Towards the Full Extensibility of Multipath TCP with eMPTCP

Bin Yang[1], Dian Shen[1](Cooresponding author), Junxue Zhang[2], Fang Dong[1], Junzhou Luo[1], John C.S. Lui[3]
[1]*School of Computer Science and Engineering, Southeast University, Nanjing, China*
[2]*iSing Lab, Hong Kong University of Science and Technology, China*
[3]*Department of Computer Science and Engineering, The Chinese University of Hong Kong, China*

2022 IEEE 30th International Conference on Network Protocols (ICNP) | 978-1-6654-8234-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICNP55882.2022.9940354

***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 security issues, or control MPTCP via user-space tools, which suffer from restricted functionality support. To address this issue, we propose eMPTCP, an *easy-to-use* framework to *fully* extend MPTCP *without security 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.**

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]–[11], path management [12]–[17], and network-application co-design [18]–[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. Franck Le, 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 user-space 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 user-space 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. 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 user-space daemons or user-defined kernel extensions, are limited by the functionalities of native

978-1-6654-8234-9/22/$31.00 ©2022 IEEE

[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 v0.95. 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.

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 security and ease-of-use. Using eBPF to extend MPTCP kernel with a user-defined program can ensure security 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**. eMPTCP supports a wide range of MPTCP operations, including controllable path establishment, traffic scheduling, etc. By allowing inspection and manipulation on network packets, eMPTCP can utilize the information from different layers of network protocols, yielding unique insights and exerting the 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 multi-tenant environment.
- **Higher pace of development**. With intent-based abstractions and security-verified helper functions provided by eMPTCP, network operators can focus on the essential policy development without worrying about the details and security issues of the MPTCP kernel. Policies like traffic scheduling in eMPTCP are written and maintained in Python and run from user space without security 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 provides a wide range of MPTCP control operations based on packet header manipulation; (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 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 nanosecond 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 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. Extended Berkeley Packet Filter (eBPF)

eBPF [28] is an emerging powerful technology, which allows developers to define programs that can be safely executed in the Linux 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 are compiled just-in-time (JIT) into the eBPF RISC instructions and finally attached to kernel functions.

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

To ensure security, eBPF incorporates a verifier to 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 pseudo-C code to develop the 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.

To facilitate programming, eBPF provides a variety of helper functions. These helper functions are implemented as part of the kernel and can be called by the BPF program with appropriate parameters. The list and functionality of helpers in the kernel are steadily increasing. Currently, there are over 100 helpers in the latest Linux kernel. However, we only need few of them to extend the MPTCP.

C. 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]–[11], path management [12]–[17] and network-application co-design [18]–[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. 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]–[21], etc.

To extend MPTCP, one can use a user-space control daemon provided by MPTCP. 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 only a set functionalities such as path management. Compared with mptcpd, eMPTCP has several advantages.

First, as a generic netlink based solution, mptcpd has a strong coupling with the MPTCP kernel stack. Its supported events and operations rely entirely on the kernel stack implementation. Its update has to be consistent with the kernel version release. On the contrary, eMPTCP is not dependent on the kernel and MPTCP versions . This is beneficial, because it usually takes a long time for kernel developers to accept code changes and release a new version. Meanwhile, eMPTCP is compatible with various already deployed Linux kernel versions, and mptcpd is not. Currently, we have tested eMPTCP with kernel versions 4.19, 5.10, 5.15, 5.19, MPTCPv0 and MPTCPv1, it works with all the versions without any modification.

Second, from the overhead perspective, eMPTCP has multiple advantages over mptcpd.

1) The overhead of the generic netlink based user space solution is much larger. In mptcpd, each individual control consists of event-triggering in the kernel, event handling in user space and finally calling a command API to enforce the control back to the kernel. Instead, eMPTCP works in the kernel for most use cases once the selector and actor chains are loaded. We implement a simple plugin using mptcpd to set a subflow to be backup and measure its processing overhead. From our measurement, this simple action takes at least 145μs on a high-end server. For comparison, the same operation implemented by eMPTCP takes 640-900ns. The overhead is approximately 150-200$\times$ higher. As a consequence, eMPTCP enables more fine-grained control over MPTCP traffic, e.g. packet level control.

