TÉCNICO LISBOA

Faastion: Secure and Elastic Multi-tenant Serverless Runtime

Rodrigo Foito de Amoreira Cidra

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Rodrigo Bruno

Examination Committee

Chairperson: Prof. Luís Veiga
Supervisor: Prof. Rodrigo Bruno
Member of the Committee: Prof. Nuno Santos

May 2024

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

Completing these five and a half years at Instituto Superior Técnico has been a challenging journey, making this accomplishment all the more rewarding. I am certain that I could not have achieved this without the support of an amazing group of people. It is a pleasure to thank those who made it a possibility:

- To Maria and Carlos, my heartfelt gratitude for your unwavering support and nurturing guidance throughout the years. Your encouragement and care have been constant sources of strength, helping me navigate life's challenges with resilience. Your presence, through both triumphs and trials, has been invaluable. I am deeply grateful for the love and wisdom you have imparted, shaping me into the person I am today.

- To Paulo, Fátima, and Lourdes, for always being there for me whenever I needed help and care. Your willingness to help and take care of me whenever I needed it has been a tremendous source of comfort and strength. I hope you can witness all my accomplishments in life, as my greatest wish is to share them with you.

- To Beatriz, I'm immensely thankful for your patience, which you graciously extend to me each day. Your nurturing care and abundant displays of love have enriched the past 7 years beyond measure. Despite the distance that once separated us, our journey has brought us closer than ever before. I eagerly anticipate the future and the countless adventures it promises to unveil. Thank you for being my constant source of strength and endless joy.

- To Lourenço, Filipe, and Miguel, my heartfelt thanks for being there through every step of our shared journey growing up together. Miguel and Filipe, your camaraderie and support have been invaluable to me. And to the little guy, Lourenço, for your enthusiasm for me to finish my work so we could play. It is a pleasure seeing you grow before my eyes. You are the most brilliant and inspiring kid I know. I cherish the memories we've created and look forward to making many more together.

- To João Santos, João Silva, João Simões, and Paulo do Ó, my deepest gratitude for all the help and laughter we shared during our time at IST. Your friendship has been a source of stability and many laughs, consistently putting a smile on my face. Special thanks to João Silva for the countless all-nighters we pulled together to finish projects on time, and for being there every day and night as we wrote our theses side by side. Your dedication and companionship made this journey memorable and rewarding. I couldn't have asked for better friends, and I am thankful for the bond we have formed.

- To André, João Cunha, Joana, and Rita, for listening to my countless complaints and encouraging me to keep moving forward. Our numerous adventures together have been unforgettable, and I hope our friendship lasts a lifetime. Special thanks to João Cunha, the best roommate I could have asked for, for enduring my whining and offering support whenvever and wherever. I am truly glad you weren't accepted in Porto, as your presence has been invaluable to me.

- To Professor Rodrigo Bruno, I am deeply grateful for your incredible knowledge, insights, and especially your time. I know how fortunate I am to have had you as my advisor. Your dedication and willingness to invest time in my work, regardless of your workload, have been invaluable. I truly appreciate you as a professor, not only for your expertise but also for showing that you genuinely care about your students.

- To my colleagues from Coalition's Scanning Engine team, I extend my gratitude for your encouragement throughout the completion of my thesis. Your words of motivation were constant reminders to persevere, even during the most challenging times. I appreciate the knowledge and insights shared with me daily over the past one and a half years. Thank you for your guidance and for contributing to my personal and professional growth.

To each and every one of you – Thank you.

Abstract

The increasing demand for scalable and elastic cloud computing as evidenced by the growing popularity of serverless has demanded language runtimes to be shared across concurrent invocations of serverless functions. Modern language runtimes are capable of running multiple isolated functions in separate heaps (often called *isolates*) which rely on Software-Fault Isolation (SFI) enforced by the compiler and runtime to confine isolation. However, such SFI guarantees only hold if functions do not rely on native libraries. Unsurprisingly, we found that functions often rely on native implementations of sophisticated algorithms, otherwise hard to implement efficiently in managed languages. Examples include image and video processing, encryption, hashing, and Machine Learning inference. State-of-the-art runtimes either conservatively rely on hardware-level isolation to isolate the entire execution of serverless functions, which imposes elasticity and scalability restrictions and does not benefit from SFI while the function executes managed code, or disable native code access altogether. In this work, we propose Faastion, a new runtime that combines SFI with hardware-based isolation by relying on hardware-based isolation only when functions execute native code. Our implementation is based on GraalVM Native Image, a Java runtime that supports isolates through SFI, combined with Intel Memory Protection Keys (MPK) for hardware-based isolation. On widely used benchmarks and workloads, we demonstrate that Faastion can safely execute concurrent functions with the elasticity and scalability benefits of SFI and the hardware-based isolation required to run native code.

Keywords

Serverless Computing; Software-Fault Isolation (SFI); Hardware-based isolation.

Resumo

A crescente demanda por computação escalável e elástica em nuvem, como evidenciada pela crescente popularidade do serverless, tem exigido que os runtimes de linguagem sejam compartilhados entre invocações concorrentes de funções serverless. Os runtimes de linguagem modernas são capazes de executar várias funções isoladas em heaps separadas (frequentemente chamados de *isolates*), que dependem do Isolamento de Falhas de Software (SFI) aplicado pelo compilador e runtime para confinar o isolamento. No entanto, tais garantias de SFI só se mantêm se as funções não dependerem de bibliotecas nativas. Não surpreendentemente, descobrimos que as funções muitas vezes dependem de implementações nativas de algoritmos sofisticados, caso contrário, são difíceis de implementar de forma eficiente em linguagens gerenciadas. Exemplos incluem processamento de imagem e vídeo, criptografia, hash e Machine Learning inference. Os runtimes tradicionais geralmente confiam de forma conservadora no isolamento em nível de hardware para isolar a execução inteira das funções serverless, o que impõe restrições de elasticidade e escalabilidade e não se beneficia do SFI enquanto a função executa código gerido, ou desativam completamente o acesso ao código nativo. Nesta dissertação, propomos o *Faastion*, um novo runtime que combina SFI com isolamento baseado em hardware, confiando apenas no isolamento baseado em hardware quando as funções executam código nativo. A nossa implementação é baseada no GraalVM Native Image, um runtime Java que suporta *isolates* através do SFI, combinado com Intel Memory Protection Keys (MPK) para isolamento baseado em hardware. Em benchmarks e workloads amplamente utilizadas, demonstramos que o Faastion pode executar com segurança funções concorrentes com os benefícios de elasticidade e es-

calabilidade do SFI e o isolamento baseado em hardware necessário para executar código nativo.

Palavras Chave

Computação Serverless; Isolamento de Falhas de Software (SFI); Isolamento baseado em hardware.

Contents

List of Figures

List of Tables

Listings

Acronyms

API	Application Program Interface
AOT	Ahead-of-Time
AWS	Amazon Web Services
JIT	Just-in-Time
IaaS	Infrastructure as a Service
IT	Information Technology
JVM	Java Virtual Machine
JNI	Java Native Interface
PaaS	Platform as a Service
FaaS	Function as a Service
CaaS	Container as a Service
VM	Virtual Machine
OS	Operating System
API	Application Program Interface
CPU	Central Processing Unit
SFI	Software-Fault Isolation
IoT	Internet of Things
MPK	Memory Protection Keys
PKRU	Protection Key Rights Register
PKU	Protection Keys for Userspace
TLB	Translation Lookaside Buffer
Wasm	WebAssembly

WASI	WebAssembly System Interface
GC	Garbage Collector
RTT	Round-trip time

1

Introduction

Contents

The serverless cloud computing model has been steadily gaining widespread adoption due to its ability to harness extreme elasticity and fine-grained billing, all while liberating developers from infrastructure management hurdles [1, 2]. According to Grand View Research [3], the global cloud computing market was valued at $368.97 billion in 2021 and is expected to experience compound annual growth of around 16% from 2022 to 2030.

Several factors are driving the evolution of cloud computing. One of the main drivers is the increasing adoption of cloud computing by businesses of all sizes. Many organizations are moving their Information Technology (IT) infrastructure and applications to the cloud to take advantage of its benefits, such as scalability, cost-efficiency, and flexibility. Technological advances, such as the increasing availability of high-speed internet and the development of new cloud-based services and platforms, also fuel the development of cloud computing. The growth of cloud computing is expected to continue as more businesses embrace the cloud and the demand for cloud-based services and solutions increases. Cloud computing is moving towards fine-grained virtualization to efficiently manage load fluctuations, optimize resource utilization, and enhance performance scalability [4]. To that end, a new and revolutionary solution was serverless computing. It was first coined by Amazon Web Services (AWS) in 2014 when they introduced AWS Lambda [5].

Serverless is often presented as an event-driven Function as a Service (FaaS) programming model. This model dissects applications into nimble, rapidly executing logic units called functions, automatically launched by the serverless platform upon invocation. As a result, serverless computing sparked significant interest among practitioners, fostering the creation of multiple serverless applications across diverse fields, including image and video processing [6], machine learning [7], data processing [8–10], and web-based applications [11], to name a few.

However, serverless functions have different performance characteristics when compared to their serverful or microservices counterparts, particularly regarding memory footprint and execution time. A recent study [12] revealed that most serverless applications run for a maximum of 1 second and consume up to 150 MBs of memory. This granularity represents a departure from the conventional virtualization use cases of microservices and serverful applications.

Traditional hardware and system isolation techniques, such as processes, containers, and virtual machines, are inefficient at protecting individual computational units from one another in the context of fine-grained computations.

Nevertheless, despite the significant gap in terms of execution time, which can extend to hours or even days in the case of microservices and serverful applications, and memory footprint, which may scale up to GBs for microservices and serverful applications, serverless applications still use the same virtualization technology that has been used to power microservices and serverful applications.

1.1 Problem

Privacy and multi-tenancy have always been a problem in Serverless computing. Existing virtualization techniques are still costly for fine-granularity virtualization. One hypothesis being considered to optimize these applications is the concurrent use of one runtime by multiple tenants. Running multiple functions from different tenants inside the same Virtual Machine (VM) or container is still not possible for security concerns. Sharing resources would reduce cold start occurrences and memory footprints. However, it unlocks the possibility of exploitations by any tenant through any security loopholes in the system. For example, a malicious tenant could potentially exploit a vulnerability in the shared runtime environment to gain unauthorized access to the data or functions of other tenants, leading to data breaches, denial of service attacks, or the unauthorized manipulation of critical operations. This hypothesis is on the table since the usual virtualisation process, despite giving security guarantees, also produces a bigger memory footprint and additional overheads such as process creation time.

Nowadays, serverless applications are deployed as tiny functions inside a VM/container. These sandboxes enable multiple function invocations to share the underlying resources. Virtual Machines share the hardware, while Containers share the operating system. These deployment methods provide some isolation, but there are still potential security concerns. When we delve into these functions' execution, we encounter two primary methodologies: sandbox execution and native execution. In sandbox execution, managed runtimes provide a layer of Software-Fault Isolation (SFI) to contain the execution of functions. For instance, memory isolates or WebAssembly [13] sandboxes ensure that multiple functions can co-execute independently within the same environment. The runtime actively enforces restrictions to prevent one function from accessing the memory of another, enhancing security. However, the challenge arises when functions rely on native code. Unlike managed runtimes, native code lacks the protection offered by SFI guarantees. This absence of containment mechanisms leaves functions vulnerable, potentially compromising the security of the entire system.

Most languages used in FaaS are restricted within a controlled environment. Regardless, simple actions, like using specific native-access interfaces, can result in sandbox escapes, leading to potential security issues. While memory safety errors are primarily a problem for applications written in unmanaged languages like C/C++, managed languages such as Java, Python, and Javascript can also suffer from memory safety issues by taking advantage of interfaces that allow them to escape the sandbox. For example, in Java, the Java Native Interface (JNI) enables applications to execute native code within a Java application, potentially compromising the security protections provided by managed languages. In addition to the risks posed by native code execution, recent hardware features like Memory Protection Keys (MPK) [14] show promise in securing such execution and preventing unauthorized memory access. However, the limitation of MPK to only 16 domains presents a significant challenge in environments with high scalability demands, such as serverless architectures. This limitation emphasizes the

need for scalable solutions that effectively address security concerns in FaaS environments, regardless of the programming language.

1.2 Objectives

Our main objective is to advance toward the ability of FaaS platforms to support multiple tenants on a single process without compromising security or performance. To do this, we aim to provide fine-grained virtualization while maintaining high levels of throughput and low latency. We seek to reduce the memory footprint compared to running separate runtimes for each invocation to increase hardware resource efficiency and lower infrastructure costs. Additionally, we are committed to solving common security monitoring problems by effectively controlling any code that escapes the sandbox. Our solution will be scalable, able to adapt and grow as needed to meet the demands of users. Overall, our goal is to make serverless computing a more secure and efficient option for businesses and individuals.

1.3 Solution

We present **Faastion**[1], an approach for controlling and isolating code execution in a serverless environment, using technologies such as GraalVM's Native Image [15] isolates to ensure that each application has isolated sandboxed memory and MPK to manage and change memory domains for complete (native) memory isolation. We recognize that serverless functions mainly execute in managed code and can take advantage of the security guarantees offered by the compiler and language runtime. Hardware-assisted isolation, specifically MPK, is only needed when applications call native code. It is important to note that determining if a transition from managed to native code is invoked is not straightforward, as it may depend on input data. To address this, we propose a system that dynamically intercepts managed-to-native code transitions and uses MPK to protect the execution, preventing native code from accessing memory from other functions. Overall, Faastion is a language runtime that focuses on capturing code transitions from sandbox execution to native execution and ensuring that native execution is controlled and isolated to maintain security and trust in the runtime.

[1] The name "Faastion" was chosen as an analogy for the word "bastion," which refers to a strong and fortified place designed to protect people or valuable assets.

2

Background

Contents

Cloud computing [16] has become an essential part of modern business and is increasingly relevant due to its ability to provide cost-effective, scalable computing resources to organizations of all sizes. By using cloud computing, businesses can focus on their core operations and applications without worrying about the maintenance and management of their infrastructure, which can be time-consuming and expensive. Cloud computing has also driven innovation in several areas, including data analytics, machine learning, and the Internet of Things (IoT). These technologies rely on the ability to access and process large amounts of data quickly and efficiently, which cloud computing makes possible.

Virtualization is a key element of cloud computing, as it allows for the creation of virtual resources that can be used to deliver cloud services. This section will explore various virtualization technologies that support cloud computing. We will discuss how these technologies work and their benefits to cloud providers and users. We will also discuss the role of cloud providers in delivering these services and the different types of cloud computing models available.

2.1 Cloud service offerings

Cloud computing [17] enables rapid provisioning with little infrastructure maintenance effort. It creates the illusion that computing resources are infinitely available on-demand, along with the ability to pay for their use in the short term. There is a trend towards moving more responsibility to the cloud provider, which allows developers to focus on business logic. Some of the services offered are Infrastructure-as-a-Service, Platform-as-a-Service, Container-as-a-Service, and Function-as-a-Service, as detailed in Figure 2.1.

2.1.1 Infrastructure as a Service (IaaS)

IaaS [18] is a cloud computing model in which a third-party provider offers Hardware, including servers, storage, and networking, along with associated software, as a service over the Internet. IaaS virtualizes the Hardware and offers virtual machines as a service, providing users with the ability to provision resources on demand without any long-term commitment. IaaS enables developers to choose the level of abstraction they desire, with the user being responsible for the operating system, applications, and runtime environment. This flexibility makes IaaS an evolution of traditional hosting, allowing businesses and organizations to scale their computing resources up or down as needed.

