SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Network Function Virtualization with UniBPF

Paul Zhang

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Network Function Virtualization with UniBPF

Network Function Virtualization mit UniBPF

Author:	Paul Zhang
Supervisor:	Prof. Dr. Pramod Bhatotthia
Advisors:	Dr. Masanori Misono, Ilya Meignan–Masson
Submission Date:	01.10.2024

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

Munich, 01.10.2024 Paul Zhang

Acknowledgments

I want to thank Prof. Dr. Pramod Bhatothia for supervising this thesis and allowing me to explore this topic.

I would also like to extend my deepest thanks to my advisors, Dr. Masaroni Misono and Ilya Meignan–Masson, for supporting me through this thesis. Their guidance and expertise were invaluable to me and the final product.

Abstract

The recent rise of Network Function Virtualization (NFV) has revolutionized modern networks, providing unparalleled flexibility, scalability, and maintainability. However, conventional deployment using Virtual Machines (VMs) introduces performance overhead, while more lightweight containers raise security concerns. Unikernels have emerged as a lightweight, high-performance alternative, but they suffer from limited flexibility, especially for dynamic updates.

This thesis presents a novel approach to integrating eBPF into the processing pipeline of Unikernel-based Virtual Network Functions (VNFs). By incorporating eBPF-based elements into the Click Modular Router, we enable custom packet processing logic to be written in common programming languages, significantly improving the system's flexibility without compromising performance or security. Additionally, we integrate a decoupled eBPF verifier to ensure secure Just-In-Time (JIT) compilation of Berkeley Packet Filter (BPF) programs while keeping startup times low.

Through qualitative and quantitative evaluations, we demonstrate that our framework offers significant improvements in flexibility and security while maintaining competitive performance metrics such as throughput, latency, and memory usage. However, further optimizations to the networking pipeline are required for the system to be fully production-ready.

The source code for this work is available at https://github.com/paulzhng/appclick-ubpf.

Contents

1 Introduction

In search of faster product development cycles, lower dependence on proprietary hardware, and difficulties in scaling their infrastructure, network operators have started adopting Network Function Virtualization (NFV) in recent years [1]. NFV solves many of these problems by moving network functions from proprietary and specialized hardware to software, allowing network operators to deploy generic hardware while running specialized software.

Many Virtual Network Functions (VNFs) are deployed using containers with their own set of problems. Containers, in contrast to more traditional VMs, are purely isolated through the operating system with each other [2]. Historically, vulnerabilities have led to container breakouts, allowing attackers root-level access to the host [3]. Ensuring critical infrastructure security, such as those of network operators, is vital. Previous work has shown that VNFs running on Unikernels show promising results, providing a lightweight, performant, and more secure alternative to containers.

Despite their benefits, Unikernels have notable drawbacks, including their lack of flexibility. The whole Unikernel must be rebuilt and deployed if the software needs to be updated. Due to their monolithic nature, partial updates without restarting the whole VM is impossible. This introduces operational hurdles and increases the costs of updates, especially if the VNF relies on internal state, such as in the case of CGNATs.

To solve these problems, this thesis proposes integrating eBPF into the network processing pipeline of a VNF running on a Unikernel. We fully incorporate an eBPF VM into the Click Modular Router, allowing us to use BPF programs for packet processing logic. We also incorporate an eBPF verifier to ensure better security of VNFs.

We face several design challenges, such as the tradeoff between eBPF verification and the involved time and complexity, integrating eBPF into Click, and retaining state across live reconfiguration boundaries. Our key idea is to add BPF-based elements to Click, which are capable of executing user-supplied BPF programs. To ensure secure execution while preserving fast startup times, we decoupled the verification process, allowing ahead-of-time verification.

We present a qualitative analysis of our framework's flexibility and ease of development, along with benchmarking key metrics, including its performance and resource utilization. Our numbers show no to low overhead caused by the eBPF VM in memory usage, latency, and throughput while providing many tangible benefits, such as

increased flexibility. By using live reconfiguration instead of restarting the full VM, the reconfiguration time is reduced by up to two magnitudes.

2 Background

2.1 eBPF

eBPF allows safely executing programs in privileged environments. Historically, it originates from the Berkeley Packet Filter (BPF), although nowadays, both terms are used interchangeably. In the context of the Linux Kernel, eBPF provides custom programs to run in response to system calls, network events, and more [4].

It consists of a RISC instruction set targetable by compilers such as *clang* or *rustc* [5]. These compiler backends allow for the compilation of source code, written in programming languages like *C* or *Rust*, to eBPF bytecode.

In the Linux kernel, the bytecode is executed with the option for JIT compilation. Before loading the eBPF program, the bytecode undergoes verification to ensure its safety by checking for properties such as termination and no memory access violations.

2.1.1 Helper Functions

eBPF programs can interact with the Kernel by calling a predefined set of *helper functions*. These helper functions provide different functionalities such as, among others, retrieving randomness, retrieving the time, or interacting with the networking stack. Notably, a set of functions is dedicated to interacting with *BPF maps* [6].

2.1.2 Maps

eBPF maps provide eBPF programs functionality to store and retrieve data, thus adding statefulness. Different specialized eBPF map types exist, such as `BPF_MAP_TYPE_HASH`, representing a key-value store, or `BPF_MAP_TYPE_ARRAY`, representing a list of elements [7].

To define a BPF map, special metadata needs to be written to a specific ELF section. These maps are materialized when the program is loaded into the Kernel. To access BPF maps from a BPF program, helper functions, such as `bpf_map_lookup_elem` for accessing data, `bpf_map_update_elem` for updating data, and `bpf_map_delete_elem` for deleting data, can be used [6].

2.2 Unikernels

Unikernels are specialized operating systems that do not separate user and kernel space. Instead, the application is packaged into a single-address-space operating system [8].

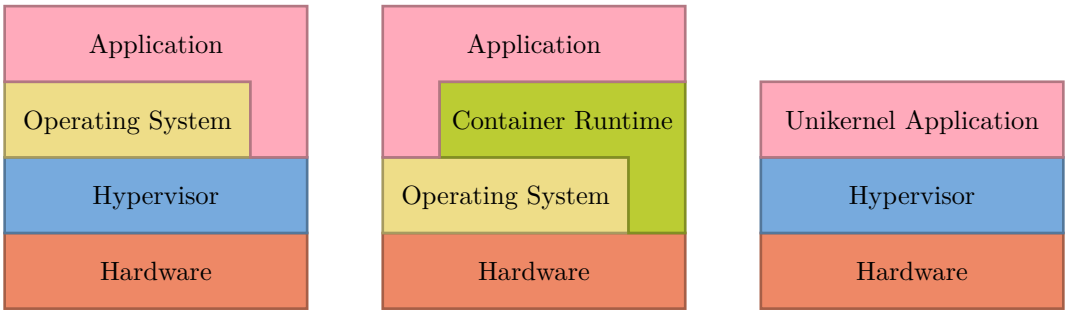

Figure 1: High-level comparison of software components of different virtualization types [8].

Unikernel images are specialized and lightweight by statically linking the application with the operating system. For example, unnecessary components can be omitted if they are not required by the application: If an application does not need a file system, the operating system does not have to provide one.

Generally speaking, Unikernels allow for better utilization of system resources by, for example, removing the need for expensive context switches between user and kernel space. Since they only contain necessary components, Unikernels are considered more secure by reducing the possible attack surface. Compared to containers, Unikernels are true virtual machines that provide better isolation from each other.

2.3 Network Function Virtualization (NFV)

Network Function Virtualization (NFV) is a concept in networking that virtualizes network middleboxes like routers, firewalls, and load balancers [9].

By utilizing NFV, network operators can abstract network functions into software installed and managed on ordinary servers instead of relying on specialized hardware appliances. This allows for more flexibility, scalability, and cost-efficiency in deploying and managing network services.

2.3.1 Click

Click is a modular framework for flexibly building network functions by allowing the composition of different *elements* [10]. Individual elements have specialized and bounded functionality, such as `IPFilter`, which filters packets by their content, or `Discard`, which drops all packets.

Click can be configured using a custom configuration language called the *Click configuration language*, as used in Listing 1. Instances of individual element classes are instantiated with arguments and composed into directed graphs describing the packet flow between the individual element instances [10].

```
FromDevice(0) // Accept packets from a network device
 -> c0 :: Classifier(12/0806 20/0001, // ARP
                     12/0800,         // IP
                     -);              // Everything else

// Answer ARP requests
c0[0] -> ARPResponder(173.44.0.2 $MAC1)
      -> ToDevice(0);

// Print IP packets
c0[1] -> StripEtherVLANHeader
 -> CheckIPHeader
 -> Print('Received packet')
 -> Discard;

c0[2] -> Discard;
```

Listing 1: A Click configuration which handles ARP requests and prints all incoming IP packets

3 Overview

3.1 Design Goals

The following design goals have been identified as key objectives for our work.

Flexibility We must be able to cover many different use cases in networking. We aim to create a framework that allows users maximum flexibility in *what* they can make.

Live reconfigurability It is crucial to allow changes to packet processing logic to be deployed without downtime. Additionally, network middleboxes such as *Load Balancers* or *NATs* depend on internal state to fulfill their purpose. Thus, we must allow *live reconfiguration* while providing the ability to preserve state.

Usability Our framework must be *easy* to use. Developers without intricate knowledge about the framework should be able to quickly understand and use our system. Additionally, developers should be able to leverage their existing knowledge.

Performance To use our solution in production workloads, we aim to maximize the throughput and minimize the latency of packets. An explicit goal is not to incur overhead compared to native Click elements.

Furthermore, we aim to minimize the startup time of each VM. The impact of eBPF verification on this goal is to be minimized.

Compatibility Our interface design should, wherever possible, conform to the decisions made by the broader ecosystem. Concepts should be easily transferable, allowing existing infrastructure to be reused while leveraging the extensive eBPF ecosystem.

Safety Ensuring the safety and reliability of networking operations is paramount. Unsafe operations such as out-of-bound access must be *prevented* to ensure the resilience of the entire system.

3.2 System Overview

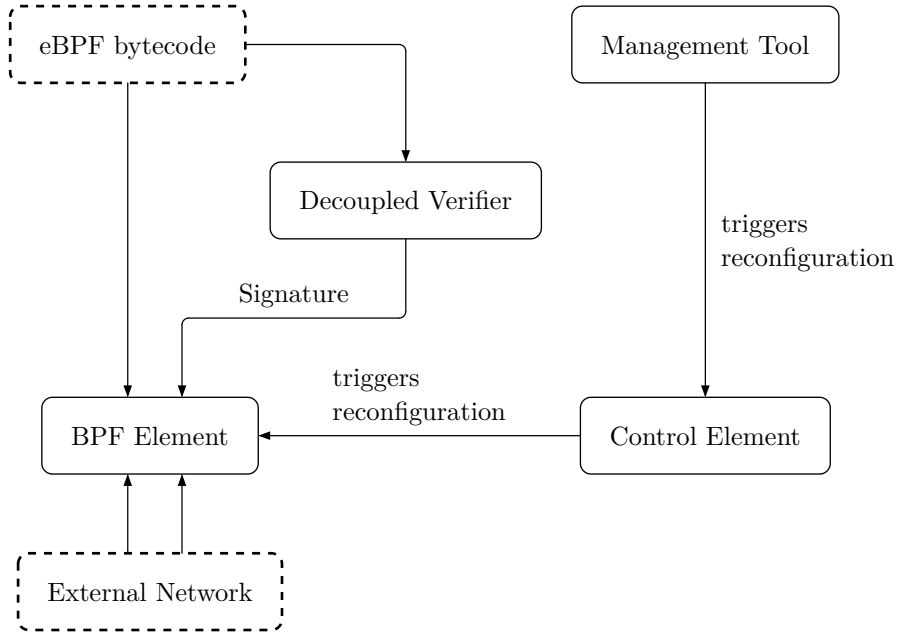

Figure 2: High-level system overview showing interactions between components.

The high-level view of the system, as shown in Figure 2, consists of the following parts:

- **Decoupled verifier:** External eBPF verifier generating a signature proving successful verification of a program
- **Management tool**: Tool capable of sending control packets to the control element
- **BPF element:** Click element that bootstraps and manages the eBPF program
- **Control element:** Element that handles incoming reconfiguration requests

3.3 Design Challenges

We have identified the following design challenges and developed key ideas to address each one.

Lengthy & complex eBPF verification Short startup times are crucial to support fast scaling for sudden load spikes. Verifying eBPF programs takes time and resources but is essential to ensure safety. Additionally, verifiers are complex and can potentially have many dependencies. Integrating a verifier into a Unikernel is difficult and makes it more heavyweight. We solve this by decoupling verification, as detailed in Chapter 4.4 (Decoupled Verification).

Integrating eBPF with Click We want to seamlessly integrate eBPF programs into Click. Thus, compatibility with the Click ecosystem is crucial. We address this by

eBPF processing using Click-native elements, as described in Chapter 4.1 (eBPF-based Click Elements).

Retaining state across live reconfiguration boundaries Our system should be capable of retaining state, even if the data structure changes. Our work addresses this by keeping the data of previously allocated BPF maps and showcasing a pattern for migrations, as outlined in Section 4.3.1 (State Retention).

4 Design

This chapter explores our approach to integrating eBPF into Click running inside a Unikernel to achieve our design goals set in Chapter 3.1 (Design Goals).

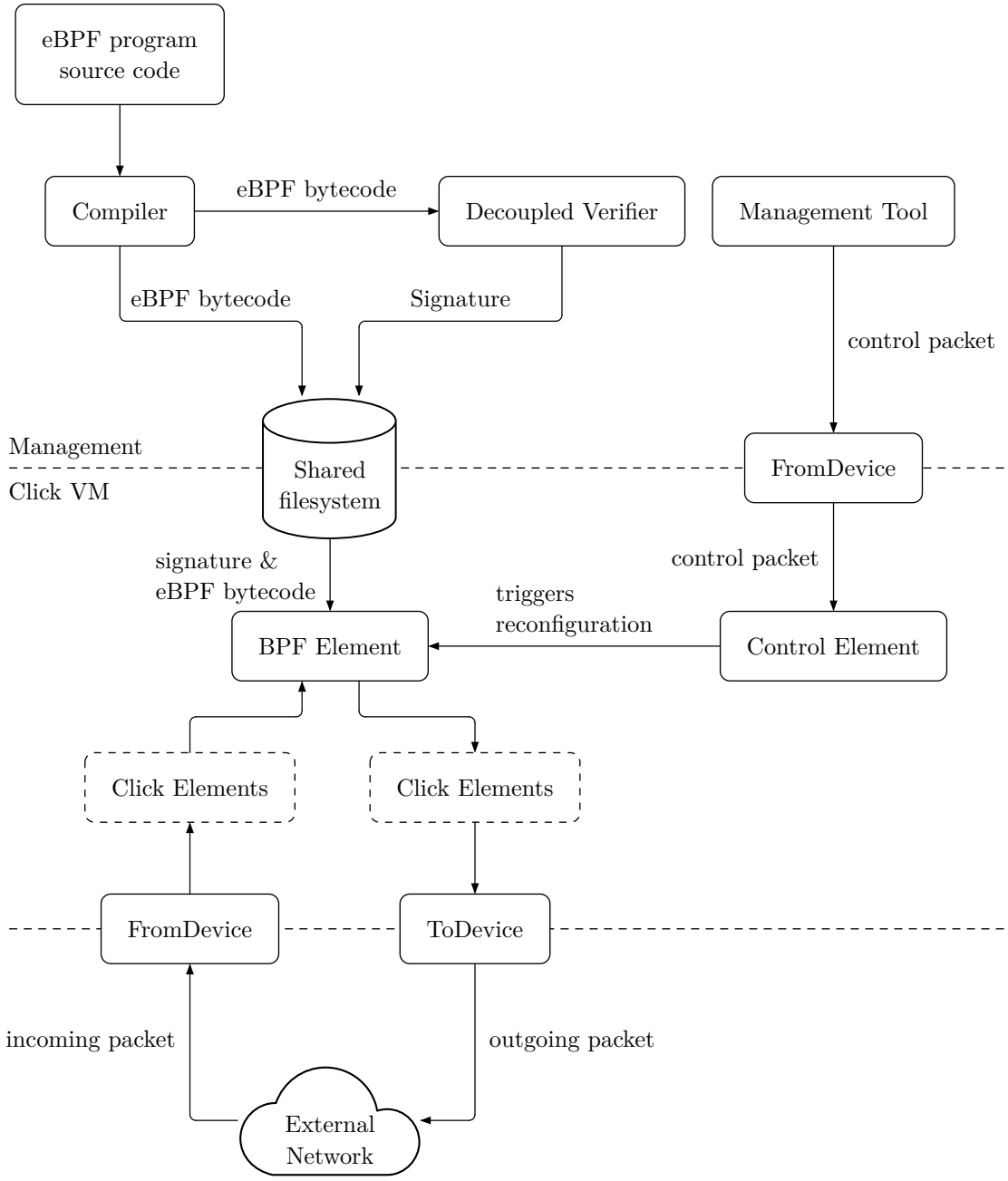

Figure 3: System components and their interactions.

