Composing eBPF Programs Made Easy with HIKe and eCLAT

Anonymous Author(s)

ABSTRACT

With the rise of the Network Softwarization era, eBPF has become an hot technology for efficient packet processing on commodity hardware. There exist production-ready tools based on eBPF that offer unrivaled performance characteristics (e.g. the Cilium framework). On the other hand, the development of custom eBPF solutions is a challenging process that requires highly qualified human resources. This situation hinders the full exploitation of the eBPF potential. In this paper, we first identify the obstacles faced by eBPF developers and propose the *HIKe* framework, where HIKe stands for "Heal, Improve and desKill eBPF". HIKe is based on an additional Virtual Machine abstraction (the *HIKe VM*) on top of the eBPF VM and on the concept of *chaining* of HIKe eBPF programs. The HIKe framework eases the writing of eBPF programs, hence improving the productivity even for expert developers. Still not content, we mapped the HIKe VM abstraction and its composition approach into a high level programming framework named eCLAT (eBPF Chains Language And Toolset). With eCLAT the developer can write scripts in a pythonish language that compose HIKe eBPF programs, with no performance penalty with respect to composing the programs using the HIKe framework.

CCS CONCEPTS

• **Networks → Programming interfaces**; **Programmable networks**.

KEYWORDS

eBPF, network programmability, network programming languages

1 INTRODUCTION

In typical datacenter networking scenarios, the packet processing nodes are based on general purpose processors (e.g. x86) and run Linux based Operating Systems. In order to support the demanding requirements of workloads running on containers and VMs, there is the need of fast and efficient packet processing solutions targeted to Linux/x86 nodes. In the last few years, the eBPF technology has gained a prominent position among the solutions to improve the packet processing performance of such nodes. For example, eBPF is the core technology of Cilium, a leading framework for secure networking in the Kubernetes container orchestration platform.

While Cilium represents an remarkable success of eBPF, there are a few annoying limitations and issues in the eBPF architecture and development model that prevent a wider application of this technology and limit the advantages that it could bring. In this paper we present two related frameworks (HIKe and eCLAT) and show how they can substantially improve the applicability and usability of eBPF.

1.1 Wizards and Muggles

The development of networking code inside the kernel of an Operating Systems is always a task reserved to domain experts and not to generalist programmers. What typically happens is that networking tools are developed by a limited number of experts (the Wizards) and then operated by a large number of users (the Muggles). Take for example the netfilter firewall in the Linux OS, written by a restricted number of Wizards and operated by tens of thousands of system administrators around the world which can use the iptables abstraction to configure the firewall rules.

The eBPF development process has its nuances and issues that makes it even more difficult than normal kernel programming and restricts it to an "elite" of experts. eBPF is inherently complex, forcing the developers to program in elementary C language and sometimes even in assembly or bytecode. eBPF programming is more an art than a science [x] and the metaphor of Wizard and Muggles fits perfectly to the eBPF ecosystem.

The development of new solutions based on eBPF and the evolution of the existing eBPF frameworks is constrained by the capacity of Wizards to develop inside the eBPF framework so that usable tools can be offered to Muggles. In turn, this is limited by the inherent complexity of the eBPF model and by a number of issues of the eBPF development process and tool-chain. The eBPF Wizards have to fight hard against the eBPF shortcomings and pitfalls to earn their keep.

1.2 eBPF made easy?

Our work aims at answering the following specific research questions, which concern either the Wizards or the Muggles in the eBPF ecosystem.

For the Wizards:

- How can we simplify the work (and the life) of the Wizards?
- In particular, why eBPF programming is so difficult and time consuming?

- How can we improve the productivity of eBPF code development?
- It is possible to compose several small eBPF programs into a large and arbitrarily complex eBPF program without having to fight with unexpected issues in the compilation/verification of the composed program?

For the Muggles:

- Is it possible to offer the Muggles a certain degree of flexibility to handle everyday scenarios, without the need of asking the Wizards to make a new version of the code for each need?
- Is it possible to offer simple programming abstractions to the Muggles, so that they can compose eBPF programs without dealing with the inherent complexity of eBPF?
- Assuming such a programming abstraction for Muggles is designed and implemented, is there a performance penalty in using the abstraction instead of low-level coding in eBPF?

Unfortunately, after a first glance to the eBPF technology one could answer that none of the above is possible. The eBPF model is very strict, severe limitations are purposely added to the characteristics of the eBPF programs that are against flexibility and dynamic "composability". Hence we need a technology breakthrough, which can provide a positive answer to the two questions that sum up the ones listed above:

- is there a way to overcome the limitations of the eBPF model and tool chain, helping the Wizards to develop eBPF solutions in a more time effective way?
- is there a way to allow Wizards to build eBPF tools that are truly "programmable" by Muggles?

1.3 Our contributions: HIKe and eCLAT

In section 2, we shortly introduce eBPF and we identify and discuss some of the issues that make the development in eBPF so complex. Then we show how to address these issues with two complementary solutions:

(1) **HIKe** (Heal, Improve and desKill eBPF), an add-on extension to eBPF that defines a new Virtual Machine on top of the eBPF one;
(2) **eCLAT** (eBPF Chains Language And Toolset), a high level programming framework and execution environment.

HIKe is designed for the eBPF Wizards and is meant to boost the productivity of the expert developers. HIKe offers the Wizards a new paradigm to compose eBPF programs, favouring modularity and code reuse. A Wizard can easily turn an eBPF program into a HIKe eBPF program, that can be re-used as a component inside the HIKe framework. The proposed *HIKe Virtual Machine* (or *HIKe execution environment*) is the key breakthrough on top of which HIKe and eCLAT can provide the answer to the questions raised in the previous subsection.

eCLAT is designed for the Muggles to take advantage of the modularity and composability features of the HIKe framework without the hassles and pitfalls of the eBPF Virtual Machine and its programming in C and assembly language. eCLAT moves the eBPF programming away from C/assembly/bytecode programs towards high-level, python-like scripts.

The fundamental motivations and general concepts of HIKe are described in section 3, while section 4 provides an in depth technical analysis of the main design and implementation aspects, focusing on a concrete use case scenario. Section 5 discusses the eCLAT framework, also describing concrete use cases.

The performance aspects of HIKe and eCLAT are analysed in section 6, while section **??** describes a fully reproducible test-bed in which both the proposed framework are deployed along with the use cases described in the previous sections. The related work is analysed in section 8, while section **??** provides the conclusions and some hints on ongoing and future work.

Summing up, we recall the main contributions of this paper:

- design and implementation of the HIKe Virtual Machine, which is the core of the HIKe framework and the main breakthrough of this paper
- the HIKe framework supports the composition of HIKe eBPF programs, with a paradigm which is simpler for the expert eBPF programmers and avoids the hassle of the eBPF verifier
- design and implementation of the eCLAT framework targeted to system administrator and developers with no in-depth experience in eBPF and Linux kernel networking
- the eCLAT framework includes a python-like language to compose HIKe eBPF programs, a run time environment that keeps track of the deployed programs and eCLAT scripts and of their dependencies, simple tools to read/write the eBPF "maps" needed to interact with HIKe eBPF programs
- performance evaluation of several key aspects of HIKe and eCLAT
- realization of a fully replicable testbed for HIKe and eCLAT, including all the use cases described in the paper

2 EBPF SHORTCOMINGS AND PITFALLS

eBPF is definitely a complex technology. Developing complex systems based on eBPF is challenging due to the intrinsic limitations of the model and the known shortcomings of the tool chain (not to mentions few bugs that can affect this tool chain). The learning curve of this technology is very steep and needs continuous coaching from experts.

In subsection 2.1 we provide a short overview on the eBPF technology, without claiming to be exhaustive. In the subsequent subsections, we identify the main shortcoming and pitfalls of eBPF and its current implementation. This analysis provides the motivations for the HIKe and eCLAT frameworks that will be presented in the following sections.

SS: @Piero: vedere nei commenti in Overleaf

2.1 eBPF overview

The extended Berkeley Packet Filter (eBPF) [8] is a low level programming language that is executed in a Virtual Machine (VM) running in the Linux kernel. eBPF has been profitably used to efficiently and safely manage packets in a very flexible way, defined by eBPF programs, without requiring any changes in the kernel source code or loading kernel modules.

eBPF programs can be written using assembly instructions that are converted in bytecode or in a restricted C language, which is compiled using the LLVM Clang compiler as depicted in figure 1. The bytecode has to be loaded into the system through the `bpf()` syscall that forces the program to pass a set of sanity/safety-checks performed by a *verifier* integrated in the Linux kernel. In fact, eBPF programs are considered untrusted kernel extensions and only “safe” eBPF programs can be loaded into the system. To be considered safe, a program must meet a number of constraints such as limited number of instructions, limited use of backward jumps, limited use of the stack and etc. The verification step assures that the program cannot crash, that no information can be leaked from kernel to user space, and it always terminates. The verifier is a static code analyzer that walks the program instruction by instruction, updating internal states and exploring all paths of conditional branches. In order to pass the verification step, the eBPF programs must be written following several rules and limitations, that can impact on the ability to create powerful network programs [4]. To sum up, an eBPF program must be compiled and then *verified* before being loaded in a running Linux system. This process is described in Fig. 1.

The execution of eBPF programs is triggered by some internal and/or external events like for example the invocation of a specific syscall or the reception of a network packet. Programs are designed to be stateless so that every run is independent from the others. The eBPF infrastructure provides specific data structures, called *BPF maps*, that can be accessed by the eBPF programs and by the userspace when they need to share some information.

