UCLouvain

École polytechnique de Louvain

A flexible implementation of OSPF

Author: **Cyril DÉNOS**
Supervisor: **Olivier BONAVENTURE**
Readers: **Mathieu JADIN, Raziel CARVAJAL GOMEZ**
Academic year 2018–2019
Master [120] in Computer Science and Engineering

Table Of Contents

Abstract

Routing protocols are at the core of every today's network. Network operators needs are evolving, but on the other hand, routing protocols implementations are not very flexible by design. This leads some operators to turn themselves toward solution such as Software Defined Networks that centralizes the control of the networks, allowing providing more complex services. In this thesis, we propose a new approach for implementing routing protocols and demonstrate its applicability on OSPF. We add a virtual machine into the protocol implementation that is able to execute some code written by a third party (most probably network operators). We show that with this approach, it is possible to add a new type of LSA and modify the Shortest Path First algorithm of OSPF based on this new LSA in a very simple manner. Our results also show that the overhead induced by the introduction of the virtual machine and the execution of the added code is acceptable.

Acknowledgments

I would like to thank my supervisor, Olivier Bonaventure, for his consistent guidance and good advises. I also would like to thank Mathieu Jadin and Quentin De Coninck for their availability to answer all my questions and help me. Finally, I would like to thank Thomas Wirtgen that was doing his master thesis at the same time, for his countless explanations.

Repository link

The following link is the repository with the code developed in the frame of this master thesis. The repository is private but will be public in August 2019 at the latest.

https://github.com/cyrden/master_thesis.

Chapter 1

Introduction

Routing protocols are at the core of every today's networks. They enable routers to exchange information that allow them to select the best routes for packets between nodes in an IP network. We distinguish Interior Gateway Protocols (**IGP**) that are used to exchange routing information between routers inside an Autonomous System (**AS**) and Exterior Gateway Protocols (**EGP**) that are used to exchange routing information between different ASes. OSPF [30] is a widely used distributed IGP routing protocol, particularly in large enterprise networks. Thanks to its distributed nature, it is able to quickly react to router/link failure [21].

More and more devices are connected to the internet every day, making the routing of networks very challenging. Most of the routing protocols in use have been designed in the late 90s. Even though they have evolved and are still performing well, networks requirements have changed and are still changing continuously [40]. Control plane protocols implementations are not very flexible by design, making network operators to sometimes have difficulties to match their requirements. It is for example impossible with current OSPF implementations for a network operator to add a new type of LSA or to tune the Shortest Path First algorithm to take other metrics than the IGP cost into account. Some big companies/network operators may have some influence on protocol developers to lead the future extensions to match their needs but it is not the case for smaller structures.

Researchers have put some efforts into yielding flexibility to routing protocols. As an example, active networks were proposed in the late 90s to bring flexibility into the control plane [45]. In active networks, packets are carrying some pieces of code that are executed on the nodes to dynamically modify the behavior of the network. But this solution was not further used in practice, mostly because of the related security concerns [42]. Indeed, processing code received in packets from unknown source comes with challenges about the safety and correctness of the execution of that code. Then Software Defined Networks (**SDN**) were proposed [27]. In SDNs, the control plane is centralized, allowing network operators to provide more complex services. On the other hand, the failure tolerance of SDNs has never reached the one from distributed routing networks.

Because of all these considerations, we want to propose a solution for routing protocols implementations themselves to provide flexibility. We would like to benefit from the flexibility of SDNs while keeping the failure tolerance of traditional routing protocols. In this thesis, we propose a general solution to make routing protocols implementations extendable. We show a way of doing so by proposing a new flexible OSPF implementation that gives the possibility of easily modifying the behavior of some key operations of the protocol. We extend OSPF with a virtual machine that is capable of executing **plugins**. These plugins are pieces of software that can extend the protocol or modify its underlying

algorithms. OSPF flexibility and extensibility has been criticized a lot and some network providers are moving to BGP for intra-domain routing because the BGP complexity provides more possibilities [28] (especially to implement routing policies). We will show that our flexible OSPF implementation can overcome many of the current limitations in a very simple manner.

Structure of the thesis

This master thesis will be organized as follows:

- In Chapter 2 we will go through the classical OSPF protocol operation, see what are the main building stones that composes it and describe their actions. We will also present the FRRouting OSPF implementation which is the one we used for the purpose of this work.
- In Chapter 3 we will describe how we modified the OSPF implementation to make it flexible. We will also describe the interface we provide to make developers able to interact with the protocol and tune it as they wish.
- In Chapter 4 we will show some interesting use cases that can be achieved thanks to our flexible OSPF implementation. We will show how it can be used to easily monitor the protocol. We will also show how it allows modifying the behavior of OSPF. For that purpose, we will add a new type of LSA to the implementation and modify the Shortest Path First algorithm based on this new type of LSA. Finally, we will analyze the performances of our flexible OSPF implementation and compare it with the vanilla OSPF from FRRouting.
- In Chapter 5, we will discuss the main issues of our implementation and give some tracks for possible future work.

Chapter 2

OSPF

The Open Shortest Path Protocol (**OSPF**) is the standard Interior Gateway Protocol (i.e. it operates inside a single AS) that can be used with both IPv4 (OSPFv2 [31]) and IPv6 (OSPFv3 [9]). It belongs to the link-state routing protocols class and is widely used in ISP backbones (even if IS-IS [34] is a good competitor) and enterprise networks. In OSPF, each router that belong to the AS builds an entire view of the network topology. This view is a directed graph with the routers being vertexes and the links between routers being unidirectional links. OSPF also implements the Dijkstra algorithm, also called Shortest Path First (**SPF**) algorithm, to compute the shortest path tree with itself as the root. In the following, both Dijkstra and SPF algorithm denomination will be used to refer to this process.

This Chapter will describe the general operation of OSPF and give more details about the main building components of the protocol. We will also list these main building blocks (we will call them **protocol operations**) in the optics of allowing their extension afterwards.

2.1 General operation

The OSPF protocol uses a range of mechanisms that allow detecting, describing and propagating state through a network to be able to know the shortest route toward every possible destination. The basic idea is for routers to exchange information about the network topology by sending Link-State Advertisements (**LSAs**), and then compute the shortest path tree for that topology. This shortest path tree is computed using the Dijkstra algorithm based on some cost on each link (this cost is usually proportional to the bandwidth of the link but it can also be linked to the delay, jitter, etc., depending on what is important in the network). This weight is administratively assigned in the configuration file of the router.

Each router implementing OSPF maintains a link state database (**LSDB**) that describes the topology map of the network. This LSDB is a concatenation of all the LSAs a router has received. From that database, each OSPF router extracts routing and topology information and stores it in three tables [6]:

- Neighbor table: stores information about OSPF neighbors (thanks to the Hello protocol).
- Topology table: stores the topology structure of a network (learned by exchanging LSAs with other routers).
- Routing table: stores the best routes (computed by applying Dijkstra on the topology).

An OSPF network can be divided into different areas to simplify administration and optimize traffic and resource utilization. These areas are identified by an integer (usually

written as an IPv4 address), and all areas must be connected to the backbone area via an Area Border Router. All the routers inside an area have detailed information about the topology of this area (they all share the same LSDB) but only learn aggregated information about the topology of the other areas and their interconnections. This limits the number of LSAs that a router needs to store and to treat, which is nice when the network is large.

2.2 OSPF packet types

To communicate with each other, routers exchange OSPF packets. All OSPF packets share a common header of 24 bytes (Figure 2.1).

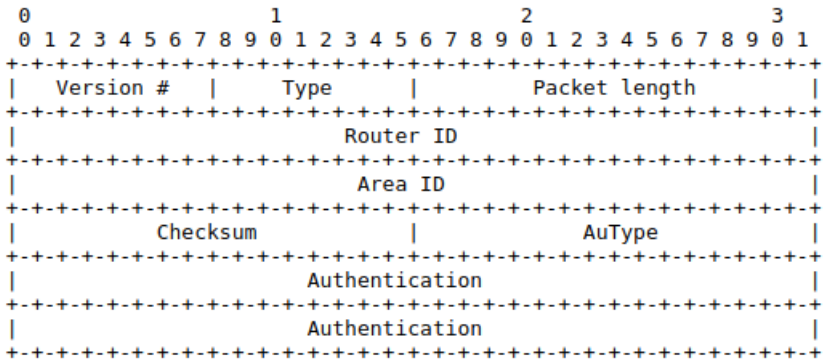


Figure 2.1: OSPF packet header [4].

There are 5 different types for an OSPF packet that determine the form of the body of the packet:

- Type 1: OSPF Hello message: used to discover other adjacent routers on its local links and networks.
- Type 2: OSPF Database Description message: contains descriptions of the topology of the AS or area. It conveys information about the content of the link-state database (LSDB) for the area from one router to another (it only contains LSA headers, not the actual content of the LSAs).
- Type 3: OSPF Link State Request message: used by one router to request updated information about a portion of the LSDB from another router. It specifies the link for which it wants more information (it contains the LSA headers of the LSAs the router wants to have).
- Type 4: OSPF Link State Update message: contains the LSAs that describes the topology. It is sent as a response to a LSR (it contains actual LSAs).
- Type 5: OSPF Link State Acknowledgment message: provides reliability to the link-state exchange process, by explicitly acknowledging receipt of a LSU (it contains LSA headers).

Other information carried in the packet header are: the OSPF version, the length of the packet, the ID of the sender router, the ID of the area the packets belongs to, a checksum and some information about the authentication scheme.
The exchange of all those packets between routers allow them to learn the needed information about the network topology they are part of.

2.3 Hello Protocol

The Hello protocol is used to maintain the adjacency state between neighbors. When a router starts an OSPF routing process on an interface, it sends a Hello packet (OSPF packet type 1) with a multicast destination address. It keeps sending those packets at regular intervals for its whole lifetime. In the default setup the interval between two Hello packets is set to 10 seconds. The dead-interval is the time after which a link is considered to be down by OSPF. It usually lasts 4 times the hello interval (40 seconds for the default value). Hello packets allow routers to know which other OSPF enabled routers are adjacent to them. When the adjacency between two routers is formed, the routers can start exchanging information with each other.
For scalability reasons, when some OSPF routers connect on the same Local Area Network (LAN), they elect a Designated Router (DR) and a Backup Designated Router (BDR) in case of failure of the DR. DR and BDR serve as the central point for exchanging OSPF routing information. Each non-DR or non-BDR router will exchange routing information only with the DR and BDR, instead of exchanging updates with every router on the network. DR will then distribute topology information to every other router inside the same area, which greatly reduces OSPF traffic [5].

2.4 OSPF LSAs

Link-State Advertisements are the core objects of the OSPF protocol. LSAs communicate the router's local routing topology to other routers in the same OSPF area so that they can update their LSDB. Thanks to a synchronization process known as **flooding algorithm**, all routers from the same area share the same LSDB. Common LSAs describe information such as routers in terms of their links, networks in terms of attached routers, routes external to a link-state domain (external routes and summary routes), etc. Figure 2.2 presents a picture of the LSA header.

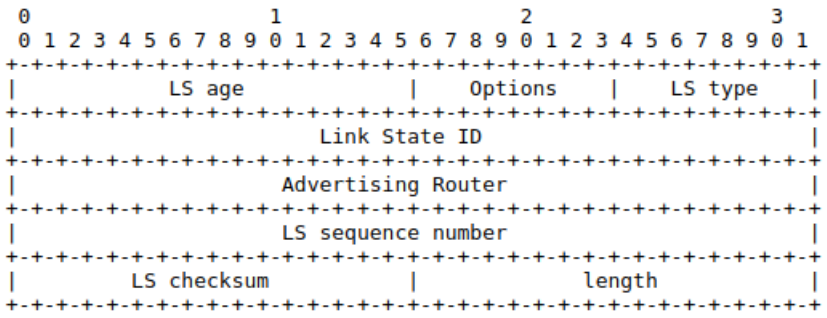


Figure 2.2: OSPF LSA header [32].

There exists 11 different types of LSAs [13] among which the most important are:

- Type 1: Router LSA - Each OSPF Router must originate a router LSA to describe the state of its own links. It sends one router LSA for each area it is part of. The link-state ID field is set to the originating router ID.
- Type 2: Network LSA - On multi-access links, routers elect a DR. The DR is responsible for originating a Network LSA, which helps to reduce the information needed to describe multi-access networks with multiple routers attached. It also acts

as a hub for the flooding of LSAs on that link, thus reducing flooding overheads. The link-state field ID is set to the IP interface address of the DR.

- Other types of LSAs are used to flood information across different areas, find the Autonomous System Boundary Router (ASBR), etc., but we do not need them to understand the basic principles.

Other information carried in the LSA header are: the age of the LSA (the time since it has been originated), some options, the link state ID which identifies the portion of the internet environment that is being described by the LSA, the advertising router (router ID of the LSA's originating router), a sequence number, a checksum and the length of the LSA.

Flooding algorithm

Routers in a same area want to have the same LSDB to be sure they have the same representation of the topology. To do so, they exchange LSAs by following a particular algorithm called flooding algorithm.
First when an adjacency between two routers is formed, they start sending each other a Database Description packet. This packet contains the headers of the LSAs contained in their LSDB. From that information, each router checks whether it has or not the LSAs and sends an appropriate Link State Request (LSR) to the other router. When a router receives a LSR, it sends all the LSAs asked in the LSR in a Link State Update (LSU). On reception of an LSU, each router follows a certain scheme described on Figure 2.3.

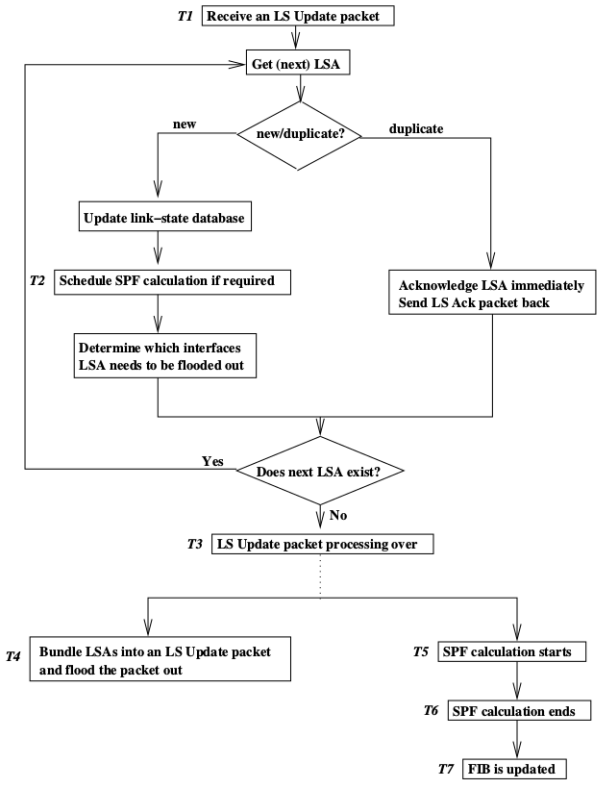


Figure 2.3: LSU processing [11].

When a router receives a LSU (*T1*), it iterates over all the LSAs contained in the LSU. For each LSA, it first checks if the LSA is a new or a duplicate one. In case it is a duplicate it just acknowledges the LSA, nothing needs to be changed in the LSDB. In case it is a new one, it has to update its LSDB. Then, it may schedule a new Dijkstra computation (*T2*) in case the reception of this new LSA gives some new relevant information for the shortest path computation. Once all LSAs have been processed (*T3*), the router floods the new LSAs in a LSU packet (*T4*). If an SPF calculation was scheduled, it will be performed some time after and will possibly update the forwarding table (*T5, T6 and T7*).
All together, this forms the flooding algorithm, which is one of the most important building block of OSPF as it allows flooding the information all across the network.

