UCLouvain

École polytechnique de Louvain

Bringing Forward Erasure Correction to IPv6 leveraging Segment Routing

Author: **Louis Navarre**
Supervisor: **Olivier Bonaventure**
Readers: **François Michel, Mathieu Xhonneux**
Academic year 2020–2021
Master [120] in Computer Science and Engineering

Acknowledgements

First of all, I would like to sincerely thank Prof. Bonaventure, for this interesting and challenging subject. I would also like to express my gratitude for his ever-growing number of ideas to improve the content of this thesis.

Secondly, I thank François Michel for his help throughout this thesis, as well as for his valuable corrections. Our many discussions on Friday morning - the day after the thesis meetings - were very instructive and allowed me to move in the right direction, both technically and psychologically.

I am also grateful to Mathieu Jadin, whose patience towards me is matched by his technical assistance.

Finally, I would like to thank my friends and my family, who have put up with my changing moods during this year and who have supported me in my work. I am also grateful to Céline for having supported, reassured and listen to me throughout this year.

Abstract

IPv6 Segment Routing (SRv6) is a recent implementation of the source routing paradigm in IPv6 network. It basically allows packets to be forwarded across specific paths possibly different from the shortest one but also permits to define network functions altering or analyzing the traffic. This thesis is based on the Linux kernel implementation of SRv6, and these functions are rigid since they are directly implemented in it. Hence, adding new behaviors means recompiling the kernel. A programmability framework has recently been added to SRv6 to enable sandbox programming directly in the kernel leveraging the eBPF support in Linux. We take advantage of this support to design, implement, and assess a Forward Erasure Correction (FEC) technique that transparently protects IPv6 packets. This extension is designed for Internet of Things (IoT) devices with resource constraints that cannot implement FEC directly and potentially rely on costly retransmission. We implement our solution as a two-parts plugin, an encoder and a decoder, using eBPF on the Linux kernel, and our design supports a generic FEC Framework with two FEC Schemes: RLC and XOR-based encoding. We evaluate the performance of our plugin with the experimental design approach in different scenarios, particularly over IoT protocols such as MQTT. Our results show that we can improve the network quality by recovering from packet losses transparently for the devices. We also design a controller that dynamically triggers the plugin when we detect losses based on the observed quality of the network, and show that we keep good recovering capabilities while decreasing the overhead induced by FEC.

Contents

Introduction

The majority of protocols rely on retransmissions in case data is lost between the sender and the receiver. These drops can be due to congestion or transmission error, for example in wireless networks. This retransmission mechanism ensures that all sent data will eventually reach the receiver but has two important drawbacks. First, the sender must keep unacknowledged data inside its sending buffer in case retransmission is needed. However, this can become a problem for embedded devices with resource constraints, such as in the Internet of Things (IoT). The second drawback concerns the increased delay due to the retransmission. Even if standard protocols such as TCP nowadays support fast retransmission techniques, the tail latency potentially increases when the sender retransmits data, and this can be a problem for delay-sensitive applications [1]. On the other hand, unreliable protocols (such as UDP at the transport layer) do not provide a retransmission mechanism, and an erased packet will simply be lost for the receiving application.

To cope with these issues, multiple Forward Erasure Correction techniques have been suggested. FEC enables recovery from potential packet losses by adding redundancy into the network. However, this principle increases the computational work of the device implementing it, as it must build a Framework, generate the redundancy and potentially recover from losses. This can be a problem for IoT devices given the CPU cost of FEC, and these embedded systems cannot always afford this overhead.

This master thesis presents a different approach to propose Forward Erasure Correction, by performing the computation inside the network and removing this burden from the host devices. With IPv6 Segment Routing [2] and its network programmability [3] in particular, we can embed function calls to SRv6 routers when forwarding packets. As the used SRv6 implementation stands in the Linux kernel, it becomes mandatory to work in the kernel space to implement FEC. However, leveraging the recently added End.BPF action inside the Linux implementation of SRv6 [4], we can add FEC as a plugin running directly inside the kernel. We implement encode and decode functions deployed on SRv6 routers, and the IoT devices only need to send the packets to protect with a Segment Routing Header with the Segment Identifiers of the routers of the plugin. In this work, we implement two coding functions with XOR and Random Linear Codes and show the benefits of the implementation in multiple scenarios. We finally present a controller that dynamically triggers the redundancy generation when we detect losses into the network.

The open-source repository containing our prototype implementation is available at the following address: `https://github.com/louisna/FEC-SRv6-libbpf.git`

This thesis contains five important chapters:

- **Chapter 1** presents the state-of-the-art technologies and the background knowledge to fully understand this thesis. We first present Forward Erasure Correction, then detail the extended Berkeley Packet Filter technology, its architecture and how we can build programs upon it. Finally, we introduce IPv6 Segment Routing and especially the End.BPF interface;
- **Chapter 2** details our solution at a high level, the design of FEC at the network layer and the challenges we face. This chapter is not specific to the Linux implementation of SRv6 in the Linux kernel, but proposes the conceptual approach of SRv6-FEC.
- **Chapter 3** presents our prototype implementation. We explain its particularities, especially concerning the FEC Schemes. We finally discuss the technical limitations of the prototype, mainly caused by the support of eBPF in the Linux kernel.
- **Chapter 4** assesses the plugin in different scenarios to show the benefits of our implementation in lossy networks. It also presents benchmarks.
- **Chapter 5** summarizes this thesis, presents the benefits and drawbacks of our approach and shows our contributions to the open-source community. Finally, we discuss the future direction of this work.

Chapter 1

State of the art and background knowledge

This section presents the technical aspects and technologies on which this thesis is constructed. We first give an introduction to Forward Erasure Correction (FEC). Then, we briefly describe eBPF, a virtual machine inside the Linux kernel. This recent technology allows the user to define at runtime specific behavior that the kernel must follow in a given region of the datapath. Finally, we present the Segment Routing technology, and how specifically IPv6 Segment Routing integrates eBPF inside the datapath to create plugins at the network layer of the kernel.

1.1 Forward Erasure Correction (FEC)

Depending on the protocol, applications use reliable or unreliable communications. Their behavior is different when considering communication in lossy networks.

Real-time applications generally rely on unreliable protocols such as UDP [5, 6]. A lost packet will not be recovered, resulting in a performance drop for the application. Over a reliable protocol such as TCP or QUIC, a dropped packet will also decrease the performance due to the need for retransmission. Different mechanisms have been developed to detect as fast as possible these losses [7, 8] but often rely on retransmission timers. However, these timers are costly since their value is often a multiple of the round-trip time (RTT), hence resulting in an increased tail latency [9].

The objective of Forward Erasure Correction (FEC) is to add redundancy to the network. When detecting a lost packet, FEC uses this redundancy to recover it, hence avoiding consuming retransmission (reliable protocols) or completely lost data (unreliable protocols). We call these redundant packets *repair symbols*, and the data packets *source symbols*. Figure 1.1 presents an example of FEC, where the fifth source symbol is lost but recovered when the repair symbol is received.

Different Forward Erasure Correction mechanisms exist, relying on specific coding algorithms to compute the repair symbols, and we call them *FEC Schemes* through this thesis.

Theoretical works and documentation about Forward Erasure Correction present algorithms to recover from lost bits/bytes. However, here we consider the use of FEC to protect entire packets in an IPv6 network. We can adapt the byte-level coding to packet-level coding by

applying the coding algorithm byte-per-byte (BpB) between two source symbols. Figure 1.3 shows how we can leverage a byte-level coding algorithm to encode entire packets. Consequently, we see that the time complexity of a FEC Scheme is multiplied by the size of the symbol to encode. Resulting from the idea to protect real-life packets, we must consider that they have different lengths. We hence may need to add padding at the end of the shortest symbols so that the coding algorithm is correctly executed byte-per-byte on all of them.

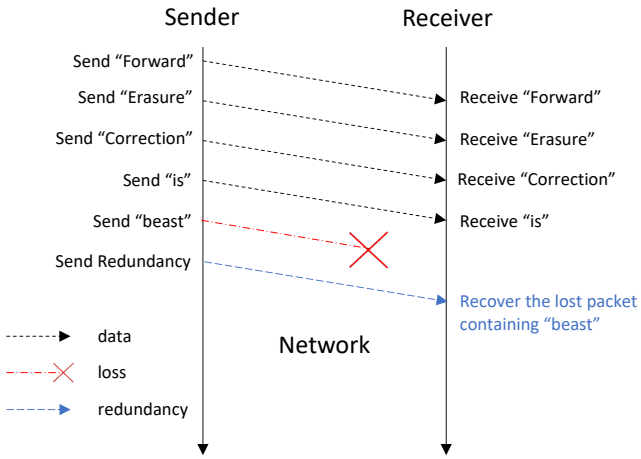


Figure 1.1: Example of Forward Erasure Correction.

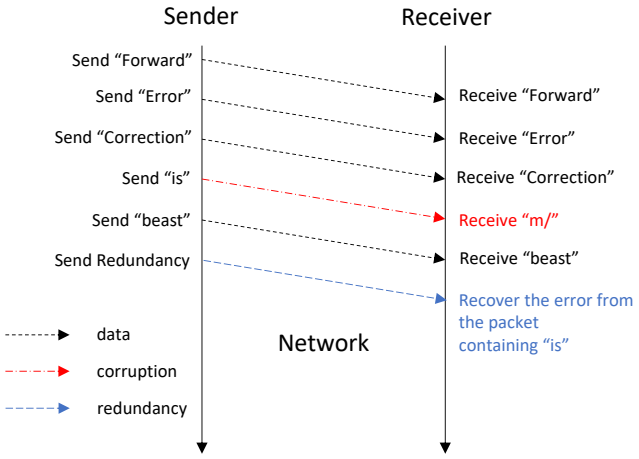


Figure 1.2: Example of Forward Error Correction.

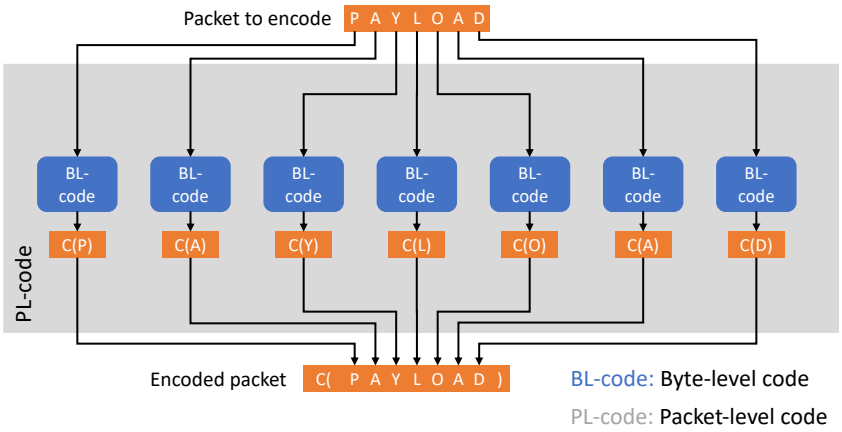


Figure 1.3: We can encode at the packet level by leveraging a byte-level erasure code byte-per-byte.

1.1.1 Forward Error Correction and Forward Erasure Correction

Two types of problems can arise when transmitting a packet: error and erasure. We talk about *errors* when a packet is corrupted, i.e., the content unexpectedly changed. Forward Error Correction techniques allow recovery from these corruptions, as shown in Figure 1.2. On the other hand, an *erasure* means that the packet is completely lost for the receiver. It can be due to congestion in the network but also happens in wireless networks [10] without congestion. We then talk about Forward Erasure Correction to recover from these losses (Figure 1.1). This thesis focuses on Forward Erasure Correction. Indeed, the majority of transport protocols (TCP and UDP) already implement a mechanism to detect these *errors*.

1.1.2 Mathematical aspects of FEC codes

Forward Erasure Correction is a field of coding theory; we also use the term *erasure code* to refer to a FEC code [11]. An erasure code transforms a message of k symbols into a longer message (codeword) of $n > k$ symbols, such that the original message can be recovered with a subset of the n symbols of the codeword [11]. We define the *coding rate* r as $r = \frac{k}{n}$. This quantity represents the proportion of useful data (i.e., not redundant) in the codeword. We also call the *reception efficiency* the proportion of symbols we have to receive (among the n symbols) to recover the initial message. Table 1.1 shows an example of an erasure code taking two symbols and outputting a codeword of length 3 (code rate of $\frac{2}{3}$). We define the alphabet Σ the set of possible values for a symbol. In the given example, $\Sigma = \{0, 1\}$.

message	00	01	10	11
codeword	101	111	110	001

Table 1.1: Example of an erasure code with $k = 2$ and $n = 3$.

message	00	01	10	11
codeword	001	010	100	111

Table 1.2: Example of a systematic erasure code with $k = 2$ and $n = 3$.

An erasure code is a *maximum distance separable* (MDS) code when the minimum Hamming distance between every possible codeword of the erasure code is equal to $n-k+1$. The minimum Hamming distance d is defined as

$$d := \min_{\substack{m_1, m_2 \in \Sigma^k \\ m_1 \neq m_2}} \Delta[C(m_1), C(m_2)], \tag{1.1}$$

Where Σ^k represents all words of length k from the alphabet Σ, C is the erasure code (the mapping), m_i the messages formed from the alphabet Σ, and Δ the *Hamming Distance*. For binary inputs, the Hamming distance is simply the sum of bitwise XOR of the two codewords:

$$\Delta(a, b) := \sum_{i=0}^{n-1} (a_i \oplus b_i).$$

The erasure code from Table 1.1 is not a maximum distance separable code, since the minimum Hamming distance is 1 ($\Delta(101, 111) = 1$) and $n - k + 1 = 2$. An MSD code is also called an optimal erasure code and has the property that reception of any k symbols from the n codeword will allow for a complete recovery of the initial k message. An example of an optimal erasure code is the Reed-Solomon [12] code, but not all codes are necessary MDS [11].

1.1.3 Encoding Algorithms

Multiple coding algorithms, or FEC Schemes, exist in the literature and we present two of them here: the XOR-based coding and Random Linear Codes (RLC). Let us note that this thesis only focuses on systematic codes. A systematic code is a code where the output codeword contains the input symbols. The example given in Table 1.1 presented a non-systematic code since the input message 00 is mapped to 101. On the contrary, Table 1.2 presents a systematic erasure code since the input symbols are always embedded at the beginning of the coded output. Systematic codes offer two main advantages. First, in a coding channel (i.e., Forward Erasure Correction channel) where the source symbols arrive sequentially, the coding function can directly forward them because the beginning of the coded output (the k first symbols of the codeword) exactly contains the k input symbols. The $n - k$ repair symbols are then generated, and their concatenation gives the complete codeword. Second, the receiver of the codeword does not need to decode the

source symbols, hence reducing the computation overhead. If no loss is detected, we remove the $n-k$ last symbols of the codeword to transmit the initial source symbols without additional processing.

1.1.3.1 XOR coding

The XOR coding function is intuitive and straightforward to implement. Only one repair symbol can be generated from the k received source symbol, resulting in $k = n-1$ and a code rate of $\frac{n-1}{n}$. The repair symbol R is generated by XOR-ing ($\oplus$) the source symbols S_1, S_2, $\cdots$, S_{n-1}, as showed in equation 1.2.

$$R := S_1 \oplus S_2 \oplus \cdots \oplus S_{n-1}. \tag{1.2}$$

The XOR coding function can recover the loss of exactly one source symbol (if only the repair symbol is lost during the transmission, the source symbols are not impacted and can be handled as if nothing happened). We recover the loss of the source symbol S_i using the other source symbols and the repair symbol, as presented in equation 1.3.

$$S_i := R \oplus S_1 \oplus S_2 \oplus \cdots \oplus S_{i-1} \oplus S_{i+1} \oplus \cdots \oplus S_{n-1}. \tag{1.3}$$

The XOR coding function is simple to implement but offers low protection as at most one source symbol can be recovered for the whole codeword. It cannot for example recover the loss of two source symbols of the initial message. Moreover, it is theoretically impossible to reach a different code rate than $\frac{n-1}{n}$.

1.1.3.2 Random Linear Codes

Random Linear Codes (RLC) is another coding function having a more adaptive code rate than XOR. We generate the repair symbols as a linear combination of the source symbols. The coefficients multiplying the source symbols are pseudo-randomly generated to increase the probability of obtaining independent equations for the different repair symbols. The seed used to obtain these values is forwarded to the decoding process to reproduce them. We can generate multiple repair symbols for the same set of source symbols, but each with a different set of pseudo-random coefficients. On the decoding process, we build a system of equations where the variables are the lost source symbols, and the received source symbols alongside the repair symbols form the independent terms. There is an equation for each generated repair symbol. We can then solve the obtained linear system containing as many equations as the minimum between the number of received repair symbols and unknowns. The solutions of the system are the recovered source symbols.

Toy example Let us consider a simple example where we need to protect S_1,S_2 and S_3 with two repair symbols R_1 and R_2. Using our pseudo-random generator, we compute the coefficients $c_i, i \in [1, 6]$. Equation 1.4 shows how we generate the repair symbols.

$$\begin{aligned} c_1 * S_1 + c_2 * S_2 + c_3 * S_3 &= R_1, \\ c_4 * S_1 + c_5 * S_2 + c_6 * S_3 &= R_2. \end{aligned} \tag{1.4}$$

Let us now consider that we lost S_2 and S_3. In this case it is impossible to recover them both using only the first repair symbol. On the decoding side, we build a linear system with the received source and repair symbols and obtain the system below (Equation 1.5). The right-end

side of the equations represents the independent terms, and the left-end side represents the variables (i.e. lost source symbols).

$$\begin{aligned} c_2 * S_2 + c_3 * S_3 &= R_1 - c_1 * S_1, \\ c_5 * S_2 + c_6 * S_3 &= R_2 - c_4 * S_1. \end{aligned} \tag{1.5}$$

The above system is of full rank if the equations are linearly independent. By using pseudo-random coefficients we have a high probability to obtain a solvable system. Solving this system with Gaussian elimination yields the recovery of the two lost symbols.

1.1.4 FEC Framework

As seen previously, multiple FEC Schemes exist and can be implemented to create a coding channel. However, even if the coding algorithms of these schemes are different, we can extract a common behavior, for example, what information must be communicated between the encoder and the decoder and how to transfer this information. We use the term FEC Framework to name this building block that provides an abstraction to the FEC Schemes.

There exist two types of FEC Frameworks: the Block and Convolutional Frameworks, respectively defined by RFC6363 [13] and RFC8680 [14]. These two Frameworks implement standard communication tools on top of Block codes and Convolutional codes.

1.1.4.1 Block codes

Block codes are the most standard way to encode source symbols, taking as input k symbols and outputting n symbols, noted (n, k). Considering a sequential flow of input symbols, a systematic Block code treats them by blocks of size k, each time generating $n - k$ repair symbols forwarded in the coding channel. Once the block is complete and the output codeword produced, we move to the next block and repeat this process. Figure 1.4 presents an example of a (3, 2) Block code, where the input symbols S_i are received sequentially, and each block j generates a unique repair symbol R_j. We send the repair symbols after the source symbols of the current block since they are computed based on the symbols of the block.

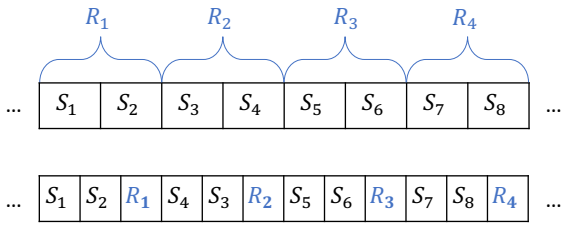


Figure 1.4: Example of a (3, 2) Block code protecting 8 source symbols. The upper part of the image shows how the source symbols are protected. The lower part shows the order in which the symbols are sent.

1.1.4.2 Convolutional codes

Convolutional codes are another strategy for Forward Erasure Correction. Instead of relying on successive blocks of source symbols to generate the repair symbols, Convolutional (or Sliding Window) codes consider the symbols as a stream and compute the repair payloads incrementally. This Sliding Window is defined by (n, k, S), with the window size S, the step value of the window

k, and the output codework length n. At each window of S source symbols, the Convolutional code outputs n symbols for the iteration and then shifts the window of k symbols. If $k < S$, the last $S - k$ symbols of the current window will be used for the next iteration. The code rate is $\frac{k}{n}$. Convolutional codes are a generalization of block codes: a (n, k) block code is the same as a (n, k, k) Convolutional code.

Figure 1.5 presents an example of a (3, 2, 4) Convolutional code protecting 8 source symbols. We see that S_3 and S_4 belong to the windows generating R_1 and R_2, while R_2 and R_3 both need S_5 and S_6. A naive implementation of a Convolutional code may require that S_3 and S_4 are forwarded once R_2 is generated, but this could lead to a long delay for these symbols since R_2 also needs S_5 and S_6 to be computed. A better approach is to store an internal copy of each input symbol in memory and forward it as soon as possible, achieving the order in which the symbols are sent, as presented in the lower part of Figure 1.5.

The main difference between Block and Convolutional codes is that the first needs a number of source symbols equal to the block size, whereas the latter only needs a number equal to the window step (because the $S - k$ source symbols have already been received for the computation of the previous repair symbols).

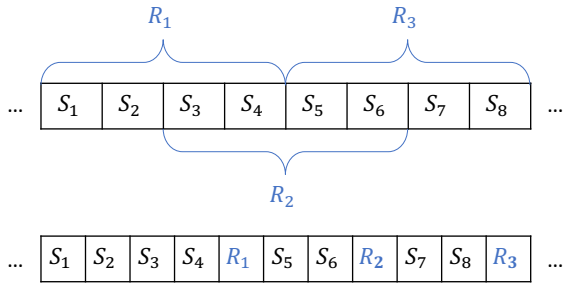


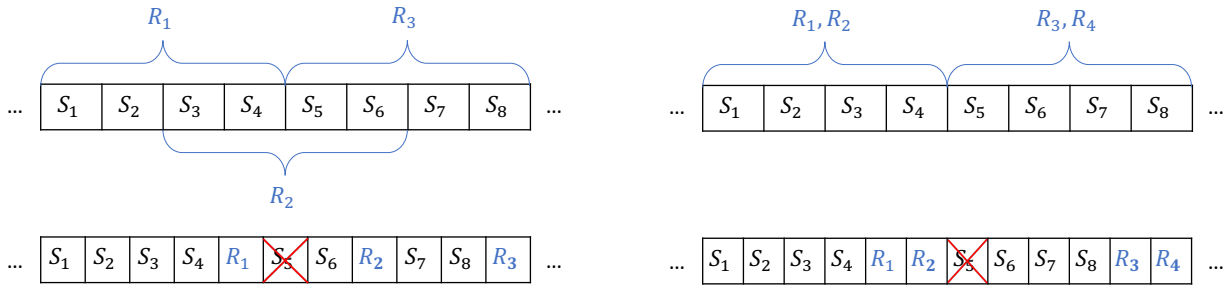
Figure 1.5: Example of a (3, 2, 4) Convolutional code protecting 8 source symbols. The upper part of the image shows how the source symbols are protected. The lower part shows the order in which the symbols are sent.

1.1.4.3 The case of Convolutional Random Linear Codes

We can adapt RLC for Convolutional codes. The algorithm for RLC is the same as presented earlier, but leveraging Convolutional codes gives strong recovering capabilities concerning burst losses. We can recover from them by combining the linear equations involving the same source symbols but with different windows by leveraging the fact that RLC constructs linear equations. From Figure 1.5, we can for example recover the burst loss of S_5 and S_6 by combining the two equations given by R_2 and R_3, with the reception of S_3 and S_4. We would obtain a linear system of two equations with two unknowns, which is solvable if the two equations are independent.

