UCLouvain

École polytechnique de Louvain

Leveraging WebAssembly to create a customizable sleep proxy

Author: **Paul-Edouard Bertrand Van Ouytsel**
Supervisor: **Olivier Bonaventure**
Readers: **Maxime Piraux, Tom Barbette**
Academic year 2023–2024
Master [120] in Computer Science and Engineering

Abstract

Since their introduction in 1992, several designs of sleep proxies have been proposed in literature. A sleep proxy is a type of proxy capable of allowing devices to enter a low power consumption state, also known as sleep mode, while still maintaining a presence on the network.

The main issue to solve when designing a sleep proxy is to adapt it to the needs of all the hosts that rely on its services. Indeed, different hosts might have different behaviors and not use the same protocols. Most current sleep proxies offer limited customization and lack support for more modern network technologies such as IPv6.

In this master's thesis, we present SLOTH (SLeep Optimizer for Tired Hosts), a highly adaptive sleep proxy. Our solution is customizable thanks to a script system based on WebAssembly modules. Those scripts allow each host to formally define how the sleep proxy handles its traffic. We present an implementation of our sleep proxy written in Rust, using the WasmEdge runtime and its API. To demonstrate the capabilities of our sleep proxy, we also present a few example scripts made to handle specific protocols. We perform experiments to measure our sleep proxy's performance and overhead on a small experimental network.

Acknowledgements

First, I would like to thank my promoter, Professor Olivier Bonaventure, for suggesting this topic and choosing me to work on it. His knowledge of computer networks and protocols, added to his constructive and continuous feedback, has been of great help during the writing of this master's thesis.

I am also grateful to Maxime Piraux for the precious advice he has given me during our meetings with Professor Bonaventure and the rest of the research team. Because of him, I was able to find new leads to explore when I was lost.

Finally, I thank my friends and family for everything they did for me this academic year. All their support allowed me to keep moving forward and maintain a positive outlook. I would like to thank in particular Guillaume Steveny, Emilie Deprez and my brother, Charles-Henry Bertrand Van Ouytsel.

Contents

1 Introduction **5**
- 1.1 Context and motivation . . . 5
- 1.2 Structure of the thesis . . . 6

2 Problem analysis and background knowledge **7**
- 2.1 Defining the requirements of our solution . . . 7
- 2.2 Sleep proxy . . . 8
  - 2.2.1 Defining the notion of sleep proxy . . . 8
  - 2.2.2 Wake on LAN . . . 9
  - 2.2.3 Previous instances of sleep proxies . . . 9
- 2.3 Packet processing in litterature . . . 11
- 2.4 Our solution . . . 11

3 Implementation **12**
- 3.1 WebAssembly . . . 12
  - 3.1.1 Contents of a WebAssembly script . . . 13
  - 3.1.2 Compiling process . . . 13
  - 3.1.3 Semantic phases . . . 14
  - 3.1.4 Runtimes . . . 14
- 3.2 Design choices . . . 14
- 3.3 Overview of the solution . . . 15
- 3.4 Sleep proxy architecture . . . 17
  - 3.4.1 Responder . . . 18
  - 3.4.2 Packet interceptor . . . 19
  - 3.4.3 Timers . . . 19
- 3.5 Structuring the rules . . . 19
- 3.6 Writing scripts . . . 21
- 3.7 Communication between the script and the proxy . . . 22
  - 3.7.1 Proxy to script communication . . . 22
  - 3.7.2 Script to proxy communication . . . 22
- 3.8 Time sequence diagrams . . . 24

4 Example scripts **26**
4.1 UDP buffer script . . . 26
4.2 DHCP script . . . 27
4.3 ICMPv4 echo requests . . . 30
4.4 IGMP . . . 31
4.5 mDNS . . . 33
4.6 TCP control : maintaining a connection . . . 36
4.7 Test scripts . . . 38
4.7.1 Simple echo . . . 38
4.7.2 Simple rule check . . . 38
4.7.3 UDP checksum . . . 39
4.7.4 Substring check . . . 39
4.7.5 SHA-256 hash . . . 39
4.7.6 AES-256 . . . 39

5 Experiments **40**
5.1 Experimental setup . . . 40
5.2 Packet processing rate . . . 41
5.3 Effect of different factors on response speed . . . 44
5.3.1 Number of rules . . . 45
5.3.2 number of sleeping hosts . . . 46
5.3.3 Optimization . . . 46
5.4 Flood ping . . . 47
5.5 Performance profiling . . . 48
5.6 Memory usage . . . 50
5.7 Experiments conclusion . . . 51

6 Conclusion **52**
6.1 Further work . . . 52
6.2 Security considerations . . . 53
6.3 Support for TCP streams . . . 53

Chapter 1

Introduction

1.1 Context and motivation

In recent years, the reality of our current environmental crisis and the consequential need to evolve toward more sustainability have become increasingly apparent.

The 2023 report of the IPCC [1] urges policymakers to take immediate actions toward climate change to maximize our chances of limiting global warming to $1.5°C$ above pre-industrial levels. This number is a threshold above which disasters related to global warming are expected to become more frequent and extreme, possibly triggering a snowball effect leading to an even faster growth of temperature levels. Considering the impact of the energy sector on greenhouse gas emissions, finding ways to optimize or reduce electricity usage seems like a promising area of research. The Information and Communication Technologies (ICT) sector in particular represented about 4%[1] of the global energy usage in 2020 [2].

On one hand, Internet access providers have already managed to optimize the telecommunication sector's energy consumption significantly. In Belgium, for instance, the energy consumption of the telecommunication sector decreased by 11% between 2018 and 2021 [3]. On the other hand, a study from the ADEME and the ARCEP in France analyzed possible scenarios regarding the evolution of the digital environmental footprint in 2030 and 2050 [4]. This study found that, in three out of the four explored scenarios, the number of connected devices is expected to greatly increase in the coming years. Despite the progress already made in reducing energy usage in the ICT sector, we need to continue working in that direction to consider the potential rise in the number of hosts.

In practice, a part of the energy used by connected devices is wasted by idle hosts providing services that are expected to be available at all times. For instance,

[1]This number only includes the use stage of ICT equipment and not other factors such as their manufacturing or transport.

a server hosting a webpage with only a few visitors per day still needs to be continuously powered on to process incoming HTTP requests. Even when those devices do not need to serve any request, they are likely to receive useless network packets from outside. Indeed, even unused IP addresses receive traffic from the Internet, although no one is supposed to contact them. These can be the results of network scans, wrong device configurations, or cyberattacks, to name a few possibilities. This phenomenon is often called the Internet background radiation and has been studied several times in literature [5, 6, 7]. This flow of packets makes it harder for devices with an Internet presence to save energy, as they need to process this background radiation constantly. In addition, maintaining TCP connections, even when they are inactive, requires the host to be capable of answering packets at all times. Finally, several network protocols such as IGMP [8] or mDNS [9] require hosts to frequently process packets to maintain a presence on the local network.

These factors make it harder for hosts to save energy during idle periods, as they cannot enter sleep mode without losing network connectivity. This is the main motivation for this master's thesis: **we wish to design a solution allowing hosts to save energy by letting them enter a sleep mode when idle, taking their needs into account without compromising regular network operation.**

1.2 Structure of the thesis

The following chapters of this master's thesis go over the following points.

- Chapter 2 presents a more detailed analysis of the problem at hand, followed by a presentation of background knowledge and works related to this master's thesis.
- Chapter 3 presents an explanation of our solution and justifies our design choices
- Chapter 4 demonstrates the capabilities of our solution by describing a few example use cases.
- Chapter 5 describes the performance evaluation of our implementation.

Chapter 2

Problem analysis and background knowledge

Before presenting our solution, we provide a detailed analysis of the problem we want to solve. We start by defining the requirements our solution should fulfill. Then, we look at concepts explored in literature and pre-existing state-of-the-art solutions. Finally, we provide a high-level description of our solution based on the notions seen in the other sections.

2.1 Defining the requirements of our solution

The problem we try to solve can be defined as follows: We want hosts on a network to consume less electricity by letting them enter a low power consumption state (sleep mode) during idle periods. The primary role of our solution is to maintain the network state of sleeping hosts transparently. The rest of the network should be able to function as usual, even if one or multiple hosts are sleeping. Solving this problem is not trivial; not every possible solution has the same potential. For this reason, we defined a list of three important characteristics for our solution.

- Our solution must be **flexible**. In practice, every host on a network has different needs because of their roles and the protocols they use. If we want to save as much energy as possible, our solution must be able to adapt itself to those needs. Ideally, each host should be able to formally define a series of rules that our solution could apply to maintain their network state.

- It is important for our solution to be **scalable**. Depending on the type of network we are working with, the number of hosts present might greatly vary. We do not expect to see the same number of devices in a home network and a large company. Additionally, different hosts might receive different amounts

of traffic. Given that enough resources are allocated, we want our solution to be capable of handling large quantities of hosts and traffic.

- Finally, our solution should be **easily deployable**. Needing additional hardware or important topology changes might make network operators reluctant to deploy our solution in a real network.

2.2 Sleep proxy

The idea of handling the traffic of a sleeping host has already been explored in literature [10]. One of the solutions defined to this end is the notion of "sleep proxy". In this section, we start by defining the concept of a basic sleep proxy. Then, we explain the Wake on LAN technology and its use in the context of a sleep proxy. Finally, we present a few existing instances of sleep proxies in literature.

2.2.1 Defining the notion of sleep proxy

Figure 2.1: Simplified representation of a sleep proxy

In the context of this document, we use the term sleep proxy to refer to an entity capable of performing actions based on the network traffic normally directed toward one or multiple hosts currently in a sleeping state. A simplified representation of the integration of a sleep proxy in a network is shown in Figure 2.1. A host sends a signal to the sleep proxy before entering sleep mode to let it know that it must start intercepting traffic. The sleep proxy can be any kind of device with packet processing capabilities and the scope of the operations performed on the traffic is dependent on the implementation of the proxy. The traffic handling can go from simply maintaining TCP connections to reproducing the behavior of the hosts for application protocols. In addition to processing the traffic of its host(s), a sleep

proxy might include a "wake-up" mechanism. When a specific condition is met, a wake-up event is triggered, which causes the sleep proxy to activate the wake-up mechanism. When waking up for any reason, the host sends a signal to its sleep proxy indicating that it wishes for the device to stop handling its traffic.

2.2.2 Wake on LAN

One possible wake-up signal that can be used with a sleep proxy is Wake on LAN. Wake on LAN is a technology allowing a host in sleep mode to be woken up by a specific signal in a network packet [11]. More specifically, a host that has entered an ACPI sleep state [12] goes back to its operating state when receiving a *magic packet*. For the operation to work correctly, the host's NIC must support the technology, and the feature must be enabled in the computer's BIOS beforehand. If all the conditions are satisfied, the NIC can detect magic packets and generate an interrupt when it receives one. The magic packet itself consists of 6 bytes set to 255 followed by 16 repetitions of the MAC address of the host to be woken up. The protocol used to transmit the packet is not specified; any packet that can be routed toward the targeted host and that contains the sequence of the magic packet will be detected by the host's NIC.

2.2.3 Previous instances of sleep proxies

The notion of sleep proxy was first introduced in a paper by Kenneth J. Christensen et al. [13]. In 1992, the US Environmental Protection Agency announced the Energy Star program, an initiative to reduce the electricity consumption of office equipment by defining standardized energy savings methods. Computers that comply with the Energy Star program can automatically reduce their power consumption to low levels during idle periods. However, this generally means turning their CPUs off and thus losing presence on the network (interrupting the established TCP connections in the process, given the sleeping time is long enough). This last part made network administrators reluctant to leave the automatic sleep mode activated on those computers. To solve this issue, the writers of this paper created a proxy server that can maintain a minimal TCP connection by advertising a size zero window. More precisely, to enter sleep mode, a host advertises a zero window on all its open connections, asks the proxy to take care of its connections, and finally enters a low power consumption state. Due to the zero-size window, while the host sleeps, its peers do not send it useful information. They only send probes to make sure the connection is still alive. Answering those probes does not require prior knowledge regarding the connection, meaning a simple device connected to the same LAN could easily answer them. This device is the sleep proxy.

In the years following the publication of this paper, numerous searchers elaborated on the concept of sleep proxy. Generally, their new proxies fell into one of two categories. Software-based proxies, which do not require additional hardware and can run on a host or an existing peripheral (such as the NIC), and hardware-based proxies, which require additional physical equipment to function. Software-based proxies generally benefit from an easier deployment in an actual network, while hardware-based proxies allow for a higher degree of freedom.

A particularly influential paper describing a software-based proxy is described in a paper by Joshua Reich et al. [14]. The authors of this paper implemented a more complex sleep proxy and deployed it in an actual company network. This sleep proxy is connected to the same subnet as the potential sleeping hosts. When a host enters sleep mode, a background service called "sleep notifier" broadcasts a particular packet that includes the open ports of the host. After receiving this packet, the proxy starts spoofing ARP packets destined for the host in order to advertise its own MAC address. This allows it to intercept incoming TCP connexions and wake the sleeping host in case of a SYN packet for an open port. The wake-up process is done using the wake on lan Magic packets. To ensure regular network operation, the proxy also implements reactions to maintain the presence of the sleeping hosts.

Another example of a popular software-based sleep proxy is the Bonjour sleep proxy [15], a technology developed by Apple to allow devices providing Bonjour-advertised network services to enter sleep mode while still maintaining a presence on the network (Bonjour, also known as mDNS, is a DNS-like protocol allowing hosts on a network to advertise services using multicast, it is defined in RFC6762 [9]). To achieve this, hosts who want to enter sleep mode must contact a bonjour sleep proxy. Generally, this role is assigned to a device that always stays on and is always connected to the network. While the host sleeps, the sleep proxy advertises its services on the network. When, at some point, a client tries to access one of the sleeping host's services, the proxy can wake it up using a Wake-on-Lan magic packet.

Finally, Yuvraj Agarwal et al. presented an implementation of a hardware-based sleep proxy called Somniloquy [16]. The idea behind it is to embed a low-power CPU on a network interface to ensure presence on the network even when the PC is in sleep mode. A daemon runs on the host and waits for sleep events. If a sleep event is detected, the current network state is transferred to the flash memory of the secondary processor. This secondary processor is powered on at all times and can impersonate the sleeping host. A set of rules defined on the secondary processor helps to define when to respond incoming packets and when to wake the host up.

2.3 Packet processing in litterature

The idea of using custom rules to filter/process network traffic is not new and practical applications of this concept have previously been explored in literature.

For instance, retina [17] is a software framework that analyzes high-speed Internet traffic with no specialized hardware. Users are able to define filters to select the data they want to receive from the network being probed. The filters themselves are written in Rust and support analysis of packets on various layers, from raw packets to reassembled application streams. An optimized pipeline for packet filtering allows Retina to quickly discard unneeded data in order to maximize its throughput.

The Android Packet Filter (APF) [18] is a part of the Android operating system allowing the Android framework to dynamically create filtering rules on the hardware level. To be more specific, the APF consists of an interpreter that can execute APF rules on a hardware level for incoming packets and a tool to generate those rules. The benefit of this packet filter is that it offers the possibility to avoid processing useless packets by directly dropping them on the hardware level and provides the ability to make new rules and change them dynamically. This way, depending on the current state of the device on which the APF is running, rules can be changed to match the needs of the system.

2.4 Our solution

Based on our analysis of the literature and existing technologies, we decided to design a *customizable sleep proxy*. This solution works on a subscription basis. A host wanting to enter sleep mode sends a request message to the sleep proxy. This message contains a list of rules that define how to handle the host's network state. The rules themselves are defined as scripts. A script is a small program that can produce **reactions** based on various **events** (An example of an event is the interception of a packet destined for a sleeping host). A reaction can, for instance, consist of waking up the sleeping host or sending a packet on the network. After defining its rules, a host can tell the sleep proxy when it enters sleep mode and send a signal when it wakes up. This way, the sleep proxy knows when to handle a host's traffic and when to ignore it. This solution could be entirely software-based and run in a device already present in the network, such as a router.

Chapter 3

Implementation

This chapter presents SLOTH (SLeep Optimizer for Tired Hosts), a sleep proxy capable of leveraging Webassembly scripts to allow for highly personalized traffic management. We start with an explanation of the WebAssembly technology. Then, we explain and justify our design choices before presenting an overview of the solution. Next, we go over the architecture of the sleep proxy and show how the program works in practice. Afterward, we describe how rules and scripts are formally defined. Finally, we examine a time sequence diagram showing a typical sleep proxy use scenario.

In this chapter, we use the term *Somnolent host* to refer to any host interested in the services of the sleep proxy. Meanwhile, *Sleeping host* is exclusively used for hosts currently sleeping and subscribed to the sleep proxy.

3.1 WebAssembly

WebAssembly is a binary instruction format originally developed to embed programs from various source languages in web applications efficiently. More specifically, it is a virtual instruction set architecture for a WebAssembly virtual machine. It was initially announced in 2015 and is maintained by a community group of the World Wide Web consortium [19]. Despite being thought for web applications, the specifications of WebAssembly do not assume a specific environment for running programs. This means the technology can be used in other contexts and not necessarily inside a web browser. The context in which a module is executed is called an Embedder. It is the name given to the host environment in which the modules are executed using a WebAssembly runtime.

WebAssembly is based on a stack machine model, meaning that the functions work by pushing values on a stack and performing operations on those values.

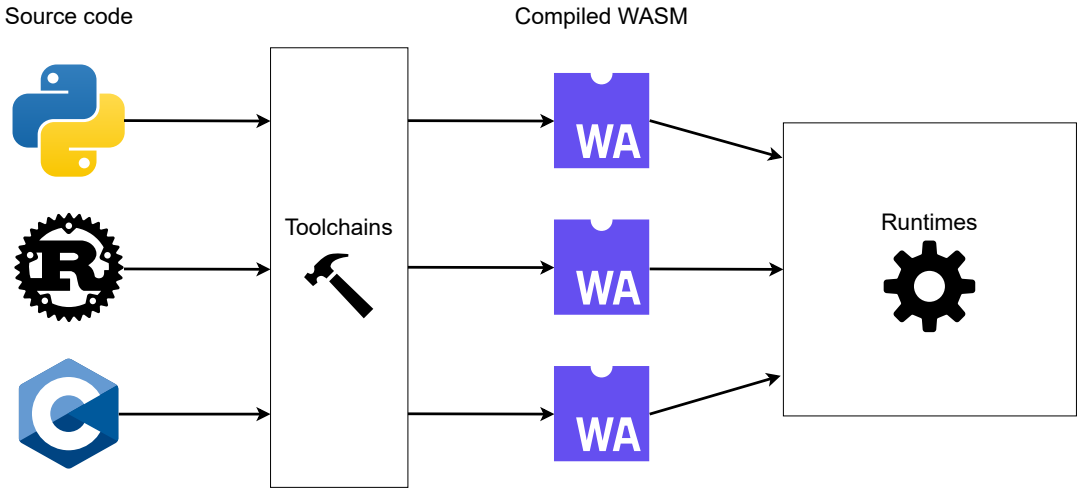

Figure 3.1: Simplified representation of the process of compiling code to WebAssembly and running it

Another important characteristic of WebAssembly is its sandboxed nature. By default, if not explicitly allowed, a script can only operate in a limited space isolated from the rest of the Embedder.

3.1.1 Contents of a WebAssembly script

A compiled WebAssembly binary is sometimes called a "module". It contains all definitions necessary for the script to run, such as its functions, data structures and variables. Modules can also import or export functions. Importing is a way for the module to access functions from the Embedder while exporting is a way to make functions available to the Embedder.

3.1.2 Compiling process

While technically possible, it is uncommon to write modules directly in WebAssembly. Instead, it is usually used as a "target." In the same way a program can be compiled for ARM32 or x86-64 architectures, it is possible to compile it for the WebAssembly virtual machine. For this purpose, multiple toolchains and compilers exist that can generate binaries from high-level languages such as C [20, 21], Rust [22] or Python [23]. The compilation process is not universal and highly dependent on the toolchain/compiler used. For this reason, the process itself is not detailed in this section.

3.1.3 Semantic phases

Before running a script, a WebAssembly runtime has to go through a series of steps to prepare the execution and ensure the script is acceptable. As of the writing of this manuscript, the WebAssembly specifications define three semantic phases for running scripts.

- Decoding: This phase represents the conversion of a WebAssembly binary into an internal representation used to process it further. This representation can be the abstract syntax defined in the WebAssembly specification or any other representation suitable for the Embedder.
- Validation: In this phase, each element of the syntax (variables, instruction, functions) is associated with a type. The WebAssembly specifications define several rules that the various types have to respect by themselves and in relation to each other. These rules are here to ensure that the module is sound. If no violations of the rules are found, the runtime moves on to the next step.
- Execution: The execution can be broken down into two sub-phases. First, the module is **instantiated**, all the data structures necessary to execute the module are created, and the Embedder creates the links with functions imported from the host. Then, an instantiated module can be **invoked** by its Embedder. That means calling one of its exported functions, giving it the right arguments, and getting its results.

3.1.4 Runtimes

The WebAssembly specifications provide general rules for verifying and interpreting syntax. However, deciding how programs are executed in practice is up to the runtime. It is responsible for converting the bytecode to machine code instructions that the Embedder can execute. The first widely deployed WebAssembly runtimes were made for javascript, allowing web browsers [24] to embed programs in web pages. The years following the release of WebAssembly have seen many new runtimes [25, 26, 27, 28] emerge for other architectures and contexts. Figure 3.1 shows the process of compiling a program in a source language into a WebAssembly binary and running it via a WebAssembly runtime.

3.2 Design choices

This section describes and justifies the design choices made to develop the sleep proxy in practice.

Regarding the custom rule system, we used a script system based on WebAssembly modules. This choice is motivated by several reasons.

First of all, WebAssembly can easily be embedded in many contexts. As seen in Section 3.1, the large variety of available runtimes makes WebAssembly a flexible choice. We decided to implement our solution using WasmEdge, as it is particularly lightweight and adapted to devices with restricted resources. This characteristic could allow us to run the sleep proxy on a small device connected to the network, such as a router. Additionally, Wasmedge has an extension system, meaning we could extend it to fit our use case better. Although we did not use this feature in the context of this master's thesis, we believe it could prove useful in future iterations of SLOTH.

Secondly, WebAssembly programs are executed in a sandboxed environment. That means the programmer controls the interaction they have with the outside world. This property limits the damage a malicious or poorly written script could do to the router/network.

Finally, WebAssembly is not tied to one specific programming language. As a binary instruction format, WebAssembly can technically be compiled from any source code, making script development easier and more accessible for developers. For instance, let's imagine that a developer wants to write a script allowing a host to keep its IPv4 address while in sleep mode. This developer can write the logic for handling the DHCP protocol in C and compile it using a toolchain such as Emscripten [21].

We chose to process packets at the lowest possible level because we would like the rules to be as customizable as possible. That means SLOTH handles *raw packets* (i.e., packets with all of the information from the datalink layer of the OSI model) and reacts based on each one of them individually. The proxy works on a "per packet" basis. This approach has the benefit of being simple but results in some limitations for our sleep proxy.

To make our sleep proxy easy to deploy, we wrote a software-based sleep proxy. It works as a daemon running in the background of a machine already in the network. This machine can be any device sharing a link with the somnolent hosts.

3.3 Overview of the solution

The program consists of two main threads: the event processor and the responder. The former handles any signal requiring a script to be executed, while the latter allows somnolent hosts to subscribe to the sleep proxy service. We separate the data plane from the control plane to further explain how the proxy works.

Figure 3.2 shows the data plane part of the sleep proxy; the orange line represents the packets intercepted by the sleep proxy. The program continuously listens on

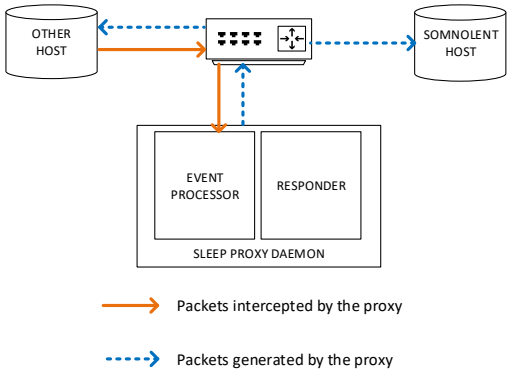

Figure 3.2: Simplified representation of the packets exchanged in the data plane part of the sleep proxy

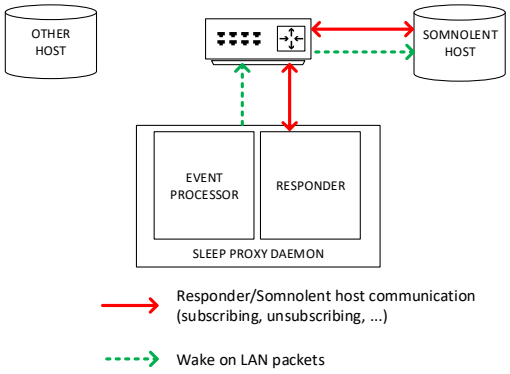

Figure 3.3: Simplified representation of the packets exchanged in the control plane part of the sleep proxy

a specific interface (passed as an argument when starting the sleep proxy) for all incoming packets. Those packets are handled by the event processor and sent down a pipeline that analyzes their headers. If the headers match against one or more rules, the event processor enters an execution phase where it executes all matching rules and collects their responses. (a list of all possible responses is given in Subsection 3.7.2) The blue dotted line represents the packets generated by the sleep proxy. The event processor can generate packets and send them to the network when necessary. We can, for instance, use this functionality to respond to a machine that tried to contact a sleeping host or to send packets to a somnolent host that has just woken up.

Figure 3.3 shows the control plane part of the sleep proxy. The red line shows the packets between the responder and the somnolent hosts. This is the part that handles host management. That includes letting hosts subscribe to the sleep proxy service, processing the rules they upload, and keeping track of which hosts entered sleep mode. While the responder is responsible for handling host requests, the state of the somnolent hosts resides in the event handler. The dotted green line shows the wake-up process. Sometimes, a rule might request the initiation of a wake-up sequence for its associated host. The event processor is responsible for handling this request by producing a wake-on-lan packet and sending it to the host that should be woken up.

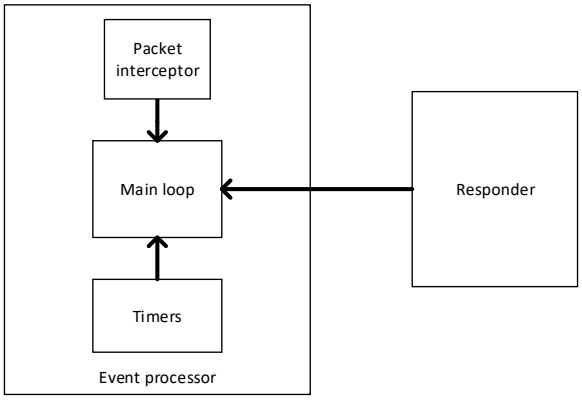

Figure 3.4: Global overview of the communications happening inside our sleep proxy

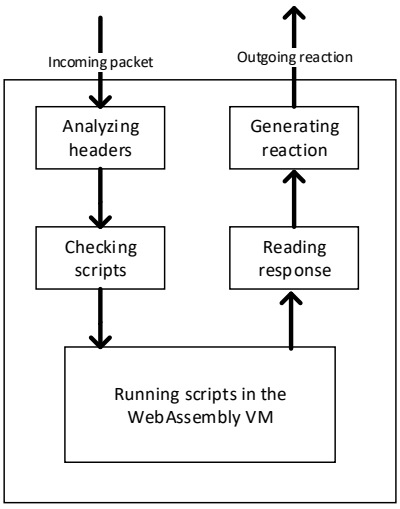

Figure 3.5: Simplified representation of the processing of a packet by our sleep proxy

3.4 Sleep proxy architecture

Figure 3.4 represents the interactions happening inside the sleep proxy. All communication inside the sleep proxy works via asynchronous message passing. The asynchronicity of the program is handled by Rust's Tokio [29] library. The event processor consists of a main loop continuously polling three message receivers.

- The first one corresponds to the packet interceptor. Its role is to intercept all incoming packets and send them towards the main loop for further analysis.
- The second one corresponds to the responder. Whenever a control message from a somnolent host is received from the responder, it parses the message and sends the parsed data to the main loop of the event processor.
- The last one corresponds to timers. Rules can set timers to perform actions at a fixed interval. This feature can notably be used to check a state regularly.

The main loop calls a handler function corresponding to the message type whenever one of those messages is received. In the following subsections, we explain in more detail how those messages are generated and how the event processor handles them.

3.4.1 Responder

The responder part of SLOTH runs over TCP and continuously waits for incoming connections from potential hosts. Communication is done using a simple protocol. Each packet consists of a one-byte opcode followed by the arguments necessary to perform the requested operation. After receiving and processing the message, the responder sends back a response code. The first byte indicates the status. The code 0 means that the operation was applied successfully. Any other code means an error occurred.

After successfully processing the request, the responder passes a message to the event processor so that it can update its information base.

Opcode 0 : add_host message

1 byte	6 byte	4 bytes	4 bytes
opcode (0)	MAC address	number of rules	json file size : n
n bytes	4 bytes	r_1 bytes	···
json file	rule 1 size : r_1	rule 1	···

This message is sent by a host to indicate that it wants to subscribe to the sleep proxy. The *MAC address* field represents the MAC address of the somnolent host. The json file is a configuration file giving additional information about the rules (as described in Section 3.5). After processing this message, the sleep proxy adds the host to its somnolent host list.

Opcode 1 : remove_host message

1 byte	6 byte
opcode (1)	MAC address

This message is sent by a host to indicate that it wants to unsubscribe from the sleep proxy. The *MAC address* field represents the MAC address of the somnolent host. After processing this message, the sleep proxy removes the host from its somnolent host list.

Opcode 2 : start_sleeping message

1 byte	6 bytes
opcode (2)	MAC address

This message is sent by a host to indicate that it is going to sleep. The *MAC address* represents the MAC address of the somnolent host. After processing this message, SLOTH marks the host as sleeping and starts to intercept its traffic.

3.4.2 Packet interceptor

The packet interceptor continuously listens on a specific interface, waiting for packets to arrive. Whenever a packet is received, it is directly sent to the main loop of the event handler. The process of handling a message from the packet interceptor is shown in Figure 3.5. First of all, the event processor analyzes the headers of the received packet to get information such as the protocols involved, the source address and the destination. It then uses these characteristics to determine which rules apply to the packet. If one or more rules is/are found, the event processor executes them. In practice, it means to execute their packet processing function with the packet as an argument via the Wasmedge API. Once the function is done running, the response is read. A reaction (as defined in Subsection 3.7.2) is then generated from the response. The program eventually interprets the obtained reaction.

3.4.3 Timers

Our sleep proxy allows us to create timers for rules that need to perform an action frequently. A timer corresponds to a task that sends messages at a fixed interval to the main loop. Those messages contain an identifier that allows the main thread to identify which rule is concerned by the message. Handling a timer message is similar to handling a packet from the message interceptor, except it exactly triggers one rule. The format of the reactions is the same as the one shown in Subsection 3.7.2.

3.5 Structuring the rules

When sending their scripts to the sleep proxy, hosts must also include a JSON file containing information about the rules. This file lets our sleep proxy know the layers at which each rule operates. Practically, the JSON file consists of a list of

```
[
    { "UDP" : { "port" : 5100, "raw" : true, "interval" : 20,
    "addresses" : ["192.168.1.1/32"]}},
    { "IPv4" : {"addresses" : ["192.168.1.1/32"]}}
]

```

Listing 1: example JSON file

rule objects. A rule object is defined by its key, corresponding to the layer the rule operates on, and a set of characteristics. Currently, the list of supported layers is as follows:

- IP: Stands for the Internet Protocol. A rule with this associated protocol processes both IPv4 and IPv6 packets.
- IPv4: Stands for version 4 of the Internet protocol.
- IPv6: Stands for version 6 of the Internet protocol.
- ICMP: Stands for the Internet Control Message Protocol.
- IGMP: Stands for the Internet Group Management Protocol.
- UDP: Stands for the User Datagram Protocol. This layer accepts an additional "port" argument to only process packets with a specific destination port.
- TCP: Stands for the Transmission Control Protocol. This layer accepts an additional "port" argument to only process packets with a specific destination port.

In addition to the layer, a rule object defines a set of characteristics via a JSON object. The keys associated with the characteristics are given in the following list.

- Addresses: A list representing the addresses the rule listens to. The addresses are encoded as strings representing prefixes. If a packet's destination address corresponds to any of the prefixes a rule is listening to; the packet gets processed by the rule. This characteristic is mandatory.
- Raw: A boolean indicating if the rule should process raw packets. If set to true, any packet given to the script as an argument is passed as a raw packet. This characteristic is not mandatory and defaults to false if not indicated.

- Interval: An unsigned 32-bit integer representing a time interval. If present in the characteristics, the sleep proxy creates a recurrent timer that calls a specific script function at a fixed interval. This fixed interval corresponds to the integer in seconds. This characteristic is not mandatory and has no default value (no timer is created if the interval key is not present).

Listing 1 shows an example of a JSON configuration file. In this case, we can see that the host subscribing to the sleep proxy defines two rules. The first is listening to UDP packets with destination port 5100 and destination address 192.168.1.1. This rule processes raw packets and sets up a timer to perform an action every 20 seconds. The second rule listens to all IPv4 packets directed towards IP address 192.168.1.1 and processes traffic on the network layer.

3.6 Writing scripts

SLOTH uses scripts compiled into WebAssembly to determine the correct reaction to an event. Those scripts can be written in any language that supports WebAssembly as a target. For instance, C/C++ code can be compiled to the WebAssembly instruction set by using the Emscripten compiler toolchain [21] or a specific flag in Clang [30]. Other languages, such as Rust, support the .wasm format as a target with their default compiler.

To have a valid script, the following functions need to be implemented.

- ***void allocate(u32)**: A function that takes an unsigned 32-bit integer and returns a pointer to a block of this size in the VM's memory.
- **void deallocate(*void)**: A function that takes a pointer of some memory block in the VM allocated via the allocate function and frees it.
- ***char process_packet(*char)**: A function that takes a message from the proxy as a pointer to an array of bytes and returns a pointer to an array of bytes representing the answer of the script. See Section 3.7.2 for more information about the message format.

Optionally, three additional functions can be implemented.

- **void initialize()**: A function that is called once when the script is received by the sleep proxy. It should be used to initialize any data structure needed in calls to the several functions.
- ***char on_wakeup()** is called whenever the host associated with the rule wakes up. The returned value is similar to the process_packet function in that it returns a message to be interpreted by the sleep proxy.

- ***char on_timer()** is called when the timer associated with the rule has expired. Similarly to the process_packet and on_wakeup function, the return value is a pointer to an array of bytes representing the response of the function.

3.7 Communication between the script and the proxy

Using the Wasmedge API, our sleep proxy is able to pass arguments to scripts and get return values from them once they finish executing. However, by default, WebAssembly only support a few primitive types [31].

To communicate arrays and more complex data structures, we had to define a protocol allowing communication from the sleep proxy to the scripts and the other way around.

3.7.1 Proxy to script communication

Proxy-to-script communication is necessary to communicate a packet when we call the process_packet function. The format of the message consists of an unsigned 32-bit integer representing the size of the packet, followed by the bytes of the packet itself. The sending process is done by first calling the *allocate* function of the script to obtain a pointer to a memory zone inside of the WebAssembly VM. The message is directly written to the corresponding memory zone. If the rule has the *raw* attribute, our proxy sends a raw packet, otherwise, a network packet is sent. We use the term raw packet to describe a packet containing all the information from layers 2 and upwards in the OSI model. Similarly, we consider a network packet to be a packet containing all the information from layer 3 and upwards.

3.7.2 Script to proxy communication

Script-to-proxy communication is necessary for the sleep proxy to return a reaction after running the process_packet, on_wakeup or on_timer function. The return value of these function is always a pointer to a memory address in the WebAssembly VM. The data at this memory address is an array of bytes representing a message. Messages are composed of an opcode on a single byte, possibly followed by additional data. In this subsection, we describe the possible messages the script can return and the way our sleep proxy interprets them.

Opcode 0: Ignore

1 byte
opcode (0)

A message indicating that the packet can simply be ignored, and no further action is required. No extra data is provided with this opcode. This message type is necessary as the event processor always expects an answer from the scripts. It should be used when a script wants to discard a packet or when it is not ready to produce a reaction yet (an example of this would be a script that accumulates packets in a buffer and waits until the buffer is full to produce a meaningful reaction)

Opcode 1: Wake up

1 byte
opcode (1)

A message indicating that the sleeping host associated to this rule should be woken up. After getting this message, the event processor produces a Wake-on-LAN magic packet to be sent to the host. No extra data is provided with this opcode.

Opcode 2: Respond

1 byte	4 bytes	n bytes
opcode (2)	packet size : n	packet

A message indicating that a packet should be sent. After getting this message, the event processor crafts a packet corresponding to the data present in the message and sends it on the network.

Opcode 3: Respond Multiple

1 byte	4 bytes	4 bytes	n_1 bytes
opcode (3)	Number of packets	packet 1 size : n_1	packet 1
...			

A message indicating that multiple packets should be sent. After getting this message, the event processor crafts all the packet corresponding to the data present in the message and sends them on the network.

3.8 Time sequence diagrams

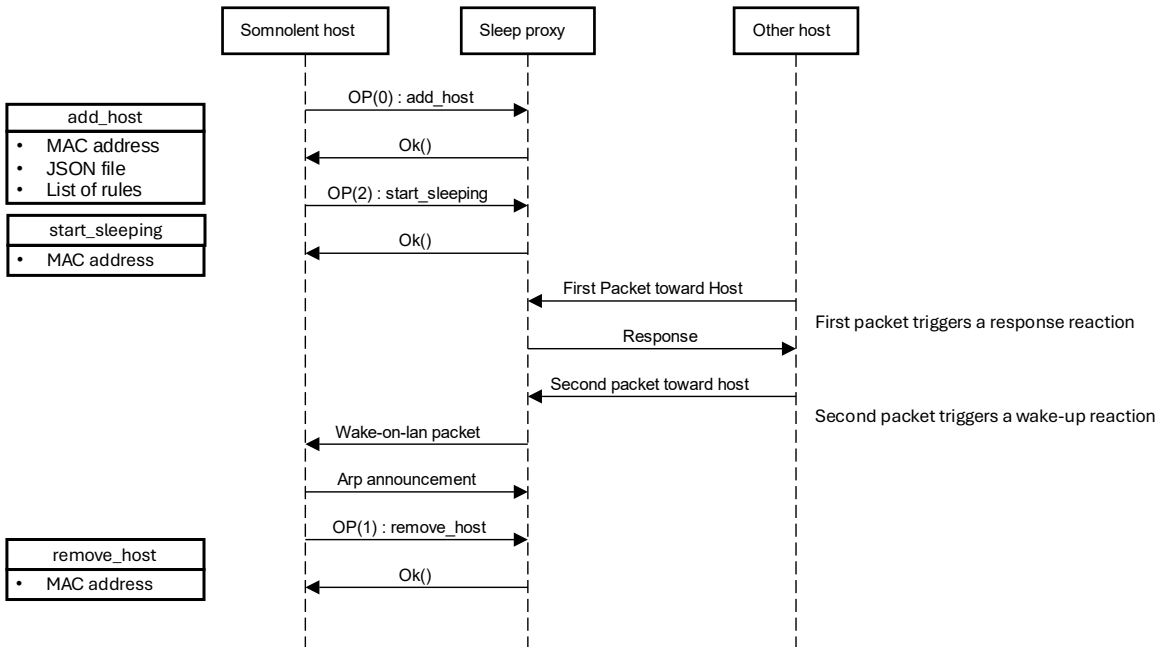

Figure 3.6: Time sequence diagram of a typical use scenario of the sleep proxy

This section features a time sequence diagram, shown in Figure 3.6, to describe a typical usage scenario of the sleep proxy.

At first, a host starts by subscribing to SLOTH by sending an add_host packet. This packet contains the MAC address corresponding to the somnolent host's network interface, followed by a JSON file describing each rule's properties and a list of WebAssembly scripts defining the rules' behavior. After receiving the packet, our sleep proxy creates an entry for this host and acknowledges the host's subscription. When the host is ready to sleep, it sends a start_sleeping packet to the proxy and waits for its answer. After getting an acknowledgment from the sleep proxy, the somnolent host enters sleep mode.

At some point, while the sleeping host is still in power-saving mode, another host tries to contact it. The other host starts by sending a first packet, which the sleep proxy intercepts. The event processor finds a rule that matches the packet's headers and runs the process_packet function of the corresponding script. The reaction generated by the script tells the sleep proxy to create a message to respond to the other host. After some time, the other host sends another packet, and the process repeats. This time, however, the reaction generated by the script tells the sleep proxy to wake up the sleeping host. The sleep proxy thus sends a wake-on-lan packet to the sleeping host. Now, if the host does not need the sleep proxy's services anymore, it can just send it a remove_host packet to be removed from the list of hosts.

Chapter 4

Example scripts

This chapter shows a few scripts we designed to demonstrate the sleep proxy's capacities. We start by describing scripts for dealing with UDP-based protocols. We then explain how the proxy can perform control operations on TCP connections. We end the chapter by describing the test scripts used as benchmarks in the rest of the manuscript. All the scripts were written in C and compiled using the wasi-sdk toolchain [20].

In the explanation of every individual script, we define the part of the sleeping host's state transferred to the sleep proxy, the way it might or might not evolve during the execution of the script , and how we make those potential changes known to the host when it wakes up. We also explain the logic for sending and processing packets while the script is active.

4.1 UDP buffer script

In some cases, to stay asleep for longer, a host might not want to be woken up every time a useful packet arrives on the sleep proxy. Instead, it can be more energy-efficient to process packets in batches. The purpose of the *UDP buffer script* is to accumulate packets up to a certain threshold and only wake up the sleeping host when that number of packets is reached.

Regarding the state transfer, the sleeping host needs to provide the sleep proxy with a threshold and potentially an IP address (or a range of IPs) and a UDP port (to filter specific packets). The only part of the proxy state that changes during execution is the packet buffer. When waking up, the sleep proxy transfers all packets received to the host.

The pseudocode of this script is shown in Listing 2. We can see that, during initialization, the script creates a new list to store packets. Whenever a packet is received, it is added to the list. The script returns a wake-up signal if the list's

length reaches a predefined threshold. Once the sleeping host has woken up, the *on_wakeup* function is triggered, and all the buffered packets are sent to the host.

```
1  THRESHOLD = N;
2  packetList;
3
4  initialize(){
5   packetList = new_list();
6  }
7
8  on_wakeup(){
9   return send_multiple(packetList);
10 }
11
12 process_packet(data){
13
14  packetSize = get_packet_size(data);
15  packet = copy_packet(data, packetSize);
16
17  packetList.add(packet);
18
19  if (packetList.len() > THRESHOLD){
20  return wakeup_signal();
21  }
22  else {
23  return ignore_signal();
24  }
25 }
```

Listing 2: Pseudocode for the UDP buffer script

4.2 DHCP script

The Dynamic host configuration protocol (DHCP) is a UDP-based protocol built on top of BOOTP [32], allowing hosts to ask one or multiple designated servers (DHCP servers) for IP addresses. The current version of the protocol is defined in RFC2131 [33]. When using DHCP, the server might decide not to give a host a permanent IP address and instead give it a temporary lease. This lease must be renewed before it expires, or the host might lose its IP address. For this reason, it is interesting to create a DHCP script that can automatically renew the lease by itself, as it can allow a host to stay asleep for longer. This section describes and discusses the choices made when designing this script.

The part of the state transferred from the sleeping host to the proxy consists of the IP address to preserve, the current timer on the lease and the bytes of a lease renewal request.

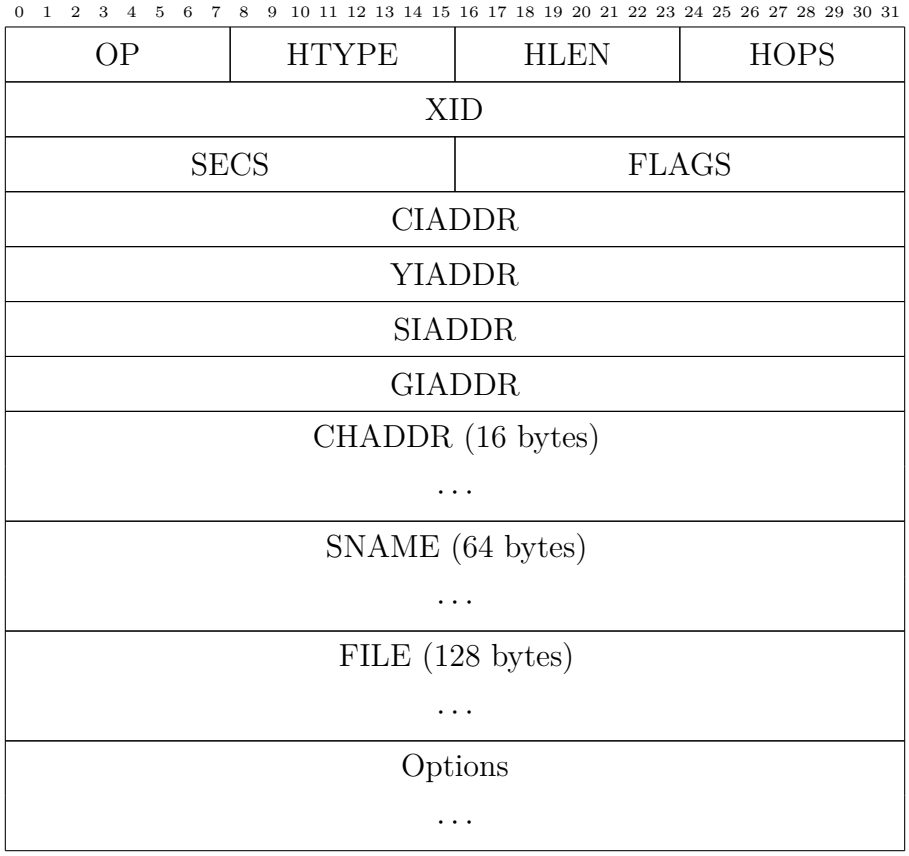

Figure 4.1: Representation of a DHCP packet

Figure 4.1 represents a DHCP packet. In practice, only a portion of the fields in these packets are of interest to us. OP represents if the message is from a DHCP client to a server (value of 1) or from a server to a client (value of 2). Since this script is made for clients and DHCP clients are not supposed to process packets from other clients, all messages with an OP value of 1 are automatically ignored. YIADDR represents the address of the client implied in the current transaction. The option field contains various additional information. Two options that we are interested in are the packet type and the duration of the lease. Additionally, the "end" option determines when the list of options ends. Other options can provide additional information about the network's state. However, we do not need to keep it in memory, as the sleeping host automatically sends a DHCP request when waking up. That means the host can acquire the additional information by itself without the help of the sleep proxy.

RFC2131 [33] defines eight different states a DHCP client can be in: INIT, INIT-REBOOT, REBOOTING, SELECTING, REQUESTING, REBINDING, BOUND and RENEWING. The script is made to help a host keep its lease, we therefore assume it starts in the "BOUND" state, which corresponds to the situation where the host owns the IP address and does not currently need to renew it. The only other states the host can transition to from this one are itself or the "RENEW" state. Once a timer has expired, the host sends a request to the server and reaches the "RENEW" state. In this state, the host might receive a confirmation (DHCPACK), a denial (DHCPNAK), or nothing. In the first case, we return to the "BOUND" state and continue the operation. In the case of denial or if no packets were received before the second timer, the sleeping host should be woken up to deal with the situation and be made aware of possible network modifications.

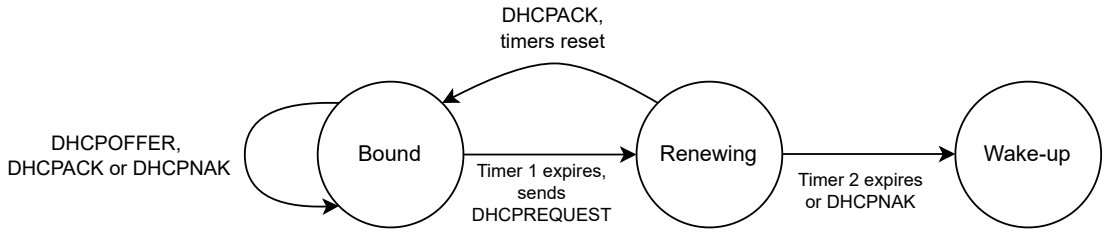

Figure 4.2: DHCP State-transition diagram adapted for a sleep-proxy script

In practice, the script's operation can be divided into two parts. First, the script needs to regularly check how long the lease is still valid. RFC2131[33] indicates that a host should have two "time thresholds" based on the duration for which

the lease is valid. A new DHCP request should be sent whenever one of those thresholds is reached. We achieve this behavior by using the timer option of the sleep proxy when defining the script to ask for a function to be executed regularly. This function checks the current state of the lease and, if one of the two "time thresholds" has been reached, sends a renewal request. Secondly, the script should be able to parse incoming DHCP packets and update its state accordingly. A client might receive only three message types from a server: DHCPOFFER, DHCPACK, and DHCPNAK. Figure 4.2 shows a simplified representation of the DHCP state diagram with the appropriate reactions and transitions depending on the situation. To simplify it, it considers that the host cannot reach all the states of the diagram.

Regarding the content of the lease renewals, they are the same as the ones the sleeping host sends by itself. The script also sends any additional options the sleeping host might send in its packets.

4.3 ICMPv4 echo requests

ICMPv4 is a protocol used to provide feedback and control information about the IPv4 protocol. RFC792[34] defines several ICMPv4 message types, each with their purpose. Notably, message types 0 and 8 correspond respectively to echo reply and request, also known as "ping". Echo requests are often used to determine if a host is reachable. Henceforth, it is interesting for the proxy to be able to respond to those requests with an appropriate reply to maintain the presence of the sleeping host on the network.

The part of the sleeping host's state sent to the sleep proxy is the address/range of addresses for which the script should respond to echo requests and a potential list of authorized IPs.

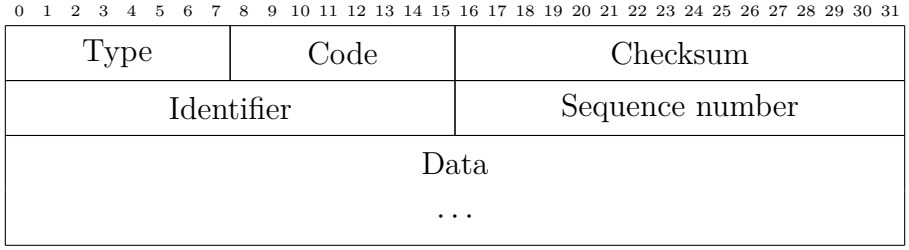

Figure 4.3: Representation of an ICMPv4 packet

Figure 4.3 shows a representation of an ICMPv4 packet. Our ICMPv4 script first looks at the "type" byte of the ICMPv4 packet and discards it if its value is not 8, as we only want to respond to echo requests. If the packet is indeed an echo request, the script crafts an echo reply with the same identifier, sequence number,

and data as the request. The reply's checksum is computed, and the packet is sent to the client requesting an echo.

We can improve the script by performing additional filtering on the echo requests. For instance, it is a well known fact that echo requests are regularly used to perform network scans and infer the topology of a network [35]. We can add a list of authorized IP ranges and verify that the distant host indeed belongs to one of them before responding to an echo request.

4.4 IGMP

The Internet group management protocol (IGMP) is the protocol used by IPv4 systems to communicate information about the state of host and router registration to multicast groups. RFC3376 [8] defines this protocol's third and latest version. A machine operating with IGMP can be a group member, a multicast router, or both. A group member is any host that wishes to receive packets from one or multiple multicast addresses, while a multicast router manages the states of its group members and forwards the required traffic to the interested hosts. In this section, we'll assume a case where the sleeping host is a simple group member.

Practically, a group member's IGMP state can be summarized as a set of tuples representing its subscriptions. The tuples consist of a multicast address, a filter mode, and a list of sources. The multicast address is the address the host wants to listen to; the filter mode is either INCLUDE or EXCLUDE. A tuple with an INCLUDE filter mode means the host only wants to receive packets from sources included in the list of source addresses. Similarly, the EXCLUDE filter mode indicates that the host wants to receive all the packets sent toward that multicast address except if they come from one of the sources included in the list. To facilitate the process of answering queries, those tuples are stored as Group Records, which can be sent "as is" in a group membership report. The format of the record is shown in Figure 4.4. We only use the record types "MODE_IS_INCLUDE" and "MODE_IS_EXCLUDE" (which indicate an "include" and "exclude" filter mode respectively) as they are the only modes needed to maintain the state (the other modes are used to update the state). It is also important to note that we always set Aux Data Len as 0, as the RFC document recommends.

The script's objective is to give the rest of the network the illusion that the sleeping host is still active. In this case, it means ensuring that the host remains a member of the multicast groups it subscribed to. As a group member, the only packets the host should send are membership reports. Those packets are sent either when the host wants to voluntarily change its state or in response to a query from a multicast router. The sleep proxy has no interest in modifying the state of the sleeping host, we thus only focus on the latter case.

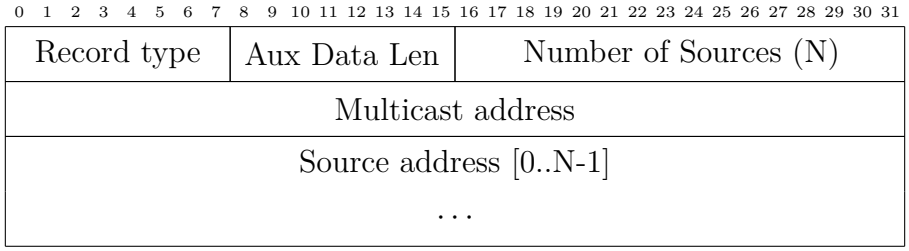

Figure 4.4: Representation of an IGMP record

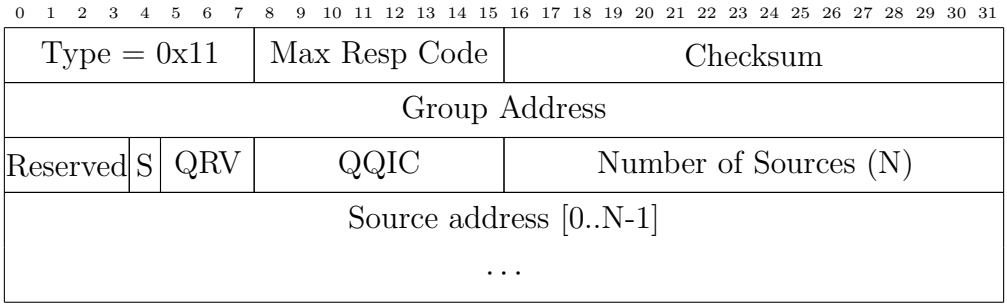

Figure 4.5: Representation of an IGMP membership query

A representation of a query is shown in Figure 4.5. When parsing a query, the fields of interest are the max resp code, the group address, and the list of sources. The S, QRV, and QQIC fields only update the state of multicast routers that receive them. The max resp code indicates the maximum time a host might wait before sending a reply. The group address indicates which multicast address is concerned by the request. Finally, the list of sources is a potentially empty list that can be used to further specify the query by asking hosts if they are interested in packets coming from any of the addresses present in the list. The group address and source address values indicate what kind of query was sent.

- If the group address is 0.0.0.0, it means the query is a *General Query*. A group member should respond to such a query with all its records.
- If the group address is not 0.0.0.0 and the number of source addresses is 0, we have a *Group Specific Query*. Those queries are sent by a multicast router to determine the reception state of the hosts regarding a specific address. If the sleeping host is indeed subscribed to the group, such a query is answered with the specific record related to that group.
- If the group address is not 0.0.0.0 and the number of source addresses is non-zero, we have a *Group-and-Source Specific query*. A multicast router

sends this type of query to ask if there are hosts interested in the traffic sent by a specific source in a specific multicast group. If the sleeping host is subscribed to the corresponding traffic, the record of the corresponding multicast group is sent to the router.

A query is never instantaneously answered to avoid the "ack implosion" problem. When several hosts sharing a link are subscribed to the same multicast address, there is an increased chance of packet collision if they all try to acknowledge a multicast packet at the same time. Instead, a timer is started based on a random value between zero and the Max Resp code. If another query is received during that period, a series of rules are checked to see if another timer should be scheduled or if the answers can be regrouped in a single one. A detailed explanation of those rules can be found in section 5.2 of RFC3376.

Once a timer expires, a membership report with all the requested records associated with it is sent. Figure 4.6 shows this packet's format.

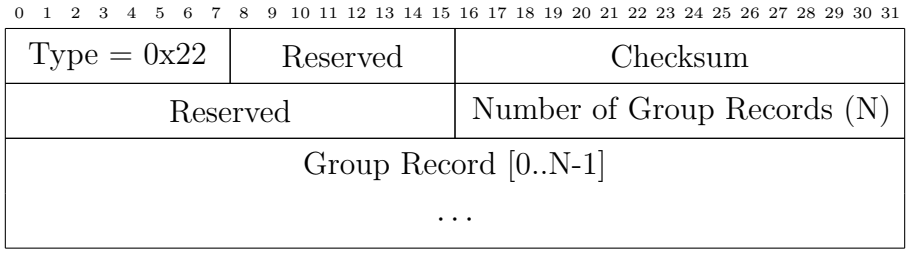

Figure 4.6: Membership report packet

Due to the way IGMP membership reports are supposed to be sent, we cannot implement this script in the current version of SLOTH. Indeed, we do not have a mechanism to precisely delay sending packets. We could achieve a similar result by setting a low-value reoccurring timer. However, that would imply performing many useless calls to the script. Considering how inefficient it would be, we decided to consider this script only on a conceptual level and not provide an implementation.

4.5 mDNS

Multicast DNS, also known as mDNS or Bonjour is a DNS-like protocol used to resolve hostnames on a local network. This protocol was defined in RFC6762[9] and is part of the technologies that make up ZeroConf[36]. In addition to its use as an alternative to DNS for local resources, the protocol can also perform DNS-based service discovery, as described in RFC6763[37].

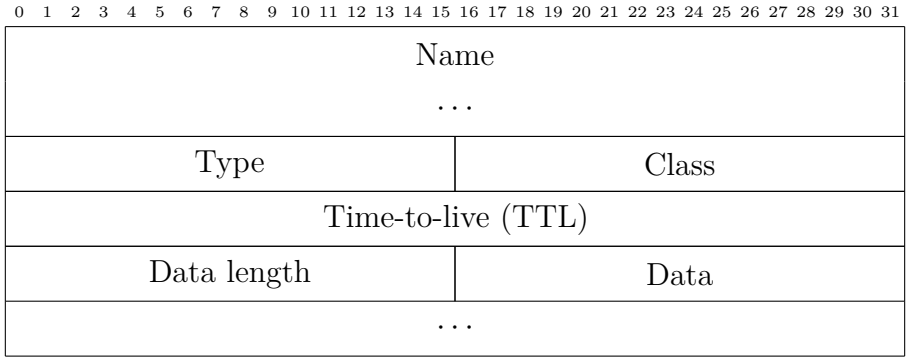

Figure 4.7: Representation of an mDNS resource record

In the context of this script, the state to be transferred from the sleeping host is the set of all the mDNS resources records (RR) that the sleeping host owns. An mDNS resource record follows the same format as a regular DNS resource, meaning it contains a name, a class, a type, a time-to-live (TTL), and the data representing the resource. The time-to-live field here represents the one the sleep proxy should advertise when answering a query for this resource. A representation of a resource record is shown in Figure 4.7. Additionally, every single one of those records should have one more field called "shared." In some cases, an mDNS resource might be shared across multiple hosts on a network; the sleep proxy needs to be aware of the shared resources to operate correctly. We explain later in this section why it is essential for the script to know which resources are shared.

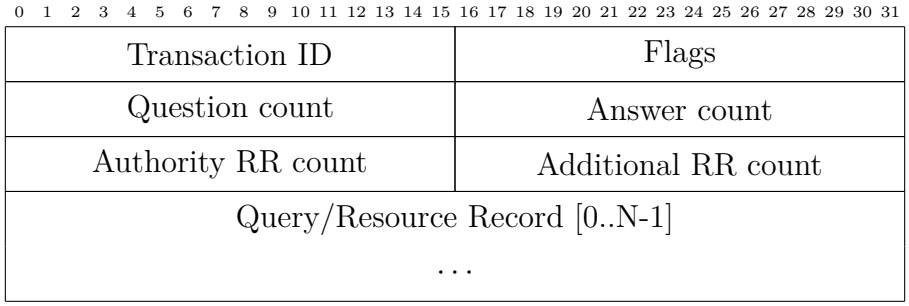

Figure 4.8: Representation of an mDNS packet

A representation of a packet is shown in Figure 4.8. The format is similar to a regular DNS packet, with some additional restrictions on the possible values of each field. The transaction ID field should be set to 0 in a query and must be set to 0 in an answer; it is not used by mDNS. The question count, answer count, authority RR count, and Additional RR count correspond to the number

of questions, answers, authority resource records, and additional resource records in the query. As with DNS, questions, answers, and additional records represent questions asked by a querier, replies to those questions, and additional resource records not explicitly asked for. The authority resources records, however, are used for a different purpose: performing tie breaking when multiple hosts want to claim a new resource record. The details of the tie-breaking process are not described in this document but can be found in section 8.2 of RFC6762 [9].

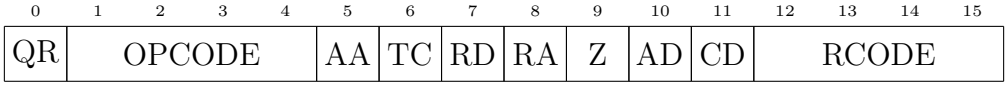

Figure 4.9: Representation of the flags of an mDNS packet

Figure 4.9 shows a more detailed representation of the flags. Fields OPCODE, RD, RA, Z, AD, CD, and RCODE are always all set to 0 and ignored by mDNS resolvers/clients. The QR (Query/Response) bit equals 1 in responses and 0 in queries. The AA (Authoritative Answer) bit has to be 0 in queries and 1 in responses. Finally, in queries, the TC (Truncated) bit indicates that an additional packet containing "known answers" may follow the current packet. That is part of the known answer suppression mechanism explained later in this section.

When receiving a query, the script operates in the following way. First, it reads all the questions it contains to see if any corresponds to one of its records. If it is indeed the case for one or more questions, the corresponding resources are added to a list of resources to send. Just like DNS, mDNS supports the "ANY" type, which acts as a wildcard for the query's type field. If the script observes it in a question, it adds all the matching resources to the list. Additionally, if the script detects a query for which it possesses the domain name but no records corresponding to its type/class combination, it adds a negative response to the additional record list of the answer packet. A negative response consists of a DNS NSEC [38] record with the domain name of the resource as the "next domain name" field and the types of the resources that do not exist in the "type bit map". After parsing every question, we need to check which answers to send. Indeed, mDNS implements a "Known-answer suppression mechanism" that limits network traffic. The host that asks the questions has to indicate which corresponding resources it has already received in the "Answers" section of its query. Any resource already known by the querier and previously added to the list of records to send is removed. A packet with additional known answers will follow if the TC bit is set in a query. In this case, the script does not generate a response and waits for the next packet. Once all of the known answers have been checked; if the sending list is not empty, the script can start crafting a response.

Usually, a response would always be sent to a multicast address. However, if

the unicast query flag is set, if the source port of the query is not 5353, or if the query was received on a unicast address, the potential answer is sent directly to the sending host in unicast, as indicated in the RFC. Now that we know which resources to include in the packet let's review all the fields of an mDNS response and explain how to fill them.

At any point, if the script detects an answer with a record whose type, class, and name correspond to one of its records but where the content differs, it checks if this resource is shared or unique. If it is shared, nothing has to be done and operation continues normally. Otherwise, it means that another host considers it owns a resource that should only belong to the sleeping host. RFC6762 defines a method to deal with these conflicts, but they result in one of the conflicting host having to rename their resource. That means the sleeping host should be woken up to deal with the conflict and know the resulting change.

The mDNS protocol is based on DNS. As such, a script that implements mDNS should also consider the entire specification of DNS and its numerous extensions. For this reason, we did not implement this script in practice and only worked on it on a conceptual level.

4.6 TCP control : maintaining a connection

The sleep proxy does not currently implement a way to process TCP streams with scripts. One could technically reimplement the TCP state machine inside of a script, but this process would be tedious and inefficient. However, the ability to process packets individually still allows us to perform some connection control.

When entering sleep mode, a host generally loses its TCP connections. Indeed, if the distant host tries to send data through the TCP stream at that time, it will not receive any acknowledgments and will, after enough attempts, consider the connection over. Even if no data is being sent, some TCP implementations might send TCP-keepalives [39] after being idle for too long. Those packets do not contain any useful information and are just used as probes to check the state of the connection. That means that the sleep proxy itself can answer them. The most simple version of a "TCP maintainer script" would wake up the sleeping host when a client sends it data and responds to TCP-keepalives on its own.

A host entering sleep mode needs to transfer part of its state to this script for it to work correctly. The state here is a set of tuples representing each established connection. Each of the tuples contains a local IP (the IP that belongs to the sleeping host in the connection), a distant IP, a local port, a distant port, a sequence number, an acknowledgment number , and an optional timestamp. The timestamp corresponds to the last timestamp sent toward the distant host if the timestamp option is present during the connection.

Option	Reaction	Reference
No-Operation	Ignore	RFC9293[39]
Maximum segment size	Syn	RFC9293[39]
Window scale	Syn	RFC7323[42]
SACK permitted	Syn	RFC2018[44]
SACK	Wake-up	RFC2018[44]
Timestamp	Respond	RFC7323[42]

Option	Reaction	Reference
Quick-Start Response	Wake-up	RFC4782[40]
User timeout option	Ignore	RFC5482[41]
TCP-AO	Wake-up	RFC5925[43]
Multipath TCP	Wake-up	RFC8684[45]
TCP Fast Open Cookie	Syn	RFC7413[46]
Encryption Negotiation (TCP-ENO)	Syn	RFC8547[47]

Table 4.1: Script reactions to the TCP options

Additionally, a TCP connection might have options that should be considered. Table 4.1 covers all options attributed to a number by the IANA [48] and officially defined in an RFC. It associates each option to one of four reaction categories. *Syn* means the option only appears during connection initialization and should, therefore, not appear in a packet intercepted by the script. *Ignored* means the option can be safely ignored because it does not impact the sleep proxy's ability to maintain the connection. *Wake-up* means the option requires direct attention from the sleeping host either because it relies on information the sleep proxy does not have or because it was not implemented by the script due to its complexity. Finally, the *Respond* reaction means that the sleep proxy can respond appropriately to a packet containing this option.

We now discuss each choice in more detail. The no-operation option contains no useful information and can, therefore, be safely ignored.

Maximum segment size, Window scale, SACK permitted, TCP Fast Open Cookie, and TCP-ENO are only used during the establishment of the connection and should not be observed by the sleep proxy.

Selective acknowledgments (SACKs) are used to indicate which segments arrived successfully. They generally imply that some segments did not make it to the remote host and should be retransmitted; we thus wake up the sleeping host when one of these is received to let it retransmit those packets.

The timestamp option is used to provide additional information regarding the time at which each packet was created. That can be useful to get an idea of the order in which received packets were sent or to estimate round trip time. The script must send its timestamp and echo back the timestamp of the remote host when

receiving a packet. Its timestamp is the previous timestamp sent incremented by one (with the first ever timestamp being communicated by the sleeping host) and the last timestamp received is saved every time in a buffer-local to the script. We use a small increment so that the proxy avoids going over the sleeping host's clock.

Quick-start response is used to negotiate a sending rate. It can be used during the establishment or in the middle of a connection. For the latter case, it means the client will send data soon, meaning the host should be woken up to receive it.

User timeout is an option that indicates how much time a host is willing to wait for data to be acknowledged until it forcefully closes the connection. While intercepting packets, the sleep proxy does not need to do anything about it. Indeed, this information is not of any use as the script will respond to all keepalives or wake up the sleeping host if needed, preventing a timeout. However, this information must be transferred to the host when it wakes up. To do this, the script stores the keepalive containing the user timeout in a buffer and sends it to the sleeping host when it wakes up.

TCP-AO and Multipath TCP would be too complex to deal with using a simple script and are out of the scope of this master thesis. Therefore, if one of the two corresponding options is seen in a packet, the script will produce a wake-up signal.

4.7 Test scripts

To test the performance of our proxy, we decided to implement six rules of increasing complexity. Those rules are not related to any application protocol. Instead, they work with arbitrary UDP packets and generally perform some computation before returning the packet to its sender. The list of rules is as follows.

4.7.1 Simple echo

This rule performs no computation and echoes a packet back to its sender. Its role is to determine the base cost of running a script.

4.7.2 Simple rule check

This rule performs a check to verify if the first byte of the sender's address is a specific number. Its purpose is to verify if adding a single instruction can have a significant cost in terms of script execution.

4.7.3 UDP checksum

This rule modifies the first byte of the payload and recomputes the UDP checksum before echoing it back to its sender. This rule provides a realistic task for the proxy, with complexity growing linearly with the payload size. The code used for this script is taken from RFC1071 [49].

4.7.4 Substring check

This rule checks the entire payload of the UDP packet, trying to find a specific substring inside of it. In all the tests, we've chosen a six-byte substring that was never included in the payload to maximize the time spent searching. This rule provides a more complicated task with linear complexity.

4.7.5 SHA-256 hash

This rule computes the SHA-256 hash of the payload of the UDP packet. The C implementation of the SHA-256 algorithm used can be found in a public GitHub repository[50].

4.7.6 AES-256

This rule encrypts the packet's payload using the AES algorithm with an arbitrary key of size 256. The C implementation of the AES-256 algorithm used can be found in a public GitHub repository[50].

Chapter 5

Experiments

This chapter presents an analysis of our sleep proxy's performance. We conducted several tests covering various parts of our implementation. We first start by describing our experimental setup. Then, we explain the results of experiments we have performed to characterize the sleep proxy. Finally, we use profiling tools to get a more in-depth understanding of the sleep proxy's internal operation.

5.1 Experimental setup

In this section, we describe the testbed used to perform our experiments. To test our implementation, we decided to set up a simple testbed consisting of a client, a Linux machine acting as a router running the sleep proxy, and a sleeping host. A representation of the topology is shown in Figure 5.1. The client and the sleeping host do not have a direct connection; they both have to go through the router to send their packets. Due to the simple nature of this network, static routing was used to connect the machines to each other. Practically, the client is a Raspberry Pi model 3B+ [51] running Alpine Linux with kernel version *6.1.55-0*. This device has a 1.4GHz 64-bit quad-core processor, 1GB of LPDDR2 SDRAM, and a Gigabit Ethernet interface over USB 2.0. According to the official specifications of the machine, its maximum throughput in practice is 300 Mbps. Meanwhile, the router and the host are both Intel® NUC Kit NUC5PPYH[52] running Ubuntu Linux with kernel version *6.5.0-41*. Both these devices have a 1.6 GHz 64-bit quad-core processor, 8 GB of DDR3L RAM, and an integrated ethernet controller that supports Gigabit ethernet. Considering that this machine only has a single ethernet interface by default, we used a TP-LINK UE306 USB network adapter [53] for the physical link between the router and the host. This additional interface also supports Gigabit ethernet. All the connections present in this testbed use wired links. Unless stated otherwise, the setup used is this one in all of the tests.

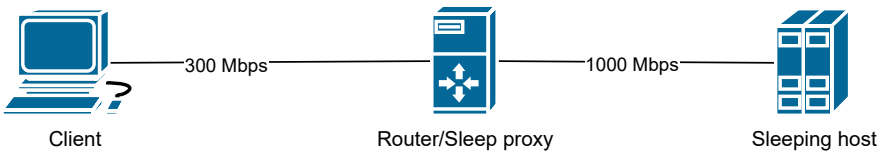

Figure 5.1: Representaton of the topology used for the experiments

5.2 Packet processing rate

One of the most important aspects of the sleep proxy to profile is the speed at which it can process packets. Indeed, with the added steps of matching the packet against a rule and running a WebAssembly script to determine the appropriate reaction, we expect it to be less efficient than a regular server processing packets with native code. Assessing the maximum rate at which our sleep proxy can process packets lets us see if our solution could be realistically used in an enterprise network and how much traffic it could handle. During the development of the sleep proxy, we tried to tune several parameters to increase its performance. This section provides an analysis of 3 different versions of our sleep proxy, each at different stages of optimization. The first objective of these tests is to understand the effects of both packet size and packet rate on the behavior of the sleep proxy. The second is to evaluate which options for the sleep proxy lead to the best performance.

Our methodology for these tests was the following: The sleeping host starts by uploading a set of test rules defined in Section 4.7 and enters sleep mode. Then, for each rule, the client sends UDP packets to the port associated with that rule toward the sleeping host, at a specific rate, for one minute. Each rule is associated with a different port to always guarantee that only one of them is triggered at a time. Additionally, we measured the response rate of a simple responder written in Rust to have a baseline for our experiments. This simple responder uses a UDP socket from the standard rust library to echo packets back to their senders.

The tests were performed starting with a rate of 256 packets per second and multiplying it by a factor of two every iteration, up to a rate of 32768 packets per second. Regarding the size of the packets sent, the tests were performed with three size categories of UDP packets: small (50 bytes payload), medium (500 bytes payload), and big (1000 bytes payload). Each test was reproduced five times, and only the median rate was kept to reduce the impact of eventual variations during testing.

For the first experiment, we used our initial and most naive implementation of the sleep proxy. In this version, we did not attempt to optimize the scripts or

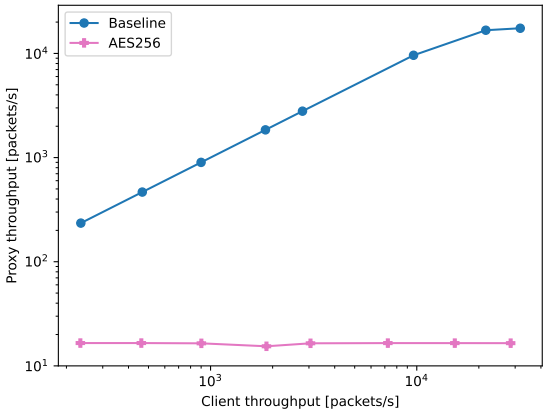

Figure 5.2: Comparison between the throughput of the client and the throughput of the sleep proxy for medium packets (500 bytes payload) when no optimizations are applied

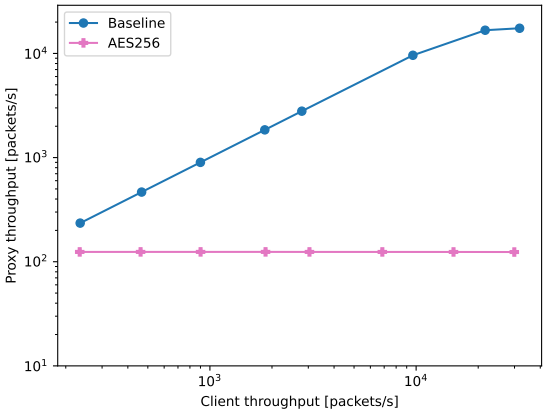

Figure 5.3: Comparison between the throughput of the client and the throughput of the sleep proxy for medium packets (500 bytes payload) when compiler optimizations are applied.

the performance of the runtime at all. The scripts were compiled using the default parameters of the toolchain, and the runtime was set to interpreted mode. Figure 5.2 shows the throughput of the sleep proxy in relation to the throughput of the client for the AES-256 rule defined in Subsection 4.7.6. We can see that, with no optimizations, the sleep proxy reaches mediocre performances for this rule. At a rate of 256 medium packets per second (1,11 Mbps), the proxy is already saturated and only manages to output approximately 16 packets per second (69,376 kbps).

For the second version of the sleep proxy, we decided to examine the possible optimizations we could perform on the scripts. We found out that the wasi sdk toolchain[20] supported optimization level flags. More specifically, as the toolchain is based on Clang [30], the optimization flags defined in the documentation of the compiler [54] can be used. To see how much of an increase in speed we could get via compiler optimizations, we decided to use the highest optimization level in regards to speed ("Ofast"). Figure 5.3 shows the performance of the AES rule compared to the baseline under these conditions. In this version, the sleep proxy manages to reach a rate of more than 100 packets per second (433,6 kbps). This represents a great enhancement compared to the performances of our naive implementation but still is not enough to handle our test loads, as the plateau seen in the graph shows.

In the final version, we wanted to see if we could enhance the performance by changing the way the runtime executes the scripts. The documentation of Wasmedge [55] indicates that the runtime supports an AOT (ahead of time) mode for script execution in addition to its interpreted mode. More specifically, using the

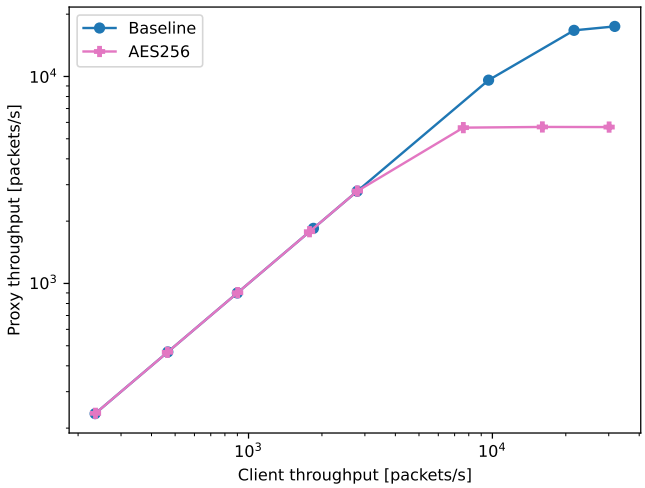

Figure 5.4: Comparison between the throughput of the client and the throughput of the sleep proxy for medium packets (500 bytes payload) when compiler optimizations and AOT are applied

Wasmedge API, it is possible to invoke an AOT compiler in a program to produce a compiled version of a script. This compiled version can then be loaded into the program and used in the same way as an interpreted script. We thus changed our implementation to use AOT compilation. In the following experiment, we enabled both compiler optimizations and AOT to achieve the best possible performance. The results of the experiment for the AES-256 rule are shown in Figure 5.4. We can observe a clear increase in performance compared to the two previous versions. The proxy is now able to handle much higher rates and does not immediately get saturated. The plateau is reached at approximately 6000 packets per second (26,016 Mbps). This test seems to indicate that this version of the sleep proxy is the most efficient one. For this reason, we kept its parameters for the final version of our proxy and used them in every subsequent test.

Figures 5.5 and 5.6 show the impact of the packet size on the processing of packets for the simple echo rule (defined in Subsection 4.7.1) and the SHA-256 (defined in Subsection 4.7.5) rule respectively. As expected, the size of the packets has an impact on the processing speed of the SHA-256 rule. This increase can be explained by the fact that the complexity of the computations necessary to compute a SHA-256 hash grows linearly with the size of the input. Despite this, we can see that the proxy's sending rate is not exactly inversely proportional to the size of the packets. This is due to the fact that the sleep proxy does not simply run the script. It has to perform other operations, which also take time but do not necessarily have a linear complexity. One of those factors is illustrated in Figure 5.5.

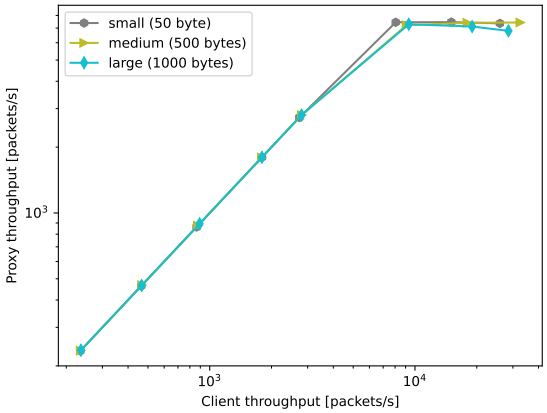

Figure 5.5: Comparison between the throughput of the client and the throughput of the sleep proxy depending on the size of the packet for the echo rule

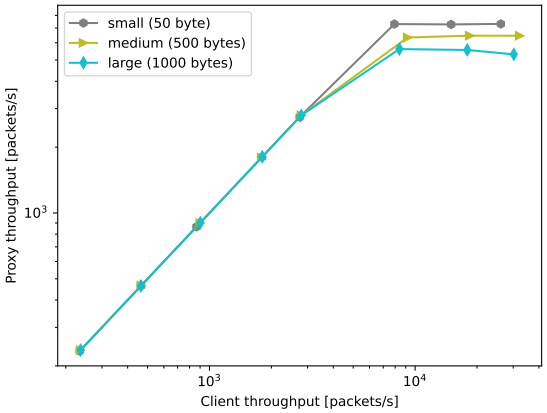

Figure 5.6: Comparison between the throughput of the client and the throughput of the sleep proxy depending on the size of the packet for the AES-256 rule

Indeed, there is a small but noticeable decrease in performance for large packets when passing through the echo rule, despite the fact that the script itself executes in constant time. One of the possible explanations for this is that the packets need to be copied in the memory zone of the WebAssembly VM to be processed. A larger packet takes more time to be copied than a small one, possibly explaining the gap in the graph.

Figure 5.7 regroups the performance of all the test rules in the experiments where the sleep proxy receives medium-sized packets (500 bytes payload). As we can see, the sleep proxy manages to handle up to 7000 packets per second for most rules, which corresponds to a throughput of 30,352 Mbps. The rule with the lowest throughput is the one that performs an AES-256 encryption. This was expected due to the more complex nature of the algorithm when compared to the other rules. One interesting observation is that, despite having different complexities, the difference between the throughput of the rules is not as high as one could expect. A possible explanation for this phenomenon would be that the optimizations we applied made the cost of executing the script low enough not to be the main factor defining our throughput. A more in-depth analysis of the performance of the sleep proxy is provided in Section 5.5.

5.3 Effect of different factors on response speed

During the previous test, we estimated the maximum packet rate a host could support to evaluate its performance. However, the complexity of the rules and the

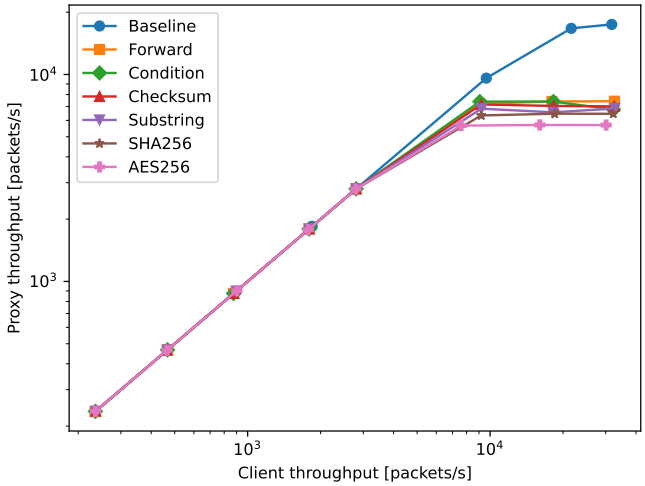

Figure 5.7: Comparison between the throughput of the client and the throughput of the sleep proxy for all of the test rules with packets of medium size (500 bytes payload)

rate at which packets are received are not the only factors that could impact the performance of the sleep proxy. In a more realistic setting, we expect multiple hosts to be connected to the sleep proxy. Each of these hosts could have varying numbers of rules to process their traffic. In this section, we try to measure the impact of two additional factors on the performance of the sleep proxy: The number of subscribed clients and the number of rules currently in application on the sleep proxy. In the following tests, all of the uploaded scripts are simply duplicates of our most test script defined in Subsection 4.7.1. This choice was made to minimize the overhead caused by the execution of the script, as it is not the focus of this section.

5.3.1 Number of rules

In this experiment, we ran several tests where the sleeping host subscribed to the sleep proxy with varying numbers of rules. Then, a client measured the response time of the sleep proxy for a specific rule by sending 10000 packets one by one and computing the time it took to receive an answer.

The results of the experiment can be seen in Figure 5.8. We can see that the latency stays stable from one up to 1000 rules, with a value of approximately $1.3ms$. This seems to indicate that, for those numbers, the cost of looking up which rules were triggered by a packet is negligible. Starting from 2500 rules, the cost is no longer negligible, and the latency progressively increases. At 10000 rules, we reached a mean latency of $3.1ms$.

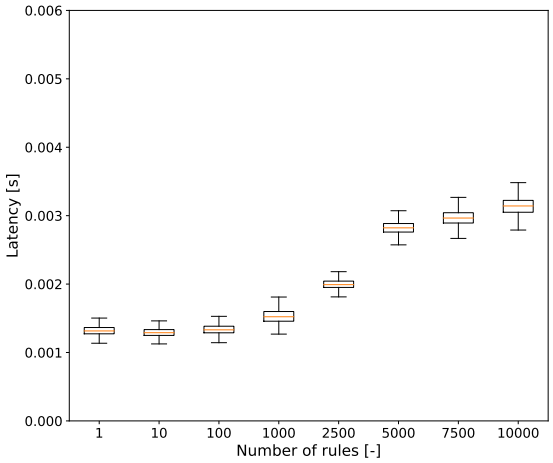

Figure 5.8: Round trip time of the sleep proxy for varying numbers of rules

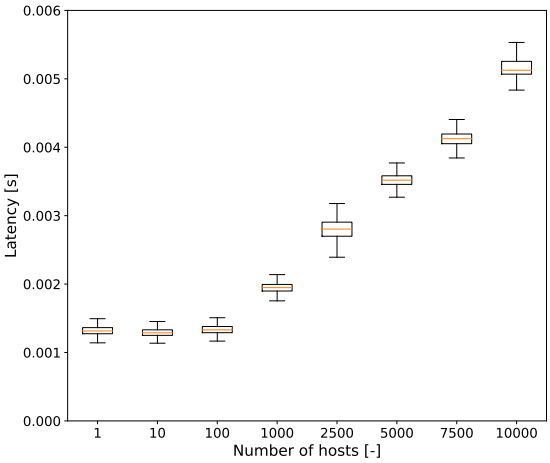

Figure 5.9: Round trip time of the sleep proxy for varying numbers of hosts

5.3.2 number of sleeping hosts

For this experiment, we ran several tests where we subscribed varying numbers of "hosts" to the sleep proxy. In reality, only one of those hosts is real (the sleeping host of our testbed); the others have fake IPs and MAC addresses. We added them in batch via a script that pretended to have their addresses. Each of the hosts has a single rule. We used the same methodology as the one described in the previous test to estimate the latency.

The results of the experiment can be seen in Figure 5.9. The observations are similar to those of the previous experiment except that, in this case, we can already observe a noticeable increase in latency when reaching 1000 hosts. In addition, the latency seems to grow faster when adding hosts than when adding rules. This makes sense as the proxy treats the rules of each host separately. Each host added means an additional data structure that the sleep proxy has to iterate over when receiving a packet. At 10000 hosts, the sleep proxy reaches a mean latency of $5.1ms$. While this increase and the one seen in the previous subsection are not large enough to make the proxy unusable, they would certainly affect the quality of the service it provides.

5.3.3 Optimization

As we have seen in the two previous subsections, a growth in latency can be observed when high enough quantities of rules/hosts are added to our sleep proxy. Our current implementations works by iterating over every single rule of every single host to see if at least one of its IP prefixes correspond to the current packet

we are handling. If this is the case, those rules should process the packet. We could improve the search by using an interval tree data structure. As its name implies, an interval tree is a tree structure containing intervals. It can be queried for a value and return all the elements of the tree containing the said value. Such a data structure could easily be modified to contain IP addresses instead of numbers. By associating every IP prefix with the rules that are listening to it, we could reduce the lookup cost to a simple search in an interval tree.

5.4 Flood ping

This section aims to compare the performance of a host responding to a large number of ping requests by itself with the performance of the ICMP script defined in Section 4.3. The point of this experiment is to analyze the performance of the sleep proxy in a more realistic situation by using an actual protocol.

For this experiment, we used *hping3* [56], a packet generator that can be used to send several types of traffic at high rates, usually to perform security tests. In our case, we decided to send the default ICMPv4 echo requests offered by the command at a fixed rate in flood mode. The default requests consist of a 42-byte packet that does not contain any payload. The flood mode means that the sender does not wait to receive the echo response of the last echo request sent before sending a new one. We measured both the ability of the sleeping host to answer pings normally and the ability of the sleep proxy to answer pings in its place. Each test consisted of 10000 echo requests sent at a rate progressively increasing with each iteration. To account for eventual variations in the performance of the sleep proxy and the sleeping host, we performed the experiment 5 times per sending rate and only kept the median results.

Figure 5.10 shows the results of our experiment. In each of the tests, we measured the loss rate of the entity that was responsible for answering our echo requests. As we can see, up to 6000 packets per second, both responders can handle the traffic with virtually no packet loss. Starting from 10000 packets per second, the sleep proxy is not fast enough to answer all of the packets in time, which means some of the incoming requests are dropped, resulting in packet loss. The loss then gradually increases without ever reaching 100%. As the sleep proxy benefits both from its NIC's buffer and the buffer of the program itself, it is always able to answer a portion of the packets received.

This point of saturation arises later for the native responder, which is the expected behavior as it does not have all of the additional overhead of the sleep proxy. In the end, we can see that, as one could have expected, the proxy is not as efficient as a native responder directly implemented in a host. Despite this, it still manages to handle several thousands of packets per second without causing losses.

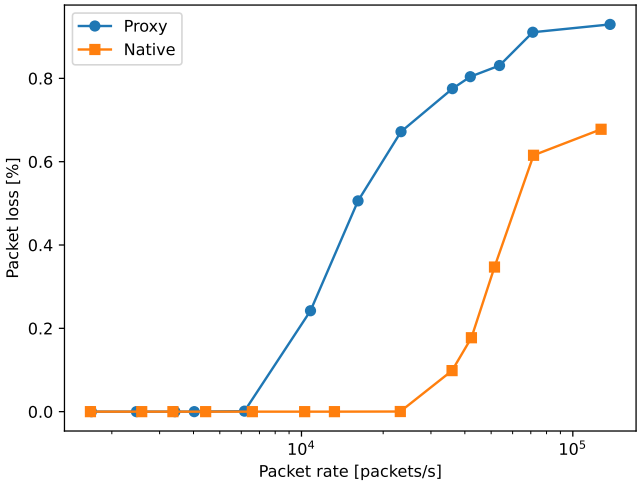

Figure 5.10: Loss rates of the sleep proxy and a native ICMPv4 responder when subjected to various rates of echo requests

5.5 Performance profiling

In order to understand the performance of the sleep proxy better, we decided to analyze where the program spent its time executing instructions. To do this, we used perf[57], a Linux command line tool that can profile programs and give various statistics about their execution by sampling CPU performance counters.

In the following section, we analyze the results obtained when running perf in a situation where a single host is subscribed to the sleep proxy and has the AES-256 rule defined in Subsection 4.7.6 as its one and only rule. To avoid taking into account the cost of compiling scripts, we first uploaded the rules, waited for Wasmedge's AOT compiler to finish compilation, and then only attached perf to the running program. The compilation of the scripts only happens once and does not affect the processing of packets once it is over. Considering that we want to see how the sleep proxy spends its time during normal execution, we estimated that it would be better to ignore this one-time cost.

Table 5.1 contains the results of our analysis. We can see that the part of the sleep proxy taking the most CPU time is the execution of the "run_func" function from the Wasmedge API. Practically, the call to this function includes all the steps Wasmedge has to take to run a script, such as looking up the name of the WebAssembly module in its list of registered modules and invoking the executor. We do not go into more details as doing so would be out of the scope of this master's thesis. We made sure to isolate the portion of this category related to the execution of the script to put it in the *Script execution* category. This category represents the CPU time spent specifically on the script itself. The *packet*

Category	CPU cycles (%)
run_func (Wasmedge)	43,103
Script execution	10,25
Packet interception	8,38
Additional Wasmedge overhead	7,66
Interpreting reactions	6,17
Proxy/VM communication	5,29

Category	CPU cycles (%)
Tokio overhead	4,25
Main responder thread operations	4,18
Memory operations	1,63
Initial packet processing	1,38
Synchronization overhead	0,58
Others	7,127

Table 5.1: Distribution of the percentage of CPU cycles in the sleep proxy

interception category represents the time spent by a thread listening to the raw socket and sending the packets it receives toward the responder thread to be processed. The *additional Wasmedge overhead* section regroups the CPU usage of the other calls to the Wasmedge shared library and the Wasemdge API at places in the code that do not correspond to one of the other categories. *Interpreting reaction* is the name given to the category encompassing the CPU cycles spent interpreting the reactions produced by the scripts. This includes ignoring packets, waking up hosts, or sending packets when necessary. The actions of writing to the memory of the WebAssembly VM and reading the results it produces are regrouped in the *Proxy/VM communication* section. The *tokio overhead* category indicates the amount of CPU taken by the Tokio runtime [29], which allows the sleep proxy to perform asynchronous tasks by handling the scheduling of those tasks. The *main responder thread operations* is the category that includes the main loop of the responder part of the sleep proxy. This is the part that verifies if the other threads have sent any messages and handles them if necessary. *Memory operations* represent the various memory allocations and deallocations present in the code that are not related to any of the other categories. *Packet processing* includes the action of parsing the headers of the received packets and matching them against the rules the proxy has to apply. The *synchronization overhead* represents the time spent dealing with synchronization primitives necessary to allow the proxy to handle multiple threads. Finally, the *Other* section regroups the calls that were not significant enough to be taken into account, the ones that were not large enough to have their own categories, and the few from which the perf tool could not get the debug symbols.

5.6 Memory usage

The last aspect we discuss in this chapter is the memory usage of the sleep proxy. Practically, if we want to deploy the sleep proxy on a piece of hardware already present in a network, we need to know approximately the impact it would have on its memory usage. If the proxy were to be deployed on a router, for instance, a high enough memory usage could potentially impact the other activities of the router, possibly causing issues in the network. To perform these tests, we used the Heaptrack [58] tool, a heap memory profiler for Linux systems that can record data about the locations of a program responsible for memory allocations and the total memory footprint at a given time in the program's execution.

We did experiments similar to the one we made in the previous section using the AES-256 script. A host starts by subscribing to the sleep proxy with a rule and enters sleep mode. Then, a client starts sending 25 packets per second for 10 seconds. The difference with the methodology of the previous section is that we performed this test twice. The first time, we started measuring after the rules were uploaded. The second time, however, we started measuring from the moment the sleep proxy started. Recording data from the start allowed us to measure the memory consumption in 3 stages of the sleep proxy. The first one is the idle stage, when the sleep proxy does not have any sleeping host subscribed to it. We measured a memory consumption of 480,5 kB on the heap for this experiment. The second stage is the compilation of the scripts. While this only happens once per uploaded rule, we believe analyzing its effect on memory consumption is still interesting to have an idea of the capacities necessary for the host to run the sleep proxy. The peak memory consumption during that stage was 7.0 MB. Finally, for the third and last stage, we analyzed the memory consumption when the proxy was receiving packets. During that time, the memory consumption did not have large variations and stayed stable at around 773.5 kB.

The other experiment we performed allowed us to isolate the part where the proxy responds to packets. We use the results of this experiment to estimate which parts of the code consume memory on the heap. The two main sources of memory allocations are the packet receiver ($8,3\%$) and Wasmedge's run_func function ($90,08\%$). The packet receiver handles the reception of packets on the raw socket, meaning it has to allocate space frequently to receive packets. The run_func function has to initiate all the structures necessary to run a function of the WebAssembly VM, explaining its large part of memory usage.

5.7 Experiments conclusion

From the experiments we conducted, we can conclude that our implementation of SLOTH achieves correct performance with low memory overhead. When using all of the optimizations we found, the sleep proxy was able to process several thousands of packets with our various test rules. This was also the case in a more realistic test scenario based on the ICMPv4 protocol. Additionally, we realized that some parts of our proxy, such as the lookup of hosts and rules, could still be improved upon. In practice, the most significant part of our proxy in terms of execution time is related to Wasmedge and its API, representing more than 50% of the CPU cycles in our test scenario.

Chapter 6

Conclusion

In this master's thesis, we tackled the problem of energy waste in computer networks. To do so, we first designed a solution consisting of a customizable sleep proxy based on a rule system. This sleep proxy processes the traffic of one or more sleeping hosts according to rules they define by themselves.

We then wrote a real implementation of this proxy in Rust, using a script system based on WebAssembly modules. To demonstrate the capacities of our sleep proxy, we wrote a few example scripts based on real protocols and explained how they could be used to maintain the state of a sleeping host.

Finally, we measured the performance of the sleep proxy in several aspects, ranging from its ability to process packets quickly to its CPU usage. We found out that our sleep proxy achieved satisfying performance despite the fact that it could still be improved in some aspects. We managed to create a solution that is **flexible**, thanks to the WebAssembly-based rule system, **scalable** as SLOTH is capable of handling several hundreds of hosts with minimal slowdowns in response speed, and **easily deployable** because of the fact that our solution is entirely implemented in software and does not require any additional equipment on the network.

6.1 Further work

The main objective of this master thesis was to provide a prototype for a script-based sleep proxy, the current implementation is not meant to be used in production due to security concerns and a lack of optimization in some aspects of the code. In this section, we discuss ways in which we could improve the security of SLOTH and add new features to it.

6.2 Security considerations

As of now, the sleep proxy uses a very simple protocol with two main flaws. First of all, the messages are not encrypted, meaning a malicious user could listen to the messages exchanged with the sleep proxy to know the state of any host that subscribed to it. This includes a compiled version of the script used to process their traffic and a list of open ports and used IP addresses. This data would be a great source of information to a cyberattacker, as it would allow them to know more about the topology of the network. The second flaw lies in the lack of authentication in our current implementation. Currently, there is no mechanism for the sleep proxy to verify if the host who sent a subscription request was actually allowed to do so and if it owned the IP addresses it subscribed to the sleep proxy. An attacker could exploit this to emit false subscriptions and perturb traffic on a network by uploading specific rules. An attack could consist of subscribing one or multiple host(s) with high numbers of complex rules to perform a denial of service attack. Another use could be to make a script that listens to all traffic and sends it back to a specific distant host to spy on the local traffic.

Both of these issues represent a great concern and are reasons the current version of the sleep proxy should only be used in an experimental environment. To solve this issue, we need to implement both authentication and encryption in our protocol. One solution could be to communicate on top of TLS, as the protocol provides both encryption and authentication. Two-way SSL authentication should be enabled to ensure that both the client and server are authenticated. Only a specific set of clients would be authorized to subscribe to the sleep proxy.

6.3 Support for TCP streams

In its current version, the sleep proxy can only work on a "per-packet" basis. This means that a script has to process every packet of a connection individually. This does not cause an issue if we want to write scripts for UDP-based protocols with a minimal state or perform TCP connection control. However, this makes the processing of TCP-based protocol almost impossible, as processing the stream would require the entire TCP state machine to be reimplemented inside of a script. This process would be tedious and highly inefficient. In future versions of the sleep proxy, we would like to implement the possibility to perform operations on TCP streams. This could be achieved by handling the TCP logic inside of the proxy program and only exposing the stream to the scripts. Another possibility would be to use the WASI [59] extension of WebAssembly to create TCP sockets inside of scripts.

Bibliography

[1] IPCC. Summary for policymakers. In *Climate Change 2023: Synthesis Report. Contribution of Working Groups I, II and III to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change*, pages 1–34. Geneva, Switzerland, 2023.

[2] Jens Malmodin, Nina Lövehagen, Pernilla Bergmark, and Dag Lundén. Ict sector electricity consumption and greenhouse gas emissions–2020 outcome. *Telecommunications Policy*, 48(3):102701, 2024.

[3] BIPT and Deloitte. Sustainability of telecommunication networks and operators in belgium. 2022.

[4] ADME and Arcep. Assessment of the digital environmental footprint in france in 2020, 2030 and 2050. 2023.

[5] Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of internet background radiation. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 27–40, 2004.

[6] Eric Wustrow, Manish Karir, Michael Bailey, Farnam Jahanian, and Geoff Huston. Internet background radiation revisited. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 62–74, 2010.

[7] Jakub Czyz, Kyle Lady, Sam G Miller, Michael Bailey, Michael Kallitsis, and Manish Karir. Understanding ipv6 internet background radiation. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 105–118, 2013.

[8] Bradley Cain, Dr. Steve E. Deering, Bill Fenner, Isidor Kouvelas, and Ajit Thyagarajan. Internet Group Management Protocol, Version 3. RFC 3376, October 2002.

[9] Stuart Cheshire and Marc Krochmal. Multicast DNS. RFC 6762, February 2013.

[10] Aruna Prem Bianzino, Claude Chaudet, Dario Rossi, and Jean-Louis Rougier. A survey of green networking research. *IEEE Communications Surveys & Tutorials*, 14(1):3–20, 2010.

[11] Amd white paper on the magic packet technology. https://www.amd.com/content/dam/amd/en/documents/archived-tech-docs/white-papers/20213.pdf. Accessed: 2024-07-26.

[12] UEFI Forum, Inc. *Advanced Configuration and Power Interface (ACPI) Specification*, 8 2022. Release 6.5.

[13] Kenneth J Christensen and Franklin 'Bo' Gulledge. Enabling power management for network-attached computers. *International Journal of Network Management*, 8(2):120–130, 1998.

[14] Joshua Reich, Michel Goraczko, and Aman Kansal. Sleepless in seattle no longer. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[15] Understanding sleep proxy service. http://stuartcheshire.org/SleepProxy/. Accessed: 2024-07-26.

[16] Yuvraj Agarwal, Steve Hodges, Ranveer Chandra, James Scott, Victor Bahl, and Rajesh Gupta. Somniloquy: augmenting network interfaces to reduce pc energy usage. In *Networked Systems Design & Implementation (NSDI)*, 2009.

[17] Gerry Wan, Fengchen Gong, Tom Barbette, and Zakir Durumeric. Retina: analyzing 100gbe traffic on commodity hardware. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 530–544, 2022.

[18] Android packet filter. https://source.android.com/docs/core/connect/android-packet-filter. Accessed: 2024-07-26.

[19] Webassembly community group. https://www.w3.org/community/webassembly/. Accessed: 2024-07-26.

[20] Wasi sdk : Wasi-enabled webassembly c/c++ toolchain. https://github.com/WebAssembly/wasi-sdk. Accessed: 2024-07-26.

[21] Emscripten. https://emscripten.org/. Accessed: 2024-07-26.

[22] Rust and webassembly documentation. https://rustwasm.github.io/docs.html. Accessed: 2024-07-26.

[23] py2wasm. https://github.com/wasmerio/py2wasm. Accessed: 2024-07-26.

[24] A webassembly milestone: Experimental support in multiple browsers. https://hacks.mozilla.org/2016/03/a-webassembly-milestone/. Accessed: 2024-07-26.

[25] Wasmer. https://wasmer.io/. Accessed: 2024-07-26.

[26] Wasmtime. https://wasmtime.dev/. Accessed: 2024-07-26.

[27] Webassembly micro runtime (wamr). https://github.com/bytecodealliance/wasm-micro-runtime. Accessed: 2024-07-26.

[28] Extism. https://extism.org/. Accessed: 2024-07-26.

[29] Tokio - an asynchronous rust runtime. https://tokio.rs/. Accessed: 2024-07-26.

[30] clang. https://clang.llvm.org/. Accessed: 2024-07-26.

[31] WebAssembly Community Group. Webassembly specification release 2.0 (draft 2024-07-18).

[32] Bill Croft and John Gilmore. Bootstrap Protocol. RFC 951, September 1985.

[33] Ralph Droms. Dynamic Host Configuration Protocol. RFC 2131, March 1997.

[34] Internet Control Message Protocol. RFC 792, September 1981.

[35] Marco De Vivo, Eddy Carrasco, Germinal Isern, and Gabriela O De Vivo. A review of port scanning techniques. *ACM SIGCOMM Computer Communication Review*, 29(2):41–48, 1999.

[36] Daniel H. Steinberg Stuart Cheshire. *Zero Configuration Networking: The Definitive Guide*. O'Reilly, 2005.

[37] Stuart Cheshire and Marc Krochmal. DNS-Based Service Discovery. RFC 6763, February 2013.

[38] Scott Rose, Matt Larson, Dan Massey, Rob Austein, and Roy Arends. Resource Records for the DNS Security Extensions. RFC 4034, March 2005.

[39] Wesley Eddy. Transmission Control Protocol (TCP). RFC 9293, August 2022.

[40] Pasi Sarolahti, Amit Jain, Sally Floyd, and Mark Allman. Quick-Start for TCP and IP. RFC 4782, January 2007.

[41] Fernando Gont and Lars Eggert. TCP User Timeout Option. RFC 5482, March 2009.

[42] David Borman, Robert T. Braden, Van Jacobson, and Richard Scheffenegger. TCP Extensions for High Performance. RFC 7323, September 2014.

[43] Dr. Joseph D. Touch, Ron Bonica, and Allison J. Mankin. The TCP Authentication Option. RFC 5925, June 2010.

[44] Sally Floyd, Jamshid Mahdavi, Matt Mathis, and Dr. Allyn Romanow. TCP Selective Acknowledgment Options. RFC 2018, October 1996.

[45] Alan Ford, Costin Raiciu, Mark J. Handley, Olivier Bonaventure, and Christoph Paasch. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 8684, March 2020.

[46] Yuchung Cheng, Jerry Chu, Sivasankar Radhakrishnan, and Arvind Jain. TCP Fast Open. RFC 7413, December 2014.

[47] Andrea Bittau, Daniel B. Giffin, Mark J. Handley, David Mazieres, and Eric W. Smith. TCP-ENO: Encryption Negotiation Option. RFC 8547, May 2019.

[48] Transmission control protocol (tcp) parameters. https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml. Accessed: 2024-07-26.

[49] Computing the Internet checksum. RFC 1071, September 1988.

[50] crypto-algorithms : Basic implementations of standard cryptography algorithms. https://github.com/B-Con/crypto-algorithms/. Accessed: 2024-07-26.

[51] Raspberry pi 3 model b+. https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/. Accessed: 2024-07-26.

[52] Intel® nuc kit nuc5ppyh. https://www.intel.com/content/www/us/en/products/sku/87740/intel-nuc-kit-nuc5ppyh/specifications.html. Accessed: 2024-07-26.

[53] Ue306, usb 3.0 to gigabit ethernet network adapter. https://www.tp-link.com/en/home-networking/usb-ethernet-adapter/ue306/. Accessed: 2024-07-26.

[54] clang - the clang c, c++, and objective-c compiler : Command guide. https://clang.llvm.org/docs/CommandGuide/clang.html#code-generation-options. Accessed : 2024-07-26.

[55] Wasmedge runtime. https://wasmedge.org/. Accessed: 2024-07-26.

[56] hping network tool. https://github.com/antirez/hping. Accessed: 2024-07-26.

[57] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2024-07-26.

[58] heaptrack - a heap memory profiler for linux. https://github.com/KDE/heaptrack. Accessed: 2024-07-26.

[59] Webassembly system interface. https://github.com/WebAssembly/WASI/tree/main. Accessed: 2024-07-26.

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