A detailed overview of the overall system components can be seen in Figure 3. The following components compose the overall system:

- **eBPF program source code:** The eBPF program's source code, written in C, Rust, or any other language compilable to eBPF bytecode
- **Compiler:** Any compiler toolchain capable of converting the source code to eBPF bytecode
- **Decoupled verifier:** External eBPF verifier generating a signature proving successful verification of a program, based on *PREVAIL* [11]
- **Shared filesystem:** Filesystem shared between the host and the Click VM storing the Click configuration, the BPF programs, and their signatures
- **Click VM:** Click modular router running inside a *Unikraft* VM [12]
- **BPF element:** Click element that bootstraps and manages the eBPF program. When receiving a packet, it executes the eBPF program
- **FromDevice & ToDevice**: Click elements that receive packets from, or send packets to, network devices
- **Control element:** element that handles incoming reconfiguration requests

4.1 eBPF-based Click Elements

The BPF extension to Click is implemented using custom Click elements that integrate seamlessly with the broader Click framework. This design enables using BPF-based elements alongside the default Click elements. For instance, it is possible to configure the Click `Classifier` to specifically route IP packets to a BPF-based element that only expects IP packets as input.

These BPF elements can be initialized with a given eBPF program, which defines the logic applied to each packet the element processes. By allowing the implementation of such logic in eBPF, users gain unparalleled flexibility in customizing packet processing tasks. For more advanced needs, expressing this logic directly in code offers greater precision and simplicity than traditional rule-based approaches.

Furthermore, traditional rule-based packet filters often linearly process packets, leading to processing costs that scale with the number of rules. In contrast, eBPF enables more efficient processing through *branching*, potentially reducing the number of executed rules and thus improving overall performance.

We introduce three distinct eBPF-based elements to support various use cases:

BPFFilter Used for *filtering* packets, i.e., dropping packets from the flow

BPFClassifier Used for *classifying* packets, i.e., adjusting the flow of packets

BPFRewriter Used for *rewriting* packets, i.e., changing their contents

With these elements, it is possible to implement BPF programs that can replace their native Click counterparts. Table 1 outlines the BPF elements that can be used to host these BPF programs.

Use case	**Key Click Elements**	**BPF element counterpart**
Load balancer	RatedSplitter, HashSwitch	BPFClassifier
Firewall	IPFilter	BPFFilter
Network Address Translation (NAT)	[IP|UDP|TCP]Rewriter	BPFRewriter
Deep packet inspection (DPI)	Classifier, IPClassifier	BPFClassifier
Tunnel	IPEncap, IPSecESPEncap	BPFRewriter
Multicast	IPMulticastEtherEncap, IGMP	BPFRewriter
Broadband remote access server (BRAS)	PPPControlProtocol, GREENcap	BPFRewriter
Monitoring	IPRateMonitor, TCPCollector	/
DDoS prevention	IPFilter	BPFFilter
Intrusion detection system (IDS)	Classifier, IPClassifier	BPFClassifier
Intrusion prevention system (IPS)	IPClassifier, IPFilter	BPFClassifier, BPFFilter
Congestion control	RED, SetECN	BPFFilter, BPFRewriter
IPv6/IPv4 proxy	ProtocolTranslator46	BPFRewriter

Table 1: Use cases, key click elements for implementing them, and their BPF-based counterparts supporting those use cases, adapted from ClickOS [13, Table 1].

4.1.1 Filtering

The `BPFFilter` element *filters* packets based on the eBPF program output, analogous to the `IPFilter` element.

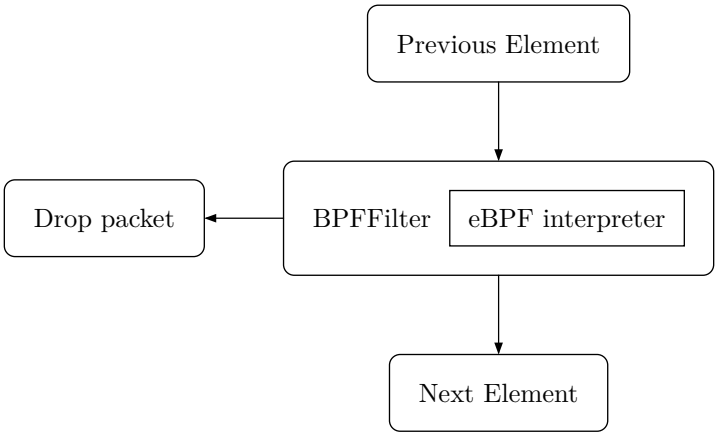

Figure 4: Input / Output of a BPFFilter.

It has one input and output and either *forwards* or *drops* the packets according to the eBPF program's criteria.

This element could, for example, be used to implement *IP-based rate limiting*, controlling traffic based on the source IP address. It could also enforce a broader set of rules for a *firewall*.

4.1.2 Classification

The `BPFClassifier` element *classifies* packets based on the eBPF program output, akin to the `Classifier` element.

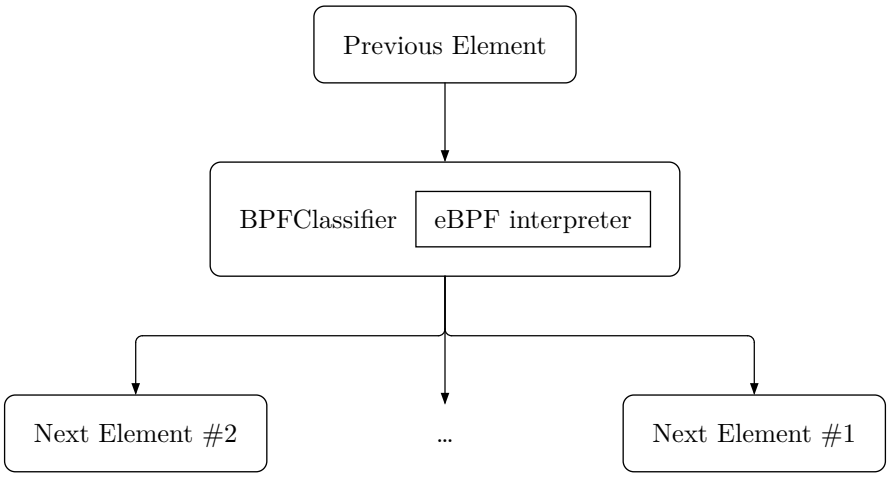

Figure 5: Input / Output of a BPFClassifier.

It has one input and multiple outputs, depending on the BPF program. The eBPF program's output specifies which output the packet is pushed to.

For example, BPF programs using this element can implement a Round Robin Load Balancer, distributing packets evenly across multiple outputs. Classifiers can also implement an IDS that redirects suspicious traffic to a monitoring service.

4.1.3 Rewriting

The `BPFRewriter` element, with a single input and output, allows eBPF programs to modify packets, enabling dynamic changes like header rewriting or payload modification.

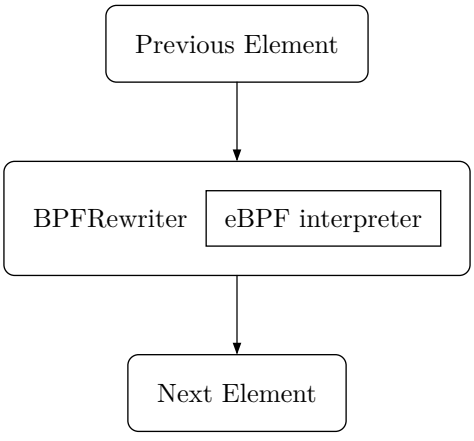

Figure 6: Input / Output of a BPFRewriter.

This functionality is useful for tasks such as *IP address translation*, *payload modification*, and other packet transformations required for advanced networking scenarios.

4.2 State Management

Many different networking appliances rely on *state* for their functionality. For example, firewalls may maintain connection states, load balancers track information to efficiently distribute requests, and NATs keep state to map and translate IP addresses.

Adhering closely to Linux's eBPF, we support BPF maps as explored in Section 2.1.2 (Maps). However, it is essential to note that the state stored in the BPF maps is not shared between different elements, ensuring isolation across elements.

As outlined in Section 2.1.1 (Helper Functions), the BPF program interfaces with BPF maps using Helper Functions such as `bpf_map_update_elem` or `bpf_map_lookup_elem`. The definitions of those Helper Functions are deliberately equal to the ones found in Linux.

4.3 Live Reconfiguration

In networking appliances, maximizing uptime is crucial, yet configuration changes frequently lead to downtime.

Often, configuration changes are only applied after a complete restart of the changed component. In contrast, our work fully supports dynamic reloading of each eBPF-powered element, which allows changing the underlying eBPF programs with nearly zero downtime or impact on the overall system.

This reconfiguration is triggered by an external network call with the information necessary to reload a single eBPF element, after which packet processing is suspended until the reload is complete.

4.3.1 State Retention

Apart from this, reconfiguration often results in *loss of state*, which is unacceptable for many networking appliances, such as CGNATs or stateful firewalls.

For example, *Click* supports live reconfiguration for elements such as `Classifier` or `ARPResponder`. Contrary to this, heavily state-dependent elements such as `AddressTranslator` cannot be reconfigured on the fly. Instead, they require a restart to apply changes, which incurs state loss and can disrupt normal operations.

Taking a CGNAT as an example, losing state will terminate all active connections, causing user interruptions and necessitating the re-establishment of sessions.

As the users of our system interact via code with the BPF maps and thus fully control how the data is accessed and modified, our system can indiscriminately allow live reloading for all eBPF elements and offload the task of ensuring the coherent use of data to the developer of the BPF programs.

Simplified, if no changes affect how the data is stored or used, the existing data in the BPF maps can be reused. In contrast, if the structure of the stored data is changed, the old data cannot be reused without any adjustments. Thus, *migrating* the data is necessary, which can be fully implemented in the BPF program.

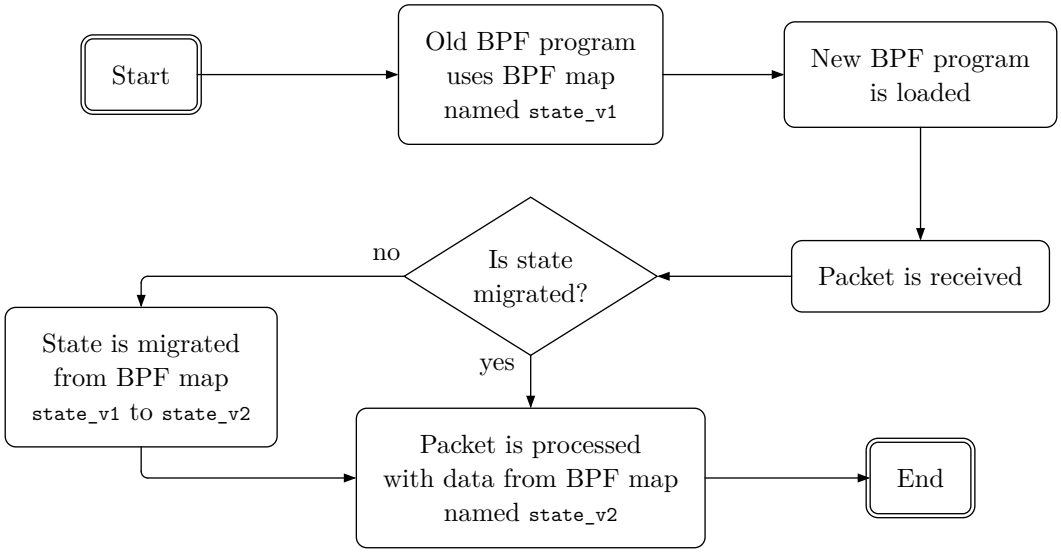

Figure 7: Possible state migration flow.

Using our system, a network operator can design a state migration similar to the one in Figure 7. If the state has not been migrated yet, the data from the old BPF map is migrated to the new BPF map. After the migration is finished, packets are processed using the state from the newly migrated BPF map.

4.4 Decoupled Verification

Linux's eBPF implementation includes a verifier that ensures the safety of eBPF bytecode by checking for issues like memory safety, proper termination, and control flow integrity before the program is loaded into the Kernel.

We include verification into our system to ensure our design goal of *safety* and thus improve overall system resilience. Typically, this would induce a tradeoff between *safety* and *startup time*, as verification happens before loading the program. Related work shows that between 91% and 99% of BPF program load time was spent on verification [14].

As short startup times are crucial for our system, our work takes a different route similar to the approach by M. Craun, A. Oswald, and D. Williams [14], in which the eBPF verifier is *decoupled* from the execution.

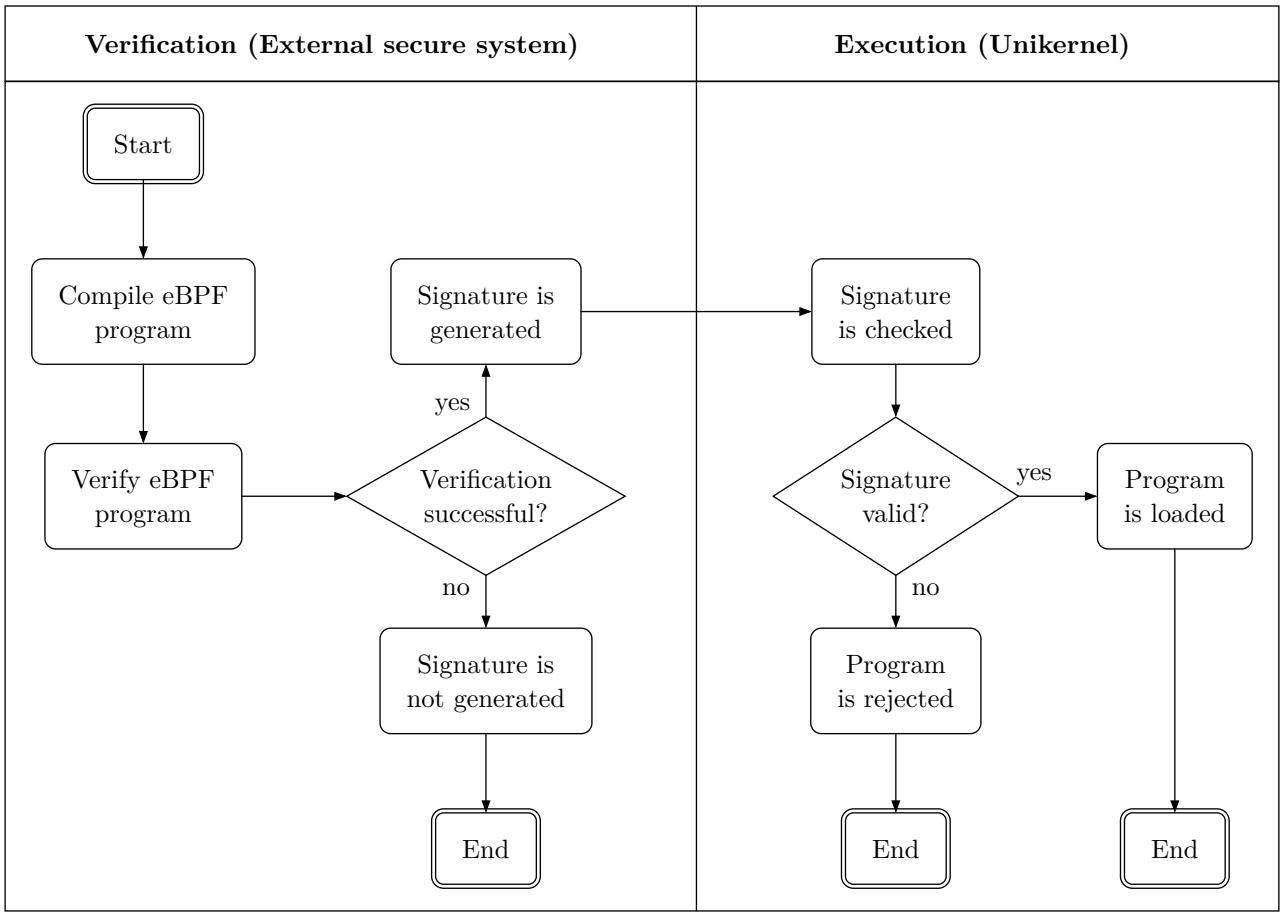

Figure 8: Flowchart showing the full verification process.

