eBPF Programming Made Easy with HIKe and eCLAT
TECHNICAL REPORT

Anonymous Authors

Abstract

With the rise of the Network Softwarization era, eBPF has become a 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 identify the obstacles faced by eBPF developers and propose the *HIKe/eCLAT* framework. HIKe stands for "Heal, Improve and desKill eBPF". It offers a Virtual Machine abstraction (the *HIKe VM*), which facilitates the *composition* of eBPF programs using a programmatic approach. Stand-alone eBPF programs can be composed into the so-called "HIKe chains" using the C language. Then we mapped the HIKe abstractions into a high level language and programming framework named eCLAT (eBPF Chains Language And Toolset). A developer can write eCLAT scripts in a python-like language to compose HIKe eBPF programs, with no need of understanding the complex details of regular eBPF programming.

Note: the text added in the extended version is included in a frame like this.

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. To support the demanding requirements of workloads running on containers and VMs, there is the need for fast and efficient packet processing solutions targeted to Linux/x86 nodes [1, 2]. 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 [3–6]. For example, eBPF is the core technology of Cilium, a leading framework for secure networking in the Kubernetes container orchestration platform [7]. Another prominent example is Katran, a layer-4 load balancer, open-sourced by Facebook [8].

```
1 flow_rate = flowmeter(packet)
2 #drop flows greater than 10Mbps
3 if flow_rate > 10:
4     droppacket()
5 else:
6     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.

While Cilium and Katran clearly demonstrate the success of eBPF, 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 [9, 10]. In particular: i) each eBPF program needs to be *verified* by the kernel before being loaded, this process is very annoying for the developer [11]; ii) it is not possible to call a loaded eBPF program from another loaded eBPF program with the normal *function call* paradigm, a program P1 can only *tail call* another program P2 [12, 13] , i.e. P1 will transfer the flow of execution to P2 with no way back and no possibility to evaluate a return value.,

The issues in the verification phase result in an increase of the development time with a big loss of productivity. The fact that function calls are not possible across loaded eBPF programs is a strong obstacle to modular code reuse. Developers have to go back into coding of a new "big" eBPF program instead of composing existing ones. In this paper, we present the HIKe / eCLAT framework that addresses the above mentioned issues. We show how it can substantially improve the applicability and usability of eBPF, lowering the barrier to develop complex eBPF based solutions.

eCLAT (eBPF Chains Language And Toolset) is a language and toolset to dynamically compose eBPF programs. An ex-

ample of an eCLAT script, which we call *chain*, is shown in Listing 1. The highlighted names refer to independently running eBPF programs which are called inside the chain as they were functions. As simple as it may seem, such form of programming is NOT possible in the normal eBPF framework, in which it is not possible to use the function call to compose independent eBPF programs. This eCLAT chain is compiled and executed in the context of an eBPF program, but it does NOT need to be verified by the kernel. Such form of composition and execution without verification is feasible thanks to the features provided by our eBPF framework called HIKe (Heal, Improve and desKill eBPF) Virtual Machine.

In section 2 we discuss the type of eBPF programmability offered by our solution and we compare it with what is offered by other eBPF frameworks. Section 3 provides some background on eBPF technology and identifies its shortcomings and pitfalls that motivate our work. Section 4 illustrates the architecture of the proposed solution, while section 5 and 7 respectively focus on HIKe and eCLAT.

2 eBPF programmability

The development of networking code inside the kernel of an Operating System 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 [14] and the metaphor of Wizard and Muggles fits perfectly with the eBPF ecosystem. With this metaphor in mind, we can state that our goal is to allow Muggles to easily *program* eBPF solutions, by writing the application logic that combines a set of eBPF programs (written by Wizards).

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.

How does our solution compare to other existing eBPF frameworks like Cilium [15] and Polycube [16]. The underlying approach behind these approaches is to employ Wizards to develop eBPF program templates (modules), leaving their final *configuration* to Muggles . All the process usually pass through a code generation engine which works in user space and is responsible for generating the final eBPF code and to inject it into the kernel. The drawbacks of this approach are:

- (for Wizards) creating new modules inside such frameworks requires more effort and ability than writing conventional eBPF programs.
- (for Muggles) the configuration does not fully cover the need for arbitrarily complex packet processing "business logic".

Polycube offers a form of composition of its *cubes*, but this is not a composition of programs *inside* the eBPF framework. A cube is a monolithic eBPF program, which receives packets on a virtual interface and emits packets on another one. Composing two cubes means connecting the output interface of a cube to the input interface of another cube.

Our approach is different, moving from *configuration* or *composition* to eBPF *programmability*. Our building blocks are conventional eBPF programs, written by Wizards and slightly adapted to work in our framework (just a few lines of code is needed). We refer to such programs as *HIKe eBPF programs*. Differently from other approaches (such as Polycube's "cubes"), HIKe eBPF programs are not necessarily complex programs (e.g., "the firewall") but can be also small utilities such as "flow meter" or "drop this packet". These HIKe eBPF programs can be called as "functions" inside a higher level program which we call *chain*, allowing the programmability of custom business logic. The chain can be expressed using a high-level program language called eCLAT. The chain executes the business logic and called eBPF programs inside the eBPF framework.

Using a Unix similarity, it is like having many standalone tools such as `cut`, `tail` or `sed`, written by wizards. Muggles can use them in custom bash scripts to cover a wide range of specific application needs, together with variables definitions and constructs such as branching conditions. In our case the eCLAT chain, written in the eCLAT language is the equivalent of a bash script. eCLAT moves the eBPF programming away from C/assembly/bytecode programs towards high-level, python-like scripts.

The benefits of this approach are:

- (for Wizards) they write HIKe eBPF programs without almost any knowledge of the HIKe framework, or they can easily adapt existing ones
- (for Muggles) thanks to the eCLAT language, they have a great flexibility for implementing any custom business logic, with no need to learn eBPF programming

HIKe chains are written in C and interpreted inside eBPF itself, thanks to the proposed HIKe Virtual Machine (HIKe VM). The HIKe VM can run on any recent Linux kernel as it is based on the available eBPF framework.

