

École polytechnique de Louvain

Leveraging eBPF to Improve the Flexibility of Routing Protocols

The Case for Border Gateway Protocol (BGP)

Author: **Thomas WIRTGEN**
Supervisor: **Olivier BONAVENTURE**
Readers: **Quentin DE CONINCK, Marc LOBELLE**
Academic year 2018–2019
Master [120] in Computer Science

Abstract

Today's Routing Protocols are the core features of the Internet. As protocols' requirements evolve with the time, network operators are faced with some challenges that can't be solved unless directly change the protocol behavior. However, convincing router vendors to extend the proficiency of the protocol is actually difficult. More, if the extension requires to redesign the protocol, the IETF has to take into consideration whether or not to adapt its standardization. It often takes many years to first obtain a standard for a new proposal, and then finally see it implemented into the vendor routers. To face with this lack of responses, some operators in data-centers adopt Software Defined Networking to obtain flexibility for routing protocols.

This master's thesis proposes an innovating approach to provide flexibility in standardized routing protocols. We leverage a virtual machine based on eBPF that is able to execute bytecode. We propose to include the virtual machine inside the implementation of the Border Gateway Protocol (BGP). This integration allows the protocol implementation to run eBPF bytecodes provided by a network operator. These codes can manipulate the protocol in order make extensions or replace completely the behavior of routing algorithms through an API that makes the link between bytecodes and the BGP protocol.

We suggest to use an existing implementation of an eBPF virtual machine, uBPF, on which we build a higher API that fits with the requirements of a real protocol such as BGP. This master's thesis describes the modifications done in order to improve its efficiency to both securely inject bytecodes on a running protocol and extend the capabilities of the existing uBPF machine.

After discussing its easy integration into the BGP FRRouting implementation, we have proven the usefulness of the modified uBPF machine, by demonstrating its benefits and extending BGP throughout several use cases.

Finally, an overview of the performances of BGP with the virtual machine is given. It shows the current advantages and limitations of the proposed approach.

Acknowledgements

The realization of this master's thesis was possible thanks to the help of several people to whom I would like to express my sincere gratitude.

First of all, I would like to thank my supervisor, Professor Olivier Bonaventure. I thank him for his trust, for his availability, his patience and his wise advice.

I would also like to thank Quentin De Coninck and Mathieu Jadin for their good advice and intellectual support throughout this work.

I also thank Sophie, Jeremy and Steven for their encouragement, proofreading and correction of my work.

To all this people, I express my thanks, my respect and my gratitude.

Contents

1	Introduction	1
2	BGP protocol	6
2.1	Internal Working	7
2.2	Type of BGP Sessions	11
2.3	BGP Filters	13
2.4	BGP Decision Process	13
3	The uBPF Machine	16
3.1	Writing eBPF C Code	20
3.2	External or Helper Functions	22
3.3	Passing Arguments To The eBPF Program	24
3.4	Calling the Virtual Machine	24
3.5	Conclusion	25
4	uBPF Integration	26
4.1	Structure of a Plugin	31
4.2	Plugin Manager	32
4.3	A mechanism to inject plugins into the implementation	34
4.4	A New Memory Manager	36
4.5	Adding Memory Bound Checker in JIT Mode	40
4.5.1	Redefining Calls to the uBPF Machine	42
4.6	Secure Communication Between the Virtual Machine and the Protocol	42
4.7	Adding an Execution Context to the Virtual Machine	43
4.8	Exposed BGP functions	47
4.9	List of getter functions created	47
4.10	Conclusion	48
5	Use Cases	49
5.1	BGP Monitoring	50
5.1.1	Monitored protocol variable	51
5.1.2	An external Daemon to prepare data for database	52
5.1.3	IPFIX Templates	53

5.1.4	IPFIX Collector	54
5.1.5	Structure of a monitoring packet	54
5.1.6	MongoDB database	55
5.1.7	MongoDB C Driver	56
5.2	Redesigning the BGP Decision Process	57
5.2.1	Building New Decisions Steps	59
5.3	A new way to add BGP import/export filters	60
5.3.1	An eBPF filter to reduce BGP router memory	64
5.3.2	Taking Advantage of uBPF to Write Complex Filters	65
5.4	Conclusion	65
6	Performance Evaluation	67
6.1	Structure of the Evaluation	67
6.2	Performance evaluation	70
6.2.1	Evolution of the time by the number of plugins attached to the protocol	72
6.2.2	Performance of Monitoring Plugins	76
6.2.3	Performance of the Decision Process Plugins	77
6.2.4	Performance of eBPF filters	78
6.2.5	A Quick Performance Analysis With the VM Interpreted Mode . .	79
6.3	Conclusion	79
7	Conclusion	81

Chapter 1

Introduction

Today's network operators work with commercial and open-source implementation of routing protocols. They act as black boxes since their mechanism is either hidden or too complex to be modified. To propose new extensions or modifications of the protocol, the operator must propose them to an organization as the IETF. Every router of the network contains a standardized implementation of protocols so that they can understand messages from other network nodes. Regardless of their complex architecture, protocols can be customized through the command-line interface (CLI), SNMP [1] or NETCONF [2]. Although these techniques are defined in literature, the IETF does not recommend these for the protocol implementation. Hence, it is quite complex to modify or to propose new extensions to the protocol because of the restricted nature of the implementation. Networks operators are only allowed to tweak what the implementation proposes. This approach significantly restricts new possibilities inside networks, and operators are often faced to problems because of this lack of protocol modification.

We can ask ourselves the following question : *"How to propose a framework that enables a network operator to easily modify the implementation with the least possible constraints?"*

A possible answer is to consider a routing protocol implementation as a grey box. Besides the current existing methods such as CLI, SNMP and NETCONF, we propose a new form of interaction through a virtual machine that allows extensions or modifications of the protocol implementation. This virtual machine is able to read and execute eBPF bytecodes to interact with the protocol through a specific API. This API must be as generic as possible to enable an operator to make modifications where he wants without asking to router vendors to include the desired functionality in the protocol as it would do with the black-box architecture.

BPF was originally introduced in the 1990s to provide an interface for users and developers to analyze network packets into the user space [3]. As each packet is treated in the kernel, it is somewhat tough to make further investigations like debugging a protocol

between two hosts. Therefore, BPF solves this problem by integrating a tool allowing copying packets into the user space. These packets are first filtered by a user predefined routine to only capture what it wants from the kernel space. When the filter is written, it is plugged where packets are received in the kernel and will decide whether the current packet is exported in the user space or not. Each implemented filter is designed to be executed on a RISC-based virtual machine, and thus are isolated from the real kernel code. However, BPF was designed and optimized for old architectures no longer used today, which does not fit into current processor implementation. Indeed, new instructions are built and a processor contains more registers and is now 64-bit large.

extended **B**erkeley **P**acket **F**ilter (eBPF) is an improved version of the classical BPF and is specifically built to take advantage of the current processor architectures. eBPF instructions are the closest to native CPU instructions, which brings a gain of performance.

This project takes the full opportunity of this virtual machine by integrating it to an existant protocol implementation. In our case, we will focus the inclusion on the **B**order **G**ateway **P**rotocol (BGP) taken from the existing **FRRouting** project. Nevertheless, this such implementation is not provided inside the kernel, but is rather implemented in user space. Hopefully, there exists a user space eBPF virtual machine which can be used to mimic the same eBPF machine contained in the Linux kernel [4].

The first step is to provide an efficient way to use this virtual machine inside the BGP protocol by giving a powerful API to ease its manipulation into the protocol. Second, we will provide an example of how to use it for monitoring the BGP protocol. Finally we show how to use eBPF plugins to modify the normal execution flow of BGP protocol by tweaking the BGP decision process and BGP filters.

Motivations

extended Berkeley Packet Filter [5] is originally used inside the Linux kernel to allow a programmer to extract information about the kernel network stack in a simple way. Indeed, directly modifying the kernel code can be cumbersome and relatively difficult. Thus eBPF proposes to the programmer to write some small plugins to extract variables from kernel space to user space. These plugins will be attached to some kernel functions (determined by the programmer) and will be executed in a kind of virtual machine. This has some advantages since the plugin is confined inside its own execution context and does not interfere directly with the kernel code. Since the plugin is directly injected into the kernel code, the eBPF virtual machine checks if a plugin has access to some unauthorized memory space. In fact, the virtual machine tries to isolate as much as possible the execution to avoid crashing the kernel. Other checks are made in this sense such as avoiding the use of loops, etc. eBPF programs are originally written into a special kind of assembly language, which is understood by the virtual machine, that actually emulates a RISC processor. Nevertheless, programming with an assembly language is difficult. Therefore, it is possible to directly write the plugin in C, a higher-level programming language. The clang family compiler proposes the users to convert the C source code

into a BPF binary file (understood by the virtual machine).

As this virtual machine is initially implemented for the Linux kernel, it is not possible to use it to monitor the BGP protocol, that is fully executed into the user space. Hopefully, there is a user space eBPF machine (uBPF) that was implemented to behave the same as the one contained in the Linux kernel. However, FRRouting that actually implements the BGP protocol, does not integrate this uBPF machine yet. Therefore, the first step is to integrate this virtual machine into the existing project. Currently, the uBPF project contains an API to attach an ELF file and executing it. However, this API does not provide the ability to load multiple implemented plugins. It is, once an eBPF code is attached to uBPF machine, not possible to replace the existing attached plugin with another one. Thus, it is necessary to redesign this virtual machine to be fully usable in order to bring flexibility into BGP.

Current implementations of protocols are not designed to be fully compatible with an extra virtual machine. It is necessary to slightly modify the BGP protocol in order to add the integration of this uBPF machine. This work will describe our methodology to provide a powerful mechanism to insert plugins.

Monitoring purpose

Today, if a network engineer wants to inspect the protocol in detail for debugging or analysis purpose, it has to pass through an external device which will capture packets or use tcpdump directly in the router [6]. The external device needs to query a BGP router to extract relevant information. Although Luckie and Beverly actually proposes a methodology to perform this kind of task [7], it is impossible to query every BGP routers since implementations may differ from one router to another. Thus the analysis might miss some important metrics. Simple Network Management Protocol (SNMP) [1] is nowadays used to monitor some variables. The idea is that router vendors implement SNMP variables in charge of observing important protocol states. The router is then polled by an SNMP manager node which will retrieve the SNMP variables. This method works fine if the vendor has implemented multiple SNMP variables that the user can retrieve. However, not all protocol variables can be monitored because the providers do not handle all the metrics that users could collect.

Therefore, eBPF might help to collect measurements that are not accessible by an external monitoring device. Furthermore it could be used to track some complex metrics which is near impossible to accurately retrieve with protocols like SNMP. To take an example eBPF could rigorously compute the time taken for the BGP decision process, which is unmanageable otherwise.

Thus, using eBPF could help users to monitor a given protocol in a fine-grained way.

Protocol modification and extension purpose

Monitoring is not the only purpose of this integration. As the written code could access the protocol variables, it is possible to **alter** and **extend** the execution path of the protocol without modifying its code since the extension is directly executed in the eBPF machine. The advantage is to provide an easier interface to the user without allowing him access to the real protocol code. Moreover, the written code is isolated and thus it is not possible to access memory. This avoids a potential crash of the protocol.

eBPF will be integrated in crucial parts of the protocol, so that a user can write and inject the modification without recompiling the whole project.

This allows a network operator to obtain flexibility to think new configurations about network protocols. The notion of plugin allows to directly test and deploy a new feature or an extension without asking router vendors to integrate the new feature on their machines.

BGP is flexible protocol in the sense that it for example supports the design of new UPDATE messages by handling errors if they are malformed [8], new routes attributes with BGP communities, exchanging the BGP speaker's capabilities [9], etc. However these extensions must be standardized by the IETF before seeing it installed on the vendor routers. The procedure might take a long time. On most of routers, the protocol implementation source code is closed, and therefore not modifiable.

Hence, our vision of the network protocol with a virtual machine tends to evolve from a black box model to a grey box model. Protocols furnished with routers are not fully extensible and the innovation is limited by what users are allowed to configure. This is a factor restraining innovation which enforces network operators to find workarounds to overcome this lack of flexibility [10, 11]. As a consequence, Software Defined Networking (SDN) [12] has been developed to provide a total flexibility of the control plane. SDN uses a fully programmable controller injecting static routes to the SDN nodes (e.g. switches, routers) according to its vision of the network. Control Plane is thus detached from the network nodes so that they only have the data plane to forward packets through their destination. This detachment allows operators to build new features that are complex to design with traditional methods [13].

Our perspective is to keep the advantages of traditional routing protocols by exposing an API to plugins in order to improve the flexibility of those protocols. SDN reaches out having too much flexibility since operators can implement their own policies to control routers. There is an absence of standardized protocols and large networks could become complex to manage without a strong base that protocols provide. This new approach is thus a compromise between flexibility and standardization.

Structure of this Master's Thesis

The master's thesis is organized as follows :

Chapter 2 presents the basic functioning of the Border Gateway Protocol. As our work is essentially focused on this protocol, we need to define the principal architecture of it. All the specific term of the protocol is described to better understand what is modified thanks our solution.

Chapter 3 describes the uBPF virtual machine. When integrated inside a program, it enables to execute other programs thanks to a specific API. Codes passed to this virtual machine are first written in a restrictive subset of the C language and then compiled in a special assembly bytecode called eBPF. This bytecode is eventually executed by uBPF. This chapter introduces the concepts of this virtual machine.

Chapter 4 explains the changes to the virtual machine. Actually, to be fully exploitable inside a network protocol, a higher level of the current virtual machine must be designed. After exposing the weaknesses of the existing uBPF machine, we present all the modifications made to strengthen uBPF and hence enabling its deployment in BGP.

Chapter 5 proves based on some use cases that extending a protocol with uBPF is easier. The previous chapter introduces the mechanism of the virtual machine, i.e. what we have done to allow extending a protocol in a better way. This chapter looks after three typical cases of extensions. First, we discuss a new approach to monitor a protocol. Second we present a new form of BGP decision process that provides flexibility to modify and extend the traditional one. In the third one we show several examples of eBPF bytecodes that are deployed to filter received routes at both the importation and the exportation side of a BGP router. The last use case proves that uBPF can simplify the design of complex filters with a few lines of C code.

Chapter 6 studies the impact of the integration of a virtual machine inside a routing protocol. This chapter first presents all the experimental evaluations made to benchmark the new protocol with uBPF enabled and then exposes a detailed explanation of the gathered results. We present the limitation of our current prototype and we finally describe suggestions for possible improvements.

Chapter 2

BGP protocol

The main subject of this master thesis is to extend the BGP protocol by injecting external code at runtime in order to modify its behavior on-the-fly. This chapter provides a description of this protocol to give a better understanding of its functioning.

The current Internet organization is such that it is divided into multiple networks. Each of these domains contains their own organization of their network and thus defines their own routing policies inside it. Indeed, in order to correctly deliver data from a computer to another one inside the domain, it is necessary to apply an Internal Gateway Protocol (IGP) which will be in charge of discovering the internal routes between every machine that takes part in the organization. Each machine contains their own identification on this network provided thanks to the IP layer. On the Internet, one of the biggest IP networks, this layer provides the role of assigning a unique identification for each node. Each machine is then identifiable both inside and outside the domains on which they belong to. The problem of transmitting data from two network nodes is thus an IP address concern. Concretely, before sending data, the machine looks on its forwarding table to see on which network interface the data must be transmitted. However, the machine that will receive the data on the other side might not be the final destination, and thus must forward the data toward the next node by again looking at its forwarding table. Nodes are actually performing hop-by-hop routing until they reach their final destination.

Each forwarding table is actually filled by protocols that discover where the IP addresses are located in the network. Inside a domain, also called autonomous system (AS), the network administrator could decide to run protocol like OSPF running an algorithm of the shortest path by defining custom metrics to compare distance between two nodes [14]. As previously said, this metric might differ from different ASes, such that two exactly same networks but with different metrics comparing the distance might generate totally different routes between two nodes.

When routes are established inside an autonomous system (i.e. each node reaches all other nodes inside the domain), it remains the connection between inter-domain.

Since each AS are responsible for a subset of IP addresses inside their networks (e.g. a fictional AS is in charge of the **10.15.0.0/16** prefix¹), if an AS want to advertise its presence on their connected neighbor ASes, a new protocol between the border routers of the two neighbor domains must be used to select the best route through the space of AS. An autonomous system is identified by a unique number assigned by IANA [15, 16]. Each node is part of one AS. Therefore, a unique association between a given IP address and an AS number could be made. Indeed, these two values are unique onto the network. This property is used to perform interdomain routing. Before digging into

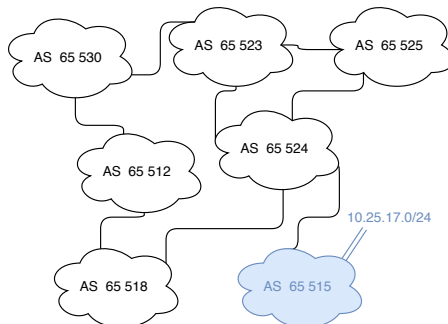


Figure 2.1: Example of network with multiple interconnected Autonomous Systems (AS)

the details of BGP, we will explain a simple procedure to advertise network reachability between multiple domains. On Figure 2.1, the blue AS is responsible for the IPv4 prefix **10.25.17.0/24**. Since it is directly connected to the AS 65 524, the blue AS announces its prefix reachability by sending a message to the other domain which will be in charge of notifying the rest of the bigger network about the location of this IP prefix. By doing this, each node of the network is able to reach each other. We would like that a packet emitted from a given AS travels the least AS possible. In this configuration, it is possible to find a path of at least four ASes. Therefore, any data coming from AS 65 530 having as a destination the AS 65 515 must take the following path :

AS 65 530 --> AS 65 523 --> AS 65 524 --> AS 65 515

The routing inside a domain is totally defined by the AS network administrator and does not interfere with the interdomain routing.

2.1 Internal Working

Border **G**ateway **P**rotocol (BGP), defined in RFC4271 [17], is the interdomain routing protocol used on the Internet. It provides a powerful mechanism for exchanging information between other BGP peers on their actual knowledge of the network. The figure 2.2 shows the connection between two networks. The edge routers of the two ASes are actually

¹Every computer inside this domain contains an IP address from **10.15.0.1** to **10.15.255.255**

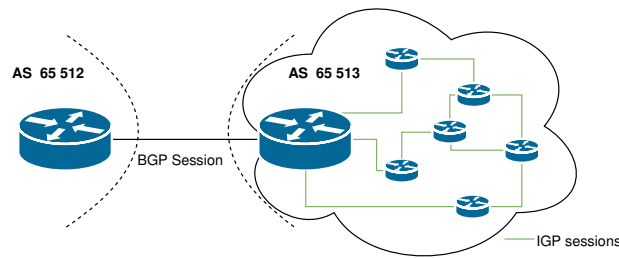


Figure 2.2: Example of BGP Session Between Two Connected AS

establishing an external BGP session to send each other various messages about their network reachability. These two nodes are physically connected and must know the IP address of their neighbor to establish a BGP session. By default the protocol opens a TCP connection on port 179 and continuously listens for incoming messages. As long as the router is alive, the connection is never closed. Thus, on non-faulty connection, routers regularly send messages. If the physical link or one of these edge routers fails, the alive one reset the connection. Every route learned from its neighbors are removed. It will then try to send an OPEN message to reestablish a BGP link with the failed router.

BGP speakers send four types of messages according to the state of the protocol :

- **BGP OPEN** : the message is sent when the router opens the session. This is the very first message sent to the other peer. It contains some information to negotiate some functionalities to use during the communication (i.e. agreeing on their capabilities [9])
- **BGP NOTIFICATION** : used to inform the other peer about a problem during the session. It could be either an error on the BGP message sent by the other peer or a problem on the router itself (e.g. not enough memory space to accept more routes). The BGP session between the two speakers is immediately stopped when sending a notification.
- **BGP UPDATE** : this message is used to inform the other peer to advertise, re-advertise or withdraw routes to the other peers such that it can update its routing table.
- **BGP KEEPALIVE** : this message is regularly sent to keep the BGP session alive. It could be the case if there is no BGP update message sent during a long time when the network is stable. In order to not to close the connection because of inactivity, routers send by default every 30 seconds a BGP keepalive message, resetting the hold timer to 90 second². This counter is used to close the connection when no message has been received during this interval. If this hold timer counts down to 0, all the routes learned from the inactive peer are deleted and BGP will retry

²These values are the one suggested by the RFC 4271 [17]

a connection by sending **OPEN** messages. Therefore, by sending BGP keepalive messages every 30 seconds, the remote router will never reach the end of the hold timer.

When a BGP speaker receives an update message, it actually contains routes for multiple prefixes. The message is divided in two parts. The first part of the message contains new routes or updates of the actual routes already sent beforehand. The second type of route is the one to be withdrawn because the router cannot reach these prefixes anymore. An update message could then carry several routes in order to optimize the computation when the message is received.

In addition to the advertisements and withdrawals, a third type of information is added to the messages, the attributes of the prefixes contained in the update. Since a router could receive the same prefixes from different routers, it must pick an AS that the packets will traverse to reach their destination. Before leaving an AS, the edge router might know different routes to reach the same destination. However, if there is no differentiation between these routes, the non-optimal route might be taken. Hence, to pick the best AS, several attributes are mentioned to achieve this task. The attributes are divided in two major groups :

1. **Well known** : attributes of this group must be recognized by every BGP implementation. Some of these attributes are mandatory and must be announced in all BGP advertisements, whereas others are not necessarily included in every update (these are called discretionary attributes).
2. **Optional** : these attributes might not be recognized by all BGP implementations. Nevertheless, among this class of attributes some of them must be forwarded (resp. not forwarded) to the next BGP peer. These optional attributes are then called **transitive** (resp. **non-transitive**).

We now present some of the common BGP attributes :

Origin It contains information about the way the prefix was learned. It can take the value 0, 1 or 2 if it is learned by IGP, another BGP peer or in another way respectively. The BGP decision process prefers the prefix having the lowest origin value. Then, an IGP route is the most preferred one. This attribute is part of the well-known mandatory attributes.

AS Path It contains a list of the autonomous systems that the prefix has traversed before reaching the edge router receiving the update. Each time a prefix exits an AS, the border router appends to the prefix its own AS number to the current AS path list. The BGP decision process prefers routes with the shortest AS path length. The AS-PATH is a well-known mandatory attribute.

Local Pref A BGP router might decide to prefer a path over another one. For a given prefix, the BGP decision process chooses the route having the higher local-preference. This well-known discretionary attribute is useful if we created a backup

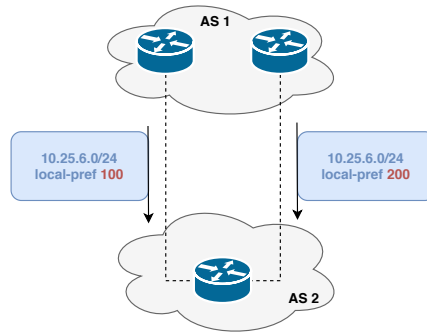


Figure 2.3: Example of ASes having Two physical links

link between two ASes. Consider the Figure 2.3 where AS 1 contains two links with AS 2. Since the AS1 routers advertise the same route ($10.25.6.0/24$) to the BGP AS2 router, this latter will either choose the left or the right interface to send packets having a destination IP within this advertised prefix. If AS 2 would like to use the left link as a backup for every packet from AS 2 having $10.25.6.0/24$ as destination, it could add a lower local preference when it receives routes from AS 1. Hence AS 2 will choose the right interface when forwarding packets. Local-pref is **never** forwarded over eBGP session. This is the local router that appends this value when it receives an update.

