SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

UniBPF: Safe and Verifiable Unikernels Extensions

Kai-Chun Hsieh

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

UniBPF: Safe and Verifiable Unikernels Extensions

UniBPF: Sichere und Überprüfbare Unikernel-Erweiterungen

Author:	Kai-Chun Hsieh
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Dr. Masanori Misono
Submission Date:	15th November 2023

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

Munich, 15th November 2023 Kai-Chun Hsieh

Acknowledgments

I want to thank Dr. Masanori Misono for advising me during my thesis, guiding me through the challenges of this work, such as thesis writing, and helping me with brilliant ideas when I encountered implementation obstacles. I would also like to express my gratitude to Yi-Ju Liu, Yueh-Cheng Liu, and Yung Hsu Yang, who actively gave me a lot of inspiration during this work. Last but not least, I would like to thank Prof. Dr. Pramod Bhatotia for supervising the thesis and taking an interest in the project's progress.

Abstract

Despite the performance and security benefits that Unikernels provide through a more compact and isolated runtime environment, some researchers argue that Unikernels are impractical for production environments due to their lack of flexibility and extensibility.

Previous research addresses these issues by proposing solutions that allow Unikernels to be dynamically reconfigured at runtime and by providing additional external system interfaces, such as kernel console and kernel hook interfaces with an extended Berkeley Packet Filter (eBPF) runtime system, similar to those used in Linux systems.

However, the lack of a suitable Berkeley Packet Filter (BPF) program verification process in the previous solution requires the BPF runtime to be enabled with an interpretation mode to effectively detect and prevent sandbox escapes. Nevertheless, relying solely on interpreters may lead to suboptimal results due to insufficient security guarantees in blocking faulty programs that may lead to runtime errors and performance limitations.

To address the above challenges, in this research, we present UniBPF - A comprehensive framework that provides BPF verifications and complementary frameworks to facilitate seamless integration of UniBPF into developers' current projects. Through decoupled BPF verification, first introduced for Unikernel systems, UniBPF provides a unified BPF verification standard while preserving the high customizability and compactness properties of Unikernels.

In this work, we showed that our verifier application can bring extraordinary security compared to the existing interpreted solution. In addition, by implementing just-in-time compilation for BPF programs with UniBPF, our control group program showed an increase of up to 900% in Unikernel BPF runtime performance. Besides, micro-benchmarks suggest that the theoretical performance at the instruction level can increase by up to 600%. Furthermore, our evaluation using a BPF program of about a thousand instructions shows a 40% performance improvement in kernel-tracing a Nginx application compared to the interpreted one.

This study generally proves that UniBPF improves BPF runtime security guarantees and performance with negligible overhead.

Contents

1 Introduction

Since Google released core cloud-computing-related papers in 2003 and the commercialization of Amazon EC2's beta version in 2006 [Qia+09], many pioneers have dedicated themselves to introducing various cloud-computing services and new concepts above them. Examples include virtual machines, containers, serverless computing, and other no-code/low-code platforms.

Among these technologies, virtual machines are typically considered the solution with the most overhead but more robust system security due to their isolated operating system nature.

To minimize operating system overhead, the concepts of library perating System (libOS) were introduced in the 1990s, alongside implementations of such as Exokernel [EKO95] and Nemesis [Han99], and were later refined by Porter et al. [Por+11] in 2011. The expected points of the above-mentioned libOS implementations, including Exokernel and Nemesis, are that both systems are designed to allow the application to communicate directly with the hardware or indirectly through utility libraries.

Based on the concepts of libOS, MirageOS [MS13] created the very first Unikernel system in 2013. Compared with libOS, MirageOS provides a more general system solution with kernel libraries that support zero-copy device I/O, block-storage I/O, networking, and concurrency with the programming language OCaml [Ler+23].

While implementing Unikernel-related applications, Misono et al. found that Unikernels are not practical due to the lack of debuggable, observable, and real-time maintainable properties. Also, their runtime extensibility is very limited [Mis23]. To address the above problems, xIO [Mis23], an abstracted *"safe overlay"* for Unikernels, has been proposed. The most notable achievements of xIO are, for example, providing Unikernels with interactive debug interfaces to allow maintainers to communicate with and reconfigure their applications at runtime, the ability to sideload and run arbitrary executables in the Unikernels side-by-side with the applications via the debug interface, and the ability to mount external file systems on the Unikernels. Their work mitigates the traditional tradeoff between Unikernels and conventional operating systems, i.e., compactness versus flexibility. For example, one of the most attractive use cases with xIO is to back up the database by running an **external utility software** on the debug console and exporting the backup files via the mounted external file systems.

Noticing the limitations of xIO's risks associated with executing arbitrary executables

without proper sandboxing, Hendrychová et al. have proposed several ideas in their master's thesis [Hen23]. One of the most critical tasks in their work is integrating a **verifiable** BPF runtime environment into Unikernel systems. Also, an extended use case implementing a kernel tracing framework with the BPF runtime is included in their work.

However, we observed several limitations in their solutions. First, they applied an interpreted runtime due to the lack of a suitable BPF verification process but a strong demand for security requirements; nonetheless, our evaluations suggest these mechanisms **cannot provide solid security assurances** such as runtime error prevention. Second, interpreters lead to **inefficiencies** that may hinder the future development of BPF runtime applications on Unikenrel systems. For example, a study, RapipPatch [He+22], found that the interpreted mode in their modified BPF runtimes experienced a slowdown of 475% to 1600%. Another research paper, BPF+ [BMG99], also supports this point of view with its own JiT compiler implementation. Third, the typical approach of integrating verifiers into the system to ensure verifier integrity and verification quality is **complicated** since major BPF verifier implementations either require specific runtime environments (e.g., C++) or are difficult to port. Worse, due to the lack of support for time-shared multi-threading and multi-processing, time-consuming verification operations can block the application from processing client requests, causing clients to experience **unacceptable latencies**.

To address these concerns, we introduced UniBPF - a comprehensive BPF verification framework. This framework features a BPF verification application and accompanying libraries that simplify the workflow for developers when integrating UniBPF into their existing projects. The included BPF verification application, UniBPF Terminal, works **independently from Unikernels** on the host system to audit the quality and validity of BPF programs. Such a design does not block the primary application services due to time-consuming BPF verifications, allows easy updates/upgrades of BPF verifiers without fear of disrupting existing systems, and preserves the compactness of the Unikernel application. In addition, by using **on-the-fly verifier initialization**, UniBPF Terminal avoids fragmented verifier solutions while providing a standard solution that applies to all Unikernel applications. Finally, with the security guarantees provided by proper verification processes, our design allows Unikernel's BPF runtime system for improved performance by JiT compiling BPF programs into platform-native code.

To assess the security commitments and quantify the performance enhancements of our design, we developed a prototype utilizing Unikraft [Kue+21] and the debug interface + BPF runtime offered by xIO [Mis23][Hen23]. In the Evaluation chapter, we integrated our prototype with an example application and several real-world applications such as Nginx [Ree08] and Redis [San09]. Based on NixOS [DL08], through these integrations, to prove the effectiveness of our system's security measures, we

tested our systems with various BPF programs containing malicious code fragments and implementation errors that could potentially cause catastrophic results. In addition to the security benefits, we also measured the verification overhead, the JiT compilation overhead, and the performance gain resulting from our system design.

In summary, our evaluations suggest that UniBPF can effectively protect against security threats that current interpreter solutions cannot, thereby preventing system crashes. At the same time, it has minimal runtime impacts on Unikernel applications and is easy to use and maintain. Also, from the performance perspective, by enabling the **Just-in-time (JiT) compilation mode**, our evaluations suggest a performance improvement of up to 600% at the instruction level and also a 900% performance improvement in one of our control group programs. In addition, we found that the performance of the current Unikernel kernel tracing solution [Hen23] is improved by up to 1.32% on real-world applications using a smaller scale BPF performance counter program and by up to 40% on the same applications using a larger scale BPF program with over a thousand instructions. What's even more surprising is that we observed possible traces of hardware acceleration on **JiT**-compiled BPF programs, which significantly accelerated some of our evaluation programs and may be pretty beneficial for real-world use cases. However, a deeper investigation to confirm this will be done in the future.

The notable contributions of our jobs are listed below.

- Integrate BPF verification processes to Unikernels with negligible runtime impact.
- The one-for-all BPF verifier solution can efficiently serve all compatible Unikernel instances and is manageable and sustainable.
- More secure BPF runtime environment achieved by BPF verification processes.
- More efficient BPF runtime environment by enabling JiT compilations.

2 Research Background

The following sections review the research background that motivates the current work. First, we explain the insufficiency of current virtualization techniques, demonstrating the necessity of Unikernels and the problems they can solve. Then, we summarize our research background by describing the practical problems developers encounter with the current Unikernel solutions and how extensibility solves those problems. In addition, we summarize the historical and background information about the main focus of this work, eBPF system, including the BPF language, BPF helper functions, and BPF program types.

2.1 Limitation of Existing Virtualization Techniques

Among modern software solutions, container technologies have become one of the top choices when developers must host multiple applications on a limited number of servers due to their efficiency, high compatibility, flexibility, and support from thriving communities.

However, recent research has addressed the Achilles heel of container technology: it relies on the virtualized operating system kernel as a shared resource among all the different instances, and once the operating system is compromised, the entire host system, including the other hosted containers, is compromised. For example, CVE-2019-5736[1] suggests that runC as a low-level container utility tool that provides features to spawn new containers on the host system used before Docker 18.09.2 and other products may allow malicious containers to escalate their privileges by overwriting it [Avr22]. Later, one of the authors of one of the most successful container runtimes, Linux Containers (LXC), also addressed this vulnerability in their project.

In 2020, Agache et al. addressed the abovementioned issue and introduced Firecraker [Aga+20]. As one of the most successfully commercialized micro Virtual Machine (VM) solutions, Firecracker is widely used by Amazon Web Services (AWS), especially in serverless computing, such as AWS Lambda and Fargate [Aga+20]. The idea behind mircoVM is to use paravirtualization and aggressive optimization techniques to achieve much stronger system isolation via **virtual machines** but with as little

[1] `https://nvd.nist.gov/vuln/detail/CVE-2019-5736`

overhead as possible.

For example, Firecracker achieved its lightweight features by eliminating superfluous driver code and modifying it for optimal performance in the limited range of emulated IO devices based on Google's *Chrome OS Virtual Machine Monitor*, which is a Linux-based project [Aga+20]. In addition, Firecracker has optimized its system for maximum efficiency to provide language runtime environments, such as removing Linux documentation, based on our observations.

2.2 Unikernel

Unikernels achieve the goal with even more aggressive ideas than microVM. Compared with conventional operating systems, Unikernel systems pose a fundamental challenge to the boundary between kernel and applications, as they assume the application's complete trustworthiness. In Unikernel systems, a series of libraries replaces the traditional operating system. For instance, Unikraft [Kue+21] implemented Linux system APIs into utility functions, which differs from conventional methods where system APIs are only accessible through system calls with processor-specific instructions such as *SYSENTER* and *INT* in the case of x86 systems. Such instructions aim to go through the traditional operating system boundaries in a controlled way but can cause significant overhead due to context switches and corresponding cache misses. Such boundaries also sometimes cause redundantly copying of data back and forth between kernel and user spaces. In addition, the compactness of Unikernel makes the application exceptionally safe from the excessive components being exploited [Bue19].

Unikraft, one of the most popular Unikernel implementations, achieves its goal by turning system calls into regular function calls, which is especially beneficial for applications requiring frequent IO operations. Moreover, Unikraft exposes all kernel resources to applications and allows the application to statically bind to these resources to further reduce overhead.

With all these efforts, *Unikraft* results in a 1.7x - 2.7x performance improvement with existing Linux-based applications such as Nginx, SQlite, and Redis [Kue+21]. With Unikraft, these applications can be started in a very short time ($\leq$ 40ms) and require very little memory to run ($<$ 10MB).

2.3 Extensible Unikernel

Unikernels achieve security, performance, and compactness by only preserving kernel components that serve the application. For example, in most Unikernel systems,

various maintenance, debugging, and state-saving tools do not exist [Mis23]. Traditionally, developers can easily access systems, such as Linux, through *ssh* [LY06] to perform operational tasks such as database backups, service reconfiguration, and service restarts. Similarly, for Windows, known solutions include Windows Remote Shell (WinRS) [Mic16] and Remote Desktop (RDP) [LXC23].

It is this sacrifice of flexibility, extensibility, debuggability, monitorability, and mainly due to the lack of runtime extensibility, that makes Unikernels impractical in production environments [Mis23][Bue19][Tal+20]. And xIO addresses these issues by introducing *safe overlay* to the Unikernels [Mis23]. Based on their ideas, xIO implemented, for example, an interactive command-line interface and support for mounting external filesystems on Unikernels using 9P protocols.

Furthermore, Hendrychová et al. proposed additional methods on top of xIO to improve the extensibility of Unikernel systems in their thesis [Hen23]. Like the feature commonly used in Linux systems, they enabled BPF programs to be downloaded to Unikernel systems and demonstrated the possibility of tracing and monitoring Unikernel systems with function hooks. They also proposed approaches for building fully extensible BPF runtime ecosystems in Unikernels. In particular, the customizable BPF helper functions.

2.4 Berkerly Packet Filter

The Berkeley Packet Filter, named initially after the BSD Packet Filter, was introduced by McCanne et al. in 1993 [MJ93]. The researchers, then working at the Lawrence Berkeley Laboratory, found that existing stack-based network monitoring solutions on UNIX were suboptimal because the network monitoring software was running in user space, requiring redundant copies of packets from kernel space to user space. In the end, they proposed the solution of attaching programs in kernel space directly to low-level network interfaces. To do this, the researchers introduced a lightweight kernel register-based virtual machine, the BPF Psudo-Machine, and its own instruction set.

As a result, their solution outperforms the existing solution, CMU/Stanford Packet Filter (CSPF) [MRA87], by at most 20 times, and even 100 times compared with the other existing solution in SunOS's Network Interface Tap (NIT) [SW94].

2.5 Extended Berkeley Packet Filter

Since its public release in version 3.18, Linux introduced a more comprehensive BPF language kernel virtual environment, also known as eBPF [Aut23] [AM14]. Building on the classic BPF, which is basically a static packet filter, eBPF allows the execution of

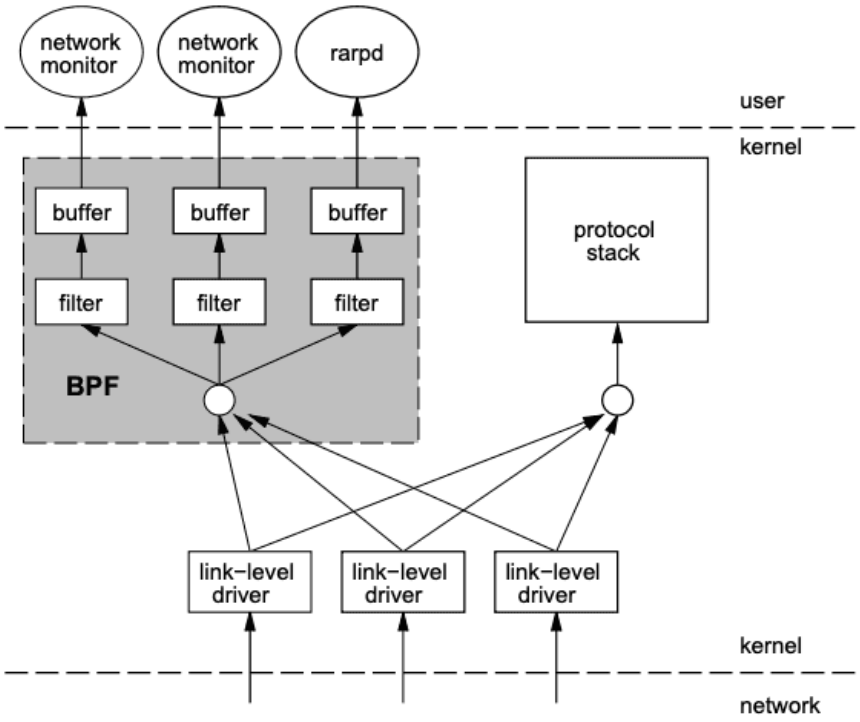

Figure 2.1: System Overview of the Berkeley Packet Filter. Image taken from [MJ93].

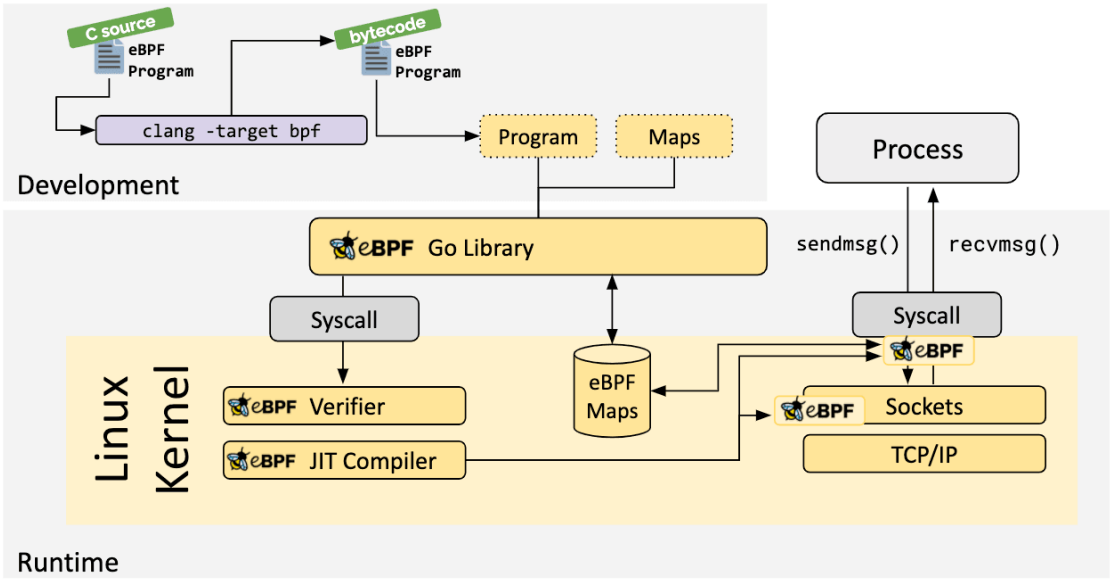

Figure 2.2: System Overview of Linux's *eBPF* System. Image taken from [Aut].

more complex BPF programs in the kernel and significantly extends the use cases of BPF. The Linux eBPF system consists of numerous system APIs and advanced features, including helper functions and key-value stores (maps), as illustrated in Figure 2.2. The most prevalent applications of the eBPF system in Linux nowadays are networking, kernel tracing, auditing, and monitoring. Two notable applications include *eXpress Data Path (XDP)* [Høi+18], which utilizes eBPF's characteristic to construct a high-speed packet processing framework, and the intriguing *sshlog* [Hil23], which logs SSH connection activity using eBPF.

