Capability-Aware Operating System Design and ZenOS

by

Peter Moore

A Thesis
Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved October 2024 by the
Graduate Supervisory Committee

Michel Kinsy, Chair
Adil Ahmad
Dhiego Andrade

ARIZONA STATE UNIVERSITY
December 2024

ABSTRACT

Memory safety is one of the most important issues in software today. It encompasses a wide range of security vulnerabilities, including several that are consistently ranked highly in MITRE's list of the most dangerous software weakness (2023). These vulnerabilities have been documented and fought against since at least 1972 (Meer *et al.*, 2010). However, previous solutions have failed to prevent memory corruption attacks because they fail to address the underlying structural issues with how we access memory (Saito *et al.*, 2016). In *A Path Toward Secure and Measurable Software*, the White House's Office of the National Cyber Director described reducing memory safety vulnerabilities at scale as an important "fundamental shift" (2024).

Capabilities represent such a fundamental shift in how memory is accessed. A capability architecture can provide guarantees about memory safety by checking an unforgeable access token on every memory access. This ensures that every actor accessing a memory location is either the owner of the location or has received permission from the owner to perform that access.

In order to ensure that the security guarantees provided by capabilities are held throughout the software stack, a capability-aware ecosystem is needed. One vital piece of any such ecosystem is an operating system. The OS is responsible for managing all the hardware and software in a system. As such, the OS must have a great amount of control over the system's hardware and software. Capabilities provide a higher degree of control than is currently possible.

This dissertation proposes a capability-aware operating system, and analyzes the work needed to harden and optimize such a system. It describes the various roles of the modern OS, and investigates how those roles might be changed by capabilities. Finally, it discusses ZenOS, an implementation of and foundation for a capability-aware OS that has been designed RISC-V non-atomic capabilities.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Memory Corruption Vulnerabilities

Every year, MITRE organizes a list of the top 25 most dangerous software weaknesses, and in 2023 four of the top twenty weaknesses were related to memory corruption.In its 2024 report *A Path Toward Secure and Measurable Software*, the Office of the National Cyber Director described reducing memory safety vulnerabilities at scale as one of the "fundamental shifts" needed to improve cybersecurity quality across the ecosystem. Additionally, this is not a new problem in the field. These problems have been documented for decades, with the first documented example as early as 1972 (Meer *et al.*, 2010). In this document, the author describes what we would now call an out-of bounds write, in which an attacker can write to regions of memory outside of the region that was intended (Anderson). Figure 1.1 shows a simple example of this kind of attack. Out-of-bounds write is still ranked as the most dangerous software weakness today (MITRE, 2023).

These vulnerabilities can be caused by a wide range of mistakes. Generally, they are caused by incorrectly handling memory access. This kind of mistake is easy to make in languages that give the programmer direct, low-level access to RAM, such as C. When programmers create buffers, they do so with intended use cases, base, and bounds. A vulnerability can occur when these intended restrictions are not stated explicitly by the programmer or not enforced properly by lower-level abstraction levels. This high-level problem - *Improper Control of a Resource Through its Lifetime*, covers a wide range of vulnerabilities that can have severe impacts on the security and

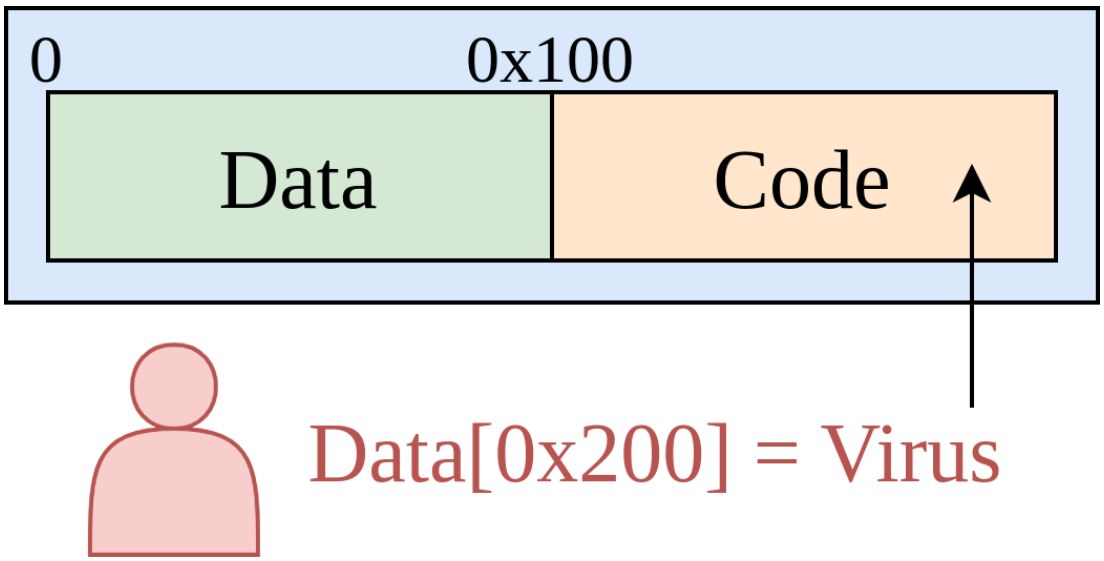

Figure 1.1: Attackers Can Utilize an Out-Of-Bounds Write Vulnerability to Cause Malicious Code to be Executed.

stability of a computing system (MITRE, 2024). As such, a number of protections have been developed to attempt to control resources and make it more difficult for attackers to use any improperly controlled resources to gain control of a system.

1.2 Current Solutions

1.2.1 Software Solutions

1.2.1.1 Stack Canaries

One such software solution is to implement stack canaries. With a stack canary, a randomized value (the canary) is placed into the stack just below the return pointer. When the return pointer is to be used, that value is checked. If the canary has been changed, that signals to the operating system that an illegal write has occurred, and the program can be killed before an attacker can gain control of it (Wagle *et al.*, 2003).

There are a few ways for an attacker to defeat stack canaries, especially because

while they mitigate the effects of a memory vulnerability, they do not prevent one from occurring. If an attacker can execute an illegal read (such as a format string vulnerability), they might be able to leak the canary value, allowing them to overwrite the return pointer without triggering the program to be killed. Stack canaries also only protect the return pointer, so other methods of jumping to malicious code such as altering the exception handler pointer may be used to alter the control flow (Lemmens, 2021).

1.2.1.2 Address Space Layout Randomization (ASLR)

ASLR was introduced to increase the difficulty of jumping to malicious code. With this scheme enabled, the sections of a program are placed into memory in random locations, thus making it difficult for an attacker to guess the location of their malicious code.

When implemented correctly, this scheme can be very effective at increasing the difficulty of memory corruption attacks. However, there are still ways around it, and many modern implementations of this scheme are insecure. For example, by default Windows and Android both choose random locations for different sections at boot and use the same location for every program that is run. In many cases, a failed attempt to break ASLR will result in the program crashing or being halted, in theory making this protocol secure. However, an attacker could use a 'dummy' program to determine the location of different segments of code, and allow such a program to crash or be marked as potentially dangerous. Then, once the different segments have been located, the attacker can attack the target program without needing to guess its randomization.

1.2.1.3 Memory Safe Languages

Many memory-safe languages can also effectively prevent memory corruption vulnerabilities. One of the most popular memory-safe languages today is Rust, which has an ownership-based resource management model. If programmers follow this model, their code is guaranteed to be memory safe. Code that breaks this model is allowed in Rust, but it must be explicitly marked as unsafe. This hopefully ensures that the programmer writing it will be aware of its danger and ensure different word that it is not exploitable. While this solution can be effective, it has a high price of entry. Many legacy codebases cannot be rewritten entirely in a new language, and yet still perform crucial tasks. A solution to help these programs be memory safe is needed.

1.2.2 Hardware Solutions

1.2.2.1 NX Bits

One hardware solution introduced in 2004 is to mark memory pages that are writable as non-executable, and vice versa. This marking is done using some unused bits in the virtual to physical memory address translation process; the processor checks those bits before writing or executing memory.

While this defense can be effective at preventing an attacker from writing an exploit to memory and then jumping to it, it cannot prevent attacks such as Return-Oriented Programming (ROP). In this type of attack, the attacker jumps to code that already exists in memory and utilizes these 'gadgets' to perform some attack. For example, one type of attack might utilize code within *glibc*, a library of code used by many programs to simplify certain operations, to create a new process. As *glibc* is already in many applications' executable memory, NX bits do not prevent this attack. Additionally, due to limitations in memory address translation, these

defenses are generally not granular, with a minimum size of 4KiB in most systems.

1.2.2.2 Pointer Authentication

In a pointer authentication scheme, used by ARM and Apple, every pointer contains not only a memory location but a hash of that memory location that is generated when the program is launched (Liljestrand *et al.*, 2019). When the pointer is to be used, its hash is verified. An incorrect hash will cause the program to be aborted, as it indicates an illegal write to the pointer. In theory, an attacker's only way to maliciously edit pointers would be to guess what the hash of the pointer is. As an incorrect guess causes the program to be aborted and the hashes are generated when the program is launched, it is very difficult to correctly guess these hashes.

A recent attack called PACMAN sidesteps this scheme by using micro-architectural side-channels. These side-channels can be used to speculatively check a pointer's hash and observe the exception handler within the Central Processing Unit (CPU) to determine if the hash was correct, without causing a crash if it is not (Ravichandran *et al.*, 2022). This allows an attacker to attempt many different PACs without restarting, and Ravichandran *et al.* were able to successfully demonstrate several attacks on Apple's M1 SOC.

1.3 Proposed Solution

Given that improper resource control problems have existed for decades, been combatted by researchers and developers for that time, and continue to cause dangerous and frequently-abused vulnerabilities, we need to fundamentally change how we control resources in order to fix these issues. This new method of resource control needs to allow the programmer to explicitly state their intentions for a resource, and ensure that every abstraction below the programmer enforces those intentions. It

must be secure and robust, to ensure that it cannot be sidestepped like some of the previous attempts. It must be performant, to ensure that the security it provides is worth the performance it requires. It must be able to scale to large-scale systems and be flexible enough to be useful on the edge. It must also be able to interface with legacy systems and provide them with some security guarantees without requiring a complete rewrite.

Capabilities meet all of those requirements. They provide a method for programmers to be clear about their intent for objects, as well as a framework for hardware to enforce those intents. Capability-aware tools for the abstraction layers between the programmer and the hardware, such as the compiler and the operating system, are needed to ensure that metadata are passed without modification or leaks. These abstraction layers will also benefit from the improved security of being capability aware. Additionally, without these tools, it will be difficult or impossible for developers to build and users to use applications secured by capabilities. This thesis seeks to discover the benefits and drawbacks of a capability-aware operating system. It will explore the role of a capability-aware operating system, and pose important resource questions for the future of development of capability systems. Finally, it will describe our progress on developing ZenOS, an operating system designed to take advantage of non-atomic capabilities and serve as a foundation for future research in this field.

Chapter 2

CAPABILITIES

A capability is an unforgeable token of authority over an object. It is created by the owner of the object, who generally also provides some metadata about the use of that object that is allowed. What metadata is stored depends on the implementation of the capability system, but the object's base, bound, and what operations may be performed on the object are usually included. For example, the owner of a capability for a program's stack might specify its base and bound addresses and that it is readable and writable but not executable. Many of these systems also include parent and children lists, which might allow inheritance of permissions and are crucial for revocation of capabilities. The owner of a capability can share it with others or create a child capability to share. That child capability must have monotonically decreased permissions. For example, if the parent capability cannot be executed, the child cannot be executable. Additionally, if the owner revokes a capability, thus disallowing its use to access the owner's memory, all of its child capabilities must also be revoked.

By giving the data owner the ability to explicitly control the object's size and use cases and enforcing those rules in hardware, capabilities can obsolete this very prevalent and potent class of vulnerabilities.

2.1 Atomic Capabilities

One type of capability system treats the capability and the object as one unit that cannot be separated from each other. In such 'atomic' capability systems, additional instructions are needed to allow for manipulation of these capabilities. Capability

Hardware Enhanced RISC Instructions (CHERI) is an example of such an atomic capability system. In this system, the size of pointers is doubled and the capability for each pointer is stored in the new upper bits. This pointer is then checked by the processor whenever it is used. This allows for fine-grained memory protection in such a manner that can be blended with conventional architectures, allowing for incremental deployment within existing ecosystems. Today, versions of CHERI exist for ARM, QEMU MIPS, and RISC-V, and a prototype software stack is currently under development (Watson *et al.*, 2019).

While atomic capabilities provide a wide range of benefits over traditional memory architectures, they also have some limitations. As the capability for the pointer is attached to the pointer, if the owner wishes to revoke a capability, all memory must be scanned to invalidate each capability individually. Also, the metadata used to validate the capability is of a fixed size, which somewhat limits the flexibility of these systems.

2.2 Non-atomic Capabilities

Non-atomic capability systems seek to assuage these limitations by separating the capability and its object. In these systems, each pointer will include the address of the object as well as the address of the metadata used to validate its access. While this increases the overhead and complexity of validating memory accesses, it simplifies revocation, as only a single location needs to be marked as invalid. Additionally, the metadata can be grown or shrunk to meet an individual system's needs. Figure 2.1 shows a broad outline of the differences between atomic and nonatomic capabilities.

