



SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

**Serverless computing in unikernels using
eBPF code injection**

Milen Vitanov



SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics


**Serverless computing in unikernels using
eBPF code injection**

**Serverloses Computing in Unikernels unter
Verwendung von eBPF-Code-Injektion**

Author:	Milen Vitanov
Supervisor:	Prof. Dr. Pramod Bhatotia
Advisors:	Ilya Meignan-Masson Masanori Misono
Submission Date:	01.12.2024

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 01.12.2024



Milen Vitanov

Acknowledgments

I would like to extend my gratitude to my advisors Dr. Masanori Misono and Ilya Meignan–Masson who provided valuable insights and feedback throughout the development of this project. Special thanks go to Prof. Dr. Pramod Bhatotia for his guidance and support in refining this research.

Additionally, I would like to thank the open-source community for their contributions to the unikernel and eBPF technologies, which were fundamental to the success of our solution.

Abstract

Unikernels are gaining more and more popularity due to their minimalist, secure and efficient environments. Such advantages are quite appealing to any cloud-native or serverless application. However, serverless frameworks require dynamic code injection, which is not allowed by unikernels due to their isolation level. This challenge can be approached by integrating eBPF, which allows attaching custom logic to hooks inside the kernel. This thesis proposes a serverless data processing framework that uses eBPF in combination with unikernels to provide fast and lightweight images while still allowing updates to the processing logic without the need to reboot or recompile the system.

The solution follows the MapReduce execution model and allows for the injection of either JIT-compiled or interpreted BPF bytecode. Through qualitative analysis, the thesis outlines the complexity and limitations introduced in comparison with traditional approaches that do not allow dynamic code injections. Quantitative analysis shows the small overhead introduced in startup-time, which becomes insignificant with the increase of re-configurations. It also demonstrates improvements in flexibility while performance, memory usage, and security still remain competitive. Additionally, the framework is evaluated with CPU-intensive tasks to demonstrate its capability to handle resource-demanding workloads, which is required in many applications such as scientific computing, machine learning or image processing. Finally, its versatility is demonstrated through implementing use cases such as a search engine and inverted index, which prove its adaptability to a wide range of big data applications.

The implementation done throughout this research is available at <https://github.com/mvitanov/MapReduce-in-unikraft-using-BPF>

Keywords: Unikernels, serverless, BPF, eBPF, Data processing, MapReduce

Contents

Acknowledgments	iii
Abstract	iv
1 Introduction	1
2 Background and related work	3
2.1 Serverless Computing	3
2.2 Distributed Data Processing: MapReduce	4
2.3 Unikernels: A Lightweight Virtualization Approach	6
2.4 extended Berkeley Packet Filter (eBPF)	7
2.5 eBPF in Unikernels	8
2.6 Existing Serverless Tools for MapReduce	8
3 Overview	10
3.1 Motivation	10
3.2 Design Goals	11
3.3 System Overview	13
3.4 Design Challenges and Key Ideas	14
4 Design	15
4.1 Architecture Components	16
4.1.1 Scheduler	16
4.1.2 Shared File System	16
4.1.3 Unikernels	17
4.2 Integrating unikernels and eBPF into the data processing workflow . .	18
4.3 Defining the BPF Helper Functions	19
4.3.1 Data structure helpers	19
4.3.2 Data helpers	22
4.3.3 Debugging helpers	22
4.4 BPF Verification	23

5	Implementation	26
5.1	Unikraft	26
5.2	uBPF	26
5.3	Software Components	27
5.3.1	Scheduler	27
5.3.2	Unikernels	27
5.3.3	Shared Data Storage	28
5.4	Communication Data Structure	29
5.5	BPF Program Context	30
5.6	BPF Helper Functions	32
5.6.1	Debugging Helpers	33
5.6.2	Hashtable Helpers	33
5.6.3	Data Helpers	35
5.7	uBPF Function Implementation and Compilation	36
5.8	Unikernel Configuration	37
5.8.1	NonBPF unikernel	38
5.8.2	Unikernel image with BPF functionality	38
5.8.3	Unikernel image with BPF JIT functionality	38
5.9	Code Verification	38
5.9.1	PREVAIL Platform	39
5.9.2	PREVAIL Usage	41
5.9.3	Secure Verification	42
6	Evaluation	43
6.1	Qualitative Analysis	43
6.1.1	Development Complexity	43
6.1.2	Debugging	43
6.1.3	Learning curve	44
6.1.4	Code Maintenance	44
6.2	Quantitative Evaluation	45
6.2.1	Unique Word Count	45
6.2.2	CPU Intensive BPF functions	55
6.2.3	Other use cases	57
7	Conclusion	63
8	Future Work	65
8.1	Parallelism	65
8.2	Beyond MapReduce	65

Contents

8.3 Enhancing Debugging Tools and Standardizing the Verifier	66
8.4 Integrated eBPF verification	66
8.5 Shared Memory between the Unikernels and the Scheduler	66
Abbreviations	67
List of Figures	69
List of Tables	71
Bibliography	72

1 Introduction

Serverless computing has emerged as a powerful tool in any major cloud provider. It allows the end users to define the core business logic of their applications in the form of stateless functions. The technology abstracts away server management, offering automatic scaling, high availability, and a pay-as-you-go billing model. This significantly decreases the time needed for an application to be developed, which makes serverless very attractive to the industry. The advantages are not only for the users, but also for the providers that offer serverless. As with any technology, serverless also requires resources and a runtime environment. However, it only uses them while executing the functions, which means that the resources are available for other serverless users after the execution. This is a significant improvement compared to the resource utilization in traditional computing models, where resources are allocated per user, independent of whether they are efficiently utilized or not [1].

Despite all these benefits, traditional serverless architectures still face challenges in performance overhead and resource utilization efficiency. Any serverless computing framework needs an execution environment for the custom short-lived functions. This execution environment should provide some required OS services that guarantee successful execution and security isolation. However, it also needs to be lightweight, consume as few resources as possible, and be performant [2].

State-of-the-art serverless frameworks are currently offering a virtual machine or a container equipped with a general-purpose OS as a serverless execution environment. This environment does not minimize resource utilization because it creates a whole general-purpose OS just for executing lightweight functions. Moreover, the usual case is that hundreds or thousands of functions may run on a single host, which would create many execution environments that provide a lot of unnecessary functionality for executing the serverless functions. Additionally, extra startup time is needed to initialize all these runtime environments and to allocate all resources needed for them [2].

In order to handle these challenges, we can use unikernels instead of general-purpose OSes as execution environments. Unikernels are VMs designed to run a single appli-

cation in the cloud. They do not come with a general-purpose OS but a customizable set of libraries that can run directly on the virtual machine and provide OS-like functionality. They also do not have expensive context switches between user and kernel space abstractions as a general-purpose OS. This lightweight library inclusion technique and single address-space approach improve the overall performance and contribute to shortening the boot time. Unikernels' level of isolation is guaranteed by hardware primitives instead of an OS, which significantly increases the security [3]. However, this also means we cannot perform any code injections without rerunning the unikernel.

Code injections are a main requirement for any serverless approach. The user provides a compiled binary code of the function, which is then loaded by the framework and executed at runtime. This cannot be directly done within a unikernel, due to its level of isolation. In order to allow for such a technique, we can make use of another technology called eBPF.

BPF is a technology developed for network packet filtering. It has been extended into a much more versatile technology called extended Berkeley Packet Filter (eBPF). eBPF is a revolutionary technology originally developed for Linux, which enables the safe and efficient execution of user-defined programs within the kernel space of an operating system [4]. We can incorporate eBPF into a unikernel in order to allow for dynamic code injections in a secure manner. This capability is essential for optimizing runtime operations and ensuring high-security standards in serverless environments.

This thesis aims to enhance serverless computing frameworks by providing a performant and secure new alternative built on unikernels with integrated eBPF code injections. As proof of concept, we decided to implement a concrete serverless data processing algorithm. We chose MapReduce because it is widespread, simple to implement, and easy to evaluate. We show how MapReduce can benefit from the lightweight and isolated nature of unikernels, combined with the dynamic capabilities of eBPF.

The remainder of the thesis is organized as follows: [Section 2](#) reviews related work in serverless computing, unikernels, and eBPF. [Section 3](#) outlines the design goals we set and the challenges we encounter as we develop our idea. [Section 4](#) provides details about the architecture design and explains how it fulfills the design goals. [Section 5](#) lists some important implementation details, whereas [Section 6](#) presents the evaluation methodology and results from our implementation. Finally, [Section 7](#) concludes with a discussion of the findings, while [Section 8](#) offers opportunities for potential future work.

2 Background and related work

2.1 Serverless Computing

Serverless computing is a cloud computing Function-as-a-Service framework. [Figure 2.1](#) illustrates its key characteristics.

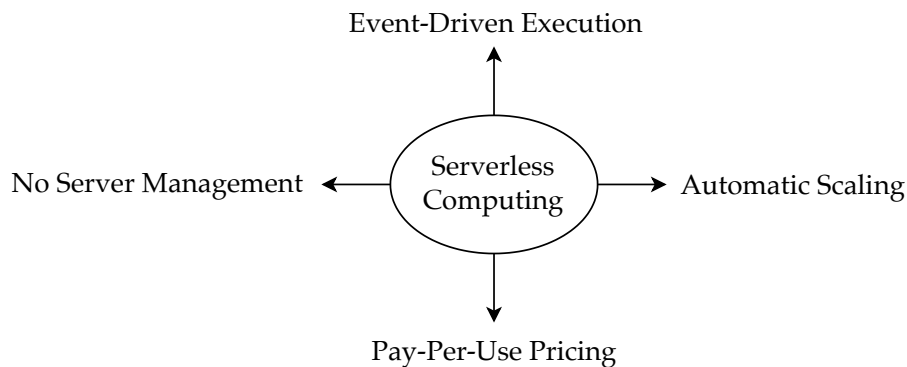


Figure 2.1: Serverless Key Characteristics

The idea behind serverless is to allow users to define their applications as stateless functions and run them on the provided infrastructure immediately or as a response to events such as HTTP requests, database updates, or scheduled tasks. This allows for allocating just as many resources from the cloud as needed for the duration of the execution. Therefore, serverless computing eliminates the problem of idle allocated resources, which exists in conventional server computing, i.e., the users are paying just as much as their serverless applications are using [5].

It is important to highlight that serverless does not mean there is no server. The execution model is designed so that the infrastructure provider dynamically manages the allocation and distribution of servers. This also allows for automatic scaling, meaning that the serverless provider allocates resources whenever the functions require that and deallocates them when no longer needed. In this way, the serverless model abstracts

away the complexities of infrastructure management, allowing the users to focus on writing only their core application functionality, which will be used to serve any client requests [1].

The first serverless platform released is AWS Lambda in 2014 [6]. It sets the standards for serverless computing. Other popular serverless solutions are Google Run Functions [7] and Microsoft Azure Functions [8]. All of these provide robust environments for deploying event-driven applications and set the industry standard for serverless computing.

2.2 Distributed Data Processing: MapReduce

Since 2014, there has been a significant increase in the adoption of serverless architectures. This increase has inevitably affected the data processing frameworks as well. Various data pipeline approaches have already been developed. The core industry requirements are **seamless data integration, transformation, and delivery**. From a technical perspective, the serverless data processing workflows must be **efficient, scalable, and cost-effective**. In general, these data pipelines are organized in various stateless processing stages, which can be handled by a serverless infrastructure [9]. Two of the most popular frameworks used for distributed data processing are Apache Hadoop [10] and Apache Spark [11].

Apache Hadoop is a scalable, fault-tolerant distributed processing framework for large data sets. Its main components are the Hadoop Distributed File System (HDFS) and the MapReduce computation model. HDFS performs data partitioning, i.e., it divides the data into smaller batches. After that, it distributes the data across different nodes and synchronizes any modifications to the batches. MapReduce takes these batches as input to its mappers and reducers in order to perform the desired data transformations and computations until the final result is reached [12].

The other popular data processing framework, Apache Spark, is built using Resilient Distributed Datasets (RDDs). These are read-only, resilient collections of objects partitioned across multiple nodes [13]. The computations are performed only when needed on these RDDs, which are also ephemeral, meaning that they utilize resources only during the computation. The computational model on the RDDs follows and extends the popular MapReduce algorithm. It enhances it by allowing iterative models on nodes and caching of some mapping phases for later uses.

As it can be seen, the popular distributed data processing frameworks are more or less relying on the traditional MapReduce algorithm. MapReduce is a programming model introduced by Google for processing and generating large datasets in a distributed manner. Traditionally, it is implemented using a distributed file system (HDFS), which provides fault tolerance and high throughput for data-intensive applications [14]. However, the increasing popularity of serverless computing has led to a big simplification of the MapReduce workloads. It has two stateless processing stages - map and reduce. [Figure 2.2](#) shows how the **Map** phase processes input data and generates intermediate key-value pairs. [Figure 2.3](#) outlines how the **reduce** phase takes these intermediate key-value pairs and merges them to produce the final output.

```
1 map(String key, String value) {  
2     for each partition p in value {  
3         EmitIntermediate(p, 1);  
4     }  
5 }
```

Figure 2.2: Map Function [15]

```
1 reduce(String key, Iterator values) {  
2     int result = 0;  
3     for each v in values {  
4         result += v;  
5     }  
6     EmitFinal(result)  
7 }
```

Figure 2.3: Reduce Function [15]

These two very simple functions are powerful enough to cover a great variety of use cases like creating search indices, recommending systems, social network analysis, web crawling, etc. All this is achieved just by changing the logic of the stateless functions while the rest of the MapReduce infrastructure stays the same.

The good performance of the model comes from its efficient data partitioning. The partitions are then distributed across a cluster of nodes in the form of tasks. This allows for robust parallel processing of very big data in finite time [15].

2.3 Unikernels: A Lightweight Virtualization Approach

Unikernels are a novel approach to virtualization. Unlike general-purpose operating systems designed to run multiple applications, unikernels are specialized, single-address-space machine images that include only the necessary components to run a specific application. There is no division into user space and kernel space, which eliminates the context switches that OSes have. This makes unikernels execute applications faster. They do not have any real operating system, but rather simulate OS-like functionality by adding lightweight custom libraries. [Figure 2.4.](#) compares unikernels to existing alternatives like traditional VMs, running standalone OSes, docker containers running on hosts and containers running in VMs.

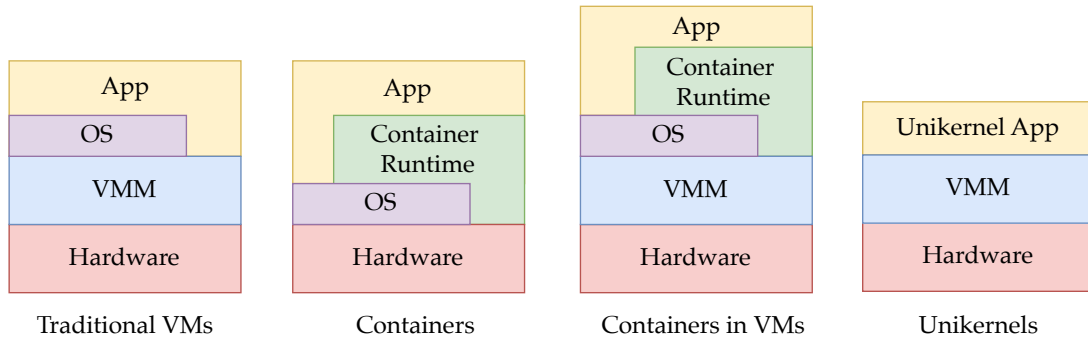


Figure 2.4: High-level overview of existing virtualization technologies [16]

As it can be seen, the unikernels provide the simplest and most direct infrastructure by directly executing the application on the hypervisor. A more detailed characteristic comparison is outlined in [Table 2.1.](#)

	VMs	Containers	Unikernels
Time Performance	Slowest	Fast	Fast
Memory Footprint	Heavy	Depends of features	Light
Security	Very Secure	Less Secure	Very Secure
Features	All	Depends on need	Only the absolute necessary

Table 2.1: Comparison between the different virtualizations [16]

We expect to have at least as good performance with unikernels as we have with containers but a lighter and much more secure execution environment. In the context of serverless, unikernels need to be able to deploy lightweight, isolated functions that

scale quickly in response to demand. Three key advantages are particularly important for this use case:

- **Minimalism:** Unikernels contain only the code necessary to execute a single application, resulting in small image sizes and reduced resource consumption [3].
- **Performance:** Due to their lightweight nature and single-address space, unikernels can boot in milliseconds, use fewer resources, and provide enhanced performance [17].
- **Security:** The reduced attack surface inherent in unikernels makes them more secure than traditional VMs or containers. Additionally, their isolation is provided by hardware primitives instead of an OS [18].

2.4 extended Berkeley Packet Filter (eBPF)

eBPF is a technology that can run sandboxed programs in a privileged context, such as an OS kernel. It is implemented as an in-kernel, event-based virtual CPU. This is different than a Virtual Machine (VM) due to the fact that it is working only within kernel space. Despite that, eBPF is often called an in-kernel VM by many sources. Historically, it is built on top of the Berkeley Packet Filter, also called BSD Packet Filter (BPF) [4]. BPF is a tool that lets you capture and filter network packets right from the operating system. It works like a network tap, making it easy to grab specific packets directly as they pass through the system [19].

eBPF is an extended version of the original BPF, which has two 32-bit registers and understands 22 different instructions [20]. eBPF extends the number of registers to eleven and increases their sizes to 64 bits. Its instruction set is also increased by adding new arithmetic and logic instructions for the bigger register size. The 64-bit registers map one-to-one to the hardware registers on the 64-bit architectures, which enables Just-in-Time (JIT) compilation into native machine code [21].

eBPF has evolved to be much more than a packet filter. It has the ability to call functions attached to specific kernel hooks. It provides a fixed-size memory region called stack (512 bytes per program) where data can be stored temporarily while the program is running. It also comes with a Linux verifier, which makes sure potentially dangerous code does not enter the kernel [4]. These advancements have transformed eBPF to have many applications beyond packet filtering, including performance monitoring, security intrusion detection or serverless and cloud-native applications [22].

