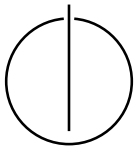
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Extending unikernels with a language runtime

Vanda Hendrychová



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

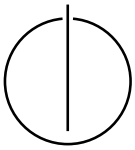
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Extending unikernels with a language runtime

Erweiterung von Unikernels mit einer Sprachlaufzeit

Author:	Vanda Hendrychová
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Dr. Masanori Misono
Submission Date:	22.5.2023



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

Munich, 22.5.2023 Vanda Hendrychová

Acknowledgments

I would like to thank Dr. Masanori Misono for advising me during my thesis as well as providing help with implementation challenges in the project. I would also like to thank Prof. Pramod Bhatotia for supervising the thesis and taking an interest in the project's progress.

Abstract

A key component of modern cloud computing is the possibility of deploying service components in isolated environments. Traditionally, cloud services were deployed in dedicated virtual machines. However, this approach introduces an overhead originating from running a whole operating system (OS) for every service instance. Containers are a popular approach to reducing the overhead by running multiple applications in a semi-isolated environment on the same OS kernel. However, the limited isolation raises security concerns.

As an alternative, unikernels can provide the same isolation level as traditional virtual machines while maintaining high performance and minimal overhead. Unikernels achieve this by directly linking necessary operating system components with the application into a minimal bootable image that can run directly on a hypervisor. However, this also means that unikernels are not easily extensible at runtime, as any code changes require recompilation of the unikernel.

In this project, we develop a method for safely loading binary programs into a unikernel at runtime. We adapt the Berkeley Package Filter (BPF) language runtime to run in unikernels built with Unikraft, ensuring isolation of the BPF binaries from the main application by virtualization. Further, we use the BPF runtime to implement a dynamic tracer for Unikraft unikernels, demonstrating how a unikernel can be simply and safely extended at runtime by loading and attaching BPF binaries to arbitrary functions. Benchmarking two different applications shows that the performance impact of including the BPF runtime is negligible and only minimally influences the image size. We believe that the BPF runtime provides a simple possibility for extending unikernels at runtime which will aid future development of unikernel tooling.

Contents

1

Introduction

The rising popularity of cloud computing has changed the way applications are deployed. Modern web applications consist of multiple layers and services with defined interfaces. In cloud environments, all components can be deployed separately to provide greater scalability and optimize resource usage by scaling services separately. A traditional approach to running services in the cloud is to use a dedicated virtual machine for each service (fig. 1.1). However, this approach introduces an overhead originating from running a whole OS for every service instance.

Containers were introduced to reduce this overhead by sacrificing some of the isolation: One could build and deploy minimal application images containing only the necessary user-space components by running multiple applications in a semi-isolated environment on the same OS kernel (fig. 1.1) [25]. Hardware interactions like block storage access or network stack are provided directly by the host OS and therefore do not need to be maintained inside the application container. However, the lack of isolation leads to an increased attack surface: A security hole in the container framework could allow a malicious or infected application to take over the whole host OS, including all containers running on it [23].

As an alternative to containerized applications, unikernels aim to combine the low resource impact of application containers with the high level of isolation of virtual machines by allowing applications to run directly on top of a hypervisor (fig. 1.1). Unikernels are minimal bootable images with only the necessary operating system components linked directly with the application. There is no division between user and kernel space. Syscalls are directly linked into the application as function calls, eliminating the need for context switching between user and kernel space. Therefore, unikernel applications can achieve higher performance than traditional virtualized applications. Further, unikernel images are often orders of magnitude smaller than containers or virtual machine images. This is because unikernel images only include the parts of an operating system that the application requires. For the same reason, the resulting virtual machines also boot faster, as fewer unnecessary OS components

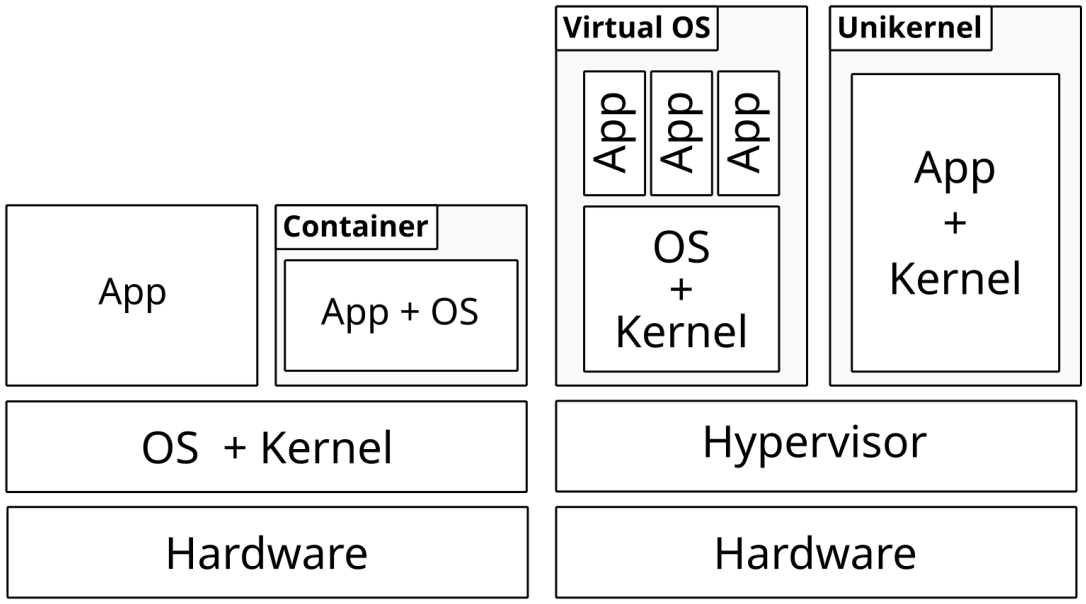


Figure 1.1: **Comparison of different virtualization approaches.** OS virtualization runs applications inside a guest operating system, including the kernel. Containerized applications use the kernel of the host operating system. Unikernels link together the application and a minimal kernel in a single image which runs directly on a hypervisor.

need to be loaded [16].

With all the advantages that unikernels offer, one might wonder why are unikernels not more widespread in the world of cloud computing. The answer is convenience: In a container or a virtual machine (VM) with an entire OS, it is easy to spawn other processes or inspect the application from a shell. Even if this additional process crashes, the application will continue to run uninterrupted. While there exist shells for unikernels, they are typically framework-specific and offer limited functionality. Further, the lack of process isolation makes adding untrusted code into the application more challenging. unikernels run in a single process with a shared address space, dynamically loaded binaries have full memory access and can potentially crash the whole application. Also, as the kernel is statically linked to the user application, any code changes require recompilation of the unikernel. In general, installing new programs in unikernels after compilation is not possible.

This also makes it difficult to debug unikernels. Many tools that can be used within a traditional operating system are difficult to include in unikernels as they need to be adapted and compiled into the unikernel application. Although this is possible, the lack of process and memory isolation means that a bug in the included program would lead to crashing or compromising the whole unikernel.

This project aims to develop a method for safely loading binary programs into a unikernel at runtime. Therefore, a language runtime will be adapted to run in unikernels built with a widely used unikernel framework.

The performance impact of including the BPF runtime will be benchmarked on different applications. Further, the included language runtime will be used for implementing a dynamic tracer for unikernels to evaluate the effectiveness and usefulness of the approach. This tracer will allow attaching new binaries to arbitrary functions at the runtime of a unikernel without needing to recompile and reboot.

Chapter 2 presents other research work that we considered – the background of our project and similar projects done in the past. Chapter 3 describes the general overview of the project, the design goals and challenges. Chapter 4 contains the high-level design of the project, whereas, in Chapter 5, you can find the implementation details. Lastly, the fulfilment of our design goals is evaluated in Chapter 6.

2

Background and related work

2.1 Unikernels

The term *unikernel* was first used in 2013 when introducing *MirageOS*[1] in [16]. Until then, the concept was mostly known under *library operating systems*. These were introduced in 1995 together with a new *exokernel* system architecture. According to [6], an exokernel is a minimal kernel on top of which run library operating systems. While in a traditional operating system, applications run in different processes and get provided with virtual memory, the exokernel architecture strives to maximize performance by handing more control over hardware resources to individual applications. The exokernel multiplexes hardware resources to allow multiple library operating systems to run simultaneously. While the exokernel provides a secure interface to the physical resources, each library operating system implements its own resource management (e.g. virtual memory). Managing resources at the application level allows more application-specific optimizations making the applications running as library operating systems more performant than running as a process within a traditional operating system.

The concept of library operating systems was revisited in 2011 by Microsoft research. *Drawbridge*[2] redesigned the Windows operating system as a library operating system. The project introduced this as a new lightweight way of sandboxing Windows applications. It also provided a new ABI (Drawbridge picoprocess) to run virtual machines in lightweight containers. Drawbridge could run the latest releases of commercial applications, such as Microsoft Excel or Microsoft PowerPoint [19].

Library operating systems faced many problems with hardware compatibility. *MirageOS* avoided this by targeting a standard hypervisor (Xen) instead of bare-metal hardware. For many years, the C language has been the obvious choice for writing a performant operating system. However, *MirageOS* decided to use OCaml due to its type-safety properties. While using a type-safe programming language was not new,

[1] `https://mirageos.org/`

[2] `https://www.microsoft.com/en-us/research/project/drawbridge`

MirageOS showed that unikernels still offer a performance advantage over traditional operating systems even when using a high-level type-safe language. The performance gain was achieved by application-specific optimizations and leaving out redundant code [16].

Like *MirageOS*, *OSv*[3] also focused on running on a hypervisor. However, they did not limit themselves to Xen and implemented support for multiple hypervisors. *OSv* authors recognized cloud computing as the primary use case for library operating systems and optimized *OSv* for the performance of network-intensive applications. *OSv* is written in C++11 and uses a similar interface as the Linux kernel to allow easier porting of applications, in many cases also running unmodified applications in various languages. Further, it explores modifying runtime environments (JVM) instead of applications [10].

IncludeOS[4] is a C++-based library operating system written entirely from scratch. It focuses on resource efficiency and virtualization platform independence. User applications in C++ need to include only a single header file without needing to select the OS components to build in. *IncludeOS* provides a customized GCC-based toolchain which automatically determines the needed operating system components at link time and includes a boot sector in the resulting image [4].

Unik[5] is a unikernel runtime that strongly focuses on cloud computing platforms and simplifies the build process of unikernels. It compiles user applications into machine images for popular cloud providers and hypervisors. *Unik* also supports Kubernetes and can be controlled via a REST API similar to Docker. Thanks to implementing the same endpoints as the Docker API, some tools should directly work with *Unik* images [12].

In this project, we use *Unikraft*[6]. It is a POSIX-compatible framework for developing unikernels [11]. While *Unikraft* offers special tools to create, manage and build new unikernel applications, it is built around Makefiles and does not require users to use their tools, such as `kraft`. *Unikraft* supports many libraries, includes over one hundred built-in libraries, and allows users to use and create external ones. Figure 2.1 shows the layered architecture of *Unikraft*. Before building the unikernel, users can configure their application – e.g. choose which libraries to include or which platforms to target. *Unikraft* provides various network stacks, filesystems, schedulers and platforms options. *Unikraft* applications can be compiled to run as Linux userspace processes, KVM virtual machines or Xen virtual machines [2].

This thesis focuses on extending unikernel frameworks that do not include a full traditional operating system. However, we can also find projects that choose a different approach of including a more complex operating system into the unikernel. One example is *Unikernel Linux*[7] (UKL) – a RedHat project which creates unikernels based on Linux [21].