Even if Block codes allow for burst loss recovering, Convolutional codes provide lower delay for recovering lost packets [15], especially with Random Linear Codes. Let us consider a variant of the example presented in François Michel's master's thesis [16] of a (3, 2, 4) Convolutional RLC and a (6, 4) Block RLC. Both schemes have a $\frac{2}{3}$ code rate, but the Convolutional code provides a better delay to recover the packets: if S_4 is lost, the Convolutional code will recover it after the reception of S_5 and R_1. On the other hand, the Block code will have to wait for the reception of S_5, S_6, S_7 and R_2 to recover the lost symbol. To reach the same latency as the

Convolutional code, we could use a (3, 2) Block code, but we would lose the burst loss recovering capability. Convolutional RLC thus offers a nice combination of delay to recover a packet and protection capabilities.



(a) Convolutional code (3, 2, 4).

(b) Block code (6, 4).

Figure 1.6: Example of Block/Convolutional codes where we lose S_5. The upper part of the images shows how the source symbols are protected. The lower part shows the order in which the symbols are sent.

1.1.5 Limitations of Forward Erasure Correction

Forward Erasure Correction is not perfect and has multiple drawbacks. Foremost, the redundancy data increases the link utilization. If the packet losses occur because of congestion, we can further increase it and the quality of the network may decrease (an internet draft under development attempts to standardize the use of FEC at the transport layer to take into account the congestion control mechanism of reliable protocols [17]). Secondly, FEC is not optimal in the sense that some lost packet cannot always be recovered, for example, due to a very long burst loss. Over a reliable transport protocol, retransmissions may be needed to overcome these failures of FEC. Finally, FEC comes along with a computational overhead: the FEC Framework must map every protected packet to a source symbol and generate the repair symbols. Considering the RLC FEC Scheme, the decoder must solve a computationally intensive linear system to recover the lost symbols. However, some real-time applications require a very short and stable delay and the value of a recovered packet could be meaningless after a few milliseconds [18].

1.2 Extended Berkeley Packet Filter

Berkeley Packet Filter (BPF) is a flexible virtual machine lying in the Linux kernel, allowing to execute user-defined programs at various hook points in the datapath while ensuring that the programs do not jeopardize the kernel integrity. The domain of applications is broad, ranging from computer tracing to networking and security. BPF solves the rigid aspect of the Linux kernel. Indeed, developing new tools requires extending and recompiling the kernel source code or load modules.

Historically, the first version of BPF (called classic Berkeley Packet Filter, or cBPF) was launched in 1992 [19] and could only perform simple tasks, for example arithmetic operations or comparison of fields of a received packet with some pre-defined constant values, over a 32-bits instruction set. Linux 3.18 deployed an extended version of BPF (called extended Berkeley Packet Filter or eBPF), and despite its name, does not only restrict to packet filtering use-cases. Concerning its networking use, the two major tools loading eBPF programs into the kernel are

`tc` (traffic control) and `XDP` (eXpress Data Path) [20].

This section first presents the eBPF architecture. Alongside its instructions set, eBPF provides a complete environment to design more powerful programs, such as maps acting as efficient key/value stores, helper functions, and program pinnings. We then give an overview of the existing eBPF development toolchain, i.e., projects aiming to easier the development and manipulation of eBPF. This section is highly inspired by the Cilium documentation [21] and ebpf.io [22]. Nowadays, cBPF is no longer used and is replaced by eBPF. That is why we interchangeably use the term BPF and eBPF.

1.2.1 Instruction set

The BPF instruction set is designed to write programs in a subset of the C language, which can be compiled into BPF instructions using an in-kernel back end compiler such as `llvm`, even if `gcc` recently got updated to support eBPF programs compilation [23]. The kernel can then map these instructions into native opcodes using a Just-in-Time (JIT) compiler so that the performance remains similar to native kernel programs.

There are multiple advantages of pushing the instruction set in the kernel, such as making it programmable without requiring to cross the kernel/userspace boundary. A BPF program makes use of the existing kernel infrastructure, for example, the drivers, tunnels, sockets, or tools such as `iproute2`. In opposition to modules, a BPF program is verified by the kernel to ensure that it will not jeopardize its integrity. Let us also note that eBPF is an event-driven execution machine. For example, a program running with `tc` for packet filtering purposes will only be triggered upon reception of a new packet.

BPF is composed of eleven 64-bits registers (named `r0`-`r10`), a program counter, and a stack of only 512 bytes. Register `r10` is a read-only register containing the BPF stack space address, and the other ten registers can be used for general purposes. `r0` holds the return value of BPF helper calls, and `r1` initially carries the pointer address of the context of the program. The context is the input argument when entering the BPF program (similar to `argc`/`argv` for a main function), and is defined by the program type. A networking BPF program receives a kernel representation of the received packet that triggers the program, as an `skb` structure. To ensure that the program does not modify the original packet, the context is a modified version of the `skb` structure containing a copy of the original socket buffer: `__sk_buff` (Appendix A.1).

1.2.2 Program load, verification and JIT

The compiled BPF bytecode can be loaded into the kernel using the `bpf()` system call, but this is generally done using higher-level libraries (see section 1.2.7). Before being attached to the desired hook, the program passes through two steps: the verification and the JIT compilation (Figure 1.7).

The verification ensures that the program will not compromise the kernel safety. The kernel verifier ensures that the program:

- Holds the required privileges;
- Does not perform invalid reads/writes that could crash the system. Each access to a structure or a buffer must check that it lies within the memory boundaries;

- Always terminates within a defined amount of instructions, i.e., does not loop *forever*, otherwise further processing in the kernel could be stopped pending. It is verified by constructing a Directed Acyclic Graph (DAG) of the program execution. As depicted in Figure 1.8, the graph must not be a tree, and different branches can lead to the same final state, but the verifier ensures that the instructions always flow in the same direction until the completion of the program. As a result of this DAG structure, for-loops were prohibited until bounded loops introduction in Linux 5.3 [24]. The maximum number of instructions that the verifier could handle was 4096, and this number raised to 1 million in Linux 5.1. More precisely, this one million limit concerns the number of instructions that the verifier can check before rejecting the program. For example in Figure 1.8, the longest path contains three instructions, but the verifier tests at least every branch once[1]. If a BPF program contains several branches, the verifier must check all of them, and a program with a big complexity will execute much less than 1 million instructions from the start to the end.

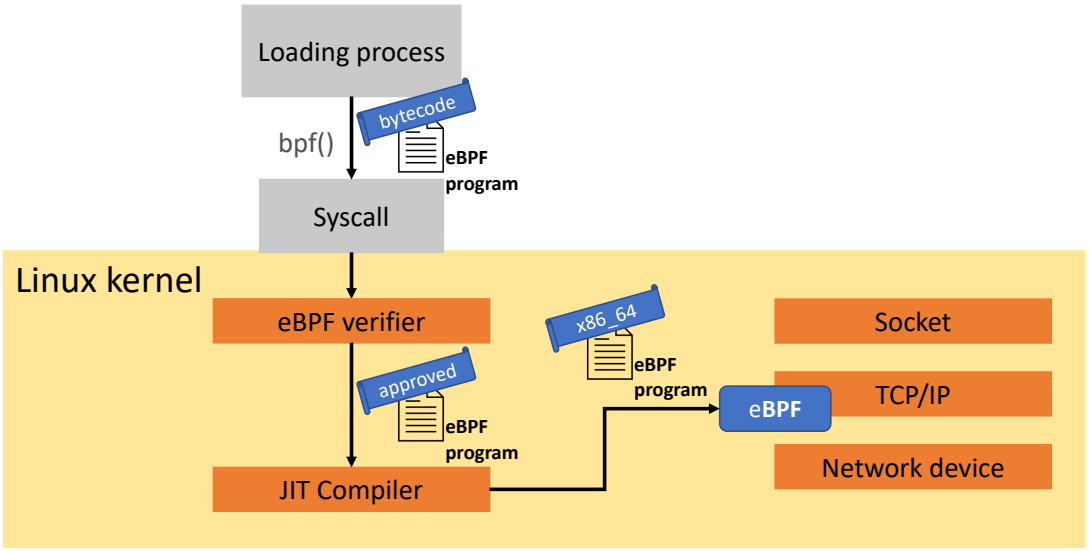


Figure 1.7: The two steps during the loading process are the BPF verification and the JIT compilation.

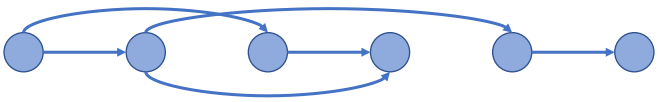


Figure 1.8: The verifier builds a Directed Acyclic Graph from the instructions of the program. Each branch increases the complexity of the program as the verifier must check all of them. The nodes and arrows respectively represent the state of the program and the executed instructions.

The second step is the Just-in-Time (JIT) compilation, to translate the generic bytecode of the program into the machine-specific set of instructions to make the BPF program run as efficiently as natively compiled kernel code. The JIT can be (de-)activated with `sysctl`.

[1]Depending on certain conditions, the verifier can skip an already checked path; this is called *state pruning* [25].

1.2.3 Helper functions

As stated earlier, a BPF program is limited to ensure kernel safety, but BPF helpers provide safe and generic functions to give more flexibility to it. They especially allow the program to retrieve/push data from/to the kernel. These functions are statically defined, meaning that we must extend the kernel to add new helpers. The `bpf-helpers` manpage [26] presents all the implemented helpers but, depending on the used hook, some of them are not always available. For example, the `bpf_skb_load_bytes()` helper provides an easy way to load data from a packet but cannot be used for tracing hooks.

1.2.4 Maps and perf events

Maps are efficient key/value stores residing in the Linux kernel (Figure 1.9). They allow saving persistent data between two calls to the BPF program (recalling that BPF is event-based). Moreover, as the stack size is limited to 512 bytes, maps can be used to store and manipulate heavier structures. Different types of maps exist, ranging from simple hash tables or arrays to more complex variants such as Least Recently Used (LRU), ring buffers, or maps of maps. Userspace applications can also manipulate these maps through the `bpf()` system call and a file descriptor. Maps can also be shared with other BPF programs of different types.

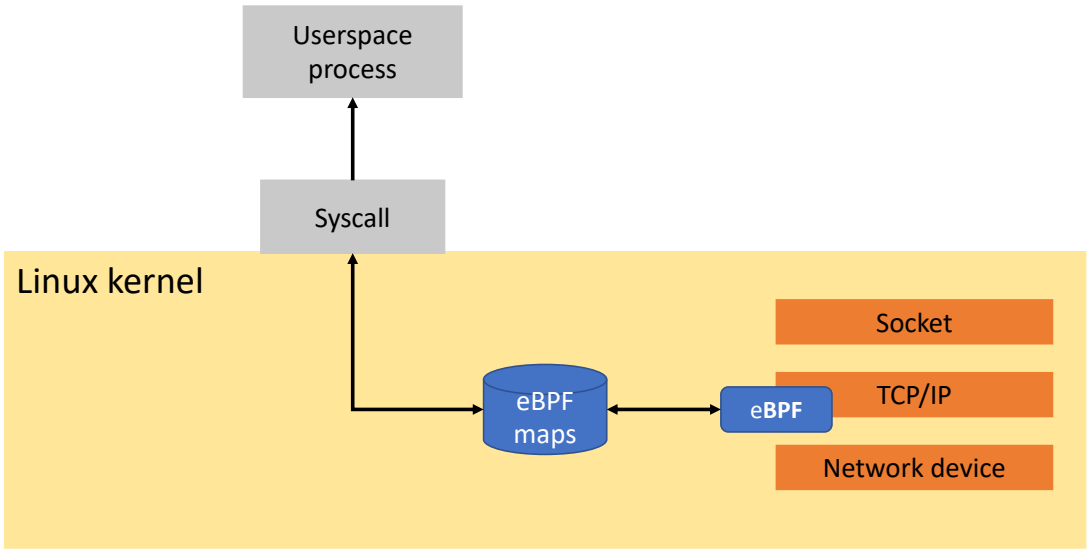


Figure 1.9: Maps are shared between the BPF program in the kernel and the userspace process, using the `bpf()` system call.

Inside a BPF program, we can get and update the content of a map respectively using the two helpers `bpf_map_lookup_elem()` and `bpf_map_update_elem()`. The BPF architecture also provides a way for a program to communicate with a userspace process using `perf events`. Indeed, some plugins may require the userspace to perform a task forbidden in the BPF environment based on intercepted information, or to collect statistics in a userspace controller program. The `bpf_perf_event_output()` helper allows a BPF program to send data to the userspace process using a `perf_event` buffer, a circular buffer map dedicated to this task. However, it is being replaced by faster BPF ring buffers starting from Linux 5.8 [27].

1.2.5 BPF tail calls

We can use another mechanism to simplify the implementation of BPF programs, named tail calls (Figure 1.10). They basically allow a BPF program to call another one (of the same type) without returning to the caller. These calls provide minimal overhead and reuse the same stack frame between jumps. The different programs are verified independently by the kernel, meaning that the communication between them must use maps for example. The jump from a program to another is implemented using a program array populated by the user with the program file descriptors as value and using the `bpf_tail_call()` helper.

As each program is verified independently, we can theoretically break the 1 million instructions limit by calling a program after another. However, the verifier also limits the number of tail calls to 33.

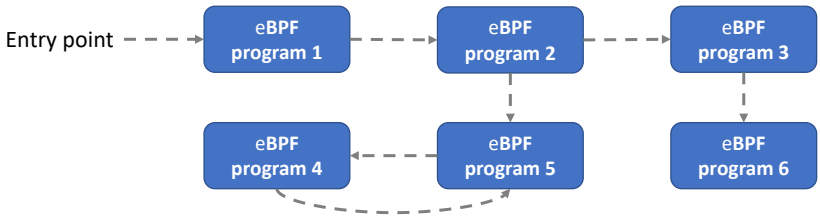


Figure 1.10: Example of BPF program tail calls. Loops between programs are authorized but the maximum number of jumps is 33.

1.2.6 Object pinning

BPF programs and maps can only be accessed using file descriptors, but their lifetime is limited to the execution of the process. It can become an issue when a userspace or third-party application regularly manipulates the content of a map since it can disappear when the BPF program initializing it finishes. To cope with this potential problem, a minimal BPF file system has been implemented, where programs and maps can be pinned to a persistent file descriptor [21]. This process, called object pinning, allows for example `tc` to share a map between an ingress and an egress program.

1.2.7 Development Toolchains

Different open-source projects have been build on top of BPF to easier program development, implementation, and deployment. We present the two toolkits used during this thesis: `bcc` and `libbpf`.

1.2.7.1 BPF Compiler collection (`bcc`)

The BPF Compiler Collection, or `bcc` in short [28] (see Figure A.1) is a framework wrapping BPF over a python interface. The user only needs to write the BPF program in C, and `bcc` handles the `llvm` compilation and the load into the kernel. The python program generates the bytecode from the given C code (either as plain text or from a source file) and triggers the `bpf()` system call. It also provides a userspace interface to communicate with the BPF program with an easy-to-use API. This API implements for example wrappers to manipulate the maps or to handle perf-events in a python userspace program. `bcc` is mainly used for efficient tracing purposes, but we can attach a `bcc`-BPF program in python on networking hooks using `pyroute2`.

1.2.7.2 libbpf

Recently, libbpf [29] (see Figure A.2) gained much interest in the BPF community. It is a generic BPF library developed by the Linux kernel community as part of the kernel source tree. libbpf is written in C and attempts to provide a vanilla API, close to standard BPF code.

Compared to bcc, libbpf reduces storage needs, as the compilation is performed by the user, where bcc needs to embed the entire llvm/clang library. Secondly, the userspace application communicating with the BPF program is written in C, leveraging the speed of this programming language compared to python. As suggested earlier, the library remains close to native BPF programming and this philosophy is quite different from bcc. A simple example is the manipulation of maps, represented in Listing 1.1 and Listing 1.2. bcc wraps maps utilization as semi-objects, where libbpf uses the native helper to access it.

```
BPF_MAP_HASH(my_array, uint32_t, uint16_t, 5);

static inline int my_bpf_function(...) {
    uint32_t k = 0;
    uint16_t *val = my_array.lookup(&k);
    if (!val) return -1;
    ...
    return 0;
}
...
```

Listing 1.1: Map manipulation with bcc.

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 5);
    __type(key, u32);
    __type(value, uint16_t);
} my_array SEC(".maps");

static __always_inline
int my_bpf_function(...) {
    u32 k = 0;
    u16 *val = bpf_map_lookup_elem(&my_array, &k);
    if (!val) return -1;
    ...
    return 0;
}
```

Listing 1.2: Map manipulation with libbpf.

Finally, libbpf supports building BPF CO-RE-enabled applications [30]. CO-RE stands for Compile Once - Run Everywhere. As its name suggests, BPF CO-RE brings together all the necessary pieces from all levels of the software stack to make it possible and easy to write portable BPF programs, handling the differences between kernels. This is for example done by the creation of a vmlinux.h file eliminating the dependencies of kernel headers.

1.3 IPv6 Segment Routing (SRv6)

Traditional IPv4 and IPv6 routers transmit packets on a hop-by-hop basis. Each router keeps in memory its routing table, where the destination address of a packet is matched according to the longest prefix match to determine the next hop. The next hop is chosen so that the packet follows the shortest path (according to some metric) from the source to the destination. This

decision is local and independent to each router forwarding the packet, based on the current information available that it knows.

A consequence of this forwarding mechanism is that at a given node, all packets with the same destination will follow the same path - or set of paths if we consider *Equal Cost Multipath* (ECMP). However, some situations may require that a flow does not necessarily follow this path to reach the destination. Some networks for example base their metrics on bandwidth [31], but real-time applications would favor a path with smaller delays. More flexibility to reach the same destination with different paths also allows for load-balancing [32], or a network operator may decide to reroute non-urgent traffic (e.g., a database backup) to a suboptimal path. Such problems are very complex to solve using hop-by-hop routing, hence the need for other tools.

Multiprotocol Label Switching (MPLS) [33] proposes an alternative mechanism to cope with this situation. With MPLS, a packet entering the network is given a label identifying a virtual path. Each router keeps a Label Switching Table to forward the packet to the desired next hop. It then becomes possible to redirect two flows having the same destination through distinct paths just by assigning a different label to the packets, and fill in the Label Switching Tables of the routers accordingly. Label Distribution Protocol (LDP) [34] and Resource ReSerVation Protocol (RSVP) [35] both implement the MPLS mechanism, but LDP does not offer enough flexibility, and RSVP especially suffers from scalability issues [36]. This led to the introduction of Segment Routing, and later its implementation over IPv6.

1.3.1 Segment Routing

Segment Routing (SR) [37] leverages the Source Routing paradigm. An SR packet contains an ordered list of instructions called segments, often referred to by their Segment Identifiers (SID). A segment can represent any local or global instruction, topological or service-based. In an IGP network, each router (node) and each link (adjacency) has an associated SID. A node Segment Identifier (Node-SID) is uniquely defined in the network and represents the shortest path (defined by the IGP) to reach the advertising node. Each router is assigned a node SID from the network operator. On the other hand, an adjacency segment (Adj-SID) is local to each router and specifies an adjacency, such as an egress interface to a neighboring router. It allows forcing the router to use this interface even if it does not lead to the shortest path to the destination. The Segment List carries these segments in order. The packet also contains a pointer indicating the active SID, and the packet is forwarded to the node represented by this segment. Once this segment destination is reached, the cursor moves to the next segment, and the destination of the packet is updated to reach the next SID of the Segment List.

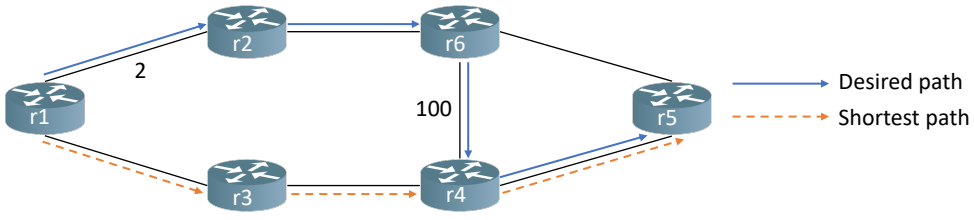


Figure 1.11: All links have a unit cost, except `r1-r2` and `r4-r6` respectively with a cost of 2 and 100. We focus on reaching `r5` from `r1`.

It then becomes possible to define arbitrary paths to reach a destination. In the topology presented in Figure 1.11, `r1` wants to reach `r5` using the path `r1,r2,r6,r4,r5` (plain arrows),

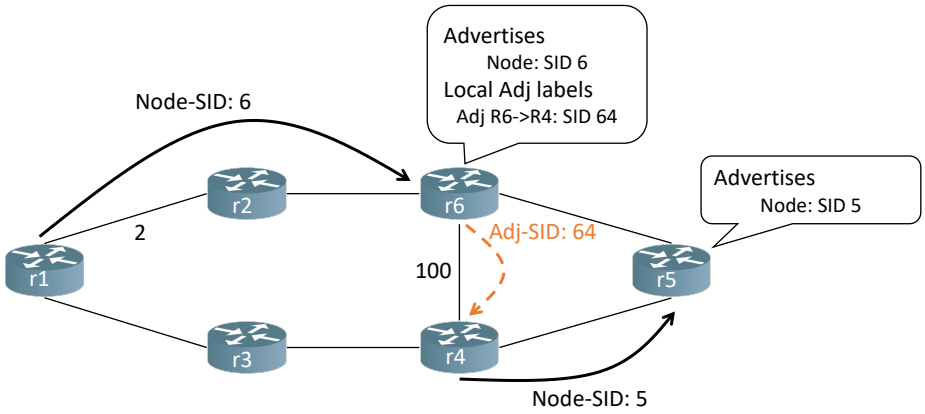


Figure 1.12: To achieve the desired path from `r1` to `r5` from Figure 1.11, we add the Segment List containing the three SIDs `<6, 64, 5>`.

but the shortest path towards this node is `r1,r3,r4,r5` (dashed arrows). Figure 1.12 shows that using Segment Routing we can add to the packet a Segment List containing the Node-SID of `r6` followed by an adjacency segment to forward the packet on the `r6-r4` link, and another Node-SID to reach the final destination.

1.3.2 IPv6 Segment Routing

We can apply Segment Routing in the IPv6 architecture by adding a Segment Routing Header in the packet. We only focus on the Linux implementation of IPv6 Segment Routing [38].

1.3.2.1 Segment Routing Header

RFC8754 [2] defines the Segment Routing Header, represented in Figure B.1. An SRv6 Segment Identifier (SID) is represented by an IPv6 address (128 bits), and stored in the *Segment List* in reverse order. The *Destination Address* of the IPv6 Header is set to the currently active segment, which is indicated by the *Segment Left* pointer of the SRH, and the *Last Entry* indicates the total number of segments in the *Segment List*. The Segment Routing Header instantiates the Source Routing IPv6 extended header, with the *Routing Type* set to 4 to indicate that we use IPv6 Segment Routing. The *Tag* marks a packet as belonging to a class or group of packets, and the *Flags* allow custom behavior of the packet. The *Next Header* field announces the type of the header following the SRH, and finally the *Hdr Ext Len* indicates the length of the extended header in 8-octet units, not including the first 8 octets [39].

The IPv6 architecture [39] also defines Type-Length-Value (TLV) options. These fields allow adding variable-length options to an extended header, such as the Segment Routing Header. As depicted in Figure 1.13, a TLV contains three fields: a *Type* identifier, the *Length* in bytes of the option (excluding the two first bytes of the TLV), and finally a variable-length *Value*. Since the *Hdr Ext Len* field indicates the length in 8-octet units, RFC8200 also defines two padding TLVs in case the extended header length is not a multiple of 8 bytes because of the presence of TLVs.

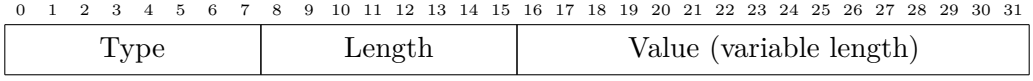


Figure 1.13: Generic representation of a TLV.

1.3.2.2 Network programming with IPv6 Segment Routing

RFC8986 [3] defines SRv6 over IPv6 Network Programming. This framework enables a network operator to define a sequence of instructions in the Segment Identifiers of the SRv6 packet. In Network Programming, we define two types of routers. A node receiving an SRv6 packet with the *Destination Address* set to an SID belonging to the node is called an *Endpoint Node*; a node forwarding a packet whose *Destination Address* is an SID not belonging to the node is called a *Transit Node*.

Leveraging the 16-bytes long addresses, routers can advertise shorter prefixes locating them in the network (the *Locator*) and use the remaining space for Network Programming purposes. When an SRv6 SID is the *Destination Address* of the IPv6 Header, it is routed in the IPv6 network as a standard IPv6 address. Using the prefix advertised by the node, the packet reaches the destination, and the remaining bits of the address are used by the router for Network Programming. More formally, RFC8986 [3] defines an SRv6 SID using the following format: `LOC:FUNCT:ARG`. The *Locator* (`LOC`) is stored on the L most significant bits of the address (this is also the prefix advertised by the router to reach it); the *Function* (`FUNCT`) defines which action must be performed by the router and is stored on the F following bits; finally, the *Argument* (`ARG`) is used to store additional arguments about the endpoint function, and is stored on A bits. It is mandatory that $L + F + A \leq 128$ to fit in an IPv6 address, and the possible remaining low-order bits are set to zero.

Section 4 of RFC8986 [3] defines 15 endpoint functions leveraging SRv6 Network Programming, ranging from the simple End behavior, to more complex functions such as End.DT4. Two other functions are defined for transit nodes: we can extend their routing tables to trigger transit functions for a given prefix, even if the packet does not contain an SRH, to insert (respectively encapsulate) a new Segment Routing Header in the packet with T.Insert (resp. T.Encaps).

In the implementation of IPv6 Segment Routing in the Linux kernel [38], the `seg6` lightweight tunnel has been designed to support the encapsulation and insertion of SRH as transit behaviors with T.Insert and T.Encaps. The `iproute2` tool provides a way to add SRH onto packets [40] using `seg6`. The `mode` is either `encap` or `inline` depending on the desired behavior:

```
ip -6 route add <prefix> encap seg6 mode <mode> segs <segments>
                                                dev <device>.
```

It is also possible to add an SHR (with TLVs) to all packets sent via a specified socket using the `setsockopt()` system call. The second lightweight tunnel of the Linux implementation is `seg6local` and has been implemented to support the Network Programming functions defined by RFC8986 [3]. It allows installing SIDs mapped to specific functions that are executed by the router. There were initially 9 functions that we could map from segments [40] using `iproute2`:

```
ip -6 route add <segment> encap seg6local action <action> <params>
                                        dev <device> table localsid.
```

1.3.3 End.BPF action in `seg6local`

Even if these functions let us construct several use-cases [41–43], they are statically defined and not flexible enough for a network operator to define custom behaviors depending on the content of a packet received by a router. Adding new functions inside the kernel for each new expected behavior is not scalable as the potential use-cases evolve continuously.

1.3.3.1 Implementation of End.BPF

With Linux 4.18, a new approach has been implemented to let network operators define their own SRv6 behaviors from a set of generic functions in the kernel [4]. Leveraging the eBPF support in Linux, the End.BPF action has been added to the `seg6local` lightweight tunnel. The new action acts as an endpoint, meaning that it only accepts SRv6 packets with the current segment corresponding to the SID triggering the eBPF action announced by the router that processes the packet. The endpoint updates the state of the SRH, i.e., advances the *Segment Left* cursor, updates the *Destination Address* of the IPv6 Header, and finally starts the eBPF program associated with the segment. Comparably to other hooks manipulating socket buffers, an End.BPF eBPF program receives as input a `__sk_buff` structure, containing the packet and associated metadata.

This endpoint was designed with two ideas in mind:(i) we must be able to implement the other SRv6 actions inside an eBPF program and (ii) the eBPF program cannot jeopardize the integrity of the Linux kernel running it. The interface hence gives restricted access to the manipulated packet but provides multiple helpers to safely read and write specific fields of the SRH:

- `bpf_lwt_seg6_store_bytes`: provide indirect write to fields of the SRH (`Tag`, `Flags`, TLVs);
- `bpf_lwt_seg6_adjust_srh`: grow/shrink the SRH to add/remove TLVs;
- `bpf_lwt_seg6_action`: execute an SRv6 action;
- `bpf_lwt_push_encap`: encapsulate the current packet with an IPv6 and Segment Routing Header.

The End.BPF action also comes along with three return values for a program:

- `BPF_OK`: the packet must trigger a FIB lookup on the next segment, and is forwarded on the egress interface of the next hop returned by the lookup;
- `BPF_DROP`: the packet must be dropped;
- `BPF_REDIRECT`: the standard FIB lookup must not be executed, and the packet must be forwarded using the destination address already set in the metadata.

Once the eBPF program terminates with a return value different than `BPF_DROP`, End.BPF performs additional checks to ensure that the integrity of the packet is maintained after the program. For example, it guarantees that $\textit{Hdr Ext Len} = (N_{\text{segments}} * 16 + \sum_{i=0}^{N_{\text{TLV}}}[i].length)/8$, and that the sum of lengths of all the TLVs is a multiple of 8 bytes. If the verification fails, the packet is dropped; otherwise, it is forwarded back to the IPv6 layer of the kernel to continue its processing.

1.3.3.2 Use-cases

Leveraging eBPF with IPv6 Segment Routing, it then becomes possible to define in-network functions on a per-packet basis. Xhonneux [4] proposes three real-life use-cases. The first presents a plugin for passive monitoring of network delay on a path defined by two routers. The ingress node sends packets with an SRH and runs a UDP server to collect information from the egress node. The SRH contains two TLVs to (i) carry a timestamp value and (ii) indicate the UDP port for the egress router to transmit the information back to the controller collecting the measured delay.

The plugin also indicates the exact path the packet must follow using intermediate node segments. The egress router computes the one-way measured delay (OWD) and forwards a new packet to the UDP controller with the result. This last step is performed in userspace as it is impossible to create a new packet in an eBPF program (Linux 4.18). The second use-case [4] implements an enhanced version of `traceroute` [44] to collect information on distant routers about the Equal Cost Multipath next-hops for a given destination. The last use-case focused on hybrid access networks.

In a second publication, Xhonneux [45] also presents a failure detection and fast-rerouting plugin with End.BPF and IPv6 Segment Routing and shows that the implementation provides good performance compared to other existing techniques for the same task.

Toy example. The mentioned use-cases each require between 50 and 150 source lines of code depending on the complexity of the plugin. To remain concise yet consistent, we present a simple example of an eBPF program triggered by the End.BPF action. Listing 1.3 shows how we can implement a modified version of the End.B6 function using End.BPF (adapted from the public repository [46]). The End.B6 endpoint inserts a new SRH on top of an existing one and forwards the packet accordingly to the active segment of the new SRH. This modified version dynamically changes the segments according to a random value given by another eBPF helper `bpf_get_prandom_u32()`.

```
__section("end_b6")
int do_end_b6(struct __sk_buff *skb) {
  char srh_buf[40]; // room for two segments
  struct ip6_srh_t *srh = (struct ip6_srh_t *)srh_buf;
  srh->nexthdr = 0;
  srh->hdrlen = 4;
  srh->type = 4;
  srh->segments_left = 1;
  srh->first_segment = 1;
  srh->flags = 0;
  srh->tag = 0;

  struct ip6_addr_t *seg0 = (struct ip6_addr_t *)((char*) srh + sizeof(*srh));
  struct ip6_addr_t *seg1 = (struct ip6_addr_t *)((char*) seg0 + sizeof(*seg1))
    ;

  // Randomly updates the intermediate segments
  u32 random = bpf_get_ranom_u32();

  unsigned long long hi = 0xfc00000000000000;
  unsigned long long lo = 0x1;
  if (random % 2 == 0) //
      lo = 0x8; //
  seg0->lo = htonll(lo);
  seg0->hi = htonll(hi);

  seg1->hi = seg0->hi;
  lo = 0x2;
  if (random % 3 == 0) //
      lo = 0xf; //
  seg1->lo = htonll(lo);

  int ret = lwt_seg6_action(skb, SEG6_LOCAL_ACTION_END_B6, (void *)srh, sizeof(
    srh_buf));
  if (ret != 0) {
    return BPF_DROP;
  }
```

```
36   return BPF_REDIRECT;
37 }
```

Listing 1.3: Modified End.B6 SRv6 action with End.BPF.

The program first creates a new Segment Routing Header with room for two segments. The new SRH will add two segments: first `fc00::2` or `fc00::f`, then `fc00::1` or `fc00::8`, depending on the random value. Line 32 of Listing 1.3 shows the use of the `lwt_seg6_action` helper to trigger the End.B6 action to encapsulate the SRH buffer. If the action fails, the packet is discarded. Otherwise, the packet must be redirected as we encapsulated it with a new SRH, hence a new intermediate destination.

This simple example shows how it becomes possible to implement standard SRv6 functions and even dynamically modify their behaviors thanks to eBPF. Once more, we can install a new route using `iproute2` to trigger the End.BPF action when an SRv6 packet contains the corresponding SID. The following command installs a route triggering the compiled eBPF code defined by Listing 1.3 when an SRv6 packet with its active segment set to `fc00::9` is processed by the router. The `sec` corresponds to the section of the eBPF program.