2.5.1 BPF Helper Functions

With the unveiling of eBPF, BPF helper functions have also been introduced to enrich the eBPF ecosystem. These functions enable the BPF program to interact with the system under dedicated controls.

`bpf_map_lookup_elem` is one example of a BPF helper function present in the Linux system, which retrieves for the BPF program the value element from the key-value store of the eBPF system with the specified key.

A noteworthy aspect of the Linux eBPF system is that the BPF verifier also validates the use of helper functions in the BPF program before they are attached to the system.

2.5.2 BPF Program Types

In addition to its helper functions, Linux allows developers to designate program types for their BPF programs by naming program sections. For example, placing a BPF program under the *socket* section will mark it as a **BPF_PROG_TYPE_SOCKET_FILTER**. With such program type, Linux will provide `struct __sk_buff` as the parameter at runtime, containing metadata and network packet data pointers.

Some BPF helper functions have limited usage in the Linux system. Only marked BPF programs of specific program types can employ certain helper functions. And the BPF verifiers also validate these.

3 Overview

The lack of appropriate BPF verification methods makes the previous BPF runtime solution [Hen23] less reliable in resisting malicious and error-prone BPF programs. And it is also the primary reason that they failed to enable a more efficient BPF runtime with JiT compiled mode.

In this work, we aim to introduce BPF verification procedures for those Unikernel applications deployed by cloud providers via hypervisor virtual machines. However, we argue that the traditional approach to integrating BPF verifiers in Unikernel scenarios is suboptimal due to implementation and usability difficulties. Instead, we propose implementing BPF verifiers as an external application. This design provides several benefits, such as improved efficiency, sustainability, and system stability. Also, the idea of creating a *one-for-all* verifier utilizing *on-the-fly* BPF verifier initialization techniques enables reduced software fragments and gives better maintainability. And since our design aims to place the verification component in the host system alongside the hypervisor, which we assume to be trustworthy, such a design does not compromise security.

In general, we aim to enhance the security features of the BPF language to Unikernels with better BPF runtime system performance without sacrificing usability and flexibility.

In the subsequent sections, we will provide an overview of our research context, discuss practical needs, and address research challenges we encountered.

3.1 Assumption and Threat Model

We assume that the Unikernel application integrated with UniBPF is free from any poisons such as backdoors injected through, for example, malicious software supply chains. The Unikernel applications offer only the features for which they were designed and were expected to provide. Simultaneously, the attack on a Unikernel application is initiated by executing insecure or malicious BPF bytecodes on it. Additionally, it is assumed that developers and maintainers of the Unikernel application will only execute BPF bytecodes within frameworks established by UniBPF, avoiding any attempt to bypass the system's security measurements. Finally, it is assumed that the cloud provider's infrastructure is reliable, meaning UniBPF components on the host system

will not be tampered with, and there will be no man-in-the-middle between UniBPF and the guest applications.

3.2 Design Goals

- **Security**: Our work prioritizes improving the security of the Unikernel runtime system, including immunity from malicious and error-prone BPF programs.
- **Sustainable Design**: Our system should be easy to use, integrate, and operate with minimal impact on the runtime performance of the Unikernel applications. It should also be easy to maintain.
- **Performance**: Our system shall result in acceptable overhead while improving the performance of the BPF runtime environment upon the existing works.

3.3 Design Challenges and Key Ideas

The following sections list the design issues we identified and our ideas for addressing them.

3.3.1 Challenge 1: Impact of Verification Processes on Unikerenl Applications' Runtime

While designing the verifier system with the Unikernel system, we identified several concerns. The lack of preemptive multitasking capabilities in the existing Unikernel system was noted, as the applications are typically the only process running exclusively in the system. Another notable concern is the absence of multiprocessor support. Consider one of the test programs from the *Verification Overhead* chapter as an example. A simple BPF program consisting of only one layer of if/else-if clauses within a for loop took 12.05 milliseconds to verify. Suppose we follow the traditional BPF verification design of integrating the verifier directly into Unikernel applications, similar to the BPF language runtime implementations in other operating systems such as Linux, as well as a BPF runtime integration project based on Windows [Mic]. When performing such heavy verification tasks within a Unikernel application, due to its single-processor nature and lack of effective preemptive multitasking capabilities, application clients may experience, in such cases, observable latency when new BPF programs are attached to the system.

Key Idea: A decoupled BPF verifier from the Unikernel application executed as a process on the host system.

3.3.2 Challenge 2: Integrate Verifier into Unikernel Application is Barely Feasible

Unikernel applications prioritize customizability and efficiency by eliminating unnecessary components and redundant procedures and focusing solely on serving applications. While developing UniBPF, we examined the complexity and size of existing verifiers' code and found that the traditional way of integrating BPF verifiers into systems was impractical, unachievable, and contrary to Unikernel's primary goal of maintaining concise and straightforward applications.

Integrating deserialization, disassembly, reconstruction of program execution paths, and BPF bytecode analysis features into a single BPF verifier is complex and may require complicated runtime environments. PREVAIL [Ger+19], for example, alone contains over 27,000 lines of code at the time of this thesis's writing. Simultaneously, integrating the standard C++ libraries required by PREVAIL poses significant challenges in the case of Unikernels. Another project that provides a BPF verification framework we have identified is KLINT [Pir+22]. It consists of about 10,300 lines of code and even requires a Python runtime environment to evaluate the verification requirements. The BPF verifier in Linux also contains over 14,000 lines of code, not including the deserializer, and is closely intertwined with Linux's internal data structures and system designs.

Key Idea: A decoupled BPF verifier from the Unikernel application utilizing the host system's runtime environments.

3.3.3 Challenge 3: Configuring Shared Verifier for Different Unikernel Applications

Assuming that we have addressed the above-mentioned **design challenge 1** and **design challenge 2** by decoupling the BPF verifier from Unikernels, our verifier system still remains barely usable due to the lack of a standardized BPF runtime system.

Of course, we can implement our system using traditional approaches, which require developers to implement their BPF runtime to conform to a standardized and unified specification, thus enabling the ability to provide a generalized BPF verifier for each application. Nevertheless, this approach forces developers to incorporate BPF features, including those they may not want to possess, such as the program types they wish to prohibit.

We believe conventional approaches conflict with the ethos of Unikernels, e.g., compactness. As such, we would like to respect the design of Unikernel systems to be freely customizable and implemented to meet the specific needs of developers.

Yet, pursuing such perfection could lead to an overabundance of fragmented BPF verifier implementations if every Unikernel adopted incompatible BPF runtime specifications. Such a scenario would make it extremely painful to maintain and update verifiers.

Key Idea: We provide a standardized BPF runtime framework that requires developers to export their BPF runtime definitions in regulated ways. At the same time, we integrate the BPF verifier into a unified BPF verification application that queries and initializes the BPF verifier on the fly, so that it can be used for any Unikernel application.

4 Design

This chapter gives a detailed introduction to the design of UniBPF. It consists of three main parts:

- Debug command interception and BPF verification with a verifier application, **UniBPF Terminal**, which is an independent component apart from the Unikernels.
- A developer-friendly BPF framework to expose the BPF runtime system specification of the existing Unikernel applications and integrate them with UniBPF.
- A highly efficient BPF runtime based on the JiT compiler.

The system overview shown in Figure 4.1 shows the BPF programs' workflow when the developers try to attach or run a BPF program using UniBPF.

The design of UniBPF relies first on a debug interface through which debug commands can be passed, and the appropriate processes can be initialized. Second, a shared and mountable external file system must be available to both the host and guest Unikernels to share BPF programs between the host and guest systems.

In summary, when application developers need to attach or run a BPF program on a Unikernel application, they need to place the BPF program on the shared file system. Then, they mount the shared file system on their guest Unikernel system, for example, using bind-mount. After all of the above, developers can finally send the debug command requesting the attachment or execution of the appropriate BPF programs.

Internally, UniBPF Terminal will intercept each debug command sent to the Unikernel applications and analyze and verify the BPF program in the given case. If the verifier accepts the BPF program, the debug command is redirected to the debug interface. Once the Unikernel system receives such a command, it loads the BPF program and sends it to the JiT compiler first. After that, the generated platform native executables are either attached to the kernel tracing module or executed immediately.

The following sections outline the requirements for integrating UniBPF into Unikernel applications first. Then, a comprehensive overview of the system design will be given, including the designs of the UniBPF Terminal and the UniBPF framework libraries.

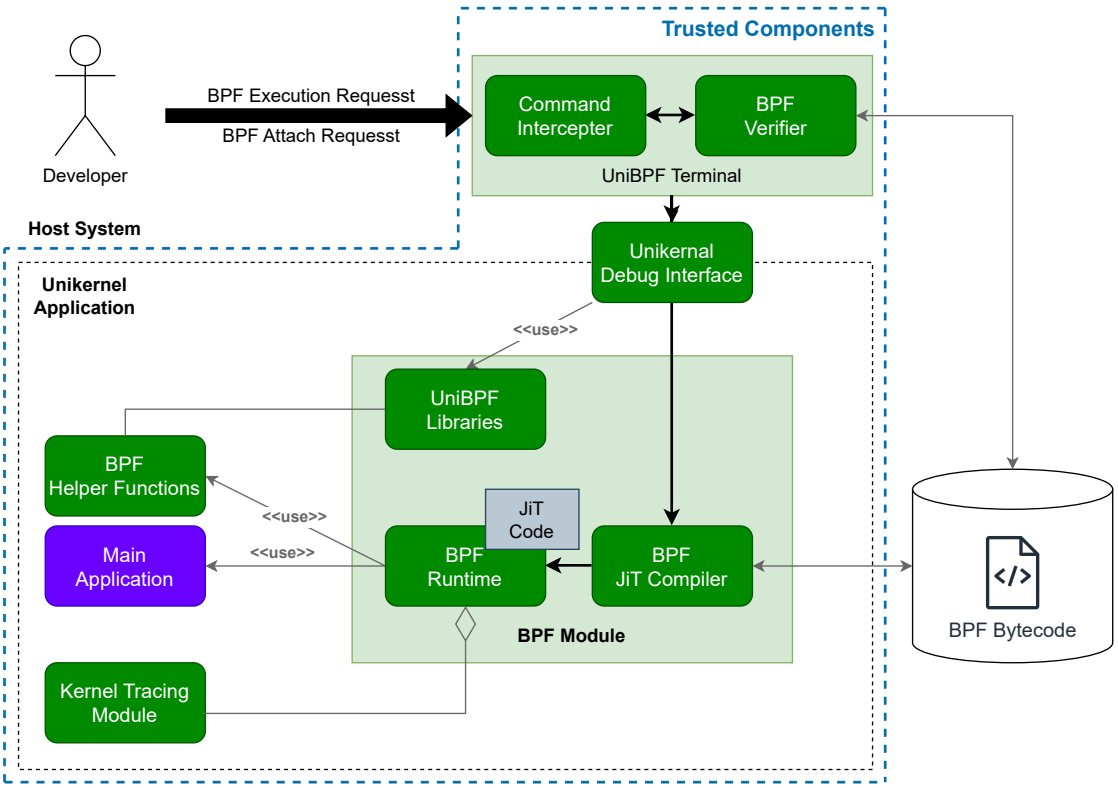

Figure 4.1: **System Overview of UniBPF.** UniBPF contains an external component that intercepts BPF program-related commands, initializes BPF verifiers to verify the specified BPF bytecodes, and then invokes execution of the specified BPF program via debug interfaces.

4.1 Prerequisite

UniBPF requires that the Unikernal application be set up with the following mechanisms in place:

1. **Debug Interface**: Unikernal applications need to be embedded with a text-based, console-like debug interface, through which Unikernal applications can process debug commands and return the appropriate responses through the same channel.

   A real-world example of such a debug interface is the Hayes AT commands for the serial port of the cellular modem, which are widely used to control the modems and consist of a series of short text strings that can be combined to produce commands for operations, such as dialing, hanging up, and changing the parameters of the connection [con23].

2. **Shared Filesystem**: Unikernel applications must be able to mount an external file system accessible from both the host and the guest system. When requested to run BPF programs for the first time in the session, **the Unikernel application is only allowed to load the BPF program in bytecode form from the shared filesystem**.

4.2 UniBPF Terminal

We developed UniBPF with a standalone BPF verifier application embedded with a command line terminal for Unikernels as shown in Figure 4.2, which, practically, will be executed on the host system. The Terminal uses the debug interface of the Unikernel applications and acts as a man-in-the-middle to intercept communication between the developers and the debug interface of the Unikernel applications.

The Terminal consists of two parts: the frontend and the backend. The frontend provides a TeleTYpewriter (TTY) terminal application, and a *command filter* acts as an arbitrator, deciding how to handle the incoming debug commands. Once the *command filter* detects a system-critical command, it detains it. Then, The command terminal immediately triggers the backend components according to the command type and checks whether executing the provided debug command is safe.

For example, in the BPF 's cases, the *command filter* will start the BPF verifier as the backend component to verify the BPF program as soon as the developer requests the Unikernel application to execute or to attach a BPF program. Then, depending on the verification results, the *command filter* will either redirect the command to the debug interface of the Unikernel application, where the debug command will be executed,

or immediately reject the command and provide appropriate error responses to the developer via the terminal application.

In summary, this design pattern solves the design challenges #1 and #2 that we addressed earlier. Such a design provides the following benefits:

- The design decision makes the BPF verifier a standalone application running on the host system, thus keeping the blocking problem caused by the verification away from Unikernel applications **(design challenge #1)**. Especially in practical cases, host systems contain more processor resources, which will minimize the impact on the applications.
- It ingeniously avoids the integration of heavy and redundant runtime libraries that serve only the verifiers from the application perspective.[1] This also reduces the effort required by developers to integrate the verifier with their BPF runtime system **(design challenge #2)**.

4.3 UniBPF Framework Library

In previous sections, we proposed a centralized BPF verifier decoupled from Unikernels. However, we realized that such an approach might hinder the practical usefulness of our verifier application, mainly due to the free design property of Unikernel applications as mentioned earlier **(design challenge #3)**.

The result of a centralized verifier without a standard specification is disastrous, and the problems come from the most customizable parts of the BPF language runtime: helper functions, key-value stores[2](maps), and program types.

On one side, forcing Unikernel developers to implement all of the above customizable parts with a fixed standard contradicts Unikernel's original aspiration to keep things simple. In addition, imposing too many restrictions will reduce our system's usability and potentially discourage other developers from integrating UniBPF. On the other hand, dropping verification support for the aforementioned BPF components renders our system meaningless.

Therefore, as a compromise, we require developers to syntactically disclose their design of BPF helper functions and program types. A brief introduction to the requirement and the information that UniBPF requires are listed below:

[1] Take our verifier application as an example. Our verifier application requires standard C++ libraries, but none of our applications (Nginx, Redis) need it.

[2] Not included in this thesis.

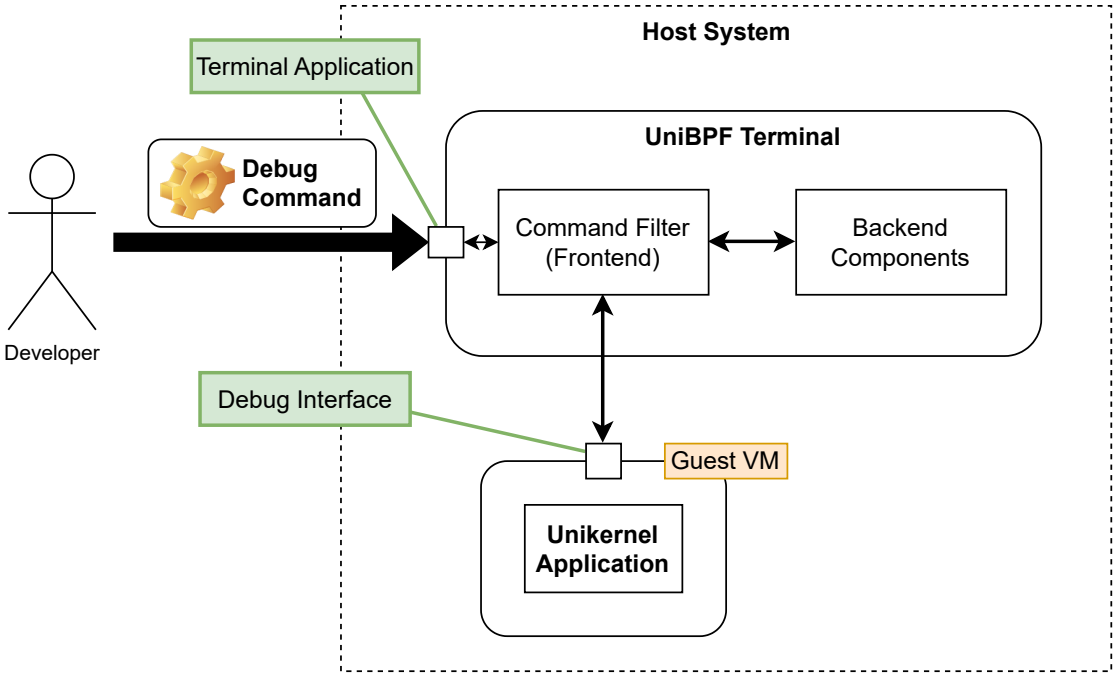

Figure 4.2: **Design of UniBPF Terminal**: UniBPF intercepts debug commands the developers made and initializes the BPF verification process once needed.

1. **Unified Syntaxes**: The syntax of the specifications exported from the Unikernel applications must follow the syntactic rules of UniBPF to be recognized by UniBPF.

2. **Required Information**: This includes the specification of **BPF helper functions** and the definition of **BPF program types**.

   On the **BPF helper functions**' side, the required information is listed below:

   a) Signature of the helper functions, including return types, number of arguments, and types of the arguments.

   b) (Optional) The ID of a BPF program type, indicating under which BPF program types the helper function can be used.

   c) The identity of the helper functions: each helper function must have a unique identification number.

   d) The name of the helper function for debugging purposes.

   On the **BPF program types**' side:

   a) Name of the program type, same, for debugging purposes.

   b) Metadata of the program type and the belonging BPF runtime context, including the size of the metadata data structure, whether such a program type is privileged, offset to the field contains runtime information, and the pointers to the data available for the BPF program.

   c) The identity of the BPF program type: Each program type must have a unique identification number.

As compensation, we provided a BPF framework library to help developers simplify the integration process when they introduce UniBPF into their existing projects, which are described in more detail in the *Implementation* chapter.

In addition, UniBPF asks the Unikernel application to export its **system information**. More specifically, the mount point information of the shared file system: Since the shared filesystem is mounted separately on the host and the guest Unikernel VM with different namespaces, details of the corresponding information are required so that UniBPF can translate between them in case of loading BPF programs for verification.

