eMPTCP: A Framework to Fully Extend Multipath TCP

Dian Shen, *Member, IEEE*, Bin Yang, Junxue Zhang, Fang Dong, *Member, IEEE, ACM*, and John C. S. Lui, *Fellow, IEEE, ACM*

***Abstract*—MPTCP provides the basic multipath support for network applications to deliver high throughput and robust communication. However, the original MPTCP is designed with limited extensibility. Various research works have tried to extend MPTCP to attain better performance or richer functionalities. These existing approaches either modify the kernel implementation of MPTCP, which involve considerable engineering efforts and may accidentally introduce safety issues, or control MPTCP via userspace tools, which suffer from restricted functionality support. To address this issue, we propose eMPTCP, an *easy-to-use* framework to *fully* extend MPTCP *without safety risks*. Internally, eMPTCP has a modular and pluggable model which allows operators to specify a comprehensive MPTCP extension as a chain of sub-policies. eMPTCP further enforces the policies through packet header manipulations. To ensure safety, eMPTCP is implemented using eBPF. Despite the stringent constraints of eBPF, we show that it is possible to implement an elaborated framework for a fully extensible MPTCP. Through verifying MPTCP in a number of real-world cases and extensive experiments, we show that eMPTCP is able to support a wide range of MPTCP extensions, while the overhead of eMPTCP operations in the kernel is in the scale of nanosecond, and the extra processing time accounts for only about 0.63% of flows' transmission time.**

***Index Terms*—Data center network, MPTCP, eBPF.**

I. Introduction

MULTIPATH transport has become a popular option in today's networks. Mobile devices usually have multiple wireless interfaces like Wi-Fi and cellular accesses [1], and it has become a norm for multihoming servers to have many parallel paths in data center networks [2]. In order to better exploit the multipath feature of networks, Multipath TCP (MPTCP) [3] was proposed to enable applications to simultaneously utilize several IP-addresses/interfaces for communication. With MPTCP, applications are able to use multiple paths concurrently to increase the aggregated capacity and to provide robustness when there is any link failure.

Despite the promising benefits of using MPTCP, the diversity of network traffic workloads and increasing performance requirements of applications significantly complicate its usage. To provide better performance or enhanced functionalities, there has been a wave of extensions over the native MPTCP, covering a wide array of use cases, including traffic scheduler [4], [5], [6], [7], [8], [9], [10], [11], path management [12], [13], [14], [15], [16], [17], and network-application co-design [18], [19], [20], [21], etc. For instance, as heterogeneous paths may cause under-utilization of the fast path and the degradation of MPTCP performance, Zhang, et al, [11] extended the traffic scheduler of MPTCP by developing an adaptive scheduler based on deep reinforcement learning. In order to improve the performance for small flows, MMPTCP [13] extended the standard MPTCP by modifying the path management module to randomly scatter packets in the network so as to exploit all available paths for small flows. Le and Nahum et al. [19] co-designed MPTCP with virtual machine (VM) migration to increase the service reachability in a cloud environment.

However, the native MPTCP[1] implementation is not designed for easy extensibility. Existing methods of implementing new extensions on the native MPTCP, including the modifications of its kernel implementation or using a userspace control module, have several undesirable drawbacks. First, to correctly modify the native MPTCP kernel code usually takes considerable amount of time and efforts, and the modification may not be compatible with new MPTCP releases. Second, by using a userspace control module (e.g., mptcpd [25]), the functionality and extensibility are highly restricted to the exposed interfaces, which is insufficient for many emerging scenarios. Recently, Extended Berkeley Packet Filter (eBPF) [26] emerges as a powerful technology to inject user-defined programs into kernel space. Viet-Hoang Tran and Olivier Bonaventure [27] have taken the first step toward extending transport protocols with eBPF. Using eBPF to extend MPTCP is a promising option, due to its safety guarantee, non-intrusiveness to the kernel, and ease of deployment.

Received 18 October 2023; revised 28 May 2024; accepted 21 September 2024; approved by IEEE/ACM Transactions on Networking Editor K. Lee. Date of publication 7 October 2024; date of current version 19 December 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62272101 and Grant 62402407, in part by the Natural Science Foundation of Jiangsu Province under Grant BK20230083, and in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization. The work of John C. S. Lui was supported in part by the Hong Kong Research Grants Council (RGC) under Grant GRF-14202923. *(Corresponding author: Bin Yang.)*

Dian Shen, Bin Yang, and Fang Dong are with the School of Computer Science and Engineering, Southeast University, Nanjing 210096, China (e-mail: dshen@seu.edu.cn; binyang@seu.edu.cn; fdong@seu.edu.cn).

Junxue Zhang is with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China (e-mail: zjx@ust.hk).

John C. S. Lui is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China (e-mail: cslui@cse.cuhk.edu.hk).

Digital Object Identifier 10.1109/TNET.2024.3469396

[1] MPTCP currently has two versions. MPTCPv0 [RFC6824] consists of a set of patches to the Linux kernel [22], the latest version of which is v0.96. MPTCPv1 is standardized by RFC8684 [23] and upstreamed to the Linux kernel recently. It is available to users using kernel version 5.6 or newer [24]. As eMPTCP does not impose any limitation on the MPTCP version, we will not make a distinction between MPTCPv0 and MPTCPv1 in this paper. Instead, we refer to both of them as the native MPTCP.

However, it remains an open question on how to use it to dynamically tune and fully extend MPTCP to best fit different users' needs. The challenges are summarized as follow:

Lack of flexibility. All existing methods to extend the native MPTCP are to handcraft a policy as a single monolithic program. With the increasing complexity of MPTCP control policies, it is difficult to know which building blocks of the policies are (in)appropriate for real-world dynamic and fluctuating workloads. Such an integrated, all-in-one monolithic model lacks the ability to fine-tune and dynamically combine modules of advanced control policies.

Limited functionalities. Current methods to extend MPTCP, either by userspace daemons or user-defined kernel extensions, are limited by the functionalities of native MPTCP stack. For example, current MPTCP and its extensions only work on end-hosts, so they have insufficient knowledge and controlability of the underlying network. Thus, current MPTCP extensions are restrictive in supporting emerging scenarios such as multi-tenant environment.

Simultaneously ensure safety and ease-of-use. Using eBPF to extend MPTCP kernel with a user-defined program can ensure safety because eBPF has a verifier to strictly check the safety and validity of the loaded program. However, eBPF also imposes many hard limits on the verifier-acceptable programs. Naively applying eBPF can be too restrictive to implement some legitimate MPTCP extensions in practice.

We believe such challenges significantly hinder experimentation and innovation in exploiting the multipath capability of networks, which motivate this research.

We propose eMPTCP, a flexible framework to extend MPTCP. This framework enables network operators to easily specify a chain of modular policies to dynamically control the behaviors of MPTCP at runtime. Extending MPTCP by eMPTCP offers the following benefits:

- **Modular and pluggable**. Instead of using a monolithic programming model, eMPTCP allows a modular specification of policies as a chain. Network operators can customize and dynamically plug their program into a chain of policies on MPTCP, without interrupting the running network services. These modules can be further shared and reused among multiple chains, thereby enhancing efficiency.
- **Adding new functionalities**. To achieve the full extensibility of MPTCP, eMPTCP extends MPTCP by a hybrid approach of direct kernel interaction and indirect packet manipulation. Thereby, it supports a wide range of MPTCP operations including controllable path establishment, traffic scheduler, etc. By allowing inspection and manipulation of network packets, eMPTCP can utilize the information from different layers of network protocols, yielding unique insights and exerting control beyond the end hosts. Specifically, we seek to add new functionalities to MPTCP, by investigating and innovating the usage of MPTCP in emerging scenarios such as the multi-tenant environment and proactive congestion control.
- **Higher pace of development**. With intent-based abstractions and safety-verified helper functions provided by eMPTCP, network operators can focus on essential policy development without worrying about the details and safety issues of the MPTCP kernel. Policies like traffic scheduling in eMPTCP are written and maintained in Python and run from userspace without safety risks.

eMPTCP delivers the above advantages by an implementation based on eBPF [26]. The key ingredients of eMPTCP include: (1) a selector-actor style policy chain, which allows operators to specify and plug in an advanced policy via a flexible combination of its building blocks; (2) A policy enforcer, which utilizes both direct, indirect, and hybrid approaches and provides a wide range of MPTCP control operations. (3) an intent-based abstraction along with a rich set of verifier-accepted helper functions.

We evaluate eMPTCP by implementing several representative MPTCP extensions. In particular, we seek to add proactive congestion control for MPTCP by embedding a customized algorithm that allocates credits for each subflow according to the bottleneck bandwidth. Compared with the default reactive congestion control algorithm of MPTCP, it achieves almost zero re-transmission under network variation. Furthermore, we investigate the usage of MPTCP in a multi-tenant cloud environment. By enabling MPTCP traffic generated from VMs to traverse through multiple physical links, we improve the throughput of baseline by up to 1.32×. We also enable some existing MPTCP extensions with eMPTCP. For path management, we extend the default path manager of MPTCP by using only one path for small flows and gradually adding subflows with user-defined parameters. The path manager can reduce the flow completion time of small flows by up to 32.1%. For the traffic scheduler, we implement an ECF-like [9] dynamic scheduler for a network with heterogeneous paths. Such a scheduler can be implemented easily using only tens of LoCs by eMPTCP and improves the application throughput by up to 1.41×. Throughout the evaluation, eMPTCP incurs only a small overhead in the level of nanoseconds on both servers and low-end devices such as Raspberry Pi, and the extra processing time accounts for as low as 0.63% of flows' transmission time. All source codes of eMPTCP and the use cases are publicly available on GitHub.[2]

II. Background and Motivation

A. Multipath TCP (MPTCP)

MPTCP is a transport layer protocol, which removes the single path limitation of conventional TCP. It enables the applications to simultaneously utilize several network interfaces for communication. Applications using MPTCP can benefit from higher aggregate throughput by exploring parallel communication paths, and achieve better robustness by seamlessly switching paths when link failures occur. It is an important protocol for critical environments like mobile communication, data center networking, etc. MPTCP is also emerging as a multipurpose next-generation transport protocol, which has the potential of replacing the current single-path TCP.

As defined by the MPTCP protocol, a separate path between the source and the destination is represented by a subflow. For example, if two communicating hosts and each has two network interfaces (and hence two IP addresses), MPTCP can establish up to four subflows between these two hosts. Among all the subflows, a primary subflow corresponds to the four-tuple TCP connection requested by the application. The

[2] https://github.com/chonepieceyb/mptcp_ebpf_control_frame

primary subflow is established first, followed by secondary subflows on the other paths. Subflows are said to have been established once their TCP connections are settled and are ready to send or receive data. If a path becomes inaccessible, its corresponding subflow is removed by MPTCP.

B. Extending the Native MPTCP

To achieve better performance or enhanced functionalities, there has been a number of extensions over the native MPTCP, which cover a wide range of use cases, including traffic scheduler [4], [5], [6], [7], [8], [9], [10], [11], path management [12], [13], [14], [15], [16], [17] and network-application co-design [18], [19], [20], etc. For example, the path-manager is the key component of MPTCP, which is responsible to decide when and which paths (or set of paths) should be used for the communication. The actual decisions about path establishment are application-specific. MPTCP by default provides four types of path-managers: `default`, `fullmesh`, `ndiffports` and `binder`. Unfortunately, all these path-managers are reported to be harmful to small flows in certain cases because it introduces additional cross-interactions with packet scheduler. Therefore, MMPTCP [13] attempted to extend native MPTCP with more intelligent path-managers. Furthermore, the native MPTCP suffers from performance degradation when there are multiple heterogeneous paths. Therefore, it is natural to extend the native MPTCP with an enhanced traffic scheduler that reacts to the network state change. Some representative traffic schedulers for MPTCP include ECF [9], BLEST [7] and STFT [10]. To adapt the native MPTCP to emerging usage scenarios, users have sought to extend MPTCP, such as in the multi-tenant environment [19], [20], cross-layer network design [18], [19], [20], [21], etc.