Our work features an external program running in a trusted and safe environment that first verifies and then, upon successful verification, generates a cryptographic signature of the bytecode's hash proving the verification.

This signature is then checked by the Unikernel, which includes a public key corresponding to the private key contained in the verification environment, before loading the eBPF program.

In contrast to M. Craun, A. Oswald, and D. Williams [14], we do *not* decouple the JIT compiler as we do not use the Linux eBPF subsystem, thus are not bound to the tight coupling between the verifier and JIT compiler. Other benefits, as explored in the paper mentioned above, are small and negligible, while portability across multiple architectures is sacrificed.

5 Implementation

This chapter presents the implementation of our eBPF extension to Click running on the Unikraft Unikernel.

We outline the key modifications to the existing system to support the eBPF-powered elements and live reconfiguration of those elements. Finally, we explore how we implemented and integrated decoupled verification into our system.

5.1 Click on Unikraft

Our implementation is based on Unikraft's `app-click` [15], which integrates the *Click modular router* [10] into the *Unikraft* Unikernel [16]. Specifically, we are using Unikraft v0.17.0 (Calypso) as the basis. Most of our changes are inside a forked version of `lib-click` [17], the library that contains the code for Click.

While based on `app-click`, we make changes to several parts of the system, including, but not limited to, adding support to read the configuration from a file inside a mounted folder, compared to only being able to read a mounted file.

5.2 eBPF Runtime

We use *ubpf* [18] to execute the compiled eBPF bytecode. To integrate *ubpf* into Unikraft's build system, we build upon prior work. V. Hendrychová [19] integrates ubpf as a native Unikraft library by creating a `Kraftfile` describing how ubpf should be built.

We choose ubpf due to its efficient and lightweight nature and ease of integration into existing systems. Additionally, ubpf is widely used in various contexts, such as eBPF for Windows [20], demonstrating its flexibility. In several instances, though, we expand upon ubpf to add missing functionality, such as adding BPF maps or supporting the relocation of read-only ELF segments.

5.2.1 Program Context

To pass arguments to the BPF program, we provide a pointer to a *context* with information regarding the packet to the program's entry function.

We use a context instead of directly passing the arguments to satisfy the constraints of our verifier library PREVAIL and adhere to the broader ecosystem. This choice is also beneficial because the eBPF instruction set has a limit of five function arguments,

so using a context ensures future extensibility, even if more than five arguments need to be passed [21].

```rust
#[repr(C)]
pub struct BpfContext {
    pub data: *mut u8,
    pub data_end: *mut u8,
}
```

Listing 2: Definition of the BPF Context in Rust.

As seen in the definition in Listing 2, the context contains pointers to the start and end of the packet. This definition is based on Linux's XDP entry point, excluding non-essential properties such as `ingress_ifindex` and `rx_queue_index`.

Unlike Linux, we use pointers to `data` and `data_end`, while Linux uses a `uint_32` offset relative to a base address. Internally, their eBPF implementation rewrites accesses to these `uint_32` fields to pointers when loading the program [22], while *ubpf* does not support this yet. As the *eBPF for Windows* project uses the same libraries as we, they also decided to use pointers instead of offsets for now [23]. The downside to this choice is negligible as PREVAIL guarantees safe in-bound accesses. Still, this is not optimal, as cross-platform compatibility is sacrificed.

5.2.2 ELF Relocation

By default, *ubpf* does not support *ELF segments* if the segment flags (`sh_flags`) are equal to `SHF_ALLOC` (e.g., `.rodata`) or `SHF_ALLOC | SHF_STRINGS` (e.g., `.rodata.str1.8`), meaning all static data must be located on the stack or in immediate values.

```rust
let string = "Hello World";
bpf_trace_printk(string);
```

Listing 3: Example of a program which fails to load

This is detrimental to the development experience as time has to be spent on debugging and rewriting otherwise fine code. Printing more detailed debug messages becomes nearly impossible as string data is often put into special segments, as exemplified by Listing 3.

ubpf specifically disallows read-only segments by checking whether the flags equal `SHF_ALLOC | SHF_WRITE` (e.g., `.data`). We augment this check by adding exceptions to the aforementioned flags so they are also allowed.

```
typedef uint64_t (*ubpf_data_relocation)(
  void* user_context,
  const uint8_t* data,
  uint64_t data_size,
  const char* symbol_name,
  uint64_t symbol_offset,
  uint64_t symbol_size,
  uint64_t imm
);
```

Listing 4: The modified data relocation function signature

After this check, *ubpf* offloads the relocation to the library user by calling a user-provided function called `data_relocation_function`. As seen in Listing 4, we also had to modify the signature of this function to add the *immediate* of the eBPF instruction, which triggered the relocation.
This had to be done to correctly relocate load and store instructions that access memory like `BPF_LDX` and `BPF_STX`. In our context, the immediate is an offset from the segment's start to the accessed memory location.

Our implementation of `data_relocation_function` then handles relocation for BPF maps, which will be detailed in Section 5.2.4 (BPF Maps), or for data segments like `.rodata`.
For data segments, we copy the whole segment into a new buffer and return the pointer to the correct position inside the buffer, as determined by the immediate value. The same buffer is reused if multiple instructions access the same segment, optimizing memory usage.

5.2.3 BPF Helpers

BPF helpers provide BPF programs facilities to communicate and interact with the rest of the system. Our BPF helper functions can be split into the following broad categories:

- **Helpers for BPF Maps:** `bpf_map_update_elem`, `bpf_map_delete_elem` and `bpf_map_lookup_elem`, as described in Section 5.2.4.2 (Access and Update Phase)
- `BPFRewriter` **Helpers**: `bpf_packet_add_space` to modify the size of the packet, which will be described in Section 5.3.4.1 (Resizing packets)
- **Utility Helpers:** `bpf_trace_printk` to print information to `stdout`, `bpf_ktime_get_ns` to get the current time, and `bpf_get_prandom_u32` to generate a random number

Like the rest of our eBPF implementation, we adhere to the helper functions the Linux Kernel provides: We reuse the function signature and helper function ID from the Kernel for all helper functions except for `bpf_packet_add_space`, which is unique to our `BPFRewriter` element.

```
bpf_register(vm, 1, "bpf_map_lookup_elem", as_external_function_t((void *) bpf_map_lookup_elem));
```

Listing 5: Helper Function Registration of `bpf_map_lookup`

We register all our implemented helper functions to the VM when initializing the eBPF VM with the help of *ubpf*'s `ubpf_register`, as seen in Listing 5.

5.2.4 BPF Maps

The use of BPF maps can be differentiated into the *creation and initialization* phase and the *access and update phase*.

To simplify our implementation, we only implement the BPF map types `BPF_MAP_TYPE_ARRAY` for a contiguous list of elements and `BPF_MAP_TYPE_HASH` for key-value associations. We chose these map types as they are versatile and address the most use cases.

5.2.4.1 Creation and Initialization Phase

In the *creation and initialization* phase, BPF maps are created and initialized. Unlike in Linux, programmatically creating maps using *syscalls* is unsupported.

```
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);          // Type of map: Array
    __uint(max_entries, 256);                  // Maximum number of elements in the array
    __type(key, __u32);                        // Type of keys: 32-bit unsigned integer
    __type(value, __u64);                      // Type of values: 64-bit unsigned integer
} my_array_map SEC(".maps");
```

Listing 6: Definition of a BPF map

From the user side, a BPF map is defined by placing the map definition with the desired parameters into a special `.maps` section, as shown in Listing 6.

As mentioned in Section 5.2.2 (ELF Relocation), *ubpf* calls our provided `data_relocation_function` to correctly handle the relocation of BPF maps. Upon being called, our `data_relocation_function`, depending on the BPF map type, initializes the correct data structure and returns a pointer to a wrapper around that data structure.

The wrapper contains relevant information for accessing a map, such as its type or key size.

In the case of element live reconfiguration, the previously instantiated BPF map, including its data, is reused. This requires that the same initialization parameters, such as the map type, cannot differ. If this invariant is not met, an error will be thrown when reconfiguring the element.

5.2.4.2 Access and Update Phase

In the *access and update* phase, i.e., during the normal execution of a BPF program, these programs access and modify the previously created BPF maps.

```
int *count = bpf_map_lookup_elem(&my_map, &key);
if (count) {
  // Increment the existing entry
  *count += 1;
} else {
  // Create the initial entry
  int initial = 1;
  bpf_map_update_elem(&my_map, &key, &initial, 0);
}
```

Listing 7: BPF program incrementing a number

Accessing and updating BPF maps is achieved using BPF Helper Functions, specifically `bpf_map_update_elem`, `bpf_map_delete_elem`, and `bpf_map_lookup_elem`.

Our implementation of these helper functions first dereferences the passed pointer to the map to get the metadata to the map, after which, depending on the map type, it executes the correct logic.

Internally, maps of type `BPF_MAP_TYPE_ARRAY` are backed by a correctly sized, contiguous memory chunk, while maps of type `BPF_MAP_TYPE_HASH` are implemented using the *C++* standard libraries' `std::unordered_map`.

5.2.5 JIT Compilation

To minimize the overhead caused by the interpretation of BPF bytecode, it is desirable to JIT-compile the bytecode into native machine code.

We can utilize *ubpf*'s built-in support for JIT compilation on *x86-64* and *ARM64* architectures to enable optional JIT compilation for our BPF programs. This can be enabled on a per-element basis when configuring the Click router.

As *ubpf* internally uses `mmap` to allocate an executable page, we must enable Unikraft's `LIBUKMMAP` feature. Alternatively, Unikraft provides the more modern and feature-rich `LIBPOSIX_MMAP` feature, which does not compile with enabled file system support as of the time of writing.

5.2.5.1 Exporting JIT-compiled code

We add the option to export the raw JIT-compiled instructions to a file. This helps to diagnose and fix potential performance issues, as well as get a better understanding of the BPF program itself.

The exported file contains raw machine code, which can be read with various utilities such as `objdump`. In the case of `objdump`, an example command is `objdump -D -b binary -mi386 -Maddr16,data16 rootfs/jit_dump.bin -Mintel`.

5.3 Click Elements

The core of our Click extension lies in our eBPF-backed elements.

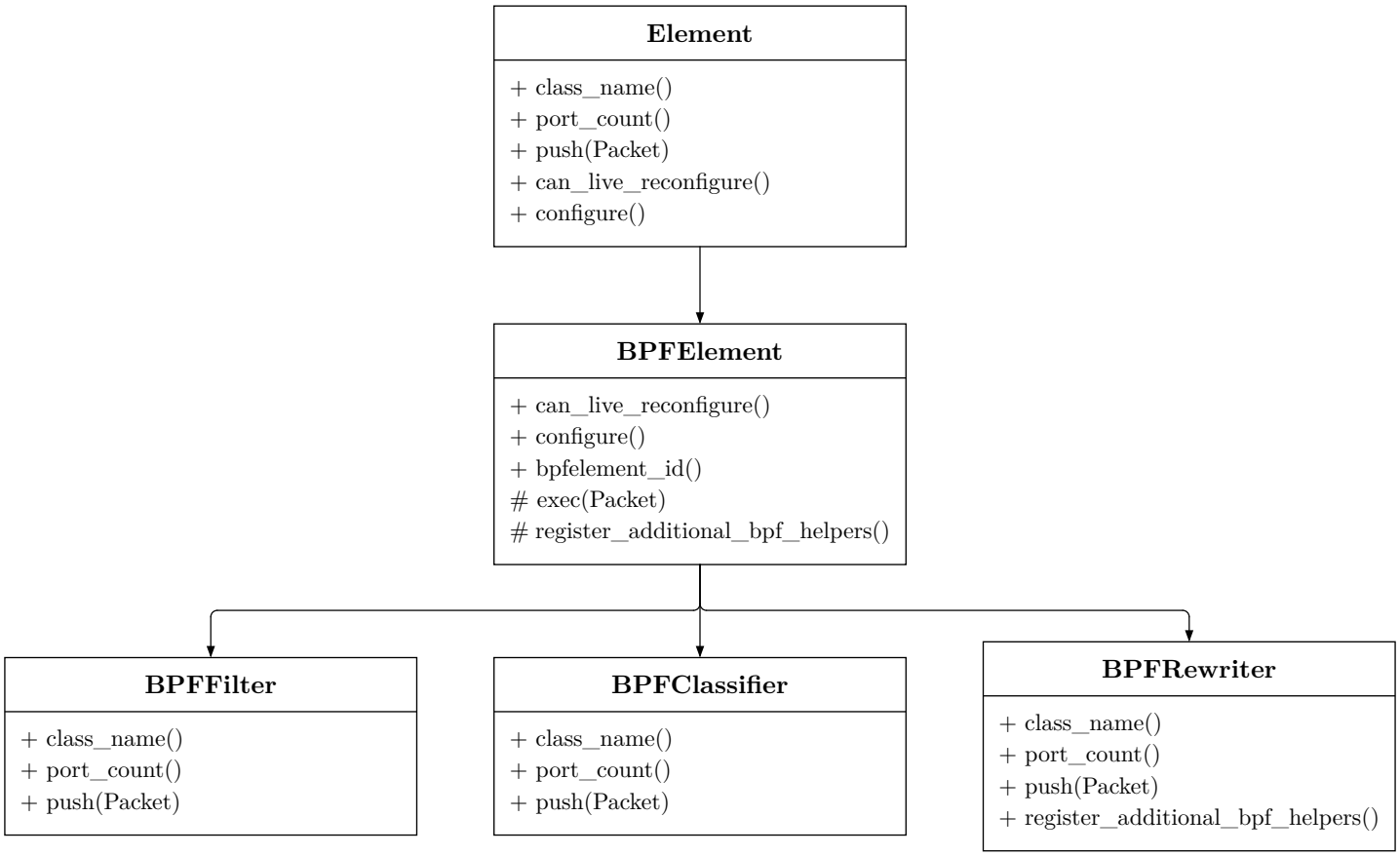

Figure 9: Simplified UML diagram of the BPF element's class hierarchy.

As many of our elements share functionality regarding configuration, eBPF execution, and more, we use an abstract parent class that contains all shared functionality, as seen in Figure 9. This helps us to deduplicate code and thus adhere more closely to the Don't repeat yourself (DRY) principle.

The parent `BPFElement` class contains the following *important* methods:

- `exec(Packet)`: Executes the configured eBPF program with arguments bootstrapped from the packet passed to the method. The method returns the return code from the BPF code.
- `configure`: Parses the configuration string with which the element was configured and initializes the element. Initialization includes setting up the eBPF virtual machine, potentially handling JIT compilation, and verifying the signature of the BPF program.