4.4 System Interface

Last but not least, UniBPF also defines two system interfaces to export the required information from the Unikernel application to the outside world. UniBPF expects these

system interfaces to be available as debug commands and to be accessible via the debug interfaces:

1. **bpf-helper-info**: This exports all BPF runtime system specifications, including BPF helper function information and BPF program type definitions.
2. **mount-info**: This exports information of the shared filesystem.

4.5 On-the-Fly Verifier Initialitzaiton

The last design decision to note is that the UniBPF Terminal has been developed as a versatile application allowing seamless switching between various Unikernel applications. Such switching is enabled via **on-the-fly initialization**: When the UniBPF Terminal is booted, it will remain as a white paper as a generalized verifier application. Only until it connects with a Unikernel application will it automatically query the system and BPF runtime system specifications using the system interface mentioned above. Then, it initializes its components, including the verifier, with the provided information and finally serves the debug console. Figure 4.3 shows a visualized workflow of this process.

In summary, such mentioned designs make UniBPF Terminal a centralized verification application, reducing software fragmentation and making it much more manageable **(design challenge 3)**. Meanwhile, the module design ensures that updating or upgrading the verification application does not affect the functionality of running applications at all and requires only simple integration tests. In addition, updating the verifier implementation doesn't require recompiling or redeploying the application, which is also a big plus.

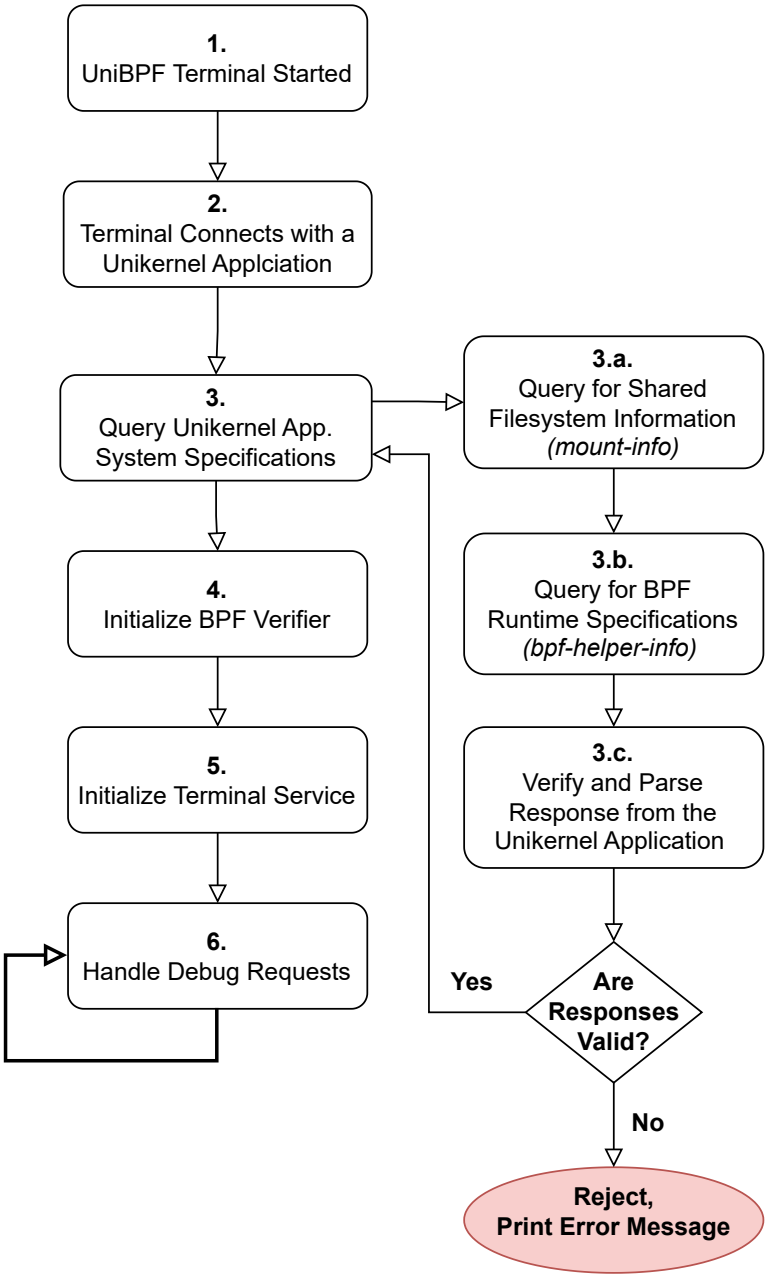

Figure 4.3: **The UniBPF *Terminal* Initialization Processes**: UniBPF initialize the BPF verifier on-the-fly with the specifications queried from the corresponding Unikernel application.

5 Implementation

In this chapter, we will present our experiences and a proposed approach for achieving a successful UniBPF implementation in a bottom-up manner. We will first outline the implementation of the UniBPF BPF framework library, such as the internal data structures and definitions. Simultaneously, we will show how the UniBPF framework library is designed to help developers integrate UniBPF into their system. Moreover, we will deliver the implementation of newly integrated debug commands that are used to export the Unikernel system and BPF runtime system specifications via the debug interface as required by UniBPF. In addition, we will show how the **UniBPF Terminal** is implemented to improve the usability of UniBPF. Finally, we will outline the process of enabling the BPF JiT compiler for Unikernel applications to reduce their BPF runtime overhead and address the challenges we encountered.

Our proof of concept is built based on **Unikraft** [Kue+21], a Unikernel framework designed to provide a Unikernel framework for applications deployed through virtual machine hypervisors. To enhance the extensibility of the Unikernels, we selected **xIO** [Mis23] as our preferred solution to provide a debug interface and to allow mounting of a shared filesystem on the Unikernel applications. In addition, we have utilized the BPF runtime solution proposed by Hendrychová et al. [Hen23], which includes not only an interpreted BPF runtime implementation but also a comprehensive kernel tracing solution with BPF programs.

For the BPF verifier, we have integrated **PREVAIL** [Ger+19]. And for the BPF JiT compiler, we used **uBPF** [Iov], which was also previously included in the BPF runtime solution proposed by Hendrychová et al. [Hen23].

The caveat is that we did not prove the correctness of the BPF JiT compiler in this work. However, related research can be found in other works, such as Jitterbug [Nel+20].

5.1 Framework Library

As mentioned in the previous chapters, our design requires detailed system and BPF runtime information to be exposed by the Unikernel applications to construct the customized BPF verifier for them on the fly. We define a set of interfaces and data structures in our framework library to regulate and support exporting this information. On the BPF runtime side, the required information includes the **function signatures**

(function name, returned data type, number of arguments, and data types of the arguments), the **unique ID** of the helper function, and the ID of the associated **program type**. On the program type side, a **unique ID**, the **program type's name**, associated **flags**, and **the BPF runtime context descriptor metadata** are required. The data structures of the above-required information can be found in the C program form in Figure 5.1 and Figure 5.2, respectively.

Similar to the Linux system, the upper limit of the number of BPF helper functions is limited to 2^{32} (Figure 5.1, line 4), although our underlying BPF runtime implementation currently supports only 64 helper functions; similarly, derived from the Linux system design, another limitation not listed in Figure 5.1 is that each helper function cannot take more than five arguments.

Finally, regarding the BPF runtime context descriptor metadata of BPF program types, more specifically, UniBPF requires Unikernel applications to provide the following four pieces of information:

1. `ctx_descriptor_struct_size`: The BPF runtime context descriptor size. This allows the verifier to ensure access to such context descriptors from the BPF program does not exceed the permitted range.

2. `offset_to_data_ptr`: In some cases, it is necessary to make data, e.g., TCP packets, accessible to BPF programs. This field indicates how to find the pointer in the context descriptor pointing to the data provided.

3. `offset_to_data_end_ptr`: This field indicates how to find the pointer in the context descriptor pointing to the above data's end.

4. `offset_to_ctx_metadata`: This field indicates how to find the metadata of the BPF runtime in the context descriptor.

5.1.1 Serializations

Figure 5.3 and Figure 5.4 show in a regular expression way how developers should serialize their BPF helper function definitions before exporting their BPF runtime information out of their systems.

From the BPF helper function specification side, when serializing them, the developer's goal should be to provide "`<HelperList>`" as suggested in Figure 5.3. `<HelperList>` shall contain one or more `<HelperDef>` separated by semicolons (`;`).

Among them, `<HelperDef>` defines a single BPF helper function. `<HelperDef>` contains the ID of the helper function (`<FuncID>`), the BPF program type (`<ProgType>`), the human-readable name of the helper function (`<FuncName>`), the data types of the

```c
// helper_function_list.h
//...

typedef unsigned int UK_UBPF_INDEX_t;

typedef struct HelperFunctionSignature {
    char *m_function_name;
    uk_ebpf_return_type_t m_return_type;
    UK_EBPF_HELPER_ARG_TYPE_NUM_t m_num_args;
    uk_ebpf_argument_type_t m_arg_types[];
} HelperFunctionSignature;

typedef struct HelperFunctionEntry {
    HelperFunctionEntry *m_next;
    const void *m_function_addr;

    // In below: UniBPF related fields
    UK_UBPF_INDEX_t m_index;

    // under which prog_type this helper function is available
    // UK_EBPF_PROG_TYPE_UNSPECIFIED for unspecified
    uint64_t m_prog_type_id;

    HelperFunctionSignature m_function_signature;
} HelperFunctionEntry;

//...
```

Figure 5.1: The internal BPF helper function related data structures defined by UniBPF in the provided framework library. These definitions provide all the BPF helper-function-related information that Unikernels must export when integrating with UniBPF.

```
1  // prog_type_list.h
2  typedef struct BpfProgType {
3      BpfProgType *m_next;
4
5      uint64_t prog_type_id;
6
7      bool privileged;
8
9      int ctx_descriptor_struct_size;
10     int offset_to_data_ptr;
11     int offset_to_data_end_ptr;
12     int offset_to_ctx_metadata;
13
14     char m_prog_type_name[];
15 } BpfProgType;
16
17 typedef struct BpfProgTypeList {
18     size_t m_length;
19     BpfProgType *m_head;
20     BpfProgType *m_tail;
21 } BpfProgTypeList;
```

Figure 5.2: The internal BPF program type related data structures defined by UniBPF in the provided framework library. These definitions provide all the BPF program-related information that Unikernels must export when integrating with UniBPF.

```
1  <HelperList> = <HelperDef>(;<HelperDef>)*
2  <HelperDef> = <FuncID>,<ProgType>:<FuncName>(<ArgTypes>)-><RetType>
3  <FuncID> = [0-9a-e]{1,8} // valid hex number
4  <ProgType> = [0-9a-e]{1,8} // valid hex number
5  <RetType> = [0-9a-e]{1,8} // valid hex number
6
7  <ArgTypes> = ϵ | <ArgeType>(,<ArgeType>)*
8  <ArgeType> = [0-9a-e]{1,8}
9  <FuncName> = [0-9a-zA-Z_-]{1,256}
```

Figure 5.3: Syntax rules for serializing BPF helper function specifications. Developers must export their BPF helper function definitions as described in this figure.

```
1  <ProgTypeList> = <ProgType>(;<ProgType>)*
2  <ProgType> = <ProgTypeID>,<ProgTypeName>:<FlagPrivileged>,
3      <SizeCtxDesc>,<OffsetDataPtr>,<OffsetDataPtrEnd>,<OffsetMetadata>
4
5  <ProgTypeID> = [0-9a-e]{1,8} // valid hex integer
6  <ProgTypeName> = [0-9a-zA-Z_-]{1,256}
7
8  <FlagPrivileged> = 0 | 1
9
10 <SizeCtxDesc> = [0-9a-e]{1,8} // valid hex integer
11 <OffsetDataPtr> = [0-9a-e]{1,8} // valid hex integer
12 <OffsetDataPtrEnd> = [0-9a-e]{1,8} // valid hex integer
13 <OffsetMetadata> = [0-9a-e]{1,8} // valid hex integer
```

Figure 5.4: Syntax rules for serializing BPF program type definitions. Developers must export their BPF program-type definitions as described in this figure.

```
1  HelperFunctionList *helper_function_list_init();
2
3  HelperFunctionEntry *helper_function_list_emplace_back(
4      HelperFunctionList *self, UK_UBPF_INDEX_t index,
5      uint64_t prog_type_id,
6      const char *functionName, const void *functionAddr,
7      uk_ebpf_return_type_t retType,
8      UK_EBPF_HELPER_ARG_TYPE_NUM_t arg_type_count,
9      const uk_ebpf_argument_type_t argTypes[]);
10
11 void helper_function_list_destroy(HelperFunctionList *self);
```

Figure 5.5: The UniBPF utility functions provided in UniBPF's BPF framework libraries assist in creating structured BPF helper function definitions.

```
1  // prog_type_list.h
2  //...
3
4  BpfProgTypeList *bpf_prog_type_list_init();
5
6  BpfProgType *bpf_prog_type_list_emplace_back(
7         BpfProgTypeList *self, uint64_t prog_type_id,
8         const char *prog_type_name, bool privileged,
9         int ctx_descriptor_struct_size, int offset_to_data_ptr,
10        int offset_to_data_end_ptr, int offset_to_ctx_metadata);
11
12 void bpf_prog_type_list_destroy(BpfProgTypeList *self);
```

Figure 5.6: The UniBPF utility functions provided in UniBPF's BPF framework libraries assist in creating structured BPF program type definitions.

arguments (<ArgTypes>) enclosed in parentheses, and finally an arrow followed by the return data type (<RetType>).

For the data type definitions, we referenced the design from PREVAIL [Ger+19]. However, due to namespace issues, we renamed them with UK_ (Unikernel) prefixes in our projects. The list of data type definitions can be found in PREVAIL's code base[1].

From the BPF program type definition side, developers are asked to specify their system's applicable BPF program types. When serializing the program type definitions, the developer's goal is to construct "<ProgTypeList>" as shown in Figure 5.4. Similarly, <ProgTypeList> contains one or more <ProgType> separated by semicolons (;).

Wherein <ProgType> defines a single BPF program type specification. <ProgType> includes the ID of the program type (<ProgTypeID>), the human-readable name of the program type (<ProgTypeName>), flags (e.g., <FlagPrivileged>) and the metadata of the BPF runtime descriptors (<SizeCtxDesc>, <OffsetDataPtr>, <OffsetDataPtrEnd> and <OffsetMetadata>).

The caveat is that in our implementation, those fields marked in Figure 5.3 and Figure 5.4 with *valid hex number* are required to contain only lower-case hexadecimal numbers, and they must fit into 32-bit integer numbers. Also, <FuncName> and <ProgTypeName> are restricted to consist only of numbers, English alphabets, low-dashes, minus signs, and to be composed with less or equal to 256 characters. The 256-character limitation on the names is only a preference in our implementation.

Integration

The framework library provides utility functions to efficiently integrate UniBPF into developers' existing applications. These help developers construct their BPF helper function specifications and BPF program type definitions systematically. These utility functions are listed in Figure 5.5 and Figure 5.6.

In real integration, developers are advised to first declare their program types in the following way:

```
1 // example pseudo-codes
2 // ...
3
4 BpfProgTypeList* g_bpf_prog_types = NULL;
5
6 // The init function:
```

[1] https://github.com/vbpf/ebpf-verifier/blob/ef234a6f25b7ca2c551b5792cd0ec4039b9457a7/src/ebpf_base.h

```
7  void init_bpf_specifications() {
8      g_bpf_prog_types = bpf_prog_type_list_init();
9      // ...
10
11     BpfProgType *prog_type_example =
12         bpf_prog_type_list_emplace_back(g_bpf_prog_types, counter++,
13                "executable", false,
14                sizeof(uk_bpf_type_executable_t),
15                offsetof(uk_bpf_type_executable_t, data),
16                offsetof(uk_bpf_type_executable_t, data_end),
17                offsetof(uk_bpf_type_executable_t, data_meta)
18         );
19     //...
20 }
21 //...
```

Whereby `uk_bpf_type_executable_t` is an example BPF runtime context descriptor provided by default in the framework library:

```
1  // uk_program_types.h
2  // ...
3
4  #define MAX_EXECUTABLE_CTX_SIZE 1024
5
6  typedef struct uk_bpf_type_executable {
7      char* data;
8      char* data_end;
9      uint64_t data_meta;
10
11     char storage[MAX_EXECUTABLE_CTX_SIZE -
12         sizeof(char*) - // data
13         sizeof(char*) - // data_end
14         sizeof(uint64_t) // data_meta
15     ];
16 } uk_bpf_type_executable_t;
17 //...
```

Regarding this BPF runtime context descriptor, take the default program type provided in the framework library as an example. Specified with such a program type, `uk_bpf_type_executable`, our system will make the arguments provided by the debug commands starting BPF programs as the readable data for BPF programs when the BPF language runtime is started:

```
1  // ubpf_runtime.c
2  //...
3
4  // In function: int bpf_exec(..., void *args,...)
5      uk_bpf_type_executable_t context;
6      context.data = context.storage;
7      context.data_meta = 0;
8
9      const size_t max_data_size = sizeof(context.storage);
10     const size_t data_size =
11         args_size > max_data_size ? max_data_size - 1 : args_size;
12
13     context.data_end = context.storage + data_size - 1;
14
15     strncpy(context.storage, args, data_size);
16     if (data_size == max_data_size - 1) {
17         context.storage[data_size] = '\0';
18     }
19
20 //...
```

By which, the `bpf_exec` function is the handler function that is called when the developer requests to run a BPF program on a Unikernel application by invoking the debug command `bpf_exec`[2][3] via the debug interface.

Last but not least, once required, the BPF program type definitions, `BpfProgTypeList`, can be freed with the provided library in the following way:

[2]Usage: `bpf_exec <bpf_filename> <bpf_function_name> [<bpf_program_argument>]`

[3]This command is slightly modified from the original version implemented by Hendrychová et .al [Hen23], where an extra argument `<bpf_function_name>` is now required to be provided since we extended BPF language runtime with support running one of a single BPF programs concentrated in the same file.

```
// example pseudo-codes
// ...

BpfProgTypeList* g_bpf_prog_types;

// The de-init function:
void free_bpf_specifications() {
    bpf_prog_type_list_destroy(g_bpf_prog_types);

    //...
}
//...
```

Using the previously defined program types, the next step is for developers to create their helper function specifications. The very first suggested step is to create a `HelperFunctionList` instance with a global variable using the utility function, `helper_function_list_init()`, then register the BPF helper functions with the utility function, `helper_function_list_emplace_back(...)`. For example, a BPF function with the signature `uint64_t bpf_map_get(uint64_t, uint64_t)` can be registered with the following code:

```
// ...

HelperFunctionList* g_bpf_helper_function = NULL;

// The init function:
void init_bpf_specifications() {
    g_bpf_helper_functions = helper_function_list_init();
    // ...

    uk_ebpf_argument_type_t args_bpf_map_get[] = {
        UK_EBPF_ARGUMENT_TYPE_ANYTHING,
        UK_EBPF_ARGUMENT_TYPE_ANYTHING,
    };

    helper_function_list_emplace_back(
```

```
16          g_bpf_helper_functions, 1, UK_EBPF_PROG_TYPE_UNSPECIFIED,
17          "bpf_map_get", bpf_map_get,
18          UK_EBPF_RETURN_TYPE_INTEGER,
19          sizeof(args_bpf_map_get) / sizeof(uk_ebpf_argument_type_t),
20          args_bpf_map_get);
21      //...
22  }
23  //...
```

From the above codes, we can also see that a BPF helper function can also be declared with `UK_EBPF_PROG_TYPE_UNSPECIFIED`, indicating that there is no specific program type restriction on using such a BPF helper function.