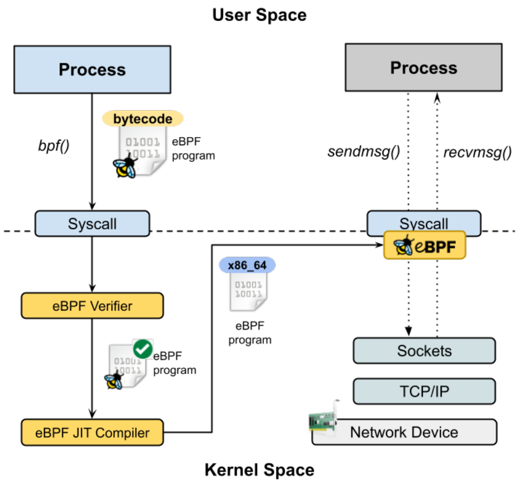

Figure 1: eBPF programs compilation and verification

Focusing on packet processing, eBPF program can be attached to different hooks and packets trigger their execution. Among these hooks, we focus for our purposes on the so called eXpress Data Path (XDP) hook. XDP [7] is an eBPF based high performance packet processing component merged in the Linux kernel since version 4.8. XDP introduces an early hook in the RX path of the kernel, placed in the NIC driver, before any memory allocation takes place. Every incoming packet is intercepted *before* entering the Linux networking stack and, importantly, before it allocates its data structures, foremost the `sk_buff`. This accounts for most performance benefits as widely demonstrated in the literature (e.g., [16] [13]). Another important hook for packet processing with eBPF is the so called Traffic Control (TC) hook. A packet handed to an eBPF program in the TC hook has been already processed to some extent by the networking code of the Linux kernel. Depending on the needs of the specific scenario, it may become preferable to use the XDP or TC hook. Note that when an eBPF programs is hooked to a certain event, it is associated with a specific execution context with given capabilities: eBPF/XDP programs are different from eBPF/TC programs.

2.2 The verification hell

The `bpf()` kernel system call is used to load an eBPF program so that it will be executed in a specific hook. The loading process starts after the compilation phase, which translates the C code into the eBPF bytecode. The loading usually comprises two steps: verification that ensures that the eBPF program is safe to run and JIT (Just In Time) compilation that translates the eBPF bytecode into the specific instruction set of the Linux machine (e.g. x86, arm, 64 or 32 bits...). On few

Linux architectures JIT compilation is not available and the eBPF bytecode will be interpreted at run time when the eBPF program is executed.

Let us focus on the verification phase. The kernel validation approach is adequate for simple eBPF programs i.e few instructions, loop-free code, no complex pointer arithmetic, while it has been shown to be a very tough obstacle to the development of complex applications [4]. As analysed in [6], there are four main issues: i) the verifier reports many false positives, forcing developers to insert redundant checks and redundant accesses, ii) the verifier does not scale to programs with a large number of paths, iii) it does not support programs with unbounded loops iv) its algorithm is not formally specified. This often causes that even a semantically correct program does not pass the validation.

One of the reasons for these problems is that the compiled bytecode that is offered to the verification step is the results of the *optimization* procedures executed by the compiler. For optimization reasons, the compiler can change the sequence of operations with respect to the C source code and this can violate some constraint that must be checked by the verifier.

For all the reasons above, the verification step is the main obstacle in the implementation of complex eBPF applications.

2.3 eBPF (un)composability

Software engineering teaches us the great advantages of modularity, composability, code/component reuse. All these best practices are dramatically out of reach for the current eBPF framework.

In normal software engineering, large systems are divided in smaller components, taming their complexity. The simpler pattern to achieve this decomposition is the *function call*. With this pattern, a caller program hands over the control flow (and possibly some parameters) to a called function/program. The called function/program then returns the control flow (possibly with return parameters) to the caller program.

In eBPF this can only be achieved at compile time, by coding a *bigger* eBPF program that includes the calls to the functions and the logic to glue them. Moreover the eBPF compiler by default uses the “inline” approach to handle the function calls. This means that there is no real transfer of the flow and of the parameters to a different execution context, but the program logic is “flattened out” into a single big execution context. The inline approach may greatly increase the size of the code block to be compiled, optimized and verified. This may cause troubles, as the probability that the verifier will complain about the compiled and optimized code dramatically increases with the size and complexity of the source code.

In order to overcome the limitation of the inline approach, the eBPF community has recently started to introduce the “bpf-to-bpf call” pattern. With this pattern the called function is not inlined in the calling code, but the traditional concept of subroutine is used. This approach is only available in recent kernels and only for x86 architecture (see [3]). Moreover there are limitations in the operations that can be executed inside the subprograms, mostly due to some issues related to the management of the stack space of the eBPF VM.

Moving away from the compile-time composition (which actually means composition by coding-compilation-verification), what about the composition of already compiled and verified eBPF programs? The eBPF framework offers the possibility to compose eBPF programs using the concept of *tail call*, which is a different pattern (i.e. less powerful) than the *function call*. Tail calls are a specific eBPF feature that allows an eBPF program to execute (or better, launch) another eBPF program. A tail call is different from a function call since the eBPF VM reuses the current stack frame and executes the new eBPF program without adding an extra call stack frame. Tail calls are commonly used to implement complex systems comprising dynamic chains of programs that dispatch the packet to perform a joint elaboration. Tail calls suffer from severe limitations: it is not directly possible to pass parameters from the called program to the called one, nor the execution flow can go back to the calling program (at most if could be re-executed from the beginning). Clearly, the developer cannot use tail calls according to the *function call* pattern that everyone is used to.

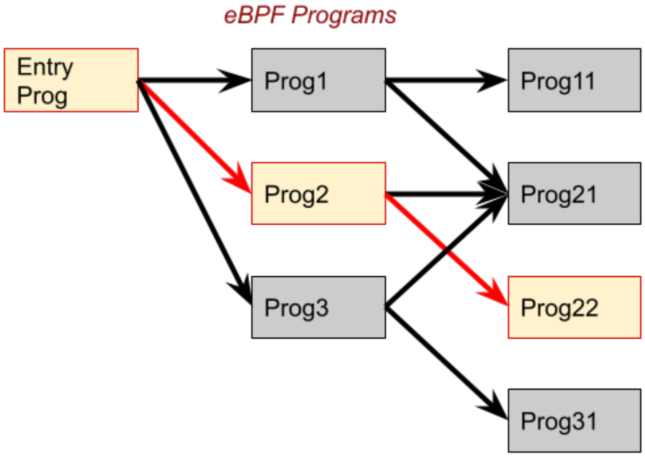

Figure 2: Chaining eBPF programs with tail calls

In Fig. 2, we can see an example of a composition of eBPF programs that invoke each other via tail calls. When an incoming packet is handed to the eBPF “Entry Prog”, it can select another program to handle the packet and execute it with a tail call. The structure of the interconnections with the next programs to be called is “statically” coded in the caller eBPF programs. From the service development point of view, we need to set up the flow logic at compile time and implement the branches of the tree inside all the eBPF programs involved in the composition. If we want to change the structure of the interconnection, we need to change the code of the involved eBPF program code, re-compile and (hopefully) pass through the verification hell. For example, this kind of strategy is implemented by the state-of-the-art eBPF composition framework called Polycube [10].

2.4 The clumsiness of BPF maps

eBPF programs need to interact with user space programs to get configured and to provide information (e.g. statistics). Moreover, eBPF programs may need to access (i.e. read/write) global state information, which represents a way to exchange information among different invocations of the same eBPF programs of among invocations of different eBPF programs. The eBPF framework provides the abstraction of *BPF maps* to these purposes. The BPF maps are key/value stores residing in the Linux kernel. Different types of BFP maps are supported, i.e. with different key and value types (see [3]). Generic BPF maps can be defined by the developers of eBPF systems, while there are specific BPF map types for maps that are needed by the eBPF framework. The use of BPF maps is not straightforward for the developer, in particular accessing BPF maps from user space programs is a cumbersome operation.

Going back to the wizard and muggles metaphor, working directly with BPF maps is a task for wizards, while muggles should be shielded from developing code to read/write the BPF maps. This means that an harder task of the wizard is to design the libraries and/or GUI tools that can be used by the muggles with basic or no programming skills to interact (indirectly) with the BPF maps.

SS: we could add here a mention to the BCC attempt to solve this problem and to its shortcomings

The risk of race conditions is another critical issue that needs to be taken into account when designing the interaction of eBPF programs and user space program with the BPF maps. As multiple eBPF programs can get executed in parallel to process a set of incoming packet (one per core), the access operations on the maps may conflict. For example the eBPF framework does not offer the possibility to get a lock on multiple BPF tables to perform an atomic set of write operation. It is a task of the developer to handle the concurrency when needed and this is in general a hard task even for wizards.

3 HIKE BREAKTHROUGH

In the previous sections, we have identified some shortcomings and pitfalls of eBPF, of its development process and tool chain. Ideally, we would like to compose eBPF programs using a *function call* pattern and without the hassles of the verification phase. These requirements cannot be met by the current eBPF framework, nor it is possible to directly enhance eBPF to meet our requirements as we would violate some fundamental assumptions (e.g. the safety guarantees provided by the verifier). The proposed breakthrough is to add a new VM abstraction on top of the eBPF VM, called HIKe (Heal, Improve and desKill eBPF) VM. Inside the HIKe VM, we can compose and execute eBPF programs (actually, HIKe eBPF programs) using the *function call* pattern, without going through the verification phase and at the same time meeting all the eBPF safety constraints!