2.5 OSPF implementations limitations

We presented the basic operations of OSPF, which is used as the IGP in many ISP backbones and enterprise networks. OSPF has been designed to be flexible and allow network operators to tune the protocol using a CLI [8]. Thanks to that, it is for example possible to set the IGP costs of the links as desired, choose the interval between two Hello packets, set the minimal timer value between two SPF calculations, etc. Although this already allows network operators to adapt the protocol to their needs, there are some intrinsic limitations due to the design of the implementations. Indeed, with this approach the set of things that can be tuned is limited to the ones that have been planned and thus made available through the CLI. It is for example impossible for a network operator to create and flood a LSA of a new type that he defined, when some event occurs. It is also impossible to completely change the operation of the Dijkstra computation so that it takes other metrics than the IGP into account when computing the shortest path tree.

The solution we propose in this master thesis tries to take off these limitations by opening the routing protocol implementations. We propose to allow the modification of the behavior of some of the main operations of the protocol such that any network operator can tune them as he needs. The impossible use cases we just presented would then become possible by tuning the appropriate parts of the protocol implementation.

2.6 An OSPF implementation: FRRouting

During this thesis we worked on an OSPF implementation to add some flexibility into it. Of course, we did not want to implement the OSPF protocol from the beginning by our self, so we had to choose an existing implementation to start from. We decided to use the one from FRRouting [35].
FRRouting (FRR) is an IP routing protocol suite for Linux and Unix platforms which includes protocol daemons for BGP, IS-IS, LDP, OSPF, PIM, and RIP. It was forked from Quagga [10], another routing protocol suite for Linux and thus includes the same fundamentals. FRR is an open source software that has been developed by a group of contributing companies in the open networking space. The choice of FRRouting rather than bird [24] or others IP routing suite is motivated by the fact that the documentation is good and there is a quite big community using it giving lots of websites and other resources with examples of configurations, etc.

Breaking up the OSPF implementation

Before going further, we will break up the OSPF implementation to bring out all the key features of the protocol. Indeed, in the perspective of adding flexibility to the protocol, we would like to identify the main components of the overall OSPF operation to make them customizable. From now, we will call such components **protocol operations**. In the article "OSPF Protocol Analysis" [29], the authors provide a good analysis of OSPF that we will use to break up the implementation.

The following list enumerates the main protocol operations. There are some secondary features that we do not describe here for simplicity. In the implementation, most of the protocol operations can be associated with a function. For each protocol operation we list here, we will give the associated function in the FRRouting OSPF implementation.

1. LSA Flooding (function `lsa_flood`): this mechanism ensures the synchronization of the LSDB of the different routers. This is primordial for OSPF to work well because

if routers have different LSDB, they might compute incompatible routes leading to unexpected behavior.

2. Shortest Path First algorithm (function `ospf_spf_calculate`): this algorithm is used to compute the shortest path tree with itself as the root based on the topology the router is aware of. This allows the router to know its best route toward each destination.

3. Adjacency bring-up (function `ospf_hello_send`): certain pairs of routers become adjacent. Then they synchronize their LSDB and keep it synchronized thanks to the flooding algorithm. Each router has a Neighbor State Machine (NSM) which is a FSM that keeps track of the adjacency state of each interface. State is changing upon some events such as reception of a hello packet, etc.

4. Designated Router (function `ospf_dr_election`): the DR is elected. Then it is in charge of originating the network LSA describing the network's local environment. All routers on the network are synchronizing their LSDB with the DR (using flooding). A Backup Designated Router (BDR) is also elected in case of failure of the DR.

Those are the main components of the OSPF protocol. Of course, there are some other components that comes on top of that, and the described features can themselves be split into other simpler steps. But for the purpose of this thesis, we will not go into more details.

Chapter 3

Extending OSPF to make it flexible

This Chapter will explain how we modified the FRRouting implementation of OSPF to add some flexibility into it. From now, for conciseness, we will denote as a **user** someone who wants to tune an OSPF implementation (most probably users will be network operators). We want a user to be able to modify the protocol implementation in a very simple manner. To do so, we propose to open the OSPF implementation at different locations called **insertion points**. Figure 3.1 shows how we propose to adapt the OSPF daemon process to support the insertion points.

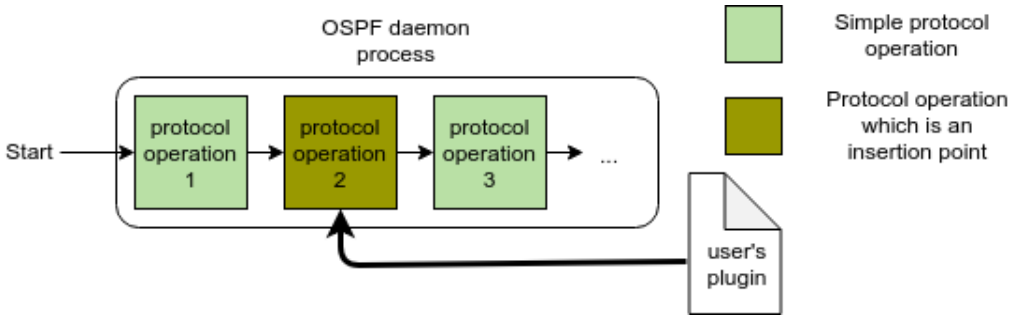


Figure 3.1: OSPF daemon process protocol operation/insertion point.

At each insertion point, users can dynamically load pieces of code that we call **plugins**. An insertion point is simply a location in the implementation's code where it is possible to load a plugin. Usually, an insertion point will correspond to a protocol operation (or a sub-part of a protocol operation) to allow its modification. On Figure 3.1, we see that the OSPF process is a succession of protocol operations (starts with *protocol operation 1*). The light squares are simple protocol operations, while dark squares are protocol operations that we turned into insertion point. *Protocol operation 2* is thus an insertion point and a user can decide to inject some code (i.e., a plugin) that will be loaded at that point in the OSPF process. Once a user's plugin is loaded at a particular location, it will be executed inside the implementation and can thus potentially modify its behavior. Choosing the insertion points we expose in the implementation is extremely important. Indeed, they correspond to the parts of the implementation that can be tuned and therefore determine how customizable the implementation will be.

Letting the user inject some code into the protocol implementation comes with several constraints. Our implementation will need to be able to execute the user's written code in a safe environment (we do not want to execute unknown code without any check). Because these codes will be independent of the OSPF implementation, we want this execution

environment to be platform independent (does not depend on the underlying hardware/OS) and to provide some isolation. The combination of the insertion points and the execution environment will allow users to easily test and deploy new protocol features without having to release new versions.
To be easily customizable, we also want our implementation to integrate a simple user interface that allows interacting with the implementation, together with a management system that deals with user's injected codes. This code management system of course needs to be internal to the implementation and invisible to the user.

This Chapter will describe how we extended OSPF to allow users to dynamically inject their own codes in the protocol. In Section 3.1, we will describe uBPF, the running environment we chose for executing user's plugins. In Section 3.2, we will explain how we built and chose the insertion points. In Section 3.3, we will describe the **plugin**, which is the entity created by a user and that abstracts the ability to modify the protocol behaviour. In Section 3.4, we will explain how we manage these plugins in our implementation and show how users can easily write plugins and integrate them in the implementation. In Section 3.5, we will discuss how we manage the memory we give access to plugins and introduce the execution context of a user's code: the **pluglet context**. Finally, in Section 3.6, we will describe the **OSPF plugins API**, which is the API we provide to allow interacting with the implementation. This API is composed of the insertion points and some function we expose to user's plugins.

3.1 User-space eBPF (uBPF) as a running environment

To allow users to inject code into the OSPF implementation, we need an environment capable of executing it in a relatively safe manner. We also want this environment to be platform independent such that it can be easily deployed everywhere. For that purpose, we choose the uBPF virtual machine [22], which is a library for executing eBPF programs. It includes an eBPF assembler, disassembler, interpreter, and JIT compiler for x86-64 architecture.

uBPF is a user-space implementation of the extended Berkeley Packet Filter (**eBPF**). eBPF [19] has been introduced in the Linux kernel in 2014. It is a small **virtual machine** which runs programs injected from user space and attached to specific hooks in the kernel. While eBPF was originally used for network packet filtering, it turned out that running user-space code inside a sanity-checking virtual machine is a powerful tool for kernel developers and production engineers [16].
eBPF was designed to interact with the Linux kernel however, FRRouting is implemented in user-space. This is why we are using a user-space implementation for the purpose of this work. The uBPF virtual machine is capable of interpreting or just in time compiling eBPF bytecodes. The uBPF implementation we use provides the following features:

- A bytecode can run on any hardware/Operating System (OS) on which OSPF would be running. Indeed, because the code is executed in a virtual machine that has its own instruction set, it is independent of the host machine which is running it. Because user's codes are autonomous, this is a very nice feature to be able to execute them independently of the host machine.

- Each bytecode has to pass through a static verifier before being executed. If the checks fail, the bytecode is rejected. The verifier especially checks that: the code contains an exit instruction, all instructions are valid, there is no division by 0, all

jumps are valid and the bytecode does not write to read-only registers. This already gives some control on what bytecode a user could write.

- A second verifier adds some specific instructions in the bytecode to check the memory accesses of the code at run-time. It checks that the bytecode memory accesses remain in its allowed area. **The allowed area is composed of its own stack and the memory the bytecode received as argument**. The verifier gives control on the memory a bytecode can access which is nice to avoid bad bytecodes to write to the OSPF memory for example.

- The uBPF VM allows to use some functions external to the eBPF programs. We call these functions the **helper functions**. In the classical in-kernel eBPF VM these functions are used to retrieve/push data from/to the kernel [7]. In our implementation, we will use these helper functions as an interface between eBPF programs and the OSPF implementation.

The uBPF VM we use offers us two possibilities: either we can use it as an **interpreter** or as a **JIT compiler**. These two solution have performance and security implications that are very different [12]:

- In interpreted mode, the eBPF bytecode is interpreted at run-time by the uBPF VM. One big advantage is that if the eBPF bytecode crashes for any reason (illegal memory access, etc), it will not affect the OSPF process but only the VM. Errors in the bytecodes stay in the VM sanity area and thus never affect the implementation. On the other hand, in interpreted mode the performances are worst than with JIT compilation.

- The uBPF JIT compiler takes the eBPF bytecode and compiles it into x86_64 (64 bits version of the x86 instruction set) assembler instructions which are the one used by most today's architectures. The compiled instruction are then executed as the ones from the OSPF process. JIT compilation gives significantly better performance but contrary to the interpreted mode, if the eBPF bytecode crashes, it will make the whole protocol to crash (this would be very bad of course).

Because JIT compiled instructions might make the protocol crash, using the JIT compiler would require going through an additional verification process before authorizing the use of a eBPF bytecode. This process would have to validate a bytecode before being able to use it in our implementation to be sure it is sane. In a real deployment perspective some effort should be put into this validation process of eBPF bytecodes to be able to take advantage of the JIT compilation performances while keeping security requirements. We measure the difference in terms of performances between interpreted and JIT compiled solutions in Section 4.3 and we discuss the validation of the bytecodes perspective in Section 5.1.

We integrated the uBPF virtual machine into FRRouting and more particularly in the OSPF implementation. This makes the OSPF implementation able to create some UBPF VMs and execute eBPF bytecodes into them. This also means that user's codes will need to be compiled to eBPF bytecodes to be usable in the implementation.

3.2 Plugins insertion points

As already introduced, we expose some part of our OSPF implementation by allowing users to inject code at particular locations called **insertion points**. Concretely, they are locations in the code where users can load their own plugins such that it is executed when the program passes on that portion of code. The choice of the insertion points we expose

is a very important part of the design because it will determine how the user can modify the protocol. We choose to associate one insertion point with each of the main protocol operations (we identified some of them in Section 2.6). At each insertion point we want to give as much flexibility as possible to the users so that they can tune each protocol operation as they want to. To do so, we propose to put three different **anchors** at each insertion points. A user can attach eBPF bytecodes at each of the different anchors of an insertion point. Figure 3.2 shows how an OSPF protocol operation can be turned into an insertion point.

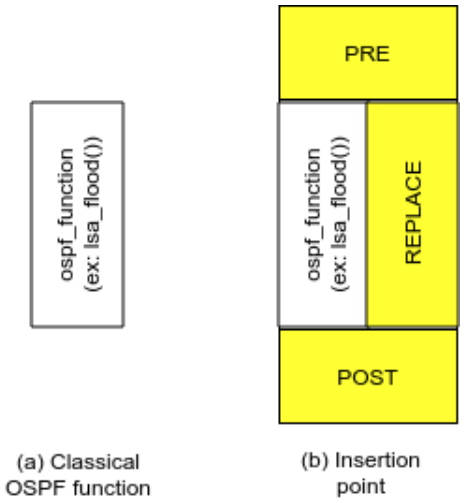


Figure 3.2(a) shows a normal OSPF protocol operation (the LSA flooding function for example). On Figure 3.2(b) we turned the OSPF function into an insertion point. The insertion point has 3 different anchors: a bytecode inserted at the `PRE` anchor will be executed before the `lsa_flood` function. A bytecode inserted in `POST` mode will be executed after. If there is no pluglet inserted at `REP` anchor, the `ospf_lsa` function will be executed normally (in white). If there is a bytecode inserted at the `REP` anchor, the bytecode will replace the function (in yellow).

Figure 3.2: Transforming OSPF protocol operation into an insertion point.

We provide the `PRE`, `REP` and `POST` anchors. A bytecode injected at the `PRE` anchor is inserted at the beginning of the protocol operation associated with the insertion point. At the `POST` anchor, it is inserted at the end and at the `REP` anchor it is inserted as a replacement of the protocol operation. We allow injecting multiple bytecodes at the `PRE` and `POST` anchors of an insertion point because it will not interfere with the execution of the protocol operation. Indeed, bytecodes injected at both `PRE` and `POST` anchors only have read access to the OSPF variables and thus they are not able to modify the protocol behaviour. On the contrary, a bytecode injected at the `REP` anchor overrides the default behaviour of the protocol operation and has full access to the OSPF variables (read and write). Therefore, we only allow to inject a single bytecode at the `REP` anchor.

Listing 3.2.1 shows the pseudo-code to transform an OSPF function into an insertion point in the implementation.

Listing 3.2.1: Pseudo-code to transform an OSPF protocol operation into an insertion point.

```
1  ospf_lsa_flood() {
2      if(plugin_tab[LSA_FLOOD_INSERTION_POINT] != NULL) {
3          if(plugin_tab[LSA_FLOOD_INSERTION_POINT]->PRE != NULL) {
4              exec_all_loaded_codes(PRE); // executes bytecodes at PRE ↩
                   anchor
5          }
6          if(plugin_tab[LSA_FLOOD_INSERTION_POINT]->REP != NULL) {
7              exec_loaded_code(REP); // executes bytecode at REP anchor
8          }
9          else { // no REP bytecode, execute built-in function
10             built_in_ospf_lsa_flood();
11         }
```

```
12          if(plugin_tab[LSA_FLOOD_INSERTION_POINT]->POST != NULL) {
13              exec_all_loaded_codes(POST); // executes bytecodes at POST ↩
                    anchor
14          }
15      }
16      else { // nothing injected at insertion point, execute built-in ↩
            function
17          built_in_ospf_lsa_flood();
18      }
19  }
```

This function is the `ospf_lsa_flood` function that implements the LSA flooding protocol operation. We bring some small modifications to the classical OSPF function: On line 2, we check if some plugin have been inserted for that insertion point. If not, we simply execute the built-in function (line 17). If yes, we check if bytecodes have been inserted at the different anchors (lines 3, 6 and 12). If it is the case, we execute them accordingly. If nothing has been injected at the `REP` anchor, we execute the built-in function (line 10).