2.5 eBPF in Unikernels

Unikernels do not come with Linux and, therefore, do not have direct support for eBPF. However, integration is possible thanks to already existing userspace BPF implementations such as uBPF [23].

When integrated with unikernels, eBPF can enhance the flexibility and functionality of these lightweight environments. It can be used to support:

- **Dynamic Functionality:** eBPF allows for the insertion of code at runtime, enabling the unikernel to adapt its behavior without needing to be rebooted or redeployed [24].
- **Performance Monitoring:** eBPF can be used to monitor and optimize the performance of tasks running within unikernels, providing real-time insights into resource usage and bottlenecks.
- **Enhancing Security:** eBPF VMs have their own isolation layer from the host, which contributes to even more secure executions.

2.6 Existing Serverless Tools for MapReduce

Current serverless tools and frameworks that support data workflows similar to MapReduce are implemented using Docker or VMs. These tools allow users to run distributed data processing tasks without worrying about the underlying infrastructure. The most popular such frameworks are listed below.

- **AWS Lambda with AWS Glue:** AWS Lambda can be used in combination with AWS Glue to run serverless ETL (Extract, Transform, Load) jobs. AWS Glue provides a serverless platform for running Spark jobs, which can be used to implement MapReduce data transformations. AWS Lambda functions can trigger these jobs, allowing for a fully serverless data processing pipeline [25].
- **Google Run Functions with Google Cloud Dataflow:** Google Cloud Dataflow is a fully managed service for stream and batch data processing. It supports Apache Beam, a unified programming model that simplifies the implementation of MapReduce. Google Run Functions can be used to trigger Dataflow jobs, enabling serverless data processing with the flexibility of containers [26].
- **Azure Functions with Azure HDInsight:** Azure Functions can be integrated with Azure HDInsight, a cloud-based service that provides Apache Hadoop, Spark,

and Kafka. HDInsight allows users to run MapReduce jobs on VMs, while Azure Functions can trigger and manage these jobs in a serverless manner, reducing the need for infrastructure management [27].

- **Open source solutions:** There are many open source data processing frameworks which can also be easily adapted to serverless MapReduce. A very popular example is Apache OpenWhisk. It is an open-source event-driven serverless platform that uses Docker containers [28].

These tools illustrate how MapReduce can be adapted to serverless computing frameworks in order to use the scalability and flexibility of cloud-native technologies like Docker and VMs.

The already implemented solutions of serverless computing, eBPF and unikernels represent significant advancements in distributed data processing. Serverless computing offers a scalable and cost-effective platform, which can be used for easy-to-implement and easy-to-use algorithms like MapReduce. Unikernels provide the performance, minimalism and security needed to optimize complex workloads in a cloud-native environment. Integrating the eBPF technology into the unikernels allows external custom functions to be loaded into the single-address space of the machine without the need to recompile or reboot. As cloud computing continues to evolve aggressively, all these technologies will likely develop even further.

3 Overview

This section presents the motivation for our proposed framework, its design goals and challenges.

3.1 Motivation

Based on the background research we did in [Section 2](#), we observed some existing weaknesses in the current serverless solutions that limit their effectiveness in distributed computing environments. They are summarized in [Figure 3.1](#).

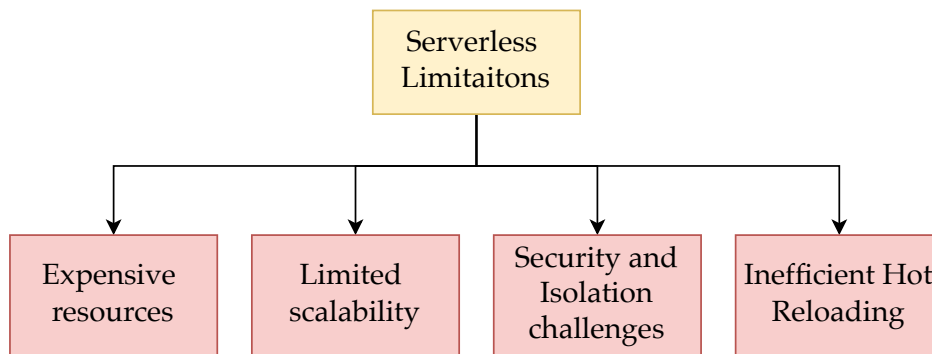


Figure 3.1: Serverless Limitations

Serverless platforms often rely on general-purpose operating systems, which can lead to unnecessary overhead. Loading and running entire OS components that are not directly needed for the execution of the stateless functions results in higher memory usage, increased resource costs, and slower execution times.

Another limitation relates to the overall performance and scalability. While automatic scaling is already offered, the reliance on traditional virtualization or containerization introduces overhead due to expensive context switches and slower instance initialization time. These inefficiencies hinder the ability to process large-scale, distributed

workloads in the most effective way.

There is also room for improvement when it comes to security. Many serverless platforms lack robust isolation mechanisms, which increases the risk of vulnerabilities spreading across distributed tasks. The use of shared resources among tasks further exacerbates the attack surface, making it difficult to enforce strong security guarantees.

Some existing solutions even struggle with flexibility in real-time environments. Modifying running applications to adapt to new use cases or workloads could require restarting services, leading to downtime and disruptions in the continuous data processing workflows.

In this thesis, we propose an innovative enhancement to the serverless frameworks that addresses all these limitations. We replace traditional VMs with unikernels to achieve image minimalization, faster performance and better security. The unikernels are equipped with the eBPF technology, which allows dynamic code injections to achieve a proper serverless nature.

3.2 Design Goals

The key goals we identified in order to provide the above-mentioned solution are

- **Efficient resource utilization:** Our system needs to compile into minimalist images. Unikernels seem to be a very good fit to achieve this goal since they compile and load only the necessary components into the machine image, i.e., they do not need a general purpose OS. Thus, minimized resource usage is ensured, leading to lower costs and faster execution times.
- **Good performance and scalability**
To make our solution relevant to the industry, we must show it is highly performant and easily scalable. We suggest that the automatic scaling capabilities of serverless platforms in combination with the short boot times of unikernels allows for fast and competitive processing of distributed data workloads. The lack of context switches additionally boosts the performance.
- **High Security and Isolation**
Our system should offer a higher isolation level and security. Unikernels provide very good security by design due to their minimal attack surface. In a serverless data processing architecture, each distributed task can be isolated within its unikernel instance, reducing the risk of security vulnerabilities spreading across

the system. Adding eBPF provides one more layer of isolation, further decreasing the system vulnerabilities. On top of that, all BPF bytecodes will undergo kernel verification. This verification will ensure the injected code is not harmful and the kernel will stay healthy upon and after the execution.

- **Dynamic Code Injection**

In the context of serverless workloads where data is processed in real time, the ability to load external code is particularly valuable in order to maintain a high availability and minimize downtime. To allow this with unikernels, we integrate the eBPF technology. eBPF gives us the opportunity to inject code into a running unikernel. This means we can modify a running unikernel application without the need for a restart or reboot, thus loading external binary code and adapting the serverless infrastructure to a custom use case.

- **Flexibility**

Our solution should be easily adaptable to different real-world scenarios. We chose the MapReduce execution model for our solution since it is generic enough to be implemented in many industry use cases. With the custom-defined map and reduced functions, the user will have the possibility to process the datasets in an entirely custom way and will also be able to chain the unikernels in a way that suits the use case.

- **Usability**

Last but not least, we want to ensure that our solution is easy to use. It should not be required that users have complex knowledge about unikernels, eBPF, or MapReduce functionality. A user of the framework should only know how to write a map and a reduce function. Then, documentation will be provided on injecting these functions and running the framework. All this should happen with a simple interface API.

3.3 System Overview

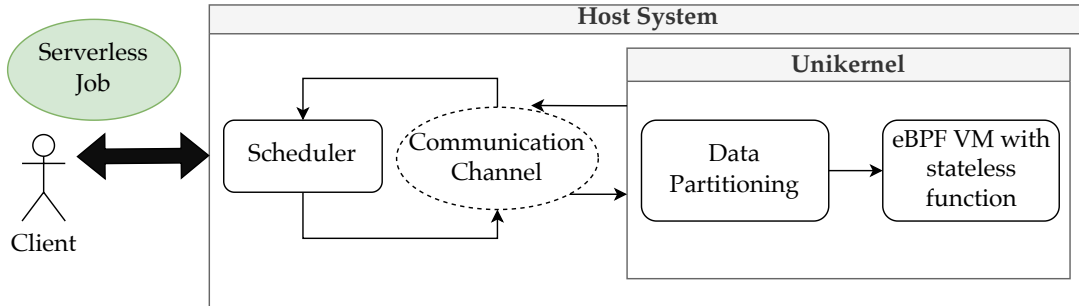


Figure 3.2: High-level overview of the framework

The high-level overview of our solution is outlined in [Figure 3.2](#). It consists of two main components linked by a communication channel.

1. Communication Channel

We use a shared file system between the scheduler and the unikernels to allow for synchronized communication and avoid data corruption.

2. Scheduler

The scheduler is responsible for receiving the request from the client and uploading it to the correct shared file system location. The unikernels process the data at that location and upload the result to another predefined shared location. Once the scheduler detects the result, it returns it to the client.

3. Unikernels

- **Data Partitioning**

Each unikernel partitions the dataset into batches. Custom logic is applied then to each of the batches.

- **eBPF VM**

Each unikernel is equipped with an **eBPF** VM that allows for dynamic code injections into the application. The dynamically updated functionality is applied to each batch, which transforms them according to a custom-defined function. In this way, the execution model can be tuned to various use cases.

3.4 Design Challenges and Key Ideas

While developing the proposed framework, we have encountered the design challenges listed below.

1. **Integrating unikernels and eBPF into a data processing algorithm**

This challenge is to use the advantages of both unikernels and eBPF technology in a data processing framework while maximizing efficiency. Our solution should provide seamless, high-performance communication between unikernels and their internal eBPF VMs, facilitating efficient, low-latency processing of any serverless tasks.

2. **Defining the eBPF helper functions**

Developing helper functions that serve as a communication layer between the unikernel environment and eBPF requires a deep understanding of both environments. These functions must not only be compatible with the eBPF execution model but also capable of handling specific tasks required by the data processing framework, such as data parsing and synchronization with the shared file system. Moreover, since helper functions are crucial for extending eBPF's capability to interact with external resources, defining them with performance and security in mind is crucial.

3. **eBPF Verification**

Since we set a goal to provide high security and allow dynamic code injections into the kernels, we need to have a robust and strict verification of the injected bytecode in order to ensure the safety of the unikernel. We can achieve this by leveraging some of the existent eBPF verification mechanisms. However, these mechanisms are usually not lightweight and require a lot of resources. They also introduce a small overhead in the system. Therefore, keeping any verification delays to a minimum is challenging while keeping the process robust and strict.

4 Design

This chapter outlines the architectural design of the proposed framework in more detail.

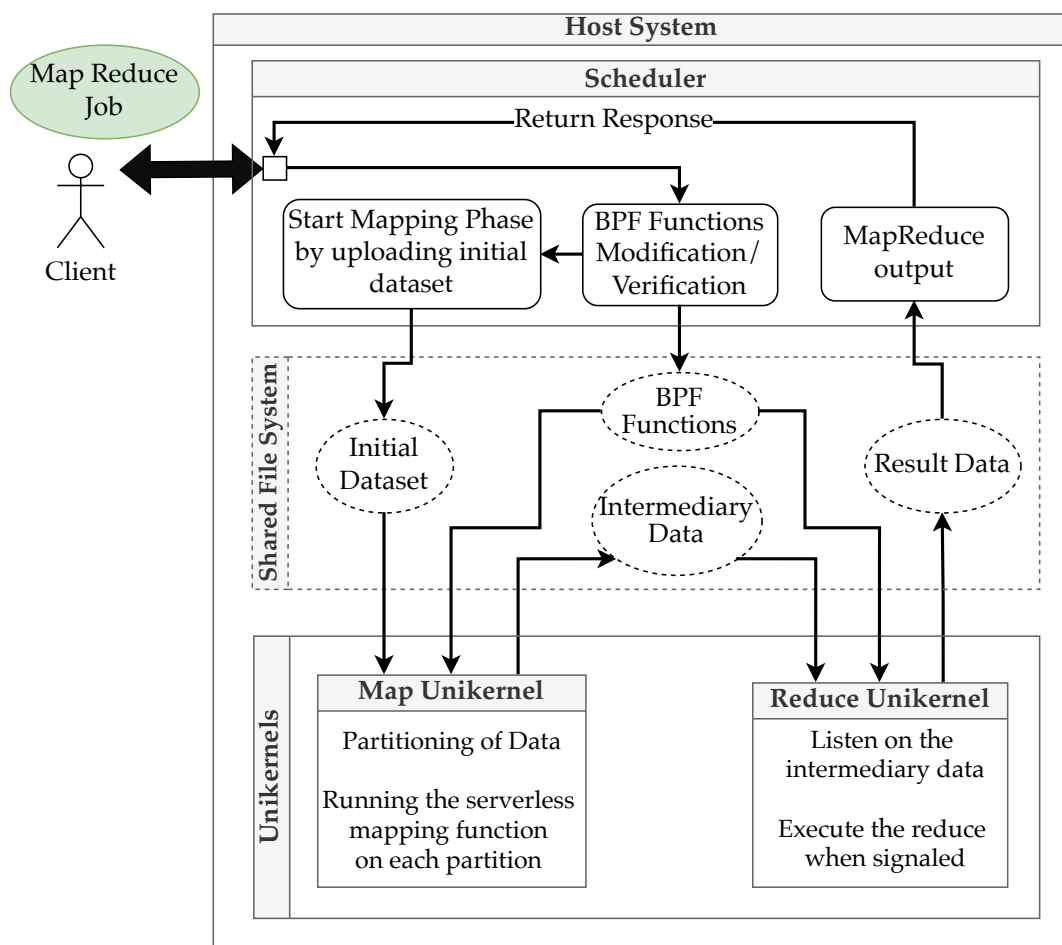


Figure 4.1: Architecture design of the framework

Our implementation chooses the MapReduce algorithm since it is a foundational execution model in the majority of the existing data processing frameworks, as discussed in [Section 2](#). We design the Map and the Reduce phases to be executed by different unikernels, which act as function workers.

4.1 Architecture Components

The following components make up the architecture shown in [Figure 4.1](#).

4.1.1 Scheduler

The scheduler is a small executable running directly on the host that processes MapReduce requests. If a request is triggered, the scheduler performs two steps:

- Uploads the BPF functions to a shared file system location and triggers a verification process.
- Retrieves the dataset and uploads it to another shared file system location for the unikernels to process.

After the upload is complete, the scheduler waits for the result location to be filled. Once this is done by the last unikernel, the result is propagated back to the client.

4.1.2 Shared File System

The shared file system is used for all the communication points between the scheduler and the unikernels. It is the only point where a state can be maintained between the different phases of the workflow. We define four main regions:

- **BPF functions**
All BPF-compiled custom bytecode is stored here. This location also holds code signatures, which are used by the unikernels to verify the bytecode is secure before loading and executing it.
- **Initial Dataset Region**
This is the FS location where the scheduler uploads the dataset to be processed by the MapReduce framework.
- **Intermediary Data Region**
Here, the mapper unikernel is uploading the transformed dataset. Upon a successful upload, a reducer unikernel retrieves this data as an input argument.

- **Result Data Region**

This is the file system location, which the Reducer unikernel uses to propagate the final result back to the scheduler.

4.1.3 Unikernels

In its simplest configuration, our framework operates with two unikernels running concurrently and monitoring the shared file system (FS) regions: one functioning as the mapper and the other as the reducer. Both unikernels share an identical architectural design, as illustrated in [Figure 4.2](#).

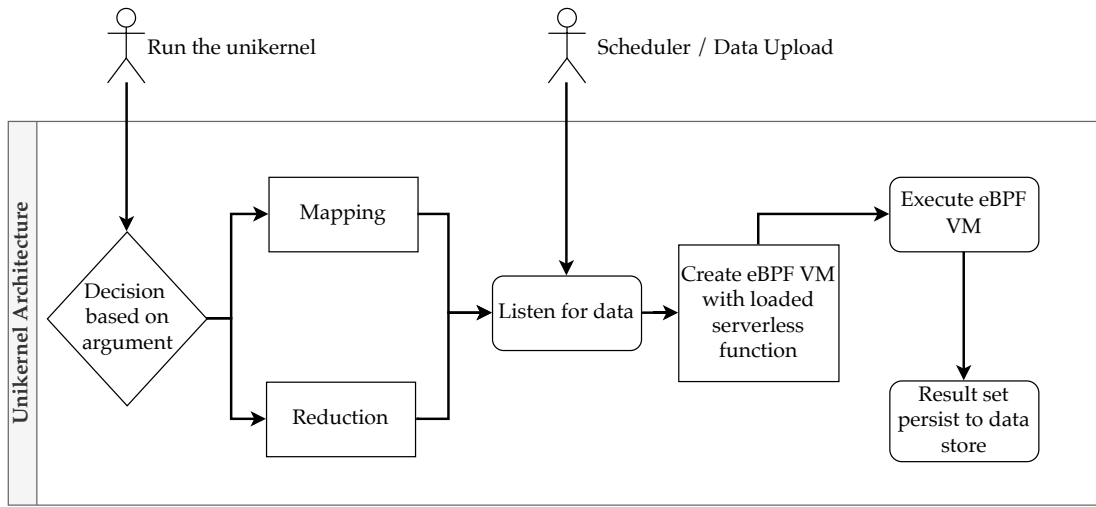


Figure 4.2: Unikernel architecture design

Before running the unikernel, its processing stage needs to be specified as either map or reduce. After that, it starts monitoring the initial dataset FS region. Once data comes, the corresponding BPF serverless function is verified by checking its signature. If the signature is invalid, the bytecode is rejected and the unikernel refuses to process the incoming dataset. If it is valid, the external code is loaded and the processing begins. First, the dataset is partitioned to the needed granularity (batch size), and then each data portion gets propagated to the loaded BPF function. The output of the function is persisted back to the shared file system.

