*x*BGP: Faster Innovation in Routing Protocols

Thomas Wirtgen[*] *ICTEAM, UCLouvain*
Tom Rousseaux *ICTEAM, UCLouvain*
Quentin De Coninck *ICTEAM, UCLouvain*
Nicolas Rybowski *ICTEAM, UCLouvain*
Randy Bush *Internet Initiative Japan & Arrcus, Inc*
Laurent Vanbever *NSG, ETH Zürich*
Axel Legay *ICTEAM, UCLouvain*
Olivier Bonaventure *ICTEAM, UCLouvain*

Abstract

Internet Service Providers use routers from multiple vendors that support standardized routing protocols. Network operators deploy new services by tuning these protocols. Unfortunately, while standardization is necessary for interoperability, this is a slow process. As a consequence, new features appear very slowly in routing protocols.

We propose a new implementation model for BGP, called *x*BGP, that enables ISPs to innovate by easily deploying BGP extensions in their multivendor network. We define a vendor-neutral *x*BGP API which can be supported by any BGP implementation and an eBPF Virtual Machine that allows executing extension code within these BGP implementations. We demonstrate the feasibility of our approach by extending both FRRouting and BIRD.

We demonstrate seven different use cases showing the benefits that network operators can obtain using *x*BGP programs. We propose a verification toolchain that enables operators to compile and verify the safety properties of *x*BGP programs before deploying them. Our testbed measurements show that the performance impact of *x*BGP is reasonable compared to native code.

1 Introduction

Internet Service Providers (ISP) are continuously challenged by their users and customers to provide value-added services that go beyond best-effort connectivity. Among others, these new services include traffic engineering techniques to prioritize some flows over others and improve network load, fast reroute mechanisms to swiftly retrieve connectivity upon failures, or anycast routing. In addition, ISPs are trying to improve their internal operations in order to provide an ever better service to their customers. This can be done by implementing a monitoring system, re-architecting or tuning the internal network.

*Thomas Wirtgen is supported by a grant from F.R.S.-FNRS FRIA.

Almost invariably deploying these services require extending routing protocols. And among all protocols, the Border Gateway Protocol (BGP) is probably the most used one given its flexibility: for many network operators, BGP has become a true "Swiss-army knife". Originally designed to distribute interdomain routes, BGP has been extended several times to support different types of services [41, 55].

While extending BGP is possible, it is certainly not easy, for two main reasons. First, ISP networks often include routers from different vendors [17, 69]. This diversity is inherent and required for technical, safety, and economic reasons. Unfortunately, this diversity means that operators can only use the *intersection* of the features set across all their routers, hindering flexibility.

Second, it can take years for even a subset of the vendors to implement new features as these need to be first standardized by the Internet Engineering Task Force (IETF). Many view this as a form of ossification of the routing protocols. As an illustration, a recent paper [79] showed that the median delay before RFC publication of BGP extensions is *3.5 years*, and that some features required *up to ten years* before being standardized.[1] This is only the tip of the iceberg though: only a small subset of the BGP extensions proposed by network operators have been discussed and later adopted by the IETF.

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) [48] 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 works [21, 42], it has not been widely adopted by ISPs. One of the main hurdles is that deploying SDN requires a major network overhaul, both at the control-plane level, to deploy scalable and robust logically-centralized controllers, *and* at the

[1] Note that this delay ignores the time elapsed between the initial idea and its first adoption by the working group, making the actual delay even longer.

data-plane level, to deploy compatible network devices. Thus far, only large cloud providers managed to perform this overhaul [36, 39].

Of course, instead of relying on commercial routers, network operators could decide to adopt open-source implementations of routing protocols [16, 23, 33, 67] running on servers or custom hardware [3]. A network operator could for instance fork a BGP implementation to add a desired feature. Maintaining this fork requires a lot of software development effort though. Such an approach is feasible for large cloud providers [62] but not for ISPs. Another approach is to use a modular routing implementation to take full control of the protocol. The network operator is responsible for the entire routing implementation. Unfortunately, it is too difficult to maintain and evolve because the network operator must have a complete understanding of the routing protocol and must have software programming skills, which they often do not have. To provide flexibility in the administration and automation of their routers, router vendors have added a Python interpreter to their operating systems [40]. However, the interpreter only handles the administration part of the router and does not provide an interface to add or modify protocol features. Finally, the use of active networks with centralized approaches or descriptive configuration languages [10, 27] is not possible in today's Internet, as autonomous systems still use decentralized protocols to establish peering links.

In this paper, we argue for much lighter weight and practical approach to network control plane programmability by *allowing the network operators to easily extend the distributed routing protocols that they already use*. Our new approach, which we call *x*BGP, is inspired by the success of the extended Berkeley Packet Filter (eBPF) in Linux [26, 35] and Windows [49]. eBPF is an in-kernel Virtual Machine (VM) that relies on 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 [26]. Similarly, in *x*BGP, different BGP implementations 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. Note that the instructions set and the in-protocol VM still need to be adopted and implemented by each vendor, but this is a one-time effort, instead of a *per-feature* effort.

Naturally, opening up BGP implementations to external programs opens the door to many (research) questions: *Which API should BGP expose? How to implement this API efficiently* or *What about the correctness and the safety of these extensions?* We answer these questions in this paper and make four main contributions.

First, we introduce the x*BGP API* which defines a set of functions that should be supported by an extensible BGP implementation. We present this API in Section 2 and describe how we modified two different BGP implementations, BIRD and FRRouting, to support *x*BGP.

Second, we present a complete validation workflow that enables operators to validate that their extensions correctly terminate, do not interfere with the memory of the host implementation, produce syntactically valid BGP messages, or only use the *x*BGP API functions authorized by the network operator. We envision this workflow to become one element of the qualification tests that operators already carry out before deploying any new BGP feature in their network.

Third, we showcase the practicality of *x*BGP by implementing eleven use cases with *x*BGP to: support a new BGP attribute; introduce new selection rules; restrict the set of paths it can compute; detect unused routes (zombies); or monitor BGP operations. Each use case involves the same *x*BGP bytecode running on both FRRouting and BIRD.

Fourth, we demonstrate the practicality of *x*BGP by measuring its overhead compared to native implementations. Even for complex extensions (re-implementing BGP Route Reflection), our benchmarks show that the overhead of *x*BGP is always under 13%, a reasonable value given the flexibility benefits.

Similarly to what OpenFlow [48] 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.

2 Architecture

At a high level, *x*BGP enables network operators to customize or extend any compatible BGP implementation by injecting and directly executing *x*BGP programs. 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 each BGP route was learned. Among others, this attribute can be used to adapt router decisions, e.g., by filtering away routes learned more than x kilometers away. Supporting such an attribute has been discussed within the IETF but never standardized [13]. Yet, large-scale ISPs reportedly use iBGP filters [71] to achieve the same effect. Using iBGP filters is risky though as doing so can lead to permanent oscillations [71].

To implement the *GeoLoc* extension, we need to support several operations in a BGP implementation. (1) When a route is received over an eBGP session, the router adds a new attribute, `Geo_Originator` that contains the geographic coordinates of the router that learns the BGP route in an import filter. (2) If the BGP route already contains the `Geo_Originator` attribute, the router needs to decode it. (3) When exporting the route to another peer, the router can use the `Geo_Originator` attribute to filter routes that are too far away. (4) To be usable by other iBGP peers, the attribute needs to be added to the BGP Update message.

To add this extension, we need to understand how BGP implementations are designed. There are many ways to organize

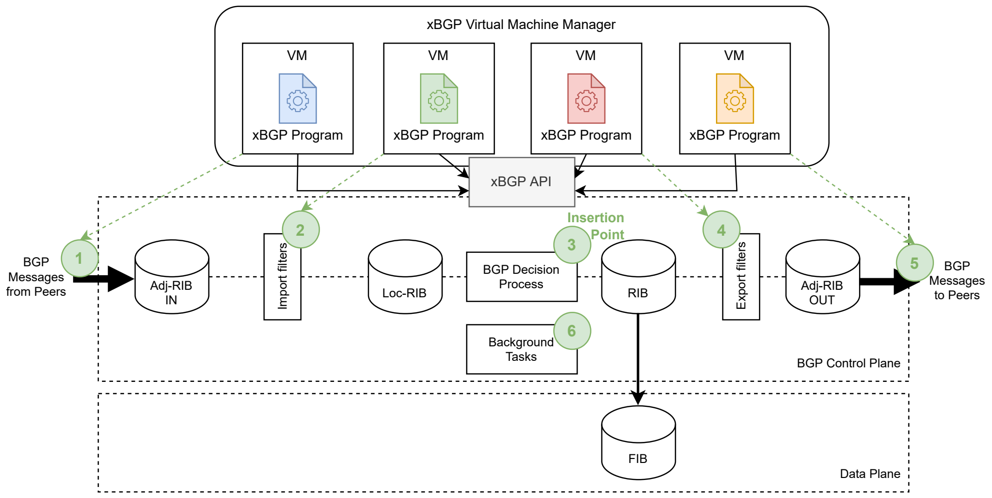

Figure 1: An *x*BGP compliant implementation exposes the abstract BGP data structures defined in RFC4271 through a generic API and uses `libxbgp`'s Virtual Machine Manager to attach the bytecode that implements extensions to specific insertion points (green circles). The four bytecodes in this example support a simple GeoLoc BGP attribute. For each *x*BGP program, we provide the set of helper functions used to retrieve data from the host implementation.

a BGP implementation. Each implementer selects a particular software architecture and the associated data structures based on their own requirements. However, all BGP implementations must adhere to the protocol specification [54]. This specification defines the format of the BGP messages, an abstract BGP Finite State Machine that manages each BGP session, and also an abstract workflow and data structures that describe how BGP update and withdrawal messages should be processed. This workflow is illustrated in black in Figure 1. Starting from the left, a received BGP message is stored in the `Adj-RIB-in` [2]. It then passes through the `import` filters that may decide to discard the message or modify attributes such as `local-pref`. If the route is accepted by the import filters, it is inserted in the `Loc-RIB`. The `Loc-RIB` contains all the BGP routes accepted by the router. The BGP decision process extracts from the `Loc-RIB` the best routes that are placed in the `RIB`. These routes then pass through the `export` filters before being advertised over BGP sessions.