At each insertion point, an argument might be passed to the bytecodes (the same argument is given to all the bytecodes injected at the same insertion point). This argument is insertion point specific and contains useful information that a user could use in its codes. The insertion points and associated arguments we provide in our implementation are briefly described in Section 3.6 and exhaustively documented in Appendix A. In the very vast majority of the cases, the arguments given to the plugins are the ones given to the OSPF function that we turned into an insertion point.

3.3 The plugin abstraction

This Section describes the abstraction we provide for manipulating bytecodes injected by the users. We will define different entities that we use to build our user's code management system.
Listing 3.3.1 defines the **pluglet** which simply represents an uBPF VM and its execution context.

Listing 3.3.1: Pluglet definition.

```
1  typedef struct pluglet {
2      struct uBPF_vm *vm;
3      pluglet_context_t *pluglet_context;
4  } pluglet_t;
```

When a user wants to inject an eBPF bytecode, the bytecode is loaded into an uBPF VM and a pluglet is created.
In our implementation, a pluglet cannot exist alone, because every pluglet is part of a **plugin** (defined in Listing 3.3.2).

Listing 3.3.2: Plugin definition.

```
1  typedef struct plugin {
2      pluglet_t *pluglets_PRE[MAX_NBR_PLUGLETS];
3      pluglet_t *pluglet_REP;
4      pluglet_t *pluglets_POST[MAX_NBR_PLUGLETS];
5      heap_t *heap;
6      void *arguments;
7      int type_arg;
8  } plugin_t;
```

A plugin is a piece of software that is **associated to an insertion point** and that can contain multiple pluglets (i.e., multiple eBPF bytecodes). A plugin contains:

- Between 0 and `MAX_NBR_PLUGLETS` pluglets pointers for both `PRE` and `POST` anchors. We limit the number of pluglets to avoid a user to inject thousands of bytecodes which would break the implementation (overflow the heap for example). The default value for `MAX_NBR_PLUGLETS` is set to 20. We think it is enough for most of the uses but it could easily be changed in case we realize it is not.
- 0 or 1 pluglet pointer for the `REP` anchor.
- A pointer to the heap common to all the pluglets that belong to the same plugin. This heap is managed in the user's written code. eBPF bytecodes from the same plugin are able to share variables using this heap (more on that in Section 3.5).
- A pointer to the arguments given to the bytecodes of the plugin.
- An identifier to know what type of argument has been given to the plugin (useful to cast).

All these informations are useful for the management of the plugins in the implementation. When a user injects a bytecode at a specific insertion point, there are two possibilities: either there is no existing plugin for that insertion point, in that case both a new plugin and a new pluglet (that belongs to this plugin) are created. If there already exists a plugin for that insertion point, a new pluglet is created and is added to the existing plugin at the specified anchor. In the following, we will denote as a **plugin** an entity able to modify the behavior of a protocol operation by being attached to an insertion point.

3.4 Plugins management

This Section will describe how plugins are managed in the implementation. As stated, plugins are sets of pluglets to be executed in their uBPF virtual machine. We would like a user to be able to inject plugins dynamically on demand. In a real network, routers cannot be stopped and restart at any time. The possibility of loading the plugins dynamically is thus a very nice feature, because it allows modifying/monitoring some protocol operations **without** having to recompile the implementation.

To keep track of all the plugins in the implementation, we define a general structure to contain all the pointers to existing plugins. This structure takes to form of an array of pointers where keys are the unique identifiers of each insertion points. This structure (presented in Listing 3.4.1) is in charge of remembering each plugin that has been inserted at a given insertion point and all its associated information. The `MAX_NBR_PLUGINS` parameter is the number of insertion points present in the implementation. Thus, each insertion point has a corresponding cell array, that can have a pointer to the associated plugin if there is one.

Listing 3.4.1: Structure containing all the loaded plugins.

```
1 typedef struct plugins_tab {
2     plugin_t *plugins[MAX_NBR_PLUGINS];
3 } plugins_tab_t;
```

Even though it is not always a good practice in C, we made this structure accessible from the whole implementation using extern variables. This allows easily accessing the structure from every insertion point in the implementation.

When the OSPF protocol starts, the tab structure is created and all the array's cells are set to `NULL` (because they are no plugins injected). When the user states that he wants to inject a particular bytecode, a message is sent to the **plugins manager** (launched as a detached thread on startup of OSPF). The plugin manager is in charge of loading the corresponding eBPF file and inserting the bytecode at the specified insertion point and anchor. When a user first loads an eBFP bytecode at a given insertion point, the associated plugin is created. The user can then inject more bytecodes in this plugin.
Thanks to this flexible approach for injecting plugins we are able to modify the behaviour of the protocol (by implementing dedicated plugins) without having to recompile the implementation.

3.4.1 Writing plugins

Writing plugins is very easy for the users. The Clang compiler supports the compilation of C programs into eBPF bytecode (the following line compiles the prog.c file into eBPF).

```
clang-6.0 -O2 -target bpf -c prog.c -o prog.o
```

This allows users to easily create platform independent plugins by writing C programs.

3.4.2 User CLI

To let users interact with the protocol implementation, we provide a simple **user CLI**. This CLI is able to read manifest files that describe what bytecodes a user wants to inject and where he wants to inject them. The user can launch the user CLI and provide the path to a simple '.txt' file. This file must contains one line for each eBPF bytecode file the user wants to inject. Each line must follow the following format:

```
[path to bytecode file] [insertion point ID] [position (PRE/REP/POST)]
```

The first empty line will be considered as end of file. Listing 3.4.2 presents an example of such a file:

Listing 3.4.2: Example of a manifest file.

```
/plugins/lsa_flood.o LSA_FLOOD PRE
/plugins/spf_time.o SPF_CALC PRE
/plugins/spf_time_post.o SPF_CALC POST
```

This manifest file asks the plugin manager to inject 3 different bytecodes. The first bytecode ('lsa_flood.o') is inserted at the `LSA_FLOOD` insertion point at `PRE` anchor and thus forms a plugin composed of one pluglet. The two other bytecodes are both inserted at the `SPF_CALC` insertion point respectively at `PRE` and `POST` anchors. Becaue these two bytecodes are inserted at the same insertion point, they form a single plugin composed of two pluglets.
The CLI reads the manifest file and sends messages to the plugins manager. The plugins manager then injects the provided bytecodes at the specified insertion points. The bytecodes are now loaded and will be executed when the protocol will pass on the given insertion points.

3.4.3 Interactions between OSPF, the uBPF VMs and the plugin manager

This Subsection aims to summarize the interactions between the core protocol process, the plugin manager thread (which is launched as a detached thread on startup of the OSPF process), the uBPF VMs that are created and the user CLI we provide to inject the bytecodes. Figure 3.3 shows how these different parts are interacting together.

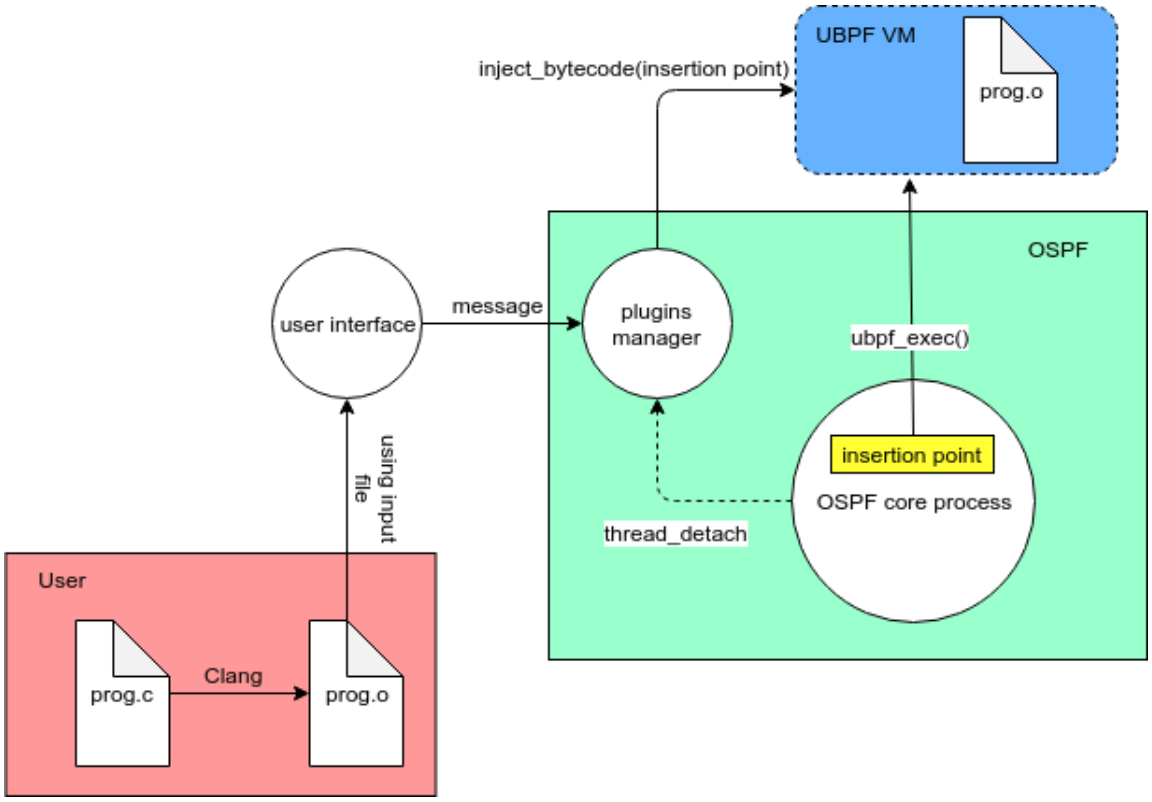


Figure 3.3: Architecture of the implementation.

In the green area (rectangle in the middle), the OSPF process launches the plugin manager as a detached thread. We also see that some insertion points (only one is drawn for visibility reasons) are present in the OSPF core process. In the red area (bottom left rectangle) we can see the user part. The user just writes a C program and compiles it to eBPF bytecode using Clang for example. Then it sends it to the user interface following the format of the manifest file previously described. When the user interface receives some input from a user, it sends messages to the plugin manager to inform it that it should inject some bytecodes. The plugin manager then injects the bytecodes at the specified insertion points. This creates one pluglet for each bytecode (it is not executed for the moment). Finally, when the core OSPF process goes on those insertion points, the bytecodes that belongs to the associated plugins are executed in their uBPF VM (using the `ubpf_exec` function).

3.5 Plugins memory management (and security concerns)

Until now, we described how a user can interact with the OSPF implementation by injecting some eBPF bytecodes at specific insertion points. Executing eBPF bytecodes inside the uBPF virtual machine gives us some control on them but unfortunately, there are some memory concerns we have to deal with. Furthermore, we need to take into account the possibility of having a malicious user that could try to inject malicious plugins into the implementation.

As introduced in Section 3.1, eBPF bytecodes executed in an uBPF VM are limited concerning the memory accesses they can perform. Indeed, they only have access to their own stack (which is 2048 bytes wide in our uBPF implementation) and the memory they receive as argument. This already raises two problems: first, 2048 bytes is very small and that can be limiting for the user when writing plugins. To address this issue, we

would like to allow an eBPF bytecode to have access to a heap. Second, this means an eBPF bytecode cannot access the OSPF process memory. Even though we do not want to give full access to this memory to the bytecodes, we would like to provide a controlled access that allow them to interact in a safe manner with the OSPF implementation internals.

We provide this services through helper functions (see Section 3.1) that we expose to the plugins. We will allow plugins to manage a heap and interact with variables from the OSPF process through them. All these functions need to be accessible from inside each uBPF VM so that plugins can call them. To register a helper function to a VM we use the `ubpf_register` function. When a C program is compiled as eBPF bytecode, the call to these external functions are translated to some `call` eBPF instructions with an opcode specific to the called function. The set of all the functions we expose to the VMs (using the register function) is one of the components of the **OSPF plugins API** (together with the insertion points).
When a user's bytecode is injected, the plugin manager creates the associated uBPF VM but it also registers all the functions of the API we want to expose. Here is an example of how to register an external function to the VM:

```
1 ubpf_register(vm, 0x01, "strcpy", strcpy)
```

This call registers the function `strcpy` to the VM '`vm`' with opcode 0x01 and allows an eBPF bytecode running in it to use it using the `strcpy` call. In our implementation it is possible to register up to 64 helper functions to the uBPF VMs. But this limitation is just an array size and could be extended in the future if we need more external functions. When a pluglet is created, we register all helper functions from the API to the asociated uBPF VM.

Registering functions will allow us to build our OSPF plugins API but there is something more we have to take care. Helper functions from our API are executed outside of the uBPF VM and have no way to get information about their caller (for example to know from what plugin they have been called, etc). We need to have some information about the execution context of the caller plugins to be able to build efficient and secure helper functions. This is why we introduce the **pluglet context**, which contains all the needed information about the current execution of a pluglet.

This Section will first introduce the pluglet context and how we deal with it. Then we will go through all the issues we described and show how we solve them by exposing new helper functions in our OSPF plugins API.

3.5.1 Execution context of the pluglets

When executing helper functions, we must know some information about the execution of the eBPF bytecode that called the helper. Indeed, let's take the example of an eBPF bytecode that calls an external function to get a variable internal to the OSPF implementation. Such functions are called **getter** functions in our implementation. They allow a bytecode to get OSPF structures by giving to the helper function a pointer to the structure it wants to access. If the helper function has no information about the execution context, how can it ensure that the pointer it gets as argument is a valid pointer, pointing to a real OSPF structure? Moreover, how can in ensure that this eBPF bytecode is allowed to access this structure?
Also, in the perspective of giving a heap to the bytecodes, we will have to write helper functions to manage this heap. How can we know where is the heap of the eBPF bytecode that called our malloc function? Because of all those concerns, we introduce the **pluglet context** (see Listing 3.5.1) that represents the context of execution of a bytecode. The

Listing 3.5.1: Definition of the pluglet context.

```
typedef struct pluglet_context {
    void *original_arg;
    int type_arg;
    heap_t *heap;
    struct plugin *parent_plugin;
} pluglet_context_t;
```

context contains the following fields:

- A pointer to the original argument given to the pluglet. This is useful to check that the eBPF bytecode gives some valid pointer to the helper functions.
- The type of argument given to the pluglet. This is useful to cast structures in the helper when the same helper can be used by pluglets from different insertion points.
- A pointer to the heap of the pluglet. This is useful for managing the heap of the plugins.
- A pointer to the plugin the pluglet belongs to. This is useful to know what plugin a pluglet is part of.

