The Case for Pluginized Routing Protocols

Thomas Wirtgen, Cyril Dénos, Quentin De Coninck*, Mathieu Jadin†, Olivier Bonaventure
ICTEAM, UCLouvain, Louvain-la-Neuve, Belgium
Email: firstname.lastname@uclouvain.be

***Abstract*—Routing protocols such as BGP and OSPF are key components of Internet Service Provider (ISP) networks. These protocols and the operator's requirements evolve over time, but it often takes many years for network operators to convince their different router vendors and the IETF to extend routing protocols. Some network operators, notably in enterprise and datacenters have adopted Software Defined Networking (SDN) with its centralised control to be more agile. We propose a new approach to implement routing protocols that enables network operators to innovate while still using distributed routing protocols and thus keeping all their benefits compared to centralised routing approaches. We extend a routing protocol with a virtual machine that is capable of executing *plugins*. These *plugins* extend the protocol or modify its underlying algorithms through a simple API to meet the specific requirements of operators. We modify the OSPF and BGP implementations provided by FRRouting and demonstrate the applicability of our approach with several use cases.**

I. Introduction

During the last decades, the requirements imposed on enterprise and ISP networks have changed drastically. The first enterprise networks simply provided a "best effort" service and were not attached to a public network. Today's enterprise networks need to support Quality of Service [13] and include security feature to protect them from attacks originating from the Internet. ISP networks also face similar problems, but at a much larger scale [31], [10]. Internet traffic continues to grow quickly and ISP networks need to scale to sustain the load. To cope with these changing requirements, network operators are forced to innovate.

Innovation is defined by the Merriam-Webster dictionary as the *introduction of something new*. As an extension, we can define *network innovation* as the introduction of new features inside an enterprise or ISP networks. The introduction of a new feature is often done in three phases: $design$, $implement$ and $deploy$. During the $design$ phase, the network operators propose new abstract solutions and evaluate them. One of the proposed solutions is then implemented before being deployed it in the network after successful tests in labs.

The Software Defined Networks (SDN) [39] accelerate innovation in networks by using centralised controllers that program flow-tables on the network switches and routers. Thanks to these centralised controllers, researchers and network operators can implement a wide range of services that are difficult to support with traditional routing protocols (see the references cited by Kreutz at al. [36] for a long list of examples). Some companies rely on SDN for parts of their network [33], but SDN has not replaced traditional routing protocols like BGP and OSPF/IS-IS.

In practice, deploying a new service inside a large ISP network can be difficult. Such a network is rarely composed of homogeneous routers produced by the same manufacturer. Most networks gather different types of routers from different vendors [10], [18]. The characteristics of a software-based virtual router used in a datacenter are different from those of an access router that connects a remote branch office or a backbone router that support hundreds of terabits/sec of traffic. Still, these different routers support the same packet format (IP) and implement the same standardised routing protocols (OSPF [41], BGP [47]). This standardisation ensures interoperability among routers from different vendors. However, it can also delay innovation as router vendors usually only implement features that have been standardised.

To illustrate the difficulty of deploying extensions to routing protocols, let us look back at the evolution of several BGP and OSPF extensions whose deployment has been documented.

The BGP communities [48] play an important role in scaling BGP routing policies by enabling network operators to tag routes and then apply the same policies to the routes that carry a given tag. Various use cases of this attribute have been documented [12], [23], [49]. This BGP attribute was designed when BGP used AS numbers encoded as a 16-bits integer and the high-order bits of the BGP communities contain an AS number. As BGP evolved to support 32-bits AS numbers [54], it became necessary to support wider BGP communities. Since 2009, the new ISPs receiving 32 bits AS numbers by default were unable to define their own BGP communities according to the existing standard [48]. Several encoding for BGP communities that support 32-bit AS numbers were proposed since 2002 [1], [46]. Unfortunately, discussions did not converge within the IETF and it took more than fifteen years to finally agree on the BGP Large Communities specification [29]. The redaction of this document took less than year, probably a record for the IETF and implementations were released in the following two years [1]. The first ISP that received a 32-bits AS number in 2009 had thus to wait more than a decade to be able to use its assigned BGP communities.

Another example is the BGP extensions to support the traffic engineering performance metrics [22]. The development of

* Quentin De Coninck is a F.R.S.-FNRS Research Fellow.
† Mathieu Jadin is supported by a grant from F.R.S.-FNRS FRIA.

978-1-7281-2700-2/19/$31.00 2019

[1] See http://largebgpcommunities.net/implementations/ for a detailed description of these implementations.

these extensions started in 2013 [58]. Six years later, it is now supported by only two vendors [2]. There are still ongoing discussions within the IETF and it remains difficult for ISPs to deploy this extension unless they exclusively use routers from the two vendors that support this extension.

A third example is the so-called add-path BGP extension [55] that enables a router to send several paths towards the same prefix over a single BGP session. The first discussions for this extension started in 2002 [56] and the IETF approved it in 2016. The first implementations [3] were reported in 2011 and then mainly in 2014.

A fourth example is the OSPFv3 LSA extensions [37] that extend the LSA format by encoding the existing OSPFv3 LSA information in Type-Length-Value (TLV). Thanks to these TLVs, it becomes easier to use OSPFv3 to flood other types of information than those covered by the standardised LSAs. The first discussions on this extension started in 2013 and only two implementations have been confirmed [4].

These examples show that while standardised routing protocols have clear benefits in terms of interoperability, it often takes half a decade or more before network operators can deploy new network services that require protocol extensions. Large companies like Facebook have reacted by implementing their own proprietary routing protocol [28], but there are no indications of its adoption outside Facebook.

In this paper, we propose a compromise between the flexibility of SDN where network operators can implement their own code and the benefits of distributed routing protocols. We focus on OSPF and BGP, but the solution that we propose is applicable to other routing protocols with some implementation effort. More precisely, we propose three main contributions in this paper.

- First, we propose in Section II to organise the implementations of routing protocols so that a network operator can extend the implementations used on her routers to support new protocol features.
- Second, we demonstrate that such an architecture can be implemented in the OSPF and BGP daemons of FRRouting.
- Third, we demonstrate in Section III, several use cases showing the benefits of our proposed implementation architecture.

We conclude the paper in Section V with a discussion of the benefits and drawbacks of our proposed approach.

II. Pluginizing a Routing Protocol

In this section, we propose a new technique to extend and enhance routing protocols and their implementations. Today's commercial and open-source implementations of routing protocols act as black boxes. From a high-level viewpoint, an implementation of a routing protocol can be represented as in Figure 1. The implementation is modelled as a Finite State Machine (FSM) that exchanges routing messages with other routers. The RFCs describe in details how and when protocol implementations should send and react to specific packets. This FSM can be configured through the command line interface, SNMP MIBs or Netconf and compute routing tables that are pushed in the FIB. With this blackbox model, any extension to the protocol requires a replacement of the FSM.