Going back to our *GeoLoc* extension, we can see that it can be added to the different parts of the BGP workflow. (1) needs to be added to the part that parses a BGP attribute. (2) and (3) must be designed as import and export filters respectively. And (4) will be added to the serialization part of the BGP implementation. The question now is how to add those subcomponents to the main BGP implementation. To answer this, we defined the insertion points depicted in Figure 1 with the green circles on which functionalities can be added or modified. These insertion points correspond to the major BGP events. It is now easy to add the four components of our simple extension to the BGP implementation in their respective insertion points. (1) is attached to the `BGP_RECEIVE_MESSAGE` ① insertion point. First, it queries the BGP neighbor's table and determines the type of the eBGP session. Then, it retrieves the contents of the received BGP update in network byte order. Finally, it attaches the new GeoLoc attribute to the route. The second program (2) is attached to the `BGP_INBOUND_FILTER` ② insertion point. It retrieves the router coordinates from the router configuration to add them to the attributes of the route. The program (3) attached to the `BGP_OUTBOUND_FILTER` ④ retrieves the neighbor information and the *GeoLoc* attribute to check if the route can be advertised to the peer. Finally, the fourth program (4) is attached to the `BGP_ENCODE_MESSAGE` ⑤ insertion point. It uses the BGP GeoLoc attribute received over an iBGP session decoded by the first program and sends it to the peer.

To be able to dynamically augment the BGP implementation, the four *x*BGP programs are executed inside a Virtual Machine and are attached to specific *insertion points* in the BGP implementation. An *x*BGP program is composed of eBPF bytecode executed by a user space virtual machine that is included in any *x*BGP compliant implementation. Thanks to this eBPF virtual machine, the same *x*BGP program can be executed on the CPUs used by different router platforms.

An *x*BGP program is not a standalone executable that performs computations autonomously. It can interact with the underlying BGP implementation, access its data structures, and call some of its functions. In contrast with operating system kernels such as Linux, FreeBSD or macOS that expose a similar POSIX interface, there is no standard API for BGP implementations. *x*BGP must then propose a common API to support several BGP implementations. If we take our ex-

[2]Some implementations do not explicitly maintain a separate `Adj-RIB-{in,out}` to reduce their memory consumption and store everything in the `Loc-RIB`. We ignore this implementation detail in this paper.

tension, when the *GeoLoc* program has finished decoding the `Geo_Originator` attribute, it must update the BGP route stored in the BGP implementation. Thus, our extension needs to fetch or set data from the host implementation. For this, the BGP implementation must propose a set of functions, the *x*BGP API [75], that enable the interactions between the extension and the internal data structures. For example, with a call to the function `set_attr`, the extension can add a new attribute to the BGP route being processed.

An important data structure of a BGP implementation is its Routing Information Base (RIB). It contains the routes selected by the BGP decision process and pushed in the Forwarding Information Base (FIB). The BGP RIB stores, for each known destination prefix, its BGP route containing its BGP attributes, including its AS-path and the address of the BGP next hop. The RIB also contains information from the intradomain routing protocol such as the cost to reach each next hop. BGP implementations use various data structures to store their RIB. Some implementations simply store the BGP attributes as they were received from the wire. Others use a specific structure for each type of attribute. To ensure that the same *x*BGP program can be executed on any compliant implementation, *x*BGP defines its own representation for IP prefixes, next hops, and BGP attributes. For the latter, *x*BGP simply relies on the wire format [54]. *x*BGP also defines a neutral representation of the BGP neighbor's table. With these representations, *x*BGP programs can access the data structures of the underlying BGP implementation. When required, *x*BGP converts the internal representation to its own format before returning data to *x*BGP programs and vice versa.

The remaining of this section describes the composition of the *x*BGP API enabling *x*BGP programs to interact with BGP implementations in Section 2.1. Section 2.2 shows how we execute an *x*BGP program inside the BGP implementation. We explain in Section 2.3 which challenges we faced to make two BGP implementations, BIRD and FRRouting, *x*BGP compatible.

2.1 The *x*BGP API

Besides some utility functions (memory management, conversion between network and host byte orders, simple math functions, etc.), most of the *x*BGP API is specific to BGP [75].

To modify internal BGP data structures, *x*BGP programs rely on `getters` and `setters` to access data structures stored on the host implementation. This ensures (i) an isolation layer between the host and the *x*BGP program and (ii) a uniform method of accessing data regardless of the BGP implementation. These functions convert the internal representation into a universal one understood by *x*BGP programs. In addition, extension codes require access to the BGP internal state (e.g., list of peers, the route attributes, the route next hop). Hence, *x*BGP requires BGP implementations to provide routines translating their internal data structures into *x*BGP ones. These include `getters` and `setters` to access/modify a BGP route including its attributes, next hop and the data that identifies a BGP peer. We also provide functions to iterate the RIB. These enable searching for a route other than those provided by the insertion points, and therefore for searching routes already installed in the BGP routing table.

Existing router OSes do not provide a common way to access internal routing data. The *x*BGP API provides functions to access IGP data, e.g., to retrieve the next hop for routes and use them in use cases described in Section 5.

An *x*BGP program can deliberately send a custom BGP message to any peer it wants. Instead of relying on an insertion point to generate the message, the *x*BGP API contains functions to send BGP messages allowing a program to send an urgent message like a BGP notification because the *x*BGP program detected a problem with a given peer.

To access non-standard data such as the geographic coordinates of the router, an extension code may require additional configuration. One approach is to directly include the data inside the code of the *x*BGP program. However, this is not scalable if the operator wants to deploy it on a large number of routers. This induces a recompilation of the code for each of its router. Instead, the *x*BGP API proposes to the network operator to include a configuration data part in a structured textual file accompanying *x*BGP programs called *manifest*. The *x*BGP program uses it later to retrieve what it needs. This extra configuration part is not directly accessible to the *x*BGP program but can be accessed through a set of API functions.

Finally, *x*BGP programs can be executed as background tasks ⑥ that are called when a timer expires. These tasks are not triggered by a specific BGP event like an insertion point but are rather executed when a timer expires. Background tasks are only used for processes that do not interact with the BGP workflow. Each task controls its timer and *x*BGP deliberately restricts one timer per task to avoid timer explosions. However, the task may ask to queue forever as long as the BGP router is alive. This is particularly interesting for *x*BGP programs that make routine maintenance for example. Each background task is executed in a dedicated thread to allow the original BGP implementation to run in parallel. If the *x*BGP program must access or update data, the *x*BGP API must be thread safe. This constraint must be respected when implementing the *x*BGP API.

2.2 Executing *x*BGP programs

An *x*BGP program is a set of eBPF bytecodes, either attached to different insertion points or executing background tasks. Each *x*BGP bytecode has its own dedicated memory, including a stack and a heap that are automatically freed after execution. This memory isolation between extension codes is guaranteed by the eBPF virtual machine. This ensures that orthogonal extensions will not interfere with each other. Yet, *x*BGP programs may need to keep persistent storage or to exchange data between the different bytecodes that compose a program. For this, the *x*BGP API provides a key-value store that is similar to the BPF maps used in the Linux kernel.

Each *x*BGP implementation includes userspace eBPF virtual machines that are controlled by a manager. The *Virtual Machine Manager* (VMM) attaches bytecode with an associated virtual machine to one specific insertion point exposed by the host implementation. Each *x*BGP program includes a manifest listing the extension codes and their insertion point. Different extension codes can be attached to the same insertion point, and the manifest defines in which order they are executed. The manifest also lists the different *x*BGP API functions that the bytecode may use.

An *x*BGP program can be attached at different insertion points, i.e., specific code locations in a BGP implementation from where the program can be called. These insertion points correspond to specific operations that are performed during the processing of BGP messages, enabling *x*BGP programs to modify the router's behavior. *x*BGP defines six generic insertion points (green circles in Fig. 1) based on the original definition of BGP [54]. The sixth insertion point is dedicated for background tasks.

By default, the VMM only runs one *x*BGP program per insertion point. *x*BGP programs must explicitly tell the host implementation to run the next *x*BGP program if any through the `next()` function. This mechanism avoids executing useless code. For example, if we attach two *x*BGP programs that parse different BGP attributes into the insertion point that processes a single BGP attribute, and the first program successfully parses the message, there is no need to run the second one.

2.3 Adding *x*BGP to BGP implementations

To demonstrate the feasibility of *x*BGP, we have adapted two open-source implementations: BIRD v2.0.7 [16] and FRRouting v7.3 [23].