- `can_live_reconfigure`: Unconditionally returns `true` to indicate to Click that the BPF program is live reconfigurable.
- `bpfelement_id`: Returns the configured (unique) ID of the `BPFElement` necessary to address a specific `BPFElement` for reconfiguration.

Each BPF-based element extends `BPFElement` and augments the *generic* logic with *specialized* functionality and behavior. The core of the child elements' logic is contained in `push`, which is called when handling a specific packet.

5.3.1 Configuration

As described in Listing 1, Click elements can be configured using the *Click configuration language*. This language not only allows for the description of the flow of the packets but also allows for the configuration of each element itself.

```
BPFFilter(ID 1, FILE rate-limiter, SIGNATURE rate-limiter.sig, JIT false, DUMP_JIT false)
```

Listing 8: Example configuration of a `BPFFilter`

In Listing 8, an example configuration for a `BPFFilter` is shown. The configuration string (i.e., `ID 1 ...`) contains the following key-value pairs:

- `ID`: A unique ID across all BPF elements used to address a specific element when reconfiguring.
- `FILE`: The path to the BPF program. This BPF program is loaded into the VM and executed for every packet.
- `SIGNATURE`: The path to the BPF program's signature confirming successful verification.
- `JIT`: The flag indicates whether the BPF program should be JIT-compiled or not.
- `DUMP_JIT`: The flag indicates whether the JIT-compiled machine code should be dumped into the file system.

5.3.2 `BPFFilter`

The `BPFFilter` element *filters* the incoming packet, as Section 4.1.1 (Filtering) describes.

```rust
#![no_std]
#![no_main]

#[derive(Copy, Clone)]
#[repr(u32)]
pub enum FilterResult {
    Abort = 0,
    Drop = 1,
    Pass = 2,
}

#[no_mangle]
#[link_section = "bpffilter"]
pub extern "C" fn main() -> FilterResult {
    FilterResult::Pass
}
```

Listing 9: Minimal `BPFFilter` program in Rust allowing all packets to pass.

The executed BPF program's return code defines the action taken upon the packet. The following valid return codes exist:

- Abort (return code `0`): An error occurred during packet processing. The packet is dropped, and an error is printed to `stderr`.
- Drop (return code `1`): The packet is dropped.
- Pass (return code `2`): The packet is forwarded to the next element.

5.3.3 `BPFClassifier`

The `BPFClassifier` element *classifies* packets according to the BPF program logic, as explored in Section 4.1.2 (Classification).

```rust
#![no_std]
#![no_main]

#[no_mangle]
#[link_section = "bpffilter"]
pub extern "C" fn main() -> u32 {
    0
}
```

Listing 10: Minimal `BPFClassifier` program in Rust forwarding all packets to output `0`.

Depending on the BPF program's output, the packet is routed to the configured outputs. The element itself can have any amount of configured outputs.

The index of the output corresponds to the return code, which has to be returned to forward to it, i.e., a program returning `1` will forward the packet to the output with index `1`. If the return code results in a forward to an undefined output, the packet is dropped, and an error is written to `stderr`.

5.3.4 `BPFRewriter`

The `BPFRewriter` element can *rewrite*, i.e., modify packets as described in Section 4.1.3 (Rewriting). In contrast to the other BPF-based elements, the `BPFRewriter` does *not* influence the *flow* of the packet but its contents themselves.

```rust
#[repr(C)]
#[derive(Copy, Clone)]
pub struct BpfContext {
    pub data: *mut u8,
    pub data_end: *mut u8,
}

#[derive(Copy, Clone)]
#[repr(u32)]
pub enum RewriterResult {
    Abort = 0,
    Success = 1,
}

#[no_mangle]
#[link_section = "bpffilter"]
pub extern "C" fn main(ctx: *mut BpfContext) -> RewriterResult {
    let mut ctx = unsafe { *ctx };

    // check whether the packet is at least 1 byte long
    if ctx.data as usize > ctx.data_end as usize {
        return RewriterResult::Abort;
    }

    unsafe { *ctx.data = 0x00; }

    RewriterResult::Success
}
```

Listing 11: Minimal `BPFRewriter` program in Rust setting the first byte of the packet to `0x00`.

The output of the BPF program is a simple boolean indicating whether the execution was successful or not. If `Abort` (return code `0`) is returned, an error is written to `stderr`, and the packet is dropped.

The program can access and modify the packet's buffer using the `data` and `data_end` pointers.

5.3.4.1 Resizing packets

Modifying packets does not only include modifying their content but also potentially adjusting the size of the packet itself.
We added a new BPF helper with the signature `bpf_packet_add_space(int32_t head, int32_t tail)` for this task. This helper allows space at the head and tail of the packet to be added or removed. If `head` or `tail` is positive, space is added; if `head` or `tail` is negative, space is removed. Both operations are allowed simultaneously, so, for example, it is permitted to add space to the front of the packet while also shrinking space at the packet's tail.

Internally, the BPF helper is implemented using Click's `Packet#push` and `Packet#pull` for modifying the head, as well as `Packet#put` and `Packet#take` for modifying the tail. As this operation affects verification, we must provide *PREVAIL* with the information that the packet length has changed.

5.4 Live Reconfiguration

Live Reconfiguration in our system, as described in Chapter 4.3 (Live Reconfiguration), consists of three separate steps:
1. The new BPF program is uploaded to the VM.
2. The BPF element reconfiguration is triggered.
3. The BPF element is reconfigured with the new BPF program.

We will discuss these three parts separately.

5.4.1 Uploading BPF programs

Upon initial configuration or reconfiguration, the BPF element reads the BPF program from the *mounted file system*. The first step when reconfiguring is to either replace the old file or create a new one containing the BPF program.

Unikraft supports the 9p passthrough filesystem (9pfs), which allows sharing a directory on the host machine with the guest Operating System [24]. We use this capability to upload BPF program binaries from the host machine to the guest Unikernel.

```
-fsdev local,security_model=passthrough,id=hvirtio1,path=PATH \
-device virtio-9p-pci,fsdev=hvirtio1,mount_tag=fs1 \
-append "vfs.fstab=[\"fs1:/:9pfs\"] --"
```

Listing 12: QEMU CLI arguments for enabling 9pfs with Unikraft

9pfs can be enabled in Unikraft by enabling the `CONFIG_LIBUK9P` flag and adding the arguments seen in Listing 12 to the QEMU command used to start the Unikernel. The `fsdev` and `device` arguments are necessary to configure and attach the file system to the VM, while the `append` argument supplies Unikraft the information to use the 9pfs as its root file system.

5.4.2 Triggering Reconfiguration

As described briefly in Chapter 4.3 (Live Reconfiguration), live reconfiguration is triggered by an external network call.

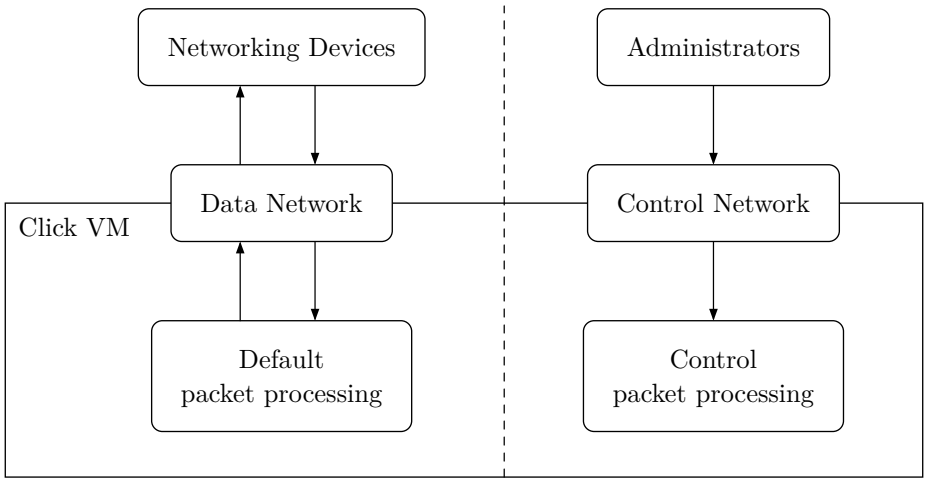

Figure 10: Diagram showing separation between the data & control network.

In contrast to prior work, such as *ClickOS* [13], we avoid tight coupling to a specific hypervisor. In *ClickOS*, *control* communication is relayed via the *XenBus*, coupling *ClickOS* to the *Xen* hypervisor.
Our work replaces this with packets, i.e., *live reconfiguration* can be triggered by sending a User Datagram Protocol (UDP) packet to the VM. This endpoint is secured by binding it to a separate and private *Control Network*.

Like Click's native `ControlSocket`, our live reconfiguration implementation uses its own element. Sadly, the existing `ControlSocket` cannot be reused for our system as

it uses Transmission Control Protocol (TCP) sockets, thus relying on an initialized networking stack. Even though Unikraft supports sockets using *lwIP* [25], Unikraft's glue code automatically initializes and binds to *all* networking devices when enabling `CONFIG_LWIP_AUTOIFACE`. This collides with the `FromDevice` element, which directly operates on networking devices, thus creating a conflict as lwIP is already attached to the networking device.

Thus, we created a custom `Control` element that works without custom sockets. We do this by separating *input* and *processing*, letting our `Control` element handle *processing*, and using the existing Click infrastructure to handle *input*.

5.4.2.1 Preprocessing of `Control` packets

Internally, the `Control` element operates on UDP packets. Thus, the packet must be preprocessed before being forwarded to the `Control` element. We do this by reusing the default Click elements, i.e., chaining elements responsible for preprocessing between the input (`FromDevice`) and the `Control` element.

```
elementclass ControlReceiver { $deviceid |
    FromDevice($deviceid)
     -> c0 :: Classifier(12/0806 20/0001, // ARP requests
                         12/0800,         // IP
                         -);              // rest

    // Answer ARP requests
    c0[0] -> ARPResponder($CONTROL_IP $MAC1)
          -> ToDevice($deviceid);

    // Handle IP packets
    c0[1] -> DropBroadcasts
     -> StripEtherVLANHeader
     -> CheckIPHeader
     -> IPReassembler
     -> SetUDPChecksum
     -> CheckUDPHeader
     -> Control;

    c0[2] -> Discard;
}

// Use the device with ID 1 as the control network
ControlReceiver(1);
```

Listing 13: Click configuration showing how the `Control` element is used.

Specifically, the following steps are taken before forwarding it to the `Control` element.

1. A raw Ethernet frame is received on the *Control Network* and accepted by the `FromDevice` element.
2. The packets are classified by their type. Depending on the type, different processing steps are applied.
   1. ARP requests are processed, and the correct response is returned. This is done so the IP address of the VM on the *Control Network* can be resolved correctly.
   2. IP packets are processed further.
   3. Other packets are dropped.
3. All *broadcasts* are dropped as we do not want to handle any undirected *control* .-packets
4. The Ethernet header is removed in preparation for further processing.
5. The IP packet header is checked for correctness (checksum, …).
6. Potential IP fragmentation is handled.
7. The UDP packet header is checked for correctness, while the checksum is explicitly ignored as the host per default offloads checksum calculation to the Network interface controller (NIC).
8. Now, the `Control` packet is parsed and handled by the `Control` element.

5.4.2.2 `Control` packet parsing

After the `Control` element receives the packet, its first step is to parse it.

prefix = `control`	…	BPF Element ID
Length of BPF program filename		BPF program filename
Length of signature filename		Signature filename

Figure 11: *Control* packet structure.

In the following, we will discuss each field of the `Control` packet as depicted in Figure 11.

- **Prefix (`control`):** Static prefix identifying the UDP packet is a `Control` packet
- **BPF Element ID:** Identifier of the BPF element which the reconfiguration is addressed to
- **Length of BPF program filename:** Character count of the BPF program filename

- **BPF program filename:** The name of the file with the BPF program that should be loaded after reconfiguration
- **Length of signature filename:** Character count of the signature filename
- **Signature filename:** The name of the file with the signature of the BPF program indicating that verification was successful

If any errors in parsing occur, e.g., if the length of the signature filename exceeds the size of the UDP payload, an error is printed to `stderr`, and processing of the `Control` packet is stopped.

5.4.3 Reconfiguring BPF elements

After parsing the packet successfully, the `Control` element invokes the reconfiguration.

First, we iterate through all active elements, filter for BPF elements, and then check whether the BPF element IDs match. If a matching BPF element is found, live reconfiguration is invoked for that specific element.

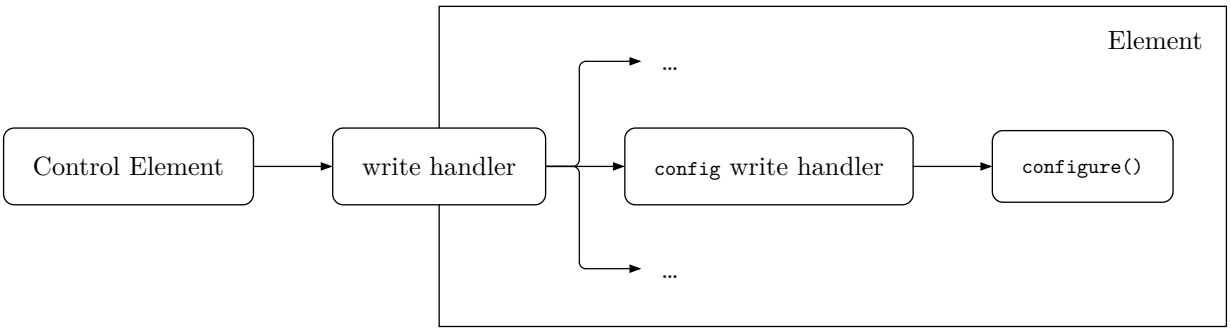

Figure 12: `Control` element integration with live reconfiguration.

We utilize Click's existing *live reconfiguration* framework, which works by invoking a special *write handler*. Handlers in Click are runtime interfaces to elements, allowing users to read statistics or write data. If `Element#can_live_reconfigure()` returns `true`, the built-in `config` write handler allows reconfiguring an element by calling it with a configuration string, as described in Section 5.3.1 (Configuration).

To trigger the write handler for a specific element, the `Control` element first obtains an instance of the `config` handler by invoking Router::handler. It then calls `call_write` with the new configuration string, causing the BPF element to be reconfigured.

The BPF element unloads the old BPF code and proceeds by first verifying the signature, then loading the new BPF code and potentially triggering JIT compilation.

5.5 Decoupled Verification

As described in Chapter 4.4 (Decoupled Verification), we *decouple* the verification of BPF programs to run externally. Instead of being embedded within the Unikernel, the verification logic resides in an external binary written in C++. This external binary performs the verification and, upon success, generates a signature. The Unikernel's role is then limited to verifying this signature.

This section will explore the processes of *verification*, *signing*, and *signature checking*.

5.5.1 Verification

Our system performs verification with the help of the *PREVAIL* eBPF verifier [11]. We embed PREVAIL as a library into the external binary, which additionally contains logic to supply PREVAIL with information about our custom framework, such as helper functions and the context definition.

PREVAIL exposes an Application Programming Interface (API) that can be used to verify a BPF program with custom platform information. Examples of platforms in PREVAIL are Linux and Windows, which both expose similar but not the same eBPF primitives. Such differences are, among other things, different BPF helpers or different program context definitions. In our case, both differences apply.

Based on the existing Linux platform with changes in the BPF helpers and context descriptors, the *Click platform* holds the information PREVAIL requires for verifying BPF programs written for our framework.

```
static const struct EbpfHelperPrototype bpf_packet_add_space_proto = {
  .name = "packet_add_space",
  .return_type = EBPF_RETURN_TYPE_INTEGER,
  .argument_type = {
    EBPF_ARGUMENT_TYPE_ANYTHING,
    EBPF_ARGUMENT_TYPE_ANYTHING,
  },
  .reallocate_packet = true,
};
```

Listing 14: Helper Prototype for the `packet_add_space` BPF helper.

