout the execution.

4.3 Virtual Memory with the New Model

Since the alternative memory model introduces such a deep and radical change to the system, and since the hardware (or emulator) is not standard RISCV, it makes sense to introduce the modifications as a separate platform, alongside x86, aarch64, and riscv64.

The implementation of the alternative memory model does not have a hierarchy of pages for the mapping but consists of one big table, depicted in figure 4.2. Since memory allocated from the pagemem is contiguous in physical memory, the idea is to allocate a big chunk spanning multiple pages when the VMM is initialized, i.e., when normally only the root pagetable page would be

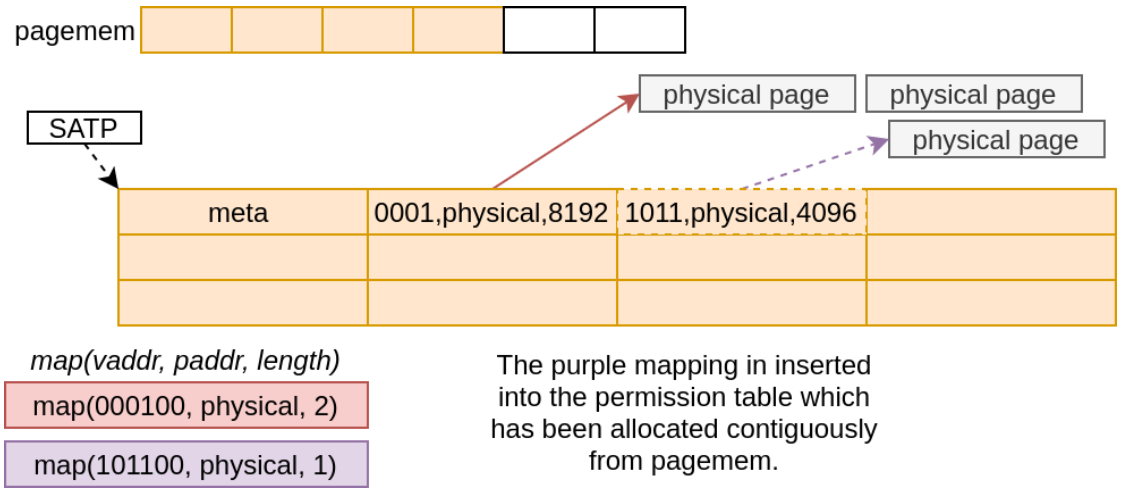

Figure 4.2: Adding a mapping to the permission table.

allocated. When a new mapping is created, it would simply be inserted into the permission table at the correct place to keep the list sorted. Additionally, the permission flags (omitted in figure 4.2) for that cell need to be inserted in the permission section of the current compartment.

Beyond `map` and `unmap`, there are a couple of kernel components that query the paging system for certain information (e.g. check if a page is marked dirty). In our Unikernel, this functionality was provided by a generic function called `traverse_ptes()` that would take a closure function. Throughout the kernel, these closures, implemented as C macros, were used frequently. Using the closures nicely decoupled the virtual memory management from the rest of the system as clients could define their own closure functions and pass them to the generic pagetable traversal function. As the closure implementation was rather obscure to our eyes, we would have opted to implement the queries as separate functions rather than trying to work out the closures since there were only a handful of clients.

Chapter 5

Evaluation

Implementation of the VMM and permission table datastructure was straight forward. We implemented `map` and `unmap` along with some helper functions e.g. for finding the cell index for a given virtual address. We performed thorough unit testing to verify the datastructure works as intended.

Implementing the MMU initialization on the other hand turned out to be trickier than expected. The code would initialize the VMM, set up the permission table datastructure and create three initial mappings depicted in figure 5.1: an identity mapping for accessing the permission table (non-executable), a kernel mapping for running the kernel code, and a device mapping (omitted in the figure). It would proceed to set the MMU mode and the permission table address in the SATP register. After that, a memory fence instruction is required to put all changes to the permission table into effect.

At this point, the system tries to translate an address that causes repeated page faults. Some digging revealed that translation fails on an address that is mapped by the identity mapping which is non-executable. The translation fails because the access is for an instruction fetch which requires the mapping to be executable. Access to the code is given by the kernel code mapping, which is executable, and we verified that the program counter (PC) is in the range of the kernel mapping prior to the memory fence instruction.

