On Integrating eBPF into Pluginized Protocols

Quentin De Coninck
University of Mons, Belgium
quentin.deconinck@umons.ac.be

Louis Navarre*
UCLouvain, Belgium
louis.navarre@uclouvain.be

Nicolas Rybowski
UCLouvain, Belgium
nicolas.rybowski@uclouvain.be

This article is an editorial note submitted to CCR. It has NOT been peer reviewed.
The authors take full responsibility for this article's technical content. Comments can be posted through CCR Online.

ABSTRACT

eBPF is a popular technology originating from the Linux kernel that enables safely running user-provided programs in a kernel-context. This technology opened the door for efficient programming in the operating system, especially in its network stack. However, its applicability is not limited to the Linux kernel. Various efforts leveraged the eBPF Instruction Set Architecture (ISA) as the basis of other networking related use cases outside of the Linux kernel. This paper focuses on the pluginized protocols' use case such as PQUIC and xBGP where the eBPF ISA serves as the basis to execute plugins providing per-session protocol behavior. It first quickly describes how the Linux kernel builds around this eBPF ISA to provide enhanced in-kernel network programmability. Then, the paper considers the case of pluginized protocols. Leveraging eBPF outside of the Linux kernel environment requires complementing the eBPF ISA to meet the pluginized protocols' requirements. This paper details these integration efforts. Based on the lessons learned from these, it finally concludes by an applicability discussion of the eBPF ISA to other use cases.

CCS CONCEPTS

• **Networks** → **Network protocol design**; *Routing protocols*; *Transport protocols*; **Programming interfaces**; **Programmable networks**;

KEYWORDS

eBPF, QUIC, BGP, Plugin, Protocol operation, Network architecture

1 INTRODUCTION

The extended Bekerley Packet Filter (eBPF) virtual machine [10] was designed with the Linux kernel in mind. It is built atop BPF [24], a lightweight virtual machine meant to efficiently filter network packets using simple rules. It initially contained two 32-bit registers and a very short stack of 64 bytes, contained a simple restricted instruction set and was dedicated for filtering network packets. Thanks to its simplicity, a just-in-time (JIT) compiler for BPF bytecode was relatively straightforward to provide and is part of the kernel since version 3.0. In version 3.4, Linux is further extended to support system calls filters provided by a user in classic BPF. While it has remained in that classic form for a long time, the idea to extend BPF came around 2014 when the Linux developers wanted to integrate dynamic tracing tools in the kernel without hurting the performances too much [6]. To this end, they began integrating the "internal BPF" (in contrast with "classic BPF") by extending the set of available registers from 2 to 10, adding instructions closer to the ones natively supported by processors, extending the register size from 32-bits to 64-bits and supporting external functions calls. The support of the classic BPF interpreter was progressively discontinued in favour of JIT compilation. This "internal BPF" was latter exposed to user-space and renamed extended Berkeley Packet Filter (eBPF). A few years later, eBPF was usable in several parts of the Linux kernel for various use cases, ranging from performance tracing to efficient network programming [3, 10, 13].

The rise of such a lightweight virtual machine attracted a lot of interest, going beyond the Linux kernel [35]. There are now standardization efforts aiming at specifying a common eBPF Instruction Set Architecture (ISA) [20, 34] and its associated bytecode file format [33]. This enabled additional implementations of the eBPF ISA, such as user-space software ones [15, 26] or hardware ones [4, 30], opening the door to use cases beyond the Linux kernel. In particular, network researchers used eBPF as the core element of *pluginized protocols* where nodes can be extended without recompiling their implementations. This paradigm shift, notably introduced by PQUIC [7] and xBGP [40], proposes an implementation design where the runtime behavior can be altered on the fly on a per-session basis by executing architecture-independent, third party bytecode in the eBPF system. Yet, given that eBPF was primarily designed for the Linux kernel, pluginized protocols' requirements needs adaptations to the eBPF ISA, such as providing permanent memory or runtime monitoring.

This paper aims to detail the different integration efforts of eBPF in the pluginized protocol implementations that started in 2018. It first reviews in Section 2 the eBPF ISA and the elements that an environment must provide to support this architecture. Section 3 elaborates on the eBPF ecosystem present in the Linux kernel. Section 4 describes how the eBPF ISA was adapted in two different pluginized protocol implementations. It concludes in Section 5 by a discussion about the applicability of the eBPF ISA for further use cases.

2 THE EBPF ISA

The eBPF Instruction Set Architecture (ISA) [20, 34] defines a lightweight, sandboxed environment running bytecode using the eBPF instruction set. The available opcodes cover basic operations such as arithmetic/logic instructions, memory loads and stores, branching/jumping instructions, external call invocation and program return. The eBPF ISA exposes 11 64-bit registers, R0 to R10. The first ten registers are designed for general purpose, with write access by the program being run. They have particular roles when

*F.R.S.-FNRS Research Fellow.

coping with call invocations. The return value of the eBPF program or of an external call invocation is stored in R0. Arguments to external function calls are set in R1 through R5, while R6 to R9 are guaranteed to remain unchanged during call invocation. R10 is read-only and contains the frame pointer to the eBPF program stack. The eBPF ISA has a one-to-one mapping for both registers and opcodes to the x86_64 architecture, making the Just-In-Time (JIT) compilation process easy on such a platform. The maximum size of the eBPF stack is bound (often to 512 bytes), but its exact value is implementation-dependent.