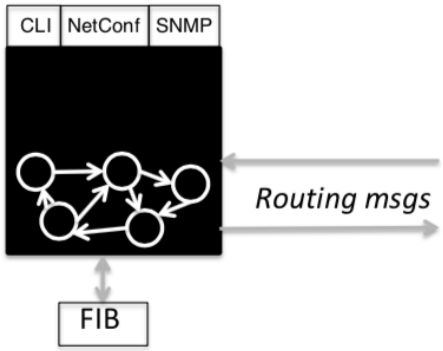

Fig. 1. Current routing protocol implementations.

We envision a different implementation model. From a high level viewpoint, our model is represented in Figure 2. We introduce three main modifications compared to the blackbox model. First, the protocol implementation provides a simple API that contains a set of functions that expose the protocol state. For example, an OSPF implementation typically includes functions to add or remove LSAs from the link state database, a BGP implementation includes functions to parse and encode BGP messages. Our second modification is that we allow the FSM that implements the protocol to be extended by adding one or more states, adding one or more transitions or replacing existing transitions. Figure 2 shows the FSM that enables the core part of the routing protocol in black and two extensions in red and blue. Our third modification is that we introduce **plugins**. A plugin is some executable code which can be executed inside a routing protocol implementation. A plugin can use the functions provided by the protocol API and extend the finite state machine. These plugins enable network operators to design their own extensions to routing protocols and deploy them in their networks without having to wait for their standardisation and adoption by multiple router vendors.

In SDN networks, operators can implement new services as software running on a centralised controller that interacts with the network devices through the Openflow protocol [39]. SDN controllers support different programming languages and a range of services have been implemented on them [36]. A network operator who wants to deploy a new service as a plugin would like to implement it once and deploy it on all routers inside her network.

The proposed deployment model has several important consequences on the implementation of our plugins. First, it

[2]See https://trac.ietf.org/trac/idr/wiki/draft-ietf-idr-te-pm-bgp\%20implementations.

[3]See https://trac.ietf.org/trac/idr/wiki/draft-ietf-idr-add-paths\%20implementations.

[4]See https://trac.ietf.org/trac/ospf/wiki/draft-ietf-ospf-ospfv3-extend\%20implementations

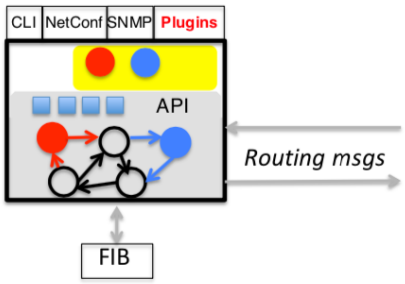

Fig. 2. Our proposed routing protocol implementations can be extended by using plugins that modify the FSM and use the API.

must be possible to execute a plugin on different types of routers that use different CPU models. This implies that either a plugin will be written using a programming language which can be interpreted by the protocol implementation or that it will be compiled into bytecode which is supported by a Virtual Machine that is included in the protocol implementation. Second, OSPF and BGP daemons are always active and it should be possible to extend them without restarting them. Third, since a plugin runs inside the OSPF/BGP daemon, there is a risk that an incorrect plugin could jeopardise the protocol state or even crash it. To cope with these three requirements, we compile the plugins into eBPF bytecode that is executed by a virtual machine that we include in the protocol implementation.

We provide more details on how we extended one implementation of BGP and OSPF to support plugins in the next sections. We first describe the key points of our solution in Section II-A. We detail the management of the memory in Section II-B. We then provide the details related to the OSPF and BGP daemons in Sections II-C and II-D.

A. Pluginizing FRRouting

To demonstrate the feasibility of this approach, we apply it to the OSPF and BGP daemons of FRRouting[5]. FRRouting (FRR) is an IP routing protocol suite for Linux and Unix platforms which includes protocol daemons for BGP, IS-IS, LDP, OSPF, PIM, and RIP. It was forked from Quagga and is actively maintained. We used FRRouting version 6.

To alter the behaviour of both OSPF and BGP, we rely on a user-space implementation of the eBPF [17] virtual machine called uBPF [32] that we linked to the FRRouting daemons. The main advantage of this virtual machine is that it supports the same bytecode as the eBPF virtual machine that is included in the Linux kernel. It can thus benefit from the different tools that have been written to compile bytecode for the Linux kernel. The uBPF VM can load executable eBPF bytecodes and either interpret them or compile them to x86 assembly with its own JIT compiler. Like the eBPF VM of the Linux kernel, it includes a verifier that checks the validity of the loaded bytecode. The uBPF verifier checks (1) all instructions are valid opcodes, (2) there is an `exit` instruction, (3) there is no forbidden operations such as division by zero, writes to read-only registers or invalid jumps, and (4) the memory accesses remain either in its stack or a provided memory area.

Now that the routing daemon includes the uBPF virtual machine, we need to discuss how the daemon must be restructured to enable it to be extended by using plugins. An implementation is organised as a series of functions that process and send packets as well as compute routing tables. These functions are the concrete implementation of specific states of the FSM protocol in a programming language such as C. Henceforth, to enable the modification of the protocol, we use these functions and make them **pluginizable**. These serve as **insertion points** where a network operator can decide to attach plugins compiled in eBPF.

More precisely, one plugin is associated with one routing function and is subdivided in three different parts, called **anchors**, offering a fine granularity on the code injection location. Consider the original function f. The anchors are illustrated in Figure 3 and are defined as follows.

- **PRE**: the eBPF code is executed just before running the body of the function f. This anchor can for example be used to load required data inside the plugin or for monitoring purposes. For example, the PRE anchor can monitor the FSM state transition to track the progress of the protocol. Any number of bytecode can be attached in this mode. They are then executed in a non-deterministic order, but they always terminate before the actual call of the function f. Bytecodes attached at PRE anchors only have read-only access to the routing daemon variables. If no eBPF code is present, this insertion point resumes to a no-op.
- **REPLACE**: the eBPF bytecode is executed instead of the original code of the function f. Only one bytecode can be attached in this mode, and the absence of injected code reduces the REPLACE mode to the original implementation of the function f. This implies that a network operator can dynamically replace one of the functions of the underlying implementation in deployed routers. Bytecodes in REPLACE anchors have read and write accesses to the routing daemon variables. This enable plugins to change main protocol algorithm such as the shortest path computation in the OSPF protocol by including custom network metrics. REPLACE can also be used to suggest new protocol features by redefining the definition of BGP import and export filters. BGP implementations typically propose a domain specific language (DSL) to design filters, but lack flexibility when designing complex filters. Traditional routing protocols implementations cannot rely on other type of information that the DSL proposes. eBPF can overcome this limitation since it has access to the arguments of the protocol function containing more information than DSL might propose.
- **POST**: this mode is similar to the PRE one, except that the eBPF code is executed just after running the body of the function f, just before returning to the function it was called from. Bytecodes attached at POST anchors only

[5]See https://frrouting.org

have read-only access to the routing daemon variables. As PRE function, the POST anchor can be used to monitor the time taken by protocol functions such as compute time in the Dijkstra algorithm in OSPF. In this case, both PRE and POST are required to track the time when the function starts and ends.