Due to time constraints, we are not able to say with certainty why the system attempts to fetch an instruction by its physical address rather than through the kernel mapping. We deem it likely that the error is related to differences in implementation between the kernel implementation and the underlying emulator. Some change to the memory fence instruction might have been introduced during the adaptation of the emulator as the original emulator worked fine. Also, changing requirements due to concurrent work on the model specification made the matter more difficult.

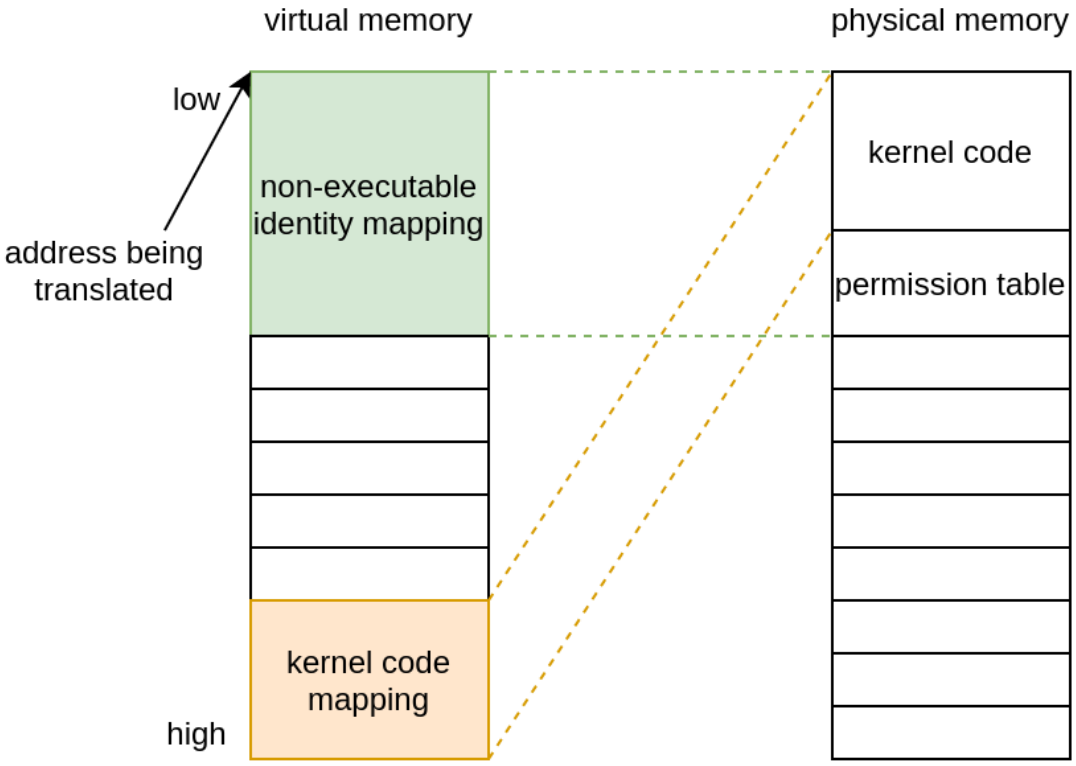

Figure 5.1: Mappings at the end of MMU initialization.

Chapter 6

Related Work

There are many techniques to approach compartmentalization. Ideas include leveraging Trusted Execution Environments (TEE), using virtualization, runtime checks (Software-based Fault Isolation), or memory-safe languages. We focus on OS and architecture-based techniques.

In general OSes, processes abstract the isolation, privilege, and execution state. However, context switching is expensive. Lightweight Contexts (lwC)[3] introduce a new OS abstraction that reduces the cost of context switches. It separates privilege and memory within the same process but each lwC has its own virtual address space, set of page mappings, file descriptor bindings, and credentials. A switch only requires change of virtual memory mapping, which is achieved by tagging entries in the TLB[1]. Threads are independent across different lwC.