[3] `http://osv.io/`
[4] `https://github.com/includeos/IncludeOS`
[5] `https://github.com/solo-io/unik`
[6] `https://unikraft.org/`
[7] https://github.com/unikernelLinux/ukl

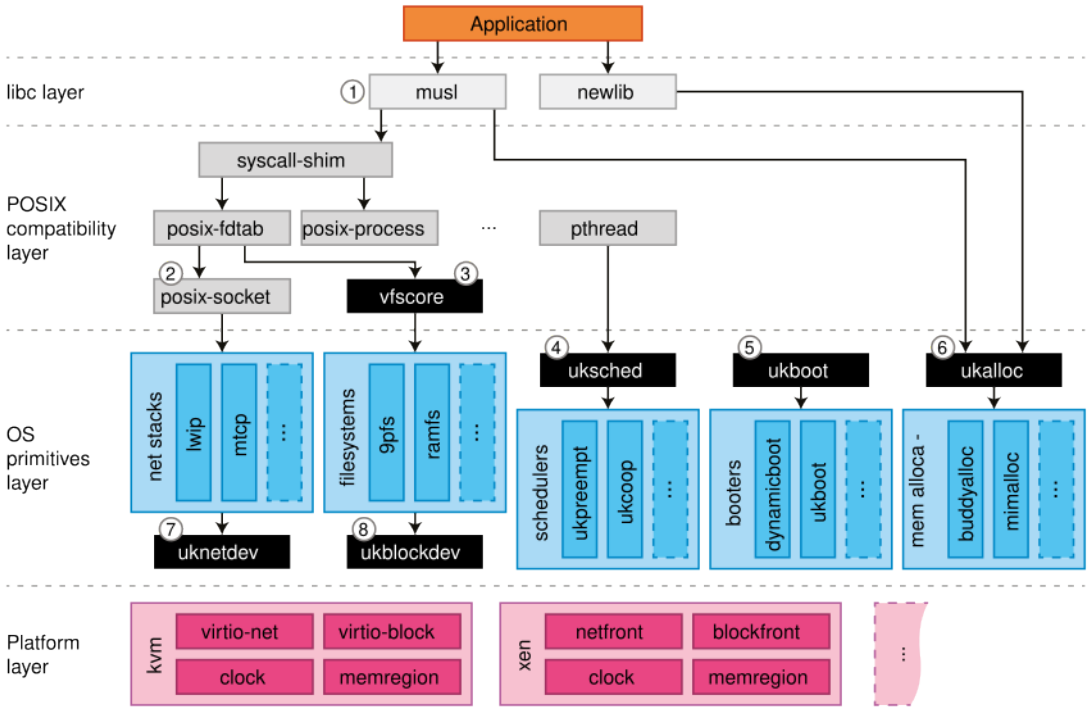


Figure 2.1: **Unikraft architecture.** Unikraft is POSIX compatible and designed in layers, allowing to choose different options for network stacks, filesystems, schedulers etc. Unikraft unikernels can be compiled for KVM, Xen and linux userspace. Image and information taken from [2].

Linux Kernel Library[8] (LKL) takes the Linux kernel code and provides it as a library usable in other applications. It allows applications to run on different operating systems and, at the same time, use features from the Linux kernel. To achieve this, the project defines an API and requires application developers to provide a small set of platform-dependent primitives [20].

2.2 Language runtimes

We consider a language runtime as the runtime environment and tools needed to execute programs in a given language on the target system. A common language runtime includes the most fundamental set of services needed for executing programs. Dynamic language runtimes support more features and offer better support for scripting languages thanks to the possibility of interpreting code on the go as the developer creates it without recompiling the whole program. Scripting languages, such as Lua, are very popular for extending compiled applications [5].

One example of a language runtime is WebAssembly[9]. WebAssembly describes an instruction set for a virtual machine. It aims to be a memory-safe and effective compilation target for web applications. In particular, WebAssembly offers a sandboxed execution environment and enforces security policies [26]. While we would like to explore including a WebAssembly runtime in unikernels, we decided to use BPF in this project due to its possibility to be integrated with a verifier.

BPF was designed at the Lawrence Berkeley Laboratory as a packet filter. At the time, there were other packet filters for Unix, but they were implemented in the user space. Due to the excessive copying of data between the kernel and user space and stack-based design, their performance could have been better. BPF provided a new optimized design with a register-based virtual machine included directly in the kernel to allow quick discarding of unwanted network packets [17].

Recently, the BPF virtual machine has been revisited to add more use cases to this in-kernel sandbox, and extended Berkeley Package Filter (eBPF) was created. eBPF can be used with more than just network packets by hooking the programs to different events in the kernel. In Linux, this includes tracing function calls, memory accesses, scheduler events and many more [8]. Figure 2.3 shows the variety of hardware and software events that BPF programs can be attached to in Linux. Brendan Gregg mentions on his blog [8] how eBPF can now be used for DDoS mitigation, dynamic tracing of Linux kernel events or intrusion detection. He focuses mainly on observability tools created with BPF Compiler Colection (BCC). These include cache statistics, monitoring TCP connections, block device and filesystem I/O latency and tracing specific syscalls.

The eBPF runtime in Linux includes a verifier, which makes sure that programs are safe before attaching them to kernel events. BPF programs communicate with the kernel indirectly using BPF maps. Different types of programs can access different

[8] https://github.com/lkl/linux

[9] `https://webassembly.org/`

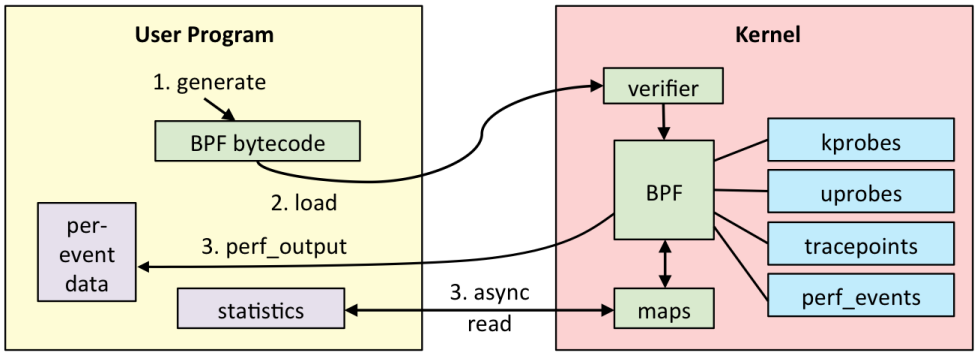


Figure 2.2: **eBPF architecture in Linux.** BPF bytecode needs to pass the verifier to be attached to kernel events. Programs can communicate either by sending per-event data or saving statistics in BPF maps. Image and information taken from [8].

BPF maps to provide isolation between BPF programs. BPF maps can contain different datatypes including arrays and histograms. Figure 2.2 shows how eBPF can be used for observability programs. The user program generates BPF bytecode and sends it to the kernel with a syscall. Within the kernel, the program is first checked by the verifier. In general the verifier checks the safety of the program and rejects unsafe programs. Safe programs are attached to selected events such as *kprobes* for tracing kernel functions and *uprobes* for user-space tracing. In Linux, the verifier considers unsafe any programs containing loops (except for sufficiently small loops that can be unrolled and still fit within the maximum amount of allowed instructions [7]) or backwards branches [8].

According to [15], there have been security incidents indicating that a verifier by itself might not be sufficient. While the design of the verifier might be safe, real-world implementations might contain security holes. Therefore, to further isolate the BPF programs from the kernel, the *MOAT* project offers a system for protecting memory using Intel Memory Protection Keys (MPK).

The *PREVAIL* verifier, introduced in [7], acknowledges the shortcomings of the Linux eBPF verifier. These include high amount of false positives (safe programs that do not pass the verifier) and very limited support for loops, where developers are often forced to experimentally determine bounds that would be accepted by the verifier or modify them in case the loop body grows. Further, programs are analyzed by exploring all possible paths, meaning that programs containing many nested branches lead to exponential time complexity. *PREVAIL* offers a solution of using a different abstract domain to track values as well as a mechanism of joining paths to avoid path explosion.

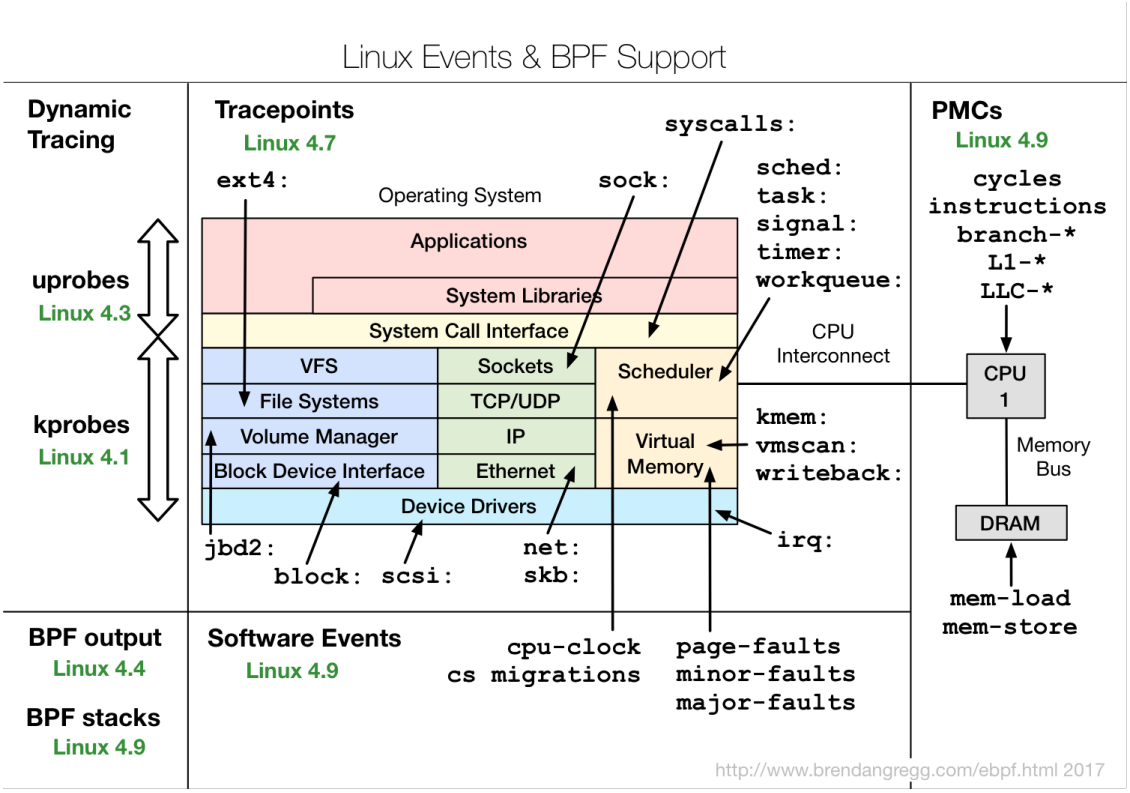


Figure 2.3: **eBPF events in Linux.** In Linux, eBPF programs can be attached to many events – apart from tracing probes, the events include also hardware and scheduling events. Image and information taken from [8].

2.3 Tracing in Linux

Tracing is a complex task: For example, in Linux there are many tools and mechanisms to allow shedding some light on what happens in the programs as well as the kernel at runtime. Tracing data can originate from either hardware and software events. Here we focus on the software side. Software data sources include *probes* and *tracepoints*. Tracepoints are statically compiled into programs and need to be activated to collect data. Probes, on the other hand, serve for dynamic tracing. They are not present in the program at compile time and are inserted by modifying the programs instructions at runtime [14].

Kernel functions are traced with *kprobes* and user-space functions with *uprobes*. Both work in a similar manner, using breakpoint handlers to execute the tracing code. Upon registering a new probe, the first instruction of the target function gets replaced by a breakpoint instruction (`int3`). The original instruction from that position gets saved along with the registered probe into a hash list. Once the target function is called, a breakpoint is invoked and a breakpoint handler finds the registered probe and executes it together with the saved instruction that was replaced [13].

These probes are executed at the entry point of functions, whereas *return probes* (*kretprobe* and *uretprobe*) are executed at function exit points. When a new return probe is registered, the return address of a function is replaced by a trampoline that executes the registered probe and then returns to the original return address [13].

Ftrace is a Linux kernel internal tracer which allows debugging and analyzing performance in the kernel-space. Enabling ftrace support in the kernel means that the kernel will be compiled with the GNU Compiler Collection (GCC) profiling flag `-pg` [9].

The GCC compiler has multiple architecture-dependent options for profiling support (here we focus on `x86_64`). The `-pg` flag enables profiling in general, which is necessary for all the other options to have an effect. The `-mrecord-mcount` option makes the compiler insert a call to a special function `mcount` at the beginning of every function. Further, instead of the call to `mcount` , we can add `-mnop-mcount` which replaces the call to `mcount` with a NOP instruction in the size of the function call in order to be replaced later. While the call to `mcount` is typically placed after the initalization of the stack frame, the `-mfentry` flag changes the location to be placed before the prologue of the function – as the first instruction of the function [27]. Other architectures feature different sets of options, e.g. `-mnops=num` inserts *num* of NOP instructions before every instruction [1].

According to [9], ftrace is highly configurable and allows setting up static tracers that collect data on every function call or dynamic tracing which means by default no data is collected but it can be dynamically changed to allow per-function. This works by using different implementations of the `mcount` function. For static tracing, `mcount` is more complex, whereas with dynamic tracing configured, it contains only a return. This ensures that dynamic tracing causes only minimal overhead. At runtime, these placeholder calls can be replaced by function calls of other ftrace functions. Tracing data

from all ftrace tracers are saved into a a special `tracefs` filesystem.

2.4 Tracing in Unikraft

There is already a tracing mechanism in Unikraft but it works with statically included tracepoints. A user has to register a tracepoint handler and include a call to it within the target function's source code. These tracepoints are only compiled into the debug image and the information collected by the tracepoints can be accessed only through GNU Project debugger (GDB) [3].

Including the tracepoints only in a debug image makes sense in terms of image size but realistically users might need to trace their applications also outside of a debugging session. Furthermore, every change to the tracepoint handler or adding a new tracepoint requires recompilation. In contrast to this, we implement a dynamic tracer which allows to add tracing probes at runtime and can also be used on a production image without debugging information.