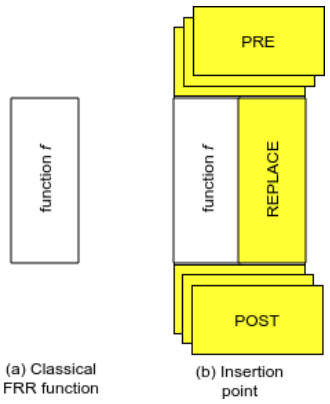

Fig. 3. Insertion points for eBPF Plugins inside routing functions.

Each pluginizable function has a name that uniquely identifies it. Such convention allows network administrators to easily attach and remove their eBPF scripts in a key-value data structure. Furthermore, such human-readable identifiers provide a convenient interface to dynamically change the plugins attached to the routing daemon without rebooting it. This latter method is effectively too restrictive, because routing sessions and routing tables must then be recomputed from scratch.

An eBPF plugin is composed of one or more bytecodes that are attached to a specific insertion point. These bytecodes are called **pluglets** and a plugin can thus contain several pluglets. A given implementation might expose many pluginizable functions. A plugin is defined in a description file listing each ELF file containing the eBPF bytecode and its corresponding function with its insertion point. The network administrator can load it through a command line interface (CLI). Several plugins can co-exist within a routing daemon. To be executed, pluglets require a VM. The current uBPF implementation provides an API to create a VM containing the loaded eBPF bytecode to execute it. However, such VM can only contain one bytecode at a time. Furthermore, with uBPF there is no API to update the code attached to a VM. The code replacement is nevertheless a required feature to dynamically update plugins, which is important for routing protocol daemons that never terminate. To solve these problems, we extend the API to manage multiple plugins. In fact, we create a specific uBPF machine which is in charge of only one pluglet. Several of these VMs can be attached to a routing daemon at a given time. These multiple VMs are stored into a map, each being associated to a plugin. This map is, of course, accessible through our extended uBPF API inside the routing daemon.

B. Memory Management

One of the motivations of using VMs is their isolation from the routing daemon they are attached to. In addition, the eBPF instruction set is quite small and simple, making it easier to control their operations. However, plugins may require more information from the implementation than the initial arguments provided to the VM. To exchange information with the routing daemon, FRRouting registers a set of functions that are made accessible to the uBPF VM, and therefore the plugins. As both pluglets and FRRouting are written in C language, plugins could theoretically access any memory location within the routing daemon. In practice, this could create stability problems if badly written eBPF code tries to access invalid memory locations. Furthermore, it would make plugins very dependent on the FRRouting internals that may change over time. To ensure the stability of the executable that combines the routing daemon and the eBPF plugins, we leverage the uBPF VM to control the memory that a given plugin can access. This is done through different techniques.

First, the routing daemon exposes through an API a set of getter and setter functions to access the main data structures (packets, LSDB for OSPF, RIB for BGP, etc.) maintained by the routing daemon. These functions are part of the modified routing daemon. They also verify the validity of their input parameters.

Second, the different pluglets composing a given plugin may need to collaborate together by exchanging information. Each pluglet is supported by one instance of the uBPF VM and has its own stack. To address this requirement, FRRouting keeps a dedicated **context** for each plugin. Thanks to this context, we can associate a plugin specific heap that is shared among the different pluglets that compose a plugin. These pluglets can allocate and free memory in their shared heap by using functions that are similar to `malloc(3)` and `free(3)`. This is illustrated in Figure 4. As we want to keep control on the memory used by the plugins, we do not directly expose the associated functions of the C library. Rather, we reimplement some of them like `memcpy(3)`, `malloc(3)` and `free(3)` and expose them to the uBPF VMs. In addition, the API of the routing daemon provides functions for pluglets to map an area of the plugin heap to a plugin specific identifier. Such mechanisms enable collaborative pluglets to retrieve a precise memory area while providing isolation between plugins.

C. Pluginizing the OSPF Daemon

The previous sections describe the generic techniques that are required to add plugins to a routing daemon. In addition, the routing daemon also needs to expose specific OSPF functions that the plugins can use. We briefly describe these OSPF functions and the insertion points in this section.

The insertion points of an OSPF daemon are the protocol functions where eBPF plugins can be attached. These insertion points depend on the features that eBPF plugins need to support. Our current prototype includes several insertion points. We briefly describe some of them. The `SPF_CALC` insertion point corresponds to the function that computes the shortest

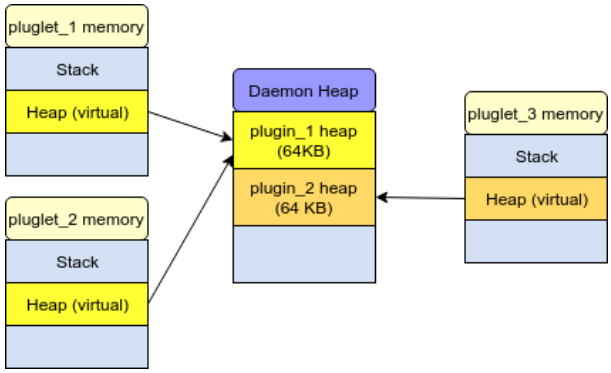

Fig. 4. The two pluglets of the left plugin share the same heap while the pluglet of the right plugin uses a separate heap.

paths. The `OSPF_SPF_NEXT` insertion point corresponds to a function which is part of the SPF calculation process that implements Section 16.1 of the OSPF specification [42]. The `HELLO_SEND` insertion point corresponds to the function that sends Hello packets. The `LSA_FLOOD` insertion point corresponds to the function that floods the received LSAs. The `ISM_CHANGE_STATE` insertion point corresponds to the function that is called when an interface changes the state of its Interface State Machine.

The OSPF API also exposes some functions to the plugins. First, we expose functions used to get/set some OSPF internal structures. For example the `get_ospf_area` function is used to get a copy of an OSPF area structure from OSPF while `get_ospf_area` can be used to set an OSPF area structure to a desired value. Such functions are provided for most of the important structure maintained by OSPF. We also expose functions from the implementation that can be useful for plugins. Examples of such functions are `plugin_ospf_flood_through_area` that allows to flood an LSA through an area and `plugin_ospf_lsa_install` that allows to install an LSA in the router's LSDB.

D. Pluginizing the BGP Daemon

The BGP daemon is also extended similarly. We add insertion points on functions receiving BGP messages from neighbours, on filters and inside the decision process. We also expose specific functions to the plugins that are executed by the uBPF VM.