```
ip -6 route add fc00::9 encap seg6local action End.BPF obj bpf_end_b6.o
                                                    sec end_b6 dev eth0.
```

Chapter 2

Forward Erasure Correction with IPv6 Segment Routing

This section presents our design of Forward Erasure Correction at the network layer, leveraging IPv6 Segment Routing. We detail the representation of source and repair symbols, as well as the FEC Framework and the information needed for the encoder and the decoder to communicate. Finally, we detail our FEC controller, a mechanism to send feedback to the encoder to dynamically enable/disable FEC when we detect packet losses.

To our knowledge, this thesis is the first work attempting to implement FEC at the network layer using IPv6 Segment Routing. As we will discuss in the following chapters, bringing FEC at the network layer allows low-capacity hosts, such as IoT devices, to benefit from FEC without implementing it themselves. Using FEC on routers enables the creation of an automatic and protected tunnel between the encoder and the decoder routers to protect packets from losses. We also present the challenges we encountered to design the FEC Framework in this context, which is different from existing implementations at the transport [47–49] and datalink layer [50, 51].

Throughout this chapter, we use the term *encoder* to refer to the plugin part that generates and sends the repair symbols. The term *decoder* corresponds to the router that receives all these symbols and tries to recover the lost packets. As presented earlier, `seg6local` allows to add SRv6 SIDs mapped to functions, especially End.BPF. In particular, we can attach SIDs to this action to start our encoder and decoder eBPF programs each time a packet is processed. The packets to protect only need to contain a Segment Routing Header with the SIDs of the encoder followed by the decoder.

If a packet contains the encoder's SID but not the decoder, it will be considered as a source symbol but the decoder eBPF program will never be triggered. It may then consider that the packet is lost during the transmission, and attempt to recover it, creating a duplicate packet. The opposite situation is less problematic as the decoder directly sees that the received packet was not mapped to a source symbol by the encoder.

2.1 Representation of source symbols

A source symbol is the data unit that is protected with Forward Erasure Correction. Using FEC with IPv6 does not offer several possibilities to define a source symbol. Indeed, as will be discussed in section 2.6, protecting the payload of the upper layer is not sufficient: we also

need to protect the routing information to transmit a recovered symbol to the correct destination.

The FECFRAME [13] states that all source symbols must have the same length. It is required to correctly decode the received symbols. However, the protected packets do not all have the same size, and we must add padding to them. We define a maximum packet size and add padding at the end of each packet shorter than this value. Packets longer than this size cannot be protected and are forwarded without relying on FEC in case of loss.

When we recover a lost packet, we end up with a pre-defined size buffer, and we need to remove the padding added during the encoding before sending the data to its destination. Therefore, we must also protect the original packet length independently of the packet itself. As we preserve the packet starting from the IPv6 Header, one could wonder why we do not simply rely on the *Payload Length* field to recover the original packet size. While this approach would slightly reduce the plugin overhead, we decided to stick with the FECFRAME requirements, as well as RFC8681 [52].

2.2 Forwarding the repair symbols

Once the encoder has received enough source symbols to generate the repair symbols, we must define a way to send them to the decoder. This router must distinguish the repair symbols from the source packets and not forward them afterward since they only carry redundant and coded information. We thought of two different strategies to carry the repair symbols.

2.2.1 Embed the repair symbols in source symbols

The first idea is to embed the repair symbols inside source symbols. We can leverage the architecture of the Segment Routing Header and create a dedicated TLV that contains the repair symbols. We do not need to create a completely new packet to transmit the FEC payload, but the packet embedding the source symbol becomes larger. Moreover, a TLV value cannot be longer than 255 bytes, meaning that we must cope with a relatively small repair symbol payload, or we must implement a TLV-fragmenting mechanism to carry multiple TLVs with a fragment of repair symbol each.

2.2.2 Repair Symbols in dedicated packets

The other strategy consists of creating a new packet that only carries a repair symbol. The repair symbol size is only constrained by the IPv6 Header limitation (the `Payload Length` constraints to 65536 bytes). On the other hand, the encoding router must create and forward new packets efficiently to the decoder. This repair symbol is carried as a payload inside a new IPv6 packet, which must contain a Segment Routing Header with a SID triggering the decoder BPF endpoint.

Our initial prototype relied on the first strategy, as it was much simpler to implement. Indeed, we did not need to craft new packets to forward the repair symbols. Due to the 255 bytes constraint of the TLVs carrying the repair payload, this prototype could only protect tiny packets, such as TCP acknowledgments. However, in order to widen the scope of our plugin, our final solution uses the second approach. We can theoretically protect longer packets as the only constraint is the IPv6 *Payload Length* maximal value.

2.3 TLV representation of protected packets

We carry FEC information inside a TLV field of the Segment Routing Header, taking advantage of the modularity of its architecture. The source symbol only needs to carry the *Source FEC Payload ID* (SFPID), a 32-bits field identifying a source symbol for the FEC Framework. Figure 2.1 defines our FEC source symbol TLV that carries this value.

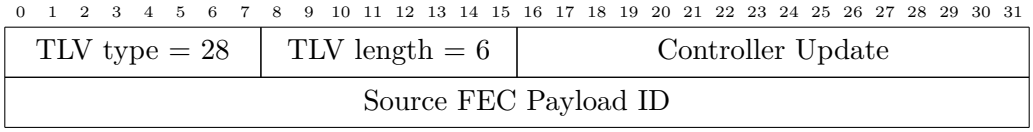


Figure 2.1: TLV for the source symbols.

The TLV is added by the encoder router and allows the decoder to detect that the packet is protected by FEC. RFC8280 [39] states that the TLV Option Type must follow the following standard:

- *The highest-order 2 bits specify the action that must be taken if the processing IPv6 node does not recognize the option type.* In our case, a processing node that does not recognize our FEC-specific TLV should skip it but not drop the packet (00).
- *The third-highest-order bit of the Option Type specifies whether or not the Option Data of that option can change en route to the packet's final destination.* This situation is specific since the decoder router removes the TLV once the source symbol is processed. Hence the packet's final destination will not receive the TLV. We decide to set this bit to 0 (the Option Data does not change).

We use as TLV Type the value 28 (`0b00011100`), in concordance with the specifications of RFC8280.

The repair symbols carry FEC Framework and FEC Scheme-specific information for the decoder router. More formally, we define it as the FEC Frame. Each repair symbol includes a FEC Frame as an SRv6 TLV, illustrated by Figure 2.2.

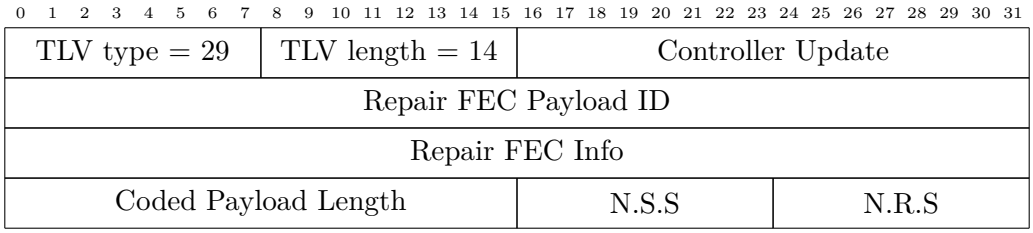


Figure 2.2: TLV for the repair symbols.

The 32-bits *Repair FEC Payload ID* field contains information to associate the repair symbol with the source symbols it protects. The *Repair FEC Info* value holds FEC Scheme-specific information, for example, the seed to generate the pseudo-random coefficients for RLC. The *Coded Payload Length* value will be recovered identically to a lost source symbol to recover its original length, as discussed previously. The *NSS* (Number of Source Symbols) indicates the number of source symbols used to generate this repair symbol. Accordingly, the *NRS* (Number of Repair Symbols) gives the number of repair symbols generated from the same set of source symbols.

The *Controller Update* field of both the source and repair TLVs indicates to the controller (section 2.5) the period between two statistics update messages. This quantity can be expressed in time or number of source symbols received, but our prototype currently uses the second approach. If the controller is disabled, its value is always set to 0. Even if we do not use this field, it ensures that both TLVs are a multiple of 8-bytes long. We hence do not need additional padding TLVs to ensure that the total SRH length is a multiple of eight bytes.

2.4 The FEC Framework

RFC6363 [13] and RFC8680 [14] define a FEC Framework at the transport layer, and we extend them for our implementation for SRv6.

The FEC Framework attempts to present a shared procedure upon which we can attach different FEC Schemes. It simplifies the FEC Schemes by providing the standard behavior of these coding algorithms, for example, adding the *Source FEC Payload ID* in the source symbols.

Figure 2.3 illustrates our design of the FEC Framework, inspired by RFC6363 [13]. The network layer transmits to the FEC Framework the packet to protect (1). In contrast to RFC6363, the FEC Framework stores a copy of the packet inside a buffer, generates the *Source FEC Payload ID* of the source symbols and adds a TLV with this value in the Segment Routing Header (2) of the original packet. This TLV is not added to the copy stored in the Framework buffer. The Framework then calls the FEC Scheme to potentially generate repair symbols (3). When it receives enough source symbols, the FEC Scheme creates repair symbols based on its specific coding function and crafts the repair TLVs (4). It then forwards the repair payloads alongside the TLVs back to the FEC Framework (5). The FEC Framework creates new repair symbol packets based on this information (7) and gives them to the network layer for transmission (8). It finally sends the source symbol back to the network layer. If the FEC Scheme does not generate any repair symbol, we can skip the steps (4, 5, 6, 7).

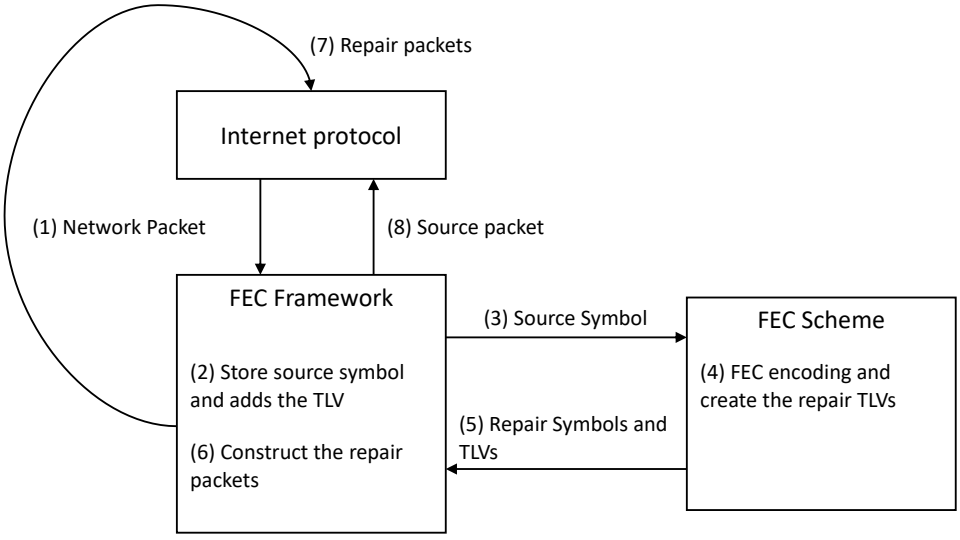


Figure 2.3: Interaction of the FEC Framework with the FEC Scheme and the network layer. We use the term *Network packet*, but RFC6363 and RFC8680 use the term *Application Data Unit*.

The rest of this section presents the Block and Convolutional FEC Frameworks for the encoder and the decoder, based on RFC6363 [13] and RFC8680 [14].

2.4.1 The Block FEC Framework

2.4.1.1 The sender Block FEC Framework

The only parameter of the Block Framework is the *Block Size*, the number of source symbols in a block. The received source symbols are stored in the Block buffer in their arrival order. Once complete, the block is passed to the FEC Scheme to generate the repair symbols for this block. The Block buffer is emptied and the next source symbols will belong to the next block.

We define the *Source FEC Payload ID* as a 16-bits *Source Block Number* and a 16-bits *Source Packet Number*, as shown in Figure 2.4.

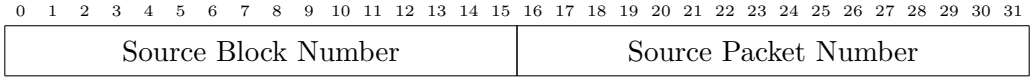


Figure 2.4: Content of the *Source FEC Payload ID* for the Block FEC Framework.

The *Source Block Number* identifies the block containing this source symbol. The *Source Packet Number* identifies the position of the packet inside this block. The block number is mandatory if we want to handle possible out-of-order arrival between the encoder and the decoder. The *Source Packet Number* is not needed by the XOR FEC Scheme, but mandatory if one attempts to add the Reed-Solomon encoding [12].

The *Repair FEC Payload ID* is also composed of the 16-bits *Source Block Number* identifying the block it protects, and the other 16-bits *Repair Symbol Number* indicating the offset of the repair symbol in the block.

2.4.1.2 The receiver Block FEC Framework

The receiver Block FEC Framework is provided with the received source and repair symbols. The decoder may receive symbols from different blocks in an incorrect order, and must hence keep room for multiple blocks simultaneously. The decoder removes the Segment Routing Header TLV from the source symbols containing the *Source FEC Payload ID* and then makes an internal copy of the packet for future decoding. Once the receiver detects that one or more source symbols from the same block can be recovered, it forwards the source and repair symbols of the block to the FEC Scheme. The recovered packets are then pushed to the network stack and forwarded *as if* there was no loss.

2.4.2 The Convolutional FEC Framework

2.4.2.1 The sender Convolutional FEC Framework

The sender Convolutional FEC Framework is similar to the sender Block FEC Framework. The Convolutional Framework is characterized by two parameters: the *Window Size* S and *Window Step* k, respectively the number of symbols in a window and the shifting value of the sliding window once the current window is full and has been processed. The source symbols are stored in a Convolutional window in their arrival order. Once the current window is complete, the encoder forwards it to the FEC Scheme for encoding and shifts the window by k source symbols. The sender Framework then waits for k source symbols before the generation of the next repair symbol, as it keeps $\max(0, S-k)$ source symbols from the previous window.

The *Source FEC Payload ID* for the Convolutional FEC Framework is simply a 32-bits field representing the ID of the source symbol in the sliding window (Figure 2.5). This value is incremented by one for each received source symbol.

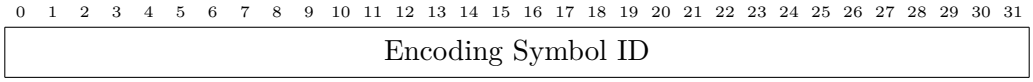


Figure 2.5: Content of the *Source FEC Payload ID* for the Convolutional FEC Framework.

The *Repair FEC Payload ID* represents the *Encoding Symbol ID* of the *last* source symbol of the current window that the repair symbol protects.

2.4.2.2 The receiver Convolutional FEC Framework

The decoder receives the source symbols - possibly in an incorrect order - and places a copy in its Convolutional window using the *Encoding Symbol ID*. The copy is made on the source symbol where the FEC TLV is removed. It also receives the repair symbols and stores them accordingly to the window they protect. Once we can recover one or more lost source symbols, the FEC Framework provides the FEC Scheme with the available source and repair symbols. The Framework can finally forward the recovered symbols *as if* there was no loss.

2.5 A controller to monitor the FEC plugin

A drawback of Forward Erasure Correction is that it consumes additional bandwidth by adding redundancy [53, 54]. Indeed, there are two sources of link utilization overhead:(i) the TLV-encoded *Source FEC Payload ID* adds eight bytes to the existing Segment Routing Header to each protected packet and (ii) each repair symbol carrying the redundancy.

This redundancy is useful if we experience losses in the network. However, we cannot predict in advance when packets will be dropped or when the network is perfect. If the decoder receives all source symbols, it will not use the repair symbols and these will unnecessarily consume bandwidth. This section discusses a causal control mechanism to reduce the plugin overhead when we detect that redundancy is not needed, using feedback from the network.

2.5.1 Related work

Cohen *et al.* propose the AC-RNLC function, a modified RLC adapting its coding rate using the feedback from the network [55, 56]. Due to the eBPF restrictions (discussed in section 3.4), we did not implement AC-RNLC in the plugin yet, but this work is a building block of our controller.

The causal controller must rely on feedback from the network to adapt the generation of repair symbols based on the observed losses. *The In-situ Operations, Administration, and Maintenance (IOAM) records operational and telemetry information in a packet while it is being processed by intermediate routers of the network* [57]. This work is still an IETF draft, but an extension of this thesis might rely on this protocol to get feedback from the entire path between the encoder and decoder routers, and not just from the decoder part of the plugin.

2.5.2 Design of our controller

We design a simple causal controller prototype that controls the generation of repair symbols. We do not provide an implementation of RLC to dynamically update the code rate of the FEC

Scheme as done by AC-RLNC, but make a binary decision regarding the generation of repair symbols.

The encoder makes the binary decision based on the feedback it gets from the decoder. If it considers that Forward Erasure Correction could improve the network reliability, it triggers the generation of repair symbols. The decision is taken by the encoder and opaque to the decoder.

The decoder regularly sends its measured statistics to the encoder. The period between two feedback messages is also defined by the encoder, using the *Controller Update* field of the source and repair TLVs. The decoder part of the controller only collects information about the losses in the network it observes. It internally keeps two counters:

- *Received counter*: the number of source symbols received since the last feedback;
- *Theoretical counter*: the theoretical number of source symbols it should have received since the last feedback.

The difference between these two values is that the *Theoretical counter* compares the *Encoding Symbol ID* of the last received symbol with the *Encoding Symbol ID* of the last packet from the previous update, where the *Received counter* is incremented for each effectively received source symbol since the last update. The decoder transmits the measured loss ratio based on these two values.

The encoder receives the loss percentage experienced by the decoder and can decide to (continue to) generate repair symbols. Conversely, it could stop the redundancy to lighten the bandwidth if the experienced loss does not require FEC protection. The current decision function is a threshold condition: if the measured loss is above a predefined value, the encoder starts/continues to generate repair symbols; otherwise, it stops crafting them. Even if we stop generating repair symbols, the encoder continues to map the received packets to source symbols (i.e., adds the TLV-encoded *Source FEC Payload ID*) as the *Encoding Symbol ID* value is used by the decoder to measure the experienced loss.

2.5.3 Communication of the statistics

The controller is a part of the FEC plugin, as it needs to manipulate the same information as the FEC implementation. The decoder updates its statistics each time the FEC BPF endpoint is called upon reception of a new source symbol. The receiver executes this routine once the source symbol is correctly handled by the receiving FEC Framework. The statistics that it forwards back to the sender must trigger the controller section of the encoder plugin, similarly to the previous discussion about the repair symbols (section 2.2.2), i.e., the message must contain an SRH to call the BPF endpoint starting the encoder eBPF program.

We leverage the IPv6 Segment Routing architecture once again and design a TLV carrying the statistics to the encoder, represented in Figure 2.6.

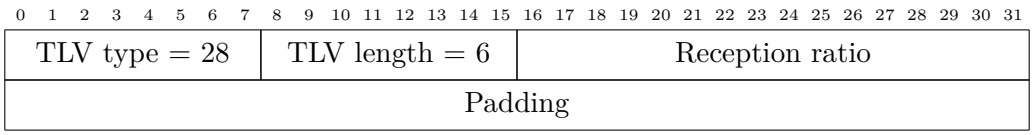


Figure 2.6: TLV for the statistics messages of the controller update.

We reuse the same TLV type as the source symbol TLV to avoid consuming too many values. The TLV currently carries the *Reception ratio*, the ratio between the *Received counter* (r) and *Theoretical counter* (t) as a 16-bits field. As eBPF currently does not support floating-point numbers, the *Reception ratio* r is the percentage of the integer division of the two values:

$$r = \left\lfloor \frac{r * 100}{t} \right\rfloor$$

Finally, we pad the TLV to ensure that it is eight bytes long. Moreover, this free space allows for further extension of the controller statistics.

2.6 The challenge of FEC for IPv6

When looking at the literature, the majority of the use cases of Forward Erasure Correction are in the application/transport layer [47, 49] or the datalink layer [51]. The protected data is the payload above the protecting header, and we must not protect any information from the current header except the payload length.

However, with IPv6 we cannot simply protect the following layers as we must also protect routing information to forward a recovered packet to its destination. Moreover, we designed our solution to protect several flows simultaneously, with potentially different sources and destinations.

Finally, the plugin does not protect only simple links, but also entire paths in a network: considering multiple IoT devices communicating with their server through a backbone network, the encoder plugin protects the IoT flows at the ingress of the backbone; the decoder then recovers lost packets at the egress of the backbone, right before reaching the server. In such networks, the encoder does not know which intermediate hops the packets will follow. RFC8200 [39] states that some fields of the IPv6 Header, and for example the Hop-by-Hop extended header, can be modified by intermediate routers even if they are not the (intermediate) destination of the packet. It implies that the packets will not necessarily arrive at the decoder in the same state as at the encoder, which will cause FEC to fail.

Figure 2.7 illustrates this in a simple scenario. Let us consider deploying our plugin to protect a single link. We receive two packets to protect using FEC. To simplify the example, we only show the 1-byte *Hop Limit* field of the IPv6 packets. Let us remember that a router first processes the IPv6 standard Header before the extended headers (1), which especially decrements the *Hop Limit* before processing the Segment Routing Header. The presence of the encoder SID triggers our FEC plugin (2), which generates a repair symbol using the simple XOR FEC Scheme. The two source symbols are forwarded with the repair symbol to the next hop, but the first source symbol (containing 01100100) is erased during the transmission (4).

When the decoder receives the remaining source symbol, it again decrements the *Hop Limit* before processing the Segment Routing Header (5). Since the repair symbol is carried inside a packet, the IPv6 processing does not modify the repair payload. The FEC plugin receives a source and a repair symbol and then theoretically can recover the lost data by decoding the received information (6). However, the received source symbol does not carry the same value as during the encoding (3). We will hence recover a corrupted packet (7). In this case, recovering a wrong *Hop Limit* is not problematic. However, as we will see in section 2.6.1, there exist other varying fields that could jeopardize the delivery of the recovered packet if not handled correctly.

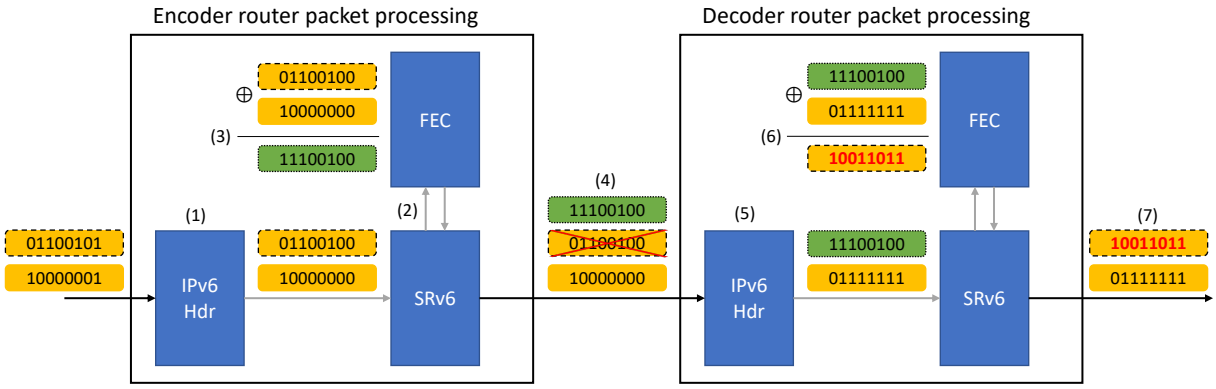


Figure 2.7: Example of wrong decoding due to a varying field in the packet.

This simple example proves that we must carefully handle varying fields of the IPv6 (extended) headers to correctly recover lost symbols. We used here the XOR operation, but this failure is not specific to this coding function. It then becomes mandatory to consider fields of the packets we attempt to protect that can vary during the transit. This section presents these varying fields and how we managed to recover them correctly with Forward Erasure Correction.

2.6.1 Varying fields in transit IPv6

The IPv6 Header and its extended headers contain several varying fields.

2.6.1.1 IPV6 Header

Excluding the *Version*, all fields of the IPv6 Header (see Figure B.2) can vary during the transfer of packet, according to RFC8200 [39].

Hop Limit. The *Hop Limit* is decremented by each node that forwards the packet. As stated earlier, we do not know in advance the number of routers between the encoder and the decoder plugin.

Destination Address. With IPv6 Segment Routing, the *Destination Address* changes for each segment of the SRH. It will change at least once between the encoder and the decoder routers because their SIDs are (successive) segments of the list.

Payload Length. The *Payload Length* can also change during the transit, especially because of the:

- Decapsulation of an IPv6 Header;
- Insertion/Encapsulation of a Segment Routing Header;
- Addition or deletion of TLV options in the extended headers.

However, we expect the packet content to remain the same between the encoder and the decoder. The only allowed event that changes the packet size is the addition of the *Source FEC Payload ID* as an SRH TLV option. As presented earlier, the decoder FEC Framework removes it so that the packet recovers its initial state. The *Payload Length* is then the same at both sides of the plugin. We do not expect this value to change for other reasons, as it would mean that the

packet received by the decoder does not contain the same payload as the encoder, hence the FEC Framework would not work as expected.

Next Header. For the same reasons as the *Payload Length*, we do not expect the *Next Header* to change between the encoder and the decoder, as it would mean that we added/removed a new extended header, or that we encapsulated/decapsulated the packet, hence changing its content.

Flow Label. RFC6437 [58] states that once the *Flow Label* is set to a non-zero value, it is expected to be delivered unchanged to the destination node(s). The only reason to modify it is for compelling operational security reasons, as described in section 6.1 of RFC6437. However, there is no way to determine if the *Flow Label* changed during the transfer and we should provide a "best-effort" quality regarding the protection of this field. As a result, we do not expect this quantity to vary between the encoder and the decoder. If it happens, we possibly recover a wrong *Flow Label*, hence offering a "best-effort" service.

Traffic Class. According to RFC8200 [39], the *Traffic Class* might change during the transfer. RFC2474 [59] describes the six highest-order bits of the field for Differentiated Services usage. As we do not have any information to recover this quantity in case of change during transit, we consider that these six bits should not change between the encoder and the decoder. For example, a network operator deploying our plugin should take this remark into account to avoid *Traffic Class* changes between the two parts of the plugin. RFC3168 [60] defines the use of the two last bits for Explicit Congestion Notification. These two bits can change if the network experiences congestion. The previous comment does not apply to these two bits, and we must propose a way to replace them in a recovered packet.

2.6.1.2 Segment Routing Header

We focus on the Segment Routing Header (see Figure B.1) as it carries the SIDs triggering the FEC plugin. RFC8754 [2] states that the *Segment Left* changes between the encoder and the decoder. This value indicates the number of remaining segments to follow after the current one. This quantity is decremented by at least one between the two parts of the plugin, depending on the presence of intermediate segments. The implementation of the End.BPF SRv6 function updates the next segment of the packet before entering the eBPF function [4]. The *Next Header* of the SRH should not change for the same reasons as the *Next Header* field of the IPv6 standard header.

2.6.1.3 Other extended headers

RFC8200 [39] states that an extended header option that may vary during the transmission must be specified by setting the third-highest bit of the *Option Type* field to one. Each extended header may contain several TLV-encoded options that could also vary during transmission [39]. A simple example is the *Hop-by-Hop* extended header. In the previous situation where we considered protecting traffic across a backbone network, it becomes impossible to track every modification of the headers and recover their initial state, which is mandatory for any FEC algorithm to recover lost symbols.

As a result, we decided to focus on the recovery of IPv6 packets carrying only a Segment Routing Header. This header can contain multiple segments not related to the FEC plugin. However, we do not ensure good packet recovering if the packet contains other extended headers

that are changed by the network between the encoder and the decoder, or if the other intermediate (i.e., between the encoder and decoder SIDs) segments add TLVs in the SRH.

2.6.2 Solution to encode the varying fields

Our solution is to replace each varying field by zero during the encoding and the decoding Schemes. These fields can change between the encoder and the decoder but will be set to zero for encoding and decoding, canceling the negative effect of their variations. We only zero these fields on the internal copy of the packet, not the initial packet that is forwarded through the network. Of course, non-varying fields are not set to zero because they do not compromise the FEC recovery. Let us precise that we only zero the fields that we expect to vary. Therefore, we do not consider the *Flow Label*, *Next Header*, *Payload Length* fields from the IPv6 Header or the *Next Header* of the Segment Routing Header. We hence replace by zero the values of the *Destination Header*, the *Hop Limit*, the two last bits of the *Traffic Class* and the *Segment Left*.

After decoding, the recovered packets will only contain zero values in these fields. We need a way to recover the information as it would be at the decoder if we did not lose the packet. We can set the *Hop Limit* to an acceptable value, 51 for example. Algorithm 1 shows how we can recover the *Destination Address* and *Segment Left* at the decoder; we only need to know the decoder SID triggering the BPF endpoint. If an SID from the *Segment List* corresponds to the SID of the decoder plugin, we know that the packet must be forwarded to the next segment of the *Segment List*. The case where the decoder plugin is the last segment of the *Segment List* will never happen because the implementation of End.BPF does not trigger the eBPF plugin if the corresponding SID is the last segment [4].

Algorithm 1: Recovering the *Destination Address* and *Segment Left* fields

```
Result: Recovered packet with the correct Destination Address and Segment Left
set curr_segleft to last segment of the SRH;
for each segment in SRH Segment List do
    if segment equals decoder SID then
        set Destination Address to the next segment;
        set Segments Left to curr_segleft - 1;
    else
        decrement curr_segleft;
    end
end
throw an error as we did not find the decoder SID
```

We finally set the two last bits of the *Traffic Class*, used for Explicit Congestion Notification, to zero. Since this is a recovered packet, one could consider that the loss is due to congestion in the network. However, an erasure, especially in wireless networks, is not always due to congestion. We hence consider that wrongly notify the packet's destination of the presence of congestion can negatively impact the network quality. On the contrary, if the packet experienced congestion and the ECN bit was set, we assume that other packets belonging to the same flow will notify the sender just with an increased delay.

Chapter 3

Implementation

This section details the implementation of our solution. As a reminder, this plugin runs inside the eBPF kernel virtual machine. Due to the limitations imposed by the verifier, we cannot currently fully implement the plugin in an eBPF program and must communicate with a userspace process for complex parts of the implementation. As part of future work, we aim at extending the kernel, especially by adding new generic eBPF helpers to replace the interactions with userspace.

Our implementation heavily relies on the `libbpf` project led by Andrii Nakryiko [29]. As only very recent kernels natively support BPF CO-RE (Compile One - Run Everywhere), we currently do not use this in our implementation; only minor changes in the *includes* of the program activate CO-RE. We mainly relied on version 5.4 of the Linux kernel [61].

We implemented two FEC Schemes: XOR and RLC, both relying on the FEC Frameworks presented in section 2.4.

3.1 The Block XOR FEC Scheme

The XOR FEC Scheme is straightforward to implement. It relies on the $(n, n-1)$ Block FEC Framework described earlier. When we receive the source symbols, we XOR byte-per-byte each source to produce a unique repair symbol for this block. The *Repair Symbol Number* is set to zero as we only generate a single repair symbol per block. For the same reason, the *Number of Repair Symbols* field of the repair TLV is set to one. The *Repair FEC Info* field is also set to zero because the XOR operation does not require any parameter.

To make the XOR computation more efficient, we operate by words of eight bytes instead of byte-per-byte. Moreover, we can leverage the mathematical properties of XOR. Indeed, we have:

$$\underbrace{S_1 \oplus S_2 \oplus ... \oplus S_n}_{\text{When we receive } S_n} = \underbrace{(((\overbrace{(\underbrace{S_1}_{\text{When we receive } S_1}) \oplus S_2}^{\text{When we receive } S_2}) \oplus ...) \oplus S_n)}_{\text{When we receive } S_n}.$$

It means that we do not need to store simultaneously all source symbols of a block to generate the repair symbol. When the encoder FEC Scheme receives a source packet, it directly XORs it with the previous symbols already coded (or copies the source symbol in the repair symbol buffer if this is the first symbol of the block). The decoder performs the identical computation, taking into account the repair symbol. Moreover, the FEC Scheme can encode/decode online

out-of-order packets, because the XOR operation is commutative and associative. This property of coding/decoding the source symbols *online* is desirable as the maximum processing time between two packets remains (almost) identical - except when we must forward information to userspace. Instead of processing $n-2$ packets (among the $n-1$ symbols of the block) quickly and then making a massive computation when they receive the last symbol, the encoder and decoder smooth the processing time over each symbol.

3.1.1 Communication with userspace

With the current version of the Linux kernel (5.4), it is impossible to craft and send new packets inside an eBPF program. To work around this problem, we encode (respectively decode) the received symbols in the eBPF program and forward the repair symbol (resp. recovered data) to userspace. We craft the desired packet and use a C raw socket to send it to the decoder (resp. next destination). The encoder communicates with userspace every $n-1$ packets once the repair symbol of the current block is generated. The decoder only triggers the userspace program if it successfully recovers a lost source symbol. If it detects no loss, or that it encountered too many losses to recover a source symbol, it does not forward any information to the userspace program.

3.1.2 Limitation of XOR

The XOR FEC Scheme is very limited regarding the recovering capabilities. Indeed, it has a $\frac{n-1}{n}$ code rate and can only recover the loss of a single packet in the block. Moreover, it is almost always unable to recover from burst losses. The only burst loss we can fully recover is the loss of the repair symbol of the block i, followed by the first source symbol of the block $i+1$, as depicted in Figure 3.1. In this situation, the repair symbol R_i is not useful since the source symbols it protects correctly arrived. The source symbol S_4 will be recovered upon the reception of R_5. Any other burst loss cannot be recovered using XOR. That is why we also consider Random Linear Codes.

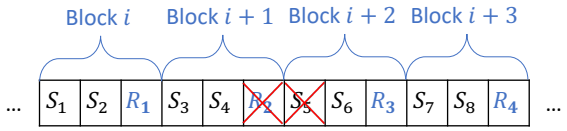


Figure 3.1: Example of a burst loss of length two with a $(3, 2)$ Block XOR code.

3.2 The Convolutional Random Linear Code FEC Scheme

We base our RLC FEC Scheme implementation on RFC8681 [52] and on the Convolutional RLC implementation from PQUIC [62]. It relies on the (n, k, S) Convolutional FEC Framework from section 2.4.2. The FEC Scheme completes the *Repair FEC Info* field from the repair symbol TLV according to Figure 3.2.

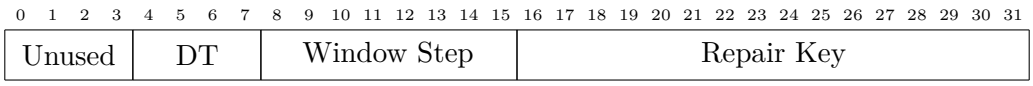


Figure 3.2: Content of *Repair FEC Info* for the Convolutional RLC FEC Scheme.

The *Window Step* reflects the number of source symbols to shift in the Convolutional window between two consecutive windows. The *Repair Key* contains the seed used to generate the

pseudo-random coefficients for the equation generating the current repair symbol. Our RLC FEC Scheme currently supports the generation and processing of only one repair symbol for a given window. This loss of generalization is not dramatic as it can be compensated with a lower *Window Step* value. As a result, the *NRS* field of the repair TLV is set to 1. The *NSS* contains the *Window Size*. RFC8681 [52] states that the *Repair FEC Info* must also contain the *Density Threshold* (DT), indicating the proportion of coefficients (generated with the seed) that are non-zero for the current equation. This value is stored in 4 bits, ranging from 0 to 15. A higher value means a smaller proportion of null coefficients, and hence an increased probability of recovery. In our implementation, we automatically replace every generated coefficient equal to 0 by 1. Our DT is hence always set to 15 as we do not have any null coefficient. The 4 highest-order bits are currently not used.

Section 3.5 of RFC8681 advises using a Pseudo-random Number Generation algorithm for the coefficients of the linear system. This step is mandatory to increase the probability of building a full-rank system that we will solve on the decoding side. We use the `tinymt32` suite from RFC8682 [63] as it defines a standard for Pseudo-random Number Generator, and is advised by RFC8681.

Given our pseudo-random values, we generate the coefficients of the linear equations as defined in section 3.6 of RFC8681, more precisely using the Galois Field $GF(2^8)$. A Galois Field (or Finite Field) is a field containing a finite number of elements. Operations such as division and multiplication will always output a number in this field. The benefit of using a finite field is that we do not deal with rational numbers, which can give inexact behavior due to rounding errors in a computer system. When attempting to recover a lost packet, we must decode exactly the lost bytes, hence the need to define operations giving exact values. In our case, the $GF(2^8)$ Galois Field means that our coefficients range from 0 to 255, but as stated earlier, we replace any null coefficient with a unit value.

To solve the linear system obtained with our coefficients, our received source and repair symbols, we use a standard Gaussian elimination approach. The implementation heavily relies on the Gaussian elimination system from the FEC plugin of PQUIC [62]. When constructing the system, we must determine the maximum number of equations to consider simultaneously. A high value increases the recovering capabilities as we consider more windows to possibly recover longer burst losses, but it also increases the computation time. Let us remember that a Gaussian elimination has a temporal complexity of $\mathcal{O}(n^3)$, with n the number of equations/unknowns. The plugin uses a tradeoff of 5 linear equations maximum in the system.

3.2.1 Communication with userspace

For the RLC FEC Scheme, the communication with user space is heavier than with XOR. We still must change the context to send the repair symbols/recovered packets to the network as it is currently impossible in the eBPF program. However, the encoding and decoding functions are more complex with RLC, and the current eBPF limitations restrict us from executing the coding part inside an eBPF program. When the sender RLC FEC Scheme detects that a repair symbol can be generated, it forwards the Convolutional window to userspace for coding and forwarding. Similarly, when the receiver FEC Scheme can potentially recover a lost packet, it forwards to userspace its entire receiving buffer, as RLC can potentially recover source symbols from multiple windows simultaneously. The userspace finally sends the repair/recovered packets to the network.

3.3 Implementation of the controller

We only implement the controller for the Convolutional FEC Framework with the RLC FEC Scheme. For the Block FEC Framework, the *Controller Update* field of the source and repair TLVs is always set to 0 and is considered as padding.

Due to the eBPF limitations (as will be discussed in section 3.4), we need to communicate with userspace to craft and send the statistics message once the period since the last update message is over. This packet contains an SRH with the encoder controller SID and the controller TLV with the statistics. The encoder controller is part of the same eBPF program as the FEC plugin but with a different section. Hence, the encoder router uses a different SID to trigger the End.BPF action and start the controller section of the eBPF program (leveraging the IPv6 Network Programming [3]). It has access to the same maps and structures, but the packet processing is separated from the FEC plugin. We extend the state of the encoder FEC plugin to contain a new field, indicating whether we must generate repair symbols or not. This field is modified by the controller upon reception of a statistics message according to the decision function, and the FEC encoder consults it each time a new repair symbol *should* be generated.

3.4 Technical limitations with eBPF

We detail here the mentioned limitations of the implementation. Most of them are caused by the eBPF environment that restricts our plugin (Linux 5.4). At the time of writing this thesis, the most recent versions of the Linux kernel do not remove these limitations. Let us also insist that these limitations are technical, but our conceptual approach and design are not their cause.

First, the stack size of an eBPF program running in the kernel VM cannot exceed 512 bytes. Hence we must store heavy data structures in maps, which can slow down the plugin.

The second limitation is the number of instructions the program can perform, limited to one million. This is sufficient for small plugins, but not anymore when we consider encoding packets using RLC, where we perform byte-per-byte computation. Moreover, let us recall that the RLC decoder must solve a linear system. The temporal complexity of our resolution algorithm is $\mathcal{O}(mn^3)$ with n the number of equations/unknowns and m the size of the packet. Considering longer packets, we easily reach the instruction limit of the kernel. We currently bypass this by performing the RLC encoding and decoding in userspace, but at the cost that we must send all the RLC receiving buffer in userspace. This operation is very costly, slows down the performance of the plugin and becomes the bottleneck of the program. This instruction limitation also restricts us to use small RLC window sizes (maximum 15). This limitation also constraints the size of the packets we can protect. As we perform coding/decoding operations byte-per-byte (or word-by-word for XOR), protecting longer buffers means more instructions to verify for the kernel. Therefore, we currently limit the maximum size of the packets we protect to 512 bytes. Let us also note that the current eBPF implementation of the Convolutional FEC Framework could support heavier packets, but the communication with userspace becomes a strong limitation since we must transmit the whole Convolutional receiving window containing the source symbols each time a repair symbol can be generated. We observed that the `perf_event` buffer used for the communication with userspace could not handle all the data correctly, and we noticed that the entire buffer could not be loaded on userspace entirely fast enough. Increasing the instruction limit will allow us to perform the encoding/decoding process entirely in the eBPF program, and avoid this costly communication with userspace.

The third limitation is the current list of eBPF helpers available. When the encoder generates a repair symbol or the decoder recovers a lost source symbol, they must create and forward a new packet in the network. As stated earlier, we currently do this in userspace as there exists no generic helper to create a packet in eBPF.

As part of future work, we aim at recompiling a custom kernel with an increased instructions limit and with new generic helpers to create new packets in the eBPF environment. We could hence avoid communication with userspace. It should improve the overall performance and make our implementation fully pluginizable. We will also try to reduce the number of branches the verifier must check, to decrease the program complexity. As we saw previously, branching (especially in for-loops) increases the number of instructions to check, thus easily reaching the instruction limit.

3.5 Deployment of the plugin

After compiling the whole plugin, we can launch the two executables on the desired routers. The kernel will verify and load the eBPF programs and start the userspace programs waiting for `perf events`. The eBPF programs and maps are pinned into the eBPF file system and attached to an IPv6 SID using `iproute2`. We can automate this process by launching the executables with the correct flags.

Currently, both the encoder and the decoder must know a priori information about each other.

The decoder. It must know the encoder Controller SID, as it is the destination of the statistics update messages. The decoder must also know its own FEC SID, first to set the source address of the controller packets, but more importantly to detect the next segment in the *Segment List* of the SRH of a recovered packet (Algorithm 1).

The encoder. It must know the FEC decoder SID as it is the destination of the repair symbol packets; its own SID is the source of these packets.

Both the encoder and the decoder also need to know the used FEC Framework to attach the correct section of the eBPF implementation to their respective SIDs, as in the eBPF implementation, the Block and Convolutional FEC Frameworks have different sections. We hence must know which one to attach when starting the plugin. An improvement would be for the decoder to detect the used FEC Framework when it receives a source symbol but we encountered problems with the eBPF verifier attempting to implement this extension. We justify this choice by stating that the FEC Framework should not change during the lifetime of the plugin, so the user should know in advance the Framework to use.

Listing 3.1 shows the different options when launching the encoder plugin. The encoder chooses the FEC Scheme parameters (block size, window size, window step).