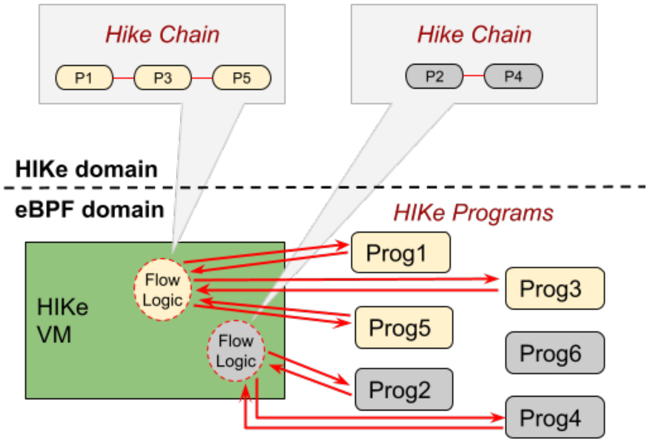

Figure 3: Composition of HIKe eBPF programs with a HIKe Chain, using the HIKe VM

The proposed solution is shown in Fig. 3. From the point of view of the eBPF framework, the HIKe VM is a regular eBPF program. The HIKe VM is capable of executing (interpreting) a bytecode that we call *HIKe Chain* that describes the composition of HIKe eBPF programs. From the point of view of the eBPF framework, a HIKe eBPF program is a regular eBPF program. To turn an eBPF program into a HIKe eBPF program, we have to “decorate” it with some helper functions contained in the HIKe library and then to recompile/verify/load it.

The HIKe chain bytecode executed inside the HIKe VM is responsible for deciding the flow logic (i.e., the eBPF programs to be executed and the logic thereof). Differently from the tail call approach described in Fig. 2: i) the eBPF programs that are composed are not aware of the structure of the interconnections; ii) the flow logic can be based on the results of the execution of the eBPF programs, but it can be changed in a flexible and modular way, independently from the HIKe eBPF programs that are compiled and verified once for ever; iii) changing the flow logic is NOT subject to the eBPF verifier, because the new HIKe chain is compiled for the HIKe VM, not for the eBPF VM.

AM: dovremmo dire che tutto questo si fa SENZA modificare la eBPF VM e quindi non andando a introdurre vulnerabilità o problemi di sicurezza per il kernel. Ciò impatta positivamente anche sul fatto che quando il kernel si aggiorna noi non dobbiamo fare niente, perchè non tocchiamo nessun suo componente **SS:** ok hai ragione

The bytecode of a HIke chain is interpreted by the HIke VM which is an eBPF program that has been previously verified before being loaded. Consequently, any instruction in that chain is interpreted by the HIKe VM which results in eBPF operations which have already been verified to be safe by the eBPF verifier itself.

The HIKe VM supports all the basic operations needed to implement arbitrary logic: if, jump, conditional jump, arithmetic operations. Let us now introduce the characteristics of the HIKe VM and of its instruction set. A fundamental breakthrough is that we have reused a subset of the eBPF VM instruction set as the basis for the HIKe VM instruction set. The the HIKe VM is bytecode compatible with the eBPF VM. We write the HIKe chains in C language and reuse the Clang/LLVM toolchain to generate the HIKe VM bytecode. Thanks to this design decision, we do not have to re invent the wheel and the HIKe framework fully benefits from the maturity of the LLVM toolset.

SS: Fare una figura che mostra il processo di compilazione e load delle HIKe chain e la relativa tool chain (magari affiancata alla tool chain eBPF?)

In the HIKe VM, a HIKe Chain can also directly invoke other HIKe Chains. The invocation of a HIKe Chain uses the same *function call* pattern used to compose HIKe eBPF programs, maintaining the same syntax and semantics. The HIKe VM, therefore, provides a unified and simple calling model for invoking both other HIKe Chains and HIKe eBPF programs, promoting and maximizing the code reuse.

Considering that the application domain of the HIKe VM is the handling of network packets, we complement the HIKe VM with some helper functions to manipulate (read and write) the packet bytes.

Finally, we mention another breakthrough of the HIKe framework. The HIKe chain that describes the flow logic to be applied to a packet is **associated** with the packet and "follows" the packet though its "journey" across the different HIKe eBPF programs that are executed inside at node. This avoids the "reclassification" issue that affects the normal function chaining approaches, which often requires to keep some state information to reclassify packets after a function has been executed to forward the packet to the next function.

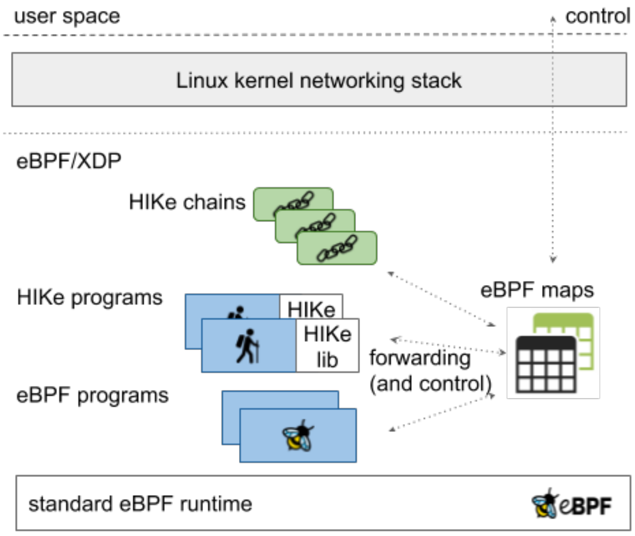

Figure 4: HIKe eBPF/XDP Architecture

4 HIKE DEEP DIVE

Fig.4 depicts the overall architecture of the HIKe framework, which is based on the following entities:

- **HIKe eBPF Program** - eBPF/XDP program which is "decorated" with the HIKe Libraries and can be composed (chained) with other HIKe Programs to process a packet;
- **HIKe Chain** - the composition (chaining) of HIKe Programs, of logical/arithmetical operations and control instructions; a HIKe Chain is written in C language and compiled in the HIKe VM bytecode;
- **HIKe VM** - an eBPF program that acts like a Virtual Machine and executes the bytecode of the HIKe Chains;
- **HIKe Chain Loader** - an eBPF program (or a set of programs) which associates a HIKe Chain to a packet and hands over the packet to the HIKe VM that will execute the chain;
- **HIKe Persistence Layer (PL)** - used to *register* the HIKe eBPF Programs so that they can be called inside the HIKe Chains and to store the bytecode of the HIKe Chains; the HIKe PL is based on eBPF Maps.

4.1 HIKe eBPF Programs

The HIKe eBPF Programs are the basic components of the HIKe framework. They are regular eBPF programs coded in C language that are decorated (i.e. they include calls to a HIKe library) so that they can be integrated in a HIKe Chain. In particular, a HIKe eBPF Program may accept a maximum of 5 input parameters which are provided by the calling HIKe Chain and it terminates by returning the execution control to the HIKe VM specifying a return code and an output parameter. On the basis of the return code, the HIKe VM can take different actions, i.e: deliver the packet to the kernel, drop the packet or even call another HIKe Chain. The output parameter may provide the calling HIKe Chain with information that influence the further execution of HIKe Chain itself.

The HIKe eBPF Programs written in C are first compiled into the HIKe VM bytecode, then they need to receive a unique (per node) Program ID and to be registered in the HIKe persistence layer. This registration phase is discussed in section 4.11.

The source code of a HIKe eBPF Program is shown in Listing 1. Two macros (i.e. pre-processor directives) are used to transform an eBPF program into a HIKe eBPF Program.

Listing 1: A HIKe eBPF program

```
HIKE_PROG(allow_any)
{
    bpf_printk("HIKe allow_any");
    bpf_printk("REG_1=%llx, REG_2=%llx,
               _I_REG(1), _I_REG(2));

    return XDP_PASS;
}
EXPORT_HIKE_PROG(allow_any);
```

The `HIKE_PROG(progname)` macro is used to define a HIKe eBPF Program. With this macro, the execution context of the eBPF/XDP program (the struct `xdp_md *ctx`) and the HIKe VM context are made implicitly available to the program.

The `HIKE_PROG` macro expansion is shown in Listing 2. The `struct hike_chain_regmem *regmem` is used for passing the input parameters to a HIKe eBPF Program when it is going to be called from a HIKe Chain. Therefore, the `HIKE_PROG` macro defines the prototype of a HIKe eBPF Program.

Listing 2: Macro that provides the signature of a HIKe eBPF Program

```
#define HIKE_PROG(progname)             \
static __always_inline int progname(    \
        struct xdp_md *ctx,             \
        struct hike_chain_regmem *regmem)
```

In order to be run by the eBPF VM, a HIKe eBPF Program must adhere to the prototype of any XDP eBPF program. Accordingly, the `EXPORT_HIKE_PROG(progname)` makes the HIKe eBPF Program `progname` compatible with the eBPF VM. Such macro passes the XDP context and retrieves the HIKe VM execution context that consists in private registers and stack allocated for each HIKe Chain execution. Then, the `EXPORT_HIKE_PROG` macro automatically adds all the machinery needed to schedule the HIKe VM execution.