MPTCP can be extended using a userspace daemon, however, it has several drawbacks, including limited functionality, high interaction overhead, and no safety guarantee. For example, mptcpd [25] is a user space daemon that performs MPTCP path-management related operations. Currently, the latest version mptcpd v0.11 (released in August 2022) supports a set of functionalities such as path management. It has the following limitations. First, as a generic netlink solution, mptcpd has a strong coupling with the MPTCP kernel stack and can only rely on events and actions supported by the stack, limiting its functionality to path management. Second, the overhead of the generic netlink-based userspace solution is significantly higher. In mptcpd, each control action involves event-triggering in the kernel, handling in userspace, and finally calling a command API to enforce the control back to the kernel. We implemented a simple plugin using mptcpd to set a subflow to be a backup subflow and measured its processing overhead. This simple action takes at least 145μs on a high-end server, making it difficult to implement per-packet decision-making. Lastly, mptcpd plugins lack safety guarantees. Compared to eBPF-based solutions, mptcpd plugins written directly in C by users can lead to issues such as out-of-bounds memory access or unsafe termination due to programming errors, as they have not undergone static analysis and verification. These issues can potentially cause the mptcpd daemon to crash, impacting overall performance.

C. Extended Berkeley Packet Filter (eBPF)

Besides the conventional method of extending MPTCP, eBPF [28], [29] is an emerging powerful and general technology to extend the kernel, which allows custom programs to be safely executed within the kernel. eBPF works in several steps. First, a standard compiler (e.g., Clang-9) is used to turn eBPF programs into BPF bytecode, whose format is independent of the underlying hardware architecture. Then, the bytecodes i.e., the eBPF RISC instructions are compiled just-in-time (JIT) into the native machine instructions and finally attached to kernel functions.

To ensure that the attached program does not crash the running kernel, eBPF incorporates a verifier to statically check whether the program can be safely attached to the kernel. The verifier is executed every time eBPF loads a program to the kernel. The goal of the verifier is to prevent the program from accessing unauthorized memory and to guarantee that the execution of eBPF programs will always terminate. From our experience, it is not easy to pass an eBPF verifier, even for a simple program. In practice, users usually leverage restricted-C code to develop the in-kernel eBPF program and then compile it into eBPF bytecode. The verifier checks the validity of program by the compiled eBPF bytecode, rather than the original program. However, the bytecode-oriented verifying information cannot be directly correlated with eBPF programs, making it difficult for troubleshooting. In fact, this issue poses significant challenges in producing a verifier-acceptable program.

Besides, extending MPTCP by using eBPF provides a promising solution. Viet-Hoang Tran and Bonaventure [27] presented an enhanced MPTCP path-manager as one representative kernel extension using eBPF. Nonetheless, it remains an open question on how to dynamically tune and fully extend MPTCP to best fit different users' needs.

D. Motivation of eMPTCP

We summarize the following challenges of existing methods to extend the native MPTCP, which motivate our design.

First, all existing methods of extending MPTCP only support the monolithic model that the policy designers need to handcraft the policy into one single program. The limitations are: 1) Network operators are unable to easily implement, test, and tune an advanced MPTCP extension (e.g., scheduling, path management, congestion control modules, and the combination of them), which consists of multiple components. For example, the extension proposed by Han et al. [30] is composed of a coupled BBR congestion control algorithm, an adaptively redundant detector, and a predictive packet scheduler. The components in this extension can be dynamically tuned or substituted for different network conditions and workloads. Unfortunately, under the current setting, it is hard to know which building blocks of the extension work improperly in real-world systems, therefore inhibiting the policies from maximizing their performance gain. 2) It lacks the flexibility to combine and reuse the components of an existing MPTCP extension. For example, shared bottleneck detection modules [31] have the potential to be reused by both path manager and traffic scheduler. However, with current methods, developers need to implement multiple kernel modules or modify the MPTCP stack from scratch.

Second, the existing methods to extend MPTCP, either by kernel modules or userspace daemons using Netlink, are limited by the functionalities of the native MPTCP stack. In particular, the naive MPTCP struggles to support extensions that require adding new MPTCP options or packet header fields. For example, it is hard to add new congestion control algorithms such as proactive congestion control (PCC), due to the inability to obtain in-network "credit" information with MPTCP. Another example is that the naive MPTCP cannot support MPTCP-SBD [32] as it introduces a new MPTCP timestamp option. Moreover, current MPTCP and its extensions work on end-hosts only, hence lacking both knowledge and controllability of the underlying network out of the host. With the presence of some bottleneck links, the users of MPTCP can not take advantage of efficient communication over multiple paths, even though the end-hosts are multi-interfaced. Especially in the multi-tenant environment, all existing MPTCP extensions work only in the guest VMs and can not utilize the aggregated network bandwidth of hypervisors. Therefore, current MPTCP extensions are restrictive in supporting emerging scenarios.

In this paper, we utilize eBPF technology instead of intruding into the kernel code-base to address the above two challenges. The reasons are: 1) In production environments, standardized kernels are commonly employed, with the deployment of modified kernels generally discouraged. 2) It takes a long time for an extension (e.g., MPTCP-SBD) to be standardized and implemented into the mainstream kernel [33]. 3) The eBPF technology facilitates non-intrusive kernel extension. Compared to kernel modules, eBPF's verifiability ensures safety, accelerating the development, testing, and deployment cycles of new extensions. However, utilizing eBPF technology to extend MPTCP also presents a new challenge as follows.

Third, although the emerging eBPF technique provides an effective mean to inject a user-defined program into the kernel with a safety guarantee, it is still very restrictive to implement an MPTCP control policy with eBPF due to its safety validation. From our experience, there are many verifier-related issues that may hinder the development of MPTCP extensions. In fact, even some simple yet valid pseudo-C code might be rejected by the verifier after compiling into bytecode due to the implicit compiler optimization. The error information is bytecode-oriented and with poor readability, which further aggravates the difficulty of troubleshooting. For example, Listing 1 demonstrates a simple and correct code, but it fails to pass the eBPF verifier when compiled into bytecode. The verifier only provides the obscure error message: "dereference of modified ctx ptr R1 off=8 disallowed.". Considering that developing MPTCP extension programs from scratch using eBPF requires significant effort to address various validation issues for users, we aim to encapsulate our experience in tackling these issues, by providing intent-based abstractions and a rich set of easy-to-use MPTCP-related helper functions.

III. eMPTCP Design

Addressing the above challenges, eMPTCP aims to achieve the following goals:

First, eMPTCP needs to enable users to easily implement an MPTCP control mechanism in a modular and pluggable manner. eMPTCP allows network operators to divide complicated MPTCP extensions into some basic and reusable components. Together with the pluggable feature, network operators can dynamically tune and combine these components on the fly, without interrupting the running network services.

```
SEC("tc")
int scancb(struct __sk_buff *ctx) {
    ...
    for (int i = 1; i < 5; i++) {
        if (curr_id == i) {
            target = ctx->cb[i];
            break;
        }
    }
    ...
 }
```

Listing 1. Example of code that fails to pass the verifier.

Second, eMPTCP needs to support the extension of a wide range of MPTCP operations, including controllable path establishment, dynamic traffic scheduling, etc. Beyond that, eMPTCP should allow an operator to define new options and add new functionalities for emerging usage scenarios.

Third, eMPTCP needs to be user-friendly to network operators, so that they can focus on the essential policy development without safety concerns. Although eMPTCP is supposed to automatically guarantee the correctness of the execution by the eBPF verifier, it needs to hide the verification issues from users as much as possible.

A. Design Choices

Overall, eMPTCP utilizes eBPF by developing all policies as eBPF programs executed within the kernel, which enables eMPTCP to securely extend MPTCP. The eBPF program needs to be attached to a specific in-kernel hook to run. Different hooks offer distinct functionalities, which correspond to design choices of eMPTCP. The design choices are described as follows.

Indirect approach by packet manipulation. At present, the Linux kernel has incorporated hooks along the data path for packet processing and redirection, including XDP [28] for ingress path and TC [34] for egress/ingress path. With these mechanisms, MPTCP can be extended indirectly by manipulating MPTCP options in packets, as the MPTCP protocol design is built upon TCP by incorporating the MPTCP option set (type equals 30) into the TCP option field (III-C). This approach provides several advantages, such as increased deployment flexibility beyond end-hosts and the capability to inspect packets for extracting detailed low-level protocol and in-network information. However, the indirect approach cannot support functionalities that require direct interaction with the protocol stack, such as congestion control and traffic scheduling.

Direct approach by in-stack kernel interaction. Currently, the Linux kernel supports a set of hooks called STRUCT_OP [35] which enables implementing kernel structure with a set of eBPF programs rather than the kernel module. For example, with STRUCT_OP, `tcp_congestion_ops` can be implemented with eBPF programs, which enables eBPF-based congestion control algorithms. Additionally, the MPTCP development branch utilizes STRUCT_OP to facilitate eBPF-based traffic scheduling. Taking advantage of this mechanism, MPTCP can be extended directly to accommodate user-defined congestion control algorithms and traffic scheduling strategies (III-D). Nevertheless, this direct approach has functional and flexible limitations.

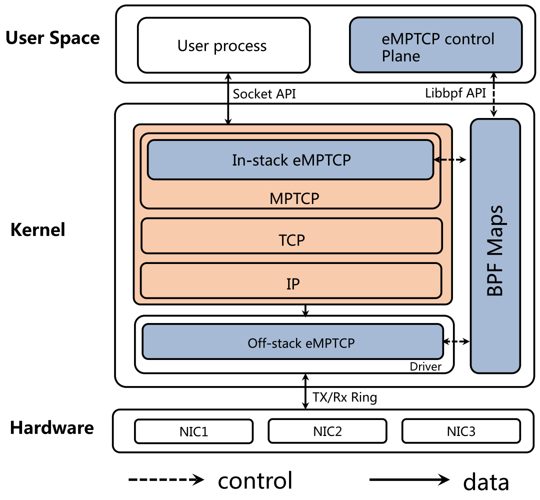


Fig. 1. eMPTCP and the networking stack.

Initially, its functionality is confined to the kernel structure exposed by the kernel through STRUCT_OP. The restricted decision context accessible to eBPF-based algorithms hampers access to in-network information for integrating new functionalities. For instance, both eBPF-based congestion control and traffic scheduling algorithms lack access to `sk_buff` for extracting in-network credits (III-E). Furthermore, eBPF-based algorithms, once attached, behave akin to a monolithic kernel module, making division into discrete building blocks challenging for fine-tuning and configuration at varying granularities.