The solution proposed in this work is decoupled in two levels:

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 runtime.

HIKe is designed for the eBPF Wizards and is meant to boost the productivity of 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 reused 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 overcome the eBPF issues and limitations that are further discussed in section 3.

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.

With the proposed HIKe and eCLAT we move from eBPF configurability/composability, to real *programmability* made easy.

3 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 subsection 3.1 we provide a short overview on the eBPF technology, without claiming to be exhaustive. In the subsequent subsections, we identify the main shortcomings and pitfalls of eBPF and its current implementation. This analysis provides the motivations for the HIKe / eCLAT framework that will be presented in the following sections.

3.1 eBPF overview

The extended Berkeley Packet Filter (eBPF) [17] is a low level programming language that is executed in a Virtual Machine (VM) running in the Linux kernel. eBPF has been profitably

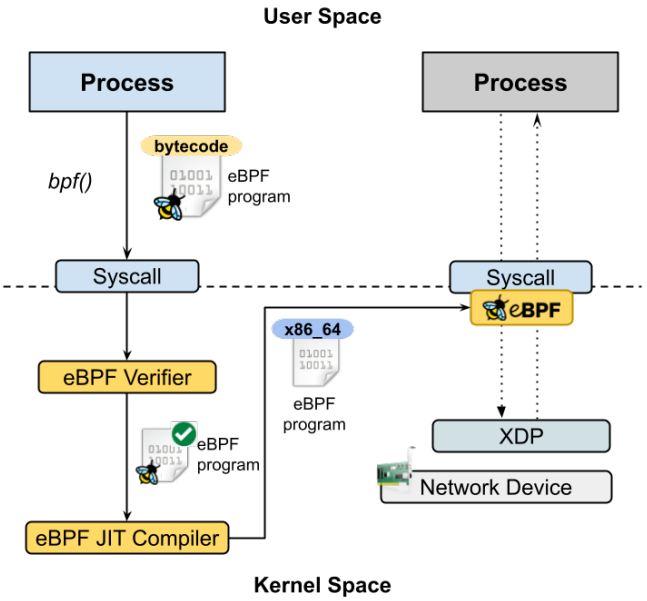

Figure 1: eBPF programs compilation and verification

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. 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 [18]. 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.

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. 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.

Focusing on packet processing, eBPF programs 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 [19] 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., [20] [21] [22] [23]). 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 program 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.

3.2 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, and no complex pointer arithmetic, while it has been shown to be a very tough obstacle to the development of complex applications [18]. As analysed in [9], 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 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 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 the reasons above, the verification step is the main obstacle in the implementation of complex eBPF applications.

3.3 eBPF (un)composability

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

In normal software engineering, large systems are divided into 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. The inline approach may greatly increase the size of the code block to be compiled, optimized, and verified. This may cause trouble, 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.

With the inline approach 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. 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 [7]). 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.

In addition to compile-time composition (which actually means composition by coding, compilation, and *verification*), the eBPF framework offers the possibility to compose already compiled and verified eBPF programs using the concept of *tail 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 flow of execution does not go back to the "calling" eBPF program. Moreover, it is not directly possible to pass parameters from the caller program to the called one. Clearly, the developers cannot use tail calls according to the *function call* pattern that everyone is used to.

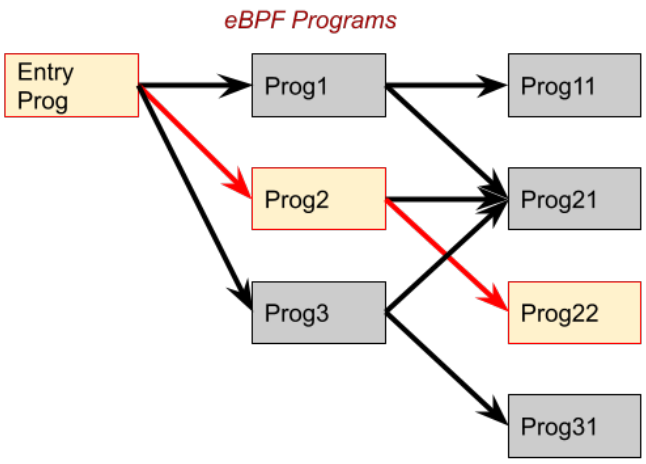

Figure 2: Chaining eBPF programs with tail calls

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.

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 [16].

3.4 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. The eBPF framework provides the abstraction of *BPF maps* to these purposes. The BPF maps are key/-value stores residing in the Linux kernel memory. Different types of BFP maps are supported, i.e. with different key and value types (see [7]). 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. The Wizards have to design the libraries and/or GUI tools that can be used by the Muggles with basic programming skills to indirectly interact with the BPF maps.

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 programs with the BPF maps. As multiple eBPF programs can be executed in parallel to process a set of incoming packets (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 operations. It is a task of the developer to handle the concurrency when needed and this is in general a hard task even for wizards. **SS:** we could add here a mention to the BCC attempt to solve this problem and to its shortcomings

4 Overview of the Solution

Let us describe the solution from the point of view of a Muggle, who wants to easily develop a specific packet processing procedure. The Muggle needs to write an eCLAT script and load it so that it will be executed on the incoming traffic.

The architecture of the solution is shown in Fig. 3, and it is composed of two layers, eCLAT and HIKe. In turn, the eCLAT layer is based on a user-level daemon, developed in Python, and on a Command Line Interface (also developed in Python) used by the Muggle to interact with the eCLAT Daemon. The interaction between the eCLAT CLI and the Daemon is based on gRPC.

The eCLAT Daemon receives the scripts from the eCLAT Chains described in the eCLAT language. The daemon first "transpiles" them into into C language, generating the source code of HIKe Chains, then it compiles the C HIKe Chains into a bytecode suitable for the HIKe VM. Actually this is not only a compilation operation, because the eCLAT Daemon also works as a linker, resolving the references to HIKe eBPF Programs and to other Chains that are called inside a Chain and writing 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 Chain refers to an HIKe eBPF Program, the eCLAT daemon checks if that program is already loaded and if not, it loads it. The compiled/linked bytecode of a Chain 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

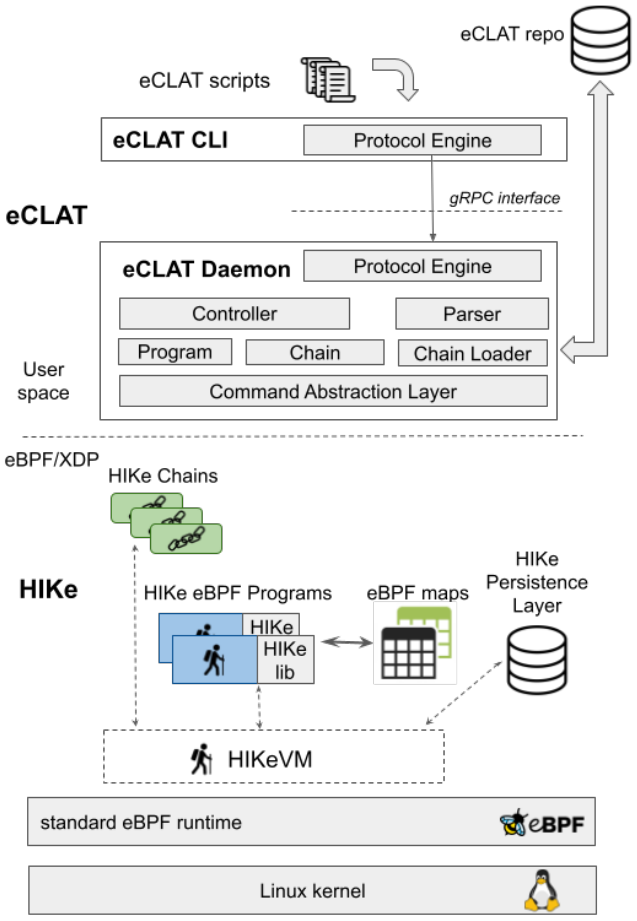

Figure 3: HIKe / eCLAT Overall Architecture

programs to read/write information. Further discussion on the eCLAT layer is reported in section 7.

The HIKe layer provides the execution environment for the bytecode of the HIKe Chains. As it will be detailed in section 5, we have designed and implemented a Virtual Machine abstraction called HIKe VM, in the form of an eBPF library that needs to be included in all HIKe eBPF programs. Therefore, to run an eBPF program in the HIKe framework, the source code of the program needs to be slightly modified by adding the calls to the HIKe VM library and the program needs to be recompiled/verified/loaded as any eBPF program.

4.1 An eCLAT Script Example

Let us consider the following packet processing logic that the Muggle 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.

The logic refers to a DDOS mitigation scenario, implementing this logic for a non-skilled eBPF programmer is not easy. In eCLAT, the non experienced programmer (Muggle) 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 parameter (`LOOKUP`). The result is 0 if the IPc6 (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 endianess automatically). Variables are typed, using the Python syntax for Syntax for Variable Annotations (PEP 526). 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 eBPF HIke programs as they were Python functions.