Improving performance is achieved by using hardware mechanisms. One mechanism is called hardware memory tagging. The idea is to use a set of keys to divide memory into different regions. Intel Memory Protection Key (MPK) and ARM Memory Tagging Extension (MTE) implement this mechanism. While it is fast, a common criticism is that the keys have too little entropy and keys are reused frequently. Several projects like libmpk [7], CryptoMPK [2], and ERIM [8] provide software frameworks to leverage Intel's MPK. They try to alleviate the key entropy problem by e.g. virtualizing the protection keys and they aim at reducing the overhead of lwC even further.

Other projects, like CHERI[11], introduce their own hardware mechanism to accommodate their needs. CHERI is a capability model. It is designed to work with current paging mechanisms and extend them which is one of its great advantages.

Finally, SecureCells, unpublished as of this writing, is the paper our work was based on. It introduces a completely new memory model as an alternative to paging, baking into the design a mechanism for compartmentalization. Unlike CHERI, it also provides support for dynamic granting of permissions between compartments.

[1] Translation Look-aside Buffer: the cache for memory address translation.

Towards providing support for compartmentalization in OSes, the authors of CHERI have developed a prototype software stack including a modified LLVM-based compiler and FreeBSD OS. Earlier examples include Mondrix[10], a Mondrian Memory Protection[9] based Linux kernel adaptation.

Chapter 7

Conclusion

Replacing the memory model of a kernel is in theory straight forward. It consists of two parts: initializing the MMU to perform translation according to the new model and implementing the datastructure the model dictates. The memory model should provide at least the same amount of functionality. By maintaining the interface and providing the same utilities, very little work has to be done.

To reflect on future work, we believe that SecureCells and CHERI largely suffice for solving the problem of an underlying mechanism to provide fine-grained compartmentalization within a virtual address space. The urgent question is how to make these mechanisms accessible and widely used; whether that be through annotations by the developer and instrumentation by the compiler, or by means of some automatic transformation tool.

Bibliography

[1] RISC-V Foundation. *Supervisor-Level ISA, Version 1.12*. 2022. URL: https://www.five-embeddev.com/riscv-isa-manual/latest/supervisor.html (visited on 06/07/2022).

[2] Xuancheng Jin, Xu Xiao, Songlin Jia, Wang Gao, Dawu Gu, Hang Zhang, Siqi Ma, Zhiyun Qian, and Juanru Li. "Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK". In: 2021.

[3] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnogomi, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. "Light-weight Contexts: An OS Abstraction for Safety and Performance". In: Nov. 2016.

[4] NanoVMs. *Nanos Github Repository*. 2022. URL: https://github.com/nanovms/nanos (visited on 06/07/2022).

[5] NanoVMs. *Nanos on the 64-bit RISCV Architecture*. 2022. URL: https://nanovms.com/dev/tutorials/nanos-and-riscv (visited on 06/09/2022).

[6] NanoVMs. *nanovms.com*. 2022. URL: https://nanovms.com/ (visited on 06/07/2022).

[7] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. *libmpk: Software Abstraction for Intel Memory Protection Keys*. 2018. DOI: 10.48550/ARXIV.1811.07276. URL: https://arxiv.org/abs/1811.07276.

[8] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. *ERIM: Secure, Efficient In-process Isolation with Memory Protection Keys (MPK)*. 2018. DOI: 10.48550/ARXIV.1801.06822. URL: https://arxiv.org/abs/1801.06822.

[9] Emmett Witchel, Josh Cates, and Krste Asanovi. "Mondrian Memory Protection". In: *SIGOPS Oper. Syst. Rev.* 36.5 (Oct. 2002), pp. 304–316. ISSN: 0163-5980. DOI: 10.1145/635508.605429. URL: https://doi.org/10.1145/635508.605429.

[10] Emmett Witchel, Junghwan Rhee, and Krste Asanovi. "Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection". In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. Brighton, United Kingdom: Association for Computing Machinery, 2005, pp. 31–44. ISBN: 1595930795. DOI: 10.1145/1095810.1095814. URL: https://doi.org/10.1145/1095810.1095814.

[11] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. “The CHERI Capability Model: Revisiting RISC in an Age of Risk”. In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), pp. 457–468. ISSN: 0163-5964. DOI: 10.1145/2678373.2665740. URL: https://doi.org/10.1145/2678373.2665740.