xBGP: When You Can’t Wait for the IETF and Vendors

Thomas Wirtgen
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium
thomas.wirtgen@uclouvain.be

Quentin De Coninck
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium
quentin.deconinck@uclouvain.be

Randy Bush
IIJ Research & Arrcus
Bainbridge Island, WA, USA
randy@psg.com

Laurent Vanbever
NSG, ETH Zürich
Zürich, Switzerland
ivanbever@ethz.ch

Olivier Bonaventure
ICTEAM, UCLouvain
Louvain-la-Neuve, Belgium
olivier.bonaventure@uclouvain.be

ABSTRACT
Thanks to the standardization of routing protocols such as BGP, OSPF or IS-IS, Internet Service Providers (ISP) and enterprise networks can deploy routers from various vendors. This prevents them from vendor-lockin problems. Unfortunately, this also slows innovation since any new feature must be standardized and implemented by all vendors before being deployed.

We propose a paradigm shift that enables network operators to program the routing protocols used in their networks. We demonstrate the feasibility of this approach with xBGP. xBGP is a vendor neutral API that exposes the key data structures and functions of any BGP implementation. Each xBGP compliant implementation includes an eBPF virtual machine that executes the operator supplied programs. We extend FRRouting and Bird to support this new paradigm and demonstrate the flexibility of xBGP with four different use cases. Finally, we discuss how xBGP could affect future research on future routing protocols.

CCS CONCEPTS
• Networks → Network protocol design; Routing protocols; Programming interfaces; Programmable networks.

KEYWORDS
BGP; Routing; Network architecture; eBPF

ACM Reference Format:

*Quentin De Coninck is a F.R.S.-FNRS Postdoctoral researcher.

1 INTRODUCTION
Put yourself in the shoes of a mobile application developer who needs to support all mobile platforms. Your application would clearly benefit from being able to seamlessly switch from Wi-Fi to cellular without impacting the established TCP connections. Multipath TCP [14] supports such handovers, but you don’t have the luxury to wait until its adoption by all smartphone vendors1. Instead, you will likely resort to integrate a QUIC library in your software to benefit from its connection migration capabilities [23, 25].

Contrast this situation with the one of network operators whose networks could really benefit from new network-level features such as improved traffic engineering protocols [1], faster convergence mechanisms [36], or improved DDoS protection [34]. Like our programmer, network operators run highly heterogeneous environments, composed of a wide variety of devices types (routers, switches, middle boxes), coming from distinct vendors, and running distinct operating systems. This heterogeneity is actually necessary, not only to avoid vendor lock-in [9], but also for increased reliability (bugs or vulnerabilities tend not to affect all OSes at once).

Unlike our programmer, though, network operators often need the entire network to support the features (not only the endpoints) and, as such, have no alternative but to wait for the required features to be: (i) standardized by the IETF; (ii) implemented by all vendors; (iii) widely tested; and (iv) widely deployed in their network. The standardization process alone often takes years. As an illustration, Fig. 1 depicts the delay between the moment the IETF working group responsible for the BGP routing protocol (IDR) started to work on a new feature and its actual RFC publication. This delay includes the time required to document two independent and interoperable implementations as required by the IETF working group. We see that the median delay before RFC publication is 3.5 years, and that some features required up to ten years before being standardized. Even worse, this delay ignores the time elapsed between the initial idea and its first adoption by the working group.

Of course, this is not a new story. Frustrated by these delays and the difficulty to innovate in networks, researchers have argued for Software-Defined Networks (SDN) [30] for more than a decade. Instead of relying on a myriad of distributed protocols and features, SDN assumes that switches and routers expose their forwarding tables through a standardized API. This API is then used by logically centralized controllers to "program" routers and switches.
While SDN has enabled countless new research [12, 24], it has not been widely adopted. While many various factors can explain this, one of the main hurdles is that deploying SDN requires a complete network overhaul, both at the control-plane level, to deploy scalable and robust logically centralized controllers, and at the data-plane level, to deploy SDN-capable network devices. We believe SDN went “too far” to enable network programmability.

