POLITECNICO DI TORINO

Master's Degree Thesis

Management Engineering

A.y. 2023/2024

February 23^{rd}, 2024

Enhancing Network Interception with Mitmproxy

An Open Source Solution for Transparent Proxy Mode on macOS and Linux

Supervisors

Prof. Marco TORCHIANO

Doc. Maximilian HILS

Candidate

Emanuele MICHELETTI

Summary

Mitmproxy is an open-source tool designed for intercepting and manipulating HTTPS traffic [1]. It allows users to intercept traffic from an entire machine or a single, specific process, offering flexibility in a range of operational modes: regular, reverse, upstream, SOCKS, DNS and transparent. In transparent mode, the tool operates at the operating system (OS) layer, making it OS-specific and increasing the complexity of implementation.

On macOS, two possible approaches were examined: the first was to redirect any packet on the *User Tunnel* (UTUN) interface: from UTUN packets could be managed and sent to Mitmproxy. The code was written exclusively in Rust, leveraging its capabilities to operate at a low level; having only one language to maintain might be an advantage but on macOS, conditionally redirections of packets are complex, this forced the redirection of all traffic to Mitmproxy only filtering requests later. This behavior is suboptimal because Mitmproxy bears the burden of processing the entire traffic even when it is only concerned with a specific process.

The second approach, currently taken, involves a combination of Rust and Swift. Swift side is the actual Redirector: a companion app that exploits Apple *Network Extensions* (NE). This allows to read the process identifier (PID) and the process name of the flow source app, deciding which packet to send to Mitmproxy and which packet to ignore. Swift is a good choice to have a perfect fit with the Apple systems. Rust side runs the Redirector sending all configuration details and forward packets received from the swift side to the core of Mitmproxy.

The initial version of this approach used *Unix pipes*, a *simplex* inter-process communication (IPC) system, but three separate channels are required: one for configuration details, another for inbound packets from the Rust side to the Swift Side, and a third for outbound packets from the Swift side to the Rust side. To enhance communication efficiency, pipes have been replaced with Unix sockets,

which support full-duplex communication, consolidating the three separate pipes into a single socket.

The serialization and the deserialization of data, both for configuration and packets, are implemented with Protocol Buffers (Protobuf): a language-neutral, platform-neutral, extensible mechanism for serializing structured data.

On Linux, the development stage is more immature than on macOS. Because of the absence of direct APIs, the strategy is to take advantage of the EBPF, allowing programs to run directly in kernel space [2] and exploiting two particular types of eBPF programs called: TC and KProbe, making them work together.

Table of Contents

List of Figures

Acronyms

HTTPS
Hypertext Transfer Protocol Secure

HTTP
Hypertext Transfer Protocol

TCP
Transmission Control Protocol

UDP
User Datagram Protocol

SSL
Secure Sockets Layer

TLS
Transport Layer Security

IP
Internet Protocol

DNS
Domain Name System

CPU
Central Processing Unit

OS

Operating System

UTUN

User TUNnel

SOCKS

Socket Secure

PID

Process Identifier

Protobuf

Protocol Buffers

IPC

Inter-Process Communication

NE

Network Extensions

eBPF

extended Berkeley Packet Filter

API

Application Programming Interface

FFI

Foreign Function Interface

CI

Continuous Integration

I/O

Input/Output

MPSC

Multi Producers Single Consumer

TCP/IP

Transmission Control Protocol/Internet Protocol

GIL

Global Interpreter Lock

ISO/OSI

International Organization for Standardization/Open Systems Interconnection

VPN

Virtual Private Network

UDS

Unix Domain Socket

HTML

Hypertext Markup Language

XDP

eXperimental Data Path

TC

Traffic Control

KProbe

Kernel Probe

cgroups

Control Groups

UProbe

User Probe

LSM

Linux Security Module

IoT

Internet of Things

Chapter 1

Introduction

1.1 Context

Proxies are extensively used in a wide range of scenarios: from security to performance, from privacy to debugging. Main difference between a regular proxy and a transparent one is that the transparent proxy does not require any configuration client side, this means that the network packets can be proxied, also if configuration can not be changed (e.g. most Android or iOS apps), or if the client does not have to or does not want to worry about the presence of the proxy (e.g. public or enterprise network).

The applications of a transparent proxy are diverse, extending beyond its foundational role. A notable use case involves optimizing and improving local network efficiency, a subject explored in literature [3]. Additionally, transparent proxies prove invaluable for debugging networks of applications which do not allow manual configuration changes, as seen in studies related to application analysis [4]. Furthermore, their practical utility extends to the analysis of malware traffic, offering insights into the workings of malicious software and contributing to effective mitigation strategies [5]. In various scenarios, transparent proxies emerge as key players, facilitating content filtering and access control without imposing explicit user configurations. This quality is particularly advantageous in public Wi-Fi networks, where transparent proxies seamlessly filter undesirable content, enhancing overall network security without disrupting end-users. In addition, the transparent proxy proves indispensable in overcoming challenges posed by devices lacking the capability for manual configuration, crucial in Internet of Things (IoT)

environments and other specialized systems, ensuring the seamless functionality and security of networks across a diverse spectrum of devices and applications.

1.2 Purpose

Mitmproxy is a robust network tool meticulously engineered for the interception, analysis, and manipulation of network traffic, stands as an invaluable asset for users seeking enhanced control and insight into their network activities. Its multifaceted capabilities empower users to inspect and modify data as it traverses through the network, offering a versatile solution for tasks ranging from debugging to security analysis. With a user-friendly interface and a rich set of features, Mitmproxy has become a go-to choice for professionals and developers aiming to understand, troubleshoot, and optimize their network interactions. However, regardless of its array of capabilities, there exists a specific nuance related to Mitmproxy's *transparent mode*. Transparent mode is currently only partially integrated on the Windows operating system (OS), and it is not implemented at all on macOS and Linux. This disparity arises from the inherent operating system-specific nature of transparent proxy development.