When a helper function is executed, it needs to know the context of the pluglet that called it. We want this execution context to be managed internally in the implementation and keep it invisible to the plugins. That means we cannot pass the context to the eBPF bytecodes (for example in their argument) so that they give it to the helpers. This is why we store the current context of execution as a global variable accessible from everywhere. This current context is set to `NULL` by default. Just before the execution of a pluglet, the current context is set to the context of the pluglet that is going to be executed so that it will be accessible by helper functions during its execution. After its execution, the current context is set back to `NULL`. We protect this variable by a mutex to ensure the consistency of the variable. The usage of a mutex ensures consistency but it also means that even in a multithreaded implementation, only one eBPF bytecode can execute at a time. This limitation could become a problem if many plugins need to run at the same time in different threads. The FRRouting implementation of the OSPF daemon is single threaded and thus does not suffer from this problem. But this could become an issue if we wanted to multi-thread the OSPF implementation for example. This issue will be discussed in more details in Section 5.4.

3.5.2 Plugins heap

As stated, eBPF bytecodes executed in the uBPF VM can only access their own stack and arguments memory. The stack size of the uBPF VM is limited to 2048 bytes, which is too small to write plugins (some structure of OSPF are already bigger than that). It would also be nice if multiple bytecodes from the same plugin could share variables. For these reasons, we would like to provide a heap for plugins. But we do not want to let eBPF bytecodes use the standard `malloc` function at will. Indeed, a malicious plugin could then be able to overflow the heap of the OSPF process by performing huge `malloc`. This is why we give a heap of limited size to each plugin. The heap we give to each plugin is set to **64 KB** in our implementation. This amount of memory looks reasonable to us but might be increased in the future if needed. Figure 3.4 shows how the plugin heap is allocated:
The plugin heap is allocated on the heap of the OSPF process and all pluglets that belongs to the same plugin share the same heap. This heap is given in the arguments of the eBPF

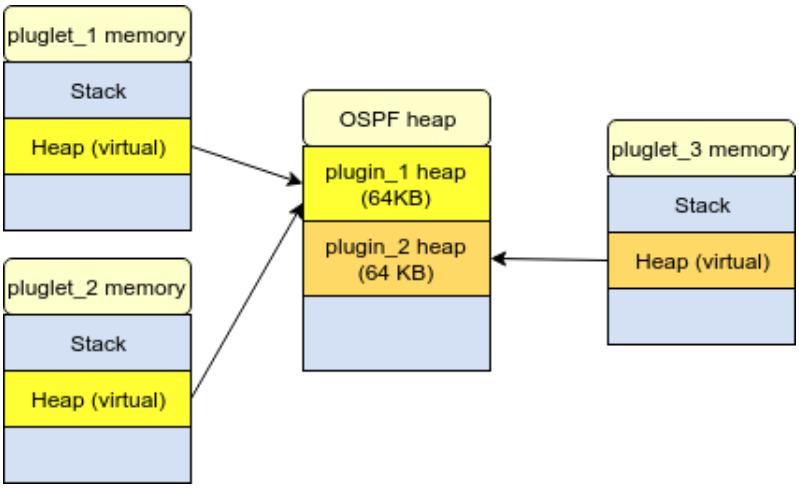


Figure 3.4: Pluglet heap management.

bytecodes. Indeed, as stated, eBPF bytecodes executed in the uBPF VM can only access their stack and the memory they receive as argument. Passing the heap in the argument ensures that the eBPF bytecodes will be able to access it. We also store a pointer to this heap in the pluglet context so that helper functions can know the address of the heap of the eBPF bytecode that is currently executed.
On Figure 3.4, we see that `pluglet_1` and `pluglet_2` belong to the same plugin and thus share the same heap which is allocated in the OSPF process heap. On the contrary, `pluglet_3` belongs to another plugin and thus has another heap.

`malloc/free` implementations

Because the plugin heap is allocated on the OSPF process heap on creation of the plugin, it is the implementation's responsibility to manage the memory blocks inside this heap and to provide `malloc/free` functions to allow plugins to use it.
Our memory management system is very simple. For each plugin heap, we store: a pointer to its start, a pointer to its end and a pointer to the last block of memory. At the beginning, the end of the heap of course corresponds to the start because the heap is empty. For each block of memory we allocate, we store some meta-data:

- The size of the memory block.
- A boolean to know if the block is available or not.
- A pointer to the next block of memory.
- A magic number (to check if a pointer to be freed is valid).
- An ID for that identifies this memory block

To allocate memory, we simply check if there is a big enough available memory block in the heap. If not, we extend the heap and allocate a new block. To free memory, we simply set the availability boolean to one to say that the memory block is now available. The management of this memory of course only affects plugins, not the OSPF implementation. To make pluglets able to manage their heap, we expose these functions in the OSPF plugins API:

```
1 void *plugin_malloc(unsigned int size)
2 void plugin_free(void *ptr)
3 void *plugin_realloc(void *ptr, unsigned int size)
```

From user's point of view, these functions are behaving like the standard ones. Users can use them in their plugin's code to have more memory than the uBPF VM stack.

Because Clang automatically compiles some function calls (such as `memcpy`) directly to their eBPF instructions instead of the call (it uses in-lining) we cannot not use some standard library functions in the plugins. This is why we also provide our own implementation of `memcpy` which operates the same way as the standard one but is not known by Clang and thus compiled as a function call:

```
1 void *plugin_memcpy(void *dest, const void *src, size_t count)
```

Sharing memory among pluglets

We also would like pluglets that belong to the same plugin to share variables. They already operate on the same heap (the plugin's heap), thus, we just need to provide a solution for them to be able to reference allocated memory areas. This is why there is an ID in the metadata of the memory blocks. We expose two new functions that allow pluglets to share variables:

```
1 void *plugin_malloc_with_id(unsigned int id, unsigned int size);
2 void *plugin_get_memory_with_id(unsigned int id);
```

The first one allows to perform a classical `malloc` but adds an identifier to the memory block. The second one allows to retrieve a memory block using its ID. Starting from the fact that the same programmer will write the different pluglets of a plugin, he will be able to share data across them. Because the heap is allocated only once, data stored in it is also persistent across multiple execution of the same plugin. This can be useful to keep track of some changes in time in the plugins. On the other hand, programmers must be careful because in case of memory leak, a plugin that is called often can quickly run out of memory. It is programmer's responsibility to manage it in a safe manner.

Optimizing the memory management

The heap management functions we implemented are operating in a very simple way. The `malloc` function is simply going through the whole heap and allocates memory when it finds a sufficiently big enough space. In the case the bytecodes from a plugin would do lots of `malloc/free` of different sizes, this could lead to a lot of fragmentation and the cost in term of performances can become big. This heap management could be improved later to avoid suffering from fragmentation after many `malloc/free` operations. Solutions such as Two-Level Segregate Fit [26] propose $\mathcal{O}(1)$ for both `malloc` and `free` functions.

3.5.3 Plugins accesses to OSPF memory

Plugins are executed in a Virtual Machine and thus have their own memory stack and cannot access the memory from the process that launched the VM (the OSPF process in our case). This is a basic principle of virtualization. Although this is a good feature to keep control on what a plugin can do, we would like to relax this constraint to allow a plugin to access some parts of the OSPF memory (for example to get some variables of interest). As stated, we use helper function as interface between plugins and the OSPF implementation. We need to be really careful when designing these function because contrary to the eBPF code which is executed in the VM, if an external function crashes (segmentation fault, etc.) it will make the whole OSPF process to crash. This is why we will need to verify the arguments the user gives to our helper functions in its plugins.

To allow plugins to get/set OSPF variables, we will feed our **OSPF plugins API** with getters and setters functions. These functions will allow eBPF bytecodes to interact with internal variables of OSPF. The following example explains how getters/setters functions operates: A user is writing a plugin for some insertion point in the code. At this insertion point, the argument passed to the plugin is the following structure (as stated, the heap is there so that it is accessible memory for the eBPF bytecode):

Listing 3.5.2: Argument example.

```
1 struct arg_example {
2     heap_t heap;
3     struct ospf *ospf;
4 };
```

The plugin can access the '`ospf`' pointer, but it cannot access the fields of the structure. Indeed, because '`ospf`' is a pointer, its fields are out of the memory the plugin is allowed to see. If the plugin tries to access these fields, the uBPF VM will see that the plugin is trying to access forbidden memory and will stop its execution. So, to allow a plugin to access the ospf structure, we need to provide an external function (a getter). The plugin will then call this getter with a pointer referencing to some memory accessible by the plugin, so that the helper can copy the ospf structure in this memory. The plugin will then be able to read the structure fields because it will access allowed memory (its stack or its heap). Still, if the OSPF structure itself contains pointers (i.e., the structure is nested), others getter functions will be needed to access the structures pointed by these pointers. The setter principle is the same, the only difference is that the helper will copy the value given by the plugin to the original OSPF variable. It just operates the other way around.
Listing 3.5.3 presents the getter function used to get an ospf structure:

Listing 3.5.3: Example of a getter function.

```
1  int get_ospf(struct ospf *ospf, struct ospf *ospf_copy) {
2      pluglet_context_t *pluglet_context = current_context;
3      if(pluglet_context == NULL) {
4          printf("NULL pointer \n");
5          return 0;
6      }
7      if(ospf == NULL) return 0;
8      switch (pluglet_context->type_arg) {
9          case ARG_PLUGIN_SPF_CALC:
10             if(ospf != ((struct arg_plugin_spf_calc *) pluglet_context->↩
                   original_arg)->ospf) return 0;
11             memcpy(ospf_copy, ospf, sizeof(struct ospf));
12             break;
13         default:
14             fprintf(stderr, "Argument type not recognized by helper ↩
                   function");
15             return 0;
16     }
17     return 1;
18 }
```

This function is used to get an OSPF structure from the implementation. The eBPF bytecode calls this helper function and gives as first argument a pointer to the OSPF structure it wants to access (it received the pointer in its arguments). The second argument the eBPF bytecode gives to the getter is a pointer to some memory it has access to (some memory on its stack or its heap). Then the helper function goes through some simple steps:

- On line 2, the helper gets the current context of execution (we are sure it is valid because it is not accessible to the user). Then it checks if the context is not `NULL`,

which would mean there is a problem.

- On line 7, it checks that the user did not give a `NULL` pointer to the function. Indeed, we do not want to produce segmentation fault, that would make the whole OSPF process to crash.
- On line 8, it checks what is the type of arguments given to the plugin. This ensures that we wanted to expose this structure to the plugin at that insertion point.
- On line 10, it checks that the pointer we received is the real OSPF pointer the bytecode received as argument, to ensure that the plugin did not send a random pointer.
- Finally on line 11, it copies the OSPF structure into the second argument which is memory accessible by the eBPF bytecode.

The eBPF bytecode is now able to read the structure pointed by '`oi_read`'.

The getter function we presented is safe. A malicious plugin could not give bad pointers to the helper without the helper detecting it. But this solution means that a plugin can only get OSPF structures it received as an argument (because they are the only pointers stored in the context). Unfortunately, this limits quite strongly what a plugin can do. Indeed, let's take the following example: A plugin calls a helper function that returns a pointer to an OSPF structure. Then it wants to access the variable behind this pointer through a getter. With the solution we described, the getter will fail because the pointer it receives was not in the arguments of the plugin. For that reason, some getters relax this constraint and allow getting structures that were not given as argument. This relaxation of course opens the implementation and a malicious plugin would now be able to make the protocol crash by giving a wrong pointer. One solution to avoid this problem would be for helper functions that return pointers to update the context by adding the returned pointer to the list of valid pointers. That way, we could check in the getters if the pointer is contained in the context. This solution could become very limiting in terms of performances if many pointers needs to be stored. Another solution (probably better) would be to perform a static validation of plugins before allowing to inject them. This validation process would for example check that a pointer given as argument of a getter function was either received in the arguments or as a return value of a helper function. This would ensure the security without having any impact on the performances because the check would be done outside of the implementation. We did not work on this issue in the frame of this work because of a lack of time, but this is for sure one of the main further work track.

3.6 OSPF plugins API

This Section presents an overview of the OSPF plugins API we expose to the plugins. This API is composed on the insertion points and the helper functions. Both together form the interface plugins can use to interact and possibly modify the implementation. This section gives a high level overview of the API. For more details about the specification of the insertion points and helper functions we expose, see Appendix A.
In Section 3.6.1, we describe some of the insertion point we propose and explain why they can be useful. Then, in Section 3.6.2, we describe some helper functions we decided to expose to the plugins.
The version of the API we propose here is a prototype that has mainly been driven by the plugins we wanted to implement, and should probably be improved in the future to make OSPF even more flexible.

3.6.1 Insertion points

Here are some insertion points we propose:

- `SPF_CALC`: this insertion point is associated to the `ospf_spf_calculate` function and thus corresponds to the SPF calculation protocol operation. This function calculates the shortest path tree for an area. Thanks to this insertion point it is for example possible to monitor the execution time of the SPF calculation by implementing two pluglets: one for the `PRE` anchor that stores the current time and one for the `POST` anchor that computes the difference. Using the `REP` anchor, it is also possible to change the behavior of the SPF calculation.

- `OSPF_SPF_NEXT`: this insertion point is associated to the `ospf_spf_next` function and is a sub-part of the SPF calculation protocol operation. The function implements *RFC2328 16.1 (2)* [31]. It is called once for each vertex of the topology and is in charge of building the shortest path tree. This insertion point could for example be used to modify the way to compute the shortest path tree while keeping the general design of the SPF calculation (i.e., without having to use the `REP` anchor of the `SPF_CALC` insertion point).

- `LSA_FLOOD`: this insertion point is associated with `ospf_flood` function and thus corresponds to the LSA flooding protocol operation. When a LSA is received (through a LSU for example), the `ospf_flood` function is called under condition that the received LSA is not yet in the LSDB or is more recent than the one stored in the LSDB. This insertion point could for example be used to monitor the LSAs that are flooded, or even to modify the flooded LSAs in a given way before sending them.

- See Appendix A for more insertion points.

3.6.2 Helper functions

The following paragraphs will present the different types of helper functions we expose together with their purpose.

Generic and memory management functions

We expose some useful standard function such as `strcpy` or `gettimeofday`.
Because Clang treats some function such as `ntohs`/`ntohl` as inline functions, they are not usable in plugins. For that purpose we expose our own implementation of such functions: `plugin_ntohs`/`plugin_ntohl`.
For memory management purposes, we expose the functions presented in Section 3.5.2.

Getter functions

Getter functions (getters) are used to get copies of OSPF internal variables by giving the pointer to the variable we want to get. We provide such functions for most of the important structure in the OSPF implementation. Examples of such functions are: `get_ospf_interface` that gets an OSPF interface structure, `get_ospf_lsa` that gets an LSA structure, **get_ospf** that gets a structure that represents an OSPF instance, etc.

Setter functions

Setter functions (setters) are used to set OSPF internal variable to a given value. They are very similar to getters except that they copy from the plugin's memory to the OSPF memory instead of the opposite. Example of such functions are `set_ospf_interface` or `set_ospf_area`.

OSPF functions

Some functions from the OSPF implementation can be really useful when developing plugins. We identified some and expose them in our API. We wrap these functions in our own functions to be able to perform some checks before executing them (checking if the pointers given by the plugin are not `NULL`, etc). Examples of such functions are: `plugin_ospf_lsa_install` that allows installing an LSA into its LSDB, `plugin_ospf_flood_through_area` that allows flooding an LSA through a given area or `plugin_ospf_nexthop_calculation` that calculates the nexthop from root through V (parent) to vertex W (destination), with given distance from root to W (useful for tuning the SPF calculation algorithm).

3.7 Conclusion

In this Chapter, we showed how we extended the OSPF implementation by adding a virtual machine capable of executing user's written plugin. This extension allows users to tune each of the protocol operations of OSPF to fit their needs.