Multi Exit Discriminator (MED) Let us consider the example shown in Figure 2.3. We are in the case where AS 1 would like to impose the link taken by AS 2 when it forwards packets destined for $10.25.6.0/24$ to AS 1. To achieve this behavior, AS 1 puts a lower MED value on the right link when it advertises $10.25.6.0/24$. When AS 2 selects the link, it will compare the MED received from AS 1. Since the route received on the left link has a higher MED than the one received on the left, AS 2 will then choose the right link. Notice that the MED comparison must only be done for prefixes received from the same AS. MED is only compared with the same prefixes received from the same AS unless routers are configured to always compare it. The MED is a way to control the incoming traffic inside its own AS. This attribute is optional-non transitive.

Community This optional transitive attribute is actually a list of 32-bit unsigned integers and is divided into equal parts of 16 bits. The first 16 most significant bits are generally used to inform which AS has injected this value and the last 16 least significant bits contain the actual value the originator AS wants to pass to other domains. Thus each value of this list could be represented as the AS number followed by a colon and then the actual value (e.g 145 : 56). Each AS can define its own type of message and have thus 16 bits to do it. For example, a given AS could define the semantics defined in Table 2.1 for some communities it appends to the route [18]. Some ranges are however reserved and have special meanings.

Value	Action on receiver side
ASn:50	Set local preference to 50 (see RFC1998 [19])
ASn:2	Prepend ASn to the AS path list [20]
ASn:78X	Do not announce this prefix to your X neighbor [20]

Table 2.1: Example of action to take when a route from ASn is received

Values from **0** and **0x0000FFFF** (0 : 65535) and from **0xFFFF0000** (65536:0) to **0xFFFFFFFF** (65535:65535) are reserved and have special meaning. For example, the value **0xFFFFFFFF02** (65535 : 65282), defined in RFC 1997 [21], corresponds to the **NO_ADVERTISE** value. It has the global meaning not to advertise the received prefix to other BGP peers.

Next-Hop Well known mandatory attribute defining the next router that packets have to reach as next hop. Each time a route announcement is sent to another AS, this attribute will be set to the IP address of the BGP speaker that has sent the update.

2.2 Type of BGP Sessions

BGP is originally intended to communicate between different ASes. Hence, such a session is called **eBGP** (**e**xternal **B**order **G**ateway **P**rotocol). Inside an AS, the protocol used is different since it does not respond to the same demands as external peering sessions. The IGP is intended to propagate the internal topology to every AS routers. Protocols like OSPF [14], RIP [22] and IS-IS [23] are IGP protocols. If the network is large, BGP could handle many prefixes coming from other ASes such that the set of prefixes is too high to be supported by an IGP protocol. Indeed, protocols like OSPF are not designed to handle a large amount of prefixes. It is not a good idea that every prefix learned from BGP must be first redistributed to the OSPF algorithm that will be flooded next to other routers. Finally, each router of the AS must flood all the prefixes to their neighbors. Although OSPF design simply does not allow this kind of utilization, BGP might be used to solve the problem. Since BGP is designed to handle very large number of routes, each non-border router of the AS could also run BGP besides the IGP protocol to handle external prefixes. In this way, IGP is exempted from this task. BGP sessions between routers of the same AS are then called **iBGP** (**i**nternal **B**order **G**ateway **P**rotocol).

To summarize, on a classical network configuration, each non-edge router of an autonomous system runs two different protocols :

- **IGP** (**I**ntradomain **r**outing **p**rotocol) : to communicate the actual topology of the internal network. Routers follow the shortest path based on the metric defined by the network administrator.
- **iBGP** : to learn best routes chosen on edge routers to contact a prefix located outside the internal network.

Border routers additionally run **eBGP** with other BGP speakers from other ASes so that the local AS has an access to the other prefixes of the other networks.

To avoid network inconsistencies, the best practice is to establish fully meshed iBGP sessions³ between each router of the AS⁴. In this way, each router contains the same actual representation of the network and the problem of network unreachability is avoided. To give an example, Figure 2.4 contains a small network establishing iBGP session on "physical" link only. Routing table for prefix **p1** is also depicted. When router **R1** receives the prefix, it will announce it to its iBGP peers. Unfortunately, local routers do not advertise routes learned from iBGP. If that was the case, internal AS' routers might create loops because of the lack of path knowledge. Indeed, when **p1** is received on **R4** from **R2** and **R3**, **R4** will re-advertise (if iBGP routes are re-advertised to other iBGP sessions) the learned prefix **p1** to both **R2** and **R3**. However, for some reason (e.g. the router ID of **R4** is smaller than the one of **R1**) both **R2** and **R3** choose **R4** as next hop. Nevertheless, **R4** has previously chosen **R1** to reach **p1**. A loop has been created between **R3-R4** and **R2-R4** for prefix **p1**. Packets having **p1** as destination coming from either **R{2,3,4}** will never reach their destination.

Figure 2.5 shows how to solve the problem of unreachability in router **R4**. Now, each router can reach any prefix of the network. Each router contains the same network information. Notice that even with the absence of iBGP link between **R3-R2**, **R4** could have reached **p1**.

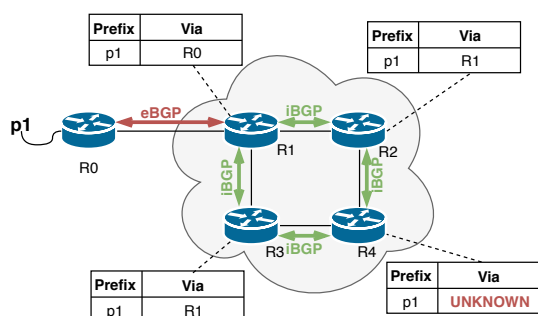


Figure 2.4: Incorrect iBGP configuration

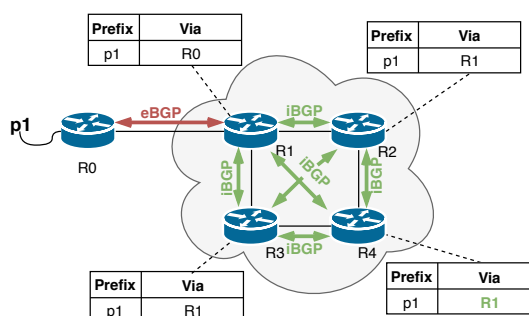


Figure 2.5: "Good bad" iBGP configuration

However, maintaining a fully meshed peering unfortunately first induces configuration overhead⁵. This is error prone and second, a significantly increasing number of routing messages sent inside the AS, leading to a bandwidth overhead when routers are converging

³This is not a full physical mesh, but rather a "logical" one. Since IGP is responsible for discovering the topology of the network, routers are able to correctly forward iBGP packets to the right peer.

⁴If n is the number of routers of an AS, there is $\frac{n(n-1)}{2}$ iBGP sessions at all.

⁵A network with 1000 routers must establish 499 500 iBGP sessions. Those sessions must be manually configured by network engineers.

to a stable network configuration. A solution with route reflectors [24] or BGP confederations [25] reduces the number of iBGP sessions inside the AS. While route-reflection is the common approach, it comes with its own caveats like the lack of route diversity, leading to non-optimal routes, increased convergence time, route oscillation and forwarding loops [26]. Similar problems with AS confederation are explained in literature [27, 28].

2.3 BGP Filters

When a route is received from a neighbor, routers have the ability to control which prefixes are advertised or received from other routers. The protocol executes thus the filters at both the import and the export side of the eBGP sessions. The purpose of these filters is very wide. They could be used to :

- reduce router memory consumption : only a part of the routes could be imported;
- prevent bad network configuration from other peers. Due to errors, other ASes might advertise wrong IP prefixes;
- establish a customer-provider relation : BGP might be used to give access to the network on other ASes;
- improve security to deny certain malicious routes. If an attack is detected from a given prefix, the router could black-hole it to stop the bad network flow.
- ...

Since filters are executed on the edge side of a router, they can also modify route attributes. To take an example, as the **Local-Pref** attribute is not shared between external BGP sessions, the import filter could modify this attribute to match the AS policies.

On the exportation side, BGP could perform AS-Path prepending [18], to voluntarily add the AS which it belongs to or add BGP communities.

2.4 BGP Decision Process

BGP is the protocol used to speak between multiple AS. On large networks such as the Internet, ASes are interconnected which each other such that they offer a plurality of paths to reach a same IP address. Figure 2.1 depicts a small interconnected network. AS 65 524 can reach a prefix located inside AS 65 530 from three links (e.g via AS 65 518, AS 65 523 or AS 65 525). This robustness could prevent network failure and ensures to reach every network nodes most of the time. As BGP routers receive multiple routes to the same network prefix, BGP needs the so-called **BGP decision process** to select the best route (from the router perspective).

BGP Update messages sent by BGP speakers actually announce a list of prefixes the router is able to reach. Inside these messages, BGP maintains the special **AS-Path** attribute indicating to the router receiving it, the list of autonomous systems to follow before reaching the prefix. BGP tries to select the shortest AS-Path length. Therefore, BGP is a "path-vector" protocol. However, besides this selection criteria, in order to give flexibility to network administrators, BGP actually makes its decision according to the attributes received for a given route.

As depicted in Figure 2.6, a BGP router is organized into several layers, each of them is used for particular purposes inside the router :

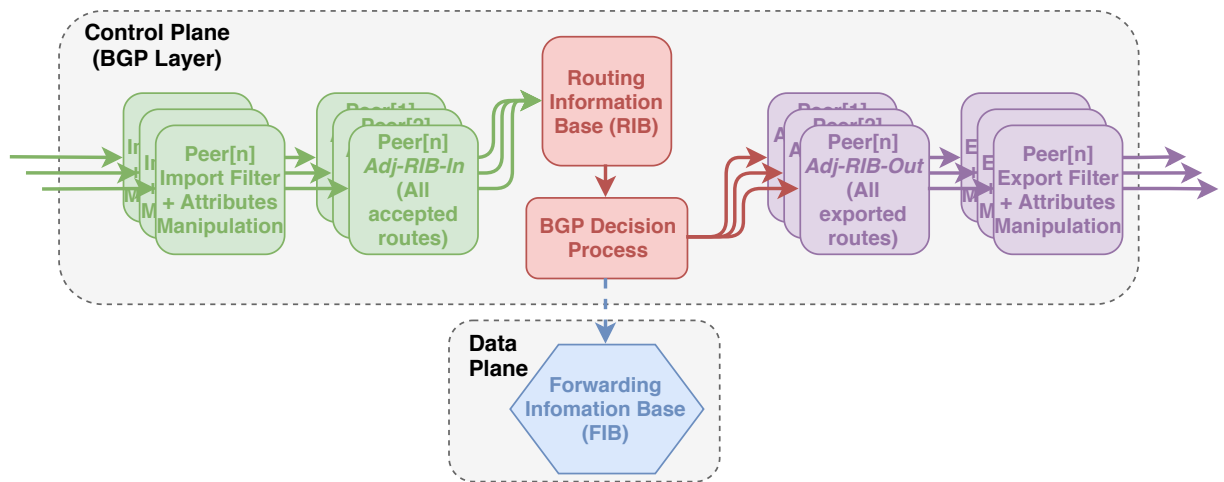


Figure 2.6: Simplified view of a BGP router

ADJ-RIB-In : for each BGP neighbor the local router contains a special table storing the received prefixes and their related attributes. Before importing the learned route, the local BGP router runs an import filter that decides to accept or deny it.

Routing Information Base (RIB) : routes from each **ADJ-RIB-In** table is then moved to the **RIB**. This latter contains all the routes considered as acceptable by the router and might contain several routes to the same prefix.

Decision Process : if multiple routes contain the same prefix, BGP chooses the best one from its perspective. Decisions steps are well defined and will be explained later. The decision process is run each time a route is added or removed from the router.

FIB : contains one route per prefix. This table is the one used to perform packet forwarding.

ADJ-RIB-Out : such as **ADJ-RIB-In**, BGP routers maintain a table for each BGP neighbor. Therefore, they contain selected routes which have been advertised to peers. Before leaving the router, routes are submitted to export filters which decides or not to export the route.

Let us consider for a moment the BGP decision steps that are a critical part of the protocol. To select the best route among those available to a given prefix, it will compare them according to their attributes in the same order.

1. **Local-Pref** : if the route is received from an external peer, this attribute can be modified by import filter. BGP selects the route having the largest local-pref value. If each route has the same value, the next step is triggered.
2. **Local routes** : prefer local routes. Those learned from IGP, statically configured and prefixes aggregated to a single one (e.g. if a neighbor announces both `206.25.160.0/19` and `206.25.224.0/19`, the local only consider `206.25.160.0/17`) are preferred.
3. **Prefer shortest AS-Path** : Route with the shortest AS-Path is selected.
4. **Origin check** : BGP prefers static, local IGP routes, from those learned by external peers.
5. **MED comparison** : prefer routes having the lowest MED value. This value is only compared from routes originated from the same AS.
6. **External check** : select the route received from an eBGP peer. Other types of peer such as iBGP or even others are less preferred.
7. **IGP weight** : select the route having the lowest IGP cost.
8. **Router-ID and Address check**: if none of the previous steps has selected a route, this check compares the router ID and finally its IP address to make the final decision.

Each time a new prefix is imported from peers, the decision process is run again to update the best route for this prefix. In case of route modification, other peers will be informed by first sending a **BGP Update** with the new attribute, and eventually sending a withdraw for the previously advertised route.

Chapter 3

The uBPF Machine

The uBPF machine [4] relies on several concepts that are worth explaining on. The basic idea with is to inject a code into an already running process. The first question that would arise is : *"How is it possible to inject code into another program ?"*. One of the answers is to rely on a virtual machine executing code just as a classical computer. Actually, the uBPF machine is a program simulating the behavior of a RISC CPU taking as input a user code which will be executed to eventually return a result. Hence, its simple description is a program executing another program. As its name might mention, this program receives a bytecode written using the eBPF instructions set.

This so-called virtual machine is simply a set of executable functions that are able to read and to execute code from the process where its API is called. As for JVM (Java Virtual Machine) first introduced to execute code written in Java, this uBPF virtual machine executes a platform independent bytecode to run on every kind of CPU architecture. Hence, the program is only needed to be translated in bytecode to be run on every supported platform. The JVM is well implemented for the three main families of operating systems such as MacOS, Windows and Unix. Actually, a JAVA bytecode produced on a Windows OS can be executed without any modification on a Linux system. The same behavior is reproduced with uBPF. If an eBPF bytecode is written on a machine **A** and given to a machine **B** to be executed on it, machine **B** will be able to execute the program because of the bytecode property.

Instruction Set A classical CPU executes machine code following a given format according to its architecture. Most of the home computers contain a **x86_64** CPU thus programs are compiled to this machine code architecture. The concept is the same with uBPF, the machine reads a program written in eBPF. To ease the execution of the program, each eBPF instruction is encoded into 64 bits and contains the following fields :

- **opcode** (8 bits) : the actual instruction to execute encoded as an integer.
- **destination** (4 bits) : to which register the result of the instruction will be stored
- **source** (4 bits) : from which register the result of the instruction will be computed

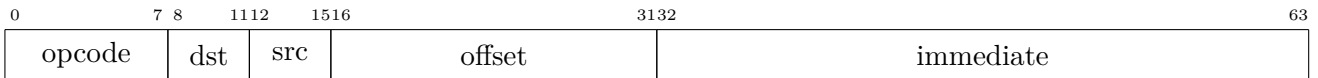


Figure 3.1: Structure of an eBPF instruction

- **offset** (16 bits) : determine an offset. For example, if a source register contains an address n and the offset 64, the instruction is applied to the memory address $n + 64$.
- **immediate** (32 bits) : the source operand, a value that might be used to execute some operation. For example, if a code would like to add 42 to the value contained in a register, the **imm** will contain 42 and the source register the value to which 42 will be added.

eBPF relies on several registers for its computation. Actually, 11 registers are used and have special meanings in the code :

- **r0**: contains the return value of the eBPF program
- **r1 to r5**: used to store the arguments passed to the eBPF program
- **r6 to r9**: so-called "savee" register which contains intermediate computations
- **r10**: is the frame pointer containing the current stack memory pointer

Due to the design of eBPF, it is only possible to pass 5 different parameters to an eBPF program. However, the current uBPF machine was designed to support **r1** as the only argument to be passed to every program running on it.

The uBPF machine is more permissive than the actual eBPF Linux kernel implementation since it allows a program to use five more extra registers to keep internal computations. They only extend the eBPF assembly architecture, ensuring that they remain compatible with the Linux eBPF VM.

The code of the uBPF machine is written in C and contains an API to communicate with it to simplify the load and the execution of an eBPF code. In fact, the current API proposed by iovisor is a basic one. Only basic functions are implemented. Among them, the project proposes :

ubpf_load() : used to integrate eBPF bytecode in the current virtual machine. The code must already be loaded in the memory. There is no helper taking a file path and read a file containing this bytecode;

ubpf_exec() : executes the already loaded code in the specified location in the code. This function takes the specified uBPF virtual machine as parameters;

ubpf_create() : instantiates a new uBPF virtual machine.

At each `ubpf_exec()` call, the function reads the bytecode by parsing every instruction located in it and reproducing the wanted behavior of the actual eBPF instruction on the machine where uBPF is located. This is one of the reasons why uBPF emulates a RISC CPU. The main disadvantage of this interpretation comes from the fact that the virtual machine rereads the eBPF bytecode in memory each time someone is calling the `ubpf_exec` function. Hence an extra overhead is induced due to the lack of direct translation to machine code. However, thanks this method the virtual machine can totally execute runtime checks on what the program is doing in terms of memory. For example, we can check what part of the memory the eBPF code is actually manipulating and stop if it makes some bad accesses. Thus, runtime checks are made easier. Furthermore, since the execution is virtualized, every memory part of a process can be reproduced and is actually handled by the uBPF machine. For example, the initial uBPF machine only contains an execution stack and every "store" instruction deals with this stack. As previously said, if the program tries to access out of this stack, this will be caught. Eventually, the virtual machine dies to avoid unauthorized memory accesses. Notice that only the virtual machine will stop but not the true process in which the VM is called. Because of this memory runtime check, we are sure that the main process will not be stopped because of an illegal memory access.

The uBPF project also contains functions to dynamically compile an eBPF code to machine code understandable by the `x86_64` architecture. This technique is called **Just-In-Time** (JIT) compilation. The code is transformed at run time, when the code is loaded into the memory. Actually, the uBPF API only proposes a translation to machine code for the `x86_64` CPU. Thus, any other CPU architecture such as `ARM` could not take advantage of this JIT compiler.

To achieve this, the uBPF JIT compiler takes each eBPF instruction and transform it to their equivalent on the machine level. The process is only done once at eBPF load-time and takes the freshly compiled code instead of the "raw" eBPF instructions. The produced machine code is stored in a particular location of the process memory since we do not want that a bad memory management on the stack or the heap process could corrupt the code. The virtual machine stores it in a particular location of the current process via the `mmap` system call. Actually, it reserves an anonymous zeroed memory area outside the heap and the stack. This memory is filled with the actual eBPF code translated to `x86_64` machine code. Once the copy is made, the permission of this memory area is changed in order to only authorize the process to read and execute the content inside this allocated space. Therefore any non authorized writes will be prohibited by causing a crash of the process (i.e. the kernel sends a segmentation fault (`SIGSEGV`) signal to the process). If no illegal write access is performed, then the code is simply executed as a normal function taking parameters and returning a value on the execution end. Its main advantage is thus to be directly understandable by the host machine that directly executes the code without having to use an interpreter like the `ubpf_exec()` function. Another advantage of the JIT compilation is that uBPF can optimize the instructions inside the bytecode before executing the machine code instructions. A further optimization procedure can be used to better produce the machine

code on the target CPU architecture. The JIT can for example directly transform stack manipulation into registers if the analysis of the bytecode determines if it is better for a question of performance.

After the JIT compilation, the code is directly executed by the processor and thus there is no extra parsing by any interpreter such as `ubpf_call()`. However, we lose the virtual execution context and the run-time memory analysis. Indeed, the eBPF code is directly using the stack of the process and thus more generally the direct memory of the program. In addition, the code could completely access to process variables that were previously not available thanks to the virtual environment. Thus, it can be noticed that the main process might crash because of an invalid memory management. Consider Listing 3.1, a simple C code that could represent a small translation of what could be inside an eBPF code.

```
1 int *evil_ptr = NULL;  
2 *evil_ptr = 42;
```

Listing 3.1: Simple pointer dereferencing

This program is intentionally given to the virtual machine to be JITted. When the program is executed, the CPU first assigns to the pointer variable a `NULL` address, then it immediately dereferences this pointer to store 42 inside the memory area pointed by `evil_ptr`. However the `NULL` address corresponds to an invalid memory area in the process that will crash after receiving a `SIGSEGV` signal. Therefore, JITted programs are not memory safe because of this lack of run-time checks.

On the contrary, the virtual execution of the eBPF bytecode could spot the illegal pointer dereferencing without having any crash of the whole process execution.

Both methods have their own advantages and disadvantages. The interpreted mode gives more flexibility in terms of execution control but uses more instructions because of the rereading of the bytecode on each eBPF execution. The JITted mode provides fast execution of an eBPF code, but cannot verify every write access to the main memory and does not provide a sufficient enclosed execution.

The virtual machine can only read and execute eBPF bytecode. However, writing a program in assembly could be quite cumbersome and is error-prone. This is why the `clang` compiler is able to convert traditional C code into eBPF bytecode. It generates an ELF object file containing every information to run the code when it is passed to a program capable of executing eBPF formatted bytecode. Thus, the programmer does not need to worry about using assembly language directly. Nevertheless, the generated code is stored in an external file that needs to be loaded in the eBPF virtual machine. The loading part must be handled from the program where the VM is located since the current uBPF version is not able to handle this part. Figure 3.2 depicts all the steps executed in order to load a custom code into the process integrating the uBPF code. The blue part of this schema is actually carried out outside the process. A user writes a simple C code which will be translated to an external file and then loaded into the

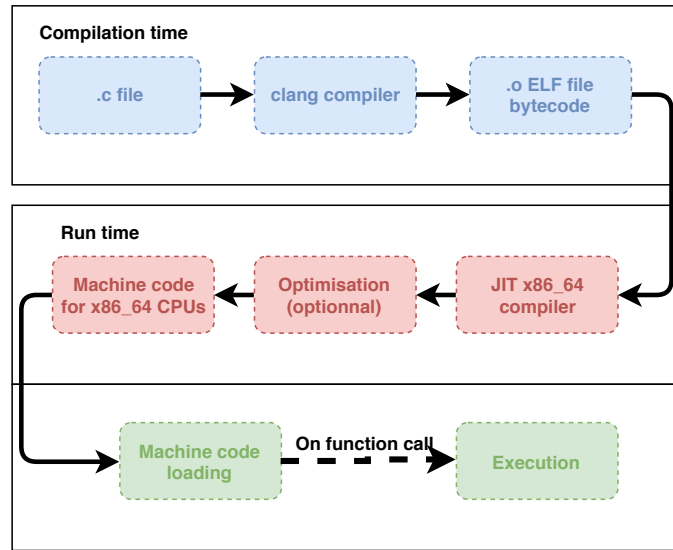


Figure 3.2: Different Steps Achieved to Inject a Plugin

process to actually perform the JIT compilation and eventually the execution at some points of the program.