A compiled eBPF program consists in an Executable and Linkable Format (ELF) file embedding two main elements. On the one hand, it contains the eBPF bytecode itself. Until 2018, eBPF programs contained a single entry point, meaning that only one function could be embedded per-bytecode. For the Linux kernel, this restriction was lifted since version 4.16 by bringing call function support in its BPF program loader, its verifier, its interpreter and its JIT compiler [18]. However, as of August 2023, this feature is not included in all eBPF ISA implementations [15, 26, 30]. The entry function can take at most one 64-bit argument, located in R1 at the beginning of its execution. On the other hand, the ELF file contains static data such as the textual identifier of the external functions related to each eBPF `call` opcode.

To be usable in a given environment, an eBPF ISA needs the following additional integration elements. First, the system must define *anchor points* (also called *hooks*) where the eBPF runtime, i.e., instance of the eBPF ISA, will be invoked. It may propose several anchor points providing different inputs and expecting specific output. Second, the system must specify the set of external functions that will be made available to the eBPF runtime. This paper names such external function set as the *eBPF runtime API*. Third, given that the eBPF runtime may operate in critical flows, the system should provide a *verification framework* ensuring the safety of the eBPF bytecode being run.

3 EXTENDING THE LINUX KERNEL

As mentioned in the previous Section, the Linux kernel provides the three integration elements to the eBPF ISA to form the eBPF Linux ecosystem. First, the Linux kernel defines hooks at different places where eBPF code can be injected. They usually work as callbacks performing specific tasks once a given kernel operation is executed or when a specific event occurs. Even if BPF was originally specified for networking extensions, eBPF now has a broader applicability area that we later discuss in this Section.

Second, the kernel provides an eBPF runtime API to interact with its internal state. Among them, some functions enable eBPF bytecode to interact with eBPF maps. These data structures are key/value persistent storage memory locations of defined size. They can be used to communicate with a user-space application or even with other eBPF bytecodes at different hooks.

Third, the Linux kernel includes a strict verifier [11] ensuring that injected eBPF programs do not compromise the kernel state (e.g., kernel lock ups due to loops or private fields modifications). This verifier prevents eBPF bytecodes from being Turing-complete, though the eBPF ISA does not prevent from having negative jumps. These checks are made easier thanks to the limited memory interactions enabled by (i) the short stack memory and (ii) the indirect BPF map interactions performed through external calls. Moreover, the verifier limits the number of instructions to be verified to one million before rejecting the program[1].

The Linux eBPF environment is in constant evolution and recent integrated features make the eBPF Linux development easier. One of them is the Compile Once, Run Everywhere (CO-RE) feature [27] to reduce the eBPF programs' dependency on the kernel version. Under the hood, it relies on the BPF Type Format (BTF) [19] that encodes debugging information of the eBPF program and maps. The BTF information indicates the features required by the eBPF program, enabling the Linux eBPF loader to rewrite the code if required. Another recent feature is the BPF kernel functions (kfuncs) [37]. It exposes any kernel function to the eBPF program under specific conditions. This enables eBPF developers to leverage existing functions in the Linux kernel not being part of the stable eBPF runtime API, similar to an unofficial API, without requiring changing the code of the Linux kernel. To simplify the on-line Linux verification process, the `bpf_loop()` helper function [21] was newly introduced in the stable eBPF runtime API. It provides easy-to-use for-loops that can iterate up to about 8 millions times without unrolling these loops, hence avoiding being stuck by the 1 million instruction limit.

With all this Linux ecosystem, it is now possible to leverage eBPF for numerous use-cases going beyond the original BPF network filtering application. On the networking side, the Linux kernel offers programmability at different levels. At the low level, XDP [13] enables a user-defined program to parse, lookup, filter and modify network packets with almost no performance penalty. It also gives control to the eBPF program regarding how the received packet should further be processed. This system covers use cases such as Distributed Denial of Service mitigation [2]. At the routing level, there is notably SRv6-eBPF [43] that exposes a new IPv6 Segment Routing (SRv6) [23] endpoint action to generate traffic engineering decisions using eBPF code. It enables implementing the IPv6 network programming paradigm [9] by executing eBPF bytecode on a packet-level basis. Example use cases include passive in-network delay measurement [43], fast failure detection and rerouting [42] and dynamic Forward Erasure Correction at the network layer [29]. At the transport layer, TCP-BPF [3] offers extensibility to the TCP kernel implementation by allowing an eBPF program to dynamically adjust its parameters on a per-connection basis using socket information such as the IP address and port numbers. Using this information, it is possible to chose appropriate values of these parameters to optimize communications, for example the receiving window or the retransmission timer. TCP-BPF was further extended by adding new hooks inside the TCP flow to add, remove, and edit new TCP options [36]. TCP Path Changer [17] leveraged this protocol extensibility to dynamically change the path of a TCP flow using the connection metrics. Antelope [45] and CCP [28] enables endpoints to dynamically change the congestion control algorithm using eBPF.