We think our solution is particularly well suited for OSPF. Indeed, in most of the cases OSPF is used for routing into enterprise and ISP backbone networks where all routers are usually configured and maintained by the same network administrator or at least the same team. It is thus easy to ensure that all routers have compatible plugins extending the implementations. Thanks to this fact, the solution is a lot easier to implement than for other protocols such as BGP, where different routers would probably be managed by different persons. Although it is well suited for OSPF, we think this solution could be generalized to other routing protocols (IS-IS, BGP, etc.), and could therefore completely change the way to implement them. Each protocol would then be a simple sequence of protocol operations easily extendable by anyone.

Chapter 4

Use Cases and performances study

In this Chapter, we will demonstrate the power of the flexible OSPF implementation by implementing some use cases. We will use the flexible OSPF implementation we just presented and take advantage of it to tune the protocol by injecting plugins.
In Section 4.1 we will show how we can use very simple plugins to perform some monitoring on OSPF. We will provide some pieces of code to demonstrate how simple it can be to write monitoring plugins. In Section 4.2 we will explain how we can use plugins to modify the behavior of the protocol. As an example, we will present an implementation of a part of the Flexible Algorithm [36] using plugins. Finally, in Section 4.3, we will study the impact on the performances of our implementation compared to the classical OSPF. We will particularly analyze the memory usage and the computation time of the Dijkstra algorithm. We will compare the vanilla OSPF implementation, our flexible implementation without any plugins injected and our flexible implementation with some plugins injected. Finally, we will measure the difference of impact in terms of performances between interpreted and JITed plugins.

4.1 Monitoring OSPF using plugins

Monitoring routing protocols is and has always been essential. Recording data about the flow of the protocol can for example help network operator to understand the weaknesses/bottlenecks of their network and thus help them to configure it in a better way. These information are also useful for protocol developers to analyze measurements in real condition systems. There of course already exists many tools that allow monitoring routing protocols and more especially OSPF. Most of them are querying information to the routers (for example using SNMP [41]) and computing some data of interest out of it. One issue with that method is that the chances are limited to what have been implemented in terms of possible requests. Using eBPF plugins as a monitoring tool allows to observe any metric, without the need of background implementation requirements (except the fact that the flexible implementation must expose the required API). Moreover, by using plugins we have access to all the internal variables of the protocol, which would not be the case using tools that are just observing the in/out packets. It is for example possible to know the execution time of the Dijkstra algorithm, which would not be possible with classical packets observation techniques. Finally, for now, if a network operator wants to monitor a new metric on its routers, he has either to poll new information using SNMP requests or to modify the implementation of the protocol, recompile it and rerun it. SNMP requests leads to more traffic on the network while stopping a router for recompiling the code is not always possible. Using our flexible implementation, a network operator could easily inject plugins and monitor in a fine-grained manner new metrics, without the need to recompile the code.

As explained in Section 3.4.1, implementing plugins is quite easy. A user just needs to write a C program, compile it into eBFP bytecode and inject it at the desired insertion point. Monitoring plugins are simple plugins that need to access some internal variables from the OSPF implementation. To do so, they simply need to use the getter functions we presented to get copies of the data they want to observe. Of course, the user must respect the constraints defined in the OSPF plugins API. The information collected in the plugins can then be treated as desired by the user. He could for example choose to store the data in a database, to send an email to the administrator when some events occur, etc.

4.1.1 Monitoring server

As an example, we implement a simple monitoring server (running on the router) that basically prints some metrics sent by the monitoring plugins. To communicate with the monitoring server the plugins are pushing some messages in a queue stored in the kernel. The monitoring server is listening on that queue to read the messages as they arrive. This is the inter-process communication (**IPC**) technique we choose even though there are many others possible solutions (unix sockets for example). A description and example of use of this IPC technique is available in that article "IPC using Message Queues" [37].

The monitoring server is launched as a separate program on the router we want to monitor. It reads on a kernel message queue with a blocking call so that it catches messages as they arrive. On reception of a message, the monitoring server checks the type of information contained in the message (using a 64bits integer). Then it casts the message body in the appropriate type and can treat it as needed. Our monitoring server is just printing the information contained in the messages it receives, but we could also imagine to store information in a database, or send an e-mail to the network manager in case a particular event occurs. We limited ourselves to a simple printing server for the purpose of this thesis.

4.1.2 Use cases

This Section presents some simple monitoring use cases we tried to handle using plugins. We will also study the advantages of using eBPF plugins compare to classical solutions.

Measuring the execution time of Dijkstra

Classical monitoring tools are usually observing the packets going through the network at some points and inferring some conclusion from the observed packets. This of course limits the information we can get to the information that is carried in the packets. In some cases it is useful to observe other metrics that are internal to the routers and impossible to infer only from the observed packets. For example an operator might be interested in measuring the execution time of the Dijkstra algorithm (or any other protocol operation). Plugins allow us to do that in a very simple manner.
We create a monitoring plugin that we inject at the `SPF_CALC` insertion point (corresponding to the `ospf_spf_calculate` function). This plugin is composed of two eBPF bytecodes (i.e., 2 pluglets). One is put at the `PRE` anchor and the other at the `POST` anchor. The code of both pluglets are presented respectively on Listing 4.1.1 and Listing 4.1.2. These codes show how easy it is to get this kind of metrics in our implementation.

Listing 4.1.1: `PRE` eBPF bytecode.

```
1 uint64_t spf_time_pre(void *data) {
2     struct timeval *t1 = plugin_malloc_with_id(3, sizeof(struct timeval))↩
          ;
3     gettimeofday(t1, NULL);
4     return 1;
```

```
}
```

The eBPF bytecode inserted at `PRE` anchor is very simple: it simply performs a `malloc` on the heap of the plugin for the pointer '`t1`'. By using the `malloc_with_id` function, an ID is assigned to the memory area that is allocated. This will allow retrieving data stored in this memory area in other pluglets of the same plugin. Then, it just stores the current time in the allocated memory.

Listing 4.1.2: `POST` eBPF bytecode.

```
uint64_t spf_time_post(void *data) {
    struct timeval t2;
    gettimeofday (&t2, NULL);
    struct timeval *t1 = (struct timeval *) plugin_get_memory_with_id(3);
    struct spf_mon mon;
    mon.time_spf = ((t2.tv_sec - t1->tv_sec) * 1000000 + t2.tv_usec) - t1->tv_usec;
    return send_data(SPF_CALC, (void *) &mon);
}
```

The eBPF bytecode inserted at the `POST` anchor is very similar. It stores the current time in '`t2`'. Then it reads in the heap using the same ID as the pluglet at the `PRE` anchor to retrieve the time stored by the first bytecode. It performs a subtraction to get the execution time of the function and stores it in a structure defined by the user. Finally, it sends it to the monitoring server so that the data can be treated.

We showed that using eBPF plugins and taking advantage of the heap accessible shared between pluglets belonging to the same plugin allow us to efficiently measure execution times of any part of the code (where an insertion point is defined of course).

Observing some internal variables

As introduced, using eBPF plugins to monitor OSPF also allow us to observe internal variables of the protocol. As an example, we implement a simple plugin that allow us to observe metrics such as the number of Hello messages sent on each interface, or the speed (bandwidth) of each interface. The C code of this plugin is show in Listing 4.1.3

Listing 4.1.3: eBPF bytecode to observe some internal variables.

```
uint64_t hello_count(void *data) {
    struct arg_plugin_hello_send *plugin_arg = (struct arg_plugin_hello_send *) data;
    struct ospf_interface *oi_read = plugin_malloc(sizeof(struct ospf_interface));
    if(oi_read == NULL) return 0;
    int ret = get_ospf_interface(plugin_arg->oi, oi_read);
    if(ret != 1) return 0;
    struct interface *ifp = plugin_malloc(sizeof(struct interface));
    if(ifp == NULL) return 0;
    int ret2 = get_interface(oi_read->ifp, ifp);
    if(ret2 != 1) return 0;
    struct hello_struct s;
    s.hello_count = oi_read->hello_out;
    s.itf_speed = ifp->speed;
    plugin_free(oi_read);
    plugin_free(ifp);
    return send_data(SEND_HELLO, &s);
}
```

This plugin is very simple as well. Here is a description of the main steps it does to retrieve the data of interest:

- Line 2: Casts the argument it receives into the appropriate type. This type depends on at which insertion point the plugin has been inserted.
- Lines 3, 7: Allocates some memory on the heap to give to the getter functions.
- Lines 5, 9: Uses getter functions. For each getter, the first argument given is a pointer to an OSPF structure the plugin wants to access (received as an argument by the bytecode) and the second argument given is a pointer to some memory accessible by the plugin (allocated on the plugin's heap in that case) where the getter will copy the data of interest.
- Lines 11-13: Stores the data of interest in a user's defined structure.
- Line 16: Sends data of interest to the monitoring server.

This simple plugin shows how easy it is to monitor OSPF internal metrics in the flexible OSPF implementation.

4.2 Modifying the behavior of OSPF using plugins

In this Section, we will show how we can customize the OSPF implementation using plugins. We will demonstrate that the flexible OSPF implementation allows achieving some use cases that would be really hard/impossible to do with the classical OSPF implementation in a quite simple manner.
OSPF is a routing protocol that allows routers to be aware of the topology they are part of. The flexibility of OSPF implementations is very limited because of the way they are implemented. For example, the traditional OSPF computes the best path toward each destination based on some IGP metric assigned to each link (usually the bandwidth). Unfortunately this does not allow computing constraints based paths that take into account other metrics/constraints than the IGP metric. This means that the traffic requirements of a network have to be represented by this single metric. Of course the IGP metric does not always represent the traffic requirements. This limitation has pushed network developers to create new Traffic Engineering (TE) protocols such as MPLS [39], RSVP-TE [2], etc. These protocols are running on top of OSPF and computing new paths that are installed in the forwarding table in addition to the OSPF paths.
This shows the limitations of the traditional OSPF implementations in terms of flexibility. Extending the flexibility of OSPF using one way or another would allow doing without the resource consuming TE protocols.

4.2.1 Use case: Flexible Algorithm

In the draft "IGP Flexible Algorithm" [36], the authors propose a solution for IGP protocols themselves to compute constraint based paths over the network (i.e., tailor the IGP computation to fit traffic engineering needs). The solution uses a set of extensions to OSPF that allow a router to send particular TLVs (Type-Length-Value) that describe other ways to compute the best path than the usual Shortest Path First algorithm. These TLVs are flooded using the Opaque LSAs (opaque LSAs have been introduced to allow some future extensibility to OSPF [3]). The authors also specify a way for routers to use Segment Routing (SR) to represent path that are computed using the presented algorithm.
In this Section, we will show the power of our flexible implementation of OSPF. The typical use case that the IGP Flexible Algorithm was trying to solve is to steer packets along paths that are not computed using only the IGP metric. One example they propose in the talk "Unleashing SR Traffic Engineering capabilities with SR Flexible Algorithms" [25] is the following: Each router marks each of its links with a color (let's say green or red).

The color could represent any metric related to the links. We could for example set the green color to the links that are encrypted and the red one to the links that are not. Or we could set the green color to the links with a speed greater than 100 MB/s and the red one for the slower links. This color just shows that we have another metric than the IGP metric to take into account. Once all links are marked with a color, we say that all packets must go to their destination using only the green links.

We will show that this kind of use case can be achieved in a quite simple way using our flexible implementation. Figure 4.1 shows a very simple topology we will use to demonstrate this use case. Each of the links have an IGP metric set to 100. The network is composed of 3 routers:

- R1: Router-ID = 1.1.1.1
- R2: Router-ID = 2.2.2.2
- R3: Router-ID = 3.3.3.3

Each router assigns a color to each of its link. Because a link has two attached routers and that each router advertises a color for this link we have one color for each direction on the link (see the arrows on Figure 4.1). In our example topology, we choose to put only green links, except for the link going from R1 to R3 that is red.

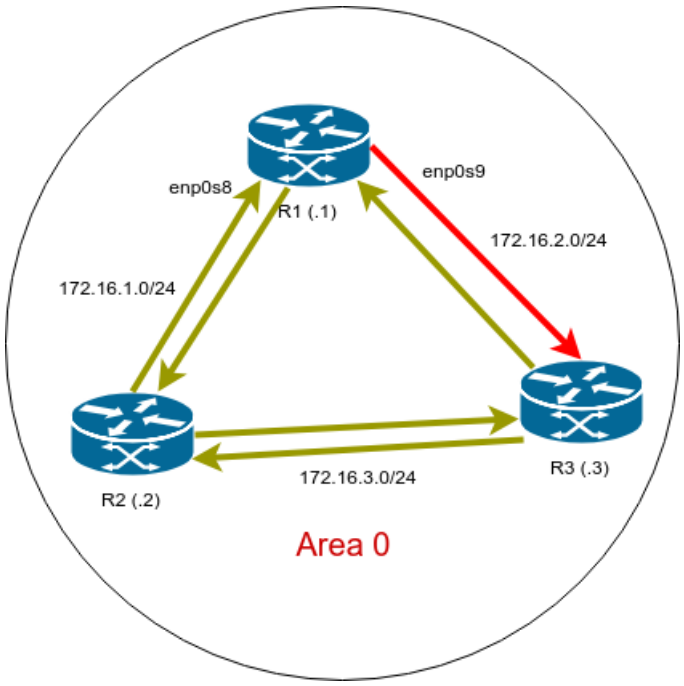


Figure 4.1: Example topology.

We would like packets to use only the green links. That means for example that if R1 sends a packet to R3, the packet should go through R2 contrary to what the traditional OSPF shortest path algorithm would do. In the following, we will explain how we achieved this use case using plugins.

Plugins

To achieve the use case we described we need to do two things:

1. Flood the information about the link color in the network. (*Plugin 1*)

2. Compute the paths taking into account the additional metric (the color of the link in our example). (*Plugin 2*)

Plugin 1 We choose to implement a new type of LSA (type 13) that does not exist in the standard OSPF to flood the information about the color of the links. To do so, we implement a plugin that creates a new LSA. The plugin then uses helper functions to install the newly created LSA into its LSDB and flood it through the area. Our new LSA is very similar to a router-LSA (type 1). The only difference is that for each link the router LSA describes, we add a color metric (as an unsigned int). For the purpose of this example the color are set by hand to correspond to Figure 4.1, but in a real implementation perspective we could assign colors depending on the speed of the interface, the delay of the link or anything else. We decide to inject this plugin at the `SPF_CALC` insertion point. That means that a new LSA will be created and flooded every time Dijkstra is computed.

Plugin 2 Now that we have our new type of LSA, we need to take the new metric into account when computing the paths. To do so, we will inject a pluglet at the `REP` anchor of the `OSPF_SPF_NEXT` insertion point. The function at this insertion point (`ospf_spf_next` function) implements *RFC2328 Section 16.1 (2)* [31]. The LSDB is represented as a directed graph with vertices being routers and networks. Each vertex in the graph has an ID, an associated LSA (router-LSA for router vertices and network-LSA for network vertices), a list of next hops and a distance from the root. The function at the `OSPF_SPF_NEXT` insertion point examines the links in the LSAs of the first vertex from the candidates list (root at first iteration). Then it updates the list of candidates with any vertices not already on the list. If a lower-cost path is found to a vertex already on the candidate list, it stores the new cost. This function is thus one of the most important from the SPF calculation process because it builds the shortest path tree.
We rewrite this function as an eBPF plugin, and we add our color constraints in its behavior. For each router-LSA that is examined, we check if there is a corresponding (same router-ID) LSA of type 13 in the LSDB. If yes, we check for each link the color of the link. If it is green, we continue normally, if it is red, we ignore this link. This causes the algorithm to prune everything that is after a red link.