Also, once necessary, the BPF helper function specifications, `HelperFunctionList`, can be freed with the provided utility function in the following way:

```
1  // example pseudo-codes
2  // ...
3
4  HelperFunctionList* g_bpf_helper_function;
5
6  // The de-init function:
7  void free_bpf_specifications() {
8      helper_function_list_destroy(g_bpf_helper_function);
9
10     //...
11 }
12 //...
```

Finally, to serialize the BPF runtime specifications, one can utilize the utility functions `marshall_bpf_helper_definitions` and `marshall_bpf_prog_types` as shown in Figure 5.7.

Both utility functions take the data structures to be serialized and serialize them into strings that conform to the syntax standard listed in Figure 5.3 and Figure 5.4, then output the stage results with the provided callback function, i.e. "`append_result(...)`".

5.1.2 Deserializations

On the verifier application side, when it receives BPF runtime information, it first tries to use the utility functions `unmarshall_bpf_helper_definitions` to deserialize

```
1  // uk_bpf_helper_utils.h
2  // ...
3
4  void marshall_bpf_helper_definitions(HelperFunctionList *instance,
5                                       void (*append_result)(const char *));
6
7  // ...
8
9  void marshall_bpf_prog_types(BpfProgTypeList *instance,
10                                      void (*append_result)(const char *));
11
12 //...
```

Figure 5.7: The UniBPF utility function provided in UniBPF's BPF framework libraries for serializing BPF helper function and program type specifications into string forms.

```
1  // uk_bpf_helper_utils.h
2  // ...
3
4  HelperFunctionList *unmarshall_bpf_helper_definitions(
5      const char *input);
6
7  //...
8
9  HelperFunctionList *unmarshall_bpf_prog_types(
10     const char *input);
11
12 //...
```

Figure 5.8: The UniBPF utility functions provided in UniBPF's BPF framework libraries to deserialize BPF helper function specifications from UniBPF regulated string forms back to program data structures.

the BPF helper function specifications and `unmarshall_bpf_prog_types` to deserialize the BPF program type definitions back into the internal data structures, and later use these information to initialize the BPF verifiers. The definition of the mentioned utility functions can be found in Figure 5.8.

Internally, we utilize Deterministic Finite Automata (DFA) to decode those syntaxes.

5.2 Debug Commands

Two additional debug commands mentioned in the previous chapters have also been implemented. As mentioned, UniBPF defined these debug commands as a system interface to export the BPF runtime information required by UniBPF from the Unikernel applications to the outside world.

5.2.1 Export BPF runtime information

The debug command `bpf-helper-info` exports the BPF runtime information. When invoked, this command prints the serialized BPF helper specifications and BPF program type definitions in the format suggested below:

```
1 bpf-helper-info=<HelperList>\n
2 bpf-prog-type-info=<ProgTypeList>\n
```

Regarding `<HelperList>` and `<ProgTypeList>` please refer to Figure 5.3 and correspondingly to Figure 5.4.

5.2.2 Export System Information: Shared Filesystem

The debug command `mount-info` exports the shared filesystem mount point information of the Unikernel application. When invoked, this command prints the serialized information in the format suggested below:

```
1 mount-info=<Path_Host>:<MountPoint_Guest>\n
```

5.3 Host-Side BPF Verifier Application - UniBPF Terminal

UniBPF Terminal is the essence of the BPF verification methodology we introduced for Unikernels in this work. This application occupies the host system's processor

resources besides the Unikernel instances while processing BPF verification requests, preventing Unikernels from being blocked by such a time-consuming operation due to the single-threading features of the major Unikernel implementations.

In addition, UniBPF Terminal also provides a maintainer and developer-friendly feature: it is flexible and general enough to be used with any Unikernel instance that implements UniBPF, utilizing the framework implemented above. In practice, this is particularly useful since, instead of maintaining a handful of BPF verifier implementations, such a **one-for-all** design can minimize the effort required to update/upgrade the BPF verification processes as soon as a bug is discovered in some of the verification workflows.

In summary, UniBPF Terminal implemented the following features:

1. On-the-Fly Initializing: Using UniBPF framework libraries, UniBPF Terminal automatically initializes and adjusts its components on the fly when connecting to some Unikernel instances.

2. Command Filter/Interceptor: This component filters, and intercepts commands that require additional handling and invokes the appropriate components to handle them.

3. Replaceable Verifier Design: UniBPF Terminal is equipped with a BPF verifier. However, it is not hard coded. With its modular design, the BPF verifier implementation can be easily replaced in the future.

4. Miscellaneous: UniBPF Terminal provides other developer-friendly features, such as "built-in" functions that make it easy for developers to gather system information.

5.3.1 Usages

UniBPF Terminal supports the following options:

- `help` or `h`: This option is exclusive to any others. With this, UniBPF Terminal will print only the manual.

- `ushell <path>` or `u <path>`: This option tells UniBPF Terminal the path to the debug interface of the Unikenrel application to communicate with. If not specified, `<path>` is by default assigned with `/tmp/port0`.

5.3.2 On-the-Fly Initialization

As suggested in Figure 4.3, when UniBPF Terminal is started, it will make no assumptions about unknown information, including the BPF runtime specification and the system information of the Unikernels it will connect to. Only when it is connected to a Unikernel application will it start the whole initialization process, including automatic query for specifications, and then use the retrieved information to launch those components that need further information. The entire process is suggested below:

Dial the Debug Interface

As a solution based on xIO [Mis23], the debug interfaces of the Unikernels can be considered as a UNIX socket and can be connected in a standard way:

```
1  // UShellConsoleDevice.cpp
2
3  // 1. Create socket
4  sockaddr_un address{};
5  address.sun_family = AF_UNIX;
6  strcpy(address.sun_path, path.c_str());
7  socketFd = socket(AF_UNIX, SOCK_STREAM, 0);
8
9  // 2. Connect to the socket
10 if (!connect(socketFd, (sockaddr *)(&address), sizeof(address))) {
11     // throws error
12     // ...
13 }
```

Gather System Information

UniBPF Terminal queries system information with the debug commands mentioned earlier in Chapter 5.2, `bpf-helper-info` and `mount-info`, and then examines and deserializes the responses with the utility functions provided by the framework library. At the same time, the syntax correction can be checked if the return values of the unmarshal functions are `NULL`. In which case, UniBPF Terminal will throw an exception.

After these, UniBPF Terminal initializes its components, such as BPF verifiers, and then prompts the user to make further requests.

5.3.3 Intecept Debug Commands

While developers issue debug commands to the Unikernels via UniBPF Terminal, UniBPF Terminal listens and analyzes each request and takes immediate action when specific debug commands are issued.

Input

In the input direction, after receiving a debug command, UniBPF Terminal first checks whether the debug command is a built-in or a BPF-specific command. If they are not, the command is redirected to the debug interface; otherwise, the command is captured and explicitly handled.

For built-in commands, they are handled directly by UniBPF Terminal.

Currently, UniBPF Terminal is only integrated with one built-in command:

- **cwd**: This command returns the Current Working Directory (CWD) of the UniBPF Terminal.

The built-in `cwd` command is helpful because the current implementation assumes that UniBPF Terminal is started under the Unikernel application folder, and developers can use this built-in command to determine if they are starting UniBPF Terminal under the correct folder.

For BPF-specific commands, if there is a need to verify specific BPF programs, UniBPF Terminal will resolve the debug command's arguments, locate the BPF programs from the parameters, and invoke the BPF verifier to ensure their validity.

BPF verifications can fail in the following situations:

1. The BPF program file cannot be found on the shared file system. In this case, UniBPF Terminal will consider it immediately as an invalid request since, according to the prerequisite (Section 4.1), Unikernal applications are allowed to execute the BPF program files only if they are placed under the shared filesystem.

2. The specified BPF program fails to fulfill the security criteria demanded by UniBPF.

In both situations, UniBPF Terminal will throw an exception indicating failures. Otherwise, the command will be redirected to the debug interface as the next step.

Filesystem Namespace Translation

Due to the isolation created by hypervisors between host and guest systems, VM running Unikernel applications have their own independent file system namespaces.

However, because the UniBPF Terminal resides within the filesystem namespace of the host system and our implementation respects the namespaces of the Unikernel applications (i.e., developers issue debug commands directly after the Unikernels' filesystem namespaces), translations between the two namespaces are necessary when the UniBPF Terminal attempts to load the BPF program for verification.

We list the use cases that UniBPF Terminal may encounter in practice and also propose our implementation below:

1. Once the developer provides the BPF program pathname in the ***relative*** path manner. Then, no conversion will be made. The UniBPF verifier application will try to locate the BPF program directly with the relative pathname under the shared filesystem.

2. Once the developer provides the BPF program pathname in the ***absolute*** path manner. It first checks whether the provided pathname belongs to the shared filesystem, which can be achieved easily by comparing the pathname prefixes. Then, the UniBPF Terminal converts the given pathname into a relative pathname under the shared filesystem and then uses the converted relative one as mentioned above.

3. Otherwise, the UniBPF verifier application will assume that the BPF program cannot be located by the Unikernel application since it definitely cannot be found under the shared filesystem mounted by the application.

Output

The output direction is straightforward. It simply pipes the response from both the UniBPF terminal and the response from the debug interface made by the Unikernal application to standard output (stdout).

5.3.4 Replacable BPF Verifier

UniBPF Terminal flexibly integrates BPF verifiers. UniBPF Terminal defines verifier interfaces and relies solely on the interfaces. Such a design makes it possible and results in less effort if the developers want different BPF verifier implementations later on. The BPF verifier interface, as shown in figure 5.9, contains two APIs:

1. **Constructor**: The constructor initializes the actual BPF verifier instance with specified configurations, including the list of the helper functions provided and the program types defined by the Unikernel application.

```
1  // EBPFVerifier.hpp
2  //...
3
4  struct eBPFVerifyResult {
5      bool ok;
6      double took;
7      double memory;
8  };
9
10 class EBPFVerifier {
11 public:
12     explicit EBPFVerifier(
13             const ebpf_verifier_options_t verifierOptions,
14             const HelperFunctionList *helperFunctionList,
15             const BpfProgTypeList *progTypes);
16
17     struct eBPFVerifyResult verify(
18             const std::filesystem::path &bpfFile,
19             const std::string &desiredProgram);
20
21     // ...
22 };
```

Figure 5.9: The verifier interface of UniBPF Terminal. To integrate a different BPF verifier implementation to UniBPF Terminal, the developers are asked to fit their desired BPF verifier implementation into the provided interface as described in this figure.

2. **Method** `verify()`: The method loads, analyzes, and verifies the BPF program file with the given pathname and program type name specified in the arguments. As a result, the BPF verifier returns whether the BPF program is valid and the telemetries of the verification processes, including time and memory usage.

5.4 JiT Compilation

Last but not least, as an attempt to achieve better performance with the BPF programs, we extend the previous work [Hen23] by enabling JiT compilations.

Our implementation switches between JiT compiled mode and interpreted mode with a macro `UK_JITTED_BPF`. Internally, if the macro `UK_JITTED_BPF` is defined, indicating that the developers are willing to enable their BPF runtime environment in JiT compiled mode, the JiT compilation processes will be started inside the Unikernel application before the BPF program is started, which can be achieved using the utility function provided by uBPF [Iov] in the following way:

```
1 // ubpf_runtime.c
2 // ...
3
4 char* compileError;
5 ubpf_jit_fn jitted_bpf =
6         ubpf_compile(bpf_program_instance, &compileError)
7
8 // ...
```

In which `jitted_bpf` points to the memory address where the JiT compiled code is located. Later, execution of the JiT-compiled code with the given BPF runtime context can simply be achieved by:

```
1 // ubpf_runtime.c
2 // ...
3
4 jitted_bpf(&context, sizeof(context));
5
6 // ...
```

Data Execution Prevention

However, the above methods cannot enable a working JiT-compiled BPF runtime environment. In fact, if Unikraft is used as the Unikernel framework, executing the above JiT code will immediately run into boot loops due to an unexpected runtime software exception.

After deeper debugging, we found that the core result causing this is that the memory allocated for the JiT code is not set up with the correct permissions. More specifically, uBPF failed to mark the memory holding the jitted code as **executable**. Actually, uBPF did allocate memory in a pretty standard way using the standard C library API `calloc` and tried to fix the permission with a standard UNIX system API `mprotect`. However, it did not work. The reason is straightforward: the system API `mprotect` was not implemented by Unikraft.

As a result, we have to rely on low-level Unikraft system functions to solve these problems:

```
1  // ubpf_runtime.c
2  // ...
3
4  struct uk_pagetable *page_table = ukplat_pt_get_active();
5  unsigned long pages = size_to_num_pages(vm->jitted_size);
6  int set_page_attr_result = ukplat_page_set_attr(page_table,
7      (__vaddr_t)jitted_bpf, pages,
8      PAGE_ATTR_PROT_READ | PAGE_ATTR_PROT_WRITE | PAGE_ATTR_PROT_EXEC,
9      0);
10
11 // ...
```

As shown in the code above, reconfiguring the permissions of the JiT-compiled code to read/write/executable is achieved by first retrieving the page table in line 4. Then, as shown in line 5, using the utility function provided by Unikraft, one can calculate the number of memory pages to update permissions with. Finally, the reconfiguration of memory page permissions can be accomplished using the Unikraft system function `ukplat_page_set_attr`, as suggested in line 6.

6 Evaluation

Lastly, we would like to evaluate whether our designs meet our design goal, including:

1. **Security**: We will evaluate whether the promised strong security can be achieved by the design of UniBPF. The evaluation will focus on two critical areas: **architectural security** and **runtime security**. Architectural security tests will determine if the verifier can successfully reject BPF programs built against architectural conventions, such as reading undefined values and incompatible operand types. Runtime security tests will determine if the verifier can identify BPF programs that could cause problems at runtime, such as zero division or programs that do not terminate.

2. **Performance**: To the next, we will test whether UniBPF allows a more efficient BPF language runtime in Unikernel applications by enabling JiT compilation, and we will also evaluate the overheads brought by the corresponding demanded verification and compilation.

6.1 Experiment Environments

For the following sections, we ran our evaluations on a server with a 12-physical-core Intel Xeon Gold 5317 CPU @ 3.00 GHz and 256 GiB of DDR4 memory. The operating system is NixOS 22.11, a distribution based on Linux 6.2.12. The hypervisor for the Unikernel application is KVM-accelerated QEMU 7.2.0.

For better reproducibility and to avoid inaccurate results due to performance loss caused by cache misses and other factors, we also disabled Intel Turbo Boost and ensured the affinity of the applications to the very first core (if not specified explicitly).

6.2 Security Analysis

Of all factors, we are interested in the security measurements the verifiers promise to provide the most. As the prerequisite to utilizing jitted BPF native code, we believe security checks are incredibly essential. One reason is: unlike the interpretation mode used in previous work, the native code generated by the JiT compiler does not provide

any security measurements, such as sandbox escape prevention. Furthermore, our previous research suggests that detecting runtime errors caused by insufficient code quality still remains challenging even when interpreters are used. So we asked ourselves, does UniBPF improve system security and robustness, as we tend to do? To prove its capability, we evaluated our system with some malicious/problematic programs, each containing some obvious implementation issues that can/might cause security problems at runtime.

In the following sections, we will first explain the target subsystem and the corresponding security dilemma to show that such a dilemma can be solved by integrating UniBPF. First, acting as a control group, we will execute the aforementioned malicious/problematic programs under a Unikernel application with only a BPF interpreter serving as the BPF runtime environment. Then, for comparison, we will run identical programs under JiT compiled mode twice, once with UniBPF protection and once without.

For the following experiments, unless noted explicitly, we will initialize the BPF verifier with its default configurations[1], where program termination is not checked, and division by zero does not cause an error.

For a better understanding of our work, evaluation results are summarized in the table below:

Program Name	*Result - Interpreter*	*Result - JiT*	*Result* - UniBPF
OOB[1]	Terminated[2]	Exploited[3]	Denied[4]
OOB + Nullptr[5]	Terminated	System crashed	Denied
Infinity Loop	System freezes	System freezes	Denied
Dvision by Zero	Error Ignored[6]	Error Ignored	Partially Denied
Instruction Type Safty	Error Ignored	Error Ignored	Denied
Program Type Safty	Error Ignored	Error Ignored	Denied
Helper Type Safty	Error Ignored	Error Ignored	Denied

[1] Out of bound memory access.
[2] The program was executed but exited early due to dangerous operations being detected.
[3] The sensitive kernel information was leaked.
[4] The System refused to execute the program.
[5] Out of bound memory access with a NULL pointer.
[6] The System tolerances the existence of the error-prone operation.

[1] `https://github.com/vbpf/ebpf-verifier/blob/ef234a6f25b7ca2c551b5792cd0ec4039b9457a7/src/config.cpp`

6.2.1 Memory Security

First, to prevent leakage of sensitive kernel information, we would like to know if UniBPF helps filter out those malicious BPF programs that "accidentally" access the data outside the allowed BPF runtime memory space. Second, we would like to know if UniBPF also helps to avoid critical memory-related runtime errors that are forbidden in most operating systems — for example, null-pointer memory access, which is also taboo in our Unikernel applications.

So we ask: ***Can* UniBPF *actively deny those BPF programs with sandbox escape intent or those with potentially unsafe memory operations?***

Assessments

To evaluate the effectiveness of UniBPF in solving the problem, we created a BPF program, **OOB**. The program aims to retrieve the "flag" by accessing a variable, `the_flag`, which lives in an example Unikernel application with a hexadecimal value of 0x42. The program's source code of **OOB** can be found in Figure 6.1.

To identify the flag variable, **OOB** uses the symbol name provided in the BPF runtime context. The symbol address is retrieved using the BPF helper function `bpf_get_addr`. Upon receiving the returned symbol address, **OOB** attempts to obtain the corresponding stored value in an unsafe C programming language manner.

The helper function `bpf_get_addr` returns the address of the given symbol once it exists in the Unikernel system; otherwise, it produces a null pointer.

In the following experiments, in the first round, we will make sure that the symbol name provided to the BPF program is valid so that if no security measure is provided, the "flag" will be captured by the test program. In the second attempt, we will test the BPF program with a symbol name that does not exist in the system, using the BPF helper function `bpf_get_addr` to simulate a memory access with a null pointer.