```
from prog.net import hike_drop, hike_pass,  \
     ip6_hset_srcdst, ip6_sd_tbmon, monitor,\
     ip6_dst_tbmon, ip6_sd_dec2zero,
    l2_redirect
from loaders.basic import ip6_sc

# send all IPv6 packets to our chain
ip6_sc[ipv6_sc_map] = { (0): (ddos_tb_2_lev) }
ip6_sc.attach('DEVNAME', 'xdp')

def ddos_tb_2_lev():
    PASS=0; DROP=1; REDIRECT=2;
    REDIRECT_IF_INDEX = 6;
    ADD=1; LOOKUP=2;
    BLACKLISTED = 0;
    IN_PROFILE = 0;

    # (src,dest) in blacklist ?
    u64 : res = ip6_hset_srcdst(LOOKUP)
    if res == BLACKLISTED:
        # redirect one packet out of 500
        res = ip6_sd_dec2zero(500)
        if res == 0:
            monitor(REDIRECT)
            l2_redirect(REDIRECT_IF_INDEX)
            return 0

        monitor(DROP)
        hike_drop()
        return 0

    # check the rate per (dst)
    res = ip6_dst_tbmon()
    if res != IN_PROFILE:
        # check the rate per (src,dst)
        res = ip6_sd_tbmon()
        if res != IN_PROFILE:
            # add (src,dest) to blacklist
            ip6_hset_srcdst(ADD)
            monitor(DROP)
            hike_drop()
            return 0