Finally, we want to test our two plugins on the example topology. To do so, we run OSPF on the routers and observe what happens on R1.
In a first phase, we run our flexible OSPF implementation on each router, and we only insert the *Plugin 1* in each router. This makes each router to flood a LSA of type 13 with one color for each of its link. The routers advertise the colors shown on the picture i.e., only green links except R1 which advertises a red link toward R3. At that point, all router run the classical Dijkstra algorithm. Table 4.1 presents R1's routing table (R = Router, N = Network):

Destination	Nexthop	Interface	Cost
[*N*] 172.16.1.0/24	itself	enp0s8:172.16.1.1	100
[*N*] 172.16.2.0/24	itself	enp0s9:172.16.2.1	100
[*N*] 172.16.3.0/24	172.16.1.2	enp0s8:172.16.1.1	200
	172.16.2.3	enp0s9:172.16.2.1	200
[*R*] 2.2.2.2	172.16.1.2	enp0s8:172.16.1.1	100
[*R*] 3.3.3.3	172.16.2.3	enp0s9:172.16.2.1	100

Table 4.1: Routing table of R1 with classical Dijkstra.

The routing table is of course what we would expect with the classical shortest path routing. Indeed, the type 13 LSAs are ignored by the classical Dijkstra algorithm.

Now, in a second phase, we inject our second plugin at the `OSPF_SPF_NEXT` insertion point as a replacement of the built-in `ospf_spf_next` function. Once R1 has computed the SPF algorithm its routing table changes. Table 4.2 presents the updated routing table:

Destination	Nexthop	Interface	Cost
[*N*] 172.16.1.0/24	itself	enp0s8:172.16.1.1	100
[*N*] 172.16.2.0/24	172.16.1.2	enp0s8:172.16.1.1	300
[*N*] 172.16.3.0/24	172.16.1.2	enp0s8:172.16.1.1	200
[*R*] 2.2.2.2	172.16.1.2	enp0s8:172.16.1.1	100
[*R*] 3.3.3.3	172.16.1.2	enp0s8:172.16.1.1	200

Table 4.2: Routing table of R1 with our pluginized Dijkstra.

We observe that R1's routing table has been updated. It now avoids going through the red link and thus all traffic from R1 will go through R2 even if the destination is R3.

We showed that with two plugins we can already implement a new type of LSA and use the data carried by this LSA to modify the Dijkstra algorithm as needed. The plugin to originate a new LSA of type 13 is about 100 lines of code long and performs 318 eBPF instructions. The one to modify the SPF algorithm is about 160 lines of code long and performs 365 eBPF instructions. We think it is quite concise compared to the impact it has on the behavior of the protocol. For more details on the implemented pluginss statistics, see Table B.1.

4.3 Impact on the performances

In the previous two Sections, we showed the power of the flexible implementation of OSPF. This section will analyze the performances of the solution compared to the classical OSPF implementation. We will also try to measure what is the overhead due to the execution of eBPF bytecodes during the protocol execution.

4.3.1 Test environment

To make our measurements, we want our test environment to match several criterions:

1. We want to simulate a real topology, especially to have a Link-State Database filled similarly to the ones of routers in real networks.

2. We want the measurements to be done on a dedicated machine which is not running anything else. From now, we will denote the router on which we are doing the measurements the **test router**.

To match the first requirement we need to fill the LSDB of our test router with many LSAs. As we are not aware of the existence of an open source tool to produce and inject concordant LSAs in an LSDB we had to find another way. We turned ourselves toward **Mininet** [44], which is a tool for creating realistic virtual network on a single machine. We also use **IPMininet** [46], a python library extending Mininet, in order to support the emulation of IP networks. In our tests, we used Mininet v2.2.2 and IPMininet v0.4.2. Using IPMininet, the creation of a network containing multiple routers running IP protocols such as OSPF can be abstracted as a simple python script. To be able to simulate some

real topologies, we use the code proposed in the "Assessing Mininet" github repository (GPL-2.0) [18] that transforms '.graphml' files to executable Mininet topologies. We extend this script to be able to create IPMininet topologies. The topology zoo data-set [1] provides many real topologies in the '.graphml' format. Thus, this script allow us to easily simulate real topologies.

To match the second requirement, we use two different machines to make our tests. One machine is dedicated for the measurements (the test router). The other one is used to emulate a topology and is connected to the test router to fill its LSDB. We use 2 VMs running in our campus cloud infrastructure. Each VM has two cores QEMU Virtual CPU 2600 MHz, 2 GB of RAM and runs Ubuntu 18.10 (with kernel v4.18.0-20). Figure 4.2 shows a picture of the setup that has been used in all the tests we present hereafter.

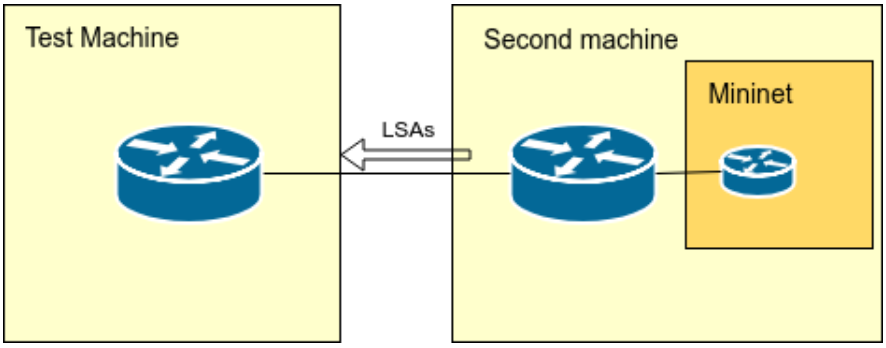
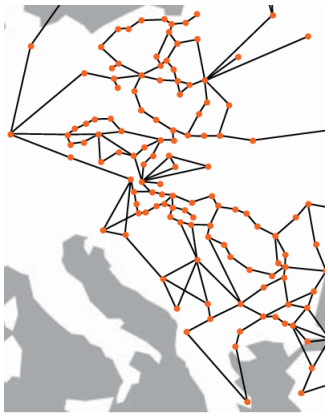


Figure 4.2: Tests setup.

The test machine (on the left) is running a router with the OSPF daemon. The router is connected to the router running on the second machine. The second machine router is connected via a virtual link to one of the routers of the mininet (we have only drawn this one in the mininet, but of course the mininet is running more than 150 routers). This allows to have the complete LSDB on the test router without having performance disturbances because of the mininet.

For all the following tests, we use the GTS Central Europe topology as our IPMininet topology (see Figure 4.3).



The GTS Central Europe topology contains about 150 nodes and the same amount of edges. We reproduce this topology in IP-Mininet by putting one router at each node. This gives a LSDB composed of about 300 LSAs (router LSAs and network LSAs). We choose this topology because it is one of the biggest in the topology zoo data-set [23], and thus should be one of the most constraining for the implementation.

Figure 4.3: GTS Central Europe topology.

4.3.2 Memory analysis

First, we would like to analyze the impact of our implementation on the memory usage of a router. To do so, we will compare our flexible implementation of OSPF with the vanilla OSPF implementation (i.e the one provided in FRRouting). More particularly, we will compare 3 different versions:

1. The vanilla version of OSPF (from FRRouting v6.0.2).
2. The flexible OSPF without any plugin injected, to see if the changes we made to the implementation brings memory overhead.
3. The flexible OSPF with 7 pluglets that form 5 different plugins injected, to see how much memory overhead plugins cause.

For each of these cases, we run OSPF for about 40 seconds. We attach psutil v5.6.2 [38] (library for retrieving information on running processes and system utilization) to the OSPF process to be able to monitor the memory utilization. We measure the Unique Size Set (USS) memory, which is the memory that is unique to a process and that would be freed if the process was terminated right now. This is the best way to know how much a process really uses because contrary to the most common Resident Set Size (RSS) memory, it does not take the memory shared between processes into account. Figure 4.4 shows the results of this experiment.

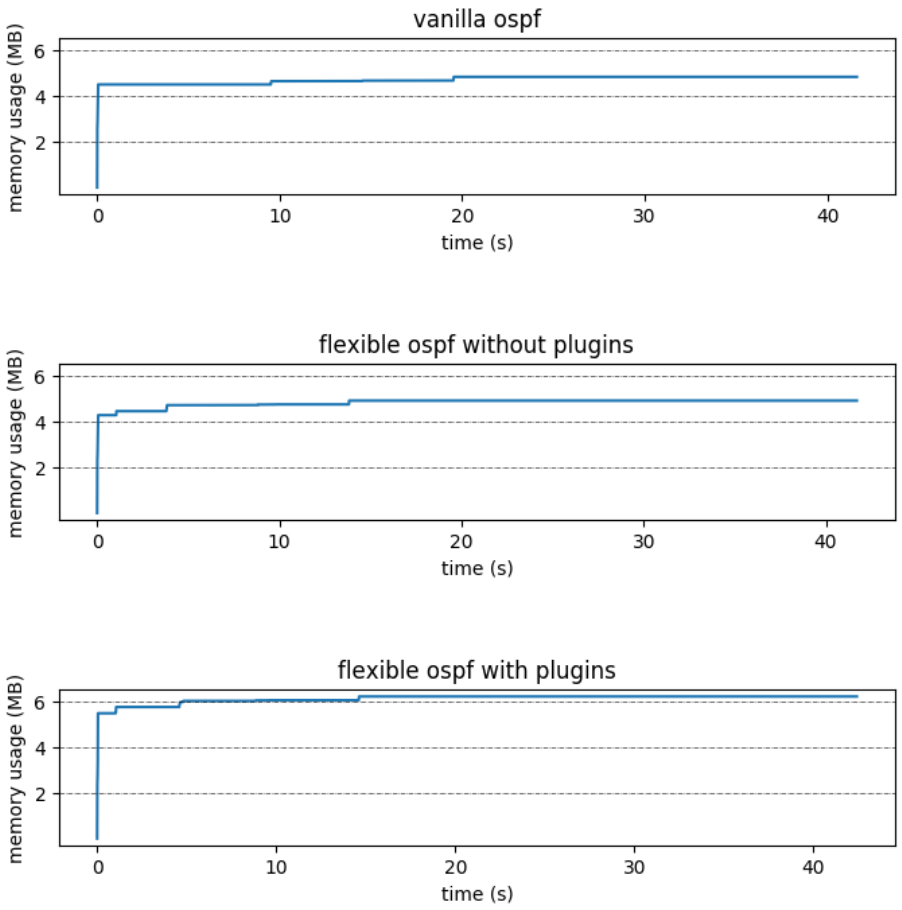


Figure 4.4: Memory consumption.

We observe that the first and second version consume very similar amount of memory and that the flexible OSPF with plugin has a non negligible overhead. The small steps we observe on the lines are due to the reception of LSAs from the neighboring router. The memory consumption after 40s of execution for the 3 versions are the following:

1. Vanilla OSPF: 4,85 MBytes.
2. Flexible OSPF without plugins: 4,93 MBytes.
3. Flexible OSPF with plugins: 6,22 MBytes.

The small difference in memory usage between the vanilla OSPF and the flexible OSPF without plugins is due to the metadata we create on start-up of the protocol to ensure the management of pluglets and plugins. This overhead is negligible.
The difference between the flexible OSPF without plugins and the one with plugins is due to multiple reasons: First, each plugin is given a 64 KB heap by the OSPF process when it is first executed. This heap then stays allocated for the future executions of the plugin. Second, when the eBPF files are loaded, they are loaded into the protocol memory. Each eBPF bytecode is about 1-10 KB heavy. Third, when bytecodes are injected, the implementation creates some more metadata related to it, and the uBPF VM also maintains data related to the VM state. Finally, some plugins call helper functions that allocates memory in the implementation. For example the plugin that originates a new LSA calls a helper function that creates this LSA and stores it in the LSDB.
All this together leads to about 1 MB of overhead for 5 plugins. This overhead seems manageable as today's routers do not suffer from memory problems and RAM is quite cheap.

4.3.3 Execution time analysis

The second thing we would like to measure is the impact of our implementation and of the plugins on the execution time of the main operations of OSPF. In OSPF, the most CPU intense operation is of course the SPF calculation. For that reason, we will compare the execution time of the SPF algorithm between different versions of OSPF. Because this time is quite low (about 6 ms for the GTS topology), as suggested in the article "How to run program or process on specific CPU cores on Linux" [33], we dedicate one core to the OSPF process and set a high process priority to it (nice value = -10). This avoids interruptions by other processes to twist our measurements. As for the memory consumption analysis, we will compare our flexible implementation of OSPF with the vanilla OSPF implementation. We will also compare the performances of interpreted and JIT compiled code. This will allow us to verify in practice if JITed code executes faster than interpreted code. The different versions we analyze are the following:

1. The vanilla OSPF from FRRouting 6.0.2 that we will use as a baseline. Indeed, this is the version we started from to implement our flexible OSPF. (*version 1*)
2. The flexible OSPF without any plugin. (*version 2*)
3. The flexible OSPF with a plugin monitoring the SPF calculation (interpreted). This plugin is composed of the two pluglets presented in Listings 4.1.1 and 4.1.2. Both are less than 10 lines of code long. (*version 3*)
4. The flexible OSPF with an **interpreted** plugin injected in the SPF calculation process. We use the plugin we presented in the use case Flexible Algorithm. The difference here is that we do not inject the other plugin to originate our new type of LSA. Thus, the only plugin we inject is simply an eBPF version of the `ospf_spf_next`

function of the protocol. The plugin is examining the links of a vertex LSA, and updating the list of candidates with any vertices not already on the list. This plugin is called for every vertex in the candidate list during the SPF calculation (i.e., more than 300 times for this topology). The plugin is about 160 lines of code long. (*version 4*)

5. The flexible OSPF with the same plugin as in *version 4* injected in the SPF calculation process but **JIT compiled**. (*version 5*)

To measure the execution time of the SPF algorithm, we launch OSPF on the test router and wait for 1 minute. This is more than enough for OSPF to receive all the LSAs from its neighbor and compute the SPF algorithm (we measure the last SPF calculation i.e., the one after having received all LSAs). To have representative results, we repeat the experiment 50 times (we wait 1 min between each experiment so that the adjacency between the two routers is lost). The time we measure is the time to compute the shortest path tree for all areas (*RFC2328 16.1* [31]). In our case there is only one area, the backbone area. Figure 4.5 shows the result of this experiment for each of the versions we presented above. The dotted line corresponds to the median execution time of the baseline (vanilla OSPF).