Results - Baseline

Our control group setup behaves as expected. The interpreter successfully refused to follow the invalid instruction in both programs with the following error messages:

```
1 uBPF error: out of bounds memory load at PC 2, addr 0x0, size 4
2 mem 0x41f05fa60/zd stack 0x400/520484720
```

```
1  __attribute__((section("executable"), used))
2  int oob(uk_bpf_type_executable_t* context) {
3          if (context->data_end - context->data < 2) {
4                  return -1;
5          }
6
7          __u64 flag_address = bpf_get_addr(context->data);
8
9          return *((int*)flag_address);
10 }
```

Figure 6.1: **oob.c**: The BPF program tries to capture "the_flag" with an unsafe direct memory access.

Results - JiT Compiled Mode w/o UniBPF

Running the first malicious programs under JiT compiled mode without the protection of UniBPF results in the flag being captured. The program returns the following responses, where 66 is exactly the decimal form of the hexadecimal number 0x42:

```
1  > load symbol.txt
2  Load 2921 symbols
3  > bpf_exec bpf-security/oob.o oob the_flag
4  BPF program compile took: 9782206 ns
5  BPF program returned: 66. Took: 13484 ns
```

In a real-world situation, this result reflects that the system is being exploited with arbitrary reads and writes to kernel memory, leading to further security crises such as privilege escalation and sensitive data leakage.

In the second attempt, the Unikernel application crashed with a page fault, indicating an invalid memory access with a NULL pointer:

```
1  > bpf_exec bpf-security/null_exception.o null_exception Abracadabra
2  //...
3  [ 412.214412] CRIT: [libkvmplat] <traps.c @ 100> page fault
```

```
4     at 0000000000000000, error_code=0000000000000000
```

In a real-world scenario, such a vulnerability will likely lead to unstable service quality and possibly to Denial-of-Service (DoS) attacks.

Results - BPF Runtime with UniBPF

Fortunately, in both cases, the BPF verifier prohibits the execution of BPF programs before any part of them is executed. The BPF verifier detects potential hazards using different methods than the interpreter, as evidenced by the error messages presented below. In the case of the interpreter, unsafe operations are detected by on-the-fly memory access boundary checks at runtime. However, in the case of our integrated BPF verifier, PREVAIL, detection is achieved by checking whether the preceding program statement can affirm the validity of the operation **at the BPF language level**. In this case, a memory access instruction's validity can only be accepted if the operand register was initialized with a potential pointer value earlier.

Although in the provided C language source code (Figure 6.1), we cast the variable into a pointer before dereferencing it, this is not sufficient after the program is translated into BPF instructions, which is a type-less programming language.

In this case, the BPF verifier observed that the program attempts to access memory by dereferencing an integer number returned by a BPF helper function. As a result, the BPF verifier refuses to accept the program.

```
1  Only pointers can be dereferenced
2                       (valid_access(r0.offset, width=4) for read)
```

Subsummary

UniBPF guarantees the same memory safety as interpreters and effectively protects the runtime system from invalid/insecure memory operations in JiT compiled mode.

6.2.2 Termination of Program

It is a significant problem in the production environment when the program gets stuck in infinite loops. This problem may not only slow down the entire application but may also make services become unreachable. The problem is particularly severe in an operating system kernel, where many critical operations can be blocked. So we ask: ***Can* UniBPF *beforehand ensure the BPF program's termination and thereby prevent the runtime system from being jammed?***

```c
#include "bpf_helpers.h"

__attribute__((section("executable"), used))
int infinity_loop(uk_bpf_type_executable_t* context) {

    asm volatile("r2 = 0x42");

    for(unsigned int index = 0; index <= 0xffffffff; index++) {
        asm volatile("r2 /= 1");
    }

    return 0;
}
```

Figure 6.2: **infinity_loop.c**: This BPF program runs into infinite loops due to an implementation error that causes an integer overflow.

In the following tests, we enabled an additional feature for our BPF verifier to check for program termination.

Assessments

This section will examine the subsystems with the program shown in figure 6.2. The program simulates an infinite loop caused by an integer overflow that a developer accidentally made. The volatile raw assembler code in the program is for the sole purpose of avoiding optimization of the loop during compilation and attempting to evade infinite loop checks performed by the runtime system.

Results - BPF Runtimes w/o UniBPF

Unfortunately, the interpreter did not live up to the promise of being able to detect infinite loops, as claimed in the previous work [Hen23]. Both the BPF runtime, either in interpretation mode or in JiT compiled mode, get stuck in an endless loop when running this program. Since the kernel of our Unikenrel application does not preempt or implement any other time-sharing mechanisms, the entire application becomes irresponsible to any service request.

Results - BPF Runtime with UniBPF

The BPF verifier successfully blocked the given test program. However, this conclusion was not purely due to the verifier's ability to detect non-terminating programs, as further investigation revealed that the verifier rejects all BPF programs with loops, regardless of whether they are constant-bounded.

We do not consider this an expected phenomenon since even the Linux eBPF verifier allows constant-bounded loops since version **5.3** [SK21]. With the additional BPF helper function `bpf_loop` introduced in version **5.17** [BS19] it is now possible to pass the verifier in the Linux eBPF system even with BPF programs containing inconstant-bounded loops (i.e. the number of iterations is decided at runtime), which allow a maximum of about eight million loop iterations [CK22].

Subsummary

UniBPF outperforms interpreters and prevents runtime systems from being blocked by those BPF programs containing infinity loops, albeit in an unexpected way.

6.3 Runtime Error Prevention

Depending on the quality of the code, runtime errors can inevitably lead to severe security crises, from crashing the entire system to economic losses due to faulty business logic. In this section, we will evaluate UniBPF with one of the most common runtime errors: division by zero.

6.3.1 Division by Zero

Division by zero throws a runtime exception in most runtime systems and platforms, including Java, Python, x86, and ARM architecture. If the exception is not handled correctly, the system will crash. Inspired by the Linux BPF specification [com], the primary BPF runtime implements division-by-zero with the resulting setting of zero in the target register. The BPF runtime included in our sample Unikernel application behaves similarly. We asked ourselves, is this expected behavior? We believe that a better way to maximize runtime security and integrity is to detect and block potential runtime errors in advance at the verification stage. Therefore, we ask: ***Can* UniBPF *reject the BPF programs with potential division-by-zero to prevent inconsistent business logic?***

Assessments

Figure 6.3 shows the assembler source code of the two test programs. Both BPF programs try to divide 0x42 placed in the **return value register**, `r0`, with zero. Unlike the first one, the second one puts the zero divisor inside a register.

```
1  # div_by_zero1.o
2  r0 = 0x42
3  r1 = 0
4  r0 /= 0
5  exit
6
7  # div_by_zero2.o
8  r0 = 0x42
9  r1 = 0
10 r0 /= r1
11 exit
```

Figure 6.3: The test BPF programs used to evaluate whether UniBPF can deny the BPF programs contain runtime errors, such as division by zero.

Results - BPF Runtime w/o UniBPF

Utilizing the BPF interpreter as the control group, both BPF programs return zero, as specified in the Linux BPF specification, as shown below:

```
1 > bpf_exec bpf-security/div_by_zero1.o div_by_zero1
2 Using BPF Interpreter
3 BPF program returned: 0. Took: 4506 ns
4
5 > bpf_exec bpf-security/div_by_zero2.o div_by_zero2
6 Using BPF Interpreter
7 BPF program returned: 0. Took: 701 ns
```

Similar to the results with the interpreter, the JiT compiled platform native code results also in the same result:

```
1 > bpf_exec bpf-security/div_by_zero1.o div_by_zero1
2 BPF program compile took: 77893 ns
3 BPF program returned: 0. Took: 117 ns
4
5 > bpf_exec bpf-security/div_by_zero2.o div_by_zero2
6 BPF program compile took: 9849167 ns
7 BPF program returned: 0. Took: 113 ns
```

Results - BPF Runtime with UniBPF

For UniBPF, the results are different. For the first program, where the constant 0x42 is divided by a constant zero, the BPF verifier does not react at all, i.e., the BPF program passes the verification. For the second one, it fails to pass the BPF verifier with the error message shown below:

```
1 entry: Possible division by zero (r1 != 0)
2
3 Program terminates within 8 instructions
4  (took: 0.000527 seconds)
5 ERROR: Verification failed for the section: executable
6 OK: 0; Failed: 1
```

The result contradicts our intuition to expect both programs to be denied. We conducted further investigation into the BPF verifier implementation, and we found that this is purely a design preference of the verifier's development team. According to the PREVAIL contributors' response to a GitHub issue ticket [TGN20], the system is designed not to react when it encounters division-by-zero with a constant denominator. The reasoning behind this decision is that if division-by-zero is done so obviously, PREVAIL developers believe it should be assumed that the developer intends it that way. To comply with the Linux BPF specification and to avoid rejecting existing valid BPF programs, the verifier is, in the end, implemented to check only for potential division-by-zero occurrences that could happen unexpectedly at runtime.

Subsummary

UniBPF can filter out those BPF programs with potential division-by-zero errors. However, it will be ignored if the error can be confirmed statically.

6.3.2 Type-Safe Codings

The BPF verifiers, including those in Linux and UniBPF, prioritize type safety as an essential criterion for verifying BPF programs, as type inconsistencies can lead to unforeseen problems in the runtime environment. For example, the operand register in a memory load instruction must contain a *pointer* value or a value legally derived from another *pointer* value. The same rule also applies to the *compare* instructions, which specify that both variables to be compared must have identical or compatible data types. Thus we ask: ***Can* UniBPF *prevent those BPF programs with inconsistent data types from being executed?***

Assessments

To determine whether our system can withstand *unsafe* code focusing mainly on type safety, we have developed three BPF programs.

The very first BPF program, **TypeSafty** (see figure 6.4), simulates a program with instructions consisting of incompatible operand types. This is done by introducing a multiply instruction on line 7, which is forcibly inserted just before a subsequent compare instruction.

The `bpf_get_addr` shown on line 8 is a BPF helper function that expects a pointer pointing to memory areas that the BPF program is allowed to read from. In summary, the given program will use the BPF helper `bpf_probe_read` shown on line 10 to read the memory data at a given address with a given length according to the name of the symbol provided in the BPF runtime context.

Next, to evaluate the functionality of the newly introduced Linux-like BPF program types in UniBPF, we also implemented a test BPF program, **InvalidContext** (see Figure 6.5). As mentioned in the previous chapters, the BPF runtime system passes the runtime context to the BPF program as an argument when the BPF program is started, so misusing the BPF runtime context descriptor with a mismatched program type will also result in undefined behaviors. Typically, the context descriptor contains runtime information and pointers pointing to the data, such as network packets, stored in a read-only *packet* region [Ger+19].

Although the program, **InvalidContext**, declared itself as a `no_context` program on line 1, it intentionally uses the runtime context descriptor `uk_bpf_type_executable_t`, which is only valid with the program type `executable`.

Finally, the third program, **HelperSignature** (Figure 6.6), misuses the BPF function `bpf_get_addr` on Line 4 by providing an argument with an integer number "0", which should be forbidden due to the mismatch data type applied.

```
1  __attribute__((section("executable"), used))
2  __u64 type_safty(uk_bpf_type_executable_t* context) {
3      if (context->data_end - context->data < 2) {
4          return -1;
5      }
6
7      asm volatile("r1 *= 1");
8      __u64 pointer = bpf_get_addr(context->data);
9
10     return bpf_probe_read(pointer, 8);
11 }
```

Figure 6.4: **type_safty.c**: The register `r1` in this BPF program, **TypeSafty**, is forced to be multiplied by an integer, which, from the BPF verifier's point of view, has already been converted to an integer.

```
1  __attribute__((section("no_context"), used))
2  __u64 invalid_context(uk_bpf_type_executable_t* context) {
3
4      if (context->data_end - context->data < 1) {
5          return -1;
6      }
7
8      return *context->data;
9  }
```

Figure 6.5: **invalid_context.c**: This BPF program, **InvalidContext**, declares the use of an incorrect BPF runtime context descriptor according to the declared program type *no_context*.

```
1 __attribute__((section("executable"), used))
2 char helper_signature(void* dont_care) {
3
4     bpf_get_addr(0);
5
6     return 0;
7 }
```

Figure 6.6: **helper_signature.c**: The BPF program **HelperSignature** calls the BPF helper function with an integer instead of using a valid pointer to a readable region of memory, as required.

Results - Baseline

As a baseline result, we first ran all of the above BPF programs, **TypeSafety**, **InvalidContext**, and **HelperSignature**, with a Unikerenal application using a BPF interpreter as the BPF runtime, but without UniBPF protection.

For the BPF program **TypeSafety**, again using *the_flag* as the parameter, the program results in returning the value 0x42 (66 in decimal form), indicating the interpreter's ignorance in dereferencing addresses from a register potentially not containing valid pointers:

```
1 > load symbol.txt
2 Load 2921 symbols
3 > bpf_exec bpf-security/type_safty.o type_safty the_flag
4 Using BPF Interpreter
5 BPF program returned: 66. Took: 19532 ns
```

InvalidContext is also executed without any error in this case since our system implementation constantly feeds BPF programs with the user-provided arguments as the BPF runtime context by default, even when the program is tagged with an invalid program type. Once the system is implemented in another way, whereas a NULL pointer is provided as the BPF runtime context upon invalid program type tagged, the execution of **InvalidContext** under interpreter mode shall end up with an *out of memory boundary load* exceptions, due to a NULL pointer dereferencing.

In the case of **HelperSignature**, unfortunately, even the on-the-fly memory boundary checker of the BPF interpreter cannot help: the Unikernel application crashed because

the BPF program called the helper function `bpf_get_addr` with a zero, a.k.a. a NULL pointer, and got trapped inside the implementation of the helper function by its security measures:

```
1  [ 1676.140104] CRIT: [libushell] <loader.c @ 316>
2                                          Assertion failure: symbol
3  [ 1676.146070] Info: [libkvmplat] <shutdown.c @ 35> Unikraft halted
```

Results - JiT Compiled Mode w/o UniBPF

For the JiT compiled mode, the BPF runtime behaves identically to the interpreter results when UniBPF protection is absent.

Results - BPF Runtime with UniBPF

By using UniBPF, we observed that all three programs with unsafe operations cannot pass the verifier.

TypeSafty failed to pass the verifier with an error message indicating that the argument provided calling the BPF helper function on Line 8 is a number instead of the requested pointer. This is because the integrated BPF verifier always treats the result of multiplication and division as a number. Thus, although the correct property was owned before Line 7, as soon as the register was passed as an argument to the helper function on Line 8, it no longer holds the valid data type as a pointer from the BPF verifier's point of view:

```
1  Invalid type (valid_access(r1.offset, width=r2) for read)
```

Similar to **InvalidContext**, the verification of **InvalidContext** failed because the BPF verifier does not know the BPF runtime context information for the given non-defined program type. The following error message indicates that the verifier asserts that the specified address shall not be read:

```
1  Only pointers can be dereferenced
2                       (valid_access(r2.offset, width=1) for read)
```

Lastly, also, **HelperSignature** is rejected by the BPF verifier, which successfully protects the Unikenrel application from crashing. The error message for the rejection is shown below:

```
1 Only pointers can be dereferenced
2                       (valid_access(r1.offset, width=r2) for read)
```

The error message clearly states that passing a number as an argument to the BPF helper function is not allowed, whereas a valid pointer is requested.

Subsummary

UniBPF can reject those BPF programs with inconsistent data types, preventing potential runtime hazards.

6.4 Verification Overhead

Next, to understand how the overhead of the newly added BPF verifier component affects the system in terms of time and memory usage, we created three BPF programs: **Nop**, **Hash**, and **Adds** to measure it. Figure 6.7, Figure 6.8, and Figure 6.9 show the source code of these programs.

Among these test programs, **Nop** contains only two BPF instructions with which we want to evaluate the lower bound of the overhead caused by the verifier; **Hash** (26 instructions) contains a `if` and a `else if` clause enclosed in a `for` loop, and we want to measure the latency and memory overhead caused by the verifier in the case of nested program logic; the rest, **Adds** (1002 instructions), consists of a thousand contigious instructions, which we use to evaluate the throughput of UniBPF's verifier.

```
1 __attribute__((section("executable"), used))
2 __u64 nop(uk_bpf_type_executable_t* context) {
3 {
4         return 0;
5 }
```

Figure 6.7: **nop.c**: A BPF program that consists of only two instructions that set the return value register to zero and then eventually return.

```
1  __attribute__((section("executable"), used))
2  __u64 hash(uk_bpf_type_executable_t* context) {
3
4      __u64 sum = 0;
5
6      for(int index = 0; index < 256; index++) {
7          char* input = context->data + index;
8          if(input >= context->data_end) {
9              break;
10         }
11
12         char to_add = *input;
13
14         if(to_add >= 'A' && to_add <= 'Z') {
15             to_add += 'A' - 'a';
16         } else if(to_add >= '0' && to_add <= '9') {
17             to_add -= '0';
18         }
19
20         sum += to_add;
21     }
22
23     return sum;
24 }
```

Figure 6.8: **hash.c**: This BPF program reads the input string from the provided BPF context and returns the Hash of the input string with its own business logic.

```
   1  r0 = 0
   2  r0 += 1
      # ... r0 += 1 * 998
1001  r0 += 1
1002  exit
```

Figure 6.9: **adds.c**: The assembler of the BPF program, which contains only simple business logic: it adds the register *r0* to a thousand and then returns it.

Observation

On average, the BPF program **Nop** took the verifier 8.82 milliseconds to verify and used 3328 kb of memory; the BPF program **Hash** took 12.05 milliseconds to verify, and we observed an increased memory usage of 4096 kb; for **Adds**, it took about 43.60 milliseconds to verify and the memory usage climbed accordingly to 5056 kb.

The caveat is that we measured the memory overhead by querying the Resident Set Size (RSS)[2] of a newly started UniBPF Terminal after verifying and executing each tested BPF program. This means the measured memory usage is only an estimated result.

We conclude that the lower bound of the verification overhead with UniBPF is slightly below 8.82 milliseconds and 3328 kb of rams. We also observe from the BPF programs **Hash** and **Adds** that programs with control branches require significantly more time to verify: the throughput of the verifier reaches 28.75 instructions/second for the BPF program **Adds**, but only 7.43 instructions/second for **Hash**.

Similarly, from a memory efficiency perspective, we found that the BPF programs with control branches may need more memory to verify in case of the same program scales.

6.5 JiT Execution Performance

6.5.1 Overhead

From the BPF programs **Nop**, **Hash**, and **Adds** given above, we observed 9.74 milliseconds, 9.79 milliseconds, and correspondingly 9.85 milliseconds of one-time JiT compilation overhead. Similarly, we observed a lower throughput in the case of the JiT

[2]In Linux systems, RSS represents the memory **allocated** in RAM.

compiling BPF programs with control branches.