While this above list could also mention TC hooks and socket filters, eBPF has now a broad scope going beyond these networking use cases. These include kernel performance tracing hooks and

[1]This value was set to 32768 until Linux 5.2: https://ebpf.io/blog/ebpf-updates-2021-02/

security filters [8], among others. In all these Linux cases, the input of an eBPF program is a pointer towards a structure containing relevant information depending on the use-case being served. For instance, in XDP the raw packet buffer with few metadata such as the interface on which it was received.

4 EXTENDING USER-SPACE PROTOCOLS

Besides the Linux kernel, several projects extracted the core eBPF ISA into a user-space library [15, 26]. These enable running eBPF bytecode either in interpreted or in JITed mode. Such projects enabled, for instance, Microsoft to natively support eBPF on Windows [35]. Still, as described in Section 2, leveraging the eBPF ISA requires defining both the anchor points and the eBPF runtime API. Furthermore, the user-space versions of the eBPF ISA come with much simplified eBPF validators. For instance, both uBPF [15] and rBPF [26] only perform syntactical checks on the injected bytecode, i.e., they report whether they observe an unknown opcode, whether the register indexes are invalid,... Hence, without additional mechanisms, it is possible to accept eBPF bytecodes implementing an infinite loop.

The initial idea of *pluginized protocols* emerged from our deployment experience of protocol extensions. Extending a network protocol often requires support from all the participating entities. Because of the heterogeneity in the protocol implementations and the participants being controlled by different entities, deploying new extensions can take years [39]. Recent works [7, 40] proposed to change that paradigm. They introduced network protocols implementations that can be dynamically extended through external bytecodes called *plugins*. To that end, the eBPF ISA offers a lightweight environment where these bytecodes can be run.

The remaining on this section focuses on two such attempts. The first one, Pluginized QUIC (PQUIC) [7], is an implementation of the QUIC protocol [16] based on picoquic [14]. It allows injecting plugins written in eBPF bytecode by leveraging an adapted version of the uBPF library [1], further denoted as uBPF† in the remaining of this paper. The second one, LibxBGP [40], offers an implementation model allowing different BGP implementations to be extended by a same plugin. It relies on another upgraded version of uBPF† , denoted uBPF‡ [41]. Each of these solutions has specific requirements and integration, summarized by Table 1. This paper first introduces their context, then discusses how the eBPF ISA was adapted and integrated, describes the verification mechanisms applied to ensure the safety of the plugins (as they may be provided by third-party entities), and finally concludes with the lessons learned from these attempts.

4.1 Pluginized QUIC (PQUIC)

PQUIC enables the dynamic tuning of a specific QUIC connection through the injection of plugins. These plugins can either replace existing behaviors (e.g., parameter tuning) or add new ones (e.g., a new extension). Example plugins include monitoring, multipath capabilities, QUIC VPNs and Forward Erasure Correction to fasten packet recovery [7] or even entirely replacing the standard retransmission mechanism of QUIC [25]. Such plugins can be remotely provided, e.g., by a server wanting the client to support a specific protocol feature on the ongoing session. Concretely, these plugins come as a set of eBPF bytecodes and a textual manifest file stating to which specific protocol functions of the implementation the bytecodes must be attached. A PQUIC plugin is hence composed by a set of eBPF runtime instances (one for each eBPF bytecode), requiring some shared communication memory to perform their operation. As presented by Figure 1, PQUIC proposes to alter the call flow of the protocol process by both overriding existing functions (e.g., `schedule_frame`) and introducing new ones (e.g., `schedule_path`). These functions hence act as the anchor points of the eBPF runtimes. To perform such call stack changes, PQUIC requires the support of *nested anchor point* invocations, regardless of whether their implementation is provided by the original PQUIC code or an eBPF runtime instance. Because plugins can provide orthogonal features, PQUIC supports their composability by ensuring isolation between plugins.

Characteristic	PQUIC [7]	LibxBGP [40]
Virtual Machine	eBPF (uBPF†)	eBPF (uBPF‡)
Persistent Memory	Per-plugin heap	Per-bytecode heap + Global shared map
Plugin content	Several bytecodes + manifest file	Several bytecodes + JSON manifest
Internals independence	✗	✓
Nested anchors	✓	✗
Chained bytecodes	✗	✓
Multiple inputs	✓	✓
Multiple outputs	✓	✗
Hidden context	✗	✓
eBPF runtime scope	Per session	On the whole endpoint

Table 1: Summarizing the main characteristics of each solution.