    monitor(PASS)
    hike_pass()
    return 0
```

Listing 2: eCLAT script for DDOS mitigation

5 HIKe: Heal, Improve and desKill eBPF

The shortcomings of eBPF and of its development process and tool chain motivate our work. We aim at composing eBPF programs using a *function call* pattern and without the hassles of the verification phase. Unfortunately, these requirements cannot be met by the current eBPF framework. Furthermore, it is not 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 approach is based on a new lightweight Virtual Machine abstraction (HIKe VM) running on top of the existing eBPF VM. Using the HIKe VM we can execute the *HIKe Chains*, which combine eBPF programs (actually HIKe

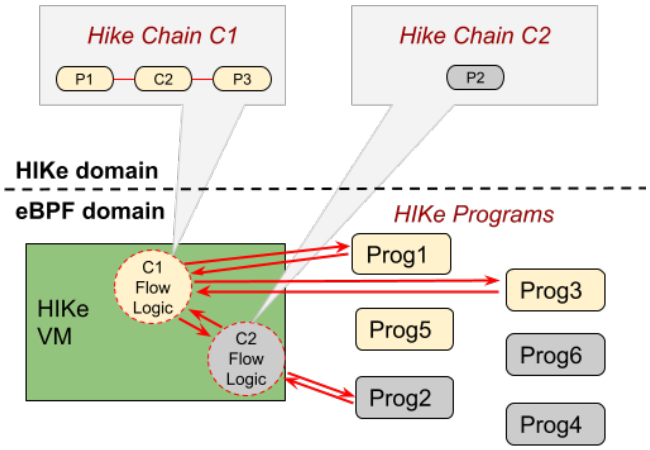

Figure 4: HIKe VM, Chains and Programs

eBPF Programs) leveraging the familiar and well-known *function call* paradigm. A HIKe chain is represented with bytecode which is interpreted by the HIKe VM. The bytecode encodes logical and arithmetical instructions, jump instructions to control the program flow, calls to HIKe eBPF programs and to other HIKe chains.

In Fig. 4, we represent two HIKe Chains, C1 (composed of two HIKe programs and one HIKe Chain) and C2 (composed of one HIKe program).

As shown in Fig. 5, the HIKe VM is included as an in-line library inside all the *HIKe eBPF programs*. The processing of a packet by HIKe starts when the packet is intercepted by the XDP Hook and handed to a special HIKe eBPF program that has the task to associate a HIKe Chain to the packet (C1 in the example), based on some classification rules. This program is called eBPF Traffic Classifier in Fig. 5. After performing its classifier logic, this program sets up the HIKe Chain C1 that has to be executed and starts to interpret the bytecode of C1 thanks to the HIKe VM library that is included in every HIKe program. As shown in Fig. 5, when the bytecode of the C1 corresponds to a function call to a different HIKe eBPF program P1, a tail call is executed and the packet is processed according to the program logic of P1. After the execution of P1 program logic, the control flow is virtually moved again to the Chain C1, this means that program P1 is executing the HIKe VM library code which is included in P1 as well. When the bytecode of C1 requires the execution of another chain C2 as a function call, this is handled by the HIKe VM, with no need to change the eBPF program in execution. When Chain C2 needs the execution of an eBPF program P2, a second tail call is performed, the program logic of P2 is first executed, then the processing of the Chain bytecode continues in the execution context of P2. When the execution of Chain C2 is completed, the execution flow returns to the calling Chain C1, and this happens in the execution context of program P2 (i.e.

in the VM library code). The execution of the HIKe Chain context associated to a packet terminates when the program logic of a Program called by a Chain takes a decision on the fate of the packet: pass to kernel, drop, send to the NIC.

The memory footprint of the HIKE VM library that needs to be included in each HIKe eBPF Program is in the order of 1K instructions, and this has a negligible impact on the run time processing performance. In addition, it does not affect the verification phase, as the root user can verify and load eBPF programs with up to 1M instructions [24].

Writing HIKe eBPF Programs is straightforward for an eBPF programmer. To turn an eBPF program into a HIKe eBPF Program, it has to be "decorated" with some helper functions contained in the HIKe library. In practical terms, this means adding 3 or 4 lines of C to the source code. Then the HIKe Program is recompiled, verified, and loaded as any eBPF program.

In the next subsection we focus on safety of the HIKe VM and we discuss the advantages of the HIKe programming model. Deeper technical details on the design and implementation of HIKe are reported in [25].

5.1 Safety of HIKe VM and Chains

While the single HIKe eBPF program needs to be compiled/verified, the bytecode of an HIKe Chain does not need to pass through the eBPF verifier. Nevertheless, the HIKe VM guarantees to meet all the eBPF safety constraints. From the point of view of the eBPF framework, the HIKe VM is seen as a normal eBPF Program running in the XDP context. Since every instruction of an HIKe Chain is executed by the HIKe VM which is in turn an eBPF program, the eBPF verifier makes sure that the HIKe VM cannot be harmful for the system. The key idea is that the HIKe VM code interpreter/executor considers every possible fault/exception raised by an instruction and it acts accordingly. In case of illegal operations (i.e.: de-referencing a NULL pointer, unbounded loops, etc), the HIKe VM aborts the execution of the faulty chain. As a consequence, HIKe Chains are safe by construction and do not force the developer to adopt an unnatural code programming style with the tricks needed in traditional eBPF programs in order to be considered valid.

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.

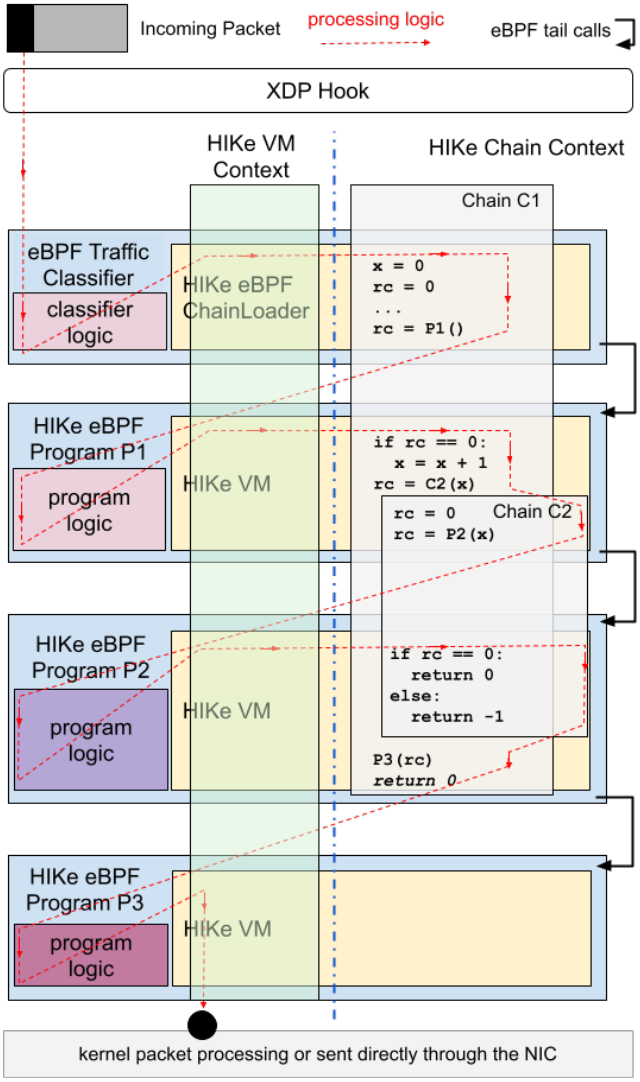

Figure 5: Composition of HIKe eBPF programs with a HIKe Chain, using the HIKe VM

5.2 PROs of the HIKe programming model

Differently from the tail call approach described in Fig. 2: i) the HIKe eBPF programs that are composed are not aware of the structure of the interconnections; ii) the HIKe eBPF Programs that are compiled and verified once for ever; iii) the Chain flow logic relies on the results of the execution of the eBPF programs, but changing the Chain logic is NOT subject to the eBPF verifier: a HIKe Chain is compiled for the HIKe VM, not for the eBPF VM!

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 semantic. 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.

The HIKe VM is designed reusing a subset of the eBPF VM instruction set, hence its bytecode is compatible with the eBPF VM. Thanks to this design choice, we can write the HIKe chains in C language and reuse the Clang/LLVM toolchain to generate the HIKe VM bytecode. Hence, the HIKe framework fully benefits from the maturity of the LLVM toolset, with no need to reinvent the wheel.

Considering that the application domain of the HIKe VM is the handling of network packets, we included in the HIKe VM some helper functions to 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* to the packet itself. Thus it "follows" the packet through its "journey" across the different HIKe eBPF programs and Chains that are executed. This avoids the "reclassification" issue that affects the normal function chaining approaches, which often require to keep some state information to reclassify packets after a function has been executed to forward the packet to the next function.

6 HIKe Deep Dive

This section is only in the extended version

The components of the HIKe architecture are listed hereafter.

- **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 by the HIKe Chains) and to store the bytecode of the HIKe Chains; the HIKe PL is based on eBPF Maps.

6.1 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 influences 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 6.5.

The source code of a HIKe eBPF Program is shown in Listing 6 in Appendix 11. Two macros (i.e. pre-processor directives) are used to transform an eBPF program into a HIKe eBPF Program.

6.2 HIKe Chains

The HIKe Chains are written in C language. The source code of a HIKe Chain is shown in Listing 3. The code is compiled into a bytecode that is subsequently stored in the HIKe Persistence Layer. During this registration 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.

The HIKe Chain defined in Listing 3 is identified by the label `HIKE_CHAIN_MON_IPV6_ALLOW`. Its goal is to identify and monitor IPv6 packets and its logic can be easily followed by reading the source code. It first reads the Ethernet type from the packet (the details for reading a field from the packet are omitted in Listing 3 and can be found in [25]). If the Ethernet type is IPv6 (`0x86dd`), then it calls the HIKe eBPF Program `mon_ipv6_pkt()`, which can perform further monitoring operations specific for IPv6 packets.

Finally, the HIKe eBPF Program `allow` is called, which accepts the read Ethernet type as a parameter. Note that this program could perform other operations depending on the Ethernet type of the packet, without the burden of accessing again to the field of the packet. The `allow` Program terminates the execution of the HIKe Chain and for this reason it is said to be *final*.

This very simple example has shown the combination of