6.5.2 Micro-Benchmark

To thoroughly understand the runtime performance improvement that JiT compilation can potentially bring, we decided to precisely measure the performance of commonly used BPF instructions at the instruction level, both in JiT compiled mode and in the interpreted mode. When studying such performance at this level, people usually present the results in terms of Instructions per Cycle (IPC). However, since there is no actual processor in the BPF architecture, we cannot easily define the cycles. So we decided to measure *Instruction per Time* (IPT), i.e., how long each instruction takes on average.

In the following sections, we will micro-benchmark some of the **arithmetic** and **memory access** instructions that we believe are most commonly used in practice.

To evaluate the duration of each instruction per runtime environment, we implemented evaluation BPF programs that wrap the BPF instructions under test in a for-loop consisting of one million iterations. By subtracting the time consumed by the control group program, which does not include any instruction under test in the for-loop, we can obtain the total execution time of the tested instructions repeated a million times. Another point to note is that at the beginning of each evaluation program, the BPF architecture register, the *r2* register, is initialized with designated values once at the beginning of each evaluation program. The *r2* register is reserved exclusively for instructions under tests that require a register as a target operand.

A template of the evaluation programs represented at the instruction level in the LLVM [LA04] style BPF assembler can be found in Figure 6.10.

In all of our evaluations, we will carry out ten runs of the evaluation programs and use the resulting averages as our final results.

Control Group

Firstly, we assessed the program's duration by utilizing a control group that did not include any instructions under test in the loop body.

As a result, we found that the interpreter takes an average of 6.054 milliseconds to complete one million iterations, while the JiT-compiled program takes an average of 0.670 milliseconds. According to the statistics, we found the JiT-compiled program is more than nine times faster than the interpreted one with our control group program.

After analyzing the disassembled interpreter and the JiT-compiled BPF program, we discovered that the significant speed-up results from the reduced time required by the jumping tables used by the interpreter to decode instructions.

```
1  // reserved for instructions require registers as operands
2  r2 = ???
3
4  r3 = 1000000
5
6  LOOP_BODY:
7  // The instruction under test
8
9  r3 += -1
10 if r3 == 0 goto END
11 goto LOOP_BODY
12
13 END:
14 r0 = 0 // return 0
15 exit
```

Figure 6.10: The template for the evaluation program. In subsequent evaluations, we insert the statement to be measured on line 7, subtract the time taken by the loop statements, then the time taken per statement can be determined.

Program	*r2*	*Instruction Under Test*
ADD	1000000	r2 += 1
SUB	1000000	r2 -= 1
DIV	1000000	r2 /= 1
MOD	1000000	r2 %= 1
OR	1	r2 |= 0
AND	1	r2 &= 1
XOR	1	r2 ≙ 0
MOV	0	r2 = 1
LSH	0	r2 «= 1
RSH	0	r2 »= 1
NEG	1	r2 = -r2
LOAD_8	0	r2 = *(u8 *)(r1 + 0)
LOAD_16	0	r2 = *(u16 *)(r1 + 0)
LOAD_32	0	r2 = *(u32 *)(r1 + 0)
LOAD_64	0	r2 = *(u64 *)(r1 + 0)
STORE_8	0	*(u8 *)(r1 + 0) = r2
STORE_16	0	*(u16 *)(r1 + 0) = r2
STORE_32	0	*(u32 *)(r1 + 0) = r2
STORE_64	0	*(u64 *)(r1 + 0) = r2

Table 6.1: The table lists the values of the *r2* register and the corresponding instruction under tests in each evaluation BPF program.

Assesments

We measured the *IPT* of fourteen BPF instructions under both JiT compiled mode and interpretation mode. The tested instructions are as follows:

- **Integer arithmetic instructions**: *ADD* (addition), *SUB* (subtraction), *MUL* (multiplication), *DIV* (division) and *MOD* (modulo).
- **Boolean instructions**: *OR* (or), *AND* (and) and *XOR* (xor) operations.
- **Miscellaneous**: *MOV* (assign) operation.
- **Bitwise operation instructions**: *LSH* (bitwise left-shift), *RSH* (bitwise right-shift) and *NEG* (boolean negation) operations.
- **Memory access instructions**: *LOAD* and *STORE*.

The initial value of register *r2* and the corresponding instruction under tests can be found in Table 6.1.

As a result, the *IPT* of the instructions mentioned above can be found in figure 6.11.

Integer Arithmetic Instructions

Based on our findings, it has been determined that the *ADD* and *SUB* instructions are approximately six times more efficient in JiT compiled mode compared to interpretation mode. However, the *MUL* instruction only experiences a speed increase of approximately 119%.

Contrary to expectation, the *DIV* and *MOD* instructions run about 4.5 times **slower** in JiT-compiled mode than in interpreter mode. After reverse-engineering the JiT-compiled BPF program and the interpreter (uBPF), we discovered that the cause could be lies in the implementation of the JiT compiler: The lack of optimization in the JiT-compiled codes causes the *DIV* and *MOD* instructions to waste significant amounts of time in context saving, such as preparing operand registers and saving arithmetic flags; similarly for the *MUL* instruction.

To confirm this issue, we took a close look at the JiT-compiled evaluation program with the *ADD* instruction, as well as with *MUL*, *DIV*, and *MOD*. From the Figure 6.12, it can be seen that the *ADD* statement yields only a single x86 instruction. However, in the case of *MUL* (Figure 6.13), it leads to a significant increase in the number of translated instructions to a total of 8. For the *DIV* and *MOD* instructions, see Figure 6.14 for their disassembled code, which produces even 16 translated instructions. For comparison, we also disassembled the interpreter in the part that handles *DIV* and *MOD* instructions (see figure 6.15). It is easy to see why the *DIV* and *MOD* BPF instructions are so slow in JiT compiled mode: both BPF instructions not only produce more platform native instructions but also use instructions that produce higher latencies; more specifically, the `pushfq` and `popfq` instructions generated by the JiT compiler produce 4 and 9 μOps with the CPU of our runtime platform [Fog22], whereas the instructions used by the interpreters (except the `div` instruction) produce only 2 μOps at most.

Boolean and Miscellaneous Instructions

The *OR*, *AND*, and *XOR* instructions, similar to the *ADD* instruction, show a speed-up of approximately 600% in JiT compiled mode compared to interpretation mode; for the *MOV* instruction, JiT compiled mode attained a 500% speed-up.

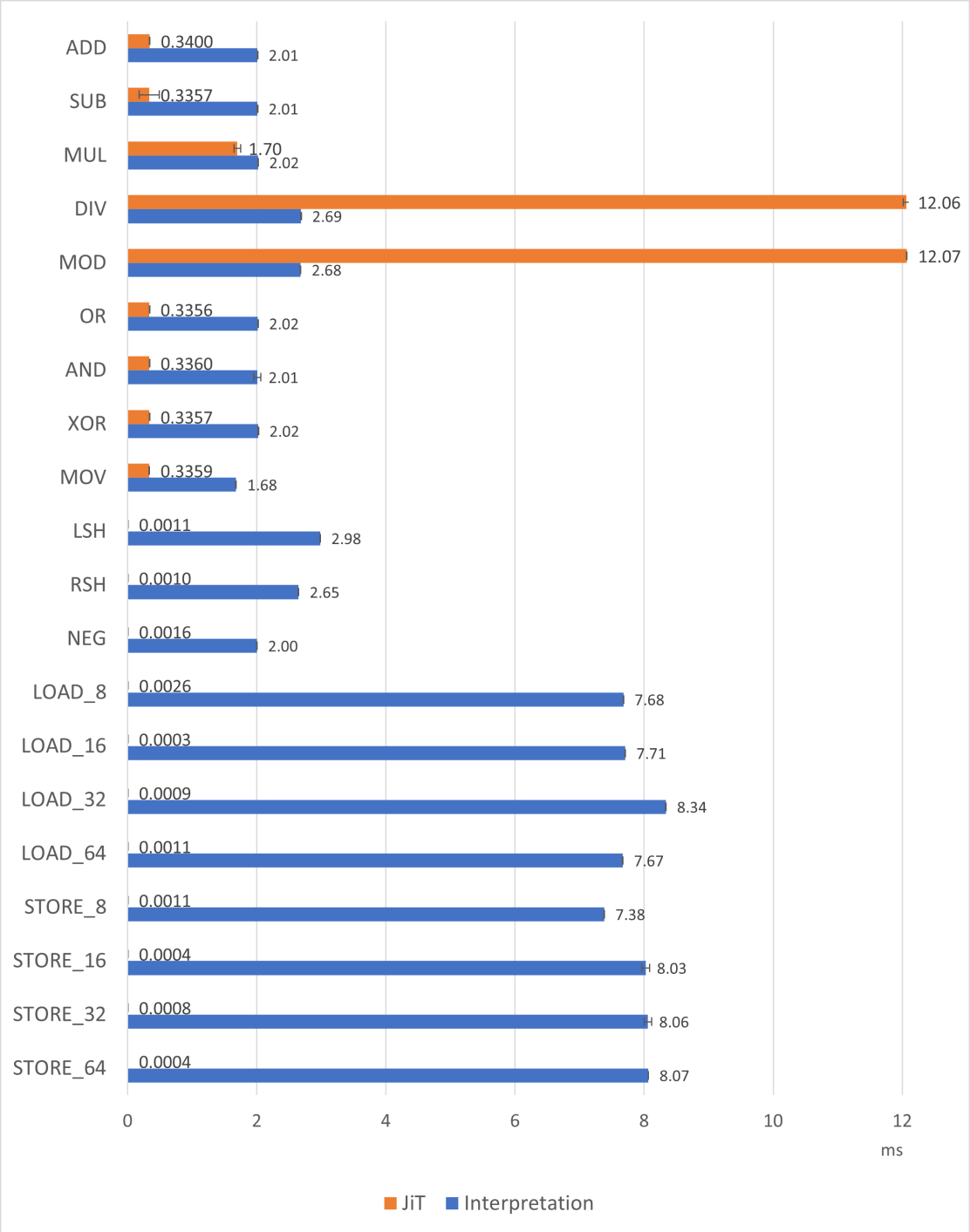

Figure 6.11: Time required for a one-million-iteration microbenchmark of the selected BPF instructions.

```
1 add rsi, 1
```

Figure 6.12: The x86 instruction of the JiT-compiled BPF instruction, *ADD*, from uBPF.

```
1 push rax
2 push rdx
3 mov rcx, 1
4 mov rax, rsi
5 mul rcx
6 pop rdx
7 mov rsi, rax
8 pop rax
```

Figure 6.13: The x86 instruction of the JiT-compiled BPF instruction, *MUL*, from uBPF.

```
 1 push rax
 2 push rdx
 3 mov rcx, 1
 4 mov rax, rsi
 5 test rcx, rcx
 6 pushfq
 7 mov rdx, 1
 8 cmove rcx, rdx
 9 xor edx, edx
10 div rcx
11 popfq
12 mov rcx, 0
13 cmove rax, rcx
14 pop rdx
15 mov rsi, rax
16 pop rax
```

Figure 6.14: The x86 instruction of the JiT-compiled BPF instruction, *DIV*, from uBPF.

```
1  # ... The jumping table
2
3  movzx ebx, bl
4  movsxd r13, r13d
5  xor eax, eax
6  lea rcx, [r15+rbx*8]
7  test r13d, r13d
8  jz short loc_537B
9
10 loc_537B:
11 mov rax, [rcx]
12 xor edx, edx
13 div r13d
14 mov [rcx], rax
15
16 # ... jump back to jumping table
```

Figure 6.15: The disassembled uBPF interpreter handling the BPF *DIV* instruction. We only list the instructions for cases where the divisor is not zero.

Bitwise Operation Instructions

The results of the *LSH*, *RSH*, and *NEG* instructions are also surprising. Our results show that these instructions are so fast under JiT compiled mode that we can hardly notice any difference compared with the control groups.

We immediately did further research by reverse-engineering the JiT-compiled BPF program, and it is confirmed that these BPF instructions are correctly translated into the x86 instructions with *shl*, *shr*, and correspondingly *neg*. In addition, we found that the initial value in the *r2* register and the instruction operand types (such as immediate value and register) have nothing to do with the results. These uncommon results did not frustrate us. Instead, they serve as a remarkable demonstration of the execution of BPF programs under JiT compiled mode for us, where the hardware can occasionally optimize the program if the JiT compiler is implemented efficiently.

Due to the diversity and lack of other analysis tools support, such as software emulation, of our underlying Unikernel framework, Unikraft, we have no choice but to exclude the results of these BPF instructions from this work. However, it is suggested to continue more in-depth research in this direction in the future.

Memory Access Instructions

For the memory access instructions, we measured *LOAD* and *STORE* instructions with different data widths, 8bits, 16bits, 32bits, and 64bits, respectively.

Similar to the bitwise instructions, we suspect that the hardware again aggressively optimized the evaluation programs. The time required for the million iterations under the JiT compiled mode is so tiny that we can neglect it. The results are, therefore, also excluded from this work.

6.5.3 Real World Examples

To discover the performance of UniBPF on the real-world applications, we implemented three different BPF programs with which to evaluate our design, including: **Nop**, **Count**, and **DummyCount**. With these test programs, during the evaluation, we will first benchmark the system without any BPF programs attached to the Unikernel application as the baseline. Then, we will hook the BPF programs to specific functions of the applications and measure their performance.

The following evaluations are performed on a Nginx [Ree08] and Redis [San09] server based on Unikraft and xIO.

For Nginx, we attach our BPF programs to the request handler function of Nginx, `ngx_http_process_request_line`, and benchmark it with the tool wrk [Glo12] with three threads, 30 connections for each thread and last for 30 seconds in each trail; for

Redis, we attach it to the Redis command handling function, `processCommand`, and benchmark it with the tool redis-benchmark [Red] with 30 clients and two million queries in each round.

The BPF program **Nop** (Figure 6.16) always returns immediately when invoked. We consider this program a baseline to compare the results of different system setups, allowing us to understand how the different BPF runtime execution strategy integrations affect the system. At the same time, by comparing the results with the results of the other BPF programs, we can consequently observe the efficiency of the BPF runtime system.

Next, the BPF program **Count** (Figure 6.17) increments a value in the internal key-value store by one with the specific keys as soon as it is invoked. We use this program to simulate a real-world performance counter application.

Finally, the BPF program **DummyCount** is implemented based on the BPF program **Count** by appending a thousand dummy instructions that add zeros to the return value register, `r0`, at the end of the program. With this program, we want to simulate a BPF program with heavy tasks.

```
1 __attribute__((section("tracer"), used))
2 int bpf_tracer(bpf_tracer_ctx_descriptor_t *ctx_descr) {
3         return 0;
4 }
```

Figure 6.16: **nop.c**: This BPF program serves as a control group and demonstrates that including the BPF runtime system into Unikernels results in negligible overhead regardless of the chosen execution strategy.

Experiment Setups

In the experiments with real-world applications, we reduced the CPU affinity limits of the application and applied CPU affinity limits to the benchmarking tool to improve reproducibility. For both applications, we assigned their affinity to the first to fourth CPU core; for the benchmarking tools, both are assigned with affinity to the fifth to eighth core.

Observations

As a result, the statistics shown in Figure 6.18 gave us many fascinating insights.

```
1  __attribute__((section("tracer"), used))
2  int bpf_tracer(bpf_tracer_ctx_descriptor_t *ctx_descr) {
3          if(ctx_descr->data_end - ctx_descr->data !=
4              sizeof(struct UbpfTracerCtx)) {
5                  return -1;
6          }
7
8          __u64 count = bpf_map_get(
9              ctx_descr->ctx.traced_function_address, COUNT_KEY);
10         if (count == UINT64_MAX) {
11                 count = 0;
12         }
13
14         count++;
15
16         bpf_map_put(
17         ctx_descr->ctx.traced_function_address, COUNT_KEY, count);
18
19         return 0;
20 }
```

Figure 6.17: **count.c**: Designed as a performance counter, this BPF program increments a value by one in the internal key-value store with designated keys each time it is invoked.

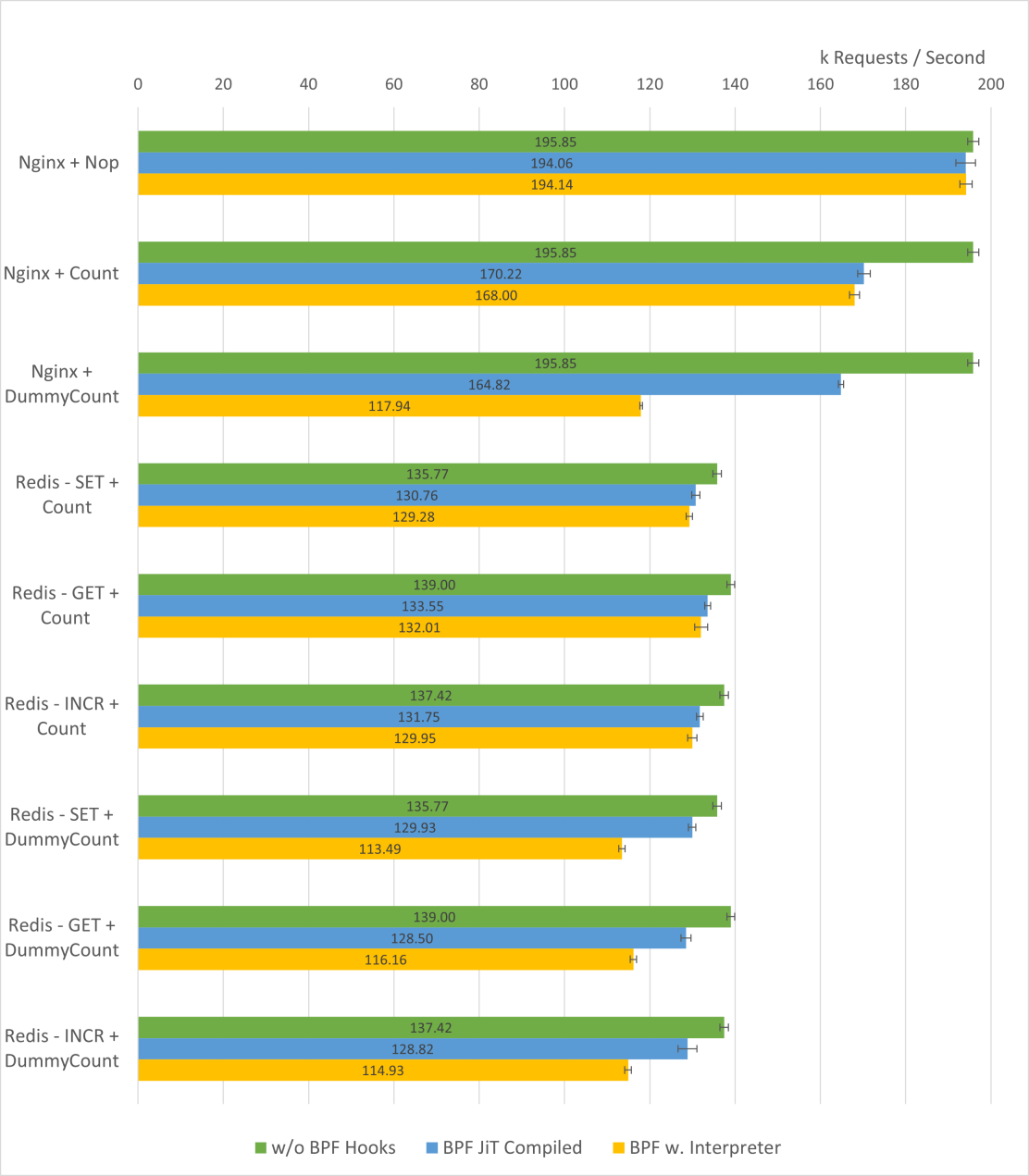

Figure 6.18: The efficiency of actual applications when combined with various BPF testing programs using differing BPF runtime execution strategies.