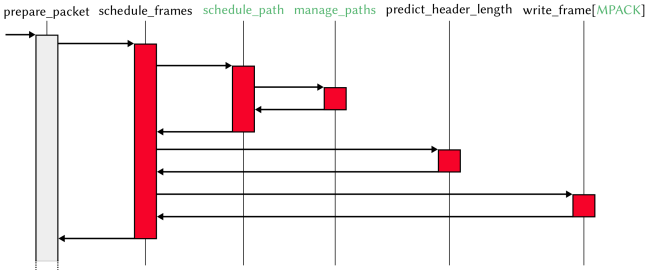

Figure 1: An example of function calling splitting in PQUIC, showing added anchor points and operations behavior provided by eBPF virtual machines.

4.1.1 eBPF Virtual Machine Inclusion. In PQUIC, the eBPF runtimes operate on a specific connection. Figure 2 overviews the integration of eBPF in PQUIC. PQUIC provides anchor points in the protocol flow process, and some default behavior. When an eBPF bytecode is attached to the anchor point, it overrides that default behavior. At most one eBPF virtual machine can replace that default behavior.

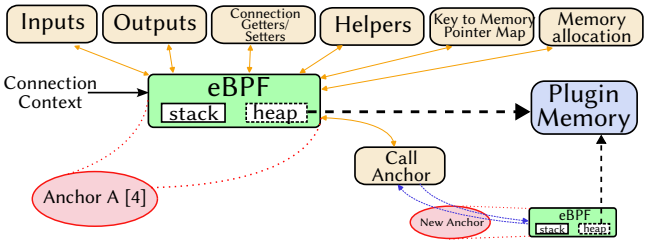

Figure 2: Overview of the integration of eBPF in PQUIC, showing the eBPF runtimes, the anchor points, the API given by external functions and the PQUIC-maintained memory.

Plugins may need to keep persistent state across different invocations, and eBPF sub-bytecodes may need a common communication channel. In addition to the limited virtual machine-specific stack, PQUIC maintains a plugin-specific heap that is shared across all its eBPF sub-bytecodes. However, PQUIC needs to keep control over the memory accessed by the eBPF bytecode. To avoid letting the eBPF bytecode access random addresses, the PQUIC uBPF† virtual machine integrates monitoring capabilities. Those are similar to works in Software-Based Fault Isolation [38, 44]. In addition to the default uBPF validation, uBPF† monitors the correct operations of the eBPF bytecodes by rewriting the original eBPF bytecode and injecting specific instructions. These monitoring instructions check that the memory accesses operate within the allowed bounds, i.e., they remain within either the plugin-dedicated memory or the eBPF stack. However, as stated in Section 2, all of the existing registers serve dedicated purposes (R0 for the return value, R1-R5 for the function parameters, R6-R9 being callee-saved, R10 holding the stack pointer). To achieve such runtime monitoring while avoiding complex eBPF bytecode rewriting, the VM relies on an additional register, R11, that cannot be used by regular eBPF code. The use of such an additional hidden register allows uBPF† to perform these checks within the eBPF bytecode without altering the content of the regular registers. Any violation of memory safety results in the call of a PQUIC-provided function removing the plugin and terminating the connection with an error. From a practical viewpoint, any memory access requested by an eBPF bytecode leads to the inclusion of 20 additional eBPF instructions and the rewriting of the offsets in the jump opcodes. These instructions are then present when the code is JIT-compiled.

When an eBPF bytecode is invoked, the system provides a pointer to the connection context as the input for the eBPF code. Given the memory checks described above, the code cannot directly dereference this pointer. Instead, it is used as a tag provided as argument to all the functions contained in the eBPF runtime API provided by PQUIC. The eBPF runtime API contains getters and setters for the different PQUIC connection fields, as well as various utility functions such as debugging ones. The eBPF runtime API also enables the eBPF bytecode to fetch the inputs and provide the outputs that are specific to the attached anchor point. To interact with the plugin-specific memory, the eBPF runtime API provides memory allocation functions within that shared space, as well as a map API where eBPF bytecodes can associate some key to the allocated memory space. Note that when different plugins are inserted on a same connection, they have distinct shared memory for each of their eBPF sub-bytecodes. Also, the memory management is spread across the different eBPF sub-bytecodes. For instance, a sub-bytecode *A* can `malloc` a memory area that is later `freed` by another sub-bytecode *B*, assuming that *B* is always called after *A*. The verification process does not assess that no memory is leaked, though the plugin-specific memory is bounded and this area is `freed` by the host implementation when the associated session is closed.

As described earlier, implementing protocol extensions often requires a combination of several eBPF bytecodes forming together a plugin. Some overridden functions may require the assistance of sub-functions. The eBPF runtime API enables an eBPF bytecode to call another anchor point, regardless of whether its behavior is provided by another eBPF bytecode or PQUIC itself. The inputs are provided as an array of integers, some actually being pointers. The outputs follow a similar format.