One such non-atomic capability system is Zeno (Ehret *et al.*, 2022). In this system, capabilities are modeled as Namespaces. The upper bits of each pointer index a Distributed Namespace Directory (DND), which contains a list of all the Namespaces

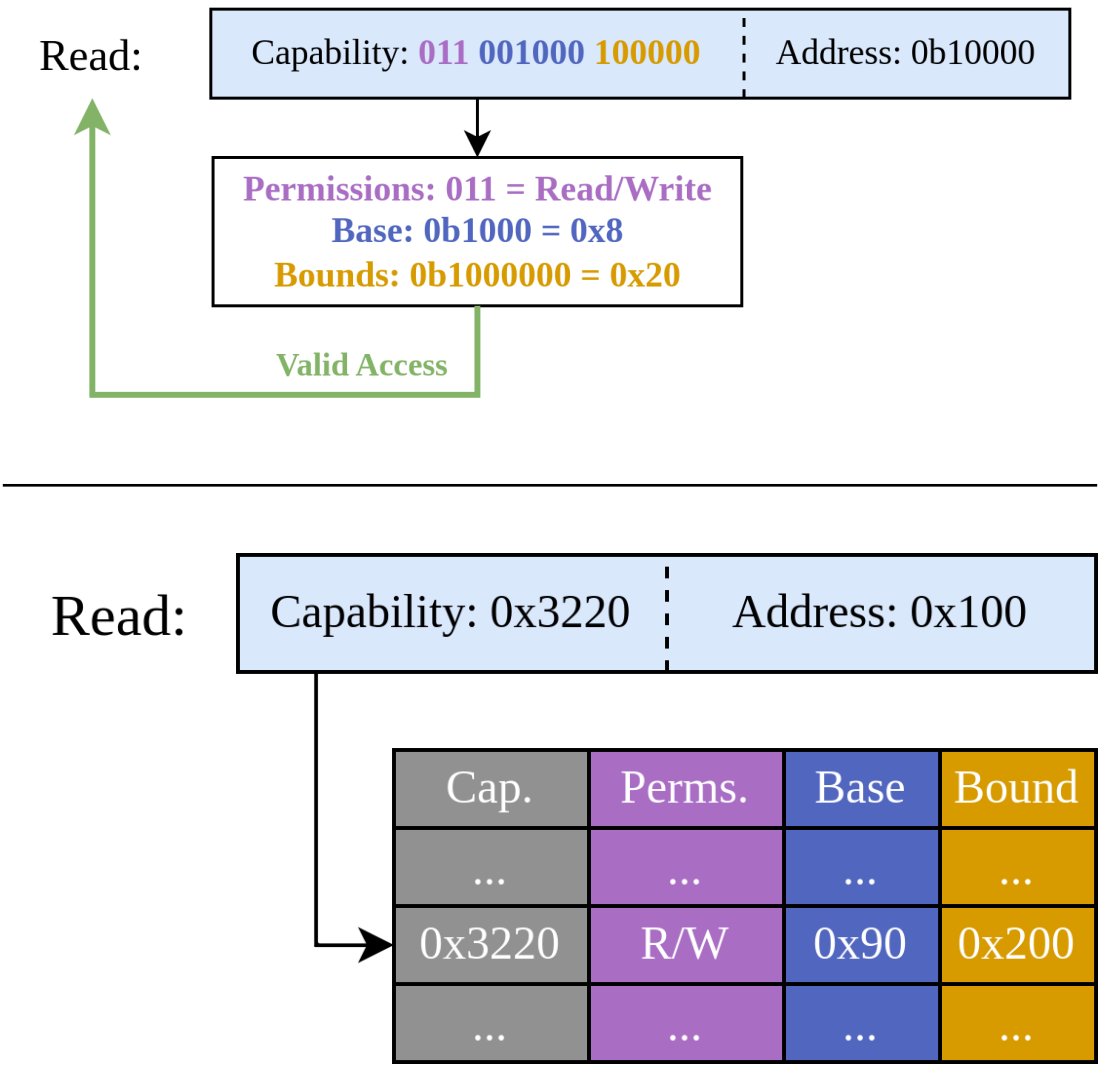

Figure 2.1: Atomic Capabilities Store Metadata Within the Pointer, While Nonatomic Capabilities Store it Elsewhere.

in the system and all their metadata. This allows for flexibility in the size, type, and amount of metadata for each Namespace, allowing for different and creative use of Namespaces to protect a wide range of objects in a system. It can also allow for resources to be more easily shared securely at scale.

Zeno has three operations related to Namespaces: NS_CREATE, NS_DERIVE, and NS_REVOKE. NS_CREATE is used to create a new Namespace – the owner of a region of memory can specify a new base, bounds, and permissions, and the

Namespace Identifier (NSID) that correctly indexes the DND will be returned. This new Namespace will be a root Namespace, i.e., it cannot have any parent Namespace and when it is created it will not have any children. NS_DERIVE is used to derive a child Namespace from one that exists currently. New metadata can be specified for this child Namespace, as long as its permissions are less than or equal to the parent Namespace. NS_REVOKE will mark the entry in the DND for the input NSID as invalid and ensure that all the input Namespace's children are also marked as invalid. This action can be performed very quickly in non-atomic capability systems, as only one region of memory needs to be invalidated, rather than a scan of all memory being required.

Chapter 3

OPERATING SYSTEM DESIGN

This chapter will describe the role of an operating system and the crucial subsystems needed for it to function. The operating system (OS) is the manager of a computing system's hardware and software resources. The OS should make it simple for applications to access such hardware by making its access abstract. That is to say, the OS should know what hardware it is running on and how to communicate with that hardware. Using that knowledge, it should provide functions for applications to access that hardware that prevent the applications from using that hardware maliciously and remove the need for each application to be designed for specific hardware.

The OS should also make it impossible for processes to access data from other processes in the system. Each process needs access to the system's hardware to function, but it cannot have access to the same hardware at the same time as other processes or risk security and functionality errors. The OS has a monopoly on control over this level of control of the hardware; it must ensure that no other applications are elevated to its privilege level.

3.1 OS Architecture

There are two main options for the overall architecture of an operating system: monolithic or microkernels. Monolithic kernels exist as a single binary that runs in a privileged mode. This binary provides all the services needed for the applications that run in unprivileged mode on the system. Microkernels, by contrast, are split into many different binaries, with as much of their code as possible unprivileged. Essential submodules, such as those that control virtual memory, scheduling, and inter-process

communication, will run in a privileged mode. Modules that do not need privileged access to the hardware, such as device drivers and file systems, will run unprivileged and use inter-process communication to send messages as needed.

Monolithic kernels are used in many systems today, famously including the Linux kernel. This design is simpler than microkernels, as different submodules exist in the same address space and can use function calls to communicate without needing to switch contexts. This simplicity not only provides significant performance benefits, but in many cases can also make it easier for the operating system to be updated by lowering the complexity of communication between submodules.

For their increase in complexity, microkernels often provide an increase in security by reducing the size of the OS's Trusted Computing Base (TCB), which refers to the components of the system that are critical to its security. Having less code running in a privileged state reduces the area of code that an attacker could use to take control of the entire system. For example, if a bug is found in an unprivileged device driver, because that server is in a separate address space from the rest of the kernel it would be much more difficult for an attacker to use that driver to gain control of the entire system than in a monolithic kernel. These benefits, along with the generally small size of microkernels, have led to their use in systems where security or size are of high importance, including Intel's Management Engine (2023) and Apple's Secure Enclave Processor (2024).

The next sections will discuss the responsibilities of various submodules within an operating system. Some of these submodules will need to run at an elevated privilege level in order to configure and control the hardware needed to perform their tasks. These submodules will be part of the 'privileged kernel.' Due to their increased level of control over the hardware, they are within the system's TCB. As such, the privileged kernel should be minimized and hardened to the greatest degree possible.

Additionally, this code should be formally verified for its security and functionality before deployment. The majority of the operating system's code will run at the lowest privilege level available; these submodules will be part of the 'unprivileged kernel.'

3.2 Boot

The boot process is crucial to a computing system's security. The operating system is an important part of the system's TCB. If it is altered or replaced during the boot process before it can exercise and enforce its monopoly on privileged hardware functions, it will not be able to enforce its security guarantees. This process begins in Read-Only Memory (ROM), typically stored in the CPU or motherboard. As this ROM cannot be updated after manufacturing, it provides a minimal amount of setup. Its main job is to verify the signature of the operating system's bootloader. If the bootloader is not signed with a valid key, it should not allow it to run. This ensures that the bootloader has not been altered. The bootloader will configure some hardware-specific details, load the operating system, and verify it in the same manner. Before jumping to the operating system, the bootloader will set up the memory to prevent the operating system from accessing or modifying the firmware. This firmware runs in the system's most privileged mode, and is generally responsible for configuring and protecting the hardware in the system. The operating system may interface with the firmware to request certain operations be performed. Most RISC-V systems use Physical Memory Protection (PMP) to ensure that no other parts of a system may access the firmware's memory. This will still allow the operating system to use system calls to request operations from the firmware but prevent it from directly executing code in a higher privilege level than the system was designed for.

It is important that the signatures and their verification be provable after boot so that applications may be sure that their security guarantees will be upheld. The

proof of verification should not be stored within the access of the operating system. Instead, many modern systems store this proof within a Trusted Platform Module (TPM), which can verify the hardware configuration of the system and securely store boot verification details.

After it has been verified, the operating system may run and set up the system to ensure that applications can run in a controlled environment.

3.3 Virtual Memory

One of the key submodules in controlling resource access in any operating system is virtual memory. When a user requests that an application be run, the OS must create a process for that application. The OS presents the system to each process as though it is the only process running. As such, each process imagines it has access to all memory from address 0 to the maximum address available. However, in reality, it only has control over the physical memory that it is currently using. Whenever a process attempts to access data at a specific address in its memory (a *virtual address*), the hardware will translate that address to a physical address and return the data at that location.

This translation is crucial for both security and functionality. It allows the OS to move applications onto and off of cores in the system, as needed for scheduling. Additionally, it is the job of the privileged kernel to configure this translation in such a manner that ensures applications are not able to access the same physical memory. This translation is set up through page tables - layers of translation that convert virtual addresses into physical addresses in a hierarchical manner. Typically, sections of the virtual address will index different levels of this hierarchical page table. It is standard for the final level of the page table to index 4 KiB of physical memory, which the final 12 bits of the virtual address index. The privileged kernel will set up

this translation when initializing a process or allocating more memory to it, and will also encode metadata into the final page table. The CPU's Memory Management Unit (MMU) will read this metadata when translating addresses to determine if this memory can be read, written, or executed. An example of this can be seen in Figure 3.1, with more specific details on the process ZenOS currently uses discussed in Section 5.1.2.

Multi-level page tables are required to translate addresses without requiring a large amount of overhead for each process. As the OS cannot know before running an application which addresses it will access, if it only has a single page table for each process it must allocate enough space for each process to access all RAM whenever a process is created. For example, a system with 64 GiB of RAM will need 16,780,000 pages to map all of its physical memory. Each page table entry is 8 bytes long, and as such the OS would need to allocate about 128 MiB per process just for translation. With a multi-level page table, the OS can simply allocate enough space for the top level of RAM and the first page of instructions to be run in a process. Then, when the process starts running and attempts to access memory that is not mapped, the hardware will stop and trigger a page fault exception. The OS may then catch that exception and map the faulting address in memory if the attempted access was valid. By having multiple levels of page table, memory can be allocated for this additional mapping at runtime instead of needing for it to be preallocated. Figure 3.1 shows an example of the process RISC-V uses for hierarchical page table translation. By default, RISC-V uses three levels of page tables, though up to five may be used in total (Waterman *et al.*, 2017).

There are some performance drawbacks from hierarchical page tables. As all the page tables are stored in memory, each translation from virtual to physical address requires that the CPU access memory once for each level of page table used. For

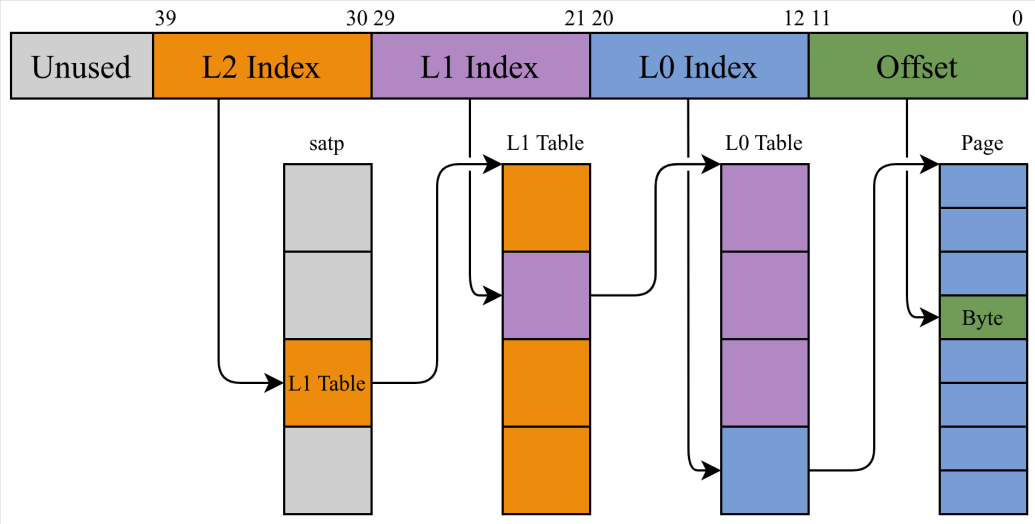

Figure 3.1: The Levels of Translation Used in RISC-V Sv39.