3.1 Writing eBPF C Code

As previously explained, `clang` enables a programmer to directly write C code instead of understanding how to use the eBPF assembler. In fact, there are some limitations on the way to write code that will be passed to the VM :

1. there must be only one function written inside a single `.c` file containing the entire program to be executed;
2. the principal function name is not taken into account when the ELF file is loaded into the VM. Therefore, any name can be chosen;
3. this function must only take one parameter as arguments and is actually a pointer to the argument memory zone. Since we have a pointer, the actual arguments are not constrained and could be anything. As the user is dealing with the C code, this argument might be a pointer to a structure. Hence, we get rid of this single pointer argument and an eBPF program can access every kind of data. This proposed technique is similar to the one used to pass arguments to a POSIX thread;
4. it is not possible to call functions from libraries inside the eBPF C program. However, there is actually a special way to call some predetermined functions inside eBPF code. Section 3.2 details this particular case. However, if the user absolutely wants to write another function that will be called inside the main eBPF program,

this latter needs to be written with the `static inline` function specifier. This tells the compiler not to create a second function but to insert the instructions of this inlined function to the location where they are originally called in the main eBPF C code (i.e. the function call is replaced by its actual instructions in its body). If it is not possible to call another function, it comes from the fact that the execution must be as enclosed as possible. Letting an eBPF function make any calls they want means that they can access to every kind of variable from the main process. This goes against the real purpose of integrating such virtual machine;

5. the allocated memory for the eBPF code when executed is small. Hence, the code should not allocate large data structures;
6. pointers dereferencing must be done on the memory bounds of the allowed VM memory area. If any illegal memory access is noticed, the interpreter will stop the execution of the eBPF code;
7. there is a limit on the number of instructions that an eBPF program contains. Actually, this constraint is only a constant on the uBPF code and could be increased if a bigger eBPF program must be loaded;
8. no infinite loop must be produced. Indeed, we do not want to freeze the process where the uBPF machine executes the external eBPF code;
9. it is not allowed to call other eBPF programs inside the main routine of another eBPF code. This restriction results from the inability to call any function otherwise than using registered external functions.

This list of constraints is required not to harden the execution of a given eBPF program and keep the virtual machine as simple as possible. These choices also keep the lightest implementation possible inside the main process.

Some problems might arise if we want to use multiple eBPF bytecode. Since a single uBPF machine created with `ubpf_create()` stores one eBPF program only, multiple uBPF machines must be created. To avoid manually calling the same functions for the creation and code loading, a higher level interface must be created to create the VM and load bytecode in a single call. More generally, manipulating the virtual machine should be simpler than the current "low level" API.

One attempt to build a higher level API is to have a function `add_bytecode()` that takes a path to the bytecode generated by a clang compiler and loads it to a new uBPF machine. Then the new created uBPF machine is inserted in a larger data structure containing already started virtual machines. In fact, to be able to retrieve a particular instance, every machine is loaded in a hash map. By using a "key-value" data structure, it is relatively easy to access every machine everywhere in the code. By storing the eBPF function manager (the previously mentioned hash map) on the global block of a UNIX process, the whole BGP protocol can access this structure containing every eBPF

function. Thus, the manager is basically composed of a hash map data structure from which every byte code will be stored. It is also a way to centralize everything related to the "eBPF aspect" of the protocol and to control which part of the protocol already contains bytecode. In this case, if a user wants to do that, the manager will refuse to load the second bytecode. Finally, the administrator must manually disable the previous eBPF bytecode and retry to load the new one.

In order to access a particular eBPF program, a given set of unique identifiers must be created. Because the virtual machine is executed at some strategic points of the code, the plugin manager needs to pick the right eBPF code. When creating an instance for a new program, the ID to choose is then an identifier of the location where the program will be executed inside the BGP protocol.

3.2 External or Helper Functions

The eBPF bytecode instructions set provides the ability to call other functions defined outside the scope of the written code using the byte code `call` instruction. These, also called helper functions, are executed on the same level as the code of the virtual machine and hence have the right to access to every memory address they want. To make a comparison, the principle could be contrasted with a system call in the Linux kernel. A given program calls a function that is executed with an elevated privilege. Since the virtual machine is executed inside the main program, external functions accessed by a plugin are directly executed inside the program code. Hence, it can access the whole process memory and every function defined and written for the original program.

What we call an external function is any part of the code that is not compiled inside the plugin but is still used inside it.

With uBPF, the same behavior happens. A set of functions is defined to perform a given task according to the argument received by the eBPF code, and returns its result to the caller. To be exploited, these external functions have to be registered onto the virtual machine before executing any plugin. Otherwise, the virtual machine refuses to load the program because the `call` instruction refers to an invalid helper. The uBPF API proposes a function `ubpf_register` taking four parameters :

1. the virtual machine on which a new external function is registered;
2. a unique ID number corresponding to the external function;
3. a string corresponding to the function name. This string is used when parsing an object file generated by `clang` (or any eBPF bytecode compiler);
4. the pointer referring to the function memory address.

The second parameter is a number used to inform which function has been called when a VM executes the bytecode. Listing 3.2 provides an example of a helper function at line 4 of the C code. After setting registers containing the three arguments (i.e. `r1`, `r2`, `r3`), a call to the external function, performed at line 7 of the disassembly text. This opcode

(`call`) takes as operands a 32-bit unsigned integer which is actually the ID registered to the uBPF API. For implementation reasons, the internal uBPF code keeps a threshold limit on the number of allowed registered functions for a single VM instance. In the current version of the uBPF machine, only 128 external functions can be passed to it. To keep it really simple, authors of the uBPF API store every registered function in a single array of 128 elements. The ID number passed on the arguments of `ubpf_register` is actually the index of this array.

On Listing 3.2, it can be noticed at line 7 that the `call` operand is equal to `-1`. Since the eBPF bytecode originates from the `clang` compilation of the above C program, the compiler actually does not precisely know which registered function it should call. Remember that functions are registered at plugin load time. The `elf` produced file by `clang` will then be rewritten to modify the operand of each `call` instruction to the right helper. This is done during the relocation phase, the uBPF API looks into the symbol table for the external function name to find which identifier it refers to.

```

1 int main(int *bgp_port){
2     bgp_test_t d;
3     d.curr_timer = 9999;
4     send_to_monitor(&d, sizeof(bgp_test_t), BGP_TEST);
5     *bgp_port = 14785;
6     return *bgp_port;
7 }

```

Disassembly of section .text:

```

<main>:
    0:      bf 16 00 00 00 00 00 00      r6 = r1
    1:      b7 01 00 00 0f 27 00 00      r1 = 9999
    2:      63 1a f8 ff 00 00 00 00      *(u32 *) (r10 - 8) = r1
    3:      bf a1 00 00 00 00 00 00      r1 = r10
    4:      07 01 00 00 f8 ff ff ff      r1 += -8
    5:      b7 02 00 00 04 00 00 00      r2 = 4
    6:      b7 03 00 00 01 00 00 00      r3 = 1
    7:      85 10 00 00 ff ff ff ff      call -1
    8:      b7 01 00 00 c1 39 00 00      r1 = 14785
    9:      63 16 00 00 00 00 00 00      *(u32 *) (r6 + 0) = r1
   10:      b7 00 00 00 c1 39 00 00      r0 = 14785
   11:      95 00 00 00 00 00 00 00      exit

```

Listing 3.2: Example of external call and its translation to eBPF bytecode

From this small example, one might see that an external function is just a simple function that is executed without any context such that they could be called by any eBPF program from everywhere. Thus, these helpers must be designed in a way to handle each possible case to avoid crashing the whole process. Indeed, since the eBPF program could execute code outside the isolated context, the main difficulty is to design external functions as safe as possible. Furthermore, the provided uBPF API is written in C, as external functions. Which means that it is impossible to capture any context when functions are registered. In other words, the C language does not provide a flexible interface to perform closure. Some alternatives must be found to avoid exposing internal execution context to the user when he wants to call a helper function.

3.3 Passing Arguments To The eBPF Program

On the traditional eBPF design, it is possible to pass five arguments to the main eBPF function. This is because of the assembly architecture. Recall that registers `r1` to `r5` are used to store the arguments passed when the code is started or when an external function is called. Unlike a `x86` assembly code, eBPF does not rely on the stack to pass and retrieve arguments but relies on the "same" mechanisms as system calls as seen before. In a `x86` architecture, syscalls also relies on CPU registers to call the right handler inside the kernel. The system call identifier is copied inside the `%rax` register and the six other registers are used to pass the arguments [29]. Unfortunately, the uBPF VM is not able to handle the full five registers when a code is being executed. However, only one (`r1`) is used and actually understood by the virtual machine. Furthermore, it considers that the argument is a pointer to contiguous memory zone where the actual argument is located. This restriction comes from the fact that the virtual machine, in interpreted mode checks every memory access in both the arguments and the stack. Therefore, to ease the memory verification, uBPF imposes to give a pointer and the total size related to the memory area where it points to. Nevertheless, there is no restriction on the data contained inside this argument block.

3.4 Calling the Virtual Machine

Until now, the current implementation of the virtual machine is quite straightforward in the sense that it provides an API to execute one eBPF byte code. The created environment is intended to execute the code in its own memory area, including a stack, a context capturing the arguments passed to the eBPF code and some extra memory pointing to a predefined user's location. During the interpretation of the instructions, the task of the uBPF machine is to ensure that every memory access made by the user program is legal (i.e. it tries to access within its memory bounds). This avoids corrupting the main process memory. However, a plugin could totally corrupt its dedicated memory without any effect on the process execution. We thus have everything to run a particular plugin.

Steps to build an instance of the uBPF virtual machine are composed of four actions :

1. load the eBPF bytecode into memory, so that uBPF VM could access the instructions. This must be done either by writing a C code compiled in eBPF thanks to the `clang` dedicated compiler, or by writing eBPF assembly directly;
2. create a new instance of an uBPF machine. This freshly initiated virtual machine actually contains one emplacement to execute one eBPF function;
3. choose the VM execution mode (i.e. Interpreted or JITted);
4. if the user chooses the JIT mode, it must explicitly compile the code, to eventually execute the program as if it was simply a newly created function.

Note that an eBPF code could be interpreted even after a JIT compilation. This leaves flexibility to the user by choosing both modes for the same plugin.

Inside the execution of a plugin, some external calls can be made to registered functions. As previously said, an uBPF machine is as isolated as possible. Therefore, it does not have any right to access the `glibc` for example. It must explicitly load the functions that a plugin is allowed to use. As the main purpose of some eBPF byte code is to send data to a monitoring server, the virtual machine must provide a way to upload the collected data. Hence some functions are built and registered to the virtual machine (every plugin could access these external functions). This created API will be recognized by the plugin and will execute the code associated with the registered function. In the monitoring case, plugins will use a special registered function that sends data through an UNIX socket to another thread or process.

3.5 Conclusion

We presented the basic functioning of a byte code execution with a virtual machine. uBPF is the first step to improve flexibility of a protocol since it allows executing bytecode inside a process. However, its current design is still limited and is not ready to be fully applied to a real protocol implementation. The uBPF project contains the interpreter and the JIT compiler, which is not sufficient enough. For instance, uBPF does not provide the ability to inject on-the-fly eBPF functions when BGP is running. Also, the injection of an ELF compiled file is not straightforward yet. Thus, the first part of our work is to improve uBPF before plugin it inside the BGP protocol.

Chapter 4

uBPF Integration

The previous chapters explained the global architecture of the current implementation of the virtual machine. The first step was to acquire the user-space eBPF machine. However, uBPF contains some flaws that must be corrected before inserting it in a real BGP implementation. Let us focus our attention on several issues to solve :

1. the current uBPF API is too low level in the sense that it provides functions to interact with a VM instance only. The main purpose of integrating the VM is to replace some already implemented behavior in the code to completely override an entire function inside the original code, for example. With the current uBPF version, it must first load the eBPF bytecode, then create a new instance of the uBPF machine, inject the ELF code and eventually execute the code with `ubpf_exec`. The task is repeated each time a new eBPF code is inserted. The procedure is quite repetitive and thus we would like that the load is made with only one API call instead of multiple ones. In addition, if we want to insert multiple instances of the virtual machine (e.g. to override multiple functions in the program), the current logic of the API is quite difficult. The "manager" must be created from scratch and must be part of the BGP code. This is not the expected behavior. The uBPF API must handle these cases such that the BGP part must only make calls to the API without adding extra overhead inside the BGP source code;
2. Section 3 explained two types of byte-code executions, that are interpreted or JITed. The first mode provides more flexibility to perform memory checks than the last one. Therefore, we need to find a way to integrate this memory checker when the eBPF code is executed. Otherwise the program could crash during the execution of the eBPF code;
3. real programs like the implementation of the BGP protocol deal with large structures capturing a lot of information concerning the connection context like information about each neighbor. Inside the program code, functions use these large structures and most of the time does not really need the entire variables in order to be correctly executed. To ease the insertion of the virtual machine, the arguments of

the eBPF functions are the same as the original one (i.e. the BPF function where the bytecode is executed). However, the actual version of the uBPF machine might expose the true variable inside the protocol, due to the design implementation of how the variables are passed to the VM. To limit programming errors, eBPF functions must have a read-access only to these structures. Thus, creating a higher level `ubpf_exec()` like function that can handle either the interpreted mode or the JIT by setting these variables in read-only status;

4. it might be the case that some function wants to modify some BPF variable. If everything is made not modifiable inside the enclosed context of the virtual machine (see previous point), our implementation must provide a way to control the modification of variables. The same reasoning could be made for variables that are not accessible by the arguments of the uBPF function;
5. by default the uBPF machine only allocates a stack to store the intermediate results of the eBPF code. This memory area is limited by a constant of 2048 bytes. The implementation stores this memory in the process stack. This explains why the allocation is very limited;
6. there is no mechanism to add extra memory to enable data to persist between two eBPF calls. Since an eBPF call could be considered as a "simple function call isolated from the main process", the eBPF execution could not store any variable in the global section of the memory program (i.e. using `static` keyword for a variable inside the eBPF program will not keep it alive on the next execution). The stack is the only memory area accessible during the eBPF execution. One might simply augment the stack allocation to allow a plugin to store more intermediate results. However, in the current implementation of the uBPF VM, the eBPF stack in interpreted mode, as well as in JIT, is allocated on the true process stack. Most of the time, the stack allocated for a program is limited in size. For example, on a machine having a `x86_64` CPU architecture that runs on a Linux kernel 5.0.12, the system reserves 8196KB of memory¹;
7. while the concept of eBPF external functions is a powerful mechanism allowing the eBPF call to talk with the main process to retrieve some information not allowed inside the virtualized execution, it might be the case that they could crash the protocol if no checks are performed on the input received by helper functions. A simple example is to pass a `NULL` pointer to the external function. If no sanitization is performed, the process will terminate its execution because of the `SEGFault` signal sent by the kernel.

The sanitization must be as accurate as possible to avoid that an eBPF function requests for unauthorized data by manipulating the pointers passed to the external function. For example, we do not want the modification of the local preference of a

¹The actual default value could be found by executing `'ulimit -s'` in a linux shell

route received from our neighbors while the actual goal of this eBPF insertion was originally to add new BGP communities;

8. we would like to handle multiple plugins. The supplied API is not very relevant to add or manage multiple plugins in the existing BGP implementation. If two eBPF plugins are created, it is the code of the protocol itself that must handle their uses. The API has to reverse this tendency and has to handle in an easy way multiple eBPF functions;
9. we would like to dynamically inject plugins when the protocol is running. Currently, the uBPF version does not provide an API to make this possible. This mechanism ought to be able to add, replace or delete virtual machines that emulates eBPF functions, already injected into the protocol;
10. ideally, we would like a kind of global manager accessible from everywhere in the project that can execute multiple eBPF functions but also listen from incoming eBPF byte code to dynamically insert/remove plugins;
11. in order to execute the plugin, it is needed to pass a memory area which will be accessible inside the plugin. Hence it can directly modify the protocol variables if the plugin is created in a bad way ! Thus, a way to protect protocol variables from plugins errors is to copy every needed variable before passing them to the plugin. Sometimes, this duplication technique is unavoidable if multiple variables must be passed to the uBPF plugin. Indeed, since the code is executed inside its own stack, which has a limited amount of space (2048 bytes), pointer related to the variable could be out of its bound. As a result, the plugin will crash. Let us consider the structure depicted in Listing 4.1 having pointers to BGP variables. If

```
1 struct args {  
2     struct peer *ptr; /* real structure used by bgp */  
3     int *pointer;  
4 };
```

Listing 4.1: Simple structure that could be passed to the uBPF plugin

this structure is passed to a plugin, obviously it will fail since pointers are associated to a memory zone outside of the uBPF VM space. Actually, an eBPF bytecode cannot access to memory addresses not belonging to its stack. We need to find a way to copy variables into this memory stack. Hopefully, it is possible to invoke external functions. A description of how these special kinds of functions operate is in Section 3.2.

As explained before, the only way to give several variables to the eBPF function is to create an intermediate structure containing wanted data, just like passing arguments to POSIX threads. The constraint of passing only one argument comes

from the fact that the uBPF project allows passing one memory variable to the code it will execute, even though a classical eBPF program has five registers dedicated to arguments;

12. one method to pass variables to a eBPF function is to pass a structure containing pointers to some variables of the protocol. Nevertheless, these pointers are not accessible by the bytecode since uBPF checks that a pointer dereferencing is made inside the bound of its memory (i.e. the stack and the context of the arguments). Each instructions contained in the code is allowed to use this stack only. This is the desired behaviour, since we do not want a BPF script to cause the crash of the whole BGP process due to an error (e.g. accessing to the **NULL** address, overriding memory by making buffer overflow, etc.). A simple way to get rid of this constraint is to use an external function. Consider the example of the eBPF function of Listing 4.2.

```
1 typedef struct args {
2     int *my_int;
3 } args_t;
4
5 uint64_t get_external_memory(args_t *a){ // (1)
6
7     int data; // (2)
8     load_mem(a->my_int, &data, sizeof(int)); // (3)
9
10    // computation with 'data'
11    return 0;
12 }
```

Listing 4.2: eBPF plugin accessing to external memory

In (1), the argument is a structure containing a pointer to an integer pointing to a memory area not accessible by the eBPF bytecode. Variables (in this case, the argument structure) passed to the `ubpf_exec` function are first copied inside the virtual machine, then the eBPF code is executed in its own context. Here, the real value passed to the plugin is a memory address pointing outside the eBPF stack. The only way to access the variable is to invoke an external function. Therefore, the instruction in (3) will take a variable declared in the memory stack accessible by the program (2) and will copy the content of the BGP variable in `data`.

`load_mem` is an external function that will be executed inside the BGP protocol, at the same level as `ubpf_exec`, `ubpf_load`, etc. This explains why it can access to non-allowed memory from eBPF plugin. However, this naive version that accesses to the main process memory is too weak. It does not prevent bad user inputs. Indeed, the user could use a fake memory address if no check is performed at the level of the external function, the main process will probably crash due to a segmentation fault.

Besides this security aspect, to avoid any mistake from the uBPF plugin, this external function copies the memory zone inside the eBPF stack. Thus, the variable is accessed in a read-only mode, which prevents the plugin from overriding the protocol variable. This function is then too weak to be used by any eBPF code, and thus, must be transformed to be a getter of a specific variable. This latter will be restricted to only one kind of structure in a specific location of the BGP protocol.

13. Before loading any eBPF program, the virtual machine is checks the validity of the bytecode. This checker verifies if :
 - it contains at least one exit instruction (necessary to stop the virtualized function);
 - all instructions are correctly formed. For example, to load a 64-bits number, it requires two eBPF instructions. The first 32 bits parts are stored in a first instruction, and the last 32 bits into a next instruction (these both parts are on the `imm` section);
 - it does not make infinite loops (or at least the eBPF jump is not referring to the previous eBPF instruction);
 - each ID passed to the call instructions refers to valid external functions;
 - is not doing a division by zero;
 - source and destination operands refer to valid registers.

The eBPF verifier thus analyzes the basic formation of the eBPF program but is not sufficient to detect any bad utilization of memory such as NULL dereferencing. Ideally, the verifier must quickly walk to every possible paths of the bytecode to detect some flaw and refuses to load the code if some instructions generate errors.

It can be noticed that the current iovisor implementation uBPF is a strong base to run other programs in a program. However, as shown previously, the number of problems is quite large and should be solved to avoid corrupting the base program where the VM is integrated. Currently, if we would like to directly embed this VM into the BGP implementation, the total amount of work would be too high and the solution could not be carried on another network protocol (or more generally to other programs). This is why it is necessary to create a strong API to handle eBPF plugins. Ideally, we would like to have a mechanism to allow a function to be "automatically reprogrammed" in order to insert every needed call to the uBPF API. Every functions like that becomes an insertion point that could be modified in such a way defined in Listing 4.3. In fact, `DEF_VM_FUN_CALL` is actually a macro in charge of the creation of the function defined by the name `function_name`, taking as arguments "`arguments`" (e.g. `(int c, char[] string)`). Hence, the modification on the BGP side is now relatively simple since a macro does everything. Indeed, its arguments take every part of a normal C function. There is, however, one argument to discuss, the `pluginID` one. As the final objective is to put multiple insertion points, this argument is used to distinguish multiple plugins. We thus

```

1 DEF_VM_FUN_CALL(function_name, (arguments), pluginID, {
2     // code of the original function
3 })

```

Listing 4.3: Modification of a function to directly include calls to the uBPF API

need to provide a kind of unique identifier on the location where eBPF instructions will be executed. Every time a new plugin is created, it must specify which ID this plugin will be part. It will provide an indication to the user that this plugin will be executed in this particular location of the code.

4.1 Structure of a Plugin

The uBPF virtual machine is intended to execute a given eBPF byte code. This machine provide an API to either interpret or JIT the eBPF bytecode and returns the computed result of this hook. This is the task of this work to leverage uBPF to execute some plugins at multiple critical location of the BGP implementation. Indeed, uBPF will be introduced in some existing functions doing different jobs during the BGP execution. One plugin is associated with one BGP function, i.e. an insertion point, and is subdivided in three different parts called anchors, having different purposes :

PRE functions : before executing any instruction of the original function, some PRE functions could be executed.

REPLACE function : this special function is intended to completely bypass the existing implementation of the function. The user could completely rewrite its own instructions to achieve the task of this function.

POST functions : before returning its results, the function executes the last functions which are part of the plugin.

It might be noticed that PRE and POST parts of the plugin could contain several eBPF hooks and not just one. Since these functions do not modify any part of the protocol, they have a read-only access and thus can be used to store information about the normal execution of BGP, prepare the replace function or even cleaning used memory. However, the replace function is the only subpart which could not receive more than one eBPF code. This restriction is due to the fact that it may modify the protocol itself, or at least its variables. If two **REPLACE** plugins modify the same protocol step, it may lead to an undefined behaviour for the rest of the BGP process execution, and thus it could completely break the normal BGP behaviour.

Each **PRE** and **POST** functions are stored in a key-value data structure, which means that every eBPF script is associated with a name that uniquely identifies them. This

choice is mainly to avoid duplicating the same code inside a plugin and to ease the dynamic deletion of a given eBPF hook. In fact, without any name, or at least an identifier associated to a function, a network administrator could not remove on the fly the concerned code without rebooting the BGP process. This latter method is too restrictive, because BGP sessions and routing tables must be recomputed from scratch.

Besides the insertion point grouping the three anchors, our implementation allows to put anchors on some part of a BGP function. If this one is critical for the correct execution of the BGP protocol, we do not want to add an **REPLACE** anchor inside it. Our higher level API is capable to support a combination of these placeholders. If no **PRE** anchors are found inside the BGP function, the virtual machine skips this anchor. The figure 4.1 shows the way a plugin is executed inside a "pluginized" BGP functions. At least the original part of the function are executed if no plugin is associated with this plugin.