4.2 Integrating unikernels and eBPF into the data processing workflow

Our design’s primary focus is on how to effectively structure unikernels and eBPF for the phases of the data processing algorithm. Since we have chosen to use MapReduce, in our case we have two phases - the map and the reduce. Unikernels are minimalist and specialized, so they aren’t well-suited to multi-role flexibility. Therefore, we have designed each unikernel to perform a specific phase in the workflow, aligning its purpose to either execute the map or the reduce logic.

This fixed-role structure simplifies the unikernel’s purpose and makes the system more predictable and efficient. However, it also requires us to carefully synchronize how each phase interacts and transitions in the overall execution. We need to ensure that each unikernel phase maintains smooth operation without the need for complex resource reallocation.

To achieve this, we design a communication layer between the host and the unikernels. We adopt a shared file system model to allow data transfer between the host and the unikernel environment. This approach provides low-latency data access for unikernels by reducing the overhead of complex networking and creating a more direct and efficient data flow. It also maintains easy access to shared resources, ensuring data availability without conflicts.

To address the serverless nature of the data processing framework, our design integrates eBPF. This technology enables dynamic injection of core functions directly into the unikernels. Through eBPF, we can manage the Map and Reduce phases as modular elements. Each of them is encapsulated within a unikernel that acts as a function worker. eBPF enables these function workers to be dynamically updated, allowing for quick adjustments to data processing logic in response to input data or processing requirements changes. This approach aligns with our goal to create a flexible, efficient system that can quickly adapt to new data processing needs without downtime.

However, eBPF-injected functionality does not have direct access to the unikernel memory. Therefore, we must design a communication layer between the unikernels and the eBPF virtual machines. Our design benefits from the possibility of creating custom eBPF helper functions, which are used by the eBPF VM to directly manipulate data in the unikernel single address space.

4.3 Defining the BPF Helper Functions

We need to create helper functions that enable the eBPF VM to interact with the unikernel environment in a way that accommodates the specific requirements of the MapReduce execution model. We approached this by carefully examining common use cases. For instance, we looked at widely used MapReduce applications such as **Word Count**, **Indexed Search**, and **Log Analysis**:

- **Word Count:** The map function counts occurrences of words, while the reduce function aggregates counts. Our BPF helpers handle string processing, data traversal and hashtable manipulation.
- **Indexed Search:** The map function emits a list of document IDs pointing to the number of occurrences of the searched word. The reduce function returns the document id with the most occurrences. BPF helpers are used for direct access to the map keys and value, as well as data traversal.
- **Log Analysis:** Map functions parse and extract log information, while reduce functions aggregate this data for analysis. BPF helpers assist with data parsing and filtering to ensure that only requested log information is processed and aggregated.

By focusing on these and other common cases, we defined a set of BPF helper functions that provide flexible, high-performance operations tailored to the data processing framework's needs. These functions are also generic enough to accommodate for other use cases. We defined three categories of BPF Helper API.

4.3.1 Data structure helpers

These helpers provide an API to read and modify the hashtable data structure, which is used as a main processing data abstraction. Its state is persisted to the intermediary data storage once the processing phase is completed.

Descriptions of currently available hashtable helpers is available in [Table 4.1](#) and [Table 4.2](#). Their technical prototypes can be found in [Table 5.2](#), [Table 5.3](#) and [Table 5.4](#).

#	Helper Name	Description
1	bpf_hashtable_get_int	Retrieve a number value linked to a key
2	bpf_hashtable_get_map_key_int	Retrieve a number value linked to a key, which is a value to another map key
3	bpf_hashtable_get_data_key_int	Retrieve a number value linked to a key, which is a substring from data at offset
4	bpf_hashtable_get_node_address	Retrieve a hashtable node linked to a key
5	bpf_hashtable_get_map_key_node_address	Retrieve a hashtable node linked to a key, which is a value to another map key
6	bpf_hashtable_get_data_key_node_address	Retrieve a hashtable node linked to a key, which is a substring from data at offset
7	bpf_hashtable_insert_int	Insert a number for a key
8	bpf_hashtable_insert_map_key_int	Insert a number for a key, which is a value to another map key
9	bpf_hashtable_insert_data_key_int	Insert a number for a key, which is a substring from data at offset
10	bpf_hashtable_insert_value	Insert a value for a key
11	bpf_hashtable_insert_map_key_value	Insert a value for a key, which is a value to another map key
12	bpf_hashtable_insert_data_key_value	Insert a value for a key, which is a substring from data at offset
13	bpf_hashtable_get_size	Retrieve hashtable bucket size
14	bpf_hashtable_get_first_node	Retrieve first node address at bucket index
15	bpf_hashtable_get_next_node	Retrieve next linked node to a given node
16	bpf_hashtable_get_node_value_int	Retrieve a number value of a node
17	bpf_hashtable_get_node_value_size	Retrieve the value size of a node
18	bpf_hashtable_get_node_value	Retrieve the value of a node
19	bpf_hashtable_get_node_key	Retrieve the key of a node

Table 4.1: Hashtable Helpers API (1)

#	Helper Name	Description
20	<code>bpf_hashtable_initialize</code>	Initialize new hashtable in host
21	<code>bpf_hashtable_free</code>	Deallocate hashtable
22	<code>bpf_hashtable_get_value_address</code>	Retrieve value address linked to a key
23	<code>bpf_hashtable_get_map_key_value_address</code>	Retrieve value address linked to a key, which is a value in another map
24	<code>bpf_hashtable_get_data_key_value_address</code>	Retrieve value address linked to a key, which is a substring from data at offset
25	<code>bpf_hashtable_insert_hashtable</code>	Inserts a map as a value in another map; key is a string
26	<code>bpf_hashtable_insert_map_key_hashtable</code>	Inserts a map as a value in another map; key is linked val to key is another map
27	<code>bpf_hashtable_insert_data_key_hashtable</code>	Inserts a map as a value in another map; key linked to a key, which is a substring from data at offset

Table 4.2: Hashtable Helpers API (2)

There are three groups of functions in the Hashtable Helpers API - `get_value`, `insert_value` and `get_node`. We allow either working directly with number 64-bit values or with generic data values. The keys for the hashtable functions can be retrieved in three ways.

- **Keys directly passed as arguments**
The simple get and insert helpers expect direct key strings passed as arguments to the helper methods.
- **Keys from the hashtable**
Helpers containing `map_key` in their names expect a key string passed as an argument. They retrieve the value of that string from the map and use it as another key to retrieve the desired value. These helpers are useful when the user passes arguments through the map. For example, if a search word is passed to the unikernel, it can be inserted into the map with a known key. The key is also known to the BPF functions, which can access and compute with it.

- **Keys from the data chunk**

Helpers containing `data_key` in their names expect a data pointer to the chunk currently processed and an offset in that chunk. These helpers compute the address with the offset and use it as a string key. They are useful when traversing chunks of data, splitting them, and inserting the split values into the map.

4.3.2 Data helpers

The batch of data to be passed to the eBPF VM is loaded in memory. Instead of passing the whole data chunk, we pass only its address location and provide an API through the BPF helper functions, which can be used to read and modify the unikernel current batch address space directly from the BPF function. The state of the address space is never persisted to the intermediary storage, therefore, we can define the modifications to the data as stateless.

#	Helper Name	Description
1	<code>bpf_data_get_next_index_delimiter</code>	Retrieve the next index pointing to a matching delimiter in the data chunk
2	<code>bpf_data_get_next_index_map_word</code>	Retrieve the next index pointing to a word linked to a map key
3	<code>bpf_data_update_index_content</code>	Update the data at index with a new byte content

Table 4.3: Data Helpers API

The helpers from [Table 4.3](#) allow for a manipulation of the data chunk passed to a BPF function. They provide the opportunity to divide the chunk by delimiters or to look for specific words in it. The update byte functionality can be used for the application of termination characters, which allow portions of the data to be used as keys for the hashtable. By providing this API, we allow for the distribution of the dataset across different map key-value pairs. The prototypes of the data helpers are available in [Table 5.5](#).

4.3.3 Debugging helpers

Debugging BPF functions can be particularly challenging due to their execution in a separate, isolated address space, which limits direct interaction with the program's runtime state. To address this, we provide specialized helper functions that enable printing diagnostic messages to the host's I/O interface. With the helpers listed in

[Table 4.3](#), developers can gain valuable insights into the behavior of their BPF programs without the need for complex debugging setups.

#	Helper Name	Description
1	bpf_print_text	Print a message up to a specified size
2	bpf_print_integer	Prints a 64-bit integer
3	bpf_print_hashtable	Prints a table content

Table 4.4: Debugging Helpers API

The prototypes of the debugging helpers are available in [Table 5.1](#).

4.4 BPF Verification

Allowing dynamic code injections in the unikernels using eBPF creates security risks similar to the ones faced by the Linux kernel when running untrusted eBPF programs. Untrusted programs could exploit the system's resources or crash the unikernel if they perform unsafe operations. In order to ensure good security of our framework and adhere to the design principles laid out in [Section 3](#), we design a verification step of all serverless code, which happens before the code can be loaded inside a running unikernel.

Our verification process has to be adequate for generic stateless functions and independent of their size. It must be robust when performing its static analysis to detect out-of-bounds memory accesses, illegal operations, type constraints or termination issues. As described in the paper "*Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions*" [29], all these requirements are fulfilled by the PREVAIL eBPF Verifier [30]. This verifier is also said to be capable of verifying loops of an arbitrary number of iterations, which would allow the users to write more complex BPF functions and suit a greater variety of use cases. Other advantages of PREVAIL are shown in [Table 4.5](#).

Safety Assurance	memory accesses, type constraints, termination, bounded loops, legal operation, limited stack size
Efficiency	fast performance, small resource usage
Precision	avoided rejection of valid eBPF programs

Table 4.5: PREVAIL Advantages

When integrating PREVAIL, we had two choices - to either decouple the process in an external software or to integrate it into the unikernel itself. We decided to decouple it mainly to keep the image minimalistic. All the benefits we have with this approach are outlined in [Table 4.6](#).

Reduced complexity	unikernel is kept minimal and optimized for its task
Flexibility	more powerful system can run the verification for resource-intensive analysis
Scalability	in multi-node environment, single centralized verifier can service multiple unikernels reducing duplication

Table 4.6: Decoupled Verification Advantages [31]

There are certainly some trade-offs to this approach. For example, integrating the verifier inside the unikernel would provide even better security by reducing the attack surface. It would also make the unikernels self-sufficient, i.e., independent from external systems.

The way the verification is designed for our proposed framework is shown in [Figure 4.3](#).

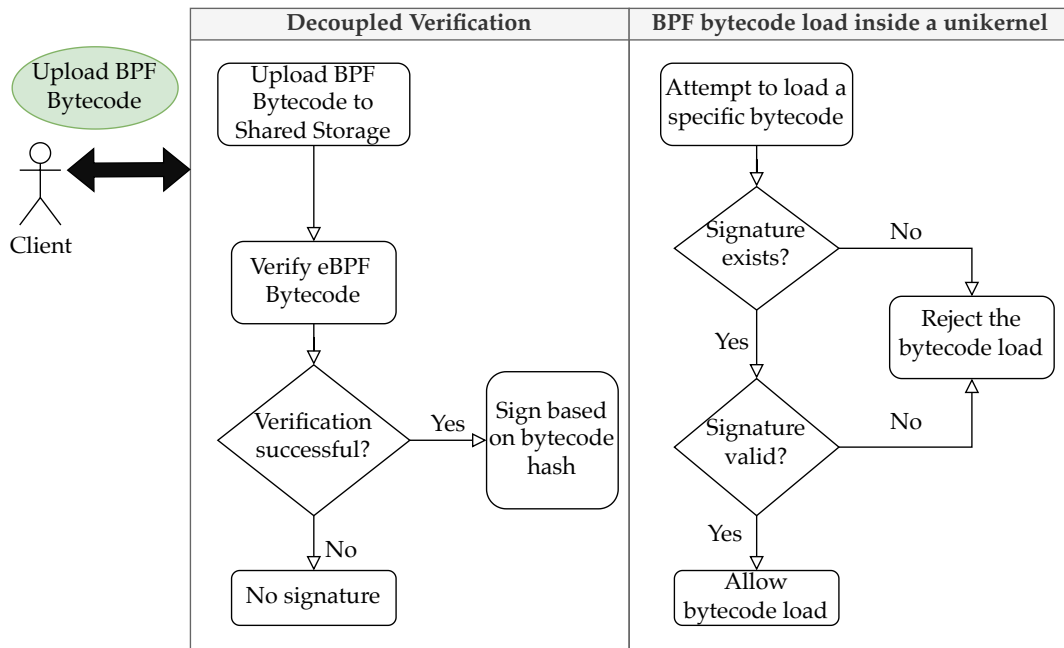


Figure 4.3: Decoupled Verification

We call the verification process on the compiled bytecode the moment it is uploaded to the shared data storage. Upon successful verification, the code is cryptographically signed using a hash of the bytecode itself. When a unikernel requests to load this bytecode, it checks whether a valid signature exists. In case there is none, the bytecode is rejected.

5 Implementation

This chapter outlines details about the implementation of our proposed solution. We present the key software components of our framework and the way they are built to fulfill the requirements outlined in [Section 3](#).

5.1 Unikraft

For our unikernel implementation, we use the Unikraft Development Kit. We chose this specific unikernel implementation because it lets us build optimized and easy-to-configure applications. Usually, creating unikernels involves a lot of manual setup and deep knowledge of OS internals. Unikraft changes this by offering a modular approach, where we only need to pick the parts we actually need. This modularity is a big advantage because it helps us keep our unikernel light and fast, trimming out any unused code that would normally slow things down [32].

Another reason we chose Unikraft is that it aims to simplify the overall unikernel development process. We can mix components, write custom unikraft libraries and reuse code across different projects. This flexibility makes it much easier to use unikernel applications for different purposes without starting a new implementation from scratch.

Last but not least, unikraft offers good performance. It is designed to have quick boot times, low memory usage, and high processing speed. That means we can achieve Linux-competitive performance for specific tasks, which is a huge advantage, especially when running big data processing workloads [32].

The unikraft version we use is v0.17.0 (Calypso) [33].

5.2 uBPF

To integrate eBPF in our unikernels, we use Userspace eBPF (uBPF) ported as a unikraft library [23]. The porting of the library is already done in another research by V. Hendrychová [34].

Since unikernels are single address-space machines, ported uBPF acts like the eBPF technology does in kernel space. It allows adding hooks at chosen places across the

unikraft application implementation and triggers the execution of loaded bytecode at runtime in these places.

This gives us the ability to inject code into the unikernel, but it also has its set of limitations. eBPF has a limited program context of 512 bytes, which restricts the complexity of the data structures that can be passed between the unikernel and the BPF programs [35]. This limitation can be tweaked by changing the upper bound accepted by the verifier. However, this is dangerous since it increases memory usage and decreases the performance of the BPF executions. Moreover, it contributes to a more complicated and time-consuming verification process.

Another uBPF limitation is the lack of helper functions and debugging capabilities. Due to these restrictions, its usage requires careful design and workarounds to achieve robust performance in big data processing use cases.

5.3 Software Components

The core software components of our implemented data processing framework are the scheduler and the running unikernels. In the Map-Reduce use case, we always have two running unikernels, each handling either the map or the reduce phase.

5.3.1 Scheduler

The scheduler is implemented in Python 3 and currently uses unikernels in a sequential way, meaning it does not implement any form of parallelism. This is sufficient for fulfilling our design goals outlined in [Section 3](#). The input datasets and BPF bytecode functions are retrieved from pre-defined locations in the file system to minimize the I/O latency.

5.3.2 Unikernels

The unikernel image is implemented to fit both phases of the MapReduce execution model. The control happens as an application argument upon running the image. Afterward, each image makes sure to validate and load the bytecode located at a pre-selected location in shared data storage between the host and the unikernel.

Unikernels run directly on bare metal. Therefore, we need an emulator to run them from a general-purpose OS and evaluate them. We chose to use QEMU as a virtualization emulator since it is platform-independent, easy to use, optimized, widely adopted in research, and well-documented [36]. The command to run a unikraft image is shown in [Figure 5.1](#).

```
1 sudo qemu-system-x86_64 \  
2   -enable-kvm \  
3   -cpu host \  
4   -m <memory-MB> \  
5   -nographic \  
6   -kernel <path-to-unikraft-image> \  
7   -append <"map"/"reduce">
```

Figure 5.1: Command to run a Unikraft image

5.3.3 Shared Data Storage

Since unikernels operate in their own isolated address space, direct communication with the host is not allowed by default. To allow data exchange, we decided to use shared data storage by mounting a file system to the unikernels. We chose the Plan 9 Protocol File System (9pfs) because it is well-supported by Unikraft and QEMU.