Our BGP API exposes two types of functions to the eBPF plugins. First, there are functions to access/modify some elements of the data structures maintained by the BGP daemon. For example, `get_cmp_prefixes` is used to retrieve two prefixes received during the BGP decision process. The first one is a prefix received from the remote peer when it has sent a BGP Update message. The second prefix is one prefix already present in the Adj-Loc-RIB. The `get_attr_from_prefix` returns the attribute structure related to the prefix sent by a remote peer. The `as_path_from_prefix` function returns the AS path related to a prefix while the `get_attr_from_path_info` returns all the attributes of the prefix passed as argument. The `get_community_from_path_info` extracts the BGP community structure associated with a given path.

Our BGP API also includes functions that manipulate and compare BGP messages or their attributes. These functions are typically used by the BGP decision process and will be used for one of our use cases. Example functions include `aspath_cmp` that compares two AS paths, `aspath_count_hops` that returns the number of ASes contained in a given path, similar functions for the MED or other BGP attributes or the `peer_sort` which determines where a peer is an eBGP or an iBGP neighbour.

III. Use Cases

In this section, we describe four examples showing how network operators and researchers can leverage the proposed plugins to extend a routing protocol.

First, we demonstrate in Section III-A that we can use plugins to extract and expose internal protocol information for monitoring. Second, we show in Section III-B plugins changing the protocol packet format and its interpretation in the OSPF route computation. Third, we show in Section III-C plugins describing more expressive BGP filters. Fourth, we demonstrate in Section III-D that plugins can also modify the BGP decision process.

A. Monitoring routing protocols

One of the most popular use cases for eBPF in the Linux kernel is to monitor various events that occur inside the kernel in an efficient and non-intrusive manner. Similarly, we added monitoring facilities to the BGP and OSPF implementations in FRRouting.

To illustrate the monitoring capabilities of our proposed plugins, we have designed and implemented both a BGP and an OSPF monitoring daemons that interact with plugins running on the routing daemons and exports statistics using IPFIX [9]. Those statistics are aggregated by the daemons and exported to an IPFIX collector.

To monitor the BGP routing daemon, we implemented several BGP plugins that are attached at PRE anchors at several insertion points. Some of these plugins monitor specific BGP messages. For example, our plugin monitoring the Open messages, used to start a BGP session, is composed of 50 lines of C code and uses 16 external calls. Similar plugins are provided for the Keepalive and Update messages. Besides monitoring the received BGP messages, one plugin also measures the time required to run the BGP decision process. We have also implemented plugins that track specific IP prefixes or analyse the received AS Paths to enable the operator to provide more detailed statistics. Finally, we built plugins in charge of both monitoring withdrawn and rejected routes. The last one provides the reason of the reject decided by BGP import filters. We study the performance impact of these plugins in Section III-C.

We also implemented similar plugins to monitor OSPF. These plugins are inserted at the PRE and POST anchors of different insertion points. With plugins shorter than 10 lines of code, we can monitor things such as the execution time of the SPF calculation process, the number of Hello packets sent or the LSAs flooded by a router.

B. More flexible OSPF route computation

One of the benefits of our proposed plugins is that it is possible to extend the routing protocol. As an illustration, we implement a new type of OSPF LSA and update the shortest path computation algorithm. This idea is similar to the flexible IGP algorithm that is currently being discussed within the IETF [44]. We do not adopt the syntax proposed in the IETF drafts, but the idea is similar.

We first define a new OSPF LSA (type 13). This LSA is similar to the normal router LSA (type 1), except that we associate an additional metric (as an integer) to each link. We use this additional metric to represent the colour of each link. This plugin is implemented by using about 100 lines of code and is inserted in the `SPF_CALC` insertion points. This LSA is then flooded inside the network.

Our second plugin is attached at the `OSPF_SPF_NEXT` insertion point at REPLACE anchor. It modifies the `ospf_spf_next()` function which implements Section 16.1 of the OSPF specification [42]. In this function, the LSDB is represented as a directed graph. The `ospf_spf_next()` function examines the links in the LSAs of the first vertex from the candidates list. Then it updates the list of candidates with any vertices that are not already on the list. If a lower-cost path is found to a vertex already present in the candidate list, it stores the new cost. We rewrite this function as an eBPF plugin to support colour constraints. For each router LSA that is examined, we check if there is a corresponding (same router-ID) LSA of type 13 in the LSDB. If yes, we check for each link the colour of the link. If it is green, we continue normally, if it is red, we ignore the link. This plugin is implemented in about 160 lines of C code.

As an illustration of the utilisation of this new LSA, we simulated the network topology shown in Figure 5. In this network, all the links have a cost of 100. With the standard Dijkstra algorithm, router `R1` uses its direct link to reach both `R2` and `R3`. With our type 13 LSA, `R1` advertises a different colour (red) for the directed link between itself and `R3`. This LSA is flooded in the network and our second plugin running on the different routers computes the routing tables without considering the red link. In this configuration, `R1`'s routing table forwards packets destined to `R3` via `R2`. On the other hand, `R3` continues to use its direct link to reach `R1`.

To evaluate the performance of these OSPF plugins, we loaded the OSPF router with the topology of the GTS Central Europe network from the Internet Topology Zoo [35]. It contains about 150 nodes and the same number of edges. This is one of the largest topologies from this public dataset which makes it a good candidate to evaluate this plugin overhead.

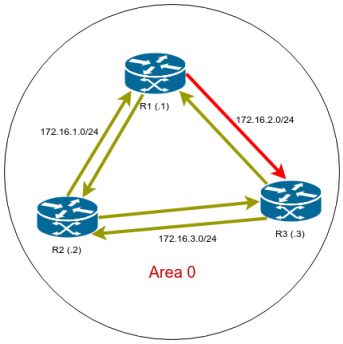

Fig. 5. Simple OSPF network.

We used this experimental setup to evaluate the memory and CPU consumption of our plugins. For this, we consider three different versions of the OSPF daemon: (i) the **vanilla OSPF** daemon from FRRouting version 6, (ii) our **flexible OSPF daemon** but without any plugin and (iii) our **flexible OSPF daemon with two plugins** installed (the two plugins that allow changing the Dijkstra computation using our new type of LSA). Looking at the memory consumption, we observe that without plugins, our **flexible OSPF daemon** consumes 4.93 MBytes of memory while the **vanilla** one only consumes 4.85 MBytes. This difference is due to the additional data structures required to support the management of plugins. With the two plugins loaded, the memory consumption grows to 5.23 MBytes. The difference between the flexible OSPF without plugins and the one with plugins is due to several reasons. First, a 64 KBytes heap is dedicated to each plugin when it starts. This heap remains allocated for the future executions of the plugin. Second, the bytecodes of the plugins consume between 1 and 10 KBytes per plugin. Third, when bytecodes are injected, the implementation stores some more metadata related to it and the uBPF VM also maintains data related to the VM state. All this together leads to about 300 KBytes of overhead for two plugins. This seems reasonable for today's routers.