2) eMPTCP relies on the eBPF verifier to ensure that all the codes can run safely in the kernel. The bilateral effect of this verifier is that it ensures control logic to meet the stringent performance requirement of the kernel. On the contrary, mptcpd does not verify the registered plugin. Thus, it cannot ensure that the extensions perform consistently in the user space. For example, some unbounded loop or thread sleep in the plugin will cause the management to not work as expected .

3) eMPTCP works by monitoring the MPTCP traffic and taking actions before the traffic enters kernel MPTCP stack, while mptcpd works after the traffic being processed in the kernel stack. As a consequence, eMPTCP enables to take early control over MPTCP before the packets are processed by the kernel stack. This will save kernel processing if some options or packets need to be filtered or dropped.

However, we believe that mptcpd and eMPTCP do not have to be an either-or choice. Actually, both solutions can work together and complement each other for wider and more intelligent use of MPTCP.

Besides, extending MPTCP by using eBPF provides a promising solution to support various use cases when multiple communication paths exist. Some previous works have attempted to use eBPF to add or modify certain specific network options into the kernel, such as TCP Timeout Option, TCP Congestion Control Option, acknowledgement option and etc. However, few of these research works have considered the multipath transport scenario. 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 the components of an advanced MPTCP extension in the kernel. For example, the scheduling algorithms of a traffic scheduler need to 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, while many other extensions incorporate the traffic classifier and measurement module, such basic components can not be reused for the new extension. With current methods, developers need to implement and recompile the whole kernel module from scratch.

Second, current methods to extend MPTCP, either by user-space daemons or user-defined kernel extensions, are limited by the functionalities of the native MPTCP stack. For example, current MPTCP and its extensions work on end-hosts only, hence lacking both knowledge and controllability of the underlying network. 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.

Third, although the emerging eBPF technique provides an effective mean to inject a user-defined program into the kernel with security guarantee, it is still very restrictive to implement an MPTCP control policy with eBPF due to its security validation. From our experience, there are many verifier-related issues which 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. Therefore, 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:

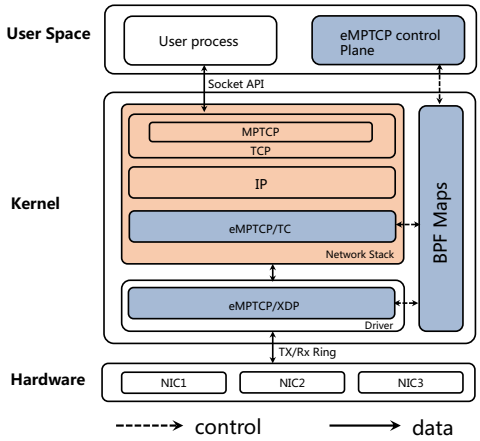


Fig. 1: eMPTCP and the networking stack.

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.

Second, eMPTCP needs to support the extension of a wide range of MPTCP operations, including controllable path establishment, dynamic traffic scheduling and 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 security 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.

As depicted in Fig. 1, eMPTCP is attached at the point below the TCP/IP stack and right above the network adapter receive (RX) queue. From the network layering perspective, it lies between L2 and L3. Thus, eMPTCP can oversee and manipulate the whole IP packets before they are processed in the network stack. The design overview of eMPTCP is shown in Fig. 2. eMPTCP delivers its desirable features through the following key designs: (1) a selector-actor style policy chain, (2) a policy enforcer based on packet manipulation , and (3) the intent-based abstraction.

A. 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, 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 imple-

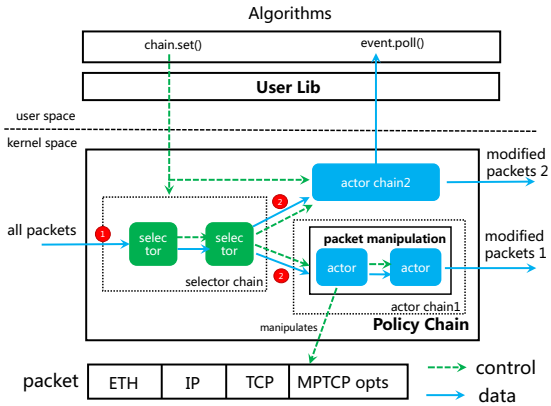


Fig. 2: eMPTCP overview.