Sv39, this would mean that each memory access would actually require four memory accesses, quadrupling memory latency. Modern CPUs use a Translation Lookaside Buffer (TLB) to alleviate some of this latency increase. When a virtual address is translated into a physical address, the result of this translation may be stored in the TLB. When a virtual address in the same page needs to be translated, the CPU can access the TLB to find the translation. This is faster than accessing memory once, and much faster than accessing it multiple times, as would be needed for virtual memory. Modern TLBs are quite efficient, with most processes spending less than 1% of their runtime on address translation (Luo *et al.*, 2015).

Another cost of translation is the added cost of switching between processes. In order to change which process is running currently, the OS needs to flush/fence the CPU pipeline and TLB. It must then write the state of the CPU's registers to memory to ensure that the process may eventually be resumed without error. The cost of these context switches can range from 40 to 1500 microseconds, depending on the process's data access pattern (Li *et al.*, 2007). The overall effect on most programs' runtime of context switches is relatively low (David *et al.*, 2007). Despite

this, it is still beneficial for operating systems to minimize the number of and impact of these switches, as they can cause a loss of timing precision (Kaffes *et al.*, 2019) and increase tail latency, especially when combined with I/O operations (Li *et al.*, 2014).

The operating system will configure this translation by allocating pages to be used for each process's page tables. It will set the entry in the last-level page table to the physical address of the region in memory where the process's data or instructions are stored. The OS can then inform the CPU of the location of the process's top-level page table, lower its privilege level, and jump to the process so it can begin execution.

3.4 Scheduling

The operating system is responsible for managing the CPU's time. The OS will have a list of programs to be run, and will have a scheduler to determine how long each process should be run for and in which order they should be run. In order to run a program, the OS must read its instructions and data from the disk and load it into memory. Once loaded into memory, each process can be run on the system's processor. As with the memory, each process should not have any knowledge of any other processes running on the same CPU as it. Each process is run on the CPU for a certain amount of time, and then put to sleep so the other processes can run. This allows the OS to present the system as though hundreds of processes are running concurrently, despite having access to a much smaller number of hardware threads. This scheduling can be done explicitly between or within processes using *yield()* or *sleep()* calls. This scheme, called green threads, relies on the threads giving up control of the CPU. For example, this might be useful when making long asynchronous function calls. A process using green threads might request input from a user and run different computations in the background while waiting.

Most applications, however, do not explicitly yield control of the CPU. Instead,

they execute instructions until their task is finished, and rely on the OS to manage everything else. The OS will have a list of processes to be run, and will have a scheduler to determine how long each process should be run for and in which order they should be run. Before running a process, the scheduler will configure the CPU's Core Local Interrupt Controller (CLINT) with a certain number of CPU cycles. After that number of cycles has elapsed, the CLINT will trigger a hardware interrupt and redirect control to the operating system. The operating system's trap handler will save the state of the process that was running, and then jump to the scheduler. The scheduler will select another process to run, configure the CLINT, load the process's previous state into the CPU, and then redirect control to it. Processes will cyclically be run in such a manner until they inform the scheduler that they have completed execution, usually by executing an *exit()* system call. The operating system may also prevent a process from being scheduled if the user requests it or if it attempts to execute an illegal instruction.

Schedulers must balance several qualities. They should aim to maximize the amount of work completed, while minimizing the amount of time needed for each process to complete. Many also strive to maximize fairness, which can mean different things depending on the system. Some systems define fairness as allocating an equal amount of time to each process. Others may find this definition unnecessarily rigid, and instead schedule based on properties such as what kind of instructions the process is executing, I/O usage, or different levels of priority as defined by the user or the OS. Schedulers are still under active development, as different workloads benefit from prioritizing different aspects of the processes on a system.

3.5 System Calls

Virtual memory management and scheduling happen transparently to processes running on the system. That is to say, most of the time processes and programmers need not be aware of and have no control over these operations. If a process wants to access other resources on the system, however, it must request this access from the operating system. This can be done through system calls. In a system call, the process will load its registers and memory with details about the request, and then execute a specific instruction to jump to the operating system. The operating system will then analyze the process's request, fulfill it if it is valid, and then return control to the process. While some processes may want to execute privileged instructions, and the operating system may want to allow them to do so, they must still make requests to the OS to execute these instructions. This is to ensure that the OS maintains its monopoly over executing in privileged mode and that processes are not allowed to interfere with each other.

The Portable Operating System Interface (POSIX) serves as a set of standards for how processes and the operating system interface with each other. These standards have been in development since 1988 (Walli, 1995). Some operating systems are officially certified to be POSIX compliant, while many more, including Linux, are mostly compliant (Atlidakis *et al.*, 2016). This standardization helps to improve software portability across different operating systems. It also allows for the use of system calls to be simplified through user libraries. These libraries may wrap a specific system call or series of system calls into a function with a larger or more clear function. For example, *glibc's malloc()* will give the process access to more physical memory by using the *sbrk* system call for smaller requests, or *mmap* for larger ones (Kerrisk, 2024). These requests will be relayed to the operating system, which may

then map free memory pages into the process's address space. Depending on the amount of memory requested in *malloc()*, *glibc* may then break up this page into smaller chunks to improve memory utilization. This greatly simplifies memory access for processes and allows for optimization effort to be concentrated on a relatively small number of libraries instead of needing to be done for every application.

3.6 Device Access

While having access to the CPU and RAM is enough for many processes, many more will need access to other devices. The OS is responsible for providing access to these devices and implementing code needed to simplify access to these devices (*drivers*). For example, while the processes running on the system can simply use *printf()* or a write system call, the OS must know which Universal Asynchronous Receiver/Transmitter (UART) is being used by the system and must know where to send bytes to be printed on the screen.

Memory-mapped I/O (MMIO) is one of the most common tools for implementing device access. In this scheme, each hardware device is assigned a range of physical memory addresses. Each memory address is linked to one of the device's Control Status Registers (CSRs). Writing a byte to this address will cause said byte to be written to that CSR, which can be used to configure or control the hardware. For example, if the output First In, First Out (FIFO) buffer of a UART is mapped to a particular address, writing 'a' to that address will cause 'a' to be printed on the screen. This scheme is much simpler and more expandable than previous schemes such as port-mapped IO (PMIO), in which specific instructions are used to access hardware devices. These specific instructions increase the complexity of the CPU's instruction pipeline, and as such are generally only found in CISC architectures like x86.

An operating system may choose to map some of those MMIO addresses to process' address spaces. Linux, for example, provides a system call to allow a process to map a file descriptor it owns into its memory. Microkernels that use userspace servers to provide access to devices will map those devices into the appropriate server's page table.

3.7 Inter-Process Communication

In order for processes to send information to the OS or other processes, there must be some way for the processes to move data outside of their address space. When virtual memory is active, any address that a process accesses will be translated by the CPU's MMU to a physical address within that process's address space. Thus, in order to communicate with another process, a process must receive assistance from the privileged kernel.

There are many ways that operating systems today handle inter-process communication (IPC). The most common methods are message passing and shared memory. In message passing, the sending process will send its message to the operating system, and the operating system will send that message to the receiving process. As the privileged kernel can configure page table translation and enable and disable virtual memory, it is able to read data from one process's address space and write it to another process's address space. This method works well for small or infrequent messages, but if processes wish to send more data or communicate more frequently, they may benefit from an IPC method that requires less intervention from the operating system.

In shared memory, a region of the sending process's address space will be mapped into the receiving process's address space. Normally, the operating system would intentionally avoid creating such an overlap, as separate address spaces are crucial

to many of the operating system's security guarantees. However, if both processes request that a region of shared memory be set up, the operating system may write the same physical address to one or more of the pages in both process' last-level page tables. This will allow the processes to easily exchange data by simply writing to and reading from the same physical memory. As this mapping occurs in the page table translation layer, the minimum amount of memory that can be shared is one page, which is typically 4 KiB. Also, as the operating system is no longer a mediator of the communication, the communicating processes will need to implement their own security and locking measures.

3.8 File System

The method of communicating outside of address spaces that is most visible to end users is through the file system. The file system simplifies and organizes access to the system's storage. In most systems, this storage is non-volatile, allowing processes to store data through power cycles. UNIX systems also allow many hardware and operating system features, such as hyperthreading and device drivers, to be accessed through the file system. As with many operating system features, this simplifies and abstracts away complex device interactions in a manner that can allow applications to be compatible with many different types of hardware without knowing it.

The file system stores its data in blocks, which are usually 512 bytes long. These blocks are organized in inodes. These data structures contain file metadata, such as the number of blocks in the file and their location. The file system compiles these inodes in a hierarchy of directories to organize and control their access. It also stores information on which blocks are free and used on the physical media.

The file system has a dual mandate of providing access to files quickly and efficiently, while also ensuring atomicity and consistency. To avoid corruption, files must

only be marked as written to disk once they have completely been written. This is especially important in the case of a system crash. The file system must enforce that the operating system's files are not corrupted by crashes to ensure that it can recover correctly from a crash. Most file systems ensure consistency by logging files as they are being written to the disk, and only committing them to the file system once the write is complete. This ensures that incomplete operations are not incorrectly marked as complete.

Another important job of the file system is to control access to its files. The standard for modern file systems is the access control list. This is a data structure associated with each file that includes each user that has some amount of access to the file, along with how they may access it. Linux, for example, stores data about each file's owner, along with read/write/execute permissions for the owner, members of the owner's group, and all other users on the system. File systems can also monitor when files are opened to ensure that only one process is modifying a file at a time. This can be crucial for ensuring consistency and synchronizing processes.

3.9 Virtualization

Modern operating systems provide such a high level of abstraction of the hardware in the system that it is possible to engineer a completely virtual system within one. This virtualization can be useful for running code designed for different architectures or operating systems, or for isolating a process and its data from the operating system. Such virtual machines are used by cloud computing providers to abstract and manage available resources, by people developing new architectures or operating systems to breaking the host system, and within some operating systems to isolate security-critical modules from the rest of the operating system.

Virtualization can be achieved wholly in software by emulating a computer within

a process. QEMU, for example, uses software emulation to run code designed for one architecture on another. It will read every instruction to be run by the virtualized program and alter the virtual machine's state accordingly. This virtual machine will be made up of data structures in memory that contain the state of the machine's RAM, as well as CSRs and simulated hardware. The Virtual Machine Monitor (VMM) will call functions for each instruction that the virtual machine executes that update those data structures in the same manner that hardware would in a non-virtualized system. This method is great for running non-compatible code and prototyping, as it does not require any new hardware. However, it comes at a significant performance impact due to the complexity of the VMM and the amount of translation involved.

VMM designs that are not architecture-agnostic can gain performance by allowing more of the VM's instructions to run directly on the system's hardware. This is accomplished by running the VM at a lower privilege level than the VMM. This lower privilege level allows the VM to manage its own state but prevents it from altering the state of the VMM or any other processes on the host. This lower privilege level can be the same as other processes on the host. In such a protocol, if the VM attempts to execute privileged instructions control will be redirected to the VMM, which can then alter the VM's state accordingly. While this is much more performant than full software emulation, frequent traps to the VMM can have significant overhead. Many CPU architectures today support hardware virtualization, with separate privilege levels for the VM and its host. This allows the VMM to control which instructions the VM is allowed to execute, and manage its page tables to prevent it from affecting other processes on the host. While the overhead of hardware virtualization varies depending on the workload of the VM, the average performance drop is roughly 5% (Huber *et al.*, 2011). Significant work has been done to minimize this overhead due to hardware virtualization's use in cloud systems. Cloud providers leverage hardware

virtualization to manage and split up compute resources among their clients and ensure that those clients are isolated from each other.

This isolation, when combined with hardware encryption and attestation features, can be used to prevent the host operating system from accessing any of the virtual machine's data. Tools such as Intel's SGX, Arm's TrustZone, AMD's SEV, and Apple's T2 create a secure enclave within the processor. These tools encrypt the memory used by the enclave. Some of them also have separate compute hardware for the enclave and the rest of the system. This Trusted Execution Environment (TEE) prevents the operating system or other processes on the system from accessing data from the process executing within the TEE. This has security benefits by preventing outsiders from altering the process's instructions, as well as preventing confidential or proprietary data from being leaked.

Chapter 4

CAPABILITY-AWARE OPERATING SYSTEM

The job of an operating system is to control how users of the system access it. Capabilities are an optimized, finely-grained tool to control access to system resources. As such, a capability-aware operating system can utilize that tool to increase security, improve performance, and provide features that would be difficult for traditional operating systems to provide. This section will cover each of the previously described submodules of an operating system, and how they might differ in a capability-aware system.