To mitigate this limitation, a deep examination of the software architecture, system intricacies, and network configurations specific to macOS and Linux is essential. The primary objective of this thesis is to elucidate the intricacies involved in enhancing the *transparent mode* functionality on Mitmproxy, guaranteeing its adaptability and efficacy across diverse operating systems. This involves in-depth analyses of networking protocols, kernel-level interactions, and other crucial elements inherent to these operating systems.

It is noteworthy that the implementation on macOS has been successfully completed, showcasing a functional *transparent mode*. However, on the Linux platform, the implementation remains a work in progress, and efforts are ongoing to achieve the same level of completeness and robustness.

Chapter 2

Mitmproxy

2.1 General overview

Mitmproxy is a free and open source interactive proxy. At the moment when this thesis is written the project is at version 10, and it provides SSL/TLS capabilities for HTTP/1, HTTP/2 and Web Sockets [1].

The tool is available with different interfaces:

- Command line interface: it is accessible with the command *mitmdump*.
  It is the quicker way to run the proxy, all configurations must be set when the application is launched, and they can not be changed during the execution
- Textual User Interface: accessible with the command *mitmproxy* represents the common way to run the proxy
- Web graphical interface: accessible with the command *mitmweb*.
  It allows user to run the proxy and to interact with it using a Graphical User Interface on the browser

Mitmproxy core is written in python, and it is available also as a python package, this means that it can be used as a python library and extended in other projects.

2.2 Modes of operations

Mitmproxy is a large tool, and it offers users a spectrum of *mode of operations* [6], each of which is tailored to a specific use case:

- One of its primary modes is the *Regular* mode, which functions as the default operational setting. In this mode, users manually configure the network to divert traffic through the proxy, enabling comprehensive control over the interception process.
- In more special situations, users can opt for the *Transparent* mode, also referred to as *Local* mode, eliminating the need for manual configuration on the client side. In this mode, the proxy adeptly intercepts traffic without requiring explicit client-side setup, redirecting the network at the foundational level. This streamlined approach ensures a hassle-free deployment of Mitmproxy, particularly useful in scenarios where user intervention for network configuration is impractical, such as IoT devices or public networks.
- Mitmproxy also introduces the innovative *Wireguard* mode, leveraging the capabilities of Wireguard to automate traffic redirection. WireGuard is a modern and lightweight virtual private network (VPN) protocol known for its simplicity and high-speed performance. It employs advanced cryptographic techniques to ensure secure communication while minimizing complexities. Unlike traditional VPN protocols, WireGuard operates efficiently without the need for additional complexities in the kernel space, contributing to reduced resource usage. The *Wireguard* mode can be compared to the *Transparent* mode, the difference is that it does not remove the need for manual configuration completely, it only minimizes it, in fact the user must only configure the wireguard client and the proxy will take care of the rest.
- In the *Reverse* mode, Mitmproxy acts as an intermediary, forwarding client requests to the target server and subsequently relaying the server's responses back to the client. This mirrors the functionality of a conventional server, making it a valuable mode for specific use cases where this behavior is desired. For instance, in load balancer testing scenarios, Mitmproxy's *Reverse* mode enables the interception and analysis of traffic between the load balancer and backend servers. By replicating the server's role, it allows for a detailed examination of how the load balancer distributes incoming requests and manages responses. Similarly, in security auditing, Mitmproxy's *Reverse* mode becomes an indispensable tool for simulating potential attacks. By mimicking the server's function, it facilitates the inspection of how the system handles

client requests and responses, aiding in the identification of vulnerabilities and security risks. This mode is also beneficial in API development and testing, providing developers with a means to closely monitor and understand how client requests are processed by the server, ensuring the seamless communication of data between clients and servers.

- In scenarios involving a chain of proxies, the *Upstream* mode in Mitmproxy becomes instrumental for orchestrating a seamless flow of intercepted data between multiple proxies. One notable use case is in the context of corporate environments employing multiple layers of security measures. By utilizing the *Upstream* mode, administrators can redirect network traffic through specified proxies, each serving a distinct security purpose. For example, one proxy might focus on content filtering and malware detection, while another enhances encryption protocols. This modular approach to proxy chaining allows for a tailored and layered security infrastructure. Additionally, in large-scale networks or distributed systems, the *Upstream* mode proves valuable for load balancing and optimizing network performance. Redirecting traffic through designated proxies ensures an efficient distribution of data processing tasks, contributing to improved overall system responsiveness. Moreover, in situations where geographical restrictions apply, such as content localization or compliance with regional data privacy laws, the *Upstream* mode allows for the redirection of traffic through proxies strategically located in different regions. This not only ensures compliance but also enhances user experience by optimizing content delivery based on geographic proximity.

- In scenarios where a network environment heavily relies on the SOCKS protocol for communication, Mitmproxy's *socks* mode becomes invaluable. For example an organization that employs SOCKS proxies to enhance security and facilitate access to restricted resources. Mitmproxy, in *socks* mode, seamlessly integrates into this environment, allowing for the interception and analysis of SOCKS traffic. This proves beneficial for security audits, debugging, and monitoring SOCKS-based applications.

- *Dns* mode finds practical application in scenarios where fine-grained control over DNS queries is essential. For instance, in a cybersecurity context, an organization may use Mitmproxy to exclusively intercept and inspect DNS queries for potential malicious activity. By resolving intercepted DNS queries using the operating system's resolver, Mitmproxy provides a layer of security against DNS-related threats without affecting other communication channels.

2.3 A brief insight into the difference between regular mode and transparent mode

Since this study is focused on the *Transparent mode* implementation, is important to analyze the architectural differences between *Transparent mode* with *Regular mode*. Even if the technical implementation is OS-specific, the general idea is the same for all OS.