2.5 BPF in Unikraft

There is already an existing project on GitHub which includes a BPF runtime into Unikraft. The *ukubpf* project uses uBPF as a BPF runtime and includes a BPF program code statically in the source code [22]. According to [24], uBPF is a userspace implementation of a eBPF runtime. While the Linux implementation is distributed under the GPL license, uBPF uses the Apache-License to allow inclusion in more projects. Apart from an interpreter, the project also includes a just in time compiler for `x86_64` and `Arm64`.

Unfortunately the ukubpf project was developed for an obsolete version of Unikraft and could not be updated to a more recent version of Unikraft. However, we drew some inspiration and used uBPF as the included runtime for this project.

2.6 Extending unikernels at runtime

As unikernels include only the most basic parts of operating systems, they generally lack a separate shell interface or implement a custom interface. However, an unpublished article [18] presents a project which extends Unikraft unikernels with a minimal shell interface that can be embedded into existing unikernels. We will further reference this project as *Xshell*. Xshell is explicitly designed for unikernels, so its built-in commands make use of the static linking of unikernels. It also enables installing new binaries at runtime and offers memory isolation based on Intel MPK. We extend Xshell by providing a sandboxed execution environment and use Xshell as an interface to our project.

3

Overview

The focus of this project is to develop a method for safely loading binary programs into a unikernel at runtime. Therefore, we explore how to adapt a language runtime to run in unikernels.

3.1 Requirements for binary instalments in unikernels

First we focused on the general requirements that a mechanism for installing binaries in unikernels should provide. We identified such requirements: *safety, performance and sandboxing*.

Safety in this context means making sure that the injected code does not crash and finishes in finite time. This is important because the crash of an injected binary would cause crash of the whole application (since unikernels only use a single process).

Performance is a key property of unikernel applications, and we need to ensure that any mechanism used for binary installation does not compromise performance or, at the very least, keeps performance limitations to a minimum.

Sandboxing is essential to protect the main application since traditional process isolation cannot be used in unikernels. We require the binary instalment mechanism to provide some form of sandboxing to ensure the security of the main application.

3.2 BPF as a sandbox

We chose to include a language runtime to provide a sandboxed environment for binary instalments. The Linux kernel now includes a BPF runtime for similar purposes. BPF also fulfils the previously stated requirements.

In the Linux kernel, the BPF runtime is coupled with a BPF verifier. Before attaching the BPF code, it is verified and only code that passes the verifier is executed. The verifier checks that the code contains no loops or unreachable instructions and that the code

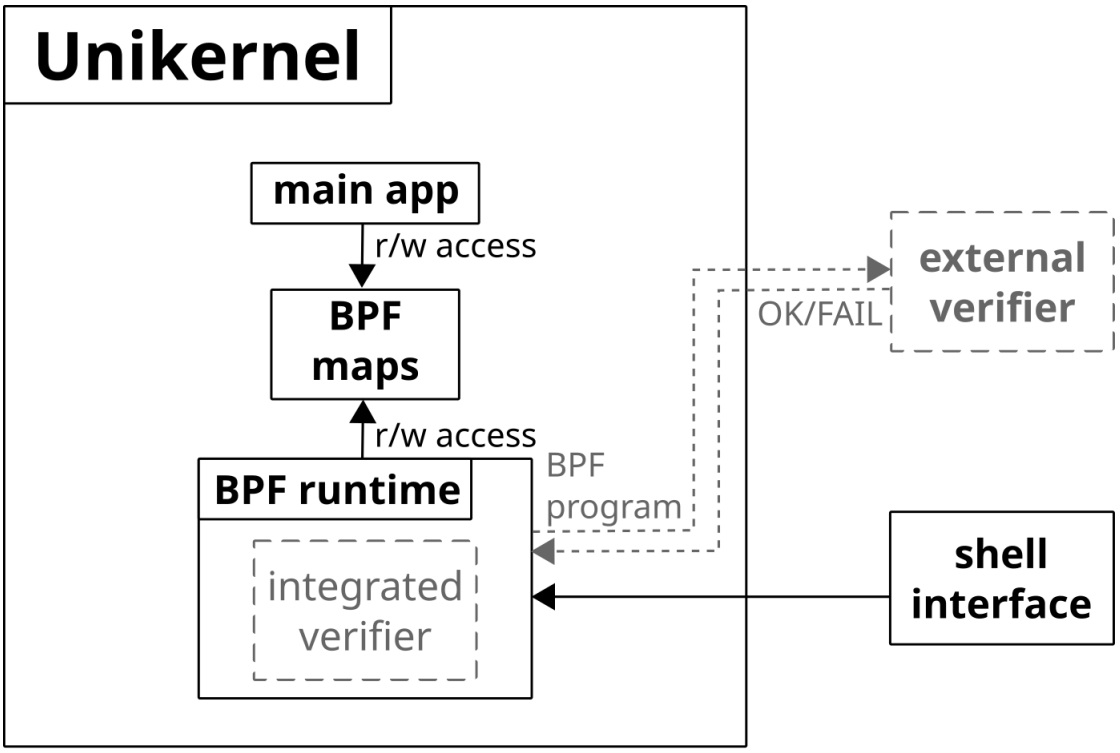


Figure 3.1: **BPF runtime in a unikernel.** BPF programs communicate with the main application through shared BPF maps. Shell interface can be used to load and spawn BPF programs at runtime. An internal or external verifier can be integrated to ensure only safe programs can be executed in the unikernel.

does not crash or leak kernel data. The possibility of integrating a BPF verifier helps us fulfil the **safety** requirement.

While BPF code can be interpreted, it can also be compiled just in time or ahead of time to increase **performance**.

Sandboxing is provided by limiting memory access of the BPF program only to specific parts of memory. Furthermore, regions of the accessible memory can be protected by the verifier.

3.3 Use case: Tracing unikernel applications

For demonstration purposes, we show how to use the BPF runtime to implement a dynamic tracer. We want to be able to load executables at runtime and attach them to arbitrary function entry points. We set the following requirements for the dynamic tracer:

- At runtime specify traced functions.
- At runtime choose actions to be done for each traced function.
- Do not change traced function behaviour or register contents after tracer actions are executed.
- Any amount of functions can be traced at the same time.
- Actions associated with each function can be modified.

3.4 Design goals

We identified three design goals that we can use to evaluate the design. Firstly, we aim to *minimize user code changes*. Secondly, we want to provide a *simple command line interface*. Lastly, we focus on making the design as *performant* as possible so that it can be included by default in production images.

3.5 Design challenges

We identified three design challenges. Firstly, there is no common interface for unikernels. Secondly, unikernels run in a single process and thus lack isolation. Lastly, the chosen language runtime is very limited in order to provide security features. In this section we explore potential solutions to these challenges.

3.5.1 Challenge #1: No common interface

Unikernels are very lightweight and specific. Unlike a shell in a traditional operating system, unikernels do not have a standard interface integrated within them. To make better use of the included BPF runtime, the unikernel should have some shell interface

that can allow users to manipulate the BPF runtime from the outside without modifying the main application.

We can solve this problem by integrating with the `Xshell` project which provides a minimal shell interface to unikernels. We do this by adding custom shell commands calling chosen functions from our library.

3.5.2 Challenge #2: Lack of isolation

Unikernels run in a single process with a single address space. This makes sense since the unikernel is supposed to be specialized to run exactly one application. However, in our project we try to install untrusted binaries at runtime to extend the main application. Untrusted binaries could potentially read or modify the main application memory in undesirable or even malicious ways. Therefore, we need to isolate the installed binaries and restrict their memory access.

In uBPF, the application memory is protected by a bounds checker. This bounds checker restricts the memory regions a BPF program can access and protects the main application from crashing because of a faulty BPF program. We include a shared storage (BPF maps) to allow communication between different BPF programs and the main application.

Even though the bounds checker restricts memory access to protect the main application, a BPF program attached to the application could still block it with e.g. infinite loops. We can provide further safety guarantees, such as that the program terminates in finite time, by verifying BPF programs before execution. Currently, we assume that the user supplies only verified BPF programs. In the future, the project could also include a verifier. We suggest either an internal verifier integrated into the BPF runtime or an external verifier outside of the unikernel (see figure 3.1).

A verifier is a potentially heavy application that we might not want to include directly into the unikernel. An external verifier would be advantageous to save resources within the unikernel and reduce image size. On the other hand, exposing more interfaces to the outside of the unikernel means bigger attack surface. Additionally, we would need to ensure that the verifier cannot be compromised or replaced by a malicious one. An internal verifier would eliminate the need to communicate with the outside of a unikernel for program verification.

3.5.3 Challenge #3: Limitations of BPF

Ensuring safety comes with limitations. BPF programs need to be written in a safe way to ensure that they do not block or crash the main application. A newly installed BPF program could for example contain an infinite loop that would block the execution of other commands or the main application. The Linux kernel verifier avoids this problem by entirely disallowing loops in BPF programs. The problem can be solved with a more advanced verifier. Additionally, supporting loops would also increase the time and space requirements of the verifier which might not be favourable for a kernel built-in.

Memory access of BPF programs is also limited. With the BPF runtime we selected, there is a very strict bounds checker which does not allow the BPF programs to access any memory except for arguments directly passed to the program. Accessing other parts of memory could be useful to share data between different BPF programs or between BPF programs and the main application.

To address these limitations, the BPF runtime provides the possibility to register helper functions that can be called from BPF programs. These helper functions do not share the limitations of BPF and also do not provide any further isolation. As they are defined statically, they form a part of the kernel code and should be reviewed by the developer before compile-time. In our project, we provide a set of simple built-in helper functions to extend the functionality of BPF programs while also trying to limit the possible negative effects the programs could have.

4

Design

The project consists of two parts. Firstly we integrate an eBPF runtime and secondly we use this runtime to implement a dynamic tracer.

4.1 BPF runtime

To integrate an eBPF runtime, we chose to use an existing project and include it as a library. Further, we extend it with adding BPF helper functions and an interface which simplifies the initialization of BPF virtual machines. We also add storage that can be accessed from both the BPF code and the main application using supplied helper functions.

Figure 4.1 shows how the BPF runtime integrates with the unikernel. The BPF runtime is a library added into the main application. This means that BPF programs can be loaded and executed directly from the main application or through a minimal shell interface that attaches to the main application (the `bpf_exec` command).

4.1.1 Memory access

BPF programs are limited to directly accessing only the memory that we pass to them (arguments). This is a limitation set by the BPF runtime we selected since it implements a bounds checker. It is good for sandboxing purposes, but it limits the possibilities of communication with the kernel or between different BPF programs. In Linux, BPF programs can share data using BPF maps. BPF maps are a shared associative memory accessible by both the kernel and the BPF programs. Kernel can access them directly and BPF programs can access them through predefined helper functions.

In Linux, there are multiple BPF maps containing different data structures. The storage we designed is a single two-level hashmap where every element needs two keys to be accessed. All BPF programs can access all regions of this hashmap. We never give

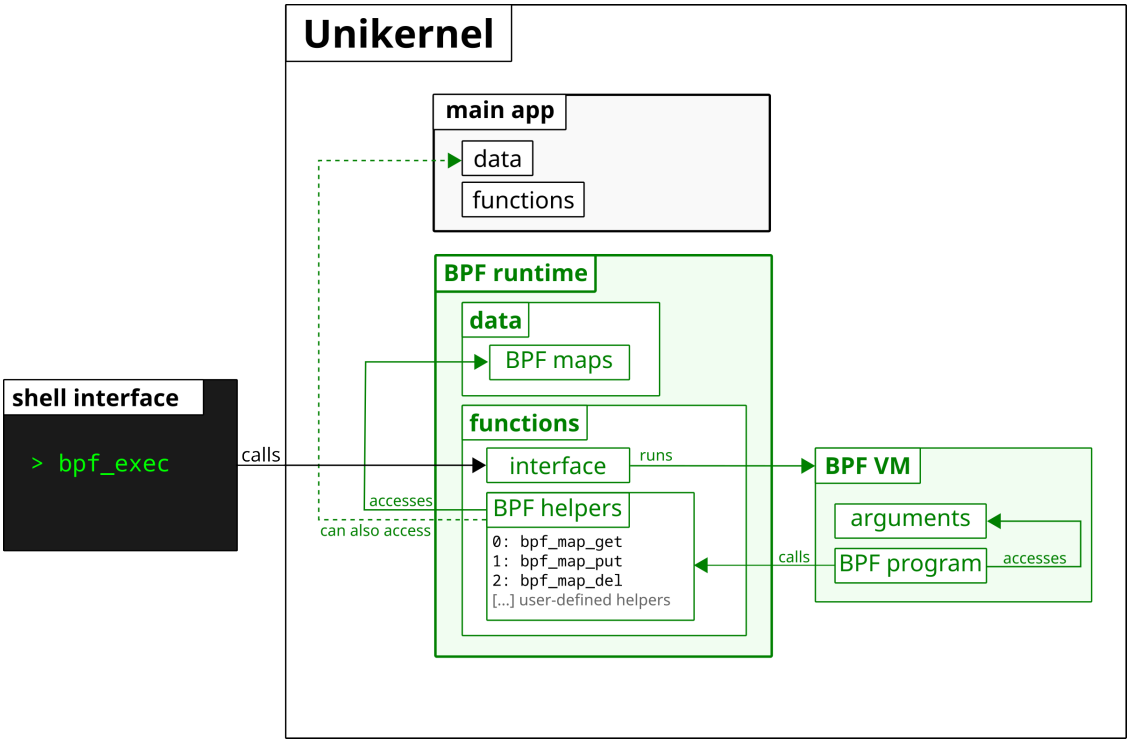


Figure 4.1: **BPF runtime in a unikernel.** BPF programs run in BPF virtual machines and can not directly access memory. BPF programs can also call predefined helper functions which have access to the whole address space. We define helpers for accessing BPF maps and allow users to define their own application-specific helper functions. BPF programs can be executed from attached shell interface with a command `bpf_exec`. Added BPF runtime highlighted in green.

the BPF program the option to manipulate the data directly. Instead, all data reads and modifications have to go through helper functions.