The hybrid approach of eMPTCP. To achieve the mentioned design goals, eMPTCP adopts both direct and indirect approaches. Notably, eMPTCP designs a hybrid mechanism to enable the collaborative integration of these approaches when designing extensions to enhance MPTCP with new functionalities. eMPTCP attaches eBPF programs to both packet processing hooks and in-stack kernel structure hooks. According to the locations and functionality of eBPF hooks used by eMPTCP, eMPTCP is divided into in-stack and off-stack parts. As depicted in Fig. 1, from the network layering perspective, the off-stack is attached to the XDP/TC hook and lies between the driver and the network stack. Thus, the off-stack part of eMPTCP can oversee and manipulate the whole IP packets before they are processed in the network stack and operate MPTCP operations indirectly. On the other hand, the in-stack part is attached to the STRUCT_OP hook and is located in the MPTCP stack to directly control it. Additionally, the two parts collaborate with each other through the BPF MAP mechanism. It should be noted that eBPF is not the only method for implementing in-stack actors. We choose it due to its superior safety and programmability compared to other approaches, such as netfilter or netlink, which is crucial for implementing new MPTCP features.

The design overview of eMPTCP is shown in Fig. 2. eMPTCP delivers its desirable features through the following key designs:

- **Selector-actor style policy chain.** In order to facilitate the modular development of MPTCP extensions, eMPTCP employs a policy chain abstraction for MPTCP extensions. The policy chain is composed of sub-policies and two types of sub-policies are supported, including selectors for inspecting and filtering packets and actors for performing MPTCP operations. Furthermore, network operators could compose arbitrary numbers and types of selectors and actors at runtime to achieve flexibility.
- **Policy enforcer based on a hybrid approach.** To ensure comprehensive support for MPTCP operations, eMPTCP employs a hybrid approach. In this design, eMPTCP deploys policies prior to the network stack for packet manipulation (indirect approach). Additionally, eMPTCP integrates policies into the stable extension interfaces provided by the kernel, allowing direct control over the MPTCP (direct approach). Moreover, eMPTCP introduces a data-sharing mechanism to facilitate collaboration among policies located at off-stack and in-stack, thereby facilitating the emerging new functionalities of MPTCP.
- **Intent-based abstraction.** eMPTCP encapsulates and provides a rich set of APIs and helper functions for the ease of policy chain manipulation and policy development while ensuring safety.

B. Selector-Actor Style Policy Chains

In order to support the modular implementation of MPTCP policies, two factors need to be considered.

Flexibility. Dividing a complex policy into several sub-policies provides flexibility. By modifying and configuring an arbitrary component of a complicated policy at runtime, network operators can perform fine-tuning on the designed MPTCP extension. Furthermore, the modular design allows the sharing and reuse of the sub-policies for different extensions. For example, an MPTCP traffic scheduler can utilize the implementation of the “shared bottleneck detection module” component in the path-manager extension. Furthermore, using a subset of the existing policies or a combination of them can generate a variety of new MPTCP extensions easily.

Granularity. The second question is how to and in what granularity to select the relevant MPTCP connections and on which the policies are enforced. The control policy can be enforced on various granularity, including connection level, sub-flow level, or even packet level. For example, a path manager works on each subflow of an MPTCP connection; the traffic scheduler works for specific packets or subflows.

Considering both flexibility and granularity, eMPTCP uses a selector-actor style policy chain for designing MPTCP extensions. As Fig. 2 shows, this design decouples the policy chain into two functionally independent components: (1) a selector chain and (2) an actor chain. And they are composed of sub-policies implemented as an eBPF program called selectors and actors respectively. The selector chain filters unrelated events, and forwards the related ones to specific action chains. In terms of functional design, a typical selector is a Connection selector which checks the 4-tuple and MPTCP token to handle only the desired MPTCP connections. Table I shows a list of supported selectors of eMPTCP. Note that selectors are also chainable. The network operator can specify an arbitrary number of selectors with logic operators like `AND` or `OR` to combine them. This multiple expression combiner is an efficient way to select different granularities of inputs.

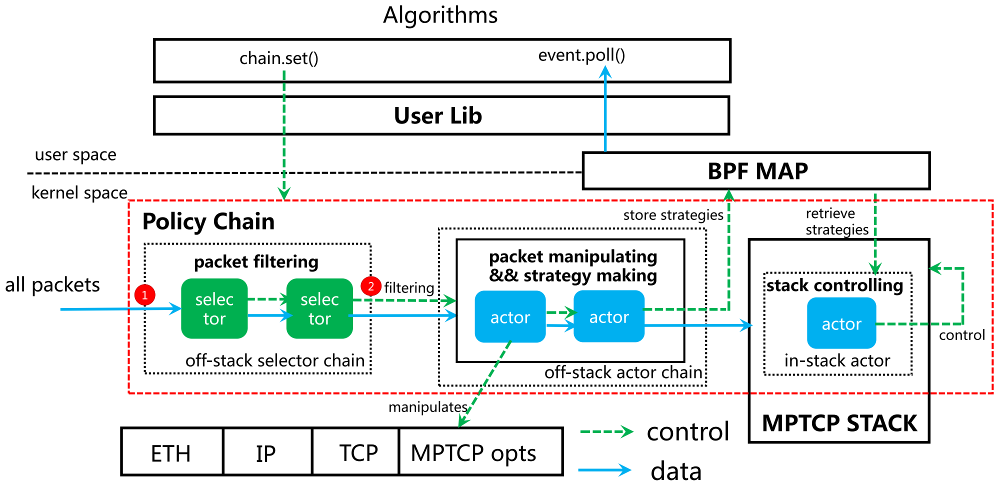


Fig. 2. eMPTCP overview.

TABLE I
SELECTORS SUPPORTED BY eMPTCP

Selector name	Functionality
`mptcp_conn`	Filter packets by the token of MPTCP connection.
`subflow`	Filter packets by the TCP 4-tuple.
`ip_pair`	Filter packets by a `(src,dst)` pair.
`src/dst`	Filter packets by source or destination IP address.
`sequence`	Filter packets by Data Sequence Number or Subflow Sequence Number
`packet_type`	Filter packets by type, e.g., MPTCP SYN, Data ACK, etc,.

For example, by combining the Port selector and Connection selector, operators can identify the MPTCP connections belonging to the same server. Then the actor chain performs operations that should be taken on the selected events. The implementation of selectors and actors is modular and can be chained with arbitrary numbers and all the sub-policies can be added or removed from chains at run-time, without interrupting the running services. Note that combining selectors and actors in a chain can form a sophisticated mechanism with high flexibility and various levels of granularity. Such a selector-actor model further benefits eMPTCP with improved performance by filtering irrelevant events as soon as possible, and forwarding the relevant ones to the suitable actor chain. The filtered events can be ignored or dispatched to the default handler, such as kernel stack.

With the general architecture of the policy chain, another important problem that needs to be solved is how to deploy the policy chain. Conventionally, different eBPF programs can be chained through tailcall mechanism as a complete eBPF program, which overcomes the limitation that a hook can only be attached by a single eBPF program at any given time. However, this direct approach is not suitable for the design of eMPTCP, and it introduces two significant challenges. Firstly, the tailcall mechanism cannot be directly used to chain policies between the off-stack and in-stack parts. This limitation arises because the tailcall mechanism only works for eBPF programs attached to the same hook. Secondly, the selector-actor architecture cannot be deployed in the in-stack part due to the absence of tailcall support in the hook used by the in-stack part, resulting in the lack of flexibility and granularity in the in-stack part.

Taking into account the two challenges, eMPTCP employs a combination of direct and indirect approaches to deploy the policy chain. In general, the main components of the selector-actor policy chain are deployed in the off-stack part. The off-stack part establishes connections between all the selectors and the majority of actors using the tailcall mechanism. Additionally, several actors are deployed in the in-stack part to directly control the MPTCP stack and they establish indirect connections with the off-stack part through BPF MAP. In this design, the policy chain is triggered by packets. The selector chain filters relevant packets and redirects them to the off-stack actor chain, which performs MPTCP operations indirectly through packet manipulation. Additionally, the actors within the off-stack actor chain develop strategies and store them in the BPF MAP. Subsequently, the in-stack actors retrieve these strategies from the BPF MAP and directly control the MPTCP stack. This design allows eMPTCP to compose sub-policies that are distributed across different locations as a policy chain. It is worth mentioning that the selector-actor policy chain is a general structure. It can be utilized to extend other protocols as required. eMPTCP supports users to develop their own selectors/actors easily and provides pre-defined, MPTCP-specific, and verifier-acceptable selectors/actors.

C. Indirect Packet Manipulation

The off-stack part of eMPTCP utilizes packet manipulation to support a wide range of MPTCP operations. Such a design is based on the rationale that MPTCP adds a new set of options to the TCP option field, which are exchanged between MPTCP-enabled end-hosts. Therefore, modifying the MPTCP-specific options in the packet header can alter the MPTCP behaviors.

Defined by the standard MPTCP protocol, the main MPTCP options include `MP_CAPABLE`, `MP_JOIN`, `MP_DSS`,[3]

[3] Currently, we do not modify MP_DSS in existing extensions. We name it here for its potential usage and eMPTCP's ability to manipulate it.

TABLE II
ACTORS SUPPORTED BY OFF-STACK PART OF eMPTCP

Actor name	Parameters	Description
rate_limit	Rate	Update the recv_win of ACKs of a subflow to control the sending rate.
set_backup	Priority	Add MP_PRIO option to packet to set or remove the current subflow
blk_subflow	N/A	Remove and store MP_ADD_ADDR to avoid creation of subflows.
add_subflow	N/A	Add MP_ADD_ADDR to packet and enable creation of subflows.
get_connect	N/A	Parse MP_CAPABLE option t send MPTCP keys to event queue.
get_subflow	N/A	Parse MP_JOIN option to send subflow token to event queue.
sched_template	N/A	Template for actor making traffic scheduling strategies and cooperating with in-stack actor
cong_template	N/A	Template for actor making congestion control strategies and cooperating with in-stack actor
record	Metric	Record specific metrics of selected packets such as RTT, flow size and etc,.

ADD_ADDR, REMOVE_ADDR, MP_PRIO, MP_FAIL, MP_FASTCLOSE and etc. Through manipulation on these options, eMPTCP can provide control on the subflow-level behaviors of MPTCP. For example, removing the ADD_ADDR packets from the communication peer will inhibit MPTCP from establishing new subflows, and re-inject that packet would automatically trigger MPTCP to establish new subflows.[4] Beyond that, rate-limiting of MPTCP subflows is implemented by modifying the receive window (RWND) on incoming ACKs. The rationale behind this design is that the protocol stack uses min(CWND, RWND) to limit how many packets it can send. This enforcement of RWND provides an upper bound to rate limit a flow in networks. This is feasible because, as RFC6824 and RFC8684 have mentioned, a host should maintain the connection-level receive window as well as all subflow-level windows.

Table I demonstrates the selectors supported by eMPTCP and their selection granularity. Table II summarizes the actors supported by eMPTCP and the corresponding packet manipulation. The selectors and actors listed in the tables are independent eBPF programs designed to perform specific actions within a policy chain. Table II also lists templates for creating new actors or selectors, such as the sched_template, which acts as a template for actors developing scheduling strategies in the off-stack policy chain. It should be noted that the actors listed in Table II can be triggered from either the sender (i.e., egress path) or receiver (i.e., ingress path) side, affecting the protocol stack of the receiver. For instance, regardless of the triggering location, the blk_subflow actor blocks the sender's address from the receiver, preventing subflow establishment, while the add_subflow actor enables the receiver to rediscover the sender's address. The method of packet manipulation enables eMPTCP to support a rich set of functionalities. For example, it enables MPTCP to interact with other cross-layer network protocols.