A regular proxy is manually configured to redirect traffic to the proxy, this practically means that 1. The client sends packets for which **the destination field is the proxy** itself. 2. The proxy receives the packets and forward them to the target server. 3. The server receives the packets and sends back the response to the proxy. 4. The proxy receives the response and forwards it to the client. This process is illustrated in Figure 2.1.

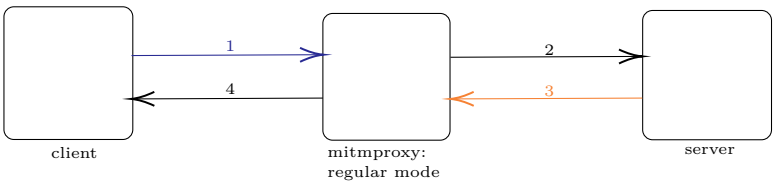

Figure 2.1: Mitmproxy in regular operation mode

A transparent proxy must handle with the redirection without the help of the manual redirection. This means that 1. The client sends packets for which **the destination field is the target server**. 2. The proxy intercepts packets and drops them after cloning its information. 3. The proxy makes a new packet from scratch, processes it from the original packet's information, and sends the new packet to the destination server. 4. The server sends the response to the client. 5. The proxy intercepts the response packet again and discards it. 6. The proxy makes a new response packet from scratch with the information from the original one and sends it to the client. This process is illustrated in Figure 2.2.

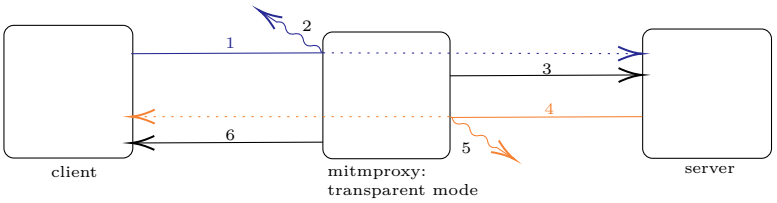

Figure 2.2: Mitmproxy in transparent operation mode

2.4 Installation and Distribution

Mitmproxy *v10.2* introduced Transparent mode as an official component within the standard modes of operation for macOS users. This significant enhancement will be seamlessly integrated by default, obviating the necessity for users to install supplementary features.

Concurrently, it is pertinent to note that the Linux version of Transparent mode is currently under development, with the release date yet to be definitively established. While the specific timeline remains undetermined, the Mitmproxy core development team is working towards the integration of this feature for Linux users in due time.

It is imperative to underscore that the entire source code for Mitmproxy, including the Transparent mode implementation, is openly accessible. The official Mitmproxy GitHub repository [7] serves as a central repository for the project's codebase, offering a comprehensive insight into the implementation of Transparent mode.

Chapter 3

Architecture and Structure

3.1 Rust as main language

The adoption of Rust as the primary programming language is motivated by the requirement for precise control over low-level system functionalities. Rust's capacity for delivering such control, coupled with high performance through zero-cost abstractions and meticulous resource management, proves essential in scenarios necessitating direct manipulation of hardware or system-level components. The language's built-in support for concurrent programming further enhances its appeal, enabling developers to efficiently manage parallel tasks and leverage the full potential of multicore architectures without compromising safety and reliability.

Additionally, Rust's compatibility with Python enhances the project's adaptability. The two languages seamlessly interface, enabling effective communication between them. This interoperability is advantageous, allowing for the integration of Rust's robust performance in low-level tasks with Python's agility and rapid development capabilities. In essence, the amalgamation of Rust and Python establishes a harmonious relationship, leveraging the distinctive strengths of each language in different facets of the project and resulting in a cohesive and potent system architecture.

3.1.1 Rust and its control over low-level system functionalities

Choosing the right programming language is a pivotal aspect in the development of software systems, especially when targeting low-level functionalities and system components. Programming languages play a crucial role in shaping a project's success, influencing factors such as performance, control over system functionalities, and overall adaptability.

In the context of Mitmproxy, and in particular of the transparent mode, the need for precise control over the network is a key aspect but not the only one. Rust, with its emphasis on *performance*, *safety*, and *zero-cost abstractions*, emerges as a valuable tool for developers seeking to enhance their control over network interactions. Beyond its utility in network programming, Rust proves beneficial for inter-process communication (IPC), providing a reliable and efficient means for different processes to exchange data seamlessly. Furthermore, Rust's capability to interoperate at the kernel level directly with kernel functions opens up new possibilities for fine-tuned control and optimization in networking applications. Rust's commitment to *safety-first* programming also ensures that developers can confidently build robust and secure network applications, while its *concurrency* model makes it easy to handle communication between different and separated component and companion apps involved in Mitmproxy's *transparent mode*, enhancing the scalability and responsiveness of the entire project.

3.1.2 Concurrency

Concurrency involves the execution of multiple tasks or processes concurrently, allowing for parallel execution and enhanced performance in software applications. In the case of Mitmproxy, a proxy tool inherently designed to handle simultaneous network requests and responses, concurrency is a fundamental requirement.

Mitmproxy operates concurrently by default to efficiently intercept, analyze, and manipulate network traffic. However, the complexity of its functionality goes beyond simple interception, since the tool incorporates companion applications such as redirectors and makes calls to kernel functions, these things must also be handled concurrently. Rust allows these kinds of operations to be performed without sacrificing code readability and especially performance.

Rust's strength in concurrency is notably attributed to its sophisticated *borrow*

checker, a dynamic analysis tool integral to the language's *ownership* model. The borrow checker operates during the compilation process, enforcing stringent rules on ownership, borrowing, and lifetimes. By scrutinizing the code for potential conflicts, it prevents issues such as data races and `null` pointer dereferences, which are prevalent challenges in concurrent programming. This feature ensures that multiple threads or processes can interact with shared data safely, eliminating the need for runtime checks and making concurrency-related errors detectable at compile-time.