For the BPF helpers, we add prototypes for all implemented helpers, such as `map_lookup_elem` or `ktime_get_ns`. As seen in Listing 14, the helper `packet_add_space`, unique

to the `BPFClassifier` element, has to set the `reallocate_packet` flag to inform the verifier to discard all invariants made about the packet size.

```
constexpr ebpf_context_descriptor_t bpfcontext_descr = {
    .size = 16, // size of context in bytes
    .data = 0,  // offset of pointer to data in context
    .end = 8,   // offset of pointer to end of data in context
    .meta = -1  // offset of pointer to metadata
};
```

Listing 15: Descriptor for our BPF context.

We define a descriptor to support our custom BPF context, as shown in Listing 14. We set `meta` to `-1` to indicate that we do not have a metadata field, contrary to Linux's or Windows' XDP.

For every executable ELF section in the supplied BPF program, we begin by calling *PREVAIL's* `unmarshal` to parse the eBPF instructions. Once the instructions are parsed, we call `ebpf_verify_program` with the instructions and our custom *Click platform*. After completion, `ebpf_verify_program` returns a `boolean` indicating whether the program was successfully verified.

If the verification step has failed, the program exits, prints an error, and *does not* generate a signature.

5.5.2 Signing

Upon successful verification, the verifier generates a signature. This step is implemented using *OpenSSL 3.0.13* and works by creating a *SHA256* hash of the file containing the BPF program and signing that hash using *ECDSA* by calling `EVP_DigestSign`.

The user supplies the private key for signing by passing a command line argument with the path to a PEM file containing the signature. After the BPF program is signed, the signature is stored in a file whose path is supplied via command line arguments.

5.5.3 Signature checking

The signature of a BPF program is checked *on the Unikernel* before the BPF program is loaded into the eBPF VM. This means that, besides the BPF program itself, the signature file must also be included inside the Unikernel's filesystem.

The signature verification inside the Unikernel is done using *OpenSSL 1.1.1c*. We could not use the same OpenSSL version as the verifier because Unikraft bundles

OpenSSL 1.1.1c instead of the newer version. Unikraft directly allows OpenSSL to be enabled by enabling the `CONFIG_LIBOPENSSL` option in the build options.

Checking the signature itself functions similarly to signing. After creating the BPF program hash, we call `EVP_DigestVerify` to check whether the signature is correct. If the signature is correct, we proceed with initializing the BPF element. However, if the signature is invalid, Click is informed that an error occurred during configuration, causing Click to exit.

6 Evaluation

This chapter presents an evaluation of our implementation *quantitatively* and *qualitatively*,.

We will benchmark key performance metrics, including *startup time* in Chapter 6.4 (Startup Time), *throughput* in Chapter 6.7 (Throughput), *reconfiguration time* in Chapter 6.5 (Reconfiguration Time), *memory footprint* in Chapter 6.6 (Memory Footprint), and *latency* in Chapter 6.8 (Latency), to provide a detailed understanding of the system's capabilities and limitations.

Additionally, we will explore aspects such as ease of development in Chapter 6.1 (Ease of Development), state migration in Chapter 6.3 (State Migration), and flexibility from a framework user's perspective in Chapter 6.2 (Flexibility), offering insights into the overall usability.

For our quantitative measurements, we conducted our measurements on a physical machine with the following specifications:

- **CPU:** Intel Xeon Gold 5317 - 12 Cores @ 3.00 GHz, turbo @ 3.60 GHz, Simultaneous multithreading (SMT) disabled)
- **Memory:** 251 GiB @ 2933 MT/s
- **Drive:** 2x Samsung MZ1L21T9 with 1.75 TiB

We pin the `qemu` process to two dedicated cores for all measurements to reduce measurement noise. Additionally, we repeat our measurements multiple times over an extended period and add a warmup phase, if applicable.

To run the experiments, we use a custom benchmarking tool written in Rust. We are using *criterion* [26], a statistics-driven benchmarking library, to measure the startup and live reconfiguration time. For the other measurements, we use a custom benchmarking harness that takes the measurements, persists them, and calculates statistics such as quantiles.

6.1 Ease of Development

Developer friendliness is core to the adoptability of a new framework. The ease of development consists of many key factors, such as *ecosystem support*, *development tools*, *debuggability*, and *learning curve*. Subsequently, we will address each factor.

6.1.1 Ecosystem Support

A robust ecosystem is critical for any technology framework's long-term success and adoption. The strength of the ecosystem surrounding a framework can significantly reduce development friction, offering pre-tested solutions and streamlining the development process. This section explores how we have tried to leverage the existing ecosystem in our work, especially concerning established technologies like eBPF.

6.1.1.1 eBPF

Our system sticks to proven libraries and interfaces for our implementation, allowing us to reuse and benefit from the broader ecosystem surrounding eBPF.

Due to the similarities between the *eBPF for Windows* project and ours, we heavily profit from existing experimentation, solutions, and contributions to core libraries like *PREVAIL* or *ubpf*. For example, during implementation, we benefit from their extensive documentation regarding *PREVAIL* and *ubpf*.

Additionally, the nature of eBPF grants us support to write BPF programs in many different programming languages, such as C, C++, or Rust, and parts of their ecosystem along with it. In our experience, we could reuse many libraries, such as *aya-bpf* [27] or *network-types* [28]. Still, constrained by eBPF runtime requirements, we had to limit ourselves to libraries without memory allocations. Additionally, it has to be noted that some libraries, such as *flex-dns* [29], which we have used to implement a Domain Name System (DNS) filter, caused the program to fail verification. We expect most more complicated libraries to face similar issues. These issues can only be addressed by forking the library, addressing the failures, or completely disabling verification.

6.1.1.2 Click

By structuring our BPF integration as a Click element, we also allow tight integration with the broader Click ecosystem, allowing us to reuse existing elements and chain them together. By reusing existing elements, we can avoid a lot of logic duplication.
If we want to build a system that performs deep packet inspection on the entire payload of IP packets, we want to reassemble fragmented packets. This is possible by using the existing `IPReassembler` element.

6.1.2 Development Tools

As eBPF programs are written in widespread programming languages, many robust Integrated Development Environments (IDEs) exist with which many developers are

already familiar. Popular IDEs such as Visual Studio Code, Vim, or CLion offer syntax highlighting, advanced refactoring support, and code completion, streamlining and easing the overall development process.

However, a significant limitation arises from the lack of tooling for the Click configuration language. While the language's modularity and flexibility are powerful, the absence of robust IDE support hampers development speed. This often forces developers to spend time manually diagnosing syntax errors and troubleshooting otherwise trivial errors.

Since our BPF programs do not run in a standard Linux environment, many existing eBPF tools like *bpftool* or *bcc* are incompatible with our framework. This limitation increases development complexity, requiring custom tools or workarounds to replicate the missing functionality.

6.1.3 Debuggability

Debugging eBPF programs can be challenging as many traditional tools like *gdb* do not support eBPF. While Linux's eBPF ecosystem provides some debugging aids, these tools still require deep familiarity with eBPF and possibly Kernel internals. However, our framework is even more constrained as these options are unavailable to us, forcing users to rely on more primitive debugging techniques like printing data to the console.

Additionally, our use of Unikraft further complicates debugging. While the Unikernel architecture provides performance benefits, it lacks the mature debugging tools and ecosystem support typically available in more established operating systems. This limitation makes diagnosing and troubleshooting issues more challenging.

That said, Unikraft still offers *some* benefits in debuggability compared to a standard Linux environment. As the operating system and application are statically linked into a single binary, tracing the source of issues can be simplified [30].
Furthermore, due to the reduced complexity of the overall system, debugging low-level issues can be potentially more accessible. This is further supported by Unikraft's straightforward compilation process, which allows developers to easily modify and experiment with code. Developers can also enable compile-time debugging options, such as Unikraft's built-in debug logs, to assist in identifying issues during development.

Besides debugging runtime issues, developers must additionally confront and diagnose verifier rejections. Although *PREVAIL* provides good verifier rejection logs, resolving each rejection can still be complex. These logs offer valuable insights into why

a program failed verification. However, interpreting and addressing the issues often requires an intricate understanding of the verifier's capabilities and constraints and a low-level understanding of the program.

6.1.4 Learning Curve

A framework's learning curve is vital to the overall ease of development. The faster developers can become productive, the more likely the framework will gain adoption. In the case of our framework, several aspects impact the learning curve.

6.1.4.1 eBPF

Developers familiar with eBPF and the surrounding ecosystem will find themselves familiar with our system, as many concepts, such as BPF maps and BPF helpers, apply. An important distinction between Linux's eBPF and our eBPF implementation lies in the verifier. According to E. Gershuni *et al.* [11], *PREVAIL* successfully verifies more complicated programs, such as those with loops, than Linux's verifier. Accepting a more comprehensive range of valid programs while offering good logs when rejecting those helps get programmers started and lessens the impact of verification on their overall workflow.

Nevertheless, verification still dramatically impacts the learning curve when writing eBPF. Developers must not only write safe code but also interpret and address rejection logs, which can be intricate and require a deep understanding of the verifier's requirements.

Additionally, eBPF programs do not support dynamic memory allocation, preventing programmers from using dynamically created lists. These restrictions are similar to those in the embedded space, so these are already widely understood and accepted by developers familiar with constrained environments where static memory management is a norm.

Despite these challenges, it is greatly beneficial to a programmer's learning curve to be able to write eBPF in programming languages such as *C* or *Rust* they may already be familiar with. Even if they are unfamiliar, these widespread programming languages are well-documented, with extensive learning resources available. This helps reduce the overall barrier to entry, making it easier to start writing eBPF programs.

6.1.4.2 Click

In the realm of NFV, Click is well-established and widely adopted in research and industry. This familiarity within the NFV space can reduce the learning curve for developers already experienced with Click, allowing them to quickly apply their knowledge to our framework.

However, for those unfamiliar with Click yet, its configuration language and understanding the concept of chaining elements can prove challenging. This is exacerbated by the scarcity of learning resources, which can slow down the process of getting up to speed with Click.

6.1.4.3 Unikraft

Most developers will be unfamiliar with Unikraft or Unikernels in general, making understanding, debugging, and using them a significant barrier to entry. Additionally, as of the time of writing, Unikraft's documentation lacks maturity and extensiveness, which means learning resources are often scarce or outdated. This limited documentation can further slow the learning process, requiring developers to invest additional time in troubleshooting and experimentation to become proficient with Unikraft and its ecosystem.

However, this challenge is alleviated because, in most cases, users of our framework will not need to interact deeply with the Unikernel upon which our system is built. This abstraction helps reduce the complexity and allows developers to focus on higher-level tasks rather than the intricacies of the underlying Unikernel.

6.1.5 Subsummary

Overall, while our framework introduces some learning curve challenges - particularly regarding eBPF verification, understanding Click's configuration language, and navigating Unikraft - the benefits of familiar programming languages, established development tools, and reduced interaction with the Unikernel help mitigate these difficulties.

6.2 Flexibility

A framework's flexibility is crucial for accommodating a wide range of use cases and adapting to varying requirements. Flexibility impacts how easily developers can customize, extend, and integrate the framework to meet specific needs.
We will explore the flexibility of our system in several aspects, such as its *extensibility*

and customizability, by showcasing a comprehensive example demonstrating our system's capabilities to adapt to different use cases.

6.2.1 Extensibility & Customizability

Extensibility and customizability are essential to NFV frameworks, as they must allow users to adapt and expand the overall system to their specific needs. This can range from choosing the *type of network function*, such as a firewall, a load balancer, or a router, to the *algorithm* a load balancer uses for its operations.

To demonstrate these capabilities, we have created a showcase that shows multiple chained Click VM instances, representing a common use case for network functions. The complete source code used for this showcase is available at https://github.com/paulzhng/appclick-ubpf/tree/main/examples/chain.

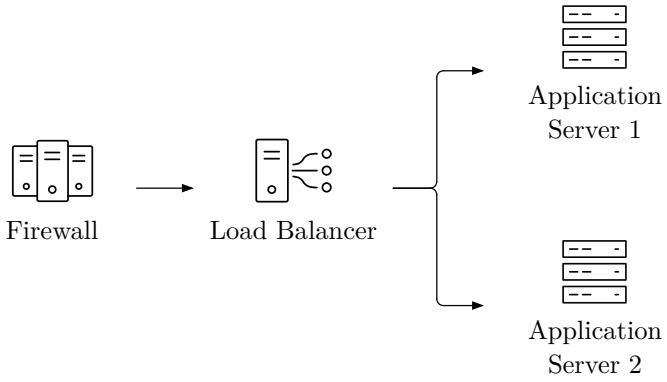

Figure 13: Infrastructure Diagram showing all components from the showcase.

The showcase, as seen in Figure 13, consists of the following components:

- **Firewall:** A firewall blocking packets addressed to port `12345` and rate-limiting incoming packets based on the source IP address.
- **Load balancer:** A *round-robin* load balancer, distributing packets to *application server 1* and *application server 2*.
- **Application server 1 & 2**: An example application server printing a log entry to `stdout` when receiving a packet.

The following will explain the *firewall* and the *load balancer* and how our BPF elements are used.

6.2.1.1 Firewall

```
-> BPFFilter(ID 1, FILE target-port, SIGNATURE target-port.sig, JIT true)
-> BPFFilter(ID 2, FILE rate-limiter, SIGNATURE rate-limiter.sig, JIT true)
```

Listing 16: Excerpt of the firewall's click config.

As seen in Listing 16, the firewall's capabilities are implemented using two chained `BPFFilters`. The first element blocks packets addressed to port `12345`; the second applies token-based rate limiting to all packets. While the `target-port` eBPF program is simple and stateless, the `rate-limiter` uses BPF maps to store the accumulated tokens for each IP address. Additionally, as tokens are distributed over time, the `bpf_ktime_get_ns` helper gets the time.

After passing the filtering steps, the packets are forwarded to the load-balancing VM.

6.2.1.2 Load Balancer

```
lb :: BPFClassifier(ID 1, FILE round-robin, SIGNATURE round-robin.sig, JIT true);
lb[0] -> ...
lb[1] -> ...
```

Listing 17: Excerpt of the load balancer's click config.

The load balancer uses a `BPFClassifier`, which uses the `round-robin` BPF program to distribute packets to the first and second output in a round-robin fashion.

The round-robin algorithm uses a BPF map of type `BPF_MAP_TYPE_ARRAY`, holding one counter. After each packet is received, the counter is incremented, and the packet is sent to the $\text{counter} \bmod 2$-th output.

Afterward, the packets are forwarded to the applicable application server.

6.2.1.3 Subsubsummary

The showcase illustrates the framework's extensibility and customizability through eBPF-based firewall and load balancer implementations. This demonstrates the framework's flexibility to adapt and extend functionality for various network use cases.

6.2.2 Integration with Orchestrators

Apart from integrating into *Click* and thus into their existing components, as already outlined in Section 6.1.1.2 (Click), we aim for our system to seamlessly integrate into existing infrastructure and management systems.

Our system is built on top of Unikraft, which runs on VMs, allowing it to be deployed on popular cloud services such as Amazon Web Services (AWS) [31]. This lets our system automatically take advantage of advanced features like autoscaling and resource management offered by these providers. Additionally, in alignment with standard practices in the NFV space, Unikraft supports running on Kubernetes [32], providing further flexibility and scalability to the deployed VNFs.

The existing support for orchestration frameworks allows efficient resource utilization and rapid adaptation to network demands, which is particularly useful in high-performance NFV environments.

6.2.3 eBPF Limitations

eBPF has inherent limitations that impact how flexible and customizable it can be. Several key constraints include:

- **Verifier Limits:** While *PREVAIL* allows more programs than Linux's verifier, false-positive rejections remain prevalent. Even after adjusting a program according to the rejection logs, satisfying the verifier checks is only sometimes possible. For example, specific memory access patterns can trigger rejections due to overly conservative safety checks. Even when these operations are logically sound, the verifier might still block them.
- **Limited Instruction Set:** The eBPF instruction set is deliberately limited in capabilities. For some use cases like load balancing with weights, *floating point operations* are crucial, though unsupported in eBPF. This forces the use of workarounds like fixed-point numbers.
- **No Input/Output (I/O) Operations:** eBPF programs cannot perform direct I/O, such as reading files or interacting with the network. Thus, interactions with external systems must use BPF helpers, which must be manually implemented inside the framework. Examples for I/O in NFV are observability, i.e., writing logs, sending metrics, etc., or also writing data to disk for caching services.