Systems with non-atomic capabilities that have flexible metadata are the focus of this study. These capability systems are extensible and could be applied to the most broad base of use cases. The ability to encode a wide array of metadata may be crucial for some of the use cases discussed in this chapter, such as the renaming discussed in Section 4.2.

Table 4.1 provides an overview of the extent to which each submodule will differ. The sections that may be significantly restructured through the use of capabilities are the priority of this chapter. Submodules that may only have minor changes are discussed, though the details of their changes may depend on implementation details. Some of these details will be discussed in Chapter 5, while others will be left for future research.

4.1 Capability-Aware Memory Access Control

The main purpose of capabilities is to control access to memory. Page tables currently allow for operating systems to control this access at a page-level (usually

Submodule	Significantly Restructured	Minor Changes or Optimized	Future Research
Virtual Memory	X		
Inter-Process Communication	X		
Virtualization	X		
Overall Architecture		X	
Boot		X	
System Calls		X	
Devices		X	
File System			X
Scheduling			X

Table 4.1: List of Submodules Within an Operating System, and How Different a Capability-Aware Version Would Be.

4 KiB or larger) granularity. Intel CPUs currently allow the operating system to specify if a page can be read, written, or executed, and if it can be accessed by the user (Intel, 2024). Capabilities allow the operating system to control access to memory at a byte-level. Non-atomic capabilities allow for greater flexibility in its level of control as well. For example, some capability metadata for executable capabilities may specify entry points into the capabilities. This could prevent return-oriented-programming (ROP) attacks (Roemer *et al.*, 2012), as attackers would not be able to execute arbitrary gadgets by jumping to the middle of these capabilities. Different systems could include different kinds of metadata to meet each system's individual security targets, and could be updated and expanded as these targets change.

One piece of metadata that could exist within the capability system is the root page table associated with that capability. This could allow one process to give direct access to a region of its memory to another process without requiring the operating system to map regions within both of their address spaces to the same physical memory. With this and the entry point metadata discussed above, one process could

directly run code in a different address space without being able to alter that address space in an undesired way. It could also allow for a process to directly access a much larger region of memory than is currently feasible. For example, a process could have its heap capability be a root capability with its own root page table, thus allowing its heap to be as large as an entire address space on the system. This idea could be expanded further to datacenters with many systems running within it. A process could transparently access memory on another system by simply using a capability that originated from that system. Such a protocol could make data access across datacenters both more efficient, secure, and easy to implement, when most protocols today have to optimize for just one of those benefits.

A number of research questions need to be answered before such a protocol could be implemented. For example, how can we ensure that the capabilities in such a datacenter are unique? If one system generates a capability that is the same as one on another system, it could make it ambiguous as to which region of memory a process is referring to. This could lead to unintended behavior and security vulnerabilities. A centralized system could store a list of the capabilities currently in use and allocate capabilities to different systems in order to prevent overlap. However, communication between the systems and this central capability controller could quickly become a bottleneck. Additionally, in a large datacenter, this would significantly limit the number of capabilities each system could create. Attaching a unique number to each system and its capabilities would also have the same drawback, and could hinder the security provided by capabilities by reducing the search space for a capability. One idea is to have each system name its own capabilities for those that will stay on the system, and rename the capabilities when they are to be shared with other systems. This could solve the limitations caused by the other ideas, but how capabilities will be renamed without requiring a central capability controller (and thus causing the

same problems) is not yet known.

Capabilities also make it possible for applications to exercise a much greater level of control over their memory than they currently can. Mentioned in Section 2.2 are Zeno's custom capability operations NS_CREATE, NS_DERIVE, and NS_REVOKE. These operations may be executed in user-mode, allowing applications to set their own rules for how their virtual memory is accessed. This can be useful when sharing memory, as discussed in Section 4.3. While applications may use these instructions directly, many will not be aware of these specific instructions. This is especially true for legacy, capability-unaware programs. It is the job of a capability-aware OS and the associated libraries to make using capabilities easy and ensure that legacy programs are secured using these instructions where possible. Future research and implementation will need to determine exactly how this should be done. For example, it is simple to say that a capability-aware *malloc()* should use an instruction like NS_DERIVE to derive a capability from the heap that will limit that pointer to the region specified in the *malloc()* call. However, many programs utilize functions in undefined or unexpected ways (Wang *et al.*, 2012), so implementing such a library may have unexpected complications depending on the program using it.

Capability memory access control may have some performance drawbacks, however. Nonatomic capabilities suffer from requiring another memory access to access the capability's metadata. This additional lookup time could be minimized through the development of a similar Capability Lookaside Buffer to cache this metadata. This lookup could also happen in parallel to the TLB lookup, thus minimizing or eliminating any additional latency caused by the additional lookup. This would add another location for the system to invalidate in the case of a revoke operation, but it is likely that the minimization of latency in the general case would result in an increase in overall performance. Future research can measure the impacts of such a

design.

4.2 Capability Virtualization and Enclaves

The main security use of virtualization today is to create enclaves within a system. These enclaves can allow for applications to run within an operating system without the operating system being able to access any of their data. This is especially useful in cloud applications, where the users may not want to trust the provider with sensitive application details. The main downside of these enclaves is that they require special development from programmers in order to separate their application into trusted and untrusted parts (Zheng *et al.*, 2021). A capability-aware operating system could leverage hardware-enforced byte-level memory access to provide these enclaves for every process running on the system. This could be done without refactoring the processes by changing how the operating system loads applications into memory. In such a scheme, the firmware would own the root capability to the application's address space. It would then derive a write-only capability, which would be given to the microkernel to load the application into RAM. Once this loading is complete, the firmware would revoke the loading capability and derives others from its root capability for the process's code and data sections. These capabilities would then be given to the application for use during its execution. Figure 4.1 shows the different sections of the system along with which capabilities they own. As can be shown in the figure, the operating system does not have access to the majority of the process's address space. It only has access to the region it needs to provide trap handling functionality. This would remove significant portions of the OS from the system's TCB. Today, operating systems not utilizing enclave tools have full access to all the physical memory within a system. Removing this access could make it significantly more difficult for attackers to gain control of a process.

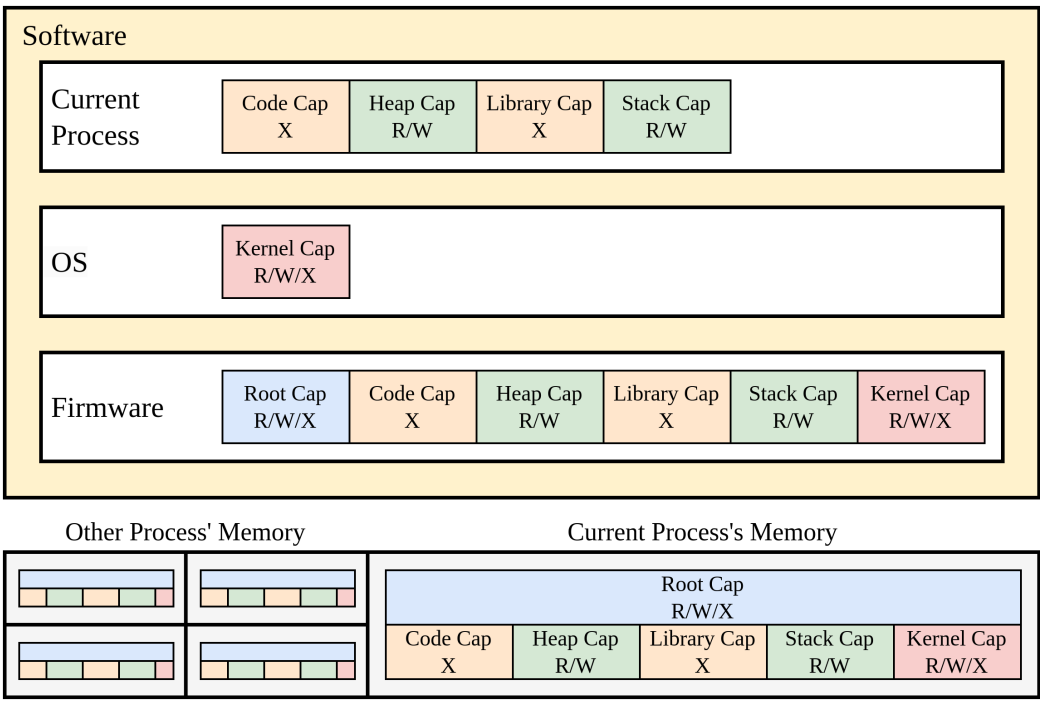

Figure 4.1: The Final State of Memory After Loading a Capability-Powered Enclave.

A number of research questions need to be answered before this plan could be implemented in a system. While the proposed protocol does prevent the unprivileged kernel from having access to the process's address space while it is running, it does not prevent it from reading the program's data as it loads that program into RAM. This data could be encrypted on the disk and decrypted by the application once the application is running and the unprivileged kernel no longer has access.

Additionally, the instructions that the process executes need to include the capabilities for the data or instructions that they access. However, it is the goal of this system that the unprivileged kernel does not have access to those capabilities. Future research will need to determine a method for the unprivileged kernel to load a program into memory without its capabilities while still allowing that program to use its capabilities. It is possible that the privileged kernel could go through the process's address space and replace the capabilities that the unprivileged kernel wrote with the

process's capabilities. However, this operation would likely be complex and require significantly increasing the TCB of the most critical part of the system. Research is ongoing into a simpler method to implement this capability renaming, either in trusted firmware or in hardware, in a manner that does not compromise the guarantees provided by capabilities.

Another question to be answered is how often such an operation would need to take place. Does this operation need to happen at every context switch? How much would that affect the performance of the system? ZenOS, the capability-aware operating system described in Section 5.1, will serve as a foundation for future research into questions such as these.

4.3 Capability-Aware Inter-Process Communication

In the inter-process communication protocols that exist today, the operating system has full access to the message(s) being sent between the processes. This is partially due to its control over physical memory, but the protocol discussed in Section 4.2 could remove this control. Capabilities can also be used to limit or prevent the OS's access to inter-process messages.

The first hypothesized protocol for capability-aware IPC involved passing capabilities between processes using the unprivileged kernel. The sending process would derive a capability with its desired bounds and permissions, send that capability to the unprivileged kernel, which would then send that capability to the receiving process. This capability and its associated shared region of memory could be used by the processes to establish a secret key using a protocol like Diffie-Hellman (Maurer and Wolf, 2000). This would allow the processes to encrypt their messages to prevent the unprivileged kernel from using a copy of the capability to see or alter the messages. A prototype of this protocol has been implemented in ZenOS, and is discussed in

Section **??**. This prototype is simple for the operating system to implement, though will require additional complexity within the user libraries that control inter-process communication. Additionally, measurements of the performance impact of encrypting these messages have not been completed.

Another potential method to allow for inter-process communication without giving access to the operating system is designed around remote procedure calls. This process would begin with the receiving process declaring that it would like to receive messages. It does so by telling the unprivileged kernel the addresses of *put()* and *get()* functions, which will be used to send or receive messages from the process, respectively. It will also give the unprivileged kernel a capability to allow for these functions to be executed. Some kind of key may be included as well if this process would like to limit from which processes it may receive messages. The sending process would then ask to send a message to the receiver, and if the operating system deems its request valid, it will give the receiver's addresses and capabilities to the sender. The sending process can then execute the receiver's *put()* call with a derived capability containing its message as the argument to that function. This protocol only gives the operating system the ability to also send messages to the receiving process, as the OS never receives the capabilities containing the messages. Along with the process loading protocol discussed in Section 4.7, this would completely isolate the operating system from the processes it is responsible for managing. This system could also be used for system calls to improve communication between user processes and the operating system.

4.4 Capability-Aware Architecture

The tools discussed above can be used to optimize and harden communication between functions within a process and between separate address spaces. As such, a

capability-aware operating system should be designed with a microkernel architecture in mind, as its tradeoffs make more sense given the security-focused nature of capability designs. Additionally, hardware capabilities can be used to ease some of the drawbacks of such a microkernel design. Separate submodules within a capability-aware microkernel can communicate efficiently and with clear standards for that communication in ways that would be more difficult for operating systems running on traditional systems to manage.

In search of improved performance, many microkernels have moved from synchronous inter-process communication to asynchronous. Many of these IPC methods still require some amount of involvement of the kernel's page table manager or copying of data (Elphinstone and Heiser, 2013). Hardware capabilities would obviate the need for the kernel to change its page table mapping in order to set up shared buffers. Additionally, as the physical memory location is not changed when a capability is shared, no data need be copied.

seL4, a formally verified microkernel based on *L4*, uses software capabilities to control communication between its different submodules (Heiser, 2020). User code in *seL4* can utilize kernel Application Programming Interfaces (APIs) to change the state of its resources; in a hardware capability system, this could be accomplished without involving the kernel. This would allow the kernel to be minimized further and reduce the number of context switches executed by a process. Capability-aware inter-process communication is discussed in more detail in Section 4.3.

4.5 Capability-Aware Boot

As discussed in Section 3.2, modern systems store boot attestation signatures in separate hardware - the TPM - in order to ensure that a compromised operating system cannot alter them. This additional hardware would not be necessary in a

capability-aware system, as the hardware or firmware could create a capability to store this information in RAM. As long as this is never shared with other parts of the system, the capability hardware can ensure its attestation without requiring other hardware. This can also be applied to the PMP that is currently used to protect the firmware's memory during the boot process. PMP entries can support regions as small as four bytes, and RISC-V supports up to 64 entries. Capabilities can support any number of regions of any size, making them a more flexible and scalable solution.