Adding the *x*BGP API. Implementing the API induced a total of 400 and 589 additional lines of code [78] on BIRD and FRRouting, respectively. The difference between both is their internal representations of the BGP data structures. The *x*BGP functions that deal with BGP messages and attributes always manipulate them in network byte order (*x*BGP's neutral representation), performing the translation to the storage format used by the implementation if required. FRRouting 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. *x*BGP simply extends this API. FRRouting does not include such an API, so we had to implement one to be able to manipulate BGP attributes in BGP updates.

Integrating `libxbgp`. `libxbgp` is a portable library, implemented as 432 lines of header code, which consists of two parts: (*i*) the utility functions of the *x*BGP API; and (*ii*) the VMM. 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 *x*BGP 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 *x*BGP 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 the special `next()` function. In that case, the VMM checks whether there are remaining codes in the ordered queue. If there are, the VMM runs the next extension code in its virtual machine. Otherwise, the behavior of the *x*BGP operation falls back to the default function provided by the BGP implementation. For instance, two extensions can attach bytecode to the `BGP_RECEIVE_MESSAGE` operation that processes their own dedicated BGP attribute, calling `next()` once they are done.

Technical challenges. While adding the *x*BGP API and integrating `libxbgp`, we encountered some interesting technical issues. To successfully use the *x*BGP API, data must be available when the function is called. In some cases, data in the host implementation was not available when the insertion point was called to execute the extension code. For example, in FRRouting, export filters are applied to a set of peers sharing the same type of outbound policies. This set is not passed to the code checking the outbound policies but is required to implement the helper function that retrieves data about the BGP peers of the router. We had to write 5 extra lines of code to get the set of peers before calling the insertion point. Also, some data structures were not flexible enough to fully support the *x*BGP API such as the function that adds or modifies a new attribute to a BGP route. However, the internals of FRRouting do not allow adding unsupported attributes that are not defined by any standard (e.g., `ORIGINATOR_ID`). We rewrote this part of FRRouting. To address those issues, we had to add 30 and 10 lines of code to FRRouting and BIRD respectively.

Each API function is called within 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 (ephemeral or not), the ephemeral memory is also automatically freed when the extension code terminates its execution. Similarly, the context enables helper functions to access data structures that 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 RIB function leverages such hidden arguments to access the data structure while being transparent to the extension code.

Limitation of *x*BGP. To better understand what can and cannot be done with *x*BGP, we analyzed the complete list of RFCs, which defines extensions to BGP, that have been published since the publication of RFC4271 [54]. The RFCs can be classified into two different types. (1) The RFCs that modify the original definition of RFC 4271 (6 RFCs) and (2) those that add features on top of BGP (30 RFCs). For (1), *x*BGP cannot be used to implement these types of RFCs because it requires a direct modification of the underlying BGP implementation. For example, increasing the internal buffer size of the BGP message size [9] is not feasible with *x*BGP. For (2), *x*BGP can be used. However, it turns out that our current prototype focuses only on the messages that BGP speakers exchange once the session is established (BGP Updates and BGP Withdraw). Not all session-level extensions to BGP can be handled by *x*BGP. For example, our current prototype cannot extend the BGP Open or BGP Route-Refresh [12] message. However, *x*BGP contains a generic insertion point, `DECODE_BGP_MSG`, that can handle future types of BGP messages. If the underlying BGP implementation does not support route refresh, we can implement it as an *x*BGP program. Modifying *x*BGP to allow it to support the session level could be implemented at a later time, but *x*BGP cannot change the architectural design of the underlying implementation. This is an important limitation of our solution. For example, no *x*BGP program can increase the size of the BGP transmit and receive buffers as defined in the corresponding RFC [9]. The internal structure of an implementation cannot be modified on the fly by a program since the definition of the structures is strongly integrated in the program binaries.

In addition to the limitation of the features that can be implemented, *x*BGP focuses mainly on the internal network, we assume that the network operator enables the necessary *x*BGP programs on the relevant routers. However, if the BGP routers decode an unknown message, it will be silently discarded and will not harm the router but will compromise the other router's computation. BGP capability negotiation messages can be exchanged to indicate whether the extension implemented by the *x*BGP program is supported by both routers implied in the BGP session. Capability support is beyond the scope of this paper.

3 Ensuring the safety of *x*BGP programs

BGP implementations generally run 24/7 and never stop. When operators deploy a new router or a new version of a router operating system, they typically run extensive tests to verify that the new feature will not break their network. From an operator's viewpoint, injecting an *x*BGP program is always risky since the program will be executed within the BGP implementation. A simple approach would consist in letting the VMM monitor their execution and stop them in case of error. This could be too late for errors that could disrupt BGP sessions. Network operators typically need some safety guarantees from the *x*BGP program. The Linux kernel copes with a similar problem by using a custom online verifier [1] that checks different aspects of eBPF programs *before* they are injected into the kernel.

Verifying *x*BGP programs. *x*BGP also relies on verification techniques to ensure that programs can be safely injected. However, instead of developing a custom verifier [24], we (*i*) establish a list of properties that an *x*BGP program should respect to be considered as safe and (*ii*) we build a toolchain embedding three existing and well-tested software verification tools allowing the verification of our properties. Our *x*BGP toolchain receives the *x*BGP programs as input. They consist of C code that uses the *x*BGP API and a manifest provided by the network operator containing the configuration data. This code is by nature untrusted and must be manually augmented with various annotations providing hints to the code verifiers, given the specificities of each one. Such annotated extensions can then enter the *x*BGP toolchain which executes in parallel each verifier. The bytecode is produced only if the code passes all of them. Once the bytecode is generated, it is added to the integrated *x*BGP store. A network operator can safely select and load *x*BGP programs coming from this store. We expect that initially each ISP will have its own store. Later, third parties or router vendors could also develop their own stores. We consider this toolchain as trusted, i.e., we select a particular compiler, `clang`, and specific verifiers, all considered as correct. Therefore, we do not need to reason about the produced bytecode and ignore problems such as handling maliciously formatted bytecode.

Embedded verification tools. The whole *x*BGP toolchain, illustrated in Figure 2, is designed to prevent four types of problems that a program can cause. First, if an *x*BGP program enters an infinite loop, it will block the underlying BGP implementation. We use the Terminator 2 (T2) automated termination checker [15] to verify the termination of *x*BGP programs.

The second set of possible problems is the way *x*BGP programs interact with the memory of the underlying BGP implementation. We use CBMC [43] and SeaHorn [31] to verify memory-related properties.

The third type of problem is related to *x*BGP and BGP themselves. *x*BGP programs can create new BGP attributes or messages that are sent over a BGP session. We use SeaHorn to verify that the BGP messages emitted by *x*BGP programs are fully compliant with the BGP RFCs and that their return values respect the *x*BGP requirements.

Finally, operators may want to be able to impose restric-

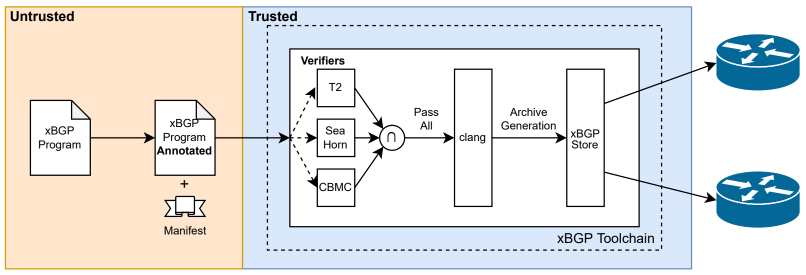

Figure 2: High-level view of the *x*BGP verification toolchain.

Property	Verifier	Type
Termination	T2	Safety
Reads/Writes within *x*BGP program's memory space	CBMC	Safety
No buffer overflow, use after free memory, invalid read, etc.	CBMC	Safety
All strings must be null terminated	SeaHorn	Safety
Correct size/buffer combination	SeaHorn	Safety
RFC-compliant syntax of BGP attributes	SeaHorn	BGP
Valid return value	SeaHorn	Safety
Checking attribute reads/writes	SeaHorn	BGP
Checking API function accesses	`libxbgp`	Safety + BGP
Call the `next()` function to trigger the next *x*BGP program	SeaHorn	Safety

Table 1: Properties that *x*BGP programs must satisfy.

tions on the *x*BGP functions and data structures that a given *x*BGP program can use. For example, a customer filter should only be able to set a `local-pref` value in a chosen range and to change nothing else. So it could never add a new BGP attribute to a route it filters. These restrictions are enforced with (*i*) SeaHorn that checks the validity of the arguments of the API functions and (*ii*) `libxbgp` which restricts the available API functions at loading time.

To be considered valid, any *x*BGP program must satisfy the properties listed in Table 1. If every *x*BGP bytecode satisfies this list, the router is guaranteed (*i*) not to crash and (*ii*) to still follow the definition of the protocol. These properties ensure the local stability of each router. Ensuring the global stability of BGP [28–30, 46] is a problem that goes beyond the scope of this paper.

Verification macros. Because of their diversity, the verification tools do not offer a common way to annotate programs. In the case of *x*BGP, this would mean annotating the plugin 3 times with different annotations and running the 3 tools manually. For a network operator, manually using several tools can be a long, tedious, and error-prone process. To ease the annotation process, we define a set of multipurpose macros `PROOF_INSTS_*()` abstracting the annotation syntax of the verifiers. Those are only expanded if the corresponding verifier is invoked. When the extension programs are compiled

```
buf[0] = attribute->flags;
buf[1] = attribute->code;
buf[2] = attribute->length;
buf[3] = attribute->data;

CHECK_ORIGIN(buf);
```

(a) Annotated Code.

```
assert(buf[0] == ATTR_TRANSITIVE);
assert(buf[1] == ORIGIN_ATTR_ID);
assert(buf[2] == 1);
assert(((buf)[3] == 0 || \
        (buf)[3] == 1 || \
        (buf)[3] == 2)));
```

(b) Expanded Code (verifier).