These limitations heavily restrict the flexibility of eBPF programs. However, some constraints may be no significant problem, such as missing I/O operations, which can be too costly for high-throughput network systems.

6.2.4 Subsummary

In conclusion, we have shown that our BPF extensions to Click and the Unikernel-based runtime environment provide a highly flexible NFV framework, supporting modern orchestrators and complex networking requirements.
However, this flexibility is constrained by eBPF, where restrictions like the lack of dynamic memory allocation and verifier limitations hinder more complex or dynamic use cases.

6.3 State Migration

One of our framework's main benefits is the possibility to reconfigure BPF programs while retaining state, even if the shape of the stored data changes. As Section 4.3.1 (State Retention) outlines, this is achieved by retaining state across live reconfigurations, which then can be used to *migrate* data from the old to the new BPF map.

This section evaluates this approach by showcasing a simple example of state migration, with the approach shown in Figure 7.

6.3.1 Example

```
#[map(name = "PACKET_CTR_V1")]
static PACKET_CTR: Array<u32> = Array::with_max_entries(1, 0);

#[no_mangle]
#[link_section = "bpffilter"]
pub extern "C" fn main() -> FilterResult {
    let Some(counter) = PACKET_CTR.get_ptr_mut(0) else {
        return FilterResult::Abort;
    };

    unsafe { *counter += 1 };

    if unsafe { *counter } > 10 {
        FilterResult::Drop
    } else {
        FilterResult::Pass
    }
}
```

Listing 18: Example BPF program with old data structure.

As a baseline, we have created a simple `BPFFilter` program shown in Listing 18 that increments a counter, and if the counter exceeds 10, it drops all further packets. The counter is stored using a BPF map of type `BPF_MAP_TYPE_ARRAY`.

```
#[map(name = "PACKET_CTR_V2")]
static PACKET_CTR_V2: Array<u64> = Array::with_max_entries(1, 0);

#[no_mangle]
#[link_section = "bpffilter"]
pub extern "C" fn main() -> FilterResult {
    if let Err(_) = handle_migration() {
        return FilterResult::Abort;
    }

    let Some(counter) = PACKET_CTR_V2.get_ptr_mut(0) else {
        return FilterResult::Abort;
    };

    // ...
}
```

Listing 19: Example BPF program using new data structure.

In Listing 19, the updated version of the BPF program can be seen. While the core logic is the same, we have changed the `u32` counter variable to a `u64`, exemplifying a change in the data's structure. We also do not use the same map anymore: We have created a new BPF map called `PACKET_CTR_v2`. Additionally, we invoke `handle_migration`, which handles the migrating data from the old BPF map to the new BPF map.

```rust
#[map(name = "PACKET_CTR_V1")]
static PACKET_CTR_V1: Array<u32> = Array::with_max_entries(1, 0);

#[map(name = "VERSION")]
static VERSION: Array<u64> = Array::with_max_entries(1, 0);

fn handle_migration() -> Result<(), ()> {
    let current_version = VERSION.get_ptr_mut(0).ok_or(())?;

    if unsafe { *current_version } != 2 {
        let old_counter = PACKET_CTR_V1.get(0).ok_or(())?;
        let new_counter = PACKET_CTR_V2.get_ptr_mut(0).ok_or(())?;

        unsafe {
            *new_counter = *old_counter as u64;
            *current_version = 2;
        };
    }

    Ok(())
}
```

Listing 20: State Migration implementation.

The implementation of `handle_migration`, as seen in Listing 20, uses one BPF map to store whether the data was already migrated, and the old BPF map which holds the old counter. If the migration has not yet happened, the old counter is stored, converted to a `u64`, and then stored in the new BPF map while additionally storing the updated version to indicate the migration was run.

After running the example, it worked as expected, and the data was migrated to the new format and BPF map.

6.3.2 Subsummary

We have shown that it is possible to not only retain state with similar shapes across live reconfigurations but also *migrate* data structures, even if the general shape of the data structure changes. This is a considerable benefit compared to the existing Click live reconfiguration, which often does not even allow live reconfiguration for stateful elements.

6.4 Startup Time

Startup Time is an essential metric for VNFs as many network functions face fluctuating access patterns, lending themselves to dynamic scaling. Instead of provisioning

resources based on the *worst-case* scenario, resources can be allocated based on the *current* demand, leading to significant savings in computational resources and costs.

6.4.1 Setup

We measure the time it takes for a Click VM to be ready to process packets. The timer begins when the VM is launched via the appropriate `qemu` command and stops when a specific string, printed when Click receives a packet, is detected in qemu's stdout. In the background, the harness sends packets to the VM at regular intervals to ensure it triggers the printing.

We measure different scenarios as startup time heavily depends on a use-case basis. Many examples are slightly modified versions of publically available Click configurations. We have chosen the following examples to cover Click configurations of various complexity, sorted by the number of lines:

- `minimal`: Minimal possible set of Click elements
- `print-pings`: Prints Internet Control Message Protocol (ICMP) echo requests [33]
- `switch-2ports`: Simple 2-port switch [34]
- `thomer-nat`: A firewall and NAT gateway [35]
- `router`: A L3 router with 4 I/O ports [34]

6.4.2 Results

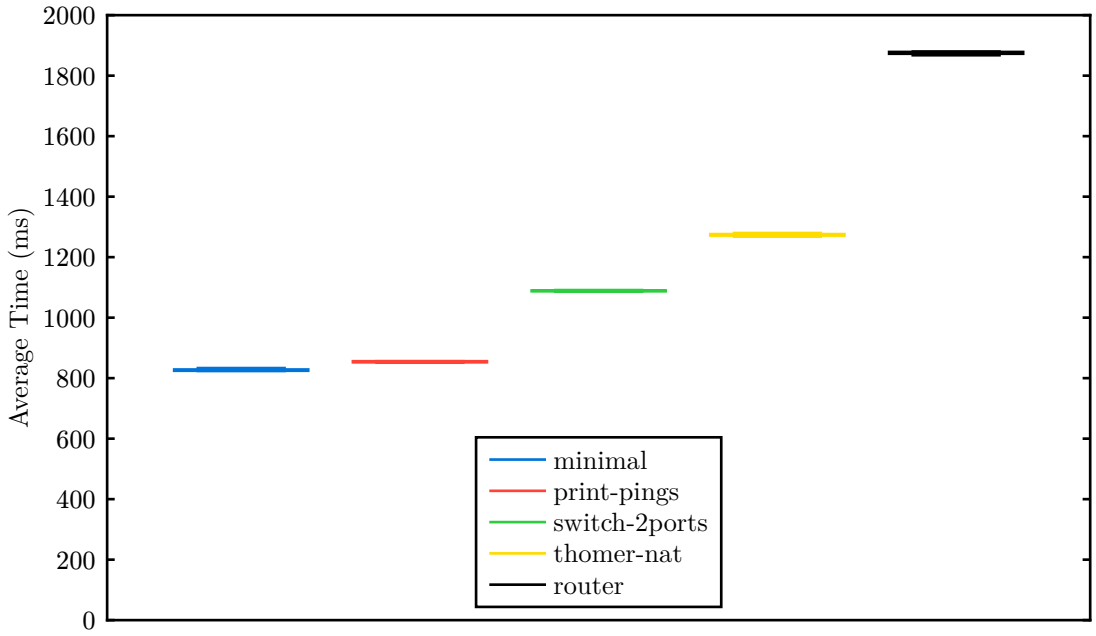

Figure 14: Startup times for the tested Click configurations.

The results shown in Figure 14 clearly show that the overall startup time of a VM heavily scales with increasingly complex configurations. While the minimal example takes roughly 0.8 seconds, the more complex router takes 1.85 seconds on average to boot.

These startup times are significantly better than a full VM running a Linux kernel can achieve [36], proving the benefits of running Click in a Unikernel. Still, these results are worse than J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, and F. Huici [13] have achieved with 30 ms, which has to be investigated further.

6.4.3 Subsummary

We have shown that short boot times can be achieved by using a Unikernel. Our minimal example took ca. 0.8 seconds to boot, while our most complex example took 1.85 seconds. These results bolster our claims that our work can effectively be used with *autoscaling*.

6.5 Reconfiguration Time

One of our core principles is that our framework enables code changes to be deployed without causing downtime. Therefore, minimizing reconfiguration times is crucial.

6.5.1 Setup

To measure the reconfiguration time, we are taking the time from the start of reconfiguration to the end, i.e., the duration that Click will delay any incoming packets due to the reconfiguration. Explicitly, we do not start reconfiguration when sending the *Control* packet, but when the *Control* packet has already been received and reconfiguration has been triggered on the element.

These measurements are run with different configured BPF programs, both with and without JIT compilation enabled. We test the following BPF programs:

- `pass`: Simple `BPFFilter` program allowing all packets to pass through
- `rate-limiter`: `BPFFilter` program, which rate-limits requests based on the IP address
- `round-robin`: `BPFClassifier` program, which does round-robin load balancing
- `strip-ether-vlan-header`: `BPFRewriter` program removing the Ethernet header

6.5.2 Results

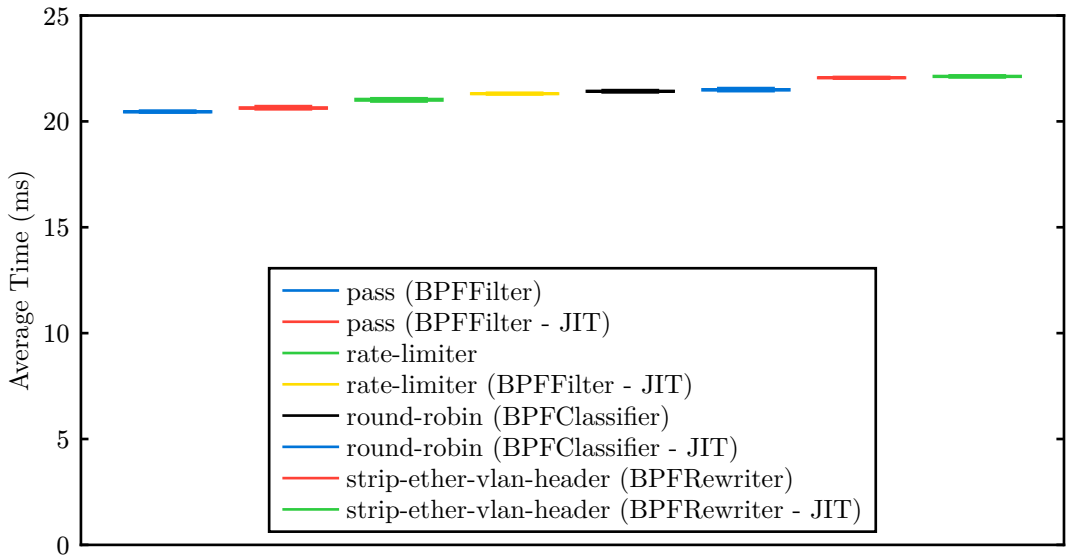

Figure 15: Live reconfiguration times for different tested Click BPF programs.

Our tests show that most live reconfigurations take around 21 ms, with more complex programs adding only a few extra milliseconds. Enabling JIT compilation also slightly increases reconfiguration times by a few milliseconds.

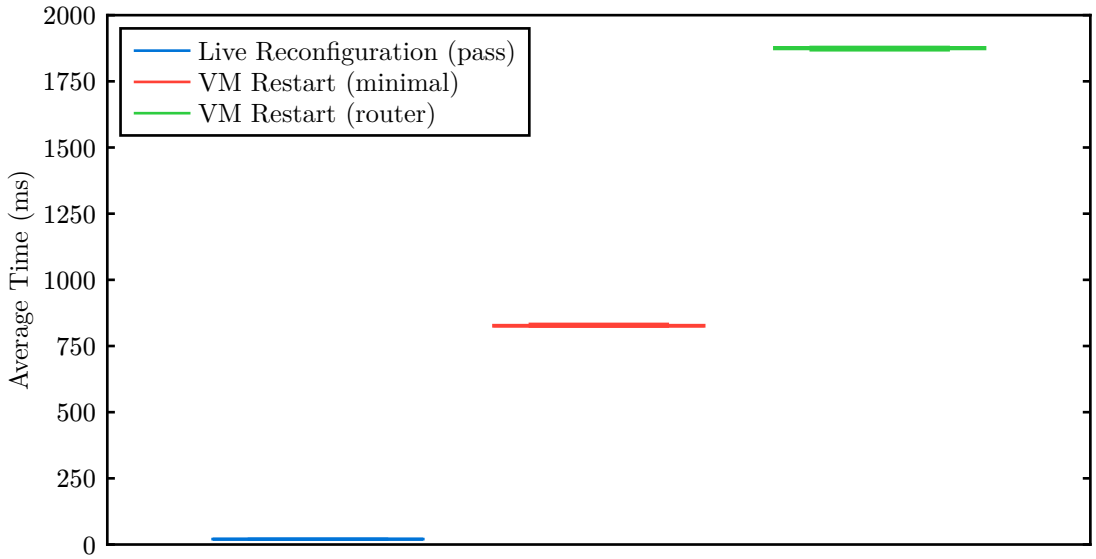

Figure 16: Live reconfiguration compared to a full VM restart.

Compared to full restarts, the live reconfiguration times are up to two orders of magnitude, showing a clear benefit.

6.5.3 Subsummary

Our tests have shown that live reconfiguration takes about 21 milliseconds, with minimal increases for complex programs or JIT compilation. This fast reconfiguration significantly outperforms complete VM restarts, ensuring efficient updates with minimal service disruption.

6.6 Memory Footprint

A low memory footprint is crucial for efficiently running many Click VMs on a single physical machine, which is common in NFV. Sharing a physical host among VNFs can have many benefits, including cost savings and latency improvements due to shorter Round-trip times (RTTs) between the VMs.

6.6.1 Setup

The measurements are run by starting the VM, then waiting until Click has finished the complete startup process, and then taking a sample of the amount of memory the VM is using. This number is estimated by measuring the Resident set size (RSS) of the qemu process and multiplying it by the system page size.

We will test the following scenarios, with and without JIT enabled, when testing BPF elements:

- `baseline`: No BPF element configured
- `pass (IPFilter)`: Built-in `IPFilter`, which allows all packets to pass
- `pass (BPFFilter)`: `BPFFilter` which allows all packets to pass
- `2x pass (BPFFilter)`: Two chained `BPFFilters` which allow all packets to pass
- `pass & drop (BPFilter)`: Two chained `BPFFilter`, one allowing all packets to pass and one dropping all packets
- `round-robin (BPFClassifier)`: `BPFClassifier` program, which does round-robin load balancing

6.6.2 Results

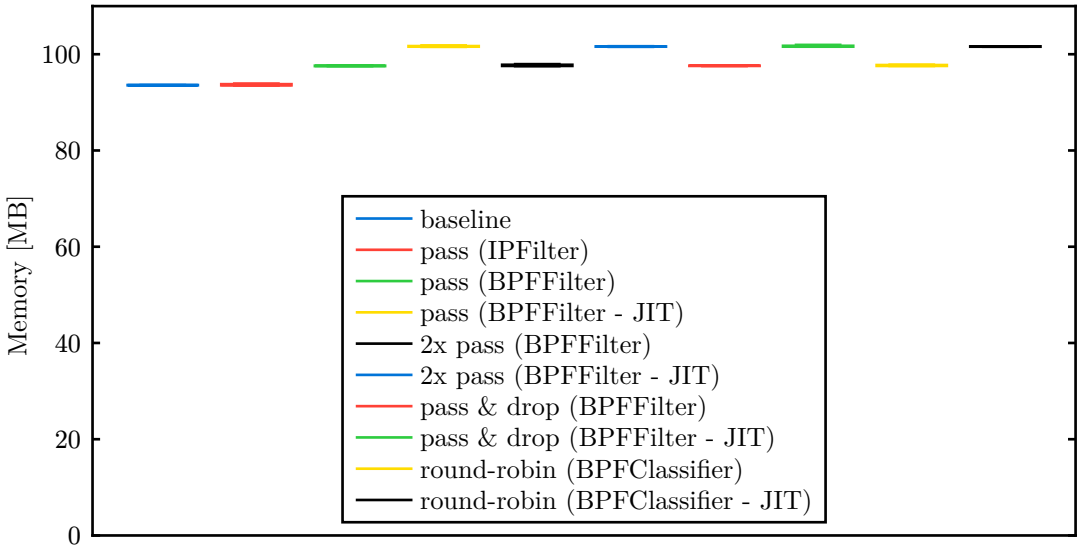

Figure 17: Memory usage for the tested scenarios.

Scenario	Memory Usage	Compared to baseline
baseline	93.59 MB	+0 MB (+0%)
pass (IPFilter)	93.59 MB	+0 MB (+0%)
pass (BPFFilter)	97.57 MB	+3.98 MB (+4%)
pass (BPFFilter - JIT)	101.62 MB	+8.03 MB (+9%)
2x pass (BPFFilter)	97.64 MB	+4.05 MB (+4%)
2x pass (BPFFilter - JIT)	101.63 MB	+8.04 MB (+9%)
pass & drop (BPFFilter)	97.62 MB	+4.03 MB (+4%)
pass & drop (BPFFilter - JIT)	101.63 MB	+8.04 MB (+9%)
round-robin (BPFClassifier)	97.62 MB	+4.03 MB (+4%)
round-robin (BPFClassifier - JIT)	101.63 MB	+8.04 MB (+9%)

Table 2: Median memory usage for the tested scenarios.

Our measurements show that, as a baseline, a Click VM without any configured BPF elements uses roughly 93.5 MB of memory. Using an `IPFilter` element does not notice-

ably increase that usage, while a `BPFElement` has a static overhead of around 4 MB, with JIT adding another 4 MB.

The memory overhead of using a `BPFElement`, with and without JIT, is only paid once. As seen with `2x pass` and `pass & drop`, the memory usage does not increase noticeably compared to `pass`. This leads us to conclude that the noticeable memory overhead consists mainly of the increase in code size.

Compared to *ClickOS* [13], our measured results are dramatically different. The authors of *ClickOS* measured ca. 15 MB of usage per VM. These differences must be investigated further.

6.6.3 Subsummary

Our tests show a base usage of 93.5 MB of memory with minimal memory overhead for using BPF elements in Click. Using BPF elements adds about 4 MB of memory usage and an extra 4 MB if JIT is enabled. Significantly, this memory usage does not increase when using multiple BPF elements or with growing BPF program complexity.

6.7 Throughput

High throughput is a critical metric in NFV, as higher throughput corresponds to smaller resource utilization and better performance in general. VNFs potentially operate in high-load environments, which our framework needs to be able to handle.

6.7.1 Setup

Our setup consists of a Click VM with an `InfiniteSource` element that generates packets as fast as possible, encapsulates them in a UDP packet, passes them through the tested Click element, and then sends the packets to the VM's host. We are measuring the achieved packets per second and bytes per second on the host using `bmon` [37].

We test the throughput of the following scenarios, once with built-in Click elements, and then with BPF elements with and without JIT:

- `baseline`: No in-between processing is happening - the packets are directly sent to the host
- `target-port`: All packets with the destination port `12345` are dropped - tests the throughput of `BPFFilter`
- `round-robin`: Round-robin load balancing to different outputs - tests the throughput of `BPFClassifier`

- `strip-ether-vlan-header`: Removes the Ethernet header from the packet - tests the throughput of `BPFRewriter`

6.7.2 Results

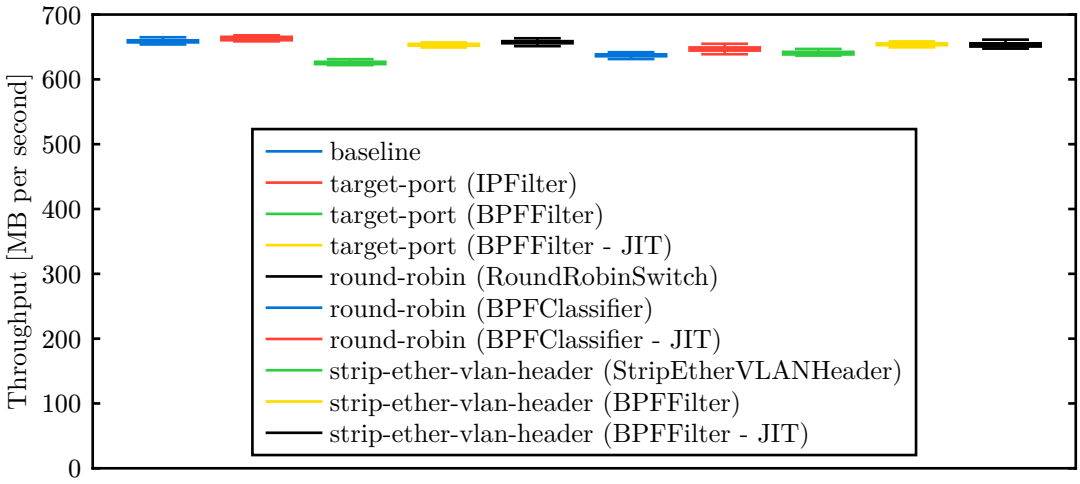

Figure 18: Processed bytes per second for the tested Click elements.

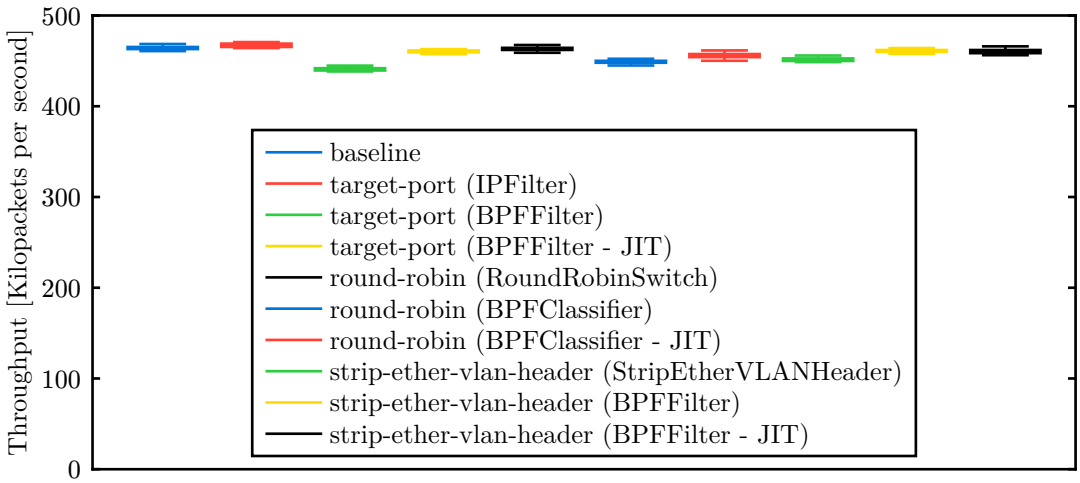

Figure 19: Processed bytes per second for the tested Click elements.

Our measurements show that, in relative terms, the use of BPF elements does not affect overall throughput, with the differences being in the realm of measurement errors. The negligible impact of the tested elements on overall system performance can explain this.

Our baseline shows that Click on Unikraft can process roughly 460k packets per second and 657 MB per second, which is worse than the results achieved by ClickOS, which achieved a throughput of up to 9.68 GB/s [13]. ClickOS had similar performance indicators before optimizing its networking pipeline, dramatically improving performance. These optimizations can be applied to our work as well.

6.7.3 Subsummary

Our results show that using BPF-based elements does not affect overall system throughput to be noticeable. However, the measured throughput of our overall Click system still needs improvement and has to be improved to be usable in production workloads. As shown by *ClickOS*, these optimizations are feasible and can improve performance so that the framework is usable even in high-load environments [13].

6.8 Latency

Besides throughput, *minimizing* the impact of our framework on latency is vital in ensuring optimal performance. High latency can degrade the overall network's performance and reduce the viability of using network functions, making it critical that our system introduces minimal delays.

6.8.1 Setup

We are measuring the RTT from the host to the Click VM by sending an ICMP Echo Request to the VM configured to respond with an appropriate Echo Reply. Before responding to the Echo Request, the Click VM is configured to pass the ICMP packet through a BPF element, effectively adding the latency caused by the BPF element.

We run the tests for the following scenarios, with and without JIT enabled:

- `baseline`: No BPF element is added
- `1 element`: The packet is passed one time through the `pass` BPF program, which allows all packets to pass through
- `10 elements`: The packet is passed ten times through the `pass` BPF program

6.8.2 Results

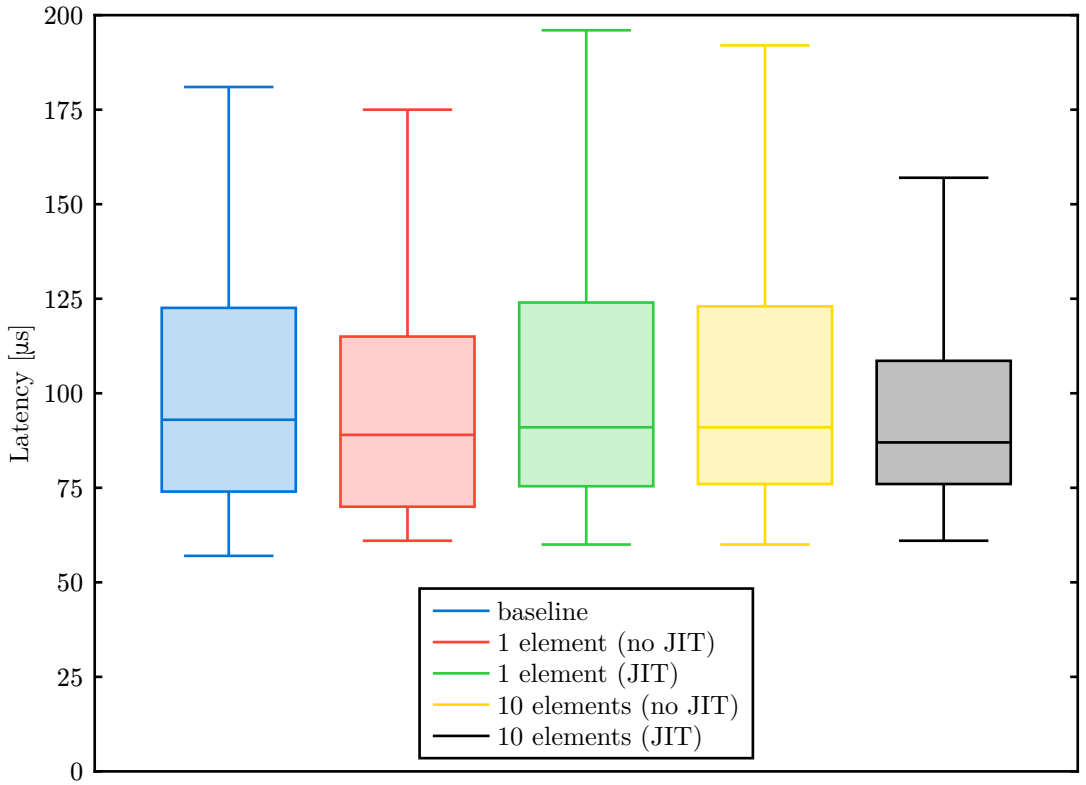

Figure 20: Measured RTT latency from the host to the Click VM for different scenarios.

Our results show that the median RTT from the host to the VM is 93 µs for the baseline. We measured no noticeable impact of BPF elements, even when chained multiple times. Similarly, JIT has no impact on the measured latency.

Compared to ClickOS [13], we see slightly worse numbers, with them measuring latency of 45 µs using similar testing methodologies. We suspect their advantage is caused by their optimized networking stack, which we have not ported to our work.

6.8.3 Subsummary

Our measurements show that, on average, the RTT between the host and the Click VM is 93 µs. Adding BPF elements to the processing chain does not measurably influence latency.

7 Related Work

This chapter explores works related to essential parts of this thesis. We will discuss their approach and explain similarities, as well as differences, to our work.

7.1 Network Function Virtualization (NFV) using Unikernels

Many related works have already addressed using Unikernels for Network Function Virtualization (NFV).

ClickOS [13] similarly ported Click to run on a Unikernel, although choosing *MiniOS* running on Xen instead of Unikraft running on KVM. Their work focused on maximizing the performance achieved by optimizing their network I/O, while ours mainly focused on integrating eBPF into Click. Nevertheless, we could effectively use their benchmarking results as a comparison point to our work. Generally speaking, ClickOS offered better overall performance for metrics such as throughput or latency compared to our work due to optimizations in their networking pipeline. These optimizations can be applied to our work as well.

T. Kurek [2] explores the benefits of using Unikernels instead of Containers for NFV, focusing on the improved security and isolation of VMs compared to containers. Their work compared different firewalls, one running on Docker, one on a Linux VM on KVM, and one based on *IncludeOS*. They found the firewall running on the *IncludeOS* Unikernel to perform the best with many rules while being comparable in performance with a smaller set of rules.

T. Kurek, M. Niemiec, and A. Lason [38] and J. B. Filipe, F. Meneses, A. U. Rehman, D. Corujo, and R. L. Aguiar [39] evaluate the performance characteristics of VNFs running on Unikernels compared to more traditional approaches such as containers or VMs. Both works observed that, in most cases, an *IncludeOS*-based Unikernel outperformed a Linux VM, sometimes achieving better results than containers. However, containers often yielded better results than their VM-based counterparts.

7.2 Network Function Virtualization (NFV) using eBPF

The use of eBPF for NFV is widely researched, although most often focused on Linux's eBPF.

InKeV [40] and Polycube [41] explore a VNF framework with an eBPF-based data plane containing the VNF's logic running inside the Linux kernel and a user space

control plane for managing those VNFs. Their frameworks support *chaining* network functions to form more complex network function chains.

N. Van Tu, K. Ko, and J. W.-K. Hong [42], N. V. Tu, J.-H. Yoo, and J. W.-K. Hong [43] and MiddleNet [44] explore Linux-based, hybrid NFV frameworks that run simpler VNFs in kernel space using eBPF while resorting to user-space processing for more complicated tasks. This approach allows sidestepping eBPF's limitations, adding overall flexibility.

7.3 Decoupled eBPF Verification

Our work decouples eBPF *verification* from its *execution*, as explored by M. Craun, A. Oswald, and D. Williams [14], although focused on the Linux kernel. Because of how Linux verifies its eBPF programs, they decouple both the verification and the JIT process. We can individually decouple the verification process as we do not share those constraints. Similar to our work, they also use cryptographic signatures to guarantee that verification has happened.

UniBPF [45] similarly decouples the eBPF verification process, running the verifier on an external system. Importantly, their implementation trusts all BPF programs to have been verified successfully, opening up the possibility of evading the verification process. Our work solves this issue by introducing signatures.

eBPF for Windows [46], which we reference several times in this thesis, also decouples the eBPF verifier. Unlike our work, this decoupling is not across *machines* but across *user space and kernel space*. The *PREVAIL* verifier runs as a system service, and if the program passes the verifier, it is loaded into the Kernel.

8 Summary and Conclusion

This work has integrated eBPF-based elements into the *Click modular router*, allowing users to program custom filters, classifiers, and rewriters, greatly enhancing Click's flexibility without creating custom elements. We have added many examples showing the system's capabilities, notably a fully integrated cross-VM network function chain, including a firewall, a load balancer, and application servers.

By taking advantage of existing eBPF verifiers, we have built a *decoupled verifier* that offloads eBPF program verification to an external step, speeding up startup and reconfiguration times while retaining eBPF's safety properties. Due to our eBPF runtime's interface closely resembling those of Linux and eBPF for Windows, knowledge about many concepts, such as BPF maps and BPF helpers, is easily transferable.

Integrating the Click framework into a Unikernel, we have created a *secure*, *lightweight*, and *performant* framework for running network functions while minimizing drawbacks in flexibility caused by the Unikernel. We have added *live reconfiguration* capabilities into our system, supporting retaining state across live reconfiguration boundaries. Even if the shape of the data changes, we have shown that it is feasible to add *data migrations*.

Our evaluations show no noticeable throughput and latency penalties when using BPF elements compared to their native counterparts. At the same time, it must be noted that the baseline performance of our system is not production-ready and can be improved.

Overall, we have built a promising and flexible eBPF-based NFV framework, allowing network operators to quickly deploy changes to their infrastructure. The source code for our work is available at https://github.com/paulzhng/appclick-ubpf.

9 Future Work

While our framework has shown promising results, several avenues for improvement remain. In this chapter, we outline potential enhancements and extensions that can be made to our system, addressing limitations encountered during the development and evaluation stages.

9.1 Optimizing the Networking Pipeline

While evaluating our throughput in Chapter 6.7 (Throughput) and latency in Chapter 6.8 (Latency), we find that our baseline performance must be improved for our framework to be usable in production workloads.

It is also possible to apply optimizations similar to previous work, such as *ClickOS* [13], to our system. The performance numbers to ClickOS before any of their optimizations are comparable to ours, which suggests we could substantially improve our performance metrics by applying similar methods.

9.2 Extending Support for BPF Helpers & BPF Maps

Currently, our support for the standard BPF helpers and BPF maps is limited. Deemed outside the scope of our work, we did not implement many, possibly useful, helpers. Similarly, we only implemented two BPF map types, while support for other map types, such as `BPF_MAP_TYPE_QUEUE`, is missing.

To make our eBPF implementation more similar to Linux and provide developers with valuable tools, it would be beneficial to extend our implementation.

9.3 Synchronizing BPF Maps across VMs

Many VNFs, such as Carrier-Grade NATs, are stateful but must be *scalable* or *crash-resistant*. As all state that needs to be persisted across packets is stored in BPF maps by design, it is possible to introduce a new BPF map flag that allows the synchronization of this state across different BPF VMs, essentially allowing to easily create distributed VNFs.

This would introduce several benefits, such as allowing seamless switchovers in case a VM crashes or horizontal scaling of stateful VNFs. Currently, this is impossible with eBPF, as state is always held on one machine only.

Differentiating the *consistency type* would also be possible, depending on the use case. For example, a rate-limiter may run with *eventual consistency*, while another network function may need to run under *strong consistency* guarantees. This would allow users to improve performance if strong consistency is not required.

9.4 Supporting other Hypervisors

In our evaluation, we did not add support to our system for different hypervisors such as *Xen* or *ESXi*. Depending on the use case, they may offer better overall performance. Performance metrics on hypervisors other than KVM would help determine whether other hypervisors are a better fit.

On a related note, replacing our use of *qemu* with *Firecracker* could yield better results for startup times and memory footprint [47], which could be helpful for some use cases.

List of Acronyms

***9pfs* – 9p passthrough filesystem**. 28, 29

***API* – Application Programming Interface**. 33

***ARP* – Address Resolution Protocol**. 5, 31

***AWS* – Amazon Web Services**. 43

***BPF* – Berkeley Packet Filter**. iv, v, vi, vii, 1, 3, 7, 8, 10, 11, 13, 14, 15, 18, 20, 21, 22, 23, 24, 25, 28, 31, 32, 33, 34, 35, 37, 38, 39, 41, 42, 43, 44, 45, 46, 49, 51, 53, 54, 55, 59, 60, 63, 64

***BRAS* – Broadband remote access server**. 11

***DDoS* – Distributed Denial of Service**. 11

***DNS* – Domain Name System**. 37

***DPI* – Deep packet inspection**. 11

***DRY* – Don't repeat yourself**. 24

***eBPF* – extended Berkeley Packet Filter**. iv, v, vi, vii, 1, 3, 6, 7, 8, 9, 10, 12, 13, 15, 16, 18, 19, 20, 21, 23, 24, 34, 37, 38, 39, 40, 42, 43, 57, 58, 59, 60

***ELF* – Executable and Linkable Format**. v, 3, 19, 21, 34

***I/O* – Input/Output**. 43, 47, 57

***ICMP* – Internet Control Message Protocol**. 47, 55

***IDE* – Integrated Development Environment**. 37, 38

***IDS* – Intrusion detection system**. 11, 13

***IP* – Internet Protocol**. 5, 31, 37, 41

***IPS* – Intrusion prevention system**. 11

***IPv4* – Internet Protocol version 4**. 11

***IPv6* – Internet Protocol version 6**. 11

***JIT* – Just-In-Time**. iv, v, 3, 17, 22, 23, 24, 25, 32, 49, 50, 51, 53, 55, 56, 58

***NAT* – Network Address Translation**. 11, 47, 60

***NFV* – Network Function Virtualization**. iv, v, vii, 1, 4, 40, 41, 43, 51, 53, 57, 58, 59

***NIC* – Network interface controller**. 31

***RSS* – Resident set size**. 51

***RTT* – Round-trip time**. 51, 55, 56, 63

***SMT* – Simultaneous multithreading**. 36

***TCP* – Transmission Control Protocol**. 30

***UDP* – User Datagram Protocol**. 29, 30, 31

***VM* – Virtual Machine**. iv, vii, 1, 2, 4, 6, 9, 10, 21, 25, 28, 31, 41, 42, 43, 47, 48, 50, 51, 52, 53, 55, 56, 57, 59, 60, 63

***VNF* – Virtual Network Function**. iv, 1, 43, 46, 51, 53, 57, 58, 60

List of Figures

List of Tables

Bibliography

[1] Equinix, "Organizations Increasing Their Adoption of Network Functions Virtualization to Reduce Complexity and Ease Management." [Online]. Available: https://www.equinix.lat/content/dam/eqxcorp/en_us/documents/resources/whitepapers/wp_techtarget_nfv_ebook_en_may2021.pdf

[2] T. Kurek, "Unikernel Network Functions: A Journey Beyond the Containers," *IEEE Communications Magazine*, vol. 57, no. 12, pp. 15–19, 2019.

[3] Y. Avrahami, "Breaking out of Docker via runC – Explaining CVE-2019-5736." Accessed: Sep. 17, 2024. [Online]. Available: https://unit42.paloaltonetworks.com/breaking-docker-via-runc-explaining-cve-2019-5736/

[4] eBPF.io authors, "What Is eBPF? An Introduction and Deep Dive into the eBPF Technology." Accessed: Aug. 01, 2024. [Online]. Available: https://ebpf.io/what-is-ebpf/

[5] Cilium Authors, "BPF Architecture — Cilium 1.16.0 Documentation." Accessed: Aug. 01, 2024. [Online]. Available: https://docs.cilium.io/en/stable/bpf/architecture/#instruction-set

[6] The kernel development community, "Bpf-Helpers(7) - Linux Manual Page." Accessed: Aug. 01, 2024. [Online]. Available: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html

[7] The kernel development community, "Bpf(2) - Linux Manual Page." Accessed: Aug. 01, 2024. [Online]. Available: https://man7.org/linux/man-pages/man2/bpf.2.html

[8] The Unikraft Authors, "Unikraft - Concepts." Accessed: Aug. 10, 2024. [Online]. Available: https://unikraft.org/docs/concepts

[9] VMware, "What Is Network Functions Virtualization (NFV)? | VMware Glossary." Accessed: Aug. 13, 2024. [Online]. Available: https://www.vmware.com/topics/network-functions-virtualization-nfv

[10] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, Aug. 2000, doi: 10.1145/354871.354874.

[11] E. 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*, Phoenix AZ USA: ACM, Jun. 2019, pp. 1069–1084. doi: 10.1145/3314221.3314590.

[12] The Unikraft Authors, “Unikraft Is a Fast, Secure and Open-Source Unikernel Development Kit..” Accessed: Jun. 01, 2024. [Online]. Available: https://unikraft.org/

[13] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, and F. Huici, “ClickOS and the Art of Network Function Virtualization.”

[14] 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*, New York NY USA: ACM, Sep. 2023, pp. 63–69. doi: 10.1145/3609021.3609299.

[15] The Unikraft Authors, “Unikraft/App-Click.” Accessed: Jun. 01, 2024. [Online]. Available: https://github.com/unikraft/app-click

[16] The Unikraft Authors, “Unikraft/Unikraft: A next-Generation Cloud Native Kernel Designed to Unlock Best-in-Class Performance, Security Primitives and Efficiency Savings..” Accessed: Jun. 01, 2024. [Online]. Available: https://github.com/unikraft/unikraft

[17] The Unikraft Authors, “Unikraft/Lib-Click: Unikraft Port of the Click Modular Router.” Accessed: Aug. 25, 2024. [Online]. Available: https://github.com/unikraft/lib-click

[18] Big Switch Networks, “Iovisor/Ubpf.” Accessed: Aug. 25, 2024. [Online]. Available: https://github.com/iovisor/ubpf

[19] V. Hendrychová, “Vandah/Uk_ubpf.” Accessed: Aug. 25, 2024. [Online]. Available: https://github.com/vandah/uk_ubpf

[20] eBPF for Windows contributors, “Microsoft/Ebpf-for-Windows.” Accessed: Aug. 25, 2024. [Online]. Available: https://github.com/microsoft/ebpf-for-windows

[21] The kernel development community, “BPF Design Q&A — The Linux Kernel Documentation.” Accessed: Aug. 25, 2024. [Online]. Available: https://docs.kernel.org/bpf/bpf_design_QA.html#id7

[22] xdp-project, “Xdp-Tutorial/Packet01-Parsing/README.Org at Master · Xdp-Project/Xdp-Tutorial.” Accessed: Aug. 25, 2024. [Online]. Available: https://github.com/xdp-project/xdp-tutorial/blob/master/packet01-parsing/README.org

[23] D. Thaler, “Xdp_md Should Be Cross-Platform Portable · Issue #3212 · Microsoft/Ebpf-for-Windows.” Accessed: Aug. 25, 2024. [Online]. Available: https://github.com/microsoft/ebpf-for-windows/issues/3212

[24] F. Bellard and the QEMU team, “Documentation/9psetup - QEMU.” Accessed: Aug. 31, 2024. [Online]. Available: https://wiki.qemu.org/Documentation/9psetup

[25] The Unikraft Authors, “Unikraft/Lib-Lwip.” Accessed: Sep. 03, 2024. [Online]. Available: https://github.com/unikraft/lib-lwip

[26] B. Heisler, “Bheisler/Criterion.Rs.” Accessed: Jun. 01, 2024. [Online]. Available: https://github.com/bheisler/criterion.rs

[27] The Aya Contributors, “Aya.” Accessed: Aug. 30, 2024. [Online]. Available: https://crates.io/crates/aya

[28] M. Rostecki, “Network-Types - Crates.Io: Rust Package Registry.” Accessed: Aug. 30, 2024. [Online]. Available: https://crates.io/crates/network-types

[29] L. Camus, “Flex-Dns - Crates.Io: Rust Package Registry.” Accessed: Sep. 05, 2024. [Online]. Available: https://crates.io/crates/flex-dns

[30] The Unikraft Authors, “Debugging a Unikernel.” Accessed: Sep. 05, 2024. [Online]. Available: https://unikraft.org/docs/internals/debugging

[31] The Unikraft Authors, “Unikraft/Plat-Aws.” Accessed: Sep. 07, 2024. [Online]. Available: https://github.com/unikraft/plat-aws

[32] The Unikraft Authors, “Container Runtimes.” Accessed: Sep. 07, 2024. [Online]. Available: https://unikraft.org/docs/getting-started/integrations/container-runtimes

[33] E. Kohler, “Click/Conf/Print-Pings.Click at Master · Kohler/Click.” Accessed: Sep. 08, 2024. [Online]. Available: https://github.com/kohler/click/blob/master/conf/print-pings.click

[34] T. Barbette, “Fastclick/Conf/Router/Router-Vanilla.Click at Main · Tbarbette/Fastclick.” Accessed: Jun. 01, 2024. [Online]. Available: https://github.com/tbarbette/fastclick/blob/main/conf/router/router-vanilla.click

[35] T. M. Gil, “Click/Conf/Thomer-Nat.Click at Master · Kohler/Click.” Accessed: Sep. 08, 2024. [Online]. Available: https://github.com/kohler/click/blob/master/conf/thomer-nat.click

[36] R. S. Auliya, Y.-L. Lee, C.-C. Chen, D. Liang, and W.-J. Wang, “Analysis and Prediction of Virtual Machine Boot Time on Virtualized Computing Environments,” *Journal of Cloud Computing*, vol. 13, no. 1, p. 80–81, Apr. 2024, doi: 10.1186/s13677-024-00646-4.

[37] T. Graf, “Tgraf/Bmon.” Accessed: Sep. 11, 2024. [Online]. Available: https://github.com/tgraf/bmon

[38] T. Kurek, M. Niemiec, and A. Lason, "Performance Evaluation of a Firewall Service Based on Virtualized IncludeOS Unikernels," *Scientific Reports*, vol. 14, no. 1, p. 557–558, Jan. 2024, doi: 10.1038/s41598-024-51167-8.

[39] J. B. Filipe, F. Meneses, A. U. Rehman, D. Corujo, and R. L. Aguiar, "A Performance Comparison of Containers and Unikernels for Reliable 5G Environments," in *2019 15th International Conference on the Design of Reliable Communication Networks (DRCN)*, Coimbra, Portugal: IEEE, Mar. 2019, pp. 99–106. doi: 10.1109/DRCN.2019.8713677.

[40] Z. Ahmed, M. H. Alizai, and A. A. Syed, "InKeV: In-Kernel Distributed Network Virtualization for DCN," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 3, pp. 1–6, Jul. 2018, doi: 10.1145/3243157.3243161.

[41] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu, "A Framework for eBPF-Based Network Functions in an Era of Microservices," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 133–151, Mar. 2021, doi: 10.1109/TNSM.2021.3055676.

[42] N. Van Tu, K. Ko, and J. W.-K. Hong, "Architecture for Building Hybrid Kernel-User Space Virtual Network Functions," in *2017 13th International Conference on Network and Service Management (CNSM)*, Tokyo: IEEE, Nov. 2017, pp. 1–6. doi: 10.23919/CNSM.2017.8256051.

[43] N. V. Tu, J.-H. Yoo, and J. W.-K. Hong, "Building Hybrid Virtual Network Functions with eXpress Data Path," in *2019 15th International Conference on Network and Service Management (CNSM)*, Halifax, NS, Canada: IEEE, Oct. 2019, pp. 1–9. doi: 10.23919/CNSM46954.2019.9012730.

[44] S. Qi, Z. Zeng, L. Monis, and K. K. Ramakrishnan, "MiddleNet: A Unified, High-Performance NFV and Middlebox Framework with eBPF and DPDK." Accessed: Sep. 14, 2024. [Online]. Available: https://arxiv.org/abs/2303.04404

[45] K.-C. Hsieh, "UniBPF: Safe and Verifiable Unikernel Extensions," 2023. Accessed: Jul. 03, 2024. [Online]. Available: https://github.com/TUM-DSE/research-work-archive/blob/0a5d018975baa7868ea29bae1c109e8055ecad74/archive/2023/winter/docs/msc_hsieh_unibpf_safe_and_verifiable_unikernels_extensions.pdf

[46] eBPF for Windows contributors, "Ebpf-for-Windows/Docs at Main · Microsoft/Ebpf-for-Windows." Accessed: Sep. 05, 2024. [Online]. Available: https://github.com/microsoft/ebpf-for-windows/tree/main/docs

[47] The Unikraft Authors, "Performance - Unikraft." Accessed: Sep. 14, 2024. [Online]. Available: https://unikraft.org/docs/concepts/performance