9pfs was originally developed for Plan 9 from Bell Labs [37]. It allows for the sharing of files and directories over a network in a lightweight manner, making it particularly suitable for simple and minimalist environments like unikernels. It does not require a complex configuration process, unlike more robust file systems such as NFS. The user simply selects a directory from the host file system, which is ported as root FS within unikraft. The necessary qemu arguments for allowing 9pfs are listed in [Figure 5.2](#).

```
-fsdev local,id=hostshare,path=<path-to-shared-folder>,security_model=none \  
-device virtio-9p-pci,fsdev=hostshare,mount_tag=fs0,disable-modern=on,disable-legacy=off \  

```

Figure 5.2: Additional QEMU arguments to run an image with 9pfs

In addition to configuring QEMU to use 9pfs, unikraft should also be compiled with the necessary components to allow proper access. The macros shown in [Figure 5.3](#) need to be present in the `.config` file of the unikraft project in order to successfully mount the shared file system with QEMU into the unikernel as a root file system.

```
1 CONFIG_LIBVIRTIO_9P=y
2 CONFIG_LIBVIRTIO_BUS=y
3 CONFIG_LIB9PFS=y
4 CONFIG_LIBUK9P=y
5 CONFIG_LIBUK9P_ERRATUM_WSTAT_FSYNC_ZERO=y
6 CONFIG_LIBVFSCORE_AUTOMOUNT_CI_9PFS=y
7 CONFIG_LIBVFSCORE_AUTOMOUNT_CI_9PFS_TAG="fs0"
8 CONFIG_LIBVFSCORE_AUTOMOUNT_CIO_DEV_ARG="fs0"
9 CONFIG_LIBVFSCORE_AUTOMOUNT_CIO_MP_ARG="/"
10 CONFIG_LIBVFSCORE_AUTOMOUNT_CIO_DRIVER_ARG="9pfs"
```

Figure 5.3: Unikraft config for mounting 9pfs as a root file system

5.4 Communication Data Structure

In order to allow application processing of generic datasets, we need to provide some data structure abstraction, which is able to translate any dataset to a program-readable input. The majority of the data processing frameworks are essentially based on mapping and transforming into <key, value> pairs. Therefore, we chose to provide a custom-implemented chained hash table in C as the main structure of our framework. Its UML diagram is shown in [Figure 5.4](#).

The implementation of the structure allows for text-based keys and generic values of arbitrary sizes. In this way, we allow enough flexibility such that the data structure can accommodate to any use case that might be implemented in the data processing execution model.

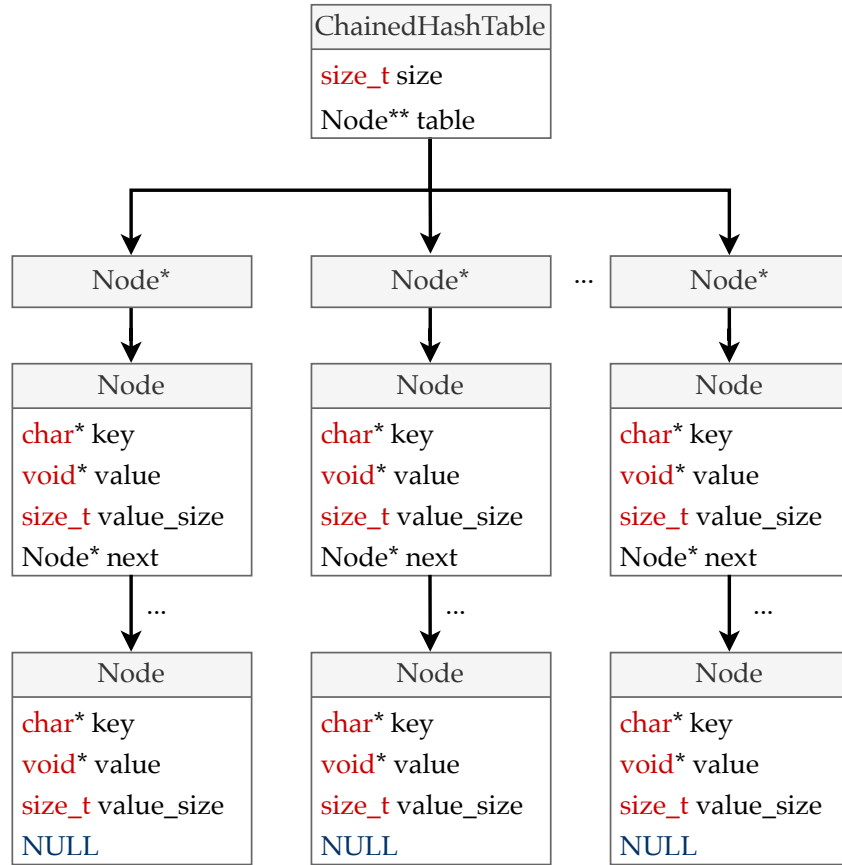


Figure 5.4: Chained Hashtable

5.5 BPF Program Context

When working with uBPF, we're dealing with a lightweight, userspace version of eBPF that gives us a flexible way to run BPF programs without the need for direct address-space access. However, without access to the host memory, communication becomes a little more complex. In order to allow for data flow, we take advantage of the 512-byte BPF program context, which is passed onto the execution stack. It allows the user to pass a limited number of statically allocated data structures. This means we are not allowed to use pointers or pointer arithmetics. Moreover, the usage of static arrays is also discouraged in the stack memory since it is very easy to surpass its limit

when using them. This leaves us with a very limited choice of a collection of built-in types, which is the reason why eBPF is great for handling small, fast interactions but less suited for operations that need large data buffers or intensive computations.

In order to overcome the stack memory limitation, we decided to pass concrete addresses onto the stack, which work together with custom-defined BPF helper functions that can access the host memory, where the passed addresses point. [Figure 5.5](#) shows the way we structured our program context.

<code>struct Context</code>
<code>uint64_t data_address</code>
<code>uint64_t data_size</code>
<code>uint64_t data_structure_address</code>

Figure 5.5: BPF program context data structure

We are passing all data using the c-type `uint64_t`. This represents an unsigned integer with 64 bits (8 bytes) of memory. This size is enough to accommodate any memory address of a 64-bit architecture. We convert to and from pointer data structures using regular c-casting. For example, the unikernels are making sure to properly cast any pointer before passing it to the BPF stack memory as shown in [Figure 5.6](#).

```
1 char* line_ptr = (char*)malloc(DYNAMIC_SIZE * sizeof(char));
2 context->data_address = (uint64_t)(uintptr_t) line_ptr;
```

Figure 5.6: Pointer cast to `uint64_t`

Then, the BPF functions have access to the memory addresses integer representation but not to the host memory itself, where the addresses can be dereferenced. Indirect access to the host memory is provided via BPF helper functions, which accept these addresses of type `uint64_t` and convert them back to normal pointers in the way shown in [Figure 5.7](#).

```
char* line_ptr = (char*)(uintptr_t) context->data_address;
```

Figure 5.7: `uint64_t` cast to a pointer

Once the correct pointer representation is retrieved, the helper functions are able to dereference and manipulate the content inside this address since it belongs to the unikernel address space.

While this approach with pointer casting is not straightforward and requires careful definition of a generic helper function, it is the most efficient one we could think of. It does not contain any copies of the data in any place, but rather manipulates directly the data in the unikernel address space. This is essential when processing big data since performance is a key point of any such computation.

After careful investigation of various data processing use cases, we decided that two address representations are enough to be passed onto the BPF stack as shown in [Figure 5.5](#). `data_structure_address` contains the address to the chained hash table, which holds the transformed dataset in the host unikernel address space. `data_address` points to the arbitrary batch of the dataset requested to be processed by the current BPF stateless function. We also need to pass `data_size` to the context in order to bring information to the helper functions about the size that is dynamically allocated in the address the data points to.

5.6 BPF Helper Functions

Allowing big data communication between the host unikernels and the eBPF VMs would be impossible without the use of BPF helper functions residing in the host address space. Registering custom BPF helpers with uBPF is presented in [Figure 5.8](#)

```
int result = ubpf_register(vm, 22, "bpf_print_text", (void*)bpf_print_text);
```

Figure 5.8: Register uBPF helper

As we can see the function `bpf_print_text` is now registered with alias 22. To be able to use this function when writing BPF C functions, we need to define a Macro as shown in [Figure 5.9](#). Note that the alias 22 is preserved. This is what creates the mapping between the MACRO and the registered uBPF helper.

```
#define bpf_print_text ((void (*)(char *msg, int size))22)
```

Figure 5.9: Macro definition for a uBPF helper

Additionally, each new BPF helper must be defined in the verifier platform. This is

explained in detail in [Section 5.9.1](#).

In our framework, we define three groups of BPF helper functions.

5.6.1 Debugging Helpers

Since the uBPF library does not have an easy way for debugging, we needed to use primitive methods in order to allow printing from the eBPF VM. These methods are defined as helper functions and explained in [Table 4.4](#). Their prototypes are listed in [Table 5.1](#). All helper names from this group are prefixed with `bpf_print_`.

#	Helper Name	Arguments	Return
1	<code>bpf_print_text</code>	<code>const char* msg</code> <code>size_t size</code>	<code>void</code>
2	<code>bpf_print_integer</code>	<code>int64_t number</code>	<code>void</code>
3	<code>bpf_print_hashtable</code>	<code>uint64_t table_address</code>	<code>void</code>

Table 5.1: Debugging Helpers API Prototypes

5.6.2 Hashtable Helpers

This group of BPF helpers acts like an API to read and manipulate the main data structure of the host unikernels. [Table 5.2](#), [Table 5.3](#) and [Table 5.4](#) show details about their prototypes. [Table 4.1](#) and [Table 4.2](#) list descriptions of their functionality. All helper names from this group are prefixed with `bpf_hashtable_`.

#	Helper Name	Arguments	Return
1	<code>bpf_hashtable_get_int</code>	<code>uint64_t table_addr</code> <code>char* key</code>	<code>integer</code>
2	<code>bpf_hashtable_get_map_key_int</code>	<code>uint64_t table_addr</code> <code>char* map_key</code>	<code>integer</code>
3	<code>bpf_hashtable_get_data_key_int</code>	<code>uint64_t table_address</code> <code>uint64_t data_address</code> <code>uint64_t data_start_index</code>	<code>integer</code>
4	<code>bpf_hashtable_get_node_address</code>	<code>uint64_t table_addr</code> <code>char* key</code>	<code>uint64_t</code>
5	<code>bpf_hashtable_get_map_key_node_address</code>	<code>uint64_t table_addr</code> <code>char* map_key</code>	<code>uint64_t</code>

Table 5.2: Hashtable Helpers API Prototypes (1)

#	Helper Name	Arguments	Return
6	bpf_hashtable_get_data_key_node_address	uint64_t table_address uint64_t data_address uint64_t data_start_index	uint64_t
7	bpf_hashtable_insert_int	uint64_t table_addr char* key int value	void
8	bpf_hashtable_insert_map_key_int	uint64_t table_addr char* map_key int value	void
9	bpf_hashtable_insert_data_key_int	uint64_t table_address uint64_t data_address uint64_t data_start_index int value	void
10	bpf_hashtable_insert_value	uint64_t table_addr char* key void* value uint64_t value_size	void
11	bpf_hashtable_insert_map_key_value	uint64_t table_addr char* map_key void* value uint64_t value_size	void
12	bpf_hashtable_insert_data_key_value	uint64_t table_addr uint64_t data_addr uint64_t data_start_index void* value uint64_t value_size	void
13	bpf_hashtable_get_size	uint64_t table_addr	size_t
14	bpf_hashtable_get_first_node	uint64_t table_addr size_t index	uint64_t
15	bpf_hashtable_get_next_node	uint64_t node_addr	uint64_t
16	bpf_hashtable_get_node_value_int	uint64_t node_addr	integer
17	bpf_hashtable_get_node_value_size	uint64_t node_addr	size_t
18	bpf_hashtable_get_node_value	uint64_t node_addr	uint64_t
19	bpf_hashtable_get_node_key	uint64_t node_addr	uint64_t

Table 5.3: Hashtable Helpers API Prototypes (2)

#	Helper Name	Arguments	Return
20	bpf_hashtable_initialize	size_t size	uint64_t
21	bpf_hashtable_free	uint64_t table_address	void
22	bpf_hashtable_get_value_address	uint64_t table_address char* key	uint64_t
23	bpf_hashtable_get_map_key_value_address	uint64_t table_address char* map_key	uint64_t
24	bpf_hashtable_get_data_key_value_address	uint64_t table_address uint64_t data_address uint64_t data_start_index	uint64_t
25	bpf_hashtable_insert_hashtable	uint64_t table_address char* key uint64_t subtable_addr	void
26	bpf_hashtable_insert_map_key_hashtable	uint64_t table_address char* map_key uint64_t subtable_addr	void
27	bpf_hashtable_insert_data_key_hashtable	uint64_t table_address uint64_t data_address uint64_t data_start_index uint64_t subtable_addr	uint64_t

Table 5.4: Hashtable Helpers API Prototypes (3)

5.6.3 Data Helpers

The third group of BPF helper functions provides functionality to read and manipulate the data portion of the dataset, which is going to be processed by the current unikernel. [Table 5.5](#) shows details about their prototypes. [Table 4.3](#) documents their functionality. All helper names from this group are prefixed with `bpf_data_`.

#	Helper Name	Arguments	Return
1	bpf_data_get_next_index_delimiter	uint64_t data_addr uint64_t data_size char* delimiters size_t delimiters_size uint64_t index	uint64_t
2	bpf_data_get_next_index_map_word	uint64_t data_addr uint64_t data_size uint64_t table_addr char* map_key uint64_t index	uint64_t
3	bpf_data_update_index_content	uint64_t data_size uint64_t index char value uint64_t data_addr	void

Table 5.5: Data Helpers API Prototypes

5.7 uBPF Function Implementation and Compilation

All predefined helper functions listed in [Section 5.6](#) in combination with the BPF program context defined in [Section 5.5](#) can be utilized for a custom definition of BPF functions. Each BPF helper has to be called `bpf_prog`, because the verifier will look for this name to use as an entry point. [Figure 5.11](#) shows a simple example function. It uses the debugging helper `bpf_print_integer()` to output the `data_size` integer from the program context to the unikernel's standard I/O.

```
1 #include "bpf_helpers.h"
2
3 struct Context {
4     uint64_t data_address;
5     uint64_t data_size;
6     uint64_t data_structure_address;
7 };
8
9 uint64_t bpf_prog(struct Context *arg) {
10     bpf_print_integer(arg->data_size);
11     return 1;
12 }
```

Figure 5.10: Simple BPF function

Once written, the BPF function has to be compiled into bytecode. This can be done by using the clang compiler. Figure 5.11 shows the compilation steps.

```
clang -c -target bpf -fno-stack-protector -o build/bpf_prog.o bpf_prog.c
objcopy -I elf64-little -O binary --only-section=.text \
    build/bpf_prog.o build/bpf_prog.bin
```

Figure 5.11: BPF function compilation

First, clang compiles the code into ELF 64-bit little-endian format. Then, the objcopy command extracts only the executable instructions. The resulting bytecode is used by unikernel's eBPF VMs and the verifier.

5.8 Unikernel Configuration

Each unikraft project contains a `.config` file in the root of its directory. This file is used to configure unikernel drivers and determine which libraries have to be included in the compiled image. It also provides the opportunity to define application options, which can be used for further customization of the resulting image. All configurations can be viewed and edited by running the command `make menuconfig`.

Our proposed framework has two important application options that are used to compile three different images.

1. `CONFIG_APPMAP_WITH_UBPF` - Run the application with uBPF core functionality
2. `CONFIG_APPMAP_WITH_UBPF_JIT` - Run the application with uBPF core functionality and JIT compilation

5.8.1 NonBPF unikernel

When none of the above-mentioned options are used, the compiled unikraft image will not use any uBPF functionality. It will fall to compiling statically pre-defined map and reduce functions, which will not be modifiable at runtime. This image is used as a baseline for the evaluation we perform in [Section 6](#).

5.8.2 Unikernel image with BPF functionality

When only `CONFIG_APPMAP_WITH_UBPF` is used in the configuration, the resulting image will have uBPF hooks at the two places where the map and the reduce core functions are called in the framework we implemented. These hooks are used as locations to dynamically inject the requested verified BPF bytecode. By using this image, the user will be able to change the functionality of the core functions without recompiling or restarting the unikernel.

This is possible because the uBPF VMs are created before the dataset processing starts. Upon creation, uBPF reads the verified BPF bytecode into memory and then directly interprets it in order to load it inside the VM. After it is loaded, the VM can be used multiple time to be applied on different context memory regions.

5.8.3 Unikernel image with BPF JIT functionality

When both `CONFIG_APPMAP_WITH_UBPF` and `CONFIG_APPMAP_WITH_UBPF_JIT` are used in the configuration, the resulting image will again have the uBPF dynamic functionality, but this time the loading of the bytecode will use the built-in JIT compilation functionality. The advantage of this approach is that it will minimize the overhead caused by the standard interpretation of the bytecode. This leads to a faster load time of the uBPF VM. JIT-compiled BPF bytecode is also equipped with further platform-specific optimizations that might decrease the execution time of the functions themselves.

5.9 Code Verification

As described in [Section 4.4](#), we are implementing our verification mechanism as an external software that will use cryptographic signatures in order to securely and reliably

verify whether a corresponding bytecode is valid or not.

We integrate PREVAIL ebpf-verifier [30] into our solution. In order to correctly configure this verifier, we first need to compile PREVAIL and statically link its built files `liblibbtf.a` and `libebpfverifier.a` to our verification solution. Since the library is written in C++, we use the same language for simplicity and define a custom platform in order to make PREVAIL compatible with our unikernel images.