Figure 4.1 shows how the BPF program runs in a BPF virtual machine and can directly access only its arguments. The BPF program calls helper functions which in our design access the data stored in BPF maps. However, in general helper functions have access to all data in the unikernel – not only the BPF maps but also the main application data and all attached filesystems. Every virtual machine contains exactly one program and a fixed set of helper functions and can be executed multiple times with different arguments.

4.1.2 Helper functions

We designed several helper functions to manipulate BPF maps and we also allow users to define their own custom helper functions in addition to the predefined ones. We provide a function to add or delete these custom helpers for any virtual machine created in the future. Since the BPF helpers are not verifiable in our design, we try to provide only a small set of primitives that need to be trusted (BPF helpers are defined statically and are part of the unikernel so they are on the same level of trust as any other function in the main application).

We provide the following predefined helper functions:

- `bpf_map_get(key1, key2)`
  - Get the value stored in map `key1` under `key2`.
- `bpf_map_put(key1, key2, value)`
  - Create or update the value stored in map `key1` under `key2`. After this operation `BPF_MAP[key1][key2]` will contain `value` regardless of if it previously contained another value or not.
- `bpf_map_del(key1, key2)`
  - Delete the value stored in map `key` under `key2` regardless of its previous (non)existence.

Other helper functions that might be useful in the future are string manipulation functions from libc which are not available in BPF programs.

4.1.3 Initializing BPF virtual machines

BPF code is executed by a virtual machine. We provide a function `init_vm` which creates the virtual machine and registers all previously supplied BPF helper functions. This is how we make sure that users can comfortably add their own project-specific BPF helpers. On the other hand, it might also be useful to be able to create virtual machines with different sets of helper functions available. Optionally, the user can provide a custom list of BPF helpers which will get appended to the built-in helpers list.

We defined three built-in helper functions that receive the indices 0, 1 and 2 as seen in figure 4.1. Additional user-defined helpers receive indices 3 and higher. As new helpers get added in the future, the built-in helpers always end up first, starting from index 0 and have fixed indices depending on the version of the project.

4.1.4 Shell interface

To execute BPF programs using the runtime, we provide the `bpf_exec` command which allows users to select a BPF program in the unikernel's file system and pass string arguments. Every BPF program also returns an integer value, which is printed into the console if the program successfully finishes its execution.

4.2 Dynamic tracer

Using the previously integrated BPF runtime we design a dynamic tracer which allows attaching BPF programs to arbitrary function entry points. As seen in figure 4.2, the tracer extends the BPF runtime with additional helper functions and shell commands.

4.2.1 Helper functions

We designed the following helper functions to support the tracer functionality:

- `bpf_notify(function_address)`
  - Issue a print statement whenever the traced function is invoked.
- `bpf_get_ret_addr(function_name)`
  - Based on the given `function_name` return an internal identifier of the function.
  - When there is some error (e.g. function does not exist or is not traced) then return 0.

The tracer maintains a separate list of helper functions, that get added to the built-in helpers. This list is also extensible through the tracer interface and any helpers in the tracer list get added also to the list of user-defined helpers that is stored within the BPF runtime (as mentioned in previous section). This has the following consequences:

- Programs that are attached to function entry points by the tracer receive the tracer list of helpers.
- Programs which are executed with `bpf_exec` should also have access to the same helpers but at the same time the lists might be different if the list within the runtime was modified by another part of the unikernel.

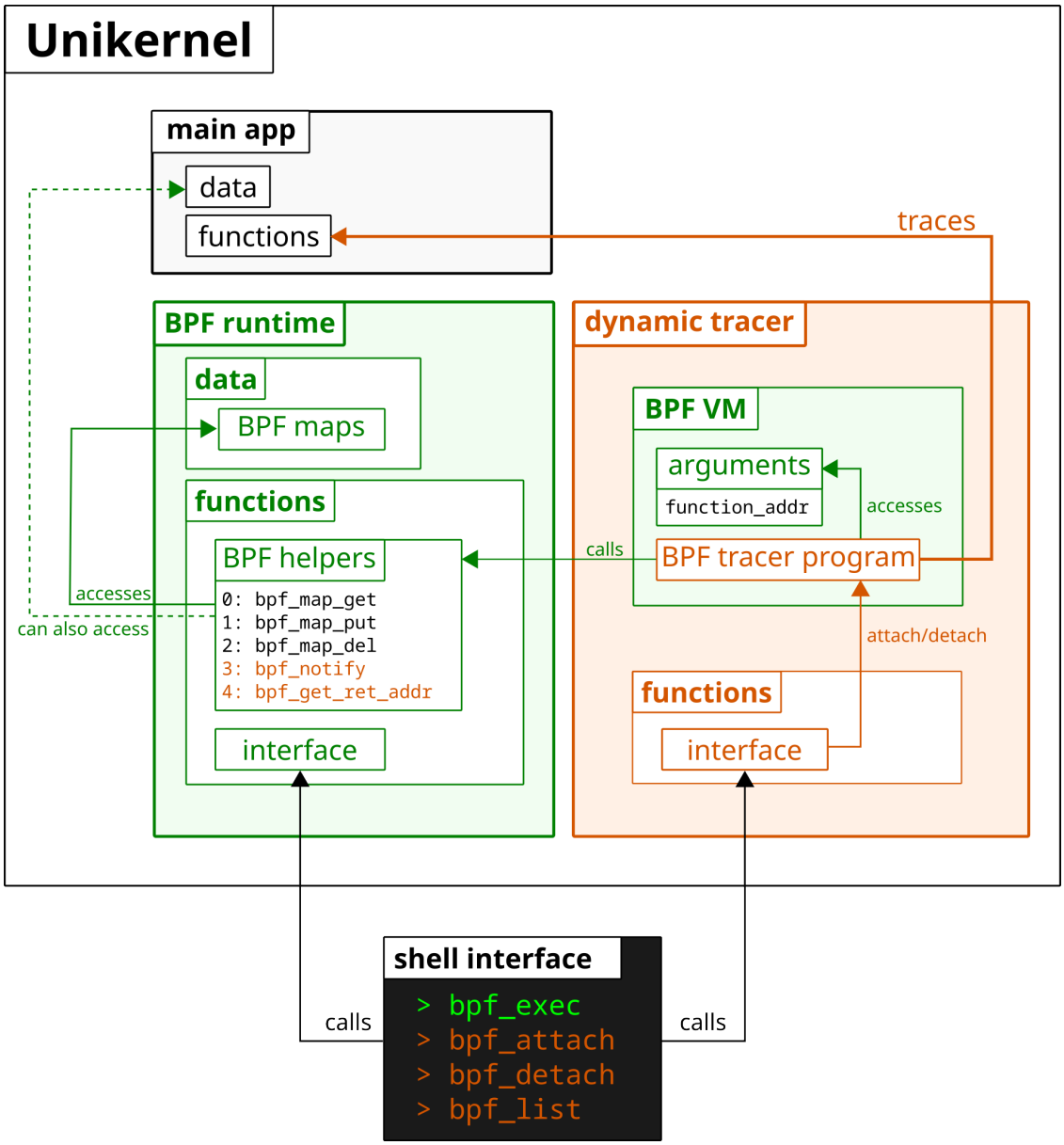


Figure 4.2: **Dynamic tracer using a BPF runtime.** We use the BPF virtual machines to trace functions in the main application. We define additional helper functions specific to the trace and additional shell commands to attach, detach and list BPF programs. This image is an extension of figure 4.1, tracer additions are highlighted in orange.

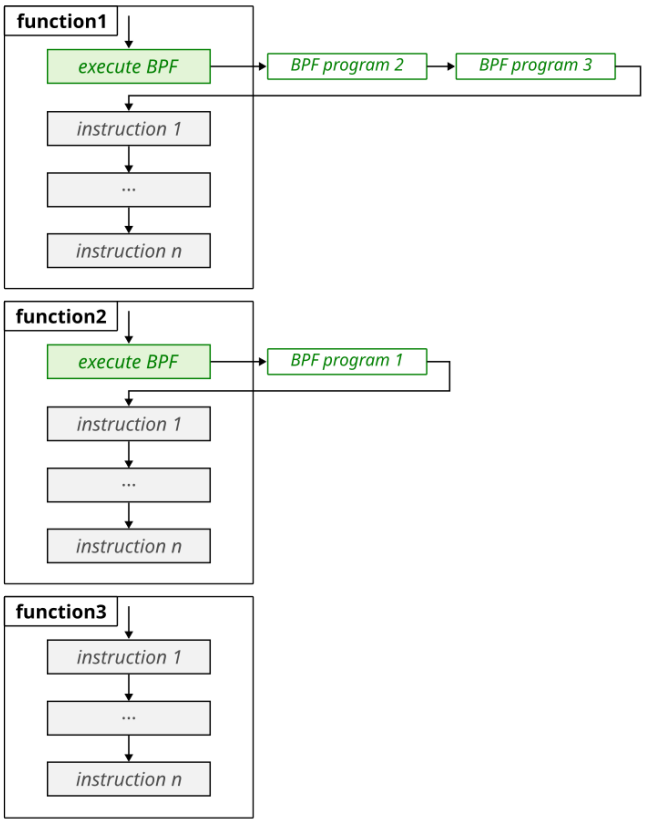


Figure 4.3: **BPF programs attached to function entry points.** The green parts show the injected code which takes care of executing the BPF programs in order of attachment before the rest of the traced function body.

- Programs executed with `bpf_exec` only get access to the tracer helpers after the tracer has been initialized (this happens automatically when any tracer command is executed - e.g. `bpf_list`).
- Tracer helpers have fixed indices which depend only on the version of the BPF runtime and not on changes possibly made by other parts of the unikernel to the runtime internal list. In the current version the tracer built-in helpers get fixed indices 3 and 4.

4.2.2 Using the tracer

As seen in figure 4.3, Each function can have multiple programs associated with it and any number of functions in the target application can be traced at the same time. We attach the BPF programs by injecting code at the beginning of the function. When the traced function is called, all attached programs are executed in the order they were attached. After all attached BPF programs are executed, the traced function resumes

and gets executed as well.

We store BPF virtual machines in a map by an identifier of each function. Every map entry contains a labeled list of BPF programs. Thus, we can not only attach programs but also detach. When all programs are detached from a function, it should return to the state before the first program was attached to it. Since every virtual machine can only contain one program, we actually save BPF virtual machines with a program loaded in them. This means that if we attach the same binary multiple times it will still end up as multiple virtual machines. It is also possible to attach the same binary to the same function multiple times and detaching a binary with that filename will detach all virtual machines with the same label.

The dynamic tracer can be manipulated by functions from within the kernel or through a minimal shell interface that we integrated with the project. We provide the following shell commands to manipulate the tracer:

- `bpf_attach <function_name> <bpf_filename>`
  - Attach BPF program `bpf_filename` to function `function_name`.
- `bpf_detach <function_name> [<bpf_filename1> <bpf_filename2> ...]`
  - Detach specified BPF programs from function `function_name`.
  - If no file names are given, detach all programs attached to this function.
- `bpf_list [<function_name> ...]`
  - List functions attached to `function_name` if supplied or to all functions if no `function_name` is given.
  - If multiple function names are given (separated with space), then the list is printed for all the chosen functions in the given order.

5

Implementation

5.1 BPF runtime

Since unikernels do not differentiate between kernel space and user space, we can use a user space BPF runtime. We decided to use the *uBPF*[1] project and make it into a unikraft library. To port it to Unikraft, we created a special Makefile containing variable definitions specific to the project (e.g. which files contain the source code and what compilation flags should be used). The unikraft library is available publicly on GitHub as `vandah/uk_ubpf` [2].

The *uBPF* project does not provide BPF maps or any other shared storage. We implemented BPF maps using a two-level hashmap with all keys and values being `uint64_t`. BPF maps in Linux can store more complex data types but the interface works differently. The built-in verifier makes it possible to give the BPF program a pointer to kernel memory where the BPF map is located. The verifier ensures that the BPF program can only access these specific regions.