First, from the combination of the **Nop** executable with Nginx, we found that the integration of the BPF runtime system itself, no matter whether it is in interpretation or JiT compiled mode, adds very little overhead. Even with two BPF instructions executing `return 0` loop nearly 193855 times per second on average, it resulted in less than 1% overhead ($\leq$ 9.2‰).

Second, the statistics themselves prove that enabling JiT compilation improves overall system performance in real-world cases: With the **Count** test program attached, the throughput of the Ningx increased by more than 1.32% in JiT compiled mode compared to interpretation mode; for Redis, the benchmark results also suggest that the performance of three different commands, *SET*, *GET*, and *INCR*, increases by 1.17% to 1.38% when the BPF program is executed in JiT compiled mode.

Finally, after analyzing statistics from the BPF program **DummyCount**, we have further confirmed our hypothesis that the JiT compiled mode yields better BPF runtime performance. As the statistics show, the efficiency of the JiT compiled mode increases proportionally with the size of the BPF program.

In **DummyCount**'s case, the Redis application shows a speed improvement between 10.63% and 14.48% when replacing the interpretation mode with the JiT compiled mode. In addition, the results indicate that the Nginx application experiences a significantly more speed increase of approximately 39.75% when **DummyCount** is attached under JiT compiled mode.

Based on the above results, it is clear that although performance gains are limited for smaller BPF programs, the use of JiT compiles does bring better results, especially as the size of the BPF program increases.

7 Related Works

7.1 Just-in-Time BPF Compiliation

Along with the official release of the eBPF system in the Linux system since version 3.18, Linux ships its own JiT compiler implementation along with its kernels [Aut23]. When a request is made to attach BPF programs with kernel functions, if the kernel option *CONFIG_BPF_JIT* is enabled, it will compile the BPF programs on the fly before attaching them to the kernel space [SBS]. Additionally, for performance reasons and to prevent the exploit where Linux's BPF interpreter is vulnerable to Spectre variant 2 [Koc+19] attacks, Linux added the option to force the kernel to always use JiT compiled mode [SB18].

However, using the GNU General Public License (GPL) license, especially in Linux's eBPF system, makes people concerned about porting it into their projects. More specifically, the GPL license makes it inapplicable to many projects [Iov]. To solve such a problem, uBPF as an Apache-Licensed project is published. uBPF provides a comprehensive eBPF runtime, including BPF interpreter, JiT BPF compiler, and support of customized BPF helper functions.

Using uBPF, the project *ebpf-for-windows* [Mic] also supports JiT compiling BPF programs in their system. Unlike Linux system JiT compiling the BPF programs in the kernel space when the system call is made to attach the BPF program to the system, *ebpf-for-windows* compiles the BPF programs proactively in the userspace, and then, only the translated platform native codes are passed away to the kernel space.

Jitterbug [Nel+20] is another research effort focused on creating BPF JiT compilers with correctness-proven models. Jitterbug has proven to be an efficient and effective framework. It also successfully located several zero-day JiT compiler bugs in the Linux system and successfully patched them with minimal changes.

7.2 BPF Verification

As soon as a request to load a BPF program into the kernel space is made in the Linux system, it is verified by the embedded BPF verifier. As one of the most critical components in the Linux eBPF system, the BPF verifier enhances system security

and integrity by providing comprehensive security and compatibility checks for eBPF programs. Especially with the increasing power and flexibility of BPF programs and the rapid evolution of the Linux system, the Linux BPF Verifier has become essential to enhance security and prevent misuse.

The *ebpf-for-windows* project also included a third-party BPF verifier implementation, PREVAIL [Ger+19], to verify the BPF program before applying them [Mic]. Instead of running the BPF verifiers directly in the kernel space, the verifier components of *ebpf-for-windows* are placed in the userspace and are designed to be protected by a secure environment such as a system service (its current implementation), an enclaved runtime, or even in a trusted VM [Mic].

The previous work integrating BPF runtime into Unikernels by Hendrychová et al. [Hen23] utilized a BPF interpreter to perform BPF program verifications on the fly.

Besides Linux's BPF verifier, several open-source BPF verification projects exist. One of them, mentioned in the previous sections, is PREVIAL [Ger+19].

PREVAIL is a static eBPF program analyzer developed based on the typical pattern of BPF programs observed by Gershuni et al. [Ger+19]. The notable achievements of PREVAIL are its **flexibility**, **accuracy**, and the capability to verify the BPF programs with loops.

For flexibility, PREVAIL can be easily reconfigured with different verification features as required by developers, such as prohibiting runtime errors like division by zero and ensuring program termination. Developers can also apply different *Zone* settings to the verifier, indicating different verification strengths. From a performance point of view, although PREVAIL had no advantage over Linux's BPF verifier in general, PREVAIL showed its advantage in programs containing loops. Unlike Linux, which verifies BPF programs by exhaustively finding every execution pattern, PREVAIL is much more applicable as a CFG-based verification solution in this case [Ger+19].

KLINT [Pir+22] is another research we noticed while searching for BPF verification-related solutions during our work. KLINT is designed to verify the software stack in networking, and most importantly, it also supports the verification of BPF programs. When using KLINT, developers write verification rules in Python, and through the lightweight Python interpreter represented by KLINT, the given software can be verified accurately and efficiently.

7.3 Decopuled BPF Auditor

ebpf-for-windows [Mic] presents its BPF verification framework differently by separating the BPF verifier and compiler processes from the kernel runtime. Also, this project proves the feasibility of achieving BPF verifications in an untrusted environment with

secure computing.

Recent research, HyperBee [WLC23], also argues in favor of moving the BPF auditing process out of the system core space, i.e., the kernel space. Their research shows the possibility of hijacking the system with malicious BPF programs by misleading the embedded BPF monitoring components. Without an effective existing solution to mitigate the issues mentioned above, HyperBee proposed their solution of placing the auditor components outside the system, and they claimed that in such a way, with more resources available, the auditor can leverage its sight and detect hidden maliciousness more easily. Focusing on virtualized systems such as cloud VMs, HyperBee implemented their system via generic virtual IO devices, making the solution portable to all other systems supporting BPF in the future.

Coincidentally, Craun et al. have also recently proposed a solution to enable computationally resource-constrained embedded systems with verified BPF runtimes by using external BPF compilation and a decoupled BPF verifier [COW23]. In their work, they also believe that: First, a JiT-compiled BPF program is useless unless verified; second, it is impractical to perform such heavy tasks of verifying BPF programs inside the target runtime system, especially when the target platform has very limited resources. In fact, in their work, they found a 90x slowdown when verifying the same BPF program on embedded systems compared to a general-purpose server.

As a result, they proposed a solution with external BPF compilations and decoupled BPF verification processes running in VMs, which are general enough to cover all compatible operating systems and are easy to use.

8 Conclusions

In this work, we successfully provide a new BPF verification framework that enhances Unikernel's BPF runtime with improved **security**. Additionally, by taking advantage of the verifier, we enabled JiT compiled mode without worrying about malicious BPF programs attached to the system and witnessed the **performance improvements** expected from JiT compilations.

Our approach decoupled the BPF verification process from the Unikernels and resulted in a minimal runtime impact on the Unikernels **(design challenge 1)**, ingeniously avoided the problems and difficulties of integrating superfluous components into Unikernels **(design challenge 2)**, and also successfully preserved the high customizability and compactness characteristics of Unikernel systems.

Finally, with the method *on-the-fly verifier initialization*, we have successfully developed a **sustainable** BPF verification application that can serve all Unikernel applications in a user-friendly and maintainer-friendly manner **(design challenge 3)**.

In conclusion, this work utilizes and extends the security benefits that the BPF language can bring to Unikernels with improved performance in their BPF runtime system while maintaining its usability.

9 Further Discussion

9.1 Runtime Configurable Unikernel Application

Typically, the business process workflow is hard-coded into the programs, leaving only a tiny portion configurable through, e.g., YAML or JSON configuration files. With the efficient BPF runtime environment created in this work, the use case implementing BPF programs as a system configuration solution has become possible.

Let's take the example of an Internet router application. Traditionally, only minimal reconfiguration of routes can be done at runtime, such as adjusting netmasks and destination interfaces. With UniBPF, however, it becomes possible to even change the routing algorithm on the fly. Such an idea also allows fine-tuning the routing algorithm in the given case with the help of compiler optimizations.

9.2 Support of BPF Maps

The verification of BPF maps remains unresolved in the verification system we formed. As the project concluded, we have yet to understand how it can be accomplished.

9.3 Integrity of Verification

Our research overlooked the importance of maintaining the integrity of verification results. This could allow an attacker to bypass the verification process using a tampered verifier application or directly communicating with Unikernel applications via debug interfaces.

We believe that digital signatures can solve this problem in the future. Our idea is as follows: First, we embed root Cerificate Authority (CA) certificates (i.e., the public keys) in the Unikernel applications; on the verifier side, we embed corresponding private keys in the verifier application. And then, cleverly, after verifying a BPF program, a digital signature and the signature information are appended to the BPF program file, **if and only if** the verifier recognizes the security of the BPF program.

Figure 9.1 shows that a signature block is appended to the end of the original BPF program, proving the validity of the BPF program. The diagram shows only a fraction

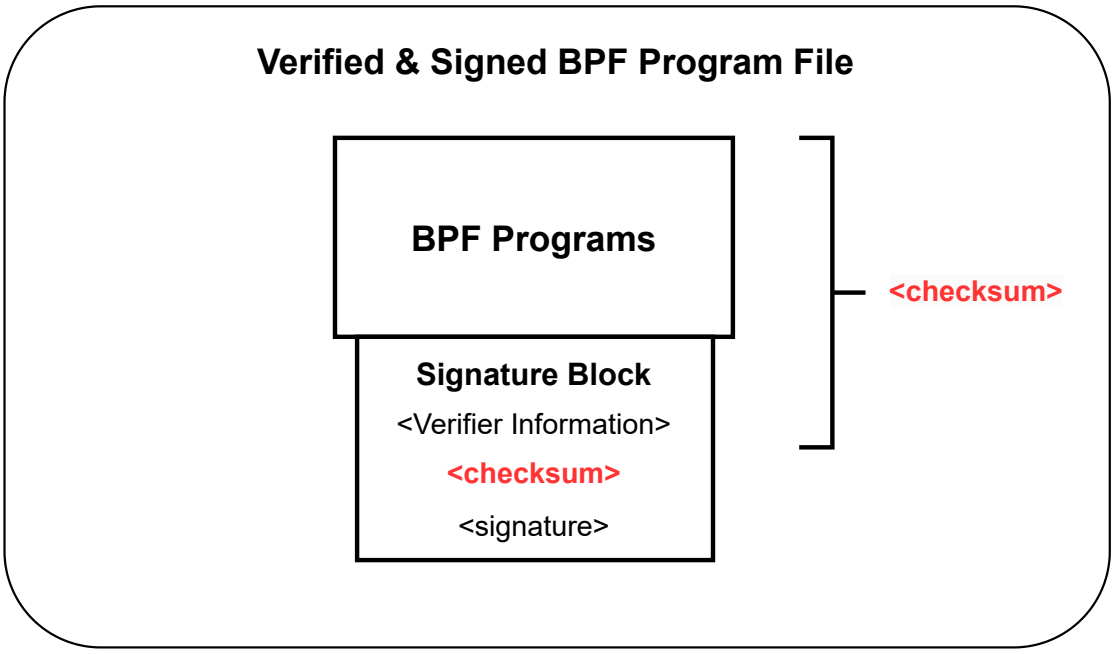

Figure 9.1: Design overview: Protect BPF program verification information with digital signatures.

of the signature elements that could be included in a single digital signature block. However, we suggest that the following information will be essential in a real-world implementation:

1. The BPF runtime specification of Unikernels, i.e., the BPF helper function specifications and BPF program type definitions.
2. The BPF verifier specifications, e.g., Are runtime errors checked? Is program termination checked?
3. Information about the key pair signing this BPF program.
4. Timestamps including *NotAfter* and *NotBefore* fields.
5. The algorithms used.
6. The checksums (as already shown in the image).
7. The digital signature (as already shown in the image).

When a Unikernel attempts to execute a digitally signed BPF program, it must first locate the digital signature block. Next, it computes the hash of the entire BPF program and the verifier parameters to prevent downgrade attacks. Finally, the Unikernel system verifies the hash against the provided digital signature and decides whether to take the BPF program.

This solution enhances the system's security and integrity and increases efficiency by eliminating redundant verification processes. Once we confirm that the program file has been signed with compatible private keys and specifications, we can skip verification and directly send the file to the application.

However, this solution also hinders the efficiency of verifying BPF programs, particularly in files containing multiple BPF programs. Our previous implementation allowed for the verification of individual pieces of programs in the file, whereas this design requires all BPF programs to be verified simultaneously due to the use of block-wise digital signatures.

Another potential issue is security: if the cloud provider's system is breached, the BPF verifier and its corresponding key pairs that sign the BPF programs may also be compromised. However, this can be mitigated by requiring developers to use complex passwords for their key pairs.

A related solution can also be found in another research work proposed with decoupled BPF verification processes for embedded systems [COW23].

9.4 Zero-Trust on Cloud Provider

Cloud providers can be compromised or evil, but all our assumptions so far are based on a trustworthy cloud provider. In the untrustable cloud provider's case, Confidential Virtual Machine (CVM) may help. With the assistance of hardware features, e.g., AMD's *Secure Nested Pages* [Sev20], Intel's *Trust Domain Extensions* [Int] along with other measurements like UEFI SecureBoot and Volume encryptions, can the CVM be protected from being tampered with and can the CVM itself remain confidential to hypervisors [Kuz23]. Our initial idea is to place the UniBPF verifier application, and the key pairs inside a CVM; simultaneously, the Unikerenl application will be executed in another CVM. The debug interfaces and shared filesystem parts remain unchanged and will still be placed on the host system, as shown in Figure 9.2.

In the new design, the debug interface will be encrypted to prevent being eavesdropped by supervisors by building a secure channel between the verifier and Unikernel applications. Regarding constructing the secure channel, we can apply existing Public Key Infrastructure (PKI) and asynchronous encryption technologies to exchange the encryption key safely. It can, however, remain unencrypted on the shared filesystem's

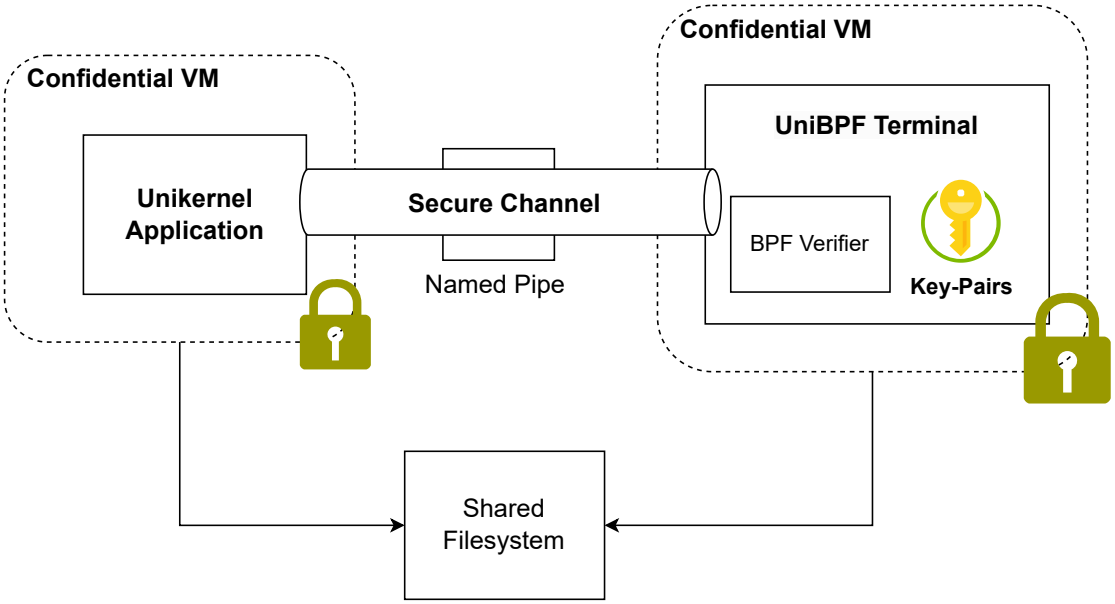

Figure 9.2: Design overview: Protect Unikernels and UniBPF secrets with CVM.

side, benefiting from the digital signatures.

9.5 Stronger BPF Runtime Isolation

While developing UniBPF, we discovered that the isolation between the embedded BPF runtime and the Unikeneral application is too weak. First, we noticed that the BPF runtime can crash the whole system when it runs into undefined circumstances, e.g., NULL pointer dereference. In addition, we found that even the BPF verifier within the Linux system, despite having a large user community, can not be error-free. For example, **CVE 2020-8835** [Ltd20] suggests that the failure of the BPF verifier to check the register bound of 32-bit operations could lead to arbitrary reads and writes to kernel memory, lateral privilege escalation, and container escape [20]. Coincidentally, **CVE 2023-2163** was recently revealed, demonstrating that the BPF verifier in the Linux system may ignore unsafe execution paths and incorrectly recognize them as safe [JI23], leading to potentially catastrophic results similar to **CVE 2020-8835**.

After careful consideration, we have come up with three possible solutions. Among the proposed solutions, the most relevant and achievable one, we believe, is Intel Memory Protection Keys (MPK).

9.5.1 Threads

Our initial instinct was to run the BPF runtime on a separate thread because of its lightweight nature, which seemed to be the most practical approach to meeting the performance requirements. However, upon further consideration, we realized that traditional threads do not provide adequate isolation from other threads under the same process. More specifically, threads cannot prevent the BPF runtime from, for example, accessing memory belonging to other threads under the same process.

