2021

Revisiting Isolation For System Security And Efficiency In The Era Of Internet Of Things

Lele Ma
William & Mary - Arts & Sciences, lelema.cn@gmail.com

Recommended Citation

Ma, Lele, "Revisiting Isolation For System Security And Efficiency In The Era Of Internet Of Things" (2021). *Dissertations, Theses, and Masters Projects.* William & Mary. Paper 1673273635.
https://dx.doi.org/10.21220/s2-sqd8-hs55

Revisiting Isolation for System Security and Efficiency in the Era of Internet of Things

Lele Ma

Jinan, Shandong, China

Bachelor of Engineering, Shandong University, 2012
Master of Engineering, University of Chinese Academy of Sciences, 2015

A Dissertation presented to the Graduate Faculty of
The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
August 2021

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Lele Ma

Lele Ma

Approved by the Committee, August 2021

Qun Li

Committee Chair
Qun Li, Professor, Computer Science
College of William & Mary

Weizhen Mao

Weizhen Mao, Professor, Computer Science
College of William & Mary

Adwait Jog

Adwait Jog, Associate Professor, Computer Science
College of William & Mary

[illegible]

Dmitry Evtyushkin, Assistant Professor, Computer Science
College of William & Mary

[illegible]

Ang Li, Computer Scientist, HPC group
Pacific Northwest National Laboratory

ABSTRACT

Isolation is a fundamental paradigm for secure and efficient resource sharing on a computer system. However, isolation mechanisms in traditional cloud computing platforms are heavy-weight or just not feasible to be applied onto the computing environment for Internet of Things(IoT). Most IoT devices have limited resources and their servers are less powerful than cloud servers but are widely distributed over the edge of the Internet. Revisions to the traditional isolation mechanisms are needed in order to improve the system security and efficiency in these computing environments.

The first project explores container-based isolation for the emerging edge computing platforms. We show a performance issue of live migration between edge servers where the file system transmission becomes a bottleneck. Then we propose a solution that leverages a layered file system for synchronization before the migration starts, avoiding the usage of impractical networking shared file system as in the traditional solution. The evaluation shows that the migration time is reduced by 56% – 80%.

In the second project, we propose a lightweight security monitoring service for edge computing platforms, base on the virtual machine isolation technique. Our framework is designed to monitor program activities from underneath of an operating system, which improves its transparency and avoids the cost of embedding different monitor modules into each layer inside the operating system. Furthermore, the monitor runs in a single process virtual machine which requires only $\leq$32MB of memory, reduces the scheduling overhead, and saves a significant amount of physical memory, while the performance overhead is an average of 2.7%.

In the third project, we co-design the hardware and software system stack to achieve efficient fine-grained intra-address space isolation. We propose a systematic solution to partition a legacy program into multiple security compartments, which we call *capsule*s, with isolation at byte granularity. Vulnerabilities in one *capsule* will not likely affect another *capsule*. The isolation is guaranteed by our hardware-based ownership types tagged to every byte in the memory. The ownership types are initialized, propagated, and checked by combining both static and dynamic analysis techniques. Finally, our co-design approach could remove most human refactoring efforts while avoiding the untrustworthiness as well as the cost of the pure software approaches.

In brief, this proposal explores a spectrum of isolation techniques and their improvements for the IoT computing environment. With our explorations, we have shown the necessity to revise the traditional isolation mechanisms in order to improve the system efficiency and security for the edge and IoT platforms. We expect that many more opportunities will be discovered and various kinds of revised or new isolation mechanisms for the edge and IoT platforms will emerge soon.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

Studying in the US is a journey with so many exciting as well as astonishing moments for me. My PhD pursuit here is very much blessed, by the help of so many people.

First, I thank my advisor, Prof. Qun Li, for his generous support on my research interests, as well as his nuanced care for my living. Professionally, he has given me many professional suggestions over the projects in my thesis as well as much free space to pursue my interests. Thanks to the valuable freedom he allowed, I have been able to learn what I dreamed of, especially for the interest of software-hardware co-design project that has brought me a deeper understanding of the computer system from bottom up. Personally, he had helped me to resolve many difficulties to adapt the new environment here in the US.

I thank Prof. John Criswell at the University of Rochester. Working with John opened a new world of system security for me and inspired me many new ideas that I am still wishing to develop in the future.

I thank Dr. Ang Li at Pacific Northwest National Laboratory. Ang taught me a new skill of how to build a complex system from scratch more efficiently and effectively – the art of growing a minimal toy system into a full-fledged complex system with spiral iterations.

I thank Senior Security Researcher Dr. Tamas K Lengyel. Tamas instructed me to learn hardware events in LibVMI and many Xen internals into the deep during my participation of google summer of code with him.

I thank many teachers and students who helped me improve over the years. To the teachers whose classes or mentoring I have learnt a lot from: Prof. Xu Liu, Prof, Adwait Jog, Prof. Gang Zhou, Prof. Weizhen Mao, Dr. Tim Davis, Mr. Chris Shenefiel, Dr. Sarah Glosson, and Dr. Leslie Bohon at William & Mary, and Prof. John Criswell, Prof. Sreepathi Pai, Prof. Chen Ding, Prof. Micheal Scott from University of Rochester, and Prof. Robert J. Walls from Worcester Polytechnic Institute. And many thanks to Zhuojia Shen, Jie Zhou, and Yufei Du for your effective help on my LLVM study and our impressive collaboration during my visit.

I thank the students in our lab, Yutao Tang, Zijiang Hao, Shanhe Yi, Nancy Carter, Cheng Li, Zeyi Tao, Yunlong Mao, and Qi Xia. You are all awesome and thank you for the interesting discussions we have been on during our group meetings.

I thank my thesis committee members, Prof. Weizhen Mao, Prof. Adwait Jog, Prof. Dmitry Evtyushkin, and Dr. Ang Li, for their willingness to serve in my committee, and their helpful comments on my presentation and thesis.

I thank Vanessa Godwin, Jacqulyn Johnson, Dale Hayes, at our Computer Science Department office and Eva Wong, Zabrina Williams, and Emily Bailey at Reves Center for their prompt help with my Ph.D. program. I thank the Graduate Coordinator Michelle Kiso for her prompt help during my visit at University of Rochester.

Last and most importantly, I thank my family for their constant encouragement. Without their unconditional support, I will not be who I am today. Special thanks to Dr. Ruiqin Tian, especially for her strong encouragement during the hard time.

To my family

LIST OF TABLES

LIST OF FIGURES

Revisiting Isolation for System Security and Efficiency in the Era of Internet of Things

Chapter 1

Introduction

1.1 Motivation

Isolation is one of the fundamental paradigms since the birth of computer systems. First, isolation plays an important rule for resource sharing on modern computing systems. For example, resource isolation between different users on a cloud machine enables the efficient sharing of the costly computing resources. Second, isolation is important to secure computing systems. It is one of a few techniques that can defend against unknown vulnerabilities in the program. Program can be separated into different function modules and each module can be isolated in a security domain. In this way, security breaches in one domain will not likely to affect other domains. For example, isolation between the kernel and user space in a computer system effectively reduces the risk of software vulnerabilities that are caused by a diversity of user applications. Until now, we have seen process-based isolation, virtual machines, Linux containers, sandboxes, software fault isolation, Intel SGX, Intel MPK, ARM TrustZone, etc., with a variety of granularities of isolation.

However, when the Internet of Things (IoT), and edge computing platforms [1] bring several new features to the computing environment, many traditional isolation mechanisms become inefficient or just do not work.

[1]This thesis will use *edge servers* to refer a cluster of computing nodes (such as *cloudlet* servers [141]) in close proximity to the end users and *IoT devices* as client device that receives services from edge servers.

On the server side, edge servers are usually not in a centralized environment as in data centers for cloud computing, and they are less powerful but more performance sensitive than traditional cloud servers. Edge servers are mostly deployed distributively in large geographical areas. Consequently, the network bandwidth and latency between different edge computing nodes can be rather limited, rendering many traditional cloud services, such as networking shared file system, live migration, etc., unrealistic to be used on the edge. Therefore, the first project (Chapter 3) aims to improve the efficiency of live migration services between edge nodes. Another problem is that the protection of a large amount of small edge servers on the edge becomes challenging. Without the centralized deployment, traditional security solutions can be inefficient to be deployed over large geographical areas. In addition, edge servers are relatively less powerful than high performance computers in the cloud. Therefore, traditional security solutions in the cloud either cannot be directly deployed to the edge, due to either execution overhead or engineering overhead. Therefore, the second project in this thesis (Chapter 4) proposes a lightweight memory monitoring solution to improve the security of edge servers.

On the client side, IoT devices are mostly equipped with low-end processors but require real-time response. Most IoT devices are embedded systems designed for domain specific tasks, where many features for a general purposed design are downgraded or simply removed. With these fundamental changes at hardware level, many traditional isolation solutions on a general purpose machine are no longer suitablefor such small devices. For example, many IoT devices does not have the Memory Management Unit (MMU) in favor of real time response. In practice, this have resulted in almost no protection between the applications and the OS kernel running on the same device. Therefore, IoT devices need either revised or new isolation mechanisms in order to be trustworthy. Following this route, our third project (Chapter 5) proposes a fine-grained isolation technique to improve the memory safety for IoT devices.

In brief, this thesis work explores different isolation techniques on the edge servers and IoT devices. On the server side, this thesis has resolved two problems, including a performance problem for live migration between container-based edge servers, and a security monitoring problem on virtual-machine based edge servers. On the client side, this thesis has described the need for the new fine-grained isolation technique and proposed a software-hardware co-design solution for efficient fine-grained intra-address space isolation on IoT systems.

1.2 Efficient Live Migration across Edge Servers

As we mentioned above, many services, such as networking shared file system, live migration, etc., are unrealistic to be used on the edge servers. Live migration has long been used in traditional cloud computing environment. Traditional migration techniques assumes all the file system are shared over high speed network between the source node and target node, which means there is no need for an explicit file system synchronization during migration. However, many edge servers are deployed over the wide area network (WAN). In such a environment, the network bandwidth and latency between different edge servers can be rather limited. Therefore, networking shared file system is not practical for edge servers. A naive solution would be to transfer the entire file system during live migration. However, as expected, the delay of migration will take too long (hours) to be useful.

To resolve the issue, we have investigated the state of the art Docker container's layered file system. We proposed a new live migration solution that does not depends on a networking shared file system. In this solution, file system are managed in different layers where most layers are read only. Therefore, we propose to transfer most of the read only file system layers before migration starts, together with a base image for the runtime memory. During migration, we only need to transfer the latest updated files and dirty memory to reduce the transmission traffic. This reduces the migration time significantly as we will show in Chapter 3.

1.3 Lightweight Security Monitor for Edge Servers

As we mentioned, most edge computing servers have limited resources comparing to cloud servers. This makes them more sensitive to heavy weight monitoring solutions such as virtual machine introspection. In addition, edge servers are distributively deployed in large geographical areas, which makes the deployment and maintenance of security monitoring solutions more challenging than in a centralized cloud center.

Therefore, in the second project, we propose a security monitoring solution for edge servers. It is deeply trustworthy without the trust of entire operating system; It is lightweight and applicable to low performance machines; It can be remotely deployed and maintained efficiently without physical intervention on the servers. The key technique is to build a virtual machine introspection monitor in in a minimized library operating system (or a Unikernel as we will introduce in Section 2.2). The evaluations on our prototype system successfully shows the effectiveness and the efficiency of our monitoring solution, as will be described in Chapter 4.

1.4 Secure IoT Systems with Hardware Ownership Tags

The client side on edge computing platforms is composed of a various kinds of IoT devices. These include desktop machines, laptops, mobile phones, as well as the much less powerful embedded devices such as smart watches, smart locks, smart bands, etc. Secure and efficient isolation between different modules on such devices are critically important for the security defense on such devices. Intra-address space isolation is much more efficient than virtual memory based isolation mechanisms. Therefore, it is widely used on IoT devices. However, existing intra-address space isolation mechanisms are rather limited. First, they usually support only a limited number of isolation domains. Second, they usually support only coarse-grained isolation in the address space. Third, they usually partially support pointer safety or support isolation in the address space, instead of support both of them. Fourth, most of them require intensive changes in the legacy code in order to enforce the security policies.

Therefore, the third project designs and implements the *Capsule* system. It is an architectural extension along with a full stack toolchain and system support. It supports fine-grained isolation in the same address space. It also supports both temporal and spatial pointer safety to harden the programs writen in unsafe languages such as C programming language. It has fine-grained memory protection at the granularity of instructions and data words. More importantly, it requires minimal porting efforts to protect legacy programs. The prototype system is built by extending an open sourced MIPS core, with the LLVM toolchain, and FreeBSD operating system. Evaluation shows the effectiveness of the system as we will discuss in Chapter 5.

1.5 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 introduces some background conceptions that are necessary to understand this work. Chapter 3 introduces the work to improve the efficiency of container migration by leveraging the Docker layered file system. Chapter 4 introduces the work where a lightweight virtual machine introspection framework is designed to monitor the security of edge servers. Chapter 5 presents the *Capsule* system for fine-grained isolation with hardware-assisted ownership tags in the memory. Chapter 6 concludes this thesis.

Chapter 2

Background

2.1 Docker Containers

Docker is a set of platform as a service (PaaS) products that deliver software in packages called containers [88] . Docker containers allows the package of the entire developing environment into container images and could be shared among different operating systems including Linux, Windows, MacOS, etc. This allows developers to build, share and run their applications anywhere without the hassle of setting up the new developing environment from time to time. Docker containers are based on OS-level virtualization where different containers on the same host share the same operating sytem kernel [137].

2.2 Xen and Unikernels

Xen Hypevisor [129] is a virtual machine monitor which allows multiple commodity operating systems to share the hardware on the same host in a safe and resource managed fashion, but without sacrificing either performance or functionality [20]. Beside commodity operating system such as Linux and Windows, Xen also supports running Unikernels such as Mini-OS, MirageOS, etc.

A unikernel is a specialized library operating system that has only a single address space and without kernel/user mode separation as in a regular operating system [111].

It is usually built with a minimal set of modules or libraries that are required by the functionality of the application. A unikernel could run directly on a hypervisor such as Xen or run directly on hardware without an intervening OS.

Xen Mini-OS [127] is one of the earliest library operating system on Xen hypervisor. It is used to build stub domains, a set of small virtual machines that disaggregate the Xen Domain 0, and it is used as the basis for the development of Unikernels [8].

2.3 QEMU

QEMU [22] is a generic and open source machine emulator and virtualizer. As a machine emulator, QEMU can run operating systems and programs written for one architecture (e.g. a MIPS processor) on a different machine (e.g. an x86 PC). By using dynamic translation, it achieves very good performance. It can be extended for fast evaluating the functionality of a new architecture design. However, QEMU cannot be used to measure the performance of the new architecture design because it lacks the architectural timing measurements (Gem5 can be used instead). In addition, QEMU can also be used as a virtualizer, where it could execute the guest code directly on the host CPU by multiplexing the resources of the host machine (e.g., using the KVM [97] kernel module in Linux).

2.4 gem5 Simulator

The gem5 simulator [5] is a modular platform for computer system architecture research. It could simulate system-level architecture as well as processor microarchitecture. It supports multiple processor models such as ARM, x86, RISC-V, SPARC, and Alpha, where each model can be extended with new functionalities for research purposes.

On the gem5 platform, users could composite a full stack computer system by combining the models for the CPU, cache, and the memory, as well as disk images for booting. Users could also play with only partial of the system. For example, gem5 can replay CPU traces or memory traces to evaluate the performance of the memory system without a CPU model or virtual address translation models.

2.5 Bluespec and *bluesim*

Bluespec [119] is a high-level hardware description language (HDL) in develop since 2000. Now it has been used in production designs like RISC-V cores Flute [4], Piccolo [9], and Toooba [10]. It has a powerful type system that can prevent errors prior to synthesis time. In addition to the System Verilog-like syntax, it also comes with syntactic flavor of the Haskell functional programming language and both flavors are interchangeable. The Bluespec compiler emits standard Verilog so it can be compatible with any synthesis toolchain.

In addition to using other simulators for Verilog descriptions, Bluespec also comes with its own cycle simulator *bluesim*. The compiled hardware description could be linked into *bluesim* simulation environment and an executable binary will be generated. When the binary is executed, the hardware module is simulated.

2.6 LLVM toolchain

LLVM [7] is a collection of modular and reusable compiler and toolchain technologies. It uses Clang [3] as front-end for C language family (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript). Clang can be extended to support user annotations in the source code. The front-end compiles the source code into LLVM intermediate code representation (LLVM IR), or LLVM bitcode. Then LLVM core libraries could provide a various of analysis and optimizations on LLVM bitcode so that they do not depend on the source code or the target.

LLVM code generator (codegen) is the backend which translates LLVM IR into target specific machine code. The code generator in LLVM contains multiple stages(or passes in LLVM): instruction selection, register allocation, scheduling, code layout optimization, and assembly emission. LLVM uses its own linker LLD [6] as drop-in replacement for the default system linkers. Linkers are responsible to generate executable file by linking multiple object files generated by the LLVM. LLD supports Link-Time Optimization(LTO) and allows LLVM to do whole program optimization during linking. LLD supports a wide range of binary formats including ELF(Unix), PE/COFF(Windows), Mach-O(macOS) and WebAssembly.

2.7 Privilege Separation and Isolation

Privilege separation [138] is an important line of defense against unknown vulnerabilities in a system. In a privilege separated system, a program is partitioned into multiple isolated security domains and each keeps its least privileges that are just enough to do its tasks. In such a system, security breaches in one domain will not likely affect other domains.

Privilege separation has been widely used in modern computer systems to ensure the secure sharing of the computer resources such as CPU, memory, storage, etc. For example, different processes are isolated inside an operating system so that different users and different programs won't interfere with each other erroneously. Another example is that inside the address space of a single process, user memory and kernel memory are separated and isolated by the operating system.

In this thesis, the term of *privilege separation* and *isolation* can be interchangeable when they are used to refer to partitioning a program in general. However, we also differentiate *isolation* from *privilege separation* in terms of the emphasized stage during program partition. We will use *privilege separation* to refer to the process to analyze the program and find out which module should be separated with other modules in a program. This process is usually determined by the security semantic of the program, and is irrelevant to what kind of isolation mechanisms will be used. On the other hand, we will use *isolation* to refer to the process where a list of separated modules and their security requirements are given as input, and these modules will be isolated into different protection domains by leveraging certain kinds of isolation mechanisms. This process is mostly focused on how to leverage specific isolation mechanisms to enforce the domain isolation, and the security requirements and how to separate the program modules are assumed to be known facts.

2.8 Intra-address Space Isolation

Intra-address space isolation is used to separate privileges inside the same process, usually used to isolate different modules or libraries inside a single user program or inside an operating system kernel. For example, Software Fault Isolation(SFI) [159] and its derived techniques, such as Native Client [173], BGI [33], etc., are pure software solution can be used to partition the entire address space into different isolated memory regions. In these techniques, every memory accesses are checked either statically or dynamically to ensure the memory address being accessed is valid (does not cross the given boundary).

There are also hardware assisted isolation mechanisms such as CPU Rings [144], Intel SGX [47], MPK [90], and ARM TrustZone [16], ARM MPU [17], etc. Hardware assistance could help to reduce the overhead significantly comparing to the pure software based solutions. In these mechanisms, the memory accesses are checked either by checking the permission bits in the page table entry (CPU Rings, Intel SGX, Intel MPK, and ARM TrustZone) or other hardware supported permission descriptors (ARM MPU). These techniques vary in how they manage the format of permission bits and how to switch the execution between different protection domains.

Chapter 3

Efficient Live Migration across Edge Servers

3.1 Overview

Mobile users across edge networks require seamless migration of offloading services. Edge computing platforms must smoothly support these service transfers and keep pace with user movements around the network. However, live migration of offloading services in the wide area network poses significant service handoff challenges in the edge computing environment. In this work, we propose an edge computing platform architecture which supports seamless migration of offloading services while also keeping the moving mobile user "in service" with its nearest edge server. We identify a critical problem in the state-of-the-art tool for Docker container migration. Based on our systematic study of the Docker container storage system, we propose to leverage the layered nature of the storage system to reduce file system synchronization overhead, without dependence on the distributed file system. In contrast to the state-of-the-art service handoff method in the edge environment, our system yields a 80%(56%) reduction in handoff time under 5Mbps(20Mbps) network bandwidth conditions.

3.2 Introduction

Edge computing has become a prominent concept in many leading studies and technologies in recent years [24, 37, 83, 84, 85, 125, 140, 141, 152, 174, 175, 176]. Since edge servers are in close proximity to the mobile end user, higher quality of services (QoS) could be provided than was possible with the traditional cloud platform [125, 140]. End users benefit from edge services by offloading their heavy duty computations to nearby edge servers [14, 53, 101, 107]. Then the end user experience with cloud services will achieve higher bandwidth, lower latency, as well as greater computational power.

One of the key challenges for edge computing is keeping quality of service guarantees better than traditional cloud services while offloading services to the end user's nearest edge server. However, when the end user moves away from the nearby edge server, the quality of service will significantly decreases due to the deteriorating network connection. Ideally, when the end user moves, the services on the edge server should also be live migrated to a new nearby server. Therefore, efficient live migration is vital to enable the mobility of edge services in the edge computing environment.

Several approaches have been investigated to live migrate offloading services on the edge. Virtual machine (VM) handoff [80, 81] divides VM images into two stacked overlays based on VM synthesis [141]. During migration, only the overlay on the top is transferred from the source to the target server instead of the whole VM image volume. This significantly reduces data transfer size during migration. However, a virtual machine overlay can be tens or hundreds of megabytes in size, thus the total handoff time is still relatively long for latency sensitive applications. For example, OpenFace [14], a face recognition service, will cost 247 seconds to migrate on a 5Mbps wide area network (WAN), which barely meets the requirements of a responsive user experience. Additionally, VM overlays are hard to maintain, and are not widely available in the industrial or academic world.

In contrast, the widely deployed Docker platform raises the possibility of high speed service handoffs on the network edge. Docker [88] has gained popularity in the industrial cloud. It employs layered storage inside containers, enabling fast packaging, sharing, and shipping of any application as a container. Live migration of Docker containers is achievable. For example, *P.Haul* [65] supports live migration containers on Docker 1.9.0 and 1.10.0. They are developed based on a user level process checkpoint and restore tool *CRIU* [52]. But *P.Haul* will transfer the whole container file system in a bundle during the migration, regardless of storage layers, which could induce errors as well as high network overhead.

In exploring an efficient container migration strategy tailored for edge computing, we focus on reducing the file system transfer size by leveraging Docker's layered storage architecture. Docker's storage allows only the top storage layer to be changed during the whole life cycle of the container. All layers underlying the top layer will not be changed. Therefore, we propose to share the underlying storage layers before container migration begins, and only transfer the top layer during the migration itself.

In this work, we build a system which allows efficient live migration of offloading services on the edge. Offloading services are running inside Docker containers. The system will reduce the transferred file volumes by leveraging layered storage in the Docker platform. Our work addressed following challenges during this project:

First, the internals of Docker storage management must be carefully studied. Few studies have been published regarding Docker storage. Reading the raw source code enables better understanding of the inner infrastructure.

Second, an efficient way to take advantage of Docker's layered storage must be carefully designed to avoid file system redundancy. We found that Docker creates a random number as local identification for each image layer downloaded from the cloud. As a result, if two Docker hosts download the same image layer from the same storage repository, these layers will have different reference identification numbers. Therefore, when we migrate a container from one Docker host to another, we must recognize whether there are any image layers with different local identification numbers yet having the same content, thus avoiding transfer of redundant image layers during the container migration.

Third, besides the file system, we also need to optimize transmission of the raw memory pages, used to restore the live status of the offloading service. Binary data are different in format then the file system, and thus must be treated separately.

Last, in terms of end user experience, we need to reduce the user-experienced connection interruption during service migration. It is possible that user-experienced interruption interval could be shorter than the actual migration time through a well designed migration process strategy. Ideally, our goal is seamless service handoff wherein users will not notice that their offloading service has been migrated to a new edge server.

We propose a framework that enables high speed offloading service migration across edge servers over WAN. During migration, only the top storage layer and the incremental runtime memory is transferred. The total migration time and user perceived service interruption are significantly reduced. The contributions of this work are listed as below (a preliminary version of this work appeared in [109]):

- We have investigated the current status of container migration and identified performance problems.
- We have analyzed Docker storage management based on the AUFS storage driver, and studied the internal image stacking methodology.
- We have designed a framework that enables efficient live migration of offloading services by sharing common storage layers across Docker hosts.
- A prototype of our system has been implemented. Evaluation shows significant performance improvement with our design, up tp 80% on 5Mbps networks.

We will briefly introduce the motivation of this work in Section 3.3. Section 3.4 reports the systematic study of Docker storage management, and the problems of previous Docker migration tools. Section 3.5 discusses the design of our system infrastructure. In Section 3.6, the prototype system is evaluated. Section 3.7 discusses related work, and Section 3.8 concludes this work.

3.3 Motivation

In this section, we seek to answer the following questions: Why do edge applications need offloading of computation? Why is service migration needed in edge computing? Why do not use VM-based migration and why do we seek to perform migration via Docker containers?

3.3.1 Offloading Service is Essential for Edge Computing

With the rapid development of edge computing, many applications have been created to take advantage of the computation power available from the edge.

For example, edge computing provides powerful support for many emerging augmented reality (AR) applications with local object tracking, and local AR content caching [85, 141]. It can be used to offer consumer or enterprise propositions, such as tourist information, sporting event information, advertisements, etc.. The Gabriel platform [82] is designed within the context of wearable cognitive assistance applications using a Glass-like wearable device, such as Lego Assistant, Drawing Assistant, or Ping-pong Assistant. OpenFace [14] is a real-time mobile face recognition program based on a deep neural network. The OpenFace client sends pictures captured by the camera to a nearby edge server. The server runs a face recognition service that analyzes the picture and sends symbolic feedback to the user in real time. More edge applications can be found in [83, 84, 140, 174, 175]. In brief, applications on the edge not only demand intensive computations, or high bandwidth, but also require real time response.

3.3.2 Effective Edge Offloading Needs Migration for Service Handoff

As mentioned previously, highly responsive services rely upon relatively short network distances between the end user and the edge server. However, when the end user moves farther away from its current edge server, offloading performance benefits will be dramatically diminished.

In the centralized cloud infrastructure, mobility of end users is well supported since end users are connected to the centralized cloud server through WAN. However, in the edge computing infrastructure, mobile devices connect to nearby edge servers with high bandwidth and low latency connections, usually via a LAN. Therefore, when the mobile device moves farther away from its edge server, the connection will suffer from higher latency, or may even become totally interrupted.

In order to be continuously served by a nearby edge server, the offloading computation service should migrate to a new edge server that is closer to the end user's new location than the current server. We regard this process as a *service handoff* from the current edge server to the new edge server. This is similar to the *handover* mechanism in cellular networks, wherein a moving user connects to the nearest available base station, maintaining connectivity to the cellular network with minimal interruption.

However, there exists one key difference between the cellular network handover and the edge server handoff. In cellular networks, changing a base station for the mobile client is as simple as rebuilding a wireless connection. Most runtime service states are not stored on the base station but are saved either on mobile client, or on the cloud. Therefore, after re-connection, the runtime state can be seamlessly resumed through the new connection.

In the edge infrastructure, mobile devices use edge servers to offload resource-hungry or computation-intensive computations. This means that the edge server needs to hold all the states of the offloading workloads. During the service handoff from one edge server to another, all the runtime states of offloading workloads need to be transferred to the new edge server. Therefore, fast live migration of offloading services across edge servers is a primary requirement for edge computing.

3.3.3 Service Handoff via VM Migration is Not Practical

One possible solution is to use virtual machine (VM) live migration [40] to migrate a VM from one edge server to another in order to seamlessly transfer the offloading workloads. However, this approach has already been shown to be not suitable for edge computing environments in [80]. First, live migration and service handoff are optimized according to different performance metrics. While live migration aims to reduce *downtime* of the VM, service handoff aims to reduce the *total time* from the time when handoff request is issued to the completion time of the migration. This is well discussed in [80]. Second, live migration is originally designed for high performance data centers with high bandwidth networks. However, this is not possible for edge servers which are deployed over the WAN. Furthermore, live migration relies on network-based storage sharing so only run-time memory state is transferred and not storage data. Apparently, network-based storage sharing across the edge computing infrastructure is not feasible due to its widely distributed nature and low WAN bandwidth between edge servers.

In order to enable handoff across edge computing servers, much research has focused on VM migration [80, 141]. However, the total handoff time is still several minutes on a WAN network. For example, it is shown that it requires 245.5 seconds to migrate a running OpenFace instance under 5Mbps bandwidth (50ms latency) network in [80].

One of the reasons for the long latency of handoff is the large transfer size during the VM migration. VM synthesis can reduce the image size by splitting images into multiple layers, and only transferring the application-specific layer. However, the total transferred size is still in the magnitude of tens, or even hundreds of megabytes. This is because the application layer is encapsulated with the whole application, including both the static binary programs and runtime memory data. We think this is an unnecessary cost.

On the other hand, the deployment of the VM synthesis system is challenging for the legacy system. In order to enable VM synthesis, the VM hypervisor needs to be patched to track dirty memory at runtime. Also, storage of VM images must be adapted to Linux FUSE interfaces in order to track file system changes inside running VMs. Those two changes are hard to deploy in practice since they change the behavior of legacy hypervisors and file systems, along with lots of performance overhead being added.

App	Total time	Down time	FS Size	Total Size
Busybox	7.54 s	3.49 s	140 KB	290KB
OpenFace	26.19 s	5.02 s	2.0 GB	2.17GB

Table 3.1: Docker Container Migration Time (bandwidth 600Mbps, latency 0.4ms)

App	Total time	Down time	FS Size	Total Size
Busybox	133.11s	9s	140 KB	290KB
OpenFace	$\sim 3200s$	153.82s	2.0G	2.17G

Table 3.2: Docker Container Migration Time (bandwidth 15Mbps, latency 5.4ms)

3.3.4 More Efficient Migration is Achievable with Docker Containers

Since VM migration poses significant performance problems to the seamless handoff of edge services, container live migration has gained recognition for being lightweight and its ability to maintain a certain degree of isolation.

In addition, Docker containers support layered storage. Each container image references a list of read-only storage layers that represent file system differences. Layers are stacked hierarchically and union mounted as a container's root file system [87]. Layered storage enables fast packaging and shipping of any application as a lightweight container based upon sharing of common layers.

These layered images have the potential for fast container migration by avoiding transfer of common image layers between two migration nodes. With container images located in cloud storage (such as DockerHub [86]), all the container images are available through the centralized image server. Before migration starts, an edge server has the opportunity to download the system and application images as the container base image stack. Therefore, we can avoid the transfer of the container's base image during the actual migration process.

Apparently, the migration of Docker containers can be accomplished with smaller transfer file sizes than with VM migration. However, as of this writing, no tools are available for container migration on the edge environment. Container migration tools for data centers can not be directly applied to the edge of WAN network.

Table 3.1 and Table 3.2 shows our experiment with previous container migration solution under two different network environments. Table 3.1 indicates that migration could be done in 7.54 seconds for Busybox, and 26.19 seconds for OpenFace. The network connection between the two hosts has 600 Mbps bandwidth with latency of 0.4 milliseconds.

However, when the network bandwidth reduces to 15 Mbps and latency increases to 5.4 ms, container migration performance becomes unacceptable. Table 3.2 shows that the migration of the Busybox container takes 133.11 seconds with transferred size as small as 290 Kilobytes and OpenFace takes about 3200 seconds with 2 Gigabytes data transferred.

We find that one of the key factors causing this poor performance is the large size of the container's transmitted file system. In this work, we propose to reduce transmission size by leveraging the layered storage provided in Docker.

3.4 Container Storage and Migration

In this section, we discuss the inner details of container storage and the problems we found in the latest migration tool. We take Docker as an example container engine and AUFS as its storage system. Docker is becoming more popular and widely adopted in the industrial world. However, as of this writing, the technical details of Docker layered storage management are still not well-documented. Therefore, in this work, we first investigate the inner details of the Docker layered storage system, and then leverage that layering to speed up Docker container migration.

3.4.1 Container Engines and Storage Drivers

In general, Linux container engines support multiple kinds of file storage systems. For example, the Docker engine supports AUFS, Btrfs, OverlayFS, etc. [87]. LXC engine supports Btrfs, LVM, overlayFS, etc. [76]. OpenVZ containers can directly run on native ext3 file system for high performance, or Virtuozzo as networking distributed storage [123]. Some of them inherently support layered storage for easy sharing of container images, such as Docker and rkt [45]. Others, such as OpenVZ, solely support regular file systems to achieve fast native performance. We leverage the layered storage of Docker containers for efficient container migration. This strategy is also applicable to other container engines supporting layered image formats, such as rkt. However, the details of layer management techniques can vary across different container engines, thus each engine requires customization to enable image layer sharing.

Different storage drivers can define their own container image formats, thus making container migration with differing storage drivers a challenging task. It must be recognized that with the efforts of the Open Container Inititive (OCI) [89], the format and structure of the container image is evolving towards a common standard across multiple container engines. For example, both rkt and Docker can support OCI images, and the container image could be migrated between rkt and Docker hosts [46].

Docker leverages the copy-on-write (CoW) features of underlying storage drivers, such as AUFS or *overlay2*. Rkt supports Docker images consistent with OCI specifications thus it can leverage the image layers for sharing. Since Docker manages container image inherently and is one of the most popular industrialized container engines, we adopt Docker as our experimental container engine to migrate containers on the edge.

3.4.2 Layered Storage in Docker

A Docker container is created on top of a Docker image which has multiple layers of storage. Each Docker image references a list of read-only layers that represent file system differences. Layers are stacked on top of each other and will be union mounted to the container's root file system [87].

3.4.2.1 *Container Layer* and Base Image Layers

When a new container is created, a new, thin, writable storage layer is created on top of the underlying read-only stack of image layers. The new layer on the top is called the *container layer*. All changes made to the container – such as creation, modification, or

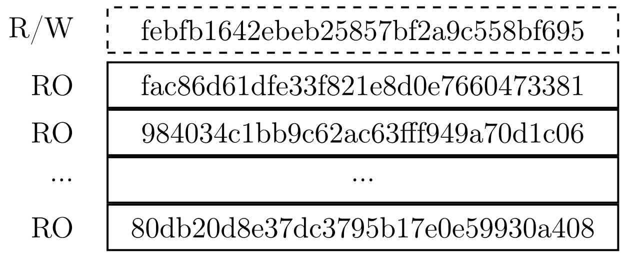

R/W
febfb1642ebeb25857bf2a9c558bf695
RO
fac86d61dfe33f821e8d0e7660473381
RO
984034c1bb9c62ac63fff949a70d1c06
...
...
RO
80db20d8e37dc3795b17e0e59930a408

Figure 3.1: OpenFace Container's Image Layer Stack

deletion of any file – are written to this *container layer* [87].

For example, Figure 3.1 shows the stacked image layers of OpenFace. Container's rootfs ID is *febfb1642ebeb25857bf2a9c558bf695* [1]. The dashed box on the top is the writable (R/W) layer – *container layer*, and all the underlying layers are readonly (RO), which are called *base image layers*. To resolve the access request for a file name, the storage driver will search the file name in order from the top layer towards the bottom layer. The first copy of the file will be returned for accessing, regardless of any other copies with the same file name in the underlying layers.

3.4.2.2 Image Layer ID Mapping

Since Docker 1.10, all images and layers are addressed by secure content SHA256 hash [87]. This content addressable design enables better sharing of layers by allowing many images to freely share their layers locally even if they don't come from the same build. It also improves security by avoiding name collisions, and assuring data integrity across Docker local hosts and cloud registries [102].

By investigating the source code of Docker and its storage structure, we find that there is an image layer ID mapping relationship which is not well documented. If the same image is downloaded from the same build on the cloud, Docker will map the original layer IDs to a randomly generated ID, called *cache ID*. Every image layer's original ID will be replaced with a unique *cache ID*. From then on, the Docker daemon will address the image layer

[1]SHA256 ID has 64 hexadecimal characters, here we truncate it to 32 hexadecimal characters in order to save space.

by this *cache ID* when it creates, starts, stops, checkpoints, or restores a container.

As a result, if two Docker hosts download the same image layer from the same repository, these layers will have different *cache ID*s. Therefore, when we migrate a container from one Docker host to another, we must find out whether those image layers with different IDs are actually referencing the same content. This is necessary to avoid redundant transfers of image layers during container migration.

3.4.2.3 Docker's Graph Driver and Storage Driver

Note that the mismatching of image layer cache IDs seems to be a flawed Docker design when it comes to container migration. However, this design is actually the image layer caching mechanism designed for the graph driver in the Docker runtime [67]. All image layers in Docker are managed via a global graph driver, which maintains a union mounted root file system tree for each container by caching all the image layers from the storage driver. The graph driver will randomly generate a cache ID for each image layer. The cache of image layers is built while the **docker pull** or **docker build** commands are executed. The Docker engine maintains the link between the content addressable layer ID and its *cache ID*, so that it knows where to locate the layer content on disk.

In order to get more details about Docker's content addressable images, we investigated the source code along with one of its most popular storage drivers, AUFS. Other storage drivers such as Btrfs, Device Mapper, OverlayFS, and ZFS, implement management of image layers and *container layers* in unique ways. Our framework could be extended to those drivers. Due to limited time and space, we focused on experiments with AUFS. The following section presents our findings about Docker's AUFS storage driver.

3.4.3 AUFS Storage: A Case Study

We conduct our experiments with Docker version 1.10 and the default AUFS storage driver. Therefore, our case study demonstrates management of multiple image layers from an AUFS point of view. For the latest Docker version (docker-20.10 as of this writing), it

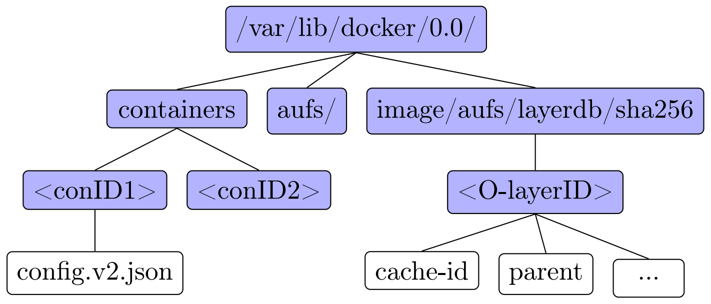


Figure 3.2: Docker Layered File System Structure

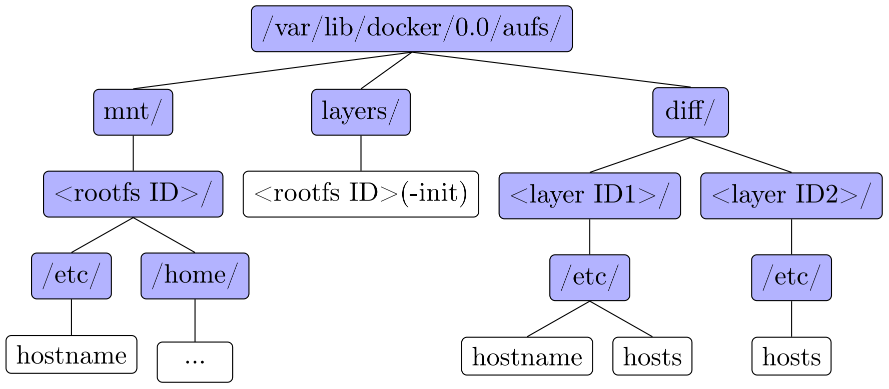


Figure 3.3: AUFS Work Directories in Docker's Layered Storage

is recommended to use *overlay2* when possible [62]. Note that the actual directory tree structure described in this section is no longer valid for *overlay2*. However, the general principles of image layer organization and access remain the same as introduced above. The scheme in this work provides a guideline to interact with the image layer addressing operations of the Docker runtime graph driver [67] which is not tightly bound to the underlying storage drivers. Therefore, it could be extended to *overlay2* with straightforward engineering efforts, consisting mostly of updating directory names.

AUFS storage driver exposes Docker container storage as a union mounted file system. Union mount is a way of combining different directories into one directory in such a way that it appears to contain the contents from all of them [121]. AUFS uses union mount to merge all image layers together and presents them as one single read-only view. If there are duplicate identities (i.e. file names) in different layers, only the one on the highest layer is accessible.

Figure 3.2 and 3.3 illustrate the Docker storage structure based on the AUFS driver. A blue box stands for a directory and a white box stands for a file. The root directory of Docker storage is by default defined as */var/lib/docker/0.0/*. We will use '.' to represent this common directory in the following discussion. The AUFS driver's work directory is located at *./aufs/*. AUFS mainly uses three directories to manage image layers:

1. *Layer directory* (./aufs/layers/): This contains the metadata that describes how storage layers are stacked together;

2. *Diff directory* (./aufs/diff/): This stores the content data for every layer, where each layer is put in one subdirectory;

3. *Mount directory* (./aufs/mnt/): This contains the mount point of the root file system for the container.

When the Docker daemon starts or restores a container, it will query the IDs of all image layers stored at the *Layer directory*. Then it will get the content of image layers by searching the *Diff directory*. Finally all image layers are union mounted together to the *Mount directory*. After this, the container will have a single view of its complete file system.

Note that the mount point for a container's root file system is only available when the container is running. If the container stops, all the image layers will be unmounted from this mount point. Then the mount point will become an empty directory. Therefore, during migration, we cannot synchronize the container's root file system directly, or the container's layers will not be mounted or unmounted correctly on the target node.

3.4.3.1 Container's Image Layer Stack List

We know that each Docker image contains several image layers. Those image layers are addressed by their layer IDs. Each Docker image has a list of layer IDs in the order of how they stacked from top to bottom. There are two files, *./aufs/layers/<rootfs ID>-init* and

./aufs/layers/<rootfs ID>, both of which store a list of layer IDs. The former stores IDs of all initial image layers when the container is created. The latter stores IDs of the newly created layers in addition to the initial layers. In Figure 3.3, we use *./aufs/layers/<rootfs ID>(-init)* as a simplified notion of the two files.

For example, for the container *OpenFace* with rootfs ID of *febfb1642ebeb25857bf2a9c55-8bf695*, it's initial layer stack is stored in the file *./aufs/layers/febfb1642ebeb25857bf2a9c-558bf695-init*. It contains all layers in the downloaded container image *bamos/openface*[2]. These layers will be read-only throughtout the whole life cycle of the container. Once a new layer is created, i.e. the *container layer*, the layer ID will be listed on the top line of the file *./aufs/layers/febfb1642ebeb25857bf2a9c558bf695*.

When the Docker daemon starts or restores a container, it will refer to those two files to get a list of all underlying Docker image layer IDs and the container layer ID. Then it will resolve those addressable IDs and union mount all those layers together in the specific order. After this, the container will get the full view of its root file system under the root mount point. We find that the two files behaves like an important handler for the union file system of the container. If any of the two file is missing or corrupted, the container will not be able to union mount the root file system correctly.

3.4.3.2 Image Layer Content Directory

AUFS manages the content of image layers in the directory of *./aufs/diff/*. The directory *./aufs/diff/<layer ID>/* stores all the files inside the specific layer identified by its *<layer ID>*. This can be either a readonly image layer or a *container layer* that is newly created. If *<layer ID>* is the same as *<rootfs ID>* of a container, then this directory is where the content of *container layer* stores, i.e. all the file system changes of the container will be stored in this directory.

[2]https://hub.docker.com/r/bamos/openface/.

3.4.3.3 Unified Mount Point

The directory *./aufs/mnt/<rootfs ID>/* is the mount point of the container's root file system. All file system layers are union mounted to this folder and provide a single file system view for the container. For example, as shown in Figure 3.3, when a container is created based on a Linux image, its mount point will contain the root directory contents like */usr/, /home/, /boot/*, etc. All those directories are mounted from its underlying storage layers, including both the read only image layers downloaded from the registry and the newly created container layer. Since this directory is a mount point for a running container's file system, it will be only available when the container is running. If the container stops running, all the image layers will be unmounted from this mount point. So it will become an empty directory.

3.4.3.4 Layer ID Mapping

Until now, the layer IDs we have discussed above are just local SHA256 IDs, or what we call cache IDs, which are generated dynamically when each image layer is downloaded by **docker pull** command. From then on, Docker daemon will address the image layer using the cache ID instead of its original layer ID (noted as *O-layerID* in this work).

We find the Docker storage system maintains a mapping relationship between the original layer IDs and its cache IDs. As shown in Figure 3.2, all the cached IDs of image layers are stored in the directory *./image/aufs/layerdb/sha256/*. For example, the file *./image/aufs/layerdb/sha256/<O-layerID>/cache-id* stores the cache ID of the image with original ID *<O-layerID>*. If a hash ID *fac86d61dfe33f821e8d0e7660473381* is stored in the file of *./image/aufs/layerdb/sha256/6384c447ddd6cd859f9be3b53f8b015c/cache-id*, this means there is an image layer with an original ID of *6384c447ddd6cd859f9be3b53f8b015c* and its cache ID is *fac86d61dfe33f821e8d0e7660473381*.

3.4.3.5 Container Configuration and Runtime State

There are several directories that store the configuration files and runtime data. As shown in Figure 3.3, the directory */var/lib/docker/0.0/containers/<conID>* contains the configuration files for each container. For example, from the JavaScript object notation (JSON) file *config.v2.json*, we can find the container's creation time, the command that was run when the container was created, etc.

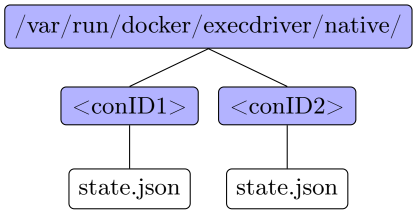


Figure 3.4: Runtime Data for Containers

Figure 3.4 shows the runtime data directory for each container. For one container with ID of *<conID>*, there will be a JSON file *state.json* that stores the runtime state of the container. For example, the initial process ID of the container is identified by key "*init_process_pid*", and the root file system mount point path can be found via key "*rootfs*". There are also some runtime cgroup and namespace meta data, etc.

3.4.4 Docker Container Migration in Practice

There is no official migration tool for Docker containers as of this writing, yet many enthusiastic developers have constructed tools for specific versions of Docker. These tools have demonstrated the feasibility of Docker container migration. For example, *P.Haul* [65] supports migration of Docker-1.9.0-dev, and Boucher [26] extends *P.Haul* to support Docker 1.10-dev migration. However, both methods simply transfer all the files located under the mount point of a container's root file system. At that point, the files in the root directory are actually a composition of all container image layers. Both methods ignore the underlying storage layers. This will cause the following problems:

1. It will corrupt the layered file system inside the container after restoration on the

target server. The tool simply transfers the whole file system into one directory on the destination, ignoring all underlying layer information. After restoration on the target host, the container cannot be properly maintained by the Docker daemon, which will fail when it tries to mount, or unmount the underlying image layers.

2. It substantially reduces the efficiency and robustness of migration. The tool synchronizes the whole file system using the Linux **rsync** command while the container is still running. First, running **rsync** command on a whole file system is slow due to the large amount of files, especially during the first run. Second, file contention is possible when process of container and the process of **rsync** attempt to access the same file and one of them is write. Contention causes synchronization errors which result in migration errors.

To verify our claim, we have conducted experiments to migrate containers over different network connections. Our experiments use one simple container, Busybox, and one complex application, OpenFace, to conduct edge server offloading. Busybox is a stripped-down set of Unix tools in a single executable file. It has a tiny file system inside the container. OpenFace [14] is an application that dispatches images from mobile devices to the edge server, which executes the face recognition task, and sends back a text string with the name of the person. The OpenFace container has a huge file system, approximately 2 Gigabytes.

Table 3.1 indicates that migration could be done within 10 seconds for Busybox, and within 30 seconds for OpenFace. The network between these two virtual hosts has a 1 Gbps bandwidth and latency of 0.4 milliseconds, transferring 2.17 GB data within a short time. We further test container migration over a network with bandwidth of 15 Mbps and latency of 5.4ms. Table 3.2 shows that migration of the Busybox container takes 133.11 seconds with transfer sizes as small as 290 Kilobytes. Migrating OpenFace requires to transfer more than 2 Gigabytes data and takes about 3200 seconds.

As previously stated, poor performance is caused by transferring large files comprising the complete file system. This performance is worse than the state-of-the-art VM migration solution we mentioned above. Migration of VMs could avoid transferring a large portion of the file system by sharing the base VM images [80], which will finish migration within several minutes.

Therefore, we require a new tool to efficiently migrate Docker containers, avoiding unnecessary transmission of common image layers. This new tool should leverage the layered file system to transfer the *container layer* only during service handoff.

3.5 Offloading Service Migration on the Edge

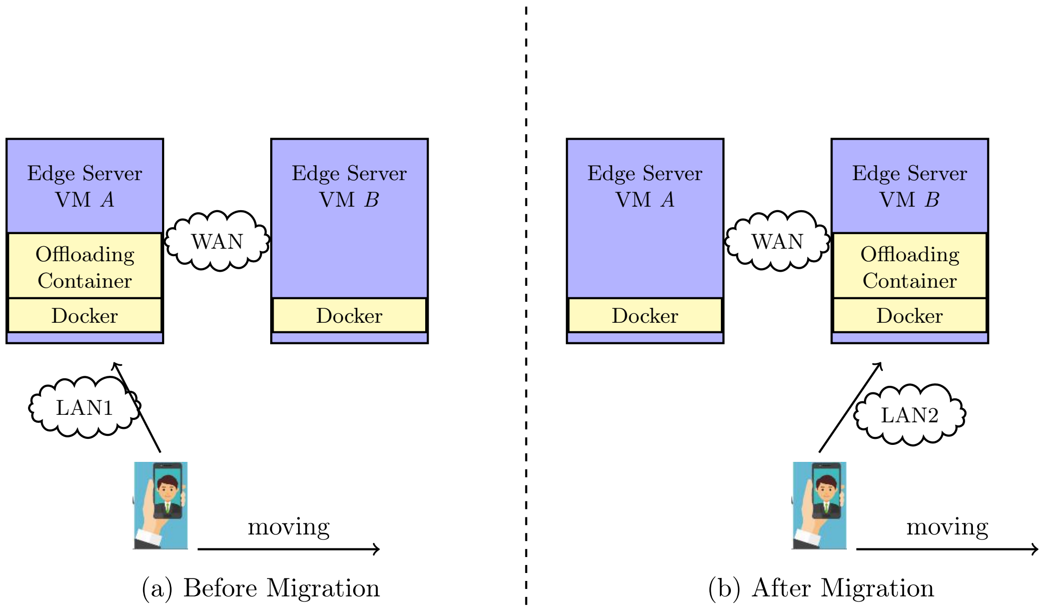


Figure 3.5: Offloading Serivce Handoff: Before and After Migration of Offloading Container.

In this section, we introduce the design of our service hand-off framework based on Docker container migration. First, we provide a simple usage scenario, then we present an overview of the system architecture in Section 3.5.1. Second, we enumerate workflow steps performed during service handoff in Section 3.5.2. Third, in Section 3.5.3 and 3.5.4, we discuss our methodology for storage synchronization based on Docker image

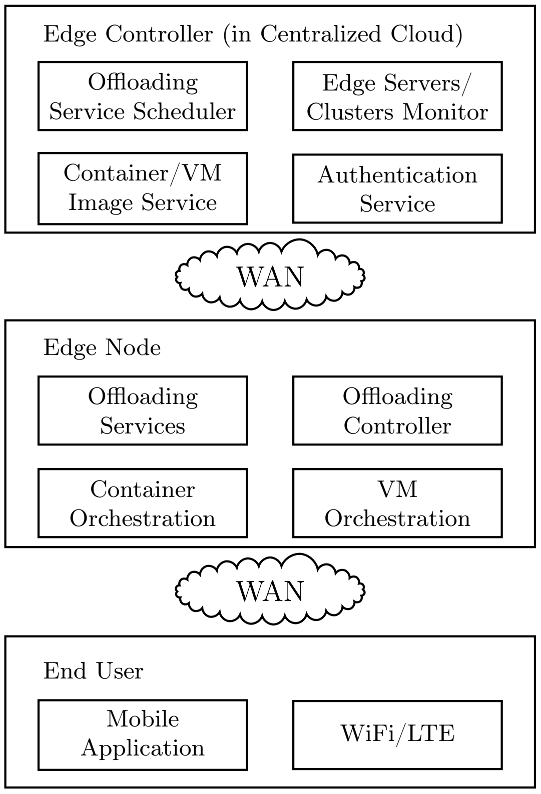


Figure 3.6: Overview of Edge Computing Platform

layer sharing between two edge servers. Finally, in Section 3.5.5, 3.5.6, and 3.5.7, we show how to further speed up the migration process through memory difference transfers, file compression, pipelined and parallel processing during Docker container migration.

3.5.1 System Overview

Figure 3.5 shows an exemplar usage scenario of offloading service hand-off based on container migration. In this example, the end user offloads workloads to an edge server to achieve real-time face recognition (OpenFace [14]). The mobile client continuously reads images from the camera and sends them to the edge server. The edge server runs the facial recognition application in a container, processes the images with a deep neural network algorithm, and finally sends each recognition result back to the client.

All containers are running inside VMs (see VM A, VM B in Figure 3.5). The combination of containers and VMs controls the isolation between applications at different levels and enables applications to scale up deployment more easily.

All offloaded computations are executed inside containers, which we call the *offloading container*. When the user moves beyond the reach of server **A** and reaches the service area of edge server **B**, its offloading computation shall be migrated from server **A** to server **B**. This is done via migration of the offloading container, where all runtime memory states as well as associated storage data should be synchronized to the target server **B**.

In order to support both the mobility of end users and the mobility of its corresponding offloading services on the edge server, we have designed a specialized edge computing platform. Figure 3.6 provides an overview of our edge platform and its three-level computing architecture. The first level is the traditional cloud platform architecture. The second level consists of the edge nodes distributed over a WAN network in close proximity to end users. The third level consists of mobile clients from end users who request offloading services from nearby edge servers.

3.5.1.1 Edge Controller

The first level contains four services running in the centralized edge controller that manages offloading services across all edge servers (or edge server clusters) on the WAN network. These four services are:

Offloading Service Scheduler: This is responsible for scheduling offloading services across edge servers. The parameters of scheduling include but are not limited to 1) physical locations of end users and edge servers; 2) workloads of edge servers; 3) end user perceived bandwidth and latency, etc.

Edge Server/Clusters Monitor: This is responsible for communicating with the distributed edge servers or clusters, and collecting performance data, runtime metadata for offloading services, and end user metadata. The collected data is used to make scheduling decisions.

Container/VM Image Service: This is the storage service for edge servers. It distributes container and VM images to the edge server for fast deployment as well as for data backup. Backup data can be saved as container volumes [61] to enable faster deployment and sharing among distributed edge servers.

Authentication Service: This is used to authenticate the identities of both edge servers and end users.

3.5.1.2 Edge Nodes

The second level in Figure 3.6 consists of the distributed edge nodes. An edge node could be a single edge server or a cluster of edge servers. Each edge node runs four services which are:

Container Orchestration Service and *VM Orchestration Service*: The two are virtualization resource orchestration services. They are used to spawn and manage the life cycle of containers and VMs. Each end user could be assigned one or more VMs to build an isolated computing environment. Then by spawning containers inside the VM, the end user creates offloading services.

Offloading Services: This is a set of container instances that execute the end user's offloading workloads.

Offloading Controller: This is responsible for managing the services inside the edge node. It could limit the number of user-spawned containers, balance workloads inside the cluster, etc. It also uploads the latest performance data to the *Edge Controller* in the cloud. Performance data includes offloading service states inside the edge node, and identification of the latest data volumes that require backup on the cloud.

3.5.1.3 End Users

The third level of our edge platform is comprised of the end user population. End users are traditional mobile clients running applications on Android, iOS, Windows, or Linux mobile devices, or embedded IoT devices. Our design will not modify the mobile client applications. The progress of offloading service handoff will be transparent to end users.

The mobile device can use *WiFi* or *LTE* to access the *Edge Nodes* or *Edge Controller*.

3.5.2 Workflow of Service Handoff

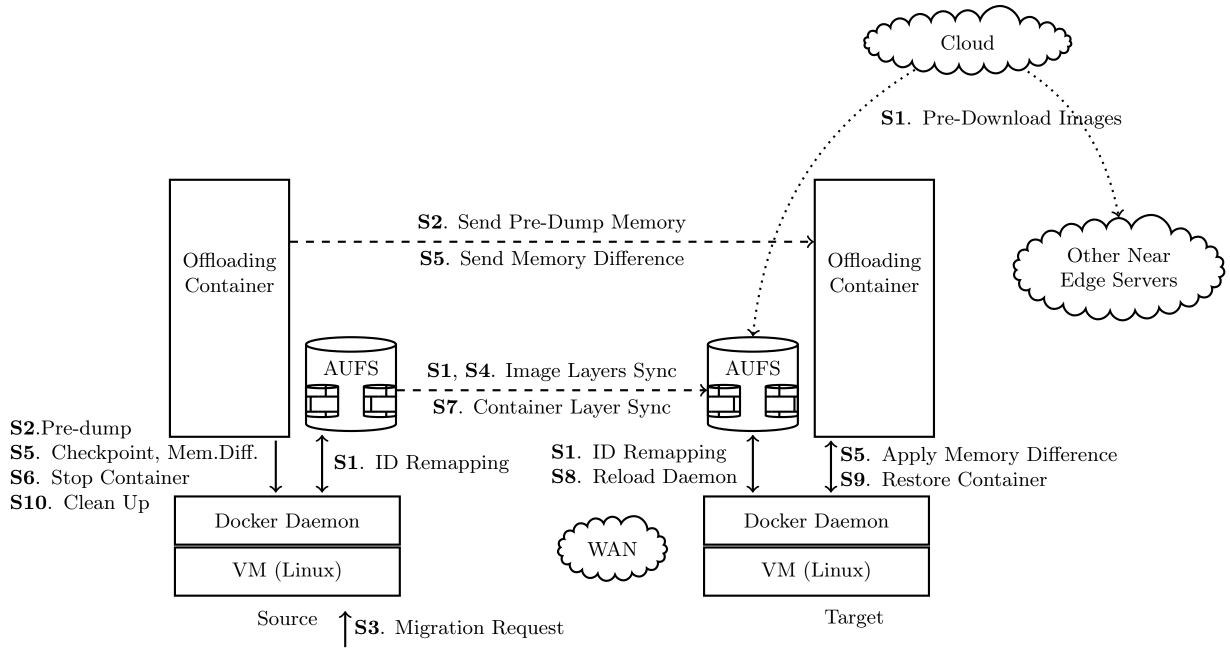

Cloud
S1. Pre-Download Images
Other Near Edge Servers
Offloading Container
S2. Send Pre-Dump Memory
S5. Send Memory Difference
Offloading Container
AUFS
S1, **S4**. Image Layers Sync
S7. Container Layer Sync
AUFS
S2.Pre-dump
S5. Checkpoint, Mem.Diff.
S6. Stop Container
S10. Clean Up
S1. ID Remapping
S1. ID Remapping
S8. Reload Daemon
S5. Apply Memory Difference
S9. Restore Container
Docker Daemon
VM (Linux)
WAN
Docker Daemon
VM (Linux)
Source
S3. Migration Request
Target

Figure 3.7: Full Workflow of Offloading Service Handoff

Figure 3.7 shows the design details of our architecture broken into individual migration steps. The source server is the edge server currently providing end user computational services. The target server is the gaining server. Computational services are transferring from the source to the target server. Details of these steps are described below:

S1 Synchronize Base Image Layers. Offloading services are started by creating a container on the source server. Once the container is started on the source server, the base image layers for that container will also be downloaded to additional nearby potential target servers. This is to begin preparation for subsequent end user movements.

S2 Pre-dump Container. Before the migration request is issued, one or more memory snapshots will be synchronized to the all potential target servers without interrupting the offloading service.

S3 Migration Request Received on Source Server. Live migration of the offloading is triggered by the migration request. The request is initiated by the cloud control center.

S4 Image Layer Synchronization. Images layers on the two edge servers are compared with each other by remapping the *cacheID*s back to the original IDs. Only the different image layers are transferred.

S5 Memory Difference Transmission. The container on the source server will be checkpointed to get a snapshot of memory. Multiple snapshots can be taken in different time slots. Two consecutive snapshots will be compared to get dirty memory. The dirty memory is then transmitted to the target server and re-assembled at the target server.

S6 Stop Container. Once the dirty memory and file system difference are small enough, such that they can be transferred in a tolerable amount of time, the container on the source server will be stopped and the latest dirty memory and files will be sent to the target edge server.

S7 Container Layer Synchronization. After the container is stopped, storage on the source server will not be changed by the container. Thus we can send the latest *container layer* to the target server. At the same time, all metadata files, such as JSON files logging the container's runtime states and configuration files, are also transferred to the target server.

S8 Docker Daemon Reload. On the target server, Docker daemon will be reloaded after receiving container configuration files from the source server. After reloading, the target node will have source configurations loaded into the runtime database.

S9 Restore Container. After the target server receives the latest runtime memory and files, the target container can be restored with the most recent runtime states. The migration is now finished at the target server and the user begins receiving services from this new edge server. At the same time, the target server will go to step **S1** to prepare the next iteration of service migration in the future.

S10 Clean Up Source Node. Finally, the source node will clean up by removing the footprints of the offloading container. Clean up time should be carefully chosen based on user movement patterns. It could be more efficient to retain and update the footprint containers if the user moves back in the future.

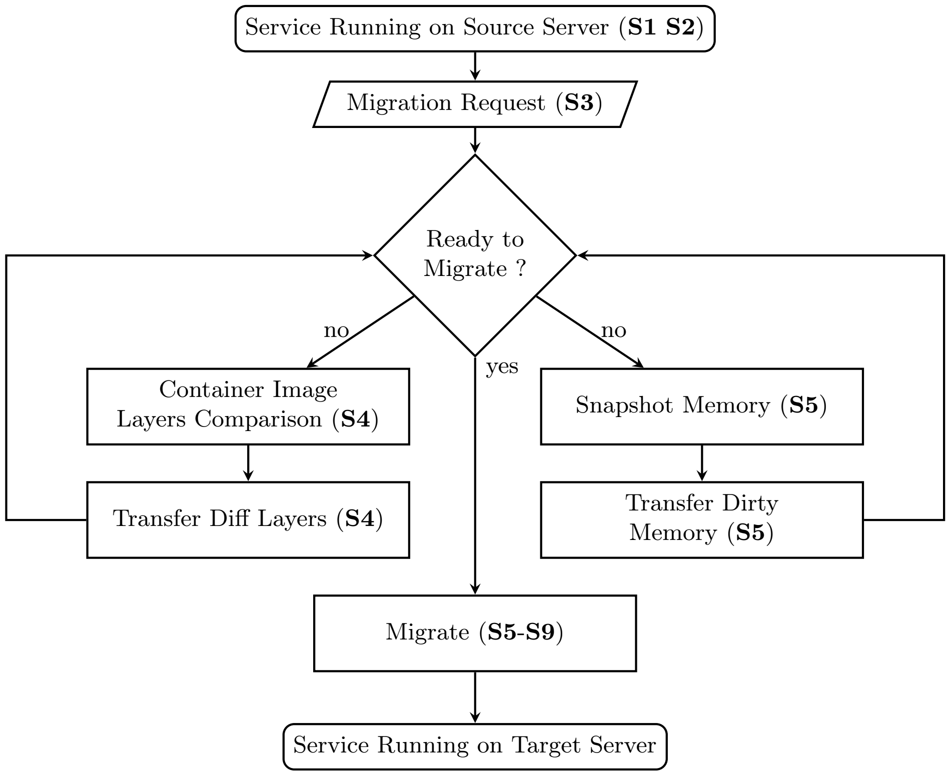

Service Running on Source Server (S1 S2)
Migration Request (S3)
Ready to Migrate ?
no
no
yes
Container Image Layers Comparison (S4)
Transfer Diff Layers (S4)
Snapshot Memory (S5)
Transfer Dirty Memory (S5)
Migrate (S5-S9)
Service Running on Target Server

Figure 3.8: Major Procedures of Migration

Figure 3.8 provides a simple overview of the major migration procedures. We assume that before migration starts, both the source and target edge servers have the application base images downloaded. Once the migration request is received on the source server, multiple iterations of transferring image layers and memory images/differences will be proceeded until the migration is done. File system images and memory snapshots are transferred in parallel to improve efficiency. The number of iterations needed can be determined empirically based on the actual offloading environment and the application's tolerance for service delay.

3.5.3 Strategy to Synchronize Storage Layers

Storage layer matching can either be implemented within the existing architecture of the container runtime, or provided as a third party tool without change to the underlying container architecture. Changing the container architecture will enable the built-in migration capabilities thus improve the efficiency and usability. However, users must update their container engine in order to benefit from the modified migration feature. Updating the software stack can be destructive in a complex environment, where the release of modified software packages usually takes a long time due to extensive testing requirements. A third party migration tool offers the advantage of faster migration feature implementation since no changes are made to the existing container engine. This is also a good option for a test environment.

In this section, we implement our migration feature as a third party tool. Of course, after the migration feature is well established, it can be embedded into the container architecture by changing the respective part of the container. One example is the graph driver of Docker [67]. One solution is to patch the graph driver by simply replacing the randomly generated cache ID with the actual content addressable hash ID of the image layer, or generate a different hash ID by hashing the same image layer content from a different hash algorithm. We leave such tool extensions to future work.

A running container's layered storage is composed of one writable *container layer* and several read only *base image layers.* The *container layer* stores all the files created or modified by the newly created container. As long as the container is running, this layer is subject to change. So we postpone the synchronization of the *container layer* to the point after the source container is stopped (in step **S7**).

All base image layers inside containers are read only, and are synchronized as early as possible. There are two kinds of *base image layers*. The first, and most common type are *base image layers* downloaded by **docker pull** commands from centralized image registries such as Docker Hub. The second type of image layer is created by the local Docker host by saving the current *container layer* as one read-only image layer via **docker commit** command.

Image layers from the centralized image registry should be downloaded before migration starts, thus download time is amortized (in step **S1**). This also reduces network traffic between the source and target edge servers. For locally created base image layers, we transfer each such image layer as it is created (in step **S4**), regardless if the migration has started or not.

3.5.4 Layer ID Remapping

As mentioned previously, an image downloaded from the common registry to multiple edge servers will have different cache IDs exposed at each edge server's Docker runtime. In order to efficiently share these common images across different edge servers, image layers need to be matched based upon the original IDs instead of the *cache ID*s. To remap image cache IDs without changing the Docker graph driver, we design a third party tool to match the randomly generated *cache IDs* to original layer IDs. We first remap the *cache IDs* to original IDs on two different edge servers. Then the original IDs are compared via communication between the two edge servers. The image layers are the same if they have identical original IDs.

After the common image layers are found, we map the original IDs back to the local cache IDs on the target server. Then we update the migrated container with the new cache IDs on the target server. Thus, the common image layers on the migrated container will be reset with the new cache IDs that are addressable to the Docker daemon on the target server. When we restore the container in the future, the file system will be mounted correctly from the shared image layers on the target server.

For the original IDs that don't match between the two hosts, we treat them as new image layers, and add them to a waiting list for transfer in step **S7**.

3.5.5 Pre-Dump & Dirty Memory Synchronization

In order to reduce transferred memory image size during hand-off, we first checkpoint the source container and then dump a snapshot of container memory in step **S2**. This could happen as soon as the container is created, or we could dump memory when the most frequently used binary programs of the application are loaded into memory. This snapshot of memory will serve as the base memory image for the migration.

After the base memory image is dumped, it is transferred immediately to the target server. We assume that the transfer will be finished before hand-off starts. This is reasonable since we can send the base memory image as soon as the container starts. After the container starts, and before the hand-off begins, the nearby edge servers start to download the application's container images. We process those two steps in parallel to reduce total transfer time. This is further discussed in section 3.5.7. Upon hand-off start, we have the base memory image of the container already loaded on the target server.

3.5.6 Data Transfer

There are four types of data that require transfer: layer stack information, the thin writable *container layer*, container metadata files, and snapshots of container memory and memory differences. Some of the data is in the form of string messages, such as layer stack information. Some data are in plain text files, such as most file contents and configuration files. Memory snapshots, and memory differences are stored as binary files. Adapting to the file types, we design different data transfer strategies.

Layer stack information consists of a list of SHA256 ID strings. This is sent as a socket message via UNIX RPC API implementation in [65]. Note that data compression is not efficient for this information because the overhead of compression outweighs the transmission efficiency benefits for those short strings.

For other data types, including the container writable layer, meta data files, dump memory images, and image differences, we use bzip2 for compression before sending out via authorized ssh connection.

3.5.7 Parallel & Pipelined Processing

With the help of parallel and pipelined processing, we could further reduce the total migration time.

First, starting a container will trigger two events to run in parallel: a) on the edge servers near the end user, downloading images from centralized registry, and b) on the source node, pre-dumping & sending base memory images to the potential target servers. Those two processes could be run at the same time in order to reduce the total time of step **S1** and **S2**.

Second, daemon reload in step **S8** is required on the target host. It could be triggered immediately after **S7** and be paralleled with step **S5**, when the source server is sending the memory difference to the target host. Step **S7** cannot be paralleled with **S8**, because daemon reload on the target host requires the configuration data files sent in step **S7**.

Third, in step **S7**, we use compression to send all files in the *container layer* over an authorized SSH connection between the source and target host. The compression and transfer of the *container layer* can be pipelined using Linux pipes.

Lastly, in step **S5**, we need to obtain memory differences by comparing the base memory images with the images in the new snapshot, then we send the differences to the target and patch the differences to the base memory image on the target host. This whole process could also be piplined using Linux pipes.

3.5.8 Multi-Mode Migration with Flexible Trade-offs

Service handoff efficiency is affected by many system environment factors. They include: 1) the network conditions between two edge servers; 2) the network conditions between end user and edge server; 3) the available resources on the edge servers, such as available CPU power. Taking these factors into consideration, we use different strategies to improve the efficiency of service handoff. We combine different metrics to dynamically adapt to various system environments.

The metrics we use to determine our strategies include:

1) *Realtime Bandwidth and Latency:* This includes the real time bandwidth and latency between the source and target edge servers, as well as between the end user and two edge servers.

2) *Compression Options:* We have a set of compression algorithms and options available for use. Different algorithms with different options require different CPU power and take differing amounts of computation time.

3) *Number of Iterations:* This defines the maximum number of iterations invoked for memory pre-dumping and storage checkpointing before handoff starts.

The end user's high quality of service is the ultimate optimization goal. Instead of providing a concrete goal for optimization under different environments and requirements, we provide multiple possible settings to enable users or developers to customize their own strategies performing tradeoffs between differing environmental factors and user requirements. The optimization goals we define for service handoff are:

1) *Interruption Time*: This is the time from user disconnection from their service on the source server to the time when the user is reconnected to their service on the target server.

2) *Service Downtime*: This is the time duration of the last iteration of the container migration. During this time interval, the service instance on the source node is stopped.

3) *Total Migration Time*: This is used to represent the total time of all iterations of container migration.

Number of Iterations needs to be carefully determined to optimize the quality of services for end users. If bandwidth is low, the time each iteration takes will be longer. So our system tends to use fewer iterations to checkpoint storage and memory. Fewer iterations mean each batch of dirty storage and memory transfers will be in large volume. Therefore, during the last iteration for service handoff, it will migrate the container in a relatively longer time, while the total handoff time at the last iteration might be less.

If bandwidth is high, more iterations could be done in a relatively short time. Then our system tends to use more iterations to send storage and memory differences. Generally the first iteration takes the longest time, say T_1. The second iteration will take a shorter time, because it only transfers the dirty memory generated since T_1, say it takes T_2, thus $T_2 < T_1$. Then the third iteration will usually cost less time, because the dirty memory generated since T_2, is smaller than the dirty memory generated since T_1. Therefore, each iteration will usually take less and less time. The last iteration's time can be minimized by increasing the total iteration number. This is how the live migration is done inside traditional data centers.

However, for live migration in an edge network, we need to consider user mobility. If we set too many iterations, it will add up to the total migration time. During this time, if the user is moving far away from its original edge server, the quality of service will also degrade despite the minimization of service downtime. Therefore we need to control the total iterations performed commensurate with user mobility and network bandwidth. Similarly, *Compression options* also need to be carefully choosen in order to optimize the service handoff process.

3.5.9 Two-layer System-wide Isolation for Better Security

It is critical to minimize security risks posed to offloading services running on the edge servers. Isolation between different services could provide a certain level of security. Our framework provides an isolated running environment for the offloading service via two layers of the system virtualization hierarchy. Different services can be isolated by running inside different *Linux containers*, and different containers are allowed to be further isolated by running in different *virtual machines*.

More thorough security solutions need to be designed before this framework can be deployed in a real world environment. These solutions include, but are not limited to efficient runtime monitoring, secure system updating, etc. We leave security enhancements for future work and focus on performance evaluation of our services.

3.5.10 Discussion

In this section, we discuss the benefits of overall system and its extended applications, and then clarify the limitations of the scope of this work.

3.5.10.1 Benefits and Applications

In this work, we propose an efficient service migration scheme based on sharing layers of the container storage, and explore several key metrics that can be used to tune migration performance. Metrics on the edge server, such as bandwidth, latency, host environment, etc., are provided to the cloud center to support decisions towards optimal performance. Cloud centers could utilize those metrics to make migration go/no-go decisions, schedule the timing of migrations, decide which target servers to choose as migration destinations in order to minimize service interruptions.

3.5.10.2 Limitations of Scope

Note that the theoretical proof of our performance optimization scheme is out of scope of this chapter. In the architecture of our edge platform, we separate the optimization problem into two tasks, one for distributed edge, and one for centralized cloud. The first one is to collect performance data from edge servers; The second one is to evaluate the performance and make optimization decisions at the cloud center. This work focuses on the edge nodes, where metrics of performance are collected. Therefore, the decision process of the cloud center is out of the scope of this work.

3.6 Evaluation

In this section, we introduce our evaluation experiments and report the results from the following investigations: 1) How can container migration performance be affected by pipeline processing? 2) How can customized metrics such as network bandwidth, latency, file compression options, and total iteration numbers, affect the migration performance? 3) Will our system perform better than state-of-the-art solutions?

3.6.1 Set Up and Benchmark Workloads

Migration scenarios are set up by using two VMs, each running a Docker instance. Docker containers are migrated from the Docker host on the source VM to the Docker host on the target VM.

In order to test our system running across WANs , we emulated low network bandwidths ranging from 5Mbps to 45Mbps. Consistent with the average latency observed on the Internet [132], we set the fixed latency of 50ms to emulate the WAN environment for edge computing. Since edge computing environments can also be adapted to LAN networks, we also tested several higher bandwidths, ranging from 50Mpbs to 500Mpbs. Latency during these tests was set to 6ms, the average observed latency on the author's university LAN. Linux Traffic Control (**tc** [28]) was used to control network traffic.

For the offloading workloads, we chose Busybox as a simple workload to show the functionality of the system, and demonstrate non-avoidable system overhead when performing container migration. In order to show offloading service handoff comparable to real world applications, we chose OpenFace as a sample workload.

3.6.2 Evaluation of Pipeline Performance

In order to demonstrate the effectiveness of pipelined processing, we tested pipeline processing with two time consuming steps: *imgDiff* and *imgSend*, where *imgDiff* receives memory difference files, and *imgSend* sends memory difference files to the target server

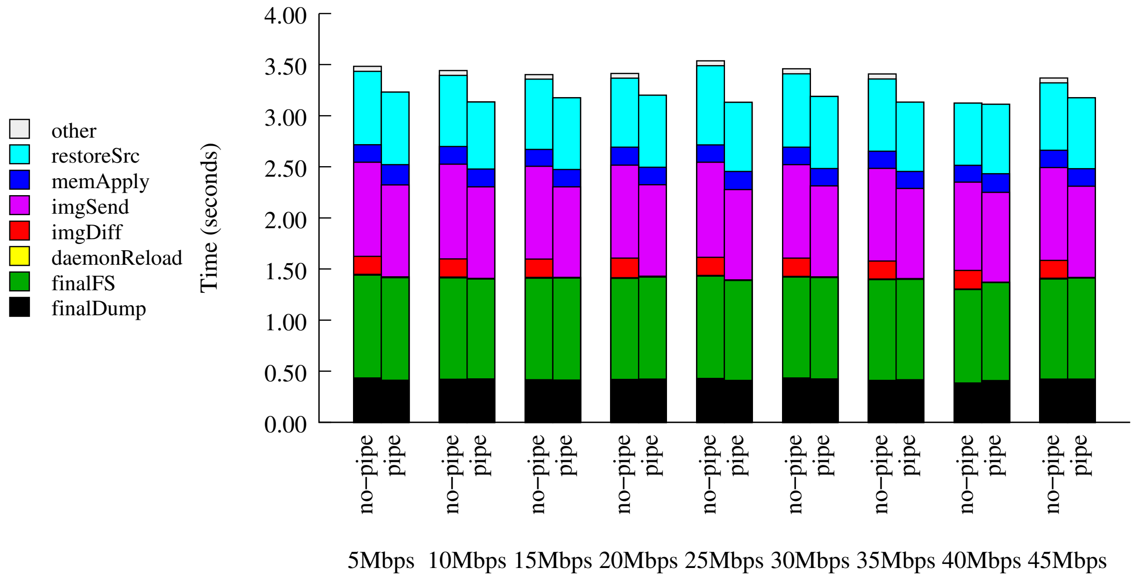


Figure 3.9: Busybox: Time Duration of Container Migration Stages with and without Pipelined Processing

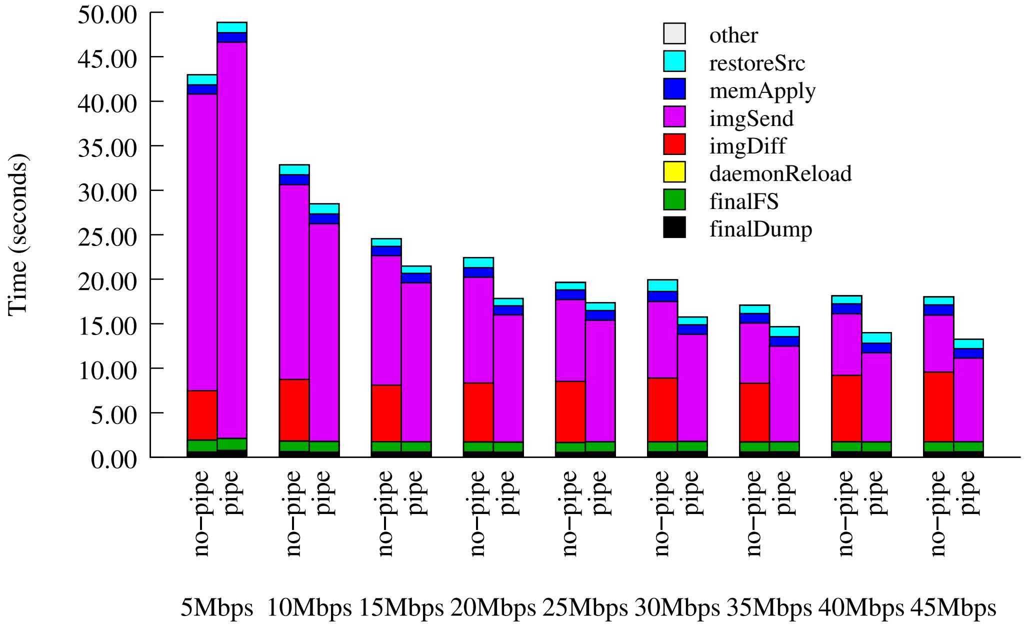


Figure 3.10: OpenFace: Time Duration of Container Migration Stages with and without Pipelined Processing

during migration. Figure 3.9 and Figure 3.10 report the timing benefits we achieved by incorporating pipelined processing. From the figure, we can see that, without pipelined processing, most time costs are incurred by receiving and sending the memory difference files. After applying pipelined processing, we save $5 \sim 8$ seconds during OpenFace migration. Busybox also saves a certain amount of time with pipelined processing.

3.6.3 Evaluation on Different Metrics

In this section, we will evaluate the service handoff time observed under different configurations of our pre-defined four metrics: 1) network bandwidth; 2) network latency; 3) compression options; 4) number of iterations. In order to evaluate the implication of different configurations, we design contrast experiments for each metric. For example, to evaluate network bandwidth effects, we keep other metrics constant in each experiment.

3.6.3.1 Evaluation of Changing Network Bandwidth

Table 3.3 and Figure 3.10 show an overview of the performance of our system under different network bandwidth conditions. Latency is set to 50ms, total number of iterations is set to 2, and the compression option is set to level 6.

In Table 3.3, *Handoff time* is from the time the source server receives a migration request until the offloading container is successfully restored on the target server. *Down time* is from the time when the container is stopped on the source server to the time when the container is restored on the target server. *Pre-Transfer Size* is the transferred size before *handoff* starts, i.e., from step **S1** until the beginning of step **S3**. *Final-Transfer Size* is the transferred size during handoff, i.e., from step **S3** until the end of final step **S9**. Average of 10 runs and relative standard deviations (RSDs, in parentheses) are reported.

From Table 3.3 and Figure 3.10 we can conclude that in general the higher bandwidth we have, the faster the handoff process. However, when the bandwidth improves to a relatively high value, the benefit of bandwidth expansion diminishes gradually. For example, when the bandwidth increases from 5 Mbps to 10 Mbps, handoff time reduces from 50 seconds to less than 30 seconds, which yields more than 40% improvement. However, when the bandwidth exceeds 50 Mbps, it becomes harder to reach higher throughput by

	Band-width (Mbps)	Handoff Time(s)	Down Time(s)	Pre-Transfer Size (MB)	Final-Transfer Size (MB)
Busybox	5	3.2 (7.3%)	2.8 (7.9%)	0.01 (0.2%)	0.03 (0.3%)
	10	3.1 (1.8%)	2.7 (1.6%)	0.01 (0.2%)	0.03 (0.6%)
	15	3.2 (1.4%)	2.8 (1.6%)	0.01 (0.5%)	0.03 (0.9%)
	20	3.2 (1.6%)	2.8 (1.8%)	0.01 (0.3%)	0.03 (0.4%)
	25	3.1 (1.6%)	2.7 (1.8%)	0.01 (0.2%)	0.03 (0.9%)
	30	3.2 (1.4%)	2.8 (1.2%)	0.01 (0.3%)	0.03 (0.5%)
	35	3.1 (3.5%)	2.7 (3.3%)	0.01 (0.3%)	0.03 (0.6%)
	40	3.1 (3.4%)	2.7 (3.5%)	0.01 (0.2%)	0.03 (0.5%)
	45	3.2 (1.9%)	2.7 (1.8%)	0.01 (0.2%)	0.03 (0.8%)
	50	3.2 (1.7%)	2.7 (1.6%)	0.01 (0.2%)	0.03 (2.7%)
	100	3.2 (1.6%)	2.7 (1.4%)	0.01 (0.3%)	0.03 (0.4%)
	200	3.1 (1.8%)	2.7 (1.8%)	0.01 (0.1%)	0.03 (0.5%)
	500	3.2 (2.0%)	2.8 (2.2%)	0.01 (0.2%)	0.03 (0.4%)
OpenFace	5	48.9 (12.6%)	48.1 (12.7%)	115.2 (6.1%)	22.6 (13.0%)
	10	28.5 (6.9%)	27.9 (7.0%)	119.4 (3.5%)	22.2 (10.9%)
	15	21.5 (9.1%)	20.9 (9.4%)	116.0 (7.3%)	22.1 (11.1%)
	20	17.8 (8.6%)	17.3 (8.9%)	116.0 (6.9%)	21.2 (12.0%)
	25	17.4 (11.5%)	16.8 (12.0%)	114.3 (7.6%)	23.7 (14.8%)
	30	15.8 (7.5%)	15.1 (7.4%)	119.3 (2.5%)	22.7 (9.3%)
	35	14.7 (13.6%)	14.0 (14.3%)	116.8 (5.9%)	22.2 (15.6%)
	40	14.0 (7.3%)	13.4 (7.6%)	112.5 (8.1%)	23.0 (8.8%)
	45	13.3 (8.6%)	12.6 (9.1%)	111.9 (9.1%)	22.6 (11.7%)
	50	13.4 (10.7%)	12.8 (11.1%)	115.2 (5.3%)	23.2 (5.3%)
	100	10.7 (9.6%)	10.1 (10.1%)	117.2 (2.4%)	21.6 (10.8%)
	200	10.2 (12.9%)	9.6 (13.5%)	116.8 (2.4%)	20.6 (17.6%)
	500	10.9 (5.6%)	10.3 (5.9%)	117.4 (1.5%)	23.0 (3.9%)

Table 3.3: Overall System Performance

simply increasing the bandwidth. This effect can be caused by limited hardware resources, such as CPU power or heavy disk workloads. When the transfer data rate of the network becomes high, the CPU power used for compression and the machine disk storage become the bottlenecks.

Note that migration time of Busybox seems to be unrelated to the bandwidths in Table 3.3. This is due to the very small transferred file size, therefore transmission is finished very quickly regardless of network bandwidth.

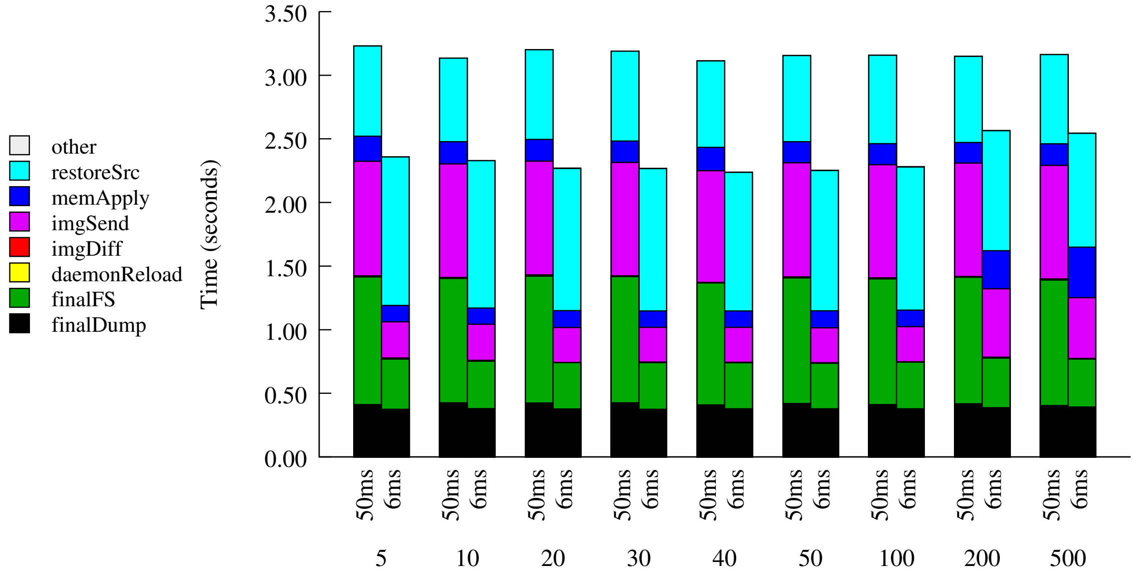


Figure 3.11: Busybox: Comparison of Migration Time.

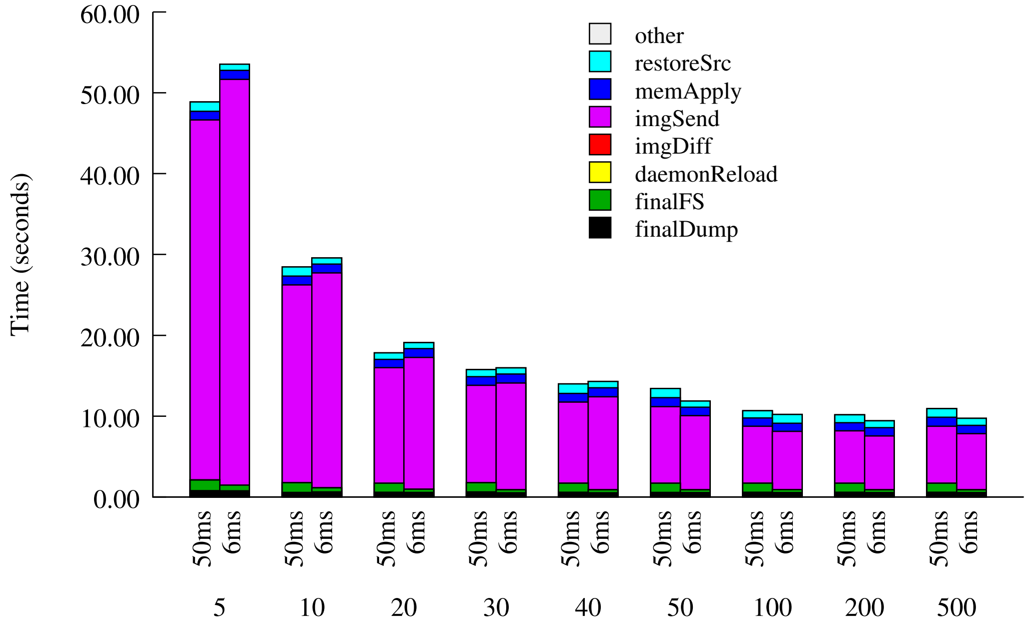
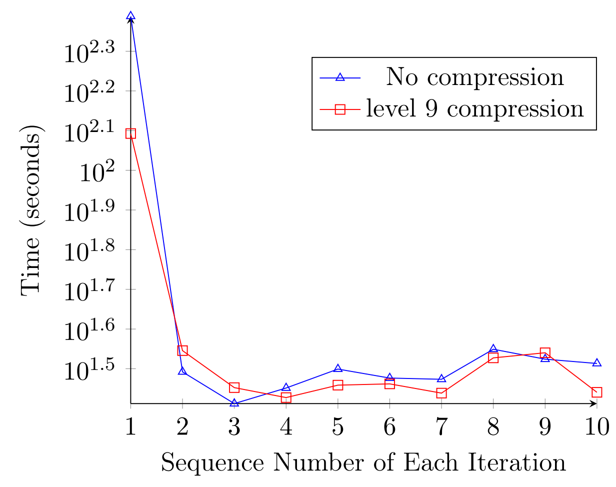


Figure 3.12: OpenFace: Comparison of Migration Time.

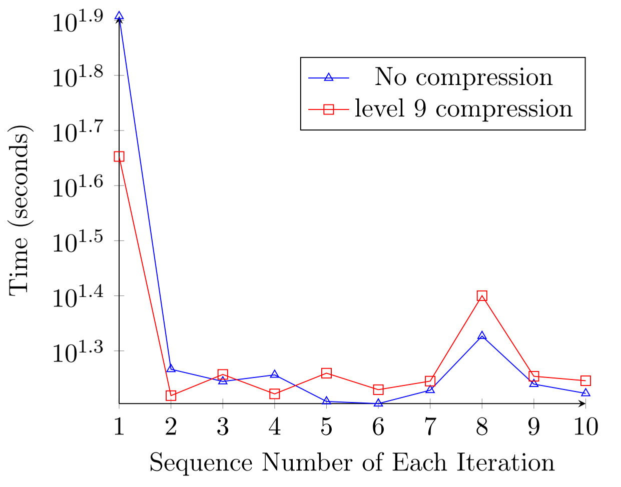
3.6.3.2 Evaluation of Changing Latency

Figure 3.11 and Figure 3.12 illustrate migration performance for BusyBox and OpenFace, respectively, under bandwidth from 5 Mbps to 500 Mbps, and latency of 50ms and 6ms, with 2 total iterations and level 6 compression. It shows a tiny difference when experiencing different latencies. This implies our system is suitable for a wide range of network latencies.

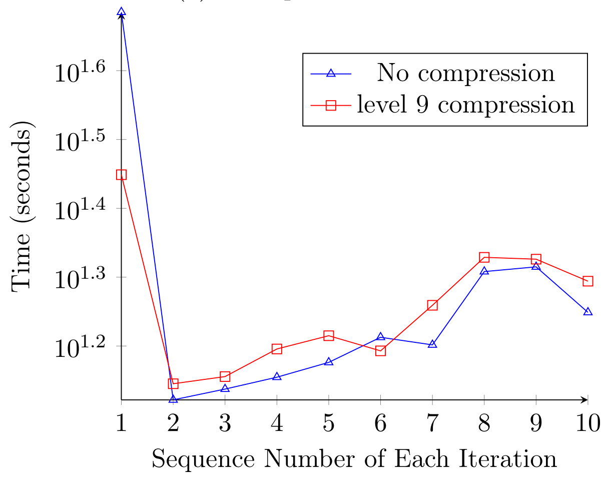
3.6.3.3 Evaluation of Changing Compression Algorithms and Options



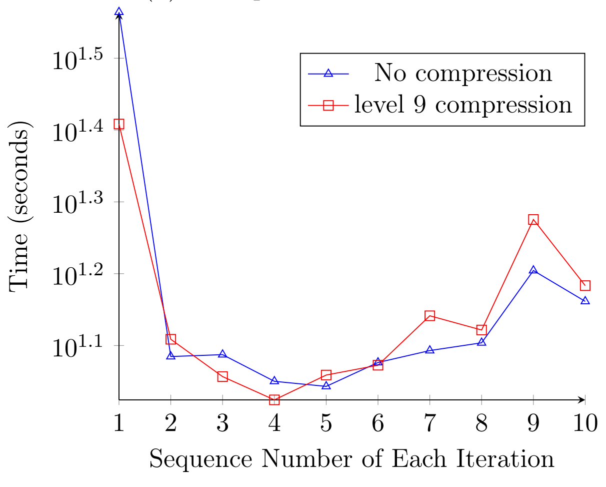
(a) 10Mbps



(b) 30Mbps



(c) 45Mbps



(d) 65Mbps

Figure 3.13: Time for each iteration during a 10 iteration memory image transfer

In Figure 3.13, each curve shows an average of 5 runs with the same experimental setup. Each run consists of the time of 10 iterations, where the first nine are memory difference transfer time before the final handoff starts. The 10th iteration equates to the final handoff time. Each figure shows different bandwidths, with no compression and with level 9 compression. Each data point is an average of 5 runnings with the same experiment parameters. Figure 3.13a shows the time of 10 iterations at the bandwidth of 10*Mbps*. We can see that with level 9 compression, we get slightly better performance than with no compression. However, for higher bandwidths, such as in Figure 3.13b - 3.13d, it is hard to conclude whether level 9 compression option is better than the no compression option.

Apparently, the higher the bandwidth we have, there are more chances that level 9 compression will induce more performance overhead. This is because when bandwidth is high, the CPU power we use to perform compression becomes the bottleneck. This also explains why with increasing iterations, level 9 compression poses greater workloads than the no compression option. When we do more and more iterations for the same container, we have to checkpoint and restore the container again and again. These activities consume many computing resources and create high workloads for the host machine‘s CPU.

Therefore, it is necessary to make the compression option flexible and choose a appropriate compression level suitable for the edge server's available hardware resources.

3.6.3.4 Evaluation of Changing Total Iterations

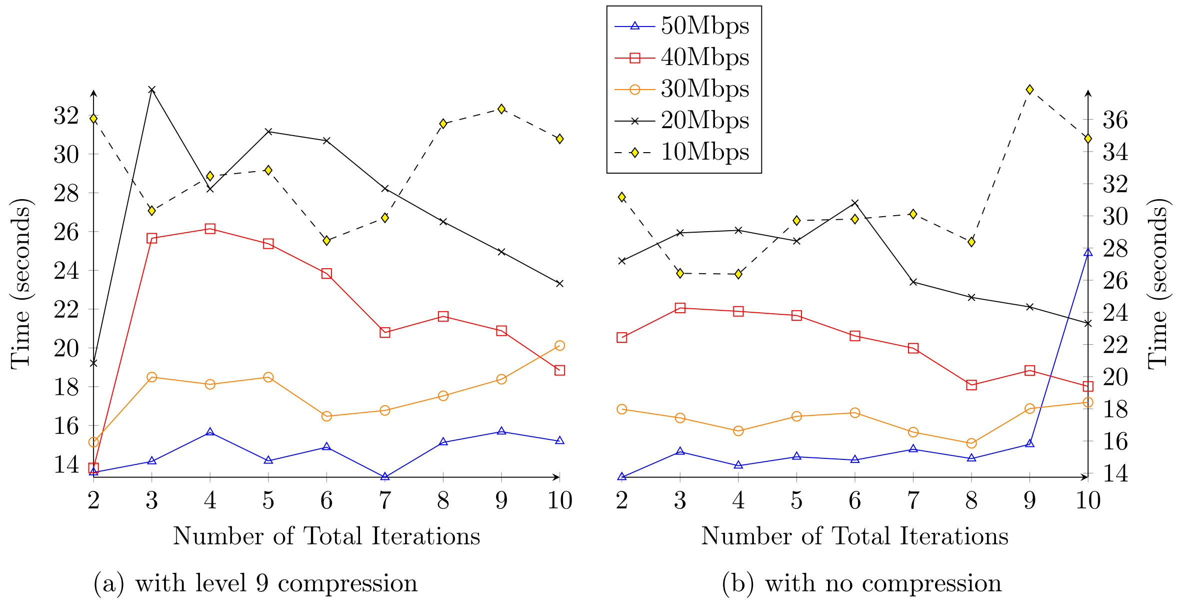


Figure 3.14: Time of Service Handoff Under Different Total Iterations.

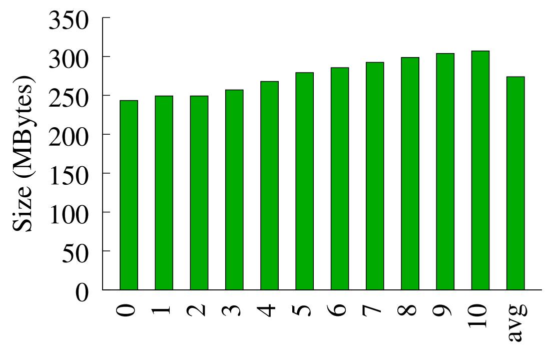
Figure 3.14 shows the handoff time when we use differing numbers of total iterations to transfer the memory image difference before handoff starts. The experiment is done on OpenFace application. Figure 3.14a shows level 9 compression of the transferred data during handoff. Figure 3.14b shows the result when no compression is used during handoff. Each point is an average of 5 runs with the same parameters.

We make two key observations from the figure: **a)** With total iteration numbers of three or more, it is rare to have a better performance than the set up with only two total iterations. **b)** With more total iterations, the final handoff time proves to be longer in most cases.

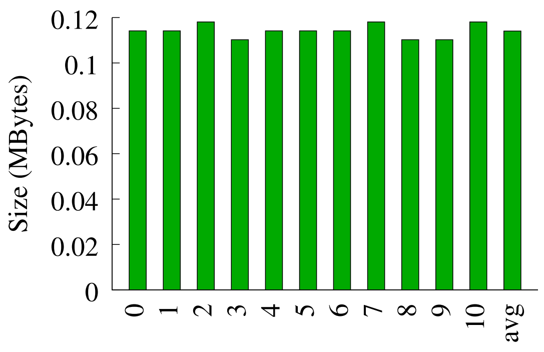
These observations can be explained by the special memory footprint pattern we shown for OpenFace/Busybox in Figure 3.15. Figure 3.15a and 3.15b show the memory size for total 11 dumps (0-10 at x-axis) for OpenFace and Busybox, respectively. Figure 3.15c and 3.15d show dirty memory size between each of dump 1 to dump 10 and the original dump 0, as well as dirty memory size between two adjacent dumps. We can see that no matter how many iterations we checkpoint OpenFace or Busybox, the footprint size in main memory changes little. Although their memory is continuously changing, the changes reside in specific areas: a 4KB area for Busybox, and a 25MB area for OpenFace.

Therefore, no matter how many iterations we perform to synchronize memory difference before handoff, at the end we will have to transfer a similar amount of dirty memory. Additionally, more iterations pose higher workload pressures for the hardware. Therefore, in most cases for OpenFace, it usually does not help to increase iterations.

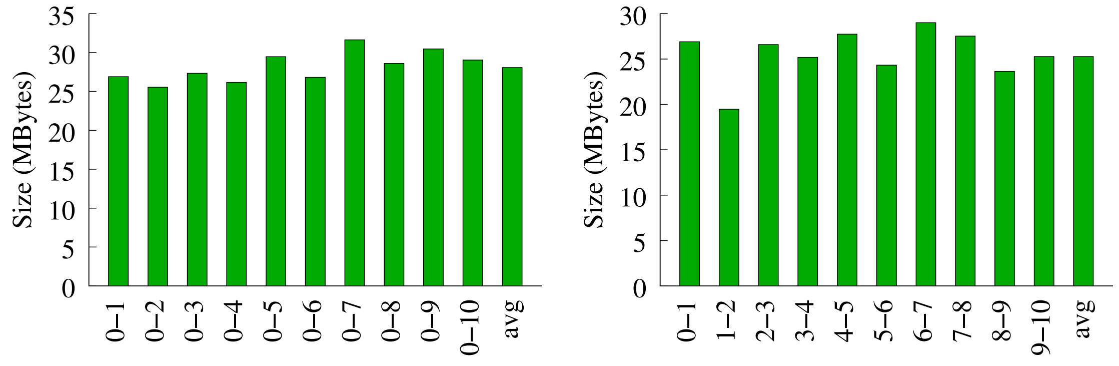
However, this does not mean we do not need more than two iterations for all applications. If the memory footprint size of the application increases linearly over time, we can get smaller memory differences with more iterations. Thus we can save more time by using more iterations.



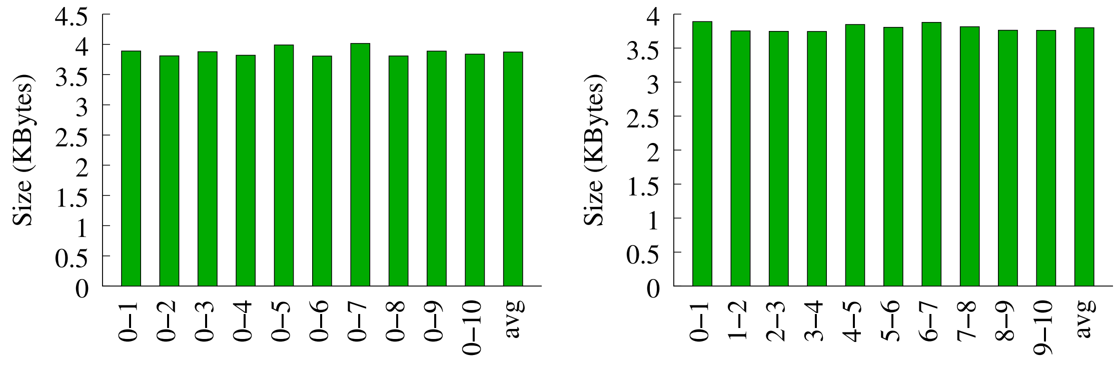
(a) Memory Size, OpenFace



(b) Memory Size, Busybox



(c) Dirty Memory, OpenFace



(d) Dirty Memory, Busybox

Figure 3.15: Dirty Memory Size Analysis for OpenFace and Busybox.

3.6.4 Overall Performance

From Table 3.3 and Figure 3.12 , we can see the OpenFace offloading container can be migrated within 49 seconds under the lowest bandwidth 5Mbps with 50 ms latency, where VM based solution in [80] will take 247 seconds. The relative standard deviations in Table 3.3 show the robustness of our experimental results. In summary, our system reduces the total handoff time by 56% $\sim$ 80% compared to the state-of-the-art work of VM handoff [80] on edge computing platforms.

3.7 Related Work

3.7.1 Edge Computing and Service Mobility

Many leading studies and technologies in recent years have discussed the benefits and challenges of edge computing. Satyanarayanan [141] proposes *cloudlet* as one of the earliest conceptions of edge nodes for offloading end-user computation. Fog computing [24] and Mobile Edge Computing [85, 125] are proposed with similar ideas whereby resource-rich server nodes are placed in close proximity to end users. The idea of edge computing has been found to offer more responsive services as well as higher scalability than cloud platforms [125, 140], thus improving quality of service (QoS) significantly. Several computation offloading schemes from mobile devices to edge servers have been investigated [14, 53, 101, 107]. By offloading to a nearby server, end users will experience services with higher bandwidth, lower latency, as well as higher computation power with less energy consumption on the mobile device.

3.7.2 VM Migration on the Edge

VM handoff solutions based on VM migration have been proposed by Kiryong [80, 81] and Machen [110]. The VM synthesis technique [141] divides huge VM images into a base VM image and a relatively small overlay image for one specific application. Based on the work of VM synthesis, Kiryong [80] proposed VM handoff across cloudlet servers (alias of edge servers). While it reduces transfer size and migration time compared to the traditional VM live migration solution, the total transfer size is still relatively large for a WAN environment. Furthermore, the proposed system requires changes to hypervisor and VMs, which were hard to maintain, and not widely available in the industrial or academic world.

A similar VM-based technique has been proposed by Machen *et al.* [110]. VM images are organized into 2 or 3 layers by pseudo-incremental layering, then layers are synchronized by using the **rsync** incremental file synchronization feature. However, it must duplicate the base layer to compose an incremental layer, causing unnecessary performance overhead. Furthermore, the **rsync** command will cause file contention problems while the service is running, as we discussed in section 3.4.4.

3.7.3 Container Migration on the Edge

Linux containers provide lightweight OS-level virtualization by running a group of processes in an isolated environment. Container runtime is a tool that provides an easy-to-use API for managing containers by abstracting away the low-level technical details of namespaces and cgroups. Such tools include LXC [104], runC [72], rkt [45], OpenVZ [122], Docker [88], etc. Different container runtimes have different scenarios of usage. For example, LXC only cares about full system containers and does not care about the kind of application running inside the container, while Docker aims to encapsulate a specific application within the container.

Migration of containers becomes possible when *CRIU* [52] supports the checkpoint/restore functionality for Linux processes. Now *CRIU* supports the checkpoint and restore of containers for OpenVZ, LXC, and Docker.

Based on *CRIU*, OpenVZ now supports migration of containers [65]. It is claimed that migration could be done within 5 seconds [156]. However, OpenVZ uses a distributed storage system [123], where all files are shared across a high bandwidth network. Due to the limited WAN bandwidth for edge servers, it is not possible to deploy distributed storage. Therefore, the migration technique of OpenVZ containers is not suitable for service handoff on edge computing platforms.

Qiu [131] proposes a solution to live migrating LXC containers in data center environments. However, LXC regards containers as a whole system container, and there is no layered storage. As a result, during container migration, all contents of the file system for that container must be migrated together, along with all memory states.

Machen *et al.* in [110] also proposes live migration of containers with layer support based on the **rsync** incremental feature in addition to their VM-based migration technique. However, their container-based solution only supports predefined 2 or 3 layers of the whole system, while Docker inherently supports more flexible amounts of storage layers. Again, it is also possible to encounter the **rsync** file contention problem when synchronizing the file system while the container is running. Furthermore, duplication of base layers could incur more performance overhead.

For Docker containers, *P.Haul* has examples supporting docker-1.9.0 [65] and docker-1.10 [26]. However, they both transmit the entire file system of the container, regardless of the underlying layered storage. This makes the migration unsatisfactorily slow across the edges of the WAN. And as we mentioned in Section 3.4.4, it also has compatibility issues with Docker storage system and can not avoid the file contention problem while synchronizing with **rsync** command.

3.8 Conclusion

We propose a framework that enhances the mobility of edge services in a three-layer edge computing environment. Leveraging the Docker container layered file system, we eliminate transfers of redundant sizable portions of the application file system. By transferring the base memory image ahead of the handoff, and transferring only the incremental memory difference when migration starts, we further reduce the transfer size during migration. Our prototype system is implemented and thoroughly evaluated under different system configurations. Finally, the hand-off time is reduced by 56% $\sim$ 80% compared to the state-of-the-art VM handoff for edge computing platforms. Our implementation is open sourced at `https://github.com/tupipa/p.haul/tree/docker-1.10-dev`.

Chapter 4

Lightweight Security Monitor for Edge Servers

4.1 Overview

Edge computing promises higher bandwidth and lower latency to end-users. However, most edge computing servers have limited resources comparing to cloud servers. This makes them more sensitive to heavy weight monitoring solutions such as virtual machine introspection (VMI). In addition, edge servers are distributively deployed in large geographical areas, which makes the deployment of security monitoring solutions more challenging than in a centralized cloud center.

In this work, we propose *EdgeVMI*, a framework to monitor and control services running on edge servers with the VMI technique. It is efficient to run on less powerful edge servers and is easy to deploy and maintain over large geographical areas. The key of our contribution is to build the monitor in a lightweight virtual machine running with a library operating system. It improves the security of the system by reducing its trusted computing base. It reduces the performance overhead by running the single-process virtual machine and by leveraging hardware events to monitor memory accesses. In addition, the minimized binary size and the small memory footprints of the monitor also reduce the runtime overhead, as well as the deployment efforts.

Inspired by unikernels, we implement a stripped down version of the state-of-the-art VMI library in a minimized operating system. Based on our tiny VMI library, we build our monitor with only the necessary system modules, libraries, and functionalities for specific monitor tasks. To reduce the security risk of the monitoring behavior, we separate the monitor into two isolated modules: one acts as a sensor to collect security information and another acts as an actuator to conduct control commands. Our evaluation shows the effectiveness and the efficiency of the monitoring system, with an average performance overhead of 2.7%.

4.2 Introduction

Edge computing platforms distribute servers much closer to end users, thus mobile clients can enjoy higher bandwidth and lower latency. Comparing to cloud servers, edge servers are less powerful, more sensitive on latencies and bandwidth, and are deloyed decentralized over the WAN instead of the centralized deployment in the LAN network. These differences introduce new challenges for the security monitoring of services running on the edge servers.

First, security monitoring of cloud services can cause prohibitive overhead on edge servers. Cloud environments host high performance computers with adequate power, and thus are more tolerable for performance overhead of traditional monitoring operations [12, 15, 68, 69]. However, edge servers usually have limited computing resources as well as power supply, which renders the expensive security monitoring solutions in the cloud unpractical for edge servers.

Second, detailed whole system monitoring are not always necessary in edge computing environment. For example, on edge servers that are running algorithms for image processing, facial recognition, object tracking, etc., the data of images or videos are usually public data. Even if the data is private, some of them can be at lower risk when the data is hard to be leaked to the cloud due to its large size and the limited network bandwidths. Therefore, the monitor of edge services should be able to run elastic monitoring, where most of time, a small scale monitoring would be enough and if needed, a larger scale monitoring can be lifted.

Third, the wide distribution of edge servers calls for efficient deployment in scale. The deployment of security solutions in large cloud data centers usually requires much engineering efforts and thus the deployment is hard to scale. However, given the large number of small clusters of edge servers and their geographical distribution, as well as the real time service requirement, it is is unlikely to manage these computing stations in the same fashion as the ones used in cloud data centers. The highly heterogeneous edge environment would make the deployment and maintenance rather complicated and budget burning. Therefore, the cost efficiency model in centralized cloud is no longer valid for edge platform.

In order to meet the aforementioned challenges, it is critical to provide a dedicated security framework that has low overhead, elastic monitoring capability, as well as effortless deployment on edge computing platforms. In this project, we present a framework, *EdgeVMI*, to meet these requirements. Our monitor is built with a unikernel [111] system, where all unnecessary software modules are eliminated in the system. More specifically, we adopt the state-of-the-art virtual machine introspection (VMI) technique which allows transparent and trustworthy monitoring from underlying of a commercial operating system. To the best of the author's knowledge, this is the first work to introduce a lightweight VMI in Unikernels for runtime security monitoring on constrained platforms such as edge servers.

The contributions of this work are briefly listed below:

- A lightweight monitoring virtual machine is designed to monitor security critical components residing in any software layers of a target virtual machine. The target virtual machines being monitored run commercial operating systems such as Windows, Linux, etc.