4.1.2 Verification Considerations. As mentioned earlier, verifications performed by the uBPF† verifier are quite limited. While the Linux environment benefits from a strong verifier, it is strongly glued with the eBPF loader, and this on-line verification process takes time, sometimes rejecting valid programs [11]. In addition, this verifier only focuses on the termination property, while other safety properties (such as protocol conformance) may be critical. To that end, instead of performing verification on the eBPF bytecode, PQUIC performs verification on the C source code.

Because of the complexity of the original picoquic implementation, deriving function specifications and assessing whether their behavior matches them is unpractical. Therefore, the two main properties being verified on PQUIC plugins are (i) their termination and (ii) the "side-effects" property. To verify these properties, PQUIC relies on external specialized tools, such as T2 [5] and SeaHorn [12]. However, these tools are generally slow and resources-intensive to produce their proofs, which is not compatible with an on-the-fly verification like what is performed with the kernel eBPF verifier. Hence, to perform complex offline verification, PQUIC authors designed a tool-chain called Plugin Distribution System (PDS) [32]. This system entirely automates the verification, compilation and distribution of PQUIC plugins. That way, verifications are performed ahead of plugin usage. PQUIC peers can fetch them from trusted registries when needed. Note that PQUIC does not verify with these external tools if direct memory accesses are valid, i.e., inside its stack and plugin heap areas. For this, uBPF† injects memory access checks to ensure that the eBPF runtime cannot escape its memory. Yet, these do not assess if computations rely on initialized values.

The non-termination of PQUIC plugins flow process may have two causes. First, a specific eBPF bytecode may not terminate (i.e., it loops forever). Second, (terminating) eBPF bytecodes may perform cyclic nested calls, resulting in an infinite call graph. To address the first case, the source code of an eBPF bytecode forming a plugin is verified with the T2 automated termination checker [5]. Note that unlike the Linux kernel that reasons on the eBPF bytecode itself, T2 works on the LLVM representation of the C code, i.e., before it gets translated into eBPF. Despite this more informed format, some terminating codes needed to be modified, e.g., to include explicit linked-list sizes, and others could not be automatically proven by T2. The second case is addressed by PQUIC monitoring the call

stack of the operations. If it notices that the called operation is already in the stack, it stops the execution of the plugin and closes the ongoing session.

The second property that is verified offline on PQUIC plugins is called "side-effects". PQUIC plugins receive a connection context as input, and may perform modifications on a subset of the fields exposed by this context. Some of them are read-only or even unreachable from within a plugin, because they lay in memory areas not reachable from within the eBPF runtime address-space. That being said, a given plugin is expected to respect a specification, i.e., a given behavior. The side-effects property ensures that a given plugin only modifies the fields of the connection context it is authorized to. For instance, a plugin providing a congestion control algorithm should not access nor leak the TLS keys used for the QUIC session. While host implementations can monitor the accessed fields, ensuring this specification enables implementations to relax such monitoring for optimization purposes. Its verification is performed with SeaHorn [12], a formal verification framework reasoning on LLVM bitcode and allowing the verification of explicitly annotated properties in C code. Such annotations are used to emulate an eBPF runtime, a dummy connection context and a witness context. Both contexts are equivalent at their initialization. Then, the verified plugin is called in the emulated environment with the dummy context as input. The dummy context is compared with the initial witness state to ensure that only authorized fields are modified by the plugin. The list of authorized fields, if any, is defined by the specifications of the plugin. Such specifications are also used, in conjunction with the header files of PQUIC, to automatically generate the annotations enforcing the side-effects property. A major limitation of this approach is the memory model used by SeaHorn. It was not able to formally represent complex memory structures such as linked lists and hashmaps, while they are widely present in the connection context exposed to the PQUIC plugins. The side-effect property is thus left unverified on such kind of fields.

Note that the verification process described in this section does not guarantee that verified plugins actually enhance the PQUIC connection. The several performed verifications ensure that a given plugin is not harmful for the PQUIC connections, but not that it performs useful or QUIC-compliant operations.

4.1.3 Lessons Learned. The choice of taking eBPF as the execution environment was motivated by its established usage in the Linux kernel. Integrating this virtual machine using a user-space implementation [15] into a QUIC implementation required lot of unforeseen efforts, such as the memory management into segments instead of arbitrary size areas to avoid memory fragmentation in the plugin heap. Still, developing eBPF plugins in PQUIC was made difficult because of the lack of proper operation definition and specification. PQUIC was built by overriding the original picoquic's functions with eBPF plugins, but these functions had some interdependencies with other ones. This made the development of eBPF plugins very dependent on the original picoquic's internals. Including a mechanism such as the recent Linux CO-RE would have simplified the plugin development process. Furthermore, because of the lack of advanced eBPF verifier, the included software-based fault isolation techniques added eBPF instructions on each memory

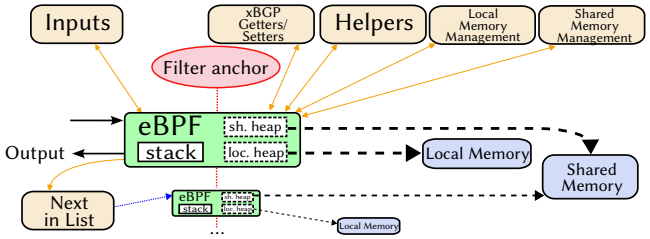

Figure 3: Overview of the integration of eBPF in xBGP, showing the eBPF virtual machine instances, the anchor points, the API given by external functions and the xBGP-maintained memories.

access of an eBPF bytecode, slowing down the execution compared to native C code. Finally, given the nested call feature of PQUIC, it was hard to assess the termination of plugins without including a runtime monitoring agent inside PQUIC, and T2 was not able to assess the termination of all plugins. Including a helper function handling for-loops, such as the `bpf_loop()`, may have helped in that verification process, as T2 assumes that external functions are always terminating.

4.2 LibxBGP

In a nutshell, LibxBGP [40] provides a software layer, called xBGP, that enables expansion of BGP implementations with user-supplied programs. For instance, xBGP plugins enable adding new BGP attributes and extending or changing the BGP decision process on the fly. Unlike PQUIC whose interface is specific to a particular implementation, xBGP aims to define a standardized layer where an xBGP plugin would correctly behave regardless of the actual BGP implementation being run. In particular, two different open-source implementations, BIRD and FRRouting, were made xBGP compliant. xBGP relies on the adherence of BGP implementations to the protocol specification [31] defining the different abstract data structures that each BGP implementation needs to keep. xBGP then defines the different anchor points based on the standardized BGP flow. There are five major anchors points: the parsing of BGP messages from peers, the import filters, the BGP decision process, the export filters, and the writing of BGP messages to peers. In xBGP, the whole host is the pluginizable unit. The BGP endpoint integrates an eBPF virtual machine manager holding the different eBPF runtimes for the anchor points. Whenever these code points are reached, the corresponding xBGP programs, if any, are executed using the uBPF‡ virtual machine.

4.2.1 eBPF Virtual Machine Inclusion. In xBGP, the eBPF virtual machine operates on the whole xBGP endpoint. Figure 3 overviews the integration of eBPF in the xBGP system.

While xBGP has a fixed, limited number of anchor points, it allows the network operator to insert a list of eBPF bytecodes that can perform very specific processing and delegate remaining operations to following eBPF bytecodes at the same hook. To achieve this, the xBGP API provides a specific `next()` function that stops the execution of the current eBPF runtime and starts executing the following one in the list. This feature is similar to the tail call

capability in the Linux kernel, except that the calling eBPF bytecode does not provide any argument specifying which function will be called next, as this only depends on the list order determined by the network operator. Note that when the `next()` function is called by the last eBPF runtime in the list, the default behavior provided by the xBGP implementation is executed.

Unlike PQUIC, xBGP does not provide any input to eBPF runtimes. Instead, it maintains a hidden context identifier that is used as the first hidden argument to all the eBPF runtime API functions. This context is then used to retrieve metadata associated to the related xBGP anchor point. Concretely, this means setting the eBPF register R1 to the value of the context identifier, but the original eBPF bytecode is not aware of this operation. To achieve this without altering the eBPF bytecode behavior, the uBPF‡ virtual machine was augmented with an additional register, R12. For each external function invocation, the registers content are shifted from one register to free R1 and store the hidden context in it. In practice, the content of R5 is saved in R12, then the content of R4 is saved in R5, and so forth until saving the content of R1 into R2. R1 is finally set to the context identifier value, known when injecting the eBPF bytecode. After the function call, further operations are performed to reset the registers to their original values, i.e., R1 takes the value of R2 and so on until R5 takes the value of R12. This requires adding 13 eBPF instructions per external function call invocation, as well as rewriting the offsets of the jump opcodes. The hidden context is then used to retrieve, e.g., the inputs specific to the anchor points or the available xBGP fields.

Some xBGP-specific structures span over thousands of bytes, and the eBPF stack alone is not sufficient to hold them. However, such structures are often specific to a given anchor point and there is no point in sharing it with other eBPF runtimes. To that end, uBPF‡ was extended to support two heap memories. The first one, called *local heap*, is specific to the eBPF runtime and allows overcoming the stack size limit while enabling data persistency. The second one, the *shared heap*, is common to all the plugins inserted in the endpoint and can serve as a communication channel through the different anchor points. xBGP provides specific API functions for each of these memory, the local memory ones having a `malloc()`/`free()`-like API while the shared memory being closer to the `shmget()` one. To ensure that uBPF‡ dereferences pointers in these area, additional memory range checks are added when injecting eBPF bytecode, similar to the process described in Section 4.1.1. Note that these additional monitoring checks may be disabled when the plugin has been certified memory-safe, as described in the next section.

4.2.2 Verification Considerations. Given the standardized data structure interface offered to the plugins, xBGP makes it simpler to perform verification going beyond plain termination. xBGP introduces a complete xBGP bytecode compilation pipeline based on the PDS [32] introduced by PQUIC, where properties verifications are performed offline. The termination property is verified with T2 [5], as for PQUIC.