Note that MPTCPv0 and MPTCPv1 have some differences in the protocol design and eMPTCP is expected to handle the difference automatically. One representative example is to work around ADD_ADDR. Specifically, MPTCP utilizes the ADD_ADDR option to announce additional addresses (and, optionally, ports) on which a host can be reached. The mechanism of the ADD_ADDR option is quite different between MPTCPv0 and v1. In MPTCPv1, there are some additional mechanisms: 1) MPTCPv1 introduces ADD_ADDR ack for reliable transmission of this option. 2) MPTCPv1 adds additional information (8 octets of truncated HMAC) with the ADD_ADDR option for authentication. eMPTCP handles the additional mechanisms. First, to block the ADD_ADDR Option, in MPTCPv1, after filtering the ADD_ADDR option, the peer won't send ADD_ADDR ack back because the ADD_ADDR was not received. The sender will keep retransmitting the ADD_ADDR if the ADD_ADDR ack is not received within a specified timeout (configurable with sysctl).

There are two methods to solve this issue:

1) Filtering subsequent retransmitted ADD_ADDR. To keep the extra remote addresses invisible to the host, a direct way is to filter the subsequent retransmitted ADD_ADDR. This approach is easy to implement and suitable for short-term blocking. It is also convenient for recovering the ADD_ADDR. We can just remove such blocking, and the retransmitted ADD_ADDR can be received by the peer.

2) Constructing the ADD_ADDR ack. The second method is that, when blocking the ADD_ADDR, we also construct the corresponding ADD_ADDR ack and send it to the peer. Constructing ADD_ADDR ack can be implemented through the eMPTCP actor. In detail, the actor attached to the XDP/TC hook constructs the ADD_ADDR ack based on the originally received ADD_ADDR. It swaps MAP PORT, sets the Echo-Flag, removes the truncated HMAC, and recalculates the checksum. After that, the actor sends the ADD_ADDR ack back to the sender through XDP/TC packet redirecting. Although this method prevents the ADD_ADDR retransmission, it requires an additional mechanism to recover the blocked addresses. The trick is to reconstruct the ADD_ADDR packet. To achieve this goal, we duplicate the latest ACK and inject the previously blocked ADD_ADDR information (including the authentication information). In this manner, the constructed packet will be accepted by the kernel stack. Note that the duplicated acks won't affect the congestion window. This is because MPTCP treats duplicated acks carrying any MPTCP option except for DSS options as control packets rather than congestion signals, according to RFC 8684.

It is worth noting that supporting path management through packet manipulation without modifying the kernel relies on the existing path management algorithm to obtain manageable paths. In this paper, we build eMPTCP's path management upon the fullmesh. Although this approach seems hacky, it has three advantages. Firstly, from a deployment perspective, it allows us to deploy eMPTCP as a middlebox, such as within a hypervisor. Secondly, from a compatibility perspective, this method does not depend on the specific implementation of the protocol stack or the particular hooks within the protocol stack. Finally, this approach has zero intrusion into the kernel.

[4] The behavior of establishing subflows is based on the assumption that both ends use fullmesh as path management algorithm.

D. Direct Kernel Stack Interaction

The ability to directly interact with the kernel stack is necessary to extend traffic scheduling and congestion control of MPTCP. Currently, this direct control can be achieved by attaching eBPF programs to the STRUCT_OP hooks which are extension interfaces exposed by the kernel including `mptcp_sched_ops` for MPTCP traffic scheduling and `tcp_congestion_ops` for TCP congestion control. Take MPTCP traffic scheduling as an example, an eBPF-based traffic scheduler can be attached to the STRUCT_OP of `mptcp_sched_ops`. Once attached, when MPTCP starts to send a segment, the associated eBPF programs are invoked and make decisions on selecting one or multiple subflows for transmitting the segment or potentially deferring the transmission. A straightforward use of this technique is to directly integrate traffic scheduling algorithms like min-RTT, BLEST [7], and ECF [9] into the eBPF programs. However, as described in Section III-B, this approach fails to leverage the design of the policy chain and lacks flexibility and granularity because of the failure of deploying the policy chain in the STRUCT_OP hook.

eMPTCP approach: eMPTCP adopts a strategy formulation and execution separation approach where the eBPF programs attached to the STRUCT_OP hooks are regarded as in-stack actors. The whole approach consists of three steps. Firstly, the in-stack actors collect metrics from the MPTCP stack and store them in BPF MAP. Secondly, the off-stack actors formulate strategies, such as traffic scheduling strategies, for the specific algorithm based on the collected metrics. The strategies are then stored in BPF MAP by the off-stack actors. Finally, the in-stack actors execute actions directly based on strategies retrieved from the BPF MAP. This design empowers eMPTCP to have direct control over the MPTCP stack as a straightforward approach while providing two benefits. First, it provides flexibility and granularity by leveraging the policy chain design of eMPTCP. For example, it is possible to specify and fine-tune traffic scheduling extensions at runtime for different MPTCP connections on demand. Second, compared with collecting metrics by optional tracing technologies like kprobe, it eliminates redundant computations and function calls by integrating metric collecting logic and taking action logic in one place.

E. Hybrid Policy Enforcer

The native MPTCP reveals limitations in emerging scenarios. For example, Xu et al. [20] demonstrate that in a multi-tenant scenario, MPTCP deployed in virtual machines (VMs) fails to detect the presence of multiple underlying physical links. Consequently, there is a need to redesign the MPTCP stack to effectively detect this information and respond accordingly. Another example is proactive congestion control [36]. Proactive congestion aims to anticipate and prevent congestion in advance by taking preemptive measures, for example, pre-allocating bandwidth to network flows in the form of credit. This approach yields improved performance, especially in data-center networks. However, the current MPTCP stack in the Linux kernel adopts traditional reactive congestion control architecture, which responds to congestion after it has already occurred, based on acknowledgments (ack clock). To enable proactive congestion control for MPTCP, modifications to the Linux kernel implementation are necessary, as the current extension interfaces exposed are designed for reactive congestion control.

We argue that supporting the addition of new functionalities to MPTCP with minimal engineering effort, particularly without modifying the kernel, is crucial for the advancement of the protocol. To achieve this kernel-modification-free extension manner, two necessary conditions need to be met. First, it requires the ability to control the protocol stack, such as controlling the scheduling procedure of MPTCP. Second, it requires the ability to perceive information beyond the existing protocol stack, such as the additional data added to packets by in-network switches. However, neither the indirect approach nor the direct approach fulfills both requirements. The indirect approach of packet manipulation fails to control the procedure of congestion control and traffic scheduling. The direct approaches of the eBPF module lack information for these new functionalities in the control block and decision context. For instance, without the necessary codes in the MPTCP stack to extract credit information, user-defined traffic scheduling, and congestion control extensions using direct approaches are unable to effectively perform proactive transportation.

eMPTCP approach: eMPTCP employs a hybrid policy enforcer, utilizing the policy chain design of itself, which combines both indirect and direct approaches to add new functionalities to MPTCP without modifying the kernel. In the policy chain of eMPTCP, the off-stack actors based on packet manipulation of indirect approach are capable of extracting beyond-stack information through packet inspection, for example inspecting "credit" information added by in-network switches. Subsequently, other off-stack actors can develop strategies based on this extracted information. Finally, the in-stack actors directly control the stack according to the strategies. By coordinating the off-stack and in-stack actors in the policy chain, which integrates both indirect and direct approaches, eMPTCP is able to effectively meet the aforementioned two requirements. To illustrate how adding new functionalities of MPTCP benefits from eMPTCP, we take proactive congestion control as an example. The policy chain enabling proactive congestion control consists of the following components: (1) MPTCP selector. An off-stack selector selects target MPTCP connections. (2) Subflow selector. An off-stack selector selects subflows belonging to the same target MPTCP connection (3) Credit auditor. An off-stack actor inspects the packet and extracts credits. (4) Traffic Scheduler. An off-stack actor that devises the traffic scheduling strategy depends on the credits and stores the strategy to BPF MAP (5) Congestion Controller. An in-stack actor that sets the congestion window to a fixed value. (6) Traffic strategy executor. An in-stack actor directly selects subflows to send a segment or defers the transmission according to the strategy. This policy chain enables proactive congestion control for MPTCP by extracting credits in the off-stack part and controlling the MPTCP stack in the in-stack part.

F. Intent-Based Abstraction

In order to accelerate the development of MPTCP extensions, eMPTCP provides a rich set of intent-based abstractions. First of all, eMPTCP incorporates a set of helper functions

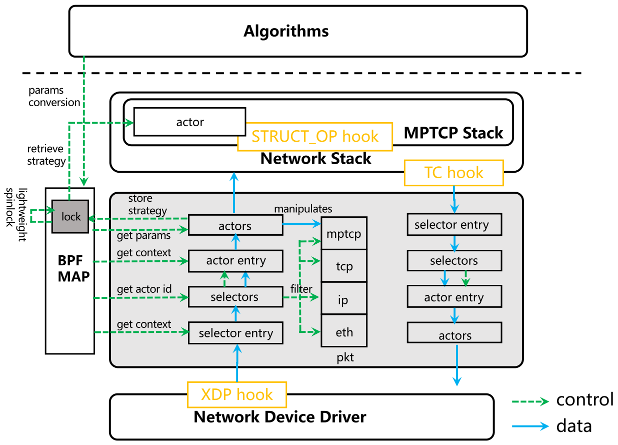


Fig. 3. eMPTCP implementation.

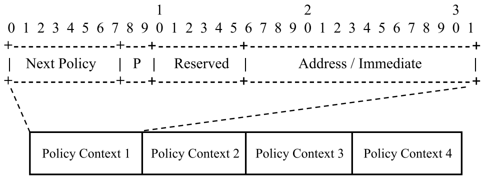


Fig. 4. Definition of the chain context. The chain context is implemented using packet metadata, `xdp_md` for XDP hook and `tc_cb` for TC hook. Next Policy is used to identify the next actor/selector. P denotes the type of parameter, which can be either Immediate (directly embedded in the context) or Address (parameters that need to be read from the BPF map).

to customize the combination of policy chains, e.g., adding, removing, or inserting a selector or an actor to an arbitrary chain. With the provided interfaces, operators can specify their desired control policy as a chain of user-defined programs. eMPTCP also provides a rich set of easy-to-use MPTCP-related helper functions to encapsulate the policy enforcers. The underlying implementation details of these helper functions are transparent to users and they have all passed the strict eBPF verifier. This design greatly eases the adoption of eMPTCP, and it decouples the policy design from underlying kernel execution. Network operators can focus on the policy essentials without worrying about the details and safety issues of the MPTCP kernel.

IV. eMPTCP Implementation

As shown in Fig. 3, the implementation of eMPTCP is primarily based on eBPF technology. In this section, we discuss the details of how to implement eMPTCP and share our experience in tackling various verifier-related issues when using eBPF to implement a complicated framework.

A. Policy Chaining Using eBPF Tail Calls and BPF MAP

The functional logic of eMPTCP is to disseminate the user-defined MPTCP control policy into multiple small building blocks locating in off-stack part and in-stack part. eMPTCP chains the building blocks in the off-stack part using the eBPF tailcall mechanism and connects the off-stack chain and building blocks in the in-stack part using BPF MAP. More specifically, first, Each building block of the policy is implemented by an eBPF program and they are analyzed and loaded independently which reduces the analysis complexity of the verifier and helps to pass the eBPF restrictions on program sizes. An eMPTCP program in off-stack part supports controlling both directions of egress and ingress network traffic. For ingress traffic, we attach eBPF programs to eXpress Data Path (XDP) [28], and for ingress traffic, we attach them to Traffic Control (TC) [34]. Second, to support the run-time combination of building blocks in off-stack parts, operators need to describe how and in what order these sub-policies are to be chained. The description of policy chaining is defined in the data structure called chain context as depicted in Fig. 4. In eMPTCP implementation, the chain context is an array of 4-byte data. The first byte represents the next sub-policy to be called. The second to fourth bytes represent the parameters of the current sub-policy. Furthermore, the context is stored as the metadata (`xdp_md` for XDP, and `cb` for TC, respectively) of packet data structure in the kernel. The entrance of the chain, either a selector or an actor, parses the metadata, acquires the index of the next sub-policy, and then queries the `prog_array` of eBPF tail calls to locate the next sub-policy. The sub-policies can be reused and combined dynamically by customizing the context metadata. Third, we attach a pre-defined eBPF program as an actor to STRUCT_OP hooks in the in-stack part. The actors implement the interfaces exposed by the kernel stack including `tcp_congestion_ops` and `mptcp_sched_ops` for congestion control and traffic scheduling respectively. Additionally, the behavior of in-stack actors is affected by the chain in the off-stack part using BPF MAP. For example, a traffic scheduler strategy maker in an off-stack chain develops a traffic scheduling strategy by every ack and stores the strategy in BPF MAP. The actor attached to `mptcp_sched_ops` performs actual traffic scheduling by retrieving the strategy for BPF MAP. eMPTCP also supports user-defined in-stack actors by encapsulating such connection mechanisms into ease-of-use APIs. The details of how to share data between the in-stack part and the off-stack part will be described in the next part.

It is worth noting that policy chains introduce additional overhead while facilitating modularity and scalability. The overheads are caused by storing and processing the chain context information. However, the overhead is quite small, because of the full use of existing data structures (XDP and TC's packet metadata) and the carefully designed policy chain context data structure.

B. Data Sharing Among Different Sub-Policies

Data sharing servers for three purposes. First, it avoids redundant computation after dividing a sophisticated extension into multiple sub-policies in a chain. Second, it enables the combination and connection between the off-stack part and the in-stack part of eMPTCP. Third, it enables the communication between userspace programs and the kernel functions. The promote challenge of data sharing lies in the coordination of multiple access to the same data from different programs running in different contexts. Specifically, the eBPF programs in off-stack part execute in softirq context while the eBPF programs in in-stack part execute within the protocol stack context. Addressing this issue, eMPTCP utilizes different mechanisms to share intermediate results or control parameters among sub-policies in different scenarios.

In the off-stack part, the sub-policies within the same chain are invoked and executed sequentially on the same CPU, because the eBPF programs connected by the tailcall still execute in the same softirq context. Thus, ensuring the independence of data between different CPUs becomes a primary concern. eMPTCP employs two mechanisms to achieve this objective. First, if the shared data is small enough, e.g., less than 2 bytes, it can be stored inline in the last two bytes of the chain context which is stored in per-packet metadata. Such a method is cost-efficient and avoids extra storage or memory access. Alternatively, if the shared data is large, we use per-cpu BPF MAPs to realize the data sharing. The per-cpu BPF MAPs are specific types of BPF MAP including per-cpu hash and per-cpu array which maintains data in each CPU independently. Using per-cpu BPF MAPs, sub-policies in the same policy chain operating on each packet can share data without concurrency issues. It should be noted that the data sharing through BPF MAP requires the reuse of MAP file descriptor of the same map among eBPF programs. To do so, we use eBPF `bpf_obj_get()` system call to obtain a file descriptor of BPF MAP and then use `bpf_map_reuse_fd()` function to replace where the same BPF MAP is used in different programs.

Unfortunately, when it comes to the connections between the off-stack part and the in-stack part, the eBPF programs in these different parts are not invoked within the same context and may run on different CPUs. This necessitates additional concurrency control mechanisms to avoid data races, as the solutions involving packet meta and per-CPU BPF MAPs are not applicable in this scenario. Commonly, BPF MAP provides common data structures such as HASH and ARRAY that can be used by eBPF programs running on different CPUs. These maps, unlike per-CPU BPF MAPs, maintain a single copy of the data that is accessed by eBPF programs across different CPUs. The HASH and ARRAY can be accessed in two patterns without data concurrency issues. Firstly, different eBPF programs access the map through the atomic replacement or deletion of the entire element using the `bpf_map_update_elem()` or `bpf_map_delete_elem()` helper functions. However, the overhead of constructing the entire elements each time is significant. Thus, the eBPF programs prefer to adopt the second approach. That is, they first retrieve the address of the target element by calling `bpf_map_lookup_elem()` helper function and then access the element directly with the address. However, multiple accesses to the same BPF MAP element by address are not directly protected by eBPF. Therefore, eBPF provides `bpf_spin_lock` mechanism. To use `bpf_spin_lock`, a lock field can be integrated into the element and the lock can be locked and unlocked on an element-wise basis using the `bpf_spin_lock()` and `bpf_spin_unlock()` helper functions, respectively. However, when using spin locks, it is advisable to avoid overly complex critical sections to prevent significant performance degradation.

eMPTCP approach: Considering the above factors, eMPTCP employs a flag-based competition mechanism. This mechanism aims to maintain a simple critical section, minimizing the time of blocking eBPF programs, particularly the packet processing program in the off-stack part. Initially, eMPTCP integrates a flag field as well as the lock into each element. And the spinlock is exclusively utilized to protect this flag field rather than the whole element. eMPTCP regards the off-stack actors as the consumers and in-stack actors as the producers. In a data-sharing procedure, multiple consumer programs contend for the flags, with only one program being successful in acquiring the flag. The winner of the contention then unsets the flag, exits the critical section, and proceeds with the remaining operations of formulating strategies. On the other hand, the consumer programs that fail to obtain the flag just abort the operation. The operation can be performed only if the flag is set by the producer program after it takes action according to the strategies. The competition for the flag is protected by the `bpf_spin_lock`. Take the traffic scheduling extension as an example, the sub-policy of traffic schedule making the traffic scheduling strategy is the consumer and the in-stack actor of traffic strategy executor performing actual traffic scheduling is the producer.

When it comes to communication between userspace programs and in-kernel eBPF programs, eMPTCP uses different mechanisms including per-cpu BPF MAPs, spinlock, and raw bpf system calls depending on the required level of concurrency control. For example, in certain scenarios where the userspace program solely updates the entire element while the eBPF program in the kernel only reads it, the default usage of bpf system call which provides concurrency control based on Read-Copy-Update (RCU), is sufficient. It is important to note that the MAP might be automatically destroyed if no program in the kernel is using it. To prevent the MAP from unintentionally being deallocated, we pin the BPF MAP to the BPF Virtual File System (VFS). BPF VFS is actually not a real file system, it only keeps the MAP alive by always referring it, incurring a small overhead.

C. Different Kinds of Packet Manipulation

The off-stack part of eMPTCP exerts a fine-grained control on MPTCP through packet manipulation. Some representative manipulations are:

(1) Modifying an existing MPTCP option. This kind of operation requires no change in the length of the header space. eMPTCP provides a set of inline helper functions to obtain pointers to header options of different protocols, such that the user-defined program can access the packet directly and modify the desired header field. To ensure consistency, a helper function is evoked to update the checksum.

(2) Removal of an MPTCP option. eMPTCP performs the removal of an option by overriding the option with `NOP` rather than shrinking the length of header space which introduces additional overhead. Thus, the removal operation reuses the packet modification helper functions, with the difference that the specific option is always modified by value `NOP`.

(3) Injection of a new MPTCP option. Since eBPF does not provide a native API to increase the length of a packet header, we implement the operation in three steps. First, increase the length of the packet by eBPF `adjust-header-room` helper functions. Second, move the original packet header data forward and reserve the space for injection of the new options.

Finally, write the MPTCP option into the reserved space and update the checksum. Similarly, we provide this functionality as an inline helper function to simplify the usage and expand the original eBPF helper functions.

It is worth noting that reinjecting options need to consider the MTU (Maximum Transmission Unit). Currently, there are several situations where MPTCP options may need to be injected. First, when the receiver gets a packet and needs to inject new MPTCP options into it, such as the ADD_ADDR option to restore a subflow. In this case, MTU does not need to be considered. Second, when adding new MPTCP options at the sender side, such as implementing the MPTCP-SBD [32] and including a new MPTCP timestamp option, this situation is similar to packet encapsulation and does require MTU considerations. The solution involves slightly adjusting the connection's MSS (Maximum Segment Size). Lastly, due to the limited TCP option space, a completely new packet may be needed to inject MPTCP options. The trick here is that eMPTCP leverages or duplicates the latest packets or ACKs to piggyback the option values. With correct `timestamp` and `checksum`, the packets will be accepted by the MPTCP stack. Moreover, the duplicated MPTCP Data ACKs won't affect the congestion window, according to RFC 8684.

D. Verifier Acceptable Helper Functions

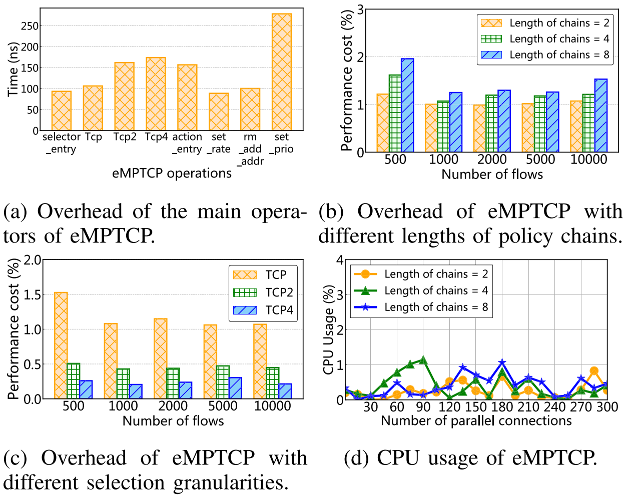
eMPTCP accepts standard user-defined eBPF programs as customized policies (actors or selectors). Beyond the basic eBPF helper functions, eMPTCP has provided a wide range of helper functions such as increasing the MPTCP header space, acquiring the specific MPTCP option, adding a new MPTCP option, and, most importantly, a set of functions to manipulate the policy chain. These helper functions are all intent-based and eBPF verifier acceptable. Thus, it significantly simplifies the development of customized policies, allowing operators to focus on designing the policy essentials.

V. Evaluation

In this section, we first evaluate the performance overhead of using eMPTCP in practice. Then we present several real-world MPTCP extensions implemented by eMPTCP and evaluate their performance.

Testbed. The testbed we use in the experiments consists of 7 servers, each of which is equipped with two Intel(R) Xeon(R) E5-2630 v4 CPUs (12 cores) and 128GB of memory. Each server is equipped with 3 10Gbps Broadcom Network Interface Card, and are connected through a Mellanox 40Gb switch. The internal network is considered to be non-blocking, and a similar setup is used by existing research [37]. Beyond high-end servers, we also deploy and test eMPTCP on low-end devices. In the test cases, we use Raspberry Pi 4B as the representative device. The Raspberry Pi 4B we use in the experiment is equipped with a Cortex-A72 (ARM v8) 1.5GHz CPU (4 cores), and 8GB memory. We use its WiFi and wire Ethernet interfaces under 300Mbps speed.

MPTCP setup. The different parts of eMPTCP have varying requirements for the kernel and the implementation of the MPTCP protocol stack. Specifically, the off-stack part needs the kernel to support XDP and TC hooks, along with corresponding helper functions. The off-stack part does not impose additional requirements on the protocol stack. For the in-stack part, eMPTCP relies on the STRUCT_OP hook to implement the in-stack actor. Therefore, the kernel's eBPF must support STRUCT_OP, and the protocol stack must accommodate STRUCT_OP hooks. Unless otherwise stated, the subsequent experiments are conducted based on MPTCP V0.96. For baseline MPTCP, we turn off MPTCP header checksumming to reduce unnecessary CPU overhead and use min-RTT as traffic scheduler, fullmesh as path management, and cubic as congestion control. We set receive buffers according to RFC6182 [38] as 256MB. The experiments are conducted five times each and the evaluation results are derived from their average values.



(a) Overhead of the main operators of eMPTCP.

(b) Overhead of eMPTCP with different lengths of policy chains.

(c) Overhead of eMPTCP with different selection granularities.

(d) CPU usage of eMPTCP.

Fig. 5. Performance evaluation on server.

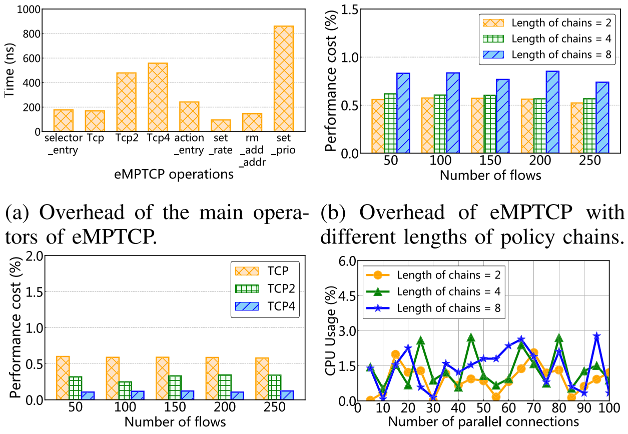
Workloads. In the experiments, we generate a large number of flows, representing network traffic of varying characteristics (e.g., packet sizes, network bandwidth usage) by Traffic Generator [37] which is widely used in many recent researches.

A. Overhead

We conduct several experiments on both high-end servers and Raspberry Pi to evaluate the overhead introduced by eMPTCP. We evaluate the time of several representative operators to process one packet using high-resolution timestamps. First, we evaluate eMPTCP on servers.

As shown in Fig. 5a, the processing time of all eMPTCP operations is at the level of nanosecond. The operation with the largest cost is `set_flow_prio` because this operator conducts packet header space adjustment. The total overhead of a policy chain is composed of all selectors and actors. It should be noted that selector_entry acts as the entry point for the selector chain, facilitating the retrieval of the selector chain context from the BPF MAP associated with a packet. The functionality of actor_entry is similar.

Further, we perform two evaluations by controlling the selection granularity and the length of the policy chain, respectively. We utilize the ratio of policy chain execution time to flow completion time as a performance metric for assessing the impact of eMPTCP on end-to-end transmission. Additionally, we use the traffic generator to generate a specified number of flows to simulating real-world scenarios. In the first evaluation, the coarsest granularity represents the worst situation when all



(a) Overhead of the main operators of eMPTCP.

(b) Overhead of eMPTCP with different lengths of policy chains.

(c) Overhead of eMPTCP with different selection granularities.

(d) CPU usage of eMPTCP.

Fig. 6. Performance evaluation on Raspberry Pi.

packets are processed by the policy chain. The length of the policy chain was set to 2 (1 selector and 1 actor), 4 (1 selector and 3 actors), and 8 (4 selectors and 4 actors), respectively. As Fig. 5b shows, the extra operation time of eMPTCP contributes to less than 2% of the total transmission time of flows, for all lengths of the policy chain. Moreover, the cost is stable even with the number of concurrent flows increasing to 10,000, demonstrating the scalability of eMPTCP. In the second evaluation, the length of the policy chain was fixed to 4 and the selection granularity is varied from coarse-grained to fine-grained with different selectors. Fig. 5c shows that the average overhead of eMPTCP is around 0.63% of the total packet transmission time. The result reveals that the finer the granularity is, the fewer packets will be selected for actors, thus incurring less overhead. It also demonstrates the effectiveness of the selector chain to reduce additional overhead by filtering the most irrelevant packets.

eMPTCP also costs a few extra CPU cycles. We measure the extra CPU usage under heavy traffic by switching on/off eMPTCP. Furthermore, we evluate CPU usage under different numbers of parallel connections. Fig. 5 demonstrates the results that eMPTCP costs less than 0.35% extra CPU usage on average.

Then, we test eMPTCP on Raspberry Pi, and the methodology is the same as server's. As shown in Fig. 6a, the processing time of the representative eMPTCP operations is just a little higher than that on servers by an average of 1.8×. It still remains around a few hundred nanoseconds, ranging from 95ns to 859ns. Further, we observe from Fig. 6b that although the absolute value of the processing time is higher, the percentage that accounts for the total packet processing time is lower. On Raspberry Pi, the performance cost ranges only from 0.52% to 0.85%. The reason is that, on low-end devices, the network throughput is much lower, such that the processing time for each packet prolongs. In this case, the performance cost, in terms of the amount of time compared to the packet processing time, decreases. For the aspects of extra CPU usage, we can see from Fig. 6d that, even on low-end devices, eMPTCP takes very little extra CPU usage of less than 3%. This is because all eMPTCP functions run within the kernel, eliminating the overhead associated with context switches, and their complexity is limited, as ensured by the verifier

```
 1 SEC("xdp")
 2 int parse_credit(struct __sk_buff *ctx) {
 3   XDP_POLICY_PRE_SEC
 4   get_credit_key_xdp(&ckey, subflow);
 5   s64 credit = parse_credit(ctx);
 6   s64 *subflow_credits;
 7   subflow_credits
 8         = bpf_map_lookup_elem(&credits_map, &ckey);
 9   __sync_fetch_and_add(subflow_credits, credit);
10   XDP_ACTOR_POST_SEC
11 }
12
13 SEC("struct_ops")
14 int BPF_PROG(sched_credit, struct mptcp_sock *msk
15                            struct mptcp_sched_data *data)
16 {
17   for (i = 0; i < MPTCP_SUBFLOWS_MAX; i++) {
18     if (data->contexts[i] == NULL)
19       break;
20     struct credit_key ckey;
21     subflow = data->contexts[i];
22     get_credit_key(&ckey, subflow);
23     s64 *subflow_credits;  s64 credits_left;
24     subflow_credits = bpf_map_lookup_elem
25                       (&credits_map, &ckey);
26     if (subflow_credits == NULL)
27       continue;
28     credits_left
29         = __sync_fetch_and_sub(subflow_credits, 1);
30     if (credits_left <= 0)
31       continue;
32       /* other sched ALG, here use min_rtt */
33       ...
34       choose_idx = i;
35   }
36   mptcp_subflow_set_scheduled
37       (data->contexts[choose_idx], 1);
38   }
```

Listing 2. Example of use case 1.

B. Use Cases

Building on top of eMPTCP, we can implement various user-defined control policies for the multipath environment with a modest size of code and zero changes to the native kernel implementation of MPTCP. Generally, there are two approaches to developing new extensions based on eMPTCP. The first is to combine eMPTCP's pre-defined actors and selectors into a policy chain, which can be implemented directly in userspace. For example, both use case 3 and use case 4 can be achieved this way. The policy chain can be easily configured using the Python interface, as shown in lines 4-5 of Listing 4 and lines 12-13 of Listing 5. The second method allows users to create new selectors or actors using the helpers and macros provided by eMPTCP. This applies to use case 1 and use case 2. To implement new helpers, users only need to use the macros provided by eMPTCP at the beginning and end of the eBPF program, as shown in lines 3 and 10 of Listing 2 and lines 3 and 8 of Listing 3.

Use case 1: Proactive congestion control for MPTCP. One promising feature of eMPTCP is to enable new functionalities of MPTCP. In this case, we investigate adding functionality of proactive congestion control for MPTCP with eMPTCP. To do so, we develop a policy chain as described in section III-E at the sender host as a simple proactive congestion control algorithm. Additionally, we develop a naive credit generator that generates credit uniformly regarding the bandwidth of the bottleneck link and allocates credit to all flows according to their arrival order. We implement the credit

```
SEC("tc")
int mac_modify(struct __sk_buff *ctx) {
  TC_POLICY_PRE_SEC
  struct mac_t new_mac;
  new_mac = PARAM.mac;
  struct ethdr = get_ethdr(ctx);
  modify_mac(ctx, ethdr, new_mac);
  TC_ACTOR_POST_SEC
}

SEC("tc")
int redirect(struct __sk_buff *ctx) {
  TC_POLICY_PRE_SEC
  int ifindex= PARAM.ifindex;
  bpf_redirect(ifindex, 0);
  TC_ACTOR_POST_SEC
}
```

Listing 3. Example of use case 2.

```
def path_management(mptcp):
  #use only one path to transmit
  if mptcp.time_from_start.lt(t1):
    sc = SelectorChain().select(0,mptcp.main)
    ac = ActorChain().add("rm_subflow")
  else:
    sc = SelectorChain().select(0,mptcp.main)
    ac = ActorChain().add("add_subflow")
  PolicyChain(sc,ac).submit()
    #only use subflows without sharing link
  for flow in mptcp.new_subflows():
    if not exist_share_link(flow, mptcp):
      continue
    sc = SelectorChian().select(0, flow)
    ac = ActorChain().add("set_priority", B = 1)
    PolicyChain(sc,ac).submit()
```

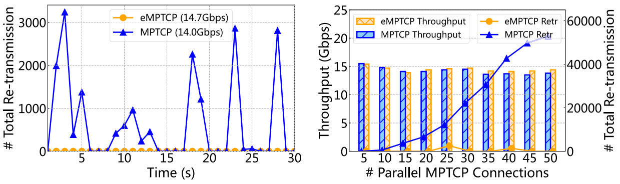
Listing 4. Example of use case 3.

```
SelectorChain.add("flow")
def ECF_scheduler(mptcp):
  sc = SelectorChain()
  ac = ActorChain()
  get_flow_info(mptcp)
  fast = get_fast(mptcp)
  slow = get_slow(mptcp)
  if fast.rtt + mptcp.k/fast.cwnd < slow.rtt:
    ac.add("rate_limit", rate = 1024)
  else:
    #enable sending data through slow flow
    ac.add("rate_limit", rate = 1024*1024)
  sc.select(0,flow)
  PolicyChian(sc,ac).submit()
```

Listing 5. Example of use case 4.

generator as an XDP program which modifies the packet by adding allocated credit information to the packet header and attaches it to the receiver host. The credit information will be inspected by an actor in the policy chain of the sender. We compare the performance including the total number of re-transmission and aggregate throughput between this extension (eMPTCP) with the default MPTCP congestion control algorithm (i.e. cubic) used in Linux kernel (MPTCP). The rest of MPTCP configuration for both is the same which includes setting fullmesh as the path management algorithm and setting BLEST as the traffic scheduler. We conduct experiments, using the MPTCP V1 maintained in mptcpnet-next, specifically with the kernel version 6.4.0. This is because the use case necessitates the `sched_ops` STRUCT_OP hook.