4.2 Plugin Manager

We have seen in the Section 4.1 that an eBPF bytecode is executed in a single entry point in the protocol. Namely, inside an already implemented function and precisely at the top (**PRE**), the body (**REPLACE**) or the bottom (**POST**) of it. Since this is an entry point which could happen from everywhere in the process, it is needed to have a manager to orchestrate the execution different plugins. To simplify the management of plugins, such as eBPF functions, all plugins are associated with a unique identifier in the form of an integer and are stored in a hash map. Hence, in any part of the BGP implementation, it is required to only mention the plugin ID and which part of the plugin to run. `run_pre_function` is an example of a call to the plugin manager in charge of executing every pre eBPF function.

```
1 int example_pluginized_function(struct_t *a, ...) {  
2     run_pre_function(ID_FUNCTION, MEM_ARGS, LENGTH_MEM_ARGS);  
3     // function body  
4     // ...  
5 }
```

Listing 4.4: Example of plugin manager call

The reasoning is quite similar to the other calls. In fact, the API is composed of three functions, in charge of running the right type of eBPF function (i.e. executing **PRE**, **REPLACE** and **POST**). Because it could be cumbersome to write the exact same pattern for every function to pluginize, a macro is defined to pre-include this integration. To transform any function to a "pluginizable" function, the macro takes several parameters : (1) the function return type, (2) the function name, (3) the ID associated with the plugin which will be interpreted by the uBPF virtual machine, (4) the body of the function, executed only if no **REPLACE** functions are injected and (5) the argument to be passed to

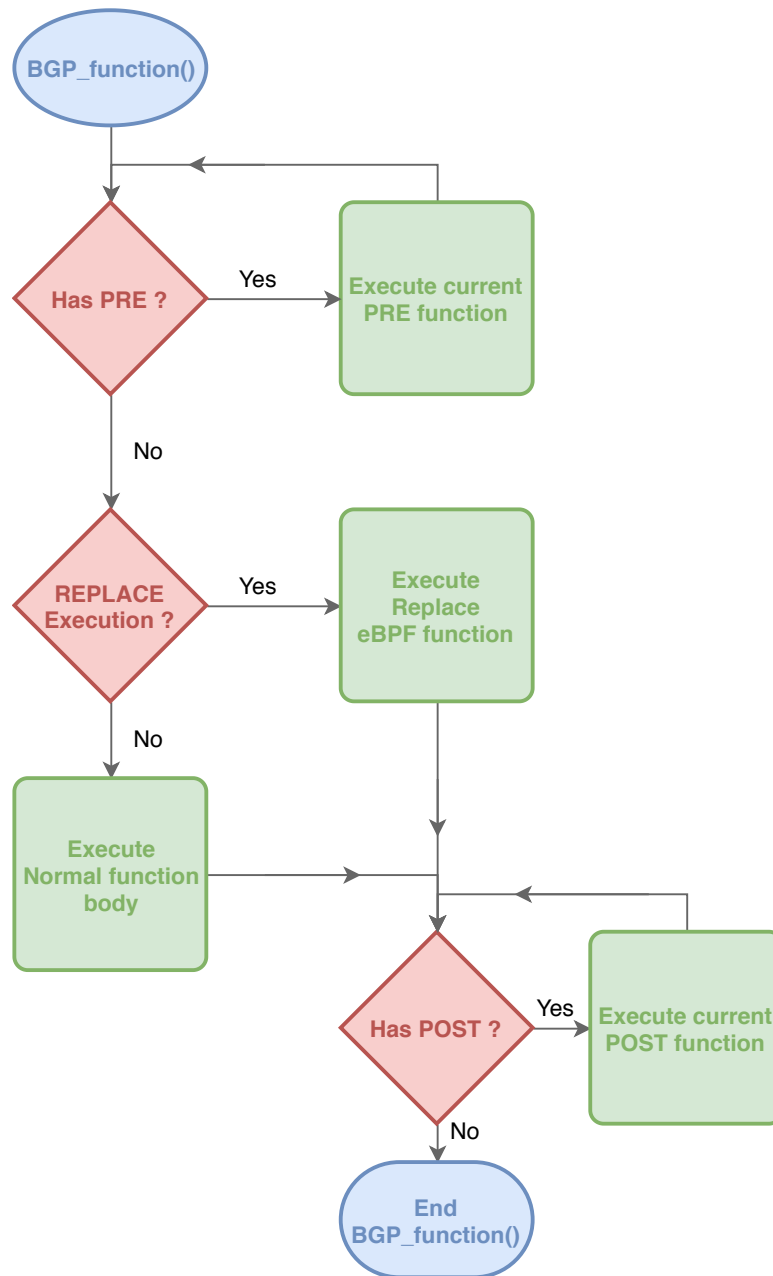


Figure 4.1: Execution flow of a pluginized function

every eBPF program. Notice that eBPF arguments are a subset of the argument of the real function. No other arguments can be passed on the virtual machine.

If an eBPF code is dynamically added, the plugin manager is in charge of converting any human readable plugin identifier to its integer representation, before adding it to the manager. This is also valid when removing a plugin, the API contains a one-to-one mapping between the integer and string (human readable identifier).

4.3 A mechanism to inject plugins into the implementation

We would like to add a mechanism to dynamically insert or remove uBPF plugins from the program without stopping it. Hence the addition or removal must be done while BGP is running and is communicating with other BGP peers.

To achieve this task, we need to include an independent thread inside BGP which listens to messages from an external process executed by a given user. Once a message is received, part of the program responsible for managing uBPF plugins reads the message and perform the action contained inside it.

These messages are sent thanks to a command-line interface (CLI). Actually, each time a user wants to interact with the virtual machine, it must execute a local program (i.e. inside the router) that will interpret the command. Ideally, on a future version of the prototype, the injector will be directly integrated to the FRRouting CLI.

The plugin manager and this external program interact through message queues and shared memory. In detail, when the user executes the program specified on Listing 4.5, the "injector" is in charge of copying the eBPF bytecode located to the specified path into the shared memory accessible by these two processes. Once the binary has been copied, the process emits a message into the queue (which is also accessible by both BGP and the injector), which notifies the BGP daemon that a new message from the injector has been emitted.

```
1 $ ./ebpf_injecter -a 'add_pre' -l 'insertion_point_id' -p  
   '/my_eBPF_bytecode_path'
```

Listing 4.5: Example of CLI for plugin injection

Once the message has been sent, the plugin manager, who was listening for incoming messages on this queue, performs the action. In this example, it adds in the PRE anchor (-a argument) the plugin into the location (-l argument) specified by the user by reading the shared memory (-p argument). When the operation is completed, the plugin manager sends back a confirmation message to the caller (the external "injector") to inform whether the operation succeeded or not. This notification is sent through the same queue and is an indication to the user of the completion of the task. The message is human readable, thus it could resolve the problem if the protocol answers back with an error.

The -a argument is the action the user would like the manager achieve. On Listing 4.5, the

action is to add the specified plugin on a PRE anchor of a given insertion point specified by the plugin identifier (in `-l`). Other actions are handled such as the **replacement** and the **deletion** of an eBPF function inside a plugin in a specified anchor (e.g. "delete this eBPF code from this plugin determined by this identifier").

Imagine a network engineer who has implemented multiple eBPF functions which are not inserted inside the protocol yet. Using the CLI is however, not user friendly enough in the sense that it must execute n times the CLI command to load these n eBPF plugins. This is subject to errors and, furthermore, if BGP already contains m plugins, the next time the protocol is reloaded, it must re-execute the command $m + n$ times. We thus provide the ability to specify a **JSON** formatted directly to the CLI.

```
1 {
2   "jit_all": true,
3   "plugins": [
4     {
5       "path": "/filter_max_routes.o", // path of the elf code
6       "extra_mem": 512,              // extra memory heap size
7       "id_plugin": "bgp_input_filter_apply_proc", // identifier
8       "type": "bpf_append_pre",      // action (add to PRE anchor)
9       "name": "filter_max_route",    // eBPF function name
10      "after": "filter_test"          // after which other eBPF
                                     function it must run
11    },
12    {...},
13  ]
14 }
```

Listing 4.6: Example of a formatted JSON to inject multiple eBPF functions at once

Listing 4.6 shows an example of a **JSON** which automatically reproduces the same behavior as the CLI line shown in Listing 4.5. A user directly specifies this type of file to the router. This latter injects all the eBPF ELF files in the BGP implementation and accordingly informs the user if the procedure has been successfully completed or not.

It is also possible to statically include some plugins that will be loaded when the BGP daemon is started. It is useful if some plugins modify the way to contact other peers for example (e.g. modify the open procedure or monitor the number of **BGP OPEN message** sent). Indeed, by using a dynamic approach (i.e. loading uBPF hooks after the BGP daemon is started), the initial connection may already have been sent. Therefore, injected plugins could never be executed for the already contacted peer.

By using the proposed API, the protocol could retrieve bytecode located on the same machine by mentioning the path in the file system. This procedure is launched before executing any BGP protocol instruction, the mentioned plugins in a JSON formatted file as explained above will be fetched and loaded to be injected somewhere in the protocol.

This JSON file is part of the BGP FRRouting configuration and is read at the same time as other static FRRouting configuration file are loaded.

The feature of dynamically injecting plugins is restricted to the local machine where the protocol is running. Indeed, limiting the plugin injection to the local router forces the administrator to first connect to the router and then injecting a plugin. This choice is to avoid that a malicious plugin is inserted without the consent of the administrator to completely corrupt the protocol. Indeed, the virtual machine does not contain a mechanism to verify the integrity of an eBPF plugin yet. Although some small checks are done by uBPF, those are not sufficient to guarantee the soundness of the eBPF code. Therefore, the router administrator only could manage which plugins to load/unload by accessing onto the machine. It forces the administrator to manually check the plugin to check if no strange behaviors are found like an infinite loop, corrupting some variables or even replacing a function of the protocol with an empty no-ops function.

4.4 A New Memory Manager

Sometimes, plugins would like to allocate extra memory whether to work on complex data structures or to compute complex procedures. In addition, it might be utterly probable that two eBPF functions belonging to the same plugin want to share data. The current version of uBPF allows a single eBPF bytecode to access a stack memory of 2048 bytes and cannot satisfy these previous requirements. To break these restrictions, we propose a new memory manager described in this section.

External functions can be used to implement a kind of shared memory between isolated uBPF sub-plugins of a plugin. With the basic uBPF API, it is impossible to share memory because each plugin is running in its own context and its own stack. Thus, we imagined a way to make it possible through the use of external functions. Figure 4.2 shows the structure of the implementation. In this picture, it is stated that a plugin can only share memory with its sub eBPF functions. In this way, different plugins in the main process are isolated and cannot access the memory of each other. This property is due to the design of the virtual machine. In fact, at the creation of a eBPF function, the API function `ubpf_load` requires the location of the extra memory (colored in purple) and its size. Internally, `ubpf_load` adds some checks in the bytecode to ensure that each function is accessing memory within its bounds (i.e. the stack, arguments and heap bounds).

Each plugin has a dedicated extra memory location, representing the arguments and the heap respectively. Since this memory is not automatically managed by the eBPF compiler like the stack, new external functions are conceived to ease the retrieval of a memory chunk on the heap. Just like the traditional `malloc` of the `libc`. The memory manager deals with this memory and contains a simplified version of the `malloc` function. Basically, it looks, from the start address of the heap for the smallest block available to host the requested size. When a memory block is freed, the manager scans the

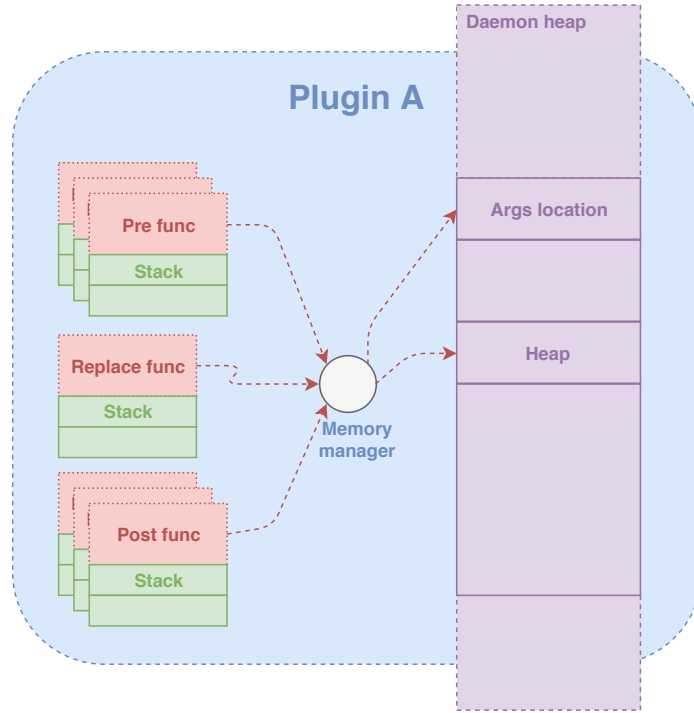


Figure 4.2: Memory structure of an eBPF plugin

entire memory to merge consecutive non-assigned blocks, to avoid as much as possible fragmentation. The memory is at most swept once for each `free` and `malloc`, leading to a complexity time of $\mathcal{O}(n)$ for both functions (n is the heap length).

We previously said that the heap is shared between the different functions of the same plugin. This allows the implementation of a shared memory between these functions. We insist on the fact that functions from different plugins cannot see each other and thus cannot share any memory.

There are some cases where the function of a given plugin would like to share information. For example, we can imagine that a PRE eBPF function would like to share computed data with another one located in POST anchor. Such plugins can create themselves a new shared memory by calling the function `new_shmem` which builds a new block inside the extra heap region. The initiator function must assign an identifier to the freshly allocated block. This ensures that this shared memory block will be accessible by other functions of the same plugin. Since a heap is created for each plugin, the same identifier could be used without any collision because the shared memory mechanism is linked to the heap it is executed. Each key (i.e. identifier) is associated with a pointer, referencing to the memory location of the shared block on the heap. This eases the implementation of this mechanism and calls the `malloc` function and stores the tuple key-pointer on a hash map to ensure an amortized constant memory size.

The next step is to fill the new shared block as a normal C pointer by either using pointer dereferencing or using the `memcpy` built-in function provided by `clang`. Getting a shared block proceeds in the same way : we need to specify the identifier and the external function `shmget` will return the pointer associated with this identifier. Finally, if memory is not used anymore, a plugin must destroy the associated entry in the table. Our implementation of the shared memory manager is greatly inspired from the XSI shared memory implementation available on many UNIX systems [30].

Using a shared memory is useful if the plugin itself does not have enough memory to store a given variable. Without the integration of this new memory manager, eBPF bytecodes only have their own stack and do not have access to the BGP heap. This stack is limited in memory and does not permit storing large data. Thus, our extra memory heap solves this lack of memory. Besides the 2048 bytes stack, an eBPF function can use the heap where its size is defined at plugin load time (see Section 4.3). Once loaded, the heap cannot be resized unless removing all eBPF functions associated to an insertion point. This prevents that the bytecode takes too much memory, leading to a memory overflow, inducing a crash of the daemon.

Having a memory in common is also useful to share data before sending data to a monitoring server for example. Imagine we want to measure the time taken by a function inside the protocol, a first function will be executed at the beginning of this monitored function and will record the start time. At the return location, another eBPF must be executed to save the total time. Since the last sub-plugin does not have any access to the start timer, the first one stores the value of the timer in the shared memory, that will be retrieved by the last plugin (located at the end of the monitored function). Finally, it sends the two timers to the monitoring servers.

Trying to monitor the time is one example of complex monitoring method. There is no other possibility to store the start time of a function since the virtual machine does not allow accesses to the protocol memory (i.e. the eBPF stack is the only source of memory). Another option that could be used is to rely on the return value of a given program. The trick is to pass the return value as arguments to the second eBPF program. This technique could work on integer values. However, if other type values must be returned (e.g. a string or a complex data structure), it is impossible by design to rely on. Indeed, the virtual machine can only store a 64-bit integer in the return register. However, a pointer is also a 64-bit value², and so it could be used to tell the other plugins where the value is stored. This is not true, without any shared memory, this pointer indicates a memory area into the eBPF stack, automatically flushed at the end of an eBPF function execution. The eBPF function receiving the pointer will crash because the stack is not the same and is recreated on each execution of an eBPF function.

Another drawback of this solution is the dependence between two eBPF programs. Since we rely on return values, we need to complicate the integration of the virtual

²On a 64-bits CPU like `x86_64` architecture

machine inside the program because it is needed to connect the return value of the first eBPF code to the arguments of the other one. This is not the desired behavior because we do not want plugins that rely on others to be correctly executed. We made the choice to let the eBPF program an isolated program without any knowledge of its environment. Furthermore, linking two eBPF plugins means to load these two programs at the same time to ensure that the second plugin could be executed. If the second was injected only, it would not be launched since it must wait the return value of the first one. Using and implementing a shared memory accessible for every plugin is thus a good compromise.

The plugin manager can detect if there is corruption inside this extra heap. The plugin heap is actually a large contiguous memory which is located in the real process heap. Like on Figure 4.3, the heap is organized in a form of linked list having as nodes, metadata headers. The actual memory of the user is stored right after the header. The figure shows

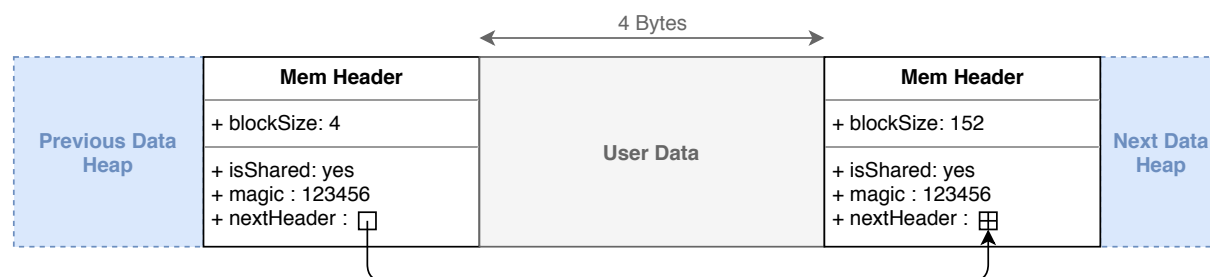


Figure 4.3: Linked list structure of the eBPF heap

a **magic** field inside each header node. This value is fixed to a determined number, that is compared when allocating, accessing and deallocating memory block. The check is simply a comparison of this field by the value it should contain. If the comparison fails, this is a clear indication of a memory corruption. In this case, our memory manager simply refuses to perform the desired task initiated by the eBPF plugin (i.e. using the memory manager to deal with the heap) by returning an error. Upon receiving the error, the eBPF can notify the eBPF programmer to tell that a measure must be taken by reloading a bug free eBPF bytecode.

Each memory header contains several information shown in Listing 4.7 such as a flag telling if the next block is shared between several plugins or not. Another flag is used to inform if the block is available for allocation or already taken by user data. Finally, the header contains a pointer to the next one to ease the heap scan. The heap is not deallocated between two plugin calls so that the eBPF code can maintain some state. First, this choice allows a plugin to keep track of the number of times it was called. This could be considered as the global section of an eBPF program (in comparison with the global section of a UNIX process). Second, allocating a large block of data in the process

```

1 typedef struct header_block {
2     uint8_t available : 1; // is this block is available ?
3     uint8_t shared : 1; // is this memory zone shared ?
4     size_t size; // size of this block
5     uint64_t magic;
6     struct header_block *next; // pointer to the next header block
7 } header_block_t;

```

Listing 4.7: Metadata associated to the allocated memory block

heap could slow and reduce the performance of the program. We then try to minimize `malloc/free` calls as much as possible. However, each time an eBPF code has finished its execution, the plugin manager must clean all the ephemeral memory that has been allocated during its lifetime (i.e. it cleans every block having the flag `shared` set to 1). Since the extra heap is quite limited in memory, we do not want any memory leak if the user forgot to clean its non shared data inside this shared memory. The overhead induced by scanning the entire memory to clean every block is balanced by avoiding returning errors because of programming errors on the eBPF programmer side.

4.5 Adding Memory Bound Checker in JIT Mode

Since the eBPF code is directly translated to the machine code format supported by the CPU architecture on which it is executed, there is no opportunity to check at run time, memory accesses made inside the function unless explicitly adding it to the eBPF code. Then the technique used by the virtual machine is to rewrite the ELF file to add eBPF instructions checking every memory accesses before JITting the program.

In addition, we would like to use the extra heap defined in Section 4.4 as normal memory and thus directly accessible from the eBPF code. Indeed, this memory is not recognized by the VM in interpreted mode unless passing the whole memory inside the arguments of the eBPF hook. Hence, arguments and the heap are sharing the same contiguous memory block. In that case, the memory bound checker will recognize the access as being a legal one.

To include the new memory block, the uBPF API call `ubpf_load()` is slightly modified to take a pointer to this area. On Figure 4.4, this value takes address `0x1031A`. The blue blocks on this schema are the only areas recognized by the memory bound checker. We thus notice that the extra memory must be first created before calling the uBPF API. This modification also impacts the way arguments are passed to the eBPF bytecode. Before, the virtual machine was able to determine at run time, the location of the arguments (i.e. by taking this memory address). This works well in interpreter mode because the

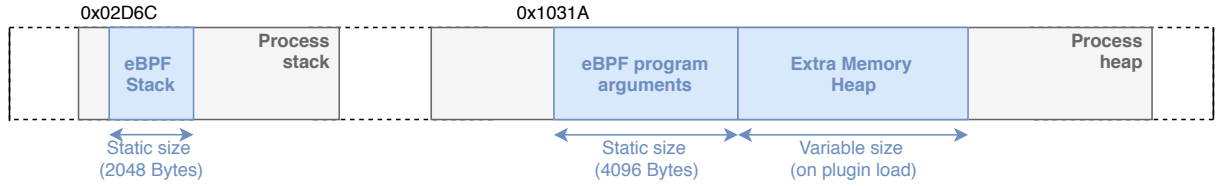


Figure 4.4: Organization of a eBPF program memory.

memcheck is external to the eBPF code and can be dynamically adapted to this memory area. However, since it is now a static verifier, the actual memory address of the extra memory plus the one of its stack³ must be directly written into the eBPF assembly code. However, there is one problem to solve. The uBPF machine must rewrite the bytecode without overriding currently used registers in the program. Indeed, to make comparisons between the concerned address and the stack or the extra memory address, the code must first store the verified address into a register. Instead of analyzing the used registers to select the ones not used at verification step, which might make it impossible to find it⁴, the virtual machine uses one of the five extra registers (**r11** to **r15**). In particular, the memory checker only needs one extra register (**r11**) to make its computation.

Let us take the example shown on Figure 4.4, the steps to verify whether address **0x1031B** is within the bounds of the virtual machine heap are :

1. store **0x1031B** inside register **r11**;
2. subtract the beginning of the heap address (**0x1031A**) on **r11**. This address is a constant value that is written inside the **imm** part of an eBPF instruction. This is the reason why the memory checker is static in JIT mode. Since, there is no possibility to dynamically pass this pointer, this latter is directly "hardcoded" inside the eBPF instructions;
3. if the result inside **r11** is strictly positive, the address is within the shared memory bounds. Then go to 5;
4. if the result is negative, then the pointer is out of bounds. In this case, an external function is called to warn the user of this illegal access before exiting the program without executing the remaining instructions;
5. compares the result of the subtraction of the total size of the shared memory. If the result is strictly greater than this maximum size, throw an error. Otherwise the program is allowed to continue its execution.