3.1.3 Zero-cost abstractions

Zero-cost abstraction is a fundamental concept in modern programming languages like Rust, emphasizing the ability to use high-level abstractions without incurring any runtime overhead. In essence, it means that developers can leverage expressive and abstract constructs in their code, such as advanced data structures or complex algorithms, without sacrificing runtime performance.

This is achieved through the compiler's optimization capabilities, which eliminate the abstraction overhead during the compilation process, resulting in highly efficient machine code.

In expansive collaborative environments like Mitmproxy, ensuring code clarity, readability, and efficiency is paramount, contributing significantly to the project's scalability, maintainability, and overall collaborative success.

3.1.4 Compatibility with Python

The choice of Rust as the primary language for the low-level components of Mitmproxy is also driven by its seamless compatibility with Python.

This compatibility facilitates a smooth transition between the existing Python codebase and the newly incorporated Rust components, ensuring a cohesive and synergistic development environment. Rust's interoperability with Python enables the two languages to work in tandem, leveraging the strengths of both.

The Rust programming language provides Foreign Function Interface (FFI) capabilities that allow developers to create bindings and call Rust functions from Python, and vice versa. This interoperability ensures that Rust and Python

components can work together cohesively, enabling developers to leverage the strengths of each language where it is most beneficial.

To achieve this, Rust exploits the capabilities of PyO3, a Rust crate that provides bindings to the Python interpreter, allowing a Rust program to call Python code and vice versa. The implementation is very simple, it is sufficient to mark the function with the `pyfunction` macro and the `pymodule` macro to mark the module. PyO3 *functions* refers to a Rust function callable from Python as if it were a native Python function, a PyO3 *module* is a Rust module that contains one or more functions, which can then be imported and used in Python as extension module.

3.2 Repository structure overview

The organizational structure of the repository is designed to ensure a clear separation between the Rust core and its companion applications. Each operating system (OS) has its dedicated folder, facilitating an organized arrangement. The Rust core, constituting the primary project, is situated in the conventional *src* folder, adhering to the typical structure of a Rust project.

For the macOS companion application, both binaries and source code are contained within the *mitmproxy-macos* folder. On MacOS, distribution of applications are allowed only after a signature process. This is done through private, non-sharable keys, which is why the *Redirector app* is available both as precompiled and as source code. The compilation process is entrusted to the GitHub CI system, and the compiled code is then uploaded to the wheel. Similarly, the Windows companion app resides in the *mitmproxy-windows* folder, encompassing both binary executables and the corresponding source code. In the case of the Linux companion app, the *mitmproxy-linux* folder exclusively houses the source code. This approach is facilitated by the GitHub *actions* system, which streamlines the compilation process.

The Foreign Function Interface (FFI) components are consolidated in the *mitmproxy-rs* folder, maintaining consistency with the naming convention of the corresponding Python module, `mitmproxy-rs`. This section encapsulates the interoperability between Rust and Python.

This hierarchical organization enhances clarity, ease of navigation, and modularity, allowing developers to efficiently work on distinct components while maintaining a cohesive structure.

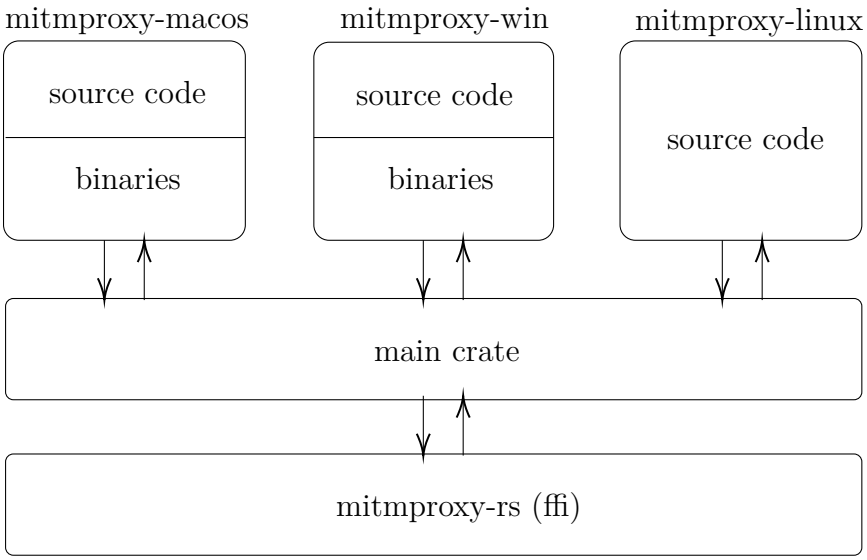

Figure 3.1: Mitmproxy repository structure

Chapter 4

MacOS implementation

MacOS implementation consists of three main parts: the first, called *mitmproxy_rs*, is responsible for forwarding packets to and from the Mitmproxy core, and it sits between the *Redirector* and the Python part of Mitmproxy. The second part, called *Redirector*, is the part that communicates with the OS, it is responsible for the interception and reinjection of packets. The third part deals with *Inter Process Communication* between the first two. The general architecture is illustrated in Figure 4.1.

4.1 The Rust side of Mitmproxy: *mitmproxy_rs*

It can be seen from figure 4.1 that *mitmproxy_rs* is divided into three parts: macOS Transparent mode, Python API and User space network stack

4.1.1 User space network stack

This part receives raw IP packets and processes them by creating UDP datagrams and TCP segments to be sent to the proxy core and vice versa. The two main crates used in this module are *Tokio* [8] and *Smoltcp* [9].