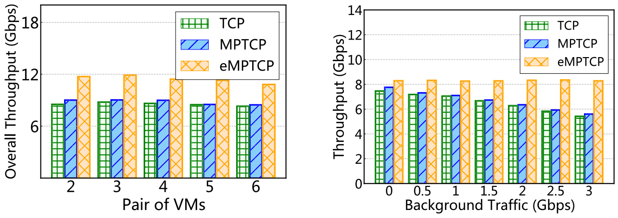
As Fig. 7 shows, we can observe that as the number of parallel MPTCP connections increases, there is a noticeable rise in the total number of re-transmissions and a slight decrease in the aggregate throughput when using the default MPTCP.In contrast, eMPTCP stands out by achieving nearly zero retransmissions while ensuring that the aggregate throughput remains equal to or slightly better than that of the default MPTCP. The reason for this is that the default congestion control algorithm of MPTCP reacts to congestion after it has already occurred. On the other hand, the eMPTCP extension allows for controlling the sending of packets based on credit, thereby preventing congestion.



(a) Re-transmission over time (number of parallel MPTCP connections equals 30).

(b) Re-transmission and throughput under varying number of parallel MPTCP connections.

Fig. 7. Effectiveness of eMPTCP enabled Proactive congestion control.



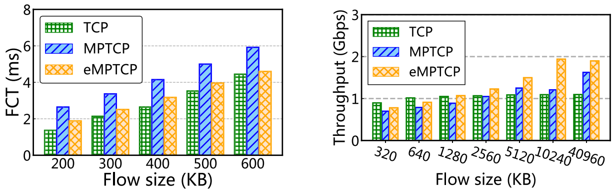
(a) Comparison of throughput with eMPTCP under different pairs of VMs

(b) Comparison of throughput with eMPTCP under different background flows.

Fig. 8. Effectiveness of eMPTCP enabled scheduler for the multi-tenant environment.

Use case 2: MPTCP in the multi-tenant environment. One promising feature of eMPTCP is to enable new functionalities of MPTCP. In this test case, we investigate the usage of MPTCP in the multi-tenant environment. With the increasing demands of VM-VM communication, there is an urge to utilize multiple paths in data center networks to improve network performance. Intuitively, the multipath transmission functionality can be added to VMs by deploying and enabling MPTCP in VMs. However, such a naive method will face two challenges. First, MPTCP is an end-host solution and the traffic of MPTCP-enabled VMs is not guaranteed to send through different physical links. Second, VMs belong to customers and we do not assume the network operators have all authority over guests' VMs. Thus, current methods to extend MPTCP by kernel modification or mptcpd are not applicable in the multi-tenant environment. Addressing this issue, we deploy eMPTCP on the hypervisors, and implement a simple traffic management policy that different subflows are sent through multiple physical interfaces.

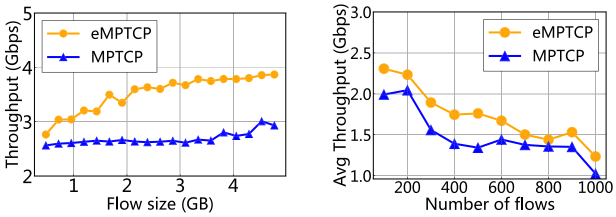
By enabling different subflows to send through multiple physical interfaces, eMPTCP delivers higher aggregate throughput for VMs. We measure the throughput of traffic between one pair of VMs with varying amounts of background traffic. Fig. 8a demonstrates that eMPTCP can improve the aggregate throughput for VMs by 23.03% when there is no background traffic. The improvement is more obvious when there is intensive background traffic. As is shown in Fig. 8b,



(a) Comparison of FCT for small flows.

(b) Comparison of throughput for large flows.

Fig. 9. Effectiveness of eMPTCP enabled path-manager.



(a) Comparison of throughput with traffic scheduler.

(b) Comparison of avg throughput for multi flows.

Fig. 10. Effectiveness of eMPTCP enabled traffic-scheduler.

when the background traffic reaches 3Gbps, the improvement can be as large as 32.3%. The reason is that, with eMPTCP and the congestion control algorithms of MPTCP, VMs can better utilize multiple paths in the multi-tenant environment while sharing the network with other tenants.

Use case 3: Path management. bottleneck shared path management Path management is a key component in the connection establishment of MPTCP. It controls when and how to establish subflows between two hosts. We design a simple path-manager, which works as follows. First, for an arbitrary flow, MPTCP uses only one path to transmit it at first and incrementally adds subflows with the number of bits this flow has sent. Second, when adding new subflows, only those sharing no common links with the existing subflows will be added. Such a simple path-manager benefits small flows with small latency and large flows with higher throughput. It should be noted that while shared bottleneck links can be detected through algorithms [31], [32], the primary purpose of this use case is to verify the effectiveness of path management, not to design a new algorithm. Therefore, we assume that information about shared bottleneck links is already known.

Fig. 9a demonstrates the effectiveness of the eMPTCP implemented path-manager in improving the Flow Completion Time (FCT) for small flows (less than 220KB). Through disabling subflows establishment at the beginning of the connection, eMPTCP provides the performance near native TCP and significantly reduces the overhead of MPTCP.

For large flows, eMPTCP further improves the capability of MPTCP by increasing the throughput of MPTCP. As shown in Fig. 9b, eMPTCP improves the throughput of MPTCP by 23.1% on average. This improvement is realized by removing the redundant paths which potentially cause congestion on the bottleneck link.

Use case 4: Traffic scheduling. Traffic scheduling is known to significantly impact the MPTCP performance, especially in the heterogeneous network environment. When MPTCP sends the packets on paths with different throughputs and delays, packets arriving at the receiver could be out-of-order. In such a case, packets sent from the fast paths have to wait for packets sent from the slow paths. Further, the re-ordering of packets also incurs extra costs. Addressing this issue, many traffic schedulers for MPTCP have been proposed. Among many of them, we implement a simple version based on the design of ECF [9], which allows for determining the sending rate on all subflows periodically at the interval of 100ms. The rate decision is defined by a vector $< r_1, r_2, \ldots, r_i >$, where r_i represents the rate of ith subflow.

In this test case, we establish two paths, one of which is set with a latency of 20ms, and the other is set with a latency of 50ms, corresponding to a fast subflow and a slow subflow, respectively. At each decision interval, the scheduler calculates the rates on each path. Fig. 10a shows that such a traffic scheduler can improve the throughput by at most 41.6% and on average 30.9%. We further evaluate the effectiveness of this scheduler with a large number of concurrent flows. As Figure 10b shows, the scheduler can improve the average throughput by 16.8% in this case.

VI. Related Work

Currently, there are a lot of efforts to enhance MPTCP, such as path management, traffic scheduler, etc. For example, as traffic scheduling has a significant impact on the performance of MPTCP, Frömmgen et al. [8] proposed a high-level programming model for MPTCP scheduler and built a corresponding runtime environment in the kernel, which enables application-aware scheduling. Zhang et al. [11] developed an adaptive scheduler based on deep reinforcement learning to schedule multi-path traffic for different scenarios. Cai and Lui [15] presented an online learning-based method to select multiple paths by learning the stochastic metrics of the paths. ECF scheduler [9] was developed which makes a prediction about transfer time through subflows and sends packets through the path with an earlier completion time. Xing et al. [39] developed an online-learning assisted scheduler tailored to accommodate diverse network conditions for mobile users. For path management, Hesmans et al. developed MPTCP path management Netlink [12] and Socket [14] API, which enables userspace and application-oriented path management. Zongor et al. [16] pointed out that when the subflows of MPTCP are not fully disjoint, the throughput will be limited by bottleneck links. Gao et al. [17] calculated the optimal path set and chose the optimal number and subflow-path assignment for MPTCP connections. Many existing works have tried to extend MPTCP in various scenarios, Le and Nahum et al. [19] utilized MPTCP to seamlessly migrate live VMs across WAN boundaries. Xu et al. [20] developed a congestion control algorithm that detects path-sharing by comparing RTT and ECN of different subflows.

Despite the promising usage of MPTCP, extending MPTCP is not easy. Existing methods either modify the kernel implementation of MPTCP, which involves considerable engineering efforts and may introduce security flaws, or control MPTCP via userspace tools such as mptcpd [25], which suffers from highly-restricted functionalities. Based on eBPF, Viet-Hoang Tran and Olivier Bonaventure [27] take the first step toward extending network protocols with eBPF. However, they only reveal the implementing details of an enhanced MPTCP path management, challenges still remain to fully extend MPTCP for more real use cases.

Beyond MPTCP, there are also many works that exploit the multipath feature of networks. For example, Gurtov and Polishchuk [40] developed a multipath scheduler called mHIP laying between IP and HIP layer which avoids many common issues in multipath environments, such as address hijacking, and vulnerability to address changing. Nikravesh et al. [41] designed a userspace multipath system called MPFlex which runs as a transport layer proxy and provides multipath services for TCP and UDP traffic. Coninck and Bonaventure [21] proposed Multipath QUIC which enables QUIC with the multipath transmission. Although these works are not directly based on MPTCP, their designs can inspire the extensions of MPTCP and can be further facilitated by eMPTCP.

VII. Future Work

Extending eMPTCP in the mobile environment. MPTCP has been most widely used on mobile devices to aggregate the bandwidth of heterogeneous paths or realize seamless handovers between networks. Therefore, extending MPTCP in the mobile environment is a potentially significant scenario. Since eMPTCP is implemented based on eBPF which has been supported since kernel version 4.9 and Android 9 [42], we believe that eMPTCP is also feasible to deploy on mobile devices. A future plan of eMPTCP is to evaluate the feasibility and robustness when deploying on mobile devices.

Extending to support more transport protocols. While the design of eMPTCP mainly targets at MPTCP, we believe that it is capable of supporting more general transport protocols with the help of XDP and TC. By enabling inspection on network packets, eMPTCP combines the view from the different layers of protocols, yielding more insights into cross-layer innovations. The implementation of eMPTCP also encourages a practical way to encapsulate more verifier acceptable, robust eBPF helper functions.

Support extensions from different developers. eMPTCP was initially designed for use by a single trusted developer, without taking into account potential conflicts that may arise from multiple developers. Ensuring compatibility among diverse extensions is a pivotal concern in numerous protocol designs, such as the concern in PQUIC [43]. Thus, we believe this as a promising direction for future exploration.

VIII. Conclusion

In this paper, we have presented eMPTCP, a framework that enables to extend MPTCP with customized control policies. eMPTCP is highly flexible and pluggable. Implemented based on eBPF, eMPTCP benefits from the security and robustness of the kernel development. We have demonstrated that several representative MPTCP extensions can be easily implemented with eMPTCP. Extensive experiments have shown that eMPTCP incurs little overhead at the level of nanosecond with negligible packet processing overhead.

References

[1] O. Bonaventure and S. Seo, "Multipath TCP deployments," *IETF J.*, vol. 12, no. 2, pp. 24–27, 2016.

[2] G. Chen et al., "FUSO: Fast multi-path loss recovery for data center networks," *IEEE/ACM Trans. Netw.*, vol. 26, no. 3, pp. 1376–1389, Jun. 2018.

[3] C. Paasch and O. Bonaventure, "Multipath TCP," *Commun. ACM*, vol. 57, no. 4, pp. 51–57, Apr. 2014.