- We build the monitor with a stripped-down version of the state-of-the-art VMI library by removing its heavy-weight dependencies on the operating system as well as the large user-space libraries, which reduces the attack surface but also improves performance, as well as reduces the package size during deployment.
- Strict control flow and data flow policies are designed across interfaces between different modules in the monitoring framework. These policies define mandatory access control constraints over the interfaces between the monitor and target, as well as the interfaces between different modules inside the monitor.
- A fast deployment pipeline is designed with user customization, building, shipment, plugin, and monitoring, where an optional feedback loop can be used between different stages to control the monitoring behaviors.
- We build the prototype system and evaluation shows an average overhead of 2.7% for the passive event based monitoring. The performance of the primary VMI operations are improved by more than 30% comparing with the state-of-the-art.

Section 4.3 presents how we identified the research problem and discusses its importance. Section 4.4 discusses our threat model, design goals, and system overview, with an example of intrusion detection to show the workflow of the framework. Section 4.5 introduces how to build, run, and deploy our monitoring framework. Section 4.6 discusses the implementation challenges and major modules of *EdgeVMI*. We evaluate the system in Section 4.7 for its security and Section 4.8 for its performance. Then we discuss the limitations and possible improvement in Section 4.9. Section 4.10 presents the related work and Section 4.11 presents concluding remarks.

4.3 Motivation

We discuss our motivation by answering the following questions: **a)** Why VMI solution is a good option for edge security monitoring? **b)** Why we need to improve VMI to meet the requirement of edge environment?

4.3.1 Why VMI?

Virtual machine introspection [74] has been widely used for malware analysis and computer forensics in the last decades [35, 74, 91, 92, 126, 150, 178]. Leveraging VMI technique, one can reconstruct the semantic details of a running virtual machine by viewing its raw memory, hardware events, and the states of the vCPU registers [128].

The widely distributed infrastructure and emerging new applications running on the edge environment pose new security requirement for edge platforms. Security monitoring solutions on the edge platform need to have low overhead and be trustworthy. The following features of VMI can benefit the monitoring operations on edge servers:

1. **Transparency.** Most malwares are sensitive to its environment and can hide themselves after intrusion into the victim. However, memory accesses to a guest VM's memory via VMI are transparent to the entire OS where the malware is running. Therefore, VMI-based monitoring is hard to be detected by malwares [92, 103].

2. **Efficient Hardware Assisted Monitoring**. Traditional software based VMI operations can be overwhelming where the memory might be scanned periodically and the raw memory must be reconstructed in order to identify any risks. However, with the event support in recent processors, we are able to monitor the memory events in a more efficient way that does not need periodical scan.

4.3.2 Why We Need Improved VMI?

To answer why we need to improve VMI, we now discuss the two problems of current VMI techniques we have identified: a) The trusted computing base is large; b) The overhead is high for runtime monitoring.

Table 4.1: VMI System Code Base in Lines of Code (LoC)

EdgeVMI	*EdgeVMI* + Mini-OS	LibVMI	LibVMI + Linux
27.2K	**54.7K**	31.0K	**15M**

4.3.2.1 The Trusted Computing Base is Large

LibVMI [126] is a popular library for VMI operations on commercial operation systems, including both Windows and Linux. It is built in a traditional operating system that is usually privileged with large code base, such as Dom 0 in Xen, or the host OS of KVM. Table 4.1 shows the line of code (LoC)[1] of our proposed system versus the state-of-the-art LibVMI system. It is clear that the introspection functionalities with the Linux operating system all together will expose a large attack surface for the platform.

4.3.2.2 The Overhead is High for Runtime Monitoring

VMI is originally designed for offline malware analysis, not online monitoring at regular execution time. Since malwares are assumed to be in any forms, the analysis logic needs to support all kinds of behaviors from any kind of code. For example, one might need to track down all the CPU or memory traces in order to analysis the instruction sequence or memory behavior of a malware. Apparently, such monitoring operations are too heavy weight to be deployed online.

In addition, running the monitor in a large commercial operating system can also introduce overhead. First, memory resources on the system can be easily exhausted. For example, a 64-bit Linux VM would normally require 4GB memory. This overhead is high for an edge server, especially when we want to run multiple different monitor VMs. Second, CPU resources can be wasted by the OS itself. OS itself run several kernel services to maintain its own functionality regardless the applications, which can occupy CPU time.

[1]All lines of code (LoC) in this report are counting only code, without comments nor blank lines, with data generated by the tool *Cloc* (`https://github.com/AlDanial/cloc`).

Third, there will be a double-scheduling overhead to run a process in such VMs. For example, when we run a single monitoring processes inside a Linux VM, the OS will need to do scheduling in order to run the monitor, given that the underlying hypervisor also needs a scheduler to run that VM. Fourth, it is apparent that the disk size of the VM image can be extremely high. For example, Linux VM image with the VMI application is at least two to three hundreds of megabytes, which can increase network traffic during deployment as well as booting delays when being executed.

Therefore, we aim to design a lightweight version of VMI with functionalities specialized for security policies and use a minimized library operating system (i.e., unikernel [95, 111, 127]) to build our monitor.

4.4 Design Overview

In this section, we first present the threat model. Then we discuss the design goals of the system. Next we introduce the overview of *EdgeVMI*. Finally we discuss its security enhancements and show an example of intrusion detection.

4.4.1 Threat Model

Here we introduce the security assumptions and define the scope of problems in this framework. We first discuss the assumptions in terms of the secure and unsecure components on the edge platform. Then we define our targeted application scenarios.

4.4.1.1 Trusted Components

The target VM in this work is a guest VM that provides services to end users. Target VMs are assumed to be securely booted with trusted booting [39, 77, 112]. During booting, all the software stack of the system in execution are checked for their integrity, including all the operating system kernels and applications running inside the VMs. If the attacker modifies the virtual machine images and try to boot the system, we assume it will be detected and prevented by trusted booting.

We assume the hypervisor provides strict isolation between VMs. *EdgeVMI* can get the execution states of target VMs by communicating with hypervisor interfaces. There is no direct communication channel between target VMs and the VMs running *EdgeVMI*. Therefore, if the guest OS is compromised, the data observed from *EdgeVMI* is still trustworthy.

We assume the service providers on the cloud are trusted. They play an important role on the cloud control center to remotely manage the edge servers. They are also responsible for the deployment and maintenance of the physical edge servers. How to protect the users from a malicious service providers in the cloud is out of the scope of this work.

We assume internet connections between the mobile devices, the edge, and the cloud are protected against leakage, corruption, or denial of service.

4.4.1.2 Threat Source

After securely booted, malicious parties might leverage vulnerabilities in the applications or the OS kernel to gain unauthorized access to the data inside the target VM, or gain control over the entire OS. Therefore, we assume the entire target VM in execution to be controlled by malicious party. This includes the applications and operating system kernels running inside the target VM. Attacks where the hypervisor, or its underlying physical hardware is controlled by malicious parties are out of the scope of this work.

4.4.1.3 Applications that Need Small Scale Monitoring

Our framework aims to provide small scale monitoring for specific edge applications. Small scale monitoring is practical for many edge services. First, end-users tend to offloading only the computation intensive tasks, such as facial recognition, image/video processing, etc. Second, the processing of private data and security critical code can be kept in the private devices, such as smart phones or IoT devices owned by users. Therefore, *EdgeVMI* aims to provide small scale system monitoring for the protection of possible small amount of security critical data or code stored on the edge servers. *EdgeVMI* has the capability to monitor the entire program, such as analyzing the entire execution traces for an application, but this is not suggested for its high performance overhead. This limited scope allows *EdgeVMI* to effectively ignore most of the execution traces and runtime memory states that are not in the monitoring range defined by the user.

4.4.2 Design Goals

EdgeVMI framework aims to be a minimized system tailored for low level security monitoring and control on edge platforms. To make it practical and efficient, we recognize the following key properties we need to achieve:

4.4.2.1 Small Code Base

A minimized code base will keep the TCB of the monitoring system as small as possible thus reduce the attack surface. Using a small operating system can reduce the code base significantly. For example, Xen Mini-OS has $27.5K$ lines of code (LoC), while Linux kernel has about $147K$ LoC (version 4.16), and a fully functioned Linux system with all the drivers and architecture support has reached more than 15+ million lines of code [49]. A small binary package with no dependencies to outside dynamic libraries would also reduce the attack surface of the system as well as improve the performance. Therefore, our system design should try to be as self-contained as possible by reducing the dependencies to outside system libraries.

4.4.2.2 Transparency

As we introduce in section 4.3, rootkits or malwares are likely to be stealthy by detecting the existence of the monitor and hiding themselves from being detected. While VMI is prone to be stealthy, we still need to carefully design the framework to make it as transparent as possible. One strategy is to design the monitor with minimal interference with the target system. Ideally, the monitor operations should never interfere with the execution of the monitored system.

4.4.2.3 Least Privileges

Least privilege is one of fundamental principles for secure system design [138]. It requires that in a particular abstraction layer of a computing environment, every module (such as a process, a user, or a program) must be able to access only the information and resources that are necessary for its legitimate purpose [138, 139, 142, 143]. Monitoring the raw physical memory of an OS requires higher privileges than the OS. If the privileged monitor is not carefully administrated, it will have a high risk to cause a crash or undesired damages to the entire software and hardware system. Therefore, our framework as a low level monitor and controller, should be strictly constrained with limited capabilities.

4.4.2.4 Isolated Components

Our framework consists of two categories of operations, one for data collection and one for control operations. Data collection operations only query the data from target system and will not change any runtime states of the target, just like a sensor observing the environmental statistics. On the other hand, the control operations at low level of the system, such as start or stop a VM, will change the runtime states of the target system, but don't need much detailed information from inside the target system. Therefore, it is reasonable to design our system containing two isolated components. Each has minimal privileges that are just enough to conduct the two different categories of operations.

Furthermore, by isolation and least privilege, our framework could get partial trustworthiness even if one of the components got compromised. This is essential for intrusion detection scenarios. Since we aims to find out abnormal behaviors after the system got compromised, a privilege separated system design would improve the trustworthiness under risky environments.

4.4.2.5 Easy to Deploy

In order to make our framework easy to deploy, we aim to build and ship images with minimal package size and leverage the cloud center as a centralized management spot to distribute the monitoring VMs. In addition, we aim to design the control operations on the distributed edge with as little human interference as possible. Then, on edge servers, the deployment and maintenance of the VM monitors should also be automated such that the on site deployment and maintenance over large geographical areas could be avoided.

4.4.2.6 Flexible with Rich Primary Mechanisms

At low level of a system, such as raw machine memory or processor register states, the semantic view of the high level information is limited. More efforts much be done to reconstruct the high level information, such as the process name in execution, etc. Since our framework is designed to live at a low level interface for more specific monitoring services, it should provide basic mechanisms to provide a rich set of low level library interfaces for high level monitoring operations to reconstruct the semantic view and build their own security policies for monitoring.

4.4.3 *EdgeVMI* Overview

Figure 4.1 gives an overview of the monitoring framework. *EdgeVMI* monitor executes directly on top of a type-1 hypervisor (such as Xen). It can introspect into the multiple layers inside the target VM, such as the Linux kernel, containers, processes running in the VM. All monitoring operations are done without direct communication with the target VM. Instead, we achieve this via the application binary interfaces (ABIs) provided by the hypervisor to receive hardware event notifications and inspect the virtualized hardware registers and memory.

Figure 4.2 shows the high level view of the monitoring network for *EdgeVMI* framework. Multiple edge servers are distributed across the wide area network (WAN). Each edge server

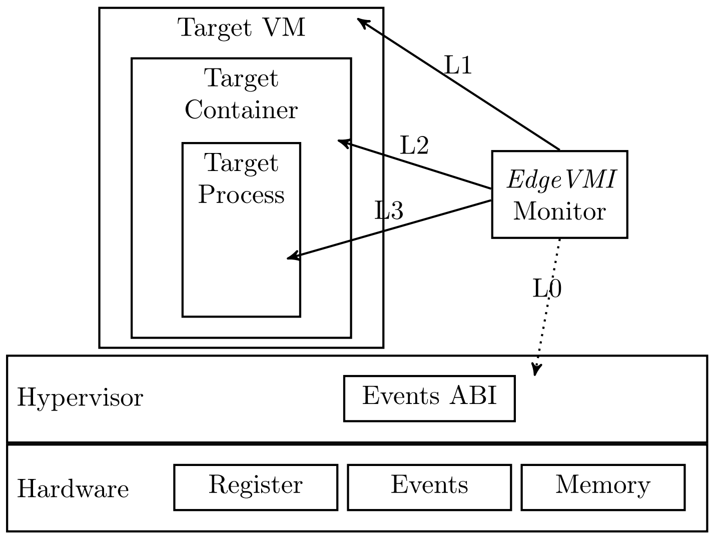


Figure 4.1: One Layer to Monitor All Layers via *EdgeVMI* .

runs multiple Target VMs. Target VMs provide services for end users. Inside the Target VMs are running traditional operating systems such as Linux, Windows, etc. Two monitor VMs called *Sensor* and *Actuator* are running a minimal operating system with tailored functionalities for specific purpose of monitor and control respectively. Monitors on edge servers are connected via a centralized control center on the cloud, which can take sanity checks on their requests and make control decisions based on the collected information. The *Sensor* VM and *Actuator* VM do not directly talk to each other in order to retain the strict isolation and reduce the overall risk that might be caused by unknown bugs in any of them.

4.4.3.1 Sensor

Sensor is the component to collect low level information by accessing the raw physical memory, virtual CPU register values, or by receiving hardware event callbacks. One or more *Sensors* can be deployed to observe different low level information of one target VM. Each *Sensor* can be built with a dedicated single purposed functionality to keep it as small as possible. On the other hand, one *Sensor* can be used to monitor multiple different target VMs.

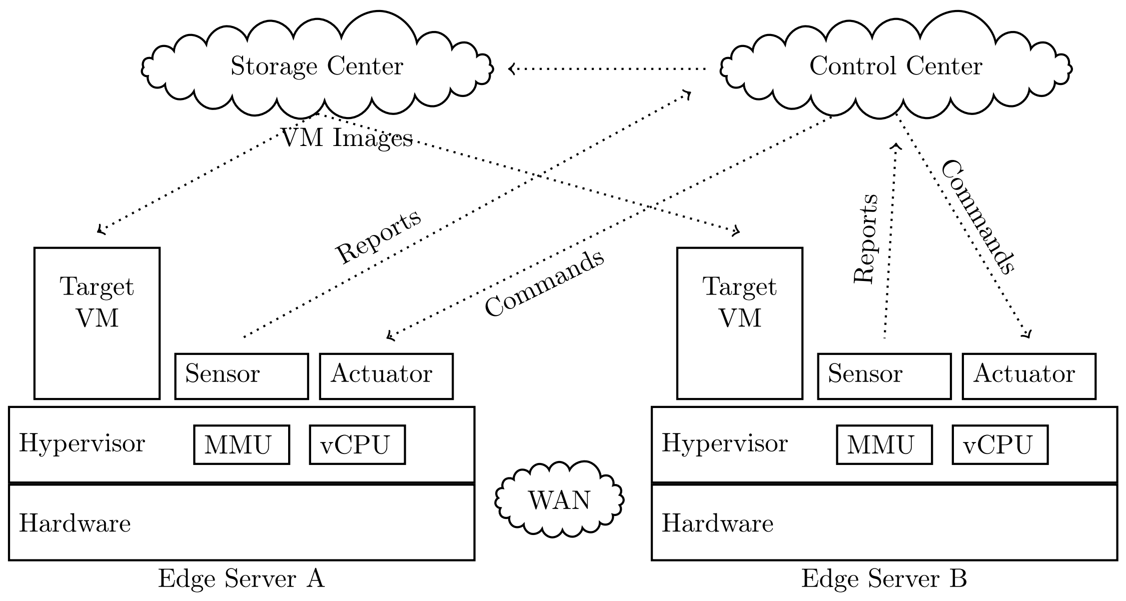


Figure 4.2: *EdgeVMI* Deploy Example under Centralized Cloud Management

4.4.3.2 Actuator

The *Actuator* is the component that receives commands from the cloud center and controls the life cycle of a VM running on the edge server. The VMs that the *Actuator* controls include target VMs which host end-users' applications, as well as the *Sensor* VMs that are conducting various kinds of monitor operations on the server. The control operations it could conduct is designed to be only life cycle management, such as to start/stop, pause/resume VMs, and download/deploy VM images. All operations must be triggered by command messages that is verified to be originated from the trusted control center in the cloud.

4.4.3.3 Cloud Storage and Control Center

EdgeVMI uses a centralized storage system for fast deployment of VM images across distributed edge servers. It has a cloud control center to analyze the low level runtime information collected from the *Sensors* on edge servers. The control center is also responsible to make decisions about the intrusion detection analysis and send explicit commands to the *Actuators* to execute the decision.

4.4.4 Harden the Security of Monitors

Monitors are usually privileged software, which can be easily exposed as attacking target by malicious parties. Therefore, the secure design of a monitor is critically important. In *EdgeVMI*, in addition to reducing the attacking surface by minimizing its code base, it is also hardened by more strategies including privilege separation, constrained information and control flows.

4.4.4.1 Privilege Separation of Monitor and Control

As shown in Figure 4.2, *EdgeVMI* is separated into two categories of VMs, the *Sensors* and *Actuators*, in order to constrain the effects of bugs in any of them. This meets the principle of privilege separation and least privilege to design secure systems. Each of the *Sensors* and *Actuators* has its own specialized functionalities and its privileges are limited only for the pre-defined purposes of monitoring or control. There are no interfaces they can talk directly to each other between the *EdgeVMI* VMs. This retains the strict isolation between each monitor module, which helps to constrain the risk of unknown vulnerabilities in any of them. Their communications, instead, need to go through the control center on the cloud, which can take sanity checks on their requests as well as their collected information.

4.4.4.2 Control and Information Flow Restrictions

As shown in Figure 4.2, the data and control flow patterns are all with single direction. With data flows in a single direction between two directly-connected components, the isolation between the two is more strictly preserved, which improves the security of the entire system. For example, the data of runtime states flows from the *Sensor* to the cloud control center. *Sensor* have to be privileged to be able to observe all the runtime states of the target VMs. But in order to change the behavior of the target system, the monitor does not rely on the functionality of the *Sensor*, but another isolated VM *Actuator* instead.

4.4.5 Example of Intrusion Detection

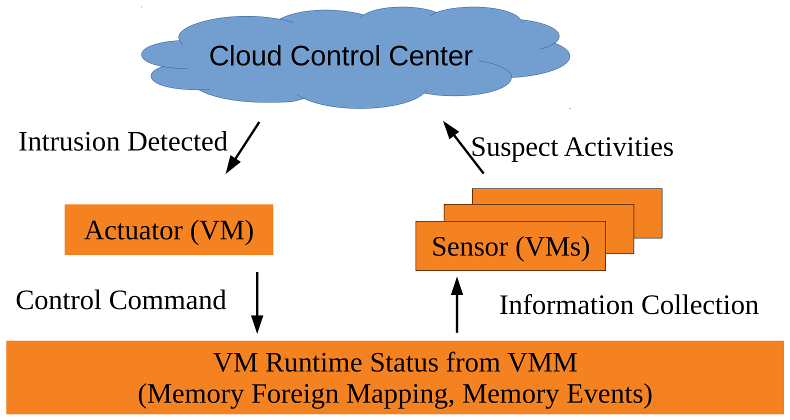


Figure 4.3: Run-time Workflow for Intrusion Detection and Emergency Response

Figure 4.3 shows an example control flow of intrusion detection and the response process in *EdgeVMI*. We assume the target VM encountered a memory corruption attach that is caused by a malicious process and the integrity of a critical data in the memory has been compromised.

Initially, one of the *Sensor* VMs is pre-built with capabilities of monitoring the critical region that the malware has changed, such as interrupt tables, or keyboard drivers, etc. It is achieved by observing a predefined memory region of the target VM's physical memory as well as the hardware event triggers registered during the booting process of the *Sensor*.

Once any of the monitored memory regions is changed, the corresponding *Sensor* VM will be notified. Then the *Sensor* VM will verify the change against the security policies it is built with. Once it decides the change might be malicious, a report will be sent to the control center. Then the control center will take further analysis.

The control center has a broader view of edge environments by observing multiple edge services' states as well as the service usage history for the end users. Therefore, based on the report received from the *Sensor* VM and the control center's previous knowledge about the execution environment, control center will determine whether there is an intrusion on the edge server.

Once the intrusion is detected at the cloud, response commands will be sent to the *Actuator* VM, where the the life cycle of target VMs or *Sensor* VMs might be updated accordingly. After the response commands are completed, *EdgeVMI* will reach to a new and starting state, where a new iteration of monitoring and control will be in place.

4.5 *EdgeVMI* in Deployment

EdgeVMI enables developers to build a set of single-purpose tiny VM monitors with ease. It provides low level system mechanisms in order to compose complex monitoring policies. These mechanisms include the monitoring of raw physical memory access, hardware events, and VM life cycle management. In addition, the framework also supports fast VM deployment over distributed servers.

4.5.1 Build Single-Purpose, Deeply Customized Monitor VMs

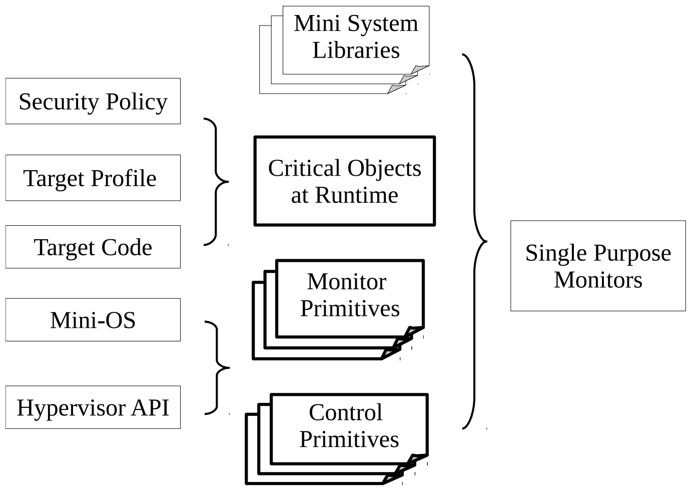


Figure 4.4: Build Single-Purpose Tiny Monitor VMs in *EdgeVMI*

Figure 4.4 presents an overview of the workflow in order to build a monitor using our toolchain in *EdgeVMI*. Key steps include the identification of security critical objects at runtime, and selecting monitor and control primitives to enforce specific security policies.

4.5.1.1 Security Policy

Security policy are usually defined by developers or users in terms of the security critical modules in the target VM. The policy will specify the portion of the target VM that needs to be monitored, the rules to evaluate the risk, as well as the actions that need to be taken once the risk condition is detected. For example, a password confidentiality policy will need to specify a) where the password is stored in the system, b) under which condition the system will consider the password is in the danger of being leaked, and c) how to response once the password is in danger.

4.5.1.2 Target Profile

Target profile contains the metadata from the operating system kernel as well as the applications which will help to reconstruct the semantic information of the target from the raw binary contents. These include the configuration of the target VM, the symbol table from its OS kernel, and debugging information from the applications' binary, etc.

4.5.1.3 Mini-OS and Hypervisor API

Mini-OS and *Hypervisor API* together are leveraged to build our *monitor primitives* and *control* primitives. More specifically, we leverage the memory management and hardware event dispatching interfaces from the hypervisor. *Mini-OS* provides us the the minimal execution environment in order to run applications on top of the hypervisor without the commercial operating system intervened.

4.5.1.4 Mini System Libraries

Mini System Libraries are our own stripped down version of several system libraries, such as the lightweight version of LibVMI [126], the stripped GLib [170], etc. They are used in order to resolve the library dependencies of the monitoring functionalities.

4.5.1.5 Critical Objects

Critical objects are the actual substances monitored by *EdgeVMI* at runtime. For example, if we need to audit the authentication process on edge servers, the critical objects can be the password related code and data. The critical objects can either be statically identified, or dynamically identified during execution by parsing the raw memory content. This can be done by combining the *security policy, target profile*, and the target source code. If the source code and the profile information of the target are not available, *EdgeVMI* can still work but will probably generate false negatives or positives during the identification of the object. However, the problem to improve the accuracy under such conditions is out of the scope of this work.

4.5.1.6 Monitor and Control Primitives

Monitor and *control primitives* are a set of simple operation units, where each one provides a basic functionality. The primitive's functionality can not be divided into smaller functionalities, or will be meaningless in terms of the monitor or control functionality, thus we call it a primitive. For example, $read\_event(addr\_start, addr\_end, proc\_id, \dots)$ is a primitive which could be used to monitor a memory region defined by the start address and end address, $[addr\_start, addr\_end]$, within the process identified by $proc\_id$. If any of the memory in this region is read in the process, the monitor will be notified by this primitive. Different primitives can be composed together in order to implement a complex function for monitor or control.

4.5.1.7 Build Monitor VMs

Finally, according to the identified *critical objects*, we could determine which kinds of primitives we need to use. Then the required primitives along with the target object information can be automatically built into the library OS along with their required portion of the VMI libarry. The final executable binary can be made pretty small which saves memory resources as well as computing resources, as we will see in Section 4.8.

4.5.2 Raw Physical Memory Access

Modern hypervisors virtualize the machine memory to provide different views of physical memory for different guest operating systems to run on. Basically it provides another layer of memory virtualization in addition to the traditional memory virtualization inside the operating system. To access the memory, a virtual address (virtualized by guest OS) of an application is translated to a machine address of a VM by the guest OS inside the VM; then the physical address (virtualized by hypervisor) of the VM is translated to the machine address (actual hardware address) by the hypervisor.

In *EdgeVMI*, we leverage the virtualization of machine memory by the hypervisor to enable one guest VM to access another guest VM's physical memory. This is done by requesting the hypervisor to map the target VM's physical memory pages into the monitor VM's memory. The mapping operation is transparent for target VM since it is operated at the hypervisor layer and no changes are made to the target VM's view of its main memory or execution states.

4.5.3 Monitor Processor Registers

Registers of virtual processors can also be obtained directly by observing the target VM's physical memory, since all these will be saved into the memory after the target VM is not occupying the real processor (due to the scheduling of the hypervisor). Furthermore, most modern hardware support events notification when the given hardware registers or memory region are read, written, or executed. The register information could be used to monitor specific system calls, the process start, stop, and context switch, as well as to traverse the address space of processes to get rich semantic view of the activities of an application. For example, on x86 platform, we could monitor the processes' context switches by monitoring the change of control register of **CR3**.

4.5.4 Hardware Event Monitoring

As we mentioned above, modern hardware support events notification whenever any hardware registers or memory regions are read, written, or executed. In addition, modern hypervisors themselves also have their rich event support for efficient communication between the guest VMs and the hypervisor, such as event channels on the Xen hypervisor [129]. *EdgeVMI* utilizes these features to monitor a various kinds of low level events. For example, a *Sensor* VM is able to receive notification when a specific page in memory has been read, written, or executed, with the event support from Xen hypervisor.

The asynchronous communication in event based monitoring is more efficient compared to traditional synchronous way where memory must be synchronized or periodically scanned in order to state memory changes.

4.5.5 VM Life Cycle Management

The *Actuator* of *EdgeVMI* is designed to manage the life cycle of guest VMs, including both the target VMs and the monitor VMs. This is the way for *EdgeVMI* to take emergency response to the possible intrusion. In our design, life cycle management includes only operations that can impact the life cycle directly, such as start/stop, pause/resume, or create/destroy a virtual machine. Each *Actuator* is limited to operate only the virtual machines running on the same hypervisor with itself.

4.5.6 Deployment

We aim to build, package, and ship the monitor images in an efficient way for fast deployment and efficient maintenance over the widely distributed edge environments. Since *Sensors* are usually tailored for a dedicated monitoring functionality, it will be specifically built with the exact functionalities being used, without any unused functionalities or modules. *Actuators* are tailored for controlling all the guest VMs on a native edge server. It will be deployed together with the installation of the hypervisor.

For both *Sensors* and *Actuators*, once they are deployed, no further changes, such as updating, reconfiguration, or reconnect to another target VMs or reconnect to another cloud center are allowed during their normal operation life cycles. The only time we can change the *Sensor* is when its target VMs have all been destroyed. The only time we can update the *Actuator* is when the hypervisor is changed or updated. This model of life cycle management simplifies deployment and maintenance and also close the door for many possible attack sources.

4.6 Implementation of a Prototype

A prototype system is built to better understand its implementation difficulties, performance, as well as its effectiveness of *EdgeVMI*. We choose Xen [20] as an example implementation in our prototype. *EdgeVMI* can also be extended to other hypervisors such as KVM [97], Bareflank [19], etc.

For the choice of the library OS, we use Xen Mini-OS [127] as an example implementation for its inherent compatibility with the Xen hypervisor. However, the *Sensor* and *Actuator* VMs of *EdgeVMI* are designed for a general library OS (or unikernel [111]), thus they are portable to many others which support the C language, such as Exokernel [66], Rumpkernel [95], IncludeOS [27], etc.

4.6.1 Challenge 1: Stripping Down Libraries

In order to enable VMI operations in a minimized library OS, we need to port several library dependencies into the OS kernel. However, in a plain library OS, there is usually no development libraries available. This is especially true for the Mini-OS in Xen, with absence of the **libc** (library for C language), which is a fundamental development library for C programs and many other libraries.

Combining the developing efforts as well as the system efficiency, we might choose one of the following solution for each library in order to port them into our Mini-OS:

1. Manually porting the library in the brute force way. This requires the most effort, where we will manually port most code base of a library into the new library OS. This is used when a library is not compatible with the OS but a large portion of it is needed in our system. This is used for the VMI library as we will discuss below in 4.6.4.4.

2. Manually create a lightweight version of the library. This requires relatively less effort than above. It is used when a lightweight version of the library is not available. Usually those libraries are very large but only a small portion of it is in use in our system, such as the GLib [170], which we will discuss below in 4.6.4.1.

3. Cross compiling an existing lightweight version of the library. This requires the least effort. We choose this only for small, important libraries, such as **libc**, or libraries that are better not to be changed, such as **libxenctrl**. We will discuss these further in 4.6.4.1.

After stripped down or cross compiled, all libraries in Mini-OS are statically linked to make it a single self-contained binary package. Note that we do not choose to simply cross-compile all the required libraries into the Min-OS, even though it is feasible in most cases. This is because cross compiling some large libraries into the system will increase the trusted computing base of the monitor which will increase the security risk of the system. In addition, including lots of unused library functions will be a waste of the memory and will increase the binary size of the final executable, which will potentially reduce the performance of the monitor system.

Table 4.2 shows an overview of our porting efforts for different libraries. For LibVMI, we have manually ported all of its key functionalities into the Mini-OS, with a total of 80 files and 20.6K lines of code (LoC) being patched into Mini-OS. We have manually crafted a lightweight version of GLib with 19 new files and 2.1K lines of code being added to Mini-OS. For other essential libraries, such as **newlib, json-c, libiconv, lwip, libxenctrl,**

Table 4.2: Libraries Ported to Mini-OS

Library	Porting Method	Files Ported	LoC Patched
LibVMI	Manual (All)	80	20.6K
GLib	Manual (Partial)	19	2.1K
newlib, json-c, libiconv, lwip, libxenctrl, libxenstore	Cross Compile	1 (Makefile)	321

and **libxenstore**, we have cross compiled them into Mini-OS by modifying the building commands in the corresponding *Makefile*.

4.6.2 Challenge 2: Input & Output without a File System

VMI operations need multiple input files to assist the reconstruction of the semantic views from the raw memory of the target VM. These files include the kernel symbol file, VM configuration file, application debugging information, etc. But Mini-OS does not have a file system. It is possible to build a file system into the Mini-OS, but it will be a waste of memory space in order to support just a few files. Therefore, we choose to find an alternative solution.

There are multiple possible solutions to feed information to Mini-OS on the Xen platform. It could go through the network interface, hypervisor interface, or hard-coded the information statically by the compiler. Network interface could potentially introduce threatens from the Internet. Using the hypervisor interface will require to add more functionality to the hypervisor, which will make the solution not compatible with legacy hypervisors and reduce its portability during delopyment. Therefore, we choose the hard-coded solution for input files. We use Linux command **xxd** to encode the file as C language string variables and cross compile it into the library OS and interpret the strings as files via **fmemopen** interface from the C library. For output, we use the internet interface to communicate with the control center in the cloud.

4.6.3 Challenge 3: Mandatory Access Control

In our design, *Sensor* VMs require the privileges of read-only access over the target VM's physical memory and the *Actuator* VM needs privileged permissions to control the life cycle of the guest VMs, but without access to the memory content of any guest VM. To reduce the security impact, *Sensors* and the *Actuator* also need to be isolated with the least privileges. However, by default, the hypervisor does not allow one VM to access the memory of another VM. To enable the new access patterns, we design new communication channels between VMs by updating the FLASK [154] policy in Xen Security Module (XSM) [41].

FLASK (Flux Advanced Security Kernel), has its origins in several trusted operating system research projects, best known through its expression in *Security-Enhanced Linux* (SELinux) [154]. XSM can group VMs by assigning different FLASK type labels to different guest VMs. Then each type of VMs could be assigned with different permissions. In our prototype, three new VM type labels are created for *Sensor*, *Actuator*, and *Target* VMs, respectively as *domSensor_t*, *domActuator_t*, and *domTarget_t*. Then between each two VM types, we define clearly how many permissions are allowed. The permission is defined by whether a type of VM could execute the hypercall or not. For example, the rule:

```
allow domSensor_t domTarget_t:mmu {map_read};
```

declares that a *Sensor* VM (labeled as *domSensor_t*) could execute a **map_read** hypercall to read the raw memory of the target guest (labeled as *domTarget_t*). In total, we have granted 15 hypercalls to the *Sensor* VM type in order for it to monitor a target VM's raw memory. The *Actuator* VM type has 8 hypercall permissions.

4.6.4 Components of the *Sensor*

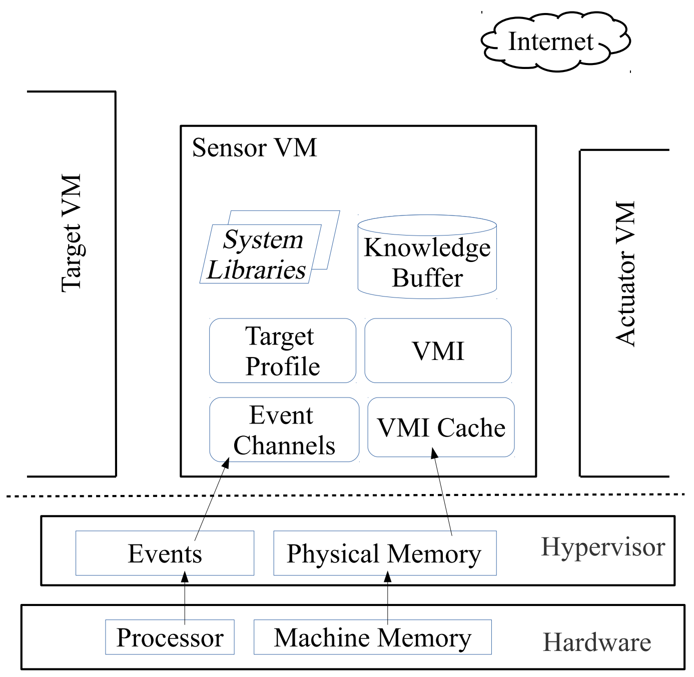


Figure 4.5: An Example implementation of *Sensor* VM

Figure 4.5 shows the composition of the *Sensor* implementation. It passively monitors the target VM: Changes in the physical memory can generate hardware events by which the *Sensor* will be notified. It contains several components which we will discuss below.

4.6.4.1 System Libraries

System Libraries are dependency libraries for VMI component as well as other components. They can be classified into 3 categories:

1. Libraries for data structure utilities. These libraries provide advanced data structures such as linked list, hash tables, etc. These libraries are essential to achieve high performance for the monitor operations. For example, VMI operations rely on several advanced data structures that reside in GLib [170]. However, as we mentioned before, there is no lightweight libraries available in Mini-OS environment that could provide these advanced data structure utilities. Therefore, we have developed our own utility

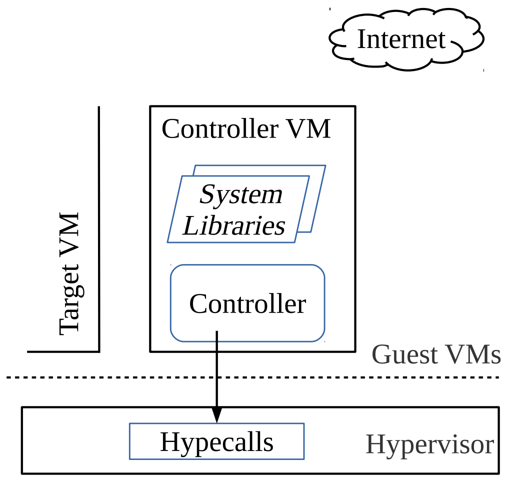


Figure 4.6: An Example of *Actuator* VM

libraries.

2. Libraries of Mini-OS kernel. These are libraries for a program to interact with the Mini-OS kernel, such as **newlib** and **lwip**. The **newlib** is a C library intended for use on embedded systems [94]. The **lwip** is a widely used open source TCP/IP stack implementation that is designed for embedded systems [63]. These two libraries provide fundamental support to develop new functionalities in Mini-OS.

3. Libraries of hypervisor. These are libraries to interact with the hypervisor, such as **libxenctrl, libxenstore** on Xen. These libraries include hypercall interfaces for the Mini-OS kernel or the applications in the Mini-OS to directly communicate with the hypervisor. To monitor the events and memory states, as well as to manage the life cycle of VMs on the hypervisor, all require to issue hypercalls via these libraries.

4.6.4.2 Target Profile

Target Profile contains the pre-known information about the operating system or applications running in the target VM. For example, in order to monitor a certain memory region in kernel space, it is necessary to know the actual memory layout of the kernel. This can be done by importing the kernel symbol information into the *Sensor*'s target profile component. In addition, we also support using the Rekall profile [2] to introspect into Windows OS kernel and applications.

4.6.4.3 Knowledge Buffer

Knowledge Buffer is provided to cache the semantic reconstruction results. It helps to avoid duplicated semantic reconstruction operations for the same requests. For example, if we receive a request to query whether there is a process exist or not, we will trigger a series of VMI operations to search the raw memory of the target VM. Then we will reconstruct a full list of the processes with their properties such as process ID and names, etc. After this, we can response to the query. At the same time, we can also store the reconstructed semantic views, here the full list of processes, into the buffer. Then if we receive a second similar query command, we could return the result by just querying this *Knowledge Buffer*.

Note that the information in the *Knowledge Buffer* should be always up to date. This means we will have an overhead of keeping these latest knowledge. For example, to keep the full list of processes up to date, we will need to update the knowledge whenever there is a process exit or created.

4.6.4.4 VMI

The VMI component in Figure 4.5 is a stripped down version of LibVMI [128]. LibVMI is designed to run on Linux or MacOS, while the most used platform is on the Linux. In order to enable VMI operations with Xen Mini-OS, we have stripped down LibVMI to run with the system interfaces of the Mini-OS. The changes we have made are mainly the library dependencies. All the dynamic library dependencies in LibVMI are replaced with their static alternatives. Some library dependencies, such as GLib [170] we will introduce below, is too large to meet our minimization goal, so we manually implement a lightweight version of GLib as replacement.

[2]Rekall Forensics: `http://www.rekall-forensic.com/`

4.6.4.5 VMI Cache

VMI Cache is used to improve the performance of VMI operations by avoiding duplicate introspection operations into the physical memory. For example, to locate a virtual address for certain processes, or the kernel threads, users need to first traverse the kernel data structure to locate the raw page tables of a certain process, then traverse the page tables to translate the address by itself. This can be time consuming and resource hungry operations. In order to mitigate this, we use a cache structure to provide the latest recent translated, or queried addresses for fast access.

Note that both the *VMI Cache* and *Knowledge Buffer* store the introspected information to improve the performance. The difference is that *VMI Cache* aims to cache the raw memory information while the *Knowledge Buffer* is designed to store semantic views of the target. In addition, the *VMI Cache* is more frequently updated and easier to be out-dated than the semantic information stored in the *Knowledge Buffer*.

4.6.4.6 Event Channels

The module of *Event Channels* provides interfaces to register event channels and interfaces to create and execute the event handlers. The type of events is determined by the underlying processor architecture as well as the hypervisor. For example, on x86 architecture, the processor supports the monitoring event of CR3 register. We could register CR3 events so that any changes to the register will trigger the event handler registered by this module. Then the event handler will conduct further analysis on the event.

4.6.5 Implementation of the *Actuator*

Figure 4.6 shows an overview of the *Actuator* architecture. It serves as the interface for life-cycle management of guest VMs, such as *restart, create,* or *destroy*, where the guest VM could be a *Sensor* VM or target VM. *Actuator* has a more lightweight structure which benefits from its simple obligations in the system. It has dependencies to some system

Table 4.3: Code Base Reduction

	LibVMI	*EdgeVMI*	Reduction	LibVMI + Linux	*EdgeVMI* + Mini-OS	Reduction
LoC	31.0K	27.2K	-12.26%	**15M**	**54.7K**	-99.63%

Table 4.4: Binary Size Reduction

LibVMI + Linux	*EdgeVMI* + Mini-OS	Reduction
25GB	**7.2MB**	-99.97%

libraries for Mini-OS kernel interactions and networking, such as **newlib**, and **lwip**. It also needs the hypervisor interfaces to conduct the command to control the life cycle of a guest VM.

4.7 Security Evaluation

In our threat model, *EdgeVMI* is secured by trusted booting technologies and a trusted hypervisor provides basic isolation and authorization of different VMs on the edge server. Networking between cloud center and the edge server are protected by assumed secure internet protocols. Both the secure and insecure aspects of *EdgeVMI* are discussed in more details below.

4.7.1 Reduced Attack Surface

First, the administration operating system, such as Domain 0 on Xen, is out of our TCB. By design, *EdgeVMI* only directly interacts with Hypervisors for memory acquisition and events acquiring. It does not depends on any of the guest operating system, including the well known ‘privileged domain‘, such as Domain 0 in Xen. In this sense, our design significantly reduces the size of TCB.

Second, the unikernel in *EdgeVMI* is minimized to contain only necessary function-

Table 4.5: Changes to the Xen hypervisor and Mini-OS kernel

	Total LoC	LoC changed	Total Files	Files Changed
Xen Code Base	157.4k	495	7977	1(added) 6(modified)
Mini-OS Kernel Base	27.5K	49	227	3(modified)

alities for monitoring purposes. Table 4.3 shows code base size of both *EdgeVMI* and the traditional LibVMI system and Table 4.4 shows the final executable size. *EdgeVMI* itself has $27.2K$ lines of code, which is smaller than the LoC of LibVMI ($31.0K$), reduced by 12.26%. However, when combining them with the underlying operating system, we could see a more significant reduction of code base: from 15 million lines of code reduced to $54.7K$ lines of code, reduced by 99.64%. The final executable size are also reduced significantly by 99.97%.

Third, our code base changes to the Xen hypervisor and the original Mini-OS kernel code are minimal, as shown in Table 4.5. We only modified 6 files and added 1 file to the Xen code base; and modified 3 Mini-OS kernel files, a very small portion of the total code base. This avoids the possible errors being introduced into the kernels.

4.7.2 MAC Isolation Based on FLASK

In *EdgeVMI*, both the *Sensors* and *Actuators* are isolated by Mandatory Access Control. This is enforced by Xen Security Module via FLASK style policy. Permissions are only granted to the object when there are policies explicitly defined. Non-defined permissions will be rejected by default. This might limited the usability for the system because system administrators have to define the policies by themselves before deploying applications. However, we argue that in *EdgeVMI*, isolation policies between the guest VMs are relatively simple and thus easy to develop and deploy. This is because that the permission required by VMI operations are very clean and simple. It has a coarse grained object with a VM as a basic element, instead of individual files or programs. Therefore, comparing with existing FLASK permission management on a regular operating system, our security policies can be more effortless to develop.

4.7.3 Least Privilege and Privilege Separation

EdgeVMI protects itself by limiting each of its components with simple operations and least privileges. We have carefully chosen a stable set of permissions for each part of *EdgeVMI*. More specifically, *Actuator* can only operate on the life cycle of VMs, and no internal information of the VM are exposed to it. *Sensors* only have permissions to read from the raw physical memory of a VM, while no permissions that will change the state of VM are granted to them. With such a setup, other portion of our system could still be trustworthy even if any of the guest VM are compromised.

4.7.4 Constrained Information and Control Flow

Communications between the edge nodes and cloud center, and the modules inside *EdgeVMI* are all implemented with single directed flow of data and control. The constrained communication graph is hard-coded into the binary package and there no options to alter the information flow or control flow directions. Even if one node in the communication path is compromised, the information leak or damage caused will be constrained to be minimal.

4.8 Performance Evaluation

This section evaluates the runtime overhead of monitoring operations. We first highlight the ultra low memory consumption of *EdgeVMI*. Next we show the performance overhead of the VMI operations in the minimized operating system with both micro-benchmarks and macro-benchmarks. Finally, we show the performance benefits gained by running monitoring operations in a dedicated single-process VM rather than a traditional VM.

Table 4.6: Runtime Machine Memory Reduction

	LibVMI + Linux	*EdgeVMI* + Mini-OS	Reduction
Minmal Size	**1GB**	**32MB**	- 96.9%
Regular Size	**4GB**	**32MB**	-99.2%

The prototype of *EdgeVMI* is implemented on top of Xen Hypervisor. Domain 0 and Target Guest VM are running 64-bit Ubuntu. Hardware setups are as following: Intel Core i3-6100 @ 3.7GHz (Skylake) with 3M L3 Cache, 16GB DDR4 main memory. Domain 0 is given 10 GB memory, and Target VM is given 4 GB memory. *EdgeVMI* VMs are given 32 MB memory each. We follow the official Xen building environment setups and use GCC as our compiler.

4.8.1 Runtime Memory Overhead

The runtime memory consumption of *EdgeVMI* monitor VMs is measured based on their machine memory occupation which is usually acquired from the host machine before one VM starts. Table 4.6 shows the overall memory allocation by the monitor VMs of *EdgeVMI* and LibVMI. Both *Actuator* VM and *Sensor* VM of *EdgeVMI* is allocated with 32MB of memory while LibVMI has 4GB as regular size and 1GB as minimal size. Since LibVMI is running in Dom0 on Xen, which is 64-bit Linux OS, thus is allocated with 4GB memory to allow regular performance. The case of minimal 1GB memory for Dom0 would be a pretty slow experience for 64-bit OS even thought it is still executable. Therefore, we can conclude that *EdgeVMI* would occupy much less memory resources and will introduce much less memory footprint than traditional VMI techniques in a traditional OS.

4.8.2 Monitoring Events

Event monitoring is evaluated with our stress testing benchmark. We manually craft benchmarks to trigger events in the hardware and then catch the events in our *EdgeVMI*

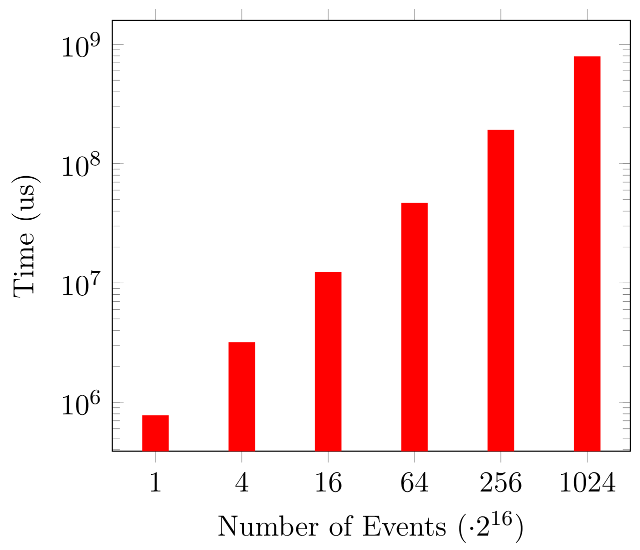


Figure 4.7: Time Cost With Different Number of Events

Sensor VMs. By triggering the events consecutively and we could see how fast an event could be handled in the monitor. Figure 4.7 and 4.8 show the scalability evaluation of event handling in *EdgeVMI*. From Figure 4.7, we can see that the absolute time being cost increases linearly when the number of events increases. This means linear scalability for the event trigger and return (after handled). From Figure 4.8, we can see that the average throughput of the event handling does not change much when the number of the events increases. Since the handler just do minimal work, the throughput reflects the maximum capability of *EdgeVMI* to handle events.

In brief, the time cost is proportional to the number of events captured in *EdgeVMI*. Throughput and latency is relatively stable under different number of events. Therefore, the event driven monitoring solution scales well and does not depends on the number of events generated in hardware. Note that during the experiment, the event handler is defined to be an empty function (with no workload) in order to show the plain overhead of the event trigger and return routine.

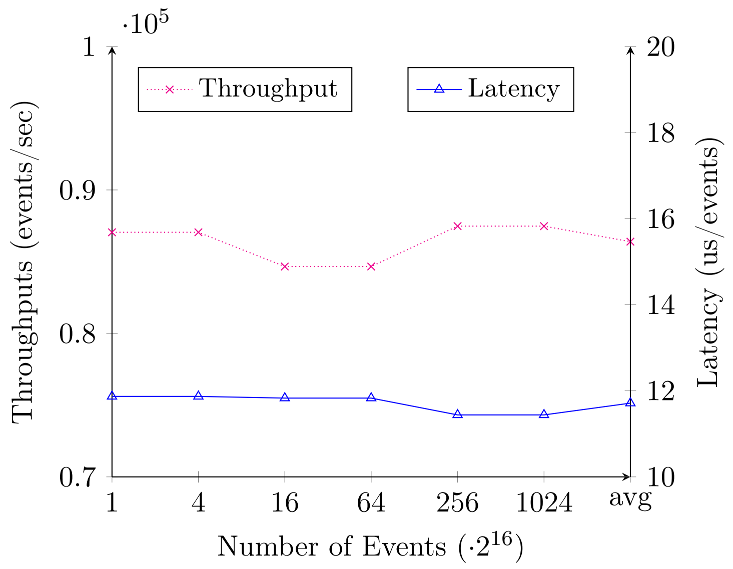


Figure 4.8: Average Throughput & Latency

4.8.3 Monitoring a Process

To evaluate the system with macro-benchmark, we have crafted a process monitoring benchmark which monitors the process creation activities and run LMBench in the target VM when the monitor is in execution. The process is monitored by registering an event handler on register CR3 on x86 machine. Whenever this register is written on the target guest VM, the CPU will generate events and the *Sensor* VM will be notified. The event handler will print information to the screen inside the *EdgeVMI Sensor*. Table 4.7 shows the performance of LMBench with an average of 2.7% overhead during the process monitoring is under execution.

4.8.4 Performance Benefits from Single-Process Mini-OS

By implementing our VMI operation in the Mini-OS, we could improve the efficiency of VMI operations. First, there is a double scheduling problem in the traditional hypervisor-based computing platform, where the hypervisor schedule time slots for the OS kernel and then the OS kernel schedules time slots for the processes running in it. This means that every time slot for the application needs at least two times of scheduling. However, our

Table 4.7: Overhead when Monitoring Single Process by CR3 Register Events

Test	Native	*EdgeVMI*	Overhead
Simple syscall	0.0386	0.0411	1.065
Simple read	0.0981	0.1005	1.024
Simple write	0.0682	0.0681	0.999
Simple stat	0.3257	0.3265	1.002
Simple fstat	0.0939	0.0928	0.988
Simple open/close	0.7853	0.7808	0.994
Select on 10 fd's	0.175	0.176	1.006
Select on 100 fd's	0.696	0.695	1.000
Select on 250 fd's	1.547	1.547	1.000
Select on 500 fd's	2.966	2.967	1.000
Select on 10 tcp fd's	0.196	0.199	1.015
Select on 100 tcp fd's	1.850	1.859	1.005
Select on 250 tcp fd's	4.611	4.613	1.000
Select on 500 tcp fd's	9.238	9.054	0.980
Signal handler installation	0.107	0.1075	1.005
Signal handler overhead	0.7595	0.7589	0.999
Protection fault	0.3283	0.3134	0.955
Pipe latency	15.4613	16.5223	1.069
AF_UNIX sock stream latency	14.0761	15.0909	1.072
Process fork+exit	69.2	87.3016	1.262
Process fork+execve	75.6757	85.7619	1.133
Average Overhead	-	-	1.027

monitor runs in a lightweight single-process VM. This means that there is no need for the OS kernel to do the scheduling. As long as the hypervisor allocates time slots for the monitor VM, the monitor functions could be executed directly without waiting for the OS to schedule it. This will potentially improve the performance of the monitor operations in our monitor VMs. Second, small memory footprints in our monitor VMs also could improve the architectural cache hit ratios and potentially improve the performance.

To verify our claim, we compare our monitor VMs with the traditional LibVMI framework. Figure 4.9 shows average time of reading one page when 500 consecutive virtual memory pages of the kernel process are traversed. On average, *EdgeVMI* costs 3.7 microseconds while LibVMI costs 5.7 microseconds introspecting one memory page. It includes the time traversing the entire page table to translate a virtual address to physical address. We can see the average time cost of *EdgeVMI* are 30% less than LibVMI.

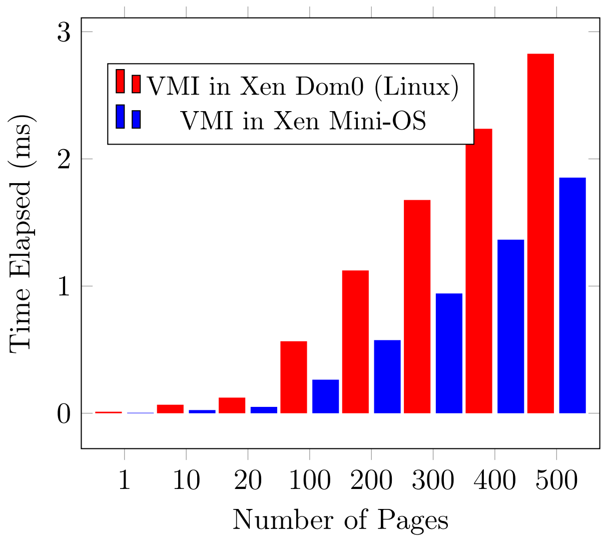


Figure 4.9: Time to Read Raw Memory Pages

4.9 Discussion

This section we discuss more challenges and possible improvements of *EdgeVMI* framework, wishing to spark more research ideas.

4.9.1 Hypervisor in TCB

The only piece of software we have to trust is the hypervisor. Currently the monitor VMs have been granted some privileges that originally are only designed for the hypervisor. But the hypervisor still holds the same privileges or even more privileges than the monitor. This is similar to the case where the traditional system auditing software residing inside the operating system gains some critical privileges that are originally only designed for the OS kernel, such as accesses to the entire file system, etc., but the OS kernel still holds all the privileges. Therefore, our monitor framework falls back to face the traditional 'same-layer' or 'peer' monitoring problem. This means a compromised hypervisor will render the monitoring operation not trustworthy.

However, the design of *EdgeVMI* can be extended by lowering itself one layer down so that it can directly talk with hardware just as the hypervisor do and effectively remove hypervisor from TCB. The feasibility have been shown in NoHype [96], where dedicated hardware resources can be pre-allocated at booting time and isolation is enforced during entire lifetime of virtual machines.

Furthermore, letting *EdgeVMI* directly talk with hypervisor can apparently improve the monitoring efficiency since it removes the indirection via hypercalls. In addition, with our static compilation and selectively composition of functional modules, unnecessary functionalities will not be included as they do in hypervisors, which will furthur improves its efficiency. We put this style of support to *EdgeVMI* as our future work.

4.9.2 Semantic Gap Challenge for VMI

The view of raw memory bits limits the visibility to the target VM's states, and reconstructing high level information from the raw binary streams can be difficult and will incur high overhead. This is well known as the problem of semantic gap [35]. Recent tools and frameworks such as Volatility [161] are able to discover high level semantic information but can incur high overhead, thus cannot be directly deployed for runtime monitoring on low performance edge servers.

Although *EdgeVMI* is designed to provide low level mechanisms to avoid the high performance overhead of reconstructing a rich semantic of processes, control centers in the cloud is still facing the semantic gap [35] problem. In order to make the intrusion detection more efficient and effective. Security policies need to be carefully designed according to the features of different kinds of memory corruption attacks, such that intrusion could be detected without a full reconstruction of the semantic view.

On the other hand, if more complicated semantic reconstruction is necessary, *EdgeVMI* could allow user to offload the semantic reconstruction task to the cloud. Apparently, this will cause another trade-off consideration between the reconstruction of raw binary memory and the amount of transferred data over the network. In general, more binary information are reconstructed, less data will be transferred while more performance overhead will be induced. How to make this trade off suitable for a certain edge server system is another challenge left to future work.

4.9.3 Orthogonality with Other Security Solutions

Framework of *EdgeVMI* is compatible with a various kinds of system security solutions to further improve the security of the system. *EdgeVMI* is assumed to be integrated with trusted booting [77, 112] as an necessary protection strategy. Additionally, hardware extensions such as Intel SGX [21, 145], RISC-V Sanctum [48], can be used to protect user applications under malicious OS. It is also possible to extend *EdgeVMI* security model where *Sensors* and *Actuator* can be protected when the hypervisor is not trusted [47].

EdgeVMI can also be extended to new hypervisor archtectures, such as Bareflank [19], which will make *EdgeVMI* support a broader spectrum of platforms. Emerging security solutions such as machine learning [73] could also help to learn and recognize the raw memory patterns for intrusion detection via the low level data collected by *Sensors* in *EdgeVMI*.

4.9.4 Weakness in Large Scale Monitoring

EdgeVMI can only protect the objects being monitored. Data that goes out of monitoring scope can not be protected. Users who need a large scale monitoring, such as the all the memory read and write operations across the entire application or operating system, would not be practical even though it is possible. This is due to the inherently high overhead of monitoring via VMI techniques, where the monitor operation happens at the lowest level, raw binary instructions and raw memory read writes, and the high volume of instruction and memory requests stream would quickly overwhelm the the speed of the semantic reconstruction progress.

4.10 Related Work

4.10.1 Security on Edge Servers

Security on the edge environment has been widely discussed in the recent literature [136, 152, 176]. The decentralized feature of edge servers poses great challenges to the security and privacy protection on the edge nodes. Shi [152] and Yi [176] discuss the missing of efficient tools to protect data privacy and security at the edge of the network. Roman [136] recognizes that most intrusion detection system has been designed for centralized computing instead of edge computing and fully autonomous security mechanisms are critical important for edge computing.

4.10.2 Virtual Machine Introspection

The idea of virtual machine introspection (VMI) was first introduced by Garfinkel and Rosenblum [74] as a way for intrusion detection. VMI has inspired many research works on the security enhancement of virtualization platforms, where an operating system is protected or checked in the hypervisor [36, 60, 92, 113, 148, 178]. Specifically, a system library, LibVMI [126] was proposed to simplify the development of VMI applications. The source code is freely available and further promotes forward the security solutions based on VMI techniques. However, it has to be installed inside a privileged host running Linux operating system, which has large TCB and memory footprints. This work ports it to a minimized operating system, which achieves a significant smaller TCB and footprints.

4.10.3 Privilege Separation on Monolithic Systems

Disaggregation of the monolithic system such as operating systems or hypervisors, such as μ-kernel [105], seL4 [98], Xoar [42], and Xen [115, 151], can improve the security by isolating different modules in the system. However, as a general purpose computing platform, those systems need to manage the communication between a large amount of isolated components. This introduces more performance overhead compared to their monolithic counterparts. In this work, we do not aim to build a general purpose computing platform, instead, we are building the system with a single purpose of security monitoring, such as intrusion detection, or service auditing. Therefore, the number of components of our system is relatively small and the communication between them can be made efficient.

4.10.4 Hardware Based Security Approaches

Hardware assisted security solutions moves root of trust into as lower level as possible, ideally in a specialized piece of hardware [34]. Then the burden of users' trust could be relieved from certain part of the system, especially from the vulnerable prone portion of a software stack or firmware. The effectiveness have been discussed in recent researches such as Scone [18], Haven [21], and VC3 [145], etc., where they eliminate the trust to OS by leveraging different hardware extensions such as Intel SGX [47]. This spectrum of solutions allow user to trust only his own application and the underlying piece of hardware. Similar solution can also be applied to the hypervisor such as in NoHype [96] which eliminated the trust on the hypervisor after the system securely booted with the help of hardware virtualization techniques.

However, hardware have limited logic budgets when facing complex security policies, which in turn delegates many security critical tasks back to software layers, resulting in a sophisticated multi-layer system stack for security tasks. In this project, we present our observation that such a complex chain of trust can be prohibitive for the emerging edge computing servers. *EdgeVMI* is our solution for trustworthy but less costly monitoring on such platforms.

4.11 Conclusion and Future Work

In conclusion, we propose *EdgeVMI*, a low level monitoring framework in a minimized library operating system to monitor a commercial operating system inside a VM on edge servers. *EdgeVMI* provides a rich set of low level mechanisms to monitor and control VMs via lightweight VMI technique. *EdgeVMI* is designed with minimized components. Each component is designed with least privileges. Strict isolations are enforced between different components via MAC policies. A prototype has been implemented and evaluation shows the efficiency and effectiveness of the monitor system. The VMI implementation in this work is open sourced at `https://tinyvmi.github.io/`.

Chapter 5

Capsule: Fine-grained Isolation and Pointer Safety with Ownership Tags

5.1 Overview

Isolation and pointer safety defense are two design strategies to build more secure computer systems. Isolation solutions can prevent illegal memory accesses across different protection domains, while pointer safety can prevent illegal memory accesses caused by type safety errors, usually inside the same protection domain. Many research efforts have been devoted to the architectural support for isolation and pointer safety, such as Intel SGX/MPK/MPX, ARM TrustZone/MPU, Capability Hardware Enhanced RISC Instructions (CHERI), etc. However, existing approaches either limit to a few number of security domains, or can only protect coarse grained memory, or support only one protection strategy (spatial/temporal pointer safety or isolation), or require tremendous porting efforts in order to protect legacy programs.

Therefore, this work proposes *Capsule*, a new architectural extension, along with the full stack toolchain and system supports, to enable both fine-grained isolation and pointer safety defense to protect legacy programs. It supports virtually unlimited number of protection domains that is proportional to the number of addresses in the address space. It

protects fine-grained memory at the granularity of instructions and data words. And more importantly, it requires minimal porting efforts to protect legacy programs. In a *Capsule*-enabled process, different protection domains can be isolated into multiple privilege-separated domains called ***capsule*s**. Each *capsule* is fingerprinted by a unique *ownership* identity, which is tagged to every instruction and data word that belongs to the *capsule*. The processor ensures that the memory in one *capsule* cannot be directly accessed by another *capsule*, thus guarantees the isolation between different protection domains. Pointers in each *capsule* are also protected with both temporal and spatial safety policies, which provides intra-domain security. A prototype system is implemented by extending a popular MIPS processor along with the corresponding LLVM-based toolchain. In addition, two tagged memory models are implemented and evaluated to explore its performance bottlenecks. Evaluations on FreeBSD with several benchmarks show that *Capsule* provides effortless backward compatibility and can successfully enforce inter-domain isolation as well as intra-domain pointer safety with moderate performance overhead.

5.2 Introduction

Programs written in unsafe languages such as C/C++ are prone to have memory safety issues [155], which makes the programs especially hard to be made trustworthy. This situation can be tracked back to the historical co-evolution of the C language and the hardware design back in the 1970s [38, 114, 135]. The ignorance of distinguishing different kinds of data in the memory from the processor's design has contributed to much of these error-prone consequences. To defend against these memory safety vulnerabilities, pointer safety and isolation are two kind of mainstream solutions.

Pointer safety defenses can protect memory object against illegal pointer dereferences caused by type safety errors at the language level [58, 78, 100, 116, 117, 118, 164, 171, 180]. However, existing pointer safety solutions require changes to the memory layout of the program to store metadata for the pointer and/or the object it points to, such as the bound information of pointers [116], or key and lock [117] on the pointer and the object. The change of the memory layout can result in compatibility issues with legacy code. In addition, from the performance side, fat-pointer based solutions are hard to be made efficient given no or limited hardware support. For example, the full pointer safety solution in BOGO [180] has 1.6x slow down on average, which is based on Intel MPX [90] with very limited hardware support (only 4 registers to store pointer metadata).

Second, domain isolation is an important line of defense against unknown memory vulnerabilities. This is achieved by applying the principle of privilege separation in a computer system [138]. A privilege separated system is partitioned into several isolated protection domains so that security breaches in one domain will not affect other domains. Intra-address space isolation is an important category of isolation that is widely used to create protection domains inside an address space. For example, CPU Rings [144], Intel SGX [47], Intel MPK [90], and ARM TrustZone [13, 16], ARM MPU [17], etc., are hardware extensions that support intra-address space isolation. They achieve isolation based on page table permissions, or permissions on other forms of memory region descriptors such as segment descriptors [56, 90] or ARM MPU registers [17]. However, these solutions suffer from one or more drawbacks below: (i) intensive manual changes to the legacy code are required in order to leverage these mechanisms; (ii) frequent domain switches on these isolation mechanisms will cause high overhead due to sophisticated context switches (e.g. 35x overhead for ECall/OCall in Intel SGX [181]); (iii) many of them only support a small number of isolation domains, such as 4 domains for x86 CPU Rings and 8 or 12 domains on ARM MPU [16]; (iv) the granularity of the isolation is still large, for instance, at page granularity on page table based mechanisms.

In contrast, pointer safety solutions can achieve finer granularity of protection which is at the object level, while with fewer manual changes to the legacy code, such as HardBound [58], SoftBound [116], and CHERI [164, 165, 168]. However, pointer safety solutions are originally designed for spatial or temporal pointer protections inside the same protection domain, and not for the isolation between different protection domains. This make it a good fit to avoid programming errors such as buffer overflows but unsuitable to isolate malicious code which may contain arbitrary instructions. One reason is that pointer metadata based permissions cannot not control non-pointer memory content such as most instructions and non-address-taken variables. This makes it not suitable to isolate a memory region that contains both pointer and non-pointer content. Even though CHERI compartmentalization shows the possibility of domain isolation by introducing the capability sealing mechanism [165, 168], its backward compatibility is mostly sacrificed: legacy control flow instructions are not allowed to cross domain boundaries and sealing and unsealing the program modules are mostly manually done. Therefore, the engineering effort to partition legacy programs with CHERI is non-negligible, where the legacy code needs to be the heavily retrofitted in order to leverage the new library interfaces for isolation (or compartmentalization).

To mitigate above challenges, we explore a new tagged architecture where each small chunk (e.g. 4 bytes) of memory are distinguished by an ownership tag. Our ownership tags not only can be used to enhance pointer safety, but also be used to isolate the address space into different protection domains. More importantly, switching protection domains can be done efficiently with no heavy context switches and safely via legacy control flow instructions. And furthermore, by incorporating the new design of the underlying hardware and software stack, our system maintains good backward compatibility and removes most the retrofitting efforts of the legacy code.

Contributions of this work are listed as below:

1. We have proposed a method for backward-compatible fine-grained intra-address space isolation at the granularity of instructions and data words. In this method, program modules can be encapsulated in almost any size with no changes to the legacy memory layout: A protection domain can be as small as only containing a single variable along with a few instructions, and each piece of memory for this domain can be allocated anywhere in the address space without having to be continuous.

2. We have proposed a new model of execution entity, *capsule*, where each *capsule* (as a

protection domain) is defined by a set of data words along with a set of instructions that can access the data. The runtime memory and execution context of the *capsule* is constrained only by the unique ownership tag on each data word and instruction, without using the page table permissions, privilege levels, etc. More importantly, the privileges that each *capsule* holds are also determined only by its set of instructions and data, which eases both the static analysis as well as dynamic enforcement of privileges for each protection domain (Section 5.4).

3. To support both inter-domain isolation and intra-domain pointer safety, we have designed three ownership tag representations. Cross-domain isolation can be guaranteed by ownership identities stored in the ownership tags, while inter-domain pointer safety can be achieved by the pointer metadata stored in the ownership tags for pointers. (Section 5.4.3).

4. To avoid heavy weight cross-domain call (such as *syscall* style calls), we have designed two domain transition modes between *capsule*s: *transient transition* without context switches and *sanitized transition* that can avoid the overhead of heavy memory context switch and have no dependence on a supervised trusted stack (as used in user/kernel transition in Unix). The new design helps to improve cross-domain efficiency as well as to eliminate the single point failure of the trusted stack (Section 5.5).

5. We have proposed an automatic isolation method to determine the boundary of *capsule*s by combining static ownership analysis and dynamic ownership propagation. Given the hints to the separated modules of a program, our compiler will reason about the ownership tags for every instructions and memory objects during compilation. During execution, the processor will propagate ownership to non-tagged memory content based on our *memory colonization* rules (Section 5.6).

6. We have designed a full stack system extension that supports the notion of *owner-*

ship tags, including the hardware processor, memory hierarchy, compiler toolchain, operating system, and user applications. Processor pipeline is extended to process the *ownership tags* to make sure that the *ownership* of the instruction matches the *ownership* of the memory it accesses. The entire memory hierarchy, including caches, main memory, as well as the binary executable files, are seamlessly extended with *ownership* tags (Section 5.7). The compiler toolchain and operating system are extended to generate and load executable files with ownership tags. Finally, legacy code in user applications can also be annotated as hints for security critical data or codes in a program.

7. A full-stack prototype system is built by extending a popular open sourced MIPS architecture (Section 5.8). We have built emulators on QEMU running FreeBSD for functional evaluation. We have built two memory models on Gem5 simulator to evaluate the performance impact. We have also extended a Verilog implementation of MIPS R4000 processor to evaluate the hardware feasibility. Evaluation shows that *Capsule* can effectively enforce fine-grained isolation as well as pointer safety for legacy programs, with negligible porting efforts posed to developers (Section 5.9). The performance of our tagged memory implementation is evaluated with overhead of 0.51x to 0.74x in adapting legacy programs to run in *capsule*s (Section 5.10).

Section 5.3 introduces our motivation. Section 5.4 presents an overview of the system design. Section 5.6 discusses our methods of how to determine the boundary of *capsule*s by static ownership analysis and dynamic ownership propagation. Section 5.5 provides different modes of *capsule*s interactions via different domain transition gates and memory sharing mechanisms. Section 5.7 introduces our design of the system stack to enable ownership tags by extending the processor architecture as well as the corresponding toolchain. Section 5.8 discusses the implementation details of our prototype system. Section 5.9 evaluates the security properties of our design. Section 5.10 evaluates the feasibility and the performance of our design. Section 5.11 discusses the related work. Section 5.12 concludes the project.

5.3 Motivation

In this section we will discuss our motivations with more details from the following aspects: 1) Why we need a new isolation mechanism? 2) Why we want fine-grained isolation? 3) Why we choose tagged memory?

5.3.1 Why a New Isolation Mechanism?

Privilege separation have long been praised as one of the fundamental principles to improve system security since 1970s [138]. It is an effective strategy to defense against unknown bugs in a system, both in the operating system design and in the applications' design. Isolation is an important technique to implement the privilege separation for a software system. This is usually achieved by separating a program into different function modules, each representing a security domain. Each domain is then assigned with the least privileges to do its tasks and communications between different domains are strictly checked. In a privilege separated system, security breaches in one domain will not affect other domains. Isolation techniques can be generally classified into two categories based on their scope of isolation: inter-address space isolation and intra-address space isolation.

5.3.1.1 Inter- and Intra-address Space Isolation

Inter-address space isolation prevents accesses from one address space to another space. They are build upon process-derived memory model, such as user processes, Linux Containers, as well as virtual machines. However, inter-address space isolation has a relatively large granularity for isolation, i.e. one address space at its smallest. Security breaches where vulnerabilities and critical data resides in the same address space cannot be contained by these techniques.

In contrast, intra-address space isolation prevents access between two domains inside the same address space, such as segmentation, capabilities, as well as page table permissions, etc. Different with the inter-address space isolation, intra-address space isolation does not need to change to a different address space during the domain switch. Therefore, it has much better performance than the inter-address space isolation. In addition, many emerging low-end IoT devices, such as ARM Cortex-M architectures, do not support virtual memory management and thus cannot implement inter-address space isolation efficiently. Therefore, intra-address space isolation becomes an important line of defense for such small devices.

5.3.1.2 Drawbacks in Existing Intra-address Space Isolation

Compiler and OS based isolation mechanisms can enable intra-address space isolation, such as SFI [159], SVA [51], Nested Kernel [54], Wedge [23], etc. But the number of security domains is still rather limited. More importantly, pure software solutions are easy to be affected by programming bugs in the large software code base when the system is lack of hardware assistance for security. Futhermore, pure software solutions are prone to introduce high overhead without hardware accelaration.

To mitigate this, several hardware assisted intra-address space isolation mechanisms have been built to provide a higher level of trust while reduce the runtime overhead, such as CPU Rings [144], Intel SGX, Intel MPK [90], ARM TrustZone [13], ARM MPU [17], etc. However, most of them suffer one or more drawbacks below, which prevents them from being widely deployed efficiently in practice. First, many of them only support a small number of isolated domains. For example, traditional CPU Rings only support up to 4 rings (on x86) and usually 2 of them are used, ARM MPU [17] only supports 8 or 12 domains, and Intel MPK [90] only supports 16 domains. These make them unsuitable to be used where many more domains are needed in order to contain possible vulnerabilities into its smallest domain [33, 133, 173]. Second, most of them require intensive manual changes to the source code. One reason is that they all rely on the page-table permission mechanisms. The configuration of page-table permissions for the memory requires deep understanding of the application logic and intensive manual changes to the source code, which makes it hard to be automated. For example, enabling Intel SGX enclaves in a system [18, 47, 145, 162] can be rather complicated even though they provide new library and system interfaces for the developers to adopt. Finally, in all these architectural supports, the granularity of these isolation mechanisms is still large, for instance, at page granularity in its smallest. This makes them unsuitable to isolate different memory objects that live on the same page.

5.3.1.3 Potentials in Pointer Safety Solutions

More flexible granularity at object level can be achived based on pointer safety solutions, such as HardBound [58], SoftBound [116], Intel MPX [124, 180], and CHERI [164, 165, 168]. The adoption of legacy programs for such solutions can be potentially made easy by adding compilation toolchain support along with none or minimal hardware extensions.

Unfortunately, fat-pointer solutions are mostly designed for spatial or temporal pointer safety, and are not easy to extend them to support isolation of protection domains [165, 168]. One reason is that fat-pointer based permissions cannot not control non-pointer memory content and thus cannot be directly used to isolate a memory region which contain both pointer- and non-pointer-derived content. Therefore, special mechanisms need to be designed in order to support isolation via fat-pointers. For example, CHERI [164, 165, 168, 171] extends their pointer safety defense (with memory capability) to support isolation (with object capability). It is achieved by introducing the capability sealing mechanism for object capabilities. However, in CHERI, legacy control flow instructions are not allowed to cross domain boundaries. Developers must manually sealing and unsealing the specific program modules with their new instructions or the new library interfaces for isolation. Therefore, its backward compatibility is mostly sacrificed and the engineering effort to port legacy programs is non-negligible, where the legacy code needs to be the heavily retrofitted in order to leverage the new mechanisms for isolation.

In addition, exiting pointer safety solutions require changes to the memory layout, especially for the representation of pointers in the program, which results in compatibility issues with legacy code. Furthermore, from the performance side, fat-pointer solutions tend to incur high runtime overhead due to limitd hardware support. For example, Intel MPX has 1.6x slow down on average as shown in BOGO [180] for full pointer safety, where only 4 hardware registers are available to store bound information.

5.3.1.4 Towards Inter- and Intra-Domain Memory Safety

Combing all above (5.3.1.1 - 5.3.1.3), we aim to design a system that can divide an address space into any number of privilege separated domains as needed with fine-grained isolation at the granularity of instructions and data words. It should handle both pointer and non-pointer memory contents and provide both inter-domain isolation as well as intra-domain pointer safety. More importantly, it should keep the human refactoring of legacy programs to be minimal by moving more responsibility onto the architecture and compiler toolchain.

5.3.2 Why Fine-grained Isolation?

```
# prologue
daddiu $sp, $sp, −48 # allocate stack frame
sd $ra, 40($sp) # save return address
...
# function body... # buggy or malicious
# epilogue
...
ld $ra, 40($sp) # restore return address
daddiu $sp, $sp, 48 # free stack frame
```

Figure 5.1: Return Address Save and Restore

Fine-grained isolation can constrain a software vulnerability in its minimal scope. Memory safety bugs in a program can live anywhere in the address space and affect arbitrary places in the same address space if not well isolated. On the other hand, the instructions and variables involved in the memory safety violations can occupy a variable size of memory with one or more memory locations, either contiguously or sparsely placed. Therefore, it is necessary to be able to protect the memory at its finest granularity, as well as to protect memory object at different sizes with different layouts.

Listing 5.1 shows our motivated example for fine-grained isolation in MIPS assembly. Upon a (non-tail) call into the callee function, the return address will be saved to the stack (line 3) in the function prologue (preparation code before the function body starts). Then at the function epilogue (restoring code after the function body ends), the return address must be restored from the stack (line 8) in order to return to the caller. It is well known that the return address being saved and restored on the stack needs to be protected from corruption. Otherwise, attackers might leverage the corrupted return address to conduct control-flow hijacking attacks such as Return Oriented Programming (ROP) [31, 32, 43, 55, 75], signal oriented programming (SOP) [25], etc. To protect the return address, we could separate the return address from the rest of the application code.

However, existing isolation mechanisms are mostly coarse-grained and will be inefficient to protect small objects such as the return addresses on each stack frame. For example, shadow stack [30] can be used to protect the return addresses. One might use traditional privilege levels (or CPU Rings) to isolate the shadow stack [160], but it incurs high overhead due to the frequent domain crossings between the shadow stack and the regular stack. Efficient shadow stack [182] can be implemented on ARM processors [16, 17] by using a special set of instructions (unprivileged load and stores) to access regular memory and another set of instructions (privileged loads and stores) to access the shadow stack. But since only two sets of instructions (privileged and unprivileged) are available for use, this isolation technique on ARM only supports two isolation domains.

Therefore, we need a new fine-grained isolation mechanisms to protect small objects. We aim to make it efficient to execute without heavy context switch for domain crossing. We aim to support unlimited number of security domains so that we can protect a various kinds of different memory objects or regions.

5.3.3 Why Choose Tagged Memory

One of the major reasons for current memory safety issues is inherited from the unawareness of security context in hardware. Under von Neumann architecture, processor treats raw memory as plain integers, and it is the programmer's responsibility to direct the processor how to interpret the memory bits. This free programming style assumed programmers understood the machines very well and it was their responsibility to avoid errors. This has enabled advanced programmers to write efficient code in the past. However, with both the hardware and the software systems are getting more complicated, it becomes unrealistic by relying solely on programmers themselves to avoid errors. New safe languages have been seen to put restrictions on the free capability of programming, but at cost of the performance with the lack of the underlying hardware support.

On the other hand, new hardware extensions, such as tagged architectures in the 1970s [70, 71, 169], and the state-of-the-art CHERI [164, 168, 171], have been proposed to bind semantic meanings to memory in order to distinguish different memory contents, such as pointers and non-pointers, type information, etc. However, these designs either heavily rely on a certain architecture or require heavy retrofitting efforts to the legacy programs. For example, CHERI extension distinguishes pointer and non-pointer with a 1-bit tag for every 256-bit memory, but it extends the pointer representation from 64-bit to 256-bit or 128-bit [164, 168, 171]. This could introduces incompatibility with legacy programs, especially when the capability-enabled application are linked with non-capability-enabled libraries.

Therefore, this work seek for an alternative implementation of the tagged memory with better backward compatibility. It should maintain backward compatibility without changing the legacy memory layout (i.e., pointers will be still 64-bit in our system). This will in turn make the adaption efforts of legacy programs to be minimal.

5.4 *Capsule* System Overview

This section first introduces our key insights for this work, then discuss the threat model with the scope of the work. Next, it presents an overview of *capsule*, the *abstract ownership*, and different categories of *capsule*s according to the private data it protects. Next, we discuss the extension of the system stack to enforce *ownership* rules. Lastly, we explain our strategy of determining the closure set for a *capsule* by combining static and dynamic information.

5.4.1 Key Observations

This work is inspired by our several key observations listed below where we have found many new security features that can be enabled by leveraging the ubiquitous tags as the ownership for every instructions and data words in the memory:

5.4.1.1 Efficient Domain Transition with Ubiquitous Ownership Tags

With ubiquitous ownership tags in memory, more efficient and secure transition gate can be designed comparing to traditional system call gates. We observe that after we apply ubiquitous ownership for each memory chunk, memory access privileges are constrained by these ownership tags and there will be no need to switch a complex context for a protection domain, including the stack pointers, page table views or page table permissions, etc. In addition, the communication between protection domains can be made more efficient by manipulating ownership tags instead of copying the actual data as used in system call gates. Therefore, more efficient transition gates can be designed with ubiquitous ownership tags in memory. Similar observations can be found in Intel MPK deployment [157], where different page table entries are given a 4-bit tag and domain transition can be as easy as updating a register (PKRU). We can leverage efficient ownership mechanisms for data sharing and the need of context switching of the processor states can be removed. These gates also allow mutually distrusted domains switches without a centralized supervisor in the OS (Section 5.5.2), which again improves performance and also avoids the risks of single point failure where a trusted stack must be used.

5.4.1.2 Partition Process Memory into Closure Sets

Data and instructions in the same address space can be grouped into two or more disjoint closure sets. A closure set here is defined as a set of instructions and data words which meets two requirements: **(1)** *All data accessed by an instruction in the set is in the set,* and **(2)** *All instructions that access a piece of data in the set are in the same set.* For example, in the traditional process model, data and code can be divided into two disjoint closure sets: one set running in kernel mode and another set running in user mode. All memory accessed by a user instruction will always have the user's permission. On the other hand, all instructions access the user's data can be restricted to be user-only instructions. Although by default all instructions in kernel mode can access user memory, this can be restricted such that the kernel will never directly access user's memory. This has been shown possible in SVA [51] and Virtual Ghost [50], where a sanitization layer is added to avoid any direct kernel access of the protected user data. Inspired by the above observations and related works, we thus aim to craft out more isolated domains based on closure sets of instructions and data.

5.4.1.3 Partition Closure Set with Transition Gates

One *closure set* of instruction and data can be partitioned into two *closure set*s by going through a special transition gate for sanitizing the communication channel. To achieve fine-grained isolation, we might need to separate one large *closure set* into two or more smaller ones, while keeping necessary communications between the separated sets.

For example, if we partitioned a set into two *closure sets*, each with different functions and their data. But a pointer-derived parameter is passed (e.g. via a call instruction) from one set to another, the pointer being passed across the boundary will probably cause data leakage or break the functionality of the original closure set before separation. So the transition between the two new sets would require special operations to cross the boundary between them. In such cases, a special transition gate can be used so that the chunk of memory being passed via pointer parameters can be made available from one set to another set, but without the authorization to both sets at the same time.

5.4.1.4 Closure Sets as Domains

Based on above observations, we define each closure set as an isolated domain, which we call *capsule*. Each *capsule* holds least privilege that operates on a fixed set of memory defined by the closure set. In this way, memory access errors in one *capsule* will not affect other *capsule*s in the process. *Capsule* can be used as general isolation mechanisms for different program modules. It can also be used to implement lock and key mechanisms to protect heap objects from temporal pointer safety vulnerabilities. Furthermore, spatial safety inside the same domain can also be defensed by the fat-pointer style ownership tags for pointer variables.

5.4.1.5 Achieve Intra-domain Security with Ownership Tags for Pointers

By defining special interpretation for ownership tags of pointers, we can implement fat-pointer like solutions to improve in-domain security. Since we will check the ownership of instruction against the ownership tag of the memory object it accesses, thus the ownership tag for the pointer that points to this object is ignored during this check. Therefore, we could store object metadata in the ownership tag of pointers to improve in-domain security. For example, by storing object size information in the tags, we could implement spatial pointer safety defense similar to these in the traditional fat-pointer approaches.

5.4.1.6 Lock and Key with Ownership Tags

Lock and key mechanisms for temporal pointer safety defense can also be achieved by encapsulating the heap objects in its own scope using *capsule*s. Lock and key mechanisms assigns a key to the subject (such as a pointer or instruction) and a lock on the object (such as a heap object). When the subject tries to access an object, it will only success when the key matches the lock. With our ownership tags, the key can be implemented as the ownership tag for an instruction, and the lock can be implemented as the tag for the heap object. Therefore, the temporary safety defense could be implemented using *capsule*s. For example, we might encapsulate 1) the code at the allocation site, 2) the code that accesses the heap object, 3) the heap object, and 4) the code at free site all together into one *capsule*. Then, if the object is freed and the memory is then re-allocated to other portion of the program (here will be another *capsule*), then the old *capsule* has no legal ownership to access that memory again, even if it could still hold a dangling pointer pointing to that location.

5.4.2 Threat Model

In brief, we aims to provide a security solution that could combine domain isolation and pointer safety in one hardware extension. By fine-grained isolation, software vulnerabilities or malicious code will be contained in a minimized security domain and will not affect other security domains, thus achieving inter-domain security. By pointer safety defense, pointer vulnerabilities in one domain could be prevented from being exploited, thus achieving intra-domain security.

In this work, the software vulnerabilities are programming errors that will cause information leakage or data corruption that does not reflect the original programming intention of developers. For example, the vulnerability can be caused by triggering undefined behaviors in a low level programming language, such as buffer overflows, use-after-free, in C programming language. Or it can be caused by malicious code from third party programs.

For example, a third-party library that have backdoors to steal or corrupt data that should not be accessed by the library based on the semantics of the program source code.

We will assume the processor is correctly implemented to follow our *ownership* requirements (see 5.7). We assume code sections are readonly and cannot be modified by the attacker. We also assume the program loading process behaves correctly during program start. We also assume the compiler is trusted in this work. Removing compiler from trusted computing base (as in SVA [51]) can be done by using a simple type checker to ensure correct isolation between *capsule*s, but this is out of the scope of this work.

5.4.3 *Capsule* Overview

The idea of *capsule* is largely borrowed from the program encapsulation feature of object oriented programming languages, where each object of a certain type (or class) is encapsulated with its own data and code and other type of object cannot access its private data except going through explicitly defined interfaces. However these high level language features will not hold at machine code level, thus can be vulnerable if any module in the program is written in unsafe language or even assembly code. In this chapter, we seek to enable program encapsulation in non-OOP languages such as C programming language by a software-hardware co-design approach which can provide a stronger guarantee of isolation as well as pointer safety in machine code level.

We use *capsule* to name a security domain which consists of a set of instructions and data words with same security requirements (thus same set of privileges). A process contain several *capsule*s and each of them represents one security domain inside the process.

Figure 5.2 shows the memory layout of runtime *capsule*s in the system memory hierarchy. The figure shows two *capsule*s in a runtime system. We use different color to represent different ownership identities. Memory content of each *capsule* is tagged with a unique color, one black and one red. Processor caches are extended with ownership tags for cached data/instructions, such that processor's data access and instruction fetch from the caches will always have the ownership tag ready at the same transaction.

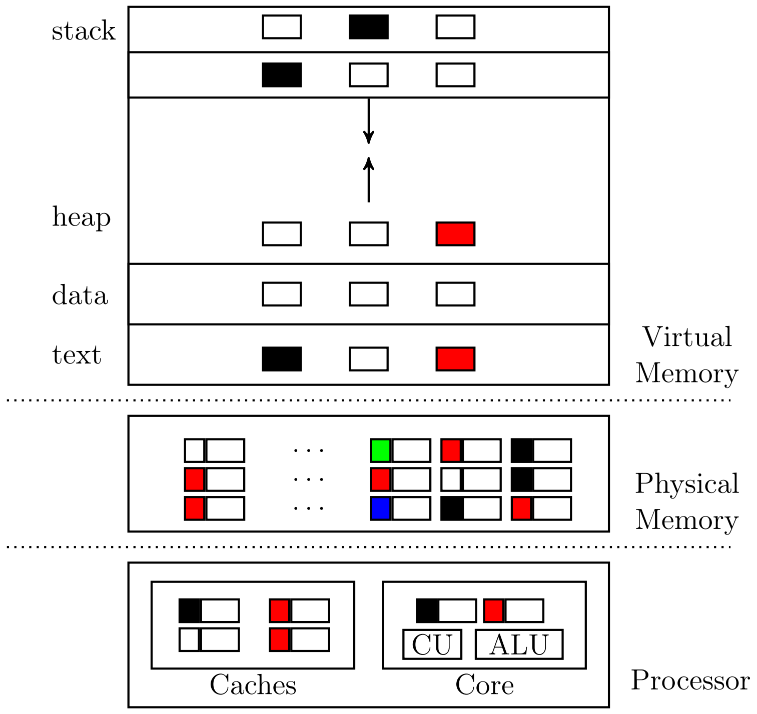


Figure 5.2: Overview of *Capsule*s in the Memory Hierarchy

Note that white colored regions represent the memory that do not belong to any *capsule* and thus are regarded not protected (or shared by all parties). However, white regions might be tagged during execution upon any *capsule*'s access. This is one of the dynamic ownership propagation rules in our processor to mitigate the weakness of static analysis capability. We will discuss this with more details in Section 5.6.

We can see that memory belongs to one *capsule* in a process can be sparsely distributed in a process's address space, i.e., they do not need to be contiguous. This design allows the legacy memory layout of the process is kept unchanged, which helps to maintain the backward compatibility with the legacy programs. This is one of the main benefits of using ownership tags at the granularity of instructions and data words.

The ownership tags of virtual memory is implemented in physical memory. As shown in Figure 5.2, entire physical memory is divided into small chunks, and each chunk contains a colored tag and the original memory for the program. The tag access is transparent for the program, which means tags are not directly addressable in a program. Instead, the processor is responsible to manage the access of tags. For example, on an 64-bit machine, if the byte address of a 64-bit word in the program is x, then processor will transparently convert this physical address to $2x$ as the program word's new memory address, and $2x+8$ as the address of the tag data for this word.

Ownership IDs in the physical memory is cached in the processor cache hierarchy, such as (L1) instruction cache, data cache, and L2 cache. This allows the processor to simultaneously access the ownership IDs along with regular instructions or data words. In addition, similar to the tag cache in CHERI [93], our processor also have a dedicated last level cache for ownership tags, which helps to speed up the tag access. Regarding the number of ownership bits for each 32-bit available memory, we expect that different security policies might need different bits of ownership tags. Therefore, we make our system to be configurable from 1 bit to 32 bits in the hardware design stage. (However, once the processor is built, the bit width will be fixed.)

5.4.4 Ownership Tag Representations and Interpretations

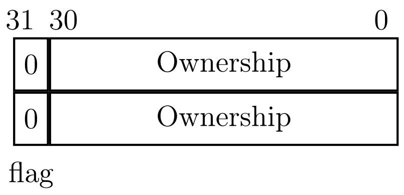


Figure 5.3: Tags for Non-Pointer (32-bit tag per 32-bit memory)

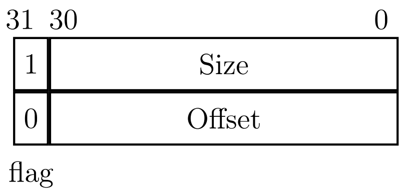


Figure 5.4: Tags for a Data Pointer (64-bit tag per 64-bit pointer)

Inside the address space, the memory content can be generally classified as non-pointers and pointers. Non-pointers are memory content that are not interpreted as addresses.

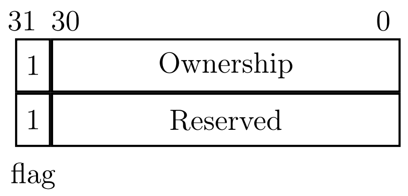


Figure 5.5: Tags for a Code Pointer (64-bit tag per 64-bit pointer)

Pointers are a special kind of data that store memory addresses. These addresses are used by the program to access different memory locations inside the address space. The pointer that refer to the location of a code snippet or an instruction is called a code pointer (or function pointer). The pointer that refer to regular non-executable memory content is called a data pointer. Distinguishing pointers and non-pointers, as well as data pointers and code pointers in the memory, plays an important role in many strategies to defend against memory safety problems [116, 155, 168].

Capsule system distinguishes them utilizing a flag on ownership tags, as shown in Figure 5.3 - 5.5. Our prototype system implements a 32-bit tag for every 32-bit memory content, in order to comply with the 32-bit instruction width in MIPS64 architecture. However, all tags are accessed on 64-bit alignment, which complies with the 64-bit architecture. During interpretation, tags for pointers (64-bit width) are interpreted as a 64-bit bundled tag, and tags for non-pointers are interpreted as two 32-bit tags individually. In order to achieve this, we use a 2-bit flag in the 64-bit bundled tag as shown in Figure 5.3 - 5.5.

5.4.4.1 Interpretation of Non-Pointer Tags

If a 64-bit application memory chunk is not a pointer, we call the tag for this chunk of memory a non-pointer tag. As shown in Figure 5.3, Non-pointer tag has the flag bits of *00*. Each 32-bit data word has a 31-bit ownership identity stored in its tag. The two chunks of 32-bit non-pointer memory are allowed to have different ownership identities, i.e. they can belong to different *capsule*s. Instructions accessing this chunk of memory must also have the same ownership identity.

5.4.4.2 Interpretation of Data Pointer Tags

Flag of *10* means this is a bundled 64-bit tag for a data pointer, as shown in Figure 5.4. This tag contains the size and offset information of the 64-bit pointer. As discussed in our insights, non-executable memory referred by data pointers are already protected via ownership tags on the instruction and the memory object, where the processor will ensure that their ownership tags will match. Therefore, data pointers do not need to be protected. They might be overwritten to arbitrary value, but the data access instruction will still fail if the ownership of the instruction does not match the ownership of the memory object. However, pointers are prone to cause spatial memory safety issues, such as buffer overflows. Therefore, here we incorporate pointer's bound information along with it and the processor will check the bound during runtime. This is similar to the spatial safety defense in the fat pointer approaches [78, 100, 116, 118].

5.4.4.3 Interpretation for Code Pointer Tags

Figure 5.5 shows a 64-bit tag for a code pointer flagged by *11*. The bundled tag contains an ownership identity and the rest 31 bits are reserved. A code pointer represents an address of an instruction that the processor can branch to. Different with data pointers, a corrupted code pointer can diverge control flow to illegal code snippets. Calling into illegal code snippets might trick the one *capsule* to share its private memory to the callee *capsule*. Considering this, we assign ownership identity to the code pointer so that the code pointer can be protected as private data in a *capsule*. This also allows the code pointers themselves being encapsulated in a different *capsule* with the instructions being pointed to. For example, we can assign unique ownership only to the return addresses of a function in order to defend against return oriented programming (ROP) attacks, which we will discuss in Section 5.5.

5.4.5 Ownership Space

Each ownership identity is an integer value. We maintain a flat ownership space where every ownership is treated as a unique entity identity with no hierarchy relationships as the type inheritance in the OOP language, nor the master-slave relationship as in user-kernel domain separation. This simplification allow us to build more efficient and highly scalable program partition, where we can automatically maintain the consistency of the ownership assignment across the system as well as using concealment of the ownership IDs to enhance the security.

5.4.6 Ownership Allocation and Relocation

In a *capsule*-enabled system, ownership IDs can be managed by several parties, such as programmers, compilers, the linker, loader, language libraries, the operating system, as well as the processor. Each of them might have the authority to determine the ownership ID for certain memory objects, either statically (during compilation) or dynamically (during binary loading). Therefore, it is important to maintain the consistency of ownership space between these parties and avoid potential conflicts with previously allocated ownership IDs.

To simplify the problem, we maintain a flat ownership space across the system and provide mechanisms to reuse ownership IDs as well as resolve collisions via ownership relocation. The problem is very similar to manage the addresses of different objects in a process's address space. With our relocation mechanism, each party could allocate ownership IDs in the entire ownership space and any ownership conflicts between different memory objects will be resolved via relocation. The implementation details can be found in Section 5.8.

5.4.7 Ownership Identity Concealment

Forged ownership can be dangerous. For example, if *capsule* identified as A forges the ownership of another *capsule* B, and reset its own code's ownership from A to B. Then the *capsule* A's code will be executed with full access to *capsule* B's memory. Therefore, as security critical resources, the ownership ID of one *capsule* should not be directly updated by other *capsule*s, nor by itself.

To defend forged ownership attacks, we propose the concealment of the ownership IDs by combining several approaches: First, programmers are not allowed to use ownership manipulation instructions that can set the ownership of variables or instructions. This will ensure that the regular *capsule*s contains application code and data will not be able to directly manipulate ownership tags during execution, including the ownership of itself (Note that developers still write application code that assign static ownership via annotations). Second, we maintain a flat ownership space and compiler tool chain can randomize the ownership at a various of stages: compilation, linking, as well as program loading: all the ownership IDs being assigned can be randomized via ownership relocation such that the integer values of ownership identities can all be randomized without changing the original separation semantics. Third, we do not provide ISA interfaces for the programmer to get the ownership ID of the memory. Therefore, an malicious *capsule* has no way to predict the concrete value of the ownership IDs, even of its own.

5.5 Secure Domain Transition

As discussed in our insights, as long as every part of memory is statically tagged with a certain ownership, and the system ensures that only the instruction with the same ownership as the memory can have access, the domain transition between *capsule*s can be as simple as just updating the program counter without any context switch of CPU states or memory regions. However, we find that tagging every instruction and reasoning about the ownership for all data words and instructions is not practical due to many corner cases that the static analysis cannot handle, such as the difficulty of pointer-to analysis for pointer intensive applications. In addition, because the requirements of some applications, the ownership being assigned to certain data might need to be changed during runtime in order for the data to be shared between different *capsule*s.

Therefore, we define domain transition in two forms: *transient transition* and *sanitized transition*. *Transient transition* happens between *capsule*s where the ownership IDs of each piece of data and instructions in the *capsule*s are fixed without any changes during the lifetime of the process. Apparently, for such *capsule*s, domain switch does not need to save or restore the CPU context, nor updating any memory mappings. This category of *capsule* will cause almost no overhead during domain switch.

Sanitized transition happens when memory sharing between the caller and callee are required. For example, a memory buffer being passed across the boundary of two *capsule*. In this case, a special sanitizing gate will come to intervene to make sure that only the data being explicitly passed following the semantic of the source code will be accessed by the callee *capsule*. We now discuss the two transition modes in more details in the following sections.

5.5.1 Transient Transition

After the address of the first instruction in the callee *capsule* is loaded into the program counter and the *ownership* tag of the instruction is loaded into processor simultaneously, a transient domain transition is done. *Transient transition* does not go through a call gate, thus it can achieve fast domain crossing without context switching. No *ownership* tag will be changed or updated during this switch. As we discussed before, this only works with *capsule*s that have no need to change the ownership of any private memory during control transfer.

For example, to protect the return address on the stack, we can create a *capsule* to just encapsulate two instructions and one memory location: the return address on the stack. This is possible because each return address on the stack is supposed to be accessed by only two instructions: one instruction to push it on the stack, another is load it from stack and put it to the program counter register. It is easy to see that the ownership of both the data and instructions are fixed and there is no need to change them (except for the initialization and reset). Therefore, a regular program counter update will be enough to transfer in or out this *capsule*, and the processor will ensure that as long as the data is valid (not being cleared), no other instructions will be able to change it.

5.5.2 Sanitized Transition

Sanitized transition between *capsule*s will go through a special call gate to handle cases where a private data of the caller *capsule* needs to be shared with the callee *capsule*, or in the reversal direction of sharing. A call gate is designed for sanitizing the parameters being passed over. Sanitizing process might include ownership transfer of the memory region being passed and control flow transfer from a caller *capsule* to a callee *capsule*. The call gate will update the *ownership* or duplicate the data being passed to the callee *capsule*. This will avoid the possible data breaches due to undefined behaviors in the code, thus make sure that only the data being explicitly passed following the semantic of the source code will be accessed by the callee *capsule*.

An example implementation of the *sanitized transition* gate is to built it on the top of *transient transitions* with assistance of several pre-defined *capsule*s. The outcome of a *sanitized transition* behaves similar to traditional context switches in a kernel-user switch via system call gates, but in an unsupervised way while with more flexibility: 1) the gate is patched to every function call site; 2) the gate can be customized for different call sites according to the ownership of the potential targets.

Figure 5.6 shows an example of cross-domain call via sanitized transition, where a pointer *data* pointing to a stack object in *capsule* A (function foo) is being passed to *capsule* B (function *libBar*). (Note that it might be possible to encapsulate the *data* along with all instructions that access it as a standalone *capsule*; However, here we assume finding all instructions to access *data* is impossible due to the weakness of static analysis,

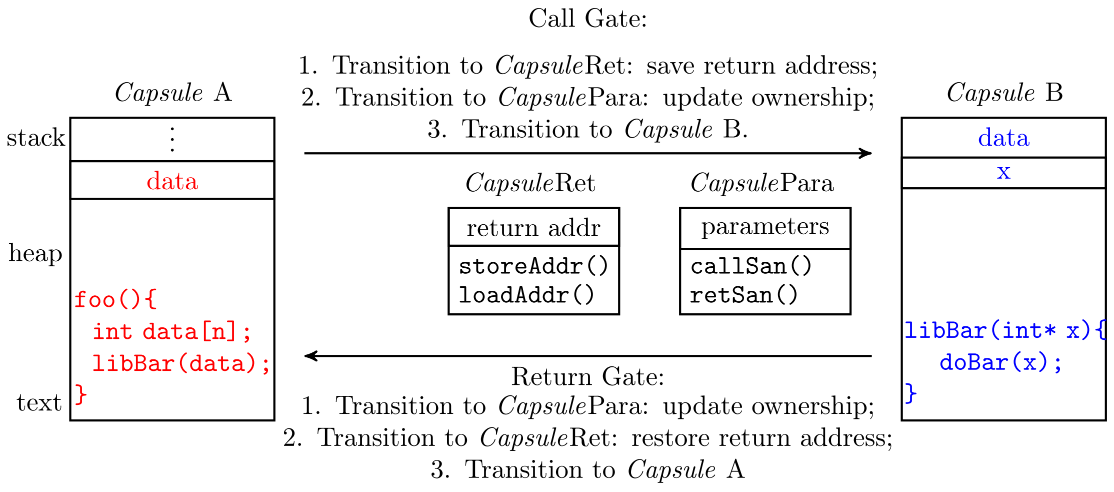

Call Gate:
1. Transition to *Capsule*Ret: save return address;
2. Transition to *Capsule*Para: update ownership;
3. Transition to *Capsule* B.
Capsule A
stack
⋮
data
heap
foo(){
int data[n];
libBar(data);
}
text
*Capsule*Ret
return addr
storeAddr()
loadAddr()
*Capsule*Para
parameters
callSan()
retSan()
Capsule B
data
x
libBar(int* x){
doBar(x);
}
Return Gate:
1. Transition to *Capsule*Para: update ownership;
2. Transition to *Capsule*Ret: restore return address;
3. Transition to *Capsule* A

Figure 5.6: An example of *sanitized transition* gate between two *capsules*.

and the ownership of *data* is being dynamically assigned, thus we need ownership transfer for *Capsule* B to access.)

As shown in Figure 5.6, the *sanitized transition* is done by going through two predefined *capsules*, each transition step is done by simply updating the program counter thus can be regarded as *transient transition*. Overall, the goal of these special *capsules* is to ensure only eligible memory should be accessed by the instructions in the caller and callee following the source code semantics:

*Capsule*Ret encapsulates the return address into a minimalized domain such that only the dedicated instructions in the function prologue and epilogue could save and restore the return address to dedicated stack entry, avoiding potential corruptions by any other instructions in a function.

*Capsule*Para minimizes the exposure of the private memory of both caller and callee *capsules* while keep the original functionality. The shared memory being passed can be sanitized in the following ways: 1) update the *ownership* tags of the shared memory; 2) duplicate the memory content but with new *ownership*. If the shared memory is being passed by only updating the tags, then the caller *capsule* will lose the *ownership* to this piece of data. If data is being passed by copying the data with new tags, the *ownership* of the original data is not changed, thus still accessible for the caller *capsule*. More details about these memory sharing mechanisms will be discussed in the following section.

5.5.3 Memory Sharing Mechanisms

By default, each byte of memory is assumed to be private to only one *capsule*. However, communications between different *capsule*s are sometimes necessary for the program to run normally. To enable data sharing as well as code reuse between different *capsule*s, we design three different memory sharing mechanisms: sharing via **memory duplication**, **ownership transfer**, and dedicated **object *capsule*s**, each with their own advantages and disadvantages in terms of runtime efficiency and burden to the compiler and programmers.

For memory duplication, the memory will be copied from the caller *capsule* to the callee *capsule* so that the callee can gain access. This can be used to support small sized memory contents being passed between *capsule*s, where the consistency between the copies are not required, such as pass-by-value function arguments in most programming languages. However, for large and complex memory objects, such as most pointer arguments, duplication of the memory objects can result in high performance overhead or even change the program behaviour. Therefore, we have the following two mechanisms to mitigate this.

5.5.4 Ownership Transfer

An ownership transfer allows private memory to be passed from one *capsule* to another *capsule* by updating the *ownership* tags. After transfer, the source *capsule* will give up its future access. Ownership transfer only happens in the domain transition gate where data from one *capsule* need to be transferred to another. The ownership transfer routine is predefined and must be strictly reasoned to ensure no abuse to the ownership transfer operations are possible from any other *capsule*.

For pointers being passed, updating the ownership of corresponding object can be challenging. To do so, we will need to decide the size information for the object being pointed to. However, the size of the object can be hard to find out via pure static analysis in the compiler. And it will also be error-prone if we require programmers to provide the size as arguments. Fortunately, we can get the size information of the object dynamically via simple program instrumentation and maintain metadata information for the pointers at runtime. Similar work has been shown effective for this solution such as HardBound [58], SoftBound [116], and CHERI architecture [168]. Therefore, we can leverage similar techniques to reason about the pointer and its size so that we can pass pointer based object between *capsule*s without breaking the system.

5.5.5 Memory Sharing via Object Capsules

Apparently, both duplication and ownership transfer require intrusive changes to the original code and can cause extra memory requests resulting in runtime overhead. This means that if the object being passed is big in size or contain a dense list of pointers, or just being passed back and forth for many times, both solutions will hurt the performance dramatically.

To mitigate this, we can create a dedicated *capsule* which holds the shared object as well as all the instructions that access the object, then this *capsule* could be passed as arguments without overhead of data duplication or ownership transfer. Note that, the dedicated *capsule* holding the shared object can hold instructions from different functions, but not necessarily all the instructions of these functions. This is one unique feature of our fine-grained ownership in contrast to existing coarse-grained privilege management (such as page table permissions), as well as pointer-based object protections.

5.5.6 Avoid Supervised Trusted Stack

Proper switch between isolated domains require two essential property: data protection and control flow integrity. As we discussed above, data protection during domain switch can be made more efficient by leveraging ownership with more options of memory sharing, and it is not always necessary to save and restore the processor context during the switch.

To ensure the control flow integrity, one can use a supervised trusted stack in the OS kernel and allow each *capsule* to make a system call to do transition. However, this require the trust of the big kernel code base, and it is well known that frequent context switches between user and kernel will result in high overhead. Therefore, to avoid the kernel's supervision, we propose to leverage the ubiquitous ownership assigned to the memory to protect the control flow. The key point is to protect the control flow critical data in dedicated *capsule*s and ensure only legal instructions could update these data.

Control flow integrity contain two folds: forward control flow and the backward control flow. The critical data involved are function addresses and return addresses. For forward control flow, we protect one or more target function addresses with dedicated *capsule*s, which can effectively limits which instruction have the privilege to change the function addresses that are stored in the memory. Similarly, backward transition is protected via the dedicated *capsule* that protects the return address, such as the *Capsule*Ret shown in Figure 5.6.

5.5.7 Decentralized Call Gates

The traditional call gate, such as the system call gate on most contemporary architectures, enables the system to audit the cross domain calls in a centralized place, which lives in a fixed location and serves as the only gate for all domain transitions. However, from the security perspective, centralized call gate is a potential single failure point for the entire system. In addition, centralized gate usually provides a general logic for saving and restoring almost all the runtime states of domains, widely known as context switch, which can become a performance bottleneck if a marvelous amount of cross-domain calls are requested.

In contrast, our call gate between *capsule*s are decentralized: Every *capsule* is accompanied with its own call gate which is emitted as a part of the calling convention. This design will avoid the single-point failure and also provides more optimization opportunities based on the specific properties of the calling site.

5.6 Determine the Closure Set for a *Capsule*

Memory objects and instructions are grouped into the same *capsule* by assigning the same ownership identity tags to their corresponding memory location. In order to determine the ownership identity for each piece of the memory content, we provide assistance to the developers by combining three approaches: a) compiler support with static ownership inference; b) dynamically propagate ownership information during execution to expand the *closure set* of a *capsule*; and c) source code annotation on critical data and code.

Algorithm 1 Static Ownership Inference

```
1:  procedure InferOwnership(Module)
2:      for all $Global \in Module$ do
3:          $Global.owner = newOwnerID()$
4:      for all $Function \in Module$ do
5:          InferFuncOwnership(Function)
6:  procedure InferFuncOwnership(Func)
7:      for all $LocalVar \in Func$ do
8:          $LocalVar.owner = newOwnerID()$
9:      for all $Instr \in Func$ do
10:         if $Global \in Instr.Operands$ then
11:             $Instr.owner \leftarrow Global.owner$
12:         else if $Pointer \in Instr.Operands$ then
13:             if $Pointer == \&LocalVar$ then
14:                 $Instr.owner\ ? \leftarrow LocalVar.owner$
15:             else if $Pointer == Param$ then
16:                 $Instr.owner \leftarrow Param.owner$
17:         else if $LocalVar \in Instr.Operands$ then
18:             $Instr.owner \leftarrow LocalVar.owner$
19:     for all $FlowInstr \in Func$ and $FlowInstr.owner! = Target.owner$ do
20:         if ShouldMerge(Target, FlowInstr) then
21:             Merge(Target.owner, FlowInstr.owner)
22:         else
23:             AddGate(Target, FlowInstr)
```

In brief, static analysis with the optional user annotations initialize the ownership of the code and data during compilation, while partial of the contents might left with no owners after compilation. After the program is loaded in the memory, the processor will propagate the ownership by first-touch first-own manner upon a memory access, which eventually ensures the protected memory of the *capsule* will always be a closure set.

5.6.1 Static Ownership Inference

The procedure of static ownership inference is shown in Algorithm 1. It starts from tracking different memory objects located either on the function stack frame, heap or global memory regions, and create a closure set for each memory object. Then it tracks all the instructions in a module that accesses each memory objects and put them into a closure set that contains the memory object. During the analysis, two closure sets might be merged if one instruction is found to access objects in two closure sets.

For cases where it is not possible to find all instructions that access the object, or not possible to find all the objects an instruction accesses, the static algorithm will not assign ownership to the instruction or memory object, and the ownership assignment will be done dynamically by the processor as we will discuss below.

In addition to the ownership inference on existing variables and instructions, compiler can also introduce transformations to the code to create new *capsule*s. The transformation will be conservative so that it will not break the system functionalities, or introduce too much overhead. For example, if a pointer passed in from one function to another, it will first try to group these two functions into one *capsule*, so that it avoids the possible overhead of updating tags.

5.6.2 Memory Colonization

However, purely relying on static analysis is not enough to find out the ownership ID for all instructions and data word in the memory. And relying on user annotation to assign ownership IDs will pose too much requirements on human efforts. Therefore we propose dynamic ownership propagation by the first-touch-first-own (FTFO) rule, which we call *Memory Colonization*: when an instruction accesses a memory object, if an instruction has ownership and the memory object does not, then the memory object will be assigned with the instruction's ownership. This colonization rule also works where an ownership of a memory object is assigned to an ownership-free instruction. The stage of dynamic propagation moves portion of the ownership reasoning responsibility down to the architectural side who inherently has all the runtime information. Note that during execution, false positives can happen where unexpected ownership violations are triggered as a result of too restrictive encapsulations. Whether the violations are false positives are highly depends on the actual semantic context of the applications. But these false positives can be avoided by adding user annotations to explicitly encapsulate the program module. The dynamic propagation rules are listed as below:

1. During execution, if an instruction with non-zero ownership tags accesses a piece of data without ownership, the processor will update the ownership of the data to be the same as the instruction's ownership.

2. If the instruction does not have ownership but the data it accesses has non-zero ownership, the processor will update the ownership of the instruction to be the same as the ownership of the data.

3. If both the data and instruction have no ownership being assigned, the processor will assume they do not belong to any *capsule*, and their ownership will not be changed. Semantically, the memory with no ownership is not protected by the ownership system.

4. Once the ownership is assigned to a memory chunk, it should remain unchanged throughout the life of the process with only limited exceptions for certain kind of data, where their ownership can be transferred from one *capsule* to another. But the ownership transfer operations are not visible to the program to avoid breaking the safety property by programmers. We will discuss the ownership transfer in Section 5.5.2.

5.6.3 Manual Ownership Assignment

We provide a set of user annotation syntax for developers to partition a program at different granularity of a program. Developers can put annotations to a certain portion of code or data and isolate them with the rest of the program. Right now, we support annotations to at the granularity of a compilation module, a function, an instruction/statement, or a certain variable.

In addition to the various granularity capability, another advantage of our partition annotations is that developers do not need to specify which portion of the program should be private to the current *capsule*, and which portion should be shared between certain *capsule*s, which greatly reduces the adoption efforts. This is achieved by our compiler tool-chain support along with the hardware ownership propagation support via memory colonization.

5.6.4 Manual Conflict Resolving

It is possible that the ownership initialization and propagation result in false positive policy violations due to incorrect boundary definitions for the closure sets. In such a case, user annotation could direct the compiler explicitly to assign ownership IDs to some critical variables or instructions such that the false positive violations can be avoided. In addition, developers could also place annotations on the source code to assign ownership IDs to the variables to support security requirements that are specific to applications. Warnings will be issued if there is conflicts between the security policy and the implementation. For example, conflicts can happen if developers explicitly annotated two objects with different ownership IDs, but the compiler founds that it is risky or impossible to split the two objects into two *capsule*s, (such as because a same store instruction operates on the two objects).

5.7 System Stack Extension for Ownership Tags

The goal of our extensions to the the contemporary system is to enforce the following *ownership rule:*

Upon a memory access instruction, the memory access is granted only when the ownership of the instruction matches the ownership of the memory it accesses.

To achieve this, we propose to assign ownership values to the memory at the granularity of instructions and data words. However, adding the ubiquitous ownership information to the entire address space is challenging. One of the challenges is how to store and check the ownership with as less refactoring of the legacy software as possible. In this section, we present two key points of our solutions: 1) The architectural design of the legacy processor is slightly extended to store and operate ownership tags in the shadow memory partition; 2) Compiler-based tool chain is extended to seamlessly infer ownership and generating binary files with ownership information.

5.7.1 Architectural Support

First, the ISA semantics of legacy memory access instructions, including load, store, and instruction fetch related instructions, are extended as following: Ownership of the accessed memory will be loaded at the same time when the instruction load any memory content into the processor; Ownership tags will also be checked before write back stage of the load instruction and before the memory write request is send out to the cache for store instructions; Ownership tags for the instruction will be loaded into the process when the instruction is fetched; Ownership tags can also be updated at runtime based on propagation rules (Section 5.6).

Second, in addition to memory access instructions, we also need to extend the control flow instructions, such as **call** and **return** instructions to support the ownership checking between callers and callees. If the caller and callee are in different *capsule*s, necessary domain transition operations will be conducted to ensure the security as well as the functionality. For example, we can extend the semantics of legacy **call** and **return** instruction pair by adding the following logic: check the ownership ID of caller instruction and callee instruction, if they are the same ownership, do control flow transfer in the traditional way; if they hold different ownership, do control flow transfer by going through a special gate, which will be discussed in Section 5.5.

Third, to seamless enable the ownership tags, we extended the system memory with ownership tags that are transparent to developers. The entire memory hierarchy including registers, caches, and physical memory, as well as binary files are extended. Joannou et. al [93] surveys several forms of tag storage methods, such as expanding word width, or using a shadow table, etc. Expanding word with stores tag bits in place the tagged memory object, for example, expand 32-bit word to 36-bit word by adding an extra 4-bit tag. Shadow table stores tag bits in a dedicated DRAM partition that is separated from the memory objects being tagged. Expanding the width is ideal option, but lacks the real world hardware support, while shadow table partition can fit into existing memory systems, but could cause high DRAM access overhead. In this work, we have explored both forms of storage in order to explore *Capsule*'s performance for different scenarios.

5.7.2 Toolchain Support

The compiler based toolchain needs to be extended to support ownership. First, the compiler parses user annotations and set ownership values for code or data as directed in the program's source code. Second, in addition to user annotation, the compiler also needs to reason about program properties and infer ownership automatically, so that we can enable certain security policies without user annotations (such as stack/heap object protections). This also requires the compiler to understand security policies and invoke corresponding analysis and instrumentation passes, such as the pass for return address protection in Section 5.5.1.

Third, the compiler needs to add ownership sections in the object files it generates, to store ownership tags for corresponding instructions and data in the binary. Internally, compiler will maintain and propagate the ownership information properly throughout its pipelines (such as IR code generation, instruction selection/lowering, machine code assembling, machine code emission, etc.) and composite proper binary sections (e.g. **.ownership.text** section for the instructions in **.text** section, and **.ownership.data** for global values in **.data** sections).

Fourth, the compiler needs to support a few new ownership instructions, specifically, the **capcolorset**, **capcolormatch**. The **capcolorset** instruction can be used to initialize the ownership tags during program start and **capcolormatch** explicitly directs the processor to check ownership during cross-*capsule* calls.

Last but not least, the linker and loader need to be extended to recognize the ownership sections (such as **.ownership.text**) from object files. It will merge the relocatable object files (such as **main.o**), and do proper relocation operations for the ownership sections, so that the ownership is properly tagged to the right instructions and data. Final executable will also be patched with the ownership initialization routines that will be executed at program start (such as code in **.init** section). During program loading, the ownership section needs to be loaded together with the regular text or data sections. In addition, whenever a new page is mapped into address space (for example, during program loading), the ownership tags in the needs to be initialized.

5.8 Implementation

Our prototype system contains 1) a processor extension supporting *ownership* tags checking and manipulation; 2) tagged physical memory to store ownership tags in the memory hierarchy including registers, caches, and main memory; 3) an LLVM based toolchain to infer ownership statically and instrument legacy code to enforce ownership at runtime, and finally generate binary executables with ownership information; 4) a set of example security policies that can be enforced without any user refactoring efforts as well as customized security policies that require minimal user annotation on security critical parts of the program.

5.8.1 Implementations for Functionality, Performance, and Feasibility

We build and evaluate our hardware system in combination of both system emulators (QEMU) and architectural simulators (Gem5), as well as the synthesizable RTL codes (in Verilog), to evaluate the system from different aspects:

1. QEMU emulator is extended to evaluate functional correctness as well as secure properties of the *Capsule* system. It contains a functional MIPS ISA extension, a shadow table based tagged memory model. It is fully compatible with our OS and compiler extensions.

2. Tagged memory models are also implemented in Gem5 simulator to evaluate the the performance of the tagged memory accesses. It contains a modified CPU extension with tag access patterns along with two memory models with ownership tags. The memory models are able to replay memory traces generated by the QEMU or CPU traces generated by GEM5.

3. An open sourced 64-bit MIPS R4000 implementation (BERI [167]) is adapted with our ownership design and we use it to evaluate the feasibility of the hardware adoption. The processor is a single core with L1 instruction and data cache, L2 cache, written in Bluespec [2]. Each L1 cache is 16K, direct-mapped, write-through, and physically indexed. The L2 cache is shared between instruction and data and is configured to be 64K, 4-way set associative, write-back, and physically indexed. The new instructions are implemented as CP2 instructions.

We build our compiler toolchain based on the latest LLVM 11.0 and the operating system is FreeBSD 13.0.

Table 5.1: ISA Extensions to Load Instructions

Load Instructions	Extension Description
Format: *ld* $\$rd, \$rs$ Example Instructions: $ld, lwu, lld, ldl, ldr,$ $lhe, lhue, lhu, lbe, lb,$ $lbue, lbu, lwle, lwl,$ $lw, lwre, lwr, lle, ll$	1. Load the ownership of source memory at [$\$rs$]. 2. Compare the ownership of source memory at [$\$rs$] against the ownership of this instruction. (a) If the ownership values are non-zeros and different, throw exception; (b) If one of them are zero ownership, update the zero-ownership value as the other non-zero value. 3. Check the tag of the source register $\$rs$: (a) If the tag of source register $\$rs$ contains non-zero ownership (code pointer tag), and its different with the ownership of the memory at [$\$rs$], throw exception; (b) If the tag of source register $\$rs$ contain object size information (data pointer tag), and the value of $\$rs$ is out of the range, throw exception; (c) Otherwise, continue normal execution; 4. Load the memory tag at [$\$rs$] and write back the ownership as the tag of register $\$rd$.

5.8.2 Processor Extension

As shown in Table 5.1 - 5.4, we extend the ISA semantics of legacy instructions to interpret and enforce ownership rules, including store, load, and control flow instructions, as well as non-memory access instructions. We also added two new instructions to initialize and explicitly check the ownership in the program, listed in Table 5.5.

Load instructions: Table 5.1 describes the extension logic to the set of load instructions. Upon every memory load, the processor will check that whether the *ownership* tag of the target data chunk matches the load instruction's *ownership tag.* A mismatch will be an *ownership* violation.

Store instructions: Table 5.2 describes how we extend the store instructions. Upon every memory store, the processor will ensure that the target memory chunk has the same

Table 5.2: ISA Extensions to Store Instructions

Store Instructions	Extension Description
Format: *st* $\$rs, \$rd$ Example Instructions: *st*, *sdl*, *sdr*, *swe*, *sw*, *sh*, *sbe*, *sb*, *swle*, *swl*, *she*, *swre*, *swr*	1. Load the ownership of destination memory at [$\$rd$]. 2. Compare the ownership of the memory at [$\$rd$] against the ownership of the current instruction: (a) If the ownership values are non-zeros and different, throw exception; (b) If one of them are zero ownership, update the zero-ownership value as the other non-zero value; (c) Otherwise, continue normal exeuction; 3. Check the tag of the destination register $\$rd$: (a) If the tag of destination register $\$rd$ contains non-zero ownership (code pointer tag), and its different with the ownership of the memory at [$\$rd$], throw exception; (b) If the tag of destination register $\$rd$ contain object size information (data pointer tag), and the value of $\$rd$ is out of the range, throw exception; (c) Otherwise, continue normal execution; 4. Store the tag of the register $\$rs$ as the tag of memory at [$\$rd$] (command issued at memory access stage).

ownership tag as the instruction. However, if the *ownership* value of the data is empty(such as 0), the store instruction will set the data's ownership the same as the instruction. This will be a convenient way to initialize the the ownership of data newly allocated on the stack or heap.

Control Flow Instructions: Table 5.3 describes the extension logic to the legacy control flow instructions. Instructions loaded from the memory into the processor will also load the ownership tag into the processor. More specifically, upon every update of the program counter register the ownership tag of this register will also be updated as the ownership tag for that instruction in the memory. All legacy control flow instructions, such as call and jump, will also being checked, such that the target code has the matching *ownership* with the control flow instruction.

Table 5.3: ISA Extensions to Control Flow Instructions

Instructions	Extension Description
Format: *inst target* Example Instructions: $j, jal, jr, jalr$, beq, bne	• Check the ownership of the target instruction and the ownership of this branch instruction; 1. If values are different and one of them is zero ownership, update the zero ownership as the other non-zero one; 2. If both values are the same, either zero, or non-zero, continue normal execution. 3. If both values are non-zero, but different, throw exception.

Table 5.4: ISA Extensions to Non-memory Access Instructions

Instructions	Extension Description
Format: $inst$ $\{\$[regs]\|[imm]\}$ Example Instructions: $mov, movn$ add, sub, mul, div and, or, nor, slt	• Check the ownership of each register operator against the ownership of this branch instruction: 1. If both values are the same, continue normal execution; 2. If values are different and one of them is zero ownership, update the zero ownership as the other non-zero one; 3. If values are different, throw exception.

Non-memory Access Instructions: Table 5.4 lists the extended logic to non-memory access instructions. Basically, for all non-memory access instructions, we update them to apply the ownership propagate rules. For example, based on the rule of *First-Touch-First-Own* (FTFO), we could update the ownership tag for the register operands. For operands that are not registers (such as immediate values), we do not process the ownership information since these content are not tagged.

New Instructions: As shown in Table 5.5, we add three new instructions to support direct memory tag process:

- **OwnershipSet(paddr, size, ownership)** (instruction **capcolorset**): this is an

Table 5.5: New Instructions for Ownership Manipulation

Capsule Instruction	Description
capcolorset \$rd, \$rs, \$rt	Set the ownership as \$rt for memory object at address starting at *[\$rd]* with size as *\$rs*.
capcolormatch \$rd, \$r1, \$r2	Set *\$rd = 0* if the ownership of the memory at *[\$r1]* and *[\$r2]* do not match; Otherwise, set *\$rd = 1*.
capbcolormatch \$rd, offset	Branch to offset *offset* if the ownership of this instruction matches with the target memory at *[\$rd]*

instruction to update the ownership ID of the memory chunk at specified address and size.

- **OwnershipMatch(result, paddr1, paddr2)** (instruction **capcolormatch**): this new instruction can be used to explicitly check whether two given memory chunks addressed by $r1$ and $r2$ have the same ownership ID.
- **BranchIfOwnershipMatch(target, offset)** (instruction **capbcolormatch**): this new instruction can be used to branch to a certain offset when the given memory address have the same ownership ID as this instruction.

5.8.3 Toolchain Extension

We build our toolchain based on LLVM infrastructure. Several tools are extended to support *capsule* ownership, including the clang frontend, backend, linker, loader, etc. This section explains how we extend the toolchain to initialize, propagate, and generating codes with *capsule* ownership information in the compilation pipelines, including passes that can automatically infer ownership values in order to automatically enforce certain security policies without user annotation.

Here is a list of important changes to the toolchain:

1. Clang support: We extended clang front-end to parse user annotation on the source code and pass the information down to the LLVM IR representations during clang code generation.

2. LLVM IR support: We create LLVM passes to reason about the ownership of memory objects and instructions according to ownership propagation rules. We also create passes to initialize the ownership values based on given security policies.

3. LLVM CodeGen pipeline support:

   (a) We updated the CodeGen passes to hand over ownership values from LLVM IR layer to LLVM DAG Nodes and finally to Machine Instruction (MIR) layer while going through the pipeline stage of instruction selection, instruction scheduling and formation (DAGISel), machine code optimization (MIR).

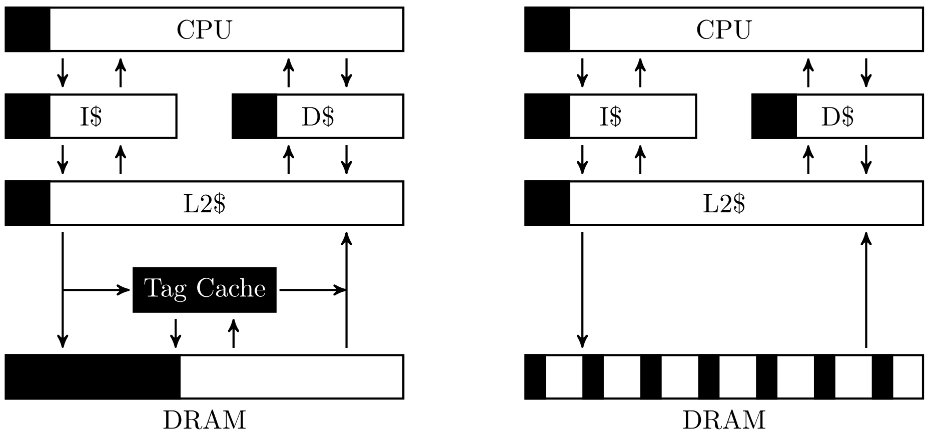
   (b) At final stage of machine code emission, we emit instructions (e.g., to section **.text**) and global values (to section **.data**) with their ownership information. A new section will be created for each section with ownership tags. For example, **.ownership.text** for the code section **.text**.

4. LLVM LLD support: During linking stage, the LLD is adapted to do necessary relocation to the ownership information corresponding to the relocation of its original sections it is mapping to. For example, relocate the **.ownership.text** section while the corresponding **.text** is relocated. The final executable or loadable object will contain a synthetic ownership section **.capsule** which is a combination of all **.ownership.x** sections in the list of input object files.

5.8.4 Ownership Space Management

A 32-bit ownership ID is implemented for every 32-bit memory chunk in our prototype system. User annotations in the source files can use arbitrary ownership IDs, but will probably be relocated during compilation to maintain the flat ownership space over all the *capsule*s being created internally in the compiler. Before linking, different object files can have same ownership IDs being assigned, similar to the virtual address assignment inside each object file. Then, during the linkage time, the linker need to be able to resolve the ownership conflicts and do necessary relocation where necessary. Each ownership entry in the object file has flags that direct the linker to take proper actions on resolving the conflict. If two object files have ownership entries that being assigned with a same ownership ID and the ownership entry is flagged with *MERGE*, then the two entries will be merged into one in the output file. Otherwise, one of the entry will be updated with a new ownership ID.

5.8.5 Two Forms of Tagged Physical Memory



(a) Shadow Memory Tagging (b) In-Memory Tagging

Figure 5.7: Two Memory Models

To explore the performance impact, our tagged memory are implemented in two forms: 1) memory tags in a shadow memory table in a dedicated physical memory partition as shown in Figure 5.7(a), and 2) in-memory tags where every 32 bit of memory is directly extended to store its 32 bit tag as shown in Figure 5.7(b).

Shadow memory table has similar implementation as the tagged memory in CHERI [164, 171]. But in contrast to the tagged memory in CHERI, we also tagged the instructions cache in addition to the data cache at L1 cache, and we use 32-bit tags per 32-bit memory instead of 1-bit tag per 128 bits (or 256 bits) memory used in CHERI. In addition, the interpretation of the tags for pointers and non-pointer data are different with CHERI as we have discussed in Figure 5.3 - 5.5.

Ownership tags is implemented in physical memory instead of the virtual memory. This is in consistency with the strategy of physical memory based processor caching approach which avoids the a various of addressing issues for virtual memory based caches. In addition, the ownership tags can be efficient in performance when implemented in physical memory along with the existing cache support in the physical address space.

5.9 Security Evaluation

The *Capsule* provides basic mechanisms to enhance memory safety of the computer system. It is designed to support as many security policies as possible. In this section, we will first compare our mechanisms with serveral state-of-the-art memory safety solutions. Then we present several exemplar security policies we can implement based on *Capsule*, from both the aspect of inter-domain isolation and intra-domain pointer safety.

5.9.1 Capsule Security Features

Table 5.6 shows an overview of *Capsule*'s advantages over a collections of state-of-the-art isolation mechanisms and Table 5.7 shows the comparison with several general memory protection mechanisms. We have four major advantages (More details of the related mechanisms will be discussed later in Section 5.11).

First, as shown in Table 5.6, *Capsule* framework is the only one that could enforce memory isolation at the granularity of both data word and instruction. Comparing to BGI [33] which can isolate a single data word by interposing every write instructions, *Capsule* in addition is capable to isolate a single instruction by the ownership tag, not just the data word. This enables *Capsule* to seamlessly isolate different instructions inside a single function, for example, during the protection of return address on the stack. As shown in BGI [33], The fine-grained isolation helps to avoid wasting memory space and keep the legacy memory layout unchanged, in contrast to the traditional page-table based memory

Table 5.6: Features of Cross-Domain Isolation Mechanisms

		Process Model	CPU Ring	Intel SGX	Intel MPK	ARM MPU	SFI	BGI	CHERI	Capsule
Architectural Ext.		●	●	●	●	●	○	○	●	●
Granularity	**Instruction**	○	○	○	○	○	○	○	○	●
	Word	○	○	○	○	○	○	●	○	●
	Sub-Object	○	○	○	○	○	○	●	●	●
	Object	○	○	○	○	○	○	●	●	●
	Page	○	●	●	●	●	○	●	●	●
	Region	○	●	●	●	●	●	●	●	●
Auto. Isolation		○	○	○	○	○	○	●	○	●
Ctx. Free Call		○	○	○	●	○	○	●	○	●
Function Part.		○	○	○	○	○	○	○	○	●
Kernel Part.		○	○	○	○	○	●	●	○	●
Application Part.		●	●	●	●	●	●	○	●	●
Unlimited Domains		●	○	●	○	○	○	○	●	●
Read Isolation		●	●	●	●	●	○	○	○	●
Write Isolation		●	●	●	●	●	○	○	●	●
Untrusted OS		○	○	●	○	○	●	○	○	●
Mutual Untrusted		●	○	●	●	●	●	○	○	●
Malicious Code		●	●	●	●	●	●	○	○	●
Backward Compatibility	Legacy Code/API	●	●	○	○	○	○	●	○	●
	Legacy Layout	●	●	○	○	○	○	○	○	●
	New ISA	○	●	●	●	●	○	○	●	●

protections. This also enables *Capsule* to protect a tiny memory location (such as return address, or a single element in a struct) without using any dedicated mechanisms that only target a single kind of vulnerability (such as stack canary, shadow stack, etc).

Second, *Capsule* supports automated deployment of isolation, where most of the mechanisms in Table 5.6 cannot (except BGI [33]). Our framework requires minimal or none user annotations and supports automated boundary discovery for program partitions by combining both the static and dynamic information. All other isolation mechanisms are mostly built on top of isolation interfaces that the compiler cannot understand, such as the

Table 5.7: Features of Intra-Domain Pointer Safety Mechanisms.

		Intel MPX	Java VM	Hard-Bound	Soft-Bound	CETS	CHERI	ARM MTE	Capsule
Spatial Safety		●	●	●	●	○	●	◐	●
Temporal Safety		●	●	○	○	●	●	◐	●
Trust Root in Hardware		●	○	●	○	○	●	●	●
Backward Compatibility	Legacy Code/API	●	○	●	○	○	●	●	●
	Legacy Layout	●	○	●	●	●	○	●	●
	New ISA	●	●	●	○	○	●	●	●
Support Separation		○	○	○	○	○	●	○	●

operating system interfaces or a new set of library interfaces. Although these interfaces for separation are generic to any applications, the adoption to the legacy programs still heavily relies on human efforts. However, *Capsule* create a new separation mechanism based on the ownership of every memory word, which is simple enough to be lowered into the language level concepts and can be easily understood by the compiler as well as the underlying hardware. This brings a higher level of automation and backward compatibility during the deployment of *Capsule* framework. Comparing with the automated isolation in BGI [33], we do not reply on a new secure library interfaces and compiler instrumentation to every memory access and redirect them to go through a access control list checking that can cause high overhead when the number of checks increases (this directs BGI not to check all the memory reads [33]).

Third, *Capsule* is the only hardware extension that supports context-free cross domain calls (where software solutions SFI [159] and BGI [33] also support). The security of the separation is achieved by creating ubiquitous ownership identities for every smallest element in memory. With each piece of memory content has its own ownership identities, we could easily switch the processor's execution from one domain to another without updating all the processor's current state and other domain related data structures, such as stack switches, extra copies, page table updates, etc. It is still safe because the access control of memory contents does not depends on any processor's state, nor any domain related data structures stored in memory (such as page table permissions), but only depends on the ownership identity of the memory, which is transparent to the running applications. These benefits are similar in BGI [33] since every byte in memory have a access control list to contorl its permissions. But BGI [33] only prevents writes and indirect jumps, does not prevent illegal read across domains. SFI [159] does not require context switches but number of domains are rather limited since the size and the placement of the isolated memory regions needs to be carefully designed in order to be fit into a convenient check in serveral instructions. More importantly, BGI and SFI rely purely on software, thus trusted root lacks insurance from the hardware.

In the following, we evaluate the security features of the *capsule* design with more detailed memory safety policies we can enforce.

5.9.2 Spatial Pointer Safety with Bound and Ownership Check

```
OWNERSHIP(111)
uint8_t global_int[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0}; /* developer assigned ownership, 111 */
char global_secret[] = "This is a secret.\n"; /* compiler assigned ownership, e.g. 112 */
int user_get_global(int index){
  //buffer overwrite if index>16
  // ownership and pointer bound will both be checked.
  return global_int[index]; /* generated instructions has compiler inferred ownership, 111 */
}

```

Figure 5.8: Spatial Safety for a Global Variable

Capsule guarantees spatial safety for both global values and local values, including the objects stored in the global section (e.g., **.data**), as well as these on the stack and the heap. The object can either be address-taken or not address-taken. Address-taken objects will be protected by both the ownership tags on instructions and the object words, as well as the data pointer tags which prevents out-of-bound pointer references. Non-address-taken objects will be protected by only the ownership tags.

Figure 5.8 shows a code snippet where a buffer overflow could happen at line 7 but can be prevented in *Capsule*. Global variable *global_int* and *global_secret* will be assigned with different ownership tags, *111* and *112*, respectively. The ownership could be given either by program annotation or by the compiler.

The array *global_int* is accessed via a pointer (thus address-taken) inside the function *user_get_global*, thus the corresponding object bound information will be tagged to the pointer. In addition, instructions accessing the *global_int* will also be tagged with corresponding ownership identity inferred by the compiler (here 111). Therefore at line 7, there will be two checks that could prevent the illegal access to the array of *global_int*:

1. Check by Bound: after the address is computed at line 7, and before the address is used to issue a memory request, the processor will check whether the resulting address is within the bound. If it is out of bound, the processor will throw an exception.

2. Check by Ownership: The ownership of the load instruction at line 7 could be checked against the ownership of the object at that address. If the address points to *global_secret*, the instruction will fail. Note that this requires to load the ownership tag of the target address before the actual data is accessed.

In our implementation, both checks will be used for pointer derived memory access, such as ones in this example. And for non-pointer derived accesses, only the former check will be in use. The spatial safety for objects on the stack, as well as on the heap are also protected with the same policies. Note that the spatial safety achieved by ownership checks are limited by the scope of *capsule*s it is living in. If different non-address taken objects have the same ownership, then spatial safety between these non-address taken objects cannot be guaranteed. This problem should be negligible since the access of non-address taken objects can be easily checked during compilation time.

5.9.3 Temporal Pointer Safety with Ownership Separation

Temporal safety errors can happen both on the heap and the stack, such as use after frees and double frees on the heap, as well as dangling pointer dereferences into the deallocated stack frame [117, 155]. These can be generally classified as two forms: dangling pointer references to deallocated stack/heap objects and dangling pointer dereferences to re-allocated heap objects.

In *Capsule*, the ownership of the deallocated stack frame or heap objects will have different ownership tags with the executing instructions. Therefore, dangling dereferences to deallocated memory will be prevented by ownership checks. In the following, we will show that *Capsule* ownership tags are also effective to protect dangling references to re-allocated heap objects.

The dynamic memory management interfaces such as **malloc()**/**free()** in **libc** is updated to support ownership. The code changes to these interfaces implementation are negligible. For example, only one instruction is inserted after the **malloc()**/**free()** call site to initialize the ownership tags for the allocated heap objects.

Every heap objects will be allocated with its ownership identity corresponding to the allocation site. The ownership of the allocated memory will be initialized before the object is returned to the caller. Upon an instruction that accesses a heap object, the ownership of the instruction will be checked against the ownership of the object. Therefore, if a chunk of heap memory is freed by current *capsule* and reallocated to another *capsule*, current *capsule* cannot access it anymore, which prevents user-after-free bugs. Before the object is freed, the ownership will be checked that the caller has the right ownership to free the object, which prevents double-free bugs. After the object is freed, the ownership tags to the corresponding memory will be cleared.

Listing 5.9 shows a code snippet for temporal safety defense. At line 7, the call to **malloc()** will return an object with an ownership tag. At line 9, the call to **free()** will clear the ownership tags for the object. At line 10, a new allocation is requested and the

```
1  void init_array(int *ptr, int size){
2    for (int i=0; i<size; i++)
3      ptr[i] = i*2+1; // bound + ownership check
4  }
5
6  int main(){
7    int *ptr = (int*) malloc(size *sizeof(int)); /* ownership tagged for the object */
8    init_array(ptr, size);
9    free(ptr); /* ownership checked, then cleared */
10   int *ptr2 = (int*) malloc(size *sizeof(int)); /* new ownership tagged for the new object */
11   *ptr; /* ownership does not match, exception */
12   free(ptr); /* ownership does not match, exception */
13   return 0;
14 }
15
```

Figure 5.9: Temporal Safety for Heap Object

memory region is tagged with a new ownership identity. A use-after-free in line 11, and a double-free at line 12. Both cases are detected by checking the ownership tags in the processor at runtime.

5.9.4 Code Pointer Integrity for Control Flow Protection

Code Pointer Integrity or Code Pointer Separation (CPI/CPS) [99] is shown to be effective in practice to defend against control flow hijacks. It provides stronger security guarantee than Control Flow Integrity (CFI) [11, 120, 177, 179], which is recently to be shown ineffective [32, 55, 75].

CPI/CPS stores all code pointers in a safe region and prevents attackers from forging code pointers by leveraging memory vulnerabilities. However, the safe region in original CPI lives in a dedicated memory protected with special designs, such as segmentation, SFI, etc. This requires program instrumentation to redirect all the original code pointer related accesses to the safe memory region, along with runtime support that facilitates the pointer accesses at runtime. Even the Code Pointer Separation (CPS) could reduce some code pointer related redirection, it still require the safe region to be separated from the original memory, which means the original memory storing code pointers are wasted.

However, with *Capsule*, the safe region in CPI/CPS could be easily implemented by tagging the region with a dedicated ownership. But in contrast, we support more smaller safe regions that can be protected with different domain identities instead of treating them as a single protected domain, which further improves the security of the design by stricter isolation. In addition, the redirection to all the code pointer access instructions in the original CPI/CPS are not needed – all code pointer access instructions just need to be tagged with the same ownership of the code pointers. In this way, we could protect the code pointer integrity in place without allocating a special memory region to store the critical pointers with access redirection.

5.9.5 Control Flow Defense against Malicious *Capsule*

CPI/CPS based control flow protection are designed to defense memory safety programming bugs but cannot defense against malicious code [99]. In *Capsule* system, we support running arbitrary malicious code in a *capsule*, similar to the isolation between sandboxes or kernel/user isolation in an OS. Cross domain calls between a malicious *capsule* and the victim *capsule* are protected by preventing arbitrary branches or returns from a malicious *capsule* into the middle of another victim *capsule*. Return addresses are already protected with our return address *capsule*, as we already discussed in the section 5.5.1. Now we discuss how to prevent a malicious *capsule* (with arbitrary code) jumping into another *capsule* in an arbitrary place.

We can pad to the head of each legal entry point with a dummy instruction that is tagged with the same ownership as the branch instruction. Therefore, branch instructions that wish to jump into this entry must have the same ownership as the this dummy target instruction. In this way, the control flow instructions will be allowed only to jump to the given target instruction at the function header. A malicious *capsule* will be unable to jump to a wrong function or jump into the middle of any instructions inside a *capsule*.

Table 5.8: Security Policy Examples Supported by Different Mechanisms

	Isolation				Pointer Safety	
	Return Address Isolation	Code Pointer Isolation	Heap Object Isolation	Malicious Code Isolation	Spatial Safety	Temporal Safety
Intel MPX	○	○	○	○	●	●
Java VM	○	○	○	○	●	●
HardBound	○	○	○	○	●	○
SoftBound	○	○	○	○	●	○
CETS	○	○	○	○	●	●
ARM MTE	○	○	○	○	◐	◐
Intel CET [149]	●	●	○	○	○	○
CFI [11]	○	○	○	○	○	○
CPI/CPS [99]	●	●	○	○	○	○
Process Model	○	○	○	●	○	○
CPU Rings	○	○	○	●	○	○
Intel SGX	○	○	○	●	○	○
Intel MPK	○	○	○	●	○	○
ARM MPU	○	○	○	●	○	○
SFI	○	○	○	●	◐	○
BGI	○	○	○	○	◐	○
CHERI	○	○	●	●	●	●
Capsule	●	●	●	●	●	●

5.9.6 Comparison in Supported Policy Examples

Table 5.8 lists several state-of-the-art isolation and pointer safety mechanisms both from the academic and industry. We can see that only *Capsule* system could support all the policy examples. This is achieved by our systematic software-hardware co-design and the ubiquitous ownership tags in the memory that most other mechanisms do not have.

5.9.7 Example: Prevent Heartbleed with Two Levels of Defenses

The HeartBleed [1, 64] vulnerability is a simple example to show our defencing effectiveness in real world attack scenarios. Heartbleed allows an attacker to reveal up to 64kB of memory to a connected client or server by sending a specially crafted heartbeat packets.

A missing bounds check in the handling of the TLS heartbeat extension can be leveraged to over read the memory.

In our *Capsule* system, the illegal accesses to memory objects can be prevented by two levels of our defenses: the spatial pointer safety check and the heap object isolation. First, each pointer pointing to the heap is tagged with based and bound information. So when the pointer is out of bound and dereferenced, an out-of-bound exception will be thrown by the processor. Secondly, after a heap object is allocated (e.g., via **malloc**), the heap object is tagged with a unique ownership. Instructions that access this object will also be tagged with the same ownership, either during compilation (via static ownership inference) or during execution (via memory colonization). Therefore, if the any of these instruction overflows to access illegal memory objects that are not tagged with the same ownership, it will be detected and prevented by the processor. Similarly, any illegal instructions that have different ownership tags cannot access these heap objects. The former defense prevents out-of-bound programming errors, while the latter defense helps to prevent malicious code or arbitrarily faked pointers being used to corrupt the heap object.

5.9.8 More Security Policies

In addition to the memory safety issues shown above, it is also notable that *capsule* system is able to enforce more flexible security policies. Given the rich ownership namespace (up to 32-bit tag for every 32-bit memory), we are expecting more use cases will be developed. For example, a stronger control flow integrity policy could be made possible to avoid control flow hijacks shown in [32, 55, 75]. We leave these in future work.

5.10 Hardware Feasibility and Performance

5.10.1 Hardware Feasibility

In addition to run our experiments on simulators, we also implemented the hardware using Bluespec hardware description language, which is built on top of the CHERI-MIPS

Table 5.9: Processor Unit Test Results

	failure	error	pass	pass ratio
BERI	49	277	2120	86.6%
Capsule-DCache	49	277	2120	86.6%
Capsule-Full	61	343	2042	83.5%

processor. The implementation of the hardware prototype is evaluation directed and not fully functional.

The processor ships with a unit test suite which is a composition of Bluespec simulator, the MIPS toolchain, and the Python Nose test framework. The test suite contains 2446 unit tests, each is testing a different ISA interface features. The original processor will pass 277 failures and 49 errors where most of them are due to certain processor features (such as FPU) are disabled during testing. Then we gradually add ownership tag support to the processor: first add to data cache, L2 cache, and tag cache; then we extend tag to the instruction cache as last step.

Table 5.9 shows the result from running the test suite before and after we add our changes to the processor. We can see that original BERI has a pass ratio of 86.6% due to certain processor features are disabled. Then after we add the ownership tag support to the L1 data cache, L2 cache, as well as tag cache (***Capsule*-DCache**), it ends up the same failure ratio. However, when we add ownership tag to the L1 instruction cache, and implemented a full fledged ownership checking logic in the processor (***Capsule*-Full)**, certain functions start to fail. This is because our ownership tags is not fully compatible with the CHERI-MIPS architecture, where a 1-bit tag is used to tag whether this memory location is stored a pointer or not. Therefore, when a memory location is a pointer and is tagged by the BERI processor, our ownership tag operation in the processor will interfere with it.

However, this does not happen in our simulator since we explicitly implemented the simulator such that we do not have conflict with the original BERI simulation. We can either do it by simply disable the CHERI feature in the simulator or by adding simple bit manipulations to avoid the interference. However, this can be non-trivial to do in the hardware implementation and not necessary for the feasibility verification. Therefore we left this to our future work.

5.10.2 Memory Performance

It is easy to see that our hardware changes to the processor is negligible in terms of performance impact, except the tagged memory we have added to the processor. This is because every memory access instruction or instruction fetch will now have two physical memory requests, one for original memory content, and another for memory tags. Therefore, our primary evaluation focuses on the memory overhead caused by ownership tag accesses.

In the processor implementation, both the L1 instruction and data cache, as well as L2 cache are extended to hold ownership tags along with every data blocks. A last level tag cache is added to store only recently accessed tags. All memory request will go through the last level cache component but only tags are processed here and regular non-tag memory requests will be simply forwarded without querying the tag cache. Therefore, the performance overhead can be observed from the memory traffic between the main memory and the last level tag cache.

To evaluate the memory overhead, we collect all the memory traces generated by the processor from QEMU simulator, converting them from virtual addresses to physical addresses, and then replay these traces on Gem5 simulator. We implemented our own cache models on the Gem5 memory simulator. One model simulates a shadow table based tag implementation and the other simulates the in-memory tag implementation. Both models contain L1 and L2 cache and use Traffic Generator in Gem5 as the driver to replay the traces and collect the results. For the in-memory model, we also added another configuration with double size cache capacity to explore the performance bottle neck.

Table 5.10 shows the result we collected for the shadow table based tag implementation. The **Baseline** is the original program execution with no *capsule* ownerships. The **Tagged**

Table 5.10: Memory Overhead with Shadow Table

	Baseline		**Tagged Cache**				avg +%
stage	warmup	iters	warmup	+%	iters	+%	
CPU R Latency(ticks)	5373.45	5373.24	11013.71	104.97	10993.76	104.60	104.78
CPU W Latency (ticks)	3157.66	3157.83	5812.05	84.06	5802.15	83.74	83.90
CPU Packets (#)	32648658	32648657	32648658	0.00	32648657	0.00	0.00
Memory R Requests (#)	1034021	1033936	2034159	96.72	2030620	96.40	96.56
Memory W Requests (#)	1025823	1033933	1947978	89.89	2024220	95.78	92.84
Memory Packets (#)	3093868	3101808	6021753	94.64	6091860	96.40	95.52
Memory Traffic (bytes)	131830016	132343616	254856768	93.32	259509760	96.09	94.71

Cache column shows the data for the configured tagged memory hierarchy with the main memory being partitioned into a shadow table and regular memory. Each trace is replayed for three times consecutively by Gem5. The first time is for system warm up. The table shows the following two iterations of run, as in column **iter 1** and **iter 2** along with the percentage of increase.

From the first two rows of Table 5.10, we could see that the CPU latency has the average overhead from 83.9% for CPU writes and 104.78 % for CPU reads. The number of CPU packets does not change because each regular packet is extended with memory tag thus there is not need to issue standalone packet just for the tag. The numbers of memory read and write requests, as well as the memory traffic are almost doubled. This traffic happens at the interfaces between the main memory and the last level cache. The traffic is doubled because every regular memory request will additionally request its memory tags that resides in another memory partition. The address of the two requests are usually far away and hard to be pre-fetched or optimized.

Table 5.11: CPU and Memory Traffic with In-Memory Tag

	Baseline		**Cache with In-Memory Tags**			
Stage	iter 1	iter 2	iter 1	+%	iter 2	+%
CPU R Latency (ticks)	5373.25	5373.25	9763.81	81.71	9763.81	81.71
CPU W Latency (ticks)	3157.84	3157.84	5105.49	61.68	5105.49	61.68
CPU R Request (#)	2230288	2230288	2230288	0	2230288	0
CPU W Request (#)	30418370	30418370	30418369	0	30418369	0
Memory R Requests (#)	1033937	1033937	2016316	95.01	2016316	95.01
Memory W Requests (#)	1033934	1033934	2016305	95.01	2016305	95.01
Mem Traffic (bytes)	132343744	132343744	258087744	95.01	258087744	95.01

Table 5.11 shows the results of our second cache model – in-memory tag implementation. From the first two rows, we can see the average CPU read/write latency drops to 81.71%/61.68% compared to 104.78%/83.90% of the shadow table implementation. This is because that in the in-memory tag implementation, the address of the tag request is adjacent with the regular memory request, which can usually be served by a single cache hit or a single memory access, instead of two cache hits or two memory accesses as in the shadow table-based implementation.

However, the in-memory tag implementation still incurs much overhead (81.71%/61.68%). To understand the bottleneck, we reconfigured the memory model with double-size cache (L1 and L2). The result is shown in Table 5.12. From the result, we can see the CPU latency overhead has decreased a little bit, from (81.71%/61.68%) to 74.70%/51.12%, so as the memory traffic, down from 95.01% to 81.71%. This tells us the cache size might not contribute to the high overhead in the in-memory tag models. We expect the performance bottleneck lies in the fixed memory bus width, where both the

Table 5.12: CPU and Memory Traffic with Double-Size Caches

	Baseline		**Double-Size Cache**			
Stage	iter 1	iter 2	iter 1	+%	iter 2	+%
CPU R Latency (ticks)	5373.25	5373.25	9387.30	74.70	9387.30	74.70
CPU W Latency (ticks)	3157.84	3157.84	4772.00	51.12	4772.00	51.12
CPU R Request (#)	2230288	2230288	2230288	0	2230288	0
CPU W Request (#)	30418370	30418370	30418369	0	30418369	0
Memory R Requests (#)	1033937	1033937	1878746	81.71	1878746	81.71
Memory W Requests (#)	1033934	1033934	1878742	81.71	1878742	81.71
Mem Traffic (bytes)	132343744	132343744	240479232	81.71	240479232	81.71

tag values and regular memory values are sharing the same bus. So we suggest to expand the width of the memory bus to improve the overall performance.

5.11 Related Work

Below we discuss existing memory protection mechanisms including pointer protections in Section 5.11.1, and then isolation mechanisms in Section 5.11.2. Separation strategies will be discussed in Section 5.11.4. Then we discuss related work on memory ownership (in Section 5.11.5), and systems with memory tags (in Section 5.11.6).

5.11.1 Memory Access Control

Access control mechanisms for memory safety have been evolving very slowly. The most traditional solution are memory segmentation and page table based memory management, where the memory space are divided into a set of segments or pages and each one can have different permissions on it, such as permission bit for read, write, and execute, or privilege bit, etc.

Page table based memory protection has been widely adopted in contemporary machines. Most of them have hardware extensions, such as TLB, that support the page permission check. Built on top of the page management, we have seen several hardware extensions that enforces domain isolation, such as CPU Rings [144] to separate user and kernel space using the privilege level bit on each page, and Intel SGX [47] to build enclaves inside an application by creating special pages in the Enclave Page Cache (EPC).

But page granularity does not fit very well when we need to protect small objects on the same page, or large objects that occupy many pages. While the traditional segmentation based memory protection might help, but many of its inherent problems (such as fragmentation) have prevented it from being widely used today. Therefore, several object-oriented memory protection solutions, such as those used in Java VM [158], HardBound [58], SoftBound [116], CETS [117], Intel MPX [124], CHERI [134], and In-Fat Pointer [172], etc., have been used as complement to the limitation of page-based protections. These solution rely on the fact that objects that needs protection are usually accessed via pointers, and thus they can be protected by checking the metadata of the pointers, such as bound size, permissions, etc.

However, we argue that memory protection at finer granularity, such as instruction and data word, is useful for many use cases in privilege separation, which these page level or object level protection solution cannot provide. For example, as shown in Programmandering [108], developers might need to separate a single function without rewriting the function into two functions. Another example is the need of sub-object protection, as discussed in CHERI [134, 166] and In-Fat Pointer [172], where an element of a data structure might have different permissions as its parent and thus need to be managed separately. But both of them only handle pointers as sub-objects, and does not handle non-pointer sub-objects.

Therefore, none of existing solutions support the access control of a single instruction or data word. Our *Capsule* framework is the first solution that can be used to meet this requirement.

5.11.2 Isolation Mechanisms

Saltzer and Schroeder's 1975 article [138] listed the privilege separation and least privilege as fundamental system design principles to improve security. Based on the underlying isolation mechanisms, we classify isolation solutions into four groups: a) **Isolation by processes**, with process management by OS kernel; b) **Isolation by memory pages**, with page permission managed either by OS kernel or application, along with hardware assistance for efficiency; c) **Isolation by address validation**, with permission check code instrumented into the program by the compiler; d) **Isolation by object oriented memory access control**, with permission embedded as object metadata.

Process-based separation model is the most widely used one. The pioneer work on privilege separation to prevent privilege escalation was purely done manually by rewriting the single process OpenSSH into two different processes [130]. Most recent privilege separation work, such as Privtrans [29], Wedge [23], Chromium [133], Capsicum [163], lwC [106], SOAAP [79], Program-mandering [108], create security domains using separated processes.

Isolation mechanisms built on top of page permissions are also very popular. The most widely used one should be the CPU Ring-style solution [144], first introduced in the Multics [44], which creates hierarchical protection domains in the address space. This is usually enforced by assigning one privilege bit in the page table entry. However, the number of domains is limited and the hierarchical protection does not support mutually untrusted domains. Many recent isolation techniques, such as Intel SGX [47], Intel MPK [90], and ARM TrustZone [16], are also built upon the page-based memory permission management.

Address validation is another mechanism used for domain isolation, which relies on checking the address range as permission check. Many widely used Software Fault Isolation (SFI) [159] techniques rely on the address validation. They usually leverage compiler instrumentation to insert address pattern checking before every memory access, such as Native Client [146, 173]. However, the number of address patterns that can be used for checking is limited by the width of the address and the size of the memory region must be power of 2. For example, Native Client [173] and its portable version [146] only supports two domains – trusted and untrusted, with one domain with 1 GB memory and another with 3 GB.

Recently, object-oriented memory access control is found to be useful for domain isolation. For example, the compartmentalization using object capabilities in CHERI [168] platform can seal the code and data together into one compartment, which creates one security domain, whose memory content is hiding against other objects and can only be accessed via predefined interfaces. However, components in the object are defined based on their pointer representations, which means that the non-pointer memory contents can be hard to be encapsulated in some cases, especially when we need to separate a single instruction or data word inside a function. In addition, CHERI's compartments have to be created manually by programming with a dedicated library. The interfaces between different compartments are also need to be created carefully to maintain its functionality.

Overall, above solutions either only support coarse grained isolation, or a limited number of domains, or cannot build mutually untrusted domains, or the refactoring efforts to the legacy programs during adoption remains high. In contrast, *Capsule* tries to avoid their disadvantages and push the limits of their capabilities further, which can a) isolate memory at finest granularity of instruction and data word; b) support mutually untrusted domains, both in kernel and user space; c) reduce the manual refactoring efforts by compiler assistance; d) provide context free domain switches.

5.11.3 Automatic Isolation

Here we use *isolation* to refer to the process where a list of separated modules and their security requirements are given as input, then these modules will be isolated into different security domains by leveraging certain kinds of isolation mechanisms. This process is mostly focused on how to leverage specific isolation mechanisms to enforce the domain isolation, and the security requirements or how to separate the programs are assumed to be known facts. In this sense, automatic isolation have been explored in several research works.

BGI [33] is one of the pioneer efforts to pursue the automation of the isolation process. It leverages compiler techniques to isolate untrusted kernel drivers automatically by instrumenting the kernel drivers with pre-defined wrapper functions. However, the underlying permission check depends on a software implemented access control list for every pointer, which could cause high overhead if the number of the checked pointers increases, which results in only supports 16 isolated domains are supported. In addition, it supports only for the isolation of different Windows kernel drivers, which leverages the special programming interfaces of kernel drivers. This make it hard to be generally applicable to finer grained isolation inside the kernel nor for the isolation of general user applications.

5.11.4 Automatic Privilege Separation

Here we use *privilege separation* specifically to refer to the process to analyze the program and find out which module should be separated with other modules in a program. This process is usually determined by the security semantic of the program, and is irrelevant to what kind of isolation mechanisms will be used.

Privilege separation are mostly done manually as in [130], where developers rewrite the program into several separated partitions or smaller sub-programs for separation. The engineering process can be divided into two steps: **boundary discovery** and **isolation implementation** [108]. The automation of boundary discovery and separation implementation are both challenging tasks. There have been pioneering work exploring automated boundary discovery but rarely do we see research work that is able to support the automation of separation implementation.

Boundary discovery refers to the problem of grouping each piece of the program into a proper partition, including how many partition of the program should be divided, to meet specific requirements. As an open problem, it has attracted many pioneering work in the past decades. For example, Privtrans [29] used static analysis and C-to-C compiler transformation to partition the input source code into two programs: one as monitor and another one as the slave. SOAAP [79] provides the reasoning capability of complex applications to find possible program partitions based on user annotations. Programmandering [108] proposes to partition a single processed program into two processes based on a control flow graph of the program and provides several options for user to choose from. However, they do not automate the separation implementation, which means lots of manual changes to the legacy code are still required in order to separate the program after partition is discovered.

Separation implementation refers to the step when the program is actually instrumented in order to separate the given multiple modules where each module contains its list of source code (such as functions and global variables in PM [108], or driver source code in BGI [33], or untrusted native code in Native Client [146, 173]). Unfortunately, all existing domain isolation solutions [23, 29, 54, 79, 130, 146, 159, 162, 164, 165, 168, 173] heavily relies on manually refactoring of the legacy program to leverage the separation logic or services. For example, Native Client [173] requires the developer to rewrite the program by replacing system services by NaCl trampolines. CHERI compartmentalization [168] requires developers to rewrite the program using libcheri interfaces to create compartments. Although BGI [33] did not require manual code changes, it could not be used for domain isolation.

In contrast, *Capsule* does not require legacy code changes during the separation implementation stage, which means we could fully automate this step. This reason we can achieve this is very similar to the BGI [33] framework where byte-granularity access control is used to enforce type safety, as well as the isolation of kernel drivers. We both have the capability of access control at byte granularity of the memory and switching between different memory domains does not require the interposition of a special programming interfaces.

5.11.5 Ownership in Memory

Latest Rust programming language shows the powerful idea of ownership to protect memory from being corrupted. However, using new languages can be hard for legacy programs, especially for large scale programs such as operating systems and device drivers, etc.

Wedge [23] also proposes ownership management for the protection of memory for application and provides a set of interface for developers to use. Although the design does not depends on a specific language, it does depends on the implementation of the operating system, as well as heavy refactoring of the legacy programs.

5.11.6 Tagged Memory

Memory Tagging has been explored both in early and contemporary days. Early tagged architectures can date back to 1960s-70s [71], such as LISP machines [169], Rice Computer [70], etc. Each of them using tag for a various kinds of purposes, such as flagging pointers or non-pointers, type information, or debugging information, etc.

Many latest architecture designs also embrace tagged memory to improve security. ARM Memory Tagging Extension(MTE) [57] provides a 4-bit tag to every 16-byte regular memory as well as a 4-bit tag to every pointer, then it implements a lock and key mechanisms to defend against both temporal and spatial pointer safety issues. However, due to the limited number of bit width, the protection is probabilistic, where different objects that share the same tag value could still be overflowed or corrupted with spatial or temporal vulnerabilities.

Intel MPX, PUMP [59], HDFI [153], CHERI [164, 168], etc. hardware supported address sanitizer HWASAN [147], etc. Although some of them can be used to implement isolation, such as CHERI and PUMP, none of them can be adopted without much refactoring efforts on the legacy programs.

5.12 Conclusion

To summarize, this chapter reports our work in exploring fine-grained domain isolation by embedding the security oriented ownership conception to the entire system stack, from the applications, operating system, compiler tool chain down to the hardware processor. We extended the processor's legacy ISA to process ownership tags and designed our toolchain and system extensions to seamless support ownership based domain isolation for legacy applications. We have built our prototype system *Capsule* and evaluation shows it is effective to defense against many kinds of memory safety issues. However, the performance evaluation shows that the current implementation has several drawbacks that can be improved, such as the memory subsystem. With more improvement to our *Capsule* system, we believe more performance optimizations can be made to reach a practical system and more security guarantees are possible with new security policies being developed.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The new computing environments, such as edge and IoT environment, require more efficient and more effective isolation mechanisms to be proposed. This thesis work describes and resolves multiple problems of the traditional isolation mechanisms that have prevented them to be used in the edge and IoT environment. These include the problem of live migration across edge servers, the security monitoring problem on edge servers, and the fine-grained isolation problem on IoT devices.

With our explorations, we have learned the necessity to revise the traditional isolation mechanisms based on the specific requirements of the edge and IoT environments. These requirements are largely originated from the new features in the new environment, such as bandwidth and latency conditions, the new efficiency requirements, the available security features, and the new hardware constraints, etc. More importantly, to compose an efficient system, all these new features and requirements must be considered in combination instead of only optimizing one of them. This has resulted in a deeply optimized system stack, usually including both the software and the hardware, that aims to serve specific tasks.

To summarize, we can see that emerging new computing paradigm brings new opportunities of revising the traditional isolation techniques. This thesis only explores a small subset of these opportunities. We expect that many more opportunities will be discovered and a various kinds of new isolation mechanisms or revisions for the edge and IoT platforms will come soon.

6.2 Future Work

The problems illustrated in this thesis are resolved but the underlying technologies will not stop evolving. With more and more new application scenarios emerging, such as autonomous driving, drone delivery, etc., along with new hardware platforms being built, such as AI processors, FPGA soft cores, RISC-V cores, etc., it is becoming more and more effortless to compose new computer platform with brand new hardware and software stacks. In the future, many newly designed systems will emerge with deeply customized software and hardware for various of domain specific tasks. Given this trend, we will need more innovations to compose efficient and effective isolation mechanisms for secure resource sharing on these heterogeneous devices and applications. Therefore, in addition to this thesis, it is urgent to continue revisiting the traditional isolation techniques. Then we will be able to revise them or propose new isolation mechanisms to meet the new requirements for these emerging applications.

Bibliography

[1] CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160., February 28 2019.

[2] Bluespec compiler. `https://github.com/B-Lang-org/bsc`, 2021.

[3] Clang: a c language family frontend for llvm. `https://clang.llvm.org/`, 2021.

[4] Flute, a free and open-source risc-v cpu. `https://github.com/bluespec/Flute`, 2021.

[5] gem5: The gem5 simulator system. `https://www.gem5.org/`, 2021.

[6] Lld - the llvm linker. `https://lld.llvm.org/`, 2021.

[7] The llvm compiler infrastructure project. `https://llvm.org/`, 2021.

[8] Mini-os - xen. `https://wiki.xenproject.org/wiki/Mini-OS`, 2021.

[9] Piccolo, 3-stage, in-order pipeline risc-v cpu. `https://github.com/bluespec/Piccolo`, 2021.

[10] Toooba, superscalar, deep pipeline risc-v cpu. `https://github.com/bluespec/Toooba`, 2021.

[11] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, 2005.

[12] Giuseppe Aceto, Alessio Botta, Walter de Donato, and Antonio Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.

[13] Tiago Alves. Trustzone: Integrated hardware and software security. *White paper*, 2004.

[14] Brandon Amos, Bartosz Ludwiczuk, and Mahadev Satyanarayanan. Openface: A general-purpose face recognition library with mobile applications. Technical report, CMU-CS-16-118, CMU School of Computer Science, 2016.

[15] Claudio A. Ardagna, Rasool Asal, Ernesto Damiani, and Quang Hieu Vu. From security to assurance in the cloud: A survey. *ACM Comput. Surv.*, 48(1), July 2015.

[16] ARM. *Arm Architecture Reference Manual: Armv8, for Armv8-A architecture profile.* ARM.

[17] ARM. *Armv8-M Architecture Reference Manual.* ARM.

[18] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, et al. Scone: Secure linux containers with intel sgx. In *OSDI*, volume 16, pages 689–703, 2016.

[19] Inc. Assured Information Security. Bareflank hypervisor, 2018.

[20] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[21] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[22] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[23] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. USENIX Association, 2008.

[24] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[25] E. Bosman and H. Bos. Framing signals - a return to portable shellcode. In *2014 IEEE Symposium on Security and Privacy*, pages 243–258, May 2014.

[26] Ross Boucher. Live migration using criu. `https://github.com/boucher/p.haul`, 2017.

[27] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, pages 250–257. IEEE, 2015.

[28] Martin A. Brown. Traffic control howto. `http://www.tldp.org/HOWTO/Traffic-Control-HOWTO/`, 2017.

[29] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72, 2004.

[30] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining Light on Shadow Stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.

[31] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow

integrity. In *Proceedings of the 24th USENIX Security Symposium*, Security '15, pages 161–176, Washington, DC, 2015. USENIX Association.

[32] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 385–399, San Diego, CA, 2014. USENIX Association.

[33] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast Byte-Granularity Software Fault Isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, New York, NY, USA, 2009. ACM.

[34] Deyan Chen and Hong Zhao. Data security and privacy protection issues in cloud computing. In *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*, volume 1, pages 647–651. IEEE, 2012.

[35] Peter M Chen and Brian D Noble. When virtual is better than real [operating system relocation to virtual machines]. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 133–138. IEEE, 2001.

[36] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review*, 42(2):2–13, 2008.

[37] Mung Chiang and Tao Zhang. Fog and iot: An overview of research opportunities. *IEEE Internet of Things Journal*, 3(6):854–864, 2016.

[38] David Chisnall, Colin Rothwell, Robert NM Watson, Jonathan Woodruff, Munraj Vadera, Simon W Moore, Michael Roe, Brooks

Davis, and Peter G Neumann. Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. *ACM SIGARCH Computer Architecture News*, 43(1):117–130, 2015.

[39] Joseph Cihula. Trusted boot: Verifying the xen launch. *Xen Summit*, 7, 2007.

[40] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[41] George Coker. Xen security modules (xsm). *Xen Summit*, pages 1–33, 2006.

[42] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 189–202, 2011.

[43] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 952–963, Denver, CO, 2015. ACM.

[44] Fernando J Corbató and Victor A Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 185–196, 1965.

[45] CoreOS. A security-minded, standards-based container engine. `https://coreos.com/rkt`, 2017.

[46] CoreOS. Running docker images with rkt. `https://coreos.com/rkt/docs/latest/running-docker-images.html`, 2018.

[47] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[48] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[49] Linux Counter. Funny statistics for the linux kernel, 2018.

[50] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual ghost: Protecting applications from hostile operating systems. In *ACM SIGPLAN Notices*, volume 49, pages 81–96. ACM, 2014.

[51] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. page 351–366, 2007.

[52] CRIU. Criu. `https://criu.org/Main_Page`, 2017.

[53] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[54] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 191–206, Istanbul, Turkey, 2015.

[55] LUCAS DAVI, AHMAD-REZA SADEGHI, DANIEL LEHMANN, AND FABIAN MONROSE. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 401–416, San Diego, CA, 2014. USENIX Association.

[56] PETER J. DENNING. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, September 1970.

[57] ARM DEVELOPER. Armv8.5-a memory tagging extension. `https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf`, 2021.

[58] JOE DEVIETTI, COLIN BLUNDELL, MILO M. K. MARTIN, AND STEVE ZDANCEWIC. Hardbound: Architectural support for spatial safety of the C programming language. pages 103–114, 2008.

[59] UDIT DHAWAN, NIKOS VASILAKIS, RAPHAEL RUBIN, SILVIU CHIRICESCU, JONATHAN M SMITH, THOMAS F KNIGHT JR, BENJAMIN C PIERCE, AND ANDRÉ DEHON. Pump: a programmable unit for metadata processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2014.

[60] ARTEM DINABURG, PAUL ROYAL, MONIRUL SHARIF, AND WENKE LEE. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.

[61] DOCKER. Docker documentation – use volumes. `https://docs.docker.com/engine/admin/volumes/volumes/`, 2017.

[62] DOCKER DOCUMENTATION. Docker storage drivers. `https://docs.docker.com/storage/storagedriver/select-storage-driver/`, 2021.

[63] Adam Dunkels. Design and implementation of the lwip tcp/ip stack. *Swedish Institute of Computer Science*, 2:77, 2001.

[64] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[65] Pavel Emelyanov. Live migration using criu. `https://github.com/xemul/p.haul`, 2017.

[66] Dawson R Engler, M Frans Kaashoek, et al. *Exokernel: An operating system architecture for application-level resource management*, volume 29. ACM, 1995.

[67] ESTESP. Storage drivers in docker: A deep dive. `https://integratedcode.us/2016/08/30/storage-drivers-in-docker-a-deep-dive/`, 2016.

[68] Kaniz Fatema, Vincent C. Emeakaroha, Philip D. Healy, John P. Morrison, and Theo Lynn. A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. *Journal of Parallel and Distributed Computing*, 74(10):2918–2933, 2014.

[69] Diogo AB Fernandes, Liliana FB Soares, João V Gomes, Mário M Freire, and Pedro RM Inácio. Security issues in cloud environments: a survey. *International Journal of Information Security*, 13(2):113–170, 2014.

[70] Edward A Feustel. The rice research computer: a tagged architecture. In *Proceedings of the May 16-18, 1972, spring joint computer conference*, pages 369–377, 1971.

[71] Edward A Feustel. On the advantages of tagged architecture. *IEEE Transactions on Computers*, 100(7):644–656, 1973.

[72] Linux Foundation. runc. `https://runc.io/`, 2017.

[73] Joseph Gardiner and Shishir Nagaraja. On the security of machine learning in malware c&c detection: A survey. *ACM Computing Surveys (CSUR)*, 49(3):59, 2016.

[74] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *Ndss*, volume 3, pages 191–206, 2003.

[75] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 575–589, San Jose, CA, 2014. IEEE Computer Society.

[76] Stéphane Graber. Lxc 1.0: Container storage [5/10]. `https://stgraber.org/2013/12/27/lxc-1-0-container-storage/`, 2013.

[77] J Greene. Intel trusted execution technology, white paper. *Online: http://www.intel.com/txt*, 2012.

[78] Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. Cyclone: A type-safe dialect of c. *C/C++ Users Journal*, 23(1):112–139, 2005.

[79] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean application compartmentalization with soaap. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1016–1031, New York, NY, USA, 2015. Association for Computing Machinery.

[80] Kiryong Ha, Yoshihisa Abe, Zhuo Chen, Wenlu Hu, Brandon Amos, Padmanabhan Pillai, and Mahadev Satyanarayanan. Adaptive vm handoff

across cloudlets. Technical report, Technical Report CMU-CS-15-113, CMU School of Computer Science, 2015.

[81] Kiryong Ha, Yoshihisa Abe, Thomas Eiszler, Zhuo Chen, Wenlu Hu, Brandon Amos, Rohit Upadhyaya, Padmanabhan Pillai, and Mahadev Satyanarayanan. You can teach elephants to dance: agile vm handoff for edge computing. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, page 12. ACM, 2017.

[82] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 68–81. ACM, 2014.

[83] Zijiang Hao and Qun Li. Edgestore: Integrating edge computing into cloud-based storage systems. In *Edge Computing (SEC), IEEE/ACM Symposium on*, pages 115–116. IEEE, 2016.

[84] Zijiang Hao, Ed Novak, Shanhe Yi, and Qun Li. Challenges and software architecture for fog computing. *IEEE Internet Computing*, 21(2):44–53, 2017.

[85] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing—a key technology towards 5g. *ETSI White Paper*, 11, 2015.

[86] Docker Inc. Docker hub. `https://hub.docker.com/`, 2017.

[87] Docker Inc. Docker images and containers. `https://docs.docker.com/storage/storagedriver/`, 2017.

[88] Docker Inc. What is docker? `https://www.docker.com/what-docker`, 2017.

[89] Open Container Initiative. Open container initiative. `https://opencontainers.org/`, 2021.

[90] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual.* Intel.

[91] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E Porter, and Radu Sion. Sok: Introspections on trust and the semantic gap. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 605–620. IEEE, 2014.

[92] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138. ACM, 2007.

[93] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W Moore, Alex Bradbury, Hongyan Xia, Robert NM Watson, David Chisnall, Michael Roe, Brooks Davis, et al. Efficient tagged memory. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 641–648. IEEE, 2017.

[94] Jeff Johnston and T Fitzsimmons. The newlib homepage. *URL http://sourceware. org/newlib*, 2011.

[95] Antti Kantee and Justin Cormack. Rump kernels no os? no problem! *USENIX; login: magazine*, 2014.

[96] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 350–361. ACM, 2010.

[97] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[98] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt,

Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[99] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 147–163, 2014.

[100] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. CCS '13, page 721–732, New York, NY, USA, 2013. Association for Computing Machinery.

[101] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *Information Processing in Sensor Networks (IPSN), 2016 15th ACM/IEEE International Conference on*, pages 1–12. IEEE, 2016.

[102] Aaron Lehmann. 1.10 distribution changes design doc. `https://gist.github.com/aaronlehmann/b42a2eaf633fc949f93b`, 2015.

[103] Tamas K Lengyel, Steve Maresca, Bryan D Payne, George D Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 386–395. ACM, 2014.

[104] Daniel Lezcano. Lxc - linux containers. `https://github.com/lxc/lxc`, 2017.

[105] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM*

Symposium on Operating Systems Principles, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[106] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An {OS} abstraction for safety and performance. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 49–64, 2016.

[107] Peng Liu, Dale Willis, and Suman Banerjee. Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge. In *Edge Computing (SEC), IEEE/ACM Symposium on*, pages 1–13. IEEE, 2016.

[108] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. Program-mandering: Quantitative privilege separation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1023–1040, 2019.

[109] Lele Ma, Shanhe Yi, and Qun Li. Efficient service handoff across edge servers via docker container migration. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 11:1–11:13. ACM, 2017.

[110] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis. Live service migration in mobile edge clouds. *IEEE Wireless Communications*, PP(99):2–9, 2017.

[111] Anil Madhavapeddy and David J Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30, 2013.

[112] Richard Maliszewski, Ning Sun, Shane Wang, Jimmy Wei, and Ren Qiaowei. Trusted boot (tboot), 2015.

[113] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and

attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 143–158. IEEE, 2010.

[114] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert NM Watson, and Peter Sewell. Into the depths of c: elaborating the de facto standards. *ACM SIGPLAN Notices*, 51(6):1–15, 2016.

[115] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving xen security through disaggregation. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 151–160. ACM, 2008.

[116] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, 2009.

[117] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler-Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 31–40. ACM, 2010.

[118] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005.

[119] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.*, pages 69–70. IEEE, 2004.

[120] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 577–587, New York, NY, USA, 2014. Association for Computing Machinery.

[121] JR Okajima. Aufs. `http://aufs.sourceforge.net/aufs3/man.html`, 2017.

[122] OpenVZ. Openvz virtuozzo containers wiki, 2017.

[123] OpenVZ. Virtuozzo storage. `https://openvz.org/Virtuozzo_Storage`, 2017.

[124] Christian W Otterstad. A brief evaluation of intel® mpx. In *2015 Annual IEEE Systems Conference (SysCon) Proceedings*, pages 1–7. IEEE, 2015.

[125] Milan Patel, B Naughton, C Chan, N Sprecher, S Abeta, A Neal, et al. Mobile-edge computing introductory technical white paper. *White Paper, Mobile-edge Computing (MEC) industry initiative*, 2014.

[126] Bryan D Payne. Simplifying virtual machine introspection using libvmi. *Sandia report*, pages 43–44, 2012.

[127] Satya Popuri. A tour of the mini-os kernel, 2014.

[128] LibVMI Project. Libvmi virtual machine introspection, fast, portable, simple, 2015.

[129] Xen Project. Why xen project? `https://xenproject.org/users/why-xen/`, 2021.

[130] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *USENIX Security Symposium*, 2003.

[131] Yuqing Qiu. *Evaluating and Improving LXC Container Migration between Cloudlets Using Multipath TCP*. PhD thesis, Electrical and Computer Engineering, Carleton University Ottawa, 2016.

[132] Akamai Releases Second Quarter. State of the internet report. *Akamai: http://www. akamai. com/html/about/press/releases/2014/press-093014. html. Accessed*, 2, 2014.

[133] Charles Reis and Steven D Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 219–232, 2009.

[134] Alexander Richardson. Complete spatial safety for c and c++ using cheri capabilities. Technical report, University of Cambridge, Computer Laboratory, 2020.

[135] D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan. Unix time-sharing system: The c programming language. *The Bell System Technical Journal*, 57(6):1991–2019, 1978.

[136] Rodrigo Roman, Javier Lopez, and Masahiro Mambo. Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges. *Future Generation Computer Systems*, 78:680–698, 2018.

[137] Rami Rosen. Linux containers and the future cloud. *Linux J*, 2014(240), 2014.

[138] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[139] Ravi S Sandhu and Pierangela Samarati. Access control: principle and practice. *IEEE communications magazine*, 32(9):40–48, 1994.

[140] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[141] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4), 2009.

[142] Fred B Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[143] Fred B Schneider. Least privilege and more [computer security]. *IEEE Security & Privacy*, 99(5):55–59, 2003.

[144] Michael D Schroeder and Jerome H Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3):157–170, 1972.

[145] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 38–54. IEEE, 2015.

[146] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. pages 1–11, 2010.

[147] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory tagging and how it improves c/c++ memory safety. *arXiv preprint arXiv:1802.09517*, 2018.

[148] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 335–350. ACM, 2007.

[149] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, New York, NY, USA, 2019. Association for Computing Machinery.

[150] Adrian L Shaw, Behzad Bordbar, John Saxon, Keith Harrison, and Chris I Dalton. Forensic virtual machines: dynamic defence in the cloud via introspection. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 303–310. IEEE, 2014.

[151] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing xen. In *NDSS*, 2017.

[152] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[153] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2016.

[154] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, Dave Andersen, and Jay Lepreau. The flask security architecture: System support for diverse security policies. 1999.

[155] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[156] Andrew Vagin. Fosdem 2015 - live migration for containers is around the corner. *Online, https://archive.fosdem.org/2015/schedule/event/livemigration/*, 2017.

[157] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium*

(USENIX Security 19), pages 1221–1238, Santa Clara, CA, August 2019. USENIX Association.

[158] Bill Venners. *The java virtual machine.* McGraw-Hill, New York, 1998.

[159] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, SOSP'93, pages 203–216, Asheville, NC, 1993. ACM.

[160] Robert J. Walls, Nicholas F. Brown, Thomas Le Baron, Craig A. Shue, Hamed Okhravi, and Bryan C. Ward. Control-flow integrity for real-time embedded systems. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, ECRTS '19, pages 2:1–2:24, Stuttgart, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum füer Informatik.

[161] Aaron Walters. The volatility framework: Volatile memory artifact extraction utility framework, 2007.

[162] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. Towards memory safe enclave programming with rust-sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2333–2350, 2019.

[163] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *USENIX Security Symposium*, volume 46, page 2, 2010.

[164] Robert NM Watson, Peter G Neumann, Jonathan Woodruff, Jonathan Anderson, Ross Anderson, Nirav Dave, Ben Laurie, Simon W Moore, Steven J Murdoch, Philip Paeps, et al. Cheri: a research platform decon-

flating hardware virtualisation and protection. Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE), 2012.

[165] Robert NM Watson, Robert M Norton, Jonathan Woodruff, Simon W Moore, Peter G Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, et al. Fast protection-domain crossing in the cheri capability-system architecture. *IEEE Micro*, 36(5):38–49, 2016.

[166] Robert NM Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W Moore, Edward Napierala, Peter Sewell, et al. Cheri c/c++ programming guide. 2020.

[167] Robert NM Watson, Jonathan Woodruff, David Chisnall, Brooks Davis, Wojciech Koszek, A Theodore Markettos, Simon W Moore, Steven J Murdoch, Peter G Neumann, Robert Norton, et al. Bluespec extensible risc implementation: Beri hardware reference. Technical report, University of Cambridge, Computer Laboratory, 2015.

[168] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[169] Daniel Weinreb and David Moon. The lisp machine manual. *ACM SIGART Bulletin*, (78):10–10, 1981.

[170] Sébastien Wilmet. The glib/gtk+ development platform. 2017.

[171] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neu-

mann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468. IEEE, 2014.

[172] Shengjie Xu, Wei Huang, and David Lie. In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. ASPLOS 2021, page 224–240, New York, NY, USA, 2021. Association for Computing Machinery.

[173] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, SP'09, pages 79–93, 2009.

[174] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog computing: Platform and applications. In *Hot Topics in Web Systems and Technologies (HotWeb), 2015 Third IEEE Workshop on*, pages 73–78. IEEE, 2015.

[175] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*, pages 37–42. ACM, 2015.

[176] Shanhe Yi, Zhengrui Qin, and Qun Li. Security and privacy issues of fog computing: A survey. In *International Conference on Wireless Algorithms, Systems, and Applications*, pages 685–695. Springer, 2015.

[177] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, L. Szekeres, S. McCamant, D. Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573, 2013.

[178] Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. Spectre: A dependable introspection framework via system management mode. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.

[179] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, Washington, D.C., August 2013. USENIX Association.

[180] Tong Zhang, Dongyoon Lee, and Changhee Jung. Bogo: buy spatial memory safety, get temporal memory safety (almost) free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644, 2019.

[181] ChongChong Zhao, Daniyaer Saifuding, Hongliang Tian, Yong Zhang, and ChunXiao Xing. On the performance of intel sgx. In *2016 13th Web Information Systems and Applications Conference (WISA)*, pages 184–187, 2016.

[182] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J. Walls. Silhouette: Efficient protected shadow stacks for embedded systems. In *Proceedings of the 29th USENIX Security Symposium*, Security '20, pages 1219–1236, Boston, MA, 2020. USENIX Association.

VITA

Lele Ma

Lele Ma is a Ph.D. Candidate in the Computer Science Department at the College of William & Mary, supervised by Professor Qun Li. His research interests lie in system support for edge computing and the internet of things, with emphasis on designing secure and efficient operating systems leveraging a wide spectrum of isolation techniques. He received a master's degree from the University of Chinese Academy of Sciences, and a bachelor's degree from Shandong University, China. He had been a visiting student at University of Rochester in 2018-2019, and a research intern at Pacific Northwest National Lab in 2020. His writings have been seen in ACM/IEEE Symposium on Edge Computing (SEC) 2017, ACSIC Symposium on Frontiers in Computing (SOFC) 2018, IEEE Transactions on Mobile Computing (TMC) 2019, and co-authored writings in Proceedings of IEEE 2019, USENIX Security Symposium 2020.