4.6 Capability-Aware Device Access

Capabilities can be used to make the memory mapping used for MMIO device access more specific and secure. Processes and drivers can be given exactly and only the access to a device that they need to operate correctly. A process could even be given access to a single CSR in such a way that allows it to, for example, check the device's status without being able to change it. Having more access control options than read/write/execute will provide for a more sophisticated and secure method of accessing devices. This could be used to harden device access and move some device servers from privileged execution modes to user mode.

4.7 Capability-Aware Scheduling

A capability-aware operating system could minimize context switches by moving state information to capabilities that are used to fetch instructions. In theory, such an OS would not need to flush the CPU pipeline and TLB to change contexts, as the capability metadata within the pointers used in the instructions stored in the pipeline would direct the CPU's memory management unit (MMU) to the correct address space. This could also allow much of the operating system's scheduling and virtual memory management code to be run at a lower privilege level, thus reducing

the OS's TCB. Further research is needed to study the viability, effectiveness, and security of such a design.

4.8 Capability-Aware File System

Current capability solutions typically focus on memory access. One of the ways that processes access the disk is through mapping files into memory. It is simple to see how capabilities could apply to this use case - by tightly controlling access to virtual memory, access to memory-mapped files could also be tightly controlled. Future research may study capabilities that control access to the disk, which could allow for a fully capability-aware file system.

Previous studies have worked to secure log files in particular, due to their importance in digital forensic investigations (Böck *et al.*, 2010) (Lantz *et al.*, 2006). It is helpful for many parts of a system to be able to write to logs. However, if attackers can overwrite the logs or delete them, they can make it difficult to discover their means of entry or what parts of the system they affected. Capability-secured logs could allow for many parts of the system to append to the logs without allowing overwrites or deletion.

This access control could be expanded to other parts of the file system. File systems currently use access control lists to control access to files. Capabilities could optimize and increase the flexibility of this solution. Future research will likely find novel ways to apply capabilities in file systems as a capability-aware file system is developed.

4.9 Remaining Research Questions

In order to implement the protocols discussed above, there are still a number of research questions that need to be answered. In a non-atomic capability system, it

is possible for the capability metadata to include the location of a process's top-level page table and its register state information. In theory, this is all the information that is needed to switch contexts. The question is this: which part of the system should be responsible for saving the previous process's context and switching to the new process's context? If it is the operating system, it will be difficult to implement such a scheme without divulging the data that this protocol was designed to keep secret. The hardware could potentially switch contexts, though the performance, power, and area penalties of such a hardware module have not been measured and could be prohibitive. Additionally, hardware cannot be updated once it has been deployed, so the design's functionality and security would need to be formally verified before it could be deployed. This job could also fall to the firmware, but would increase the firmware's TCB, which could be dangerous for such as security-crucial portion of a system.

There are additional questions to be answered regarding capability metadata. If processes are allowed to jump to other process' address spaces, this could potentially lead to confused deputy or return-oriented-programming attacks, in which the attacking process uses a capability it has been given to jump to code that it is allowed to execute but is not supposed to start at. By stringing together parts of another program in this manner, an attacker can induce arbitrary behavior in another process without injecting code (Roemer *et al.*, 2012). The context switcher could potentially verify that the process is being entered at a valid entry point. If the context switcher is in hardware, the inherent lack of flexibility could be unnecessarily limiting for programs that want to allow for multiple entry points or entry points under certain conditions. The firmware could allow for increased flexibility at the cost of an increased TCB. It might make more sense for this information to be stored in the capability metadata itself. The capability metadata definitely needs to include the

capability's base, bounds, and permissions. In order to switch contexts using only a capability and virtual address, the metadata would need to include the owner's top-level page table and register state information. If that data is being included, it makes sense to also store valid entry locations. However, when being resumed after being paused by the scheduler, a process may be at any location within its executable address space. With that in mind, does it make more sense to allow for capabilities that say any location is valid to start at, or should capabilities that are going to be used for remote procedure calls be a separate kind of capability?

Similarly, does every capability need to include values in every metadata field? It might not make sense, for example, for a process's stack capability to include an entry location. How and where this metadata is best stored also remains to be answered. An intuitive answer is to store it in RAM, as is standard practice for traditional memory metadata. If it is stored in RAM, how is this RAM protected from the rest of the system? A system like RISC-V's Physical Memory Protection could be used, or the CPU's memory management could own the only valid capability to that region. One idea hypothesized was to include a region of volatile memory built into the CPU's MMU for storing capability metadata. While this could be made very secure, it would likely come at too high of a cost of area and/or expandability of the system.

It is core to the capability model that only the data owner may write the capability's metadata, but is the same true for reading? In the remote procedure call example discussed above, the protocol could mandate that the only valid entry location is the base address of the capability and allow the sending process to read that value so that it can jump to that location. While allowing for this metadata to be read could allow for increased flexibility and potentially new features, further research is needed to determine if this access could break the security guarantees provided by

the capability model.

All of these questions are core to implementing a capability-aware system. However, once implemented, significantly more questions arise regarding the viability of such a system. The security benefits of capabilities will need to be verified by conducting attacks against the system and its capability model to determine if/where it fails to meet its guarantees. These attacks should be run against applications that are aware of capabilities and utilize them explicitly, along with applications that are not but are running with capability-aware libraries in a capability-aware operating system. One would assume that the former would resist more of the attacks; if/where the latter fails will reveal much about the capability model and where the capability-aware systems can be improved. Capability systems will also need to be tested for side-channels, as such methods of attack could sidestep the protections afforded by capabilities.

The performance impacts of these capabilities will need to be measured and minimized as well. While security is important, many consumers and companies are not willing to give up significant amounts of performance to prevent attacks, depending on their threat model and the importance of their data. Future research should measure the performance difference between a capability-aware system, systems that are optimized for performance, and capability-unaware systems that are optimized for security. This difference should be measured between both capability-aware applications and legacy applications.

ZenOS, discussed next, is a capability-aware operating system designed for Zeno, a RISC-V based non-atomic capability system. Today, it can boot and run a number of different applications. As its development continues, it will be a foundation to study the capability model and answer these and other questions related to it.

Chapter 5

ZENOS

ZenOS is a capability-aware operating system built from scratch to take advantage of hardware-enforced non-atomic capabilities. Its current implementation is designed for Zeno (discussed in Section 2.2), though it could be adapted to other capability systems. A version of ZenOS also exists for QEMU. An overall view of the architecture of ZenOS can be seen in Figure 5.1. Today, this operating system can boot, print to the screen, take input from users, and run a variety of applications, as discussed in **??**.

5.1 ZenOS Architecture

5.1.1 Machine-Level Code

One of the goals of ZenOS is to have a minimal TCB. Code that runs in machine mode in a computing system is part of its TCB because of its reach and power - machine mode in RISC-V has full access to the system's physical memory and hardware. As such, ZenOS only runs in machine mode when one of the lower privilege levels could not perform the same task.

5.1.1.1 Entry

The first code that runs upon boot must run in machine mode; this code is referred to as Entry in ZenOS. It consists of a small amount of handwritten assembly designed to do simple tasks that are needed to enable ZenOS to run in supervisor mode. It is responsible for initializing timer interrupts, the stack, RISC-V's Physical Memory Protection, and jumping to supervisor-level code by executing *mret*. RISC-V's *xret*

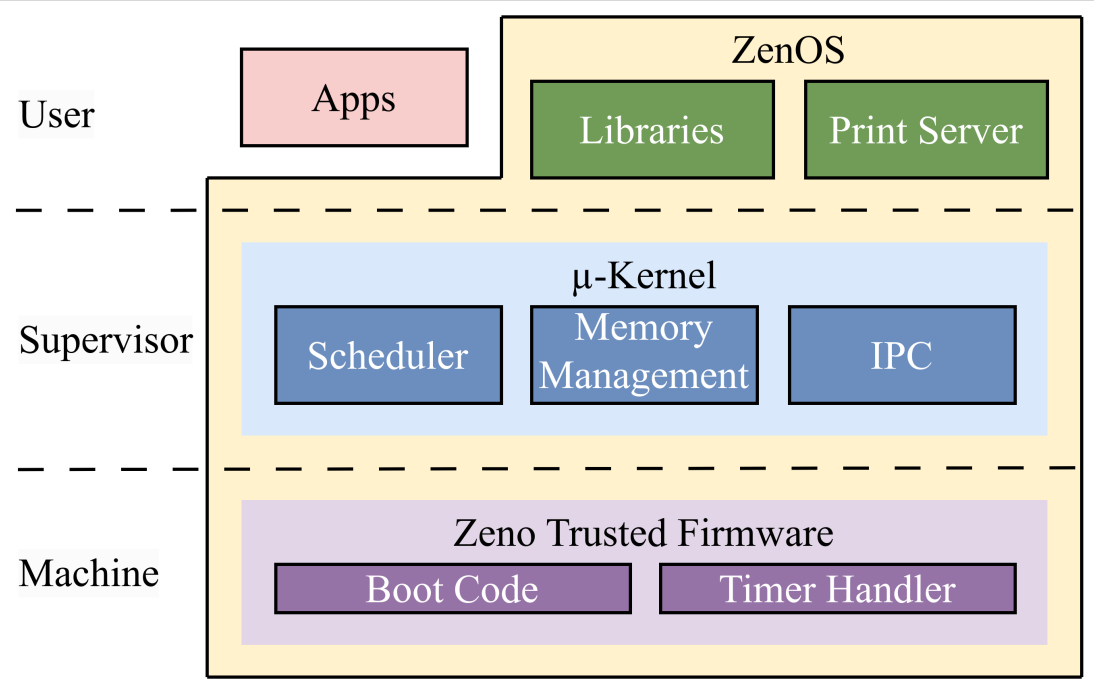

Figure 5.1: Overall View of the Architecture of ZenOS.

instructions are used to move to a lower privilege level, while ecall is used to move to a higher permission level. (x here refers to the privilege level when the instruction is executed - *mret* returns from machine mode, *sret* from supervisor mode.) *ecall* will jump to the higher level's trap handler, while *xret* will jump to the location stored in the *xepc* register.

5.1.1.2 Timer Interrupts

ZenOS's CLINT handler also runs in machine mode. The CLINT works by comparing the current number of cycles, MTIME, to its stored number of cycles, MTIMECMP. If MTIMECMP is greater than MTIME and the CPU is in supervisor or user mode, it will trigger an interrupt and jump to whichever trap handler has been configured to handle it. This means that these interrupts must be handled in machine mode, as if the CLINT jumps to supervisor mode it will immediately trigger another interrupt

and the trap handler will not any time to change the value stored in MTIMECMP. As such, when a timer interrupt is triggered, the CLINT jumps to the value stored in mtvec in machine mode. The code at this location sets the next value of MTIMECMP based on the initialization done in Entry. It then executes mret to jump to the supervisor-level trap handler, which can schedule the next process to run. Other traps do not have this limitation, so Entry delegates all other traps to supervisor mode.

5.1.2 Supervisor-Level Code

Much more of ZenOS runs in supervisor mode. This mode generally has enough permissions for ZenOS to perform the tasks it needs to do to manage the system, while still being less potent than machine mode. It is still part of the TCB for now, though future research may minimize or remove supervisor-level from ZenOS entirely by moving it to user mode.

5.1.2.1 Virtual Memory Management

Supervisor code is responsible for managing virtual memory, ensuring that applications have access to as much memory as needed and that their physical memory does not overlap (except when requested). Upon boot, the virtual memory management will map the kernel's binary, trampoline page, stack, user applications, and the UART to the kernel's page table. Each user-level process is also assigned a page table that is mapped with its binary, stack, trampoline page, and trapframe. Binary here refers to the application's code and its data, the stack is a region of memory used for storing local, short-term data, and the trampoline and trapframe are regions of memory used when handling interrupts and exceptions (traps). These locations are mapped using ZenOS's *walk()* function, which emulates in software the page table walk executed

in hardware, discussed in Section 3.3. This allows ZenOS to determine the last-level page table entry (PTE) that a virtual address will access before being translated into a physical address. Once this has been determined, *walk()* can be used to either return the physical address that corresponds to the input virtual address or set a specific physical address that should be used for a virtual address. This is crucial in some applications, such as when mapping the UART into virtual memory – the virtual address used by the UART server must be translated to the MMIO address that corresponds to the UART.

The system used by ZenOS is *Sv39*. This system utilizes the bottom 39 bits of the virtual address to find a physical address. These 39 bits are split into three 9-bit indexes and a 12-bit offset. The three indexes are used to find a specific page table entry within a page table. The highest-level page table in the system is stored in the supervisor address translation and protection (satp) register. This page table contains 512-page table entries, and the top index in the virtual address selects one of these entries. This entry is another page table, which contains another 512-page table entries. Metadata used by the hardware to determine if the page table entry has been set up by the OS already and determine which kinds of accesses to that region of memory are allowed is also stored within the page tables. The second 9-bit index will select a page table entry from this list, and the third 9-bit index will select a page table entry from that page table entry's list. The lowest-level of the page table will return a page of memory instead of a page table entry. These pages are 4 KiB long, and as such 12 bits are needed to select a specific byte within them. This is the purpose of the offset within the virtual address (Waterman *et al.*, 2017). Each process's page table is configured by the OS, and set up such that even if one process's virtual address is the same as another's the physical addresses will translate to different locations in memory. When a process is running, it is not involved in this

translation. The hardware will simply walk through these page tables and return the byte that each virtual address is translated to.