Our project currently does not use a verifier and the *uBPF* project does not include one either. However, it protects kernel memory by implementing bounds checking. It makes sure that the BPF program only accesses its arguments and memory within the BPF program. If the helper function would have had returned a pointer to the kernel memory outside of these regions, the BPF program would still not be able to access it due to the built-in bounds checker. Therefore, we use simple data types and pass the data directly through the helpers. This way, the helper function is accessing the BPF maps directly whereas the BPF code can only access them indirectly through the helper.

The hashmap in this project is just a simple fixed size array of linked lists (hashing with chains).

[1] `https://github.com/iovisor/ubpf`
[2] `https://github.com/vandah/uk_ubpf`

```
int bpf_exec(const char *filename, void *args, size_t args_size,
             void (*print_fn)(char *str));
```

Figure 5.1: **BPF runtime interface for Xshell.** The `bpf_exec` command in Xshell is supported through providing a function that accepts a filename and an array of string arguments. The function further expects Xshell to supply a custom print function which would allow printing messages directly into the Xshell console.

5.1.1 Loading data at runtime

We need to load new programs at runtime so users can create the programs on-demand and add them to the unikernel without restarting. We can do this easily by attaching a filesystem to the unikernel. Users can then store their BPF programs in this filesystem and we can load them by filename. Unikraft has a built-in support for the `9pfs` filesystem which is what we decided to use. When using KVM, we can mount a folder from our guest OS filesystem and change its contents at the unikernel's runtime.

5.1.2 Integrating with Xshell

To integrate with Xshell we provide a simple interface, in which Xshell passes our interface functions the command arguments and a print function to print out messages into the Xshell console. Each function corresponds to one shell command. Using conditional compilation blocks, we only include the commands if the respective library is included.

In Xshell we can add new built-in shell commands. Xshell contains a function that recognizes the built-in commands by string comparison. When defining a new command, we first define the header of our interface functions and then call them with the right arguments that we receive from Xshell as an array of strings. For the BPF runtime we only include one built-in command `bpf_exec`. More details on the interface are described in figure 5.1.

5.2 Dynamic tracer

We use the BPF runtime to implement a dynamic tracer. The BPF code can be loaded from filesystem and then executed within unikraft. We attach a 9pfs volume to the unikernel and store all BPF programs there. Further, we also need to access the debugging symbols during runtime of the application in order to identify addresses of traced functions. To support multiple traced functions and multiple BPF programs per function, we implemented a key/value storage that we use as a map of BPF virtual machines. We further offer helper functions that access another key/value storage to allow exchanging data between the unikernel and the BPF program.

```
struct UbpfTracer {
  // { function_name -> function_address }
  struct DebugInfo *symbols;
  uint32_t symbols_cnt;

  // { (nop_address + CALL_INSRUCTION_SIZE) -> List<(label, ubpf_vm)> }
  struct THashMap *vm_map;

  // { function_address -> nop_address }
  struct THashMap *nop_map;
};
```

Figure 5.2: **Tracer data.** The tracer stores information about debugging symbols (addresses of symbols in the unikernel binary), as well as a VM map that contains information about which BPF virtual machines are attached to which functions and a NOP map mapping function starts to NOP addresses.

5.2.1 Tracer data

The tracer is stateful and therefore needs to store and access data about its state. We store the tracer state within a data structure `UbpfTracer` defined as shown in figure 5.2. Most functions expect a tracer argument to be passed to them, but we also provide several functions to be used as an interface to the tracer. These use a global instance `g_tracer` of the `UbpfTracer` structure and initialize it when needed.

The state of the tracer consists of debugging symbols (`symbols` and `symbols_cnt`), map of attached programs (`vm_map`) and where within the function did we attach the programs (`nop_map`).

The debugging symbols can be generated by a tool `nm` and Unikraft even has a configuration option to generate the symbols file at compile time. To make the symbols available to the unikernel, we need to copy them to the attached filesystem. For the sake of simplicty, the tracer expects the symbols to be in a file called `debug.sym` .

5.2.2 Attaching a program

The process of attaching a BPF program to a selected function is depicted by figure 5.3.

The tracer needs two arguments to attach a BPF program to a function: *function name*(name of the target function) and *bpf filename*(path to the BPF program that should be attached to the selected function).

Using the function name we search the previously loaded debugging symbols to find the function address. Next, we insert a call to a function that executes the attached BPF programs.

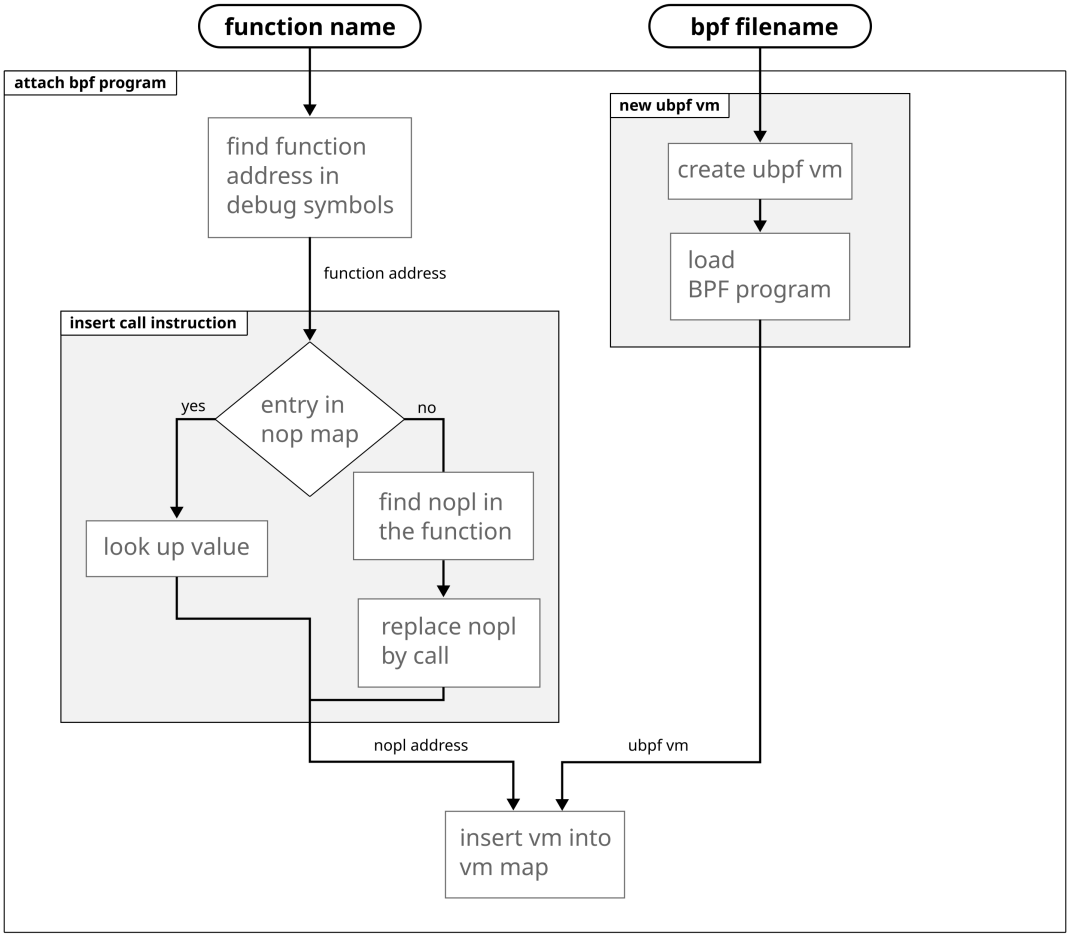


Figure 5.3: **Attaching a BPF program to a function.** A new BPF virtual machine is created for every program. With the help of debugging symbols we find the address of a NOP instruction that we replace by a call. We use this address to insert the new virtual machine into a map.

There are two common approaches to inserting code into functions:

- **insert a breakpoint instead of the first instruction**: at runtime replace the first instruction of a function with a breakpoint instruction and implement a breakpoint handler that also executes the replaced instruction at its end
- **insert extra `nop` instructions**: at or before compile time insert extra `nop` instructions at the start of a function that we intend to trace and then change these extra instructions at runtime

In this project, we choose the second option. We use profiling options of the GCC compiler to insert a 5 byte long nop instruction (for `x86_64` this is `nopl 0x0`). We require that users add the following compiler flags to their applications:

- `-pg` : enable profiling
- `-mrecord-mcount` : add a call to an `mcount` function at the start of each function
- `-mnop-mcount` : instead of a call to `mcount` , insert a `nop` instruction of the same size (5B)

The 5 bytes are just enough for inserting a `call` instruction with a relative address.

We could also just use the `mcount` function to launch our BPF programs but this would mean an extra function call at the start of each function in the application. By using `nop` instructions instead, we add less overhead to functions without any attached programs.

Figure 5.4 shows how the profiling options affect a program. In part **A** the program is compiled without profiling options and thus contains no extra instructions. Part **B** shows how the same program could look like when compiled with the above mentioned profiling options that insert `nopl` instruction at the start of every function. In part **C** the program was modified by the tracer and a BPF program `notify.bin` was attached to `function2` (figure 5.3 shows what happens between **B** and **C** in more detail).

The drawback of adding the instruction with the GCC profiling options is that the `nopl` instruction is not always inserted at the very start of the function. We observed that depending on the function, it can be inserted at offsets of 0, 4 or 8 bytes. This depends on whether the function in question calls other functions within its body or whether it needs to allocate space on stack for local variables.

Information about locations of these instructions should be available, however the only way we found to access them is to run the application once and let it create a `gmon.out` file which can then be parsed with `gprof` or manually. To avoid having to restart the application, we search for the `nopl` instruction manually at runtime. In our case it was these 5 bytes: `0f 1f 44 00 00` . Of course this can vary by architecture or compiler version but there should be a limited amount of available instructions to search for and we know the architecture at compile time which allows us to define a different byte-sequence depending on target architecture. In this project however we focus only on `x86_64` .

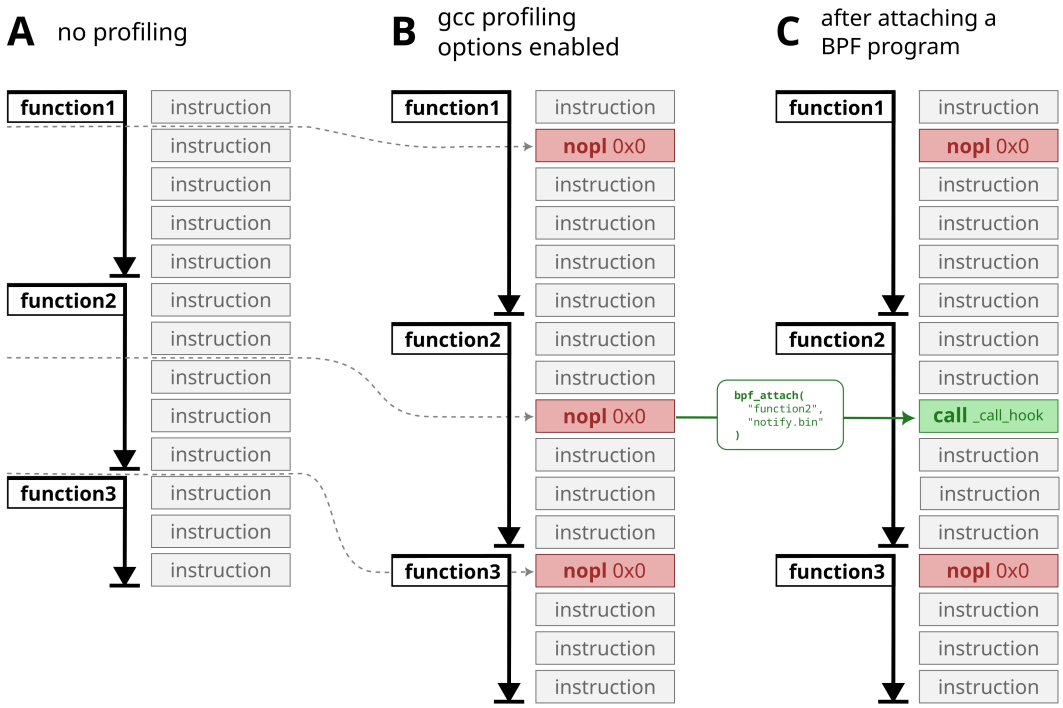


Figure 5.4: **Inserting `nopl` instructions and attaching a BPF program.** In part A the program is compiled without profiling, in part B profiling options caused extra NOP instructions to be inserted. A BPF program is attached to the program in part B and the resulting binary is shown in part C. (Instructions are not fixed-sized.)