```
#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 */
}
```

Listing 3: A HIKe chain

two already compiled eBPF programs by means of the *function call* pattern, which *solves a fundamental problem of the eBPF framework*. With the same programming easiness, a Chain can invoke another Chain (further details on the Chain-to-Programs and Chain-to-Chain function calls are provided in [25]).

We note that the HIKe chain is defined 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 the 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 `chainid` which is the only parameter for this chain.

The function calls to the `mon_ipv6_ptk()` and `allow()` HIKe eBPF Programs are also coded 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.

The HIKe framework also provides 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`.

6.3 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. 6. The HIKe VM retrieves the references to the HIKe eBPF Programs and the bytecode of the HIKe Chains from the HIKe Persistence Layer. Fig. 6 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.

Using the HIKe VM and the function call pattern to compose Programs and Chains, it may seem that we can overcome the intrinsic limitations imposed by the eBPF VM and provide a Turing complete execution environment. This is NOT the case, otherwise we would have found a way to overcome the security constraints of the eBPF environment. What happens is that we have: i) a static limitation in the number of instructions that compose the bytecode of a chain; ii) a dynamic limitation on the number of instructions that can be executed by an instance of a chain. The static limitation can be changed by recompiling the HIKe VM, we have set it to 32. The dynamic limitation is set to 32. The dynamic limit affects the verification phase when the HIKe VM is compiled and verified as any eBPF program. The higher the limit, the longer the time the verifier will take to check the validity of the HIKe VM before loading it.

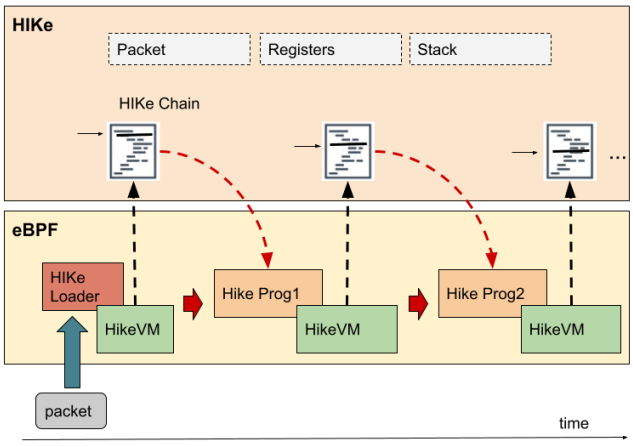

Figure 6: HIKe VM executing a HIKe chain

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.

6.4 HIKe Chain Loader

Upon the arrival of a packet on a network interface (configured for HIKe), an eBPF program called *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 Chain 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).

6.5 HIKe Persistence Layer

The HIKe Persistence Layer (HIKe PL) solves two fundamental problems: i) store the references to the HIKe eBPF Programs so that the HIKe Chains can call them by their ID; ii) identify the HIKe Chains and store their bytecode, so that the 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 in the HIKe framework, which can simply be abstracted as two key-value stores, represented in Table 1. The key-value stores support the retrieve/delete/set/update operations and they are implemented as eBPF maps.

The registration of a HIKe eBPF program or the storage of a HIKe Chain requires the assignment of an ID to the Program or Chain. This is managed by an entity which is external to the HIKe PL and implemented in user space. The main motivation for this design choice is that the operations on the eBPF tables need to be synchronized to avoid concurrent writes leading to inconsistency. It is very hard to achieve synchronization at eBPF level, as there is no way to lock multiple tables. It is much cleaner and easier to implement the ID management and the registration operation in a user space program/daemon.

Table 1: Key-value stores in the HIKe Persistence Layer

Key	Value
HIKe eBPF Program ID	Reference to the HIKe eBPF Program (file descriptor)
HIKe Chain ID	Bytecode of the Chain

6.6 HIKe development process

With reference to Fig. 7, the implementation of a complete HIKe eBPF system requires the following steps:

- **Step 0**: initialization of system-wide HIKe maps with pinning.
- **Step 1**: the first step is devoted to the loading of the HIKe eBPF 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 ("hardcoded") or read from a map. Finally, the program needs to be loaded and eventually attached to the XDP hook.

6.7 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 the handling of 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.

6.8 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

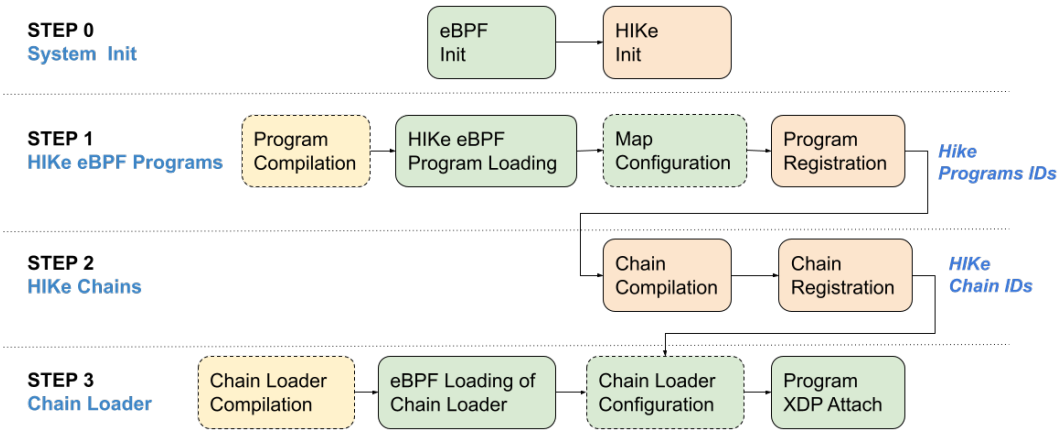

Figure 7: Steps required to set up a complete eBPF/HIKe system

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)  |
+--------------+------------------------+
```

6.9 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 4. `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.

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

Listing 4: Access to packet fields in a HIKe Chain

With reference to the Listing 4, 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 5 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.

```
[...]

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

    [...]
}
```

Listing 5: Getting the packet length

6.10 HIKe Chain-to-Program calls

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 6, 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.

6.11 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.

6.12 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.

7 The eCLAT abstraction

HIKe introduces programmability and modularity in eBPF, however the technical entry barrier for a developer that wants to implement a custom logic in eBPF network functions is still high. For this reason we designed eBPF Chains Language And Toolset (eCLAT), a framework and a language that supports eBPF/HIKe based network programmability offering a high level programming abstraction.

eCLAT simplifies the reuse of the HIKe eBPF programs by defining a python-like scripting language used to specify all HIKe operations, such as composing eBPF programs and configuring eBPF maps, hiding the difficult technical details (i.e. avoiding the need of low level programming eBPF using the C language and simplifying the retrieval, compilation and injection of HIKe eBPF programs).