A major difference with PQUIC verifications is the BGP protocol enforcement. xBGP plugins are annotated with the xBGP verification toolchain to verify that they respect the BGP specifications, e.g., the BGP message format. This property is verified with the formal verification SeaHorn framework [12]. The correct termination of strings with a null-byte is also performed with this framework.

Another difference with PQUIC plugins verification is the memory safety enforcement. This property is not related to the pluginization process by itself but is a generic C-code property verification. xBGP toochain leverages CBMC [22], a Bounded Model Checker, to ensure the memory safety of xBGP plugins. Various memory-related checks, such as the verification of array bounds, integers overflow or use of null pointers, are performed.

4.2.3 Lessons Learned. Unlike PQUIC, xBGP defines a common representation layer on which eBPF bytecodes operate on. While requiring some initial efforts to include it, this makes xBGP applicable to several implementations and simplifies the development of plugins. It also makes possible to assess protocol specifications through the verification toolchain. For the sake of simpler verification process and less possible misuse, xBGP hides the session context pointer in a register unavailable to the eBPF bytecode, still requiring changes in the eBPF ISA. Also, the verification tooling effort is still consequent, due to the use of the C programming language. Adopting languages enforcing safety properties, such as Rust, and applying verification on them may make this process simpler.

5 DISCUSSION

The Linux eBPF ecosystem enables in-kernel programmability by injecting user-provided bytecode. Since 2018 when we started to look at eBPF to tune the execution of protocol implementations, the Linux kernel integration evolved at a fast pace. The development of BTF and the eBPF tools (such as BCC) made the use of eBPF programs easier and less dependent to the specific kernel version. Furthermore, by adopting a simple RISC instruction set, it makes JIT compilation easy for x86_64 processors.

In parallel, we developed our own tooling based on the specific pluginized protocol use case, starting from the state where eBPF was in 2018. Neither PQUIC nor xBPG relies on recent Linux advances; plugins written in C were compiled into eBPF with only the plain clang compiler. The plugins metadata were given in separate manifest files instead of being encoded in the ELF. Furthermore, the user-space implementation we relied on [15] gave us some flexibility, but also lot of integration work as well, notably when providing some heap memory.

We however believe that our experience would benefit to trying to achieve similar eBPF integration in other use cases than the Linux kernel one. While user-space eBPF implementations follow the eBPF ISA, they often need to be complemented by additional features, such as persistent memory, support for hidden values associated to the eBPF runtime, and monitoring of memory accesses. While the Linux kernel includes such mechanisms, these should maybe be part of the eBPF ISA itself to benefit to other use-cases.

REFERENCES

[1] PQUIC Authors. 2020. uBPF: Userspace eBPF VM (PQUIC version). (2020). https://github.com/p-quic/ubpf/.

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

[3] Lawrence Brakmo. 2017. Tcp-bpf: Programmatically tuning tcp behavior through bpf. In *NetDev 2.2.*

[4] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2022. hXDP: Efficient software packet processing on FPGA NICs. *Commun. ACM* 65, 8 (2022), 92–100.

[5] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Terminator: Beyond Safety: (Tool Paper). In *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18*. Springer, 415–418.

[6] Jonathan Corbet. 2014. BPF: the universal in-kernel virtual machine. *Linux Weekly News* (May 2014). https://lwn.net/Articles/599755/, Accessed: 2021-02-04.

[7] Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, and Olivier Bonaventure. 2019. Pluginizing QUIC. In *Proceedings of the ACM Special Interest Group on Data Communication - SIGCOMM '19*. ACM Press, Beijing, China, 59–74. https://doi.org/10.1145/3341302.3342078

[8] Jake Edge. 2015. A seccomp overview. *Linux Weekly News* (September 2015). https://old.lwn.net/Articles/656307/.

[9] Clarence Filsfils, Pablo Camarillo, John Leddy, Daniel Voyer, Satoru Matsushima, and Zhenbin Li. 2021. Segment Routing over IPv6 (SRv6) Network Programming. RFC 8986. (Feb. 2021). https://doi.org/10.17487/RFC8986

[10] Matt Fleming. 2017. A thorough introduction to eBPF. *Linux Weekly News* (Dec. 2017).

[11] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1069–1084.

[12] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn verification framework. In *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Springer, 343–361.

[13] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*. 54–66.

[14] Christian Huitema. 2023. picoquic. (2023).

[15] IOVisor. 2023. uBPF: Userspace eBPF VM. (2023). https://github.com/iovisor/ubpf.

[16] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. (May 2021). https://doi.org/10.17487/RFC9000

[17] Mathieu Jadin, Quentin De Coninck, Louis Navarre, Michael Schapira, and Olivier Bonaventure. 2022. Leveraging eBPF to make TCP path-aware. *IEEE Transactions on Network and Service Management* 19, 3 (2022), 2827–2838.