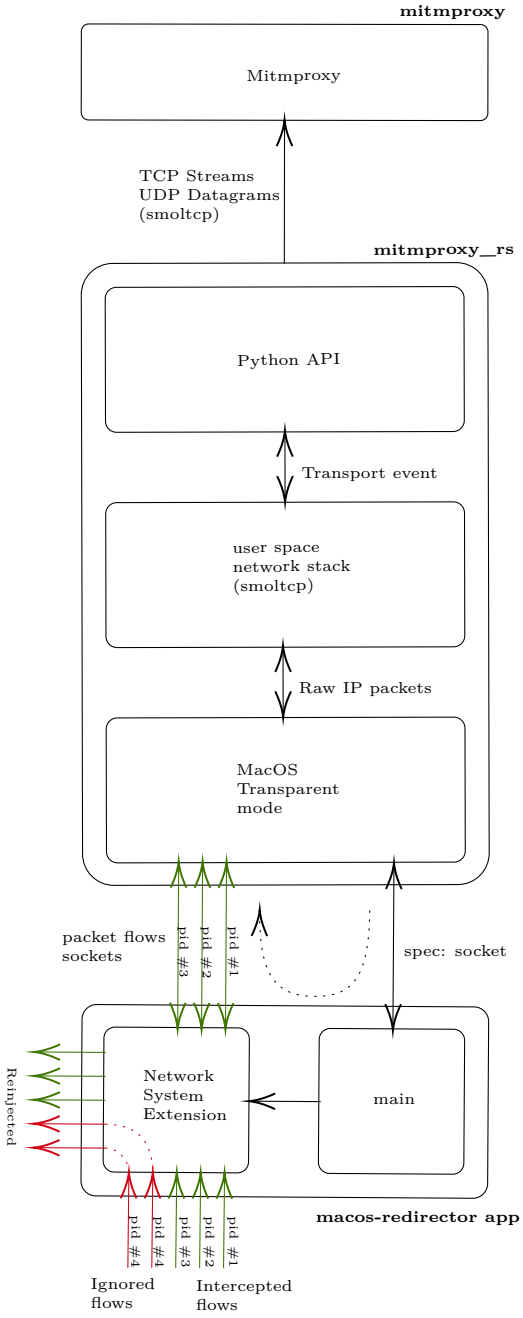

Figure 4.1: macOS implementation

Tokio

Tokio is an event driven and non-blocking I/O platform designed to write asynchronous applications. Since in *mitmproxy_rs*, many tasks are performed asynchronously but need to communicate with each other, *Tokio* is used with *Tokio::Sync* [10] feature that provides synchronization primitives.

The communication methods used are essentially *Broadcast* Channels and *Multi Producers Single Consumer* (MPSC) channels: the first one involves multiple senders sending data to multiple receivers while the second one involves multiple senders sending data to a single receiver. MPSC channels can be *with back-pressure* if there is a congestion control or *unbound* if there is no congestion control. It is good to note that they can also be used without any problem for the use case with a single producer and a single consumer.

Broadcast channels are used to trigger and propagate the *shutdown* event, while the main use-case for MPSC channels is the communication between *Mitmproxy* and *mitproxy_rs*.

There is one channel from *Mitmproxy* to each TCP connection within *mitmproxy_rs* to send commands related to the connection itself and because the information transmitted is connection-specific, each channel is logically and physically separated from the others: this is the typical *single-producer, single-consumer* use-case.

There is also a channel that departs from the individual connection and is directed to *Mitmproxy* for sending datagrams; in this case the information is primarily useful to *Mitmproxy*, consequently all connection channels are logically considered a single-producer and the communication is the standard MPSC use-case.

The figure 4.2 illustrates a simplified diagram of MPSC channels between *Mitmproxy* and *mitmproxy_rs*.

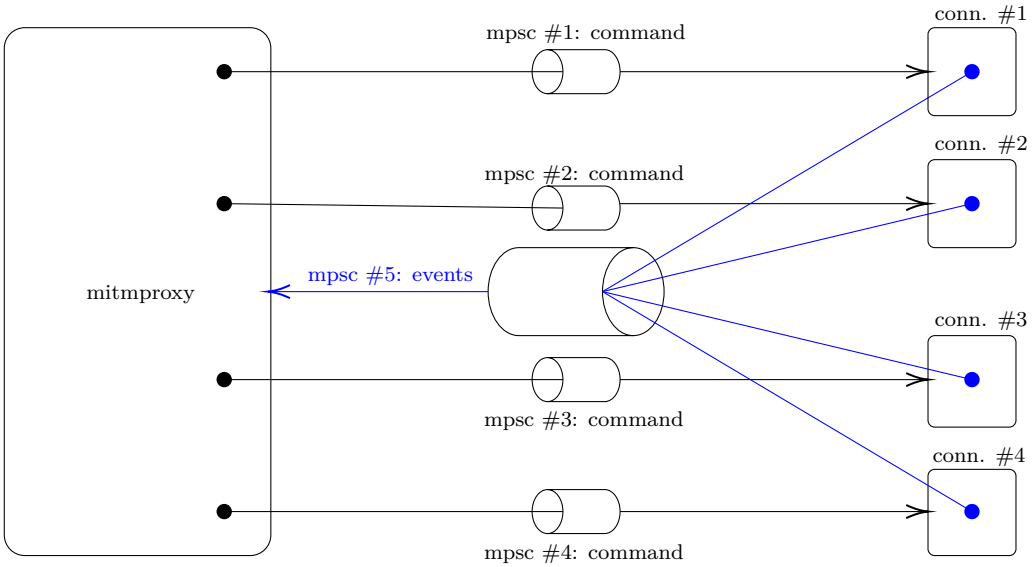

Figure 4.2: MPSC channels between *Mitmproxy* and the *mitmproxy_rs*

Smoltcp

Smoltcp [9] is an event-driven TCP/IP stack that is designed to be used in bare-metal, real-time systems with emphasis on simplicity and performance.