In this paper, we argue for a much lighter weight and practical approach to network programmability by allowing easy extension of the distributed routing protocols themselves. Our new approach, which we call xBGP, is inspired by the recent success of extended Berkeley Packet Filter (eBPF) in the Linux kernel. eBPF is an in-kernel Virtual Machine (VM) which supports a custom instruction set. Thanks to eBPF, programmers can easily (and securely) deploy new programs that can access a subset of the kernel functions and memory [17]. Similarly, in xBGP, distributed routing protocols expose an API and an in-protocol VM with a custom instruction set to access and modify the intrinsic protocol functions and memory. Thanks to this API and the VM, the same code can be executed on different implementations of a given routing protocol. Note that the instructions set and the in-protocol VM would still need to be adopted and implemented by each vendor, but this would be a one-off effort, instead of a per-feature effort.

Of course, opening up distributed routing protocols to external programs opens the door to many (research) questions: Which instructions set should we offer? How to implement this instruction set efficiently—so as to minimize the overhead or What about the correctness and the security of these extensions? We start to answer these questions in this paper in the context of BGP because this protocol plays a key role in ISP and data center networks, supporting both Internet access and value-added services. We make two main contributions.

First, we introduce our vision of xBGP that consists of three core elements. The first is the xBGP API. It defines a set of functions that expose the key features and data structures that are supported by any BGP implementation. The second is a set of insertion points. These identify the specific locations in a BGP implementation where user-provided bytecodes (xBGP programs) can be attached. The third is an eBPF Virtual Machine Manager that actually executes these bytecodes at runtime.

Second, we showcase the practicality of xBGP by adding support for it to two distinct BGP implementations (FRRouting [35] and BIRD [13]) and by implementing four use cases: (i) a reimplementation of route reflection; (ii) a new attribute that encodes geographical coordinates; (iii) route origin validation; and (iv) an extension that restricts paths inside data centers to be valley-free. Each use case involves the same xBGP bytecode running on both FRRouting and BIRD. We further show that the overhead of xBGP is within 20%, which is very reasonable given the flexibility benefits. In one use case, our extension was even faster than the native code thanks to a more optimized datastructure.

Similarly to what OpenFlow [30] achieved, we believe that programmable distributed routing protocols have the potential to open up many promising avenues for research, while being fundamentally more practical and deployable. We discuss several of these in Section 4.

2 XBG: OVERVIEW

We now provide an overview of xBGP and its prototype implementation 11bxbgp. In a nutshell, xBGP is a software layer that enables to expand a BGP implementation with user-supplied programs. Similarly to eBPF, xBGP programs are compiled to a specific instruction set, can call functions specified in the xBGP API and are attached to specific insertion points in the BGP implementation. Whenever these code points are reached, the corresponding xBGP programs (if any) are executed using the xBGP virtual machine.

API The xBGP API defines a set of functions enabling xBGP programs to access key BGP data structures. The xBGP API leverages the fact that all BGP implementations must adhere to the protocol specification [32] which defines, among others, the abstract data structures that each BGP implementation must maintain. We illustrate the three main data structures in Fig. 2 (in blue). Incoming BGP update messages first pass through an import filter before being installed in the Adj-RIB-In. Valid BGP routes are then stored in the Loc-RIB. This Loc-RIB is used by the BGP Decision process together with the router’s RIB to select the best path towards each prefix. The routes selected to be sent are stored in the Adj-RIB-Out. They are filtered by the export filters attached to each session before being advertised to a peer. The xBGP API exposes simple functions (e.g. get, set or write_buf) to access each of these structures.

Insertion points As their names indicate, the xBGP insertion points correspond to specific code locations in a BGP implementation from where xBGP programs can be called. These insertion points correspond to specific operations that are performed during the processing of BGP messages, enabling xBGP programs to modify the router’s behavior. These insertion points are represented using green circles in Fig. 2. Other insertion points might be defined to support other types of BGP extensions.

Virtual Machine An xBGP implementation includes user space eBPF virtual machines that are controlled by a manager. The virtual machine manager attaches bytecode with an associated virtual machine to one specific insertion point exposed by the host implementation. 11bxbgp provides several helper functions that ease the development of xBGP programs.