2.1.2 Platform as a Service (PaaS)

PaaS [19] is a cloud computing model that provides a platform for developers to create and manage applications and services. PaaS offers a range of tools and technologies, such as development environ-

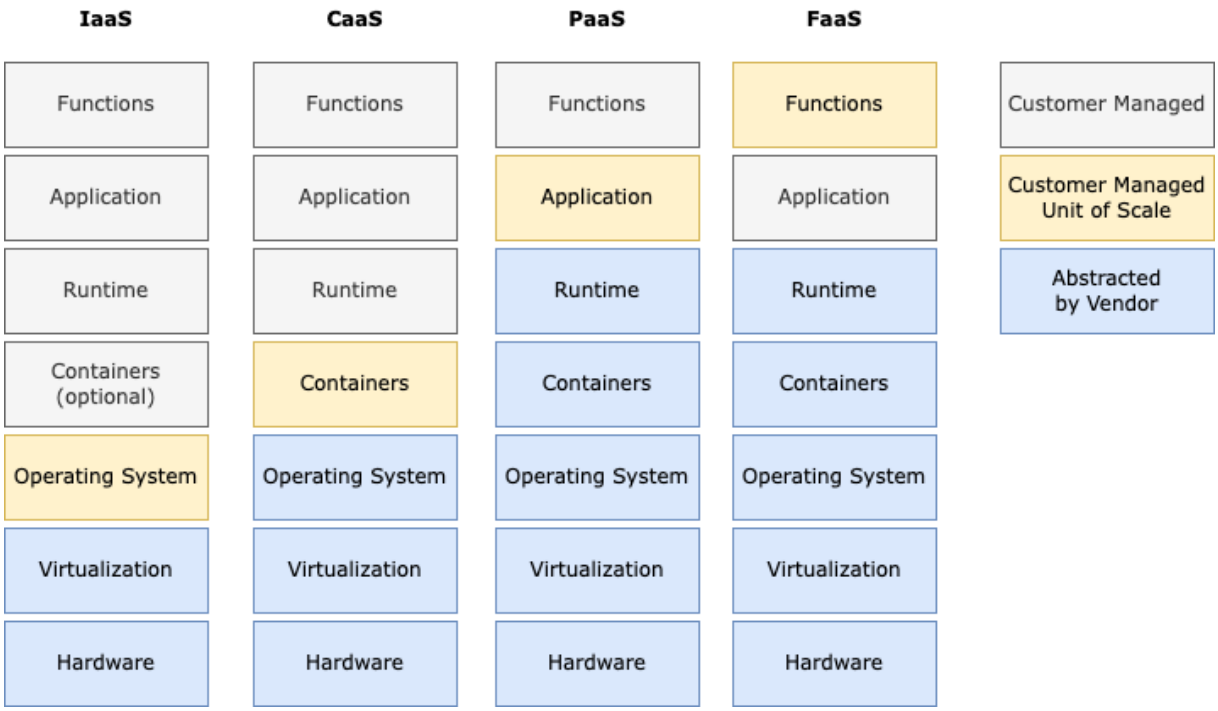


Figure 2.1: Abstraction level stack.

ments, database management systems, and middleware, that developers can use to build and deploy their applications. Providers take care of the underlying infrastructure, including the hardware and operating systems, allowing developers to focus on building and managing their applications. However, this can also be a drawback, as developers may not have the flexibility to install custom system dependencies or access a wide range of technologies.

2.1.3 Container as a Service (CaaS)

CaaS [20] is a cloud computing service that allows users to run containerized applications in a cloud environment. CaaS provides users access to a platform and infrastructure for deploying, managing, and scaling containerized applications. It is common to classify CaaS as a subset of IaaS. Instead of using VMs as the main deployment unit, CaaS uses containers as the primary resource. CaaS also typically includes an orchestration platform to help manage the containers and ensure they run optimally.

2.1.4 Function as a Service (FaaS)

FaaS [21] is an event-driven architecture that is known for its simplicity and has earned the nickname ”serverless” architecture. One of the main benefits of FaaS is that it allows users to focus on the application code without worrying about infrastructure management. Additionally, users are only charged for

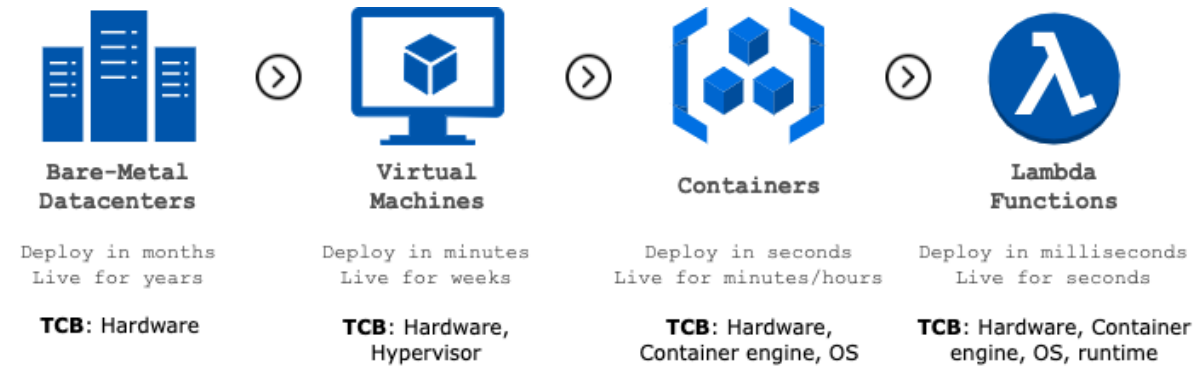


Figure 2.2: Serverless evolution (TCB stands for Trusted Computer Base and represents the continuous growth of components users need to trust across every new stage).

the actual consumption of resources rather than for idle computation time or declared resources. FaaS can automatically and independently scale up or down functions based on demand. Code deployment takes milliseconds. However, it is important to note that FaaS applications are composed of multiple microservices, so it is necessary to consider their latency. Cold-start latency is another potential issue, as each invocation requires starting a new virtual machine or container. Security can also be a concern since multiple customers' functions may run on the same server.

Elastic and scalable virtualization techniques are crucial for FaaS platforms to handle dynamic workloads and meet performance demands. Since FaaS applications rely on the automatic scaling of functions based on demand, the underlying virtualization infrastructure must be able to rapidly provision and de-provision resources in response to fluctuations in workload. Without such techniques, the platform may struggle to maintain responsiveness during peak usage periods or inefficiently allocate resources during periods of low demand, leading to increased costs or degraded performance. Additionally, scalable virtualization techniques ensure that FaaS platforms can accommodate the growth of microservices-based applications without sacrificing performance or reliability. Implementing these techniques enables FaaS providers to offer users a more seamless and cost-effective serverless experience.

A visual representation of the evolution of serverless computing is depicted in Figure 2.2.

2.2 Virtualization technologies

IBM [22] pioneered virtualization [23] more than 60 years ago as a method of logically partitioning mainframe computers into distinct virtual machines (Section 2.2.1). The development of distributed computing led to the temporary decline of virtualization between the 1980s and 1990s. By the 1990s, x86 servers had become the industry standard due to the wide adoption of Windows and Linux. New IT infrastructure and operation challenges have arisen as x86 server deployments grew. Companies were bound to underuse physical hardware because each server could only run one vendor-specific task.

Besides that, most of their computing infrastructure must remain up and running, which results in power consumption, cooling, and facilities costs.

To overcome these difficulties and turn x86 systems into a general-purpose, shared hardware infrastructure that provides isolation, mobility, and Operating System (OS) choice for application environments, VMware [24] introduced virtualization to x86 systems in 1999. It was at this point that virtualization took off.

2.2.1 Virtual Machines (VMs)

Virtual machines [25] purpose is to improve the efficiency of computer resource sharing and maximize available machine capacity, eliminating costs associated with buying or maintaining underused servers. Virtualization allows users to run multiple OS instances on a single physical platform. Each instance is an isolated environment, meaning they cannot tamper with anything outside their box. The application code deployment still requires the installation, configuration, and OS setup beforehand. As virtualization grew more popular, service providers quickly began supporting VMs, and new technologies, such as containers, emerged.

2.2.2 Containers

Containerization [26] is the packaging of software code with all its necessary components and dependencies. Containers are lightweight and only incorporate high-level software, making them far faster to alter and iterate than VMs. Underneath the OS layer, all containers share the same underlying hardware system, hence it is feasible for an exploit to escape one container and affect the shared OS. They simplified the process of deploying code directly into production. Services can now provide a platform enabling consumers to deploy containers, decreasing the entry barrier. An illustration of both VMs and Containers architecture is in Figure 2.3.

2.2.3 Micro virtual machines

Compared to regular VMs, micro VMs offer enhanced security and isolation while enabling containers' speed and resource efficiency. They cannot communicate with other processes and can only access limited OS resources. Firecracker [27] is a practical example of this concept. It takes a micro VM sandbox and restricts it with Seccomp, cgroup (mentioned in section **??**), and Namespace (mentioned in section **??**) policies. These historical developments contributed to providers offering new kinds of services.

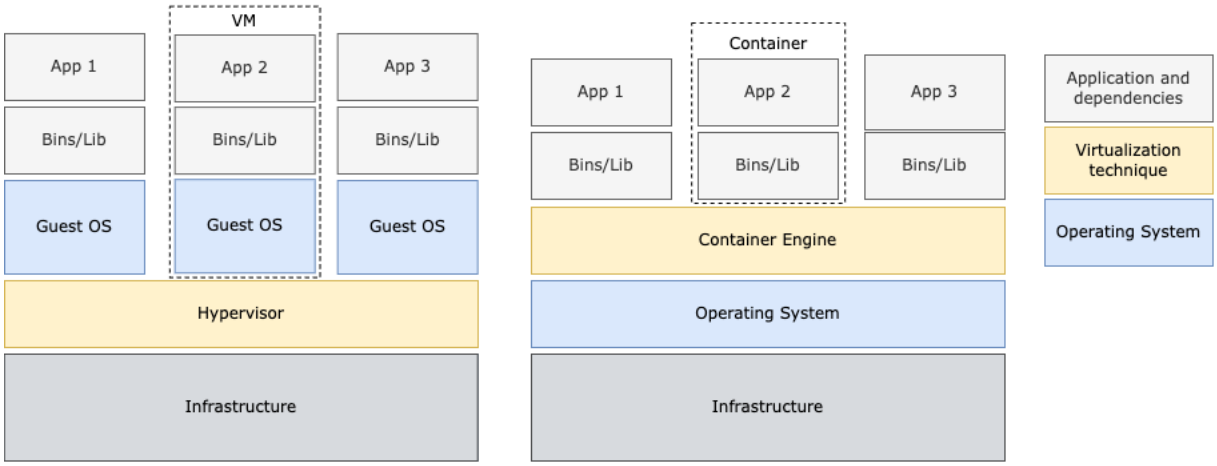


Figure 2.3: Virtual Machine and Container architectures.

2.3 Runtime-level virtualization

Sharing resources is a key benefit of multi-tenancy, but it can also be a security concern. Users often want more fine-grained security isolation for each function at the system level, which can be challenging to provide while maintaining a short startup time. Existing isolation techniques can reduce startup times to a few milliseconds, but it still needs to be determined whether they offer the same level of security as traditional virtual machines. Ongoing research and development focus on finding effective isolation mechanisms with low startup overhead. On the positive side, using provider management and short-lived instances in serverless computing allows for faster patching of vulnerabilities. Some users may prefer physical isolation to protect against co-residency attacks, and recent hardware attacks have made reserving a whole core or physical machine more appealing. Cloud providers may offer a premium option for users to launch functions on physically dedicated hosts.

Multi-tenancy is a software architecture in which a single instance of a software application serves multiple tenants. In this model, each tenant has access to the same software, but each tenant's data is kept separate and secure. Fine-grained security refers to the level of security isolation provided to each tenant. In a multi-tenancy environment, it is important to ensure that each tenant's data is kept separate and secure and that one tenant cannot access or modify another tenant's data. This can be a challenge in a multi-tenancy environment, as it requires maintaining a short startup time while also ensuring that the execution environments are not cached in a way that shares state between repeated function invocations. Fine-grained security refers to providing this level of security isolation at a very granular level, such as at the function level. In practise, one way to integrate security with efficiency is to run different applications on the same runtime, fully isolated. There are many techniques and tools that, combined, could help us achieve this integration.

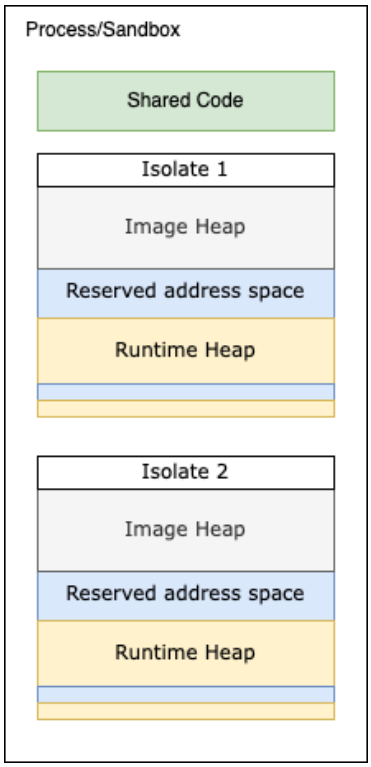


Figure 2.4: Isolate's memory management.

2.3.1 Memory Isolation

Memory isolates are a virtualization feature that allows multiple applications to share the same runtime and efficiently use hardware resources. An *isolate* is a sandboxed execution environment used to isolate code and resources from other *isolates* within a single virtual machine instance.

Each *isolate* has its own memory space, thread pool, and set of loaded libraries. They run in a separate execution context from other *isolates*, as shown in Figure 2.4. This allows multiple language runtime environments to run concurrently within a single virtual machine instance without interference. This concurrency can improve applications' scalability and resource utilization by allowing multiple workloads to share a single VM instance.

Isolates offer several other benefits. For example, when an *isolate* is torn down, its allocated memory is automatically freed without needing garbage collection. This can improve the performance and efficiency of applications. Additionally, *isolates*' memory isolation guarantees that objects from isolates associated with different users cannot be accessed, which helps to ensure the security and privacy of data within the virtual machine.

Virtualizing the runtime will enable platforms to pack additional tenants per server, reducing infrastructure costs. Furthermore, it will reduce memory footprint since different tenants share some dependencies (i.e. libraries).

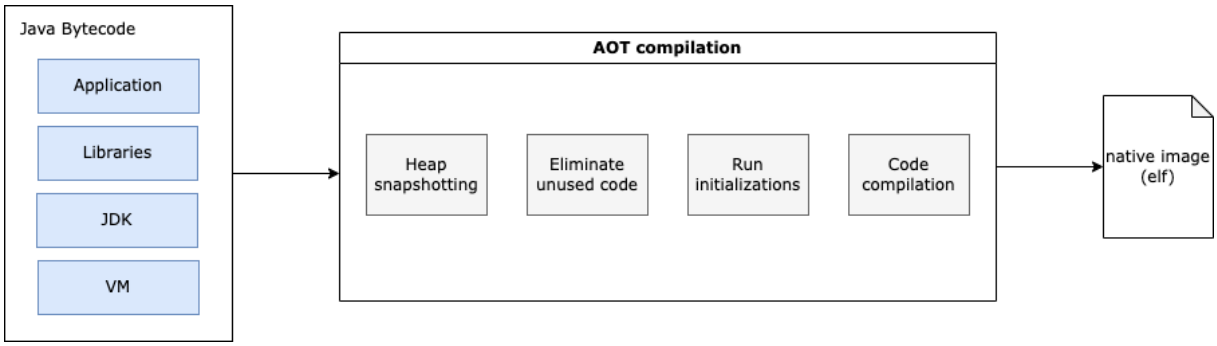


Figure 2.5: Native Image creation.

Despite all these benefits, memory isolates only offer SFI when executing managed code. However, when executing native code, these memory isolation guarantees do not apply. Native code operates directly on the hardware and memory, bypassing the managed runtime's protections. This can lead to potential security vulnerabilities, as the native code can perform unsafe memory operations, access restricted areas, or corrupt memory, thereby compromising the isolation provided by memory isolates.

GraalVM's Native Image [15] is an Ahead-of-Time (AOT) compiler used to create standalone executables from Java applications and includes support for *memory isolates*. It uses a fraction of the resources that the Java Virtual Machine (JVM) requires, making it cheaper to compile and run. The AOT compilation statically analyses the application's code and performs aggressive optimizations such as eliminating unused code, heap snapshotting, and static code initializations. These optimizations can result in faster start-up times, better overall performance, a smaller memory footprint, and lower resource consumption.

One of the main benefits of *Native Images* is their use of the "closed world assumption," which means that they complete the build-time monitoring of all bytecodes that could be called during execution before creating the native image. This means there will not be any additions or changes to the bytecode at runtime, resembling a closed world (no changes, additions, or losses). Consequently, dynamic language capabilities are not supported, which is a security benefit. Among these capabilities are Java Reflection and Java Unsafe. Java Reflection enables a program to discover the values of field variables and change them, while Java Unsafe allows developers to circumvent the safety guarantees provided by Java. Figure 2.5 shows the *Native Image* proceedings to create an executable.

2.3.2 Resource Isolation

Resource isolation is fundamental in modern computing environments to ensure fair allocation of system resources and prevent interference between different tasks or applications. Effective resource isolation mechanisms are essential for maintaining system stability, security, and performance across diverse workloads. Features such as control groups (cgroups) and namespaces encapsulate applications and

enforce resource constraints.

Control Groups (cgroups) are a Linux kernel feature that enables fine-grained control over resource allocation for groups of processes. By assigning resource limits to cgroups, system administrators can prevent processes from monopolizing system resources. For instance, one could enforce CPU and memory limits to ensure fair sharing among multiple concurrent applications.

Namespaces provide isolation for system resources, such as processes, network interfaces, and file systems, by creating separate instances of these resources for each container or application. Network namespaces, for example, enable each container to have its own virtual network stack, preventing network conflicts and enhancing security.

Resource isolation is a critical aspect of modern computing environments, involving various technologies and approaches to ensure fair allocation and secure execution of applications.

While these are valuable mechanisms, it's important to note that the detailed implementation of such mechanisms is beyond the scope of this work. However, in the context of our specific application or system, we can speculate that each sandbox or execution environment could be enclosed within its own cgroup and namespace. This would enable precise control over resource allocation and isolation, ensuring that each component operates independently and securely within its designated boundaries. Further exploration of this implementation detail could be a topic for future research or system design considerations.

2.3.3 System calls and Libc

Applications primarily interact with the operating system through system calls, the fundamental interface between an application and the kernel. These system calls can be invoked directly by the application or, more commonly, through the C standard library (libc), which provides a higher-level API.

Libc, including implementations like Musl libc, serves as a mediator, offering standard functions that internally invoke the necessary system calls. This abstraction simplifies application development but also means that libc can be a critical control point for system interactions.

In the context of sandboxing, where the goal is to confine applications to a limited set of resources and capabilities, it is essential to intercept both direct system calls and those made through libc. By doing so, we can enforce security policies and prevent unauthorized actions. Seccomp plays a vital role here by restricting the system calls available to an application, thereby reducing the attack surface.

Musl libc's design, which allows each thread's execution and use of libc to be unique, further enhances this security by ensuring resource isolation. This means that even within the same application, different threads do not share the same state, reducing the risk of unintended interactions and potential security breaches.

Combining seccomp with a robust libc implementation like Musl creates a defense mechanism, en-

suring that applications operate within their intended boundaries and mitigating the risk of exploitation through system call interfaces.

2.3.4 Hardware Assisted

MPK offer hardware-assisted mechanisms to strengthen the protection provided by SFI. SFI is a technique used to enforce security policies at the software level by isolating potentially unsafe code and preventing it from impacting the rest of the system. While SFI is effective, it can be enhanced with additional hardware support to improve its reliability and efficiency.

The MPK feature enhances SFI's protection by allowing one to assign specific protection keys to virtual memory pages. These keys control the access rights of different threads to those memory regions, ensuring that only authorized threads can access data. This hardware-level enforcement adds a layer of security, making it harder for malicious code to bypass restrictions imposed by SFI.

However, MPK has a significant scalability issue due to its limitation of supporting only 16 protection keys or domains. This restriction can be problematic in large multi-tenant workloads where applications will require a large number of isolated domains. One must carefully manage each protection key to maximize the security benefits without exceeding the hardware limits.

Moreover, assigning a page to a specific protection domain in MPK involves a costly operation because it requires flushing the Translation Lookaside Buffer (TLB). Frequent TLB flushes can degrade performance, especially in systems with high memory access demands.

Changing a thread's permissions also implies writing to the Protection Key Rights Register (PKRU), which holds the access rights for the current thread's protection keys. Although updating the PKRU register is quick, it introduces additional overhead and complexity in managing thread-specific permissions.

Furthermore, work like libmpk [14] referred that MPK have been found to have vulnerabilities in the protection key assignment and management system. These vulnerabilities can allow protection keys, which control access to groups of pages in memory, to be re-used after they have been deallocated, leading to confusion and potential security issues. In addition, it is possible for attackers with access to an arbitrary write vulnerability to corrupt the protection keys stored in a variable, potentially allowing them to manipulate the permissions of a target group of pages.

Despite these limitations, MPK is a robust tool for enhancing memory protection and security by combining hardware-assisted mechanisms with software-based techniques like SFI, achieving a more secure and efficient environment for modern applications.

3

State of the art

Contents

This chapter examines the efforts of various projects to implement secure multi-tenancy with different approaches and a brief discussion on why we believe they ultimately fell short in achieving this goal.

3.1 Runtime/compiler-based SFI

Photons [28]. Today's serverless platforms initialize and schedule each invocation separately, disregarding the execution of the same code and environments across multiple invocations. A significant portion of the execution time is consumed by initializing each invocation's runtime and application state. Furthermore, there is no memory sharing across invocations, which leads to the replication of large amounts of state, increasing overall memory utilization. These inefficiencies are a natural consequence of strict isolation.

Photons exploit this redundancy while still providing the same serverless abstractions as today's platforms. Photons allow safe runtime sharing across multiple invocations of the same function by the same tenant (functions from the same tenant are assumed to be harmless toward each other).

To support data separation and preserve the serverless abstraction, the authors propose a function loader, a wrapper around JVM that intercepts and instruments the application's bytecode. Using the concept of static tables, the function loader modifies access to global static state at class loading time. These tables are introduced to map static fields, shared between function executions, to local copies accessible through a unique identifier attributed for each invocation, creating private states. Ultimately, static tables grant isolation in a sharing environment with small overheads.

Thanks to this isolation, Photons can reuse application-specific resources and runtime for future invocations of the same function. Fast startups can now occur when a warm container is in memory.

Photons implementation does not handle Java reflection or Java Unsafe, which is a significant security vulnerability. For instance, certain libraries may use reflection to access static fields. This can disrupt the usual order and assumptions of class loaders and bytecodes. In these situations, users must modify the code manually and change access to static fields. Unfortunately, the Photons implementation does not provide automatic handling of these scenarios.

WaVe [29]. WaVe is a secure and fast WebAssembly (Wasm) [13] runtime system. Every Wasm runtime is trusted to implement theWebAssembly System Interface (WASI), which is used to access the filesystem and network directly and to make it possible to run Wasm code across all different OSs. Fundamentally, Wasm relies on automated verification to make sure that the runtime code upholds Wasm's memory isolation guarantees and appropriately limits each sandbox's access to OS resources, tasks that are frequently not accomplished by WASI alone.

WaVe implements memory, file system, and network isolation. It does not rely on developers to put

all the proper safety checks; rather, WaVe uses a single safety policy file enforced with an automatic verifier.

The creators assume that each allocated sandbox has an exclusive directory (root directory) containing all the data it has permission to access, ensuring filesystem isolation. Similarly, to preserve network isolation, WaVe verifies that a sandbox's network access can only make outgoing network connections to addresses present in an allow list created by the host application. When a sandbox makes a hostcall, WaVe dynamically checks if its arguments pointers are all within the sandbox's memory to grant memory isolation.

After all, WaVe still has significant limitations. Despite allowing host applications to run multiple sandboxes concurrently, it cannot guarantee safety if a single sandbox runs multiple threads concurrently, jeopardizing efficiency. Also, Wasm's sandbox, by default, can be exploited, and code can escape from it using WASI or other appropriate Application Program Interfaces (APIs) such as the WebAssembly C/C++ API. The creators did not consider this exploitation, making it non-Multi-tenant proof. Code escaping the sandbox can be particularly dangerous when calling certain native methods, the exact problem of using a language like Java.

FAASM [30]. Most existing FaaS platforms isolate functions in stateless containers. This particular decision presents additional challenges for data-intensive applications. As discussed in Section 3.4, containers introduce data access overhead by forcing state to be maintained externally or passed between function invocations. Besides, the large memory footprint of containers limits scalability since the maximum number of containers is usually capped by the amount of available memory on a machine.

FAASM aims to solve these challenges by introducing a serverless runtime that uses Faaslets (a function isolation mechanism that accomplishes efficient and safe execution on a single machine) to execute distributed stateful serverless applications across a cluster.

A FAASM runtime instance schedules, executes, and maintains a pool of Faaslets. Each Faaslet in the pool has a dedicated thread and a respective address space. Wasm compiles the functions associated with Faaslets, along with their libraries and dependencies, to ensure memory safety and control flow integrity. The CPU cycles of each thread are constrained using cgroups, and network access is limited using network namespaces, granting resource isolation.

Faaslets support shared memory regions within the constraints of WebAssembly's memory safety guarantees. Faaslets accomplish this by extending a function's linear byte array and remapping its pages to create a new shared region. Furthermore, since Faaslets share the same overall address space, they can access shared memory regions, allowing the colocation of functions and reducing prevalent memory footprints caused by previous data duplication. Sharing memory and colocating functions allows more instances to run in one machine and avoids serialization overheads.

The FAASM runtime pre-initializes a Faaslet ahead-of-time and snapshots its memory to obtain a Proto-Faaslet, which can be used to create a new Faaslet instance in hundreds of microseconds, avoiding the time to initialize a language runtime and reducing cold-start times.

Faasm relies on language-level isolation provided by Wasm, which is weaker than container-based isolation. Faasm uses threads and shared memory with software-fault isolation to provide thread-private memory partitions. While this approach can be effective at protecting against certain types of attacks, it is heavier and slower than hardware-based isolation techniques such as MPK's domain checks. Additionally, and as explained in WaVe, section 3.1, Wasm code can call native libraries, potentially compromising the security of a multi-tenant environment.

3.2 VM-based Isolation

REAP [31]. REAP aims to tackle the cold start problem inherent in serverless computing environments. Serverless architectures, such as AWS Lambda, Azure Functions, and Google Cloud Functions, rely on temporary containers or virtual machines to execute functions. Each time the system invokes a function, it must initialize the environment, loading the necessary code and dependencies. This initialization process, known as a cold start, can introduce significant latency, negatively affecting performance and user experience. Cold starts are especially detrimental for applications requiring real-time responsiveness or high interaction levels, as they undermine the expected on-demand scalability and availability of serverless functions.

To address these challenges, the authors optimized the snapshotting process used in serverless platforms. Snapshotting involves capturing the state of a VM or container at a specific time, which can then be quickly restored, thus reducing the initialization time for subsequent function invocations. The solution consists of several components. Firstly, they developed a benchmarking suite to evaluate the performance of existing snapshot mechanisms across various serverless platforms. This benchmarking helps identify the main factors contributing to cold start latency. Based on the insights from benchmarking, the authors have proposed several optimizations. They aim to reduce the size of snapshots, which will decrease the time needed to save and restore these states. They improved memory management techniques to make capturing and restoring snapshots more efficient. One of the techniques is page-sharing, which allows multiple VM instances to share memory pages. This works by identifying and consolidating identical memory pages across different VM instances. These strategies simplify the snapshotting process, making it more efficient and less resource-intensive, thus minimizing cold start times.

The proposed solutions show potential, but they also have some limitations. Some suggested optimizations could increase resource usage, like memory or CPU. In situations where resources are limited,

these increases could cancel out the performance improvements gained from reducing cold start times. Furthermore, the specific techniques and optimizations may not work well for all serverless workloads. Applications with unique characteristics or dependencies may not benefit as much from these proposed improvements.

3.3 Process-based isolation

Nightcore [11]. Existing FaaS systems share a generic high-level design. Frontends receive all incoming traffic and forward it to separate backend servers for fault tolerance purposes, requiring significant invocation latencies that include at least one network round trip, making them a poor choice for latency-sensitive microservices. Nightcore's goal is to make it possible to efficiently support this type of microservices by achieving low invocation latency overheads and high invocation rates while maintaining low CPU usage. The authors introduce two concepts, internal and external function calls. Executing a microservice generates internal function calls. An external function call is generated by the client and received by a gateway.

Nightcore's engine responds to function requests from both the gateway and a runtime library within function containers. Function containers provide isolated environments and run worker processes that receive requests from the engine and execute application code. In order to eliminate a trip to the gateway, the internal function calls directly contact a dispatcher that adds them to a queue to be later executed on the same backend server, making most communications local and, therefore, more efficient. This improvement enables Nightcore to achieve lower overall latency. In order to reach high invocation rates, Nightcore's engine uses event-driven concurrency, which makes it handle many concurrent requests with very few threads.

The main drawback of Nightcore is using different containers to run different functions, hinting that some memory could be shared between functions. However, this does not happen because the creators did not find a way to isolate the memory used by each function fully.

SOCK [32]. Programmers' posture towards their code has seen several modifications due to efforts to improve developer elasticity and scalability. Serverless platforms provide reasonable solutions to improve developer velocity, but they also create new infrastructure problems. Specifically, their techniques tend to increase cold start occurrences. Most serverless platforms currently wait minutes to recycle idle lambda instances. Furthermore, reusing code introduces additional startup latency from library loading and initialization.

The authors performed two studies to better understand what interferes with efficient cold starts. The first study led to the uncovering of scalability bottlenecks in the network and identified lighter-weight

alternatives. The second study found that, in most Python projects, 36% of imports are to just 0.02% of packages.

These findings led to the implementation of SOCK. It is integrated with the OpenLambda serverless platform and is based on the Python ecosystem. SOCK creates lean containers for lambdas by avoiding the expensive operations identified in the first study that are only necessary for general-purpose containers. SOCK avoids these operations by:

- Using a chroot operation to change the apparent root directory for a running process and its children since it is not necessary to selectively expose other host mounts within the container, making mount namespaces unnecessary;

- Using Unix domain sockets (UDS) for the communication between the OpenLambda manager and the processes within a container. This way, relevant resources, such as namespaces, may be represented as file descriptors, which can be passed via UDS, making network namespaces unnecessary;

- Generating a pool of cgroups (necessary for isolation yet expensive), amassing all overhead during pool generation and evading it when provisioning a new container.

SOCK also provisions Python handlers using a generalized Zygote-provisioning strategy to avoid the Python initialization costs identified in the second study. Zygote provisioning is a technique where new Zygotes (processes) are launched as forks of an initial Zygote that already has pre-imported libraries that subsequent programs are likely to require. This technique prevents child processes from doing the same initialization tasks more than once and utilizing excessive memory. These optimizations improve the container boot process to achieve cold starts in the low hundreds of milliseconds.

SOCK's main drawback is that the Zygote provisioning strategy only works for simple runtimes like Python. For more complex runtimes, such as JavaScript engine or JVM, the root runtime would have to be restarted to use the Garbage Collector (GC) and Just-in-Time (JIT) compiler threads that are typically in the background.

SAND [33]. SAND tackles complex workflows dispatched to serverless platforms. The authors discovered that the total runtime drastically outweighs the functions' execution time, implying the presence of additional overheads.

The authors claim that one reason behind these overheads is that today's serverless platforms execute each application function within separate containers. By itself, this leads to two standard practices (each with a drawback), either the usage of cold containers (long startup latency) or the usage of warm containers (resource inefficiency due to idle execution).

The goal of SAND is to significantly reduce latency overhead and improve resource efficiency by addressing both of these issues. The authors began by designing a fine-grained application sandbox mechanism in which each application (set of functions) has its own sandbox on a given host. When a sandbox hosts a particular function, it runs a dedicated OS process loaded with function-associated code and libraries and waits for event messages. Upon receiving an event message, the process creates an instance to handle the request by forking itself. This approach makes resource allocation and deallocation more efficient, as the OS automatically reclaims resources when a function is terminated. It also reduces memory footprint by sharing initialized code, as each forked process inherits everything from its parent. In addition, the forking strategy is relatively fast and lightweight and grants memory isolation.

SAND, nonetheless, has limitations. Multiple instances of an application's functions are executed concurrently as separate processes inside the same sandbox, meaning that different functions may compete for the same resources and interfere with one another's performance. Besides, languages that use JIT compilers and Garbage Collectors can lead to memory divergence, which occurs when different processes have different views of the same memory region.

3.4 MPK-based isolation

Faastlane [34]. Generally, FaaS applications are based on complex workflows. A workflow specifies how a set of functions must process input in an orderly manner. These functions interact by passing transient states between each other, meaning they are mutually dependent.

Most providers execute workflow functions in separate containers, whether or not they belong to the same workflow instance. Therefore, copying the transient state across containers or transmitting it via cloud-based storage services is necessary, contributing to function interaction latency. Unfortunately, when workflow instances become more complex, this setup struggles to scale accordingly, making it ill-suited for many applications.

Faastlane aims to map functions in a workflow instance to threads that share a process address space, thus providing faster communication and avoiding function interaction latency. However, using threads for efficient state-sharing compromises the isolation of sensitive data and the concurrency in parallel workflows. So, Faastlane's runtime separates every thread into its portion of address space. It uses MPK to provide thread-granularity memory isolation at low overheads for functions that share a virtual address space.

Furthermore, in order to accommodate for interpreted languages that use a global interpreter lock (prevents concurrent execution of threads), Faastlane utilizes a workflow composer. This composer is a static tool that analyzes the workflow structure and forks processes instead of threads wherever the

workflow allows parallelism.

Faastlane's main drawback is its hardware limitations. The MPK's protection key rights register (PKRU) can only manage up to 16 protection keys, meaning that, at most, 16 functions can run concurrently, limiting scalability. Despite this, Faastlane remains our biggest competitor because we both leverage MPK within a single process runtime to achieve similar objectives.

ERIM [35]. The ERIM paper introduces a novel approach to ensuring the integrity of the Linux kernel in real-time, particularly crucial for safety-critical fields like automotive systems and industrial control.

ERIM's framework enables real-time execution of security-sensitive code within the Linux kernel, leveraging real-time scheduling policies to minimize performance impact. It achieves this by integrating security checks directly into the scheduling process. When a security-sensitive task is scheduled for execution, ERIM dynamically adjusts the scheduling parameters to prioritize its processing. This ensures that critical operations are executed promptly while still maintaining system integrity. By tightly coupling security checks with the scheduling mechanism, ERIM effectively prevents disruptions to real-time requirements, ensuring that security measures do not compromise system performance.

Despite its advancements, ERIM does not directly address the multi-tenancy problem inherent in serverless environments. This challenge involves efficiently sharing resources among multiple users while maintaining isolation and performance guarantees. Nonetheless, ERIM's insights into efficient memory isolation and execution within a single instance lay the groundwork for potential solutions to the multi-tenancy problem. Future research could build upon ERIM's principles to develop more comprehensive approaches to multi-tenancy in serverless architectures, bridging the gap between real-time integrity and scalable, multi-user environments, such as our project.

Jenny [36]. Jenny aims to address syscall filtering challenges for Protection Keys for Userspace (PKU)-based memory isolation systems. These systems are increasingly used in modern web browsers and cloud computing to improve performance through in-process containers. However, existing research has identified significant flaws in how these systems manage syscalls, making them vulnerable to various attacks. By uncovering previously unknown PKU-related syscall attacks and comparing different syscall interception mechanisms, the authors seek to demonstrate the practicality and security of syscall filtering for PKU systems.

In response to these challenges, the authors introduce Jenny as a solution that provides comprehensive and efficient filter rules for protecting a PKU sandbox. Jenny offers various filtering capabilities, such as file system virtualization, in-process namespaces, and browser site isolation. It includes a faster mechanism for syscall interception tailored to the needs of PKU systems. Jenny enables filtering on the same thread, simplifying impersonation of syscalls, nested filtering, and signal handling, ultimately

showcasing that syscall filtering for PKU systems can be practical and secure.

While Jenny shows promise, there are inherent limitations. The performance overhead, although minimal in testing, could vary with different applications and usage patterns. Moreover, similar to Faastlane, Section 3.4, the number of protection keys supported by Intel MPK is limited, which imposes constraints on the number of worker threads that can be used in complex scenarios.

3.5 Discussion

In this analysis, we compare all the works reviewed using four different perspectives. The limitations and drawbacks identified in the state-of-the-art (Section 3) highlight that our solution is the only one that addresses all four perspectives. To the best of our knowledge, no existing tools or platforms can provide this unique combination of security and efficiency. The comparison will be based on the following four perspectives:

- **Invocation Colocation**: allows the concurrent execution of different tasks on the same process runtime;
- **Multi-tenant Support**: supports multiple tasks from different tenants, fully isolated from each other;
- **Memory-efficient Solution**: uses a minimal amount of memory to perform a given task without replicating unnecessary components;
- **Scalable Solution**: can handle a growing number of tasks without becoming inefficient or requiring a significant increase in resources.

As represented in Table 3.1, ERIM fails in the ”invocation colocation” part, which refers to running multiple tenants concurrently and efficiently. Although it improves real-time integrity, it does not directly address multi-tenancy challenges such as in serverless environments. Nonetheless, ERIM’s insights lay the groundwork for potential solutions to multi-tenancy issues such as Faastion.

REAP, SOCK, and SAND all lack certain features we classify as important. They focus on isolating tasks within individual processes, VMs, or containers, not addressing the ”invocation colocation” aspect. Regarding memory efficiency, REAP relies on page-sharing mechanisms. While page sharing can reduce memory overhead by allowing multiple VM instances to share memory pages, it requires significant memory resources to maintain multiple copies of the application state. SOCK and SAND also fall short in this aspect. SOCK attempts to address the issue of memory replication by forking an already initialized process. SAND, on the other hand, runs each tenant on a unique process, resulting in memory duplication.

Table 3.1: Comparing Faastion with similar works.

	Invocation Colocation	Multi-tenant Support	Memory-efficient Solution	Scalable Solution
Photons [28]	✓	✗	✓	✓
Faastlane [34]	✓	✓	✓	✗
FAASM [30]	✓	✗	✓	✓
REAP [31]	✗	✓	✗	✓
Nightcore [11]	✓	✗	✓	✓
Wave [29]	✓	✗	✓	✓
SOCK [32]	✗	✓	✗	✓
SAND [33]	✗	✓	✗	✓
ERIM [35]	✗	✓	✓	✗
Jenny [36]	✓	✓	✓	✗
Faastion	✓	✓	✓	✓

Photons, FAASM, Nightcore, and WaVe, have security issues when code escapes the sandbox, potentially endangering different tenants. This suggests that these methods are weak in protecting the isolation of different tenants, which is a critical concern in a multi-tenant system.

Faastlane and Jenny are not scalable solutions due to a MPK limitation of 16 tenants that can run concurrently. In contrast, Faastion was thoughtfully designed with these issues in mind and does not exhibit these weaknesses. It addresses the concerns of memory replication and duplication by utilizing isolates. Additionally, Faastion delivers excellent performance and robust isolation, leveraging MPK to protect tenants against harmful executions. We significantly enhance its scalability due to a strong insight described in Section 4.4.

4

Solution Design

Contents

This Chapter explores the details of our proposed solution, Faastion (strategy is depicted in Figure 4.1). We will explain why we have chosen to utilize Intel MPKs and discuss strategies for addressing any limitations they may have.

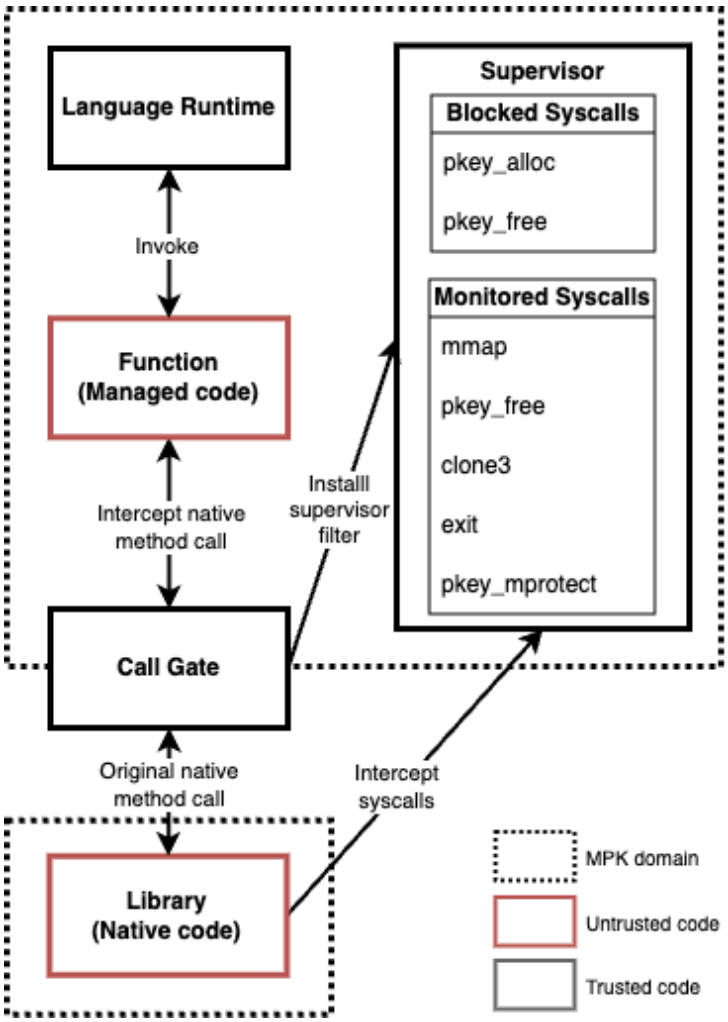


Figure 4.1: Faastion's Basic strategy.

4.1 Design overview

Faastion utilizes the built-in sandboxing capabilities of widely used serverless languages such as Java, JavaScript, and Python to enhance security. Although these languages have sandboxes, code can potentially break out, posing a security risk. Potential escape methods include native method calls and reflection, which can circumvent the sandbox's protections. To address this, Faastion incorporates strong control measures to prevent and manage such escape attempts.

Faastion's security strategy revolves around the ”call gate,” a mechanism designed to monitor and

control interactions attempting to bypass the sandbox. Faastion begins by searching for an available MPK domain to create an isolated execution environment. Once it finds an empty MPK domain, Faastion installs a seccomp (secure computing mode) filter. This filter restricts the system calls that untrusted code can make, ensuring that only safe and authorized interactions are allowed. Faastion assigns a dedicated supervisor to monitor the system calls of the executing thread, which will operate within the constraints defined by the seccomp filter. The executing thread's domain is then switched to the isolated MPK domain. This step ensures that the thread operates within a controlled environment, minimizing the risk of unauthorized access to system resources. With the supervisor in place and the execution domain secured, Faastion calls the original native method. Throughout this process, the supervisor maintains control over the execution, ensuring that any system call made by the native method adheres to the security policies enforced by the seccomp filter.

By employing this layered approach, Faastion ensures comprehensive isolation of potentially unsafe code while maintaining control over its execution. This method prevents security concerns from code escaping the sandbox since it leverages real-time monitoring and intervention if any security policies are violated. Through these measures, Faastion provides a robust and secure environment for executing serverless functions, effectively mitigating the risks associated with native method calls and reflection in sandboxed languages.

4.2 Trust Boundaries

In our architecture, trust boundaries are carefully delineated. The language runtime and compiler are considered trusted entities, exempt from monitoring during interactions with function code. This trust extends even to components written in non-managed languages like C/C++. However, user-provided code is treated with caution due to the potential for bugs or malicious code. This type of code undergoes thorough scrutiny to ensure system integrity and security. It is important to note that while tenants do not need to trust each other, they must trust the runtime. If the runtime has vulnerabilities or bugs, it could compromise the security of all tenants.

4.3 Static Analysis

In our current implementation, we focus exclusively on Java applications. Upon code submission, we use Javassist, a Java library for bytecode manipulation, to inspect the application's classes. Javassist allows us to interact with the bytecode directly and perform various transformations and analyses.

One of the primary checks we perform is to determine whether the method calls are native or not. After identifying native methods, we further verify if these methods are part of the trusted native methods

from the Java runtime. Trusted native methods are those provided by the standard Java libraries and are considered safe. We compare the identified native methods against a predefined list of trusted native methods. If a native method is not in the list of trusted methods, we flag it for additional handling. For any native methods that are not trusted, we generate a code snippet, referred to as a call gate, which will replace the original native method by serving as a wrapper. It applies necessary isolation measures to safeguard against potential security risks, ensuring proper isolation beyond Java's inherent sandbox capabilities. After the isolation measures are in place, the call gate executes the original native call. After Javassist generates call gates for all identified non-trusted native methods, it integrates them into the application's bytecode. This ensures that the application runs while maintaining enhanced security protocols.

Using Javassist, we ensure that Java applications maintain security when interacting with native code. This process is crucial for protecting against potential vulnerabilities of executing untrusted native methods.

4.4 MPK isolation

MPKs allow us to partition the virtual address space into 16 sets of pages, theoretically enabling the safe concurrent execution of up to 16 tenants within the same process. This claim is only valid if all users run in their own domains. However, to support a greater number of concurrent tenants, we leverage this capability by implementing runtime-level virtualization to create a scalable solution. This involves virtualizing a single runtime environment and enabling multiple tenants to run concurrently, with all initially assigned to the same domain.

Although this approach may expose all tenants to shared risks due to the common domain, we mitigate these risks by utilizing MPKs to switch domains. Specifically, when an application needs to execute native code, it triggers a domain switch to isolate its execution environment. This domain-switching mechanism ensures that potentially harmful operations do not affect other concurrent applications, enhancing security. By employing this strategy, we effectively manage a larger number of tenants within a single runtime environment, balancing performance and security. This method allows for efficient resource utilization and improved scalability in multi-tenant systems.

4.5 Data Structures

We employ two primary data structures to manage our system. The first is a map that tracks the association between libraries and applications. The second is an array that monitors domain occupation by threads. To populate the map, we intercept the `dlopen()` call via LD_PRELOAD, which is invoked by

Java's `System.loadLibrary()` method. Our custom `dlopen()` function searches the /proc/self/maps file for the loading library, capturing its start address and size. Although this process can be computationally intensive, it only needs to be performed once, significantly speeding up all subsequent calls.

At the beginning of the runtime, we perform a series of initializations. One initialization to mention specifically is the initialization of the stacks. This step proves required, as the default stack exclusively accommodates threads operating within the monitor domain. Any access attempts into the stack from an executing thread within other domains could result in a segmentation fault. To prevent this, Faastion prepares a domain-specific stack at runtime's initialization. This involves allocating distinct memory regions for each domain, which are then protected using the `pkey_mprotect()` mechanism.

4.6 Special domains

To improve clarity, we specify the role of each domain. We utilize three types of domains:

1. The monitor domain, also known as domain 0, functions as the default domain where the Graalvisor is launched and operated. This domain has complete access rights and permissions, making it very powerful. Only trusted or untrusted but managed code, such as Java, should execute in it.

2. The zombie domain, designated as domain -1, acts as a repository for applications with revoked permissions. This domain is necessary because we are unable to restore library permissions to their original state (monitor). Furthermore, keeping these applications in their previously assigned domains could pose a security risk, as a new thread selecting the same domain could potentially access residual data.

3. Domains 2-15 operate as standard domains where untrusted code undergoes execution, resulting in 14 domains for us to manage.

4.7 Optimizations

A more efficient way of managing permissions for native methods is to do it lazily, taking advantage of the Graalvisor optimization of reusing threads. Instead of setting permissions at the start and revoking them at the end of each method, we dynamically assess whether the current application matches the preceding one executed within the same domain. If there's a match, no modifications are necessary. However, if the applications differ, the cached application permissions are revoked, relocating it to domain -1, and is replaced by the current application in the cache. We then configure the permissions of the current application for the designated domain.

By doing this, we no longer need to waste time revoking permissions. With thread reuse facilitated by Graalvisor, when the same thread executes the same application, we retain the existing permissions from the cached application, removing the need to modify permissions for the target domain. Consequently, this streamlines the search for available domains and the permission adjustment process. Under optimal conditions, this strategy achieves performance akin to utilizing isolates (no isolation), thus enhancing overall efficiency.

4.8 Musl and Seccomp

To understand Faastion's proper functioning and security, we must discuss the roles of Musl and Seccomp. The delayed loading of Musl's libc in response to function calls is a pivotal feature that guarantees the safe execution of untrusted methods. To better understand this concept, one important insight is that the standard C library (libc) is initially set to be accessed by threads executing in the default domain (known as domain 0 or "monitor").

Before invoking a native method within our system, we take a critical step to modify the thread's domain through the PKRU. This modification is crucial for guaranteeing memory isolation for the untrusted method. Without this mechanism, there is a substantial risk that it could attempt to access a function from the standard libc, resulting in a segmentation fault and a system crash. This is where Musl comes in as a necessary solution. For our system to function seamlessly and securely, we compile every native library with Musl. One fundamental principle of Musl is that it defers the loading of its libc until a function from the compiled library is invoked. This contrasts with the standard libc, which is loaded into domain 0 at the beginning of execution.

The significance of this approach is that Musl's libc is loaded within the domain of the calling thread when calling a relevant function. This ensures that even if an untrusted method initiates a libc function, it will not result in a system crash. Instead, the Musl libc operates within the context of the invoking thread, preserving the entire system's stability and security.

The Seccomp tool provides us visibility into the actions performed by native code. We use Seccomp eBPF filters in each native executing thread to prevent undesired behavior. These filters restrict the system calls that a process can make, which reduces the attack surface of potentially malicious code. Seccomp is a tool that allows a supervisor thread to receive notifications whenever an untrusted thread executes a filtered system call. The supervisor has complete control over how to handle the situation. This is how we ensure that every untrusted native execution has limited power.

4.9 Domain Management

It is crucial to effectively manage resources and allocate domains within the call gate to maintain security and operational efficiency. This section will explore the complex processes involved, beginning with the role of supervisors and their critical functions. To gain a better grasp of the upcoming concepts, please refer to Figure 4.2.

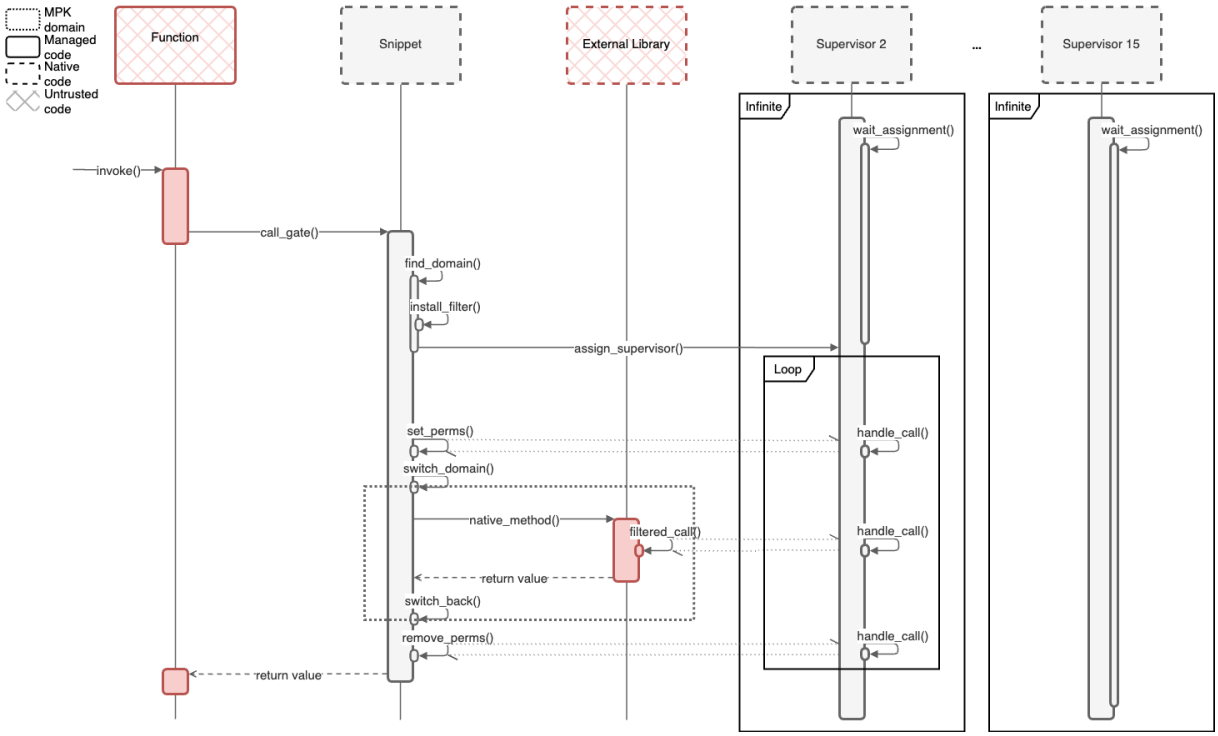


Figure 4.2: System Workflow.

4.9.1 Supervisors

Supervisors are essential because, although Javassist allows us to track code transitions (Section 4.3), we lack visibility into the exact actions performed by native code. To mitigate this limitation and prevent undesired behavior, we install a Seccomp eBPF filter (described in Section 4.8) in each native executing thread. This filter enables the assigned supervisor to monitor and control the native code's actions effectively. The installed seccomp filter restricts almost all system calls, except for a few that are important for us to monitor, ensure proper isolation of resources, or prevent their use altogether. If a native method attempts to modify permissions (excluding `pkey_mprotect()`), the corresponding supervisor sends a SIGINT signal to terminate the thread immediately. This precaution prevents potential risks to other

tenants. The same response is triggered when a thread calls `pkey_alloc()` or `pkey_free()`. The supervisor gets notified for subsequent method calls that require monitoring or resource isolation, including `pkey_mprotect()`, `mmap()`, `munmap()`, `clone3()`, and `exit()`.

```
1  static void handle_notifications() {
2      struct Supervisor* sp = &supervisors[domain];
3
4      wait_sem();
5
6      for (;;) {
7          if (ioctl(sp->fd, ..., req) == -1) {
8              ...
9          } else if (sp->status) {
10             threadCount[domain]--;
11             break;
12         }
13
14         switch(req->data.nr) {
15                 ...
16         }
17
18         ioctl(sp->fd, SECCOMP_IOCTL_NOTIF_SEND, resp);
19     }
20 }
```

Get supervisor of domain

Wait supervisor assignment

Wait for native syscall

Empty domain if app status is done (1)

Handle filtered syscalls

Submit response

Listing 4.1: Notification handler.

At the beginning of the runtime, we assign a supervisor to each of the 14 available domains. These supervisors remain inactive until an application is allocated to a specific domain (see Listing 4.1, line 4). At that point, they begin supervising the executions within that domain, entering an infinite loop (line 6). When a thread attempts any notification-flagged syscalls, the supervisor receives a signal (line 7). If the notification is valid, the supervisor handles the call using a switch case (lines 14 to 16). After handling the call, the supervisor submits the response (line 18). The supervisor remains in this loop of handling notifications until the native method finishes. The native method has finished when its status is 1, indicating it is done (line 9). When that happens, the supervisor frees the domain and breaks out of the loop.

The following Listings provide a detailed explanation of why the supervisors get notified of these specific system calls and the procedures we use to handle them.

```
1  static void
2  handle_pkey_mprotect(struct seccomp_notif *req,
3                       struct seccomp_notif_resp *resp)
4  {
5      struct Supervisor* sp = &supervisors[domain];   <--------- Get supervisor of domain
6
7      if (sp->execution == NATIVE) {                  Return error if execution is native
8          resp->error = -errno;    <---------
9          return;
10     }
11
12     int prot = pkey_mprotect((void *)req->data.args[0],
13                              req->data.args[1],
14                              req->data.args[2],
15                              req->data.args[3]);
16     if (prot == -1) {                                                  return error if mprotect not successful
17         resp->error = -errno;   <----------------------------
18     } else {
19         resp->error = 0;   <------------------   error 0 means success
20         resp->val = (__s64)prot;
21     }
22 }
```

Listing 4.2: `pkey_mprotect()` handler.

`pkey_mprotect()` is very important permission's syscall in our supervision process. Instead of immediately stopping a calling thread that tries to execute it with SIGINT, we notify the supervisor, since Graalvisor reuses threads for optimization purposes. When threads are reused, they keep the seccomp filter, which becomes a challenge when they try to modify library permissions for the running application while in the call gate. Killing the thread would be problematic as the call is still coming from a trusted part of the execution. To distinguish between a `pkey_mprotect()` call originating from a call gate (managed) and one from an untrusted native method, we use a boolean field that is set by the thread when on the call gate. This field has two modes of execution: MANAGED and NATIVE. In NATIVE execution,

where the call comes from an untrusted source, we immediately stop the thread and raise an error, as described in lines 7-9. In contrast, in MANAGED execution, the supervisor continues with the call execution as usual.

```
1  static void
2  handle_mmap(struct seccomp_notif *req,
3              struct seccomp_notif_resp *resp)
4  {
5      void *mapped_mem = mmap(...);
6
7      if (mapped_mem == MAP_FAILED) {
8          resp->error = -errno;   <---------  Return error if mmap fails
9      } else {
10         if (pkey_mprotect(mapped_mem, ..., domain) == -1) {   <----  Protect mapped memory
11             resp->error = 1;
12             perror("pkey_mprotect");
13             return;
14         }
15
16         MemoryRegion memReg;
17         memReg.address = mapped_mem;
18         memReg.size = req->data.args[1];
19         memReg.flags = req->data.args[2];
20         insert_app(supervisors[domain].app, memReg);   <---------  Insert region into map
21
22         resp->error = 0;   <------------------  error 0 means success
23         resp->val = (__s64)mapped_mem;
24     }
25 }
```

Listing 4.3: `mmap()` handler.

The supervisor executes the `mmap()` call and uses `pkey_mprotect()` to adjust the permissions of the allocated block, enabling threads within the supervised domain to execute, see **else** statement in starting in line 9. By default, `mmap()`-allocated blocks only accept threads running in domain zero. This approach ensures that memory regions allocated by `mmap()` are properly segregated based on the domain of exe-

cution, preventing unauthorized access from threads in other domains. Additionally, to enhance security, the region allocated by `mmap()` is added to the application map, lines 16 to 20, restricting access to the data within it to the thread executing the application that initiated the `mmap()` call. Furthermore, its permissions are synchronized with the library's permissions to safeguard users who may forget to free the allocated memory.

```
1 static void
2 handle_munmap(struct seccomp_notif *req,
3                struct seccomp_notif_resp *resp)
4 {
5     struct Supervisor* sp = &supervisors[domain];   <---- Get supervisor of domain
6
7     remove_app(sp->app, req->data.args[0]);   <---- remove region from map
8     resp->flags = SECCOMP_USER_NOTIF_FLAG_CONTINUE;
9 }
```

Listing 4.4: `munmmap()` handler.

`munmmap()` `mmap()` by removing the `mmap()` address from the application map, indicating that the memory contained therein will be freed. Consequently, this process contributes to reducing the time required to modify the memory permissions of an entire application.

```
1 static void handle_clone3(struct seccomp_notif_resp *resp)
2 {                                          Increment
3     threadCount[domain]++;    <---------- thread number
4                                           on domain
5     resp->flags = SECCOMP_USER_NOTIF_FLAG_CONTINUE;   <---- Allow normal execution of syscall
6 }
```

Listing 4.5: `clone3()` handler.

`clone3()` is called upon `pthread_create()` so the supervisor increments the count of threads within the supervised domain. We employ a counter rather than a boolean to accurately track active threads, ensuring that the domain is only freed when the count reaches zero, indicating the absence of active threads. This mechanism allows us to manage system resources efficiently, releasing domains for reuse when we are certain they are no longer in use.

```
1 static void handle_exit(struct seccomp_notif_resp *resp)
2 {
3     threadCount[domain]--;   <---------  Decrement thread number on domain
4
5     resp->flags = SECCOMP_USER_NOTIF_FLAG_CONTINUE;   <----  Allow normal execution of syscall
6 }
```

Listing 4.6: `exit()` handler.

The `exit()` call complements `clone3()` by decrementing the count of threads within the supervised domain. Upon detecting a thread's exit, the supervisor adjusts the thread count accordingly, facilitating the proper cleanup and resource management within the domain.

4.9.2 Call gate

The call gate is the key component that enables Faastion to manage execution environments within a multi-domain system. Its functionality is essential for maintaining the illusion of continuous, uninterrupted domains and facilitating seamless transitions between managed and native execution. By orchestrating these transitions, the call gate ensures optimal resource allocation, robust security, and smooth operational flow across diverse execution contexts.

To fully understand the following Listings, it's important to note their simplification for clarity. Each thread contains a thread-local variable called "domain" used to store the domain used by the running application. It's worth noting that the "domain" variable may appear without being explicitly declared. Additionally, each thread contains a thread-local variable called "fd" which represents its file descriptor after the seccomp filter's installation.

```
1 void call_gate() {
2     acquire_domain(app, &fd);
3     change_stack_and_domain(domain);
4     native_method();   <----------------  Call original native method
5     change_stack_and_domain(0);
6     reset_env();
7 }
```

Listing 4.7: Call Gate.

This function is the crucial point where managed code transitions out of the sandbox and into the call gate, as explained in Section 4.3. Upon initiation, the call gate goes through carefully planned actions to guarantee strong security and operational integrity. Primarily, the call gate starts by obtaining a domain using a custom algorithm, explained in detail in Listing 4.8. Once the domain acquisition process is complete, Faastion transitions the thread's original stack into a domain-specific stack, mentioned in Section 4.5. After this change, the risk of segmentation faults is effectively eliminated, allowing the executing thread to seamlessly transition to its designated domain. With the necessary security measures in place, the original native method's execution can proceed. Following execution, Faastion restores the stack and the thread to the monitor domain and resets the environment (function explained in Listing 4.11).

```
1  void acquire_domain(const char* app, int *fd) {
2      if (is_app_cached(app) == 0 || threadCount[domain] > 0) {
3          find_domain(app, fd);
4      } else {
5          threadCount[domain]++;   <---------  Populate domain
6          assign_supervisor(app, fd);
7      }
8  }
9
10 void find_domain(const char* app, int *fd) {
11     while(1) {
12         for (int i = 2; i < NUM_DOMAINS; ++i) {
13             if (threadCount[i] == 0) {
14                 domain = i;
15                 threadCount[domain]++;   <---------  Populate domain
16                 assign_supervisor(app, fd);
17                 return;
18             }
19         }
20         usleep(100);   <----------------------  No empty domain found, active wait for 100 us
21     }
22 }
```

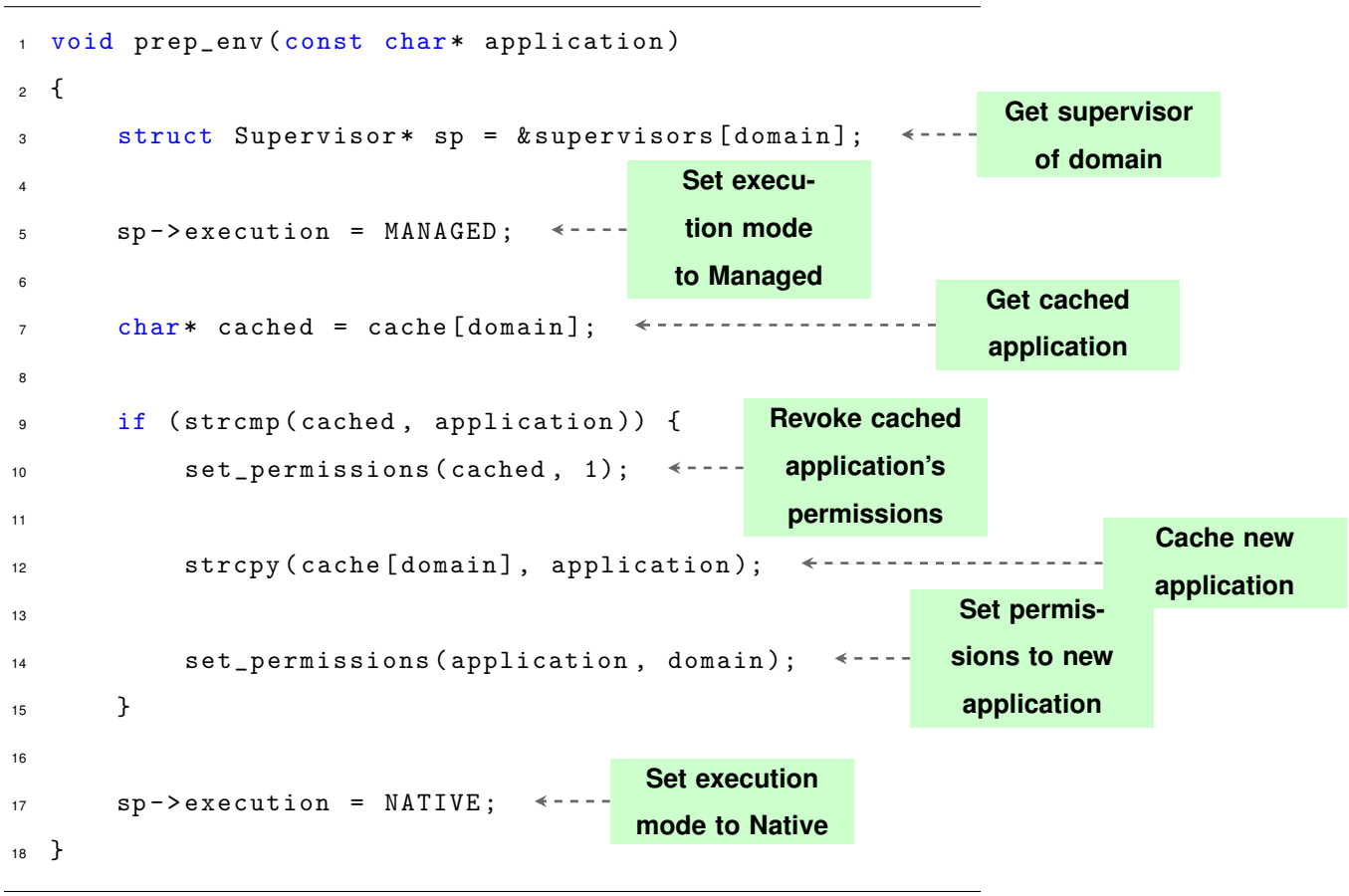
Listing 4.8: Domain acquirement.