³The actual top stack address is saved in **r10** register

⁴It is perfectly possible to find some programs using the whole registers

To check the stack address, the logic remains the same. However, this latter is stored in its own register (**r10**), therefore the comparison is easier in the sense that we do not need to store the address inside the eBPF instructions. A first consequence is that the dynamic allocation of the stack is now made possible. Hence, the eBPF stack works as for usual function calls. The actual frame pointer is stored inside **r10** and the program can allocate needed memory. Second, we have the ability to limit the stack expansion when the code is JITted. Before, the code acted like a traditional C function, in which the memory stack can be freely allocated⁵. Now, with this checker, the total amount of memory allocated on the stack is bounded by the defined size inside the implementation of the virtual machine.

4.5.1 Redefining Calls to the uBPF Machine

The integration of this memory bound checker and the extra shared memory induced a redesign of the API to run a given eBPF code. This combination forces the user of the **ubpf_exec()** (or the jitted function) to think about the way the arguments are passed to the function. On the untouched uBPF machine, the location of the arguments is not restrained because of the lack of static memory checker. In Figure 4.4, the only part allowed for the eBPF code is shown in blue. The block responsible for the arguments is now in the "extra memory" and thus statically assigned on uBPF VM creation. This is a small restriction from the original implementation that enables memory safety inside an eBPF function. Each time **ubpf_exec()** is called, the right pointer to the extra memory must be passed. To avoid programming errors when adding insertion points in the BGP source code, **ubpf_exec** is wrapped into a function in charge of copying the arguments to the extra memory of the program.

4.6 Secure Communication Between the Virtual Machine and the Protocol

We would like that the uBPF machine is as isolated as possible from the main process where it is executed. That is, any attempt to access to unauthorized memory area results in the termination of the currently executed eBPF code. When a given eBPF code is called, some argument might be passed into the virtual environment. As the byte-code might be considered as a function executed at a certain point in the main process, there is no restriction on the structure of the memory passed to the virtual machine. Therefore, it is totally legal to pass a pointer to the code, and this one could be read inside the VM. However, as previously explained, if the eBPF hook could not access the memory related to this pointer, it will stop its execution to give back the hand on the protocol program. This is the required behavior because it guarantees that the eBPF script could not access to the entire memory process. However, this restriction could be cumbersome if the eBPF arguments include complex structure containing pointers to other protocol

⁵Until the per process maximum stack size is reached

variables. It might be the case that a certain plugin wants to have access to different pointers inside this structure. Nevertheless, we do not want to be too permissive in terms of accessed memory. A compromise between isolation and "free access" must be adopted. To cope with this problem, we use the power of external functions. Section 3.2 briefly explained this concept. Recall that these kind of functions are in fact helpers that can be called inside eBPF code and are registered when the virtual machine is created. These helpers are not part of the virtual machine code and thus are executed at the same level as the uBPF machine. In other words, when called, these functions are not executed in the enclosing environment of the eBPF code but rather from the main process where the virtual machine is created and executed. Thus, the entire memory process could be accessed thanks to these helpers.

Back to the main problem, these functions are used to access to some part of the memory originally not authorized for the eBPF byte code. Since the virtual machine imposes that every pointer must point to the dedicated memory area, helper functions must copy the requested protocol variables to the authorized zone for this plugin, so that every memory accesses are on the bounds of the virtual machine memory. Since a copy is performed, the variable is only readable and cannot be modified inside the uBPF machine.

The problem of accessing to nested variables is now solved. A BPF code is now able to read restricted parts of the memory. However, this solution is too simple and does not care about bad pointers that could give a user to these getters. If a user makes a mistake, inside the virtual machine, the error must be confined in the virtual execution and must not be spread to the entire process. Hence, this simple solution could lead to the crash of the entire protocol due to a segmentation fault, if checks are not performed. Worse, the user could use this function to access memory area which is not handled by the helper and leads to an attack against the protocol. Getter functions must sanitize user inputs before performing their original task. Hence, it is needed to revise the way to run a given plugin inside the virtual machine.

4.7 Adding an Execution Context to the Virtual Machine

The basic uBPF API only provides functions to run the virtual machine and register some helper functions. Thus, there is no verification at this step. Therefore, an upper layer must be created to prevent bad eBPF code to perform bad actions inside the protocol process. To do so, this upper layer, which we call the plugin manager, stores one context for each eBPF function. The context structure is then stored in a manifesto and contains several information needed to correctly run a given plugin defined in listing 4.8:

- (1) **a pointer to the passed arguments** : each time the code is executed, this pointer changes according to the location of the arguments. Of course, a plugin only has a read copy of every variable and does not have the opportunity to modify the real process variables. Implemented "Setters" functions are provided to achieve this task;

```

1 struct bytecode_context {
2     void *args;      /* (1) */
3     int type_args;   /* (2) */
4     heap_t *heap;    /* (3) */
5     ...
6 };
7
8 typedef struct bytecode_context context_t;

```

Listing 4.8: Plugin context structure

- (2) **a unique identifier related to the argument passed to the program.** Since plugins do not necessarily have the same type of argument, this number is used to cast the previous pointer in the right structure to retrieve the copy of each argument passed to the uBPF machine. With this unique number, registered function could simply use "pattern matching" to cast the previous (`void *`) pointer;
- (3) **the associated heap :** each plugin does not share the same extra memory area. Sub eBPF programs related to the same plugin can access the same heap. There are thus multiple heaps managed by the virtual machine manager that is in charge to give the right heap to the right eBPF program.

This context is extensible by design and future information may be added if new features regarding the plugin manager are included in the existing project.

Once the context is created, it is needed to update it every time its associated plugin is executed. To do so, at every call to `ubpf_exec()`, the plugin manager has to retrieve the arguments to be passed to the plugins, update the associated context (i.e. the field `args` and `type_args`) and then execute the code via `ubpf_exec()`. As previously said, the related execution context is inserted in a set at "load-time". This choice was made so that helper functions can be safely used inside an eBPF bytecode. As explained in Section 3.2, external functions cannot capture any state or context related to the location where it was called. The solution is then to pass an extra argument to every plugin, that is the execution context pointer. Since this context information contains sensitive data in its own way, any attempt to dereference this pointer must crash the current execution of the plugin. This pointer might be seen as a kind of unusable label inside the eBPF code. Thus, the plugin manager needs to provide a pointer that is not readable and writable inside the virtual machine.

This pointer, now becoming a kind of identifier, is then used for some helper functions such as getters, setters, and the one that manages the plugin heap. These types of functions actually take as the first parameter a pointer to the context. By passing this unique number⁶, the external function can check if any pointer passed to the function is

⁶Pointers are actually memory addresses, which are 64-bit numbers (in the case of a 64-bit architecture)

valid and can be trusted. The verification steps are quite straightforward :

1. we first check if the context pointer is non-null. If it is the case, it is not possible to continue because, we cannot determine from which plugin, this function was called;
2. if the pointer contains a non-null address, the function checks if the pointer is related to any pointer contained in the set containing every recorded context. If it was not the case, accessing the memory area of this bad pointer could lead, to a segmentation fault, leading to the crash of the entire process;
3. once the pointer could be trusted (verified at step 2), we can access the content of the context. For example, let us imagine that a plugin wants to have read access to the nested structure `s` from the main structure shown in Listing 4.9. The `proto_t` structure is passed to the arguments of a certain eBPF hook and thus has a read access to it. This means that it can read every field in this structure. However, it might be noticed that the field `s` (of type `struct nested_struct`) is a pointer that cannot be directly accessed from a plugin. That's why a getter is needed to copy the field into an authorized memory area, either its stack or its heap.

The getter definition has one argument in this case, a pointer to a memory area where the nested structure will be copied. The caller must ensure having read access to the memory pointer by this pointer. Otherwise, the problem remains the same, the virtual machine will refuse to dereference the pointer.

Next, the helper function, will thus access to the context structure, cast the argument field (see Listing 4.8) in the right structure (here, `proto_t`), access to the nested structure and copy it in the memory area pointed by the second parameter of the external function.

It might be noticed in this small example that the context itself could retrieve the right structure, that is, there is only one variable corresponding to the structure to be fetched. However, some plugin argument contains more than one pointer with the same type to retrieve. In this case, the getter function has a third argument corresponding to the memory address of the right field to get. This is then the task of the getter to compare the pointer passed as an argument with the ones contained in the arguments of the eBPF code.

```
6 typedef struct protocol_struct {
7     struct nested_struct *s;
8     int proto_var;
9     char[50] extra;
10 } proto_t;
11
12 struct nested_struct *
13 get_nested_struct(struct nested_struct *s);
```

Listing 4.9: Example of nested structure and a getter to access it

Since the arguments inside the context structure are a copy of the ones passed to the eBPF code, and since the context is not accessible inside the executed code, the comparison between the pointer passed to the getter and the one in the context structure certifies us whether the helper could safely retrieve the variable.

In fact, every helper involving pointers must check their validity because pointers are the most vulnerable structure when programming with the C language. This is particularly true when dealing with code that must be executed in an isolated context. Since we are exposing some functions that are executed in the main process, a programming error (i.e. bad utilization of external functions) inside the eBPF code might crash the whole process. This is why external functions must sanitize every user input. If we consider again the getter function shown in Listing 4.9, the user must provide a pointer to the memory area where the nested structure will be copied. The pointer here must be checked if it is in the virtual machine memory bounds, to avoid overriding the protocol variable. However, the eBPF code could mess up its entire allowed memory up to crashing the uBPF execution, the BGP protocol will not stop its execution, though. The error is confined and the user simply reloads the plugin once the error has been fixed.

Helper functions are not related to one eBPF function and could be called by every authorized plugin. One might ask how such functions could retrieve the context associated with the right bytecode? In fact, the C language does not provide the ability to build closures⁷, to enclose the context before building a "custom" function dedicated for the eBPF program. In this way, every bytecode could access to the same API but the context is enclosed in the contextual environment of the external function. Thus, there is one copy of the same external function for every eBPF functions. Their contextual environment contains the context associated to the eBPF bytecode.

The solution proposed against this lack of closure support is to store into a global variable, the address of the current eBPF bytecode context. Hence, each external function can access this global variable. This ensures that the integrity of the context pointer and that no eBPF programmer could access to the context of another plugin. The global variable is automatically replaced by our uBPF API before each call to an eBPF bytecode (`ubpf_exec` or the JITted function). This solution brings, however, a drawback. Only one eBPF function can be executed at the same time. We made the solution mono-threaded to prevent race conditions when two simultaneous calls to the eBPF functions try to modify this global variable. This clearly impacts the performance of the overall program if this one is multithreaded. In the future, the solution needs to be redesigned to support multithreaded operation.

⁷This is actually implemented in GCC as an extension [31] but is very limited. However not every compilers support it. clang don't provides this feature for example.

4.8 Exposed BGP functions

To allow a plugin to perform some simple tasks more efficiently, we let the plugins use several already implemented functions inside the **FRRouting** project. As for getter functions, they are wrapped into the mechanism that checks if pointers are valid (explained in section 4.7). Our exposed functions are :

aspath_cmp this function takes two BGP ASPATH: A and B. It compares their paths and returns if A is longer, equal or shorter than B;

aspath_count_hops counts the number of ASes contained in the path passed in the argument of the function;

aspath_cmp_left it takes two routes at arguments. This function checks if theses routes originates from the same AS. It is useful to compare MED value;

aspath_cmp_left_confed same as the above function but compare two confederations as-path;

aspath_count_confeds count the number of confederations inside an as-path;

bgp_is_valid_label related to MPLS [32], checks if a given label concerning a given route or prefix is well formed;

bgp_flag_check used to check if a given flag is set into a BGP structure. For example, this function might be called to check if a given BGP session must always compare the MED value;

peer_sort determine if the peer is an eBGP or iBGP neighbor.

The above list are functions especially exposed to compute the BGP decision process. The choice to reveal these functions on plugins is due to the fact that it might be used by multiple plugins. Therefore, this avoids redefining the same instructions each time a plugin has access to the **ASPATH** length. Indeed, the limitation of the eBPF code does not allow the import of other functions defined in another module. Thus, the only way to define a function accessible by multiple eBPF program is to go through external functions.

4.9 List of getter functions created

This section explains which structures are involved with getter function :

get_cmp_prefixes this function is used to retrieve two prefixes received during the BGP decision process. The first one is a prefix received by the remote peer when it has sent a BGP Update message. The second prefix is one prefix already located in the ADJ-Loc-Rib;

get_attr_from_prefix get the attribute structure related to the prefix sent by a remote peer;

extra_from_prefix according to the documentation of FRRouting "*Ancillary information to struct bgp_path_info, used for uncommonly used data (aggregation, MPLS, etc.) and lazily allocated to save memory.*". Also contains data about route dampening;

as_path_from_prefix retrieve the AS path related to a prefix;

get_attr_from_path_info retrieve all the attributes of the prefix passed as arguments;

get_cluster_from_attr retrieve data related to the route reflector configuration;

get_community_from_path_info get the BGP community structure associated to a route;

get_bgp_instance get the variables concerning the BGP instance of the router;

peer_from_prefix retrieve information about the peer who has sent the prefix given as arguments;

get_maxpath_cfg get configuration about the maximum number of paths authorized per route;

peer_from_args if there is a pointer related to a peer structure inside the arguments of a given eBPF program, this function will retrieve it;

get_raw_prefix_from_args retrieve the prefix contained in the argument of an eBPF program.

4.10 Conclusion

This chapter described our new high level API to solve problems of the original uBPF machine. Our API provides an easier and better integration on the protocol side. We created a manager supporting multiple instances of uBPF VM in different BGP functions. We also provided a flexible way to add eBPF functions by providing a simple tool to inject them inside the protocol. The biggest change is the integration of a context that gives extra information to the virtual machine when it executes the eBPF bytecode. This context allows first to make integrity checks at plugin run-time, and second to add a new extra memory and get rid of the uBPF 2048 bytes stack.

We also presented an API that plugins used to retrieve information about both the protocol and the new memory. This API is key because it enables a plugin to extend BGP. Hence it must be as generic as possible to let an eBPF programmer design new features for the protocol.

Chapter 5

Use Cases

The previous chapter explained the modification of the uBPF virtual machine aiming first to meet the requirements of a network protocol and next ease its integration inside a given protocol implementation. Until now, the protocol has been redesigned to include our virtual machine inside BGP. However, no plugins have been proposed to extend its capabilities yet. This chapter now introduces several use-case showing how to extend the Border Gateway Protocol. We begin by proposing a new technique to monitor BGP variables that are complex to retrieve with the available solution. Then we present our modified version of the flexible BGP decision process that implement the one proposed by [Retana and White \[33\]](#) through a set of eBPF plugins. We prove its extensibility by developing an alternate form of the hot-potato routing that uses geographical coordinates between BGP routers. We finally close this chapter by introducing eBPF filters, a new form of importation and exportation filters taking the advantage of a programming language to build complex filter policies.

Our work is based on an already implemented version of the BGP protocol maintained by the FRRouting community. The code is open source and is available on the GitHub platform [\[34\]](#). Indeed, other projects could have been used for the eBPF integration such as Bird, OpenBGPD or Quagga. Nevertheless, the chosen project is actively updated by a community of maintainers.

FRRouting is actually a fork of the Quagga Routing Software Suite implementing multiple IP routing protocol like BGP [\[17\]](#), OSPF [\[14\]](#), LDP [\[35\]](#), etc. The first version of FRRouting was released in April 2017 and brings a lot of improvements to its predecessor. Their Github page [\[36\]](#) describes all the modifications performed for the first release of FRRouting. One of these changes resolves around performance issues in BGP by integrating kernel threads [\[37\]](#). Since the first release, they have added a lot of functionality to the BGP protocol by including both standardized RFC extensions or drafts and VPN support [\[38\]](#). Therefore, this BGP extension provides a lot of functionality which is not implemented in the Quagga routing suite.

Protocols are in continuous development and contain the most recent features as well as innovative ones for data centres to ease their configurations. Thus, FRRouting provides

a flexible routing suite with innovative technologies. Because the advantages that bring this project, Cumulus Networks, a company providing an "open source" way to design networks in enterprises, has decided to integrate FRRouting in their solutions [39].

5.1 BGP Monitoring

Monitoring the BGP protocol could be useful to extract some variables to have a better understanding of the network state. There are some methodologies to extract some important features about BGP routers like the protocol described by Luckie and Beverly [7]. Nevertheless, the proposed technique is error-prone and only works for a fraction of the BGP routers since it does not have direct access to the protocol variables in the router. Hence, we propose a new use-cases of monitoring which can be used in conjunction with eBPF plugins, and thus, allowing a better fine grained monitoring.

Our monitoring server is launched at the BGP daemon start and is executed in its own process. The monitoring server is listening through a PIPE for all incoming messages of the eBPF plugins. When a BGP session is established, the monitoring server creates a new BSON object to send to a database for further analysis. The figure 5.1 shows the desired architecture. Unlike SNMP, with our design of monitoring, this is the routers

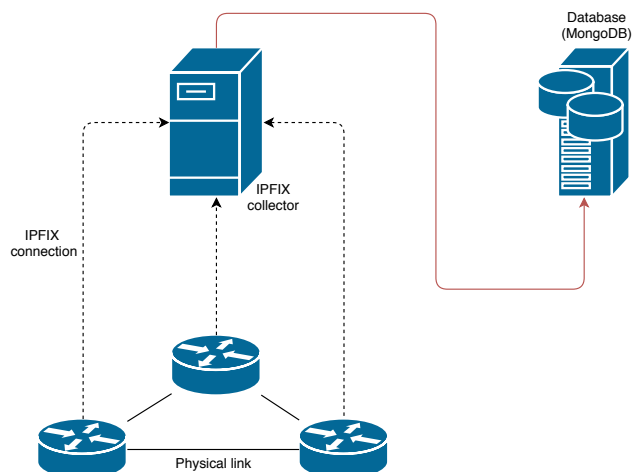


Figure 5.1: General Architecture of Monitoring BGP

itself that send data to an **IPFIX** collector in charge of persisting information in an external database. The collector is passive and does not need to poll every x second a given configured SNMP variable in the router.

The monitoring server is running in its own process to avoid saturating the BGP process. Although the server does not require a lot of memory, it is better to execute it to another process. The reason is straightforward, if the daemon in charge of receiving data

from eBPF plugins crashes, it will not affect BGP which could continue its execution even with after failure.

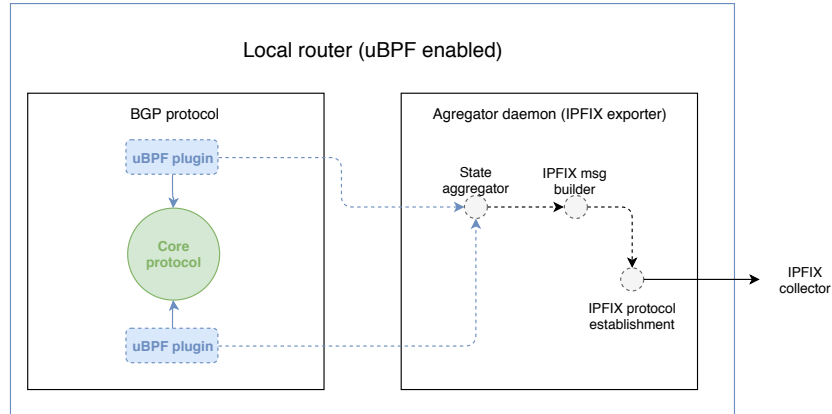


Figure 5.2: General Architecture of Monitoring BGP With eBPF Plugins

The figure 5.2 represents the general configuration of the protocol to enable monitoring for multiple routers with uBPF virtual machine deployed inside them. As depicted, the protocol is only modified to include the virtual machine, and the main logic of the monitoring is located outside it. This allows the protocol to focus on its main task without overloading it with everything related to the monitoring. Hence, all the "monitoring" plugin establish a connection with a local daemon (which is running in parallel with the protocol) to send collected data. To keep the architecture as generic as possible, the local aggregator is using the IPFIX protocol, implemented in today's routers. The proposed architecture can then fit to most of the existing routers.

On the collector side, we opted for a setting that includes an IPFIX collector which communicates with a MongoDB server. Every collected data is stored in a persistent way to support analysis that could detect some potential configuration issues (for example).

5.1.1 Monitored protocol variable

To allow a network engineer to analyse the BGP protocol in a fine-grained way, the uBPF machine has been deployed in multiple BGP functions (i.e. insertion points).

Receiving Open messages Plugins attached to this location in the BGP code can measure the time taken to process this type of message sent by a remote peer until the router write the answer to the remote peer socket. This monitoring information might be useful to detect remote router failures. The more a peer router sends open messages, the more it has restarted and this might be an indication of a system maintenance or a reboot due to a crash. The recorded time could be used to know if the local router is overloaded or not (i.e. if the processing time is longer than usual).

Receiving Keepalive messages The eBPF code collects the time taken to receive a new keepalive message from every remote peer. Again, this information could help to see which router has trouble to send new keepalives because it is congested or is down due to various reasons.

Receiving Update message Plugins will monitor the time taken to process an update message from a remote peer. It does not include time to run the decision process, nor the time to advertise new routes to peers. In fact, it is just an indication that an update message is received. Later, these monitored information might be used to count the number of total packets sent by a specific peer

Prefix Update handler When an update message is received, there are two kinds of information inside it, prefix updates and prefixes withdrawals. In this case plugins will capture an update of the aspath for a given prefix. Other plugins are in charge to do the same thing for a prefix withdrawal. BPF programs will only send the prefix related to the removed route to the monitoring server.

BGP decision process eBPF program will capture the time taken to run the BGP decision process. The latter will not run just after receiving an update message but will be executed when the worker thread of FRRouting decides to run the planned task. Therefore, the calculated time does not take into account this "waiting time". In fact, since multiple thread are implied in the process from the reception of an update until the BGP decision process, the only way to monitor the true time is to directly modify the protocol code to add information to distinguish which update message the planned decision process execution belongs to.

Parsing AS Path A plugin will retrieve the AS path parsed by the protocol to send it to the monitor. A plugin only sends one AS path per packet even if multiple prefixes are mentioned in the update message. Indeed, if it contains several prefixes, they reference the same AS path and it would not be necessary to duplicate it in the monitoring database.

Some plugins also monitor prefixes rejected by inbound filters defined by the network administrator. If any of them are rejected, a location for a plugin is provided and could be used to monitor the reason for the rejection of the concerned prefix.

5.1.2 An external Daemon to prepare data for database

This functionality allows the eBPF machine to send data collected from every plugin to an external daemon. The function in charge of contacting the local daemon contacts BGP process through a UNIX pipe. As the daemon is external to the BGP process (i.e. these are two different processes), this is a simple way to send data through these two processes. Interprocess Communication could be done by multiple ways in a UNIX system like sockets, semaphore, shared memory. Nevertheless, the daemon is a child

process and thus it is possible to use pipes as the simplest structure to communicate data. BGP is pushing formatted data coming from eBGP functions to the output stream of BGP that will be received on the input stream of the daemon. To be correctly parsed, data from eBPF monitoring plugins must follow the format defined in Figure 5.3.

The external daemon is a server waiting for data from uBPF hooks. Once a segment has been sent to the daemon, it will be first parsed. Second, if data needs to be processed, the program will generate final values to be sent to the database (i.e. some computation must be done because uBPF sends raw data to be as fast as possible).