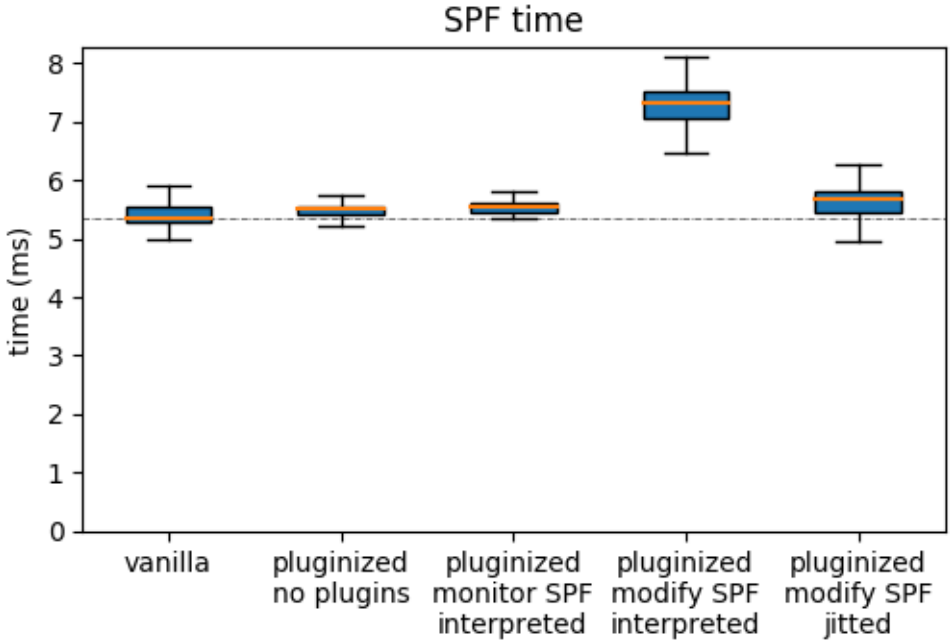


Figure 4.5: SPF algorithm execution time.

We observe that the execution time for version 2 (flexible ospf without plugins) is a little higher than for version 1 (vanilla ospf). With version 3, the monitoring plugin in interpreted mode leads to a small execution time overhead compare to version 2. Version 4 suffers from a significant overhead because of the execution of the plugin that modifies dijkstra (interpreted) more than 300 times. Version 5 which is the same as version 4 but with the JITed plugin experiences a significantly smaller overhead than version 4. These observations already allow us to draw some conclusions:

- The overhead of the flexible OSPF without plugins in terms of execution time is very small. This was excepted as the only difference between this version and the vanilla one is the initialization of some structures on start-up and the checks due to the insertion points (see Listing 3.2.1).

- The overhead induced by small plugins such as the monitoring plugin we inject in the third version is negligible. This is just a few more instructions to execute and thus has a very small impact. For more details about the complexity of the monitoring plugins, see Table B.1.
- In the version 4 we have a 160 lines of code interpreted plugin that is executed about 300 times during a single execution of the SPF calculation. In that case, the overhead is not negligible (about 33% compare to version 2). This high overhead can be explained by multiple factors: First, the cost of interpreting the bytecodes is high. Second, the plugins are using the provided API to get OSPF structures and manage their heap.
- Version 5 shows that when the code is JITed, the performances are a lot better. The overhead drops to about 3% compare to version 2. This version especially benefits from the fact that with JIT compilation, the eBPF bytecodes are compiled only on the first execution of the plugin, not on all the next ones. The overhead is explained by the fact that the plugin needs to call helpers from the API to get OSPF structures and manage its memory.

We still need to put things in perspective. Even though about 2/3 of the execution time of the SPF algorithm is spent in the `ospf_spf_next` function (the one overridden by the plugin), our plugin is only 160 lines of code long and calls many helper functions that do a lot of things. These helper functions are not executed in the VM and thus the part of the SPF calculation process executed in the uBPF VM is not the majority. The overhead we observe is therefore definitely not negligible for version 4. The more the plugins will do, the more this overhead will rise and will probably become unbearable. We can conclude that interpreting plugins is good for developing new protocol plugins and has some security advantages, but on the other hand in a real deployment perspective, the overhead in terms of execution time is huge. This convinces us even more that work should be put into the validation of plugins so that we can use them in JIT compiled mode and therefore benefit from the higher performances without leading to security breaches (this track will be discussed in the next Chapter).

Chapter 5

Discussions and future work

This Chapter aims to discuss the limitations of the solution we presented in this master thesis. We will also identify some possible future research tracks in the perspective of improving the presented solution.

In Section 5.1, we will talk about the validation process of plugins that could be put in place to verify the validity of plugins. In Section 5.2, we will discuss the possibility of sharing plugins between users. In Section 5.3, we will explain how the OSPF implementation should be revisited in the perspective of putting as much flexibility as possible in the protocol. In Section 5.4, we will talk about the implications of using our solution in multi-threaded implementations. Finally, in Section 5.5, we will highlight the lack of existing open source tools for generating OSPF topologies.

5.1 Validation of plugins

In this Section, we will use the term plugin as defined in Section 3.3. A plugin is thus a piece of software able to extend the OSPF implementation by being attached to a given insertion point. It can contain multiple eBPF bytecodes (i.e., multiple pluglets) and thus the validity of a plugin includes the validity of multiple eBPF bytecodes together with the validity of their association.
We showed in Section 4.3 that when plugins start to be big, the overhead in terms of execution time goes up when we interpret them. We also showed that using JITed plugins, the overhead is significantly smaller compared to the interpreted solution. One issue is that if a JITed bytecode crashes, the whole protocol would crash (see Section 3.1). In the perspective of deploying our solution, some work should be put to be able to take advantage of the JIT compilation possibility while keeping security. Indeed, the performance overhead of the plugins when interpreted could quickly become unsustainable when plugins start to be big. To allow the secure usage of JITed plugins, we propose to put in place a validation process for the plugins. Each plugin would need to be validated by an entity to get a certificate. This certificate would be a proof of sanity and would ensure that its execution would not lead to undefined behavior. We highlight some constraints that a plugin must respect to be considered as valid. This list is probably not exhaustive but gives the most important things the validator would have to check.
Each bytecode that belongs to the plugin has to fill the following conditions:

- A bytecode must terminate in any case. A bytecode that loops indefinitely would of course be unsafe to inject in the implementation. Some tools such as KITTeL [15] have been developed and allow to prove the termination of a C program.
- Each time a bytecode calls a helper function, the pointers given to the helper must be verified. These pointers must either have been received as argument of the

plugin (from the implementation), received from another helper function (from the implementation as well) or being a pointer to the bytecode's heap memory.

- Each bytecode must be memory safe. Each memory area allocated on the plugin's heap must be freed before the end of the bytecode. Exception can be accepted when a bytecode allocates memory with an ID (to share this block of memory among multiple bytecodes of the same plugin). In that case, the constraints is relaxed to the fact that this memory must be freed before the end of the whole plugin instead of the end of the bytecode (i.e., it can be freed in another bytecode from the same plugin).

Having a way to perform this checks to give some kind of certificate to a plugin would allow trusting plugins and execute them with fewer security concerns for the implementation internals. Several tools that rely on formal methods have started to appear to validate eBPF code [17]. These tools could help in building a static verifier for protocol plugins.

5.2 Sharing plugins among users

This Section discusses the possibility to share plugins among users (i.e., probably among network operators). Users that develop plugins to implement new protocol features could decide to publish them under an open source license. These plugins could then be used by other users to extend their own implementation. Different network operators often use routers from different vendors running different implementations. The possibility of sharing plugins is thus conditioned by some agreements between vendors:

- They must agree on the virtual machine they use to execute the bytecodes. In the presented solution we choose the uBPF VM but other virtual machines could be used for this purpose. Example of such other solution is WebAssembly [20]. It is a binary code format that offers compact representation, efficient validation and compilation, and safe low to no-overhead execution. As for eBPF, it is possible to compile C programs to WebAssembly.

- They must agree on the API the implementation exposes to the plugins. This includes the insertion points and helper functions to be standardized among their implementation.

The possibility of sharing plugins among users would allow network operators to easily tune protocols such as OSPF as they need by reusing open source plugins and sharing their own plugin with others. We think this openness would quickly improve protocols in different ways because of the emulation it would create.

5.3 Redesigning the OSPF implementation and careful choice of the API

To make OSPF as flexible as possible, the design of the implementation should be revisited. In its actual form, the OSPF implementations are not very customizable by design. As an example, when we wanted to implement a plugin to create a LSA of a new type (see Section 4.2), we faced an issue. In FRRouting's implementation of OSPF, the LSDB contains an array of databases where each cell corresponds to an LSA type. This array has a fixed size which is the number of different LSA types that exist. To be able to add new LSAs of type 13 (the one we created), we had to extend this array's size. This is just an example but there are many features in OSPF that are not extendable by nature. This is why the general design should probably be revisited by trying to think about all the possible

extensions users could want to implement.

As we already briefly discussed, the choice of the API the implementation provides is also one of the most determining thing in how flexible the implementation will be. When developing our flexible OSPF, the API we chose to expose has mainly been driven by the new functionalities and plugins we wanted to create. We think there is a lot more work to put into carefully choosing all the insertion points and the helper functions that are exposed. In the perspective of deploying the solution for network operators to use it, it would probably be IETF's role to specify the basic protocol operations and associated insertion points of the protocols. They also should define the set of helper functions that an implementation must expose such that plugins are able to interact with the implementation in an efficient and exhaustive way. This would radically change routing protocols by transforming them to a set of simple operations each of which would be easily customizable by anyone.

5.4 Plugin management in multi-threaded implementation

In Section 3.5.1, we introduced the pluglet context. We also explained that because our implementation uses a unique global variable to store the current pluglet execution context and that this variable is protected by a mutex, only one eBPF bytecode can execute at a time. Although the FRRouting implementation of the OSPF daemon is single threaded, having at most one bytecode executed at a time would be limiting if we wanted to multi-thread the OSPF implementation or to apply our solution to other multi-thread routing protocol.

A possible solution to remove this limitation could be to have one current context variable for each thread. Helper functions would then have to know from which thread they have been called and check the appropriate current context. Thread-Local Storage [14] is a mechanism by which variables are allocated such that there is one instance of the variable per extant thread. This easily implementable solution would probably fix this issue. Unfortunately, it requires significant support from the linker, dynamic linker, and system libraries and is thus not available everywhere.

Another solution to avoid this limitation would be to find a way for helper functions to have the information contained in the pluglet execution context in another way than using the global current context variable. But we did not find such other mechanism for the moment.

5.5 LSA generation tool

In Section 4.3, we presented the tests we made to measure the performances of our implementation. As described, we had to simulate a topology using IPMininet to be able to generate a LSDB that can be used by the SPF algorithm. Although this solution works, it is quite complicated in terms of configuration (we had to interface a Mininet router with the host machine, etc.) and has limitations in terms of size. Indeed, each simulated router runs processes and uses resources, making the size of the Mininet not infinite [47]. For our tests we simulated a topology of about 150 routers. This is already quite big but the SPF calculation time for that topology is only about 6 ms. Being able to generate a lot bigger topologies (e.g. 10 000 routers) would allow to put the algorithms of the protocol under higher stress making it easier to observe and analyze them.
Thus, a better way to perform this kind of tests would be to generate some concordant

LSAs to create a LSDB that simulates a certain topology without having to actually run it. Such tool would generate sets of LSAs that appear to be a logical, representing realistic topologies, and would send it to the router under test in the form of Link State Update packets. This would allow to generate a lot bigger topologies because there would be no need to actually run the topology we are simulating. Although some proprietary solutions exist [43], we are not aware of any similar open source tool. We think that developing such open source tool would be really useful for the community.

Chapter 6

Conclusion

In this master thesis, we showed how it is possible to adapt a routing protocol implementation such as OSPF to make it flexible by allowing the insertion of plugins at particular locations called insertion points. We splitted OSPF in different subroutines of the protocol that we call protocol operations and associate such insertion point to each of them. We allow users (most likely network operators) to tune each of these protocol operations using plugins. Plugins are software modules independent of the implementation, that can be hooked to the insertion points in a certain manner. Plugins are then executed inside the implementation in an execution environment: the uBPF virtual machine.

We demonstrated how easy it is to take advantage of this flexibility to modify the protocol in a given way. We especially showed that plugins can be used for monitoring purposes, allowing doing fine-grained monitoring without the need of recompiling the code or restart the router. We also implemented plugins to add a new type of LSA and to modify the SPF calculation process so that it takes into account the additional metric carried in the new LSA. This case shows the power of our solution by completely changing the behavior of one of the most important protocol operation of OSPF: the SPF calculation.

We studied the impact of our modified implementation and the execution of plugins in terms of performances and memory. We showed that the overhead in terms of memory is limited and should not cause problems for today's routers. We also showed that the overhead in terms of execution time can be high when interpreting plugins, but fortunately it heavily drops when using JIT compilation. We observed that overhead is proportional to the complexity of the plugins, making it negligible for simple monitoring plugins.

Although the solution we propose is a prototype and that there are some more tracks to explore (Chapter 5), we think the proposed approach could change the game for protocols development. Indeed, in an increasingly open internet world, networks needs are evolving very quickly. Traditional protocols implementations are very "static" by nature and the only way to tune them is by modifying configuration variables through the CLI, which limits the flexibility to what has been intended. We are convinced that in the current networks world, anyone should be able to tune routing protocols to best fit its needs. We also think that the fact that these plugins implementing protocol extensions could be shared among users, would create some emulation and make protocols evolve faster.

Bibliography

[1] University of Adelaide. *The Internet Topology Zoo: dataset*. URL: http://www.topology-zoo.org/dataset.html. [Online; accessed 02-June-2019].

[2] D. Awduche et al. *RSVP-TE: Extensions to RSVP for LSP Tunnels*. RFC 3209. RFC Editor, Dec. 2001.

[3] L. Berger et al. *The OSPF Opaque LSA Option*. RFC 5250. RFC Editor, July 2008.

[4] Amit N. Bhagat. *OSPF Packet Types*. URL: https://sites.google.com/site/amitsciscozone/home/important-tips/ospf/ospf-packet-types. [Online; accessed 03-June-2019].

[5] CCNA. *Designated Backup Designated Router*. URL: https://study-ccna.com/designated-backup-designated-router/. [Online; accessed 02-June-2019].

[6] CCNA. *OSPF overview*. URL: https://study-ccna.com/ospf-overview/. [Online; accessed 02-June-2019].

[7] Cilium. *BPF and XDP Reference Guide*. URL: https://scanfcilium.readthedocs.io/en/latest/bpf.html. [Online; accessed 02-June-2019].

[8] Cisco. *IP Routing: OSPF Configuration Guide*. URL: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_ospf/configuration/xe-16/iro-xe-16-book/iro-cfg.html. [Online; accessed 07-June-2019].

[9] R. Coltun et al. *OSPF for IPv6*. RFC 5340. http://www.rfc-editor.org/rfc/rfc5340.txt. RFC Editor, July 2008. URL: http://www.rfc-editor.org/rfc/rfc5340.txt.

[10] Quagga community. *Quagga Routing Suite*. URL: https://www.quagga.net/. [Online; accessed 07-June-2019].

[11] SANTA CRUZ and Aman Shaikh. "MANAGEMENT OF ROUTING PROTOCOLS IN IP NETWORKS". In: (Apr. 2019).

[12] Nick Desaulniers. *Interpreter, Compiler, JIT*. URL: http://nickdesaulniers.github.io/blog/2015/05/25/interpreter-compiler-jit/. [Online; accessed 03-June-2019].

[13] Bob Vachon Diane Teare Rick Graziani. *OSPF Implementation*. URL: http://www.ciscopress.com/articles/article.asp?p=2294214&seqNum=2. [Online; accessed 02-June-2019].

[14] GCC online documentation. *Thread-Local Storage*. URL: https://gcc.gnu.org/onlinedocs/gcc-3.3/gcc/Thread-Local.html. [Online; accessed 03-June-2019].

[15] Stephan Falke, Deepak Kapur, and Carsten Sinz. "Termination Analysis of C Programs Using Compiler Intermediate Languages". In: *22nd International Conference on Rewriting Techniques and Applications (RTA'11)*. Ed. by Manfred Schmidt-Schauß. Vol. 10. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011, pp. 41–50. ISBN: 978-3-939897-30-9. DOI: 10.4230/LIPIcs.RTA.2011.41. URL: http://drops.dagstuhl.de/opus/volltexte/2011/3123.

