KØBENHAVNS UNIVERSITET

Writing eBPF Programs with High-level Libraries: A Comparison of Aya and fun eBPF

Caroline Kierkegaard (`qlj566`) and Mikkel Willén (`bmq419`)
Supervisor: Ken Friis Larsen
Co-supervisor: Matilde Broløs

March 24, 2025

Abstract

This paper presents a comparative study of two high-level frameworks for developing eBPF code: Aya, implemented in Rust, and fun eBPF, implemented in Haskell. We developed an experimental setup with a Rust-based server and client to replicate previous throughput experiments originally performed with fun eBPF. Our investigation focused on evaluating performance differences, particularly the impact of filtering in kernel space versus user space, and assessing the usability of specific library features. In addition to throughput measurements, we conducted packet loss experiments to address a potential threat to the validity of both our and the original throughput experiment. The results indicate that, while the Haskell implementation shows minor differences in packet loss between filtering modes, the Rust-based system maintained practically no packet loss. Furthermore, our analysis highlights a difference in ease of use and control. Aya offers higher-level abstractions that simplify stack management and map operations, whereas fun eBPF provides more control over low-level operations. Overall, this work provides insights into the differences in developing eBPF programs with these two libraries and suggests directions for future experiments to further expand on the study.

Contents

1 Introduction

This project examines two high-level frameworks for developing eBPF code, focusing on a comparative analysis of their performance and usability. eBPF enables safe execution of programs in the kernel space, thanks to its verifier that guarantees code safety before execution. eBPF is used for a variety of tasks such as performance optimisation, debugging and system monitoring, where executing code directly in the kernel can yield improvements in efficiency.

Due to the wide variety of use cases, but inherent complexity of writing raw eBPF code directly, high-level abstraction libraries have been developed. This project focuses on two such libraries: Aya, implemented in Rust, and fun eBPF, implemented in Haskell.

Our contributions centre around establishing an experimental setup to compare the two libraries. The goal is to assess whether the performance benefits of filtering in the kernel observed with fun eBPF are reproducible with Aya, while also evaluating the developer experience. To reach this goal, we developed our own Rust-based server and client to complement the existing Haskell server and client, thereby creating an environment for comparative experiments.

The Rust server we built is compiled with an eBPF filter via an environment variable. It incorporates a pre- and post-processing functions for handling tasks such as map management and uses a hash map as its core data structure. Complementing the server, the Rust client replicates the "Midgård" behaviours of the Haskell client as well as extending the `Frey` behaviour to accept the total number of packets as a variable and introducing a new `Frigg` behaviour that sends a mix of valid and invalid packets rather than a sequential pattern.

We have also implemented a series of eBPF programs. We have implemented four socket filters; a simple socket-filter that accepts all packets, a filter for accepting valid commands, a more robust valid command filter and a parameterised filter. Besides filters, we have have also created snoopers for counting received packets and for copying packet contents into a map for user space inspection.

Our experimental framework was designed to test the throughput gains observed with fun eBPF and determine if similar improvements hold for Aya. We replicated a throughput experiment originally conducted in Haskell and extended it to include variations in the number of packets. Recognising the threat against validity posed by packet loss, we conducted an experiment to evaluate how packet loss affects throughput measurements.

Lastly, our implementation allowed us to assess the ease of use of the two libraries. We have looked at some concrete examples from both libraries, where we looked at the necessary steps needed to implement and use maps, hook points and stack memory in eBPF.

The report is organised as follows: Section 2 provides background on kernel and user space, introduces eBPF, and reviews the Aya and fun-eBPF libraries. Section 3 details our implementation, covering the Rust-based server, the suite of eBPF programs, and the client's behaviour modes. Section 4 presents our experimental results on throughput, packet loss, and ease of use, while Section 5 outlines future work opportunities. Finally, Section 6 concludes the report with a summary of our findings. Our implementation can be found on this GitHub repository: `https://github.com/mikkelwillen/POCS---ebpf-is-fun`. To make sure you are getting a working version, use commit with hash: `49b01af`.

2 Background

This section provides the background for our study. We begin by outlining the distinctions between Linux kernel space and user space, followed by an introduction to eBPF and its significance in kernel-level programming. We then present the high-level libraries for writing eBPF code, Aya and fun eBPF.

2.1 Kernel space and user space

Linux clearly distinguish user space and kernel space in memory and execution. Kernel space is strictly reserved for the operating system kernel and low-level components, running with elevated privileges [5]. In contrast, user space is where regular application processes execute, in unprivileged user mode, each in its own isolated memory address space [5]. This separation is fundamental for both stability and security. A bug in a user space program typically cannot crash the whole system, and user processes cannot directly access sensitive kernel memory or hardware. They must interact with the kernel through controlled mechanisms, primarily system calls, which enforce checks and safety [9].

System calls are the primary mechanism for user space to request services from the kernel. When a user space program needs to perform an operation that requires higher privileges, it performs a system call, which triggers a controlled transition from user mode to kernel mode [9]. The entire transition is invisible to the program aside from the fact that it incurs some overhead and the operation might block if it had to wait for e.g. I/O or other events.

While the strict separation of user and kernel space is essential for safety, it does come with performance costs. Transitioning between user mode and kernel mode e.g. entering and exiting a system call, incurs overhead compared to a normal function call. The CPU must perform a series of actions on each transition: switch privilege levels, switch stacks and save and restore registers. This overhead means that system calls are relatively expensive operations [4].

2.2 eBPF

eBPF is a Linux kernel technology that allows running sandboxed programs inside the kernel without modifying kernel source code or loading kernel modules [3]. eBPF provides a safe and efficient way to extend kernel functionality at runtime.

eBPF introduces an in-kernel virtual machine with a custom instruction set. eBPF programs are compiled into a bytecode format consisting of 64-bit RISC-like instructions, executed on a VM with ten 64-bit registers [1]. The kernel can interpret this bytecode or use a Just-In-Time (JIT) compiler to translate it into native CPU instructions for speed. In fact, eBPF instructions are typically mapped 1:1 to real CPU instructions, so JIT-compiled eBPF code runs nearly as fast as built-in kernel code. When an eBPF program is loaded, the kernel sets up a dedicated memory region for it, e.g. packet data or context structures, and only that memory is accessible to the program. This ensures the program operates within a constrained environment [1].

An eBPF program is loaded into the kernel via the `bpf()` system call. Before it can run, it must pass through a verifier and then optional JIT compilation. Upon successful load, the kernel returns a file descriptor referring to the eBPF program [2]. The program can then be attached to a specific hook point in the kernel where it will execute. eBPF defines many hook points throughout the kernel, which are triggered by events such as system calls, function entry/exit, network packet processing, tracepoints, etc [2]. Each eBPF program is a designated program type corresponding to the kind of hook or subsystem it attaches to, for example, a socket filter program runs on incoming packets, whereas a kprobe program runs on a kernel function entry [2]. If a predefined hook is not available for a desired event, developers can create custom hooks using kernel probes or user space probes to attach eBPF programs almost anywhere in kernel or user code [2].

eBPF programs cannot directly call arbitrary kernel functions. Instead, the kernel exposes a limited set of helper functions that eBPF programs may invoke safely. The available helpers depend on the program type – for instance, a tracing program has helpers to get stack traces, while an XDP program has helpers to redirect or drop packets. This design gives eBPF programs controlled access to kernel functionality without compromising stability [2] [3].

eBPF maps are a core part of the architecture that enables state and data sharing between eBPF programs and user space. A map is a generic data structure, like hash table, array, ring buffer, etc., that lives in kernel memory [2]. user space programs create maps, and eBPF programs can lookup or update elements in these maps through special instructions and helper calls. This mechanism allows eBPF programs to output data that user space can read, or to consume configuration data that user space writes. Each map type has defined semantics; for example, per-CPU maps store separate values per CPU for efficiency, ring buffer maps support streaming data to user space, etc [2].

Because eBPF programs run in kernel space, safety and security were top priorities in its design. Several layers of defence ensure that eBPF extensions cannot compromise the kernel [3]. By default, only privileged users can load eBPF programs. This prevents untrusted or unprivileged code from ever reaching the kernel [3]. Every eBPF program must go through the in-kernel verifier before it is accepted. The verifier performs static analysis on the eBPF bytecode to prove safety properties for all possible execution paths. It checks that the program will always terminate, meaning no infinite loops, does not perform out-of-bounds memory accesses or use uninitialized memory, and meets constraints on program size and complexity [3]. For example, loops are permitted only if they have a statically provable bounded iteration count. The verifier also ensures that all pointer arithmetic stays within allowed structures and that any calls are to valid targets. If any of these checks fail, the program is rejected and never loaded. These guarantees prevent eBPF programs from crashing the system or reading/writing unintended memory [3]. eBPF maps and program contexts are designed so that all memory accessed by eBPF is accounted for. Pointers obtained within eBPF come with bounds information that the verifier uses to prevent out-of-range access. All memory accesses are bounds-checked by the verifier. Moreover, eBPF programs can't access user space memory directly and vice versa except through well-defined mechanisms, so there's isolation between kernel eBPF execution and user space [3].

One of eBPF's biggest advantages is that it can inject custom logic into the kernel with minimal performance overhead. eBPF programs run in kernel space, avoiding the frequent context switches between user and kernel space that traditional solutions require. This is especially important for tasks like packet filtering or system call tracing. Eliminating user-kernel context transitions greatly reduces latency and CPU overhead [2]. For example, an eBPF program attached at a socket can decide to drop a packet right in the kernel, whereas a user space packet filter would require handing the packet to user space and then back to kernel. Running entirely in-kernel gives eBPF a fundamental speed advantage.

2.3 Aya

Aya is a library written in Rust, made for developing eBPF applications. It provides a framework for developing both the user space programs, that interact with eBPF programs, and can compile Rust code to eBPF bytecode. [6]. Unlike previous frameworks, Aya does not depend on the traditional C-based libraries, such as libbpf or BCC. Instead it invokes Linux eBPF system calls directly via Rust's libc bindings [6]. This design yields a self-contained ecosystem. In most cases, a Rust nightly toolchain is the only build dependency required to compile eBPF programs, with no need for a C compiler or clang/LLVM at runtime [10].

A core motivation for Aya is to bring Rust's strong safety guarantees and type system to eBPF development. Rust's memory safety and strict compile-time checks help prevent common bugs that might slip through in C. Although the Linux BPF verifier ensures certain memory safety properties, using Rust provides an additional layer of type safety and consistency checking [10]. Aya's design leverages this by providing Rust abstractions for eBPF program contexts, map types, and helper functions, so that many mistakes are caught at compile time rather than at runtime or during eBPF verification. For instance, each eBPF hook type, such as a socketfilter hook, expects a specific context structure. Aya encodes this in the Rust type system. Developers annotate eBPF functions with macros, like `#[socket_filter]`, and the function signature must use the corresponding context struct provided by Aya. If the wrong context type is used, the Rust compiler will emit a type error. In a concrete example, an XDP program in Aya must take an `XdpContext` argument. If a programmer mistakenly uses a `SkBuffContext` for an XDP hook, the mismatch is caught as a compile-time error [10]. This contrasts with C, where such an error might only be caught when loading or running the program, since the C compiler would not know the semantic difference [10].

Aya also introduces idiomatic Rust patterns into eBPF programs to improve safety and clarity. One notable area is error handling. In C eBPF code, the convention is to return integer error codes, since an eBPF program's return value often must indicate success or failure to the kernel. Aya allows developers to write eBPF logic using Rust's `Result` type for error propagation. The approach is to write an inner function that returns a `Result<T, E>`, and then have the actual eBPF entry-point function call the inner function and translate any error into an error code [10].

Aya's developers have designed the system to match libbpf's capabilities by implementing support for all eBPF program types and features in Rust [7]. When compiled with BPF Type Format (BTF) support and static linking, an Aya-based binary can operate on multiple Linux kernel versions without requiring re-compilation [6]. This is achieved by leveraging BTF to identify kernel data structure layouts at load time and adjust eBPF program accesses accordingly [6].

Another goal is to streamline builds and deployment. Because Aya does not require kernel headers or a kernel build at compile time, and avoids any external C toolchain, building eBPF programs is fast [6]. This fast, self-contained build process simplifies integration of eBPF into larger Rust projects and makes it possible to ship eBPF functionality as a single binary. Aya also provides opt-in asynchronous support in user space, with tokio or async-std, so developers can easily integrate eBPF event handling into asynchronous Rust applications [6].

2.4 fun-eBPF

The fun eBPF library is a high-level library for developing eBPF programs using Haskell, and is a part of ongoing research at the University of Copenhagen. The library is designed to simplify the process of developing eBPF programs into user space applications. The initial purpose was to create a use-case for eBPF, and to investigate whether moving part of a web-servers behavior to kernel space via eBPF can result in any advantageous changes in performance and resource use, such as time spend, memory used and energy used. Currently, much of the eBPF functionality needs to be implemented as eBPF bytecode.

The library includes a server instance, operating as a bag server, that can process and filter network messages. This is achieved by the server having a state map, that client application can interact with by sending UDP packets with `PUT`, `GET`, `DELETE` and `STOP` commands. The library then includes a variety of filters, that are created to test the difference in performance by moving some of the server features to kernel space.

One such filter is the `socketfilter`. This filter checks that the command in the incoming packets match one the allowed commands. If they do, the packet is send through, if not, they are discarded. Since bad messages will then be discarded in kernel space, the hope is, that we can get a performance gain compared to filtering in user space, since we then do not have to transfer the packets from kernel space to user space.

Other eBPF programs have also been developed, each with different purposes. One such program is the `snoop-filter` or just `snoop`, since the program is not a filter, but uses the `tracepoint BPF_PROG_TYPE`. This program uses an eBPF map to count the number of packets received. When a packet is received, the map is incremented, and the values of the map, can then be read from user space.

The fun eBPF library uses the ebpf-tools library [8] to connect the high-level Haskell code to low-level eBPF bytecode. A key part of this integration is quasiqouting, which allows developers to embed eBPF assembly directly into Haskell code using a syntax similar to native bytecode. This makes it possible to interpolate Haskell values directly into the instruction set, reducing some of the repetitive boilerplate and saving values, such as the numeric code for function calls, in a much more readable format.

When using quasiquoting, each block of eBPF assembly is parsed into an abstract syntax tree (AST) that represents the program. This tree is then converted into raw bytecode through the `encodeProgram` function. The developer is still responsible for managing stack offsets, what helper functions are available to the specific eBPF program and needs to ensure the resulting program complies with the eBPF verifier. The ebpf-tools library also exposes function, that use Haskell's Foreign Function Interface (FFI), to call low-level C functions. This enables the fun eBPF library to directly invoke the kernel's BPF system calls to e.g. load, attach and interact with the eBPF programs.

Our project is based on an artifact, which is a subset of the developed functionality of fun eBPF, provided by our supervisors, Ken Friis Larsen and Matilde Broløs. The handed out artifact can be found on our GitHub repository in the directory `/handout-artifact`: `https://github.com/mikkelwillen/POCS---ebpf-is-fun/tree/main/handout-artifact`.

3 Implementation

In this section, we provide an overview of the implementation of our experimental framework. We have implemented a server, a client and a variety of filters and eBPF programs using the Aya library in Rust. The design of the implementation is inspired by the Haskell code, we were provided.

3.1 The Server Library

We have implemented a server for the project, that is able to handle UDP packets. Firstly, we have made an enumerator for the valid commands.

3.1.1 Message enumerator

The `parser::Message` enumerator contains the set of valid commands, that our system can process. It defines four different commands for different operations.

```
pub enum Message {
  Get(u32),
  Pet(u32, i64),
  Delete(u32),
  Stop,
}
```

`PUT` associates the key with the value for insertion or update in the state map. `GET`, gets the value corresponding to the key in the state map. `DELETE` removes the key-value pair from the state map, and finally `STOP` signals the server to stop operating.

3.1.2 Main function

In the main function, we begin by parsing the command-line arguments, `verbose` and `capacity`, using the `Clap` library. We then load the eBPF bytecode. Following this, the function binds a UDP socket to a designated local address `127.0.0.1:12345`. After the socket is set up, we retrieve the socket filter program from the loaded eBPF bytecode and attach it to a hook in the UDP socket in the kernel. This makes the filter program run, when a packet is received on the UDP socket.

```
let prog: &mut SocketFilter = ebpf.program_mut("socket_filter")
                                      .unwrap().try_into()?;
    prog.load()?;
    prog.attach(&socket)?;
```

We then initialise an empty state map that is used to save the key-value pairs in the server.

After all this is set up, the `main` function calls the `serve` function, but before this, the `pre` function is called, and `serve` is finished, the `post` function is called.

3.1.3 Pre and post functions

Before and after the call to `serve` in the `main` function, the `pre` and `post` functions are called respectively. These functions allow us to modify the behaviour of the server, without having to change the implementation of the `server_lib`. `pre` and `post` are e.g. used to setup eBPF maps in user space. These functions could also be used for many other purposes, where we either want to interact with the eBPF program, or if we want to change server behaviour.

3.1.4 Serve function

The `serve` function handles incoming network messages. It contains the core server loop, continuously listening for UDP messages and then processing them, when they are received. Within its loop, `serve` waits for incoming data using `recv_from` on the UDP socket, and writes the data to a buffer.

```
match socket.recv_from(&mut buf) {
    Ok((num_bytes, sender)) => {
        let msg_option = parse_message(&buf[..num_bytes]);
        match msg_option {
```

We parse the received bytes into the enumerator `Message` by calling `parse_message` on the buffer. We match on the result. If it is a `STOP` command, we break out of the loop. If it is some other command, we call `process_message` on the command.

3.1.5 Parsing of messages

The `parse_message` function is responsible for interpreting raw byte streams and converting them into the commands from the `Message` enumerator. The function takes one parameter, `request` of type `[u8]`, which contains the incoming message to be parsed.

The function `parse_message` begins by scanning the input byte slice for the first non-alphabetic character. This character acts as a delimiter, that separates the command from the subsequent data. This delimiter is used to split the byte slice into the two parts for the command and the data.

```
let pos = request.iter().position(|&b| !b.is_ascii_alphabetic())
                                        .unwrap_or(request.len());
let (cmd_bytes, rest) = request.split_at(pos);
let cmd = str::from_utf8(cmd_bytes).ok()?;
```

The command is the converted into a UTF-8 string, and we then match on the command. For the `PUT` command, the function verifies that the remaining byte slice contains atleast 12 bytes - 4 bytes for the key and 8 bytes for the value. We then use `from_le_bytes` to decode the numerical key and value from little-endian, and then return a `Message::Put`. In the cases for `GET` and `DELETE`, we check that the remaining byte slice contains atleast 4 remaining bytes for the key, decode it from little-endian, and returns either a `Message::Get` or a `Message::Delete`. The `STOP` command requires no additional data, and thus we just return a `Message::Stop` immediately.

If the command is unrecognized or if the input does not meet the required byte-lengths for the command, the function returns `None`.

3.1.6 Processing of messages

The `process_message` function is responsible for updating the state map on incoming commands. The function takes four parameters. First the state map, which is a mutable reference to the state map. Then the message to be parsed, as the enum type `Message`. A boolean flag to control logging, and lastly a capacity parameter, that limits the maximum number of keys the state map can hold.

The implementation of `process_message` uses a `match` statement on the input message, to ensure all possible message types are handled.

For the PUT command, the function first checks if the state map's size is below the allowed capacity or if the key already exists in the map. If the condition is met, it uses the builtin hashmap API to either update the existing key or insert a new one with a default value of zero, which is then updated with the value provided. If the state map is full and the key does not exist, a log message is written, stating that no key can be added.

```
if state.len() < capacity || state.contains_key(&key) {
    *state.entry(key).or_insert(0) += value;
} else {
    logging(verbose, "State map is full, cannot add new key");
}
```

In the case for the GET command, the function retrieves the current value associated with the given key. The integer value is the converted into a string, and then into a byte vector. This byte-encoding is then meant to be sent back over the network. For DELETE, the function removes the key from the state map with the hashmap API. The STOP command is not supposed to be caught in this function, and is logged as an anomaly. The log is an alert for issues in message routing, since the STOP command is supposed to be managed in the server loop in the `serve` function.

The function returns an `Option<Vec<u8>>`, where only the GET operation produces a reply.

3.2 The eBPF programs

3.2.1 Filters

We have implemented a catalogue of filters to use in our experiments. More specifically socket filters, which are a type of eBPF program that attach to a socket, allowing us to intercept and analyse network packets as they enter the kernel's networking stack. By operating at this level, they provide access to the context from higher-level protocols, which can be very useful for both inspection and filtering packets. This means we can apply filtering rules based on the content of the packets. We have implemented 4 different socket filters: `simple-socket-filter`, `valid-command`, `robust-valid-command` and `parameterised-filter`.

All of the socket filters utilise the `[#socket-filter]` macro. The filters work, by returning some integer, specifying how many bytes we are letting through. If the packet is larger than the number of bytes allowed through, the packet is discarded. Returning -1 allows a packet of any length through.

`simple-socket-filter`

This filter allows all packages through by always returning -1.

`valid-command`

This filter only allows packets containing a valid command through. The valid commands are PUT, GET, DELETE and STOP. The content of each of the commands an size of each element in bytes is shown in figure 1.
The filter retrieves the length of the incoming packet. If the packet length is zero, it is immediately rejected. To perform the inspection, the function copies the first 64 bytes of the packet into a buffer. The validation process then checks for specific byte sequences at fixed positions in the packet payload. The command should be 8 bytes in as seen in figure 1, in the buffer. We then verify whether the command in the packet matches one of the predefined commands.

```
if command[8..11] == *b"PUT"{
    Ok(())
} ...
```

If any of these command sequences are found at the correct position, the packet is allowed. If a packet does not contain any of the expected commands, the function returns an error, leading to its rejection. Additionally, the program includes a simple panic handler that enters an infinite loop in case of an unexpected failure.

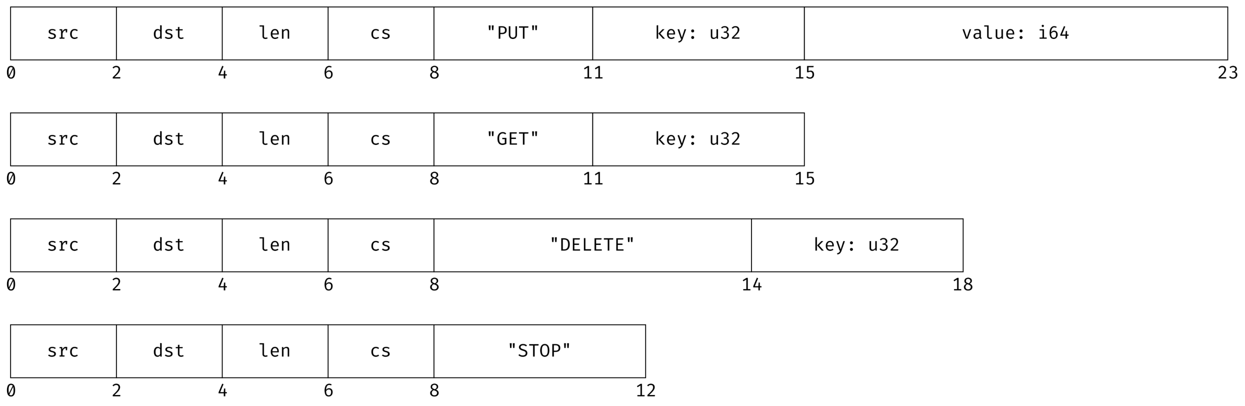

Figure 1: Content of UPD packet for each valid command

`robust-valid-command`

After creating the first version of a valid command filter, we discussed if we could make a more robust version. The first valid command filter only checks if the packet length is above 0 and if the correct ascii characters from the valid commands are in the correct places. This could lead to a series of invalid commands slipping through, even when they pass these checks e.g. the command `PUTT`.

For our more robust filter, we added a check of the length of the packet. Every UPD packet has a length field, which we can check against knowledge we have of the length of each command as seen in figure 1. E.g. the `PUT` command is 23 digits long due to the header (8) + `PUT` (3) + the u32 key (4) + the i64 value (8) = 23.

```
if length == 23 && command[8..11] == *b"PUT" {
    Ok(())
}
```

With this check there will not be room enough for the cheat command `PUTT`. However, the final `T` would be interpreted as part of the key and for now, our implementation does not handle correct types in the key and value parts of the commands.

For future work, one could create a more robust filter. For now, the protocol is written in binary, which can be interpreted in a number of ways. One could change the protocol to use ascii characters instead. This way each digit will be interpreted in a specific way and one can ensure that the keys and values are not matched e.g. letters or special characters.

`parameterised-filter`

Since Aya does not allow dynamically loading eBPF program, we have created a filter, that can change behaviour depending on a parameter set in user space. This does not allows us to dynamically load eBPF programs, but if we know the different types of behaviours we want at compile time, we can change the filters behaviour at runtime.

To achieve this, we have setup a map in the pre-function in user space, where we are able to interact with the eBPF map. Here we change a value in the map, depending on a flag set at runtime.

We use the verbose flag to change the behaviour at runtime. It would make more sense, to have a separate flag for this, but since this is just a test, we do not have to modify the `server_lib` implementation.

```
let mut array = Array::try_from(ebpf.map_mut("parameter").unwrap())?;
...
if verbose {
    println!("Setting parameter[0] to 0");
    array.set(0, 0, 0)?;
} else {
    println!("Setting parameter[0] to 1");
    array.set(0, 1, 0)?;
}
```

In the eBPF program, we simply check, if they value is 0 or not. If it is set to 0, we allow the packet through. If the value is set to anything other than 0, we do not allow the packet through.

```
 unsafe {
    match PARAMETER.get(0) {
        Some(0) => Ok(()),
        _ => Err(()),
    }
} ...
```

This behaviour in itself is not particularly interesting, but could be used in combination with some user space checks, where we can temporarily stop receiving packets, if we want to.

3.2.2 Snooper for counting

We have also implemented an eBPF program, that does not filter, but instead snoops on the packets and thus can provide valuable information.

Specifically, we have implemented a snooper that can count the number of packets received - both valid and invalid. The program uses a `PerCpuArray` eBPF map to store the count in. This allows us to easily access the array, by getting a pointer to the index of the array, and increment it.

```
#[map(name = "counter")]
static mut PACKET_COUNTER: PerCpuArray<u32> = PerCpuArray::with_max_entries
   (CPU_CORES, 0);
...
unsafe {
    let counter = PACKET_COUNTER.get_ptr_mut(0).ok_or(())?;
    *counter += 1;
}
```

A `perCpuArray` defines a BPF map where each index holds a separate copy of data for every CPU thread. This design minimizes lock contention by allowing each CPU to update its own value independently without interfering with others. [11] As described in the section about the implementation of the server, the servers has a pre and post function, which comes in very handy in this case. The post-function reads the count from the map and outputs it to the user.

```
let counterArray: PerCpuValues<u32> = array.get(&0, 0)?;
let mut total: u32 = 0;
for i in 0..nr_cpus().expect("failed to get nr_cpus") {
    total += counterArray[i];
}
```

This is not a very advanced program, but it lays the ground for future work. One could use such a counter for dynamic filters such as e.g. we can at most get a certain amount of puts or a request should only get through if its requested key exists.

3.2.3 Snooper for printing packets

We have created another program, which snoops as well and prints the content of a packet to the user. It works by copying the content of the packet to a `PerCPUArray` map, which is then retrieved and printed in user space with the servers post-function. In the kernel space, a data structure named `Buf` is defined to hold up to 500 bytes of packet data, and a `perCPUarray` map is created to store this data. When a packet arrives, the socket filter function calls a helper routine that first determines the packet length and then safely reads up to 500 bytes into a temporary buffer. This buffer is then copied into the first entry of the per-CPU array map, ensuring that only the available bytes are stored without violating the verifier's constraints.

```rust
#[repr(C)]
#[derive(Clone, Copy)]
pub struct Buf {
    pub buf: [u8; 500],  // Buffer to store packet data
}
...
let mut temp_buf: [u8; 500] = [0; 500]; // Use a temporary buffer that fits
     the verifier's constraints

// Load only available bytes, up to 500 bytes max
ctx.load_bytes(0, &mut temp_buf[..read_len as usize])
    .map_err(|_| ())?;

// Retrieve the eBPF map entry
let buf = unsafe {
    let ptr = BUF.get_ptr_mut(0).ok_or(())?;
    &mut *ptr
};

// Copy only the loaded portion into the eBPF map
buf.buf[..read_len as usize].copy_from_slice(&temp_buf[..read_len as usize
   ]);
```

After the eBPF program processes the packets, a post function retrieves the map and iterates over the per-CPU values and prints the contents of the buffers.

This design was implemented for two primary reasons. First, printing data directly from the eBPF program allowed us to verify the correctness of the packet filtering logic and to observe the contents of the packets, which was invaluable during debugging and testing. Second, accessing the data more easily in user space might be important for future experiments where we want to use the data to e.g. which sort of invalid (and maybe malicious) packets a client is sending.

3.3 The Client

The client is implemented in Rust as a testing tool for evaluating the server and its filtering mechanisms. It sends packets via UDP and supports multiple behaviour modes that can be set with command-line arguments. The implementation is divided into two components: a library and a main function.

The main file defines the execution logic of the client and manages the configurable behaviours through command-line arguments using the `clap` crate. We have implemented four parameters, which can be set through the command-line:

- `--verbose (-v)`: Enables logging output.
- `--behaviour (-b)`: Determines the client's behaviour mode.
- `--percent (-p)`: Specifies the percentage of invalid packets for the `Frey` and `Frigg` behaviours.
- `--number_of_packets (-n)`: Specifies the number of packets for the `Frey` and `Frigg` behaviours.

The client initializes a UDP socket and selects a behaviour mode based on the user's input. The behaviour modes correspond to different testing strategies, such as repeatedly sending `PUT` commands, when the default mode `thor` is chosen.

The library file is responsible for encoding and decoding messages exchanged between the client and the server. It provides functions to serialize different message types into byte arrays before transmission over UDP. The encoding scheme ensures that requests such as `PUT`, `GET`, and `DELETE` are correctly formatted for interpretation by the server. Additionally, the library includes a logging utility that enables verbose output, when debugging is enabled in the command-line.

3.3.1 Client Behaviour Modes

The client supports different behaviour modes, each simulating a distinct pattern of interactions with the server:

- `Thor`: Sends multiple `PUT` commands followed by a `STOP` command.
- `Odin`: Alternates between `PUT` and `GET` commands before stopping.
- `Loki`: Introduces invalid messages (`BAD`) alongside `PUT` and `GET` commands.
- `Njord`: Sends `PUT`, `GET`, and multiple `DELETE` commands in a structured sequence.
- `Sylvie`: Sends 10 million `PUT` requests.
- `Sif`: Focuses on key-specific `GET` and `PUT` requests.
- `Frey`: Implements a flexible ratio between valid and invalid messages and sends a flexible number of packets
- `Frigg`: Similar behaviour to Frey, but sends a mix of `BAD` and `GOOD` packets, as opposed to first all `GOOD` then all `BAD` packets.

`Thor`, `Odin`, `Loki`, `Njord`, `Sylvie` and `Sif` are all behaviours implemented in the provided Haskell artifact. These were reimplemented in the Rust client, in order to ensure that the servers worked similarly with similar behaviours.

The `Frey` and the `Frigg` behaviours is where it gets interesting for our experiments. These two behaviours makes it possible for us to perform experiments, testing how the valid/invalid ratio affects the performance of the server and the filters.

Both the `Frey` and `Frigg` introduces a command-line argument for setting the percentage of invalid messages as well as one for setting the number of packets. Where `Frey` first sends all it's valid packets and then all its invalid packets, `Frigg` mixes these. With `Frigg` the aim is to ensure that in the case, we lose the last packets, e.g. if the buffer fills up, we don't end up with a skewed amount of valid/invalid packets.

A `Frey` behaviour, taken only the invalid/valid ratio as argument, already existed in the Haskell artifact. To match our implementation in Rust, we have implemented `Frigg` and a `Frey2`, which also take the number of packets as input, in Haskell.

4 Experiments

In this section, we describe the experiments conducted to compare the performance and usability of Aya and fun eBPF. We aim to determine whether the throughput gains observed with fun eBPF can be reproduced with Aya, while also identifying potential threats to validity. Using our implemented set-up in Rust and existing Haskell set-up, we designed a series of experiments that replicate throughput measurements, evaluate the impact of packet loss, and assess ease-of-use factors. These experiments provide us with an approach to understanding both the performance and the developer experience associated with each library.

Our experiments where run on a machine equipped with a 8-core AMD Ryzen 7 5800U with 16 MiB L3 cache and with 16 GiB RAM running nixos.

4.1 Throughput

We aim to verify the throughput gains observed with fun eBPF and determine if similar improvements hold for Aya. Our framework replicates a throughput experiment originally performed with fun eBPF in Haskell.

The original hypothesis from fun eBPF states: *The hypothesis is that we can improve performance by moving parsing of invalid messages into kernel space. Here, performance is time and memory, but could also be energy consumption.*
This hypothesis was verified with an experiment performed in a set-up written in Haskell. In this project, we aim to verify it in Rust focusing on Wall Clock Time as our performance measure, thereby providing a starting point for an assessment of the relative efficiencies of these libraries.

4.1.1 Assumptions and hypotheses

We suspect that moving the filtering from user space to kernel space, will improve the time it takes to receive packets in our server set-up, especially when a large percentage of invalid packets are send. This is due to the suspicion that it takes time to transfer the received packets back from kernel space to user space. When we filter invalid packets in the kernel space, we will no longer spend time transferring the invalid packets.

With these assumptions in mind, these are the variables we can vary upon; the number of packets, the percentage of invalid packets and the binary option of either filtering in kernel space or in user space.

We formulate three hypotheses based on our assumptions:

1. We expect an increase in the number of packets will lead to an increase in the wall-clock time it takes to receive the packets.

2. We expect the work of the filters to have negligible impact on the total time spend.

3. We expect an increase in the percentage of invalid packets will lead to the wall-clock time spend receiving packets, while filtering in kernel space, being less than or equal to the wall-clock time spend receiving packets, when filtering in user space.

The first and second hypotheses will be performed in order to establish a baseline and test that the time receiving valid packets do not differ from when we filter in kernel space to when we filter in user space. The third hypothesis is the one we actually want to test, in order to verify our assumptions.

4.1.2 Results

First, we performed an experiment to verify the first hypothesis. We varied the number of packets send, using both a server that filters the packets in kernel space and one that filters in user space. We fixed the percentage of invalid packets to 1%, as we are not testing the effect of the filtering with this experiment. Specifically, in this experiment set-up the number of packets is a number of `PUT` commands minus the 1% of invalid packets which are `BAD` commands. These are followed by a `GET` and a `STOP` command. We used the `Frigg` behaviour to do this. The experiment was run 2 times and the measurements shown are the average.

Number of packets	100	1000	10000	100000	1000000	10000000
User space (`simple-socket-filter`)	2,000	2,000	2,010	2,055	2,500	7,955
Kernel space (`valid-command`)	2,005	2,000	2,010	2,055	2,575	8,140

Table 1: Wall clock time in seconds for filtering in user space and kernel space with 1% invalid packets

The results in table 1 show an increase in the time spend receiving packets, when increasing the number of packets, thus verifying our first hypothesis.

We calculate the aggregate percentage difference in time spend

$$\sum_n \frac{|t_u - t_k|}{\frac{t_u+t_k}{2}} \cdot 100 = 0,9174\%$$

where n is the number of packet, t_u is the Wall Clock Time of the user space filter and t_k is the Wall Clock Time of the kernel space filter. We see that it is below 1%, and thus are sufficiently close together, thus verifying our second hypothesis.

Then we performed an experiment to test our third hypothesis. We fixed the number of packets to 100,000, as we have just established that number of packets does not affect the filtering. We varied the percentage of invalid packets and performed the same tests on both the server that filters in kernel space and user space. Again, the experiment set-up is a client with `Frigg` behaviour sending a specified number of `PUT` commands minus the set percentage of invalid packets which are `BAD` commands. The experiment was run 2 times and the measurements shown are the average.

Percentage of invalid packets	1	25	50	75	99
User space (`simple-socket-filter`)	2,055	2,055	2,050	2,055	2,060
Kernel space (`valid-command`)	2,055	2,055	2,055	2,050	2,060

Table 2: Wall clock time in seconds for filtering in user space and kernel space sending 100,000 packets

As seen in table 2, the results does not show the expected tendency. We expected that when increasing the percentage of invalid packets, it would show that it takes less time send and receive packets, when we filter in kernel space. We calculate the aggregate percentage difference

$$\sum_p \frac{|t_u - t_k|}{\frac{t_u+t_k}{2}} \cdot 100 = 0,0974\%,$$

where p is the percentage of invalid packets, t_u is the Wall Clock Time of the user space filter and t_k is the Wall Clock Time of the kernel space filter. This shows, that it takes approximately the same time. The small differences can be described with the variations in run time averaged over the 2 run times.

We have also performed the second experiment for a variation of number of packets in order to show continuity. We performed the experiment for all combinations of number of packets in the list [100, 1000, 10000, 100000, 1000000, 100000000] and percentages of invalid packets in the list [1, 25, 50, 75, 99]. We also generated graphs in order to show the development over the increase in number of packets. The results can be found in the appendix A and B.

4.1.3 Threats against validity

As our experiment conclude, we have not verified our third and main hypothesis. However, this could be due to a variety of threats against the validity.

Firstly, a threat could be rate limiting. We have not taken any measures to assure a certain rate of which our packets are send. This means that we could flood the server before it has time to process all the packages. To test if this is a valid threat, one could perform an experiment where we control the sending rate of the packets from the client and look for a turning point in the rate, where the throughput is maximised. To mitigate this threat, we could then use this sending rate.

Another threat is the size differences of the valid and invalid packets. In our experiment set-up the valid commands are `PUT` commands, which are 23 bytes long. The invalid commands are `BAD` commands, which are 19 bytes long. This might cause a difference in how to packets are handled by the server and how many of them it can handle at a time. To mitigate this threat for this experiment specifically, we could make the packets equal length. However, in many real-life appliances, one cannot necessarily control the length of the packets, so it would be relevant to make an experiment to assess if packet size differences is a valid threat. We could for example set up an experiment, where we test varying invalid packet sizes.

Our third hypothesis are based on the assumption, that the time spend filtering in the kernel is negligible compared to the time spend transferring the packets from the kernel to user space. With our second hypothesis and the corresponding experiment, we tried to eliminate this as a threat against validity. However, this experiment does not take into account, that there are still 1% invalid packets. This means, we can't be sure on the actual percentage difference. To eliminate this threat, we could perform an experiment, where there are no invalid packets, thus giving us a more accurate percentage difference in the time spend for the two server versions.

Another threat to the validity of our experiment is the fact that we are running the test on a machine, which are also running background processes. Due to shared resources, we cannot be sure, that our measurements are not affected by this.

Even though we run all tests on the same machine, the wall clock time can could vary due to OS scheduling, context switches, or other system activity.

Another significant threat is also the number of times we have run the experiment. This experiment was only run 2 times and this might bring an uncertainty to the results. Ideally, we should run the experiments far many times and average over the results. This might also lessen the threat of OS scheduling and shared resources as these will be distributed over the many runs.

The final threat, which we have identified, is the threat of packet loss. If we lose more packets in kernel space compared to user space, we cannot be sure, that the time saved is due to higher throughput, or due to losing packets, and thereby not spending time processing them. The same is true, if we lose more packet in user space compared to kernel space, which could lead to our experiment showing a similar Wall Clock Time between user space and kernel space in our experiment. We have conducted an experiment to asses if this is a valid threat against the validity of the throughput experiment.

4.2 Packet loss

Recognising that packet loss poses a threat to the validity of the throughput experiments, we wanted to conduct an experiment testing this. We also wanted to see if packet loss posed a threat to the throughput experiment for fun eBPF. Therefore we performed two experiments with the same set-up for both the Haskell server and the Rust server. The two experiments were performed independently and we are not comparing the results of the two servers.

4.2.1 Assumptions and hypotheses

Sending multiple packets to a server can lead to packet loss. When packets arrive at a rate that exceeds the server's processing or buffering capacity, some packets might be dropped. This is especially relevant when we are working with UDP, which do not guarantee delivery or provide built-in retransmission mechanisms. We assume that, when filtering in the kernel, we do not have to transfer the invalid packets from kernel space to user space, reducing packet loss, since we can move on to the next packet faster.

The variables we can vary upon is; the number of packets, the percentage of invalid packets and the binary option of either filtering in kernel space or in user space.

We have formulated these two hypotheses:

1. We expect an increase in the number of packets send to lead to an increase in the percentage of packets lost.
2. We expect that as the percentage of invalid packets increases, the packet loss when filtering in kernel space will be less than or equal to the packet loss when filtering in user space.

4.2.2 Results

First we performed an experiment to test the first hypothesis, in order to see if the increase in packets would lead to an increase in packet loss. We varied the number of packets send from 100 to 10,000,000 and fixed the percentage of invalid messages to 1%. We used the `Frigg` behaviour to send the packets. We performed this experiment independently for both the Haskell and the Rust server 10 times, and found the average.

Number of packets	100	1000	10000	100000	1000000	10000000
Filtering in user space	0,00%	0,04%	0,00%	0,00%	0,00%	0,00%
Filtering in kernel space	0,00%	0,04%	0,00%	0,00%	0,05%	0,01%

Table 3: Packet loss in the Rust server with 1% percentage of invalid packets

Number of packets	100	1000	10000	100000	1000000	10000000
Filtering in user space	0,00%	0,00%	0,00%	1,39%	1,76%	2,90%
Filtering in kernel space	0,00%	0,00%	0,00%	1,46%	1,86%	3,07%

Table 4: Packet loss in the Haskell server with 1% percentage of invalid packets

In table 4 we can see that for the Haskell server, the number of packets lost is increasing along with increasing the number of packets send. This is what we expected. However, with the Rust server in table 3, we are basically not losing any packets, even though we increase the number of send packets. As expected, filtering in respectively user and kernel space, does not make much of a difference when we are sending 99% valid packets as these will not be filtered.

We then performed another experiment to test the second hypothesis. Here we fixed the number of packets to 100,000 and varied the percentage of invalid packets from 1% to 99%. Again, this experiment was performed independently for the Haskell and the Rust server using the `Frigg` behaviour and averaging over the results from 10 runs.

Percentage of invalid packets	1	25	50	75	99
Filtering in user space	0,00%	0,04%	0,00%	0,00%	0,00%
Filtering in kernel space	0,00%	0,04%	0,00%	0,00%	0,00%

Table 5: Packet loss in the Rust server when sending 100,000 packets

Looking at table 5, the results are showing what we expected with the Rust server. We are barely losing any packets, both in user space and in kernel space, thus showing our second hypothesis. From this, it appears we can eliminate packet loss as a threat against validity for the throughput experiment of the Rust server.

Percentage of invalid packets	1	25	50	75	99
Filtering in user space	1,39%	1,32%	0,42%	0,00%	0,00%
Filtering in kernel space	1,46%	1,42%	0,30%	0,10%	0,00%

Table 6: Packet loss in the Haskell server when sending 100,000 packets

The results are not showing what we expected with the Haskell server in table 6. We are losing slightly more packets in kernel space compared to user space, thus disproving our second hypothesis for the Haskell server.

We would argue, however, that since the difference in percent points is so small, it alone wont affect the throughput experiment as a whole. This is because the throughput gain observed in the Haskell throughput experiment, is magnitudes greater than the difference in this experiment. We would therefore argue, that a small amount of the throughput gain, can be attributed to packet loss, but packet loss does not have a big enough impact, to discredit the throughput experiment. We cannot ultimately reject the threat, since if it turns out other threats against validity also contributes to the performance gain, this result, in combination with those, would then pose a threat against validity.

We also performed the experiment for all shown combinations of number of packets and percentages of invalid packets. The results can be found in the appendix C.

4.2.3 Threats against validity

We identified a variety of threats against the validity of the experiment.

Firstly, rate limiting is also a concern in this experiment. Without a controlled sending rate, we might be overwhelming the system's capacity to process incoming packets, leading to packet loss that is more reflective of a saturated network rather than the performance of our filter. To test if this threat is valid, one could perform an experiment where we carefully control the packet transmission rate and identify a threshold beyond which packet loss begins to increase significantly.

Another threat arises from the overhead introduced by the work of the filter itself. Although our filter is implemented in kernel space with eBPF, its processing overhead might affect the packet handling. To validate this, we could measure the processing time attributed solely to the filtering operations and compare it with the overall packet handling time.

As in the previous experiment, shared resources on the machine might still pose a threat. Since the test environment is subject to background processes, this could lead to additional packet loss that is not directly related to the filtering mechanism. Isolating the experiment or reducing background interference could help assess the impact of this threat. OS scheduling is also again a potential threat.

Again, a concern is the limited number of experimental runs. This experiment was conducted 10 times, which may introduce uncertainty in the results. Ideally, we should perform the experiments many more times and average the outcomes. Increasing the number of runs could also mitigate the effects of OS scheduling and shared resources, as these factors would be more evenly distributed.

Many of the same threats applies to both the throughput and the packet loss experiment. Both experiments share nearly identical setups because the packet loss experiment was specifically designed to assess its potential threat to the validity of the throughput experiment. As a result, many of the same risks are present in both cases.

4.3 Ease of use

We want to examine the ease of use the libraries when developing eBPF programs, especially the functionalities available and how ease they are to use. We discovered some notable differences in the ease of use of Aya compared to fun eBPF during implementation of our server, client and particularly the eBPF programs.

The aim is to compare the experience of developing eBPF programs and the corresponding user space set-up with the two libraries. The points below are inherently subjective.

4.3.1 Maps

Aya supports all eBPF map types out of the box. It provides high-level Rust APIs to define and use maps both in eBPF programs and in user space. Developers can declare maps in Rust with type-safe interfaces. Aya then also allows developers to access these maps in both user space and kernel space with convenient methods, like `Get` or `Set`. It is worth noting, that whenever we want to access the map in kernel space, we need to wrap it in an unsafe code block. This is because the map is mutable static variable, specifically a pointer to the map, meaning we have to perform a raw memory access. This bypasses Rust's normal safety guarantees.

fun eBPFs support is more low-level. The ebpf-tools library mainly provides bytecode assembly utilities. It allows creating maps by calling `bpf_map_create` via Foreign Function Interface (FFI), but there is no rich API for different map types comparable to Aya's. The developer must manage map key-value pairs manually in eBPF bytecode, and map features, such as ring buffer or perf event arrays, are not abstracted away by a Haskell library. This makes map usage less intuitive and more elaborate to use in Haskell compared to Aya.

To compare between the two libraries, we will now do a comparison of a program implemented in both Aya and fun eBPF. We will specifically look at the snooper program. In both versions, this program has been implemented with a map, though the Aya version uses a `PerCpuArray` where as the fun eBPF version uses a `HashMap`. Regardless of the map type, the ease of use is determined by how the steps to use the maps are managed, so the fact that we use different map types should not have an impact.

In Aya, maps are created with macros. In the snooper program, it looks like this:

```rust
#[map(name = "counter")]
static mut PACKET_COUNTER: PerCpuArray<u32> = PerCpuArray::with_max_entries
    (1, 0);
```

These two lines does several things. It declares a static mutable map, or a pointer, named `PARAMETER` in the eBPF program. It declares the type of map as an `Array<u32>`, with a maximum number of entries initialized to zero. Note, the zero argument to `with_max_entries` is not the initial value, it is a flag. Aya's build process then automatically generates the corresponding eBPF bytecode for the maps creation. Since the eBPF program are executed on the CPU thread, that handled the event, that triggered the program, and since the OS could schedule any thread to process the UDP packet, we need a separate index in the map for each thread, to avoid race conditions. This is why we chose a `PerCpuArray`. We can then access the map in kernel space, by getting a pointer to a specific index in the CPU's array with

```rust
unsafe {
    let counter = PACKET_COUNTER.get_ptr_mut(0).ok_or(())?;
    *counter += 1;
}
```

which is then incremented. Aya abstracts away the need to get a thread identifier, and automatically indexes in its respective array. Accessing the map in user space is as straightforward. First we need to get a pointer to the array. In Aya, this is done by calling `map_mut` on the eBPF program and giving it the a string with the name of the map in kernel space.

```rust
let array = PerCpuArray::try_from(ebpf.map_mut("counter").unwrap())?;
```

When we want to read from the map, we can just use the `Get` method to get a specific index from each of the `CpuArrays` at the same time.

```
let counterArray: PerCpuValues<u32> = array.get(&0, 0)?;
```

We can then iterate over this new array with a for loop.

In fun eBPF, Maps are created and accessed using functions that wrap low-level BPF syscalls. In the snooper program, the map is created with

```
bpfMap <- newMap 16 BPFMapTypeHash
```

Here, the function `newMap` takes a maximum number of entries and a map type as parameters. If we then want to write to the map, we have to do several things. The ebpf-tools library exposes several function to aid with this. We first have to get the file descriptor for the map, which is done with the following function from ebpf-tools

```
let map_fd = mapToRawFd bpfMap
```

We then need to get the numeric identifiers for the helper functions we need, to be able to read and write to the map.

```
map_lookup_elem = fromIntegral c_BPF_FUNC_map_lookup_elem
map_update_elem = fromIntegral c_BPF_FUNC_map_update_elem
```

Again since eBPF programs can be executed on any CPU thread, we need a separate index for each thread, thus we need some way to distinguish between the threads. This is done with

```
get_pid = fromIntegral c_BPF_FUNC_get_current_pid_tgid
```

These variables can then be used in the eBPF bytecode using a quasiquoter, and we are ready for the first step of updating an element in the map, which is getting the current value of the map.

```
mov r2, r10
add r2, -4
stxw [r2], r3
lmfd r1, $map_fd
call $map_lookup_elem
```

To get the value, we first have to set up the stack, so the key is stored at the frame pointer -4. The key in this case is the process ID of the Haskell program. The map's file descriptor is moved into $r1, and then we call the helper function to look up an element in the map. This function returns a pointer to the current map element in $r0.

The next step is to increment the value.

```
mov r8, r0
jeq r8, 0, +4
ldxdw r6, [r8]
add r6, 1
mov r8, r6
jmp +1
mov r8, 1
```

The first thing here is to check whether the lookup returned a valid pointer. If not, it means that the map value has not been initialized yet, so we just move 1 into $r8, which then contains the new value to be stored in the map. If it is a valid pointer, the value the at the pointer is loaded into a register, incremented and then again saved in $r8.

We are now ready to update the value in the map. First thing is to ready the key.

```
;; pid is key
call $get_pid
mov32 r3, r0
mov r2, r10
add r2, -4
stxw [r2], r3
```

This is done, by first obtaining the key, by calling the get process ID function. This key is then stored at the same stack location reserved earlier.

Now it is time to ready the value.

```
mov r3, r10
add r3, -16
mov r4, r8
stxdw [r3], r4
```

The value was as mentioned stored in $r8. This value is now loaded onto the stack at the location of the frame pointer -16.

Finally, we can call the update function.

```
mov r4, $flag
lmfd r1, $map_fd
call $map_update_elem
```

The program first sets the flag in $r4, then loads the file descriptor for the map again. Then it calls the helper function to update the map, which uses the key and new value, stored at the specific offsets in the stack, to update the corresponding map entry.

While both approaches generate valid eBPF bytecode, we would argue that Aya is far easier to use, due to high-level methods to interact with maps. The fun eBPF approach, requires much more manual effort and has a much higher risk of bugs, which also this snooper program is the victim of.

4.3.2 Hook points

Aya supports a wide range of eBPF program types and makes attaching these programs to hook points in the kernel straightforward. Each hook type is represented by a Rust type, and Aya provides macros, like `#[tracepoint]` and `#[socketfilter]`, to hook specific functions to those contexts. The user space side of Aya also offers methods to load and attach programs, abstracting away the complexity of BPF attach syscalls. Attaching an eBPF program often only requires calling a load method and perhaps providing an attachment target, like a specific UDP socket for socket filters, or a network interface for XDP programs.

The Haskell library's support for hook points is limited compared to Aya. It does not have built-in abstractions for the various eBPF program types. Instead it exposes a generic interface to load a program with a given program type. This means all eBPF program types are unstable, but the library doesn't provide abstractions to attach bytecode to hooks, so the developer must specify this themselves, by using the correct Linux API program type, which then only tells the kernel which hook context to associate with the bytecode. Therefore it is up to the user to adhere to the expected context of the hook, like what helpers are allowed, since the library wont warn you, if use these incorrectly.

We will again look at an example. Below are how to specify and attach an eBPF socket filter.

In Aya, you annotate the eBPF function with a macro.

```
#[socket_filter]
pub fn socket_filter(ctx: SkBuffContext) -> i64 {
```

This first of all tells the Aya compiler what type of program it is, but it also specifies the starting point of the eBPF program. Thus, when a hook is invoked on the attached UDP socket, the eBPF program will start execution, from the specified function.

In the user space program, the code is loaded by specifying the eBPF byte code file.

```
let mut ebpf = aya::Ebpf::load(aya::include_bytes_aligned!(concat!(
    env!("OUT_DIR"),
    "/socket-filter"
)))?;
```

This implies the eBPF program is compiled in advance. The program is then loaded by name.

```
let prog: &mut SocketFilter = ebpf.program_mut("socket_filter").unwrap().
    try_into()?;
prog.load()?;
```

and then attached to an UDP socket, by simply specifying the socket.

```
let socket = std::net::UdpSocket::bind("127.0.0.1:12345")?;
prog.attach(&socket)?;
```

The eBPF program will now be run, whenever a packet is received on the socket.

In fun eBPF, the programmer manually writes the socket filter's bydecode, and there is no way to annotate, what type of eBPF program it is, in the eBPF bytecode. This is only specified, when the program is loaded. The program is loaded by wrapping the eBPF instruction into a `BPFProgLoad` record.

```
let prog = BPFProgLoad { progType = c_BPF_PROG_TYPE_SOCKET_FILTER
                        , insns = encodeProgram insns
                        , license = s_Dual_MIT_GPL
                        , logLevel = Just 2
                        , kernVersion = Nothing
                        }
```

This specifies the program type, in this case `c_BPF_PROG_TYPE_SOCKET_FILTER`, meaning it will be loaded as a socket filter. The instructions are then encoded into bytecode with `encodeProgram` function. The program is then loaded into the kernel.

```
fres <- bpfProgLoad prog
```

If this load is successful, the resulting program file descriptor is attached to the target socket `sock`.

```
case fres of
    Right (Fd progFd) ->
      setSocketOption sock (SockOpt c_SOL_SOCKET c_SO_ATTACH_BPF) (
    fromIntegral progFd )
```

In both Aya and fun eBPF the libraries expose higher-level functions to deal with loading and attaching the eBPF programs, making them both easy to work with, though the developer needs to specify a few extra things in fun eBPF, like the kernel version, and the option of BPF or eBPF, in this case eBPF with `c_SO_ATTACH_BPF`.

4.3.3 Stack memory

In eBPF, programs are limited to a 512 byte stack, or 256 bytes in some cases, like tail calls. Allocating memory on the heap is not allowed. Aya handles these constraints by not allowing the standard library and features that allocate to the heap, like `alloc` or `collections`, are not allowed either. Though, when you write an eBPF program in Aya with these restrictions, local variables and temporary storage are compiled to use the stack automatically. The compiler helps ensuring that you do not exceed the stack limit. For instance, if you declare a local array, the compiler calculates the size of it, and if the total stack usage exceeds the stack limit, you get a compile-time error.

In fun eBPF, if you want to use the stack, you must manually write instructions that use the frame pointer with a negative offset. Because you are directly controlling the offsets, you must carefully track how much space is used. If you use too many bytes or overlap different variables, the kernel's verifier will reject the program. There is no automatic allocation or safety net, so the developer must manually compute every offset and guarantee that the total does not exceed 512 bytes.

We will now look at two different examples of the use of the stack.

The first example we will look at, is using the stack to allocate a local array in eBPF. In Aya, we can declare a local fixed-size array with the standard rust syntax.

```rust
let array: [u32; 4] = [10, 20, 30, 40];
```

Accessing the array is also done with the standard rust syntax. The Aya compiler takes care of the allocation of the array on the eBPF stack and computes the correct offsets automatically when indexing in the array.

In Haskell, we manually have to reserve a contiguous block of memory on the eBPF stack by subtracting from the frame pointer. We either have to remember the offset to index of the start of the array, or we can use the quasi-quoting functionality to save the offset to the start of the array.

```haskell
let array_offset = -16
    insns = [ebpf|
      mov r2, r10
      add r2, $array_offset

      mov32 r0, 10
      stxw [r2], r0

      mov32 r0, 20
      add r2, 4
      stxw [r2], r0

      mov32 r0, 30
      add r2, 4
      stxw [r2], r0

      mov32 r0, 40
      add r2, 4
      stxw [r2], r0
```

If we then want to index in the array, we again have to load the frame pointer and the array offset, and then we also have to add the offset to the specific index we want (8 in this example, to get the third index).

```
mov r2, r10
add r2, $array_offset

add r2, 8
ldxw r3, [r2]
```

This approach requires that the offsets are calculated and managed by the developer.

Another example, is when we want to use the maps from earlier. In the Aya case, we again don't need to think about the stack, we just use the high-level methods to get, set and so on, with the keys and values we want.

In the fun eBPF case, we have manually reserve a 4-byte slot on the stack to store the key and an 8-byte slot to store the value, which again requires us to manage and calculate the offsets.

Aya's abstractions make managing the eBPF stack almost invisible to the developer, we only need to make sure, we don't reach the stack limit, while the fun eBPF approach requires us to manage every step of these operations with the correct offsets.

5 Future Work

This project establishes an experimental set-up for comparing Aya and fun eBPF. Initial experiments were conducted, but the set-up provides the possibility for many more. In this section, we will discuss some ideas for future experiments one could conduct and ideas for how one would extend the set-up to do so.

5.1 Testing threats against validity of our experiments

As discussed in the experiments section, we identified many threats against the validity of our experiments. As future work, we could set up experiments for each of these threats in order to asses their validity and ideally mitigate them.

5.2 Throughput comparison of Aya and fun eBPF

The implementation of the two servers vary on a number of points. The main difference is that the servers are build upon two different data structures for storing the state of the server. Therefore, in order to make a useful comparison of the throughput, we would have to align the implementation of the two servers to make sure we are examining the difference in the libraries and not in the server set-up.

5.3 Dynamic filters

In this project we have taken the first steps in order to implement dynamic filters. As described in the implementation section, we have created a filter for counting packets using a map. As our experiments of ease of use showed, the use of maps in Aya is very straight forward and accessible to the user. Maps are the basis for many, much more interesting dynamic filters, which could be implemented as future work.

5.4 UDP vs. TCP

The server implemented for this project is build around UPD, which means we designed it to operate in a connectionless mode, where data is sent in individual packets without establishing a dedicated connection. This results in lower latency and faster communication, but does not guarantee packet delivery or order. In contrast, using TCP would have required a three-way handshake to establish a reliable connection. TCP ensures that all data is received correctly and in order through error-checking, retransmission, and flow control, but these features will then add overhead and increase latency.

For this project we chose UPD, mainly because the server implemented for the fun eBPF is an UDP server and the initial aim was to make the servers comparable. For future work it would be interesting to perform our experiments on a TCP server as well, as this would likely change the results of our throughput and packet loss experiments.

5.5 Socketfilter vs XDP

For this project we have focused on socket filters, but these are not the only type of eBPF filters - another interesting type is eXpress Data Path (XDP) filters. Socket filters and XDP are both eBPF programs used for processing network packets, but they operate at different stages of the network stack. As mentioned earlier, with socket filters, we attach our eBPF program to a socket, allowing us to inspect or filter packets once they've already entered the kernel's networking stack. In contrast, XDP runs at the earliest possible stage - directly on the network interface card. This enables us to drop, modify, or redirect packets before they enter the broader network stack.

5.6 Ease of use of Dynamically loading eBPF programs

Aya provides a robust API for loading and attaching pre-compiled eBPF programs. However, unlike in Haskell, where you can generate code at runtime using functions that take an input argument, Aya does not offer similar features. In Haskell, this also means, that you can write two different filters in separate functions and choose at runtime which filter to run.

A possibility for future work would be to explore whether is is possible in Aya, to change the active filter at runtime, by providing an alternative path to the eBPF bytecode. We would also like to explore, if it is possible to have match case/if statement, that loads and attaches a eBPF bytecode program depending on runtime inputs. This would e.g. allow us to move the parameterisation of the filters from kernel space to user space. If this is possible we could potentially cut down on a lot of instructions in parameterised eBPF programs, since each behaviour would be put in different eBPF programs, and loaded in user space.

6 Conclusion

This study compared two libraries for developing eBPF bytecode interacting with eBPF programs in user space - Aya, implemented in Rust, and fun eBPF, implemented in Haskell. We replicated a throughput experiment originally performed with fun eBPF. We performed a packet loss experiment, because we identified it as a threat to the validity of both throughput experiments. This experiment was performed for both the Rust and Haskell set-up independently, to assess whether packet loss would have an impact on the throughput experiment results. We also compared specific features available in both Aya and fun eBPF, and assessed the ease of use of these features when developing eBPF programs.

In order to perform these experiments, we implemented a server and client in Rust and used Aya to develop eBPF programs, both filters and snoopers. The server, client and eBPF programs was inspired by the handed-out Haskell implementation.

When we replicated the original throughput experiment with Aya, we expected the same results, which showed an increase in throughput, when filtering in kernel space compared to user space. However, the results of our experiment did not show the same results as the original experiment. Our results showed a similar throughput for filtering in kernel space compared to user space. We identified a number of threats to the validity of the experiment.

One of the identified threats was packet loss. Losing packets may screw with the throughput, since we will not measure the time it takes to receive packets correctly, if we are losing a significant amount of them. Thus we performed an experiment, to tests the number of packets lost, when varying on the same variables as in the throughput experiment. The results showed that the Rust server barely lost any packets and thus we could conclude that it is not a threat for the throughput in the Rust server. The Haskell server lost slightly more packets when filtering in kernel space than user space. However, the differences were very small so we cannot conclude it as a definite threat to the throughput of the Haskell server.

We compared the ease of use of some specific feature of the libraries. Our comparison showed, a significant amount of time is spend managing the stack, input and output registers for function calls, and what helpers to call to access elements of an eBPF map, when developing in fun eBPF. This is mostly abstracted away in Aya, since Aya handles stack management and standard methods for accessing arrays replace the helper calls. fun eBPF however, inherently provides more control, e.g. by allowing the user to directly write to the stack.

For future work we want to focus on developing the experiments and asses threats such as rate limiting, background system variability, and the relatively small number of test runs. We would also like to extend our set-up with more eBPF programs and a variation of server instances, so we can perform even more interesting experiments in the future.

References

[1] Dominik Scholz et al. "Performance Implications of Packet Filtering with Linux eBPF". In: *30th International Teletraffic Congress (ITC 30)* (2018). URL: https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/ITC30-Packet-Filtering-eBPF-XDP.pdf.

[2] Bolaji Gbadamosi et al. *The eBPF Runtime in the Linux Kernel.* 2024. arXiv: 2410.00026 [cs.OS]. URL: https://arxiv.org/abs/2410.00026.

[3] 2025 eBPF.io authors. *eBPF Documentation.* URL: https://ebpf.io/what-is-ebpf/.

[4] Wikipedia authors. *Protection ring.* URL: https://en.wikipedia.org/wiki/Protection_ring.

[5] Wikipedia authors. *User space and kernel space.* URL: https://en.wikipedia.org/wiki/User_space_and_kernel_space.

[6] *aya-rs/aya Github.* URL: https://github.com/aya-rs/aya.

[7] Donald Hunter and Sanjeev Rampal. *eBPF application development: Beyond the basics.* URL: https://developers.redhat.com/articles/2023/10/19/ebpf-application-development-beyond-basics#.

[8] Sebastian Paarmann Ken Friis Larsen Matilde Broløs and korreman. *eBPF-tools library.* URL: https://github.com/kfl/ebpf-tools/.

[9] Andy Kuszyk. *Linux fundamentals: user space, kernel space, and the syscalls API surface.* URL: https://www.form3.tech/blog/engineering/linux-fundamentals-user-kernel-space.

[10] Michal Rostecki and Thomas Legris. *Aya: your tRusty eBPF companion.* URL: https://www.deepfence.io/blog/aya-your-trusty-ebpf-companion.

[11] Docs.rs Team. *Rust Documentation.* URL: https://docs.rs/.

7 Appendices

7.1 Appendix A

Packets	simple-socket-filter	valid-command
100	2.000	2.005
1000	2.000	2.000
10000	2.010	2.010
100000	2.055	2.055
1000000	2.500	2.575
10000000	7.955	8.140

Table 7: Throughput measured in seconds (WCT) with 1% invalid packets

Packets	simple-socket-filter	valid-command
100	2.005	2.000
1000	2.000	2.000
10000	2.010	2.010
100000	2.055	2.055
1000000	2.530	2.525
10000000	8.000	8.160

Table 8: Throughput measured in seconds (WCT) with 25% invalid packets

Packets	simple-socket-filter	valid-command
100	2.010	2.005
1000	2.000	2.000
10000	2.010	2.010
100000	2.050	2.055
1000000	2.545	2.560
10000000	8.220	8.295

Table 9: Throughput measured in seconds (WCT) with 50% invalid packets

Packets	simple-socket-filter	valid-command
100	2.010	2.000
1000	2.000	2.000
10000	2.010	2.010
100000	2.055	2.050
1000000	2.540	2.600
10000000	8.285	8.245

Table 10: Throughput measured in seconds (WCT) with 75% invalid packets

Packets	simple-socket-filter	valid-command
100	2.010	2.005
1000	2.000	2.000
10000	2.010	2.010
100000	2.060	2.060
1000000	2.570	2.520
10000000	8.375	7.985

Table 11: Throughput measured in seconds (WCT) with 99% invalid packets

7.2 Appendix B

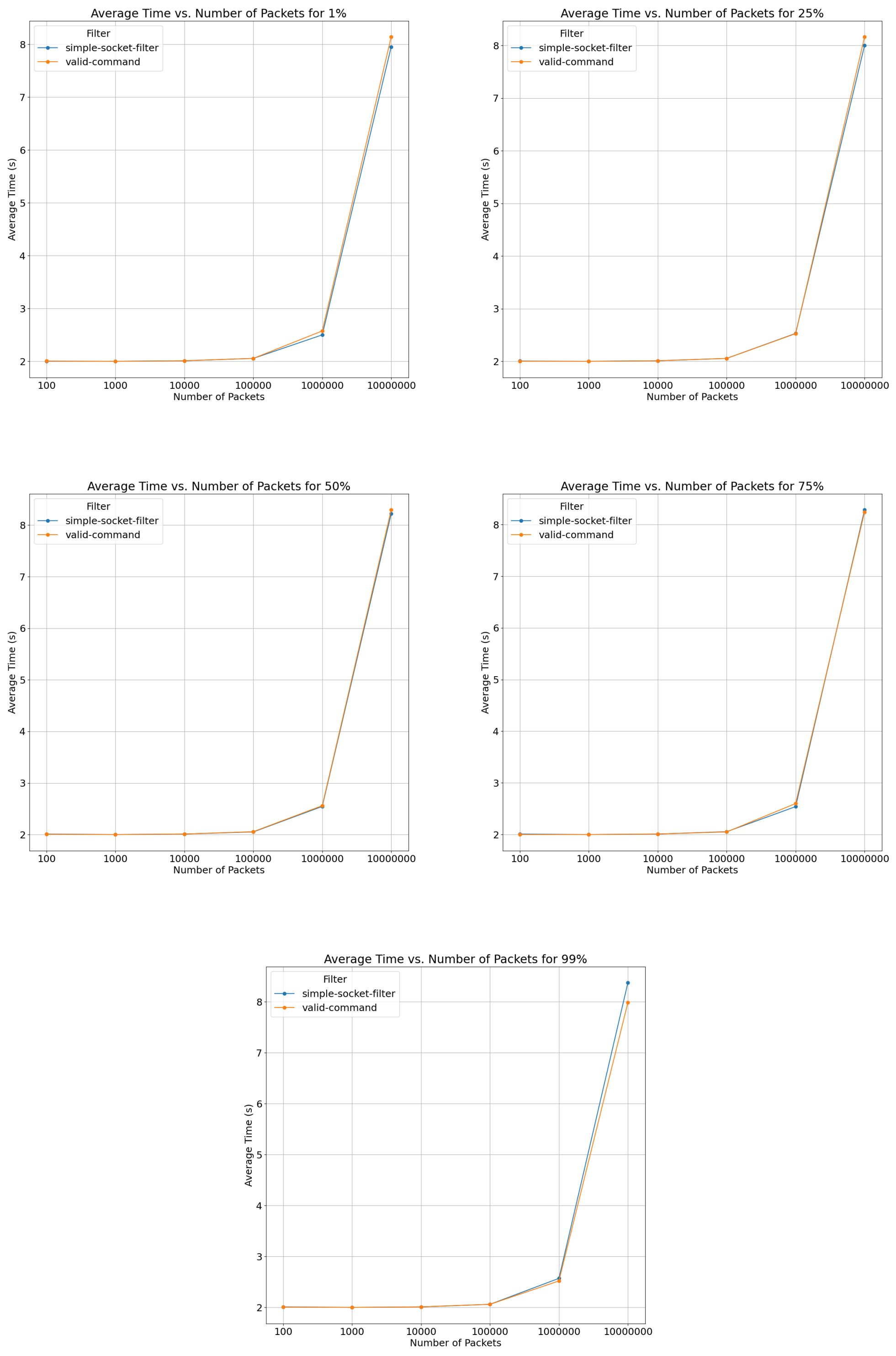

Figure 2: Graphs for throughput measured in seconds (WCT) for varying percentages of invalid packets

7.3 Appendix C

Percentage	Filter	100	1000	10000	100000	1000000	10000000
1	Kernel	0.00%	0.00%	0.00%	1.46%	1.86%	3.07%
	User	0.00%	0.00%	0.00%	1.39%	1.76%	2.90%
25	Kernel	0.00%	0.00%	0.00%	1.42%	1.74%	1.38%
	User	0.00%	0.00%	0.00%	1.32%	1.64%	1.30%
50	Kernel	0.00%	0.00%	0.00%	0.30%	1.59%	1.26%
	User	0.00%	0.00%	0.00%	0.42%	1.59%	1.30%
75	Kernel	0.00%	0.00%	0.20%	0.10%	0.69%	0.60%
	User	0.00%	0.00%	0.00%	0.00%	0.67%	0.60%
99	Kernel	0.00%	0.00%	0.00%	0.00%	0.00%	0.03%
	User	0.00%	0.00%	0.00%	0.00%	0.00%	0.03%

Table 12: Packet loss in percentage for the Haskell server, when filtering in either user and kernel space

Percentage	Filter	100	1000	10000	100000	1000000	10000000
1	Kernel	0.00%	0.00%	0.00%	0.00%	0.05%	0.01%
	User	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
25	Kernel	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	User	0.00%	0.00%	0.00%	0.04%	0.00%	0.00%
50	Kernel	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	User	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
75	Kernel	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	User	0.00%	0.00%	0.00%	0.00%	0.01%	0.00%
99	Kernel	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	User	0.00%	0.00%	0.00%	0.00%	0.00%	0.01%

Table 13: Packet loss in percentage for the Rust server, when filtering in either user and kernel space