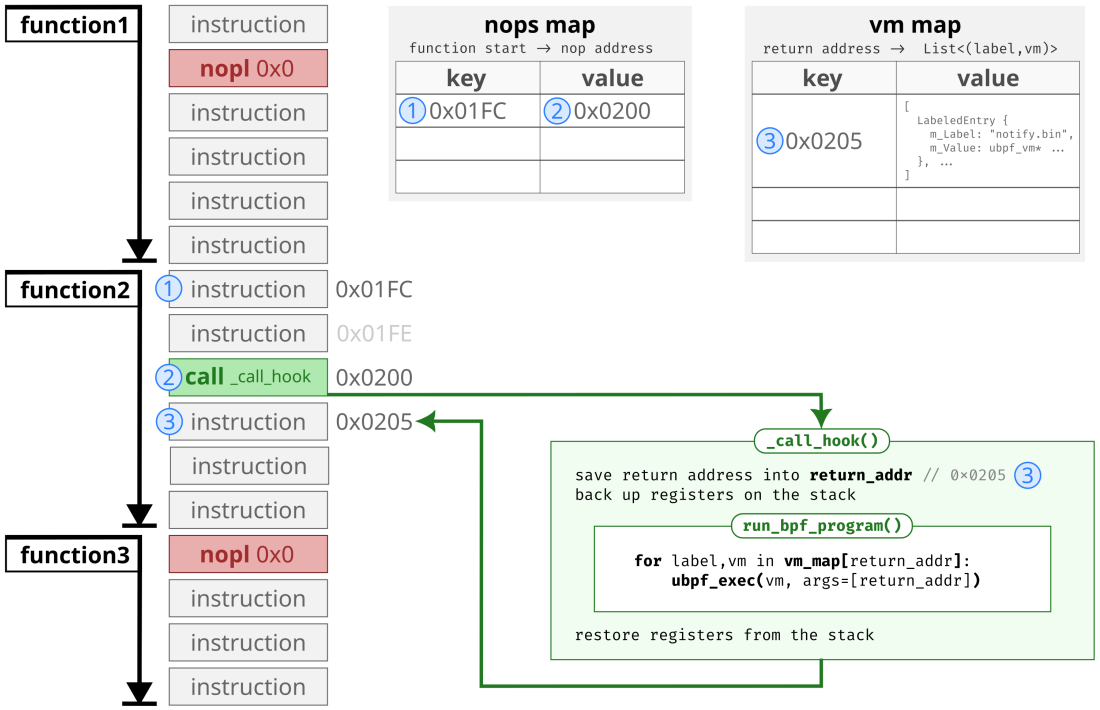


Figure 5.5: **BPF program attached to a function.** A BPF program `notify.bin` was attached to the function `function2`. The nops map now contains a mapping of the start of `function2` to the location of the `nopl` / `call` instruction. The vm map contains a mapping of the instruction after the `nopl` / `call` to a list of BPF virtual machines, now containing one entry labeled with the name of the attached program: `notify.bin`. The `nopl` instruction in `function2` has been replaced by a `call` instruction. When `function2` is executed, current registers are saved on the stack, all attached BPF programs are executed and registers are restored from the stack. Instructions are not fixed-sized which is why the addresses in the figure do not increase by a fixed amount of bytes.

When attaching the first program to a function, we search for the above mentioned sequence of bytes so that we can replace it with a `call` instruction to `_call_hook`. Once replaced, we would not be able to find it when attaching a second program. Hence, we save it into `nops_map` which maps function start addresses to the corresponding `nopl` addresses (shown in 5.5). When attaching a new program to a function we can check if the location is already known and search for the bytes in case it is not (shown in figure 5.3).

Next, we need a lookup key for obtaining the list of BPF programs which should be executed for a particular function. As the lookup key we chose the address of the instruction after the `nopl`, since it is equal to the return address of the `_call_hook` call and therefore trivially obtainable.

5.2.3 Saving register contents

Most registers in the used calling convention are callee-saved, meaning that a called function is responsible for taking care of backing up and restoring any registers it modifies. However, a small number of registers is caller-saved. Ignoring the caller-saved registers would lead to damaged register contents and application crashes. The compiler would normally take care of this and insert the appropriate instructions before the function call. However, due to the five bytes space limit of the `nopl` instruction, there is no space for additional operations. Therefore, instead of calling `run_bpf_program` directly, we wrap it in an assembly function `_call_hook` which first backs up all registers, then calls `run_bpf_program` and finally restores the register contents to their state before the function call.

As stated in the previous section, we need the return address of the `_call_hook` call as a lookup key inside the `run_bpf_program` function. Since entering `run_bpf_program` would overwrite the return address to an address inside `_call_hook`, the original return address is saved by `_call_hook` in a variable that is accessible from `run_bpf_program`.

5.2.4 Listing and detaching programs

When we attached a BPF program, we saved a label (filename) together with the `ubpf` virtual machine. This makes it possible to *list* BPF programs attached to functions and *detach* specified BPF programs from functions.

5.2.5 Shell interface

The tracer can be manipulated either from the main application or through a shell interface. We provide an interface that can be used to add custom shell commands into Xshell. We extend the set of Xshell commands we already defined for the BPF runtime by `bpf_attach`, `bpf_list` and `bpf_detach` which allow attaching and detaching BPF programs and listing attached programs. Commands accept string arguments and Xshell can pass a custom print function which will be used to print out messages directly into

```
int bpf_attach(const char *function_name, const char *bpf_filename,
               void (*print_fn)(char *str));
int bpf_list(const char *function_name, void (*print_fn)(char *str));
int bpf_detach(const char *function_name, const char *bpf_filename,
               void (*print_fn)(char *str));
```

Figure 5.6: **Dynamic tracer interface for Xshell.** Functions accept a string arguments and a custom print function to allow printing directly into the attached shell interface instead of the unikernel's standard output.

the Xshell console. See figure 5.6 for details of the function interface. The `bpf_list` command in Xshell can also be invoked with a list of function names. This results into multiple calls to the `bpf_attach` function that we define here.

5.3 Using helper functions in BPF programs

BPF code can be created from many different sources. Since this project uses C for the helper function implementation, we show how to use the helper functions in C that is later compiled into BPF. The problem with helper functions is that we need the BPF code to call them by specific addresses that we define when setting up the BPF virtual machine. In BPF assembly this would be e.g. `call 0` for the first helper function or `call 1` for the second helper. This showed to be problematic in C. We found a way to simplify creating function calls to the helper functions with C macros. In the following example (figure 5.7) we created a header file with the list of our helper functions. Figure 5.8 shows how these helper functions can be used within a BPF program. In this example we implement a BPF program to count the function calls of a certain function. The tracer passes the function address as an argument. We use the function address as a key for the BPF map and the second-level key is an arbitrarily chosen constant that we define in another header file and share among our BPF programs. The program first retrieves the current value from the BPF map. Absence of the value in the map is indicated by the value `UINT64_MAX` in which case we assume the count to be zero. Next, we increment the counter and store the new value in the map.

The program only updates the BPF map but does not show any output. To access the resulting values, we need to check the BPF map. We can create a simple BPF program to do this. Figure 5.9 shows a program that takes a function name as an argument and using a helper function `bpf_get_ret_addr` retrieves the first-level key into the BPF map for this function. The program returns the amount of times the requested function has been called since attaching the `count.bin` program that we defined earlier. If the function is not traced or not known or the user does not give any argument, the program returns -1. When executing the program with `bpf_exec`, the return value will be printed into the shell.

```c
#include <stdint.h>

// bpf map helpers
#define bpf_map_get ((uint64_t(*)(uint64_t key1, uint64_t key2))0)
#define bpf_map_put ((void (*)(uint64_t key1, uint64_t key2,\
                                                uint64_t value))1)
#define bpf_map_del ((void (*)(uint64_t key1, uint64_t key2))2)

// tracer helpers
#define bpf_notify ((void (*)(uint64_t function_address))3)
#define bpf_get_ret_addr ((uint64_t(*)(const char *function_name))4)
```

Figure 5.7: **Defining BPF helpers with specific addresses in C.** Calls to BPF helper functions need to be compiled in a way that the functions have addresses that we define within the BPF virtual machine. We use C macros to achieve a function-like syntax for helper functions and the correct instructions being emitted.

```c
#include "bpf_helpers.h"
#include "config.h"

int bpf_prog(void *arg) {
    struct UbpfTracerCtx *ctx = arg;
    uint64_t count = bpf_map_get(ctx->traced_function_address, COUNT_KEY);
    if (count == UINT64_MAX) {
        count = 0;
    }
    count++;
    bpf_map_put(ctx->traced_function_address, COUNT_KEY, count);

    return 0;
}
```

Figure 5.8: **Example BPF program: Saving values into a BPF map.** The program `count.bin` counts the amount of times a function has been called by incrementing a counter in the BPF map.

```c
#include "bpf_helpers.h"
#include "config.h"

int bpf_prog(void *arg) {
    if (!arg) {
        return -1;
    }
    uint64_t addr = bpf_get_ret_addr((const char *)arg);
    if (addr == 0) {
        return -1;
    }
    uint64_t count = bpf_map_get(addr, COUNT_KEY);
    if (count == UINT64_MAX) {
        count = 0;
    }

    return count;
}
```

Figure 5.9: **Example BPF program: Getting a value from a BPF map.** The program `get_count.bin` uses a tracer-specific helper function to get the map index and retrieves the count value from the BPF map for a function specified by function name.

```c
#include "config.h"

int bpf_prog(void *arg)
{
    struct UbpfTracerCtx *ctx = arg;
    bpf_notify(ctx->traced_function_address);

    return 0;
}
```

Figure 5.10: **Example BPF program: Notify.** The BPF program `notify.bin` takes a function address as an argument and uses the `bpf_notify` helper function to prints out the function's name into the unikernel console.

The `bpf_notify` helper function also accepts the traced function address as an argument and prints out a notification into the unikernel's standard output whenever this helper is called. We use this in a simple BPF program `notify.bin` (see figure 5.10).

5.4 Limitations

The BPF runtime does not include a verifier which makes it possible to execute or attach unverified BPF code.

To optimize the runtime performance, many compilers try to avoid expensive function calls of small functions by increasing the code size and inlining the functions at their call sites. The tracer does not work with inlined functions since we can only trace function calls and in the case of an inlined function the function call is eliminated. This is a general limitation of tracing programs. To address this, we recommend to either label the function as `noinline` or insert a static tracepoint.

Lastly, self-modifying code would possibly not work properly with the tracer – depending on what parts of the binary get modified.

6

Evaluation

We tested the functionality by including the runtime and tracer in existing applications. These applications include a simple `demoapp` that calls a dummy function `function1` in a loop as well as an `sqlite_benchmark` application which executes 60,000 SQL queries on an SQLite database to represent a more complex application. All benchmarks were run with a KVM hypervisor on an AMD EPYC 7742 CPU. We pinned the execution to the first CPU core and set the CPU governor to "performance" in order to avoid CPU throttling.

6.1 Design goals

Earlier we have set three design goals, here we discuss how well these goals were met.

6.1.1 Minimal user code changes

We aimed to make it as easy as possible for users to include our project. Currently, for the functionality of the BPF runtime, there are no user code changes necessary – users can include the `ubpf` library in their project's configuration and functions for creating and executing BPF virtual machines will be available to use.

Our tracing mechanism is limited to functions that have not been inlined. This is a general limitation of tracing tools. For most functions, no changes are necessary. For smaller functions we recommend notifying the compiler not to inline the function, e.g. with the `noinline` attribute:

```
__attribute__((noinline)) void myfun() { ... }
```

Further, for the dynamic tracer we do require users to change their compilation flags to include extra `nop` instructions at the function start. Depending on what functions are intended to be traced, it is possible to add these options either to the application

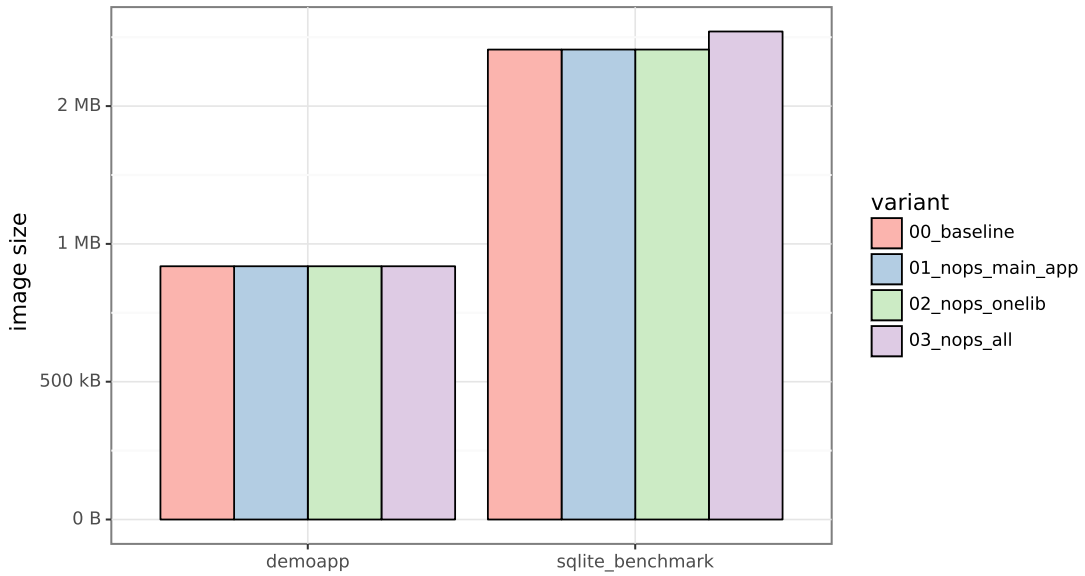


Figure 6.1: **Comparison of image size with different compilation flags.** Image sizes are mostly unaffected by the profiling options. `00_baseline`: No profiling options. `01_nops_main_app`: Profiling options enabled for the main application but not for any libraries. `02_nops_onelib`: Profiling options enabled only for one library (`libsqlite` for application `sqlite` and `libuksched` for application `demoapp`). `03_nops_all`: Profiling options enabled for the whole unikernel.