5.9.1 PREVAIL Platform

The PREVAIL verifier provides an API, which can be used to write a platform configuration. This configuration serves as a bridge between the verifier and the intended runtime environment by providing an execution context and system-specific configurations.

In order to create our own platform configuration, we used as an example the already implemented Linux platform provided in the repository of the project [30].

One of the important specifications is a descriptor for our stack memory. It is shown in [Figure 5.12](#).

```
1 constexpr ebpf_context_descriptor_t bpf_prog_descr = {  
2     .size = 256,  
3     .data = 0,  
4     .end = 255,  
5     .meta = -1  
6 };
```

Figure 5.12: BPF Context Descriptor

Our context object is valid only if it is less than 256 bytes. The `.data` and the `.end` fields of the descriptor provide offsets to the beginning and the end of the stack data structure. We set them to 0 and 255, respectively, which means we allocate all 256 bytes for the data structure passed via the stack memory (our Context struct from [Figure 5.5](#)). We are not using any metadata fields (unlike linux). Therefore, we set `.meta = -1`.

Another important platform specification part is related to the custom BPF helper functions. PREVAIL requires prototypes for all helpers that our platform is allowed to verify successfully. An example of such a prototype is shown in [Figure 5.13](#).

```

1 static const struct EbpfHelperPrototype bpf_print_text_proto = {
2     .name = "print_text",
3     .return_type = EBPF_RETURN_TYPE_INTEGER_OR_NO_RETURN_IF_SUCCEED,
4     .argument_type = {
5         EBPF_ARGUMENT_TYPE_PTR_TO_READABLE_MEM,
6         EBPF_ARGUMENT_TYPE_CONST_SIZE
7     },
8 };

```

Figure 5.13: eBPF Helper Prototype in PREVAIL

This is one of the debugging helper functions we presented in [Section 5.6.2](#). The prototype specifies the the name, the return type and the arguments of the helper. The return type can be one of the possibilities listed in [Table 5.6](#).

_ebpf_return_type	Description
EBPF_RETURN_TYPE_INTEGER	Must always return an integer
EBPF_RETURN_TYPE_PTR_TO_MAP_VALUE_OR_NULL	Returns ptr to BPF map value or NULL
EBPF_RETURN_TYPE_INTEGER_OR_NO_RETURN_IF_SUCCEED	Returns void or an error code
EBPF_RETURN_TYPE_UNSUPPORTED	Returns an unsupported type

Table 5.6: PREVAIL return types

We have selected `EBPF_RETURN_TYPE_INTEGER_OR_NO_RETURN_IF_SUCCEED` since the printing function does not really return anything. The `.argument_type` field is a collection of allowed argument types. A breakdown of each type is listed in [Table 5.7](#). These argument types define the constraints on what values or memory regions can be passed as parameters to ensure the safety and correctness of eBPF programs. For our function, we select `EBPF_ARGUMENT_TYPE_PTR_TO_READABLE_MEM` for the `msg` argument, which means that the verifier will allow only initialized memory. For the second argument `size`, we select `EBPF_ARGUMENT_TYPE_CONST_SIZE` in order to instruct the verifier to apply a 64-bit restriction and to disallow the value 0, since it does not make sense to allow printing of 0 bytes.

<code>_ebpf_argument_type</code>	Description
<code>EBPF_ARGUMENT_TYPE_DONTCARE</code>	Argument type is ignored
<code>EBPF_ARGUMENT_TYPE_ANYTHING</code>	Any 64-bit argument type is valid
<code>EBPF_ARGUMENT_TYPE_CONST_SIZE</code>	Constant size argument
<code>EBPF_ARGUMENT_TYPE_CONST_SIZE_OR_ZERO</code>	Constant size argument or 0
<code>EBPF_ARGUMENT_TYPE_PTR_TO_CTX</code>	Pointer to a BPF context structure
<code>EBPF_ARGUMENT_TYPE_PTR_TO_MAP</code>	Pointer to a map object
<code>EBPF_ARGUMENT_TYPE_PTR_TO_MAP_OF_PROGRAMS</code>	Pointer to a map of program objects
<code>EBPF_ARGUMENT_TYPE_PTR_TO_MAP_KEY</code>	Pointer to the key of a map entry
<code>EBPF_ARGUMENT_TYPE_PTR_TO_MAP_VALUE</code>	Pointer to the value of a map entry
<code>EBPF_ARGUMENT_TYPE_PTR_TO_READABLE_MEM</code>	Pointer to initialized readable memory
<code>EBPF_ARGUMENT_TYPE_PTR_TO_READABLE_MEM_OR_NULL</code>	Pointer to initialized readable memory or NULL
<code>EBPF_ARGUMENT_TYPE_PTR_TO_WRITABLE_MEM</code>	Pointer to writable memory
<code>EBPF_ARGUMENT_TYPE_UNSUPPORTED</code>	Argument type is not supported by the verifier

Table 5.7: PREVAIL argument types

Analogously, we write prototypes for all predefined helper functions. After the prototype definitions, we have to ensure their mapping to `int32_t` from their `ubpf` registration is also preserved. These are defined in the prototype functions

- `bool is_helper_usable_ukmapreduce(int32_t n)`
- `EbpfHelperPrototype get_helper_prototype_ukmapreduce(int32_t n)`

5.9.2 PREVAIL Usage

After a valid platform is specified, we can already use the PREVAIL statically compiled verifier as a linked library to our custom `ubpf-verifier` in the following way

- We load the bytecode of the BPF program we verify
- Parse the bytecode using the PREVAIL `read_elf()` function, which will give us a list of all functions.
- Find the function with name `bpf_prog` from the list of parsed functions
- Unmarshall the found program into an acceptable type by the verifier (PREVAIL provides `unmarshall()` function)
- Call `ebpf_verify_program()` function, which performs the actual PREVAIL verification according to the specified platform

5.9.3 Secure Verification

Having verified the eBPF bytecode, we need to ensure that the externally verified outcome is securely propagated to the unikernels without the possibility of malicious tampering with the result. To achieve this, we decided to rely on a hash value and the RSA public-key cryptographic algorithm.

If the bytecode verification fails, we log the result and do nothing else. However, in case it is successful, we proceed with signing the code in the following way

1. We use OpenSSL to create a SHA256 hash value from the bytecode of the verified program
2. Sign the hash using the private RSA key
3. Upload the signed hash to the same shared directory as the bytecode.

Whenever a unikernel wants to load a bytecode in its uBPF VM, it looks for a signature. If it cannot find any, no successful verification is run and the bytecode is rejected. In case there is a signature, it proceeds to verify it by using the public RSA key. Upon successful signature verification, it loads the bytecode and proceeds with its execution. Otherwise, the function is rejected and an error is propagated.

6 Evaluation

This section outlines the evaluation of our proposed framework. It is divided into two parts - qualitative analysis and quantitative measurements. The qualitative analysis addresses aspects of usability and development experience. The quantitative measurements present different experiments with various use cases and configurations, which compare initialization time, memory footprint and performance. Additionally, we provide a benchmark analysis against a traditional Linux VM implementation. At the end, we also extend the framework with more use cases.

6.1 Qualitative Analysis

6.1.1 Development Complexity

Developing serverless BPF functions for unikernels introduces challenges due to the restricted uBPF environment and the verifier constraints. Unlike Linux eBPF, uBPF does not come with standard headers and helpers, which means the developer needs to work with a custom-defined API. The stack memory region shared between the BPF VM and the host unikernel is very limited and does not support any dynamically allocated structures. Moreover, the eBPF verifier applies strict checks on program structure, including limitations on program size, types, complexity, and data access types. These limitations increase the difficulty of implementing data processing functions like `map()` or `reduce()`, especially as traditional error messages and logs are often inaccessible in unikernels. Adjusting the implementation logic of a BPF function to pass the verification requires a deep understanding of BPF internals, such as its instruction set, as well as knowledge about the way its address space works. This increases the initial learning curve for developers unfamiliar with the technology.

6.1.2 Debugging

Debugging BPF programs within unikernels is limited, as standard tools such as `gdb` or `strace` are unavailable. Our uBPF integration provides minimal debugging support by using `print` redirects through helper functions. This makes it difficult to isolate issues or verify program logic during runtime. For instance, diagnosing verifier failures,

understanding memory access issues, and ensuring proper function execution within the unikernel forced us to use very basic debugging practices, which significantly increased the development time.

6.1.3 Learning curve

Writing eBPF functions

Writing eBPF functions for our data processing tasks involves understanding the framework we implemented, low-level BPF programming requirements, as well as complicated error messages coming from the verifier. The developer needs to be familiar and comfortable with the predefined BPF context and all helper functions provided by the platform. Since the verifier performs strict checks on data access patterns and stack usages, developing is also restricted to the limited instruction set and legal memory accesses. The functions must be designed to process big data batches only by passing addresses around using stack memory and helper functions. Understanding all these complexities requires time and extra design considerations.

Unikraft

We anticipate that users of our framework will rarely need to modify the unikernels directly. However, should such a need arise, it's important to note that Unikraft has a steep learning curve, particularly for developers without prior experience in operating systems or low-level programming. Configuring and building a unikernel for a specific application also requires familiarity with tools like `make` and `Kconfig` in order to properly include all needed libraries. More challenges could arise from the lack of proper documentation. Most of the unikraft documentation that can be currently found is either very limited or outdated. This might force developers to read the source code, which could be time-consuming and frustrating.

6.1.4 Code Maintenance

The ability to dynamically inject eBPF functions within unikernels allows us to update data processing logic on the fly. However, as more functions are added or updated, maintaining code modularity and managing dependencies becomes more complex. Keeping each updated function successfully verified by the verifier also adds to the maintenance complexity. We observed that frequent adjustments are required to ensure that new functions pass verification and work seamlessly with the existing unikernel logic.

6.2 Quantitative Evaluation

Our quantitative evaluation focuses on key performance metrics for different configurations of our framework. We chose a simple use case of the MapReduce algorithm in order to perform a comprehensive evaluation and benchmark the different solutions.

All measurements in this section are collected from experiments performed on a physical machine with the specifications listed in [Table 6.1](#).

CPU	Intel Xeon Gold 5317 (12) @ 3.000GHz, SMT disabled
GPU	ASPEED Technology, Inc. ASPEED Graphics Family
Memory	257231MiB @ 2933 MT/s
Disk	2x Samsung MZ1L21T9 with 1.75 TiB
OS	NixOS 24.05.20241018.07f7c53 (Uakari) x86_64
Kernel	Linux 6.6.56
QEMU	qemu-system-x86_64 emulator version 8.2.6

Table 6.1: Machine Specifications

Additionally, all QEMU processes are given only one CPU core since we do not have parallelism in the current version of the framework.

6.2.1 Unique Word Count

The simple case we chose for our data processing framework is Unique Word Count. There are two BPF functions defined for it. The map function performs a count of unique words and then the reduce returns the size of the map data structure. The dataset we chose for our experiments is [Enron Email Dataset](#) [38].

We performed our evaluation on three different configurations:

1. **MapReduce in unikernels without eBPF**

This baseline configuration uses statically compiled Map and Reduce functions within unikernels without the dynamic flexibility of eBPF. We refer to it as NonBPF configuration later in the section.

2. **MapReduce in unikernels with eBPF**

The second configuration still uses the same unikernels, but this time, each of them is equipped with an eBPF virtual machine, which is able to dynamically

inject the core functionalities of the mapping and the reduction phase of the algorithm at runtime.

3. MapReduce in unikernels with JIT-compiled eBPF

While the previously mentioned configuration uses a classic interpreting of BPF functions, this one is taking advantage of the JIT compiler, explained in detail in [Section 5.8.3](#).

With the following experiments, we are evaluating how the integration of eBPF and JIT affects the cold and warm initialization time, the overall computation performance, and the memory footprint.

Cold Initialization Time

Cold Initialization measures the time required to start the software from a completely uninitialized state until it is ready to start processing data. Initialization time should indicate both unikernel efficiency and the overhead introduced by eBPF and JIT mechanisms. In order to correctly measure all of these, we added a log just before the serverless functions are called in the Mapper and the Reducer unikernels. This means that our measurements include the I/O access to the shared file system for the dataset, the setting up of the uBPF VMs and the loading of the serverless functions into them. We hypothesize that eBPF will introduce a slight overhead and that its JIT-compiled version will be a bit slower due to its additional compilation function `ubpf_compile()`.

The experiment was run 10 times, and the results for the unikernels are summarized in [Figure 6.1](#). As it can be seen, the uBPF VM did not introduce significant overhead, only around 2ms. The JIT-compiled BPF introduced approximately the same with a difference to the non-JIT version of 0.1ms. The standard deviations are also approximately the same across the three different configurations.

Looking carefully at the numbers, we see that the Reducer unikernel consistently adds around 1-1.5ms overhead to all configurations. This happens due to the deserialization of data from the intermediate data store to the communication data structure. The deserialization process is before the uBPF VM load, so it gets included in our measurements.

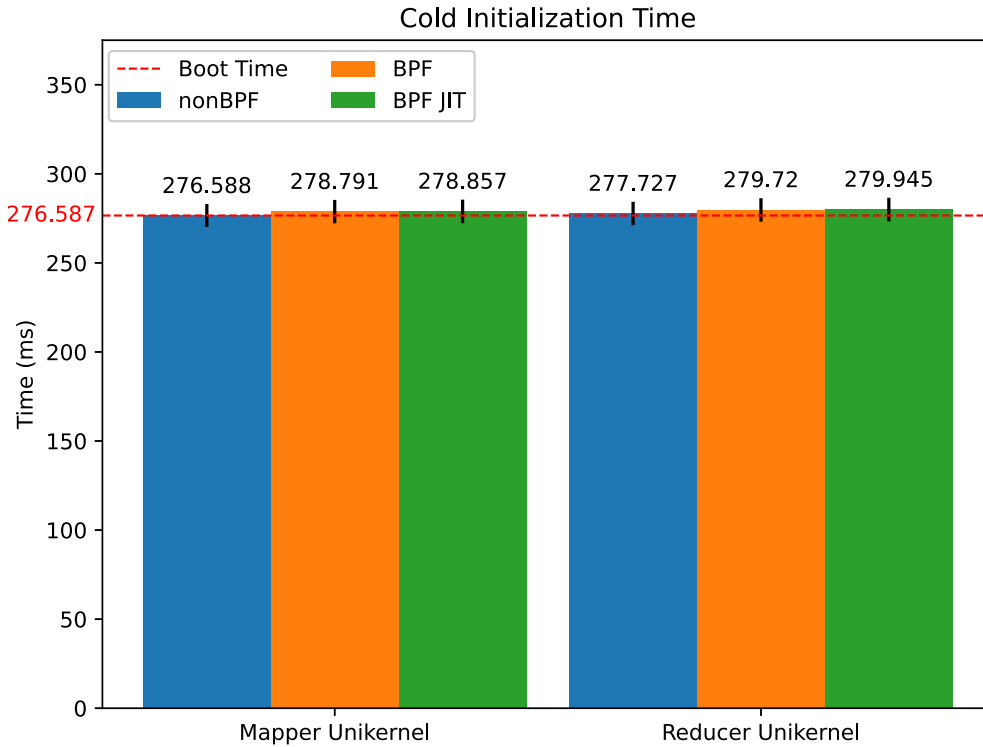


Figure 6.1: Cold initialization time

Most of the cold initialization is attributed to the boot time of the different unikernels. The average time we measured only for the booting is 276.5ms. If we subtract this, we are left with very small differences between the configurations, which are all within the range of the standard deviations. This is why we can conclude that uBPF does not affect cold initialization time when integrated with unikernels.

Warm Initialization Time

Warm Initialization time is the duration needed to start the software when some resources or context have been retained from a previous run. In our experiment, this is the time taken to reconfigure a data processing task before the dataset is transformed. In configurations with eBPF, this metric is crucial for evaluating how quickly new functions can be loaded. We expect eBPF-based configurations to be much faster than the baseline, as they eliminate the need for static recompilation and reboot.

We collect measurements again from 10 runs and outline our results from the unikernels in Figure 6.2. The results clearly show how much performance gain we get from using BPF functions. The improvement mainly comes from the lack of boot time in the BPF configurations since they are able to dynamically load the new bytecode without restart.

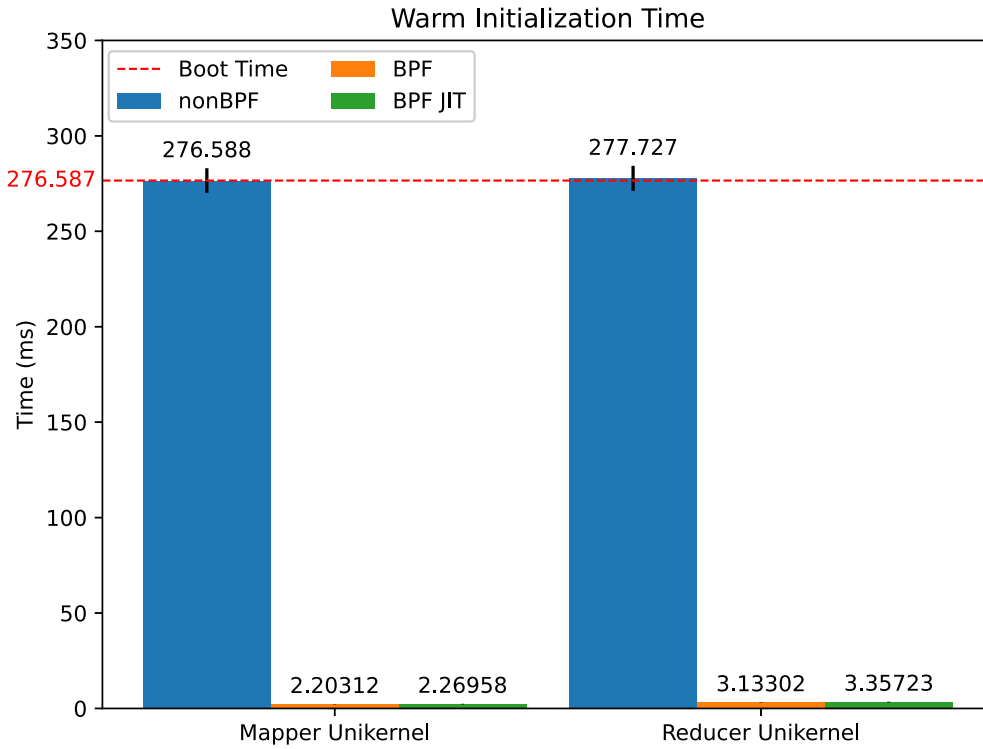


Figure 6.2: Warm initialization time

On average, the uBPF configurations decrease the warm initialization by **276ms**.