In a nutshell, eCLAT provides an easy-to-use python-like language to simplify the writing of HIKe chains, avoiding the need of explicit interaction with the HIKe Persistence Layer.

7.1 Features

- **Easy-to-use language to express HIKe Chains** using a python-like scripting language, and providing a library for easy access to the packet fields.

- **Download and manage the HIKe eBPF Programs** by automatically collecting the imported programs from the eCLAT code repository, taking care of compilation and byte-code injection. Calling another programs appears to the eCLAT developer as a conventional Python function call, masquerading the needs to register and use numerical program IDs needed by the HIKe persistence layer.
- **Simple access to BPF maps**, viewed as simple python dictionaries.
- **Define an entry point** by specifying the HIKe Chain Loader responsible to trigger the HIKe VM, and its configuration. As an example we can have a classifier based on the destination IPv6 address: according to the IPv6 destination address, different chains can be executed.

7.2 Architecture

Fig. 3 shows the architectural view of eCLAT.

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 [26]. 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 eCLAT CLI allows users to query the daemon about the current status of eBPF maps.

The daemon-based architecture is also consistent with a security limitation of eBPF which grants only to the process that mounted the eBPF filesystem (or one of its children) to have access to the filesystem itself. In our case, the eCLAT daemon initially mounts the eBPF filesystem and then is responsible of all related operations.

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**: it 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 programs and chains. These identification numbers will be fundamental for the chain compilation phase since the HIKe Chains rely on numerical IDs for calling programs, rather than on the names which are used in the eCLAT domain.
- **Program**: wraps and manage an 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 programs, the information about the structure of the program (variables, functions, structs, etc.) are automatically extracted and registered in a JSON file. This file is also parsed to obtain all map-program associations.
- **Chain**: it 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) HIKe program. Then compiles it to generate artifacts (object files) through the execution of a dedicated Makefile. Finally, it registers the output in the HIKe Persistence Layer.
- **Chain Loader**: this component handles the HIKe Chain Loader Programs and interacts with their maps. Using the eCLAT scripting language, the programmer can specify the program that has to be loaded and attached to the XDP hook and can configure its maps.
- **Parser**: it 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 of the chains.
- **Command Abstraction Layer**: provides an abstraction over the different shell commands that needs to be invoked on the operating system.

8 Evaluation

8.1 HIKe data plane performance evaluation

Which is the performance penalty in using HIKe abstraction instead of low-level coding in eBPF? To answer to this question, we focus on data plane performance and we consider the HIKe framework and the HIKe VM. We measured the processing overhead caused by executing (interpreting) operations inside the HIKe VM, which in turn is executed by the eBPF VM. We expect that writing the Chains using eCLAT will not decrease performance, because the transpiling operation from eCLAT to C does not introduce overhead and the transpiled C code is compiled as a HIKe chain.

Our performance evaluation consists in measuring the maximum forwarding throughput of a node. This is defined as the maximum packet rate (measured in packets per second - pps) for which the packet drop ratio is smaller than or equal to 0.5%. Further details on the methodology are reported in Appendix 13 and in [27].

8.1.1 Testing scenario: DDoS mitigation with HIKe

We consider a realistic DDoS mitigation scenario which is a popular application of eBPF XDP [28]. We assume that a software Linux router is forwarding packets and we have to identify the packets that belong to a *blacklist* of source IP addresses. In particular we want to mark the packets in the blacklist with a given IP TOS. The marking program adds a processing burden to the normal forwarding operations, the obvious goal is to keep this burden as low as possible. We compared three solutions for the marking program: i) a conventional eBPF program ("raw"); ii) a HIKe program ("hike"); and iii) an ipset based ([29]) approach ("ipset"). The eBPF raw solution is the most difficult to be programmed, only the *wizards* can do it. The HIKe solution could even be developed by the *muggles* using eCLAT. The IPSet solution has an intermediate development complexity. Further details on these implemented solutions are described in [25], their source code is available at [30].

	Raw eBPF	**HIKe**	**Ipset**
Av. Throughput (kpps)	**2570**	**1867**	**991**
Std. Dev. (kpps)	2.1	3.9	1.4
overhead w.r.t. raw eBPF	-	27.7%	61.4%
overhead w.r.t. HIKe	-	-	46.7%

Table 2: Performance of DDoS implementations

Table 2 reports the achieved speed in terms of maximum throughput. As we can see, HIKe presents a $\sim 27.7\%$ of performance degradation with respect to the raw eBPF approach (1.8 Mpps versus 2.6 Mpps). However, HIKe outperforms ipset whose performance is $\sim 46.7\%$ worse than HIKe and $\sim 61.4\%$ worse than the raw eBPF program. These figures of course depend on the specific scenario, in particular on the processing cost of the "background" node operations and of the additional operations that we want to add to eBPF or HIKe/eCLAT. The higher the processing cost of background operations, the lower will be the difference of the impact of HIKe vs. raw eBPF. Moreover, if we use HIKe to combine a number of eBPF programs, if the processing cost of a single program is high, the cost of the composition is relatively low, and vice versa.

8.1.2 Testing scenario: HIKe worst case overhead

Considering that different scenarios for the background operations and for the program to be composed will result in different estimations of the HIKe overhead with respect to raw eBPF, we consider a reference synthetic scenario which represents the worst case situation for HIKe. In fact, we compose a number of null programs that do nothing, both with HIKe function calls and with raw eBPF tail calls. 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.

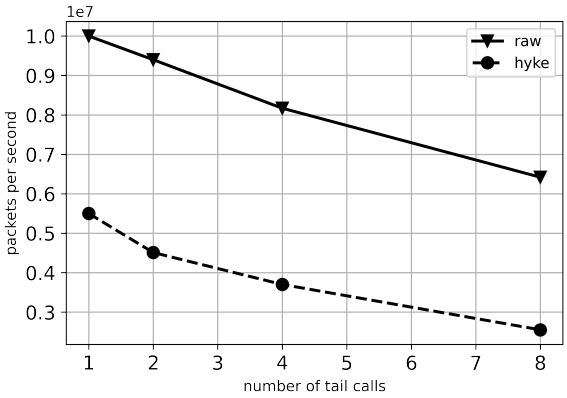

Figure 8: HIKe vs. raw eBPF performance

Fig. 8 shows the max throughput of both HIKe VM ("hike") and the eBPF programs ("raw"), which decreases with the number of tail calls performed. The HIKe VM presents a relative performance degradation of $\sim 40\%$ in case of 2 tail calls (10M pps vs 6.1M pps), and reaches $\sim 54\%$ when 8 tail calls have been considered (6.8M pps vs 3.1M pps). Roughly, we can estimate that the cost of composing programs under the HIKe VM is twice the cost of raw eBPF tail calls. We are very satisfied with this result, considering that: i) the HIKe VM is an interpreter, ii) we have a substantial gain in programming easiness; iii) we are already working to improve the performance with respect to our first prototype; iv) the current performance is already adequate in several real scenario scenarios as shown in the previous subsection.

8.2 Modularity

The greatest benefit of adopting the HIKe/eCLAT framework is in the flexibility and modularity it offers. Table 3 objectivize the benefits of this approach by comparing the presented solution with popular frameworks, Cilium and Polycube, across different dimensions, and specifically:

- **Define application logic**: how a user (muggle) 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 allows to define custom configuration. The difference is that programming flows allows 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 support

Table 3: eCLAT benefits in terms of modularity

Dimension	Cilium	Polycube	eCLAT
Define application logic	configuration and API	configuration of modules and topology	programmatic
Composition approach	user-space code generation	configuration of eBPF programs	inside eBPF
Composition topology	-	linear (tail call)	arbitrary
Modularity	monolithic	big modules (cubes)	any eBPF program
Extensibility	submit a patch to the main project	creation of a new template (cube) within the framework	create a conventional standalone eBPF programs and slightly modify (decorate) it

arbitrary topology as the data flow can follow different branches and loops. Other frameworks like Polycube are limited by a linear topology: data passes through a predefined set of eBPF programs which hands over through a set of tail calls.

- **Composition approach**: where the composition of different modules happens? eCLAT is the only one which permit composition *inside* eBPF, i.e. each module is an eBPF program, and the logic programs are called is written in another eBPF program. This is different from models where the composition happens in user space and then, through code generation, a eBPF program monolith is injected.
- **Modularity**: Which is a module? Cilium is monolithic, and Polycube relies mainly on "big" modules (e.g., "the firewall") as they can be only chained together. Conversely, for eCLAT a module is a standalone eBPF program that can be also quite small (e.g., a flow meter) as its utility must not be absolute, but functional of the context where it will be placed in the HIKe chain (e.g., in an *if* expression to decide a branch).
- **Extensibility**: How a user (wizard) can 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.

9 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.

9.1 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. [18] 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 [18] are part of the motivations which led us to the design and development of HIKe and eCLAT. Gershuni et al. [9] 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. We also recognize the relevant role of the "validation hell" and we believe that the HIKe architecture can help to mitigate the problem.

9.2 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 [16, 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 [36] where eBPF pro-

grams 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 [37]. *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 flows 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 be certainly 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 [7]. 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 the new approach proposed by HIKe.

Risso et al. proposed an eBPF-based clone of iptables [38]. 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 [39, 40]. The works propose to move the existing flow processing features in OVS kernel datapath into an eBPF program attached to the TC hook. They also investigate how to adopt kernel bypass using `AF_XDP` and moving flow processing into the user space. HIKe/eCLAT can be used inside OVS as well but it is not our focus to specialize the framework for a particular application.

Finally, several authors implement eBPF Hardware Offload to SmartNICs [41, 42].

10 Conclusions

Using the HIKe framework, the eBPF *wizards* can provide HIKe eBPF Programs that can be easily composed by *muggles* using programmable HIKe Chains. The HIKe VM executes the Chains without passing through the eBPF verifier. Programmers can thus call eBPF programs as "simple" function calls. The benefits of the HIKe architecture are twofold: a HIKe eBPF Program can be reused "as is" in several different application contexts with no code change needed. Different application logic can be implemented just by writing an HIKe Chain as the HIKe VM takes care of the runtime composition.

eCLAT is a Python-like scripting language and framework that allows *muggles* to write eBPF applications just specifying the application logic, reusing programs, and relieving them from the difficulties of eBPF programming. eCLAT helps network programmers in the creation of complete applications that mesh up the programs made by wizards.

Both HIKe and eCLAT frameworks are available under a liberal open source license at [30].

The overhead of the HIKe architecture has been evaluated in terms of computational overhead and performance degradation considering a typical scenario and a worst case. The HIKe performance is already better with respect to competing tools for muggles (e.g. netfilter ipset). The performance penalty with respect to raw eBPF programming is already acceptable considering the huge gain in simplicity for some applications. We expect that further work on the HIKe framework will also reduce our gap with respect to raw eBPF.

11 HIKe eBPF Program Example

This section is only in the extended version

```
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);
```

Listing 6: A HIKe eBPF program

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 extended version [25] includes the listings of the two macros and discusses the technicalities of the HIKe eBPF programs.

The `HIKE_PROG` macro expansion is shown in Listing 7. 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.

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

Listing 7: Macro that provides the signature of a HIKe eBPF Program

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.

```
#define EXPORT_HIKE_PROG(progname)          \
__hike_vm_section_tail(progname)            \
int __HIKE_VM_PROG_EBPF_NAME(progname)      \
    (struct xdp_md *ctx)                    \
{                                           \
    struct hike_chain_regmem *regmem;       \
    int rc;                                 \
                                            \
    regmem = hike_chain_get_regmem();       \
    [...]                                   \
                                            \
    rc = progname(ctx, regmem);             \
    switch (rc) {                           \
    case XDP_ABORTED:                       \
    [...] /* remaning XDP_* retcode */      \
    case XDP_REDIRECT:                      \
        return rc;                          \
    }                                       \
                                            \
    [...]                                   \
    hike_chain_next(ctx);                   \
    /* fallthrough */                       \
aborted:                                    \
    return XDP_ABORTED;                     \
}
```

Listing 8: Macro that provides the code to interact with the HIKe VM

Listing 8 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 7, 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 8 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 7 and 8 are pretty easy to use (despite their internal complexity). They offer the possibility to the wizard developer to write HIKe eBPF Programs or to adapt existing eBPF programs very easily.

12 eCLAT Script Example

This section is only in the extended version

Listing 9 provides an example of eCLAT script. In the first 5 lines of the script we import some HIKe eBPF programs (line 1-2), together with a classifier (line 3) that we set as entry point, using the loader object (imported in line 4). The Packet object (line 5) allows easy read and write operations on data packet.

Then we set up the loader by specifying the interface, the hook, the type of classifier and its configuration (lines 9-13). The configuration is specific to the classifier but ultimately needs to associate a classification outcome with a given chain. In this case, packets coming from the two given IPv6 addresses will be processed by the `chain_main` chain.

The `chain_main` (lines 35-43) calls the `chain_tos` to get the TOS of the packet and, according to the returned value (saved in the `tos` variable), calls `chain_fast`, `chain_slow` or `deny` to drop the packet. While the first two are chain calls, the third one is implemented as a tail call for the related HIKe eBPF program.

Line 15 defines the variable `__offset_ttl` which has a global scope and is used inside `chain_fast`. Such chain calls the `packet_counter` HIKe eBPF program which returns a counter value, stored in the `cnt` variable. Then according to this value the packet can be processed by the `fast` or by the `slow` HIKe eBPF programs that respectively route the packet using eBPF (with a fast looking up on a map and bypassing ARP), or hands over the packet to the slower Linux networking stack. Before entering into the slow path, the TTL is overwritten on the packet (line 22). The slow path is also the action performed in the `chain_slow` chain (lines 25-26).

```
from monitoring import packet_counter
from net import fast, slow, deny
from loaders import ipv6_classifier
from eCLAT import Loader
from HIKe import Packet

```

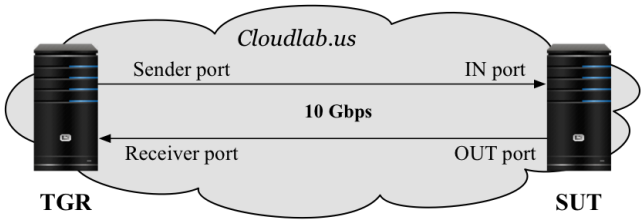

Figure 9: Testbed architecture

```
# set the entry point
Loader.set('enp6s0f0', 'xdp',
    ipv6_classifier, {
    '12:1::2': chain_main,
    '13:1::2': chain_main
})

__offset_ttl = 64

def chain_fast():
    cnt: u32 = packet_counter()
    if cnt < 1000:
        return fast()
    else:
        Packet.writeU8(__offset_ttl, 10)
        return slow()

def chain_slow():
    return slow()

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:
        # block IPv4 packets
        return deny()
    elif tos < 3:
        return chain_slow()
    elif tos >= 3:
        chain_fast()
```

Listing 9: eCLAT script example

13 Testing environment

This section is only in the extended version

We set up a testbed according to RFC 2544 [43], which provides the guidelines for benchmarking networking devices. Figure 9 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, 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 [44], 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 4.

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 [45] that is an open source traffic generator powered by DPDK [46]. We used SRPerf [27], 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 than 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 [27]. wo macros and discusses the technicalities of the HIKe eBPF programs.

References

[1] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. High-speed software data plane via vectorized packet processing. *IEEE Communications Magazine*, 56(12):97–103, 2018.

[2] Minseok Kwon, Krishna Prasad Neupane, John Marshall, and M Mustafa Rafique. Cuvpp: Filter-based longest prefix matching in software data planes. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 12–22. IEEE, 2020.

[3] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. Accelerating linux security with

Table 4: 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

ebpf iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 108–110, 2018.

[4] Federico Parola, Sebastiano Miano, and Fulvio Risso. A proof-of-concept 5g mobile gateway with ebpf. In *Proceedings of the SIGCOMM'20 Poster and Demo Sessions*, pages 68–69. 2020.

[5] William Tu, Joe Stringer, Yifeng Sun, and Yi-Hung Wei. Bringing the power of ebpf to open vswitch. In *Linux Plumbers Conference*, 2018.

[6] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. Bmc: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *NSDI*, pages 487–501, 2021.

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

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

[9] 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*, page 1069–1084, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8. IEEE, 2018.

[11] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi 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)*, pages 41–61, 2020.

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

[13] Daniel Borkmann. Advanced programmability and recent updates with tc's cls bpf. *Proc. NetDev*, 1, 2016.

[14] Gianluca Borello. The art of writing ebpf programs: a primer. https://sysdig.com/blog/the-art-of-writing-ebpf-programs-a-primer, 2019.

[15] Cilium Project. Cilium project home page. https://cilium.io/, 2020. Accessed: 15-01-2021.

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

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

[18] 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)*, pages 1–8, New York, US, 2018. IEEE.

[19] Toke 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*, pages 54–66, New York, 2018. ACM.

[20] 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)*, pages 1–6, New York, 2019. IEEE.

[21] D. Scholz et al. Performance implications of packet filtering with linux ebpf. In *2018 30th International Teletraffic Congress (ITC 30)*, volume 01, pages 209–217, New York, 2018. IEEE.

[22] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.

[23] Nguyen Van Tu, Jae-Hyoung Yoo, and James Won-Ki Hong. Accelerating virtual network functions with fast-slow path architecture using express data path. *IEEE Transactions on Network and Service Management*, 17(3):1474–1486, 2020.

[24] Linux kernel Git - commit: c04c0d2 bpf: increase complexity limit and maximum program size. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c04c0d2b968ac45d6ef020316808ef6c82325a82.

[25] Anonymous Authors. Composing eBPF Programs Made Easy with HIKe and eCLAT - Technical report (anonymized version, 2021.

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

[27] 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*, pages 414–419, New York, US, 2018. IEEE.

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

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

[30] Anonymous. Hike eclat github organization home. https://github.com/hike-eclat, 2021.

[31] Zuo Xiang, Frank Gabriel, Elena Urbano, Giang T Nguyen, Martin Reisslein, and Frank HP Fitzek. Reducing latency in virtual machines: Enabling tactile internet for human-machine co-working. *IEEE Journal on Selected Areas in Communications*, 37(5):1098–1116, 2019.

[32] Shie-Yuan Wang and Jen-Chieh Chang. Design and implementation of an intrusion detection system by using extended bpf in the linux kernel. *Journal of Network and Computer Applications*, page 103283, 2021.

[33] Joshua Levin and Theophilus A Benson. Viperprobe: Rethinking microservice observability with ebpf. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–8. IEEE, 2020.

[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*, pages 67–72. ACM, 2018.

[35] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, Jianwen Pi, and Aasif Shaikh. 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)*, pages 1–9. IEEE, 2019.

[36] Anonymous authors. Anonymized reference. 2021.

[37] Zaafar Ahmed, Muhammad Hamad Alizai, and Affan A. Syed. Inkev: In-kernel distributed network virtualization for dcn. *ACM SIGCOMM Computer Communication Review*, 46(3), jul 2018.

[38] Matteo Bertrone, Sebastiano Miano, Jianwen Pi, Fulvio Risso, and Massimo Tumolo. Toward an eBPF-based clone of iptables. In *Netdev 0x12, THE Technical Conference on Linux Networking*, Nepean, Canada, 2018. The NetDev Society.

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

[40] Cheng-Chun Tu, Joe Stringer, and Justin Pettit. Building an extensible open vswitch datapath. *ACM SIGOPS Operating Systems Review*, 51(1):72–77, 2017.

[41] Jakub Kicinski and Nicolaas Viljoen. ebpf hardware offload to smartnics: cls bpf and xdp. *Proceedings of netdev*, 1, 2016.

[42] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hxdp: Efficient software packet processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990, 2020.

[43] S. Bradner and J. McQuaid. Benchmarking Methodology for Network Interconnect Devices. RFC 2544, RFC Editor, March 1999.

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

[45] TRex Team. TRex realistic traffic generator, 2021.

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