When acquiring a domain, Faastion does a series of checks to know if the thread needs to waste CPU power searching for it. These checks involve checking the cache optimization described in Section 4.7. A cached application indicates that it was the most recently used application by the thread, which implies that the native library permissions for the corresponding domain remain active. If the application is in cache but the number of threads executing in the previous domain is not zero, the program must seek an alternative domain using the `find_domain()` function. Contrarily, if the number of threads executing for the previous domain is zero, the executing thread can reclaim the previous domain and assign the corresponding supervisor. The `find_domain()` function operates through a straightforward algorithm that scans for the first available domain. If there are no available domains, the thread waits for 100 microseconds before restarting the search. This loop continues until there is an available domain. This method ensures the re-usage of domains based on real-time native usage, preventing bottlenecks and promoting efficient execution of concurrent threads.

```
1  void assign_supervisor(const char* app, int* fd)
2  {
3      struct Supervisor* sp = &supervisors[domain];   <---- Get supervisor of domain
4      if (*fd == 0) {
5          *fd = install_notify_filter();   <--------- Install seccomp filter
6      }                                    Assign file
7      sp->fd = *fd;   <----                descriptor to
8                                           supervisor
9      signal_sem();   <------------------   Signal supervisor to
10     prep_env(app);                        start handling
11 }
```

Listing 4.9: Supervisor assignment.

The `assign_supervisor()` function plays a pivotal role in setting up the monitoring infrastructure for thread execution. This function is responsible for installing the seccomp filter (Section 4.8), which restricts system calls to the predefined set mentioned in 4.9.1. If it's a first-time executing thread, Faastion installs the filter in the executing thread and obtains a file descriptor. The file descriptor acts as a communication channel between the executing thread and the supervisor. It allows the supervisor to monitor the thread's activities and intervene to handle system calls that require supervision. Once the file descriptor is obtained, the supervisor can begin handling notifications. We then signal the supervisor (receiving the signal is in line 4 of Listing 4.1).



Listing 4.10: Environment preparation.

Before executing untrusted code, the environment needs preparation via the `prep_env()` function. Initially, Faastion sets the execution mode to MANAGED. We expedite permission modifications in lines 9-15 using the optimization strategy mentioned in Section 4.7. All subsequent `pkey_mprotect()` calls originating from this thread are attributed to untrusted code. To accommodate this, the supervisor's execution field is adjusted to NATIVE.

```
1 void reset_env()
2 {
3     struct Supervisor* sp = &supervisors[domain];
4     sp->status = DONE;
5 }
```

Listing 4.11: Environment reset.

To signal the supervisor to stop handling notifications for the finished native method, we set the

supervisor status to DONE. This action allows the execution flow to enter the else-if statement in line 9 of Listing 4.1.

5

Evaluation

Contents

In this section, we will systematically analyze the performance and effectiveness of Faastion through various benchmarks and workloads. Our evaluation aims to demonstrate Faastion's ability to securely execute concurrent applications by leveraging SFI with Memory Isolates and utilizing hardware-based isolation mechanisms for native code execution. We will use GraalVM as the managed runtime environment, with Java benchmarks transitioning control flow to native code via the JNI. The evaluation will cover the following aspects:

- Configuration details of the experimental setup.
- Analysis of page migration overheads.
- Detailed study of native execution.
- Comparative assessments against different isolation mechanisms, including Memory Isolates, process-based isolation, and Faastlane.
- Comprehensive benchmarking results, focusing on native execution, managed execution, and mixed execution scenarios.

Each subsection will provide insights into specific performance metrics and security aspects, highlighting the strengths and potential limitations of Faastion in various operational contexts.

5.1 Evaluation Environment

We conducted all our experiments on a system equipped with two Intel Xeon Silver 4114 10-core CPUs running at 2.20 GHz and 200GB of ECC DDR4-2666 Memory. To ensure reproducibility, it is essential to run the benchmarks on systems with MPK support. As for software requirements, we ran all our benchmarks using Java JDK 17.0.7, GraalVM for JDK 17.0.7, and MUSL version 1.2.4. Additionally, benchmarks utilizing the OpenSSL library were executed using OpenSSL version 3.3.0. We used Ubuntu 22.04.4 LTS with kernel 5.15.0-86-generic, as our system utilizes `seccomp_unotify`, a Seccomp user-space notification mechanism introduced with Linux 5.x. To conduct the experiments, we disabled additional Intel features such as Turbo Boost and Hyperthreading. Disabling Turbo Boost prevents the CPU from fluctuating clock speed while disabling Hyperthreading prevents physical cores from acting as two logical cores.

In our evaluation, we utilized a variety of benchmarks to assess the performance and behavior of Faastion under different workloads. Each benchmark has two versions. One that executes in native code and another in managed code. The benchmarks used in our study are detailed below:

- **Image manipulation**: Processes images and performs tasks such as resizing, filtering, and color adjustments.

- **Matrix multiplication**: Performs matrix multiplication operations.
- **File hashing**: Computes cryptographic hashes for files.
- **AES encryption**: Performs AES encryption and decryption operations.
- **Factorization**: Factorizes large integers.
- **Zip compression**: Compresses and decompresses data using ZIP algorithms.

5.2 Page Migration Overheads

To interpret this work's results accurately, it's important to understand how long each component is expected to take. In this study, we looked at how quickly the `pkey_mprotect()` function operates when altering the permissions of different numbers of memory pages. We also assessed how changing the number of threads accessing the same amount of memory affects performance.

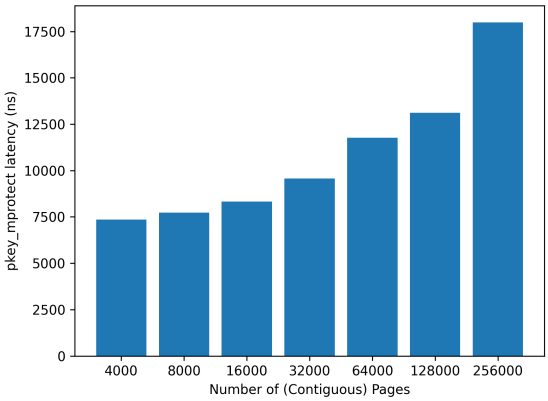


Figure 5.1: `pkey_mprotect()` latency with varying number of pages.

Figure 5.1 illustrates the latency of the `pkey_mprotect()` function with a varying number of contiguous memory pages. To obtain these results, we developed a simple script that executes `pkey_mprotect()` at least 100 times for each page count, subsequently averaging the results to ensure accuracy and consistency. The results demonstrate that the latency of `pkey_mprotect()`, increases with the number of pages. This trend is expected due to the increased overhead associated with managing larger memory regions. The plot exhibits an exponential growth pattern because the number of pages doubles with each iteration. Consequently, the latency growth appears steep, reflecting the computational complexity involved in handling larger contiguous memory regions.

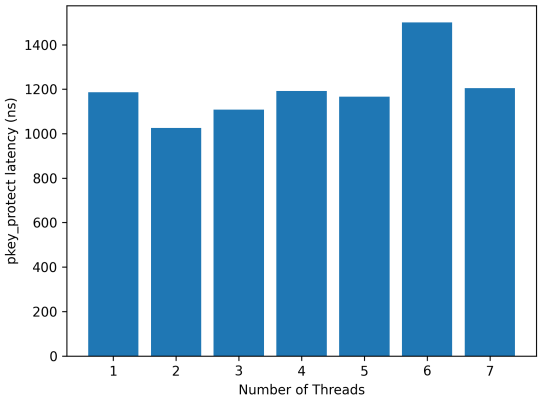


Figure 5.2: `pkey_mprotect()` latency with varying number of threads.

Figure 5.2 presents the latency of the `pkey_mprotect()` function with a fixed number of pages as the number of threads varies. Similar to the previous experiment, we executed `pkey_mprotect()` at least 100 times for each thread count and reported the average results to ensure reliability.

Interestingly, the results indicate that the latency of `pkey_mprotect()` remains almost unchanged regardless of the number of threads. This observation suggests that the function's performance is primarily influenced by the number of pages rather than the number of concurrent threads. The consistency in latency across different thread counts can be attributed to the fact that `pkey_mprotect()` operates on a per-page basis, and the overhead associated with managing memory permissions does not significantly scale with the number of threads.

This study provides detailed insights into the performance implications of the `pkey_mprotect()` function under different conditions. Understanding this behavior is essential for optimizing resource allocation and ensuring efficient execution within our multi-domain system. As we were able to see, `pkey_mprotect()` adds little latency to out final solution if compared with solutions that use forking, facilitating our decision of choosing MPK.

5.3 Native execution study

To substantiate our insight that not all applications execute native code, and those that do spend minimal time in it, we conducted the evaluation presented in Figure 5.3.

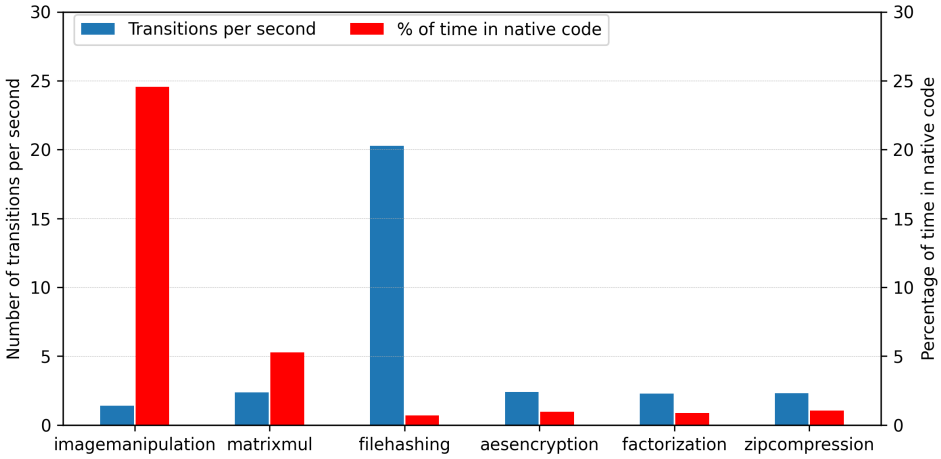


Figure 5.3: Native code execution.

This Figure corroborates our statement, revealing that the majority of native code executed originates from the runtime itself, which is inherently trusted. To gather this data, we used a custom Javassist tool to count the number of native methods in each application and wrap each method in a timer. This allowed us to track the percentage of execution time spent in these methods. The depicted values specifically highlight untrusted native methods, which are the primary focus of our isolation efforts. We excluded Java-specific native methods, such as those from the System package, from our analysis, as we consider them trusted.

5.4 Comparison Targets

In this section, we outline the various approaches against which we are going to compare Faastion to evaluate its performance, security, and efficiency.

5.4.1 Memory Isolates

Memory Isolates offer a method to run multiple independent VM instances within a single process, facilitating intra-process isolation. This approach ensures that objects in one isolate are not accessible by objects in another, providing strong security guarantees. However, these guarantees weaken when there is a control flow transition to execute native code. Native code execution can bypass the isolation, potentially exposing vulnerabilities that Memory Isolates alone cannot mitigate. To note that Faastion is built on top of *isolates* which should make it slower in every comparison.

5.4.2 Process-Based Isolation

Process-based isolation relies on the inherent memory isolation facilitated by individual processes, each possessing its own distinct address space. This strategy guarantees that one process cannot directly access the memory of another, thereby fortifying data privacy and security. In this approach, an initial process, serving as a fork server, generates a new child process for every invocation. While this ensures robust isolation, it introduces notable overhead primarily attributable to the forking process. This overhead can manifest as increased latency and memory consumption, particularly in scenarios of heightened workload, potentially impacting performance.

5.4.3 Faastlane

To facilitate a fair comparison, we implemented our version of Faastlane. It is an approach that uses MPKs to reserve a protection domain for the entire duration of a function's execution. Unlike Faastion, which assigns domains on-demand based on execution needs, Faastlane allocates domains statically for the duration of the function. This static allocation can lead to rapid exhaustion of available domains, reducing overall availability and efficiency. It is important to note that this is not the actual implementation of Faastlane, but rather a simplified version. This simplifications involves reutilizing Faastion's logic with the optimization mentioned in Section 4.7. The key distinction lies in Faastlane's acquisition of a domain prior to the entire application code execution, releasing it only upon completion of execution.

5.5 Benchmarking

To closely simulate the workloads that a serverless system might encounter, we evaluate benchmarks that exclusively run managed code and benchmarks that require the execution of native code, where the control flow transitions out of the managed runtime sandbox to execute native code. Blending benchmarks with executions in both managed and native code ensures that our evaluation closely captures the execution requests that production-ready serverless systems encounter, as it is unlikely that every single request requires the need for native libraries.

We focus on four key metrics to comprehensively assess Faastion's performance, each providing critical insights into aspects of the system's behavior under various conditions. These metrics are:

- **Client-Side Latency**: This metric measures the Round-trip time (RTT) from the client's perspective, encompassing the total time taken for a request to travel from the client to the server and back. We will evaluate how Faastion handles an increasing number of concurrent requests, observing how client-side latency scales with load and assessing the system's responsiveness under high-traffic conditions.

- **Server-Side Latency**: This represents the time taken to process each request on the server, from the moment it receives a request until it sends a response. We will analyze the impact of escalating concurrent requests on server-side latency to understand how Faastion manages processing efficiency and workload distribution as the number of requests increases.

- **Domain Usage Over Time**: This metric examines how effectively Faastion utilizes available execution domains compared with *Faastlane*. By tracking domain usage over time, we can determine the efficiency of Faastion's dynamic domain allocation strategy, ensuring optimal resource usage and minimizing the risk of domain exhaustion, especially under varied and demanding workloads.

- **Memory Footprint**: This assesses the memory resources during execution. Monitoring the memory footprint helps us evaluate the system's resource efficiency and ability to maintain performance without excessive memory consumption. This metric is essential for understanding the scalability and cost-effectiveness of Faastion in different deployment scenarios.

Each of these metrics will be analyzed under controlled benchmarking scenarios using the **wrk** benchmarking tool. We will simulate a range of workloads by varying the number of threads and connections to mimic real-world usage patterns. By running these tests for sustained periods (e.g., one minute per configuration) and averaging the results, we ensure the reliability and accuracy of our measurements. Through this detailed benchmarking process, we aim to highlight Faastion's strengths and identify any areas for improvement, providing a comprehensive understanding of its performance, efficiency, and scalability compared to other isolation and execution management approaches.

5.5.1 Native Execution

This type of execution focuses solely on benchmarks that escape the sandbox into untrusted native methods, representing the worst-case scenario for Faastion. This type of execution is critical as it tests the system's ability to handle untrusted code securely and efficiently. In such cases, we expect Faastion's performance to be comparable to that of *Faastlane*, as both systems should manage and acquire the same amount of domains at the same speed.

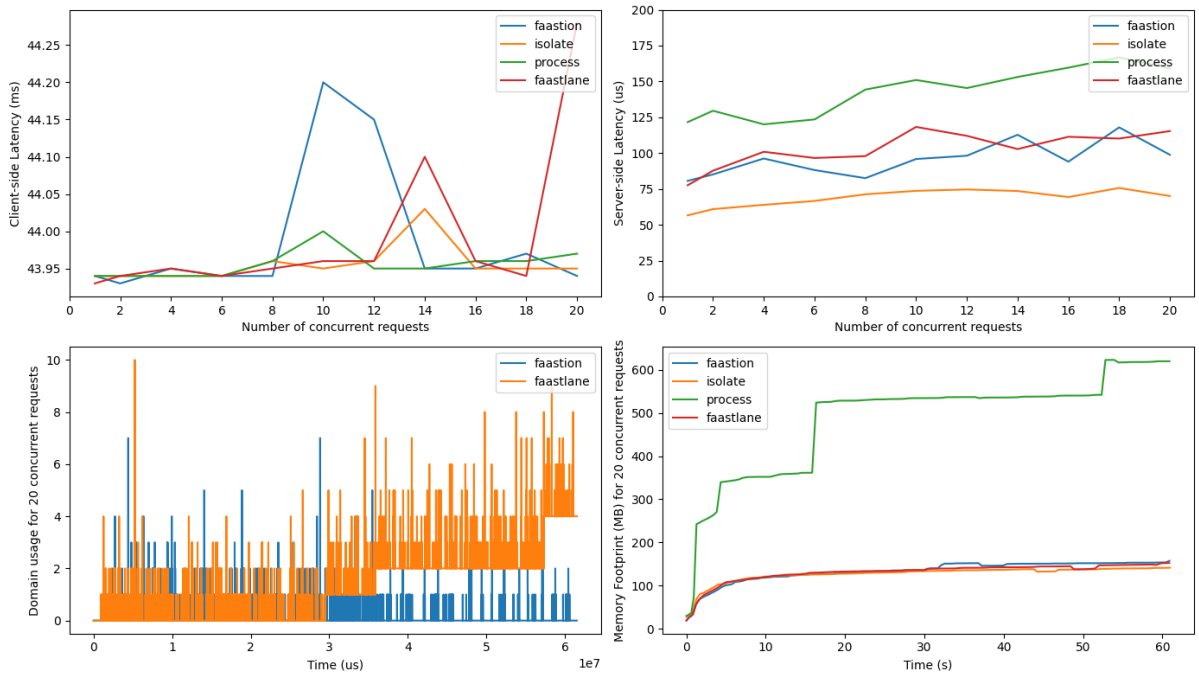


Figure 5.4: Native execution metrics.

Figure 5.4 presents results from a high-speed native Factorization benchmark. Upon analyzing client-side latency (top-left), we see that all approaches exhibit similar average latency times across various levels of concurrent requests. We attribute this consistency to network noise, as evidenced by the server-side graph (top-right), where requests take approximately 90 microseconds to execute. However, on the client side, latency hovers around 50 milliseconds, indicating a significant difference attributable mainly to inherent RTT overhead. While Faastion and *Faastlane* may appear to show slight increases in certain requests, this observation can be attributed to the scale of the graph. In absolute terms, the difference in latency values among the four frameworks is negligible, with the maximum value being only around 0.2 ms greater than the lowest.

As for the server-side latency (top-right), which results don't include network overheads and only include the time it takes to handle the request, we observe different outcomes. Each framework shows a slow and steady increase in execution time as the number of concurrent requests grows, due to the added overhead of managing more concurrent executions. *Isolates* consistently has the lowest average execution time, as expected, followed by Faastion and Faastlane, and Process at last, which also exhibits the steepest increase in execution time, probably because of the inherent overhead of forking. Faastion and *Faastlane* consistently have similar performance, with Faastion having the lower edge with an average decrease of 6.73%. Comparing Faastion's average execution time for 1 and 20 concurrent requests, we see an increase of 20.82%, 7.25% more than the same comparison with *isolates*. This additional percentage seems normal considering Faastion runs on top of *isolates*. Furthermore, when

executing with more than 14 concurrent requests, Faastion has to resort to active waiting.

Given that the concurrent applications tested for the basic strategy always include native code execution, both frameworks showed similar domain usage (bottom-left), with Faastion holding a slight advantage. This distinction can be attributed to Faastion exclusively reserving domains for native execution, whereas *Faastlane* allocates domains for the entire application lifecycle. Notably, despite subjecting the systems to 20 concurrent requests, the domain count did not consistently reach our threshold of 14. This discrepancy may stem from our method of data collection, which involves checking domain usage every millisecond, while requests typically complete within 100 microseconds on average. Consequently, there's a likelihood that we miss the precise moment when domains are exhausted. Nonetheless, debugging confirms the exhaustion of domains. Another contributing factor is the fast execution time of each function, making it challenging to exhaust the number of domains. The accompanying graph provides no more than an overview of how each approach manages domain allocation.

The Memory Footprint (bottom-right) for *isolates*, *Faastlane*, and Faastion are all very similar. In practice, this is because all three frameworks run on a single process, leveraging execution threads to manage different concurrent requests. In comparison, Process-based isolation requires significantly more memory than the other frameworks. This is expected and is due to the creation of multiple processes to handle concurrent requests. Creating processes demands substantially more memory compared to the single-process approach of the other three frameworks, which manage concurrent requests at the thread level. Thread-level management is much more efficient in terms of memory utilization. This metric should be consistent regardless of the type of execution, therefore we are only going to mention it here.

5.5.2 Managed Execution

Centered exclusively on benchmarks that remain within managed code for their entire duration, this execution mode represents the optimal conditions for Faastion, where we expect performance to be virtually identical to that of *isolates*. Furthermore, we anticipate a significant performance advantage over *Faastlane*, as Faastion's domain management and isolation strategies are optimized for managed runtimes, thereby avoiding the overheads associated with handling untrusted native methods.

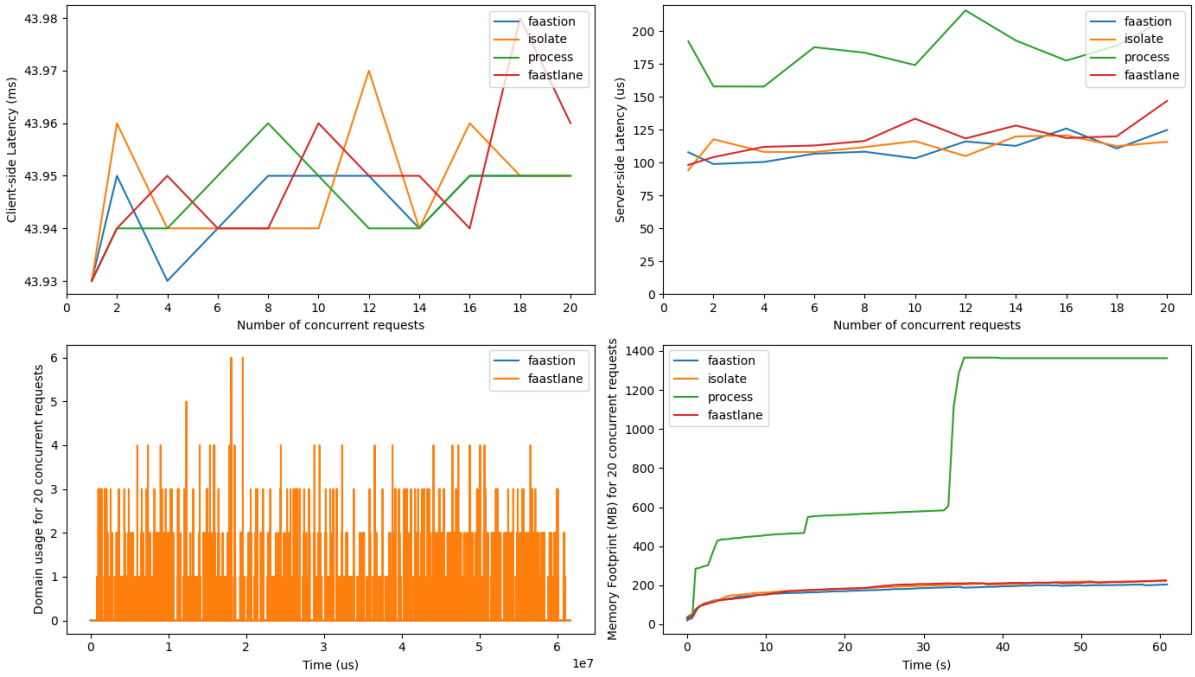


Figure 5.5: Managed execution metrics.

Figure 5.5 illustrates the evaluation results from a high-speed managed Matrix Multiplication benchmark. When examining client-side latency (top-left), we note similarities to the previous execution type, with some variations attributed to network noise.

For server-side latency (top-right), we observe distinct patterns. Each framework experiences a gradual increase in execution time as the number of concurrent requests rises. *Isolates* and Faastion consistently exhibit the lowest average execution time, as anticipated due to its efficient management and utilization of MPK domains. Faastion leverages Java's sandbox with Memory Isolates to ensure complete isolation when executing managed applications, thereby maintaining secure execution. Process-based isolation exhibits the higher server-side latency, again likely due to the overhead of forking new processes for each request. *Faastlane* exhibit similar performance trends as before. For instance, comparing the average execution time with Faastion's shows an increase of 11.52%. *Faastlane* does not differentiate based on the type of execution, making these results anticipated. It always reserves a domain preemptively, regardless of whether it is needed for managed or native code.

In terms of domain usage (bottom-left), Faastion does not utilize domains in this type of execution since all code is managed. This efficiency stems from its ability to rely solely on Java's inherent sandboxing capabilities as stated earlier. Conversely, *Faastlane* exhibits consistent domain usage across different execution types, as it reserves a domain preemptively for every execution.

5.5.3 Mixed Execution

Including managed and native benchmarks, this form of execution represents the most realistic scenario. In this mixed environment, we expect Faastion's performance to surpass that of *Faastlane* due to more efficient domain management. While Faastion's performance may be slightly lower than *isolates*, the difference should be minimal, demonstrating Faastion's capability to handle diverse workloads effectively. We will test this execution type using two distinct workloads: one relying on high-speed benchmarks and the other on moderate to low-speed benchmarks. Since *Faastlane* is our primary competitor, this comparison aims to highlight Faastion's performance advantages.

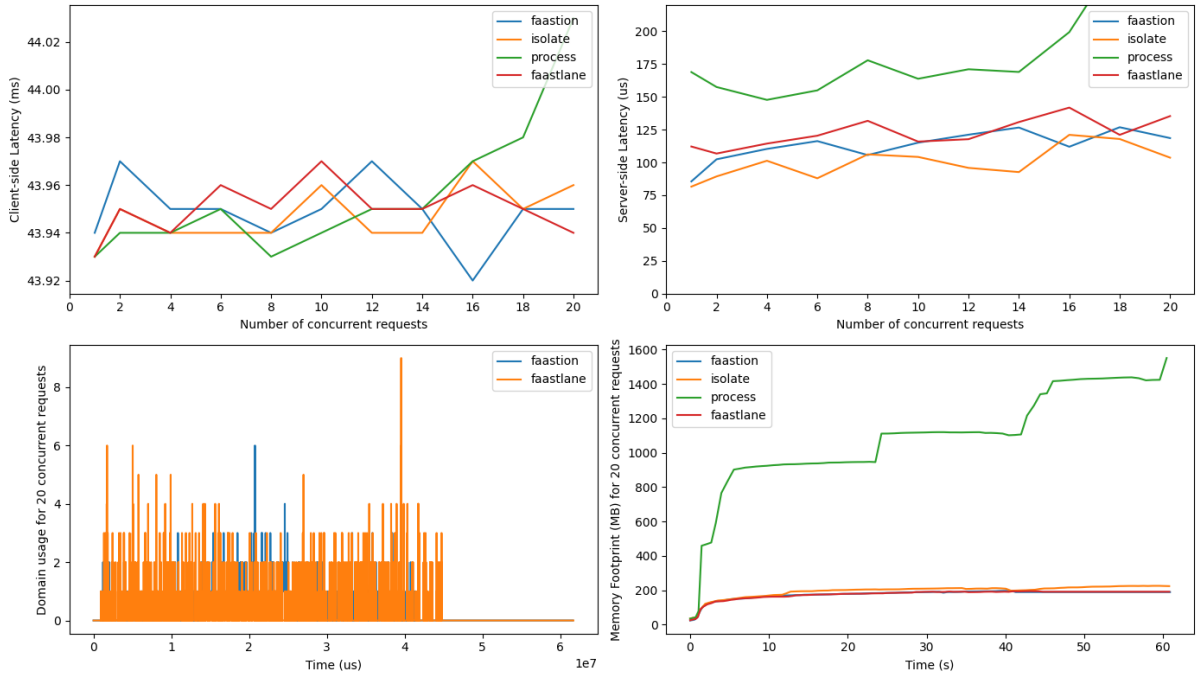


Figure 5.6: Mixed execution metrics (high-speed).

High-speed workload. Figure 5.6 uses the same Factorization benchmark as in native execution alongside a managed Matrix multiplication benchmark. When examining client-side latency (top-left), we observe similar results as the other executions. Due to the high speed of these benchmarks, network noise is the primary source of latency.

Regarding server-side latency (top-right), the outcomes differ. Regardless of the number of requests, isolates keep consistently achieving the lowest average execution time, followed by Faastion, *Faastlane*, and process-based isolation. In this scenario, the graph indicates that Faastion's average latency is 25.48% higher than *isolates*'. Additionally, Faastion shows a 7.51% lower latency than *Faastlane* and a 41.67% decrease compared to process-based isolation.

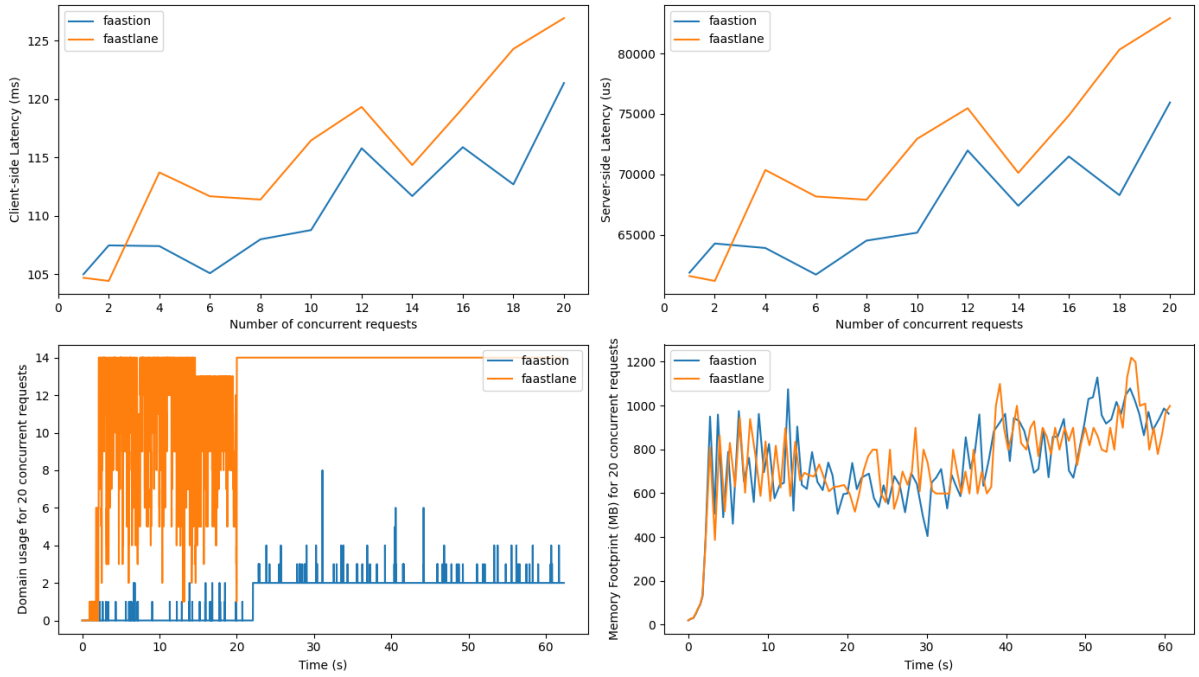


Figure 5.7: Mixed execution metrics.

Moderate speed workload. Figure 5.7 uses the same Factorization benchmark and a managed File hashing benchmark. Upon observing client-side latency (top-left) and server-side latency (top-right), we note a different trend from the other execution types, with the graphs appearing very similar. This confirms our earlier assertion that the differences were primarily due to networking noise. The processing of a large file for hashing now influences the execution, dictating client-side latency values. Analyzing both plots, we observe a clear advantage of Faastion over its direct competitor, *Faastlane*. As the number of concurrent requests increases, both approaches exhibit a steady increase in latency. However, Faastion demonstrates an average execution time of 12.31% less than that of *Faastlane*.

In this realistic scenario, we observe a distinct difference in domain usage (bottom-left). Faastion utilizes domains much less frequently than Faastlane due to the managed benchmarks, as explained in the previous Sections. This discrepancy becomes even more pronounced due to the extended duration required to complete the file hashing benchmark, resulting in prolonged domain retention. Faastion capitalizes by acquiring a domain only when executing untrusted native code. Conversely, Faastlane's longer domain retention leads to active waiting, contributing to inferior performance compared to Faastion.

5.5.4 Lazy-Faastion Optimization

In this evaluation, we'll discuss the execution times of Faastion with and without the cache optimization detailed in Section 4.7. The contrast between eager- and lazy-Faastion lies in their handling of native

library permissions. Eager-Faastion adjusts these permissions each time it begins and concludes executing native code, whereas lazy-Faastion bypasses these adjustments if the application is cached, theoretically resulting in significantly faster performance.

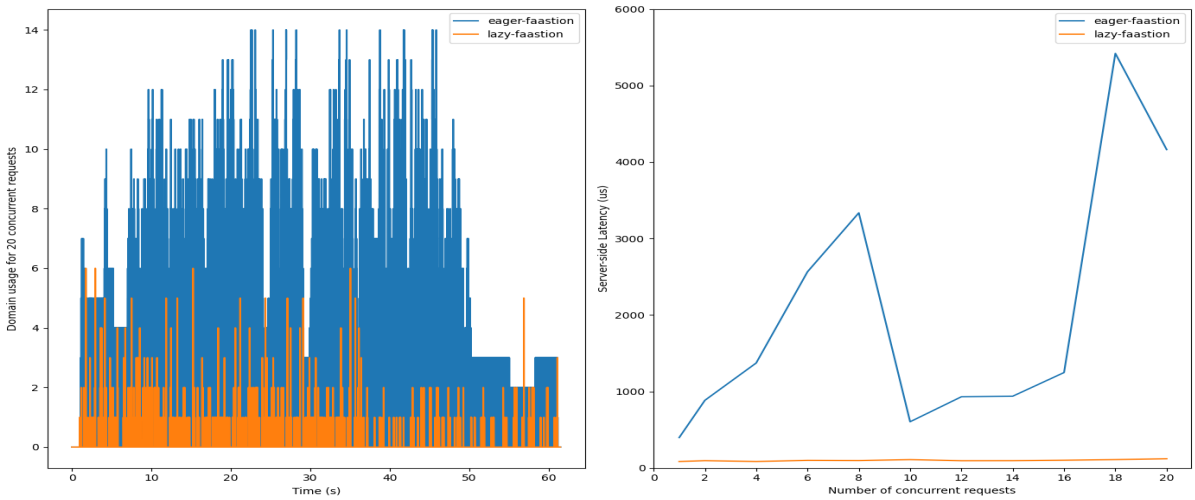


Figure 5.8: Eager- vs Lazy-Faastion metrics.

Figure 5.8 depicts the outcomes of conducting a native Factorization benchmark. Notably, there is a substantial contrast in domain usage, as evidenced by the graph on the left. This variance aligns with the discrepancy in workload complexity between eager- and lazy-Faastion. Since eager-Faastion undertakes more work, it logically requires more time for execution. Consequently, there is a higher probability of additional domain acquisitions during its execution cycle. This behavior was anticipated, and the observed 72.22% reduction in domain usage disparity underscores its significant influence on overall execution times.

The server-side latency data (right) provides a clearer illustration of the substantial time disparities between eager- and lazy-Faastion. This distinction is primarily attributed to the augmented workload inherent in eager-Faastion, resulting in accelerated domain exhaustion and frequent resorting to active waiting. On average, this manifests as a remarkable 93.31% reduction in latency, with an even more impressive 97.82% decrease in the worst case (18 concurrent requests). These figures underscore the profound impact of cache optimization on minimizing latency and optimizing overall system performance.

The significant reduction in domain usage and the corresponding reduction in instances requiring active waiting underscore the effectiveness of the cache optimization. These improvements directly translate to enhanced performance across the board.

5.6 Conclusion and Findings

In our evaluation, Faastion emerges as the leading performer compared to process-based approaches, showcasing superior performance and exceeding *Faastlane* by a small margin. We expect this margin to become more pronounced in scenarios with consistently more than 20 concurrent requests, particularly in mixed environments. However, our testing was constrained by the hardware configuration of the machine, featuring 2 CPUs with 10 cores each, which prevented us from exceeding 20 threads with 20 connections without encountering undefined behavior. Although one thing we can conclude, is that Faastion is more likely to have a better performance than *Faastlane* if the applications executed are not high-speed. Moreover, a significant observation lies in the vast disparity in memory footprint between process-based isolation techniques and the other approaches. Despite a minor performance dip observed in *isolates*, the enhanced protection and isolation from other tenants delivered by Faastion provide a substantial overall improvement and benefit. These findings underscore Faastion's effectiveness. It is an efficient serverless computing solution, particularly in environments where security and performance are paramount concerns.

6

Conclusion

Contents

Serverless computing has been limited by issues related to privacy and multi-tenancy, particularly in terms of securely running multiple functions from different tenants in the same environment. To address these challenges, we propose a solution that utilizes a combination of technologies, including GraalVM's Native Image isolates and MPK. These tools help to isolate resources and control the execution of native code, allowing us to maintain security and trust in the runtime while also improving performance. Through our experimentation with this approach, we have achieved an equilibrium between security and performance in serverless computing. Our findings underscore the efficacy of this strategy in mitigating the inherent risks associated with multi-tenant environments while optimizing system performance to meet the demands of modern cloud computing architectures.

6.1 System Limitations and Future Work

The current implementation is limited to Linux systems and requires specific Intel Central Processing Units (CPUs) that support MPK technology. Therefore, platforms lacking these prerequisites are incompatible with the system, even though other similar features are planned for AMD.

To address some of the limitations of MPK, such as performance and scalability, a new approach to memory protection in computer systems called Efficient Protection Keys (EPK) [37] was proposed. EPK uses a new set of hardware-based keys to provide more granular memory protection, resulting in improved memory resource utilization and enhanced security. This approach uses existing virtualization hardware features to expand the number of available protection domains, making it a valuable solution to enhance MPK and overcome its limitations.

Expanding the functionality beyond its current scope holds great promise for future development. Currently, the system is focused on Java static analysis and snippet generation using Javassist. However, broadening its capabilities to include additional programming languages such as JavaScript and Python would significantly improve its accessibility and usefulness. This expansion would make the tool available to a wider audience of developers across different programming ecosystems. Moreover, integrating support for multiple languages could lead to new applications and use cases, enhancing the tool's versatility and potential impact.

A promising route for future work in Faastion involves incorporating mechanisms similar to ERIM's binary analysis for enhanced security and isolation. The analysis would scan application binaries for specific assembly code patterns indicative of security vulnerabilities, such as attempts to modify thread domains. By integrating such analysis mechanisms into Faastion, we would be reinforcing isolation boundaries between concurrent applications, making it an interesting feature.

Lazy Process Isolation is a feature we've been developing in collaboration with another student. Although we have a prototype, it is not yet ready for testing with real workloads. This feature aims to

replace the current active waiting mechanism. Here, we provide a brief overview of its purpose, the progress we have made so far, and our vision for the final solution.

Faastion ensures memory isolation during native code execution by relying on the availability of empty MPK domains for optimal performance. However, due to the limited number of MPK domains, a contingency plan becomes necessary when demand exceeds availability. In such cases, a fallback mechanism comes into play. Native code awaiting execution could be queued until an MPK domain becomes available. Yet, this approach introduces potential delays, as the release of MPK domains depends on the completion of functions utilizing them, leading to unpredictably prolonged wait times. In Faastion's current architecture, if all domains are occupied, the system checks every 100ms to see if a domain has become available. This strategy inherently has its flaws. This active waiting mechanism leads to inefficient CPU usage. Continuously checking for domain availability consumes CPU cycles that could be used for processing actual tasks. This degrades overall system performance, especially under high-load conditions where many domains are frequently occupied.

To find the best solution, we researched different approaches. We started by looking into the `fork()` mechanism, a traditional choice for process creation. This method creates a complete duplicate of the parent process, resulting in a 1:1 replication that, while simple in concept, proved inefficient for our needs. The duplication introduced security concerns, as the forked process inherited full access permissions to the parent's memory space. This raised the alarming possibility of unintended data exposure, potentially compromising sensitive information set by other applications.

To mitigate this risk, we evaluated alternative strategies. One option was to use MPK in the new process, ensuring isolation. However, this approach would involve significant overhead, considering that the overhead of fork already exists, making this solution impractical. One alternative solution involves using *zygote* processes, which are "blank slate" processes without any pre-existing memory or state. By starting from scratch, *zygote* processes provide a clean environment for carrying out new tasks, free from any inherited baggage from parent processes. However, implementing and managing *zygote* processes can add complexities and overhead, which may make them less practical in our specific situation. However, this method was not considered because each time a new process is needed, forking would occur in the critical path, leading to performance issues. By using a process pool processes are always ready, avoiding these issues.

We conducted tests to evaluate the effectiveness of combining `vfork()` with `execve()`. `vfork()` is a lighter version of `fork()` that establishes a direct connection between the child and parent processes instead of creating a duplicate. While this approach showed improvement over the traditional `fork`, it did not fully meet our optimization goals. Additionally, we examined the `posix_spawn()` function, which utilizes `vfork()` internally. We chose `posix_spawn()` over `vfork()` because, as suggested by Unix, using `exec` immediately after `vfork()` is recommended, but not feasible for us as we need to create

pipes for communication. `posix_spawn()` allows us to use `vfork()` while still enabling the creation of pipes.

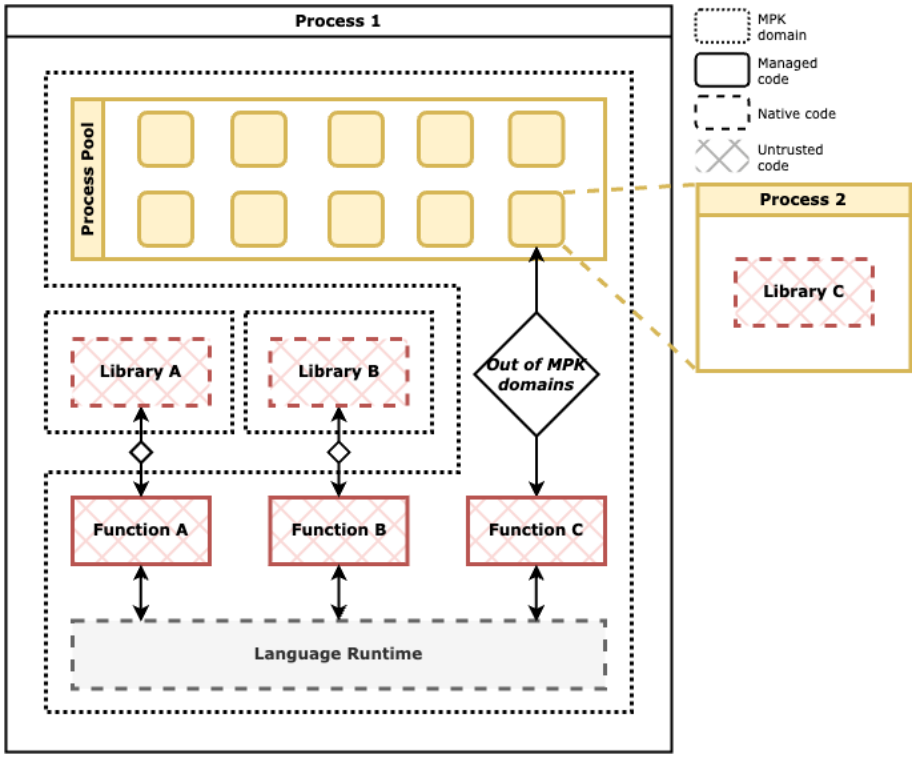


Figure 6.1: Lazy Process Isolation.

We have decided to create a process pool consisting of 20 processes that are ready to be used whenever we run out of domains, Figure 6.1. Rather than wasting time forking, we should use pipes to send a payload indicating which method we want to run inside the process. The overhead of communicating through pipes is much lower than the overhead of forking. We have incorporated this approach into our final solution as a fallback mechanism.

To have an ultimate fallback mechanism for worst-case scenarios, we have implemented the `posix_spawn()` as a fallback method to the fallback method. This way, even if we run out of processes in the process pool, we can create a new one and keep the native code completely isolated. It is important to note that these scenarios are extremely unlikely, as even in complex workloads, many applications do not rely on native code, and if they do, they spend little on it, making our on-demand domain switching available most of the time. Thus, to accommodate the limitations of the MPK domains, Faastion runs in three different execution modes.

- **Hot Execution**. When the number of native code executions is lower than the available number of domains, Faastion uses hot execution. In this mode, the native code transitions are intercepted and isolated within an empty MPK Domain. Hot execution is the default execution mode of Faastion

(see Figure 4.1).

- **Warm Execution**. When the number of MPK domains is exhausted, and a new request is required to execute native code, Faastion overcomes the limitation by using inter-process isolation.
  - Initialization: Our system maintains a process pool of 20 pre-initialized processes kept on standby to execute native code for incoming requests. Initializing the processes ahead of time helps exclude the overhead of calling the *fork()* system call from the critical path. The pre-initialized processes in the pool await a payload from the function call gate, which notifies the corresponding process to execute a native function.
  - Task Assignment: The function call gate sends a payload to a process in the pool. This payload contains the shared library file and the function name called by the application. The dormant process uses the payload to execute the function at runtime. In warm execution, we establish inter-process communication through named pipes.
  - Cleanup: To guarantee complete memory isolation, Faastion cannot reuse processes for future tasks. As a result, Faastion utilizes a background thread to clear zombie processes and promptly replenish the process pool by replacing used processes with new ones.
- **Cold Execution**. In the worst-case scenario, a function invocation might not be able to use either Hot or Warm execution mode for native execution, as both the MPK domains and process pool might be depleted. In such extreme scenarios, our system employs *posix_spawn()*, which utilizes the *vfork()* system call to create a new child process and execute the native function. Using *vfork()* to create a new child process overcomes the cost of copying page tables from the parent to the child process.

Bibliography

[1] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, vol. 62, no. 12, p. 44–54, nov 2019. [Online]. Available: https://doi.org/10.1145/3368454

[2] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," *Commun. ACM*, vol. 64, no. 5, p. 76–84, apr 2021. [Online]. Available: https://doi.org/10.1145/3406011

[3] "Cloud computing market size, share trends analysis report by deployment model, by service, by vertical and segment forecasts, 2020 - 2027," 2021. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/cloud-computing-industry

[4] Cloudflare, "Introducing cloudflare workers: Run javascript service workers at the edge," https://blog.cloudflare.com/introducing-cloudflare-workers/, 2017.

[5] A. W. Services, "Aws lambda," https://aws.amazon.com/lambda/, 2021.

[6] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. USA: USENIX Association, 2017, p. 363–376.

[7] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 13–24. [Online]. Available: https://doi.org/10.1145/3357223.3362711

[8] I. Müller, R. Marroquín, and G. Alonso, "Lambada: Interactive data analytics on cold data using serverless cloud infrastructure," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 115–130. [Online]. Available: https://doi.org/10.1145/3318464.3389758

[9] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden, “Starling: A scalable query engine on cloud functions,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 131–141. [Online]. Available: https://doi.org/10.1145/3318464.3380609

[10] M. Wawrzoniak, I. Müller, R. Bruno, and G. Alonso, “Boxer: Data analytics on network-enabled serverless platforms,” in *International Conference on Innovative Data Systems Research*, 2021.

[11] Z. Jia and E. Witchel, “Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 152–166.

[12] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, “Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider,” in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 205–218. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad

[13] W. C. Group, “Webassembly,” https://webassembly.org/, 2021.

[14] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, “libmpk: Software abstraction for intel memory protection keys (intel {MPK}),” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 241–254.

[15] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger, “Initialize once, start fast: Application initialization at build time,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360610

[16] B. Hayes, “Cloud computing,” 2008.

[17] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, “A view of cloud computing,” *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[18] S. Bhardwaj, L. Jain, and S. Jain, “Cloud computing: A study of infrastructure as a service (iaas),” *International Journal of engineering and information Technology*, vol. 2, no. 1, pp. 60–63, 2010.

[19] E. Keller and J. Rexford, “The” platform as a service” model for networking.” *INM/WREN*, vol. 10, pp. 95–108, 2010.

[20] M. K. Hussein, M. H. Mousa, and M. A. Alqarni, “A placement architecture for a container as a service (caas) in a cloud environment,” *Journal of Cloud Computing*, vol. 8, pp. 1–15, 2019.

[21] “Serverless Computing - AWS Lambda - Amazon Web Services — aws.amazon.com,” https://aws.amazon.com/lambda/.

[22] “IBM,” https://www.ibm.com/.

[23] M. Rosenblum, “The reincarnation of virtual machines: Virtualization makes a comeback.” *Queue*, vol. 2, no. 5, pp. 34–40, 2004.

[24] “VMware,” https://www.vmware.com/.

[25] J. E. Smith and R. Nair, “The architecture of virtual machines,” *Computer*, vol. 38, no. 5, pp. 32–38, 2005.

[26] M. J. Scheepers, “Virtualization and containerization of application infrastructure: A comparison,” in *21st twente student conference on IT*, vol. 21, 2014.

[27] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, “Firecracker: Lightweight virtualization for serverless applications,” in *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, 2020, pp. 419–434.

[28] V. Dukic, R. Bruno, A. Singla, and G. Alonso, “Photons: Lambdas on a diet,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 45–59.

[29] E. Johnson, E. Laufer, Z. Zhao, S. Narayan, S. Savage, D. Stefan, and F. Brown, “Wave: a verifiably secure webassembly sandboxing runtime.”

[30] S. Shillaker and P. Pietzuch, “Faasm: Lightweight isolation for efficient stateful serverless computing,” in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 419–433.

[31] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, “Benchmarking, analysis, and optimization of serverless function snapshots,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 559–572.

[32] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, “{SOCK}: Rapid task provisioning with {Serverless-Optimized} containers,” in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 57–70.

[33] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, “{SAND}: Towards {High-Performance} serverless computing,” in *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 923–935.

[34] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, “Faastlane: Accelerating {Function-as-a-Service} workflows,” in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 805–820.

[35] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, “Erim: Secure, efficient in-process isolation with memory protection keys,” in *Proceedings of USENIX Security Symposium*.

[36] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, “Jenny: Securing syscalls for {PKU-based} memory isolation systems,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 936–952.

[37] J. Gu, H. Li, W. Li, Y. Xia, and H. Chen, “{EPK}: Scalable and efficient memory protection keys,” in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 609–624.