Figure 3: Example of a verification macro that checks the origin attribute of a BGP route. The macro can be extended or not according to its use. (a) is the original source code and (b) is the code viewed by a verifier.

for routers, the annotations are not expanded and thus will not interfere with the normal BGP execution. Figure 3 shows an example of such verification macro.

Aside from the verifier syntax abstraction, we mainly bring two contributions. First, we define a set of macros helping network operators to verify the properties listed in Table 1. Network operators can use them to annotate their xBGP programs. The macros are translated to their corresponding annotation to the right software verifier. For example, a network operator can use the `BUF_CHECK_*` macros to verify if the BGP attributes sent to a BGP peer are formatted as stated in the RFCs.

Second, we set up a verification toolchain that automatically performs verification on the xBGP programs. It automatically and transparently calls all the verification tools and verifies the annotations contained in the source code of the programs. If all properties are satisfied, the system stores the verified plugins in a "plugin store", which the programmer or network operator can use to inject into their routers. The routers will only accept plugins that have been verified and signed by the plugin store.

Those macros, in conjunction with our verification toolchain, allow a complete abstraction of the verification process. This makes the usage of *x*BGP simpler for network operators. The entire set of verification macros is defined in Appendix B.

3.1 Proving *x*BGP Programs' Termination

T2 (TERMINATOR 2) is a program analysis tool for termination [15] and temporal property [7] verification. We were successfully able to prove the termination of every *x*BGP program that implements the use cases defined in this paper. Table 3 reports the total time taken by the verification toolchain to verify all the properties defined for the *x*BGP programs, including the termination checks. However, to check the termination we had to slightly modify the source code since some specific features of the C language were not supported by the prover. First, when using fixed-width integer types (e.g., `uint8_t`), T2 was not able to generate the proof of termination. We had to convert those types to their primitive type. Second, all the loops of the program must be explicitly bounded. For example, if the *x*BGP program needs to parse a BGP attribute, we must explicitly bound it to 4096 iterations, the maximum size of a BGP message [54]. Third, T2 does not handle bit shift operations. To solve this issue, we encapsulated the bit shift computation in a non-deterministic function. This is a function that is not defined in the source code of the *x*BGP program but simply tells T2 that it returns an arbitrary integer value that T2 can handle. Such non-deterministic function is also considered to terminate by T2.

3.2 Preventing Memory and C Errors

C is a permissive language and programmers can easily make mistakes in their programs while handling memory. A bug in an *x*BGP program that causes a buffer overflow or leads to using freed memory could crash the underlying BGP implementation. An earlier prototype automatically instrumented the eBPF code to verify these properties online [79]. The verification was made at runtime by adding memory check instructions to the *x*BGP bytecode. However, this had a performance impact. Our new *x*BGP toolchain uses the CBMC bounded model checker [43] to verify the absence of simple memory-related issues and SeaHorn [31] to detect more complex issues. That being said, the online verification of the memory bounds remains in *x*BGP and the operator can enable it or not at *x*BGP bytecode load time.

CBMC is a C Bounded Model Checker that uses loop unwinding methods. It requires that loops are strictly bounded [43] which implies that the code of *x*BGP programs must be adapted. It automatically annotates code, generates a formula and proves it *via* an integrated SAT solver. It can spot common C programming errors [14]. However, more complex properties cannot be checked automatically. For example, *x*BGP programs can log data to syslog. The functions used for that take a string as arguments. Nevertheless, C strings are not safe by design. We must ensure that each string is correctly null-terminated to prevent buffer overflows. Another example is the alteration of BGP attributes. An *x*BGP program needs to call an API function that takes as arguments a pair `<buffer, length>`. Those two values must correlate: if the actual buffer length is shorter than the announced length, the host implementation is vulnerable to a buffer overflow. We use SeaHorn [31] to prove properties written directly in the code as assertions.

We provide a set of C macros that operators call in the *x*BGP programs they verify. The first one is verified by searching for a null byte within the string. For the second one, we verify before each API call if the length passed to the function matches the buffer length. This is achieved by inserting custom annotations in the *x*BGP programs and passing them to SeaHorn.

3.3 Ensuring BGP and *x*BGP Compliance

*x*BGP programs can (i) send new BGP messages or (ii) modify the internal representation of BGP routes. If such a program sends a message deviating from the standardized BGP syntax [54], it could disrupt BGP sessions and have a huge impact [47]. For (i), we verify that the syntax of the BGP message generated by an *x*BGP program conforms to the BGP RFCs. For (ii), we check that the modification is correctly formatted. A corrupted BGP route accessed outside the *x*BGP program could result in a crash of the *x*BGP implementation. For this, the code is annotated with assertions representing BGP invariants that are checked by SeaHorn. We also created a set of C macros verifying that standard attributes comply with their definitions (correct flags, size, etc.). For non-standard attributes, we check that they respect a TLV format. For BGP messages, we check that the buffer containing the message conforms to the BGP syntax [54].

In addition, *x*BGP programs also have to comply to *x*BGP requirements. Some insertion points require a "communication channel" with `libxbgp` to change the behavior of the host BGP implementation. This is achieved by using the return values of the executed bytecode. Therefore, bytecodes cannot deviate from predefined values. For example, an *x*BGP filter returns a specific value to tell the host to reject the current route. This property is verified with SeaHorn by considering the *x*BGP program as a function called inside a "fake" `main`. The return value is then retrieved and verified using a custom assertion.

3.4 Enforcing Operator-Imposed Restrictions

Thanks to the manifest, the operator can list the *x*BGP API functions and the data structures that each *x*BGP program can use. Imagine a filter that only checks the validity of the route without modifying any data related to this route. To decrease the risk of introducing bugs in *x*BGP programs, the operator can restrict the set of API functions the program can call. In this example, the filter should have a read-only view, and thus should not call any function altering BGP data structures.

To settle this, we implemented a permission manager inside `libxbgp` that verifies, at load time, the functions that a given *x*BGP program calls according to its manifest. Just before being loaded, `libxbgp` checks the *x*BGP bytecode to look for unauthorized API function calls.

Network operators use BGP communities [20, 64] to enable their customer to activate specific features such as setting `local-pref`, AS-path prepending, or selective advertisements on a per-route basis. With *x*BGP they could provide even more advanced services. Imagine you are a network provider that proposes to attach filters developed by its clients to their eBGP sessions. You define a set of BGP attributes the client can modify such as MED, `local-pref`, etc. When they use communities, operators establish policies on the attributes which can be modified in a BGP route. For example, they define ranges of possible values for the `local-pref` attribute [20]. To modify an attribute for a route, an *x*BGP program calls the `set_attr` API function. When the *x*BGP toolchain processes such a program, SeaHorn verifies if the arguments of the API functions respect the policies defined in the manifest, i.e., if both the argument to change and its new value are legitimate. This is done by adding assertions in the source code of the *x*BGP program supplied by the customer.

4 Overhead of the current *x*BGP prototype

Using *x*BGP in BGP implementations brings flexibility for a network operator since they can use a simple abstraction to program their router. However, this flexibility has a price in terms of performance. To evaluate the overhead of `libxbgp`, we consider three different features that are already implemented in both native FRRouting and Bird to have a fair comparison with *x*BGP. The first is a simple filter adding an arbitrary MED value to all exported routes. The second provides support for extended communities [59]. The third is a complete implementation of Route Reflection [4]. While we expect operators to mostly develop simple plugins such as the first two, the Route Reflection extension demonstrates the of flexibility *x*BGP by covering the whole BGP workflow described in Section 2. Furthermore, since Route Reflection is supported by both FRRouting and BIRD, this extension enables us to compare the overhead of an *x*BGP implementation with native ones.

To evaluate the performance impact of *x*BGP, we use the simple network described in Figure 4. We measure the delay between the first BGP update sent by the Upstream router and the last update received by the Downstream one. This reflects the time needed for the Device under Test (DuT) router to process the routes sent by the Upstream router. The Upstream and Downstream routers are running an unmodified implementation of BIRD v2.0.8 while the DuT router is running the *x*BGP version of BIRD or FRRouting according to the test. The DuT router is running an Intel® Xeon® X3440 @2.53GHz with 16 GB of RAM, Linux kernel v5.15.29 and Debian 11.

The Upstream router sends a full routing table from a recent RIPE RIS snapshot (June 3, 2021, at 4:15 PM) containing 873k IPv4 routes and 120k IPv6 routes. We consider multiple executions of the BGP daemon located in the DuT router.

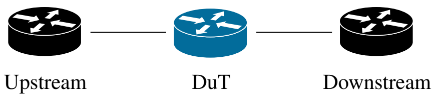

Figure 4: Simple network used for *x*BGP evaluations.

Use Case	Processing Time	
	xFRR	**xBIRD**
No *x*BGP program	+1.05%	+1.6%
Filter Set MED	+6.67%	+2.59%
Extended Communities	+5.93%	-0.67%
Route Reflection	+12.97%	+7.43%

Table 2: Performance impact of running *x*BGP programs to *x*BIRD and *x*FRR.