(without libraries), to any libraries that the user selects or to all functions in the whole unikernel. To ensure the code is compiled with the right options, we recommend forcing a rebuild of the unikernel after adding these options. The required compilation options can be set by adding a variation of the following lines to the project's `Makefile.uk`:

```
# enabling tracing for application `hello`
APPHELLO_CFLAGS-y += -pg -mrecord-mcount -mnop-mcount

# enabling tracing for library `sqlite`
LIBSQLITE_CFLAGS-y += -pg -mrecord-mcount -mnop-mcount

# enabling tracing for the whole unikernel including all libraries
CFLAGS-y += -pg -mrecord-mcount -mnop-mcount
```

As these options add an instruction to every function, we have inspected the influence that the extra instructions would have on image size and runtime performance (figures 6.1 and 6.2). While there is a slight increase in image size, we did not observe any significant difference in runtime performance.

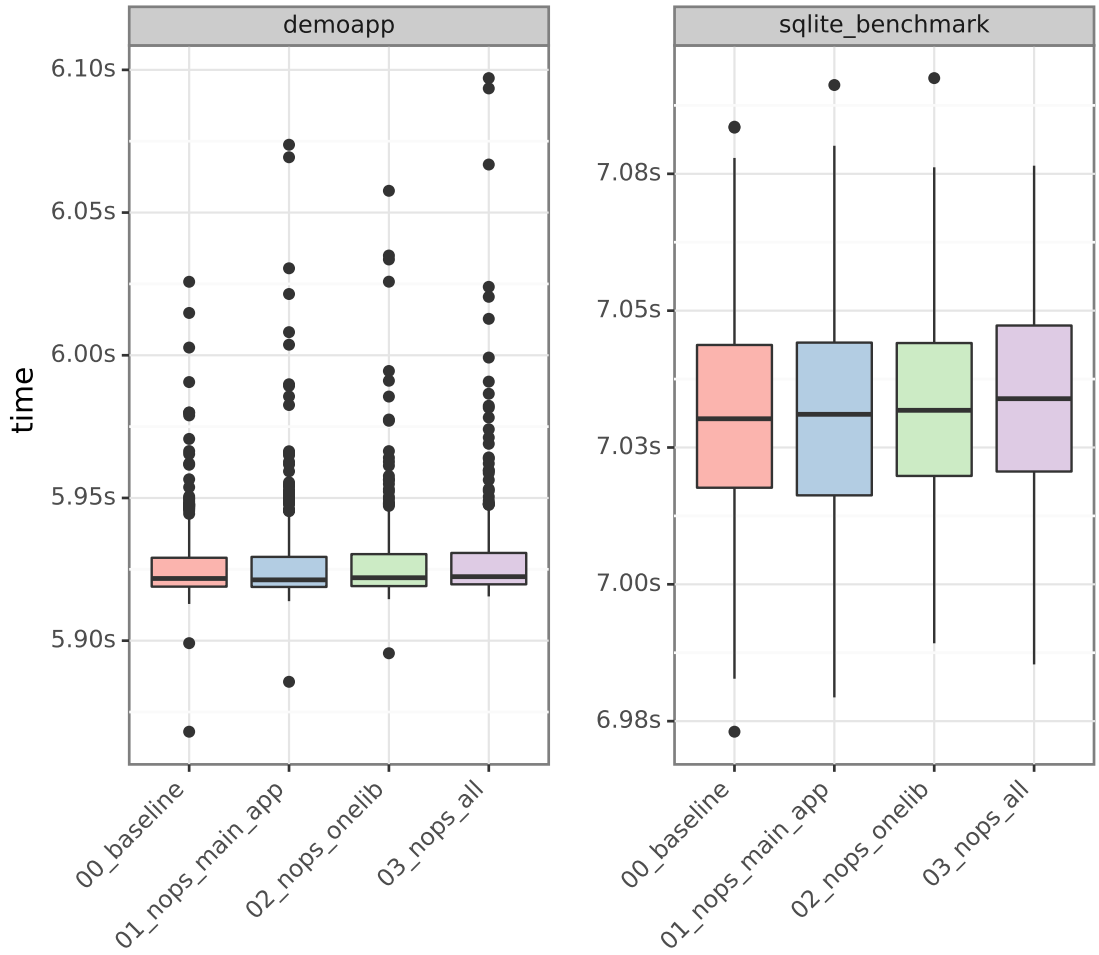


Figure 6.2: **Performance impact of profiling options.** Profiling options do not have any significant impact on performance, probably due to processor optimizations. `00_baseline` : No profiling options. `01_nops_main_app` : Profiling options enabled for the main application but not for any libraries. `02_nops_onelib` : Profiling options enabled only for one library (`libsqlite` for application `sqlite` and `libuksched` for application `demoapp`). `03_nops_all` : Profiling options enabled for the whole unikernel.

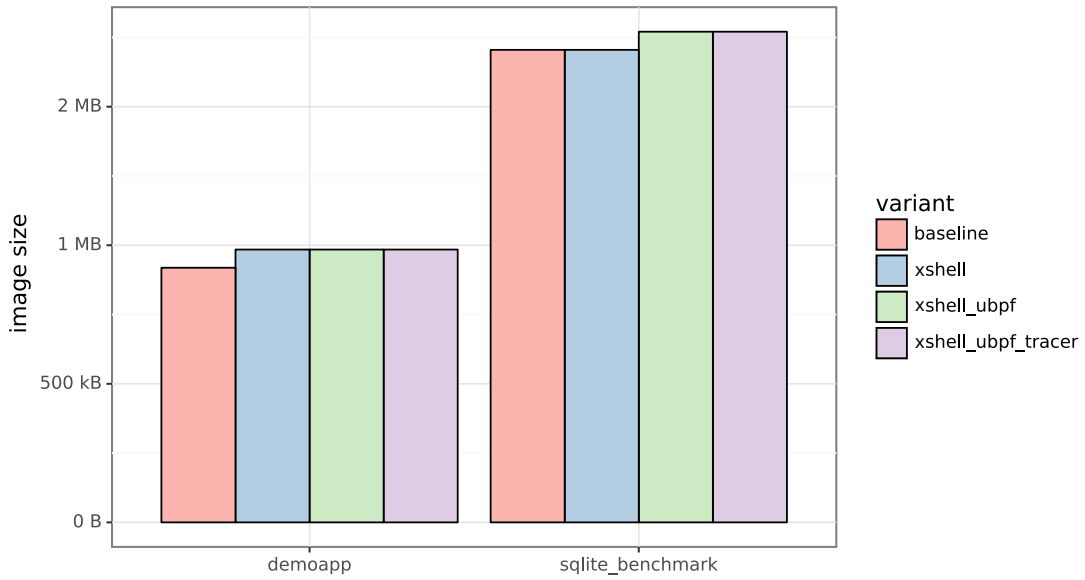


Figure 6.3: **Comparison of image size with different libraries included.** Image sizes are mostly unaffected by the inclusion of different libraries. `baseline` : No additional libraries. `xshell` : Including the `xshell` library. `xshell_ubpf` : Including Xshell and the BPF runtime (uBPF). `xshell_ubpf_tracer` : Including Xshell, BPF runtime and dynamic tracer. The image with the tracer library does not include the profiling options.

We also tested the effect of including different libraries on image size. Since the project uses Xshell as interface, we always need to include also the `xshell` library. Additionally, we tested the inclusion of both Xshell and the BPF runtime as well as combining combining Xshell, BPF runtime and the dynamic tracer in one image. Our results show that the image size does not change very much by including either of the libraries (fig. 6.3).

6.1.2 Simple command-line interface

We integrated the project with Xshell and provide several commands to execute BPF programs and attach them to arbitrary functions. In addition, we also add the possibility to list and detach attached programs. The following example shows how to use the Xshell interface to attach programs, list attached programs and execute BPF programs. BPF programs can be modified at the runtime of the unikernel. However, already attached programs are loaded into the unikernel memory and can no longer be modified. To update an attached BPF program to a new version, the program needs to be detached and attached again to force reloading.

```c
int bpf_prog(void *arg)
{
    return 0;
}
```

Figure 6.4: **Example BPF program: No operation.** The program `noop.bin` is the simplest BPF program we can create and use for benchmarking purposes.

```
> bpf_attach myfun count.bin
Program was attached successfully.
> bpf_attach myfun notify.bin
Program was attached successfully.
> bpf_list myfun
myfun:
 - count.bin
 - notify.bin
> bpf_detach myfun notify.bin
Program was detached successfully.
> bpf_exec get_count.bin myfun
The program returned: 2
```

We designed the interface in a way that the user only has to provide the most essential arguments and the tracer instance gets created automatically in the background without the user having to worry about initializing it or passing some extra information to the commands.

6.1.3 Performance

For benchmarking the runtime impact of attached BPF programs, we first created a dummy function:

```c
__attribute__((noinline)) void function1()
{
    static int count = 0;
    count++;
}
```

To this dummy function, a simple BPF program `noop.bin` was then attached multiple times (see figure 6.4 for details on the program `noop.bin`).

We then benchmarked the dummy function with different numbers of attached instances of `noop.bin` by running the function 10,000 times for each test case and measuring the elapsed nanoseconds. As shown in figures 6.5 and 6.6, the effect of

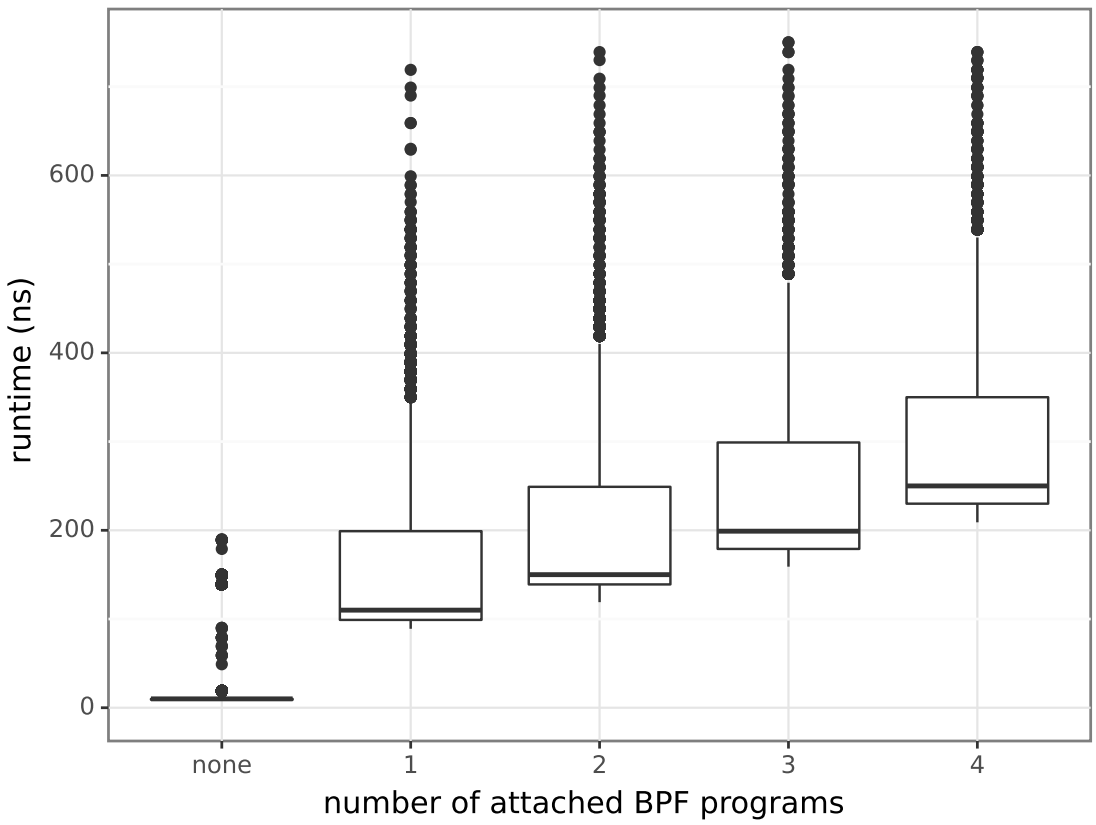


Figure 6.5: **Function runtime depending on the number of attached BPF programs.** We attach the same BPF program `noop.bin` multiple times to a dummy function. The first attached program has an exceptionally high impact on the runtime, all additionally attached instances increase the runtime at a lower rate.