Listing 3: Macro that provides the code to interact with the HIKe VM

```
1  #define EXPORT_HIKE_PROG(progname)        \
2  __hike_vm_section_tail(progname)          \
3  int __HIKE_VM_PROG_EBPF_NAME(progname)    \
4      (struct xdp_md *ctx)                  \
5  {                                         \
6      struct hike_chain_regmem *regmem;     \
7      int rc;                               \
8                                            \
9      regmem = hike_chain_get_regmem();     \
10     [...]                                 \
11                                           \
12     rc = progname(ctx, regmem);           \
13     switch (rc) {                         \
14     case XDP_ABORTED:                     \
15     [...] /* remaning XDP_* retcode */    \
16     case XDP_REDIRECT:                    \
17         return rc;                        \
18     }                                     \
19                                           \
20     [...]                                 \
21     hike_chain_next(ctx);                 \
22     /* fallthrough */                     \
23 aborted:                                  \
24     return XDP_ABORTED;                   \
25 }
```

Listing 3 shows the code, simplified for the sake of discussion, of the `EXPORT_HIKE_PROG` macro that makes the HIKe eBPF Program `progname` compatible with an eBPF/XDP program runnable on the eBPF VM.

At line 9, the HIKe VM execution context for the specific HIKe Chain being executed is retrieved. Next at line 12, the HIKe eBPF Program `progname`, defined earlier through the macro shown in Listing 2, is executed.

When `progname` returns, the return code is examined. If the return code matches one of the well-known codes such as `XDP_ABORTED`, `XDP_PASS`, etc, the execution of the HIKe eBPF Program and consequently of the HIKe VM ends by giving the control back to the eBPF VM. Conversely, if the returned code does not match any of those handled by the `switch` at line 13, the flow control is passed to the HIKe VM which continues the execution of the remaining HIKe Chain instructions.

It is worth noting that Listing 3 shows how the HIKe VM and consequently the HIKe Chains are executed. A HIKe eBPF Program is divided into two logical parts: i) the execution of the specific program code; ii) the execution of the HIKe VM which is run *on behalf* of the HIKe eBPF Program and processes the instructions contained in the active HIKe Chain.

The macros shown in Listing 2 and 3 are pretty easy to use (despite to their internal complexity), and they give us the possibility to write HIKe eBPF Programs or to adapt existing eBPF ones very easily.

4.2 HIKe VM

The HIKe VM provides the “glue” between the HIKe Chains and the HIKe eBPF Programs. The HIKe VM executes the flow logic of the HIKe Chains by calling the HIKe eBPF Programs (possibly providing input parameters) and receiving their output parameters, as depicted in Fig. 5. The HIKe VM retrieves from the HIKe Persistence Layer the references to the HIKe eBPF Programs, as well as the bytecode of the HIKe Chains. The HIKe VM can communicate with the HIKe eBPF Programs through a set of eBPF maps which persist in the kernel space and offers a service which is similar to the shared memory for inter-process communication. Fig. 5 also shows that the HIKe VM comprises a set of Registers, the Stack needed to manage the function calls, and it can directly access the Packet information.

AM: dove parliamo delle limitazioni intrinseche della HIKe VM?

AM: qui dentro forse è il caso di far vedere come è fatta questa VM, i registri quali sono, lo stack, la shared memory per "l’IPC" e l’accesso al pacchetto?

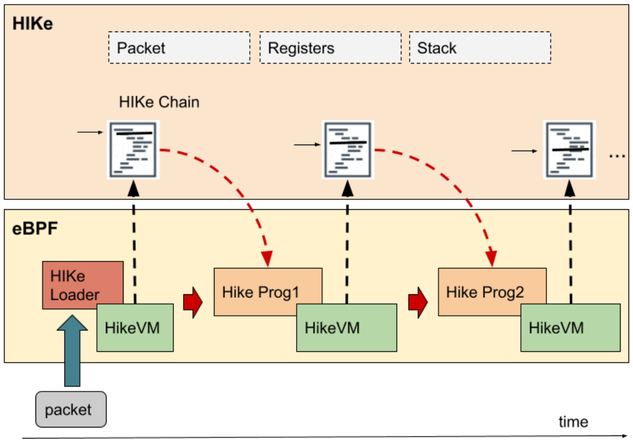

Figure 5: HIKe VM executing a HIKe chain

4.3 HIKe Chain Loader

Upon the arrival of a packet on a network interface (configured for HIKe), an eBPF program called the Chain Loader is invoked. The Chain Loader makes use of the APIs provided by the HIKe VM to associate a HIKe Chain to the packet and start the execution of the Flow Logic of the Chain.

The Chain Loader selects and executes the correct HIKe Chain based on an arbitrary logic that can be programmed in the Loader itself. For example, a Loader may invoke a given HIKe Chain based on the destination IP address of the packet, or the TOS value and so on. As soon as the Loader identifies the ID of the HIKe Chain, it calls the `hike_chain_bootstrap()` function, provided by the HIKe VM, passing the chain ID. Then the HIKe VM associates the Chain to the packet and starts executing the bytecode instructions of the Chain (which are retrieved from the HIKe Persistence Layer).

AM: bisgnerebbe fare un semplice diagramma che mostra quali sono i componenti fondamentali della HIKe VM. La figura 5 tende a ritrarre una sorta di sequence diagram, ma ci manca una parte strutturale. La HIKe VM da dove va a leggere i programmi? Dove sono memorizzati? Chi li memorizza? Quali sono le interfacce con lo userspace per caricare un nuovo programma?

4.4 HIKe Chains

The HIKe Chains are written in C language. The source code of a HIKe Chain is shown in Listing 4. The code is compiled into a bytecode that is subsequently stored in the HIKe Persistent Layer. During this phase, an unique Chain ID is obtained and associated with the Chain itself: it will be used for referring to that Chain from the HIKe VM, the Chain Loader and the other HIKe Chains.

From the point of view of the HIKe VM, a HIKe chain is an executable bytecode, based on the HIKE VM Instruction Set. To make the life easier for the HIKe developers, the definition of a HIKe Chain in C language is extremely simple and intuitive thanks to the use of special macros that are provided by the HIKe framework.

Listing 4: A HIKe chain

```
#define allow(ETH_TYPE)                     \
hike_elem_call_2(HIKE_PROG_ALLOW_ANY,       \
                 ETH_TYPE)

#define mon_ipv6_pkt()                      \
hike_elem_call_2(HIKE_EBPF_PROG_PCPU_MON,   \
                 MON_IPV6_PACKET_EVENT)

HIKE_CHAIN_1(HIKE_CHAIN_MON_IPV6_ALLOW)
{
    [...]
    /* read Ethernet type from packet */
    eth_type = [...];

    if (eth_type == 0x86dd)
       mon_ipv6_ptk();

    /* call the HIKe eBPF Program 'allow'
     * providing the input parameter
     * 'eth_type'.
     */
    allow(eth_type);

    return 0; /* fallthrough */
}
```

In Listing 4 we provide the definition of a HIKe Chain identified by the label HIKE_CHAIN_MON_IPV6_ALLOW by means of the macro HIKE_CHAIN_1. Thanks to this macro, the developer is relieved from the need to deal with the signature of function that represents a HIKe Chain and with the retrieval of the input parameters. In this case, HIKE_CHAIN_1(chainid) declares a HIKe Chain identified by the ID progid which is the only parameter for this chain.

The HIKe framework also provides other macros for defining chains that accept more than one parameters; for example the macro HIKE_CHAIN_2(chainid, type_arg2, arg2) is used to define a HIKe Chain identified by chainid that accepts a second parameter of type_arg2 whose formal parameter name is arg2.

The logic of the aforementioned HIKe Chain is straightforward: when executed, it identifies the type of traffic (i.e. IPv4, IPv6, etc) by reading the Ethernet type. If the Ethernet type is IPv6 (0x86dd), then it counts the number of packets by calling the HIKe eBPF Program mon_ipv6_pkt() .

Finally, the HIKe eBPF Program allow is called, which accepts the analyzed traffic type as a parameter. The allow Program, finally, terminates the execution of the HIKe Chain and for this reason it is said to be *final*.

Note that both the mon_ipv6_ptk() and allow() programs are in turn presented as macros for sake of clarity. Indeed, they expand the HIKe hike_elem_call_2() helper function. This helper function is in charge of invoking the execution of a HIKe eBPF Program, updating the context of the HIKe VM, handling the return code and continuing with the chain flow execution.

4.5 HIKe VM Instruction Set

The bytecode of the HIKe Chains is binary compatible with the eBPF VM. This means that the same Instructions Set of the eBPF VM is used by the HIKe VM, they have the same registers and the ABI (Application Binary Interface) of the eBPF VM, including for example the calling conventions for the functions is preserved. The HIKe VM uses an internal stack to support function calls and to perform the register spilling operations and automatic variables. The HIKe VM is actually a restricted eBPF VM, meant to work only in the eBPF/XDP hook offering to the developer a subset of the functionality of the eBPF VM. By default, the helper functions that are available inside the eBPF VM are not available inside the HIKe VM, unless they have been explicitly exported by the the latter. Note that in the HIKe framework this restriction only applies to HIKe Chains, while the HIKe eBPF Programs are regular eBPF programs and can benefit from all the features (e.g. helper functions) provided by eBPF in the XDP hook.

4.6 HIKe VM Memory Management

In order to execute the HIKe Chains associated with the packets being processed, the HIKe VM implements memory management mechanisms that make it possible to: (i) isolate the execution contexts of the HIKe Chains; (ii) support the function call pattern typical of imperative programming; (iii) provide transparent access to the bytes of a packet for read/write operations; (iv) provide shared memory areas through which eBPF programs, HIKe eBPF Programs, and HIKe Chains can exchange information.