Table 2 shows the relative performance impact of running the extensions with *x*BGP programs compared to their native implementation in both BIRD and FRRouting. For each *x*BGP compatible implementation, we run 10 times the *x*BGP programs and compute the convergence time. The convergence time is the time between the first BGP update message is received from Upstream to DuT and the last BGP update message sent from router DuT to Downstream.

Before even loading any *x*BGP extensions, bringing support of *x*BGP in a BGP implementation involves an initial overhead. More specifically, the host implementation must first construct the argument to be passed to the *x*BGP program, then request execution of the corresponding insertion point, and finally execute the *x*BGP termination routine. These additional steps increase the total number of instructions to be executed compared to the native non-*x*BGP implementation. To quantify the cost that `libxbgp` takes in BIRD and FRRouting, we ran both implementations of xBIRD and xFRR without plugins and compared them to their non-*x*BGP compatible versions. Making both implementations of *x*BGP compatible adds a cost in the convergence time of 1% and 1.6% in FRR and BIRD respectively.

We now consider the MED filter (one insertion point) and the extended communities (two insertion points) extensions. When implemented as *x*BGP programs, these slightly increase the convergence time compared to their native version. The Just-In-Time compiler used inside the virtual machine does not optimize as efficiently as the one producing native code. In particular, computation-intensive bytecode involving additions, subtractions, and multiplications take 50% more time to run than native code. This overhead is even worse when considering division and modulo operations.

Yet, we observe a higher convergence time increase for FRRouting than BIRD. By analyzing the execution of each *x*BGP bytecode with a code profiler, we identified two main reasons for this difference. First, to communicate with the host implementation, the *x*BGP program must pass through a dedicated *x*BGP API. For security reasons and because of the internal mechanism of `libxbgp`, the data of the host implementation are first translated into a neutral representation, then copied into a dedicated memory area, accessible in writing and reading by the bytecode. Translation and copying play an important role in the execution of a plugin but are needed to run the same *x*BGP program in several BGP implementation. BIRD internally uses data structures that are closer to the *x*BGP neutral representation than the FRRouting ones, hence involving less translation overhead. Second, FRRouting and BIRD have different internal architectures. The interactions between the `libxbgp` API and the BGP implementations are different. FRRouting is less flexible as its implementation is not designed to be quickly extended with new functionalities. While in BIRD, most of the insertion points map to a specific place, in FRRouting some insertions points must be repeated at different code places, involving up to four times more *x*BGP program calls than BIRD.

We now consider the Route Reflection extension covering the whole BGP workflow. Supporting this feature requires a list of all iBGP client peers. Routers' implementations use their dedicated CLI syntax to define all their iBGP client peers. `libxbgp` does not have access to this CLI configuration since it is implementation-dependent. Instead, it relies on its configuration data within the manifest that can be accessed at any time by the *x*BGP program. On average, the BIRD's convergence time is 7.5% slower than the native code while FRRouting's one is 13% slower. The previous elements still hold to explain the difference between BIRD and FRRouting. In particular, there are more calls to *x*BGP programs in FRRouting due to its code architecture than in BIRD (`BGP_ENCODE_MESSAGE` is called 4 times more), and the translation time to convert data structures is non-negligible in FRRouting (up to 40% overhead for the import filter). Still, the performance overhead of *x*BGP remains in acceptable bounds.

5 Use Cases

Section 2 presented the *GeoTLV* feature to demonstrate that *x*BGP programs can create new attributes that influences the router. Section 4 presented the MED filter, extended communities, and route reflectors to make a performance comparison. This section presents other use cases, which are not implemented natively in FRRouting and BIRD, that illustrate the advantages of *x*BGP for various classes of problems that the operator wants to solve. It is true that the features of this section can be implemented in any BGP implementation without *x*BGP. However, feature support depends on the pace of implementation by all vendors. Thanks to the *x*BGP design, an operator can quickly design its features and introduce them into the network before they are implemented by the vendor. xBGP is the first step to bring extensibility to the network. The first use case defines an *x*BGP program (Section 5.1) to influence the decision process and the import and export filters from the BGP client point of view. The second use case detects zombie routes (Section 5.2). These are routes that are installed in the routing table but are no longer reachable. Third, operators always try to understand the state of their network to improve it as much as possible. We present two use cases (Section 5.3 and 5.4), where BGP is monitored using communities. Due to space limitations, we detail three other extensions in appendix. The fifth use case is related to route filtering in data-centers (Section A.1). It demonstrates that *x*BGP can provide a programmable interface to design complex import and export filters. Our sixth *x*BGP program (Section A.2) gives another example of a special filter that checks the origin of a route. Finally, our seventh use case (Section A.3) shows that an *x*BGP compatible implementation can leverage IGP information to make routing decisions.

Use Case	**C LoC**	**eBPF Insts**	**Total Verif Time(s)**
Geo TLV (§2)	388	1340	664
MED Filter (§4)	55	149	79
Extended Communities (§4)	196	322	86
Route Reflection (§4)	509	3853	27
Route Selection (§5.1)	62	148	27
Zombie Detection (§5.2)	1071	5697	277
Decision Monitor (§5.3)	306	437	29
Propagation Time (§5.4)	560	805	73
Valley Free (§A.1)	143	960	182
Prefix Origin (§A.2)	150	661	57
IGP Data (§A.3)	36	149	3

Table 3: Verification of the *x*BGP programs supporting our use cases.

Table 3 reports the size of the *x*BGP bytecode, the number of lines of code and the time taken to validate every *x*BGP program according to the properties defined in Section 3.

5.1 Customer Selecting Routes

A BGP router only selects one route for each prefix even though it learns multiple routes. As a result, it will only send one route to each BGP neighbor, which decreases the path diversity. Consider Figure 5 to illustrate the situation. AS1, a multihomed stub network having peering links with `Transit 1` and `AS2`. We are interested in the propagation of the routes to the destination network depicted in gray. To maximize path diversity in AS1, it should learn the purple path from AS2 to leverage the two different transits. However, AS1 cannot influence the decision process of AS2's routers.

Enabling the dissemination of multiple routes can bring several benefits such as load-balancing [45], avoiding route oscillation [29] and faster local recovery upon a network failure [61]. With *x*BGP it becomes possible to influence the border router to announce the route the client prefers. To design such an *x*BGP program, all edge routers must enclose their BGP client to one Virtual Routing and Forwarding table

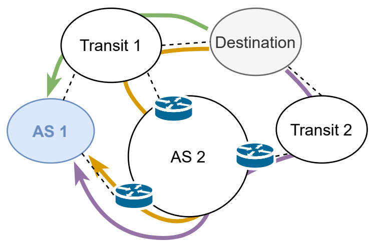

Figure 5: Path Diversity in a Network.

(VRF) [70]. All the routes learned from all neighbors will be exported to the main BGP-VPN RIB's router and then exported to the VRF of each client so that they can have a full view of the routes. Since all clients are in their respective VRFs, the BGP decision process is different for each of them and can therefore be influenced by an *x*BGP program that decides which route to advertise.

We designed a simple *x*BGP program that randomly selects one of the available routes in the VPN RIB thanks to the *x*BGP API function `get_vrf`. It demonstrates that *x*BGP allows the operator to create a customized and more powerful route selection compared to the traditional router CLI. Accessing the VPN RIB through a simple router configuration is something that cannot be done with traditional BGP implementation. Furthermore, an *x*BGP program has access to the entire BGP route and the internal data structure of the BGP router. *x*BGP therefore provides greater flexibility compared to the classical CLI.

We were successfully able to check the termination with T2, the C errors with CBMC, its compliance to the *x*BGP return values. We also verified that the program does not use API functions altering the BGP internal state.

5.2 Detecting BGP Zombies

When a route becomes unavailable, a BGP router sends a withdraw message to all its peers. Because of software bugs [22], it may happen that one of these BGP peers fails to process such a withdraw message. As a result, the route is still considered reachable by the failed router. This is an operational problem because the withdrawal is not propagated, and part of the network still believes that the route is available. If packets still follow this zombie route, they will be black holed. Measurements indicate that these zombie routes are common and affect many ASes [50].

To detect zombie routes, we designed an *x*BGP program that is executed periodically. It uses the timestamp of the arrival of a route in the RIB to detect the routes that are older than *x* days. Our threshold is arbitrarily fixed to a day. Our *x*BGP program is configured to be executed during the maintenance window. It parses the entire BGP RIB thanks to the API functions `*_rib_iterator`. If a route is older than the configured threshold, it is flagged as a possible zombie. To confirm the status of the route, the router needs to request it again from the peer that announced it. This could be done with standardized mechanisms such as Graceful Restart [58] or Route Refresh [12, 51]. However, those two approaches require the remote router to announce again its entire BGP routing table. For the sake of performance, we decided to only ask the remote peer to reannounce the routes flagged by the *x*BGP program. We introduce a new type of BGP message called `BGP Refresh`. It contains a list of prefixes that the router wants to confirm. The peer receiving the `BGP Refresh` message will announce a withdraw or an update message if the routes are not available anymore or still in its BGP routing table respectively. *x*BGP allows sending BGP messages via the `schedule_bgp_message` API function.

It is difficult with a traditional BGP implementation to detect such a zombie route. Indeed, there is no mechanism to analyze and perform an action according to the state of the BGP routing table. To include this feature, the network operator must convince each router vendor to add this feature into its implementation. This use case demonstrates that *x*BGP can outperform the current configuration method that is proposed in classical BGP implementation.

This *x*BGP program successfully passes the T2 and CBMC verifications. As it manages BGP messages, we verified their compliance to the RFC. We also checked that the size of the buffers announced to the API function matches their real size. This program *x*BGP is an example of functionality that cannot be performed with the traditional router CLI while the router is running.

5.3 Monitoring the BGP Routing Decision

Currently, if a network operator would like to debug its BGP routers, he only has monitoring information from the routers it directly controls. This is due to the fact that the traditional BGP specification only provides the exchange of local routing information but does not provide any abstraction to send monitoring information about the routing process. Yet, a support of a dedicated monitoring channel has been proposed [60] but this is still not implemented on all vendor's routers. In a nutshell, a BGP router can ask its neighbor to give different metrics such as its number of reachable prefixes, its ADJ-RIB-IN, its current state, etc. This shows that many network operators need to monitor the BGP session to enable better control of routing information in the network. Some router vendors actually provide commands to retrieve the local state of a router. However, the information is restricted to the router view only and does not include the status of routers that are outside the operator's management scope. Having statistics from other routers could bring many benefits such as the selection of a better route. More specifically, if the BGP router sends its best routes with the step at which routes have been decided, the remote BGP router can learn much information about the route diversity in a network. If the routes are always decided at the very end of the BGP decision process, it indicates a lack of diversity in the remote network. On the contrary, if the routes are decided early in the process such when the AS_PATH is shorter than the previous best route, it can indicate a higher level of diversity. Thanks to this information, the BGP router can adapt its behavior to prefer a path with more diversity to be more resilient to a router or link failure.

We leverage *x*BGP to instrument the BGP implementation to retrieve at which step of the BGP decision process the route has been chosen. Each time it runs, an *x*BGP program retrieves the reason of the decision in the process. It can be retrieved through the arguments passed to the *x*BGP program. To inform other BGP routers, this information is added as a BGP community when sent to other BGP speakers. This is done by the API function `set_attr_to_route`. This way, other routers can parse and use this information to adapt their routing strategies. This *x*BGP program also collects statistics about the other steps of the BGP decision. Each time a route is selected, the *x*BGP program increments an internal counter. It repeats the operation for each decision step. When a route is sent to any peer, these statistics are attached as a community. The BGP router receiving the statistics can have a broader view of the current routing table of its peer. If all the routes have been decided by the BGP tiebreaker that compares the IP address of the router that sends the route then it shows a lack of path diversity. The network operator could then attribute a lower preference to the route advertised by the remote router.

This *x*BGP program successfully passes all the verifiers. Since it handles the BGP community attribute, we verified if the format is respected according to the corresponding RFC.

Since this information cannot be retrieved with traditional router CLI, this new approach could enable more fine-grained routing decisions. Indeed, this new type of active monitoring cannot be achieved with traditional monitoring tools such as BMP, SNMP, etc. as these later tools do not modify the BGP message they sent to the BGP neighbor. Network operators have thus at their disposal new information from outside their network.

5.4 Measuring BGP Route Propagation Times

For mission critical systems, the convergence time of a routing protocol is an important metric to know. It helps to better understand what could be the cause of a slow convergence. Discussions with network operators indicated that commercial router vendors provide undocumented CLI commands to access profiling points. However, this profiling information is local to each router. It could be useful to exchange such information within an entire network. This could open new opportunities to better understand the current state of the network. One example of such monitoring is the time taken by a BGP route to traverse an AS. To support such monitoring information, BGP must be augmented to add in each route its arrival time at each AS border router. Our *x*BGP program defines a new non-transitive BGP attribute, called `RECEIVED_TIME`. It adds this attribute when a route is received over an eBGP session (thanks to the `set_attr` *x*BGP API function family). It traverses the AS with the BGP route until it reaches an edge router. The `RECEIVED_TIME` attribute is removed when the associated route is sent over an eBGP session and the border router computes the difference between its current NTP time and the one of the attribute. As for the previous use case, exchanging such monitoring information is not currently feasible with traditional routers. These two use cases show that *x*BGP can perform a new type of active monitoring by exposing the internal data of the BGP implementation itself to inform the other neighbor of the current BGP routing state.

6 Related Work

Protocol programmability. In the late nineties active networks were proposed as a solution to bring innovation back inside the network that was perceived as being ossified [65]. Most of the work in this area focused on the possibility of placing bytecode inside network layer packets. PLAN [66], ANTS [73] and router plugins [19] are examples. In the control plane, researchers built upon this idea to propose new solutions such as the 4D architecture [25], the Routing Control Platform that centralizes routing [11] or Metarouting [27] that proposed to open the definition of routing protocols using a declarative language. While these previous works propose configuration languages or centralized approaches to deal with network programmability, *x*BGP relies on an existing decentralized control plane protocol on which an operator can add its new functionality to locally influence the routing.

Bringing flexibility to an implementation of a network protocol has been studied in the literature. Researchers have proposed using extension codes to extend transport protocols like STP [52], QUIC [18] and the FRRouting implementation of OSPF and BGP [80]. However, the architecture of these pluginized approaches is close to the internal architecture of a single protocol implementation and does not offer the flexibility to pluginize different implementations of the same protocol. *x*BGP goes one important step further by enabling very different implementations to execute the same *x*BGP program. *x*BGP tries to determine what all implementations of a protocol have in common to try to find a common usable interface.

To ease the automation and the configuration of their devices, routers vendors added scripting languages that enable the network operator to execute recurrent tasks [6]. However, this acts as a simple shell that cannot be used to extend the router implementation. Other vendors integrated the python language into their router OS [40] to perform automation task more easily, such as configuring the router or executing a monitoring routine when a particular event occurs.

Reducing the BGP implementation to its minimum has been studied with CoreBGP [74]. However, it only manages the basic BGP Finite State Machine on which plugins written in the Go language are inserted. The remainder of the BGP logic such as sending BGP messages or managing the routing table is passed to the plugins. CoreBGP plugins react to an FSM events while *x*BGP programs react to protocol events defined by the insertion points depicted in Figure 1.

XORP [33, 34] was introduced to propose an open-source software router platform. This solution has been designed to allow researchers to easily develop their own extensions to a routing protocol. Other open-source routing stacks have been developed such as Quagga [2], FRRouting [23] or BIRD [16]. While these open-source stacks allow modifying the source code of the routing software, *x*BGP goes one step further by introducing a simple API to interact with the routing software. There is no need to look directly in the code of the implementation to understand how to integrate an extension. Anyone who wants to add their own extension will interact with the router through *x*BGP. Throughout this paper, we demonstrated that an *x*BGP extension code written only once can be successfully executed by two open-source routing stacks, FRRouting and BIRD.

Virtual Machines. `libxbgp` is based on a user-space implementation of the kernel eBPF VM [53]. In recent years, Linux kernel developers have integrated a virtual machine called eBPF [63] which enables programs to inject executable bytecode at specific locations inside the kernel. It was initially targeted at monitoring kernel operations [35], but also for fast packet processing [35]. Researchers have used eBPF to support networking programming with IPv6 Segment Routing [83] and extend TCP [68]. Other frameworks could have been used such as WebAssembly [32] or lua [38] that is widely used in industrial systems. Using another type of VM can be studied to measure its performance and its relevance to routing protocols.

Verification tools. The PDS (Plugin Distribution System) [57] provides secure verification and distribution of extension code for Pluginized QUIC [18]. It allows the automation of different types of verification for several extension codes at the same time. Our *x*BGP toolchain includes more verifiers and checks more properties. While the PDS uses a Merkel tree to secure the distribution of plugins, the *x*BGP toolchain simply keeps them in a store that is used by the network operator.

7 Conclusion

We presented *x*BGP, a new paradigm that enables network operators to innovate in routing protocols. *x*BGP allows them to write their extensions or modifications in the form of an *x*BGP program that can be executed inside the protocol implementation. This programmability could help network operators innovate with existing distributed routing protocols as Software Defined Networking lead to the development of programmable switches. Our solution has been proposed for BGP but could also be adapted to support other routing protocols. We further introduced the *x*BGP toolchain that allows operators to annotate *x*BGP programs to verify their safety. It checks if the *x*BGP program meets the local properties of the router such as the termination, the memory constraints and if the *x*BGP program meets the definition of BGP. If it passes the verification step, the *x*BGP program can be safely added to the BGP implementation and is guaranteed not to corrupt the router. Finally, we demonstrated *x*BGP's capabilities by proposing several use cases that have been implemented with our solution. Among them, *x*BGP enables the operator to add new attributes to a BGP route, implementing complex filters, allowing a client to influence the BGP decision process and executing background tasks.

Future Directions. We see two directions to improve *x*BGP. The first would be to look at how to structure an existing BGP implementation to support *x*BGP more efficiently. The second is related to the virtual machine used. eBPF was the most mature virtual machine during the development of *x*BGP. However, other virtual machines such as WebAssembly seem more promising and start to perform well. It might be interesting to see the advantages of using them in the context of *x*BGP.

Software artifacts

To encourage other researchers to reproduce and extend our results we provide the entire source code of `libxbgp` [78] composed of 3506 LoC, the eBPF virtual machine we use (2236 LoC), the two versions of FRRouting [77] (+2675 LoC) and BIRD [76] (+2083 LoC) *x*BGP compatible, the whole *x*BGP programs (15 programs) we developed on top of *x*BGP [81], the experimental scripts we use to evaluate the impact of the performance with our approach (853 LoC) [78] and our verification toolchain based on the PDS [56]. We will also provide the set of annotation to verify *x*BGP programs (1121 LoC) [82].

Acknowledgments

This work has been partially supported by the French Community of Belgium through the funding of a FRIA (Fund for Research training in Industry and Agriculture) grant. We thank the anonymous reviewers and our shepherd, Phillipa Gill, whose helpful comments improved the quality of this paper.

References

[1] The linux kernel static checker. https://github.com/torvalds/linux/blob/master/kernel/bpf/verifier.c.

[2] Quagga software routing suite. https://www.nongnu.org/quagga/.

[3] "Microsoft Azure". Software for open networking in the cloud. https://azure.github.io/SONiC/.

[4] T. Bates, E. Chen, and R. Chandra. BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP). RFC 4456 (Draft Standard), April 2006. Updated by RFC 7606.

[5] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 437–451, 2017.

[6] Raymond Blair, Arvind Durai, and John Lautmann. *Tcl scripting for Cisco IOS*. Cisco Press, 2010.

[7] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. T2: temporal property verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 387–393. Springer, 2016.

[8] R. Bush and R. Austein. The Resource Public Key Infrastructure (RPKI) to Router Protocol. RFC 6810 (Proposed Standard), January 2013.

[9] R. Bush, K. Patel, and D. Ward. Extended Message Support for BGP. RFC 8654 (Proposed Standard), October 2019.

[10] 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.

[11] 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, 2005.

[12] E. Chen. Route Refresh Capability for BGP-4. RFC 2918 (Proposed Standard), September 2000. Updated by RFC 7313.

[13] Enke Chen, Naiming Shen, and Robert Raszuk. Carrying Geo Coordinates in BGP. Internet-Draft draft-chen-idr-geo-coordinates-02, Internet Engineering Task Force, October 2016. Work in Progress.

[14] Edmund Clarke and Daniel Kroening. Ansi-c bounded model checker user manual. *School of Computer Science, Carnegie Mellon University*, 2006.

[15] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: beyond safety. In *International Conference on Computer Aided Verification*, pages 415–418. Springer, 2006.

[16] CZ.NIC, z.s.p.o. BIRD internet routing daemon. https://gitlab.nic.cz/labs/bird.

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

[18] Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, and Olivier Bonaventure. Pluginizing QUIC. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 59–74. 2019.

[19] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. Router plugins: A software architecture for next generation routers. *SIGCOMM Comput. Commun. Rev.*, 28(4):229–240, October 1998.

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

[21] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to SDN: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, 2014.

[22] Romain Fontugne, Esteban Bautista, Colin Petrie, Yutaro Nomura, Patrice Abry, Paulo Gonçalves, Kensuke Fukuda, and Emile Aben. BGP zombies: An analysis of beacons stuck routes. In *International Conference on Passive and Active Network Measurement*, pages 197–209. Springer, 2019.

[23] The Linux Foundation. FRRouting project. https://frrouting.org/.

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

[25] 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. *SIGCOMM Comput. Commun. Rev.*, 35(5):41–54, October 2005.

[26] Brendan Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[27] Timothy G. Griffin and Joäo Luís Sobrinho. Metarouting. *SIGCOMM Comput. Commun. Rev.*, 35(4):1–12, August 2005.

[28] Timothy G Griffin and Gordon Wilfong. An analysis of bgp convergence properties. *ACM SIGCOMM Computer Communication Review*, 29(4):277–288, 1999.

[29] Timothy G Griffin and Gordon Wilfong. Analysis of the med oscillation problem in bgp. In *10th IEEE International Conference on Network Protocols, 2002. Proceedings.*, pages 90–99. IEEE, 2002.

[30] Timothy G Griffin and Gordon Wilfong. On the correctness of ibgp configuration. *ACM SIGCOMM Computer Communication Review*, 32(4):17–29, 2002.

[31] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The SeaHorn verification framework. In *International Conference on Computer Aided Verification*, pages 343–361. Springer, 2015.

[32] 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. *SIGPLAN Not.*, 52(6):185–200, June 2017.

[33] Mark Handley, Orion Hodson, and Eddie Kohler. Xorp: An open platform for network research. *SIGCOMM Comput. Commun. Rev.*, 33(1):53–57, jan 2003.

[34] Mark Handley, Eddie Kohler, Atanu Ghosh, Orion Hodson, and Pavlin Radoslavov. Designing extensible ip router software. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 189–202, 2005.

[35] 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*, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[36] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, pages 15–26, 2013.

[37] G. Huston and G. Michaelson. Validation of Route Origination Using the Resource Certificate Public Key Infrastructure (PKI) and Route Origin Authorizations (ROAs). RFC 6483 (Informational), February 2012.

[38] Roberto Ierusalimschy. *Programming in lua*. Roberto Ierusalimschy, 2006.

[39] 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.

[40] Junos. Junos PyEZ developer guide. https://www.juniper.net/documentation/en_US/junos-pyez/information-products/pathway-pages/junos-pyez-developer-guide.html, July 2021.

[41] K. Kompella, B. Kothari, and R. Cherukuri. Layer 2 Virtual Private Networks Using BGP for Auto-Discovery and Signaling. RFC 6624 (Informational), May 2012.

[42] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2014.

[43] Daniel Kroening and Michael Tautschnig. CBMC–C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[44] P. Lapukhov, A. Premji, and J. Mitchell (Ed.). Use of BGP for Routing in Large-Scale Data Centers. RFC 7938 (Informational), August 2016.

[45] Petr Lapukhov and Jeff Tantsura. Equal-Cost Multipath Considerations for BGP. Internet-Draft draft-lapukhov-bgp-ecmp-considerations-07, Internet Engineering Task Force, June 2021. Work in Progress.

[46] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding bgp misconfiguration. *ACM SIGCOMM Computer Communication Review*, 32(4):3–16, 2002.

[47] Robert Mc Millan. Research experiment disrupts internet, for some. *Computerworld*, pages August, 28, 2010. https://www.computerworld.com/article/2515036/research-experiment-disrupts-internet--for-some.html.

[48] 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.

[49] Microsoft Corporation. ebpf for windows. https://github.com/microsoft/ebpf-for-windows.

[50] Porapat Ongkanchana, Romain Fontugne, Hiroshi Esaki, Job Snijders, and Emile Aben. Hunting BGP zombies in the wild. In *Proceedings of the Applied Networking Research Workshop*, ANRW '21, page 1–7, New York, NY, USA, 2021. Association for Computing Machinery.

[51] K. Patel, E. Chen, and B. Venkatachalapathy. Enhanced Route Refresh Capability for BGP-4. RFC 7313 (Proposed Standard), July 2014.

[52] Parveen Patel, Andrew Whitaker, David Wetherall, Jay Lepreau, and Tim Stack. Upgrading Transport Protocols using Untrusted Mobile Code. *ACM SIGOPS Operating Systems Review*, 37(5):1–14, 2003.

[53] IO Visor Project. Userspace eBPF VM. https://github.com/iovisor/ubpf.

[54] Y. Rekhter (Ed.), T. Li (Ed.), and S. Hares (Ed.). A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006.

[55] E. Rosen and Y. Rekhter. BGP/MPLS IP Virtual Private Networks (VPNs). RFC 4364 (Proposed Standard), February 2006. Updated by RFCs 4577, 4684, 5462.

[56] Nicolas Rybowski. Plugin Distribution System. https://github.com/nrybowski/SPMS.

[57] Nicolas Rybowski, Quentin De Coninck, Tom Rousseaux, Axel Legay, and Olivier Bonaventure. Implementing the plugin distribution system. In *Proceedings of the SIGCOMM '21 Poster and Demo Sessions*, page 39–41. Association for Computing Machinery, New York, NY, USA, 2021.

[58] S. Sangli, E. Chen, R. Fernando, J. Scudder, and Y. Rekhter. Graceful Restart Mechanism for BGP. RFC 4724 (Proposed Standard), January 2007. Updated by RFC 8538.

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

[60] Rob Shakir, Robert Raszuk, Rob Shakir, and David Freedman. BGP OPERATIONAL Message . Internet-Draft draft-frs-bgp-operational-message-00, Internet Engineering Task Force, July 2011. Work in Progress.

[61] M. Shand and S. Bryant. IP Fast Reroute Framework. RFC 5714 (Informational), January 2010.

[62] Rachee Singh, Muqeet Mukhtar, Ashay Krishna, Aniruddha Parkhi, Jitendra Padhye, and David Maltz. Surviving switch failures in cloud datacenters. *ACM SIGCOMM Computer Communication Review*, 51(2):2–9, 2021.

[63] A Starovoitov. BPF-in-kernel virtual machine. *Linux Kernel Developers' Netconf*, 2015.

[64] 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, 2018.

[65] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *SIGCOMM Comput. Commun. Rev.*, 37(5):81–94, October 2007.

[66] D.L. Tennenhouse and D.J. Wetherall. Towards an active network architecture. In *Proceedings DARPA Active Networks Conference and Exposition*, pages 2–15, 2002.

[67] The OpenBSD Project. Openbgpd. `http://openbgpd.com/`.

[68] Viet-Hoang Tran and Olivier Bonaventure. Beyond socket options: making the linux TCP stack truly extensible. In *2019 IFIP Networking Conference (IFIP Networking)*, pages 1–9, 2019.

[69] Yves Vanaubel, Jean-Jacques Pansiot, Pascal Mérindol, and Benoit Donnet. Network fingerprinting: Ttl-based router signatures. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 369–376, 2013.

[70] Laurent Vanbever. Customized BGP route selection using BGP/MPLS VPNs. In *Routing Symposium, Cisco Systems*, 2009.