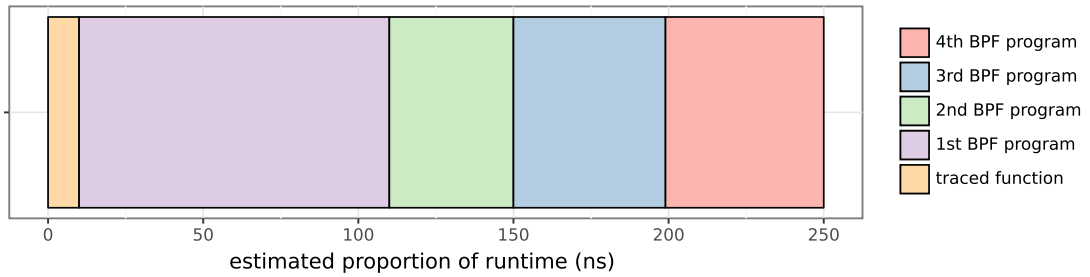


Figure 6.6: **Estimated composition of runtime with four attached BPF programs.** The figure shows an estimate of the proportional runtime of a benchmark with four attached `noop.bin` programs. Estimates of the runtime proportion were calculated by taking the median of the benchmark runtimes (see figure 6.5). The proportion of runtime of the first attached program is higher than of the following program instances.

Attached BPF program	Mean runtime (ns)	Median runtime (ns)
no attached program	24	19
`noop.bin`	185	119
`count.bin`	664	619
`get_count.bin`	7932	7798
`notify.bin`	2597458	2892427

Figure 6.7: **Runtime of different BPF programs.** The table shows the mean and median runtime of the dummy function with different BPF programs attached.

the first attached program on runtime is higher than the effect of subsequently attached programs. This can be explained by the overhead of looking up the BPF programs attached to the function in `run_bpf_program()`.

Next, we benchmarked the dummy function with different BPF programs attached. In this case, only one program was attached to the function at any time. We used the following sample programs:

- `noop.bin`, which does nothing and simply returns (fig. 6.4)
- `count.bin`, which counts the number of times a function has been executed using BPF maps (fig. 5.8)
- `get_count.bin`, which based on a function name retrieves the value saved by `count.bin` (fig. 5.9)
- `notify.bin`, which prints out the name of the traced function on standard output (fig. 5.10)

In figure 6.7 we show that manipulating the BPF maps (`count.bin`) does not impact the runtime as much as finding the address of a function based on its name (`get_count.bin`). Further, printing into console (`notify.bin`) is significantly slower than any of the other operations.

```
int bpf_prog(void *arg)
{
    int *p = 0;
    return *p;
}
```

Figure 6.8: **Example BPF program: Accessing a NULL pointer.** The program `oob.bin` dereferences a NULL pointer which leads to a failed execution but does not result into crashing the main application.

6.2 Safety and isolation

The goal of the project was to create a safe way to install new binaries at runtime. Here we evaluate how safe the implementation is and how are the binaries isolated from the rest of the application.

6.2.1 Out of bounds memory access

While the BPF bytecode is currently not verified, the included BPF runtime features a bounds checker which checks for out of bounds memory accesses. If a program tries to access a memory region outside of its arguments, the program execution fails. In this simple example (see figure 6.8), the program is dereferencing a NULL pointer which should result into a crash of the program as it is undefined behaviour. Accessing other memory regions (except for the arguments) ends up with similar results.

When we execute this program with Xshell, we get a message that the program execution failed:

```
> bpf_exec oob.bin
BPF program execution failed.
```

Additionally, the bounds checker prints an error message on standard output of the unikernel:

```
uBPF error: out of bounds memory load at PC 4, addr 0x0, size 4
mem 0x0/zd stack 0x0/520223744
```

While the BPF program crashes, the main application continues its execution without crashing. A verifier would help in this case to allow the program to access more memory regions and move a part of the workload of checking memory accesses to the time when the program gets loaded.

```c
int bpf_prog(void *arg)
{
    if (!arg) {
        return -1;
    }
    const char *str = (const char *)arg;
    unsigned i = 0;
    int sign = 1;
    int result = 0;
    if (str[0] == '-') {
        sign = -1;
        i = 1;
    }
    while (str[i]) {
        result = result * 10 + (str[i] - '0') * sign;
        i++;
    }
    return result;
}
```

Figure 6.9: **Example BPF program: Loops.** This program converts a string argument into an integer, making use of the fact that loops are allowed, since there is no integrated verifier. This program is compiled and executed without problems with the included runtime.

6.2.2 Loops

One significant limitation of BPF in Linux is that the verifier does not allow loops in the BPF programs. Since we do not use a verifier, it is possible to execute BPF programs with loops in them. We take as an example a program in figure 6.9 which converts a string argument to a number. This is useful because we designed `bpf_exec` to pass string arguments to the programs (similar to any shell implementation). However, we do not have a C library implementation for BPF programs which makes converting the string arguments to numbers more difficult. If we also do not allow loops, then this task becomes nearly impossible without an additional helper function.

Inclusion of loops is a question of safety since a program containing an infinite loop would likely block or otherwise negatively affect the main application or the usability of our shell. It is also not easy to solve with a verifier as supporting loops requires a more complex verification process.

The bounds checker within the included BPF runtime contains a simple check to recognize the most obvious infinite loops. The program in figure 6.10 will not execute successfully. It will practically immediately fail with the following error message:

```
int bpf_prog(void *arg)
{
    while (1)
        ;
    return 0;
}
```

Figure 6.10: **Example BPF program: Loops (fail).** This program includes an infinite loop with no body. The bounds checker can recognize this and the program execution will fail without blocking the main application.

```
> bpf_exec loop_fail.bin
Failed to load code: infinite loop at PC 2
```

More complex infinite loops are not recognized by the bounds checker as we can see with the program in figure 6.9 which could potentially also result in an infinite loop.

7

Conclusion

In this project we developed a method for safe loading of binary programs into a unikernel at runtime. We adapted the BPF language runtime to run in unikernels built with Unikraft. BPF binaries are being isolated from the main application by virtualization. A built-in bounds checker ensures that BPF binaries can interact with the main application only through a carefully selected set of helper functions. Thanks to running in a virtualized environment, crashing BPF binaries can not impair the functionality of the main application.

We used the included BPF runtime to implement a dynamic tracer for Unikraft unikernels. The tracer allows loading and attaching BPF binaries to arbitrary functions at the runtime of a unikernel without needing to recompile and reboot.

Benchmarking two different applications showed that the performance impact of including the BPF runtime is negligible and only minimally influences the image size.

In conclusion, the included BPF runtime provides a simple possibility for extending unikernels at runtime. We believe that this will aid future development of unikernel tooling, ultimately increasing the adaptation of unikernels in cloud environments.

8

Future Work

8.1 Verifier

BPF is designed to be formally verifiable, but the project we decided to use does not implement a verifier, only a bounds checker. It would be beneficial to either integrate the *uBPF* project with a verifier or use a standalone verifier to check the code when loading it into the unikernel. Obviously, verifying code at insertion brings an additional performance overhead when loading BPF programs. However, verification also provides safety guarantees that are necessary for the BPF runtime to be useful in real-world applications.

8.2 Other language runtimes

In this project we explored integrating BPF into unikernels. However, similar results could also be achieved by including other language runtimes such as Lua or WebAssembly. The challenge here is to provide sufficient safety. This is in part provided due to the sandboxed environment any dynamic language runtime provides but needs to be enforced by other language features or tools such as a verifier or at least a bounds checker.

8.3 Using `-mfentry` instead of `-mrecord-mcount`

We use the `-mrecord-mcount` flag of GCC to include space for a tracing function call at the beginning of each function. This flag causes the function call to be placed after the function prologue. The newer `-mfentry` flag places the function call before the function prologue as the very first instruction of the function. Since it eliminates the need to search for the NOP instruction in the whole function body, the `mfentry` option would simplify the tracer design and is worth exploring in the future.

8.4 BPF runtime usecases

While originally intended as a network packet filter, BPF gained a lot of new usecases within the Linux kernel. Even though not all of these usecases are necessarily useful in unikernels, it would be interesting to explore other ways unikernels could benefit from having an integrated BPF runtime. In this project, we explore extending the unikernel with tracing capabilities. We trace entry points of functions but the tracer could be extended by tracing the exit points of functions which would allow building function call graphs and measuring time spent in a function. Future projects could serve to optimize performance of unikernels by observing performance and latency in different areas – e.g. monitor I/O operations or network traffic.

8.5 More code attachment options

Currently, we explored how to attach BPF code to the entry point of functions. Providing more events to attach to brings more extensibility to the kernel. In the Linux kernel, BPF code can be attached to various events, such as function exit points or hardware events. This is also necessary to enable more usecases.

Abbreviations

BPF Berkeley Package Filter

eBPF extended Berkeley Package Filter

BCC BPF Compiler Colection

GDB GNU Project debugger

GCC GNU Compiler Collection

OS operating system

VM virtual machine

MPK Memory Protection Keys

List of Figures

Bibliography

[1] *Adapteva Epiphany Options*. URL: https://gcc.gnu.org/onlinedocs/gcc/Adapteva-Epiphany-Options.html (visited on 05/10/2023).

[2] The Unikraft Authors. *Docs - Unikraft*. 2023. URL: https://unikraft.org/docs/ (visited on 01/30/2023).

[3] The Unikraft Authors. *Session 03: Debugging in Unikraft*. 2023. URL: https://unikraft.org/community/hackathons/sessions/debugging (visited on 03/30/2023).

[4] Alfred Bratterud et al. "IncludeOS: A minimal, resource efficient unikernel for cloud services." In: *2015 IEEE 7th international conference on cloud computing technology and science (cloudcom)*. IEEE. 2015, pp. 250–257.

[5] *Dynamic Language Runtime Overview*. URL: https://learn.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/dynamic-language-runtime-overview (visited on 05/09/2023).

[6] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. "Exokernel: An operating system architecture for application-level resource management." In: *ACM SIGOPS Operating Systems Review* 29.5 (1995), pp. 251–266.

[7] Elazar Gershuni et al. "Simple and precise static analysis of untrusted linux kernel extensions." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 1069–1084.

[8] Brendan Gregg. *Linux Extended BPF (eBPF) Tracing Tools*. URL: https://www.brendangregg.com/ebpf.html (visited on 04/20/2023).

[9] RedHat Inc. *ftrace - Function Tracer*. 2018. URL: https://www.kernel.org/doc/html/latest/trace/ftrace.html (visited on 05/13/2023).

[10] Avi Kivity et al. "OSv—optimizing the operating system for virtual machines." In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 61–72.

[11] Simon Kuenzer et al. "Unikraft: fast, specialized unikernels the easy way." In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021, pp. 376–394.

[12] Idit Levine. *UniK: Build and Run Unikernels with Ease*. 2018. URL: https://medium.com/solo-io/unik-build-and-run-unikernels-with-ease-2c7344115cbb (visited on 05/16/2023).

[13] *Linux tracing - kprobe, uprobe and tracepoint*. URL: https://terenceli.github.io/%E6%8A%80%E6%9C%AF/2020/08/05/tracing-basic (visited on 05/12/2023).

[14] *Linux tracing systems & how they fit together*. URL: https://jvns.ca/blog/2017/07/05/linux-tracing-systems (visited on 05/12/2023).

[15] Hongyi Lu et al. "MOAT: Towards Safe BPF Kernel Extension." In: *arXiv preprint arXiv:2301.13421* (2023).

[16] Anil Madhavapeddy et al. "Unikernels: Library operating systems for the cloud." In: *ACM SIGARCH Computer Architecture News* 41.1 (2013), pp. 461–472.

[17] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture." In: *USENIX winter*. Vol. 46. 1993.

[18] Masanori Misono et al. "Xshell." unpublished. 2022.

[19] Donald E Porter et al. "Rethinking the library OS from the top down." In: *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. 2011, pp. 291–304.

[20] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. "LKL: The Linux kernel library." In: *9th RoEduNet IEEE International Conference*. IEEE. 2010, pp. 328–333.

[21] Ali Raza et al. "Unikernel Linux (UKL)." In: *Proceedings of the Eighteenth European Conference on Computer Systems*. 2023, pp. 590–605.

[22] sysml. *ukbpf*. 2019. URL: https://github.com/sysml/ukubpf (visited on 03/30/2023).

[23] Joshua Talbot et al. "A security perspective on unikernels." In: *2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*. IEEE. 2020, pp. 1–7.

[24] *uBPF*. URL: https://github.com/iovisor/ubpf (visited on 05/20/2023).

[25] Junzo Watada et al. "Emerging trends, techniques and open issues of containerization: a review." In: *IEEE Access* 7 (2019), pp. 152443–152472.

[26] *WebAssembly*. URL: https://webassembly.org/ (visited on 05/16/2023).

[27] *x86 Options*. URL: https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html (visited on 05/10/2023).