For performance reasons, the HIKe VM maintains the information about the currently running (active) HIKe Chain separately for each logical CPU (i.e. for each core). In particular, the VM maintains the pointer to the currently running HIKe Chain as well as the private stack that supports calling a HIKe Chain from another HIKe Chain.

The HIKe VM maintains for the active HIKe Chain an execution context that includes the state of the VM registers, a stack, and the machine instructions that compose that Chain. The stack of a HIKe Chain is used to store automatic variables and for register spilling purposes. The VM registers specific to the active HIKe Chain are used in conjunction with the stack to pass parameters to and from the HIKe eBPF Programs.

Such an organization of memory enables the HIKe Chains to be independent and isolated from each other. Also the HIKe VM can maintain the state needed to invoke HIKe eBPF

Programs from a HIKe Chain and, eventually, returning back the flow control to the Chain.

A HIKe Chain is able to access both the contents of the packet and the memory shared between HIKe Chain and HIKe eBPF Programs due to the HIKe MMU component that transparently remaps the virtual addresses to the actual addresses where the data to be accessed is present.

Each virtual memory address is 32 bits long and consists of two parts: the first most significant byte of the memory address represents the “segment” or “bank”, while the remaining bits represent the offset within that segment. As a result, there can be a maximum of 256 different segments, in each of which up to 16 M bytes can be addressed.

```
      MSB                             LSB
  +--------------+----------------------+
  | bank (8 bit) |       offset (24 bit) |
  +--------------+----------------------+
```

4.7 Simple and Efficient Packet Handling

The eBPF machine in the XDP hook can read/write the packet (i.e. frame XDP) directly using pointers to the start and the end of the packet, that are contained in the context structure of XDP (`struct xdp_md`). In the HIKe VM, the packet bytes are mapped into a Virtual Memory address of the VM address space, through a simplified MMU (Memory Management Unit) implemented in the HIKe VM.

For the developer of the HIKe Chain, the final result is that the access to the packet bytes is similar to what it is possible to do in an eBPF/XDP program. For example, is it possible to access the *Ethernet type* field using the code shown in Listing 5. `HIKE_MEM_PACKET_ADDR_DATA` is the virtual memory address of the start of the packet, `nthos` is a macro that converts from network byte order to host byte order.

Listing 5: Access to packet fields in a HIKe Chain

```
__u8 *p = HIKE_MEM_PACKET_ADDR_DATA;
__u16 eth_type = nthos(*(__be16*)(p+12));
```

With reference to the Listing 5, in a HIKe Chain it is not strictly necessary to verify that a pointer `p + off` refers to a valid memory area as it would be mandatory to do in any eBPF program, under penalty of rejection by the verifier. The HIKe VM automatically verifies that any access to any memory location is valid and safe. If it is not, the HIKe VM immediately terminates the execution of the HIKe Chain by taking a default action that causes the packet to be dropped.

However, in packet processing operations it is very common to check for the length of the packet in order to decide whether or not to access specific parts of the packet. For this reason, the HIKe VM makes available to the programmer of a HIKe Chain the current length of the packet to which the Chain is associated. The packet length is mapped to a specific HIKe VM memory address (`HIKE_MEM_PACKET_ADDR_LEN`).

Listing 6 shows how to access the Ethernet type of a packet in a HIKe Chain (offset +12 from the beginning of the frame) only if that packet is at least 14 bytes long.

Listing 6: Getting the packet length

```
[...]

__u32 *pkt_len = HIKE_MEM_PACKET_ADDR_LEN;
if (*pkt_len >= 14) {
    eth_type = [...];

    [...]
}
```

4.8 HIKe Chain to HIKe eBPF Program communication

Inside a HIKe Chain each HIKe eBPF Program can access the registers and the stack of the currently running HIKe Chain (active Chain) in both read and write mode. The HIKe VM is designed to honor the ABI (Application Binary Interface) of the eBPF VM and respect the same calling convention. Consequently, the input parameters of the HIKe eBPF Programs are passed in the *scratch registers* `r1-r5`, while the output is returned to the Chain in the register `r0`.

In Listing 1, the HIKE eBPF Program `allow_any` accesses registers `r1` and `r2` via the `_I_REG(REG_NUMBER)` macro to fetch their contents and print them onto the trace pipe. Similarly, a HIKe eBPF Program can return the `x` value to the HIKe Chain simply by writing into register `r0`, i.e: `_I_REG(0) = x`.

As with the eBPF VM, the registers in the HIKe VM are natively 64 bits long. In the future we plan to also support 32-bit register mode to speed up operations that do not involve the upper 32 bits of a register.

4.9 HIKe Chain-to-Chain calls

The HIKe VM implements call functionality between HIKe Chains. This means that a HIKe Chain can invoke the execution of another HIKe Chain and therefore the HIKe VM supports the execution of nested HIKe Chains.

From the developer’s point of view, it is possible to group complex functionality in Chains by creating logical functional structures that simplify both the writing and the management of the network application to be implemented. Note that the call to a HIKe Chain does not exploit the tail calls mechanism and consequently does not impact on the total limit of available tail calls that is set by the eBPF VM to 32. In other words, a number of calls to the HIKe Chain can be made that does not depend on the number imposed by the tail call mechanism.

The maximum number of nested HIKe Chains is however limited by the stack of the HIKe VM which currently provides for a maximum nesting of 8 HIKe Chains. This value can changed by the Wizard during the compilation of the HIKe eBPF Programs.

The HIKe Chain-to-Chain call supports both the passing of input parameters and the return value to the calling HIKe Chain, enabling the realization of complex and structured packet processing logic.

In the implementation of the HIKe Chain to HIKe Chain the same conventions of the HIKe eBPF Program have been followed. However, since each HIKe Chain is an independent execution context, the calling HIKe Chain copies the input parameters into the registers `r1-r5` of the called HIKe Chain. Consequently, the HIKe Chain that is about to return the control copies the return value into the `r0` register of the calling HIKe Chain.

4.10 Code portability

HIKe Chains are written to be able to run on different kernel versions. While in principle HIKe eBPF Programs may suffer from portability problems due to different kernel internals from version to version, in practice this is not the case since the API for eBPF/XDP is quite stable. On the contrary HIKe Chains do not have any portability problem because the HIKe VM takes care of running the HIKe Chains adapting to the kernel version, to the available libraries etc. With HIKe Chains, the concept of "write once, run everywhere" is pushed to the limits.

For eBPF HIKe Programs that need features that are present in some kernel versions and not in others, the Wizards can provide the right program version for the right kernel.

4.11 HIKe Persistence Layer

The HIKe Persistence Layer (HIKe PL) solves two fundamental problems: i) identifies the HIKe eBPF Programs so that the HKe Chains can call them by their ID; ii) identifies the HIKe Chains and stores their bytecode, so that HIKe VM can retrieve the bytecode and the HIKe Chains can call other Chains by their ID.

We have purposely implemented a very thin persistence layer, which can be abstracted as two (key, value) maps, represented in table

AM: qui non ho idea di cosa mettere...IL management è fatto da tool esterni ad HIKe VM come ad esempio eclat o tool userspace come bpftool, script e olio di gomito...

SS: dobbiamo raccontare il processo di registrazione dei Programmi e di memorizzazione delle Chain
il framework HIKe "kernel level" non offre la gestione degli ID... abbiamo pensato che sia meglio gestirlo "user space level"
dobbiamo raccontare il discorso degli ID dei programmi e delle Chain che (allo stato attuale) devono essere noti a priori se scriviamo i programmi e le Chain in C in HIKe
questo ci serve per vendere anche meglio eCLAT

XXX annotation per dire che mappe usa un programma organizzazione delle mappe in directory

5 THE ECLAT ABSTRACTION

Although HIKe allows the programmer to structure its code in a flexible and modular fashion, the technical entry barrier for a developer that wants to use this concatenation for the development of network functions is still high. In fact, using HIKe there is still the need to interact directly with the code of eBPF programs and eBPF tables/maps that are used to store configuration and status of different programs and HIKe. For this reason we designed eBPF Chains Language And Toolset (eCLAT), a framework and a language that supports eBPF/HIKe based network programmability providing a high level programming abstraction to the HIKe framework.

eCLAT provides a high-level programming abstraction of the HIKe framework that simplifies the reuse and the composition (chaining) of the HIKe EBPF programs. The eCLAT scripting language is used to compose eBPF programs and to configure eBPF maps hiding the difficult technical details (i.e. avoiding the need of low level programming eBPF using the C language).

In a nutshell, eCLAT provides an easy-to-use python-like language to simplify the writing of HIKe chains.