This small server running on the router maintains state for every data sent by plugins. In particular, it stores a state for each peer connected with the local router plus a state for local information not specific to a given peer such as the time to process a BGP update message, the number of routes contained in the RIB, how many decision processes have been run, etc. Each time a plugin sends a notification, the corresponding state is updated.

The last step is to send gathered data to an external database for further analysis. Since BGP is mostly running on routers, it is needed to find a simple way to send data. Hopefully, the IETF has standardized IPFIX (**IP Flow Information Export**) [40] as being a universal standard, to export and collect information flows from routers. Therefore, our external daemon implements an IPFIX exporter. At a specified rate¹, the exporter retrieves every stored state and puts them in an IPFIX message targeted for an external collector. Once the exporter has sent the desired message, it resets its stored state. Indeed, since data has been correctly sent it is not used anymore inside the exporter.

5.1.3 Ipfix Templates

The IPFIX protocol relies on templates to send data. Since IPFIX is extensible and could be used to monitor protocols like TCP, HTTP, UDP, etc., a template defines the structure of the IPFIX message. Hence, with this information, the collector can understand the type of data sent by the exporter, and extract the right data in the message. Both the exporter and the collector use the same specified template. This step is done before the two processes are launched and cannot be done dynamically.

Actually, the IPFIX protocol comes with a set of predefined information variables defined by the IANA that can be used to create a custom template. Among them, the list maintained by IANA defines some information elements (IE) like the destination IP address, destination port, vlan ID, etc. Every IE, is associated with a unique identifier and is used to build the template. Nevertheless, this list does not include some desired IE to capture some BGP variables like the BGP decision processes time, prefixes advertised by a remote peer, etc. Actually, there is a way to predefine custom IE thanks to the flexibility of IPFIX. The trick is to create multiple "enterprise-specific" information

¹e.g. every x second

elements associated with a given enterprise number ID.

IPFIX defines two kinds of IE. As explained above, there are IANA-assigned and enterprise-specific Information Elements. To distinguish different enterprises, the protocol assigns a unique ID per enterprise from which custom IE would be defined. This is exactly the way we proceed to design IE for the BGP protocol. An unassigned enterprise number is used in which some custom IE are defined.

5.1.4 Ipfix Collector

Once data has been collected by the external exporter, this latter uses the IPFIX protocol to send in a formatted way the aggregated BGP variable. The external daemon of the previous section (5.1.2) sends this data to another process located on the same machine or in another one. In fact, the desired architecture is to have one collector that listens to multiple exporters (i.e. multiple BGP routers).

Thus, the task of the collector is to retrieve messages sent by exporters to store them in a persistent way in a database. This process is executed outside the context of a router, on a server. Hence, it is possible to install a MongoDB client inside it to make the interaction with the database.

Once an IPFIX message is received, the collector is in charge of creating a new BSON from this data. Finally, the new generated BSON is sent to the MongoDB database.

5.1.5 Structure of a monitoring packet

Since BGP routers now send monitoring data to a global aggregator, and eBPF plugins collect different types of data, we need to create a way to efficiently build a protocol to send data toward the database. Each plugin creates a structure to store the data it has collected. Listing 5.1 provides an example of structure which is sent to the local daemon. Since a PIPE is used, client and server must agree on a packet structure. In our case, this is simpler since the local daemon is local. Indeed, memory must not be formatted in "network order" because these two processes are executed on the same processor architecture.

The figure 5.3 shows the simple segment structure imagined to send data variables to the server. The header is quite simple but required; **Length** contains the total size of the data and **Type** mention what kind of data is represented. Finally, the last part of this segment is the data itself. It is parsed according to the **Type** value on the header, since the content of the data could vary according to it. Data is not strictly formatted because protocol variables might differ. Therefore, a parsing step is required on the server side. Sometimes, this step is quite simple, because a C structure is copied in the segment, containing fixed size data only. Thus it is simple to decode information, since C can

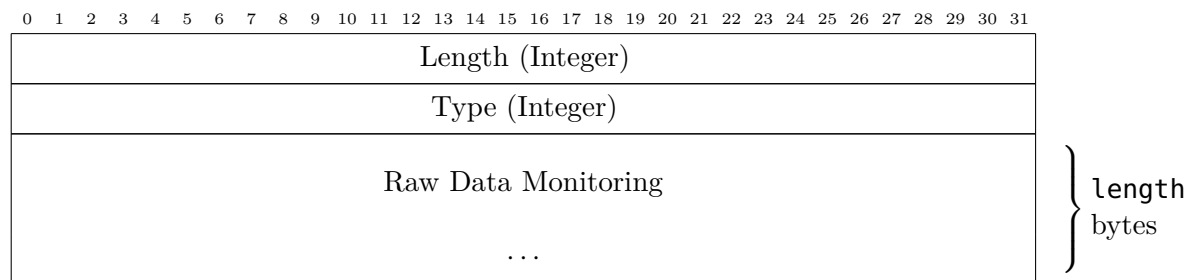


Figure 5.3: Structure of a segment sent to the daemon server

automatically cast the memory zone in the right structure². Sometimes, we are dealing with variable length data, such as strings. This is the case when sending the ASPATH. Indeed, this length cannot be predetermined and cannot be stored in the structure³ because this one stores a pointer to the as-path memory area. Then, this example shows that parsing must sometimes make some extra computation.

Having the Type inside the header ensures the modularity of both the client and the server. Later, if a new type of monitoring is created inside the BGP protocol, we only need to implement the parsing procedure inside the server.

5.1.6 MongoDB database

The local daemon sends data received by eBPF plugins into a global database server running a MongoDB instance. The goal of having a database is to perform the analysis of every monitored variable of several BGP routers. If users want to use multiple independent databases to isolate some routers from others, the actual implementation of the local daemon allows contacting a specific database to send data.

Using a MongoDB database also provides powerful tools to make data analysis. It contains a query interface to aggregate data having some value in common or that satisfies to the written. MongoDB is a document-oriented database that differs from traditional SQL database. Every document is stored in a BSON format, a JSON like document supporting binary data types, useful to store pictures, for example. In that sense, it can be seen that BSON is a superset of the traditional JSON format.

Unlike JSON, BSON is not human readable, so each document is stored as a binary blob. However, it is possible to use the JavaScript Object Notation (JSON) to represent

²Structures are generally defined in a header C file (.h). Both server and client could recognize how the structure is built.

³Actually, the full ASPATH could be stored inside the structure. However, buffer length must be statically fixed. If this plain structure is sent to the server, some bytes inside the buffer are not used, leading to send these non-used bytes through the local TCP connection. For performance reasons, data must be as small as possible.

```

1 struct monit_bgp_keepalive {
2     struct timeval elapsed; // time elapsed between two consecutive
        keepalive
3     struct timeval last; // time when last keepalive has been sent
4     uint32_t keepalive_interval; // normal interval between two
        keepalive
5     struct in_addr remote_id; // remote router IP
6     struct in_addr local_id; // local router IP (which send a
        keepalive)
7 };

```

Listing 5.1: Example of structure containing monitoring information when a keepalive message is sent to a remote peer

```

1 {
2     'myBool': true,
3     'myInt': 10,
4     'myArray': [1,2,6]
5 }

```

Listing 5.2: Example of JSON object

the BSON being stored in the database. This representation might differ from the true object in MongoDB since values are not represented as a string. For instance, a boolean variable is not stored as the string "true" but rather with the value `\x01`. Listing 5.2 depicts a traditional Javascript object, representing what a binary document looks like inside MongoDB.

5.1.7 MongoDB C Driver

Since the monitoring server is written in C, it is needed to use an API which is compatible with the language. There is an official supported driver called `mongoc` that is relatively quite simple to use. In our case, this driver is used to create new **BSON** documents from the structure sent by eBPF plugins. The simplest way to build a new document is to use a predefined macro inside the `mongoc` API. Listing 5.3 depicts an example of the BSON creation. The `BSON_NEW` macro is the simplest to use and try to mimic the syntax of JSON. It might be noticed the lack of types' checking with this method, the produced code is minimal, though. Of course, the API offers other functions to deal with type checking, however, creating a BSON is more cumbersome. Both methods are valid and offer some advantages that the other one does not propose.

```

1 bson_t *document = BCON_NEW(
2     "type",      BCON_UTF8("open"),
3     "remoteID",  BCON_UTF8(buf),
4     "localID",   BCON_UTF8(inet_ntoa(open->local_id)),
5     "time",      BCON_DOUBLE(diffticks / (CLOCKS_PER_SEC /
6         1000.0)),
7     "status",    BCON_INT32(open->status)
8 );

```

Listing 5.3: Creating a new BSON document in C

5.2 Redesigning the BGP Decision Process

Thanks to the dynamic plugins insertion mechanism, it is possible to rearrange how the decision process is performed internally. The original BGP decision process is very standardized and does not provide the flexibility to rearrange how the different steps are evaluated. In addition, there is only a restricted part of the variables taken into consideration during the decision to keep the new advertised prefix or not. Including a new feature into this process automatically requires to dig into the implementation of BGP, understand its structure and recompile the process to run the new modifications. Of course, the whole procedure is cumbersome and precludes fast tweaks into the protocol. Hence, the use of eBPF might get rid of these constraints by completely modularizing the BGP decision process.

To be fully compatible with the uBPF machine, the code used to perform the decision process has been rewritten. Internally, FRRouting implements the complete decision process in a single function. All sub-steps are then executed sequentially in this same function. However, to exploit the full capacity of plugins, it was necessary to rewrite each original steps into separate functions because we would like to either change the execution order, changing the default behavior of a given step or even adding new steps. Therefore, a function with an insertion point is created for each BGP decision step. These new created functions are wrapped into the original one and are executed through a new manager handling the pluginized decision process. When the decision process is called, it first contacts our manager to execute decision functions turn by turn. This supervisor uses both BGP decision step functions and new extra eBPF **REPLACE** functions added by the user. This "decision manager" is relying on the return value of the BGP decision steps to determine the next plugin to be executed right after. Since each sub-step is associated with a unique identifier, plugins could use it to indicate the next step to run. For example, if the return value of a function is **External_Check**, then the next step is the one preferring an eBGP peer over routes received from other types of peers. The decision manager is designed to stop the decision step when one of its sub parts returns either :

DECISION_NEW : indicates to the manager that the new prefix is preferred over the existing ones contained inside the BGP-Loc Rib.

DECISION_OLD : the considered prefix of the BGP-Loc RIB is better than the new one. Then, the protocol must compare the other prefixes of the BGP-Loc RIB still not considered with the freshly received one.

DECISION_ERROR : a plugin encountered an error. So the decision step must be rerun from scratch without any intervention of the virtual machine. The original decision step will be considered, and a network administrator is notified of the faulty plugin.

Procedure 1 shows how our decision process looks like. About its implementation, the

Procedure 1 DECISION PROCESS : Compute the BGP decision process

Input: p the prefix from which the best route will be computed

Input: $routes \leftarrow \{r : r \in \text{RIB} \wedge r.\text{prefix} = p\}$

Output: The best new route associated with the prefix p

$d \leftarrow \text{getDecisionManager}().\text{newInstance}()$

while $d.\text{state} = \text{IN_PROGRESS}$ **do**

$d.\text{nextStep}(routes)$

(executes the current decision step and stores its return value in d)

if $d.\text{state} = \text{ERROR}$ **then**

$d.\text{executeOriginalSteps}()$

return $d.\text{selectedRoute}$

decision manager is a structure with an array containing the fifteen decision step functions of the original decision process. Its task is to take the first function to execute and compare the return value. If one of the three values described above is encountered, it will either return the computed result of the primary function (i.e. **DECISION_NEW** or **DECISION_OLD**), or recompute the whole step but by deactivating all the **REPLACE** functions of every plugin that are part of the decision process. Since steps might not be executed in the original order, simply deactivating the faulty plugin does not solve the problem because switching to the original instructions could create a loop inside the decision process. Let us take the example depicted in Figure 5.4. In (b), the user decided to begin the decision process by checking the AS path attribute, then comparing the weight between the two prefixes. Nevertheless, for some reason, the plugin crashes and returns an error. If the manager only deactivates the red plugin (i.e. the internal weight check will be used), like in (c), since the original procedure wants to continue with the AS PATH check (the normal procedure is shown in (a)) a loop is created. The protocol is indefinitely stuck and cannot speak with their peers anymore. This explains the re-computation of the whole process with the original functions of BGP **FRRouting** implementation.

It is interesting to notice that the design of the decision manager provides a way to pluginize only a sub part of the decision step. Thus, if a user only decides to re-implement

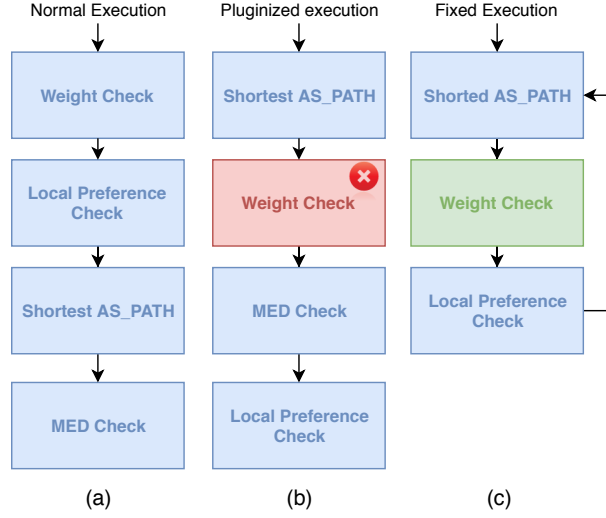


Figure 5.4: Example of loop creation when attempting to recover a plugin failure

a single step of the decision, without creating any loop inside the computation, the manager could perfectly handle the user request.

5.2.1 Building New Decisions Steps

The characteristic of the decision manager is its flexibility to accept new decision process steps which are not originally part of the traditional BGP decision process. This allows new opportunities to fully tweak the decision process by leveraging the power of a Turing Machine. In this sense, the only limitation is the restriction when accessing memory. Indeed, a plugin could only access what the virtual machine allows inside a plugin. In other words, plugins parts of the decision process could compare every attribute that is only accessible through external functions (getters).

Since these new steps do not belong to any BGP function, the manager needs to differentiate plugins that are part of a BGP function from those that do not. Since an identifier is assigned to every pluginized function, extra decision step plugins must take a free ID. The decision manager has to explicitly call every eBPF code of this "artificial" **REPLACE** dummy plugin (i.e. in the sense that no function is associated with them) instead of executing the pluginized functions. Hence, the decision manager is interacting directly with the plugin manager.

To give an example of this functionality, we explicitly added in the decision process, a comparison between the geographical location of learned prefixes. This was possible by opening the BGP communities attribute to the plugins via getter external function. Since a BGP community value is encoded on 32 bits, and that the first 16 bits, by convention are reserved for the AS that has originated this community value. The 16

remaining bits are left to encode the geographical location. In some cases, a value of 32 bits is too restrictive. Although, multiple BGP community values could be inserted for a unique prefix advertisement, we also provide an API to handle extended [41] and large communities [42] allowing encoding values on 64 bits and 96 bits respectively. Our solution works for each type of communities but we demonstrate this use case with 32 bits BGP communities.

Since the first 16 bits are taken for the AS number, we decided to reserve 1 bit of the lower 16 bits to indicate whether or not this value encodes a geographical location. Finally, we end up with 15 bits. This is quite poor to obtain precise information about the exact location of the router that originally advertised a given prefix. We found a workaround by giving an ID to some predefined cities that might potentially contain a BGP router. Then, the plugin must maintain an internal correspondence with the geographic coordinates belonging to a given ID. Consequently, we just found a way to compare two prefixes with this trick. Finally, the new extra decision step prefers the closest distance between the location of the router and the BGP community contained in the route. If it cannot make a decision, they just hand over the next decision step, as usual.

5.3 A new way to add BGP import/export filters

With the current implementation of FRRouting, filters are defined via CLI and can be defined in four categories :

1. **access-List** : the network engineer defines lists of prefixes to filter (or authorize). Each access list contains one or more prefixes to allow or deny. For example, if one list contains the prefix `229.31.0.0/16`, an incoming route with prefix `229.31.228.0/24` might be discarded if the list is configured to match every route containing at least the same first 16 bits. Some lists could also define an exact match, which means the prefix must completely correspond to the one contained in the filter. So our previous example will not be matched with our small access list. Thus for each received route, the router analyzes these access lists to find whether the prefix must be discarded or not. This is a router filter only based on the prefix.
2. **prefix-List** : their behavior are the same as Access-Lists but allow putting matching conditions on the range of the received prefix. Adding a prefix in a prefix-list could be done with the following command in the FRRouting CLI :

```
1 ip prefix-list name (permit|deny) prefix [le len] [ge len]
```

If the user types "`deny 192.168.23.0/24 le 26 ge 25`", this tells to the router to match the first 24 bits of the route (i.e. `192.168.23`). Furthermore, the filter will be only applied to the routes having a prefix length between 25 and 26 included.

Hence this prefix list rule is the same as the access-list containing the following IPv4 prefixes:

- 192.168.23.0/26;
- 192.168.23.64/26;
- 192.168.23.128/26;
- 192.168.23.192/26.

3. **AS-List** : The last type of filtering is made on the received AS Path. The user could define a regular expression matching some ASes inside the received path. The regex can then define several use cases such as matching routes :

- that passed through a specific AS;
- that originates from a specific AS;
- passing to a specific sequence of ASes;
- etc.

4. **route-map** : this is certainly the most flexible type of filtering inside a router. It allows filtering routes according to several attributes such as :

- the BGP communities (large and extended included);
- the interface where the route is received;
- the route prefix;
- the route origin;
- the peer announcing the route.

In fact, route-maps are a kind of fine-grained access-list, since they can read more attribute to decide the importation (or exportation) of a route.

Although Route-Maps add flexibility to the process of filtering routes, they provide a few extra information about the current state of the received prefix such as communities and general facts about the way the prefix was received in the Router. Some export rules such as avoiding sending a particular prefix to a certain neighbor are possible. However, the configuration might become cumbersome and complex. Then, having the ability to exploit the power of eBPF byte-code could ease some of today's complex configurations. Furthermore, eBPF could expose more information about the current state of the protocol. Filters will also have the power of a Turing-machine to perform current filter tasks in a simpler way, but could also handle more complex configurations. We thus provide a flexible way to add more filters by configuring dedicated eBPF hooks.

The filtering process is made per peer and per received routes and defined in a single function inside FRRouting. Adding an insertion point for the virtual machine is thus quite simple. However, since we would like to add more than a single eBPF filter, we need to add a new functionality for our virtual machine API. Filters must be executed turn by turn in a predefined order. The current uBPF API provides three types of execution modes (**PRE REPLACE** and **POST**). We could use the **PRE** execution to add more filters because it can carry multiple eBPF hook. Nevertheless, there are two problems with this execution mode. First, this placeholder is not intended to modify the protocol state and second, the execution order of these eBPF functions is undefined. We thus introduce a special execution mode of **PRE** which both can modify protocol variables and execute bytecode in a deterministic way. We call this new execution mode **PRE_{det}** in this section. In this configuration, eBPF bytecodes are still executed at the beginning of the insertion point. The biggest change is that plugins have access to external functions capable of modifying protocol variables. We can consider this **PRE_{det}** as several dummy **REPLACE** no-op anchors executed at the beginning of the filter function. The user can also provide the execution position of each eBPF functions when injecting an ELF file into the protocol implementation. For example, three filters, namely f_1 , f_2 and f_3 must be executed in this order. Nevertheless f_2 is already attached to the protocol. Hence, it is possible to tell the virtual machine to place f_1 before f_2 and f_3 after f_2 .

Since we are dealing with filters, like the BGP decision process, it could be the case that some of them do not need to be executed because a previous filter has already decided to accept or reject the route. Like the flexible BGP decision process, eBPF functions rely on their return value to indicate to the virtual machine whether or not it must continue the execution of filters. Three return values are taken into consideration :

FILTER_DENY : the last executed plugin has determined not to import the route into the protocol. The execution of the filter part of BGP must stop without adding the route into the ADJ-RIB-in.

FILTER_ACCEPT : the filter accepts the route. The virtual machine stops the executions of the filter and tells the pluginized protocol function to import the route into the ADJ-RIB-In.

BGP_CONTINUE : no decision has been taken for this route. The uBPF VM must execute the next filter on the execution queue.

If **PRE_{det}** has finished its execution without any decisions, the virtual machine executes the normal part of the original function (or the replaced one defined in the **REPLACE** part of the plugin) and eventually the eBPF functions located in the **POST** part.

We previously mentioned that a filter could modify protocol variables. As for traditional filters, the virtual machine enables an eBPF filter to modify some attributes of the route such as : the local-preference, the MED, the AS-PATH (to support path prepending for example), BGP communities, etc. We also grant to eBPF filters a read access to the

current RIB of the BGP router. This new opportunity could bring new filter possibilities based on the RIB content. In order not to degrade too much performance, the RIB access is only authorized through external functions. Walking into the BGP routing table inside a plugin needs to copy all routes inside it. Since copying memory induces a lot of latency, performances could be affected.

The choice not to give write accesses to the RIB for eBPF filters is quite straightforward. Programming errors could lead to major inconsistencies such as deleting a route to a particular prefix. Since BGP is an incremental protocol, routes might never be re-announced by other BGP neighbors. Ergo, parts of the network won't be reachable anymore and the error might spread to other BGP peers.

Manipulating BGP communities is now easier. Traditional BGP filters rely on route-maps to select every prefixes matching a condition about a particular community. Filters can contain several route-maps, which could be resource intensive. Since our filters can be implemented in a state-less way (i.e. the eBPF programmer defines which communities to match without maintaining a list of prefixes as a route-map does), resources are saved. Furthermore adding, deleting and comparing communities are simpler because uBPF brings the flexibility of a programming language to perform these actions.

The same use case with geographic position as the BGP decision process might be used to decide to accept/reject a route. A network engineer first encodes the geographic position of a given prefix into large or extended communities. Second, he defines a filter to compute the "as-the-crow-flies" distance between the location of the router and the prefix. Finally, the filter might or might not accept the route according to a distance limit set by the network administrator. This type of filter is, in practice, complex to perform without the uBPF virtual machine.

If an error occurs on a `PREdet` function, the next eBPF filter is executed. The errors that could arise inside a filter function are related to the memory of the plugin. To retrieve attributes related to a received route, the filter must request for variable via an external function that will copy the requested variable inside the authorized memory area of the plugin. Due to programming errors, the memory plugin could be overfilled, leading to the crash of the current eBPF program. This is the only case that could arise when dealing with eBPF filters and might happen if the plugin calls more than one the same external function copying multiple time the same protocol variable into the plugin memory area. Therefore, it might be the case that the filter already has modified an attribute of the received route. In this case, if the next filter decides to import the route, attributes will be partially correct (i.e. the failed eBPF filter has not fully finished its execution, the next non executed instructions could have modified other attributes). However, consequences are local to the router. Indeed, during the execution of the BGP decision process, the router could change the next AS to reach for the prefix having partially correct attributes. Consider that the failed filter was in charge of modifying the `local-pref` attribute. However, due to memory overflow, it is unable to make the

modification. Hence, received prefixes do not contain the right local preferences. The correct decision process will then choose a different prefix from another AS than the one expected via eBPF filters since **local-preferences** are not correctly set.