Example As an illustration, we consider how to expand a BGP implementation to support a new BGP attribute, GeoLoc, that stores the geographic location (i.e., longitude and latitude) of where the BGP route was learned. Among others, this attribute can be used to adapt router decisions, e.g., by filtering away routes that are more than x kilometers away. Supporting such an attribute has been discussed within the IETF but was not standardized [7]. Yet,

Figure 1: Delay between the publication of the first IETF draft and the published version of the last 40 BGP RFCs.
Figure 2: An xBGP compliant implementation exposes the abstract BGP data structures defined in RFC4271 through a generic API and uses libxbgp’s Virtual Machine Manager to attach the bytecode that implements extensions to specific insertion points (green circles). The four bytecodes in this example support a simple GeoLoc BGP attribute. For each bytecode, we provide the set of helper functions used to retrieve data from the host implementation.

several ISPs reportedly use iBGP filters [37] to achieve the same effect. Using iBGP filters is risky, though as doing so can lead to permanent oscillations [37].

Adding support for the GeoLoc attribute is easy with xBGP, and can be done with only four simple extensions. The first one adds GeoLoc attribute to BGP routes. It is attached to the BGP_RECEIVE_MESSAGE ① insertion point. It mainly uses three functions of the xBGP API. First, it uses peer_info to query the BGP neighbors table and determine the type of the eBGP session. Then, it retrieves the contents of the received BGP update in network byte order with get_arg. Finally, it attaches the new GeoLoc attribute using the add_attr function. The second bytecode is attached to the BGP_INBOUND_FILTER ② insertion point. This bytecode also uses peer_info, retrieves the router coordinates from the router configuration using the get_xtra function and manipulates the attributes with get_attr and set_attr. The bytecode attached to the BGP_OUTBOUND_FILTER ④ also retrieves the neighbor information and the attribute. Finally, the fourth bytecode is attached to the BGP_ENCODE_MESSAGE ⑤ insertion point. In addition to the xBGP API function already described, it uses write_buf to write the BGP GeoLoc attribute over an iBGP session.

2.1 xBGP meets actual BGP implementations

To demonstrate the feasibility of xBGP, we have made two different open-source implementations xBGP-compliant by integrating a generic libxbgp in both FRRouting [35] and BIRD [13].

Adding the xBGP API Implementing the API induced a total of 400 and 589 additional lines of code on BIRD and FRRouting, respectively. The difference between the two is due to the internal representation of the BGP data structures in memory. The xBGP functions that deal with BGP messages and attributes always manipulate them in network byte order (the neutral xBGP representation), performing the translation to the storage format used by the implementation if required. FR Routing uses an internal representation that is different from our neutral one. We thus had to implement several functions to do the conversion between the two representations. Another difference is the handling of BGP attributes. BIRD includes a flexible API to manage BGP attributes. xBGP simply extends this API. FRRouting does not include such an API, and we had to implement one to be able to manipulate BGP attributes in BGP updates.

Integrating libxbgp This library, implemented as 432 lines of header code, consists of two parts: (i) extension utilities; and (ii) the Virtual Machine Manager (VMM).

First, libxbgp provides generic utility functions. Aside from access to xBGP data structures, extension code may require additional persistent storage. For instance, an extension could need to keep track of the dedicated attributes it sends and receives to update its behavior. The xBGP API includes such memory allocation functions. An extension code has its own dedicated memory space and it cannot directly access the memory of other extension codes or the host implementation. This isolation is guaranteed by the eBPF virtual machine that we use. This ensures that orthogonal extensions will not interfere with each other. Extension code belonging to the same xBGP program can, however, share a dedicated persistent memory space available through helper functions defined by libxbgp. Other
The VMM is in charge of executing the right extension code according to the state of the host implementation. This layer acts as a multiplexer. To include xBGP operations, the BGP implementation calls the VMM to execute the associated extension codes.

Then, the VMM proceeds as follows. It first checks if there are attached extension bytecodes to the called xBGP operation. If not, the VMM executes the default function provided by the implementation. Otherwise, it runs the first extension code mentioned in the manifest. Two outcomes are possible. First, the extension code provides a result for the operation and the VMM returns the output to the caller. Second, the extension code delegates the outcome to another one by calling a special next() function. In that case, the VMM checks that there are remaining execution codes in the ordered queue. If there are, the VMM runs the next extension code in its virtual machine. Otherwise, the behavior of the xBGP operation falls back to the default function provided by the BGP implementation. For instance, two extensions can inject code to the BGP_RECEIVE_MESSAGE operation to process their own dedicated BGP attribute, calling next() once they are done. While running extension codes, the VMM also monitors their execution and stops them in case of error. In this case, it falls back to the default function and notifies the host implementation of the error.

Technical challenges While adding the xBGP API and integrating 11bxbgp, we also encountered some technical issues that were interesting. To successfully use the xBGP API, data must be available when the function is called. Some data in the host implementation were not available when the insertion point was called to execute the extension code. For example, in FRouting, export filters are applied on a set of peers sharing the same type of outbound policies. This set in not passed to the code checking the outbound policies but is required to implement the helper function get_peer_info. We wrote 5 extra lines of code to retrieve the set of peers before calling the insertion point. Also, some data structures are not flexible enough to fully support the xBGP API. For example, the function add_attr adds a new attribute to a BGP route. However, the internals of the host BGP implementation do not allow adding unsupported attributes that are not defined by any standard (e.g., ORIGINATOR_ID). We rewrote this part of the host implementation to support it. To address those issues, we had to add 30 and 10 lines of code to FRouting and BIRD respectively.

Each API function is called with a context of execution. This context is hidden within the extension code but visible in the host BGP implementation. This makes it possible to control which extension code has called the function. The context is also used to retrieve variables that cannot be directly used inside the extension code. For example, if an extension code needs to allocate extra memory (either ephemeral or not) the VMM can allocate the requested memory in the right memory space thanks to the context information, without requiring cooperation from the extension code. The ephemeral memory is also automatically freed when the extension code terminates its execution. Similarly, the context enables helper functions to access data structures which are out of the extension code’s scope. For instance, a dedicated helper function enables an extension to add a new route to the RIB. When setting an insertion point, the BGP implementation can pass a set of arguments. While some are visible inside the extension code, others are not. The BGP function leverages such hidden arguments to access the data structure while being transparent to the extension code.

Related Works The architecture of routing protocol implementations such as XORP was designed with extensibility in mind [19], but it does not expose a vendor-neutral API. We previously added eBPF to the FRouting implementation [40], but the proposed eBPF programs could only be used inside this implementation. Transport protocols researchers have proposed using extension codes to dynamically extend transport protocols like STP [31] or QUIC [10]. These proposals focused on extending a particular implementation of a protocol. xBGP goes one important step further by enabling very different implementations to execute the same xBGP program. In parallel to our work, CoreBGP [39] proposed to structure BGP implementations as a small core that implements the BGP Finite State Machine and a set of plugins that implement the entire BGP logic. It would be interesting to evaluate whether CoreBGP can support xBGP.

3 USE CASES

In addition to the GeoLoc attribute described in the previous section, we illustrate the benefits of xBGP by considering four different use cases implemented using extension code.

3.1 Filtering Routes Based on IGP Costs

Since the xBGP API provides access to the data structures maintained by a BGP implementation, network operators can leverage it to implement new filters. As a simple example, consider an ISP having worldwide presence that wants to announce to its peers the routes that it learned in the same continent as the advertising BGP router. This policy can be implemented by tagging routes with BGP communities on all ingress routers and then filtering them on export. While frequently used [11], this solution is imperfect. Consider an ISP having two transatlantic links terminated in London (UK) and Amsterdam (The Netherlands). This ISP has a strong presence in Europe and two links connect UK to other European countries. If these two links fail, packets between Germany and London will need to go through Amsterdam, the USA and then back in UK. When such a failure occurs, the ISP does not want to advertise the routes learned in UK to its European peers. With BGP communities, it would continue to advertise these routes after the failure.

Using the xBGP API, the operator could implement this policy as follows. First, configure the IGP cost of the transatlantic links at a high value, say 1000 to discourage their utilization. Second,
We now evaluate the performance penalty of using extension code VMs on the same laptop as depicted in Fig. 3. The listing below shows that on average over 15 runs, our extension code is less than 20% slower than native code on both FRRouting and BIRD.

```
uint64_t export_igp(bpf_full_args_t *args Unused) {
    struct ubpf_nexthop *nexthop = get_nexthop(NULL);
    struct ubpf_peer_info *peer = get_peer_info();
    if (peer->peer_type != EBGP_SESSION) {
        return FILTER_REJECT;
    }
    // next filter will decide to export route
    return FILTER_REJECT;
}
```

Listing 1: An export filter rejecting BGP routes having a too large IGP nexthop metric.

Figure 3: Experimental setup. Routers establish iBGP (Route Reflectors) or eBGP (Origin Validation) sessions on links L1 and L2 according to the tests shown in Fig. 4.

Figure 4: Performance impact of extension bytecode versus native code.

Figure 5: A simple data center.

implement a simple export filter that checks the IGP cost of the nexthop before announcing a route. The complete source code of such a filter is shown in Listing 1. It is attached to the BGP_OUTBOUND_FILTER insertion point. If the IGP cost to the BGP nexthop distance is acceptable, the filter calls the special function next(). This informs the VMM to execute the next bytecode attached to the insertion point. If the extension code is the last to be executed, the insertion point falls back to the native code. To reject the route, the extension code returns the special value FILTER_REJECT to the host implementation.

### 3.2 BGP Route Reflection

We now evaluate the performance penalty of using extension code versus native code. For this, we implement BGP Route Reflection [3], i.e., the support for the ORIGINATOR_ID and CLUSTER_LIST BGP attributes entirely as an extension code.

We use a simple network composed of three routers (upstream, Device Under Test (DUT) and downstream) running in different VMs on the same laptop as depicted in Fig. 3. The upstream and downstream routers use FRRouting v7.3.1. The DUT acts as a route reflector. The upstream router is first fed with IPv4 BGP routes from a recent RIPE RIS snapshot of June 2020. The upstream router sends all its routes over an iBGP session to the DUT that reflects them to the downstream router.

We measure the delay between the announcement of the first prefix by the upstream router and the reception of the last prefix of the BGP table on the downstream router. We then report the relative performance impact between the native implementation on both FRRouting and BIRD and our extension code. The blue boxplots in Figure 4 show that on average over 15 runs, our extension code is less than 20% slower than native code on both FRRouting and BIRD.

#### 3.3 BGP in data centers

Although BGP was designed as an interdomain routing protocol, it is now widely used as an intradomain routing protocol in data centers [26]. This is mainly because BGP scales better since it does not rely on flooding in contrast with OSPF or IS-IS. Another benefit of BGP is its ability to support a wide range of configuration knobs and policies. However, BGP suffers from several problems that forced datacenter operators to tweak their BGP configurations [26]. These tweaks make BGP configurations complex and more difficult to analyze and validate [4]. To illustrate this complexity, let us consider the data center shown in Fig. 5. Routers S1 and S2 are the Spine routers, L10...L13 the leaf routers and T20...T23 the top-of-the-rack routers. In such a data center, there is no direct connection between the routers at the same level of the hierarchy. Data center operators usually want to avoid paths that include a valley (e.g. L10 → S1 → L11 → S2). To achieve this, they run eBGP between routers, but configure the same AS number on S1 and S2 (even if these routers are not directly connected). Similarly, L10 and L11 (resp. L12 and L13) use the same AS number. With this configuration, when S2 receives a BGP update with an AS-Path through S1, it recognizes its AS number and rejects the route. This automatically blocks paths that include a valley and also helps to prevent path hunting.

Unfortunately, using the same AS number on separate routers can cause problems. First, operators can no longer look at the AS Paths to troubleshoot routing problems since different routers use the same AS number. Second, by prohibiting valley-free paths, the operator implicitly agrees to partition the network when multiple failures occur. Consider again Figure 5. If the links L10 – S1 and L13 – S2 fail, then the only possible path between L10 and L13 is
The interdomain routing system is regularly affected by disruptions caused by invalid BGP advertisements due to manual errors and other problems. Examples include the AS7007 incident in 1997, the announcement of a more specific prefix covering the YouTube DNS servers by Pakistan Telecom in 2008 or the BGP prefixes leaked by Google in 2017 that disrupted connectivity in parts of Asia. These problems and many similar ones were caused by configuration errors.

To cope with these (mainly manual) errors, network operators and the IETF developed three types of solutions. First, they enhanced the address registries to include cryptographically signed certificates that associate IP prefixes to origin ASes. This is the basis for the Routing Public Key Infrastructure (RPKI) [27]. Thanks to the RPKI, an operator can verify whether ASx is a valid originator for prefix p1. Second, the SIDR working group developed techniques to allow a router to query the RPKI to validate the origin of the routes that it receives [22]. The work on validating the origin of prefixes started in 1999 [29], the first RFC was adopted in 2012. It is slowly being deployed [8, 33]. This approach is now being extended to also use the RPKI to validate other elements such as the AS-Path [2]. The third long-term solution, which should also cope with malicious BGP hijacks, will be to extend BGP to carry digital signatures inside the BGP messages [28]. This extension is far from being deployed.

To evaluate the performance of our prefix origin validation, we use the same testbed as in Section 3.2 except that we use eBGP sessions for links L1 and L2. Our DUT does not implement the RPKI-Rtr protocol [6, 38] but loads a file that considers 75% of the injected prefixes as valid. For this test, our extension code checks the validity of the origin of each prefix but does not discard the invalid ones.

Figure 4 compares our extension codes running on BIRD and FRRouting to their native implementations. On BIRD, our prefix validation extension code provides similar performance as BIRD’s native code. Surprisingly, on FRRouting, our extension is 10% faster than the native code. A closer look at FRRouting’s source code revealed that it browses a dedicated trie for validated ROAs (Route Origin Authorization) each time a prefix needs to be checked. Our extension uses a hash table as in BIRD to retrieve the validated ROAs.

4 FUTURE RESEARCH DIRECTIONS

Although this paper focused on BGP, our methodology of defining a minimal vendor-neutral API that exposes the key functions and datastructures of any implementation could be applied to any routing protocol. Combined with eBPF, such an API would enable network operators to program their routing protocols by injecting xBGP programs that are executed by the eBPF virtual machine. This programmability could help network innovators innovate with existing distributed routing protocols as Software Defined Networking lead to the development of programmable switches.

From a high-level perspective, xBGP allows the classic separation of mechanism from policy [20], with the base BGP code dealing with transport, OS data structures, etc. and the xBGP extensions providing the policy. With xBGP, a BGP implementation becomes like a micro-kernel operating system that exposes a simple but powerful API that supports a variety of user supplied programs.

From a system’s viewpoint, there are different directions to extend xBGP. First, xBGP should be applicable to BGP implementations written in safer languages than C/C++ [15, 16, 21]. Second, eBPF should be compared with other Virtual Machines [18, 41] by considering performance but also isolation and safety constraints. The safety of the xBGP programs will be an important concern for network operators. It would be useful to develop automated verification techniques that allow to validate the safety of these xBGP programs.

From a deployment viewpoint, network operators will only benefit from xBGP once it has been integrated in several commercial BGP implementations. Our open-source libxbgp could be a starting point for this integration, but we expect that commercial vendors will first want an approved IETF specification of xBGP before committing to such an implementation effort. This could require some years of discussion. The same applies to other existing routing protocols.

From a research viewpoint, it would be much more interesting to design new protocols by assuming that they would expose a vendor-neutral API that allows extending them. This change of paradigm would force us to focus our efforts on designing a small set of truly extensible functions instead of adding new features on top of each other. The specification of such protocols would likely need to be different from today’s informal specifications. A fully formal specification of xBGP or a new extensible routing protocol could be a long-term goal.

In this paper, we have assumed that xBGP interacts with a fixed dataplane to retrieve routes from the RIB and push entries to the forwarding tables. As the dataplane becomes more and more programmable [5], it becomes possible to imagine a completely new network layer protocol implemented in P4. xBGP could then interact with the new dataplane by pushing P4 programs.

Software artefacts

To enable other researchers to reproduce and extend our work, we release all the source code of xBGP, our modified eBPF virtual machine and our modifications to BIRD and FRRouting on https://www.pluginized-protocols.org/libxbgp.
REFERENCES