Performance (Execution Time)

We measure the total processing time needed to complete the MapReduce execution model in each of the [three configurations](#). Execution time will indicate whether the flexibility of eBPF (especially with JIT) provides performance benefits or whether it adds overhead compared to statically compiled MapReduce core functions. This metric helps us understand the real-world trade-offs of using dynamic code injection in performance-critical applications.

Additionally, we extend our evaluation to compare the performance of our unikernel data processing configurations with a traditional Linux VM. We preserve the same data structure, code base and file communication flow in the Linux VM in order to compare as fairly as possible. Such evaluation allows us to understand how unikernels perform in general against more conventional and widespread approaches in terms of both resource usage and processing efficiency.

Experiments with three different dataset sizes were conducted.

1. Small Dataset (25KB)

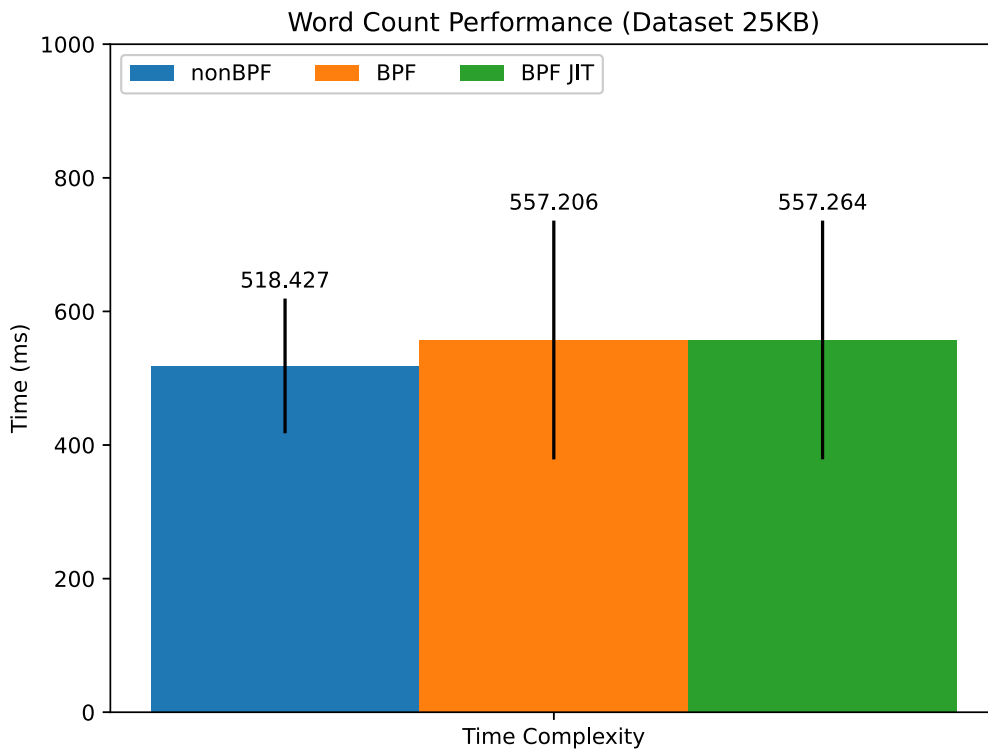


Figure 6.3: Word Count performance (25KB dataset)

Our results from [Figure 6.3](#) show that the baseline configuration without BPF injections performs slightly better when processing very small datasets (25KB). This is because the limited number of BPF reconfigurations and executions does not result in sufficient saved boot time to compensate for the overhead introduced by the bytecode loading. The JIT compiled BPF bytecode performance is approximately the same as the interpreted, meaning that there is no significant benefit to

using JIT with very small data and few BPF executions.

Additionally, we ran the same small dataset on a Linux VM. The average execution time for 10 runs was 7.78 seconds. This is significantly slower than any of the unikernel approaches from [Figure 6.3](#). This big difference comes from the amount of time Linux needs in order to boot its general-purpose OS.

2. Medium-sized Dataset (1MB)

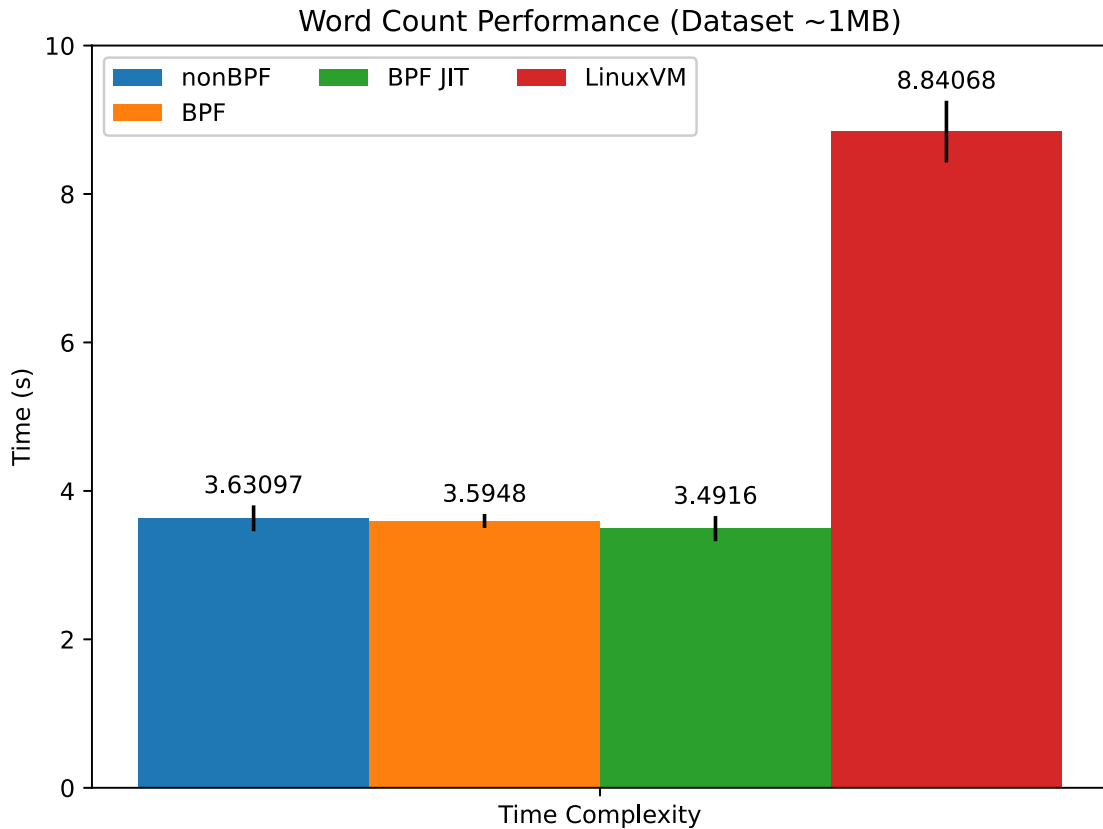


Figure 6.4: Word Count performance (1MB dataset)

With a 1MB dataset, we already observe a slight advantage of using BPF injections. The saved boot time makes up for the BPF overhead and boosts the performance of the BPF functions. We can also see a small advantage of using JIT-compiled bytecode compared to the interpreted BPF code. All results are illustrated in [Figure 6.4](#).

We also conducted a Linux VM experiment with this dataset. The result of the VM was again very slow (more than 2x) due to the boot time, as shown in [Figure 6.4](#).

3. Bigger Dataset (5MB)

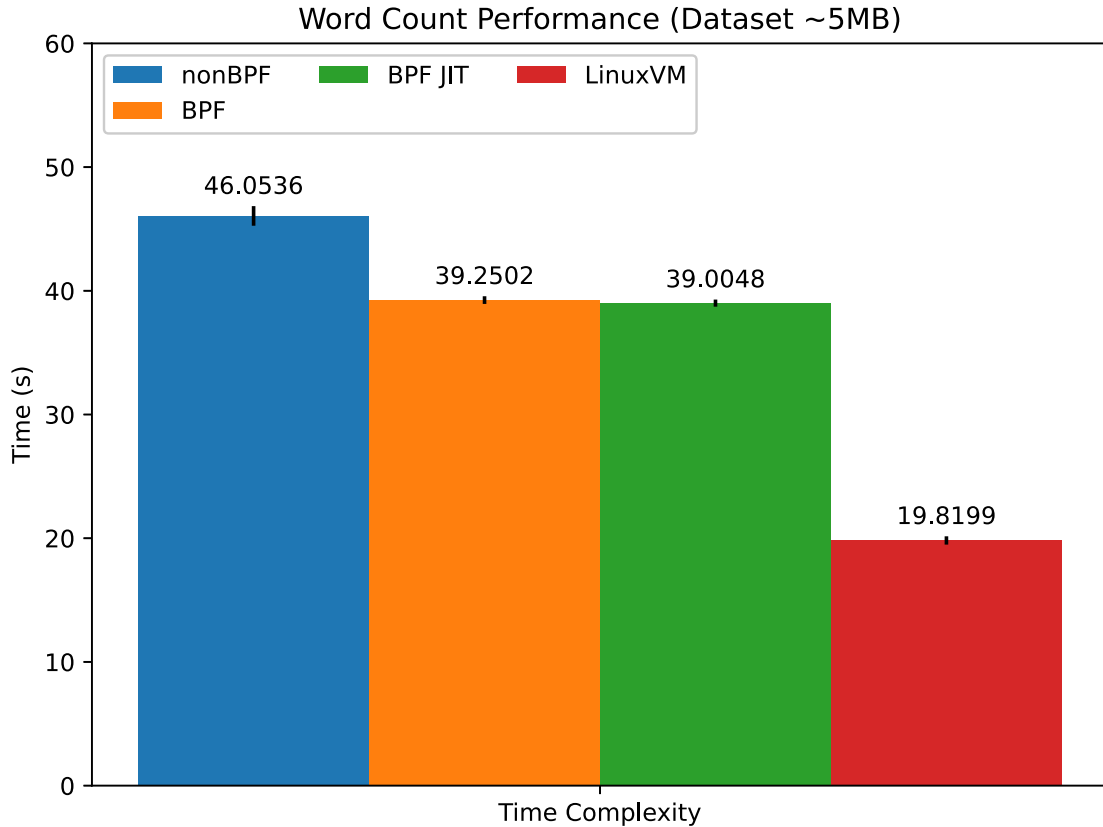


Figure 6.5: Word Count performance (5MB dataset)

It can be seen from [Figure 6.5](#) that once we increase the size of the dataset, the uBPF VMs are getting more and more performant compared to the unikernel baseline without BPF. This means that for much bigger datasets, we will observe a proportional performance benefit from the dynamic code injections.

However, during the Linux VM experiment with a 5MB dataset, we discovered that it executes our data processing framework implementation two times faster than

the Unikraft unikernel. The reason for this is likely attributed to Linux mature optimizations, including advanced CPU scheduling, efficient memory management, and a highly optimized I/O stack. Unikraft, while minimal and lightweight, may lack these optimization techniques, leading to under-utilization of the provided CPU resources.

Memory Footprint

We also measured the memory footprint of each unikernel configuration and the Linux VM. This metric is important for understanding resource efficiency, especially since unikernels are intended to be lightweight. The addition of eBPF or JIT compilation may increase memory usage, but we aim to quantify the exact impact in order to determine if it remains within acceptable limits for high-performance and resource-constraint environments.

1. Small Dataset (25KB)

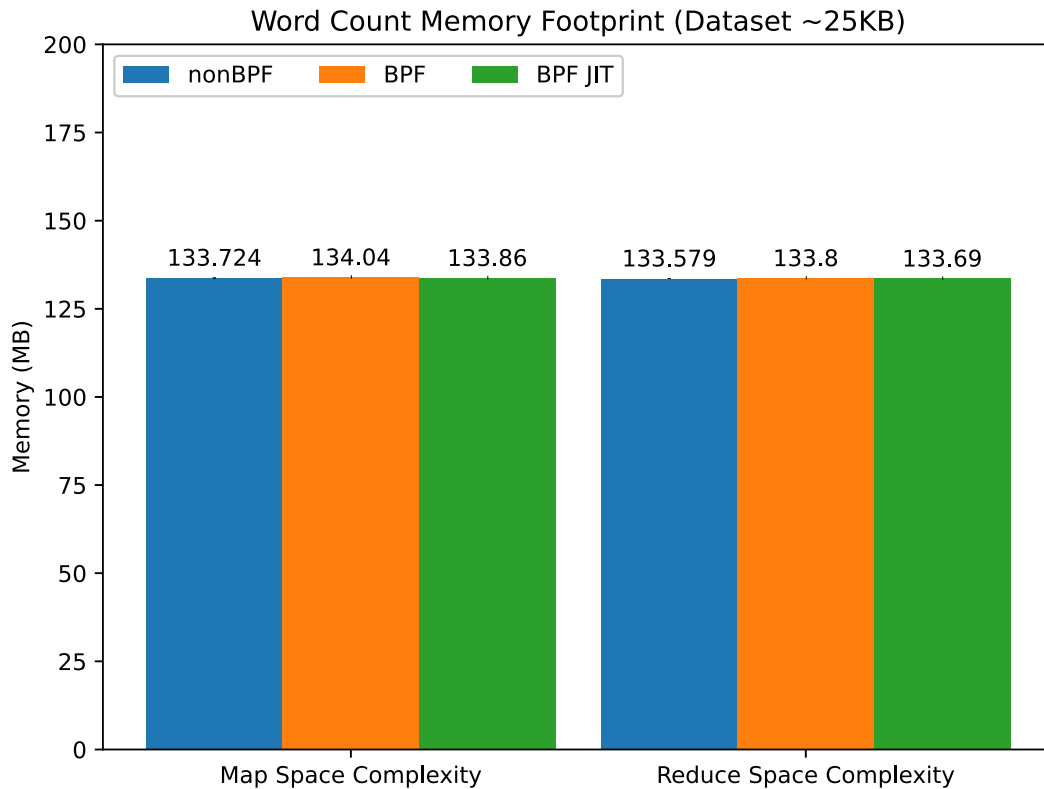


Figure 6.6: Word Count memory footprint (25KB dataset)

As demonstrated in Figure 6.6, the unikernels maintain a consistent memory consumption when processing small datasets. Even if we run a dummy hello world example, the unikernels reserve approximately 130MB of memory, which means this is inherent to Unikraft, regardless of the computational workload.

The Linux VM also has an inherent reservation of approximately 1447.218MB, regardless of the computational workload. This shows us the efficiency we can achieve when our framework does not require a general-purpose OS.

2. Medium-sized Dataset (1MB)

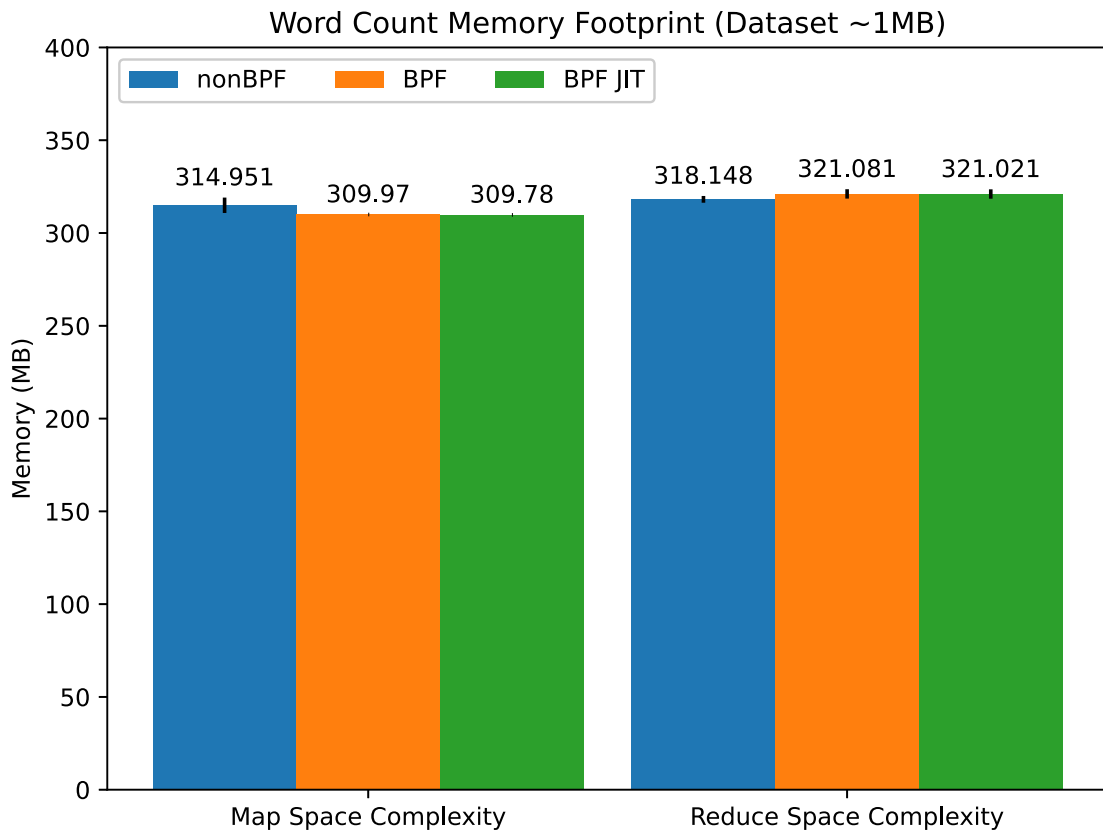


Figure 6.7: Word Count memory footprint (1MB dataset)

Since the dataset is now bigger, the unikernels need to also occupy more resources to fulfill their use case. However, all configurations increased evenly as it can be seen from Figure 6.7. Therefore, we can say that the BPF integration had no effect

on the memory footprint.

The Linux VM computation resulted in approximately the same 1446.732MB memory allocated, which means it uses some already allocated memory by the VM.

3. Bigger Dataset (5MB)

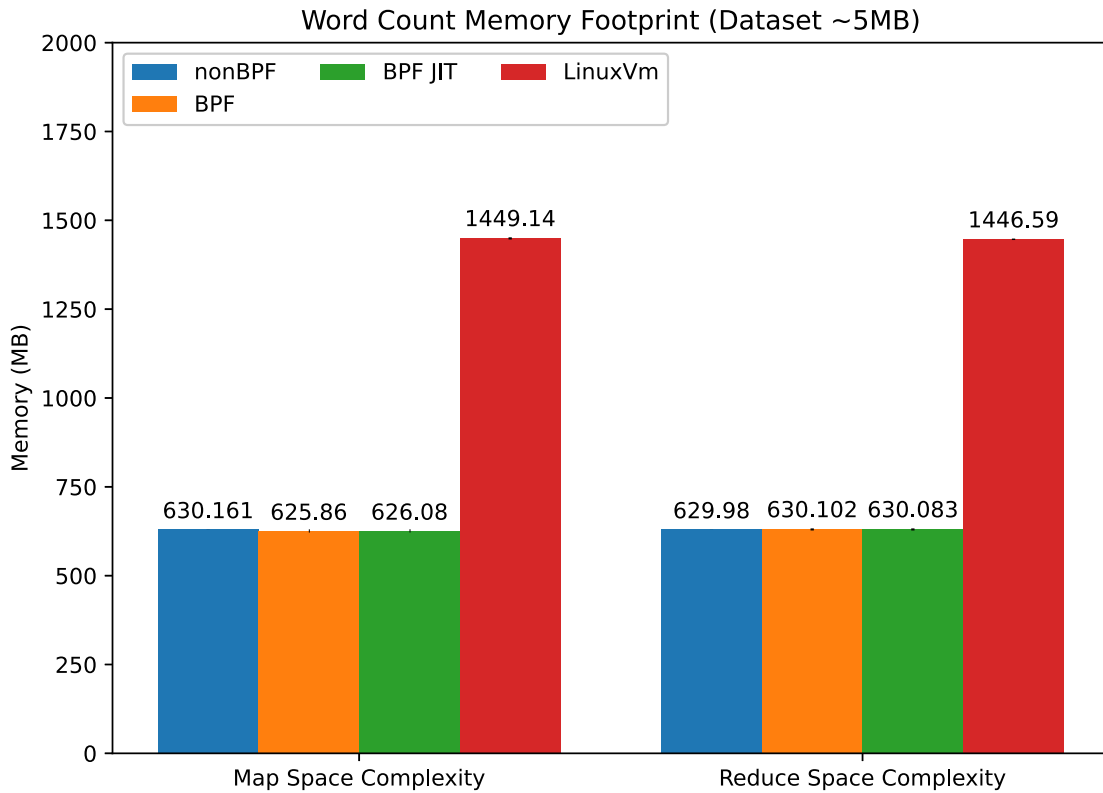


Figure 6.8: Word Count memory footprint (5MB dataset)

With the increase of the dataset to 5MB, our unikernels' memory consumption increased as well as illustrated in Figure 6.8. This is normal because we load all the data in memory without implementing any optimizations. More importantly, using uBPF VMs for dynamic injection of the data processing bytecode does not negatively affect the space complexity, independent of the input data.

In the Linux VM, we continued having 1450MB reserved, which is more than two

times the memory needed by any of our unikernel configurations. It is important to note that general-purpose VMs allocate a lot of memory to the OS they run. This is inefficient since resources are acquired without actually being used. Unikraft, on the other hand, consumes memory much more efficiently. The reason for this is the lack of OS, resulting in a minimalist image that bundles only the necessary libraries and drivers for its single-purpose functionality.

6.2.2 CPU Intensive BPF functions

We ran an experiment with CPU-intensive BPF functions to explore the potential of implementing resource-heavy use cases in our serverless data processing framework. Implementing such use cases is not the focus of this thesis. However, we want to show that this is still possible. Many popular data processing applications used for scientific computing, image processing or machine learning rely very heavily on CPU-intensive operations. We ran simple but computationally expensive tasks to simulate this without the usage of helper functions or the BPF context. This helped us understand how each unikernel configuration performs and handles resource utilization under heavy computational loads.

In our experiment, we used a 5MB dataset to trigger the BPF CPU-intensive functions. The function used instead of the regular map functionality is the calculation of Fibonacci numbers shown in [Figure 6.9](#). The value we chose for N is 50000.

```
1 uint64_t bpf_prog(struct Context *ctx) {  
2     volatile uint64_t prev = 0, curr = 1, next = 0;  
3  
4     for (uint64_t i = 2; i <= N; i++) {  
5         next = prev + curr;  
6         prev = curr;  
7         curr = next;  
8     }  
9  
10    return curr;  
11 }
```

Figure 6.9: Fibonacci number calculation

The reduce function was replaced by a CPU-intensive operation of calculating π using the Leibniz Series shown in [Figure 6.10](#). With this calculation, we encountered the BPF's

inability to handle floating point numbers. To solve this, we applied a scaling factor for fixed-point arithmetic, which allowed us to work only with integers. The values we used for the C Macros are $ITERATIONS = 1000000000$ and $SCALE = 100000000$.

```
1 uint64_t bpf_prog(struct Context *arg) {
2     int64_t pi_scaled = 0;
3     for (int i = 0; i < ITERATIONS; i++) {
4         int64_t term_scaled = SCALE / (2 * i + 1);
5         if (i % 2 == 0) {
6             pi_scaled += term_scaled;
7         } else {
8             pi_scaled -= term_scaled;
9         }
10    }
11    volatile int64_t result_scaled = pi_scaled * 4;
12    return (uint64_t) result_scaled;
13 }
```

Figure 6.10: π calculation using the Leibniz Series

We used the `c` keyword `volatile` in both CPU-intensive functions in order to instruct the compiler not to apply any optimizations and always execute the code involving these variables. This allows for a fair comparison.

The performance we achieved for our three unikernel configurations is shown in [Figure 6.11](#). The execution times we measured show that the BPF JIT configuration performs as well as the nonBPF unikernel. It even provides a slight improvement coming from the saved boot time at each reconfiguration. However, we observed a significant delay in the BPF VM that does not use JIT compilation. It finished three times slower than the JIT-compiled code. This overhead is introduced by the way the code is interpreted instead of being directly translated into native machine code. Each instruction is separately read from memory, then decoded and finally executed through a series of conditional checks and function calls. Due to this complicated process of interpretation, we can conclude that non-JIT compiled BPF bytecode is not appropriate for CPU-intensive functions.

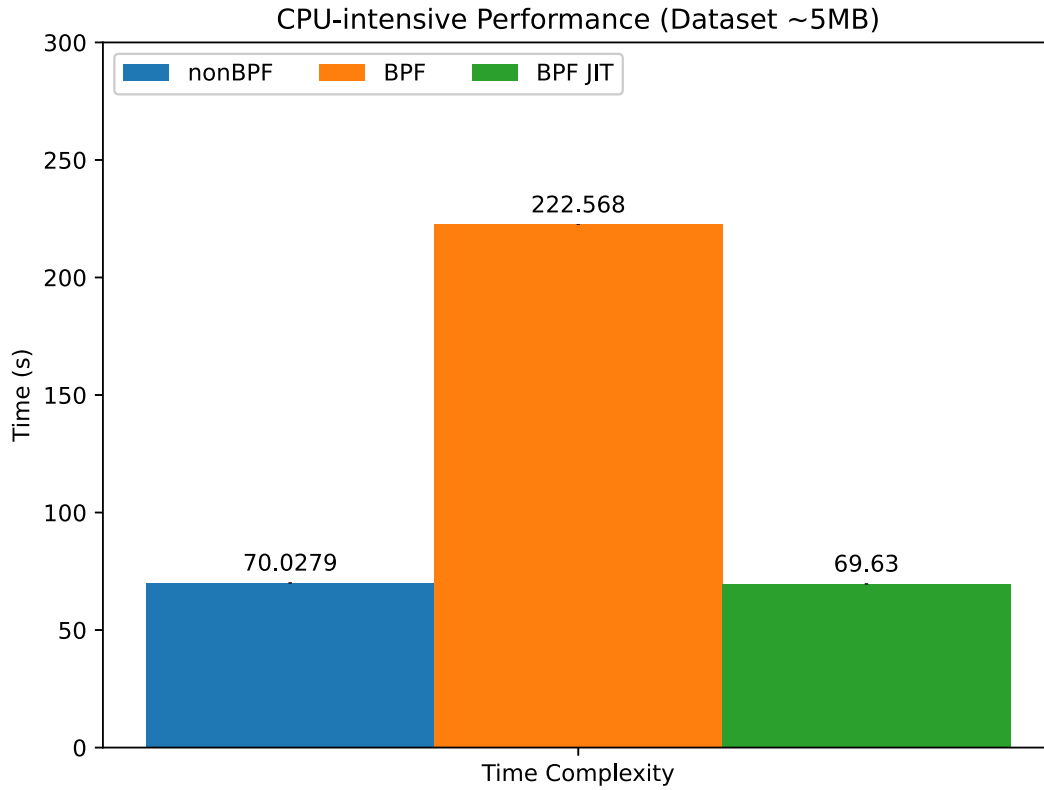


Figure 6.11: CPU-intensive functions Performance

The memory footprint of all configurations' computations did not change, which is what we expected since we are not using any non-primitive dynamic data structures. The memory consumption results across all configurations are shown in [Table 6.2](#).

	Average across 10 runs
Mapper Unikernel	133.877MB
Reducer Unikernel	133.855MB

Table 6.2: Memory Footprint with CPU-intensive functions

6.2.3 Other use cases

Finally, we expand our evaluation by applying, implementing, and evaluating BPF functions for other use cases beyond the unique word count used until now. Testing with more MapReduce scenarios helps us validate the framework's adaptability. It provides

a broader perspective on performance characteristics and gives us the opportunity to determine how flexible the BPF Helpers API is.

Search Engine

Very popular data processing operations are filtering and reduction. To utilize them for a use case, we decided to implement a small primitive search engine. We would still use the dataset from the Word Count problem, but will define new map and reduce BPF functions.

1. Mapper

The dataset consists of many files with emails and text inside. The idea of the map function is to filter and organize the files by matches of a searched word. Two additional arguments are needed inside the BPF function - the current document name and the searched word. In order to avoid modifying the Context from [Figure 5.5](#), we can propagate as much data as we want through the shared communication data structure. By utilizing the BPF helper functions, we avoid memory copy or transfer between the host and the BPF VM. The control is exercised entirely through addresses and static memory. The whole implementation of the Map function is available in [Figure 6.12](#).

```
1 uint64_t bpf_prog(struct Context *arg) {
2     char MAP_SEARCH_KEY[] = "MR_Search_Key";
3     char MAP_DOC_ID_KEY[] = "MR_Document_Id";
4     for(uint64_t i = 0; i < arg->data_size; i++) {
5         uint64_t token_start = i;
6         uint64_t match_idx = bpf_data_get_next_index_map_word(
7             arg->data_address, arg->data_size,
8             arg->data_structure_address, MAP_SEARCH_KEY, i);
9         if(match_idx == arg->data_size) { break; } // no match
10        int value = bpf_hashtable_get_map_key_int(
11            arg->data_structure_address, MAP_DOC_ID_KEY);
12        bpf_hashtable_insert_map_key_int(
13            arg->data_structure_address, MAP_DOC_ID_KEY, ++value);
14        i = match_idx;
15    } return 1;
16 }
```

Figure 6.12: Map Function for a Search Engine use case

MAP_SEARCH_KEY and MAP_DOC_ID_KEY are our predefined and statically compiled hashtable keys pointing to the dynamic data. The user can always change the search word by changing the shared file `search.in`. The documents are always changing dynamically as the traversal continues.

Each line of data is processed separately, and any match with the searched word is persisted to the intermediary data store using the key-value form

(document_id, occurrences)

2. Reducer

The reduce function for this use case is trivial. It simply traverses the intermediary store and returns all document_id values ordered by their corresponding number of occurrences.

This MapReduce use case is quite efficient since we do not copy and keep all the data. We only preserve the data the user searches for. This is also reflected in the evaluation plot, where we can see that the performance is much faster in comparison with the word count use case. Execution time from 10 searches for the word Allen (first name in the [Enron Email Dataset](#)) is shown in [Figure 6.13](#).

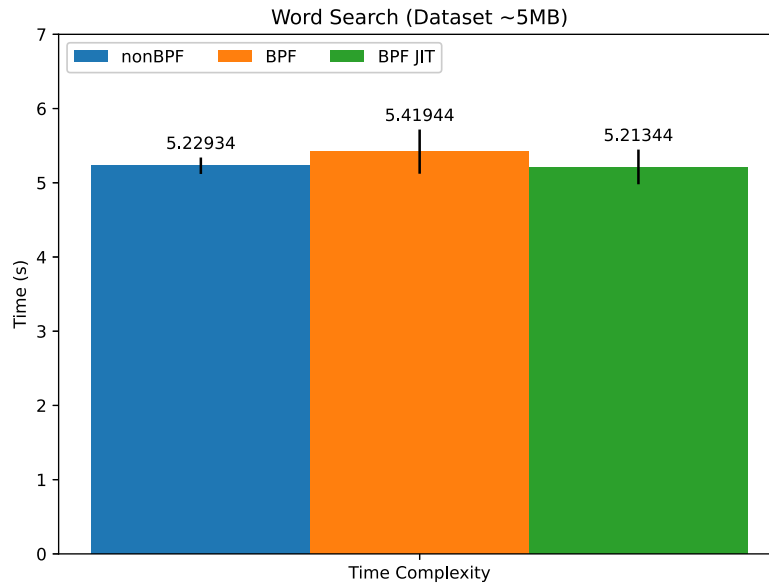


Figure 6.13: Search use case performance

Here again, we see that the JIT BPF and nonBPF configurations are faster than the

BPF interpreted bytecode. The memory footprint was preserved across all unikernels with averages shown in [Table 6.3](#).

	Average across 10 runs
Mapper Unikernel	149.877MB
Reducer Unikernel	153.786MB

Table 6.3: Search use case Memory Footprint

The search use case can be applied to analyzing big data where we must filter for words of interest and compute the repetition frequencies. Such an example in industry is Log Analysis.

Inverted index

Finally, we would like to present a bit more complicated use case. An Inverted Index is a data structure used in search engines like Google to map words (or terms) to the documents/webpages in which they appear.

The idea is that the data is traversed completely, and each word points to a list of documents. We implement this use case in our data processing framework by using nesting in the communication data structure. The hashtable we use can point to generic data values. In order to adjust it to accommodate this use case, we need to allow maps to point to other maps. By allowing this complexity, we can achieve an intermediary store after the mapper in the form shown in [Figure 6.14](#).

```
word_1 -> [(doc_id_1, occurrences), ... (doc_id_n, occurrences)]
word_2 -> [(doc_id_2, occurrences), ... (doc_id_k, occurrences)]
...
word_m -> [...]
```

Figure 6.14: Inverted Index intermediary form

1. Mapper

The mapper relies completely on the Helpers API. Implementation is shown in [Figure 6.15](#).

```

1 uint64_t bpf_prog(struct Context *arg) {
2     char delimiters[] = {'_', '\t', '\0', '\r', '\n', '\v', '\f'};
3     char MAP_DOC_ID_KEY[] = "MR_Document_Id";
4     char terminating_char = '\0';
5     uint64_t line_size = arg->data_size;
6     for(uint64_t i = 0; i < arg->data_size; i++) {
7         uint64_t token_start = i;
8         uint64_t token_end = bpf_data_get_next_index_delimiter(
9             arg->data_address, line_size,
10            delimiters, sizeof(delimiters), i);
11        if(token_start == token_end) {
12            continue; // when the current index is a delimiter
13        }
14        bpf_data_update_index_content(
15            line_size, token_end, terminating_char, arg->data_address);
16        uint64_t value = bpf_hashtable_get_data_key_value_address(
17            arg->data_structure_address, arg->data_address, token_start);
18        uint64_t currentDocumentAddress = bpf_hashtable_get_value_address(
19            arg->data_structure_address, MAP_DOC_ID_KEY);
20        if(value == 0) {
21            uint64_t newMap = bpf_hashtable_initialize(64);
22            bpf_hashtable_insert_data_key_int(
23                newMap, currentDocumentAddress, 0, 1);
24            bpf_hashtable_insert_data_key_hashtable(
25                arg->data_structure_address,
26                arg->data_address, token_start, newMap);
27            i = token_end;
28            continue;
29        }
30        int repetitions_on_doc = bpf_hashtable_get_data_key_int(
31            value, currentDocumentAddress, 0);
32        repetitions_on_doc++;
33        bpf_hashtable_insert_data_key_int(
34            value, currentDocumentAddress, 0, repetitions_on_doc);
35        i = token_end;
36    }
37    return 1;
38 }

```

Figure 6.15: Map Function for Inverted Index

Lines 1-15 are simply traversing a line and processing it in tokens. In this case, a token is a word (defined by the delimiters in line 2). Then, the code tries to retrieve a nested map to which the word points. If there is none, it will create one. It will fill it with the current `document_id` and occurrence 1. Then, an insertion is done into the original hashtable. Any subsequent encounter of that word either creates a new (`document_id`, `occurrence`) pair in case the document is new or increments the occurrence found.

Upon completing the data traversal, we end up with a fully computed inverted index inside our intermediary storage.

2. Reducer

The inverted index reducer can serve many applications, such as computing word frequencies, returning Top-N words per document, or documents with the least or no occurrences.

We believe inverted index implementation is a very good example to demonstrate the flexibility offered by the helper functions we defined for our data processing framework.

7 Conclusion

In this work, we presented an approach of integrating unikernels and eBPF in order to build a serverless framework, aiming to leverage unikernel efficiency and eBPF's flexibility for dynamically updating core data processing functions. Our evaluation included qualitative and quantitative assessments to provide a comprehensive view of the framework's strengths and limitations.

Based on our qualitative analysis, we found that using eBPF in a unikernel environment introduces certain development challenges. The eBPF verifier imposes strict constraints that can complicate the creation and adaptation of BPF data processing functions. Debugging is also limited due to the lack of traditional OS tools within unikernels, making it harder to isolate issues when developing or updating eBPF code. Additionally, the learning curve for creating verifier-compliant eBPF bytecode adds complexity, especially for developers unfamiliar with the BPF specifics and the minimalist design of unikernels. Despite these challenges, eBPF provides significant benefits. It allows us to dynamically update functions without the need for a reboot, which increases development flexibility and reduces downtime.

Our quantitative evaluation provided insights into the framework's performance and resource efficiency. We observed that the unikernel-based data processing implementations with eBPF, especially with JIT compilation, showed distinct advantages in terms of warm initialization time. The ability to dynamically update functions without restarting the unikernel made this approach highly adaptable and suitable for scenarios requiring frequent updates. In terms of cold initialization time, our results indicated that while the eBPF and JIT versions introduced some overhead, it remained manageable and did not significantly hinder overall system performance. The memory consumption in all scenarios where JIT and eBPF were used appeared to be consistent with the approaches that did not use the BPF technology.

Performance measurements of execution time and data throughput showed that the JIT-compiled eBPF configuration is slightly slower than the baseline unikernel setup when we use a very small input dataset. However, with the increase of the data size, the JIT-compiled BPF, as well as the non-JIT-compiled BPF approaches, are getting faster,

with the potential of significant performance benefits when big data is processed.

We also compared our unikernel-based data processing framework with a similar implementation in a standard Linux VM. While the results showed a huge decrease in memory consumption when using unikernels, the execution with a 5MB input dataset appeared to be two times slower than a Linux VM. We assume this happens due to internal Linux optimizations that are not available in the unikernels we used.

Finally, we implemented additional MapReduce use cases in our framework, which revealed that the eBPF-based approach is adaptable across various tasks. We also demonstrated the flexibility and usability of our BPF Helpers API by providing implementations of more complicated use cases, such as building an inverted index.

Overall, our experiments suggest that integrating eBPF with unikernels to build a data processing framework offers a promising balance between flexibility and efficiency despite some trade-offs. The adaptability of BPF functions is particularly useful for dynamic data-processing environments. However, the limitations imposed by the verifier and the challenges in debugging within unikernels highlight areas where further improvements could enhance usability and developer productivity. Future work could explore improved debugging tools for unikernels and BPF. Other areas could be implementing data processing parallelism for better performance or alleviating the verification process.

8 Future Work

This thesis develops a unikernel-based data processing framework with integrated BPF. While our evaluation shows huge performance benefits of applying this solution to big data, it is still in an initial state, and many improvements can be made. Some are directly relevant to our solution, while others are related to the technologies we use.

8.1 Parallelism

Any data processing framework that works with big datasets benefits from efficient resource utilization and scalable performance. In our current implementation, we focus on showing the advantages of unikernels and uBPF technology by running the system in a sequential manner. In the future, our implementation can be extended to support worker unikernels, which are running and processing the input dataset in parallel. One way to achieve this is to adapt the original parallelism model introduced by Google in their MapReduce framework [15]. This would significantly increase both the performance and the efficiency of the framework, making it more relevant for industry-wide use cases.

8.2 Beyond MapReduce

Our current data processing implementation follows the MapReduce execution model. However, many real-world applications require more generalized workflows. Extending the framework to support other execution models, such as data flow graphs or streaming data, will make it even more applicable to the various serverless data processing use cases.

For instance, data flow graph models enable deeper dependencies between tasks, such as complex branching and merging patterns. Such workflows are very common in machine learning pipelines. On the other hand, adding streaming data processing capabilities would allow the framework to perform data computation in real time, which would make it suitable for use cases such as sensor data analytics, event-driven architectures, and financial market analysis.

8.3 Enhancing Debugging Tools and Standardizing the Verifier

Debugging eBPF programs within unikernels creates significant challenges due to the lack of traditional OS-level tools. Developing debugging tools specifically designed for unikernels with uBPF will make the development process more efficient. These tools could include sophisticated logging, profiling or stack tracing for BPF functions. Furthermore, standardizing the eBPF verifier to reduce the learning curve and allow for greater compatibility would increase developers' productivity.

8.4 Integrated eBPF verification

Currently, we keep the verification process decoupled from the unikernels using the BPF bytecode. While this approach has advantages like keeping the images minimal and less complex, there are also certain trade-offs. By integrating the verifier directly into the unikernel, the framework could become self-sufficient, simplifying the build process and reducing the reliance on external libraries. Such integration would require a lightweight implementation to maintain the unikernel's minimal resource footprint while still ensuring high security.

An example approach for achieving this would be to adapt the verification as a unikraft C++ library. This can be done by allowing `libcxx` in the unikraft configuration file and integrating a statically compiled version of the PREVAIL ebpf-verifier inside the image. By exposing a wrapper external C function, we should theoretically be able to call the verifier directly from the unikernel logic.

8.5 Shared Memory between the Unikernels and the Scheduler

Currently, our framework provides a communication channel between the scheduler and the unikernels by mounting a shared file system. This is a convenient and working approach. However, it can introduce latency and resource overhead, particularly for high-frequency or large-scale data tasks. The communication channel can be improved by implementing some shared cache or memory. This would enable direct interactions between the scheduler and the unikernels and increase data-sharing speed.

Abbreviations

9pfs	Plan 9 Protocol File System 28, 29, 69
API	Application Programming Interface 12, 19–23, 33–36, 39, 58, 60, 64, 71
AWS	Amazon Web Services 4, 8
BPF	Berkeley Packet Filter iv–vi, 2, 7, 8, 12, 16, 17, 19, 21–23, 27, 30–33, 35–41, 43–46, 48–51, 53, 55–60, 63–66, 69
BSD	Berkeley Software Distribution 7
CPU	Core Processing Unit iv, vi, 7, 45, 52, 55–57, 71
eBPF	extended Berkeley Packet Filter iv, v, vii, 2, 7–9, 11–14, 18, 19, 22, 23, 26, 27, 30–33, 37, 40, 42–48, 52, 63, 64, 66, 69
ELF	Executable and Linkable Format 37
FS	File System 16, 17, 28
GPU	Graphics Processing Unit 45
HDFS	Hadoop Distributed File System 4, 5
HTTP	Hypertext Transfer Protocol 3
I/O	Input/Output 22, 27, 36, 46, 52
ID	Identification 19
JIT	Just-in-Time iv, vi, 7, 38, 46, 48–50, 52, 56, 59, 63
NFS	Network File System 28
OS	Operating System 1, 2, 6, 7, 10, 11, 26, 27, 45, 50, 53, 55, 63, 66
QEMU	Quick Emulator 27, 28, 45, 69
RDDs	Resilient Distributed Datasets 4
RSA	Rivest-Shamir-Adleman 42
uBPF	Userspace eBPF vi, 8, 26, 27, 30, 32, 33, 36, 38, 42, 43, 46–48, 51, 54, 65, 66, 69
UML	Unified Modelling Language 29
VM	Virtual Machine 1, 6–9, 11, 13, 14, 18, 19, 22, 32, 33, 37, 38, 42, 43, 46, 49–56, 58, 64

VMM Virtual Machine Monitor 6

List of Figures

2.1	Serverless Key Characteristics	3
2.2	Map Function [15]	5
2.3	Reduce Function [15]	5
2.4	High-level overview of existing virtualization technologies [16]	6
3.1	Serverless Limitations	10
3.2	High-level overview of the framework	13
4.1	Architecture design of the framework	15
4.2	Unikernel architecture design	17
4.3	Decoupled Verification	25
5.1	Command to run a Unikraft image	28
5.2	Additional QEMU arguments to run an image with 9pfs	28
5.3	Unikraft config for mounting 9pfs as a root file system	29
5.4	Chained Hashtable	30
5.5	BPF program context data structure	31
5.6	Pointer cast to uint64_t	31
5.7	uint64_t cast to a pointer	31
5.8	Register uBPF helper	32
5.9	Macro definition for a uBPF helper	32
5.10	Simple BPF function	37
5.11	BPF function compilation	37
5.12	BPF Context Descriptor	39
5.13	eBPF Helper Prototype in PREVAIL	40
6.1	Cold initialization time	47
6.2	Warm initialization time	48
6.3	Word Count performance (25KB dataset)	49
6.4	Word Count performance (1MB dataset)	50
6.5	Word Count performance (5MB dataset)	51
6.6	Word Count memory footprint (25KB dataset)	52
6.7	Word Count memory footprint (1MB dataset)	53

List of Figures

6.8	Word Count memory footprint (5MB dataset)	54
6.9	Fibonacci number calculation	55
6.10	π calculation using the Leibniz Series	56
6.11	CPU-intensive functions Performance	57
6.12	Map Function for a Search Engine use case	58
6.13	Search use case performance	59
6.14	Inverted Index intermediary form	60
6.15	Map Function for Inverted Index	61

List of Tables

2.1	Comparison between the different virtualizations [16]	6
4.1	Hashtable Helpers API (1)	20
4.2	Hashtable Helpers API (2)	21
4.3	Data Helpers API	22
4.4	Debugging Helpers API	23
4.5	PREVAIL Advantages	23
4.6	Decoupled Verification Advantages [31]	24
5.1	Debugging Helpers API Prototypes	33
5.2	Hashtable Helpers API Prototypes (1)	33
5.3	Hashtable Helpers API Prototypes (2)	34
5.4	Hashtable Helpers API Prototypes (3)	35
5.5	Data Helpers API Prototypes	36
5.6	PREVAIL return types	40
5.7	PREVAIL argument types	41
6.1	Machine Specifications	45
6.2	Memory Footprint with CPU-intensive functions	57
6.3	Search use case Memory Footprint	60

Bibliography

- [1] V. Lannurien, L. d’Orazio, O. Barais, and J. Boukhobza, “Serverless cloud computing: State of the art and challenges,” in May 2023, pp. 275–316, ISBN: 978-3-031-26632-4. DOI: 10.1007/978-3-031-26633-1_11.
- [2] B. Tan, H. Liu, J. Rao, X. Liao, H. Jin, and Y. Zhang, “Towards lightweight serverless computing via unikernel as a function,” in *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, 2020, pp. 1–10. DOI: 10.1109/IWQoS49365.2020.9213020.
- [3] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, “Unikernels: Library operating systems for the cloud,” in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’13, Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 461–472, ISBN: 9781450318709. DOI: 10.1145/2451116.2451167.
- [4] eBPF.io authors. “What is eBPF? An Introduction and Deep Dive into the eBPF Technology.” (), [Online]. Available: <https://ebpf.io/what-is-ebpf/> (visited on 10/01/2024).
- [5] G. Adzic and R. Chatley, “Serverless computing: Economic and architectural impact,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, Paderborn, Germany: Association for Computing Machinery, 2017, pp. 884–889, ISBN: 9781450351058. DOI: 10.1145/3106237.3117767.
- [6] Amazon.com authors. “AWS lambda.” Accessed: 2024-10-15. (), [Online]. Available: <https://aws.amazon.com/lambda/> (visited on 10/15/2024).
- [7] Alphabet Inc. “Google run functions.” Accessed: 2024-10-15. (), [Online]. Available: <https://cloud.google.com/functions> (visited on 10/15/2024).
- [8] Microsoft. “Microsoft azure functions.” Accessed: 2024-10-15. (), (visited on 10/15/2024).
- [9] Z. Shojaei Rad and M. Ghobaei-Arani, “Data pipeline approaches in serverless computing: A taxonomy, review, and research trends,” *Journal of Big Data*, vol. 11, no. 1, p. 82, Jun. 2024, ISSN: 2196-1115. DOI: 10.1186/s40537-024-00939-0.

- [10] A. S. Foundation, *Apache hadoop*, Accessed: 2024-11-10, 2024.
- [11] A. S. Foundation, *Apache spark*, Accessed: 2024-11-10, 2024.
- [12] N. Ahmed, A. L. C. Barczak, T. Susnjak, and M. A. Rashid, "A comprehensive performance analysis of apache hadoop and apache spark for large scale data sets using hibenach," *Journal of Big Data*, vol. 7, no. 1, p. 110, Dec. 2020, issn: 2196-1115. doi: 10.1186/s40537-020-00388-5.
- [13] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12, San Jose, CA: USENIX Association, 2012, p. 2.
- [14] M. R. Ghazi and D. Gangodkar, "Hadoop, mapreduce and hdfs: A developers perspective," *Procedia Computer Science*, vol. 48, pp. 45–50, 2015, International Conference on Computer, Communication and Convergence (ICCC 2015), issn: 1877-0509. doi: <https://doi.org/10.1016/j.procs.2015.04.108>.
- [15] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008, issn: 0001-0782. doi: 10.1145/1327452.1327492.
- [16] M. F. Alexander Jung, *Unikernel concepts*.
- [17] D. Williams, R. Koller, M. Lucina, and N. Prakash, "Unikernels as processes," Oct. 2018, pp. 199–211. doi: 10.1145/3267809.3267845.
- [18] A. Randal, "The ideal versus the real: Revisiting the history of virtual machines and containers," *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020, issn: 0360-0300. doi: 10.1145/3365199.
- [19] F. authors, *Bpf(1) freebsd manual pages*, 13.2, Accessed: 2024-10-15, FreeBSD, Oct. 2021.
- [20] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture," in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, ser. USENIX'93, San Diego, California: USENIX Association, 1993, p. 2.
- [21] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18, Heraklion, Greece: Association for Computing Machinery, 2018, pp. 54–66, isbn: 9781450360807. doi: 10.1145/3281411.3281443.

- [22] K. Zandberg, E. Baccelli, S. Yuan, F. Besson, and J.-P. Talpin, “Femto-containers: Lightweight virtualization and fault isolation for small software functions on low-power iot microcontrollers,” in *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, ser. Middleware ’22, Quebec, QC, Canada: Association for Computing Machinery, 2022, pp. 161–173, ISBN: 9781450393409. DOI: 10.1145/3528535.3565242.
- [23] I. Team, *Ubpf: Userspace ebpf vm*, Accessed: 2024-10-15, 2024.
- [24] B. Sharma and D. Nadig, “Ebpf-enhanced complete observability solution for cloud-native microservices,” Apr. 2024.
- [25] Amazon.com authors, *AWS lambda*, Accessed: 2024-11-01, 2024.
- [26] Alphabet Inc, *Create user-defined functions for dataflow templates*, Accessed: 2024-11-01, 2024.
- [27] Microsoft, *Run the mapreduce examples included in hdinsight*, Accessed: 2024-11-01, 2024.
- [28] A. S. Foundation, *Apache openwhisk documentation*, Accessed: 2024-11-01, 2024.
- [29] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv, “Simple and precise static analysis of untrusted linux kernel extensions,” in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1069–1084, ISBN: 9781450367127. DOI: 10.1145/3314221.3314590.
- [30] P. Team, *Prevail - a new ebpf verifier*, Accessed: 2024-11-15, 2024.
- [31] M. Craun, A. Oswald, and D. Williams, “Enabling ebpf on embedded systems through decoupled verification,” in *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, ser. eBPF ’23, New York, NY, USA: Association for Computing Machinery, 2023, pp. 63–69, ISBN: 9798400702938. DOI: 10.1145/3609021.3609299.
- [32] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, Ș. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, and F. Huici, “Unikraft: Fast, specialized unikernels the easy way,” in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys ’21, Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 376–394, ISBN: 9781450383349. DOI: 10.1145/3447786.3456248.
- [33] U. GmbH, *The fast, secure and open-source unikernel development kit*.
- [34] V. Hendrychová, *Ubpf for unikraft*, Accessed: 2024-06-15, 2023.

- [35] A. S. Jay Schulist Daniel Borkmann, *Linux socket filtering aka berkeley packet filter (bpf)*.
- [36] T. Q. P. Developers, *Qemu docs*.
- [37] P. 9. Foundation, *Plan 9 from bell labs*.
- [38] D. C. M. Library of Congress and S. Division, *Enron email dataset*.