mentation of the "traffic classifier" 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 enforce 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 work 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. The selector mainly decides *which* granularity the policies are enforced. The selector chain filters unrelated packets or packet groups, and forwards the related packets to specific action chains. A typical selector includes the Connection selector which checks the 4-tuple to handle only the desired MPTCP connections. 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 network packets. For example, combined with the Connection selector, operators can specify a subflow selector to identify those subflows belonging to the same MPTCP connection by checking the MPTCP token. Then, the actor chain performs the operations that should be taken on the selected packets. The implementation of selectors and actors is modular and can be chained with arbitrary numbers. All the sub-policies can be configurated with different parameters, added or removed from chains at run-time, without interrupting the running services. Note that combing 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 packets as soon as they reach the NIC, and forwarding the relevant ones to the suitable actor chain. The filtered packets can be dropped or dispatched to the native kernel stack. 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 well-designed, MPTCP-specific and verifier-acceptable selectors/actors.

B. Packet Manipulation

eMPTCP policy enforcer 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.

TABLE I: Selectors supported by eMPTCP.

Selector name	Functionality
`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,.

Defined by the standard MPTCP protocol, the main packet header options include `MP_CAPABLE`, `MP_JOIN`, `MP_DSS`[3], `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. Beyond that, rate-limiting of MPTCP subflows is implemented through 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 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,

[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 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.
record	Metric	Record specific metrics of selected packets such as RTT, flow size and etc,.

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 DSS options as control packets rather than congestion signals, according to RFC 8684.

C. Intent-based Abstraction

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

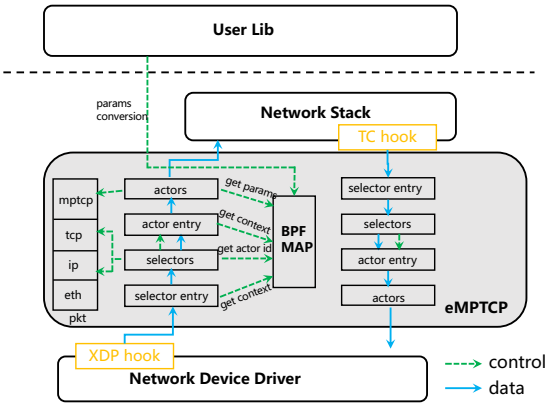


Fig. 3: eMPTCP implementation.

First of all, eMPTCP incorporates a set of helper functions 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 security 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

The functional logic of eMPTCP is to disseminate the user-defined MPTCP control policy into multiple small building blocks and chain them together using eBPF tail calls. First, each building block of the policy is implemented by an eBPF program. An eMPTCP program supports controlling both directions of egress and ingress network traffic. For ingress traffic, we attach eBPF programs to eXpress Data Path (XDP) [29], and for ingress traffic, we attach them to

Traffic Control (TC) [30]. These small eBPF programs are analyzed and loaded independently, such that it reduces the analysis complexity of the verifier and helps to pass the eBPF restrictions on program sizes. Second, to support the run-time combination of these building blocks, 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 bytes 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 meta data (`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 meta data, 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 meta data. 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

eMPTCP works by chaining multiple eBPF programs and executing the policy sequentially. Since eBPF prohibits the use of global variables, sharing data among these eBPF programs is highly restrictive. Addressing this issue, eMPTCP has two mechanisms to share intermediate results or control parameters among sub-policies. First, if the shared data is small enough, e.g., less than 2 bytes, it can be stored inline with the chain context using the last two bytes. Such a method is cost-efficient and avoids extra storage or memory access. Alternatively, if the shared data is large, we use BPF MAP to realize the data sharing. Conventionally, a BPF MAP can not be shared among separate programs, as the MAP file descriptor itself is unable to share among eBPF programs. Hence, 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. Another usage of BPF MAP is to share data between user space programs and kernel functions. In such a scenario, 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

eMPTCP exerts a fine-grained control on MPTCP through packet manipulation. Some representative manipulations are:

(1) Modifying an existing MPTCP option. This kind of operations require no change on the length of header space.

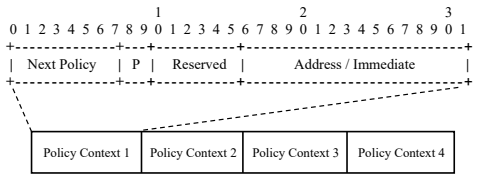


Fig. 4: Definition of the chain context.

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.