[4] G. Sarwar, R. Boreli, E. Lochin, A. Mifdaoui, and G. Smith, "Mitigating receiver's buffer blocking by delay aware packet scheduling in multipath data transfer," in *Proc. 27th Int. Conf. Adv. Inf. Netw. Appl. Workshops*, Mar. 2013, pp. 1119–1124.

[5] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure, "Experimental evaluation of multipath TCP schedulers," in *Proc. ACM SIGCOMM Workshop Capacity Sharing Workshop*. New York, NY, USA: Association for Computing Machinery, Aug. 2014, pp. 27–32, doi: 10.1145/2630088.2631977.

[6] F. Yang, Q. Wang, and P. D. Amer, "Out-of-order transmission for in-order arrival scheduling for multipath TCP," in *Proc. 28th Int. Conf. Adv. Inf. Netw. Appl. Workshops*, 2014, pp. 749–752.

[7] S. Ferlin, Ö. Alay, O. Mehani, and R. Boreli, "BLEST: Blocking estimation-based MPTCP scheduler for heterogeneous networks," in *Proc. IFIP Netw. Workshops*, 2016, pp. 431–439.

[8] A. Frömmgen et al., "A programming model for application-defined multipath TCP scheduling," in *Proc. 18th ACM/IFIP/USENIX Middleware Conf.* New York, NY, USA: Association for Computing Machinery, Dec. 2017, pp. 134–146, doi: 10.1145/3135974.3135979.

[9] Y.-S. Lim, E. M. Nahum, D. Towsley, and R. J. Gibbens, "ECF: An MPTCP path scheduler to manage heterogeneous paths," in *Proc. 13th Int. Conf. Emerg. Netw. Exp. Technol.* New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 147–159, doi: 10.1145/3143361.3143376.

[10] P. Hurtig, K.-J. Grinnemo, A. Brunstrom, S. Ferlin, Ö. Alay, and N. Kuhn, "Low-latency scheduling in MPTCP," *IEEE/ACM Trans. Netw.*, vol. 27, no. 1, pp. 302–315, Feb. 2019.

[11] H. Zhang, W. Li, S. Gao, X. Wang, and B. Ye, "ReLeS: A neural adaptive multipath scheduler based on deep reinforcement learning," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2019, pp. 1648–1656.

[12] B. Hesmans, G. Detal, S. Barre, R. Bauduin, and O. Bonaventure, "SMAPP: Towards smart multipath TCP-enabled applications," in *Proc. 11th ACM Conf. Emerg. Netw. Exp. Technol.* New York, NY, USA: Association for Computing Machinery, Dec. 2015, pp. 1–7, doi: 10.1145/2716281.2836113.

[13] M. Kheirkhah, I. Wakeman, and G. Parisis, "MMPTCP: A multipath transport protocol for data centers," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, Apr. 2016, pp. 1–9.

[14] B. Hesmans and O. Bonaventure, "An enhanced socket API for multipath TCP," in *Proc. Appl. Netw. Res. Workshop*. New York, NY, USA: Association for Computing Machinery, Jul. 2016, pp. 1–6, doi: 10.1145/2959424.2959433.

[15] K. Cai and J. C. S. Lui, "An online learning multi-path selection framework for multi-path transmission protocols," in *Proc. 53rd Annu. Conf. Inf. Sci. Syst. (CISS)*, Mar. 2019, pp. 1–2.

[16] L. Zongor, Z. Heszberger, A. Pašić, and J. Tapolcai, "The performance of multi-path TCP with overlapping paths," in *Proc. ACM SIGCOMM Conf. Posters Demos*. New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 116–118, doi: 10.1145/3342280.3342328.

[17] K. Gao, C. Xu, J. Qin, S. Yang, L. Zhong, and G.-M. Muntean, "QoS-driven path selection for MPTCP: A scalable SDN-assisted approach," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, Apr. 2019, pp. 1–6.

[18] F. Duchene and O. Bonaventure, "Making multipath TCP friendlier to load balancers and anycast," in *Proc. IEEE 25th Int. Conf. Netw. Protocols (ICNP)*, Oct. 2017, pp. 1–10.

[19] F. Le and E. M. Nahum, "Experiences implementing live VM migration over the WAN with multi-path TCP," in *Proc. IEEE Int. Conf. Comput. Commun. (INFOCOM)*, May 2019, pp. 1090–1098.

[20] C. Xu, J. Zhao, J. Liu, and F. Chen, "Revisiting multipath congestion control for virtualized cloud environments," in *Proc. IEEE/ACM 28th Int. Symp. Quality Service (IWQoS)*, Jun. 2020, pp. 1–10.

[21] Q. De Coninck and O. Bonaventure, "Multipath QUIC: Design and evaluation," in *Proc. 13th Int. Conf. Emerg. Netw. Exp. Technol.* New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 160–166, doi: 10.1145/3143361.3143370.

[22] C. Paasch and S. Barre. *MultiPath TCP (MPTCP)—Linux Kernel Implementation*. Accessed: May 6, 2022. [Online]. Available: http://www.multipath-tcp.org

[23] *RFC 8684 TCP Extensions for Multipath Operation With Multiple Addresses*. Accessed: May 6, 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8684.html

[24] Netdev Group. (2020). *MPTCP Linux Kernel Upstream*. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/netdev

[25] *Multipath TCP Daemon*. Accessed: May 6, 2022. [Online]. Available: https://github.com/intel/mptcpd

[26] *Extened Berkeley Packet Filter (eBPF)*. Accessed: May 6, 2022. [Online]. Available: http://ebpf.io

[27] V. H. Tran, "Measuring and extending multipath TCP," Ph.D. thesis, Louvain School Eng. (EPL), UCLouvain, Louvain-la-Neuve, Belgium, 2019.

[28] T. Hoiland-Jorgensen et al., "The express data path: Fast programmable packet processing in the operating system kernel," in *Proc. 14th Int. Conf. Emerg. Netw. Exp. Technol.* New York, NY, USA: Association for Computing Machinery, Dec. 2018, pp. 54–66, doi: 10.1145/3281411.3281443.

[29] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proc. USENIX Winter Conf. (USENIX Winter Conf.)* San Diego, CA, USA: USENIX Association, Jan. 1993, pp. 259–270. [Online]. Available: https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet

[30] J. Han et al., "Leveraging coupled BBR and adaptive packet scheduling to boost MPTCP," *IEEE Trans. Wireless Commun.*, vol. 20, no. 11, pp. 7555–7567, Nov. 2021.

[31] *Shared Bottleneck Detection for Coupled Congestion Control for RTP Media*. Accessed: May 11, 2024. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8382.html

[32] S. Ferlin, Ö. Alay, T. Dreibholz, D. A. Hayes, and M. Welzl, "Revisiting congestion control for multipath TCP with shared bottleneck detection," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, Apr. 2016, pp. 1–9.

[33] W. Tu, Y.-H. Wei, G. Antichi, and B. Pfaff, "Revisiting the open vSwitch dataplane ten years later," in *Proc. ACM SIGCOMM Conf.* New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 245–257, doi: 10.1145/3452296.3472914.

[34] Linux Programmer's Manual. *tc-bpf(8)—Linux Manual Page*. Accessed: May 6, 2022. [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html

[35] M. K. Lau. (2020). *Introduce BPF STRUCT_OPS*. Accessed: Oct. 9, 2023. [Online]. Available: https://lwn.net/Articles/809092/

[36] B. Montazeri, Y. Li, M. Alizadeh, and J. K. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *Proc. ACM SIGCOMM Conf. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 221–235, doi: 10.1145/3230543.3230564.

[37] W. Bai, L. Chen, K. Chen, and H. Wu, "Enabling ECN in multi-service multi-queue data centers," in *Proc. 13th USENIX Symp. Netw. Syst. Design Implement.* Santa Clara, CA, USA: USENIX Association, Mar. 2016, pp. 537–549. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/bai

[38] *RFC 6182: Architectural Guidelines for Multipath TCP Development*. Accessed: May 6, 2022. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6182

[39] Y. Xing, K. Xue, Y. Zhang, J. Han, J. Li, and D. S. L. WeiMember, "An online learning assisted packet scheduler for MPTCP in mobile networks," *IEEE/ACM Trans. Netw.*, vol. 31, no. 5, pp. 2297–2312, Oct. 2023.

[40] A. Gurtov and T. Polishchuk, "Secure multipath transport for legacy internet applications," in *Proc. 6th Int. Conf. Broadband Commun., Netw., Syst.*, Sep. 2009, pp. 1–8.

[41] A. Nikravesh, Y. Guo, F. Qian, Z. M. Mao, and S. Sen, "An in-depth understanding of multipath TCP on mobile devices: Measurement and system design," in *Proc. 22nd Annu. Int. Conf. Mobile Comput. Netw.* New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 189–201, doi: 10.1145/2973750.2973769.

[42] *Android Open Source Project: Using EBPF Extensions*. Accessed: May 6, 2022. [Online]. Available: https://source.android.com/devices/architecture/kernel/bpf

[43] Q. De Coninck et al., "Pluginizing QUIC," in *Proc. ACM Special Interest Group Data Commun.* New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 59–74, doi: 10.1145/3341302.3342078.

Dian Shen (Member, IEEE) received the bachelor's, master's, and Ph.D. degrees from Southeast University, China, in 2010, 2012, and 2018, respectively. He was a Visiting Researcher with The Chinese University of Hong Kong from 2021 to 2022. He is currently an Associate Professor with the School of Computer Science and Engineering, Southeast University. His research interests include cloud computing, virtualization, and data center networks.

Bin Yang is currently pursuing the integrated master's and Ph.D. degree with the School of Computer Science and Engineering, Southeast University. His research interests include data center networks, OS kernel design, and implementation.

Junxue Zhang received the Ph.D. degree in computer science and engineering from iSINGLab, The Hong Kong University of Science and Technology (HKUST), supervised by Prof. Kai Chen. He is currently a Research Assistant Professor with the Department of Computer and Engineering, HKUST. His research work has been published in many top venues, such as SIGCOMM, NSDI, CoNEXT, and IEEE/ACM TRANSACTIONS ON NETWORKING. His research interests include data center networking, machine learning systems, and privacy-preserving computation.

Fang Dong (Member, IEEE) received the B.S. and M.S. degrees in computer science from Nanjing University of Science and Technology, China, in 2004 and 2006, respectively, and the Ph.D. degree in computer science from Southeast University, China, in 2011. He is currently a Professor with Southeast University and also the Director of the Big Data Computing Center, Southeast University. His current research interests include edge intelligence, cloud computing, and industrial internet. He is a member of ACM. He serves as the Co-Chair for the ACM Nanjing Chapter, China.

John C. S. Lui (Fellow, IEEE) received the Ph.D. degree in computer science from the University of California at Los Angeles. He is currently the Choh-Ming Li Chair Professor with the Department of Computer Science and Engineering, The Chinese University of Hong Kong (CUHK). His current research interests include quantum networks, machine learning, online learning (e.g., multi-armed bandit and reinforcement learning), network science, future internet architectures and protocols, network economics, network/system security, and large-scale storage systems. He is elected as a member of the IFIP WG 7.3, a Fellow of ACM, and a Senior Research Fellow of the Croucher Foundation. He received various departmental teaching and research awards, including the CUHK Vice Chancellor's Exemplary Teaching Award and the Research Excellence Award from CUHK.