[18] The kernel development community. 2023. BPF Architecture. (2023). https://docs.cilium.io/en/stable/bpf/architecture/#bpf-to-bpf-calls.

[19] The kernel development community. 2023. BPF Type Format (BTF). (2023). https://www.kernel.org/doc/html/v6.2/bpf/btf.html.

[20] The kernel development community. 2023. eBPF Instruction Set Specification, v1.0. (2023). https://www.kernel.org/doc/html/latest/bpf/instruction-set.html.

[21] Joanne Koong. 2021. [PATCH v3 bpf-next 0/4] Add bpf_loop helper. (2021). https://lore.kernel.org/bpf/87tuft7ff7.fsf@toke.dk/T/.

[22] Daniel Kroening and Michael Tautschnig. 2014. CBMC–C Bounded Model Checker: (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*. Springer, 389–391.

[23] David Lebrun and Olivier Bonaventure. 2017. Implementing ipv6 segment routing in the linux kernel. In *Proceedings of the Applied Networking Research Workshop*. 35–41.

[24] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture.. In *USENIX winter*, Vol. 46.

[25] François Michel, Alejandro Cohen, Derya Malak, Quentin De Coninck, Muriel Médard, and Olivier Bonaventure. 2022. FlEC: Enhancing QUIC with application-tailored reliability mechanisms. *IEEE/ACM Transactions on Networking* (2022).

[26] Quentin Monnet. 2023. rbpf: Rust (user-space) virtual machine for eBPF. (2023). https://github.com/qmonnet/rbpf.

[27] Andrii Nakryiko. 2020. BPF CO-RE (Compile Once – Run Everywhere). (2020). https://nakryiko.com/posts/bpf-portability-and-co-re/.

[28] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 30–43.

[29] Louis Navarre, François Michel, and Olivier Bonaventure. 2021. SRv6-FEC: bringing forward erasure correction to IPv6 segment routing. In *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*. 45–47.

[30] Richard Prinz et al. 2023. hBPF = eBPF in hardware. (2023). https://github.com/rprinz08/hBPF.

[31] Yakov Rekhter, Susan Hares, and Tony Li. 2006. A Border Gateway Protocol 4 (BGP-4). RFC 4271. (Jan. 2006). https://doi.org/10.17487/RFC4271

[32] Nicolas Rybowski, Quentin De Coninck, Tom Rousseaux, Axel Legay, and Olivier Bonaventure. 2021. Implementing the plugin distribution system. In *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*. 39–41.

[33] Dave Thaler. 2023. *eBPF ELF Profile Specification, v0.1*. Internet-Draft draft-thaler-bpf-elf-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-thaler-bpf-elf/00/ Work in Progress.

[34] Dave Thaler. 2023. *eBPF Instruction Set Specification, v1.0*. Internet-Draft draft-thaler-bpf-isa-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-thaler-bpf-isa/00/ Work in Progress.

[35] Dave Thaler and Poorna Gaddehosur. 2021. Making eBPF work on Windows. (May 2021). https://cloudblogs.microsoft.com/opensource/2021/05/10/making-ebpf-work-on-windows/.

[36] Viet-Hoang Tran and Olivier Bonaventure. 2020. Beyond socket options: Towards fully extensible Linux transport stacks. *Computer Communications* 162 (2020), 118–138.

[37] David Vernet. 2023. [PATCH bpf-next v3] bpf/docs: Document kfunc lifecycle / stability expectations. (2023). https://www.spinics.net/lists/kernel/msg4676660.html.

[38] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1994. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review* 27, 5 (1994), 203–216.

[39] Thomas Wirtgen, Quentin De Coninck, Randy Bush, Laurent Vanbever, and Olivier Bonaventure. 2020. Xbgp: When you can't wait for the ietf and vendors. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 1–7.

[40] Thomas Wirtgen, Tom Rousseaux, Quentin De Coninck, Nicolas Rybowski, Randy Bush, Laurent Vanbever, Axel Legay, and Olivier Bonaventure. 2023. xBGP: Faster Innovation in Routing Protocols. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*.

[41] xBGP Authors. 2023. LibxBGP. (2023). https://github.com/pluginized-protocols/libxbgp.

[42] Mathieu Xhonneux and Olivier Bonaventure. 2018. Flexible failure detection and fast reroute using eBPF and SRv6. In *2018 14th International Conference on Network and Service Management (CNSM)*. IEEE, 408–413.

[43] Mathieu Xhonneux, Fabien Duchene, and Olivier Bonaventure. 2018. Leveraging ebpf for programmable network functions with ipv6 segment routing. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. 67–72.

[44] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 30th IEEE Symposium on Security and Privacy*. 79–93. https://doi.org/10.1109/SP.2009.25 ISSN: 2375-1207.

[45] Jianer Zhou, Xinyi Qiu, Zhenyu Li, Gareth Tyson, Qing Li, Jingpu Duan, and Yi Wang. 2021. Antelope: A framework for dynamic selection of congestion control algorithms. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. IEEE, 1–11.