To evaluate the CPU cost of the plugins, we focus on the computation of the shortest paths, which is the most important algorithm used by an OSPF daemon. To measure the CPU time required to compute the shortest path, we rely on the following experiment. We start the router under test, let it download the entire LSDB from its neighbour and measure the time required to compute the shortest path after the full transfer of the LSDB. Figure 6 provides the CPU times measured over 50 different runs with four variants of our flexible OSPF daemon. Our baseline is the **vanilla OSPF** daemon which takes on average 5.34 msec to compute the shortest paths. Our flexible OSPF daemon takes roughly 5.49 msec without plugins and slightly more with the monitoring plugins.

The eBPF plugins can either be interpreted or compiled by the JIT compiler of the uBPF VM. The plugin that supports our Type 13 LSA inside the shortest path computation is composed of about 160 lines of C code. It is executed for each node/edge visited during the shortest path computation. When this plugin

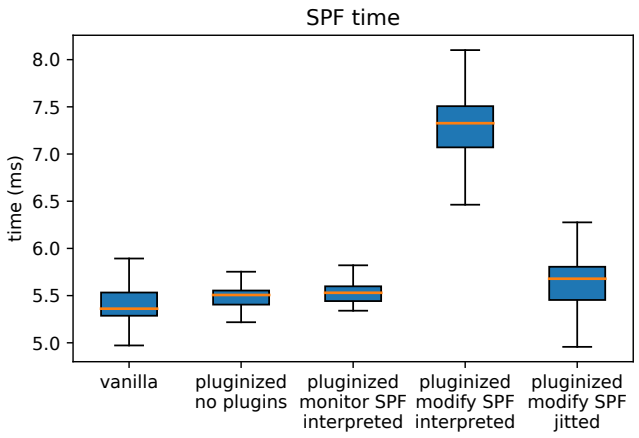

Fig. 6. SPF execution times over 50 runs on the emulated 150 nodes GTS Central Europe topology.

TABLE I
DATA ABOUT EBPF PLUGINS FOR THE DECISION PROCESS.

eBPF Function	API Calls	eBPF Insts	LoC
Local Pref	21	85	51
As Path	19	91	52
MED Check	25	147	58
IGP Weight	76	338	134
Router ID cmp	26	110	54

is interpreted, the shortest path computation increases up to 7.3 msec. However, once the plugin is compiled by the JIT, the shortest path computation time drops to 5.66 msec and thus the overhead remains small compared to vanilla OSPF.

C. More flexible BGP filters

FRRouting, like most BGP implementations, support a range of import and export filters. A network operator can define access-lists that defined the list of prefixes which are accepted/rejected. It is also possible to specify a prefix-list which can also match on the prefix length. FRRouting also supports filters that match on the AS-Path and route-maps which can match on other attributes such as BGP Communities, the origin of a route, the peer that announced a route. Such filters are widely used by network operators [14] and some router configurations contain thousands or tens of thousands of lines of configuration files to specify them.

Although route-maps are the most flexible BGP filters, their configuration might become cumbersome and complex [14]. Our proposed eBPF plugins enable network operators to write filters in C code which is much more expressive than the ad-hoc languages that have been defined by router vendors to support filters. Furthermore, such eBPF filters could access to additional information, such as the current state of the protocol.

The filtering process is supported per peer and per prefix and defined in a single function inside FRRouting. We added an insertion point for the eBPF virtual machine inside this function. However, there are situations where an operator could want to attach several eBPF plugins to this filtering process. Given that the order of the application of the filter functions can be important for the decision of the filter, we allow the network operator to specify the order in which different REPLACE plugins will be executed for this filtering function. These REPLACE functions are actually multiple dummy no-ops anchors that are executed before actual FRRouting filtering function.

An eBPF plugin for the filtering function can return three different results: `FILTER_DENY` if the filter has decided to reject the route, `FILTER_ACCEPT` if the filter has decided to accept the route and `BGP_CONTINUE` if the next filter needs to be applied. The uBPF virtual machine executes the different BGP plugins in the order specified by the network operator when it loaded them and stops the processing as soon as one of them returns `FILTER_DENY` or `FILTER_ACCEPT`.

We previously mentioned that a filter could modify protocol variables. As for traditional filters, the virtual machine enables an eBPF filter to modify attributes such as the local-preference, the MED, the AS-PATH (for path prepending for example), BGP communities, etc. The eBPF filters can also read the current RIB of the BGP router. This could bring new filter possibilities based on the RIB content.

Another advantage of eBPF filters is that it becomes easy to manipulate BGP communities. Many network operators use BGP communities for a wide range of purposes [12]. Measurements indicate that BGP routers rarely remove the BGP communities that upstream routers attached. This increases the size of the BGP routing tables and the memory consumption on routers and opens a range of operational problems [49]. With eBPF, the programmer defines which communities to match without maintaining a list of prefixes as current route-map. An eBPF filter can easily add, delete and compare BGP Communities since the filter is written in C.

We developed several BGP import filters as eBPF plugins to illustrate the flexibility of our proposed architecture and evaluate the performance impact of these new filters. Our first plugin is used as an import filter. It simply parses the AS-Path as only accepts the routes advertised by an odd-numbered AS. This filter only requires 5 lines of C code. We do not expect that network operators who want to use it in their network, but we use it as a simple benchmark to evaluate the performance impact of the eBPF plugins.

Our second eBPF plugin is more useful for network operators. The BGP routing tables in the default-free Internet continue to grow. Recent data [6] shows that routers of RouteView project need to carry more than 800k routes. Recent routers can easily handle such large routing tables, but many smaller ISPs and enterprise networks still use older routers that have limited memory. On such routers, it makes sense

[6]See http://bgp.potaroo.net/.

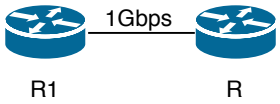

Fig. 7. Network topology used to evaluate the performance of the BGP filters implemented as eBPF plugins.

to only accept a subset of the routes to avoid overflowing the available memory. Many ISPs use filters to block IPv4 prefixes that are too long (e.g. `/24`) [53], [7]. However, these filters block some legitimate prefixes. Measurement studies have shown that a small fraction of the ASes that advertise prefixes are responsible for the pollution of the BGP routing tables by advertising many more specific prefixes that are covered by a less-specific one [7]. Some ASes advertise both a `/20` IPv4 prefix and all the `/24` subprefix that it contains. Our second eBPF plugin automatically detects those ASes that de-aggregate their large prefix and only accepts the first 4 more specific prefixes that are included inside a larger one that is already included in the router's RIB. This eBPF plugin is implemented in 19 lines of C code. When a BGP route is received, the eBPF plugin verifies whether it is already covered by a less specific prefix that already includes 4 more specific prefixes. If so, the route is rejected, otherwise it is accepted.

To evaluate the performance of these two filters, we consider the simple scenario shown in Figure 7. Router `R1` uses `exabgp` to inject a BGP routing table [7] containing 200K entries to router `R` over an eBGP session. Router `R` uses different versions of FRRouting. The machine running router `R` is equipped with Intel(R) Core(TM) i3 CPU 540 @ 3.07GHz running on Linux kernel 5.0.13, 12GB of RAM and 1 Gbps NIC.