5.1.2.2 Trap Handling

The other responsibility of ZenOS's supervisor code is to handle traps not handled by machine mode. RISC-V uses the term traps to include both events that are triggered by hardware (interrupts) and by software (exceptions). Standard RISC-V allows traps to be delegated to either machine mode or supervisor mode (Waterman *et al.*, 2017), so ZenOS delegates all traps other than timer interrupts to supervisor mode. Future development may minimize the supervisor trap handler to simply store a little of the trapping process's state and redirect traps to user mode. The trapping process's state is stored in its *trapframe*, a region of memory controlled by the kernel and used to prevent the kernel from overwriting data that was in use by the process. The trapframe stores every CPU register, the app's entry location, and the location of the trap handler that should be used for the app. The kernel's supervisor address translation and protection (satp) register is also stored into each trapframe, as the trap handling code will need to switch to the kernel's page tables before it is able to access kernel code.

When a supervisor trap is triggered, the CPU jumps to *save_registers*, which stores the CPU's current state into the process's trap frame and switches to the kernel's page table. This function must be written in assembly to ensure that none of the CPU's state is lost by accidentally overwriting registers that are in use by the trapping process. While *ecall* instructions are called explicitly by a program, the trap handler will also be used to handle interrupts and unintentional exceptions. Therefore, it cannot rely on the typical application binary interface (ABI) for caller vs. callee saved registers. Instead, it must ensure that every register is 'callee' saved — i.e.,

saved by the trap handler. When *load_registers* is called to return to a process, it will restore all these registers before jumping to the process. The process may then resume execution as if it was never interrupted.

After switching page tables, *save_registers* will jump to *trap_handler()*. This function will read the supervisor trap cause (*scause*) register to determine what kind of trap has been triggered, and direct execution to whichever trap handler function is required for the triggered trap. In the case of an *ecall*, *scause* is set to 8. Therefore, the trap handler will jump to the system call handler (discussed in Section 5.1.2.3). Other *scause* values indicate that the trap was not triggered intentionally by the process. If the top bit of *scause* is 1, that indicates that the trap was an interrupt and not an exception. For example, a *scause* of 0x80000007 indicates that a timer interrupt occurred. If ZenOS's kernel was executing at the time of the interrupt, the kernel will resume its previous activity. However, if a user application was being run at the time, ZenOS will instead request a print buffer flush and restart the cycle (see **??**) at the shell.

Another commonly triggered type of unintentional trap relates to ZenOS's use of virtual memory. When a process is initialized, only the first page of its virtual memory is mapped. Then, if a process attempts to access memory outside that page, an exception is thrown as the hardware cannot translate that virtual address into a physical address. When the OS receives this exception, it must determine the location within memory that the process attempted to access (in RISC-V, this is stored in the *stval* register) and configure the translation for that page of memory. This configuration includes loading the associated region of the application's binary into the correct region of memory, setting the application's page table entry to the correct physical memory location, and setting the appropriate read/write/execute bits. After this mapping has been set up, the trap handler can return to the faulting instruction

and allow the application to continue. This on-demand paging allows ZenOS to efficiently use physical memory and the automatic exception handling functions in such a way that processes are not and need not be aware of this mapping occurring.

5.1.2.3 ZenOS System Calls

ZenOS identifies system calls by a unique number stored in the a7 register. The system call handler will read this register to determine which system call has been executed. These system calls are used by applications to request functionality they do not have access to in user mode. Seventeen system calls are currently supported by ZenOS — twelve that are used by *glibc* and five that are unique to ZenOS. A further three system calls are partially implemented as part of efforts to improve compatibility with *glibc*. Work is ongoing to improve compatibility and implement more system calls. One of ZenOS's goals is to become POSIX-compatible, which would allow for a wide range of capability-aware and unaware applications to run on and benefit from capability systems. A list of the syscalls that are supported today can be seen in Table 5.1.

The custom system calls that exist in ZenOS today center around its UART server. This server exists as a proof-of-concept for capability-aware inter-process communication. It, and the system calls it utilizes, are discussed in Section 5.1.3.1.

5.1.3 User-Level Code

All the sections of ZenOS that do not need access to supervisor permissions are run in user mode. This reduces the impact of bugs in these sections of code, both in terms of system reliability and security.

Syscall Name	#	Result	Notes
Send UART Functions	457	Used by the UART server. Sends the kernel the location of its syscall functions.	Syscalls with numbers greater than 456 are custom to ZenOS.
Request Print Buffer	458	Requests a capability from the UART server for printing data.	
Buffer Request Complete	459	Used by the UART server. Returns a printing capability.	
Request Print Buffer Flush	460	Requests that the UART server flush saved printing data.	
Buffer Flush Complete	461	Used by the UART server. Returns control flow after flush.	
Exit	93	Terminates calling process, resets its state, and flushes its print buffer(s).	
Exit Group	94	Terminates all threads in the process. In ZenOS, this is the same as exit().	Multithreading is not currently supported.
Execve	221	Executes the application whose ID is stored in a0 at the time of the call.	Invalid ID load the shell. The calling process's state is not lost and may be resumed.
Sbrk	2	Increments the process's data space by a0 number of bytes. Returns the previous program break location.	
Brk	214	Sets the end of the process's data space to a0. Returns 0 on success, -1 on error.	
Uname	178	Returns system information.	"ZenOSv1.0 rv64ima"
Get TID	177	Returns thread ID.	ZenOS does not have separate identifiers for thread/group/user/process. Currently, syscalls 160-177 return the process ID.
Get EGID	176	Returns effective group ID.	
Get GID	175	Returns group ID.	
Get EUID	174	Returns effective user ID.	
Get UID	172	Returns user ID.	
Get PID	160	Returns process ID.	
Openat	56	Open or create file.	ZenOS does not currently support files. Syscalls 56, 57, and 63 are in development and only supported for specific files.
Close	57	Close a file descriptor.	
Read	63	Read from a file descriptor.	

Table 5.1: List of Supported Syscalls.

5.1.3.1 UART Server

ZenOS's UART server was designed to be a model of capability-aware inter-process communication. In ZenOS, each process does not have access to the UART. The only process that may access it is the UART server. When it is first run, it creates several write-only capabilities for regions of its data memory. In order to print to the screen, a process must first request one of these capabilities by executing the Request Print Buffer syscall. Upon receiving this request, ZenOS will request one of the capabilities from the UART server. The UART will mark one of its capabilities as allocated and give it to ZenOS using Buffer Request Complete. ZenOS will then pass this capability to the requesting process. The process may then write its printing data to

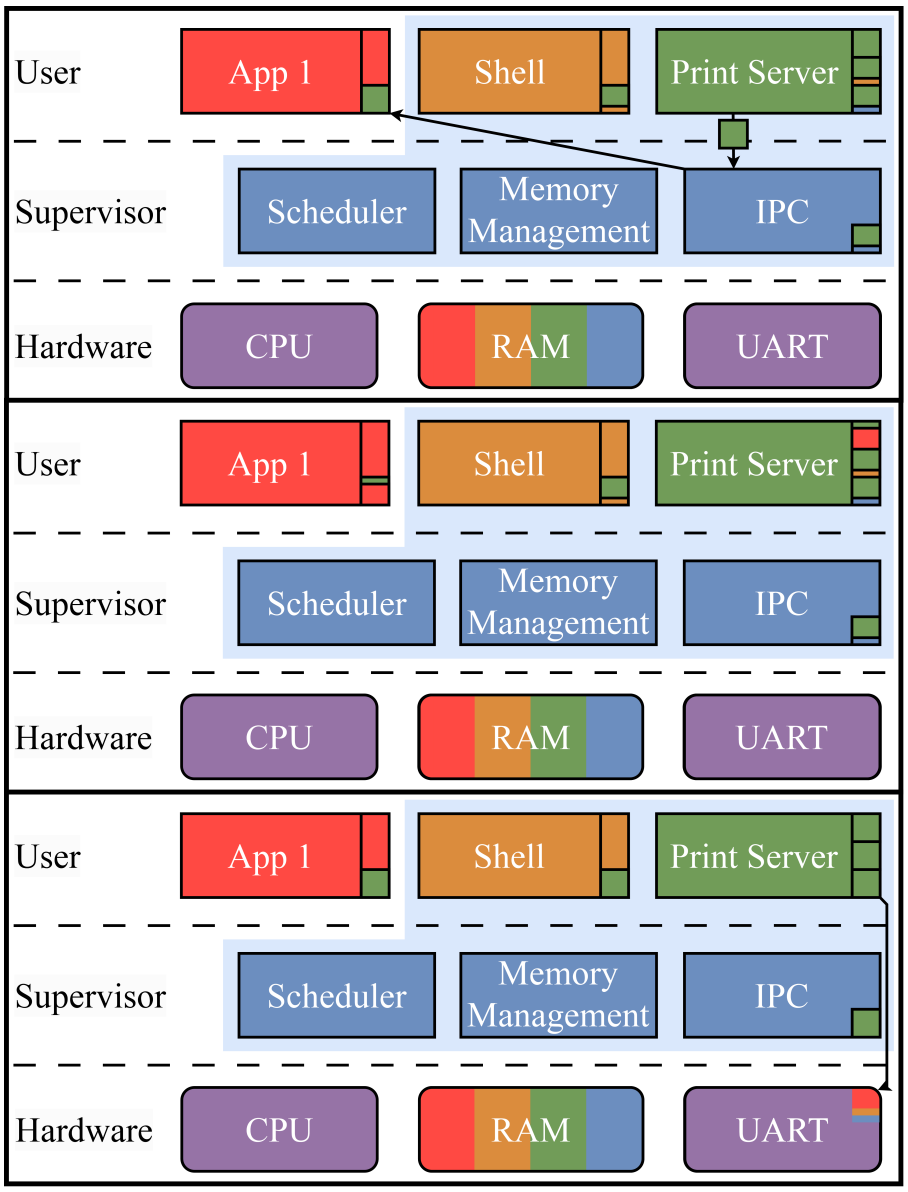

Figure 5.2: ZenOS and its Applications Print Through Capabilities Shared by the UART Server.

this capability. The UART server will periodically flush the data in these capabilities to the screen, or a flush may be specifically requested with the Request Print Buffer Flush syscall. After such a flush has been completed, the UART will execute a Buffer Flush Complete call and return control to the operating system. This process can be seen in Figure 5.2.

A similar process is being developed for taking input. Currently, processes can request individual characters from the UART. Once a read-line function has been fully written, the UART server will be able to share read-only capabilities with processes to share input in a similar manner.

5.1.3.2 Shell

Another section of ZenOS that runs in user mode is the shell. The shell will ask the user which application they would like to run and wait for user input. This user input can then be compared to the names of valid applications loaded in ZenOS. If a match is found, the shell will load that application's ID into register a0, load a7 with *execve*, and execute an *ecall*. ZenOS will then load this application into memory and begin its execution.

5.1.3.3 User Libraries

ZenOS includes some code usable by applications to simplify access to its resources. These user libraries simplify the process of reading input, making system calls, and printing to the screen. For example, if an application wants to print to the screen, it will call *print_string()*. This function will then check if the application has access to a valid capability from the UART server, and request one if it does not. Once a valid capability has been acquired, the library will write the string to that capability, and handle outlier cases like if the string to be printed is larger than the capability. All of this happens seamlessly without the application needing to be capability-aware.

5.1.4 Applications in ZenOS

Due to the limited number of system calls currently implemented by ZenOS, many programs will fail to run on it. Programs need not be capability-aware to be com-

patible with ZenOS, but complex programs that require functionality implemented by system calls not yet implemented on ZenOS will fail to run. As such, the applications that are currently running on ZenOS were created in house in order to test its functionality. All the implemented system calls returned the expected results, as shown in Table 5.1.

More work is needed to improve ZenOS's compatibility and functionality. There are many system calls that libraries like *glibc* rely on, and implementations of some of these calls are being worked on currently. Additionally, while ZenOS supports writing to storage media, it does not include a file system currently. A capability-aware file system could provide the same benefits to file security that capabilities provide to memory security, but such a system is outside the scope of this paper.

5.2 Walkthrough of ZenOS

Upon reset, ZenOS will begin at Entry. It first enables the CLINT and configures its interrupt period. Entry also sets up a region in memory so that the machine mode interrupt handler can add to *MTIMECMP* when an interrupt is triggered without losing any register data that was in the CPU when the interrupt was triggered. Once this has been completed, Entry can write the address of the machine mode trap handler to the machine trap vector (mtvec) register.