Note that eMPTCP can work with very long MPTCP connections. The trick here is that, when it injects any MPTCP option, 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 TCP stack. Moreover, the duplicated ACKs won't affect the congestion window, according to RFC 8684.

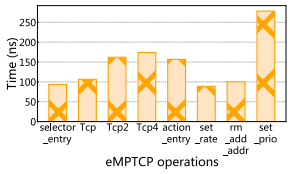
D. Verifier Acceptable Helper Functions

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.

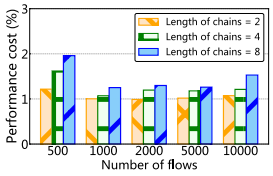
V. Evaluation

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.

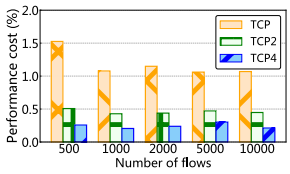
Testbed. The testbed we use in the experiments consists of 7 servers, each of which is equipped with two Intel(R)



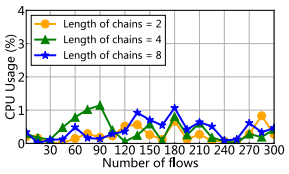
(a) Overhead of the main operators of eMPTCP.



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

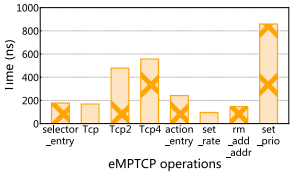


(c) Overhead of eMPTCP with different selection granularities.

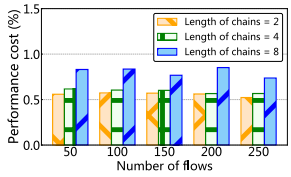


(d) CPU usage of eMPTCP.

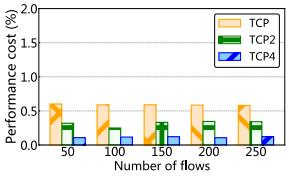
Fig. 5: Performance evaluation on server.



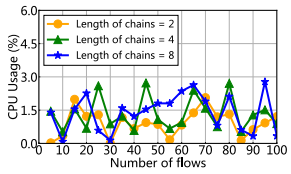
(a) Overhead of the main operators of eMPTCP.



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



(c) Overhead of eMPTCP with different selection granularities.



(d) CPU usage of eMPTCP.

Fig. 6: Performance evaluation on Raspberry Pi.

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 many existing researches [31]. 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), 8GB memory. We use its WiFi and wire Ethernet interfaces under 300Mbps speed.

MPTCP setup. eMPTCP does not impose any limitation on the MPTCP version. We have tested eMPTCP on Linux kernel version 4.19, 5.10, 5.15, 5.19 with MPTCPv0 and MPTCPv1. To reduce unnecessary CPU overhead, we turn off MPTCP header checksumming while keeping the conventional TCP checksums enabled. In most experiments, receive buffers are set according to RFC6182 [32] as 256MB.

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 [31] which is widely used in many recent researches.

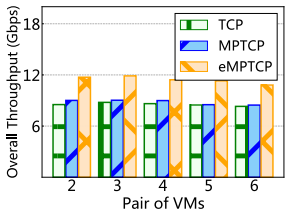
A. Overhead

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.

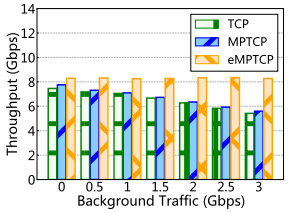
As shown in Fig. 5a, the processing time of all eMPTCP operations is in the level of nanosecond. The operation with the largest cost is `set_flow_prio`, because this operator conducts packet header space adjustment. The total overhead of a policy chain is composed of all selectors and actors.

Further, we perform two evaluations by controlling the selection granularity and the length of the policy chain, respectively. In the first evaluation, the coarsest granularity represents the worst situation when all packets are processed by the policy chain. The length of 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 contribute to less than 2% of the total transmission time of flows, for all length of the policy chain. Moreover, the cost is stable even with the number of concurrent flows increasing to 10000, 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 most of irrelevant packets.

eMPTCP also costs a few extra CPU cycles. We measure the extra CPU usage under heavy traffic by switching on/off eMPTCP. Fig. 5d demonstrates the results that eMPTCP cost less than 0.35% extra CPU usage on average. Then, we test eMPTCP on Raspberry Pi. 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 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 it 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 the eMPTCP functions have passed the strict verifier of eBPF, and it meets the stringent resource constraints of the kernel.