[71] Stefano Vissicchio, Luca Cittadini, and Giuseppe Di Battista. On iBGP routing policies. *IEEE/ACM Transactions on Networking*, 23(1):227–240, 2014.

[72] Matthias Wählisch, Fabian Holler, Thomas C Schmidt, and Jochen H Schiller. Rtrlib: An open-source library in c for rpki-based prefix origin validation. In *Presented as part of the 6th Workshop on Cyber Security Experimentation and Test*, 2013.

[73] D.J. Wetherall, J.V. Guttag, and D.L. Tennenhouse. Ants: a toolkit for building and dynamically deploying network protocols. In *1998 IEEE Open Architectures and Network Programming*, pages 117–129, 1998.

[74] Jordan Whited. Corebgp - plugging in to bgp. `https://github.com/jwhited/corebgp`, July 2020.

[75] Thomas Wirtgen. xBGP api documentation. `https://github.com/pluginized-protocols/xbgp_plugins/blob/master/xbgp_compliant_api/xbgp_plugin_api.h`.

[76] Thomas Wirtgen. xBGP bird. `https://github.com/pluginized-protocols/xbgp_bird`.

[77] Thomas Wirtgen. xBGP frrouting. `https://github.com/pluginized-protocols/xbgp_frr`.

[78] Thomas Wirtgen. xBGP source code. `https://github.com/pluginized-protocols/libxbgp`.

[79] Thomas Wirtgen, Quentin De Coninck, Randy Bush, Laurent Vanbever, and Olivier Bonaventure. xBGP: When you can't wait for the ietf and vendors. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 1–7, New York, NY, USA, 2020. Association for Computing Machinery.

[80] Thomas Wirtgen, Cyril Dénos, Quentin De Coninck, Mathieu Jadin, and Olivier Bonaventure. The case for pluginized routing protocols. In *27th International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2019.

[81] Thomas Wirtgen and Tom Rousseaux. xBGP plugins source code. `https://github.com/pluginized-protocols/xbgp_plugins`.

[82] Thomas Wirtgen and Tom Rousseaux. xBGP verification. `https://github.com/pluginized-protocols/xbgp_plugins/tree/master/prove_stuffs`.

[83] 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*, CoNEXT '18, page 67–72, New York, NY, USA, 2018. Association for Computing Machinery.

A Additional use cases

This section describes some additional use cases for *x*BGP.

A.1 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 [44]. 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 forces the network operators to tweak their BGP configurations [44]. These tweaks make BGP configurations complex and more difficult to analyze and validate [5]. To illustrate this complexity, let us consider the data center shown in Fig. 6. Routers $S1$ and $S2$ are the Spine routers, $L10 \dots L13$ the leaf routers, and $T20 \dots$ the top-of-the rack routers. In such a data center, there is no direct connection between the routers at the same level in the hierarchy. Data center operators usually want to avoid paths that include a valley (e.g. $L10 \rightarrow S1 \rightarrow L11 \rightarrow S2$). To achieve this, they usually run eBGP between routers, but configure the same AS number on $S1$ and $S2$ (even if these routers are not 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

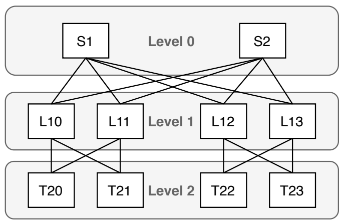

Figure 6: A simple data center.

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 6. If both links $L10-S1$ and $L13-S2$ fail, then the only possible path between $L10$ and $L13$ is $L10 \rightarrow S2 \rightarrow L12 \rightarrow S1 \rightarrow L13$. If the same AS number is used on $S1$ and $S2$, this path will never be advertised.

With *x*BGP, a network operator can use different AS numbers for their routers and implement specialized filters on the spine and leaf routers. For example, if $S1$ and $S2$ are both connected to transit providers and can reach the same prefixes, then $L10$ should never reach $S2$ via $S1$ and $L11$. However, this path should remain valid if the final destination is a prefix attached to below $R13$.

To implement such a filter, we load a manifest containing every eBGP session from a router of level i to a router of level $i+1$ in a pair having the following form: $(AS_{li}, AS_{l(i+1)})$. For each route, the filter checks each consecutive pair of the AS-Path. If a pair of this manifest is included in the AS-Path, the filter rejects the route since it is not valley-free.

This *x*BGP program successfully passes T2 and CBMC checks. SeaHorn confirms its *x*BGP compliance relative to its use of the API functions and the return values.

A.2 Validating BGP Prefix Origins

The interdomain routing system is regularly affected by disruptions caused by invalid BGP advertisements originated from ISPs. 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 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 illustrate the flexibility of *x*BGP, we consider a RPKI-based route origin [37] validation variant. The network operator includes in the configuration data of the manifest all prefixes it knows the origin. We assume the operator has themself validated the ROA signatures before generating the file. This file is used by the *x*BGP program each time a BGP route is received by a peer to check if the origin AS of the route matches with the one contained in the file.

To evaluate the performance of our prefix origin validation, we use the same testbed as in Section 4 excepted that we use eBGP sessions for links L1 and L2. Our *DuT* does not implement the RPKI-Rtr protocol [8, 72] but loads configuration data 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.

Table 2 compares our extension codes running on BIRD and FRRouting to their native implementations without any prefix validation. We do not compare our solution with the RPKI-Rtr protocol since we do not totally implement the RPKI protocol. We only check the origin of the route. The difference in execution between the two implementations is also explained by the difference in the internal representation of the data structures used.

The termination and absence of C errors were proved with T2 and CBMC. SeaHorn also confirms that the *x*BGP program does not write any data in the memory of the host implementation and its compliance on the return values.

A.3 Filtering Routes Based on IGP Costs

Since the *x*BGP 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 a worldwide presence that wants to announce to its peers the routes that it learned in the same continent as the advertising BGP. This policy can be implemented by tagging routes with BGP communities on all ingress routers and then filtering them on export. While being frequently used [20], this solution is imperfect. Consider an ISP having two transatlantic links terminated in London, UK, and Amsterdam in The Netherlands. This ISP has a strong presence in Europe and two links connect the 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 to the UK. When such a failure occurs, the ISP does not want to advertise the routes learned in the UK to its European peers. With BGP communities, it would continue to advertise these routes after the failure.

Using the *x*BGP API, the operator could implement this policy as follows. First, he configures the IGP cost of the transatlantic links at a high value, say 1000 to discourage their utilization. Second, he implements a simple export filter that checks the IGP cost of the next-hop 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` ④ insertion point. If the IGP cost to the BGP next hop distance is acceptable, the function 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 proposes to fall back to the native code. To reject the route, the extension code returns the special value `FILTER_REJECT` to the host implementation.

For this *x*BGP program, we used SeaHorn to ensure return

Flag	Enables
PROVERS	Verification macros.
PROVERS_ARGS	next() call verification macros.
PROVERS_T2	T2 related macros.
PROVERS_CBMC	CBMC related macros.
PROVERS_SEAHORN	SeaHorn related macros.

Table 4: Verification flags.

Macro	Code only provided to
PROOF_INSTS_SEAHORN	SeaHorn.
PROOF_INSTS_CBMC	CBMC.
PROOF_INSTS_T2	T2.

Table 5: Macro allowing to provide pieces of code for a specific verifier.

values were meaningful to `libxbgp`. T2 and CBMC are also used to check the termination and the absence of any C errors. We also verify that the *x*BGP program has only a read-access to the host implementation.

```
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) {
    next(); // Do not filter on iBGP sessions
  } if (nexthop->igp_metric <= MAX_METRIC) {
    next(); // the route is accepted by this filter;
  }         // 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.

B Verification macros

This appendix provides in Tables 5, 6, and 7 exhaustive lists of our custom-made verification macros. Those are enabled at compile time with different flags described in Table 4.

Macro prefix	Attribute name/macro suffix	Check
BUF_CHECK_*	LENGTH	The correct formatting of the attribute which is stored in a buffer.
	ORIGIN	
	ASPATH	
	NEXTHOP	
	MED	
	LOCAL_PREF	
	ATOMIC_AGGR	
	AGGREGATOR	
	COMMUNITY	
	ORIGINATOR	
	CLUSTER_LIST	
	EXTENDED_COMMUNITIES	
	AS4_PATH	
	AS4_AGGREGATOR	
	AIGP	
	LARGE_COMMUNITY	
CHECK_*	LENGTH	The correct formatting of the attribute which is stored in a `path_attribute` structure.
	ORIGIN	
	ASPATH	
	NEXTHOP	
	MED	
	LOCAL_PREF	
	ATOMIC_AGGR	
	AGGREGATOR	
	COMMUNITY	
	ORIGINATOR	
	CLUSTER_LIST	
	EXTENDED_COMMUNITIES	
	AS4_PATH	
	AS4_AGGREGATOR	
	AIGP	
	LARGE_COMMUNITY	
CHECK_IN_BOUNDS_*	LOCAL_PREF	The given attribute lies in the range specified by the operator.
	MED	
CHECK_*	ARG	The next() function is called if the xBGP program cannot parse the current attribute.
	ARG_CODE	
	OUT	
	RET_VAL_FILTER	

Table 6: BGP attributes verification macros used by SeaHorn.

Macro prefix	Target	Check
CHECK_*	BUFFER	The given buffer respects the specified size.
	STRING	The given string is null-byte terminated.
	COPY	The copied buffer is unchanged.

Table 7: Memory check macros used by SeaHorn.