Entry then configures *sp* to point to a region of memory that C code can use for stack data. This region of memory is a 4096-byte long character array that can be used by functions to store transient data. The kernel has such an array within its memory, and each user program is also allocated one within its address space. Another responsibility of Entry is disabling RISC-V's Physical Memory Protection (PMP) by writing to the CPU's *pmpaddr0* and *pmpcfg0* CSRs. By writing to these registers, Entry can instruct the CPU that any process can access any memory with any type

of instruction. While these protections may be helpful in some systems, they are not needed in a capability-aware system and lack the granularity and easy configurability provided by capabilities. Configuration of these protections is still needed, however, as the default state is to deny any kind of access to memory unless in machine mode. Finally, Entry loads the address of Start into the machine exception program counter (*mepc*) CSR, stores 1 into the machine previous privilege mode (*mpp*) CSR, and executes *mret* to jump to supervisor mode.

The supervisor initialization must initialize all the connected devices and set up virtual memory before user-level applications can be run on it. The main device of concern in the current version of ZenOS is the UART. Like the CLINT, it too contains several CSRs used to define its behavior. Upon boot, ZenOS configures its baud rate, sets the word length to 8 bits (as each character sent will be 1 byte long), flushes and then enables its input and output queues, and enables it to transmit and receive interrupts. After this, ZenOS may print to the screen by checking the UART's line status register and sending characters as long as it is ready to receive them. This allows ZenOS to print to the screen and inform the user that it has begun the initialization process.

ZenOS begins initializing virtual memory by creating a list of pages in RAM that are not currently being used. This list is a singly linked list that begins with the first page after the kernel's binary in memory. ZenOS's linker descriptor provides the kernel with the address of the end of its binary. Each object in the list contains a pointer to the next object in the list, which is one page higher in RAM. The final object in the list is the last page before the Top of Range (TOR) – the last virtual address that can be translated to a physical location in memory. Figure 5.3 shows the state of memory after this list has been initialized

After this list has been created, pages from it may be allocated to processes as

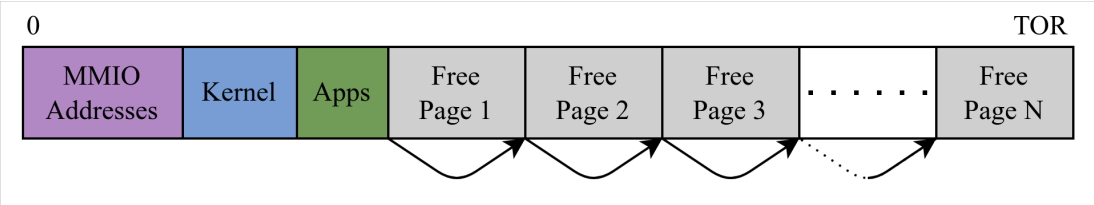

Figure 5.3: The State of RAM After ZenOS Finishes Initializing its Free Memory List.

needed to create a page table. The first page table created is for the ZenOS kernel. After a page has been allocated, ZenOS walks through the new page table to map all the regions of memory that it uses (as discussed in 5.1.2) in order to ensure that it will not page fault.

After these structures have been initialized, virtual memory can be enabled and ZenOS can begin setting up page tables for user-level processes. Each user-level process is assigned a page table, and ZenOS maps its regions of memory using *walk()* as with the kernel page table, though the user page tables have access to much less physical memory by default.

Setup for the UART server begins immediately after ZenOS finishes initializing virtual memory. ZenOS loads the UART server into memory, then jumps to *load_registers* to set up the UART server's state. At boot, the data registers of the CPU should just be 0, so *load_register*'s main job at this time is to switch to the UART server's page table and jump to its main function.

The UART server starts by creating several capabilities to regions of its memory. This list of capabilities is stored in its memory until requested by an application. Once these capabilities have been created, the UART server returns to the kernel the locations of three of its functions: *allocate_print_buffer*, *flush_print_buffers*, and *request_input_char*. The locations of these functions are stored within the kernel so that it may call them by jumping to their location within the UART server's binary. To send the locations of these functions to the ZenOS kernel, the UART server loads

the addresses of these functions into its a registers and executes the Send Print Buffer Functions system call. Future work will expand the UART server to process and handle requests similarly to how ZenOS handles traps/syscalls in order to make the UART server more expandable.

The system call handler will store the server's function addresses within ZenOS's memory, and then return to load_registers and load the shell. The shell will ask the user which application to run, and once it receives valid input, it will execute the appropriate execve syscall. Once loaded into memory and the CPU, a user application may run uninterrupted until a timer interrupt occurs.

This loop describes the majority of ZenOS's execution – the shell is loaded, an application is selected and loaded, the application makes some system calls, ZenOS handles those calls and returns to the application, a timer interrupt occurs, the UART buffers are flushed, and the cycle restarts. This cycle can also restart if the application finishes its execution and triggers an exit syscall. This cycle can be seen in Figure 5.4.

5.3 Evaluation

Several tests have been devised to measure ZenOS's performance and ensure its proper functionality. These tests are designed to measure differences between ZenOS's methods and those used by conventional operating systems today. As such, they have all been run in RISC-V QEMU, and the performance results are presented in terms of execution cycles. While the exact number of cycles can vary from run to run, the different methods measured here are distinct enough that conclusions about their efficiency can be made. Each of these tests was run at least 10 times, with an average standard deviation of 191.7 cycles.

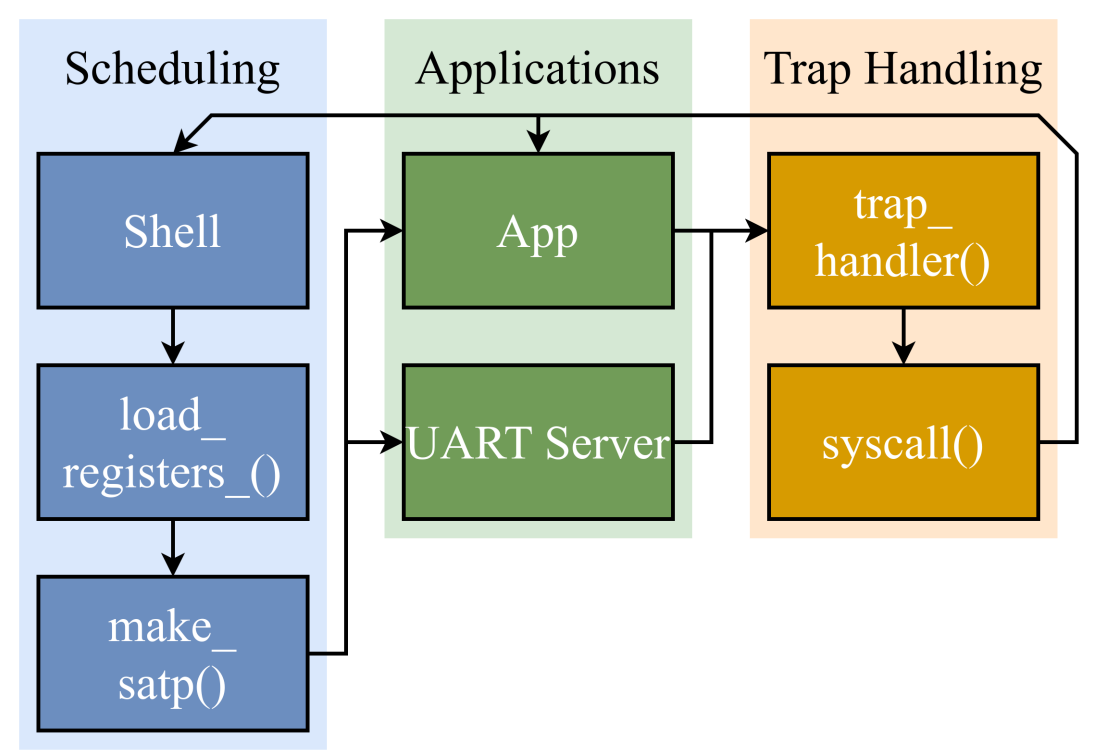

Figure 5.4: The Standard Cycle of Execution Within ZenOS.

5.3.1 Loading Methods

Section 4.2 describes a method of loading applications into memory using capabilities that would prevent the operating system from being able to access their memory. While the remaining research questions discussed in that section prevent such a protocol from being implemented today, its performance can still be emulated by following the steps described and using placeholder functions for the hardware features that are still being developed.

This performance was measured by loading an application into memory in the standard method, then loading the same application into memory and following the steps that would be needed to coordinate the OS loader and the firmware in a full enclave system. The instructions used to move application data into a runnable location of memory are the same in both methods, and as such have been removed from

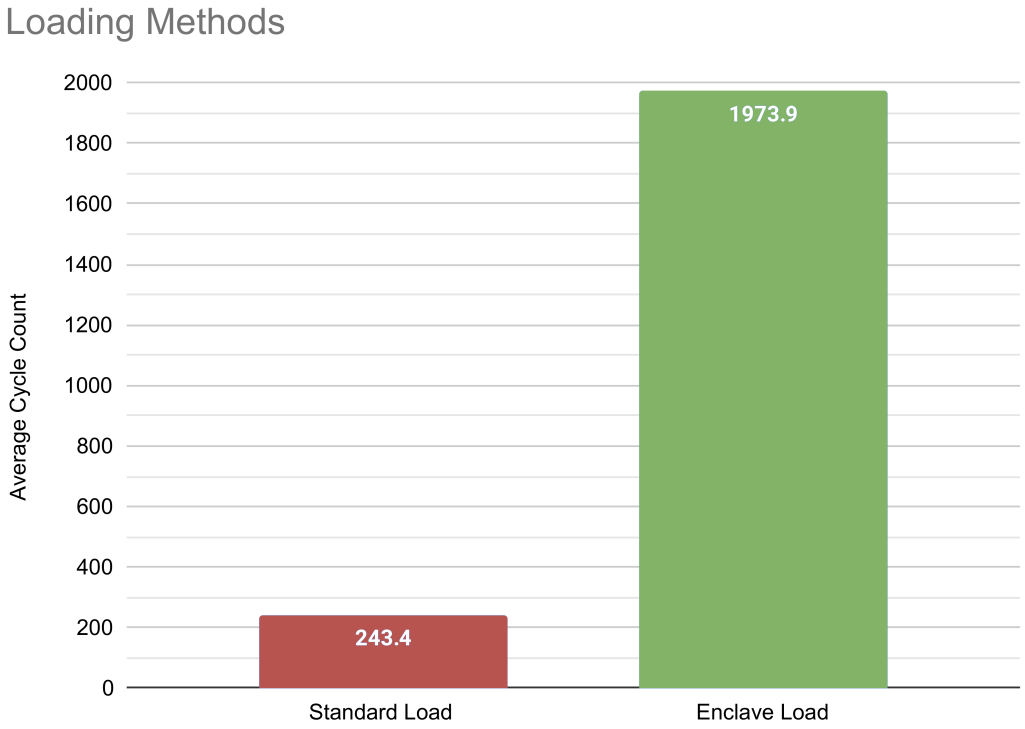

Figure 5.5: Overhead in Traditional Application Loading and Capability-Based Enclave Loading.

both methods' cycle count. This ensures that only the overhead in both systems is measured. Figure 5.5 shows that the enclave loading protocol produces more than 1700 additional cycles of overhead. The standard deviation of the standard loading method was 21.1 cycles, and that of the enclave loading was 245.2 cycles. The additional overhead when loading an enclave is mainly due to the three additional context switches needed to switch control between the firmware and the operating system. This could be greatly mitigated by running the two privilege levels simultaneously, perhaps on different threads. Even if such a performance overhead is inherent to enclave loading, the security benefits of reducing applications' TCB will outweigh this one-time cost at load time for many systems.

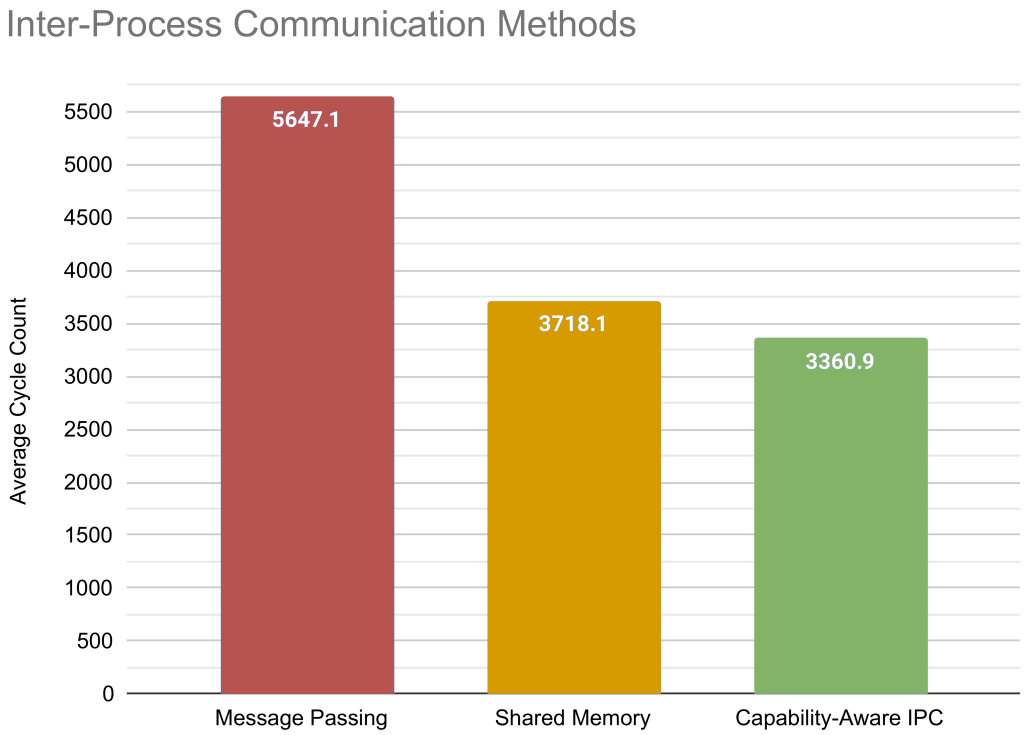

Figure 5.6: Overhead in Message Passing, Shared Memory, and Capability Sharing.