[16] Matt Fleming. *A thorough introduction to eBPF*. URL: https://lwn.net/Articles/740157/. [Online; accessed 03-June-2019].

[17] Elazar Gershuni et al. "Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions". In: *PLDI'19*. https://research.vmware.com/publications/simple-and-precise-static-analysis-of-untrusted-linux-kernel-extensions. June 2019.

[18] *Graphml to mininet topology script*. URL: https://github.com/sjas/assessing-mininet/blob/master/parser/GraphML-Topo-to-Mininet-Network-Generator.py. [Online; accessed 02-June-2019].

[19] Brendan Gregg. *Linux Extended BPF (eBPF) Tracing Tools*. URL: http://www.brendangregg.com/ebpf.html. [Online; accessed 03-June-2019].

[20] Andreas Haas et al. "Bringing the Web Up to Speed with WebAssembly". In: *SIGPLAN Not.* 52.6 (June 2017), pp. 185–200. ISSN: 0362-1340. DOI: 10.1145/3140587.3062363. URL: http://doi.acm.org/10.1145/3140587.3062363.

[21] Gianluca Iannaccone et al. "Analysis of Link Failures in an IP Backbone". In: *Proceedings of the 2Nd ACM SIGCOMM Workshop on Internet Measurment*. IMW '02. Marseille, France: ACM, 2002, pp. 237–242. ISBN: 1-58113-603-X. DOI: 10.1145/637201.637238. URL: http://doi.acm.org/10.1145/637201.637238.

[22] iovisor. *Userspace eBPF*. URL: https://github.com/iovisor/ubpf. [Online; accessed 02-June-2019].

[23] S. Knight et al. "The Internet Topology Zoo". In: *Selected Areas in Communications, IEEE Journal on* 29.9 (Oct. 2011), pp. 1765–1775. ISSN: 0733-8716. DOI: 10.1109/JSAC.2011.111002.

[24] CZ.NIC Labs. *The BIRD Internet Routing Daemon*. URL: https://bird.network.cz/. [Online; accessed 07-June-2019].

[25] Jose Liste. *Unleashing SR Traffic Engineering capabilities with SR Flexible Algorithms*. URL: https://www.youtube.com/watch?time_continue=1&v=l_RThbdDnko. [Online; accessed 02-June-2019].

[26] M. Masmano et al. "TLSF: a new dynamic memory allocator for real-time systems". In: *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.* July 2004, pp. 79–88. DOI: 10.1109/EMRTS.2004.1311009.

[27] Nick McKeown et al. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[28] Michael Morris. *Making BGP Our Core Enterprise Routing Protocol*. URL: https://www.networkworld.com/article/2348693/making-bgp-our-core-enterprise-routing-protocol.html. [Online; accessed 02-June-2019].

[29] J. Moy. *OSPF Protocol Analysis*. RFC 1245. RFC Editor, July 1991.

[30] John Moy. *OSPF Version 2*. RFC 1247. http://www.rfc-editor.org/rfc/rfc1247.txt. RFC Editor, July 1991. URL: http://www.rfc-editor.org/rfc/rfc1247.txt.

[31] John Moy. *OSPF Version 2*. STD 54. http://www.rfc-editor.org/rfc/rfc2328.txt. RFC Editor, Apr. 1998. URL: http://www.rfc-editor.org/rfc/rfc2328.txt.

[32] John Moy. *The Link State Advertisement header*. URL: https://www.freesoft.org/CIE/RFC/1583/110.htm. [Online; accessed 03-June-2019].

[33] Dan Nanni. *How to run program or process on specific CPU cores on Linux*. URL: http://xmodulo.com/run-program-process-specific-cpu-cores-linux.html. [Online; accessed 02-June-2019].

[34] D. Oran. *OSI IS-IS Intra-domain Routing Protocol*. RFC 1142. RFC Editor, Feb. 1990.

[35] Linux Foundation Collaborative Projects. *FRRouting*. URL: https://frrouting.org/. [Online; accessed 02-June-2019].

[36] Peter Psenak et al. *IGP Flexible Algorithm*. Internet-Draft draft-ietf-lsr-flex-algo-02. Work in Progress. Internet Engineering Task Force, May 2019. 23 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-lsr-flex-algo-02.

[37] Shubham Rana. *IPC using Message Queues*. URL: https://www.geeksforgeeks.org/ipc-using-message-queues/. [Online; accessed 02-June-2019].

[38] Giampaolo Rodola. *Psutil*. URL: https://pypi.org/project/psutil/. [Online; accessed 02-June-2019].

[39] E. Rosen, A. Viswanathan, and R. Callon. *Multiprotocol Label Switching Architecture*. RFC 3031. http://www.rfc-editor.org/rfc/rfc3031.txt. RFC Editor, Jan. 2001. URL: http://www.rfc-editor.org/rfc/rfc3031.txt.

[40] M. El-Sayed and J. Jaffe. "A view of telecommunications network evolution". In: *IEEE Communications Magazine* 40.12 (Dec. 2002), pp. 74–81. ISSN: 0163-6804. DOI: 10.1109/MCOM.2002.1106163.

[41] A. Shaikh et al. "An OSPF topology server: design and evaluation". In: *IEEE Journal on Selected Areas in Communications* 20.4 (May 2002), pp. 746–755. ISSN: 0733-8716. DOI: 10.1109/JSAC.2002.1003041.

[42] J. M. Smith and S. M. Nettles. "Active networking: one view of the past, present, and future". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 34.1 (Feb. 2004), pp. 4–18. ISSN: 1094-6977. DOI: 10.1109/TSMCC.2003.818493.

[43] Spirent. *Spirent TestCenter Protocols*. URL: https://www.spirent.com/products/testcenter/protocols#routing. [Online; accessed 08-June-2019].

[44] Bob Lantz Mininet Core Team. *Mininet*. URL: https://github.com/mininet/mininet/wiki/Documentation. [Online; accessed 02-June-2019].

[45] David L Tennenhouse and David J Wetherall. "Towards an active network architecture". In: *ACM SIGCOMM Computer Communication Review* 26.2 (1996), pp. 5–17.

[46] Olivier Tilmans. *IPMininet*. URL: https://github.com/cnp3/ipmininet. [Online; accessed 02-June-2019].

[47] Volkan Yazıcı. *Benchmarking Mininet*. URL: https://vlkan.com/blog/post/2013/04/19/benchmarking-mininet/. [Online; accessed 08-June-2019].

Appendix A

Detailed OSPF plugins API

A.1 Insertion points

For each of the insertion point we propose, we give: its ID, the OSPF function in which it has been integrated and a small description of what this function does.

- `SPF_CALC`: `ospf_spf_calculate` function. This function calculates the shortest path tree for an area. The argument given to the plugin is the following:

```
1    struct arg_plugin_spf_calc {
2        heap_t heap;
3        struct ospf_area *area;
4    };
```

- `HELLO_SEND`: `ospf_hello_send` function. This function is the function in charge of sending hello packets. The argument given to the plugin is the following structure:

```
1    struct arg_plugin_hello_send {
2        heap_t heap;
3        struct ospf_interface *oi;
4    };
```

- `LSA_FLOOD`: `ospf_flood` function. This function is used to flood the received LSAs. When a LSA is received (through a LSU for example), `ospf_flood` is called if the received LSA is not yes in the LSDB or if the LSA is more recent than the one stored in the LSDB. The argument given to the plugin is the following structure:

```
1    struct arg_plugin_lsa_flood {
2        heap_t heap;
3        struct ospf_lsa *lsa;
4    };
```

- `ISM_CHANGE_STATE`: `ism_change_state` function. This function is called when the state of the interface state machine needs to change (this FSM is used for the adjacency formation). The argument given to the plugin is the following structure:

```
1    struct arg_plugin_ism_change_state {
2        heap_t heap;
3        struct ospf_interface *oi;
4        int new_state;
5    };
```

- `OSPF_SPF_NEXT`: `ospf_spf_next` function. This function is part of the SPF calculation process. It starts from a vertex (router of network) and updates the list of candidates with any vertices not already on the list. The argument given to the plugin is the following structure:

```
struct arg_plugin_ospf_spf_next {
    heap_t heap;
    struct vertex *v;
    struct ospf *ospf;
    struct ospf_area *area;
    struct pqueue *candidate;
};
```

- `DR_ELECTION`: `ospf_dr_election` function. This function is in charge of electing DR and BDR routers for a network. The argument given to the plugin is the following structure:

```
struct arg_plugin_dr_election {
    heap_t heap;
    struct ospf_interface *oi;
};
```

A.2 Helper functions

This section lists all the helper functions we expose to the plugins in our API. We give the signature of each function with a small description.

Generic and memory management functions

gettimeofday: standard `gettimeofday` function.

strcpy: standard `strcpy` function.

```
uint16_t plugin_ntohs(uint16_t value);
```

Operates as the standard `ntohs` function. The standard `ntohs` fails compiling with Clang.

```
uint32_t plugin_ntohl(uint32_t value);
```

Operates as the standard `ntohl` function. The standard `ntohl` fails compiling with Clang.

```
void *plugin_malloc(unsigned int size);
```

Operates as the `malloc` function but on the plugin's heap.

```
void plugin_free(void *ptr);
```

Operates as the `free` function but on the plugin's heap.

```
void *plugin_realloc(void *ptr, unsigned int size);
```

Operates as the `realloc` function but on the plugin's heap.

```
void *plugin_malloc_with_id(unsigned int id, unsigned int size);
```

Operates as the `malloc` function but on the plugin heap and associates an id to the memory block.

```
void *plugin_get_memory_with_id(unsigned int id);
```

Gets a pointer to the memory area identified by '`id`'. Returns `NULL` if there is no such memory area.

Getter functions

Getter functions are used to get copies of OSPF internal variables by giving the pointer to the variable we want to get. We only document the functions for which the outcome is not completely obvious.

```
int get_ospf_interface(struct ospf_interface *oi, struct ospf_interface *oi_copy);
```

```
int get_interface(struct interface *ifp, struct interface *ifp_copy);
```

```
int get_ospf_interface_list(struct list *oilist, struct ospf_interface **oilist_copy, struct ospf_interface **oi_list_addresses);
```

Put copies of all the ospf interfaces in '`oilist`' in '`oilist_copy`'. The pointers to the real OSPF variables are stored in '`oi_list_addresses`'. Storing the real pointers is useful in cases when the plugin needs to call some other helper functions and give them the pointers to the OSPF variables and not to a copy.

```
int get_ospf_lsa(struct ospf_lsa *lsa, struct ospf_lsa *lsa_copy);
```

```
int get_lsa_header(struct lsa_header *lsah, struct lsa_header *lsah_copy);
```

Copies the LSA header structure pointed by '`lsah`' into '`lsah_copy`'.

```
int get_lsa_with_length(struct lsa_header *lsah, struct lsa_header *lsah_copy);
```

Same as `get_lsa_header` but copies also what is after the LSA header (looking at length field).

```
int get_ospf_area(struct ospf_area *area, struct ospf_area *area_copy);
```

```
int get_ospf(struct ospf *ospf, struct ospf *ospf_copy);
```

```
int get_vertex(struct vertex *vertex, struct vertex *vertex_copy);
```

```
struct vertex *get_candidate(struct pqueue *candidate, int stat);
```

Returns a pointer to the candidate number '`stat`' in the pqueue. This helper is needed because of the particularity of the pqueue structure.

Setter functions

Setters are used to set OSPF internal variable to a given value.

```
int set_ospf_interface(struct ospf_interface *oi, struct ospf_interface *oi_copy);
```

Copies the ospf interface structure pointed by '`oi_copy`' to the pointer '`oi`'.

```
int set_ospf_area(struct ospf_area *area, struct ospf_area *area_copy);
```

Copies the ospf area structure pointed by '`area_copy`' to the pointer '`area`'.

OSPF functions

Some functions from the OSPF implementation can be really useful when developing plugins. We identified some and expose them in our API. We wrap these functions in our own functions to be able to perform some checks before executing them (checking if the pointers given by the plugin are not `NULL`, etc).

```
struct ospf_lsa *plugin_ospf_lsa_install(struct ospf *ospf, struct ↩
    ospf_interface *oi, struct ospf_lsa *lsa);
```

Installs the LSA pointed by '`lsa`' into the LSDB for the OSPF instance represented by '`ospf`'. This function can for example be used in plugins tuning the treatment of received LSAs or to install some newly created self-originated LSAs.

```
unsigned int plugin_ospf_nexthop_calculation(struct ↩
    arg_plugin_ospf_spf_next *s, struct vertex *w, struct router_lsa_link↩
     *l, unsigned int distance, int lsa_pos);
```

Calculates nexthop from root through V (parent) to vertex W (destination), with given distance from root to W. Useful for tuning the SPF calculation algorithm.

```
int plugin_ospf_flood_through_area(struct ospf_area *area, struct ↩
    ospf_neighbor *inbr, struct ospf_lsa *lsa);
```

Floods the LSA '`lsa`' into the area '`area`'. Useful to propagate newly created LSAs for example.

```
int plugin_lsa_link_broadcast_set(struct stream **s, struct ↩
    ospf_interface *oi, uint32_t metric);
```

```
struct ospf_lsa *plugin_ospf_lsa_new_and_data(struct stream *s, struct ↩
    ospf_area *area);
```

Creates a new LSA in the OSPF implementation based on the stream received as argument.

```
int plugin_ospf_lsa_has_link(struct lsa_header *w, struct lsa_header *v);
```

Returns index of link back to V from W, or -1 if no link found.

```
void plugin_trickle_up(int index, struct pqueue *queue);
```

Trickle-sort up the node of index '`index`' from the queue towards root.

```
struct ospf_lsa *plugin_ospf_lsa_lookup_by_id(struct ospf_area *area, ↩
    uint32_t type, struct in_addr id);
```

Returns a pointer to the LSA identified by '`id`' in the LSDB. Return `NULL` if nothing was found.

```
struct ospf_lsa *plugin_ospf_lsa_lookup(struct ospf *ospf, struct ↩
    ospf_area *area, uint32_t type, struct in_addr id, struct in_addr ↩
    adv_router);
```

```
void plugin_pqueue_enqueue(void *data, struct pqueue *queue);
```

Enqueues '`data`' in the queue '`queue`'.

```
struct vertex *plugin_ospf_vertex_new(struct ospf_lsa *lsa);
```

Creates a vertex from the LSA pointed by '`lsa`' and return a pointer to this vertex.

Appendix B

Implemented plugins statistics

The following table presents the statistics of the implemented plugins. Each line corresponds to an eBPF bytecode (i.e, a pluglet). Multiple bytecodes that are at the same insertion point belong to the same plugin.

Table B.1: Statistics of implemented pluglets

eBPF bytecodes	**Insertion point**	**Anchor**	**External calls**	**eBPF Insts**	**ELF (Bytes)**	**LOC**
Originate LSA type 13	SPF_CALC	PRE	41	318	5088	96
Modify Dijkstra	OSPF_SPF_NEXT	REP	45	365	5920	157
Monitor LSA flood	LSA_FLOOD	PRE	3	24	1080	9
Monitor Hello packets	SEND_HELLO	PRE	9	43	1416	20
Monitor ISM	ISM_CHANGE_STATE	PRE	8	39	1408	15
Monitor Dijkstra (PRE)	SPF_CALC	PRE	2	8	880	7
Monitor Dijkstra (POST)	SPF_CALC	POST	5	39	1320	15

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**