Our baseline for the evaluation of the performance impact of the eBPF filters is the utilisation of FRRouting without any filter. We add to FRRouting an eBPF plugin that monitors the insertion time of each prefix in the router FIB. Thanks to this plugin, we plot on Figure 8 (dotted blue curve) the time required to process the BGP updates received from `R1`. The green curve shows the time required to process the same BGP updates with our second eBPF plugin that filters the more specific prefixes. This filter rejects 13k of the 200k routes and only increases the processing time by 5.4%. The eBPF filter that rejects the routes advertised by an odd-numbered AS processes all the BGP routes in only 12.23 seconds, but it rejects half of them. Since it rejects many routes, the BGP daemon has to perform less computation than when there is no plugin.

The dotted magenta curve on Figure 8 shows the time required to process all the BGP updates sent by `R1` without any filter but with the 7 eBPF monitoring plugins described in Section III-A installed.

[7]For this evaluation, we really on the BGP routing table from Spotify's `super-smash-brogp` project, see https://github.com/spotify/super-smash-brogp.

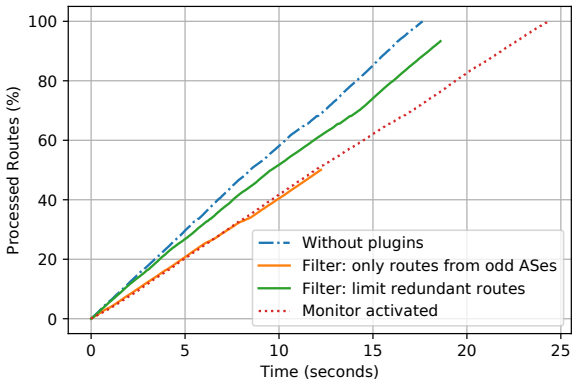

Fig. 8. Performance of the BGP filters implemented as eBPF plugins.

D. Pluginizing the BGP Decision Process

Our last use case is the BGP decision process. This is a key part of the BGP daemon that controls the selection of the best path towards each destination prefix. Network operators use various techniques to influence the selection of these best paths [15], [45], [50]. Some routers can be configured to skip some steps of the BGP decision process or slightly modify their behaviour [16]. For example, many BGP implementations support a configuration parameter to always compare the MED attribute even between routes that were received from different peers.

In the FRRouting BGP daemon, the decision process is implemented as a single function (`bgp_path_info_cmp`). We refactor the FRRouting code to organise this function such that it now calls one specific function per step of the BGP decision process. Each of these steps is then implemented as a separate function. These functions are all implemented following the same pattern. They compare a new path with the best one that is already present in the BGP routing table. If the new path is strictly better based on the attributes that are compared in this step of the decision process, then the function returns `BGP_COMP_SPEC_2`. If the best current path is strictly better than the new one, the function returns `BGP_COMP_SPEC_1`. If the two paths are equivalent according to the attributes considered in this step of the decision process, then the return value of the function returns the next step that needs to be executed. This makes it possible to fully customise the BGP decision process, not only replacing one step with another, but changing the order of the rules of the BGP decision process.

Thanks to the utilisation of eBPF plugins, network operators can easily tune the BGP decision process of their routers. Many network operators use BGP communities to tag the Point of Presence (PoP) or the city where a given route was learned [12]. We leverage this to implement a variation on hot-potato routing that uses the geographical distance between PoPs to prefer one route over the other. Each PoP is encoded as a BGP community and our eBPF plugin contains a table with the latitudes and longitudes of all the PoPs of the ISP. When two routes are compared, the eBPF plugin computes the

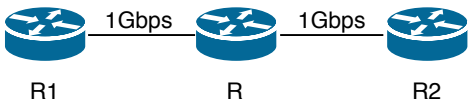

Fig. 9. Network lab used to evaluate the pluginized BGP decision process.

distance between them based on the geographical coordinates of the PoPs where they were received and always prefer the closest one. This eBPF plugin is implemented in 148 lines of C code.

To evaluate the cost of using eBPF plugins within the BGP decision process, we reimplement all the steps of the decision process as plugins. In FRRouting version 6, there are 14 different steps of the decision process. The eBPF plugin that supports the `local-pref` attribute requires 51 lines of code. This is one of the simplest steps of the BGP decision process. The most complex eBPF plugin is the one that compares the IGP cost towards the BGP nexthop. This eBPF plugin is implemented using 134 lines of C code. We do not expect that network operators will replace all the steps of the BGP decision process with eBPF plugins, but use this as our worst-case scenario to evaluate the performance penalty of these eBPF plugins.

We consider the network show in Figure 9. Router `R1` sends a full BGP routing table containing 200k routes. Once router `R` has accepted all the routes announced by `R1`, router `R2` starts to announce exactly the same routes. Every route sent by `R2` must be evaluated by all the steps of the BGP decision process on router `R` that eventually prefers the new one because of its `router-id`, i.e. the last step of the BGP decision process.

We consider different variants of our modified version of FRRouting and measure the execution time of the BGP decision process to fully process routes sent by `R2`. As for evaluating filter performances in Section III-C, we use the same eBPF plugin that monitors the insertion time of each re-advertised prefix in the router FIB. With the vanilla BGP daemon, the dotted blue curve of Figure 8 shows that it takes on average 17.5 seconds to accept all the routes sent by router `R2` and install them in the FIB of router `R`. If we use our modified version of FRRouting that supports eBPF plugins but do not install any of them, router `R` needs up to 28 seconds to install all the routes sent by `R2` in its FIB. Finally, when all the steps of the BGP decision process are implemented as plugins, router `R` needs almost 34.8 seconds to install the same number of routes in its FIB. We instrumented our code to analyse the reason for this high cost of the pluginized BGP decision process and have identified that the simple linked-list used by the memory allocator of our prototype was the culprit. We are currently rewriting this part of the code and expect that the new memory allocator will improve the performance of the pluginized BGP decision process.

IV. Related Work

Based on feedback from their customers, router vendors have implemented various techniques to control the operation of routing protocols. The Command Line Interface (CLI) is the classical way for network operators to tune the configuration of the routing protocols running on their routers. Some also rely on SNMP MIBs to gather statistics and some simple configuration tasks [27], [34]. Over the years, router vendors have added new techniques to enable their customers to interact with the router software. Some vendors provide scripting facilities [38], [6] and the industry is now heading towards the utilisation of Yang models [8]. However, these approaches do not enable network operators or researchers to extend the underlying protocols.

The eBPF virtual machine has been introduced in the Linux kernel a few years ago. It is now mainly used for configuration and monitoring purposes [19]. Looking at the networking use cases, eBPF is used to provide fast programmable data packet processing [30], improve firewalls [2], implement network services [40], support IPv6 extensions [59], extend TCP [52] or implement Multipath TCP schedulers [20]. We are not aware of applications of eBPF to routing protocols.