5.3.2 Inter-Process Communication Methods

Sections 3.7 and 4.3 discuss four methods of inter-process communication - message passing, shared memory, data capability sharing, and instruction capability sharing. The first three have been tested, and the results are shown in Figure 5.6. Instruction capabilities represent an exciting possibility for future inter-process communication, though the software hardware needed to support such a system does not yet exist. As capability ecosystems continue to mature, instruction capabilities may enable a wide range of new functionality.

Figure 5.6 shows that capability-aware inter-process communication represents an improvement on current shared memory of around 10%. The standard deviations for message passing, shared memory, and capability-aware IPC are 198.0, 278.6, and

273.1 cycles respectively. Message passing has the largest amount of overhead by far, and also suffers as this overhead must run for every message to be sent. As such, this protocol should be used when a small number of messages need to be sent or in systems where simplicity of implementation is more important than raw performance. Shared memory requires 65% of the instructions of message passing, and this overhead only runs when the IPC is first set up. Capability-aware IPC further improves on shared memory both in terms of performance and in terms of the level of control afforded to the communicators.

5.3.3 Printing Methods

Capability-aware IPC is used in ZenOS to allow for communication between processes running on ZenOS and its UART server. This protocol is discussed in Section 5.1.3.1. Figure 5.7 shows how this protocol compares to that of a traditional, monolithic kernel.

In the monolithic protocol measured in Figure 5.7, the printing application sends its string to the kernel, which then writes it directly to the UART. This requires far fewer cycles than the "First Server" protocol, which measures the overhead of the first print sent to the UART server in ZenOS's protocol. The overhead of the first print is so much higher due to the setup and context switching needed to allocate and share a capability between the two processes. After the first print, however, the overhead of capability-aware printing falls to far below that of the monolithic protocol. If, as ZenOS's shell does, the application always requests a print buffer flush after writing to its capability, its overhead is 32.6% of the first print and 50.1% of the monolithic protocol. Most applications, however, do not need user input as frequently as the shell and as such do not regularly need to request print buffer flushes. This lazy flushing takes just 113 cycles on average, the lowest by far. This is because the user

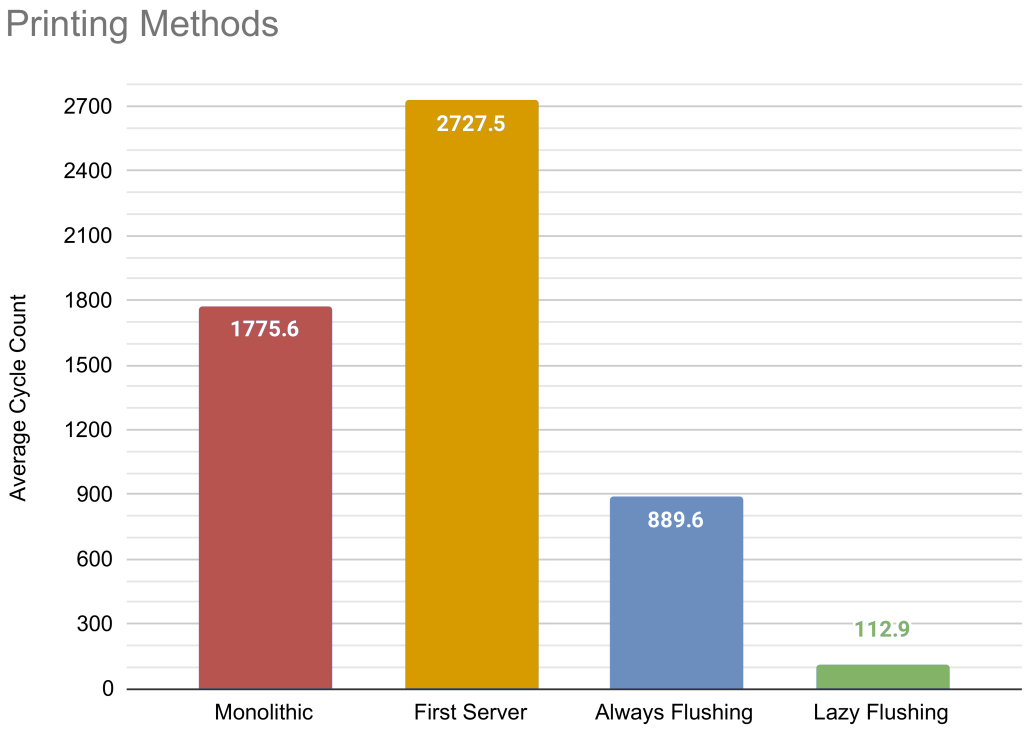

Figure 5.7: Overhead Caused by Printing with and without Capabilities.

library needs only to write the string to be printed to a buffer, which can be flushed when the process exits, its buffer is filled, or another process requests a buffer flush. The standard deviations of these four tests were 96.1 (monolithic), 523.2 (first server), 81.1 (always flush), and 8.7 (lazy flush) cycles.

The tests completed here show that ZenOS's inter-process communication performance is on par with or exceeds that of traditional operating systems. While the most secure method of capability-aware loading does require more setup than the loading systems that exist today, this overhead only occurs when the process is first loaded into RAM. As such, its performance impact will represent a very small percent of the total number of instructions executed by an application during its runtime.

ZenOS is available for use at {GITLAB}. Compilation of ZenOS requires the riscv64-gnu toolchain to be installed; instructions and a script to set up this toolchain

can be found in the docs folder of the aforementioned repository. Several test applications are included with ZenOS; these may be modified or additional added provided that they only use the system calls found in Table 5.1.

5.4 Conclusion and Future Work

This dissertation has provided an overview of the architecture and potential of a capability-aware operating system, including how capabilities can enhance memory security through more fine-grained access control. Analyzing ZenOS illustrated how some of these improvements might be implemented outside of theory, and how ZenOS will serve as a foundation for future research into capabilities.

ZenOS has strong foundations in its management of virtual memory, scheduling, inter-process communication, trap handling, libraries, and system calls. However, due to its recent inception, significant opportunities for improvement still exist. Future research should focus on improving compatibility with applications that were not designed for Zeno, such as libraries like *glibc*. Improving compatibility will not only increase the security of the applications that run on ZenOS, but also improve researchers' ability to test ZenOS and improve its security. It will also allow for future research to better answer some of the questions asked in Chapter 4.

Overall, ZenOS represents a robust framework for advancing the state of memory security. Its ongoing development could pave the way for substantial improvements in safeguarding systems against increasingly sophisticated threats to how we store and use data. As this framework continues to be built upon and improved, it will become a keystone in future computing systems that are more resilient, reliable, and expandable than the systems we rely upon today.

"Back to the building blocks: A path toward secure and measurable software", Tech. rep., The White House (2024).

Anderson, J. P., "Computer security technology planning study", Tech. Rep. ESD-TR-73-51, Electronic Systems Division, AFSC, L. G. Hanscom Field, Bedford, MASS. 01730 (1972).

Apple, "Secure enclave", `https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web` (2024).

Atlidakis, V., J. Andrus, R. Geambasu, D. Mitropoulos and J. Nieh, "Posix abstractions in modern operating systems: The old, the new, and the missing", in "Proceedings of the Eleventh European Conference on Computer Systems", pp. 1–17 (2016).

Böck, B., D. Huemer and A. M. Tjoa, "Towards more trustable log files for digital forensics by means of "trusted computing"", in "2010 24th IEEE International Conference on Advanced Information Networking and Applications", pp. 1020–1027 (IEEE, 2010).

David, F. M., J. C. Carlyle and R. H. Campbell, "Context switch overheads for linux on arm platforms", in "Proceedings of the 2007 workshop on Experimental computer science", pp. 3–es (2007).

Ehret, A., J. Abraham, M. Isakov and M. A. Kinsy, "Zeno: A scalable capability-based secure architecture", URL `https://arxiv.org/abs/2208.09800` (2022).

Elphinstone, K. and G. Heiser, "From l3 to sel4 what have we learnt in 20 years of l4 microkernels?", in "Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles", pp. 133–150 (2013).

Heiser, G., "The sel4 microkernel–an introduction", The seL4 Foundation **1** (2020).

Huber, N., M. von Quast, M. Hauck and S. Kounev, "Evaluating and modeling virtualization performance overhead for cloud environments.", CLOSER **11**, 563–573 (2011).

Intel, "What is intel management engine?", `https://www.intel.com/content/www/us/en/support/articles/000008927/software/chipset-software.html` (2023).

Intel, *Intel® 64 and IA-32 Architectures Software Developer's Manual* (2024).

Kaffes, K., T. Chong, J. T. Humphries, A. Belay, D. Mazières and C. Kozyrakis, "Shinjuku: Preemptive scheduling for {μsecond-scale} tail latency", in "16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)", pp. 345–360 (2019).

Kerrisk, M., "malloc(3) - Linux manual page — man7.org", https://www.man7.org/linux/man-pages/man3/malloc.3.html, [Accessed 21-09-2024] (2024).

Lantz, B., R. Hall and J. Couraud, "Locking down log files: enhancing network security by protecting log files", Issues in Information Systems **7**, 2, 43–47 (2006).

Lemmens, M., "Stack canaries - gingerly sidestepping the cage", https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/ (2021).

Li, C., C. Ding and K. Shen, "Quantifying the cost of context switch", in "Proceedings of the 2007 workshop on Experimental computer science", pp. 2–es (2007).

Li, J., N. K. Sharma, D. R. Ports and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency", in "Proceedings of the ACM Symposium on Cloud Computing", pp. 1–14 (2014).

Liljestrand, H., T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg and N. Asokan, "PAC it up: Towards pointer integrity using ARM pointer authentication", in "28th USENIX Security Symposium (USENIX Security 19)", pp. 177–194 (USENIX Association, Santa Clara, CA, 2019), URL https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand.

Luo, T., X. Wang, J. Hu, Y. Luo and Z. Wang, "Improving tlb performance by increasing hugepage ratio", in "2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing", pp. 1139–1142 (IEEE, 2015).

Maurer, U. M. and S. Wolf, "The diffie–hellman protocol", Designs, Codes and Cryptography **19**, 2, 147–171 (2000).

Meer, H. *et al.*, "Memory corruption attacks: The (almost) complete history", Blackhat USA (2010).

MITRE, "CWE - 2023 Top 25 Most Dangerous Software Weaknesses", https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html (2023).

MITRE, "Cwe-664: Improper control of a resource through its lifetime", https://cwe.mitre.org/data/definitions/664.html (2024).

Ravichandran, J., W. T. Na, J. Lang and M. Yan, "Pacman: attacking arm pointer authentication with speculative execution", in "Proceedings of the 49th Annual International Symposium on Computer Architecture", ISCA '22, p. 685–698 (Association for Computing Machinery, New York, NY, USA, 2022), URL https://doi.org/10.1145/3470496.3527429.

Roemer, R., E. Buchanan, H. Shacham and S. Savage, "Return-oriented programming: Systems, languages, and applications", ACM Transactions on Information and System Security (TISSEC) **15**, 1, 1–34 (2012).

Saito, T., R. Watanabe, S. Kondo, S. Sugawara and M. Yokoyama, "A survey of prevention/mitigation against memory corruption attacks", in "2016 19th International Conference on Network-Based Information Systems (NBiS)", pp. 500–505 (IEEE, 2016).

Wagle, P., C. Cowan *et al.*, "Stackguard: Simple stack smash protection for gcc", in "Proceedings of the GCC Developers Summit", vol. 1 (Citeseer, 2003).

Walli, S. R., "The posix family of standards", StandardView **3**, 1, 11–17 (1995).

Wang, X., H. Chen, A. Cheung, Z. Jia, N. Zeldovich and M. F. Kaashoek, "Undefined behavior: what happened to my code?", in "Proceedings of the Asia-Pacific Workshop on Systems", pp. 1–7 (2012).

Waterman, A., Y. Lee, R. Avižienis, D. Patterson and K. Asanović, *The RISC-V Instruction Set Manual Volume II - Privileged Architecture*, RISC-V Foundation (2017).

Watson, R. N., S. W. Moore, P. Sewell, B. Davis and P. Neumann, "Capability hardware enhanced risc instructions (cheri)", `https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/` (2019).

Zheng, W., Y. Wu, X. Wu, C. Feng, Y. Sui, X. Luo and Y. Zhou, "A survey of intel sgx and its applications", Frontiers of Computer Science **15**, 1–15 (2021).