5.1 Features

The main eCLAT features of are:

- **Import the HIKe programs** and humanizing them by associating their IDs to them some human-friendly names (unique strings). Then eCLAT grabs from the HIKe programs the maps schema which are specific for each HIKe programs and describe the status needed for the execution of the program (eBPF programs are stateless);
- **Configure and interact with HIKe programs** by populating and interacting with the maps of the HIKe programs in a simple and and user-friendly way;
- **Define an entry point** which is the eBPF program responsible to trigger an HIKe chain. This can be for instance a classifier such as the HIKe classifier which are based on the destination IPV6 address: according to that, different chains can be executed, implementing SRv6 network programming model.
- **Define and load the HIKe chains** which can be lists of HIKe programs or of HIKe chains (chain-in-chain);

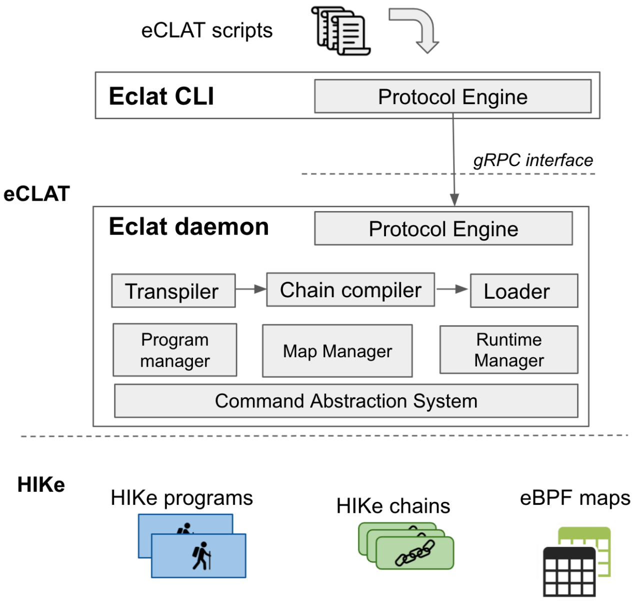

Figure 6: eCLAT Architecture

5.2 Architecture

Figure 6 shows the architectural view of eCLAT. At the bottom of the architecture we have the eBPF/XDP domain where we can find the HIKe eBPF programs, the HIKe chains and the eBPF maps used by HIKe and the HIKe programs.

Eclat has been implemented as a daemon (*eclatd*) which is in charge of compiling and configuring the HIKe programs, keeping the state of the ones that are running and interface with that for instance to acquire information about their status (e.g., the current measure in case of monitoring programs). One of main reasons behind the demon-based architecture resides in a security limitation of eBPF. Indeed, just the process that mounted the eBPF filesystem (or one of its children) can have access to the filesystem itself. Thus, the demon architecture simplifies the management of maps and programs, and also provides a serialized access to the eBPF environment.

The Eclat daemon receives user commands from a CLI (*eclat-cli*) through a gRPC interface. The structure of the data is described through a protocol buffer language [1]. We implemented a `run` service which transcompiles an eCLAT script into an HIKe program, compile the HIKe program and the load it into the memory.

As shown in figure 6, internally the eCLAT daemon is composed by the following functional blocks:

- **Transpiler**: is in charge of translating the source code, from the (python-like) Eclat script to a (C) HIKe program. This module is composed of a lexer and a parser. It creates the Abstract Syntax Tree (AST) and then generates the source code of the corresponding HIKe program, ready to be compiled.
- **Chain compiler**: it takes the output of the previous component and compiles it, to generate the artifacts (object files) through the execution of a dedicated Makefile.
- **Chain loader**: loads the output of the previous component in memory. This is done through the execution of a loader shell script.
- **Command Abstraction System (CAS)**: provides an abstraction over the different shell commands that needs to be invoked on the operating system.
- **System configurator**: is responsible to set up the operating system. This includes the mounting of the directory `/sys/fs/bpf/` and `/sys/kernel/tracing`.
- **Protocol engine**: implements the gRPC protocol service and is responsible for the communication with the CLI
- **eBPF HIKe Program Manager (PM)**: loads and unloads HIKe eBPF programs. In particular, it loads the programs imported by the eCLAT script (if not already loaded) and generates/retrieves their IDs (which are passed to the Runtime Manager). These identification numbers will be fundamental for the transpiler phase since the HIKe programs rely on the concept of numerical IDs rather than on the names such as in the eCLAT world.
- **eBPF Map Manager (MM)**: handles the management of the maps at the eCLAT level. During the compilation of the HIKe programs all the information about the structure of the program (variables, functions, structs, etc.) are registered in a JSON file. The file is also parsed to obtain all map-program associations.
- **Runtime Manager (RM)**: collects all the information about the runtime state. The state information is registered through a JSON structure, which contains information regarding: i) the classifiers that have been downloaded and which has been selected for the current case; ii) a static structure for the association of IDs to programs (pairs {`program_name - ID`}); iii) the downloaded/installed HIKe programs; iv) the downloaded chains, together with the list of HIKe programs used and whether of not each chain has been installed; v) the defined maps, together with the list of programs associated with, flagging the programs that are already been installed and the privacy of the programs (shared or private); vi) the environment (ports, interfaces, etc.)

[1] https://developers.google.com/protocol-buffers/docs/overview

5.3 The life of an eCLAT programmer

The implementation of a complete HIKe eBPF system requires several steps:

- **Step 0**: initialization of system-wide HIKe maps with pinning.
- **Step 1**: the first step is devoted to the load of the eBPF HIKe programs. Internally this requires the compilation of the eBPF bytecode, the generation of a program ID, the loading of the program, the pinning of the program and its maps, and the registration to the HIKe framework.
- **Step 2**: once all the HIKe eBPF programs have been successfully registered, we must take care of the HIKe chains. This requires the compilation of the chains with the IDs of the programs within, the generation of the chains IDs and their loading in the HIKe chain map.
- **Step 3**: the final step requires the compilation of the eBPF entry point (usually a packet classifier). The task of this program is to call the right chain according to the classification result and thus it needs to be configured with the related chain IDs. Such IDs can be included directly in the entry point source code ("hard-coded") or read from a map. Finally, the program needs to loaded and eventually attached to the XDP hook.

eCLAT automatize and simply such process. In particular the eCLAT programmer does not need to worry about the HIKe programs/chains IDs but can just refere to them using their names. The eCLAT framework takes care of deriving unique IDs and associate them with the programs/chains. Moreover eCLAT framework will automatically initialize the system (step 0). Then, by interpreting to the python-like script, it will load the required HIKe eBPF programs (step 1), setup the chains (step 2) and configure the classifier (step 3). This greatly simplify the life of the eCLAT programmer.

5.4 Example

There is a node that will classify packets based on TOS. The eCLAT / HIKe node will receive these packets and activate a chain based on the number of TOS found. The Mon program will count the packages that have a certain TOS according to the following logic:

```
from net import Packet
from HIKe import allow, mon, slow, fast
__offset_ttl = 64

def chain1(tos: s16):
    cnt: u32 = mon(tos)
    if (cnt < 1000): fast()
    else: slow()
    return -1

def chain2(tos: s16):
    mon(tos)
    slow()
    return -1

def chain3(tos: s16):
    Packet.writeU8(__offset_ttl, 10)
    chain2(tos)
    return -1

def chain_tos():
    eth_type: u16 = Packet.readU16(12)
    tos: s16 = -1
    if (eth_type == 0x800):
        tos = Packet.readU8(16)
    return tos

def chain_main():
    tos: s16 = chain_tos()
    if (tos < 0):
        slow()
        return -1
    # IPv4 traffic, go ahead
    if (tos == 4):
        chain1(tos)
    elif (tos == 12):
        chain2(tos)
    elif (tos == 16):
        chain3(tos)
    elif (tos == 28):
        fast()
    else:
        allow()
    return -1
```

6 HIKE AND ECLAT PERFORMANCE

SS:
* descrivere gli obiettivi delle misura di performance
* cambiare l'ordine: mettere prima le misure che mostrano che va bene (HIKe real scenario) e poi quelle in cui, volendo spingere le prestazioni al massimo andiamo a valutare le limitazioni correnti della piattaforma
* individuare quello che si deve fare per migliorare
* mettere la descrizione del tesbed come ultima sottosezione
* ridurre il numero di cifre significative dei valori in tabella

6.1 Testing environment

We set up a testbed according to RFC 2544 [12], which provides the guidelines for benchmarking networking devices. Figure 8 depicts the used testbed architecture that comprises two nodes denoted as *Traffic Generator and Receiver (TGR)* and *System Under Test (SUT)* respectively. In our experiments,

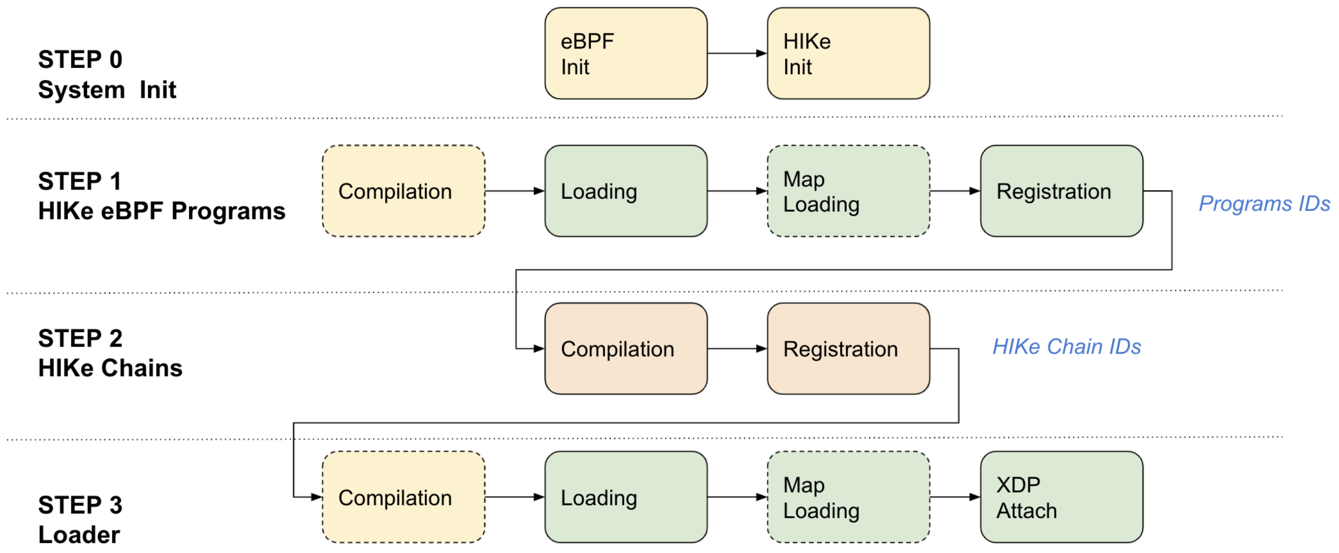

Figure 7: Steps required to set up a complete eBPF/HIKe system

Table 1: Example of the operations performed according to the different values of IP TOS field.

TOS	Operations
4	Mon + Fast Routing
12	Mon + Slow Routing (towards Kernel + TC)
16	TTL + chain2: (MON+SLOW) *(Does not recalculate the Checksum)
28	Fast Routing

Table 2: SUT Hardware characteristics

Type	Characteristics
CPU	2x Intel E5-2630v3 (8 Core 16 Thread) at 2.40 GHz
RAM	128 GB of ECC RAM
Disks	2x 1.2 TB HDD SAS 6Gbps 10K rpm 1x 480 GB SSD SAS 6Gbps
NICs	Intel X520 10Gb SFP+ Dual Port Intel I350 1Gb Dual Port

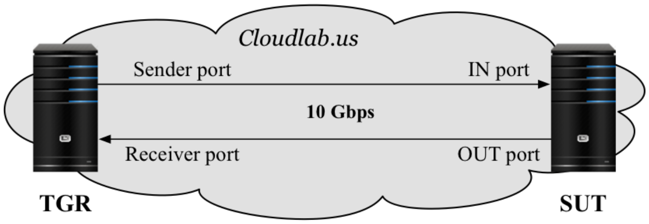

Figure 8: Testbed architecture

the packets are generated by the TGR on the Sender port, enter the SUT from the IN port, exit the SUT from the OUT port and then they are received back by the TGR on the Receiver port. Thus, the *TGR* can evaluate all different kinds of statistics on the transmitted traffic including packet loss, delay, etc. The testbed is deployed on the CloudLab facilities [11], a flexible infrastructure dedicated to scientific research on the future of Cloud Computing. Both the TGR and the SUT are bare metal servers whose hardware characteristics are shown in the table 2.

The SUT node runs a vanilla version of Linux kernel 5.10 and is configured as a specific node on this the SRv6 network. In the TGR node we exploit TRex [14] that is an open source traffic generator powered by DPDK [5]. We used SRPerf [1], a performance evaluation framework for software implementations, which automatically controls the TRex generator in order to evaluate the maximum throughput that can be processed by the SUT. The maximum throughput is defined as the maximum packet rate measured in Packet Per Seconds (PPS) for which the packet drop ratio is smaller then or equal to 0.5%. This is also referred to as Partial Drop Rate (PDR) at a 0.5% drop ratio (in short PDR@0.5%). Further details on

PDR and insights about nodes configurations for the correct execution of the experiments can be found in [1].

6.2 HIKe computational overhead

We evaluated the computational overhead of the HIKe framework by comparing it with a conventional eBPF tail call approach. To this aim we setup a simple "for cycle" benchmark program implemented with tail calls. The eBPF program reads an integer value i on a map and decrements it; then the program exits if $i = 0$ or tail call itself if $i > 0$. Conversely, the HIKe eBPF program implements the for cycle inside the HIKe VM. We measured the execution speed of both the approaches by counting the achieved packets per second (pps).

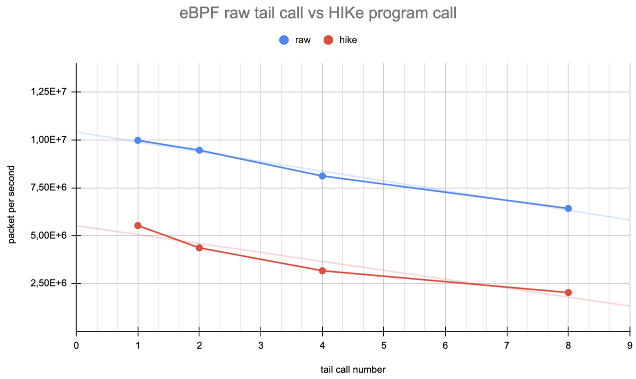

Figure 9: HIKe computational performance overhead

As we can see by figure 9 the overall throughput of both HIKe VM ("hike") and the eBPF programs ("raw") clearly decreases with the number of tail calls performed. The HIKe VM presents a performance degradation accounts for a $\sim$ 45% in case of 2 tail calls (10M pps vs 5.5M pps), and reaches $\sim$ 68% when 8 tail calls have been considered (6.4M pps vs 2M pps). This performance metric gives us the pure HIKe VM overhead, since the execution of the almost-empty eBPF program is minimal with respect to the overhead given by the HIKe VM (worst-case scenario). For this reason in the next subsection we evaluate the performance in a more realistic scenario.

6.3 HIKe Real Scenario

We considered a more realistic scenario of a DDoS mitigation program which is a popular application of eBPF XDP [2]. In this scenario we created a filtering program which marks packets coming from a denylist. We compared three basic approach: i) a conventional eBPF program (called "raw"); ii) an HIKe program (called "hyke"); and iii) a ipset [2] based approach (called "ipset").

Table 3 reports the achieved speed in terms of processed packets per second. As we can see, HIKe presents a $\sim$ 29.6% of performance degradation with respect to the conventional eBPF approach (1.8M pps versus 2.6M pps). However, HIKe outperforms ipset which performance is $\sim$ 45.8% worse than HIKe and $\sim$ 61.8% worse than the raw eBPF program. The HIKe overhead generally has a lesser impact on complex eBPF programs because of HIKe VM execution. The price to pay in terms of speed is however reasonable given the feature the framework provides in terms of fast composition, code re-usability and easiness-to-use.

6.4 eCLAT Performance

eCLAT performance does not affects the speed rate of packet processing, as eCLAT acts only on the transpiling, compilation and loading of HIKe programs and chains. In other words, it deals only with the "management" part.

However in the eBPF world, such management may happens at real-time, for instance in reaction of SDN operations which may load/unload eBPF chains and programs. For this reason we evaluated the performance of the main eCLAT features, reporting the results in table 4.

XXX rivedere numeri che sono sbagliati

7 HIKE & ECLAT PLAYGROUND

Nello sviluppo delle nostre soluzioni HIKe ed eCLAT abbiamo sviluppato due scenari utili a mostrare le potenzialità dell'architettura. Per verificare le perfomance in HIKe, già descritte nella sezione 6, è stato predisposto un sistema a due nodi mediante network namespace. Il primo nodo è un generatore di traffico ed ha lo scopo di fornire i pacchetti da processare con HIKe. Nel secondo nodo sono caricati i programmi HIKe e la HIKe VM. In figura **??** è mostrata l'architettura del sistema con i network namespace tra loro. L'interconnessione avviente con un coppia di virtual link

8 RELATED WORK

8.1 eBPF limitations and investigations

eBPF presents several limitations that has been tackled by researchers to provide mitigation or propose re-design. Miano et al. [4] analyzes the use of eBPF to create complex networking services, pinpointing the main technological limitations for specific use cases, such as broadcasting, ARP requests, interaction between control plane and data plane, identifying alternative solutions and strategies. They present a way to speed up programs by recompiling and loading them after

[2] https://ipset.netfilter.org/

	Raw eBPF	HIKe	Ipset
Av. Throughput (pps)	**2596417,31**	**1827420,55**	**990969,96**
Std. Dev. (pps)	2599,58	2978,99	1444,17

overhead w.r.t raw eBPF	-	29,62%	61,83%
overhead w.r.t. HIKe	-	-	45,77%

Table 3: Performance Comparison of DDOS implementation: Raw eBPF vs HIKe vs IPset

Table 4: eCLAT feature execution time

HIKe Chain Compilation Time	0.015 s
HIKe Two Chain Loading Time	0.285 s
HIKe One Chain Loading Time	0.211 s
HIKe eBPF Program Compilation Time	0.022 s
HIKe eBPF Program Loading Time	0.507 s

any change, to save the time to access to the maps. Gershuni et al. [6] analyse a design of eBPF in-kernel verifier with a static analyzer for eBPF within an abstract interpretation framework, to overcome the current verifier limitations. The authors' goal is to find the most efficient abstraction that is precise enough for eBPF programs and their choice of abstraction is based on the common patterns found in many eBPF programs with several experiments that were performed with different types of abstractions. One of the limitations, intrinsically present in eBPF, is related to the limited size of the programs, which can extend to a maximum of 4096 assembly instructions. This problem is overcomed by dividing the network function into several eBPF programs that call each other with tail calls, up to a maximum of 32 jumps. In [9], authors deals with the evaluation of the performance of tail calls between eBPF programs before and after the optimizations introduced to mitigate some vulnerabilities discovered, named Spectre and Meltdown. he comparison was made between two kernel versions: 5.4 and 5.5. Version 5.4 uses Retpoline for tail calls, which is the method introduced to mitigate vulnerabilities, while 5.5 converts tail calls into direct calls. They measure the duration of tail calls and then compare the execution time of a program when making numerous tail calls, showing that each tail call is 80% faster after the optimization in the kernel version 5.5.

8.2 eBPF frameworks and applications

There are several eBPF based projects aimed to bring code reusability/chaining and easyness-to-use. The most popular ones are three: Polycube, Cilium and Inkev.

Polycube aims to provide a common framework for network function developers to bring the power and innovation promised by Network Function Virtualization (NFV) to the world of in-kernel processing, thanks to the usage of eBPF [10]. Network functions in Polycube are called Cubes and can be dynamically generated and inserted into the kernel networking stack. Polycube enables the creation of arbitrarily complex network function chains, where each function can include an in-kernel dataplane and user space control plane with characteristics of persistence, isolation and composability. Differently from us, ...

Cilium is an open source software for transparently securing the network connectivity between application services deployed using Linux container management platforms like Docker and Kubernetes [3]. Differently from us, ...

InKeV is a network virtualization platform based on eBPF, an in-kernel execution engine recently upstreamed into linux kernel.

The following work [10] deals with Polycube, a software framework that aims to bring the power of NFV to applications that require in-kernel processing, ensuring greater flexibility and customization. Polycube allows us to arbitrarily create complex network function chains, where each function can include an in-kernel dataplane and user space control plane with characteristics of persistence, isolation and composability. Network functions in Polycube are called Cubes and can be dynamically generated and inserted into the kernel networking stack, without the need for custom kernels. Polycube aims to provide a common framework for network function developers to bring the power and innovation promised by NFV to the world of in-kernel processing, thanks to the introduction of eBPF. However eBPF was not created for this, so Polycube wants to cover the following features, not covered by the eBPF subsystem natively.
1) Common structure and abstractions of in-Kernel Network Functions
2) Programmable and extensible NF chaining
3) Simple Management and Execution of NFs
4) Simple Development of control and management plane
The authors show in the paper the characteristics of the framework, its performances and the advantages brought to the NFV world.

eBPF can also be used to give greater flexibility to the OVS Datapath. In [15] is addressed the possibility of using eBPF in OVS datapath development through two projects: OVS-eBPF project and OVS-AFXDP project.
1) OVS-eBPF project consists in rewriting the flow processing features, present in the OVS kernel datapath, in an eBPF

program and attaching them to Linux TC.
2) OVS-AFXDP project instead aims to adopt kernel bypass using AF_XDP and moving flow processing into user space.

In [15] is addressed the possibility of using eBPF in OVS datapath development through two projects: OVS-eBPF project and OVS-AFXDP project.
1) OVS-eBPF project consists in rewriting the flow processing features, present in the OVS kernel datapath, in an eBPF program and attaching them to Linux TC.
2) OVS-AFXDP project instead aims to adopt kernel bypass using AF_XDP and moving flow processing into user space.

8.3 SOA originale (archivio)

There are several projects and researches related to the area mentioned so far. Wanting to focus on the limitations of eBPF, it is necessary to mention some interesting articles in this regard, such as [4], which analyzes the use of eBPF to create networking services. The authors analyze the main limitations of the technology for specific use cases, such as broadcasting, ARP requests, interaction between control plane and data plane, identifying alternative solutions and strategies. They clearly show how the possibility to change the programs executed at runtime allows to optimize in an "aggressive" way the code that is executed. For each change (eg. a new rule inserted, a modification of an existing rule, etc.) the code is recompiled and loaded again, saving access to the maps. Another important limitation of eBPF is related to the in-kernel verifier; it is highlighted by the following article, [6] which describes the design of a static analyzer for eBPF within an abstract interpretation framework. The choice of abstraction is based on the common patterns found in many eBPF programs. To do this, the problem of verifying untrusted kernel extensions was considered. Many modern OSes get many of the features through dynamically loaded extensions that have support for I / O devices, file systems, networking etc. Extensions running in privileged CPU mode must be trusted by the system to make sure they do not contain malicious code. With eBPF the verifier takes care of ensuring that the code is secure, but this leads to some native problems. Therefore the authors of the paper have developed a verification algorithm for real world eBPF programs, which overcomes the current limitations of the verifier and based on an abstract domain and associated with abstract transformers that capture the relevant state of an eBPF program. The authors' goal is to find the most efficient abstraction that is precise enough for eBPF programs. Several experiments were performed with different types of abstractions. Using the Relational Domain, Zone Abstract Domain showed the best balance between speed and accuracy. One of the limitations, intrinsically present in eBPF, is related to the limited size of the programs, which can extend to a maximum of 4096 assembly instructions. This problem is overcomed by dividing the network function into several eBPF programs that call each other with tail calls, up to a maximum of 32 jumps. The following paper [9] deals with the evaluation of the performance of tail calls between eBPF programs before and after the optimizations introduced to mitigate some vulnerabilities discovered, named Spectre and Meltdown. The comparison was made between two kernel versions: 5.4 and 5.5. Version 5.4 uses Retpoline for tail calls, which is the method introduced to mitigate vulnerabilities, while 5.5 converts tail calls into direct calls. So some experiments were done using two benchmarks; the first to measure the duration of tail calls and then compare the execution time of a program when making numerous tail calls. They showed how each tail call is 80% faster if the retpolines are converted to direct calls (ie after optimization, kernel version 5.5). The second benchmark instead was used to send traffic on a chain of two large programs but with only one tail call between them and in this case the gain compared to version 5.4 was 5%. It is worth mentioning a work concerning the applications of eBPF in the networking field. The following work [10] deals with Polycube, a software framework that aims to bring the power of NFV to applications that require in-kernel processing, ensuring greater flexibility and customization. Polycube allows us to arbitrarily create complex network function chains, where each function can include an in-kernel dataplane and user space control plane with characteristics of persistence, isolation and composability. Network functions in Polycube are called Cubes and can be dynamically generated and inserted into the kernel networking stack, without the need for custom kernels. Polycube aims to provide a common framework for network function developers to bring the power and innovation promised by NFV to the world of in-kernel processing, thanks to the introduction of eBPF. However eBPF was not created for this, so Polycube wants to cover the following features, not covered by the eBPF subsystem natively.
1) Common structure and abstractions of in-Kernel Network Functions
2) Programmable and extensible NF chaining
3) Simple Management and Execution of NFs
4) Simple Development of control and management plane
The authors show in the paper the characteristics of the framework, its performances and the advantages brought to the NFV world. eBPF can also be used to give greater flexibility to the OVS Datapath. In [15] is addressed the possibility of using eBPF in OVS datapath development through two projects: OVS-eBPF project and OVS-AFXDP project.
1) OVS-eBPF project consists in rewriting the flow processing features, present in the OVS kernel datapath, in an eBPF program and attaching them to Linux TC.

2) OVS-AFXDP project instead aims to adopt kernel bypass using AF_XDP and moving flow processing into user space.

[Toward an eBPF-based clone of iptables] rifanno iptables con ebpf - vedere come comunicano con il programma (se hardcoded o via mappa) idea da vendere potrebbe essere i nostro prigramma loader

REFERENCES

[1] A. Abdelsalam et al. 2018. Performance of IPv6 Segment Routing in Linux Kernel. In *1st Workshop on Segment Routing and Service Function Chaining (SR+SFC 2018) at CNSM 2018, Rome, Italy*. IEEE, New York, US, 414–419.

[2] Gilberto Bertin. 2017. XDP in practice: integrating XDP into our DDoS mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, Vol. 2. The NetDev Society, Nepean, Canada, 1–5.

[3] The cilium project. 2021. BPF and XDP Reference Guide. available online at https://docs.cilium.io/en/latest/bpf/#bpf-and-xdp-reference-guide.

[4] Sebastiano Miano et al. 2018. Creating Complex Network Services with eBPF: Experience and Lessons Learned. In *IEEE International Conference on High Performance Switching and Routing (HPSR2018)*. IEEE, New York, US, 1–8.

[5] The Linux Foundation. 2021. Data Plane Development Kit (DPDK). https://www.dpdk.org/.

[6] E. Gershuni et al. 2019. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *PLDI 2019: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 1069–1084.

[7] Toke Høiland-Jørgensen et al. 2018. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. ACM, New York, 54–66.

[8] Jay Schulist, Daniel Borkmann, Alexei Starovoitov. 2021. Linux Socket Filtering aka Berkeley Packet Filter (BPF). https://www.kernel.org/doc/Documentation/networking/filter.txt

[9] Clement Joly and François Serman. 2020. Evaluation of Tail Call Costs in eBPF. In *Linux Plumbers Conference 2020*. The Linux Foundation, San Francisco, California, 14.

[10] S. Miano et al. 2021. A Framework for eBPF-Based Network Functions in an Era of Microservices. *IEEE Transactions on Network and Service Management* 18, 1 (2021), 133 – 151.

[11] Robert Ricci, Eric Eide, and the CloudLab Team. 2014. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE* 39, 6 (2014), 36–38.

[12] S. Bradner and J. McQuaid. 1999. *Benchmarking Methodology for Network Interconnect Devices*. RFC 2544. RFC Editor. https://tools.ietf.org/html/rfc2544

[13] D. Scholz et al. 2018. Performance Implications of Packet Filtering with Linux eBPF. In *2018 30th International Teletraffic Congress (ITC 30)*, Vol. 01. IEEE, New York, 209–217.

[14] TRex Team. 2021. TRex realistic traffic generator. https://trex-tgn.cisco.com/

[15] William Tu et al. 2018. Bringing the Power of eBPF to Open vSwitch. In *Linux Plumbers Conference 2018*. The Linux Foundation, San Francisco, California, 11.

[16] N. Van Tu et al. 2019. eVNF - Hybrid Virtual Network Functions with Linux eXpress Data Path. In *2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, New York, 1–6.