In the late nineties active networks were proposed as a solution to bring innovation back inside the network that was perceived as being ossified [51]. Most of the work in this area focused on the possibility of placing bytecode inside network layer packets. This bytecode was then executed by virtual machines running on routers. The idea of placing code inside packets was not adopted by the industry [5], but P4 [3] could be considered as a modern variant of this idea. In the control plane, researchers built upon this idea to propose new solutions such as the 4D architecture [24], the Routing Control Platform that centralises routing [4] or Metarouting [25] that proposed to open the definition of routing protocols using a declarative language.

Although the eBPF plugins proposed in this paper were applied to BGP and OSPF, the same technique could be used with other control plane protocols. There are several ongoing efforts to develop new routing protocols that could benefit from such plugins. Some examples include Facebook's Open/R routing platform [28] or the protocols that are being designed within the LSVR, RIFT or BABEL IETF working groups.

V. Discussion and next steps

We started the paper by explaining that although standardised routing protocols are important from an interoperability viewpoint in multivendor networks, the need to reach agreements between all these vendors in standardisation bodies delays the deployment of new protocol extensions which hinders innovation. To enable network operators who are responsible for Internet Service Provider networks to innovate more quickly, we have proposed that routing protocols support platform independent protocol plugins that extend them. We have demonstrated how one particular implementation of BGP and OSPF, namely FRRouting, could be extended to support these plugins and thus made one step towards the realisation of our vision. We now discuss several research problems that need to be solved to fully realise this vision.

How to validate plugins? Network operators, notably in enterprise and ISP networks, usually test new versions of the router operating systems in their labs before deploying them in production networks. These tests are intended to prevent problems when new versions of the software are deployed in production. They typically run for days or weeks and are only used before major upgrades. With protocol plugins, network operators will operate in a more agile manner. They could change deployed plugins on a daily basis. However, they are unlikely to perform manual tests with such plugins. A better approach would be to use automated tools that leverage formal methods to validate that the plugins are correct. The Linux kernel, which also includes an eBPF virtual machine, solves this problem by strictly limiting the memory area that eBPF programs can access and the number of instructions that they execute. The latter limitation is being removed to support more complex use cases. Researchers have recently proposed software tools that rely on formal methods to validate such eBPF code [21]. Pluginized QUIC [11], which applies a similar idea to extend the QUIC protocol also uses verification technique to prove than plugins terminate. This is a first step for the validation of plugins.

How can we increase the performance of plugins? Plugin performance should be as close as possible to the performance of native code, but this performance gain should not come at the expense of the safety guarantees. There are two possible directions to improve performance. A first approach is to try to extract as much performance as possible from our modified eBPF virtual machine that runs in userspace. One of the current performance bottlenecks is the memory accesses that are audited by our virtual machine to prevent out-of-bound memory accesses. These checks consume CPU time and affect performance. One possibility to reduce them would be to develop a plugin compiler that ensures that JITed plugins only access authorised memory areas. Such tools exist for standard executables [43]. Another approach would be to replace the eBPF virtual machine with a faster one. Some processor architectures, such as CHERI-MIPS [57], provide such memory checks as built-in assembly instructions. The WebAssembly virtual machine that is being deployed by web browsers could be another alternative [26].

Can we use the same plugin on different implementations? Network operators often use network equipment from multiple vendors. For them, it would be ideal if the same bytecode could be executed on different routers. This is not yet possible with the current version of our prototype, but this could become possible in the future provided that: (i) router vendors agree on including the same virtual machine in their software and (ii) they expose exactly the same API. We believe that this could be possible in the long-term. In different domains, browser vendors have agreed on supporting the WebAssembly virtual machine in their browsers [26] and operating systems vendors have agreed on using the POSIX API. Our next step to reach that goal will be to add the support for plugins in a different BGP/OSPF implementation and then align the API of these two implementations to produce plugins that are implementation dependant. We will then be able to propose a first plugin API for discussion with other vendors and within the IETF.

How should future protocols leverage this extensibility? Most standardised protocols are designed by committees. This process has advantages and drawbacks. By involving more people in their design, the new protocols have a higher probability of meeting the user/operator's requirements. However, these committees sometimes tend to over specify and add new features that require long discussions before reaching an agreement. The design of an extensible routing protocol could be done differently. A small committee could start with a small set of basic functionalities, a virtual machine and a simple API. Based on these specifications, the community could then openly develop extensions to meet their specific requirements. Instead of trying to match all the expressed requirements, the protocol designers would focus on getting the basic principles of the core protocol right. If some of these extensions become popular or require better performance than the one achievable with virtual machines, they could later be included in a new version of the protocol.

Software artifacts

To encourage other researchers to reproduce and extend our results, our patches to FRRouting version 6 and the scripts required to reproduce our measurements are available at the GitHub repository https://github.com/twirtgen/pluginized-frrouting.

ACKNOWLEDGEMENTS

This work was partially supported by FRIA. We thank the anonymous reviewers and our shepherd, Kamil Sarac, for their valuable comments. We would also like to thank Job Snijder for his comments on BGP implementations and the authors of the FRRouting and the uBPF open-source projects.

REFERENCES

[1] Lange Andrew et al. Flexible bgp communities. Internet draft, draft-lange-flexible-bgp-communities-03, work in progress, August 2010.

[2] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. Accelerating linux security with ebpf iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 108–110. ACM, 2018.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[4] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 15–28. USENIX Association, 2005.

[5] Ken Calvert. Reflections on network architecture: an active networking perspective. *ACM SIGCOMM Computer Communication Review*, 36(2):27–30, 2006.

[6] Cisco. Cisco ios scripting with tcl configuration guide. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ios_tcl/configuration/12-4t/ios-tcl-12-4t-book/nm-script-tcl.html, 2011.

[7] Luca Cittadini, Wolfgang Muhlbauer, Steve Uhlig, Randy Bush, Pierre Francois, and Olaf Maennel. Evolution of internet address space deaggregation: myths and reality. *IEEE Journal on Selected Areas in Communications*, 28(8):1238–1249, 2010.

[8] Benoit Claise, Joe Clarke, and Jan Lindblad. *Network Programmability with YANG*. Addison-Wesley, 2019.

[9] B. Claise (Ed.), B. Trammell (Ed.), and P. Aitken. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. RFC 7011 (Internet Standard), September 2013.

[10] Guy Davies. *Designing and Developing Scalable IP Networks*. John Wiley & Sons, 2004.

[11] Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, and Olivier Bonaventure. Pluginizing quic. In *SIGCOMM'19*, pages 59–74. ACM, 2019.

[12] Benoit Donnet and Olivier Bonaventure. On bgp communities. *ACM SIGCOMM Computer Communication Review*, 38(2):55–59, 2008.

[13] John William Evans and Clarence Filsfils. *Deploying IP and MPLS QoS for multiservice networks: Theory and practice*. Elsevier, 2010.

[14] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 43–56. USENIX Association, 2005.