To avoid this memory overflow problem, an eBPF analyser, running before the injection step, could detect the bad utilization of memory. This analyser would simply refuse to load the ELF file if such an error is spotted. We assume for now that the written eBPF functions are first analyzed by a human "inspector" before injecting them, so that no memory error can occur. In a future version of the virtual machine, this pre-analysis should be integrated such that they could quantify the total amount of heap needed to execute the eBPF program. Actually, the **clang** compiler already contains a mechanism to compute the total amount of memory allocated to the eBPF stack. This compiler throws an error by not transforming the C file into eBPF bytecode. As the heap is a new extension to uBPF, **clang** cannot prevent memory overflows.

5.3.1 An eBPF filter to reduce BGP router memory

Allowing an eBPF filter to have a read access to the BGP Routing Information Base has several advantages. This section describes a particular use-case taking this benefit.

Memory management in BGP routers is crucial. Figure 5.5 depicts the current FIB of a router maintained by the Route-Views project [43]. Some of their routers exceed 800K

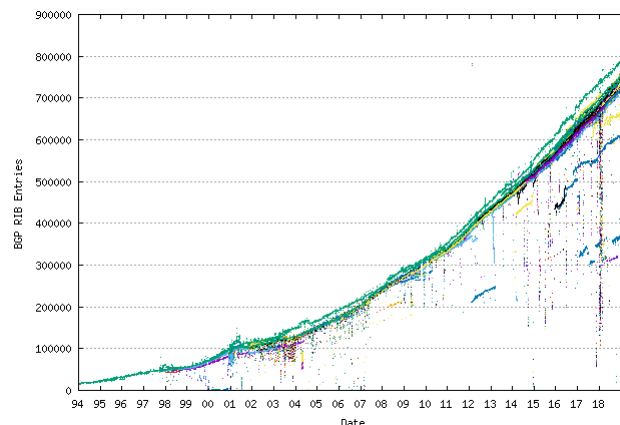


Figure 5.5: Evolution of BGP IPv4 FIB entries from 1994 to 2019 seen by Route-Views routers. Source: <https://bgp.potaroo.net/>

IPv4 routes. Since other BGP peers could modify which prefixes they send to other BGP speakers, some redundant routes could be added inside the FIB. For example, imagine a neighbor able to reach **10.26.0.0/16**. For internal reasons, it decides to advertise this prefix plus other more specific prefixes such as **10.26.1.0/24**, **10.26.2.0/24** etc. BGP routers consider routes since traditional filters cannot have full access, in an easy way, to the RIB content. This leads to a waste of memory. The same reasoning could be

applied to other cases such as a peer advertising `10.26.0.0/16`, but other peers sending more specific routes. We would like to keep some of these redundant routes, for failure recovery, but not all of them.

The proposed eBPF filter is plugged as an import filter. For each received prefix, it looks into the RIB if the prefix is already present. If it is the case, the filter accepts it. If BGP contains a route to this prefix, the filter will determine the number of routes containing this prefix. For example, if the current prefix is a `/16`, the filter looks for other more global prefixes $\leq /16$ in the RIB. It determines the numbers of less specific prefixes that incorporate this `/16`. If the number exceeds a given threshold determined by the eBPF programmer, e.g. 4 maximum more global prefixes, the route is not accepted.

Let us discuss about an interesting case that could happen when filters apply this policy. It might be the case the router receives more specific routes (e.g. multiples `/24`) before receiving a more global one (`/16`) that incorporates all the other previously received `/24` prefixes. Actually, the exposed API could detect this behavior. To respect the threshold previously defined, some routes must be removed from the RIB. However, if routes are removed, this implies to send a withdraw to other peers at the same time. This is not a good solution because two messages will be advertised to BGP neighbors which will, in turn, spread the network updates to other peers in the network. An extra overhead of BGP messages will be added in the global network because of the configuration drift caused by an importation filter. We thus decided to omit this particular exception.

5.3.2 Taking Advantage of uBPF to Write Complex Filters

With the ability to write filters in a programming language like C, we provide to the filter designer a powerful interface to set up filters that previously required a lot of configurations. To illustrate this flexibility, we implemented a filter that is able to parse the AS-Path attribute to accept routes advertised by odd-numbered AS. This is certainly not "usable" on a real network configuration but demonstrates the simplicity of conceiving such filter. Listing 5.4 shows the C code to achieve this task. With only 4 lines of code, the filter is successfully implemented. Without the integration of this virtual machine, to reproduce the same filter, the BGP configuration must include an `AS access list` of 2^{15} ASes⁴ that will be checked each time a route is learned. Hence this induces a lot of overhead against a constant time check with eBPF filter.

5.4 Conclusion

This chapter enumerates the three kinds of eBPF plugins that have been implemented for BGP. Although we do not implement eBPF plugins at every critical location of BGP, our use cases prove that enhancing BGP through eBPF is easy. We also show that eBPF is used to overcome the lack of flexibility to design complex filters in a few lines of C code.

⁴Half of the number of total 16-bits AS identifier

```

1 #include "ubpf_tools/include/public_bpf.h"
2
3 /**
4  * ON IMPORT FILTER
5  * Allowing route received from odd ASes
6  * @return FILTER_ACCEPT if route has been learned from odd AS
7  *         FILTER_DENY if route has been learned from even AS
8  *         BGP_CONTINUE in case of error (OUT OF MEMORY)
9  */
10 uint64_t filter_router_aspath(bgp_input_filter_t *args) {
11     as_t a = from_as();
12     return a == 0 ? BGP_CONTINUE : (a % 2 == 0 ? FILTER_DENY :
13                                     FILTER_PERMIT);
14 }

```

Listing 5.4: Simplicity of creating complex filters

The API that communicates between eBPF bytecode and the BGP implementation is designed to open new possibilities as we have seen with the alternate form of hot-potato routing in the BGP decision process. We also provide RIB read access to BGP filters to expand their scope. This API is thus critical to allow network operators to extend as much as possible the protocol with an eBPF virtual machine.

Chapter 6

Performance Evaluation

Previous sections explained the insertion of a virtual machine inside a routing protocol to overcome limitations of the current implementations. As the virtual machine must be executed and thus integrated inside an existing implementation of BGP, more instructions are executed by the CPU when the code is run. This leads to an extra overhead. The question is then to quantify this overhead to assess the performance of our new virtual machine. This chapter evaluates the performances of our current implementation.

6.1 Structure of the Evaluation

In order to provide accurate performances, we need to consider the design of BGP. Indeed, the protocol is doing most of its job when it receives prefixes from other BGP speakers. For each received prefix, it runs the well-known **BGP Decision Process**, by testing whether the freshly received route is better than the ones already present in its routing table. To test the stability of the virtual machine, we need to think about a network configuration that will stress BGP. In other words, our protocol must receive a sufficiently large routing table from its neighbors to be able to observe its behavior under stress. [44] shows the actual FIB (Forwarding Information Base) for an average router located on the internet. At the time of this writing, approximately 800k routes are stored and known¹ by BGP routers of the Route-View project. This means that BGP has to process these 800k routes contained in multiple **BGP Update** messages. Hence designing a protocol able to process such a high quantity of messages as fast as possible is crucial.

In order to test the implementation, at least two routers must be used. If only one router is used to advertise its prefixes to our test router, the decision process is not correctly traversed. Indeed, since the test router does not know any route to any prefix sent by this unique router, the route will be instantly added to the Loc-Rib. However, if a second router, containing the same routing table is added to the neighbors of our test

¹On average

router, this latter will traverse the entire BGP decision steps by eventually comparing router identifiers. Thus we use the network topology shown in Figure 6.1 (a).

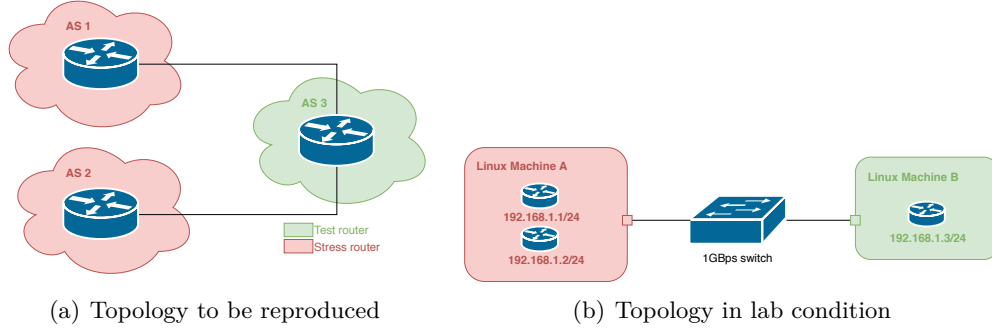


Figure 6.1: Network topology used for the evaluation phase.

This figure shows, however, a topology that might exist in the real Internet. Since we are not directly dealing with routers but Linux machines running BGP, the real topology is different and shown in Figure 6.1 (b). In fact, the two tests routers are running on a single machine isolated from the test machine to avoid having interferences when we collect statistics on the green router. To be more precise, the red machine uses BGPerf [45] based itself on ExaBGP [46]. Exabgp is a framework allowing users to define a BGP configuration via a simple markdown language. The particularity of this tool is its ability to transform BGP messages in human readable format like JSON and generate BGP messages with JSON. Thus, this project could be used to inject big routing tables on the target router by generating custom BGP Update messages.

BGPerf is using the power of ExaBGP to design a strong test architecture as described in Figure 6.2. It creates two different kinds of BGP instances. The first one uses ExaBGP to directly speak with the target router (i.e. the one containing the uBPF machine). According to the configuration passed to BGPerf, it creates one or more ExaBGP "routers" with a custom BGP routing table. In our case, we create two BGP routers with the exact same routing table (i.e. the same prefixes are advertised to the remote target).

The second type of BGP instance is used to record the updates of the remote target and forwarding the number of routes received on the main script in charge of building the entire topology. So far, the remote target must configure three BGP connections with the actual test framework. Two of them listen to incoming BGP update messages and the last one to announce the routes learned by the first two ExaBGP routers. With this current configuration, we can reach every insertion point of the uBPF machine.

In the classical version of BGPerf, it is possible to monitor system information of the target router because all processes run on the same machine. However, we do not want to have interferences from other processes when retrieving these data. So, the monitored

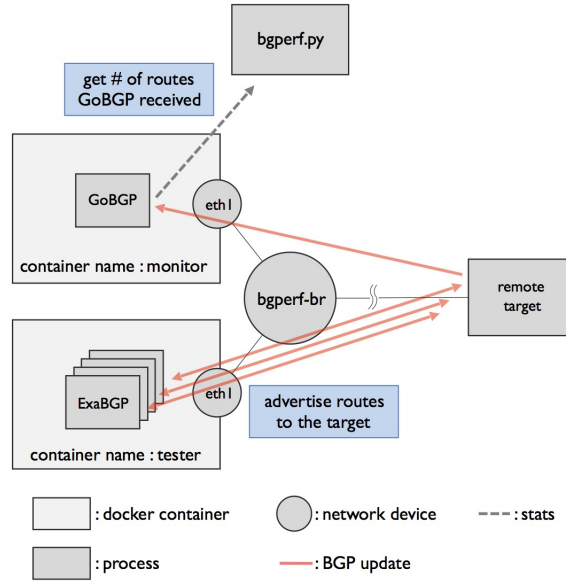


Figure 6.2: BGPerf structure. (Source: https://github.com/osrg/bgperf/blob/master/docs/benchmark_remote_target.md)

router is isolated in its own machine without any extra processes unless the ones to gather system utilization. Therefore inside the target router, some scripts are added to retrieve the current load of the monitored system (CPU, memory, context switches) to reproduce the original behavior of BGPerf. These scripts rely on `psutils` [47].

In addition to the scripts recording the system utilization of a given process, we also use the Linux tool called `perf` [48]. This is a fine-grained performance analysis tool that can retrieve hardware performance counters to analyze a given process. This is a special set of 40-bits wide CPU registers specially implemented to help software designers to optimize their programs. HPC (Hardware Performance Counters) can be leveraged to count a lot of events such as TLB misses, cache access, instructions executed per CPU cycles, etc. The basic utilization of these hardware counters is controlled by the Intel **x86** instruction `RDPNC` [49]. A user specifies the type of event specified in the **ECX** register and the CPU will copy the contents of the HPC into registers **EDX:EAX** (EDX contains the high-order 8 bits and EAX has the low-order 32 bits).

To avoid directly programming with assembly instructions, `perf` already contains a high level API to retrieve desired events from a running process. Furthermore, instead of printing the contents of these registers, it precisely finds the instructions affected by the counter by looking at the process source code. Among them, four hardware events are interesting :

- the instructions the most executed on the CPU;
- the instructions the most executed per CPU cycle;
- the instructions inducing the most cache misses;
- the instructions doing a lot of memory accesses.

The full list of events that could be recorded by **perf** can be obtained by executing the command **perf list** on a Linux terminal (**perf** must be installed first on the machine). The previous list is sufficient to compare the performances of the integration of the uBPF machine. We can directly find if the memory accesses or the number of instructions added with uBPF are bottlenecks for our implementation. Furthermore, these events are directly related to hardware. Hence we have precisely the behavior of both memory and executed operations.

Platform Used to Benchmark BGP With uBPF Enabled

We conclude this section by providing information about the machine used during the tests. Table 6.1 provides its configuration. As mentioned before, the machine contains a 1Gbps interface and is connected to a 1Gbps link.

Perf version	5.0.g1c163f4
OS Release	Linux 5.0.13
Linux Distribution	Arch Linux
# CPU cores	4
CPU description	Intel(R) Core(TM) i3 CPU 540 @ 3.07GHz
Clock Speed	3.07GHz
Total Memory (RAM)	12GB
CPU L1 Data	32KB
CPU L1 Instruction	32K
CPU L2 Cache	256K
CPU L3 Cache	4096K
Architecture	x86_64

Table 6.1: Characteristic of the computer used to perform the tests

6.2 Performance evaluation

Before doing tests with any part of the code including with an uBPF plugin, we need to have a baseline reference to compare our implementation against the original one. Figures 6.3 and 6.4 show in the dashed blue curve the time taken to process a routing table without any injected plugin. For tests of Figure 6.3, only one router of Figure 6.1 sends its 200K routing table to our "uBPF" router.

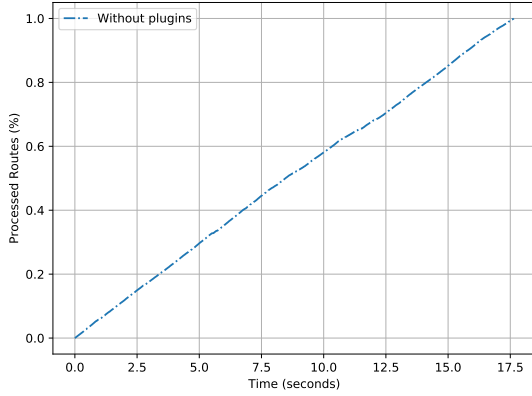


Figure 6.3: Performance of the BGP filters implemented as eBPF plugins.

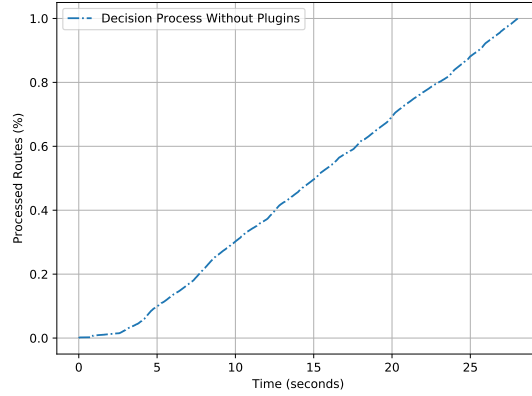


Figure 6.4: Performance of the BGP decision process implemented as eBPF plugins.

On Figure 6.4, as stated in Section 6.1, two routers are used to assess the performance of the decision process. To produce the curves depicted in this figure, we first wait that the first router sends its entire routing table before enabling the second router which advertises its 200k prefixes to our uBPF router. We then measure the time taken to process every route from the second router only. The two test routers advertise the same routing table composed of 200k IPv4 prefixes². Since the same routes are sent, the 14 decision steps are evaluated. The uBPF router eventually reaches the **Router-ID**, the last decision step to make a choice. As most of the job is performed when the two stress routers send their routing table, this period is only taken into account. The rest of the time, BGP will either wait for incoming messages or send keepalive messages. Thus this part is not relevant because nothing is computed inside BGP unless waiting for incoming messages from other peers.

The baseline time is not the same on these two figures since the tests are different. On Figure 6.3, the decision process does not run because no routes are known yet. This explains why the processing time is longer on Figure 6.4

In order to compute the overall processing time of routes received from the tests routers, we plug a monitoring plugin on the function advertising a forwarding entry in the **Linux Routing Entries** (FIB). The eBPF function only sends the time when the prefix is actually announced to the kernel. Listing 6.1 shows the C code used to retrieve some data about the advertised route in the Linux' FIB. The code of this eBPF function is JITted to guarantee the best performance of the protocol.

²This table is available on the Spotify's Github project **super-smash-brogp** <https://github.com/spotify/super-smash-brogp>

```

1 uint64_t pre_on_announce(zebra_announce_ubpf_t *args) {
2     ebpf_message_t msg;
3     struct announce_msg content =
4         {.clock = clock(), .prefix = args->cp_p.u.prefix4.s_addr};
5         // ^^^^^^ prefix contains the prefix of the
6                 route
7     memcpy(&msg.mesg_text, &content, sizeof(struct announce_msg));
8
9     send_ipc_msg(&msg); // send to external process a message
10    return EXIT_SUCCESS;
}

```

Listing 6.1: Monitoring filter used to retrieve processed routes

6.2.1 Evolution of the time by the number of plugins attached to the protocol

Before focusing our attention to the performance of the protocol per use cases, we will increasingly plug eBPF bytecode of different uses case in BGP and observe the overhead induced by the addition of further eBPF programs. Figure 6.5 reports the time taken

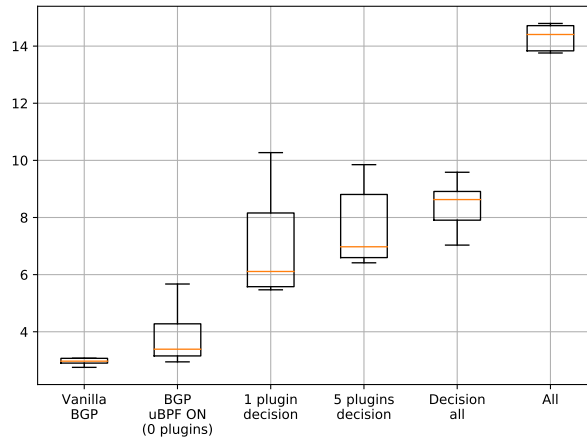


Figure 6.5: Overhead of running different plugins (over 15 runs) in JITed mode

to process incoming BGP updates for one up to 25 attached plugins. Tests are made for both interpreted and JITted mode. Information about eBPF plugins can be found in Table 6.2. We can see that the insertion of data structure manipulating the uBPF machine slightly increases the execution time compared to the vanilla implementation, i.e. the original FRRouting implementation. This is due to the fact that more instructions are performed. For each functions with an insertion point, it must check if there are

eBPF Function	External Calls	eBPF Insts	ELF (Bytes)	LoC
Weight Check	11	47	1856	37
Local Pref	21	85	2488	51
Local Route	14	42	1800	25
As Path	19	91	2624	52
Origin Check	9	41	1680	27
MED Check	25	147	3376	58
IGP Weight	38	338	6280	134
Multi Path check	15	66	2200	38
Prefer already selected route	29	103	2784	48
Router ID cmp	26	110	2896	54
Cluster ID cmp	24	97	2736	53
Neighbor addr cmp	9	52	1816	38
Confed Check	27	94	3024	50
[Extra] BGP Communities cmp	24	203	4664	148

Plugins	External calls	eBPF Insts	ELF (Bytes)	LoC
FIB Announce	2	12	1184	14
Decision Process Time	13	59	3702	41
Rejected Route	7	81	2552	31
Keepalive Msg	5	49	2088	28
Update Msg Time	13	71	3920	45
Open Msg	16	93	4472	50
Update Msg Contents	11	123	3088	45
Received AS Path	1	10	1160	5
Filter Redundant Routes	4	35	1696	19
Withdraw content	4	55	2120	24
Accept Odd AS	1	14	1200	5

Table 6.2: Data about eBPF code for use cases. The decision process plugins are shown in the left table and use case about monitoring and filters are listed in the right table.

some attached plugins.

The insertion of one plugin of the decision steps increases the execution time by two. The eBPF function run is the one computing checking the IGP weight. This is, according to Table 6.2, the heaviest plugins in terms of lines of code and external calls. However, the length of the eBPF code is not related to the overhead. Indeed, if we pluginize the first five steps of the decision process (i.e. the first five plugins of Table 6.2), an increase of almost 1 second is noticed. Furthermore, if every plugin computes all the steps of the decision process, the overhead is not much higher in JITed mode.

Thus from one to 15 decision steps plugins, the BGP update message processing time is quite stable. Approximately 2.6 seconds are added to the total execution time. A deeper understanding of the hardware event with **perf** tool indicates that the overhead comes from both the uBPF memory manager and the retrieval of plugins in our data structures. First of all, each argument must be first copied into the extra memory defined in Section 4.4. In addition, all retrieved variables must also be copied in this memory. The virtual machine must first check if the plugin has the right accesses to the protocol variable. If it is the case the uBPF machine also verifies if the pointer given at arguments of "getter" functions are within the bounds of the plugins (i.e. whether the pointer is sufficiently large enough to host the variable and fits in the extra memory granted to it).

Second, retrieving the plugin code in the data structure induces a lot of "cache misses". By looking at the results generated with **perf**, there are 68% more cache misses on average with only the "IGP Weight check" plugins activated. Having more cache misses induces more delays when accessing to the data because the CPU must fetch data from main memory (or other cache levels) to the primary CPU cache. This is especially true when the CPU must execute instructions of JITed plugin. Indeed, the code of each plugins are separated from the "real" code of FRRouting. JITed codes are located on a memory area allocated by **mmap** and the protocol one is located in the **.text** segment of the process memory. Therefore, the plugin code, located in a different memory area must first be loaded before the CPU executes its instructions.

Third, each plugin is doing more memory accesses than regular BGP functions. If the eBPF bytecode would like to request a variable from the protocol, the virtual machine must ensure that it has the right accesses, make some checks and then perform a read only copy into the plugin memory area (see Section 4.7). Dealing with memory causes a drop of the overall performance of the running process and could be observed with the number of memory accesses done by the program. The memory is accessed 4 times more than the original protocol. This is due to our memory manager. When a plugin wants to access to a protocol variable, it must allocate a space to the extra memory. We have identified that the simple linked-list used by the memory manager drops the performances because it must scan in a linear time the memory bloc (heap) to find a free one.

This memory access bottleneck could be observed when recording the number of instructions retired per processor cycle. In other words, this HPC is actually counting the number of instructions **completely** executed between two clock ticks [50]. This counter measures the latency triggered by specific instructions in the program because of cache misses, I/O, memory accesses, etc. Without any surprises, the slowest part of the code is generated by our uBPF API doing most of the check about memory integrity and memory copy. Among the slowest functions of this API, the one making the call of the plugins also induces latency. This function is actually performing multiple accesses to the central memory :

1. flush the memory arguments of a previous eBPF execution;
2. copy arguments to the authorized memory area;
3. executing eBPF code located to a non-standard part of the memory;
4. clean the memory taken by the executed eBPF hook.

In fact, the API interacting with the virtual machine leads to a significant augmentation of the execution time due to these memory bottlenecks.