```
./encoder [-e encoder_IPv6] [-d decoder_IPv6] [-a -i interface]
    -f framework (default: convo): FEC Framework [convo, block]
    -e encoder_ip (default: fc00::a): IPv6 of the encoder router
    -d decoder_ip (default: fc00::9): IPv6 of the decoder router
```

```
    -b block_size (default: 3): size of a FEC Block (if block)
    -w window_size (default: 4): size of the FEC Window (if convo)
    -s window_slide (default: 2): window slide (if convo)
    -a attach: if set, attaches the program to *encoder_ip*
    -i interface: interface to attach (if -a)
    -c controller_ip (default: fc00::b): activate the controller
    -l latency (default: 1000): latency of the controller
    -t threshold (default: 98): loss threshold to activate repair symbols
```

Listing 3.1: Utilization of the encoder plugin.

Listing 3.2 does the same for the decoder plugin. We do not need to specify the FEC Scheme parameters as they are detected by the eBPF program upon reception of the repair symbols.

```
./decoder [-e encoder_IPv6] [-d decoder_IPv6] [-a -i interface]
    -f framework (default: convo): FEC Framework [convo, block]
    -e encoder_ip (default: fc00::a): IPv6 of the encoder router
    -d decoder_ip (default: fc00::9): IPv6 of the decoder router
    -c controller_ip (default: fc00::b): IPv6 of the encoder controller
    -a attach: if set, attaches the program to *encoder_ip*
    -i interface: interface to attach (if -a)
```

Listing 3.2: Utilization of the decoder plugin.

Once both plugins are installed on the routers, we can add an IPv6 Segment Routing Header for the destination we want to protect using `iproute2`, inserting the two segments related to the encoder and decoder parts of the FEC plugin with the following command, or relying on `setsockopt` options of a socket:

```
ip -6 route add <destination IPv6> encap seg6 mode inline
    segs <encoder IPv6>,<decoder IPv6> dev dev <device>.
```

Chapter 4

Experiments

This chapter presents the performance of our implementation of the FEC plugin. We analyze it under different aspects and scenarios. We first explain our experimental setup. Then, we experimentally compute the solution overhead and present benchmark results. Finally, we assess the recovering capabilities in lossy networks under various situations and analyze the impact of the controller.

4.1 Experimental setup

We detail here the experimental setup we used for our experiments, as well as how we simulate packet losses. Our design makes these experiments reproducible for further analysis. Except stated otherwise, we run all our experiments using the IPMininet [64] environment, a python library extending Mininet [65] that supports the emulation of complex IP networks, and SRv6 in particular. This environment lets us create complex topologies on a single machine.

4.1.1 Experimental topology

We use the same network topology for all experiments, represented in Figure 4.1. The encoder plugin runs on `rE` and the decoder on `rD`. The loss model, discussed next, is attached at the ingress of `rD` on the `rE-rD` link to simulate packet losses between the encoder and the decoder for traffic coming from `rE`. We use the router `rA` to encapsulate packets to the desired destinations with a Segment Routing Header with segments related to the encoder and decoder plugins[1]. To emulate a real-life network, we add a small delay on the `rE-rD` link; except stated otherwise, this value is 10 ms. We finally attach clients (respectively servers) to `hA` (resp. `hD`) for our experiments.

Figure 4.1: Linear topology to test our FEC plugin.

4.1.2 The loss model

As discussed by Borella [66], real-life packet losses in networks generally happen in bursts. To model this, we design a two-states Markov model [67] to simulate such loss patterns, represented in Figure 4.2. When a packet is in the `PASS` state, it is correctly forwarded. In the `DROP` state,

[1] This is just a design choice. It is also possible to add the SRH insertion on `hA` directly.

it is discarded to simulate a loss. We configure the model with two parameters: k represents the probability of remaining in the `PASS` state if we are already in this state. We therefore have a probability of $1-k$ to enter `DROP` from `PASS`. Conversely, the d parameter influences the probability to remain in the `DROP` state, simulating a burst loss. We can create dropping models with burst losses of length $\frac{1}{1-d}$ on average. The lower (resp. higher) the value of k (resp. d), the more the network will suffer from losses. We use a pseudo-random number generator using a seed to simulate the losses, making the erasures reproducible. Regarding the implementation, this Markov loss model is an eBPF program attached using `tc`. Table 4.1 shows the range of values of the parameters used for our experiments.

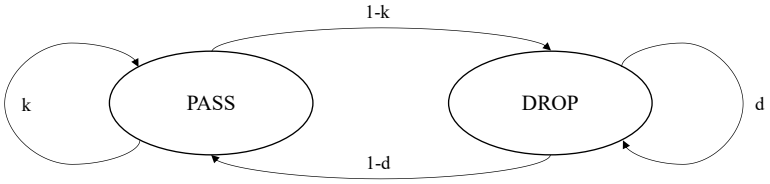


Figure 4.2: Two-states Markov model used to drop packets.

	Min value	Max value
k	0.9	0.99
d	0.02	0.5

Table 4.1: Parameters range of the Markov model used for the experiments.

4.1.2.1 Properties of the model

Leveraging the Markov chain properties, we can compute the static distribution to show the expected number of losses of the model. Our one-step transition matrix $P_{k,d}$ is defined by Equation 4.1, where the first state corresponds to `PASS` and the second to `DROP`.

$$P_{k,d} = \begin{pmatrix} k & 1-k \\ 1-d & d \end{pmatrix}. \tag{4.1}$$

The stationary distribution π is such that it does not change if we apply the transition matrix on itself. We can find the stationary distribution by solving the system given by Equation 4.4, where I is the identity matrix.

$$\pi = \pi \cdot P_{k,d}, \tag{4.2}$$

$$0 = \pi \cdot (P_{k,d} - I), \tag{4.3}$$

$$0 = \pi \cdot \begin{pmatrix} k-1 & 1-k \\ 1-d & d-1 \end{pmatrix}. \tag{4.4}$$

This system is undetermined, but we can add a new equation knowing that $\pi_1 + \pi_2 = 1$. The final system hence becomes:

$$\begin{cases} (k-1)\pi_1 & +(1-d)\pi_2 & = 0 \\ \pi_1 & +\pi_2 & = 1 \end{cases} \tag{4.5}$$

Solving this linear system gives the static distribution of the two states:

$$\begin{pmatrix} \pi_1 & \pi_2 \end{pmatrix} = \begin{pmatrix} \frac{d-1}{k+d-2} & \frac{k-1}{k+d-2} \end{pmatrix}. \tag{4.6}$$

Packets are dropped whenever the Markov chain is in the `DROP` state, whose stationary distribution is given by π_2. Depending on the values of k and d, π_2 shows the expected proportion of lost packets during an experiment.

4.1.3 Simulations and experimental design

During our experiments, we present different scenarios that can benefit from FEC. For each situation, we measure the evolution of different metrics with the parameters of our Markov loss model. Even if we present the detailed results for specific values of the model, we also use the experimental design approach [68] to give a global overview of our solution.

Indeed, we defined our loss model with two parameters but we do not know their values in real-life situations. The experimental design approach removes this burden by sampling values from the defined range of the parameters and reports the Experimental Cumulative Distribution Function (ECDF) associated with the results, allowing to explore numerous scenarios that could happen in real-life communication. In our case, we uniformly sample the space defined by Table 4.1 for our Markov loss model. We initially sampled this space with 104 points, but some experiments required more runs to get accurate results, and we sample them with 260 points - we detail for each graphical result the number of samples in the caption. For each sample of k and d, we repeat the experiment three times and only report the median value to decrease the variance of the results. The experimental design approach is computationally intensive but is increasingly used in scientific publications nowadays [47, 69].

4.2 Plugin speed benchmarks

The End.BPF SRv6 action processes packets in the kernel, taking advantage of the eBPF virtual machine. Running user-defined programs directly in the kernel allows manipulating the packets without requiring context changes between the kernel and userspace. However, stopping the packet processing to start the eBPF program associated with the current SID adds some overhead, increasing the packet processing time, and therefore reducing the maximum number of packets the router can handle in a defined amount of time [4]. The current section presents three benchmarks. First, we analyze how the encoder plugin behaves when dealing with intensive traffic. Even if IoT devices do not consume much bandwidth [70], we find it interesting to explore the limits of our implementation. Then, we analyze the time required for the RLC FEC Scheme to encode/decode packets and finally, we compare the end-to-end delay increase using our plugin with XOR and RLC.

4.2.1 Encoder maximum performance

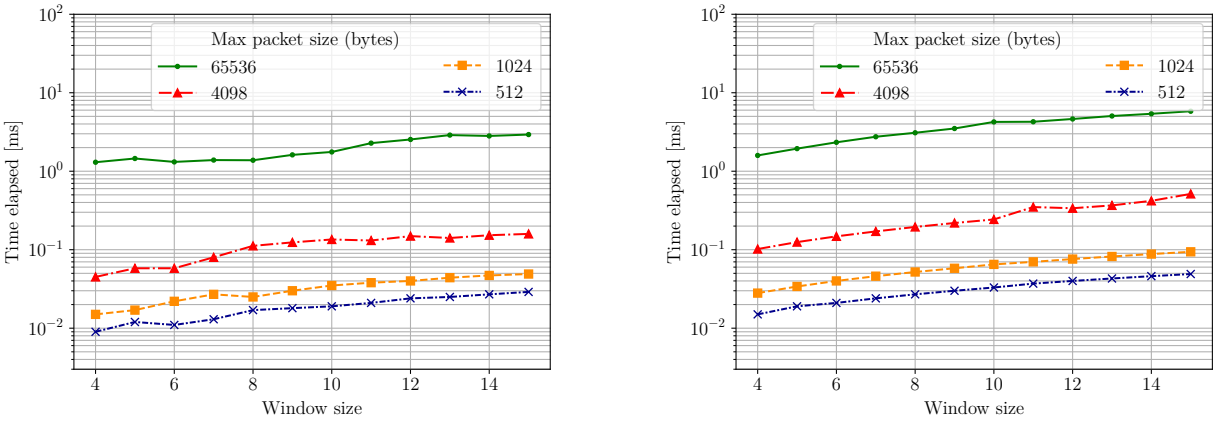
To analyze how many packets the encoder plugin can process, we use two Raspberry Pi 3B+ connected via an Ethernet link with a maximum capacity of 300 Mbps on this link. The first device generates traffic using `trafgen`, an efficient tool to generate artificial traffic into a network [71], and the second runs the encoder plugin. The encoder generates and sends repair symbols, but these are not taken into account in the packet counting. We generate 110 bytes packets containing a Segment Routing Header with the encoder SID. When we send traffic at a speed of 25 Mbps, we measured that the encoder plugin processes all of them without any problem. However, we started noticing a performance drop when sending packets at 60 Mbps: 10 % of the packets are not processed by the plugin, and dropped since the waiting queue of the plugin is full. Finally, when `trafgen` generates traffic at 110 Mbps, the plugin could only process half of the packets sent. We could not generate traffic at a speed between 60 Mbps and 110 Mbps

using the Raspberries since the `-b` option of `trafgen` - controlling the bandwidth - constraints the utilization to only 1 CPU, which was insufficient to generate traffic above 30 Mbps alone.

4.2.2 RLC FEC Scheme benchmark

As explained in section 3.2.1, the RLC FEC Scheme is partially implemented in userspace. More precisely, the encoder encodes the source symbols from its window to generate the repair symbol, and the decoder creates the linear system composed of all received symbols to recover the lost source symbols. Since we perform these operations in userspace, we can measure their processing times independently of the plugin. To this end, we created a small benchmarking program that simulates the other parts of the plugin and evaluates the encoder/decoder RLC FEC Scheme. We compare the performance for various values of the *window size* w of the convolutional buffer and for multiple packet sizes. Indeed, as we work in userspace, we are not constrained anymore by eBPF and can manipulate bigger packets. The *window step* s is equal to 2.

For the encoder, we fill in a window buffer of size w containing dummy values and call the encoder RLC program to create a repair symbol. Concerning the decoder, we build an entire convolutional buffer of $w + s * 4$ source symbols so that we potentially construct a system of several equations (one for each considered repair symbol). We first simulate the loss of one source symbol to evaluate the decoder RLC performance.



(a) Encoder benchmark. We encode the full window specified by the *window size*.

(b) Decoder benchmark. We simulate the loss of one source symbol in the convolutional buffer.

Figure 4.3: Encoder and decoder RLC FEC Scheme benchmarks. Each point of the figures is the median of 4000 experiments.

Figure 4.3 shows the time elapsed by the encoder and decoder RLC FEC Schemes for different values of w. Firstly, we see a linear increase of the processing time with the window size for both programs (except the encoder with a packet size of 65536 bytes, where we do not observe a true linear increase). As expected, the time (mostly linearly) increases with the packet size. The encoder takes 2 ms to encode 15 source symbols of 65536 bytes each, meaning that it can encode at a speed of 491 MBps for this packet size. We also observe that the slope of the curves is more or less the same no matter the packet size, which only adds a constant factor to the computation time. Concerning the decoder, we also see that the linear increase is the same for all considered packet sizes. We see that the decoding process takes slightly more time to recover a packet than the encoder to generate a repair symbol. It is due to the pre-processing of the

decoder to construct the linear system and finally solve it. However, the order of magnitude is the same for both processes.

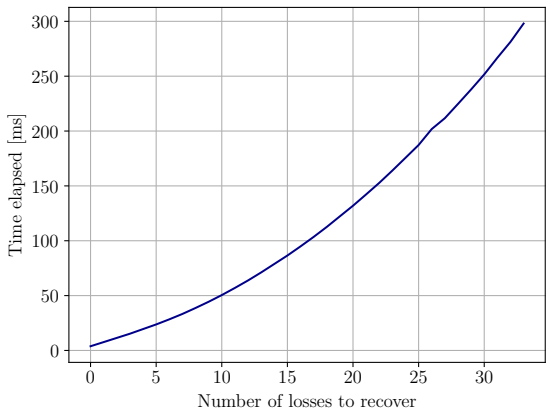


Figure 4.4: Elapsed time for the RLC decoder to recover the lost source symbols, with a constant $w = 2$, $s = 10$ and a packet size of 65536 bytes.

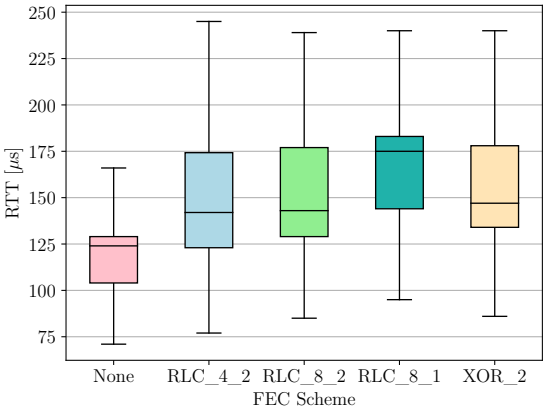


Figure 4.5: RTT of 1000 `ping` requests. `RLC_X_Y` means the Convolutional RLC FEC Scheme with a window size of `X` and a window step of `Y`. `XOR_X` means the Block XOR FEC Scheme with a block size of `X`.

We also measured the impact on the computation time when increasing the number of lost source symbols to recover. For this experiment, we keep the RLC parameters $w = 2$ and $s = 10$. Figure 4.4 shows that the time needed to recover the lost source symbols increases with a polynomial factor with the number of the lost symbols. We expected this result because the Gaussian elimination has a time complexity of $\mathcal{O}(n^3)$ with n the number of lost symbols.

4.2.3 Impact of the plugin on the RTT

A last interesting metric to analyze is the additional delay encountered by a packet that is protected by the plugin when we do not encounter any loss - the next section will analyze the behavior of the plugin in lossy networks. We use the topology presented in section 4.1.1, where we remove the 10 ms delay, and use the `ping` command to measure the round-trip time (RTT) for `hA` to send an *Echo Request* to `hD` and receive the *Echo Reply* response. The client sends an *Echo Request* every 100 ms:

```
ping -6 -c 1000 -i 0.1 <hD IPv6 address>
```

Even if the network does not encounter any loss, the encoder plugin generates and sends repair symbols as we do not use the controller in this section. Figure 4.5 shows the measured RTT over 1000 tests for different FEC Schemes. We first see that using the plugin increases the RTT compared to the baseline, from 120 μs to 140 μs for the fastest analyzed FEC Scheme. As expected, increasing (resp. decreasing) the window size (resp. window step) of RLC also increases the median RTT as the plugin generates and manipulates more repair symbols. Surprisingly, the XOR FEC Scheme is not faster than RLC. We explain this behavior by the fact that the XOR FEC Scheme encodes and decodes in the eBPF program, whereas the RLC encoding/decoding is implemented in userspace. Therefore, we expect the RTT to increase if the RLC FEC Scheme is fully implemented in the eBPF program.

4.3 The benefits of FEC in lossy networks

This section analyzes how our FEC implementation behaves in networks with packet losses. We present different experiments that simulate real scenarios where Forward Erasure Correction can be helpful. Due to the current limitation of packets of 512 bytes maximum, we focus on the plugin in IoT networks. Indeed, most IoT devices send packets of relatively small size, do not have a lot of resources and do not send much traffic [70]. Some devices cannot even provide a fully reliable transfer and use only UDP to exchange data. We measure the benefits of the plugin in such situations. We also analyze the utility of using our implementation with TCP, where the packet losses increase the end-to-end delay and reduce the throughput due to the need for retransmissions. Finally, we assess our plugin with MQTT [72], a standard protocol for IoT communication.

4.3.1 SRv6-FEC over UDP

We analyze in this section the benefits of our implementation with UDP traffic to measure the impact of our plugin with an unreliable transport protocol. We run on `hD` a server using `iperf3` [73]. With the topology from Figure 4.1 (hence with the 10 ms delay back), we start on `hA` the transmission of data for 30 seconds, at a rate of 1 Mbps:

```
iperf3 -c 2042:dd::1 -t 30 -b 1000000 -udp
```

We run our benchmark with UDP alone and with our SRv6-FEC plugin. We test our two FEC Schemes: XOR with a block size of 2 and RLC with a window size of 4, and a window step of 2.

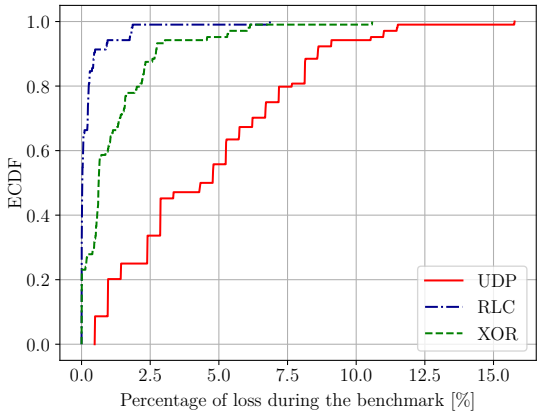
4.3.1.1 Recovering capabilities

The first interesting metric is the number of lost packets. This quantity is given by `iperf3` after each test. The experimental cumulative distribution function of our tests (Figure 4.6a) demonstrates that we can drastically decrease the losses using our plugin. In more than 90 % of our experiments, RLC over UDP decreases the number of losses to less than 1 % of the total number of packets. On the contrary, the UDP-only transmission experiences 5 % or more losses in more than 50 % of our tests.

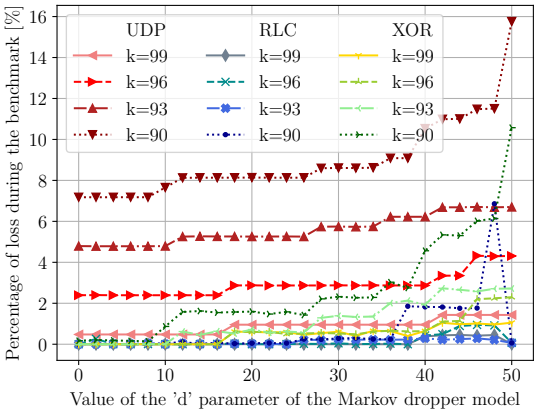
Figure 4.6b shows the results for specific values of k and d. We first notice that both k and d impact the loss percentage for the standard UDP traffic, as expected from section 4.1.2.1. As presented earlier, our RLC implementation behaves almost perfectly when we do not experience many losses but starts suffering when the network becomes too sharp ($k = 90$, $d > 35$). We explain this inefficiency with the small window size (4) of the Convolutional Framework used during the experiments. By looking more closely at the traces, we see that the failures of RLC happen when we experience loss bursts longer than 3, which cannot be recovered with the tested FEC Scheme, comforting us in our hypothesis. With the simpler XOR FEC Scheme, we also reduce the number of losses compared to UDP but still experience many failures. Figure 4.6b shows that it especially happens when the network is more subject to burst drops (d increasing), confirming the mentioned limitations regarding this simple scheme. Even in these complex situations, the plugin (with XOR but especially with RLC) still provides very good recovering capabilities.

4.3.1.2 Jitter

UDP is used for real-time applications [5, 6], where the variation in delay of packet arrival must be small [74, 75]. The previous section showed that we increase the number of received packets



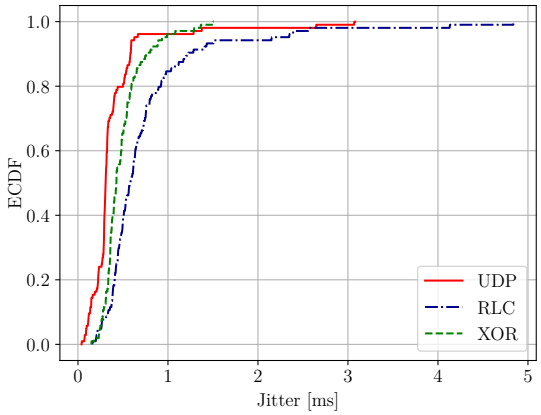
(a) Experimental CDF over the range of parameters from Table 4.1. 104 samples.



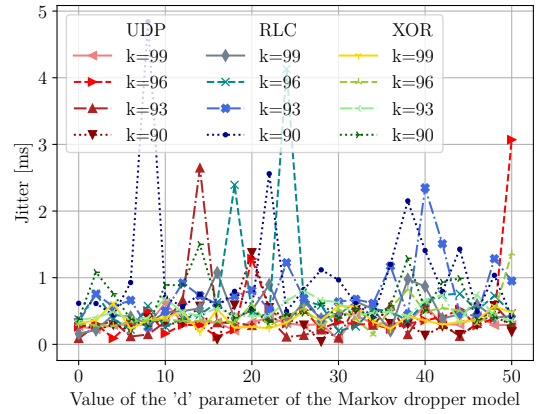
(b) Detailed results for various values of k and d.

Figure 4.6: Evolution of the number of losses during a UDP transmission with UDP and our SRv6+FEC plugin with our Markov model. The plugin runs a (3, 2, 4) RLC or a (3, 2) XOR.

with our implementation, but we must also take into account the plugin overhead. `iperf3` also provides the *jitter* metric computed during an experiment. According to the documentation [76], the *jitter* corresponds here to *the difference in end-to-end one-way delay between selected packets in a flow with any lost packets being ignored* (RFC3393 [77]).



(a) Experimental CDF over the range of parameters from Table 4.1. 104 samples.



(b) Detailed results for various values of k and d.

Figure 4.7: Evolution of the *jitter* (as defined by RFC 3393 [77]) during a UDP transmission with UDP and our SRv6+FEC plugin. The plugin runs a (3, 2, 4) RLC or a (3, 2) XOR.

Figure 4.7 shows the *jitter* ECDF of our solution compared to the UDP-only traffic. As expected, we experience a higher *jitter* with RLC/XOR than simple UDP. As the *jitter* is only computed on the received packets, the standard UDP traffic does not suffer from much delay variation. On the contrary, we see that the packets protected with FEC experience the implementation overhead. However, this *jitter* increase is still low, as it is only doubled for RLC compared to UDP and remains below 1 ms for 85 % of the experiments. Figure 4.7b shows the *jitter* for specific values of k and d of the Markov model. Even if it is not easy to distinguish

clearly the different curves, we observe that the UDP *jitter* is always slightly below XOR and RLC. We tried to analyze the strong increases from the figure, but it seems to be an artifact of the experiment, maybe due to the IPMininet environment. These strong variations could explain the few experiments from Figure 4.7a with a *jitter* above 2 ms for UDP and RLC.

4.3.1.3 RLC vs XOR

Figure 4.6 and Figure 4.7 also compare the two implemented FEC Schemes: RLC and XOR, both having the same code rate of $\frac{2}{3}$. The simple XOR encoding provides a smaller *jitter* than RLC (Figure 4.7a) because the XOR FEC Scheme needs less computation. Additionally, the XOR computation is done *online*, smoothing it over all source symbols, whereas the symbols triggering the repair symbols generation/source symbols recovery of RLC experience an increased delay as they are forwarded at the end of our FEC Framework. On the other hand, RLC shows better recovering capabilities as the UDP server receives on average 1 % more data than XOR, even with small window size. Figure 4.6b also provides insight into the limitations of XOR: the performance drop is correlated to the increase of burst loss probability. When $d < 10$, burst drops are relatively small and XOR performs almost identically to RLC. However, the performance necessarily decreases, as the XOR FEC Scheme cannot recover from burst losses.

In conclusion, even if the XOR FEC Scheme gives lower overhead than RLC, the latter provides better redundancy to protect traffic flows. Previous works have also reached the same conclusion [47, 78]. For these reasons, we only analyze the performance of our implementation over the Convolutional RLC FEC Scheme for the rest of this chapter.

4.3.2 SRv6-FEC with TCP connections

We now analyze the benefits of SRv6-FEC with TCP connections. With this reliable protocol, we know that all sent data will reach the destination. However, the connection latency can rise whenever we experience losses during the transmission. Using exactly the setup from Figure 4.1, we run a TCP server on `hD` and a client on `hA` using `iperf3` [73]. We add a bandwidth constraint of 15 Mbps on the `rE-rD` link using the `tc` system call in IPMininet, and configure the client to send 14 MBytes to the server:

```
iperf3 --client 2042:dd::1 --bytes 14000000 -M 280
```

As we do not constrain the speed at which `iperf3` runs, the TCP client tries to send the packets as fast as possible, given the 15 Mbps bandwidth limit. Again, we constrain the size of the TCP packets as our plugin only applies on packets of at most 512 bytes, including the Segment Routing Header. Even if real TCP transmissions send packets of larger size, we find it interesting to evaluate how our implementation can improve TCP connections in a lossy network. We also limit the duration of an experiment to 350 seconds, after which the run is stopped. The plugin runs a (3, 2, 8) RLC.

4.3.2.1 Quality of the TCP connection

The most important metric when analyzing the quality of a TCP transfer is the amount of time required to complete the transmission since packet losses will trigger retransmission and potentially increase the tail delay. We measured a median baseline connection time - without the plugin nor losses - of 9.54 seconds over 20 runs in the same setup. With the plugin - again without losses - the connection time increases to 17.54 seconds (also the median value of 20 experiments). The results presented by `iperf3` showed that with the plugin, the client bitrate is

almost half the baseline value (6.56 Mbps against 12.5 Mbps). By analyzing the packet traces, we observed that with a TCP bitrate of 6.56 Mbps, the `rE-rD` link saturated at 15 Mbps using the plugin. We explain this phenomenon by the plugin overhead adding more bytes to the link (further discussed in section 4.3.3.2). As a result, we consider the second baseline of 17.54 seconds for the TCP client with the plugin.

Figure 4.8 shows that the quality of the TCP connection duration remains close to the baseline of the plugin (17.54 seconds) even with losses. On the other hand, more than half of our experiments without the plugin did not finish within 350 seconds, more than 35 times the TCP-only baseline time. These unfinished experiments occurred when $k < 96$, i.e. when the network experiences (at least, also depending on d) 4 % of packet losses. More importantly, we observe that no run with TCP alone reaches the baseline: even a few losses drastically decrease the performance. By digging further into the results, we notice that the different plateaus correspond to the same value of k. It means that the value of d does not hugely impact the TCP performance. We explain this phenomenon by the TCP sending window that contains several packets waiting for an acknowledgment. A single lost packet will trigger retransmission from the TCP sender, but multiple consecutive unacknowledged packets could be retransmitted at the same time. A longer burst loss will thus have a smaller impact since the client needs to retransmit at least one packet anyway.

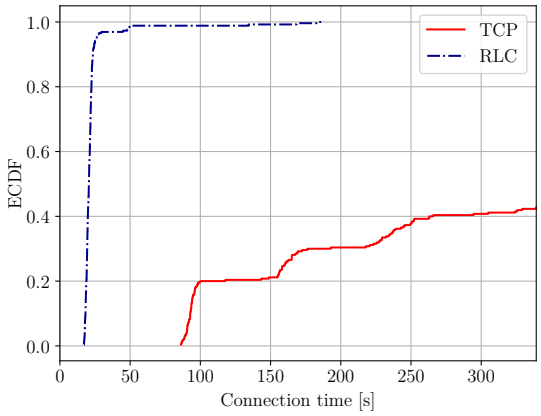


Figure 4.8: ECDF of the TCP connection time using `iperf3`. The plugin runs a (3, 2, 8) RLC. 260 samples.

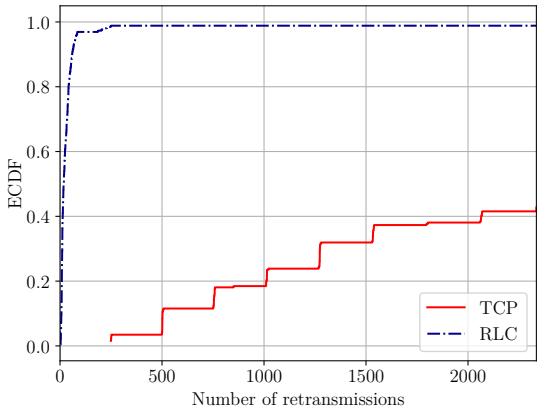


Figure 4.9: ECDF of the TCP retransmissions using `iperf3`. The plugin runs a (3, 2, 8) RLC. 260 samples.

With a window of size 4, we experienced a strong degradation of the performance compared to the window of size 8, and the completion time rarely reached the baseline. With our window of size 8, the completion time remains below 30 ms for 95 % of the experiments. Even with RLC, we suppose that the TCP sender needs to retransmit some data and for the most complex values of the Markov model ($k = 90$, $d = 50$), the completion time explodes even with the plugin, reaching 185 seconds. However, even in this extreme scenario, the plugin outperforms the TCP-only connection.

4.3.2.2 Number of retransmissions

The completion time is related to the number of retransmissions during a TCP connection [79]. Figure 4.9 shows the number of retransmissions during the connection between the client and

the server under our Markov loss model. The baseline only retransmitted a few segments when the congestion control (*cubic*) reached the bandwidth limit. As expected, the TCP client sends much fewer retransmissions when we attach our plugin. Once again, the ECDF of TCP-only is not complete as more than half our experiments did not finish within 350 seconds. The figure shows that the number of retransmissions is correlated to the connection time, especially with the plugin: we also observe that 95 % of the experiments need less than 100 retransmissions with RLC. TCP alone needs 250 of them in the best case, whereas this happens in the worst case with the plugin.

Figure 4.9 highlights a negative point. As suggested above, even with the plugin, the TCP client still needs to retransmit some packets. The used RLC FEC Scheme can recover any burst loss of length up to 4. Beyond that value, RLC cannot recover all of them, hence the need for retransmissions. These few retransmissions slightly increase the connection time as we see from the figures.

Despite this, we see that the plugin helps to decrease the number of retransmissions for a TCP client. Let us recall that our implementation is not flow-specific, and we can reach the same conclusions by aggregating multiple TCP connections simultaneously.

4.3.2.3 FEC over small TCP connections: Apache benchmark

We also consider the case of relatively short TCP connections, such as HTTP queries [80]. For such small connections, the impact of a lost packet is more important, as the TCP sender cannot simultaneously retransmit unacknowledged segments alongside new segments. Indeed, for bulky connections sending a great quantity of data, retransmission will be smoothed by the sending of new data. We consider small connections such as HTTP queries where the client sends only one packet, and retransmission means waiting at least for one RTT without sending any other information. A naive solution would be to send multiple times the same query and hope that at least one reaches the destination. This method works but heavily increases the network utilization as the same data is sent several times. Using FEC at the network layer, we can aggregate flows from different senders simultaneously and create intelligent redundancy to recover a specific query thanks to the overall traffic in the network.

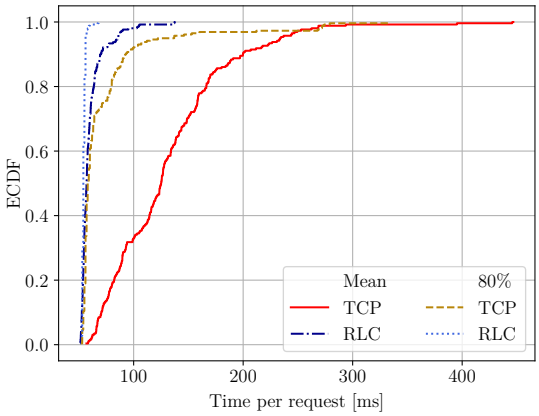


Figure 4.10: ECDF of the mean and 80$^{\text{th}}$ percentile time per request the of the Apache Benchmark. The plugin runs a (3, 2, 4) RLC. 260 samples.

We use the Apache Benchmark tool on `hA` to simulate the queries to a SimpleHTTPServer running on `hD`. A test consists of 2000 requests with 10 concurrent clients. The TCP connection is always closed after the client receives the HTTP response. The queries are performed with HTTP/1.0 and are 171 bytes long. The length of the single HTTP response is 580 bytes but it is not protected by FEC since the plugin protects packets from `rE` to `rD`.

As each experiment is relatively short (a request and response), a single run can be unaffected

by the losses, where another request may be retransmitted multiple times. Indeed, we observed a strong variance by computing the mean time per request (TpR) over the 2000 runs of each sample of the Markov parameters. We hence decided to also report here the ECDF of the 80^{th} percentile of each experiment, i.e., the time per request such that 80 % of the 2000 requests fall below this value for a sample of the Markov parameters.

Figure 4.10 shows the results with our plugin running a (3, 2, 4) RLC. The baseline needs 43 ms for the entire query/response, and we see that using the plugin we remain close to this value in 80 % of the experiments. The 20 % remaining concern the most complex scenarios, where some queries cannot be recovered by RLC due to a long burst loss, and experience a higher tail delay. The 80^{th} percentile curve with RLC also confirms that the mean TpR is influenced by a small number of longer queries: even in the most complex scenario ($k = 90$, $d = 50$), 80 % of the HTTP connection times fall below 59 ms, only 16 ms above the baseline, whereas the mean TpR is almost tripled.

We once again see the plugin benefits when comparing with TCP-only. In 10 % of the experiments, the 80^{th} percentile TpR is higher than 100 ms, and the mean TpR above 200 ms. This analysis of small TCP connections shows that using FEC at the network layer we can protect multiple small flows together in a coding channel, and recover lost packets thanks to the other connections.

4.3.3 MQTT benchmark

As stated earlier, a use-case of our implementation is to protect the transmission of data in IoT networks. This part focuses on the advantages of using FEC to protect data from MQTT clients. Message Queuing Telemetry Transport (MQTT) [72] is a publish-subscribe network protocol to exchange information between IoT devices. The clients send messages to a *broker* (i.e. an MQTT server) and subscribe to specific channels to get the messages from other clients. This lightweight protocol is heavily used in networks with resource constraints [81]. MQTT runs over TCP to provide a reliable data transfer. However, the devices may suffer data losses, heavily increasing the tail delay.

To measure the performance of our plugin, we use an open-source MQTT benchmark implemented in `go` [82]. This tool simulates MQTT clients attempting to push messages to the broker as realistic IoT devices would do. We run this benchmark on `hA`. To simulate our MQTT broker, we use the Eclipse Mosquitto project [83], one of the most common brokers in practice. We run `mosquitto` with the initial configurations on `hD`.

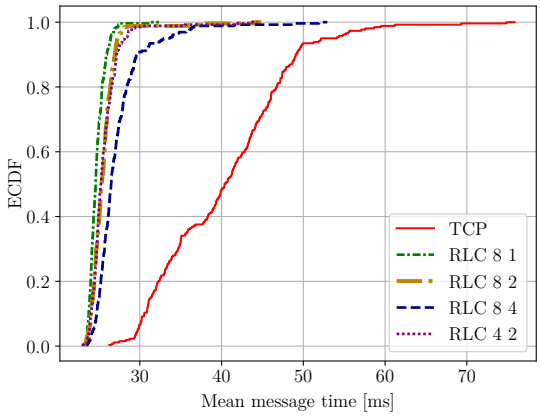
We start each benchmark with ten clients, each sending a total of five hundred messages. We decided to simulate ten clients simultaneously to show the benefits of FEC at the network layer, where we can aggregate multiple flows together to protect them. Each message contains an MQTT payload of 100 bytes:

```
mqtt-benchmark --broker tcp://[<broker IPv6 address>]:1883 --clients 10 --count 500
```

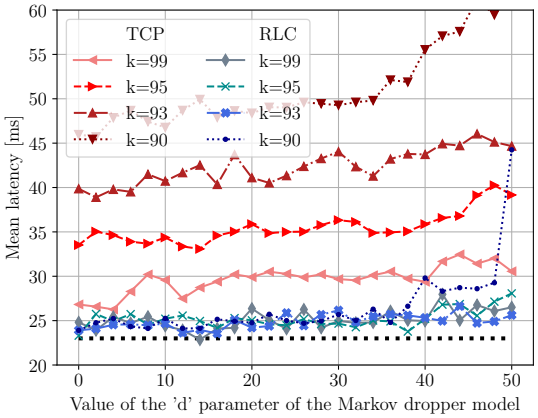
This real-life use case is also the opportunity to compare the performance of our plugin with RLC for different parameters. To get more accurate results to analyze the impact of these parameters, we uniformly sample the parameter space of our Markov loss model with 260 points.

4.3.3.1 Latency evolution

As MQTT runs over TCP, we know that all data will eventually reach the destination. We first focus on the mean latency for a client to send a message to the broker. More precisely, we analyze the message mean time (MMT) of the MQTT benchmark.



(a) Experimental CDF over the range of parameters from Table 4.1. 260 samples.



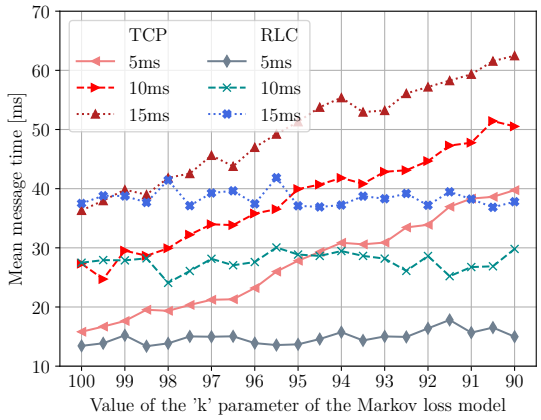
(b) Detailed results for various values of k and d.

Figure 4.11: Evolution of the message mean time for an MQTT client to send a message to the Mosquitto broker, under our Markov loss model. `RLC X Y` is the RLC FEC Scheme with a window size of `X` and a window step of `Y`. Figure 4.11b only shows the results for RLC (3, 2, 4) (or `RLC 4 2`).

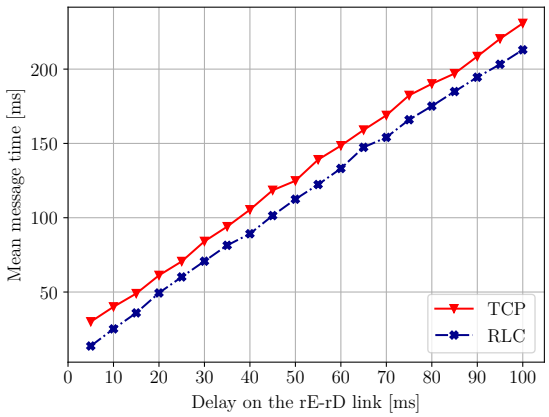
Figure 4.11a shows that we drastically decrease the mean latency by adding our plugin when the network experiences packet drops. Without FEC, more than 50 % of the experiments suffer from a delay superior to 40 ms, where the baseline - without the plugin nor losses - is 23 ms. On the contrary, depending on the RLC parameters, the maximum MMT lies in 99 % of the experiments between 34 ms and 40 ms. Moreover, this quantity is lower than 30 ms for 90-99 % of the experiments, also depending on the RLC parameters. We see that FEC recovers numerous lost packets, hence decreasing the need for costly TCP retransmissions. Figure 4.11a shows a general tendency, but we can dig further and look at specific values of the Markov loss model. Figure 4.11b presents the evolution of the same metric but now specifically looking at some values of k and d. First, we see that the performance with the plugin is comparable to the baseline in most situations, meaning that our implementation does not slowdowns the performance of the transmission. The real benefit of FEC reflects for smaller values of k: with the plugin, the MMT remains almost identical and slightly increases for the more complex situations ($k = 90$, $d > 40$) but stays below 30 ms (except when $k = 90$, $d = 50$). Without the plugin, the mean latency constantly increases as we decrease (respectively increase) k (resp. d). It can even become more than twice as high as the baseline value.

Figure 4.11a also shows the impact of the window size and window step on the results. As expected, we provide better recovering capabilities when the window step decreases: we generate more repair symbols for the same set of source symbols, thus increasing the probability of recovering longer burst losses. The dotted line (`RLC 4 2`) illustrates that reducing the window size from 8 to 4 slightly decreases the performance, but its impact is lower than the window step. By raising the window step from 2 to 4, we increase by 10 % the number of experiments

with an MMT over 30 ms, which shows the importance of the step value to recover burst losses.



(a) We vary the value of k, and analyze a delay of 5 ms, 10 ms, 15 ms.



(b) We keep $k = 95$ and vary the delay in the range 5 ms to 100 ms by steps of 5 ms.

Figure 4.12: Evolution of the message mean time for an MQTT client to send a message to the Mosquitto broker. We vary the delay on the `rE-rD` link and test with our Markov loss model with a constant value of $d = 30$ for graphical reasons.

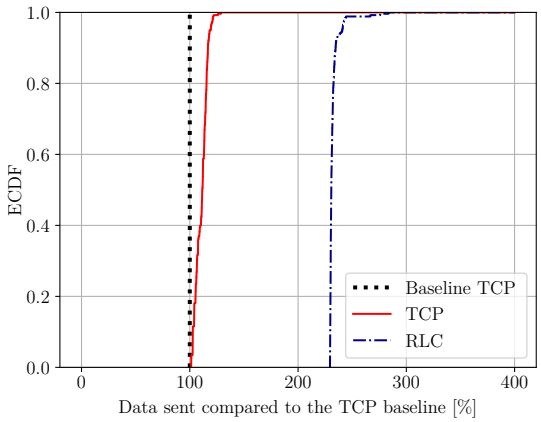
The delay on the `rE-rD` link may also impact the evolution of the message mean time for the MQTT clients. Figure 4.12a shows how this delay influences the MMT of our MQTT clients with our Markov model. With the plugin, the MMT remains almost identical for a given link delay even when the network encounters more losses (as k decreases). On the contrary, MQTT clients above TCP-only suffer when we add more losses, and the MMT linearly increases as we decrease k. However, this linear increase stays the same no matter the initial delay on the link.

We also perform this experiment with a delay varying from 5 ms to 100 ms in steps of 5 ms (Figure 4.12b). No matter the value of the delay, the gap between RLC and TCP remains the same. However, the figure shows that the MMT rise with respect to the delay follows a slope of almost 2, which is expected since the RTT is approximately two times the one-way delay.

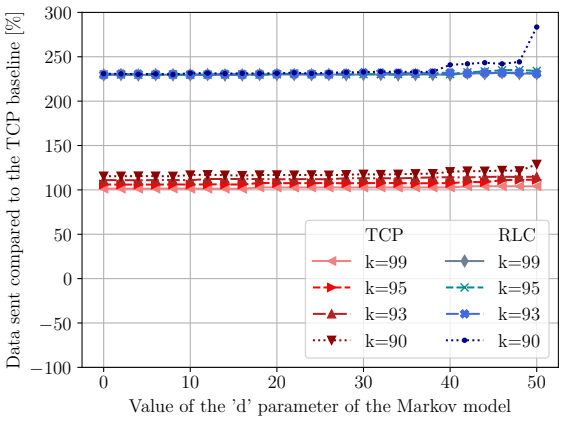
4.3.3.2 Network utilization overhead

The redundancy added by our FEC plugin comes at the cost of higher link utilization. More redundancy will improve recovering capabilities, but each repair symbol is a new packet in the network. It then becomes important to consider this overhead when deploying FEC in a network.

We analyze this overhead on the `rE-rD` link. Figure 4.13a shows that the plugin sends on average 1.25 times more data than our baseline, the same test without the plugin and in a perfect network. When the network experiences losses, the TCP connections send more data proportionally to the losses on the link. Figure 4.13b shows the same results, but for specific values of k and d. Without the plugin, the quantity linearly increases with the complexity of the network. When k drops from 99 to 90, the increase is about 10 % for the same value of d. It also seems that the value of d does not impact this quantity very much for TCP alone. A possible explanation is that the range of d of the Markov model does not allow for really long burst losses (the maximum average burst loss is only 2, even if we also measured longer burst losses during the experiments with a lower probability), as we start seeing a small increase with the highest values of d ($d > 45$).



(a) Experimental CDF over the range of parameters from Table 4.1. 260 samples.



(b) Detailed results for varius values of k and d.

Figure 4.13: Evolution of the number of bytes sent by MQTT clients to the Mosquitto broker, depending on the parameters of the Markov loss model. The FEC plugin runs a Convolutional RLC (3, 2, 4).

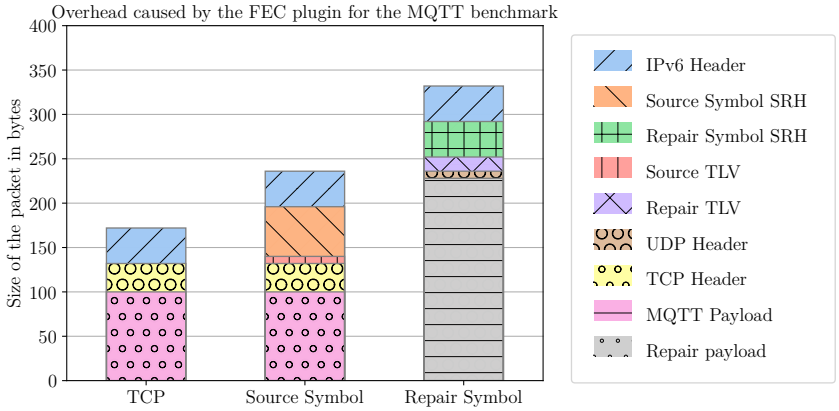
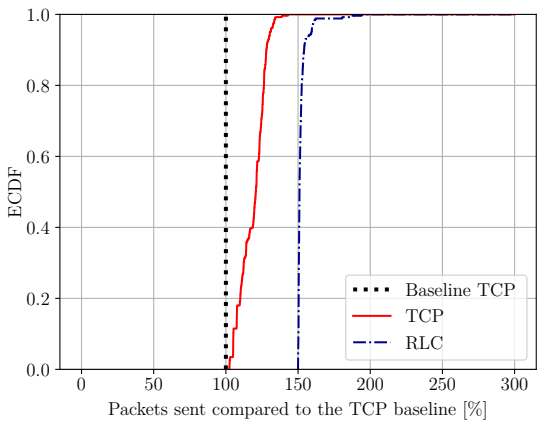


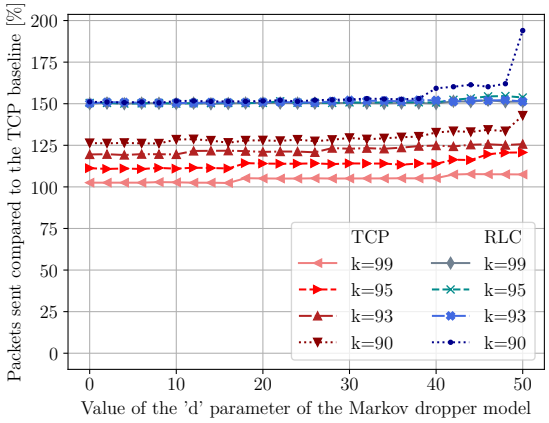
Figure 4.14: Overhead of FEC in SRv6 for an MQTT-publish message.

Our plugin over the TCP connections sends much more data. This observation is common in previous works implementing FEC [47]. Here the theoretical code rate is $\frac{2}{3}$, meaning that we should only see a 50 % increase with the plugin. However, the plugin is deployed using a Segment Routing Header, thus increasing the size of the protected packets. We must also take into account the TLV-encoded *Source FEC Payload ID*, or the IPv6 packets carrying the repair symbols. Figure 4.14 shows the content of an MQTT *publish message* packet, its mapping to source symbol and the corresponding repair symbols. To deploy the FEC plugin, the packets must contain the SRH that adds a constant overhead of at least 72 bytes. On the other hand, the messages only carry 100 bytes of MQTT payload. Thus, the cost of the Segment Routing Header is not amortized by the payload we protect, explaining this 125 % increase in the use of the link. Protecting longer packets would amortize the cost of the SRH, and hence decrease the relative overhead from Figure 4.13a.

To support our hypothesis, we also measured the link overhead in terms of packets forwarded in the network. In this situation, we do not consider the impact of the SRH on the packets, but only the cost of the generated repair symbols and the retransmitted packets. Figure 4.15a



(a) Experimental CDF over the range of parameters from Table 4.1. 260 samples.



(b) Detailed results for various values of k and d.

Figure 4.15: Evolution of the number of packets sent by MQTT clients to the Mosquitto broker, depending on the parameters of the Markov loss model. The FEC plugin runs a Convolutional RLC (3, 2, 4).

confirms that our plugin respects the $\frac{2}{3}$ code rate, i.e., we send 50 % more packets corresponding to the repair symbols. The difference with TCP alone is smaller when considering the number of sent packets during the benchmark. By considering protecting longer packets, Figure 4.13a would more closely correspond to Figure 4.15a.

Figure 4.15b shows that without the plugin, the MQTT clients must send more data when the network experiences more losses. On the contrary, with FEC, this quantity remains constant most of the time, meaning that the MQTT clients do not need to retransmit many packets. In the context of IoT devices with constrained resources, it is interesting to avoid costly retransmissions from the devices themselves. With FEC, this redundancy is added by intermediate routers, hence hiding the losses to the devices. Even if we send on average more data and more packets, the plugin offers more stability and we better control the link utilization as it remains constant whatever the losses of the network, not even considering the positive impact on the message mean time (section 4.3.3.1). The only exception presented by Figure 4.15b is when the parameters of the Markov model give strong losses ($k = 90$, $d > 40$). By analyzing the logs of our model, we see that these scenarios give longer loss bursts that are theoretically impossible to recover with our $\frac{2}{3}$ code rate and a window of 4, hence the performance drop.

4.4 Impact of the controller

The controller helps to decrease the byte overhead on the protected link/path by disabling the sending of repair symbols. Our objective is to analyze how the plugin behaves in this lossy environment with our controller.

4.4.1 Analyzing a single trace

Figure 4.16 shows the impact of the controller on the utilization of the `rE-rD` link and the number of bytes received by the server `hD`. We use the same setup and experiment as in section 4.3.1 but running for 180 seconds. The decoder sends an update message every 1024 source symbols and

the controller triggers the repair symbols generation when it observes 2 % or more losses. The Markov loss model runs with $k = 95$ and $d = 30$. We also modified our eBPF dropper to change its state every 8000 received packets to alternate between the Markov model and a fully reliable network.

The figure shows that the controller has a positive impact on the link utilization. When the controller does not detect any loss, the number of bytes sent by `rE` is almost identical to what `hD` receives, meaning that the plugin overhead is limited. We explain this slight difference by the presence of the *Source FEC Payload ID* in the protected packets between `rE` and `rD`, which is not present when the packet reaches `hD`. When the controller detects losses, it sends an update notification to the encoder to start the repair symbols generation, and the encoder sends as many bytes as without the controller.

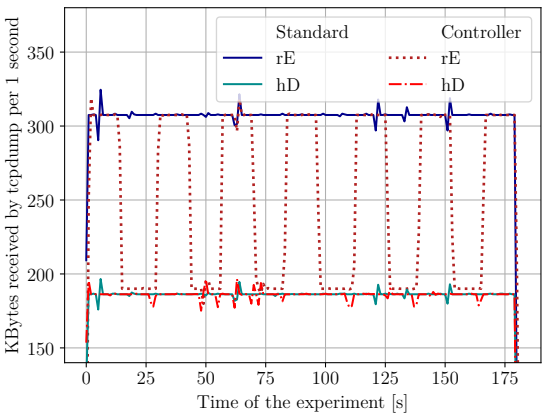


Figure 4.16: Evolution of the bytes sent (respectively received) by `rE` (resp. `hD`) during a UDP transmission, with (i.e. the *Controller* curves) and without (i.e. the *Standard* curves) our controller.

Concerning the data received at `hD`, the figure shows that the plugin performs identically with and without the controller. We only experience a performance drop every time the encoder starts to generate the repair symbols (i.e., the *controller-rE* curve suddenly rises). We explain it with the design of the controller on the decoding side, as we expected the recovering capabilities of the plugin to decrease with the controller. Indeed, when the controller first experiences losses, the encoder does not generate any repair symbol yet. We will experience a performance drop between the first losses and the update message sent by the decoder to the encoder controller to trigger the repair symbols generation. It is the only scenario where the controller will negatively impact the performance.

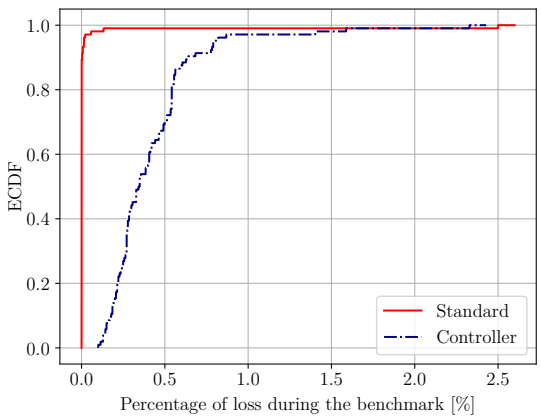
However, this performance drop can be controlled by the two parameters of the controller: the threshold above which we trigger the repair symbols generation and the controller update notification rate. The lower the first quantity, the more susceptible the controller will be to trigger repair symbols generation. The update notification rate is defined by the number of source symbols between two updates. If we send an update message every x source symbols, the approximation of the maximum number of packets we can lose, l is given by Equation 4.7.

$$l = \underbrace{x}_{\text{Number of symbols to trigger an update}} + \overbrace{t * p}^{\text{Symbols sent during the update transmission}} . \tag{4.7}$$

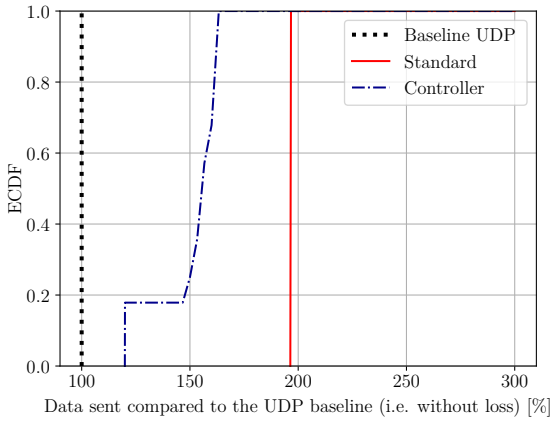
Where t indicates the time (in seconds) between the update packet generation and the reception by the encoder, and p the number of source symbols sent by the encoder per second. Conversely, decreasing x also means that the decoder controller sends more update messages, meaning that we increase the controller overhead and bandwidth utilization.

4.4.2 The controller on the long run

We test the controller over our plugin in the same scenario as discussed in section 4.3.1, i.e., over a UDP traffic for 30 seconds. We use the modified version of the eBPF dropper presented earlier: the program continues to switch between the Markov model and a fully reliable network for every 8000 source symbols received. The controller triggers repair symbols generation with a 2 % loss threshold and sends an update message for every 1024 source symbols. The plugin runs a (3, 2, 8) RLC. We thus expect the overall number of losses to decrease compared to section 4.3.1 as we are using a larger window and we alternate between the Markov model and a fully reliable network.



(a) Experimental CDF of the number of losses.



(b) Experimental CDF of the link overhead (in bytes) compared to the baseline.

Figure 4.17: Comparison of the Controller performance with the plugin over a UDP traffic in a lossy network. The plugin runs a (3, 2, 8) RLC. 104 samples.

Figure 4.17 shows the trade-off between the recovering capabilities and the utilization of the `rE-rD` link. As expected, we experience more losses with the controller, but this value remains acceptable since more than 90 % of our experiments suffer less than 1 % losses. The traces confirm that these additional drops occur during the period between two updates triggering the repair symbols generation.

However, Figure 4.17b shows the advantage of the controller on the link utilization. The standard plugin (i.e., without the controller) always sends almost twice as many bytes as the baseline[2]. On the contrary, with the controller, our encoder sends on average much less data. When we do not detect any loss, the global overhead is less than 25 % compared to the baseline, and we send at most 75 % more bytes in the most complex situations. In this particular example, we see that we can unload 25 % of bytes overhead in the worst case, while keeping relatively good recovering capabilities. The beginning of the Controller curve of Figure 4.17b is also particular. Let us remember that we run the controller with a drop threshold of 2 %, and when it detects less than 2 % loss, the controller does not trigger repair symbols generation. By inspecting our results, we see that all experiments giving a link utilization overhead of less than 25 % used a Markov loss model with $k = 99$. The controller hence never detects sufficient losses to trigger

[2]The bytes overhead is less significant than what we observed in Figure 4.13a, which presented a 225 % overhead. By looking at the exchanged packets, we see that the current experiment forwards bigger packets than the experiment from section 4.3.3. As expected, the relative overhead decreases as the SRH cost is more amortized here.

the repair symbols generation. The plateau in the curve also results from this analysis: when the controller starts seeing drops, repair symbols are forwarded through the link, and the link overhead suddenly increases.

Chapter 5

Conclusion

In this thesis, we have shown that using IPv6 Segment Routing and its networking programmability, we can develop in-network Forward Erasure Correction services to protect delay-sensitive applications that are unable to implement them. It allows recovering lost data without the need for retransmissions. We designed an architecture to support FEC at the network layer using SRv6, relying on existing works and standardizations and taking into account the specific challenges occurring at this layer. Our implementation supports different FEC Schemes, and we already implemented two of them: the XOR Scheme for Block codes and RLC for Convolutional codes.

We provide a prototype implementation leveraging the eBPF support in the Linux kernel and the recently added End.BPF interface to the IPv6 Segment Routing `seg6local` lightweight tunnel. We evaluated the plugin in lossy networks by implementing a reproducible two-states Markov model simulating real-life losses. Our experiments focused on the protection of UDP and TCP traffic in lossy networks to assess the plugin, and we also analyzed its impact on an Internet of Things protocol with MQTT. The results showed that, in these situations, FEC can improve the network quality, either by decreasing the number of lost packets (for UDP) or by decreasing the end-to-end delay (TCP, MQTT). We measured the overhead of Forward Erasure Correction and showed that the Segment Routing Header cost can be amortized by protecting longer packets.

Finally, we designed and implemented a simple controller allowing to disable FEC depending on the estimated quality of the network. We showed that our controller reduces the FEC overhead when we do not encounter losses by deactivating repair symbols generation and reactivate it when we notice packet drops on the protected link/path. We measured that the only negative impact of the controller occurs when we start observing losses while the repair symbols generation is still disabled, but the plugin still improves the network quality.

Benefits and drawbacks of the solution

By implementing FEC at the network layer, it becomes possible to aggregate multiple traffics for protection, no matter the transport protocol of these flows. Moreover, this plugin can protect simple links but also entire paths just by setting an ingress and egress routers in the network. SRv6-FEC is attached to routers, hence removing the computational overhead from the hosts that are unable to implement FEC due to resource constraints. The only task that the hosts must execute is to add a Segment Routing Header inside each packet with the two SIDs of the routers implementing the plugin. This can also be performed by an upstream router between the host and the encoder. One of the biggest challenges of FEC at the network layer is to

keep routing information to recover the initial state of the lost packet, and we proposed an efficient way to retrieve the correct next destination of a recovered packet. Finally, leveraging the BPF support in SRv6, our implementation works as a plugin that a network operator can attach/detach without requiring to restart the kernel of the machine.

On the other hand, we faced multiple limitations of the BPF verifier of the kernel. Our plugin currently only supports packet protection of at most 512 bytes and needs to communicate with a userspace process for repair symbols generation and recovered packets transmitting. Conceptually, we also restrain the extended headers we can protect with our plugin, as we only ensure the protection of the SRH and the IPv6 Header. Any extended header remaining constant between the ingress and egress routers of the plugin will be correctly recovered, but we do not ensure perfect recovery if their content can change during the transit.

Open-source contributions

This thesis was also the opportunity to contribute to the open-source community. We discovered and corrected a bug in the End.BPF interface implementation of `pyroute2` [84]. We also started an issue on the Scapy project [85] because the SRv6 implementation did not follow RFC8754 [2] but still the draft-06 from March 13, 2017 [86]. The issue has been corrected by the community. Our prototype implementation is also publicly available.

Future work

As part of future work, we aim at extending the Linux kernel to cope with the current limitations we face due to the support of eBPF. More particularly, we could try to (i) add a new helper to craft and send completely new packets in an eBPF program, (ii) increase the instruction limit of an eBPF program and (iii) optimize the current prototype complexity. Our objective is more fundamentally to support the protection of longer packets to widen the domain of application of the plugin. Our current implementation could support such packets for the Convolutional FEC Framework (hence the RLC FEC Scheme) but struggles from the communication with userspace that decreases the plugin efficiency. We aim at entirely implementing the plugin in the eBPF program to avoid such transfer to userspace. However, this prototype implementation of FEC at the network layer leveraging IPv6 Segment Routing already works well and can be of great benefit especially for IoT devices with resource constraints.

Bibliography

[1] J. Frnda, M. Voznak, and L. Sevcik, "Impact of packet loss and delay variation on the quality of real-time video streaming," *Telecommunication Systems*, vol. 62, no. 2, pp. 265–275, 2016.

[2] C. Filsfils, D. Dukes, S. Previdi, J. Leddy, S. Matsushima, and D. Voyer, "IPv6 Segment Routing Header (SRH)." RFC 8754, Mar. 2020.

[3] C. Filsfils, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima, and Z. Li, "Segment Routing over IPv6 (SRv6) Network Programming." RFC 8986, Feb. 2021.

[4] M. Xhonneux, F. Duchene, and O. Bonaventure, "Leveraging ebpf for programmable network functions with ipv6 segment routing," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pp. 67–72, 2018.

[5] L.-A. Larzon, M. Degermark, and S. Pink, *UDP lite for real time multimedia applications*. Citeseer, 1999.

[6] P.-K. Lam and S. C. Liew, "Udp-liter: an improved udp protocol for real-time multimedia applications over wireless links," in *1st International Symposium onWireless Communication Systems, 2004.*, pp. 314–318, IEEE, 2004.

[7] P. Sarolahti, M. Kojo, and K. Raatikainen, "F-rto: an enhanced recovery algorithm for tcp retransmission timeouts," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 2, pp. 51–63, 2003.

[8] A. Kesselman and Y. Mansour, "Optimizing tcp retransmission timeout," in *International Conference on Networking*, pp. 133–140, Springer, 2005.

[9] L. Zhang, "Why tcp timers don't work well," *ACM SIGCOMM Computer Communication Review*, vol. 16, no. 3, pp. 397–405, 1986.

[10] D. P. Blinn, T. Henderson, and D. Kotz, "Analysis of a wi-fi hotspot network," in *Papers presented at the 2005 workshop on Wireless traffic measurements and modeling*, pp. 1–6, 2005.

[11] Wikipedia, "Erasure code." `https://en.wikipedia.org/wiki/Erasure_code`. Accessed: 2021-05-31.

[12] S. B. Wicker and V. K. Bhargava, *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[13] V. Roca, M. Watson, and A. C. Begen, "Forward Error Correction (FEC) Framework." RFC 6363, Oct. 2011.

[14] V. Roca and A. C. Begen, "Forward Error Correction (FEC) Framework Extension to Sliding Window Codes." RFC 8680, Jan. 2020.

[15] V. Roca, B. Teibi, C. Burdinat, T. Tran, and C. Thienot, "Less latency and better protection with al-fec sliding window codes: A robust multimedia cbr broadcast case study," in *2017 IEEE 13th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 1–8, IEEE, 2017.

[16] F. Michel and O. Bonaventure, "Improving quic,"

[17] N. Kuhn, E. Lochin, F. Michel, and M. Welzl, "Coding and congestion control in transport," Internet-Draft draft-irtf-nwcrg-coding-and-congestion-08, Internet Engineering Task Force, Apr. 2021. Work in Progress.

[18] C. Huitema, "The case for packet level fec," in *International Workshop on Protocols for High Speed Networks*, pp. 109–120, Springer, 1996.

[19] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture.," in *USENIX winter*, vol. 46, 1993.

[20] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th international conference on emerging networking experiments and technologies*, pp. 54–66, 2018.

[21] "Cilium." `https://github.com/cilium/cilium`. Accessed: 2021-02-24.

[22] Cilium, "What is ebpf? an introduction and deep dive into the ebpf technology." `https://ebpf.io/what-is-ebpf/`, 2021. Accessed: 2021:06:06.

[23] J. Corbet, "Compiling to bpf with gcc." `https://lwn.net/Articles/800606/`. Accessed: 2021-02-24.

[24] M. Rybczynska, "Bounded loops in bpf for the 5.3 kernel." `https://lwn.net/Articles/794934/`, 2019. Accessed: 2021:06:06.

[25] Q. Monnet, "ebpf updates #4: In-memory loads detection, debugging quic, local ci runs, mtu checks, but no pancakes." `https://ebpf.io/blog/ebpf-updates-2021-02`, 2021. Accessed: 2021:06:03.

[26] "bpf-helpers(7) — linux manual page." `https://man7.org/linux/man-pages/man7/bpf-helpers.7.html`, 2021. Accessed: 2021:06:06.

[27] A. Nakryiko, "Bpf ring buffer." `https://nakryiko.com/posts/bpf-ringbuf/`, 2021. Accessed: 2021:06:06.

[28] "Bpf compiler collection." `https://github.com/iovisor/bcc`. Accessed: 2021-02-24.

[29] "libbpf." `https://github.com/libbpf/libbpf`. Accessed: 2021-04-15.

[30] A. Nakryiko, "Bpf portability and co-re." `https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html`. Accessed: 2021:03:17.

[31] R. Baumann, S. Heimlicher, M. Strasser, and A. Weibel, "A survey on routing metrics," *TIK report*, vol. 262, pp. 1–53, 2007.

[32] Y. Desmouceaux, P. Pfister, J. Tollet, M. Townsley, and T. Clausen, "Srlb: The power of choices in load balancing with segment routing," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 2011–2016, IEEE, 2017.

[33] A. Viswanathan, E. C. Rosen, and R. Callon, "Multiprotocol Label Switching Architecture." RFC 3031, Jan. 2001.

[34] B. Thomas, L. Andersson, and I. Minei, "LDP Specification." RFC 5036, Oct. 2007.

[35] R. T. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification." RFC 2205, Sept. 1997.

[36] M. Welzl and M. Muhlhauser, "Scalability and quality of service: a trade-off?," *IEEE Communications Magazine*, vol. 41, no. 6, pp. 32–36, 2003.

[37] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, "Segment Routing Architecture." RFC 8402, July 2018.

[38] D. Lebrun and O. Bonaventure, "Implementing ipv6 segment routing in the linux kernel," in *Proceedings of the Applied Networking Research Workshop*, pp. 35–41, 2017.

[39] D. S. E. Deering and B. Hinden, "Internet Protocol, Version 6 (IPv6) Specification." RFC 8200, July 2017.

[40] D. Lebrun, "Srv6 - linux kernel implementation." `https://segment-routing.org/`. Accessed: 2021-05-31.

[41] D. Lebrun, M. Jadin, F. Clad, C. Filsfils, and O. Bonaventure, "Software resolved networks: Rethinking enterprise networks with ipv6 segment routing," in *Proceedings of the Symposium on SDN Research*, pp. 1–14, 2018.

[42] F. Duchene, M. Jadin, and O. Bonaventure, "Exploring various use cases for ipv6 segment routing," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pp. 129–131, 2018.

[43] F. Duchene, M. Jadin, and O. Bonaventure, "Exploring various use cases for ipv6 segment routing," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pp. 129–131, 2018.

[44] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira, "Avoiding traceroute anomalies with paris traceroute," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pp. 153–158, 2006.

[45] M. Xhonneux and O. Bonaventure, "Flexible failure detection and fast reroute using ebpf and srv6," in *2018 14th International Conference on Network and Service Management (CNSM)*, pp. 408–413, IEEE, 2018.

[46] M. Xhonneux, "ebpf updates #4: In-memory loads detection, debugging quic, local ci runs, mtu checks, but no pancakes." `https://github.com/Zashas/Thesis-SRv6-BPF/blob/e9f9624ade3f94eb22cbfaffa050506692eaad06/seg6-bpf-tests/tests_bpf.c#L65`, 2018. Accessed: 2021:06:05.

[47] F. Michel, Q. De Coninck, and O. Bonaventure, "Quic-fec: Bringing the benefits of forward erasure correction to quic," in *2019 IFIP Networking Conference (IFIP Networking)*, pp. 1–9, IEEE, 2019.

[48] M. Van Der Schaar, S. Krishnamachari, S. Choi, and X. Xu, "Adaptive cross-layer protection strategies for robust scalable video transmission over 802.11 wlans," *IEEE Journal on selected areas in communications*, vol. 21, no. 10, pp. 1752–1763, 2003.

[49] B. Liu, D. L. Goeckel, and D. Towsley, "Tcp-cognizant adaptive forward error correction in wireless networks," in *Global Telecommunications Conference, 2002. GLOBECOM'02. IEEE*, vol. 3, pp. 2128–2132, IEEE, 2002.

[50] L. Lopacinski, M. Brzozowski, and R. Kraemer, "Data link layer considerations for future 100 gbps terahertz band transceivers," *Wireless communications and mobile computing*, vol. 2017, 2017.

[51] L. Lopacinski, M. Brzozowski, and R. Kraemer, "A 100gbps data link layer with an adaptive algorithm for forward error correction," *IEICE Proceedings Series*, vol. 22, no. SESSION4-3, 2015.

[52] V. Roca and B. Teibi, "Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC) Schemes for FECFRAME." RFC 8681, Jan. 2020.

[53] C. Barakat and A. Al Fawal, "Analysis of link-level hybrid fec/arq-sr for wireless links and long-lived tcp traffic," *Performance Evaluation*, vol. 57, no. 4, pp. 453–476, 2004.

[54] A. Al Fawal and C. Barakat, "Simulation-based study of link-level hybrid fec/arq-sr for wireless links and long-lived tcp traffic," in *WiOpt'03: Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks*, pp. 8–pages, 2003.

[55] A. Cohen, D. Malak, V. B. Bracha, and M. Médard, "Adaptive causal network coding with feedback," *IEEE Transactions on Communications*, vol. 68, no. 7, pp. 4325–4341, 2020.

[56] A. Cohen, G. Thiran, V. B. Bracha, and M. Médard, "Adaptive causal network coding with feedback for multipath multi-hop communications," *IEEE Transactions on Communications*, 2020.

[57] F. Brockners, S. Bhandari, and T. Mizrahi, "Data Fields for In-situ OAM," Internet-Draft draft-ietf-ippm-ioam-data-12, Internet Engineering Task Force, Feb. 2021. Work in Progress.

[58] S. Amante, J. Rajahalme, B. E. Carpenter, and S. Jiang, "IPv6 Flow Label Specification." RFC 6437, Nov. 2011.

[59] F. Baker, D. L. Black, K. Nichols, and S. L. Blake, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers." RFC 2474, Dec. 1998.

[60] S. Floyd, D. K. K. Ramakrishnan, and D. L. Black, "The Addition of Explicit Congestion Notification (ECN) to IP." RFC 3168, Sept. 2001.

[61] L. Torvalds, "Linux kernel." `https://github.com/torvalds/linux`. Accessed: 2021-04-15.

[62] Q. De Coninck, F. Michel, M. Piraux, F. Rochet, T. Given-Wilson, A. Legay, O. Pereira, and O. Bonaventure, "Pluginizing quic," in *Proceedings of the ACM Special Interest Group on Data Communication*, pp. 59–74, 2019.

[63] M. Saito, M. Matsumoto, V. Roca, and E. Baccelli, "TinyMT32 Pseudorandom Number Generator (PRNG)." RFC 8682, Jan. 2020.

[64] "Ipmininet." `https://github.com/cnp3/ipmininet`. Accessed: 2021-05-16.

[65] B. Lantz, "Mininet & all." `https://http://mininet.org`. Accessed: 2021-05-16.

[66] M. S. Borella, D. Swider, S. Uludag, and G. B. Brewster, "Internet packet loss: Measurement and implications for end-to-end qos," in *Proceedings of the 1998 ICPP Workshop on Architectural and OS Support for Multimedia Applications Flexible Communication Systems. Wireless Networks and Mobile Computing (Cat. No. 98EX206)*, pp. 3–12, IEEE, 1998.

[67] E. O. Elliott, "Estimates of error rates for codes on burst-noise channels," *The Bell System Technical Journal*, vol. 42, no. 5, pp. 1977–1997, 1963.

[68] R. E. Kirk, "Experimental design," *Handbook of Psychology, Second Edition*, vol. 2, 2012.

[69] C. Paasch, R. Khalili, and O. Bonaventure, "On the benefits of applying experimental design to improve multipath tcp," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pp. 393–398, 2013.

[70] Y. Chen and T. Kunz, "Performance evaluation of iot protocols under a constrained wireless access network," in *2016 International Conference on Selected Topics in Mobile & Wireless Networking (MoWNeT)*, pp. 1–7, IEEE, 2016.

[71] E. L. Helvey, *Trafgen: An Efficient Approach to Statistically Accurate Artificial Network Traffic Generation*. PhD thesis, Ohio University, 1998.

[72] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-s—a publish/subscribe protocol for wireless sensor networks," in *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*, pp. 791–798, IEEE, 2008.

[73] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu, "iperf3, tool for active measurements of the maximum achievable bandwidth on ip networks," *URL: https://github.com/esnet/iperf*, 2012.

[74] P. Marti, J. M. Fuertes, G. Fohler, and K. Ramamritham, "Jitter compensation for real-time control systems," in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*, pp. 39–48, IEEE, 2001.

[75] M. Claypool and J. Tanner, "The effects of jitter on the peceptual quality of video," in *Proceedings of the seventh ACM international conference on Multimedia (Part 2)*, pp. 115–118, 1999.

[76] N. B. ?ational Laboratory, "iperf - the ultimate speed test tool for tcp, udp and sctp." `https://iperf.fr`. Accessed: 2021:05:13.

[77] C. M. Demichelis and P. Chimento, "IP Packet Delay Variation Metric for IP Performance Metrics (IPPM)." RFC 3393, Nov. 2002.

[78] C. Rezende, M. Almulla, and A. Boukerche, "The use of erasure coding for video streaming unicast over vehicular ad hoc networks," in *38th Annual IEEE Conference on Local Computer Networks*, pp. 715–718, IEEE, 2013.

[79] M. Allman, V. Paxson, W. Stevens, *et al.*, "Tcp congestion control," 1999.

[80] N. Cardwell, S. Savage, and T. Anderson, "Modeling the performance of short tcp connections," *Techical Report*, 1998.

[81] D. Thangavel, X. Ma, A. Valera, H.-X. Tan, and C. K.-Y. Tan, "Performance evaluation of mqtt and coap via a common middleware," in *2014 IEEE ninth international conference on intelligent sensors, sensor networks and information processing (ISSNIP)*, pp. 1–6, IEEE, 2014.

[82] A. Krylovskiy, "Mqtt benchmarking tool." `https://github.com/krylovsk/mqtt-benchmark`. Accessed: 2021:04:30.

[83] R. A. Light, "Mosquitto: server and client implementation of the mqtt protocol," *Journal of Open Source Software*, vol. 2, no. 13, p. 265, 2017.

[84] L. Navarre, "Ignore other encap_types load_netlink - pyroute2." `https://github.com/svinota/pyroute2/pull/760`, 2020. Accessed: 2021:06:02.

[85] L. Navarre, "Ipv6exthdrsegmentroutingtlv still based on draft and not rfc8754 - scapy." `https://github.com/secdev/scapy/issues/3117`, 2021. Accessed: 2021:06:02.

[86] S. Previdi, C. Filsfils, K. Raza, J. Leddy, B. Field, D. Voyer, daniel.bernier@bell.ca, S. Matsushima, I. Leung, J. Linkova, E. Aries, T. Kosugi, Éric Vyncke, D. Lebrun, D. Steinberg, and R. Raszuk, "IPv6 Segment Routing Header (SRH)," Internet-Draft draft-ietf-6man-segment-routing-header-06, Internet Engineering Task Force. Work in Progress.

Appendices

Appendix A

BPF

A.1 The ___sk_buff structure

```
/* user accessible mirror of in-kernel sk_buff.
 * new fields can only be added to the end of this structure
 */
struct __sk_buff {
  __u32 len;
  __u32 pkt_type;
  __u32 mark;
  __u32 queue_mapping;
  __u32 protocol;
  __u32 vlan_present;
  __u32 vlan_tci;
  __u32 vlan_proto;
  __u32 priority;
  __u32 ingress_ifindex;
  __u32 ifindex;
  __u32 tc_index;
  __u32 cb[5];
  __u32 hash;
  __u32 tc_classid;
  __u32 data;
  __u32 data_end;
  __u32 napi_id;

  /* Accessed by BPF_PROG_TYPE_sk_skb types from here to ... */
  __u32 family;
  __u32 remote_ip4; /* Stored in network byte order */
  __u32 local_ip4;  /* Stored in network byte order */
  __u32 remote_ip6[4];  /* Stored in network byte order */
  __u32 local_ip6[4]; /* Stored in network byte order */
  __u32 remote_port;  /* Stored in network byte order */
  __u32 local_port; /* stored in host byte order */
  /* ... here. */

  __u32 data_meta;
  __bpf_md_ptr(struct bpf_flow_keys *, flow_keys);
  __u64 tstamp;
  __u32 wire_len;
  __u32 gso_segs;
  __bpf_md_ptr(struct bpf_sock *, sk);
  __u32 gso_size;
```

```
41 };
```

Listing A.1: The `__sk_buff` structure is given as context for the majority of networking hooks.

A.2 BPF development toolkits

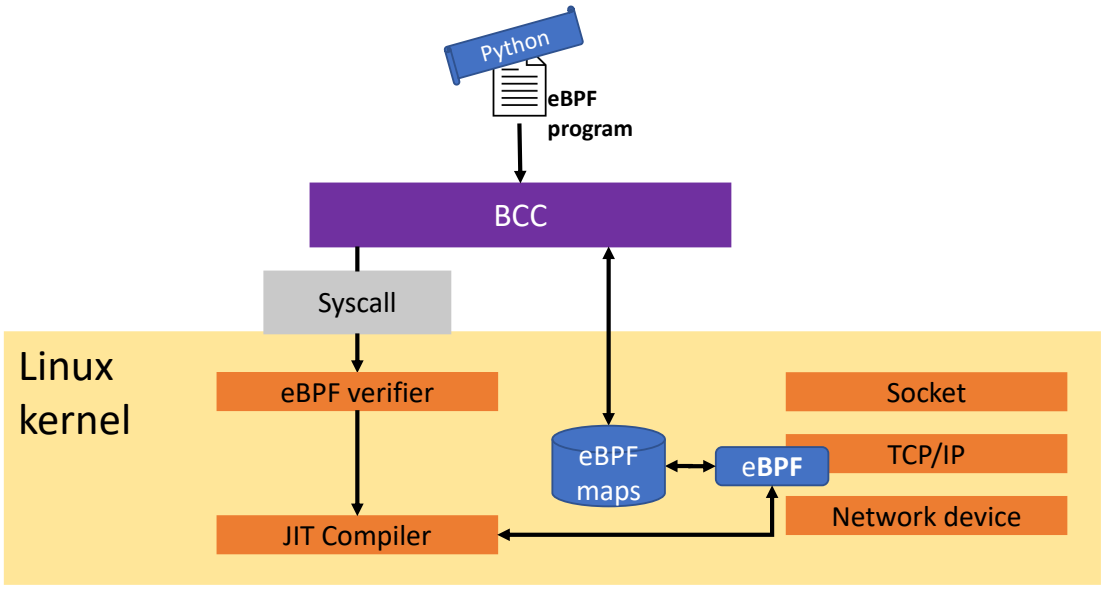


Figure A.1: `bcc` architecture where the eBPF hook is installed on the TCP/IP stack.

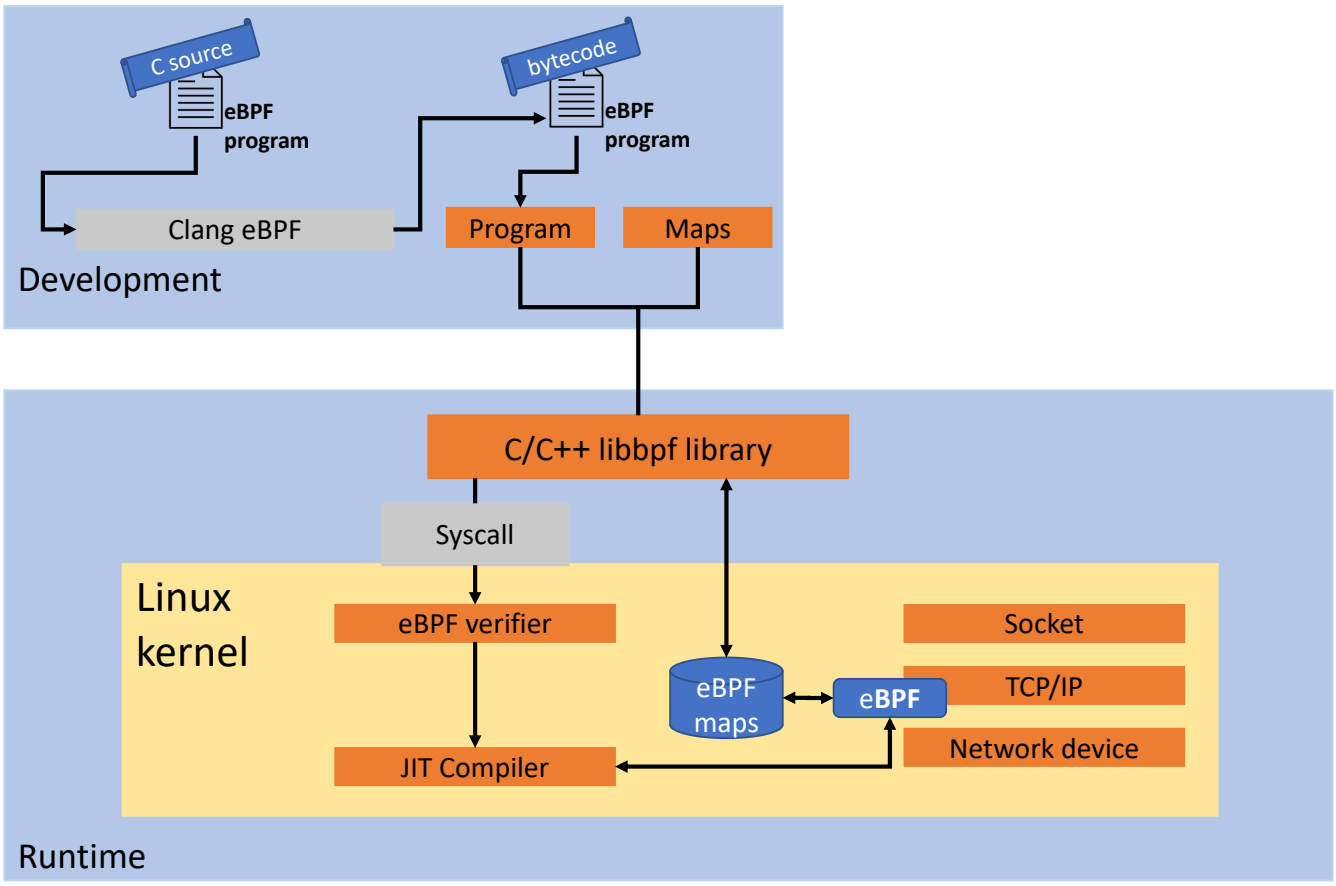


Figure A.2: `libbpf` architecture where the eBPF hook is installed on the TCP/IP stack.

Appendix B

IPv6 Headers

B.1 Segment Routing Header

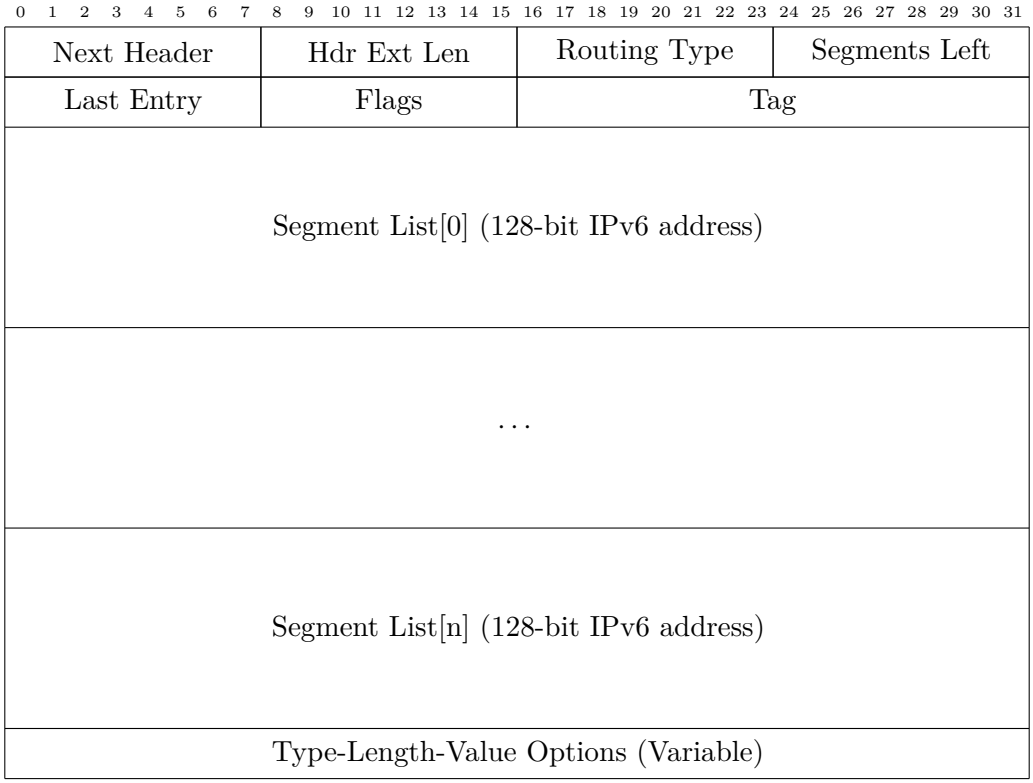


Figure B.1: IPv6 Segment Routing header.

B.2 IPv6 Standard Header

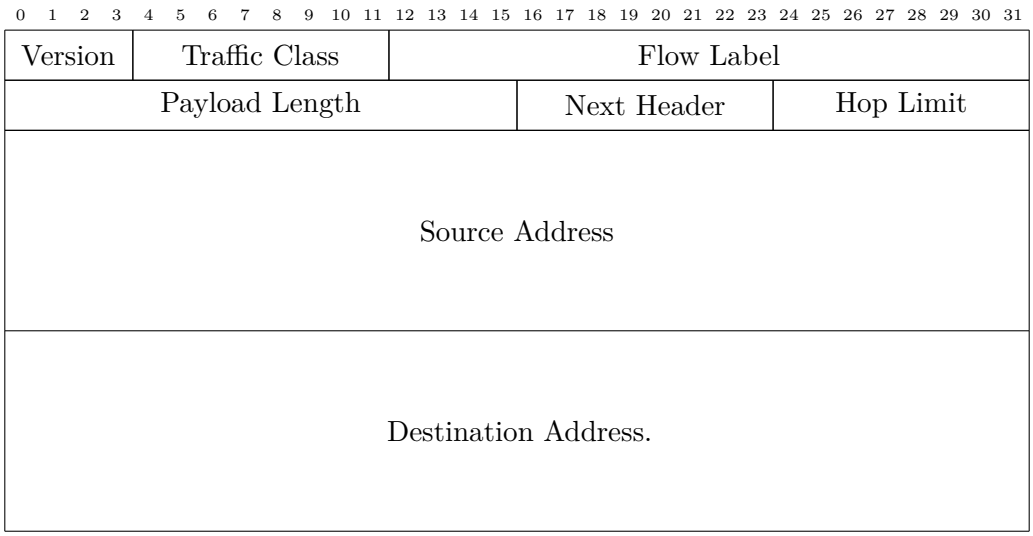


Figure B.2: IPv6 header

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