[15] Nick Feamster, Jay Borkenhagen, and Jennifer Rexford. Guidelines for interdomain traffic engineering. *ACM SIGCOMM Computer Communication Review*, 33(5):19–30, 2003.

[16] Nick Feamster and Jennifer Rexford. Network-wide prediction of bgp routes. *IEEE/ACM Transactions on Networking (TON)*, 15(2):253–266, 2007.

[17] Matt Fleming. A thorough introduction to ebpf. *Linux Weekly News*, December 2017. https://old.lwn.net/Articles/740157/.

[18] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 469–483, 2015.

[19] Lorenzo Fontana and David Calavera. *Linux Observability with BPF*. O'Reilly, 2019.

[20] Alexander Frömmgen, Amr Rizk, Tobias Erbshäußer, Max Weller, Boris Koldehofe, Alejandro Buchmann, and Ralf Steinmetz. A programming model for application-defined multipath tcp scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 134–146. ACM, 2017.

[21] Elazar Gershuni et al. Simple and precise static analysis of untrusted linux kernel extensions. In *PLDI'19*, June 2019. https://research.vmware.com/publications/simple-and-precise-static-analysis-of-untrusted-linux-kernel-extensions.

[22] L. Ginsberg (Ed.), S. Previdi, Q. Wu, J. Tantsura, and C. Filsfils. BGP - Link State (BGP-LS) Advertisement of IGP Traffic Engineering Performance Metric Extensions. RFC 8571 (Proposed Standard), March 2019.

[23] Vasileios Giotsas, Georgios Smaragdakis, Christoph Dietzel, Philipp Richter, Anja Feldmann, and Arthur Berger. Inferring bgp blackholing activity in the internet. In *Proceedings of the 2017 Internet Measurement Conference*, pages 1–14. ACM, 2017.

[24] Albert Greenberg, Gisli Hjalmtysson, David A Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4d approach to network control and management. *ACM SIGCOMM Computer Communication Review*, 35(5):41–54, 2005.

[25] Timothy G Griffin and Joäo Luís Sobrinho. Metarouting. *ACM SIGCOMM Computer Communication Review*, 35(4):1–12, 2005.

[26] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. *ACM SIGPLAN Notices*, 52(6):185–200, 2017.

[27] J. Haas (Ed.) and S. Hares (Ed.). Definitions of Managed Objects for BGP-4. RFC 4273 (Proposed Standard), January 2006.

[28] Saif Hasan, Petr Lapukhov, Anuj Madan, and Omar Baldonado. Open/R: Open routing for modern networks. https://code.fb.com/connectivity/open-r-open-routing-for-modern-networks/.

[29] J. Heitz (Ed.), J. Snijders (Ed.), K. Patel, I. Bagdonas, and N. Hilliard. BGP Large Communities Attribute. RFC 8092 (Proposed Standard), February 2017.

[30] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David 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*, pages 54–66. ACM, 2018.

[31] Geoff Huston. *ISP survival guide: strategies for running a competitive ISP*. John Wiley & Sons, Inc., 1998.

[32] IO Visor Project. Userspace ebpf vm. https://github.com/iovisor/ubpf, 2018.

[33] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[34] D. Joyal (Ed.), P. Galecki (Ed.), S. Giacalone (Ed.), R. Coltun, and F. Baker. OSPF Version 2 Management Information Base. RFC 4750 (Proposed Standard), December 2006.

[35] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[36] Diego Kreutz, Fernando MV Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[37] A. Lindem, A. Roy, D. Goethals, V. Reddy Vallem, and F. Baker. OSPFv3 Link State Advertisement (LSA) Extensibility. RFC 8362 (Proposed Standard), April 2018.

[38] Jonathan Looney and Stacy Smith. *Automating Junos Administration: Doing More with Less*. " O'Reilly Media, Inc.", 2016.

[39] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[40] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network service with ebpf: Experience and lessons learned. *High Performance Switching and Routing (HPSR). IEEE*, 2018.

[41] J. Moy. OSPF Version 2. RFC 1247 (Draft Standard), July 1991. Obsoleted by RFC 1583, updated by RFC 1349.

[42] J. Moy. OSPF Version 2. RFC 2328 (Internet Standard), April 1998. Updated by RFCs 5709, 6549, 6845, 6860, 7474, 8042.

[43] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 2–14. IEEE Press, 2019.

[44] Peter Psenak et al. IGP Flexible Algorithm. Internet draft, draft-ietf-lsr-flex-algo-02.txt, work in progress, May 2019.

[45] Bruno Quoitin, Cristel Pelsser, Louis Swinnen, Olivier Bonaventure, and Steve Uhlig. Interdomain traffic engineering with bgp. *IEEE Communications magazine*, 41(5):122–128, 2003.

[46] Robert Raszuk et al. BGP Community Container Attribute. Internet draft, draft-ietf-idr-wide-bgp-communities-05, work in progress, July 2018.

[47] Y. Rekhter (Ed.) and T. Li (Ed.). A Border Gateway Protocol 4 (BGP-4). RFC 1654 (Proposed Standard), July 1994. Obsoleted by RFC 1771.

[48] S. Sangli, D. Tappan, and Y. Rekhter. BGP Extended Communities Attribute. RFC 4360 (Proposed Standard), February 2006. Updated by RFCs 7153, 7606.

[49] Florian Streibelt, Franziska Lichtblau, Robert Beverly, Anja Feldmann, Cristel Pelsser, Georgios Smaragdakis, and Randy Bush. Bgp communities: Even more worms in the routing can. In *Proceedings of the Internet Measurement Conference 2018*, pages 279–292. ACM, 2018.

[50] Renata Teixeira, Aman Shaikh, Tim Griffin, and Jennifer Rexford. Dynamics of hot-potato routing in ip networks. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):307–319, 2004.

[51] David L Tennenhouse and David J Wetherall. Towards an active network architecture. *ACM SIGCOMM Computer Communication Review*, 26(2):5–17, 1996.

[52] Viet-Hoang Tran and Olivier Bonaventure. Beyond socket options: making the linux tcp stack truly extensible. In *IFIP Networking'19*, 2019.

[53] Gaurab Raj Upadhaya. Best practices for ISPs. http://www.pch.net/resources/tutorial/ispbcp.

[54] Q. Vohra and E. Chen. BGP Support for Four-Octet Autonomous System (AS) Number Space. RFC 6793 (Proposed Standard), December 2012.

[55] D. Walton, A. Retana, E. Chen, and J. Scudder. Advertisement of Multiple Paths in BGP. RFC 7911 (Proposed Standard), July 2016.

[56] Daniel Walton et al. Advertisement of Multiple Paths in BGP. Internet draft, draft-walton-bgp-add-paths-00, work in progress, May 2002.

[57] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 457–468. IEEE, 2014.

[58] Qin Wu et al. BGP attribute for North-Bound Distribution of Traffic Engineering (TE) performance Metrics. Internet draft, draft-wu-idr-te-pm-bgp, work in progress, Oct 2013.

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