The last test of Figure 6.5 shows an interesting result when plugins are loaded at every eBPF insertion point implemented in the protocol. The general tendency of the real execution time ought to be quite stable. However, we can clearly observe a significant gap up to 14 second. An analysis of the CPU load informs us that the process is waiting most of the time, as Figure 6.6 could depict. The red curve shows the average CPU load of the

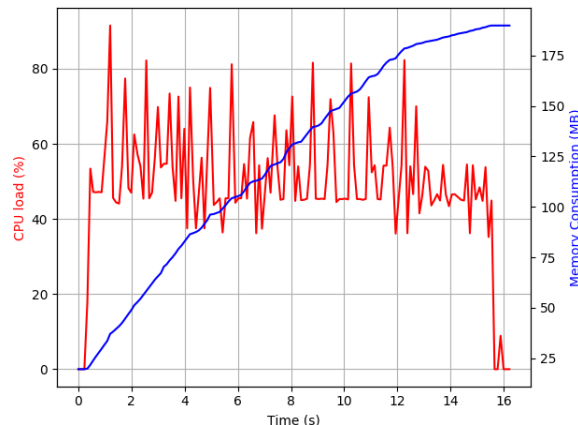


Figure 6.6: CPU load and memory consumption of the protocol with all insertion points populated by eBPF bytecodes.

four threads running to the BGP process. This curve shows that on average 2 threads are running and 2 are sleeping. A new factor must be taken into account. The BGP **FRRouting** implementation is by design multithreaded. There is a thread to read packets coming from the network, one thread to write data for BGP neighbor, one to handle the protocol and the last one to listen to new plugins to be injected to the protocol.

Our current implementation of the virtual machine imposes that only one eBPF function is executed at once on the system (See Section 4.7). To avoid race conditions between two plugins, a lock is acquired by a thread which will eventually be released at the end of the eBPF execution. Waiting threads cannot continue their execution and are thus limited by the actual design of the virtual machine. This consequence is even more visible on the right part of the graph with the blue curve showing the memory consumption. From four seconds before the start of the stress, the curve has a "staircase" shape. This means that the local routing table is not filled during this short amount of time, therefore, the protocol is either receiving BGP updates from neighbor BGP speakers or computing the BGP decision process, but cannot handle both at the same time. The consequence of this mono thread limitation caused by the virtual machine impacts by a factor of approximately two the performance of the protocol.

6.2.2 Performance of Monitoring Plugins

This section gives an analysis of performance of the monitoring part use case. As is the previous section, we focus our attention on the general performance of the insertion of monitoring plugins in BGP.

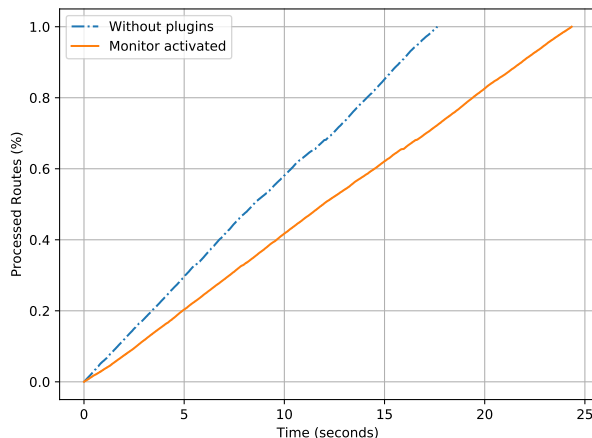


Figure 6.7: Performance of eBPF monitoring

Each plugin calls an external function that will send data passed as argument through a UNIX pipe. This call is then a function doing the interaction with the external process. On Linux systems to interact with an external process, several options are valid. They both have their own advantages and drawbacks. On the first version of the virtual machine, the used method was to communicate through a UNIX socket on which the monitoring daemon was listening once FRRouting process establishes a connection to transmit data. Time taken to send monitoring messages was too slow because of (1) the overhead of the transmission protocol used, and (2) the lack of buffering when transmitting data. Indeed, TCP segments are directly sent to the monitoring daemon without taking any benefit or any buffering. Hence, the system call function ‘`send`’ [51] (used to transmit data blocks) until the data is correctly passed to the external process. This undesired behavior might be fixed with the introduction of a sufficiently large buffer to limit the useless waiting time. This is the main reason of introducing UNIX pipes in the architecture of our monitoring system. The kernel itself keeps a buffer of default size of 1MB³ [52]. The operation is then non-blocking as long as the buffer is not completely filled. Figure 6.7 shows one run when eleven filters are activated in the orange curve. Even if there is a buffer preventing most of blocking induced by a lack of space in buffers, writing into the buffer requires to copy variables into kernel before continuing the execution of the user space program. Again, by looking at the slowest instructions with `perf` tool, it can be noticed that the kernel `pipe_write` function takes

³Since Linux 2.6.35, the maximum size of pipe kernel buffer is located on `/proc/sys/fs/pipe-max-size`

part of the slowest instructions of BGP.

The inter-process communication is the main bottleneck on this use case because of the necessity to pass on the kernel space before communicating with the second process. This process induces a lot of voluntary context switches when switching between user and kernel mode. Indeed, the CPU does a lot of actions such as flushing the TLB, recover the context of the kernel process by restoring the values in the registers, and update structures concerning the data related to the process. These operations influence the performance of BGP protocol.

6.2.3 Performance of the Decision Process Plugins

Another interesting use case to benchmark is the "pluginization" of the BGP decision process. Section 5.2 explained the detailed internal process to make this important part of the protocol as flexible as possible. The figure 6.8 shows the results of one run of our new flexible decision process in the orange curve.

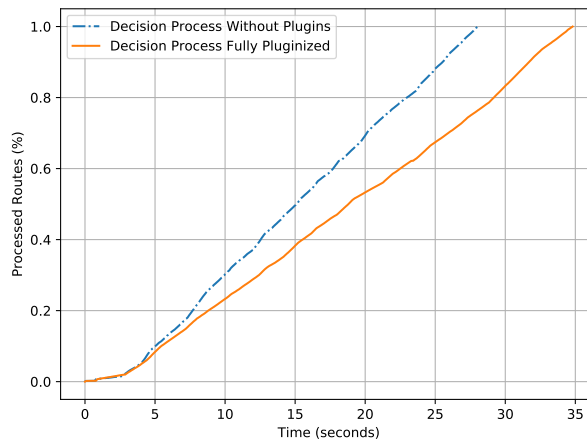


Figure 6.8: Performance of the BGP Decision Process Implemented as eBPF

As for every uses case, the overhead is also due to the memory management doing checks for security reasons and looking for available free blocks in $\mathcal{O}(n)$ time to copy variables into the plugin memory. However, a new interesting constraint must be taken into consideration. Since the full BGP decision process is replaced by eBPF bytecodes, the virtual machine, must load instructions into another part of the memory and then copy arguments on the uBPF memory. Furthermore, the BGP decision process uses BGP attributes and must then request a lot of data structure in the protocol. This increases the global execution time of the routing table computation.

On the original BGP `FRRouting` code, the BGP decision process is a single large function including all steps. As the procedure is quite important in BGP, its execution must be as fast as possible. The `FRRouting` optimization then includes every sub-steps inside a single function and avoids adding extra instructions linked with the call of function as we are doing with our new design of BGP decisions step. To call a function, the process must load the arguments on the execution stack, save the current stack pointer and then make a jump to the address of the function's first instruction. If BGP must make these extra instructions for every 15 BGP steps per prefixes received, the performance will significantly decrease, as Figure 6.8 proves with the two curves. When the decision process is fully pluginized we have an overhead of 34.8%, compared to the one without any plugins (the dashed blue curve). First because there is more executed instructions, and second because the memory is more accessed as previously explained.

6.2.4 Performance of eBPF filters

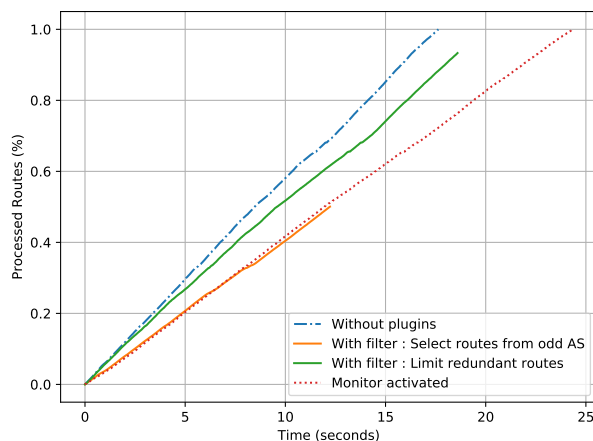


Figure 6.9: Performance of the BGP filters implemented as eBPF plugins.

Figure 6.9 contains the evaluation of the two filters described in Sections 5.3.1 and 5.3.2. The orange curve represents the number of processed routes by the filter removing the ones originated from odd ASes. The graph actually shows that half of routes are sent to the FIB. This is the correct behavior since ASes inside the AS-Path associated with routes are randomly generated from 0 to 2^{16} . The execution time is then lower than the baseline one because the protocol has to process fewer routes and thus has to perform less computations.

The green curve represents the time taken when the filter limiting redundant routes inside the router's RIB. An overhead of 5.4% is noticed against the baseline dashed blue curve. However, it can be seen the filter is doing its job correctly because it has rejected 13k of the 200k advertised routes. Given that the filter actually performs 35 eBPF instructions

including 4 external calls (see Table 6.2), the total execution time is slightly increased.

6.2.5 A Quick Performance Analysis With the VM Interpreted Mode

We close the performance analysis by making a short comparison with the interpreted mode of the virtual machine. So far, the protocol was executing all the plugins with the JIT part. This means that the eBPF instructions are directly executed on the CPU without decoding the eBPF file each time this has to run as for the interpreted mode. Obviously, the total number of instructions executed per eBPF code will increase and thus should take more time to execute an eBPF program. Furthermore, that the eBPF byte-code must be proofread each time it is executed. The program must then retrieve the instructions from the memory, which might lead to cache misses. Figure 6.10 confirms our

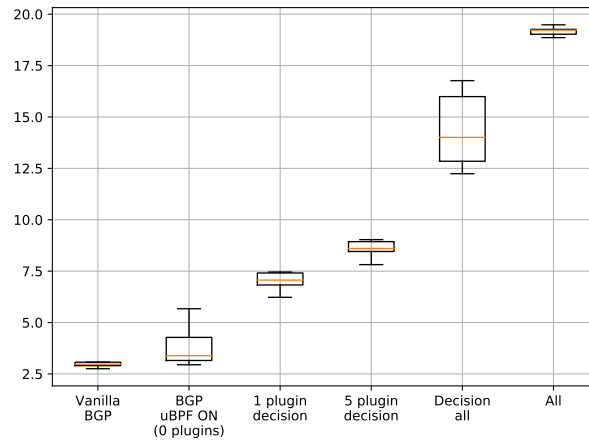


Figure 6.10: Overhead of running different plugins (over 15 runs) in interpreted mode

hypotheses. With every activated plugin, the total execution time significantly increases. The `perf` tool indicates that the slowest function in the code is `ubpf_exec()`, the uBPF API doing the interpreted execution of the bytecode.

The interpreted mode contains thus every flaw of the JITed execution but also adds an extra overhead with the rereading of the bytecode eBPF instructions each time the program is called.

6.3 Conclusion

This chapter studied the impact of the integration of a virtual machine inside BGP. We saw that including it induces a non-negligible impact on the overall performances of the protocol. Our study proves that our prototype still contains some flaws that must be solved. The memory manager is still weak and must be revised to significantly increase the performances of the protocol. Furthermore, the design of the eBPF manager has to be rewritten in order to support multi-thread implementation such as FRRouting. To be

fully usable, our virtual machine should add as less overhead as possible and must not add extra useless execution constraints on the original implementation.

Ideally, to support this idea, the performance of our plugins should be as close as possible to native code. We compared the two execution modes of the uBPF VM. JIT seems to be better in terms of performance. However, its lack of security induces that run-time checks must be done. However, it accordingly decreases the performance of the implementation. Several alternatives might be taken such as (1) using our uBPF machine on a special processor architecture like **Chery-Mips** [53] that provides memory checks on the CPU directly or (2) implementing a powerful static analyzer that refuse to load the plugin if it detects bad memory accesses.

Chapter 7

Conclusion

This master's thesis describes how to improve IP routing protocols through the integration of a virtual machine capable of executing plugins.

We demonstrated the advantages of the virtual machine on the Border Gateway Protocol (BGP) through several use cases, going from monitoring to the redesign of a static, well-defined and standardized procedure : the BGP decision process. Through several use cases, we also proved that the protocol is extensible by implementing a new form of hot-potato routing, using the geographical distance between routers used to decide which route to take during the decision process. Finally, we demonstrated that the power of a Turing machine could offer new possibilities such as creating importation filters that previously were hard to build.

To be able to design these use cases, the original implementation of the uBPF virtual machine has been revised in order to fit the requirements of a protocol. eBPF functions belonging to the same plugin are isolated and cannot access what the protocol exposes to eBPF bytecode. In addition, the plugin memory has been secured to guarantee its integrity, to avoid crashes, memory leaks and plugin interferences. The virtual machine has been extended to provide a mechanism to modify a protocol to insert multiple insertion points and anchors into BGP in the easiest way possible. Finally, we added a manager capable of adding, replacing and removing eBPF plugins while BGP is running.

Discussion

Our uBPF API does not provide a mechanism to check the integrity of an eBPF bytecode before loading it into the virtual machine. This lack of static analysis prevents network operators to automatically deploy plugins. A human must first validate the plugin before its insertion, and check whether it has programming errors or not. There is actually a bytecode analyzer included in the uBPF machine. However, this one is less competitive than the eBPF Linux Kernel verifier [\[54\]](#) and does not accurately verify the presence of an infinite loop, ill memory management, memory overflows, etc. This type of verifier

could allow a network operator to deploy, with a centralized mechanism, all its eBPF plugins.

Even though eBPF is platform independent, every router must expose the same API with the same insertion points. This framework is valid if an operator owns routers of the same vendor. This is, however, rarely the case. Vendors must agree on the same virtual machine API in order to be fully capable of supporting plugins written by operators.

There is a trade-off between flexibility and performance. The virtual machine could execute plugins based on two modes: JIT or interpreted. In interpreted mode, the flexibility is higher because the security concerns are better managed. The virtual machine could have a better understanding of what the plugin is trying to do. In JIT mode, the performances are closer to native code but do not offer the flexibility to apply complex security checks. Our workaround was to directly rewrite the eBPF bytecode to ensure memory protection by using a register to store the beginning of the extra heap. However, we prefer to have both flexibility and performance, and thus opt for an architecture capable of fulfilling both requirements.

Future Work

The realized work is still in a prototype stage. It provides a foundation of a "gray-box" networking approach. To be fully viable, some development must be done to improve the performance of the current virtual machine. Some ideas have been presented at the end of Chapter 6. The virtual machine must not add extra overhead that is useless to the correct functioning of the protocol (e.g. checks to guarantee the integrity of an eBPF bytecode). We believe that using a new secured CPU architecture or designing a powerful static eBPF analyzer could be a good approach.

We proved that using a virtual machine inside a routing protocol opens new opportunities for networks. This is why we encourage the network community to reproduce or continue our work. The prototype, as well as our patches regarding the BGP FRRouting version 6.1 implementation are available on https://bitbucket.org/twirtgen/frr_ubpf/. For confidentiality reasons, the repository is now stored in closed sources until August 2019. We believe that new features can be added if network operators are interested to contribute to what has been started in this thesis.

Bibliography

- [1] J.D. Case, M. Fedor, M.L. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP). RFC 1157 (Historic), May 1990. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc1157.txt>.
- [2] R. Enns (Ed.), M. Bjorklund (Ed.), J. Schoenwaelder (Ed.), and A. Bierman (Ed.). Network Configuration Protocol (NETCONF). RFC 6241 (Proposed Standard), June 2011. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc6241.txt>. Updated by RFCs 7803, 8526.
- [3] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, 1993.
- [4] IO Visor Project. Userspace ebpv vm. <https://github.com/iovisor/ubpf>, . Accessed: 2019/06/06.
- [5] Matt Fleming. A thorough introduction to ebpv. Linux Weekly News, December 2017. <https://old.lwn.net/Articles/740157/>.
- [6] Tcpdump. Tcpdump. <https://www.tcpdump.org/>. Accessed: 2019/06/10.
- [7] Matthew Luckie and Robert Beverly. The impact of router outages on the as-level internet. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 488–501. ACM, 2017.
- [8] E. Chen (Ed.), J. Scudder (Ed.), P. Mohapatra, and K. Patel. Revised Error Handling for BGP UPDATE Messages. RFC 7606 (Proposed Standard), August 2015. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc7606.txt>.
- [9] J. Scudder and R. Chandra. Capabilities Advertisement with BGP-4. RFC 5492 (Draft Standard), February 2009. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc5492.txt>.
- [10] Albert Greenberg, Gisli Hjalmtýsson, David A Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4d approach to network control and management. *ACM SIGCOMM Computer Communication Review*, 35(5):41–54, 2005.

- [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. USENIX Association, 2005.
- [12] 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.
- [13] Diego Kreutz, Fernando MV Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.
- [14] J. Moy. OSPF Version 2. RFC 2328 (Internet Standard), April 1998. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc2328.txt>. Updated by RFCs 5709, 6549, 6845, 6860, 7474, 8042.
- [15] R. Housley. Internet Numbers Registries. RFC 7249 (Informational), May 2014. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc7249.txt>.
- [16] IANA. Autonomous system (as) numbers. <https://www.iana.org/assignments/as-numbers/as-numbers.xhtml>, December 2018. Accessed: 2019/05/01.
- [17] Y. Rekhter (Ed.), T. Li (Ed.), and S. Hares (Ed.). A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc4271.txt>. Updated by RFCs 6286, 6608, 6793, 7606, 7607, 7705, 8212.
- [18] Benoit Donnet and Olivier Bonaventure. On bgp communities. *ACM SIGCOMM Computer Communication Review*, 38(2):55–59, 2008.
- [19] E. Chen and T. Bates. An Application of the BGP Community Attribute in Multi-home Routing. RFC 1998 (Informational), August 1996. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc1998.txt>.
- [20] Philip Smith and Barry Greene. Using bgp communities, February 2017. <https://nsrc.org/workshops/2017/apricot2017/bgp/bgp/preso/09-BGP-Communities.pdf>. Accessed: 2019/05/02.
- [21] R. Chandra, P. Traina, and T. Li. BGP Communities Attribute. RFC 1997 (Proposed Standard), August 1996. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc1997.txt>. Updated by RFC 7606.
- [22] G. Malkin. RIP Version 2. RFC 2453 (Internet Standard), November 1998. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc2453.txt>. Updated by RFC 4822.

- [23] D. Oran (Ed.). OSI IS-IS Intra-domain Routing Protocol. RFC 1142 (Historic), February 1990. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc1142.txt>. Obsoleted by RFC 7142.
- [24] T. Bates, E. Chen, and R. Chandra. BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP). RFC 4456 (Draft Standard), April 2006. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc4456.txt>. Updated by RFC 7606.
- [25] P. Traina, D. McPherson, and J. Scudder. Autonomous System Confederations for BGP. RFC 5065 (Draft Standard), August 2007. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc5065.txt>.
- [26] Marc-Olivier Buob, Steve Uhlig, and Mickael Meulle. Designing optimal ibgp route-reflection topologies. In *International Conference on Research in Networking*, pages 542–553. Springer, 2008.
- [27] Rohit Dube. A comparison of scaling techniques for bgp. *ACM SIGCOMM Computer Communication Review*, 29(3):44–46, 1999.
- [28] John G Scudder and Rohit Dube. Bgp scaling techniques revisited. *ACM SIGCOMM Computer Communication Review*, 29(5):22–23, 1999.
- [29] syscall(2) linux user’s manual, April 2018.
- [30] W Richard Stevens and Stephen A Rago. *Advanced Programming in the UNIX Environment: Advanc Progra UNIX Envir_p3*. Addison-Wesley, 2013.
- [31] Free Software Foundation. Extensions to the c language family – nested functions. <https://gcc.gnu.org/onlinedocs/gcc/Nested-Functions.html>. Accessed: 2019/06/06.
- [32] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031 (Proposed Standard), January 2001. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc3031.txt>. Updated by RFCs 6178, 6790.
- [33] Alvaro Retana and Russ White. Bgp custom decision process. Internet-Draft draft-ietf-idr-custom-decision-08, IETF Secretariat, February 2017. URL <http://www.ietf.org/internet-drafts/draft-ietf-idr-custom-decision-08.txt>. <http://www.ietf.org/internet-drafts/draft-ietf-idr-custom-decision-08.txt>.
- [34] FRRouting Project. The frrouting protocol suite. <https://github.com/FRRouting/frr>, . Accessed: 2018/11/26.
- [35] L. Andersson (Ed.), I. Minei (Ed.), and B. Thomas (Ed.). LDP Specification. RFC 5036 (Draft Standard), October 2007. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc5036.txt>. Updated by RFCs 6720, 6790, 7358, 7552.

- [36] Donald Sharp. Quagga 1.1 → frr 2.0. <https://github.com/FRRouting/frr/wiki/Quagga-1.1-%E2%86%92-FRR-2.0>. Accessed: 2018/12/16.
- [37] FRRouting. Use kernel threads for bgp packet writes & keepalive generation #302. <https://github.com/FRRouting/frr/pull/302>. Accessed: 2018/12/16.
- [38] Martin Winter. One year of frrouting. https://ripe76.ripe.net/presentations/125-one_year_FRR.pdf, May 2018. Accessed: 2018/12/16.
- [39] Cumulus Networks. What is FRRouting? <https://cumulusnetworks.com/learn/frrouting/>. Accessed: 2018/12/16.
- [40] B. Claise (Ed.), B. Trammell (Ed.), and P. Aitken. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. RFC 7011 (Internet Standard), September 2013. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc7011.txt>.
- [41] S. Sangli, D. Tappan, and Y. Rekhter. BGP Extended Communities Attribute. RFC 4360 (Proposed Standard), February 2006. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc4360.txt>. Updated by RFCs 7153, 7606.
- [42] J. Heitz (Ed.), J. Snijders (Ed.), K. Patel, I. Bagdonas, and N. Hilliard. BGP Large Communities Attribute. RFC 8092 (Proposed Standard), February 2017. ISSN 2070-1721. URL <https://www.rfc-editor.org/rfc/rfc8092.txt>.
- [43] Routeviews. University of Oregon Route Views Project. <http://www.routeviews.org/routeviews/>. Accessed: 2019/05/26.
- [44] Bgp statistics from route-views data. <http://bgp.potaroo.net/bgprrpts/rva-index.html>. Accessed: 2019/05/12.
- [45] osrg. bgperf. <https://github.com/osrg/bgperf>. Accessed: 2019/05/12.
- [46] Exa Networks. exabgp, the bgp swiss army knife of networking. <https://github.com/Exa-Networks/exabgp>. Accessed: 2019/05/12.
- [47] Giampaolo Rodola'. Pstutils. <https://github.com/giampaolo/psutil>. Accessed: 2019/05/12.
- [48] perf(1) linux user's manual, March 2019.
- [49] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 2B: Instruction Set Reference*, 2019.
- [50] Intel®. Instructions retired event. <https://software.intel.com/en-us/vtune-amplifier-help-instructions-retired-event>. Accessed 2019/05/13.
- [51] send(2) linux user's manual, September 2017.

- [52] pipe(7) linux user's manual, September 2017.
- [53] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468. IEEE, 2014.
- [54] 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. 2019.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl