eBPF Programming Made Easy with eCLAT

Andrea Mayer*†, Lorenzo Bracciale*†, Paolo Lungaroni*†,
Pierpaolo Loreti*†, Stefano Salsano*†, Giuseppe Bianchi*†
*University of Rome Tor Vergata, †CNIT

***Abstract*—With the rise of the Network Softwarization era, eBPF has become a hot technology for efficient packet processing on commodity hardware. However the development of custom eBPF solutions is a challenging process that requires highly qualified human resources. In this paper we propose the *eCLAT* framework with the goal to lower the learning curve of engineers by re-using eBPF code in a programmable way. eCLAT offers a high level programming abstraction to eBPF based network programmability, allowing a developer to create custom application logic in eBPF with no need of understanding the complex details of regular eBPF programming. To support such modularity at the eBPF level, we created an eBPF library that implements a virtual machine, called HIKe VM. The HIKe VM library extends the conventional eBPF programs so that they can be integrated in eCLAT. The eCLAT/HIKe solution does not require any kernel modification.**

I. Introduction

Extended Berkeley Packet Filter (eBPF) [1] is a technology for packet processing in Linux/x86 nodes of datacenters, which has recently gained a prominent position among the solutions to improve the packet processing performance [2], [3], [4], [5]. eBPF has been successfully adopted for the development of Cilium [6], a leading framework for secure networking in the Kubernetes container orchestration platform, for the GKE Dataplane V2, and for Katran [7], a layer-4 load balancer open-sourced by Meta.

Developing eBPF programs is not easy. There are a few annoying limitations and issues in the eBPF architecture and development model that generate complexity, preventing a wider application of this technology and limiting the advantages that eBPF could bring [8], [9]. In particular, eBPF programs need to be *verified* by the kernel before being loaded, this process is very annoying for the developer [10] and it can increase the development time with a significant loss of productivity.

In this work, we propose an approach where "small" and independent eBPF programs can be easily arranged together to build complex workflows, without changing their source code but just composing them together in a programmable way. Using a Unix similarity is like having many standalone programs such as cut, tail, grep, sed, and using bash scripts to compose them for a wide range of specific application needs.

The proposed approach is called eCLAT (eBPF Chains Language And Toolset). eCLAT offers a python-like scripting language for composing eBPF programs. An example of an eCLAT script, which we call *chain*, is shown in Listing 1. In the eCLAT scripts it is possible i) to define variables; ii) to implement looping/branching operations, and iii) to execute independent eBPF programs (highlighted in green in Listing 1).

```
flow_rate = flowmeter(packet)
#drop flows greater than 10Mbps
if flow_rate > 10:
    droppacket()
else:
    allowpacket()
```

Listing 1: Example of an eCLAT *chain*. Inside a chain, eBPF programs are called as they were functions, allowing an easy and flexible programming of application logic.

In our vision, there will be two types of developers: i) the expert *eBPF developers*, a minority of developers highly skilled in eBPF programming that can develop the eBPF components; ii) the *eCLAT developers*, the large majority that writes eCLAT scripts using the eCLAT language and toolkit to compose the custom applications. We believe that the high-level python-like abstraction offered by eCLAT greatly simplifies the learning curve for developers. System administrators accustomed to command line tools can easily become eCLAT developers and benefit from the power and the speed of eBPF, without the difficulties of becoming eBPF programmers. Using the eCLAT framework, a large number of novice programmers can implement complex application logic exploiting the eBPF powerful capabilities. Also the expert developers can benefit from using the eCLAT scripts because it can boost their productivity when a given problem can be solved by combining existing eBPF programs.

In order to turn our vision to reality, we need a way to execute the eCLAT Chains. We have designed a Virtual Machine (VM) abstraction called HIKe (HIKe stands for Hide, Improve and desKill eBPF). The HIKe VM is the execution environment for the eCLAT Chains and it is developed as an eBPF library. The eBPF programs that are composed inside a chain (as shown in Listing 1) are more properly referred to as *HIKe eBPF programs* or *HIKe programs* for short. In fact, an eBPF program needs to be (easily) extended with proper "calls" to the HIKe VM library in order to be used inside the eCLAT framework. A HIKe program can be called as a function inside an eCLAT chain and it will be executed in the HIKe Virtual Machine.

The contributions of the paper are as follows: i) we propose eCLAT as the first framework enabling the composition of precompiled and pre-verified eBPF code ii) we designed a Python-like scripting language that is transpiled in bytecode for implementing network eBPF applications iii) we support

978-3-903176-51-5 © 2022 IFIP

the composition of programs at eBPF level with a VM abstraction called HIKe VM iv) we devised a distributed architecture for the eBPF code management integrating a packet manager for the eCLAT framework.

The paper is organized as follows: in Section II we introduce eBPF and discuss some shortcomings, then in Section III we present the eCLAT framework and scripting language. Section IV provides some implementation insights, while Section V discusses the evaluation of the proposed solution. We present the related work in Section VI and finally conclusions are drawn.

II. Background: eBPF shortcomings

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 mention a few bugs that can affect this tool chain). The learning curve of this technology is very steep and needs continuous coaching from experts. In this section, we provide an overview on the eBPF technology, then we discuss some shortcomings.

A. eBPF overview

The extended Berkeley Packet Filter (eBPF) [11] 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 Fig. 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. The verification step assures that the program cannot crash, that no information can be leaked from the kernel to the user space, and it always terminates. 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 [12]. After the verification, JIT (Just In Time) compilation translates the eBPF bytecode into the specific instruction set for a given architecture (i.e. x86, arm, 64 or 32 bits).

Once loaded, 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. 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.

Focusing on packet processing, eBPF programs can be attached to different hooks and packets trigger their execution.

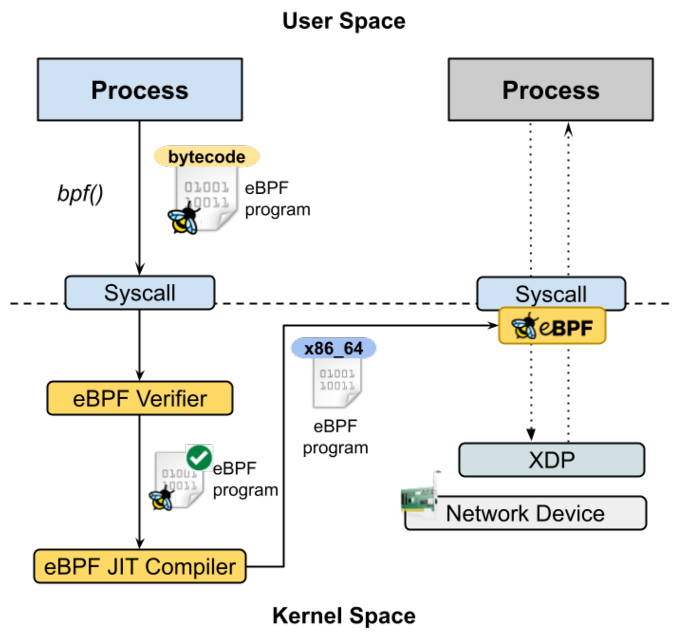

Fig. 1: eBPF programs compilation and verification

Among these hooks, we focus for our purposes on the so-called eXpress Data Path (XDP) hook. XDP [13] 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 structure, foremost the `sk_buff`. This accounts for most performance benefits as widely demonstrated in the literature (e.g., [14] [15] [16] [17]).

B. The verification hell

The verification phase is the one creating major issues in the eBPF programming model. The kernel validation approach is almost adequate for simple eBPF programs, i.e. few instructions, loop-free code, and no complex pointer arithmetic, while it has been shown to be a very tough obstacle to the development of complex applications [9]. As analyzed in [8], there are four main issues: i) the verifier reports many false positives, forcing developers to insert redundant checks and assume quite contrived programming solutions; ii) the verifier does not scale to programs with a large number of logical paths (i.e.: nested branches); 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 offered to the verification step is the results of the *optimization* procedures executed by the compiler/optimizer. For optimization reasons, the compiler/optimizer can change the sequence of operations (preserving the correctness of the program) with respect to the C source code and this can violate

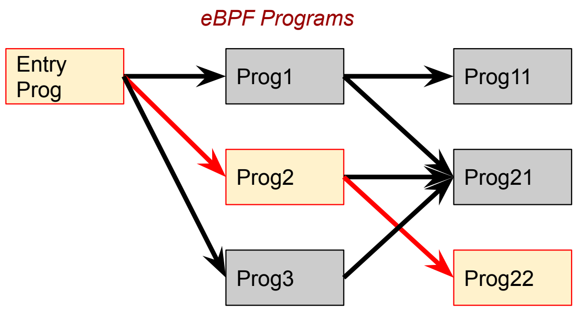

Fig. 2: Chaining eBPF programs with tail calls

some constraint that must be checked by the verifier. The final bytecode obtained from the compilation of an eBPF program often depends on the version of the used compiler/optimizer (toolchain). Different versions of the compiler/optimizer may not produce identical bytecode from the standpoint of the individual instructions used as well as their order. At the same time, the eBPF verifier evolves as new features are added in the later releases of the kernel. All of this can affect the possibility that a given program compiled with a specific version of the toolchain is correctly verified on one specific kernel version but is rejected on another one.

C. Poor program composition abstractions in eBPF

In the eBPF framework, a program can call another program using the *tail call* approach. Figure 2 provides an example of 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 execution control is handed over to the *callee* eBPF program and the processing continues along all the *calling tree*. With a tail call, the flow of execution is passed to the called program and the execution context of the calling program is terminated. There is no possibility of executing a program and receiving return parameters to be processed.

If the logic to concatenate the programs needs to be changed, the affected eBPF programs need to be recompiled (and pass the verification). For example, this strategy is implemented by the state-of-the-art eBPF composition framework called Polycube [18].

Very recently, the possibility of invoking eBPF global functions that are verified independently has been introduced in the eBPF model. The use of eBPF global functions can simplify the work of an expert eBPF developer, easing the code reuse, but it requires in any case the verification of the eBPF program that contains the calls to the global functions. In other words, the eBPF global functions do not fulfill the requirements that led us to design and develop the eCLAT solution.

D. Lack of a package manager

Considering the limitations in the reuse and composition of eBPF programs, there is no surprise that there is not a package manager tool for eBPF, with functionality similar to pip (for Python modules) or to npm (for Javascript packages). The eBPF package manager should facilitate the reuse and the management of eBPF components, promoting the development of an eBPF ecosystem.

III. The eCLAT abstraction

eCLAT (eBPF Chains Language And Toolset) is a framework that offers a high level programming abstraction to eBPF. Such abstraction is implemented with a python-like scripting language, called eCLAT Scripting Language, devised for composing eBPF programs and configuring eBPF maps. The overall goal is to lower the learning curve of engineers, allowing them to re-use eBPF code in a programmable way. This is possible thanks to a modular approach where eBPF programs are modules and can be composed arbitrarily to implement flexible application logic. To support such modularity at the eBPF level, we have designed a virtual machine, called HIKe VM that is able to execute the eCLAT programs, called *Chains*, as discussed in the next subsections.

A. The eCLAT Scripting Language

In the eCLAT framework, the complex operations and heavy lifting must be done *within* eBPF programs which are programmed in C by experts and stand "outside" eCLAT. Within eCLAT is it possible to call such eBPF programs as they were conventional *functions*, passing to them some input values (arguments) and receiving from them their return value. Such values can be used in the eCLAT script to define the application logic in a simple but *programmable* way.

To this aim, the eCLAT scripting language supports the definition of variables, arithmetic operations, branching/looping conditions and *function calls* which, as we said, masquerade the call to eBPF programs. The eCLAT language also offers the capability of importing modules, interacting with a package manager. An example of an eCLAT script is shown in Listing 2. The full specification and the formal definition of the eCLAT language grammar are in the documentation [1].

An eCLAT script is processed by the eCLAT framework (in particular by the eCLAT deamon), performing the following operations:

1. the package manager fetches the eBPF programs that are imported
2. the code of the eCLAT script is transpiled to C language
3. the C code is compiled in bytecode for the HIKe Virtual Machine
4. the bytecode is stored in eBPF maps ready to be executed

B. Architecture and modularity

The three main elements of the eCLAT framework are: *Chains, loaders and HIKe programs*.

- **Chains**: an eCLAT script is organized in chains which are defined like python function definitions. A chain is a simple set of instructions to be executed on a packet

[1] https://hike-eclat.readthedocs.io/en/latest/eclat_doc.html

and contains the application logic. In the code listing 2, the chain `ddos_tb_2_lev` is defined in lines 10-45. eCLAT transpiles the code of each chain to generate the bytecode which will be executed by the HIKe Virtual Machine.

- **Loaders**: a loader is an eBPF program which is responsible for calling the execution of a specific chain. For instance in the code listing 2, the loader `ip6_sc` intercept all the IPv6 packets and for each packet call the chain `ddos_tb_2_lev`. eCLAT gives the possibility to configure each loader. For instance the `ip6_sc` loader can be configured to call different chains according to the destination IPv6 address of the received packet.
- **HIKe Programs**: a HIKe program is an eBPF program which can be called in an eCLAT chain. The chain can pass some arguments to the program which in turn can return a value. Such programs are conventional eBPF programs which must be slightly adapted (few lines needed) to be executed inside the HIKe Virtual Machine. Usually, each HIKe program is devised to do one job such as count packets in a flow, classify, blacklist an IP address or encapsulate. In the code listing 2 there are 8 different HIKe programs used such as `l2_redirect`. It is important to note that eCLAT is not an eBPF code composer: all the programs called from a chain are compiled and loaded *separately and independently* in memory, to avoid verification problems given by the creation of a monolithic “large” eBPF program. Nicely, no change in the code of the HIKe programs is required when they are composed.

Chains, loaders and programs are stored on public repositories together with their documentation. In this way, the package manager of eCLAT can automatically download the imported programs on demand. eCLAT programmers can view the catalog of available eBPF programs (organized in packages) and consult their documentation [19].

Fig. 3 shows the architectural view of eCLAT where we can see eCLAT as composed by a daemon and a client line interface (CLI). Section IV discusses the internal architecture of the daemon and the CLI.

C. The HIKe Virtual Machine (HIKe VM)

To make eCLAT possible, it is necessary to find a way to compose (pre-compiled and pre-verified) eBPF programs using the *function call* pattern and without the hassles of the verification phase. Unfortunately, these cannot be done with the current eBPF framework. The idea we propose is to decorate eBPF programs with specific code that allows them to be composed in a *chain* specified with eCLAT. The logic of the chain will be interpreted at run time, for this purpose we designed a lightweight Virtual Machine abstraction, named HIKe VM. The HIKe VM is a library of eBPF code that is executed by the eBPF programs that are composed in the chain. In what follows, we will refer to the decorated programs as HIKe eBPF programs or simply HIKe programs. The HIKe VM is designed as a register-based Virtual Machine using a subset of the eBPF VM 64-bit RISC instruction set. The eCLAT scripts are transpiled into a bytecode that is loaded in memory (through eBPF maps). This bytecode is interpreted by HIKe VM which fetches, decodes and executes the instructions. The bytecode codifies logical and arithmetical instructions, jump instructions to control the program flow, calls to HIKe Programs and also calls to other chains. At run time, the HIKe VM counts the number of instructions of the chain that are executed. When the number of executed instructions exceeds a configurable threshold (e.g. 64 instructions) the processing is stopped (and the associated packet is dropped). In this way, kernel blocking is avoided with a “run-time” check by the Hike VM and not with a static analysis of code sanity as done by the eBPF verifier.

The HIKe VM provides also a set of helper functions (e.g., for packet handling) which are in turn made available to eCLAT programmers. A deeper technical discussion on the HIKe VM can be found in [20], full details are provided in [21]. An earlier version of the HIKe VM was used in [22].

```
1  from prog.net import hike_drop, hike_pass, \
2       ip6_hset_srcdst, ip6_sd_tbmon, monitor,\
3       ip6_dst_tbmon, ip6_sd_dec2zero, l2_redirect
4  from loaders.basic import ip6_sc
5
6  # send all IPv6 packets to our chain
7  ip6_sc[ipv6_sc_map] = { (0): (ddos_tb_2_lev) }
8  ip6_sc.attach('DEVNAME', 'xdp')
9
10 def ddos_tb_2_lev():
11     PASS=0; DROP=1; REDIRECT=2;
12     REDIRECT_IF_INDEX = 6;
13     ADD=1; LOOKUP=2;
14     BLACKLISTED = 0;
15     IN_PROFILE = 0;
16
17     # (src,dest) in blacklist ?
18     u64 : res = ip6_hset_srcdst(LOOKUP)
19     if res == BLACKLISTED:
20         # redirect one packet out of 500
21         res = ip6_sd_dec2zero(500)
22         if res == 0:
23             monitor(REDIRECT)
24             l2_redirect(REDIRECT_IF_INDEX)
25             return 0
26
27         monitor(DROP)
28         hike_drop()
29         return 0
30
31     # check the rate per (dst)
32     res = ip6_dst_tbmon()
33     if res != IN_PROFILE:
34         # check the rate per (src,dst)
35         res = ip6_sd_tbmon()
36         if res != IN_PROFILE:
37             # add (src,dest) to blacklist
38             ip6_hset_srcdst(ADD)
39             monitor(DROP)
40             hike_drop()
41             return 0
42
43     monitor(PASS)
44     hike_pass()
45     return 0
```

Listing 2: eCLAT script for DDoS mitigation

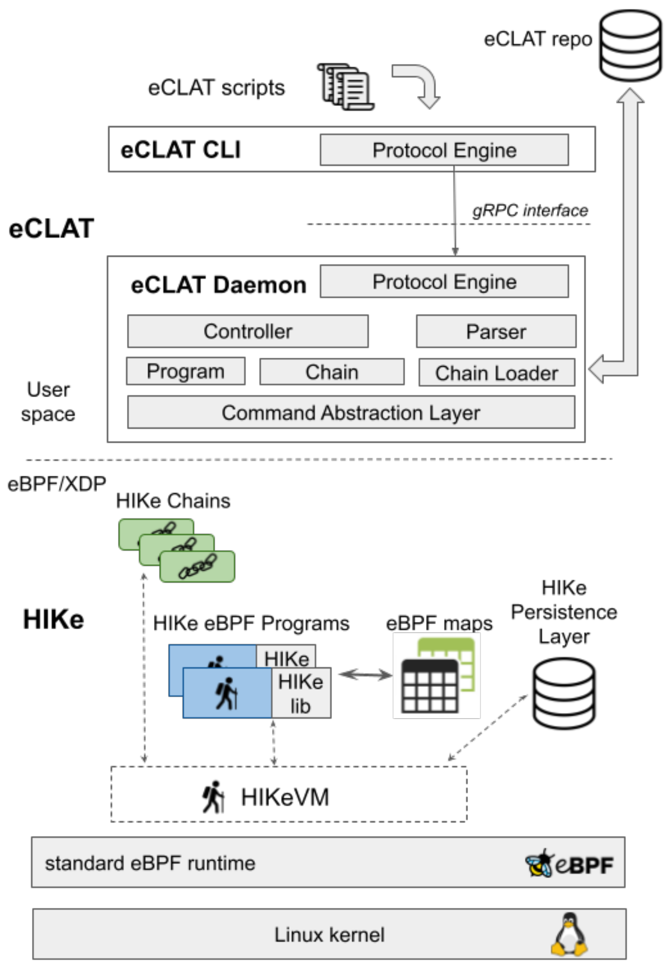

Fig. 3: eCLAT Overall Architecture

D. An eCLAT Script Example

DDoS mitigation is a popular application of eBPF on XDP [23]. Let us consider the following packet processing logic that a developer wants to implement using eCLAT.

> If the packet rate for any IP destination D is over a threshold R1, analyze all IP sources S_{any} that are sending packet for this "overloaded" IP destination D.
> If an IP source S in S_{any} is sending packets with a packet rate over a threshold R2, put the IP (S, D) in a blacklist for a duration of T seconds.
> During this interval T, drop all packets in the blacklisted (S, D) couple and send a sample of the dropped packets (e.g., one packet every 500 packets) to a collector.

Implementing this logic for a non-skilled eBPF programmer is not easy. Using eCLAT, the non experienced programmer can write a script like the one shown in Listing 2.

Specifically, the script `ddos_tb_2_lev` (DDoS with two levels token bucket) uses and combines in a custom way 7 different eBPF HIKe programs, which are imported in lines 1-3. The Chain loader is called `ip6_sc` (line 4) and it selects all the IPv6 packets. The Chain loader is configured in line 10, which binds the Chain `ddos_tb_2_lev` to the classifier.

In line 11 the `ip6_sc` is attached to the `XDP` hook of `eth0` interface.

The logic of the `ddos_tb_2_lev` chain is defined starting from line 13, as follows.

1 call the `ip6_hset_srcdst` program with a parameter (`LOOKUP`). The result is 0 if the IPv6 (src, dst) is blacklisted.
2 if the packet is blacklisted, send one packet every 500 to an interface that collect packet samples and drop the others (line 14); count the REDIRECT and the DROP events
3 if the packet is not blacklisted, check the IPv6 destination against a token bucket. If the rate is out of profile for the token bucket (per destination), check the IPv6 (source, destination) against another token bucket. If the rate of the (source, destination) flow is out of profile, put the (source, destination) flow in the blacklist by calling again the `ip6_hset_srcdst`, this time with parameter ADD, and then drop the packet (increasing the DROP events counter.
3 if the packet is not out of the profile, increment a counter of the passed packets and exit the eBPF program by handing the packet to the regular kernel processing.

eCLAT scripts support branching and looping instructions (if, for, while, although in the limits set by the HIKe VM and eBPF verifier), and simplify the operations to read/write packets (resolving the endianness automatically). Variables are typed, using the Python syntax for Syntax for Variable Annotations (PEP 526) [24]. The data returned by Chains and Programs are 64 bit long but can be cast to shorter subtypes.

As we can see already by this simple example script, eCLAT provides the flexibility to define custom application logic in an easy way, by reusing different standalone HIKe eBPF programs as they were Python functions.

IV. Implementation

eCLAT has been implemented in Python as a daemon (*eclatd*). The eCLAT daemon receives user commands from a CLI (*eclat*) through a gRPC interface. The structure of the data is described through a protocol buffer language [25]. Through the CLI, users can load an eCLAT script which instructs the daemon to i) import all necessary HIKe eBPF Programs by collecting their code, compile, inject and register them to the HIKe VM; ii) translate the high-level code of the chain in C language, compile and load them in the HIKe VM; iii) manage the entry point (chain loader) by retrieving its code, compile, inject and configure according to custom parameters. The daemon is needed to assign run time IDs to HIKe programs and chains and to use these IDs when compiling/linking the chains. The daemon keeps the state of eCLAT consistent and avoids concurrency issues in the loading of programs and chains. The eCLAT CLI allows users to query the daemon about the current status of eBPF maps.

As shown in Fig. 3, the eCLAT daemon is composed by the following functional blocks:

- **Protocol engine**: implements the gRPC protocol service and is responsible for the communication with the CLI;
- **Controller**: is responsible to set up the networking environment, to interact with the parser and to execute the scripts invoking the managers. It generates/retrieves IDs for HIKe eBPF Programs and HIKe Chains. Such identification numbers will be fundamental for the chain compilation phase since the HIKe Chains rely on numerical IDs for calling HIKe eBPF Programs, rather than on the names which are used in the eCLAT domain;
- **Program**: wraps and manages a HIKe eBPF Program. The component fetches programs from the eCLAT public repository, compiles them, and takes care of the loading and unloading operations. Finally, it registers the output in the HIKe Persistence Layer. During the compilation of the HIKe eBPF Programs, the debug info about the program (i.e.: variables, functions, structs, etc.) are automatically extracted and registered in a JSON file. This file is parsed to obtain all map/program associations as well as the number of input parameters accepted by the specific HIKe eBPF Program;
- **Chain**: handles the script part related to HIKe Chains. It is in charge of translating the source code, from the (python-like) eCLAT script to a C-defined HIKe Chain. Then compiles it to generate artifacts (i.e. ELF file object) through the execution of a dedicated Makefile. Finally, it registers the output in the HIKe Persistence Layer. The HIKe Persistence Layer contains a catalog between all the HIKe Chains loaded (and thus their bytecodes) and the Chain IDs assigned by the eCLAT Runtime Environment;
- **Chain Loader**: this component handles one or more HIKe Chain Loader(s) and interacts with their maps. Using the eCLAT scripting language, users can specify the chain loader that has to be loaded, attached to the XDP hook as well as the configuration that has to be enforced through configuration maps;
- **Parser**: has the task of analyzing the eCLAT scripts and creating the Abstract Syntax Tree (AST), in order to interpret the provided commands and generate the C code which defines the HIKe Chains;
- **Command Abstraction Layer**: provides an abstraction over the different shell commands that need to be invoked on the operating system to deal with eBPF / HIKe.

The eCLAT daemon automatically fetches the required programs from the **Package Repository**. The repository contains *packages* which in turn may contain different programs, chains, or chain loaders. Few examples of programs are shown in Table I, the full list is in [19].

When a user wants to execute an eCLAT script the flow is the following. The eCLAT Daemon receives the scripts from the eCLAT Chains described in the eCLAT language. The daemon first download the required code from the eCLAT Repo and then "transpiles" the eCLAT chain code into into C language, generating the source code of HIKe Chains, which is then compiled into a executable format suitable for being loaded and executed by the HIKe VM. Actually this is not only a compilation operation, because the eCLAT Daemon also works as a *linker*: it resolves the references to HIKe eBPF Programs and to other HIKe Chains called inside a Chain and writes the HIKe eBPF Program IDs and Chain IDs into the bytecode. Moreover, the eCLAT daemon manages the dynamic compilation, verification and loading of the HIKe eBPF Programs that are referred in the Chains. In fact, when a HIKe Chain refers to a HIKe eBPF Program, the eCLAT daemon checks if that program is already loaded and if not, it loads it. The executable of a HIKe Chain (i.e. the bytecode with some additional info) is stored by the eCLAT Daemon in the HIKe Persistence Layer, which is based on eBPF maps. The eCLAT Daemon also interacts with eBPF maps in the HIKe layer, that are used by the HIKe eBPF Programs to read/write information. The HIKe layer provides the Runtime Environment for executing the bytecode of the HIKe Chains.

V. Evaluation of the Solution

A. Prototype

We have implemented a full prototype, running on a single Docker container [26]. Inside the prototype, it is possible to develop and test HIKe eBPF programs and eCLAT Chains. In particular, we emulate a node implementing the eCLAT framework and a node that generates traffic to be processed. We provide a number of HIKe eBPF packages and programs (see examples in Table I) and demonstrate how they can be easily combined in eCLAT Chains to implement fairly complex packet processing scenarios (like the DDoS example coded in Listing 2). The technical documentation and the instructions to replicate the experiments are available at [19].

B. Modularity

The greatest benefit of adopting the eCLAT framework is in the flexibility and modularity it offers. Table II objectivize the benefits of this approach by comparing the presented solution with popular frameworks, Cilium and Polycube, across different dimensions, and specifically:

- **Application logic definition**: how an eCLAT user can define/implement a custom application logic? eCLAT allows users to define their business logic in a programmable way through eCLAT scripts. Conversely, other frameworks allow defining custom configuration. The difference is that programming flows allow much more expressibility than relying on a pre-defined set of parameters to configure;
- **Composition topology**: which topology of the data pipeline is supported by the framework? eCLAT supports arbitrary topology as the data flow can follow different branches and loops. Other frameworks like Polycube are limited by a linear topology: packets flow through a predefined set of eBPF Programs which are connected over through a set of *tail calls*. If developers want to implement a custom calling logic with specific interaction

HIKe eBPF Program	Package Name	Description
`ip6_dst_meter`	meter	Counts the packets per IPv6 destination
`ip6_sd_tbmon`	meter	Token bucket monitoring per IPv6 (source, destination) couple
`ip6_sd_dec2zero`	sampler	Implement a counter-to-zero per IPv6 (source, destination) couple
`show_pkt_info`	info	Print debug information about a packet
`ip6_alt_mark`	alt_mark	Decode the Alternate Mark TLV in the Hop-by-hop Options Extension Header

TABLE I: Examples of HIKe programs available in the package repositories.

TABLE II: Comparison of the modularity features for different eBPF frameworks

Dimension	**Cilium**	**Polycube**	**eCLAT**
Application logic definition	configuration and API	configuration of modules and topology	programmatic
Composition approach	assembling and compiling different building blocks	interconnection of cubes through ports (e.g., veth pairs)	dynamic composition of eBPF programs with no recompilation.
Composition topology	-	linear (tail call)	arbitrary
Code generation	BCC-based	BCC-based	transpiled from eCLAT script to C, and compiled with CLang/LLVM
Modularity	pre-defined programs	big modules (cubes)	any eBPF program
Extensibility	submit a patch to the main project	creation of a new cube within the framework	conventional eBPF programs with minor modifications

patterns (i.e. a program calls another program accordingly to given conditions), they must do it on their own;

- **Composition approach**: where the composition of different modules happens? eCLAT is the only one which permits composition *inside* eBPF, without requiring any eBPF Programs (modules) recompilation. This is different from models where the composition happens in user space and then, through code generation, eBPF programs bytecode is injected;
- **Code generation**: differently from others, eCLAT is *not* based on BCC [27] but on CO-RE [28] which is fostered and maintained by the Linux kernel community;
- **Modularity**: What is a module? for Cilium there are pre-defined generated programs, and Polycube relies mainly on “big” modules (i.e. “the firewall”) as they can be only chained together. Conversely, for eCLAT a module is a standalone HIKe eBPF Program that can be also quite small (i.e. “flow meter”) as its utility must not be absolute, but functional of the context where it will be placed in the HIKe Chain (i.e. in an *if* expression to decide a branch);
- **Extensibility**: How can an expert eBPF programmer create a new module to extend the framework? HIKe/eCLAT module can be any legacy eBPF program with very few changes (3 or 4 lines of C needs to be added). Extending other frameworks requires more skills.

C. Dataplane performance

In order to evaluate the dataplane performance of the eCLAT framework, we defined a benchmark with a set of processing operations. We called this benchmark MMLF (Match, Mark, Lookup and Forward). We assume that a node is forwarding packets and needs to identify the packets that belong to a *blacklist* of source IP addresses. The packets in the blacklist have to be marked with a given IP TOS. After the classification and marking the packets are forwarded with a lookup in the routing table. The classification and marking operations add a processing burden to the normal forwarding operations, the obvious goal is to keep this burden as low as possible. We developed and compared three solutions: i) an IP set [29] based approach (**IP Set**), ii) a chain of HIKe eBPF programs (**HIKe**); and iii) a conventional eBPF program (**eBPF**). According to our experience, the conventional **eBPF** solution is the most difficult to be programmed, only the expert eBPF developers can do it. The **HIKe** solution is simpler and it can be programmed by the eCLAT developers. The **IP Set** solution has an intermediate development complexity.

Our performance evaluation consists in measuring the maximum forwarding throughput of a node executing the MMLF benchmark, for the three solutions (IPset, HIKe and raw eBPF). The maximum forwarding throughput (R_{max}) is defined as the maximum packet rate (measured in kilo packets per second) for which the packet drop ratio is smaller than or equal to 0.5%, according to the methodology reported in [30].

In the experiments, we considered two types of packets: i) packets that need to be marked; ii) packets that do not need to be marked (their source address is not in the blacklist). For reference, we have evaluated in the same conditions of our experiment the maximum forwarding throughput of plain IPv6 forwarding performed by the Linux kernel (**Plain**). Table III reports the R_{max} (averaged over 30 experiments). To give evidence of the reliability of the measurements, we report the Coefficient of Variation (the Standard deviation divided by the average value).

	IP set	**No match**	**Plain**	**HIKe**	**eBPF**
R_{max}	**0.99**	**1.32**	**1.39**	**1.87**	**2.57**
Cv	0.14%	0.10%	0.10%	0.21%	0.08%

TABLE III: eCLAT Dataplane Performance: R_{max} in Mpps

The R_{max} for the IP Set solution is 0.99 Mpps (for packets that need to be marked). The reference R_{max} for the plain

IPv6 forwarding operation (with a lookup in the routing table) in the Linux kernel is 1.39 Mpps. The degradation accounts for the cost of classification and marking using the IP Set framework. We observe that the throughput of the HIKe based solution is 1.87 Mpps (for packets that need to be marked), with an increase of performance of 88% (a factor 1.88x) with respect to the IP Set solution. The HIKe solution performs the lookup in the kernel routing tables by using an eBPF helper function. HIKe is faster than the plain IPv6 forwarding in the kernel, despite the fact that it also performs the classification and the marking in addition to the route lookup. This is because it benefits from the advantages of XDP/eBPF processing compared to regular Linux kernel. As expected, the custom eBPF program achieves the highest throughput for packets that needs to be marked (2.57 Mpps), at the price of requiring expert eBPF programming skills. The second column (**No match**) reports the R_{max} for packets that do not need to be marked, which is the same for the HIKe and eBPF solutions. In this case, only the initial (unsuccessful) match operation is performed and then the packet is left to the kernel for the regular processing. The R_{max} is 1.32 Mpps, only a 5% reduction with respect to plain IPv6 forwarding. This result shows that the performance penalty introduced by the initial classification made by XDP/eBPF is minimal.

VI. Related Work

eBPF has been extensively used for building fast and complex applications in several domains such as tactile [31], security [32], cloud computing [33] and network function virtualization [34]. In what follows, we limit our analysis on the limitations of the system and on the relevant framework.

A. eBPF limitations and investigations

eBPF provides advantages to network programmers but it also presents several limitations that have been highlighted by researchers and often tackled to provide mitigation or propose re-design. A comprehensive review of eBPF technology opportunities and shortcomings for network applications is provided in Miano et al. [12] that analyzes the use of eBPF to create complex services. The authors pinpoint the main technological limitations for specific use cases, such as broadcasting, ARP requests, interaction between control plane and data plane, and when possible they identify alternative solutions and strategies. Some of the problems reported in [12] are part of the motivations which led us to the design and development of HIKe and eCLAT. Gershuni et al. [8] analyze 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. We also recognize the relevant role of the "validation hell" and we believe that the HIKe architecture can help to mitigate the problem.

B. eBPF frameworks for networking

There are several eBPF based projects and frameworks devoted to simplify or manage the networking using eBPF. The most popular ones are three: Polycube, Cilium and Inkev. *Polycube* aims to provide a 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 [18], [35]. Network functions in Polycube are called Cubes and can be dynamically generated and inserted into the kernel networking stack. Like us, Polycube is devoted to implement complex systems through the composition of cubes. However, Polycube's goal is not to reconstruct functional programming but to build chains of independent micro-services. The absence of function calls does not allow eBPF programs to return values or accept input arguments, and thus it is not possible to change the flow logic according to the output of a given program.

We have been inspired by the work [22] where eBPF programs can be chained but our ideas of the HIKe VM and of function calls are missing.

Another approach for using eBPF inside the NFV world is provided by Zaafar et al. with their InKeV framework [36]. *InKeV* is a network virtualization platform based on eBPF, devoted to foster programmability and configuration of virtualized networks through the creation of a graph of network functions inside the kernel. The graph which represents the logic flow, is loaded inside a global map. The logic implemented by the graph is merely related to the function composition, while we provide a more complex flow within the HIKe VM (e.g., branch instructions, loops, and in general programmable logic). Such as for Polycube, the goal of InKeV is to provide network-wide in-kernel NFV, which is not our framework main goal but that can certainly be one of the most important applications of it.

Cilium is an open source application of the eBPF technology for transparently securing the network connectivity between cloud-native services deployed using Linux container management platforms like Docker and Kubernetes [1]. With respect to this work, Cilium has a totally different target as it is focused on the security of applications running in containers. Conversely, our target is the reusability of different eBPF programs and their composability inside the chains, separating the composition logic flow from the eBPF (HIKe) programs themselves. We think big applications like Cilium could greatly benefit from this new approach.

Risso et al. proposed an eBPF-based clone of iptables [37]. The approach uses an optimized filtering based on Bit Vector Linear Search algorithm which is a reasonably fast and consolidated programming interface based on iptables rules. Clearly, the focus of the work is not composability, but an extended version of such an approach could be used to define the entry point for the HIKe applications.

It is worth mentioning the application of eBPF to provide a greater flexibility to Open vSwitch (OVS) Datapath [38], [39]. The works propose to move the existing flow processing features in OVS kernel datapath into an eBPF program attached to the TC hook. Finally, several authors implement eBPF Hardware Offload to SmartNICs [40], [41].

VII. Conclusions

eCLAT simplifies the creation of complex eBPF applications by providing a scripting language for implementing custom application logic. With eCLAT it is possible to mesh up eBPF programs, seen by the eCLAT script developers as "simple" function calls. Each of these programs can be reused in several different application contexts with no code change needed. A Virtual Machine built inside eBPF and called HIKe VM takes care of the runtime composition with a minimal overhead. As further extension, we are considering the possibility to "push" eCLAT chains in remote nodes to achieve network programmability. Both HIKe and eCLAT frameworks are available under a liberal open source license, the pointers to the source code are in [19].

References

[1] The Cilium project, "BPF and XDP Reference Guide," 2021, available online at {https://docs.cilium.io/en/latest/bpf/\# bpf-and-xdp-reference-guide}.

[2] M. Bertrone, S. Miano, F. Risso, and M. Tumolo, "Accelerating linux security with ebpf iptables," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, 2018, pp. 108–110.

[3] F. Parola, S. Miano, and F. Risso, "A proof-of-concept 5g mobile gateway with ebpf," in *Proceedings of the SIGCOMM'20 Poster and Demo Sessions*, 2020, pp. 68–69.

[4] W. Tu, J. Stringer, Y. Sun, and Y.-H. Wei, "Bringing the power of ebpf to open vswitch," in *Linux Plumbers Conference*, 2018.

[5] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Muller, "Bmc: Accelerating memcached using safe in-kernel caching and pre-stack processing." in *NSDI*, 2021, pp. 487–501.

[6] The Cilium Project, "Cilium Project Home Page," https://cilium.io/, 2020, accessed: 15-01-2021.

[7] Engineering at Meta, "Open-sourcing Katran, a scalable network load balancer," 2021.

[8] E. Gershuni *et al.*, "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*. New York, NY, USA: Association for Computing Machinery, 2019, p. 1069–1084.

[9] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with ebpf: Experience and lessons learned," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–8.

[10] L. Nelson, J. Van Geffen, E. Torlak, and X. Wang, "Specification and verification in the field: Applying formal methods to {BPF} just-in-time compilers in the linux kernel," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 20)*, 2020, pp. 41–61.

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

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

[13] T. Høiland-Jørgensen *et al.*, "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*. New York: ACM, 2018, pp. 54–66.

[14] N. Van Tu *et al.*, "evnf - hybrid virtual network functions with linux express data path," in *2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. New York: IEEE, 2019, pp. 1–6.

[15] D. Scholz *et al.*, "Performance implications of packet filtering with linux ebpf," in *2018 30th International Teletraffic Congress (ITC 30)*, vol. 01. New York: IEEE, 2018, pp. 209–217.

[16] M. AM Vieira et al., "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[17] N. Van Tu, J.-H. Yoo, and J. W.-K. Hong, "Accelerating virtual network functions with fast-slow path architecture using express data path," *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1474–1486, 2020.

[18] S. Miano *et al.*, "A Framework for eBPF-Based Network Functions in an Era of Microservices," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 133 – 151, 2021.

[19] The eCLAT project, "eCLAT HIKe Technical Documentation," https://hike-eclat.readthedocs.io/en/latest/index.html, 2021.

[20] A. Mayer et al., "Composing eBPF Programs Made Easy with HIKe and eCLAT - Technical report," 2021. [Online]. Available: https://github.com/hike-eclat/docs/blob/master/tech-docs/tr-hike-eclat.pdf

[21] A. Mayer, "Network programmability in Software Routers," PhD Thesis, https://tiny.one/mayer-phd-thesis, 2022.

[22] A. Mayer *et al.*, "Performance Monitoring with H^2: Hybrid Kernel/eBPF data plane for SRv6 based Hybrid SDN," *Computer Networks*, vol. 185, 2021.

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

[24] R. Gonzalez, P. House, I. Levkivskyi, L. Roach, and G. van Rossum, "Python syntax for syntax for variable annotations," PEP 526, 2016. [Online]. Available: \url{https://www.python.org/dev/peps/pep-0526/}

[25] Google Developers, "Protocol Buffers," https://developers.google.com/protocol-buffers, 2021.

[26] The eCLAT project, "eCLAT docker Github Page," https://github.com/netgroup/eclat-docker, 2021.

[27] "Bcc project," https://github.com/iovisor/bcc.

[28] "BPF CO-RE (Compile Once – Run Everywhere)," https://nakryiko.com/posts/bpf-portability-and-co-re/.

[29] Netfilter Project, "IP Sets Home Page," https://ipset.netfilter.org/, 2021.

[30] A. Abdelsalam *et al.*, "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*. New York, US: IEEE, 2018, pp. 414–419.

[31] Z. X. et al., "Reducing latency in virtual machines: Enabling tactile internet for human-machine co-working," *IEEE JSAC*, vol. 37, no. 5, pp. 1098–1116, 2019.

[32] S.-Y. Wang and J.-C. Chang, "Design and implementation of an intrusion detection system by using extended bpf in the linux kernel," *Journal of Network and Computer Applications*, p. 103283, 2021.

[33] J. Levin and T. A. Benson, "Viperprobe: Rethinking microservice observability with ebpf," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. IEEE, 2020, pp. 1–8.

[34] M.Xhonneux, F.Duchene and O. Bonaventure , "Leveraging ebpf for programmable network functions with ipv6 segment routing," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. ACM, 2018, pp. 67–72.

[35] S. Miano et al., "A service-agnostic software framework for fast and efficient in-kernel network services," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2019, pp. 1–9.

[36] Z. Ahmed, M. H. Alizai, and A. A. Syed, "Inkev: In-kernel distributed network virtualization for dcn," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 3, jul 2018. [Online]. Available: https://doi.org/10.1145/3243157.3243161

[37] M. Bertrone, S. Miano, J. Pi, F. Risso, and M. Tumolo, "Toward an eBPF-based clone of iptables," in *Netdev 0x12, THE Technical Conference on Linux Networking*. Nepean, Canada: The NetDev Society, 2018.

[38] W. Tu *et al.*, "Bringing the Power of eBPF to Open vSwitch," in *Linux Plumbers Conference 2018*. San Francisco, California: The Linux Foundation, 2018, p. 11.

[39] C.-C. Tu, J. Stringer, and J. Pettit, "Building an extensible open vswitch datapath," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 1, pp. 72–77, 2017.

[40] J. Kicinski and N. Viljoen, "ebpf hardware offload to smartnics: cls bpf and xdp," *Proceedings of netdev*, vol. 1, 2016.

[41] M. Spaziani Brunella et al., "hXDP: Efficient Software Packet Processing on FPGA NICs," in *USENIX OSDI 2020)*, 2020, pp. 973–990.