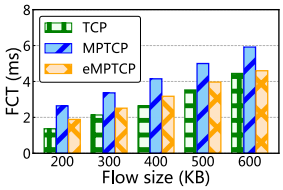


(a) Comparison of throughput with eMPTCP under different pairs of VMs

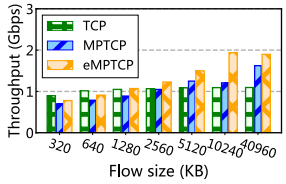


(b) Comparison of throughput with eMPTCP under different background flows.

Fig. 7: Effectiveness of eMPTCP enabled scheduler for multi-tenant environment.

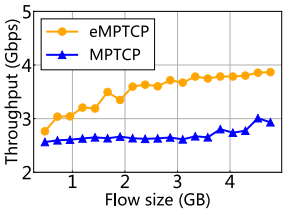


(a) Comparison of FCT for small flows.

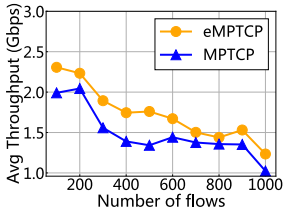


(b) Comparison of thoughput for large flows.

Fig. 8: Effectiveness of eMPTCP enabled path-manager.



(a) Comparison of throughput with traffic scheduler.



(b) Comparison of avg throughput for multi flows.

Fig. 9: Effectiveness of eMPTCP enabled traffic-scheduler.

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.

Use case 1: 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, it is an urge to utilize multiple paths in data center networks to improve the 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. 7a 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. 7b, 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 2: 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.

Fig. 8a 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. 8b, 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 3: 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 throughput and delay, 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 $< r_1, r_2, ..., r_i >$, where r_i

represents the rate of ith subflow.

In this test case, we establish two paths, one of which is set with the latency of 20ms and the other is set with the 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. 9a 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 9b 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 and 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 et al. [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. 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, Franck 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 user-space 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 et al. [33] 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. Ashkan et al. [34] designed a userspace multipath system called MPFlex which runs as a transport layer proxy and provides multipath services for TCP and UDP traffic. De Coninck et al. [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. Conclusion

In this paper, we have presented eMPTCP, a framework which 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 in the level of nanosecond with negligible packet processing overhead. Some future directions to improve eMPTCP include the following.

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 [35], 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.

Acknowledgements

This work is supported by National Key R&D Program of China 2018AAA0100500, National Natural Science Foundation of China under Grants, No. 61902065, 61972085, 61906040, the Natural Science Foundation of Jiangsu Province under grant BK20190345, BK20190335, Jiangsu Provincial Key Laboratory of Network and Information Security under Grants No.BM2003201. We also thank the Big Data Computing Center of Southeast University for providing the experiment environment and computing facility. John C.S. Lui is supported in part by the GRF 14200321.

We also thank the shepherd Prof. Olivier Bonaventure and anonymous reviewers of this paper for their constructive comments to improve the quality of the paper.

REFERENCES

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

[2] G. Chen, Y. Lu, Y. Meng, B. Li, K. Tan, D. Pei, P. Cheng, L. Luo, Y. Xiong, X. Wang, and Y. Zhao, "FUSO: Fast Multi-Path Loss Recovery for Data Center Networks," *IEEE/ACM Transactions on Networking*, vol. 26, no. 3, pp. 1376–1389, 2018.

[3] C. Paasch and O. Bonaventure, "Multipath TCP," *Communications of the ACM*, vol. 57, no. 4, pp. 51–57, 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 *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, 2013, pp. 1119–1124.

[5] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure, "Experimental Evaluation of Multipath TCP Schedulers," in *Proceedings of the 2014 ACM SIGCOMM Workshop on Capacity Sharing Workshop*, ser. CSWS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 27–32. [Online]. Available: https://doi.org/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 *2014 28th International Conference on Advanced Information Networking and Applications 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 *2016 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE, 2016, pp. 431–439.

[8] A. Frömmgen, A. Rizk, T. Erbshäußer, M. Weller, B. Koldehofe, A. Buchmann, and R. Steinmetz, "A Programming Model for Application-Defined Multipath TCP Scheduling," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, ser. Middleware '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 134–146. [Online]. Available: https://doi.org/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 *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 147–159. [Online]. Available: https://doi.org/10.1145/3143361.3143376

[10] P. Hurtig, K.-J. Grinnemo, A. Brunstrom, S. Ferlin, Alay, and N. Kuhn, "Low-Latency Scheduling in MPTCP," *IEEE/ACM Transactions on Networking*, vol. 27, no. 1, pp. 302–315, 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 *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1648–1656.

[12] B. Hesmans, G. Detal, S. Barre, R. Bauduin, and O. Bonaventure, "SMAPP: Towards Smart Multipath TCP-Enabled Applications," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2716281.2836113

[13] M. Kheirkhah, I. Wakeman, and G. Parisis, "MMPTCP: A multipath transport protocol for data centers," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, 2016, pp. 1–9.

[14] B. Hesmans and O. Bonaventure, "An Enhanced Socket API for Multipath TCP," in *Proceedings of the 2016 Applied Networking Research Workshop*, ser. ANRW '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1–6. [Online]. Available: https://doi.org/10.1145/2959424.2959433

[15] K. Cai and J. C. Lui, "An Online Learning Multi-path Selection Framework for Multi-path Transmission Protocols," in *2019 53rd Annual Conference on Information Sciences and Systems (CISS)*, 2019, pp. 1–2.

[16] L. Zongor, Z. Heszberger, A. Pašić, and J. Tapolcai, "The Performance of Multi-Path TCP with Overlapping Paths," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, ser. SIGCOMM Posters and Demos '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 116–118. [Online]. Available: https://doi.org/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 *2019 IEEE Wireless Communications and Networking Conference (WCNC)*, 2019, pp. 1–6.

[18] F. Duchene and O. Bonaventure, "Making multipath TCP friendlier to load balancers and anycast," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, 2017, pp. 1–10.

[19] F. Le and E. M. Nahum, "Experiences Implementing Live VM Migration over the WAN with Multi-Path TCP," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1090–1098.

[20] C. Xu, J. Zhao, J. Liu, and F. Chen, "Revisiting Multipath Congestion Control for Virtualized Cloud Environments," in *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. IEEE, 2020, pp. 1–10.

[21] Q. De Coninck and O. Bonaventure, "Multipath QUIC: Design and Evaluation," in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 160–166. [Online]. Available: https://doi.org/10.1145/3143361.3143370

[22] C. Paasch and S. Barre, "MultiPath TCP (MPTCP)- Linux Kernel implementation," http://www.multipath-tcp.org.

[23] "RFC 8684 TCP Extensions for Multipath Operation with Multiple Addresses," https://www.rfc-editor.org/rfc/rfc8684.html.

[24] "Netdev Group. (2020) MPTCP Linux kernel upstream. ," https://git.kernel.org/pub/scm/linux/kernel/git/netdev.

[25] "Multipath TCP Daemon," https://github.com/intel/mptcpd.

[26] "Extened Berkeley Packet Filter (eBPF)," http://ebpf.io.

[27] V. H. Tran, "Measuring and Extending Multipath TCP," Ph.D. dissertation, PhD thesis. UCLouvain, Louvain-la-Neuve, Belgium, 2019.

[28] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture." in *USENIX winter*, vol. 46, 1993.

[29] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 54–66. [Online]. Available: https://doi.org/10.1145/3281411.3281443

[30] W. Almesberger *et al.*, "Linux Network Traffic Control—Implementation Overview," 1999.

[31] W. Bai, L. Chen, K. Chen, and H. Wu, "Enabling ECN in Multi-Service Multi-Queue Data Centers," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, Mar. 2016, pp. 537–549. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/bai

[32] "RFC 6182: Architectural Guidelines for Multipath TCP Development," https://datatracker.ietf.org/doc/html/rfc6182.

[33] A. Gurtov and T. Polishchuk, "Secure multipath transport for legacy Internet applications," in *2009 Sixth International Conference on Broadband Communications, Networks, and Systems*, 2009, pp. 1–8.

[34] 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 *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 189–201. [Online]. Available: https://doi.org/10.1145/2973750.2973769

[35] "Android Open Source Project: Using eBPF Extensions," https://source.android.com/devices/architecture/kernel/bpf.