9.5.2 Processes

Modern operating systems use virtual memory to separate processes from each other. Each process has its own page tables that map its virtual address space to linear addresses. While process-based technologies achieve perfect memory space isolation, they are not the optimal solution due to their inefficiency. The overhead of switching between different processes, known as context switching, is significant. This includes the time it takes to switch virtual memory spaces, which invalidates the cache and results in substantial overhead when reloading data from memory. For example, a single memory read on an Intel Nehalem processor takes about 185 CPU cycles, i.e., about 63 ns on a 2.933 GHZ processor [Mol+09].

We also considered using *global page entries* to mitigate these phenomena. Characterized by that in an x86 CPU, the page entries will never be evicted from the Translation Lookaside Buffer (TLB) when utilizing this technique. Another solution we came up with was to use *segments* instead of virtual memory to avoid page table swaps and the associated latencies. However, these two solutions do not help at all since changing the read/write permission during such a so-called lightweight context switch still cannot prevent TLB from being flushed. Also, implementing processes in the Unikernel system, from zero to hero, could be a significant challenge.

9.5.3 Intel Memory Protection Keys (MPK)

Intel MPK is a new hardware primitive feature available from Skylake server CPU to support thread-local permission control on groups of pages with only little switching overhead [Par+19][PLK23]. With MPK, memory can be divided into up to 16 regions, and each part can only be accessed if the appropriate memory protection key is set in the protection key rights register (PKRU) [PLK23]. Conversely, MPK takes only about 23 CPU cycles to change the permission view using the `WRPKRU` instruction, which can speed up the context switch by 11x compared to the process solution [Pen+23].

Due to the lightweight and the promise of memory isolation, we think MPK might be the most feasible solution. Our solution is as follows: First, we dedicate the memory

space into multiple regions and sort the components of the Unikernel system into the memory regions. Then, we protect the memory regions with MPK, making them accessible only to the components under the same region or to specific components according to the design.

For example, as shown in Figure 9.3, we would give the system kernel access to all other memory regions (the blue marked regions) and the BPF runtime limited access to specific BPF functions in the *BPF Helper Function Stub* (the yellow marked region). In fact, if we take a closer look at the figure, the BPF programs have access to only a tiny part of the BPF functions, which we call *call-gates*. We assume that these *call-gates* check the access rights, ensure the validity of the helper function call, perform the necessary configurations, and finally perform the actual BPF helper function calls. The similar ideas can also be found in μSWITCH [Pen+23] and ERIM [Vah+19].

At the same time, we believe that MPK as a CPU hardware feature is reliable, and the security promises are concrete. Although one can still argue that the hardware feature may still be bugged and vulnerable, such as Meltdown [Lip+18] and Spectre [Koc+19], undeniably, such possibilities are much lower due to the intensive verifications done on the CPU hardware designs before the rollout.

Despite the advantages of MPK, the disadvantage of MPK is that the Unikernel systems can only host about 14 isolated BPF programs simultaneously since MPK supports only 16 different isolated encryption keys, assuming that one of them is reserved for the system kernel/application and the other for the helper function call-gates. However, further research works [Gu+22][Par+19] suggests the possibility of increasing this number.

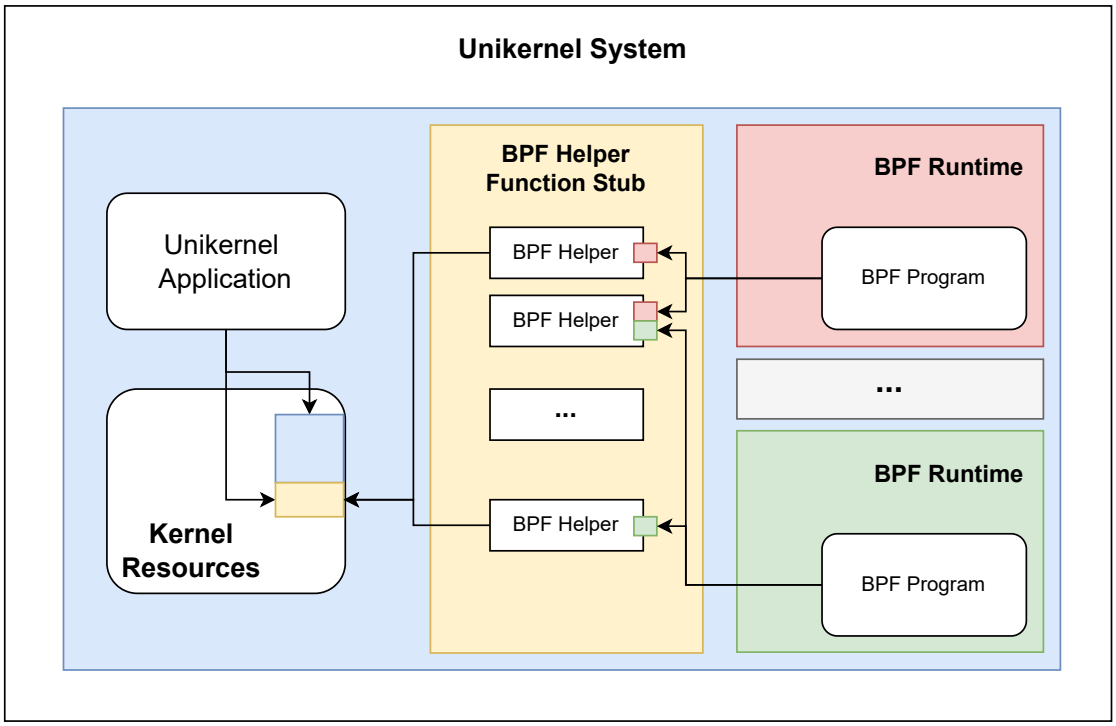

Figure 9.3: Design overview - isolation of Unikernel components with MPK: Blocks with different background colors indicate that the component has exclusive access only to components marked with the same color.

Abbreviations

AWS Amazon Web Services

VM Virtual Machine

eBPF extended Berkeley Packet Filter

BPF Berkeley Packet Filter

XDP eXpress Data Path

DFA Deterministic Finite Automata

CWD Current Working Directory

CA Cerificate Authority

CVM Confidential Virtual Machine

PKI Public Key Infrastructure

TLB Translation Lookaside Buffer

MPK Memory Protection Keys

IPC Instructions per Cycle

JiT Just-in-time

WinRS Windows Remote Shell

RDP Remote Desktop

GPL GNU General Public License

DoS Denial-of-Service

RSS Resident Set Size

LXC Linux Containers

CSPF CMU/Stanford Packet Filter

NIT Network Interface Tap

TTY TeleTYpewriter

libOS library perating System

List of Figures

List of Tables

Bibliography

[20] *CVE-2020-8835: LINUX KERNEL PRIVILEGE ESCALATION VIA IMPROPER EBPF PROGRAM VERIFICATION*. Apr. 2020. URL: https://www.zerodayinitiative.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification (visited on 10/27/2023).

[Aga+20] A. Agache, M. Brooker, A. Florescu, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. "Firecracker: Lightweight Virtualization for Serverless Applications." In: *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*. NSDI'20. Santa Clara, CA, USA: USENIX Association, 2020, pp. 419–434. ISBN: 9781939133137.

[AM14] D. B. Alexei Starovoitov and D. S. Miller. *net: filter: split filter.h and expose eBPF to user space*. Sept. 2014. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=daedfb22451dd02b35c0549566cbb7cc06bdd53b (visited on 10/27/2023).

[Aut] eBPF.io Authors. *What is eBPF? An Introduction and Deep Dive into the eBPF Technology*. URL: https://ebpf.io/what-is-ebpf/ (visited on 10/16/2023).

[Aut23] C. Authors. *BPF and XDP Reference Guide — Cilium 1.15.0-dev documentation*. Apr. 2023. URL: https://docs.cilium.io/en/latest/bpf/ (visited on 10/27/2023).

[Avr22] Y. Avrahami. *Breaking out of Docker via runC – Explaining CVE-2019-5736*. June 2022. URL: https://unit42.paloaltonetworks.com/breaking-docker-via-runc-explaining-cve-2019-5736/ (visited on 10/27/2023).

[BMG99] A. Begel, S. McCanne, and S. L. Graham. "BPF+: Exploiting Global Data-Flow Optimization in a Generalized Packet Filter Architecture." In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '99. Cambridge,

Massachusetts, USA: Association for Computing Machinery, 1999, pp. 123–134. ISBN: 1581131356. DOI: 10.1145/316188.316214.

[BS19] D. Borkmann and A. Starovoitov. *bpf: introduce bounded loops*. June 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/?id=2589726d12a1b12eaaa93c7f1ea64287e383c7a5 (visited on 10/22/2023).

[Bue19] P. Buer. *Unikernels Aren't Dead. They're Just Not containers*. May 2019. URL: https://www.infoq.com/presentations/unikernels-includeos/ (visited on 10/31/2023).

[CK22] A. Colomar and M. Kerrisk. *BPF-helpers(7) - linux manual page*. Sept. 2022. URL: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html (visited on 10/22/2023).

[com] T. kernel development community. *1 BPF Instruction Set Specification, v1.0*. URL: https://docs.kernel.org/bpf/standardization/instruction-set.html (visited on 10/22/2023).

[con23] W. contributors. *Hayes AT Command Set*. Sept. 2023. URL: https://en.wikipedia.org/wiki/Hayes_AT_command_set (visited on 10/04/2023).

[COW23] 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*. eBPF '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 63–69. ISBN: 9798400702938. DOI: 10.1145/3609021.3609299.

[DL08] E. Dolstra and A. Löh. "NixOS: A Purely Functional Linux Distribution." In: *SIGPLAN Not.* 43.9 (Sept. 2008), pp. 367–378. ISSN: 0362-1340. DOI: 10.1145/1411203.1411255.

[EKO95] D. R. Engler, M. F. Kaashoek, and J. O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management." In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: Association for Computing Machinery, 1995, pp. 251–266. ISBN: 0897917154. DOI: 10.1145/224056.224076.

[Fog22] A. Fog. *4. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs*. Nov. 2022. URL: https://www.agner.org/optimize/instruction_tables.pdf (visited on 10/27/2023).

[Ger+19] 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*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1069–1084. ISBN: 9781450367127. DOI: 10.1145/3314221.3314590.

[Glo12] W. Glozer. *wg/wrk: Modern HTTP Benchmarking Tool*. 2012. URL: https://github.com/wg/wrk (visited on 10/31/2023).

[Gu+22] J. Gu, H. Li, W. Li, Y. Xia, and H. Chen. "EPK: Scalable and Efficient Memory Protection Keys." In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 609–624. ISBN: 978-1-939133-29-36.

[Han99] S. M. Hand. "Self-Paging in the Nemesis Operating System." In: *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*. New Orleans, LA: USENIX Association, Feb. 1999.

[He+22] Y. He, Z. Zou, K. Sun, Z. Liu, K. Xu, Q. Wang, C. Shen, Z. Wang, and Q. Li. "RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices." In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2225–2242. ISBN: 978-1-939133-31-1.

[Hen23] V. Hendrychová. "Extending unikernels with a language runtime." MA thesis. Technical University of Munich, 2023.

[Hil23] M. Hill. *SSHLOG/agent: SSH session monitoring daemon*. Apr. 2023. URL: https://github.com/sshlog/agent (visited on 10/27/2023).

[Høi+18] 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*. CoNEXT '18. Heraklion, Greece: Association for Computing Machinery, 2018, pp. 54–66. ISBN: 9781450360807. DOI: 10.1145/3281411.3281443.

[Int] Intel. *Intel® Trust Domain Extensions (Intel® TDX)*. URL: https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html (visited on 10/31/2023).

[Iov] Iovisor. *Iovisor/UBPF: Userspace EBPF VM*.
URL: https://github.com/iovisor/ubpf (visited on 10/04/2023).

[JI23] J. J. L. Jaimez and M. Inge.
Introducing a new way to buzz for eBPF vulnerabilities. May 2023.
URL: https://security.googleblog.com/2023/05/introducing-new-way-to-buzz-for-ebpf.html (visited on 10/27/2023).

[Koc+19] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom.
"Spectre Attacks: Exploiting Speculative Execution."
In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[Kue+21] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, Ş. 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*. EuroSys '21. Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 376–394. ISBN: 9781450383349.
DOI: 10.1145/3447786.3456248.

[Kuz23] V. Kuznetsov. *Introduction to confidential Virtual Machines*. June 2023.
URL: https://www.redhat.com/en/blog/introduction-confidential-virtual-machines (visited on 10/06/2023).

[LA04] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation." In: *CGO*.
San Jose, CA, USA, Mar. 2004, pp. 75–88.

[Ler+23] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon.
Understanding the Remote Desktop Protocol (RDP) | Microsoft Learn.
Sept. 2023. URL: https://v2.ocaml.org/manual/ (visited on 11/08/2023).

[Lip+18] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg.
"Meltdown: Reading Kernel Memory from User Space."
In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[Ltd20] C. Ltd. *NVD - CVE-2020-8835*. Apr. 2020.
URL: https://nvd.nist.gov/vuln/detail/CVE-2020-8835 (visited on 10/27/2023).

[LXC23] H. Liang, S. Xu, and L. Chen.
Understanding the Remote Desktop Protocol (RDP) | Microsoft Learn. Feb. 2023. URL:
https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/hh875630(v=ws.11) (visited on 10/31/2023).

[LY06] C. M. Lonvick and T. Ylonen. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. Jan. 2006. DOI: 10.17487/RFC4251.

[Mic] Microsoft.
Microsoft/EBPF-for-windows: Ebpf implementation that runs on top of windows. URL: https://github.com/microsoft/ebpf-for-windows#architectural-overview (visited on 10/04/2023).

[Mic16] Microsoft. *Winrs | Microsoft Learn*. Aug. 2016. URL:
https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/hh875630(v=ws.11) (visited on 10/31/2023).

[Mis23] M. Misono. "xIO." unpublished. 2023.

[MJ93] S. McCanne and V. Jacobson.
"The BSD Packet Filter: A New Architecture for User-level Packet Capture." In: *USENIX Winter 1993 Conference (USENIX Winter 1993 Conference)*. San Diego, CA: USENIX Association, Jan. 1993.

[Mol+09] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller.
"Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System." In: *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 2009, pp. 261–270.
DOI: 10.1109/PACT.2009.22.

[MRA87] J. Mogul, R. Rashid, and M. Accetta.
"The packer filter: an efficient mechanism for user-level network code." In: *ACM SIGOPS Operating Systems Review* 21.5 (1987), pp. 39–51.

[MS13] A. Madhavapeddy and D. J. Scott. "Unikernels: Rise of the Virtual Library Operating System: What If All the Software Layers in a Virtual Appliance Were Compiled within the Same Safe, High-Level Language Framework?" In: *Queue* 11.11 (Dec. 2013), pp. 30–44. ISSN: 1542-7730.
DOI: 10.1145/2557963.2566628.

[Nel+20] L. Nelson, J. V. Geffen, E. Torlak, and X. Wang. "Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel." In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 41–61. ISBN: 978-1-939133-19-9.

[Par+19] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. "libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)." In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 241–254. ISBN: 978-1-939133-03-8.

[Pen+23] D. Peng, C. Liu, T. Palit, P. Fonseca, A. Vahldiek-Oberwagner, and M. Vij. "μSwitch: Fast Kernel Context Isolation with Implicit Context Switches." In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, pp. 2956–2973. DOI: 10.1109/SP46215.2023.10179284.

[Pir+22] S. Pirelli, A. Valentukonytė, K. Argyraki, and G. Candea. "Automated Verification of Network Function Binaries." In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 585–600. ISBN: 978-1-939133-27-4.

[PLK23] S. Park, S. Lee, and T. Kim. "Memory Protection Keys: Facts, Key Extension Perspectives, and Discussions." In: *IEEE Security & Privacy* 21.03 (May 2023), pp. 8–15. ISSN: 1558-4046. DOI: 10.1109/MSEC.2023.3250601.

[Por+11] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. "Rethinking the library OS from the top down." In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*. Ed. by R. Gupta and T. C. Mowry. ACM, 2011, pp. 291–304. DOI: 10.1145/1950365.1950399.

[Qia+09] L. Qian, Z. Luo, Y. Du, and L. Guo. "Cloud Computing: An Overview." In: vol. 5931. Jan. 2009, pp. 626–631. ISBN: 978-3-642-10664-4. DOI: 10.1007/978-3-642-10665-1_63.

[Red] Redis. *Redis benchmark*. URL: https://redis.io/docs/management/optimization/benchmarks/ (visited on 10/31/2023).

[Ree08] W. Reese. "Nginx: The High-Performance Web Server and Reverse Proxy." In: *Linux J.* 2008.173 (Sept. 2008). ISSN: 1075-3583.

[San09] S. Sanfilippo. *Redis*. May 2009.
URL: https://redis.io/ (visited on 10/31/2023).

[SB18] A. Starovoitov and D. Borkmann.
bpf: introduce BPF_JIT_ALWAYS_ON config. Jan. 2018.
URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=290af86629b25ffd1ed6232c4e9107da031705cb
(visited on 10/27/2023).

[SBS] J. Schulist, D. Borkmann, and A. Starovoitov.
Linux Socket Filtering aka Berkeley Packet Filter (BPF). URL:
https://www.kernel.org/doc/Documentation/networking/filter.txt
(visited on 10/27/2023).

[Sev20] A. Sev-Snp.
"Strengthening VM isolation with integrity protection and more."
In: *White Paper, January* 53 (2020), pp. 1450–1465.

[SK21] A. Starovoitov and J. Koon. *bpf: Add bpf_loop helper*. Nov. 2021.
URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?h=e6f2dd0f80674e9d5960337b3e9c2a242441b326
(visited on 10/22/2023).

[SW94] W. Stevens and G. Wright. *TCP/IP Illustrated: The protocols*.
Addison-Wesley professional computing series. Addison-Wesley, 1994.
ISBN: 9780201633467.

[Tal+20] J. Talbot, P. Pikula, C. Sweetmore, S. Rowe, H. Hindy, C. Tachtatzis,
R. Atkinson, and X. Bellekens. "A Security Perspective on Unikernels." In:
June 2020, pp. 1–7. DOI: 10.1109/CyberSecurity49315.2020.9138883.

[TGN20] D. Thaler, E. Gershuni, and J. Navas.
Division by zero verifies as valid · Issue #133. Dec. 2020.
URL: https://github.com/vbpf/ebpf-verifier/issues/133 (visited on
10/22/2023).

[Vah+19] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler,
P. Druschel, and D. Garg.
"ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)."
In: *28th USENIX Security Symposium (USENIX Security 19)*.
Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1221–1238.
ISBN: 978-1-939133-06-9.

[WLC23] Y. Wang, D. Li, and L. Chen. "Seeing the Invisible: Auditing EBPF Programs in Hypervisor with HyperBee."
In: *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*. eBPF '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 28–34. ISBN: 9798400702938. DOI: 10.1145/3609021.3609305.