In *mitmproxy_rs* it allows the implementation of the network and socket layers in user space. Specifically, it is used to create a network I/O that receives and sends packets to and from the sockets connections.

The network I/O is a structure that contains, among others things, 1. a `std::collections::HashMap::<ConnectionId, SocketData>` with all the keys of the connections coupled to all the sockets that are associated with the connection itself, 2. a device to buffer packets and to prepare them for transmission and 3. a MPSC bounded channel to communicate with the proxy core.

Going down the TCP/IP stack, *smoltcp* provides the ability to build IP packets, check their validity and build TCP segments and UDP datagrams from IP packets and vice versa.

4.1.2 Packet source

This step of the *mitmproxy_rs* module is the one responsible for direct communication with the *Redirector*.

This module begins by creating the Unix socket with which communication with the *Redirector* app will take place.

After this step, the socket officially runs the *Redirector* app, which must be located in the *Applications/* folder, this particular requirement of the macOS implementation is due to the fact that on macOS there is a subtle restriction for some APIs that allows them to run correctly only if the *.app* executor is located within the *Applications/* folder.

The *Redirector* app is started by passing via arguments also the path to the Unix socket that allows the inter process communication between the two.

After the application startup, the *packet_sources* module creates two types of structures:

1. The parent structure responsible for the *redirector* app communications and for creating and initiating the connections. It runs a loop that, with the help of the *tokio* crate, listens for the arrival of new and possible configurations to be sent to the redirector, listens for the arrival of new transport commands to handle existing connections, and listens for the arrival of new connections to be created and launched by the *redirector*. Specifically, this structure stores each packet flow in a *std::collections::HashMap* with the corresponding `ConnectionId`, the unique key in the map.
2. The *connection* structure that allows the single connection to be managed by storing the socket and MPSC channel to communicate with the proxy core. This structure is created when the *Redirector* app sends a packet that belongs to a connection that has not yet been registered.

The figure 4.3 illustrates a simplified schema of the *packet_source* module:

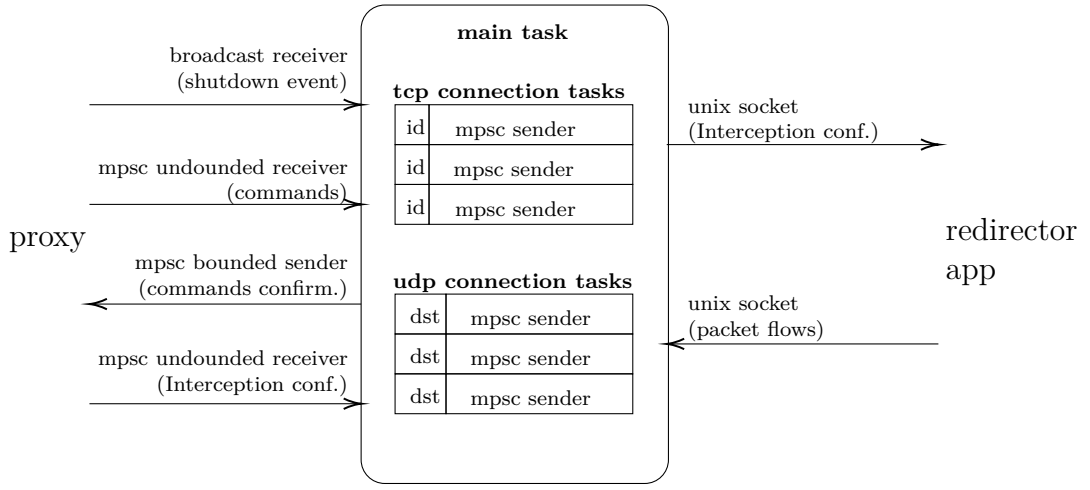

Figure 4.3: Packet source module schema

4.1.3 Python API

The communication with mitmproxy is handled with the *PyO3* [11] crate, which provides Rust bindings for the Python interpreter for the purpose of allowing native Python modules to be written directly in Rust.

PyO3 provides the ability to create Python functions and modules in Rust and to call them from Python very trivially: marking them only with specific macros. It handles the conversion of data between the two languages independently by automatically converting Python objects to Rust counterparts and vice versa (e.g. `PyString` in `String` and Python `List` in Rust `Vec`).

It allows a correct *error propagation* between the two languages: Python `Exceptions` are converted in Rust `Result::Err` and `Result::Err` are converted in Python `Exceptions`.

Moreover, it deals with the memory management of Python objects, counting the references to objects and dropping them when the reference count reaches zero

Among other things, it takes care of the *Global Interpreter Lock* (GIL) [12], a mechanism used by the *CPython* [13] interpreter: it is a *mutex* that prevents multiple native threads from executing Python byte codes at the same time, normally this is a performance constraint in Python, since the lock makes it easier for the interpreter to be multithreaded, at the expenses of the true parallelism.

For this reason with the *PyO3* crate it is possible to release the GIL and to reacquire it when needed, thus executing truly parallel Python byte codes directly and simply using Rust code.

4.1.4 Listing of active processes information

Process selection is another challenging but parallel part that is not directly inherent to the proxy itself, but competes with a feature that *mimtproxy* provides.

In fact, the user can directly specify the processes he or she wishes to intercept.

The idea is to have a constantly updated list of active processes showing some information about each process.

To further enhance the user experience it was chosen to also show the application icon, obviously only if present.

Processes information

Information about processes and icons are stored in two different structures and are handled separately.

The process structure stores some useful information such as the *path* to the *executable*, the *display_name* string, the boolean variable *is_visible* which indicates whether the process has a visible opened window and the boolean variable *is_system* which indicates whether the process is a system process or not.

The first and the second variable are obtained by using the *sysinfo* [14] crate. This crate allows obtaining system's information for all major OS, and it allows listing all processes installed and running on the machine. On macOS the *sysinfo* crate obtains the details of each process by calling *proc_pidbsdinfo* [15]: a system call that returns a lot of data about a specific process, including information such as resource usage, process execution status, and other attributes.

Through the *sysinfo* crate is it possible to obtain all the necessary information directly, except for the *is_visible* attribute, in fact of macOS systems this variable is bound to the window element and not to the process itself.

For this reason it is necessary to go to a lower level and use the *Core Graphics*

[16] API, wrapped in the *core_graphics* crate in rust [17], which provides a lot of functions to manage windows and screens.

The *core_graphics* crate provides the ability to get the list of all windows and the process identifier that owns each visible window; all given PIDs are then stored in a `std::collections::HashSet`, when it is desired to know whether a process is visible or not just check if the PID of the process is present the set.

Processes icons

The icons are not directly reachable from the *proc_pidbsdinfo* system call, so a different approach must be used. Of course, the icons are not files stored in the *.app* folder but are embedded directly in the compiled binary file of the application.

The *appkit* macOS framework provides the class `NSRunningApplication`, with an important property called *icon* that returns the application icon as `NSImage`, this property though is only accessible using the Swift or Objective-C language, i.e., the language underlying all Apple systems.

In Rust, there is the *objc* crate, a wrapper of the objective-c language, with which it is possible to call the *icon* method on the `NSRunningApplication` class obtaining the icon as `Id`, a generic object in Objective-C.

It is then necessary to convert the `Id` to an `NSImage` and convert the `NSImage` to a `TIFFRepresentation`, which is a representation of an image in memory using the *TIFF* data format: the standard format used by macOS to store icons image.

Finally, it is possible to convert the `TIFFRepresentation` to a `Vec<u8>` that is the raw representation of the icon in memory.

Since *Mitmproxy* uses *PNG* icons, it is needed to convert the icons from *TIFF* to *PNG*, this is done with the use of *image* crate [18] which provides many functions to manipulate raw images. Conversion is also possible using objective-c functions, but the *image* crate is faster [1]

[1]The complete process of getting the PNG data using the *image* crate for conversion takes about [57.228 ms 58.251 ms 59.329 ms] while the process with objective-c only takes [69.748 ms 70.997 ms 72.255 ms], benchmarks have been done on 100 iterations with *criterion* [19] crate

4.2 Redirector

The *Redirector* is the actual core of the macOS implementation, it is responsible for the interception and reinjection of packets.

The initial idea was to keep the codebase in Rust, to avoid communication issues between processes and to avoid big complications in the implementation.

This approach was abandoned because the macOS ecosystem is easier to handle with Swift or Objective-C than with Rust, above all the interaction with the system APIs.

4.2.1 First approach forcing traffic redirection to Utun

TUN/TAP interfaces [20] are virtual network devices at kernel layer. They were originated as *Universal TUN/TAP Driver* in 2000 as a merge of the corresponding drivers in Solaris, Linux and BSD [21]. The driver is still maintained and currently part of the Linux and FreeBSD kernel.

The key difference between TUN and TAP interfaces is that a TUN interface is at *Network Layer* of the ISO/OSI model while the TAP interface is at *Data Link Layer*.

On macOS, only the TUN interface is natively supported, and it is called *UTUN*. This approach was discarded because it was not possible to force a little and specific portion of the traffic to go through the *UTUN* interface, but the filter was applied to all the traffic only after the interception and this is not optimal.

The ideal situation would be to have a filter that intercepts the traffic before it is sent, in this way it should be possible to simply ignore the packets that are not of interest without sending them to the proxy and reinjecting them immediately afterward.

4.2.2 Second approach with Apple Security Extension

The approach with *Apple Security Extension* is definitely the most intricate from an intuitive point of view but also the smoothest from a technical perspective.

It assumes to use as much as possible what Apple provides without finding shortcuts and giving up the idea of having a standard, common approach for all Unix based systems.

The language used to implement this approach is Swift, a standard, along with Objective-C, for Apple OSs.

The key of this approach is the *Apple Network Extension* [22]: a framework that provides a set of APIs to customize and extend the core networking features of Apple OSs.

The framework is very big and to implement the *Redirector* app only the part related to the Virtual Private Network (VPN) is used with a particular focus on the different *Provider* types: the *Packet Tunnel Provider* [23] and the *App Proxy Provider* [24].

Packet Tunnel Provider

This *Packet Tunnel provider* allows the management of packets at the *Network Layer*. To use it is necessary to include the network extension entitlement and add the corresponding capability within the Redirector app.

Before launching the provider it is necessary to set a proper configuration by setting the `NETunnelProviderManager` [25] object, in this object is it possible to set the `appRules`: a vector of `NEAppRule` elements, that will be used to filter the traffic.

The problem with `NEAppRule` is that the rule cannot be specified during the interception of the flow, but it must be set before the interception.

This new approach is better than the former because it is managed with official APIs, but it is still not optimal.

One of the goals is to change the `InterceptConfig` at runtime, then it is necessary to filter the traffic inside the Network extension and not just at the beginning.

App Proxy Provider

The App Proxy Provider allows the management of packets at the *Transport Layer*, one layer above the *Network Layer*.

Even if it could be strange to manage the packets at the *Transport Layer* instead of the *Network Layer*, `NEAppProxyProvider` is the only *Provider* that allows a deep management of packets flows at runtime. Moreover, `NEAppProxyProvider` provides the class `NETransparentProxyProvider` [26]: an inherited class, available on macOS 11.0 and later, which has a critical benefit: when inside the `handleNewFlow` method `False` is returned, the flow is not rejected and blocked as it happens in the `NEAppProxyProvider`, but it simply ignored and processed normally as it be never intercepted. This is exactly the behavior that is desired.

4.2.3 From extension to system extension

On macOS there is a distinction between *extension* and *system extension*, specifically to NE, the simple *extension* allows the distribution only through the *App Store*, this is a big limitation for the *Redirector* app, which being only a companion app, it would be useless to be distributed through the official Apple App Store.

The transition from the extension to the system extension, allow on macOS 10.15 or later a distribution outside the *App Store* by signing the app with a developer ID.

Of course both versions are distributable only after signing, and this is the reason why the *Redirector* is distributed as binary and not only as source code.

4.3 Inter process Communication

Also, the Inter process communication is OS specific. On Windows named pipes (more details in appendix C) are used, and this choice affected the adoption, in the very early version, of Unix pipes on macOS as well. Successively the choice was changed to Unix sockets and in this section the reasons will be explained.

Serialization and of packets is done using *Protocol Buffers* [27] on each platform. After the adoption of deserialization protobuf on macOS version, the Windows version was also updated accordingly.

4.3.1 Data structures

Protobuf is a serialization and deserialization mechanism developed by Google [27]. The structure must be written in a *PROTO* file and then compiled with the *protoc* compiler. *Protoc* compiler translate the *PROTO* file into a language specific file, and it is available for all major programming languages. In this case the *PROTO* file is compiled in Rust and Swift files.

All the communications between the *Redirector* and the *mitmproxy_rs* are serialized and deserialized using Protobuf. There are structures for the configuration of the *Redirector* with the list of the *pids*, the *process_name*, and of course there is the structure to serialize and deserialize inbound and outbound packets.

4.3.2 Unix pipes

Unix-like pipelines are a mechanism for inter-process communication in Unix systems. A pipeline is unidirectional, and it is composed by two file descriptors, the read-only end and the write-only end. The unidirectionality is not the only difference with the *named pipes* used in the Windows implementation, the main difference is that Unix pipes survives only as long as the process that created them is alive, while the named pipes are persistent and can be used by multiple processes.

In the Windows implementation the named pipes are used to communicate between processes because the communication is quite straightforward, as read and write operations. On macOS and Linux the main problem is that it is necessary to create two pipes for each communication between the *Redirector* and the *mitmproxy_rs*, considering configurations and packets, the unidierectionality of the pipes is a big limitation.

For this reason, a subsequent update altered the behavior to employ Unix sockets instead of Unix pipes.

4.3.3 Unix sockets

Unix sockets are a mechanism for inter-process communication, they are also known as *IPC Sockets* or *Unix Domain Sockets* (UDS).

UDS are literally handled as files, they are created from a path, and they are

deleted when the process that created them is terminated.

Main reason for the adoption of UDS is that they are bidirectional, but there are also three little disadvantages: the first is that in terms of performance they are a little slower [28], the second is that they are not available on Windows and the third is that they could be considered a little bit more complicated to use.

Disadvantages are not a huge problem because the performance difference is not noticeable and since the goal of the *Redirector* app is not to be cross-platform, but to be available on macOS, Unix sockets are a good choice.

Chapter 5

Linux implementation

On Linux, the development is in an even more initial phase. An official version hasn't been released yet, and both the code and the development process could undergo significant changes.

Unlike macOS, Linux doesn't have any API that can be used to achieve the goal. For this reason, it was necessary to leverage a capability of the Linux kernel: *extended Berkeley Packet Filter*[29] (EBPF).

5.1 eBPF

Extended Berkeley Packet Filter (EBPF), is a versatile and programmable lightweight virtual machine deeply integrated into the Linux kernel, initially found its roots in efficient network packet filtering. Over time, its capabilities have expanded, transforming it into a technology that facilitates dynamic and secure execution of custom code within the kernel space. This evolution has positioned EBPF as a pivotal tool, offering fine-grained visibility and control over system events, thereby making it instrumental for diverse applications such as performance monitoring, security enhancement, and network analysis.

EBPF's programmability, coupled with its minimal performance overhead, has resulted in its widespread adoption across various domains, contributing significantly to advancements in observability and security within modern computing environments. Its modular and extensible design allows developers to harness

its power for a myriad of purposes, extending beyond its initial focus on packet filtering.

EBPF programs come in various types, each catering to specific functionalities within the kernel. Here's a brief overview:

- *Probe Programs*: These programs attach to specific kernel functions, enabling developers to collect data or perform actions when those functions are executed, there are two different types of probe programs: 1. *Kprobe*: Kernel probe programs attach to kernel functions, allowing for the interception of kernel function calls and the collection of data. 2. *Uprobe*: User probe programs attach to user-space functions, enabling the interception of function calls and the collection of data. A further distinction can be made between *Probe* and *ProbeRet* programs, the first ones are executed before the probed function while the second ones are executed after the probed function, collecting the return value of the probed function.
- *Tracepoints*: Tracepoint programs capture events at predefined points in the kernel, offering insights into the system's behavior and performance.
- *Socket Programs*: Designed for manipulating network sockets, these programs provide control over network traffic at the kernel level.
- *Classifiers* or *Traffic Control Classifier* (TC): eBPF programs can be used with *Traffic Control* to classify and manage network traffic based on specific criteria, Traffic Control enables sophisticated management of network traffic, allowing administrators to exert fine-grained control over bandwidth allocation, packet prioritization, and quality of service parameters. EBPF programs used within Traffic Control serve as dynamic classifiers, facilitating the categorization and prioritization of network packets based on predefined criteria. This integration empowers system administrators to optimize network performance, enhance resource utilization, and enforce tailored traffic management policies.
- *Cgroup* Programs: Control Groups, commonly referred to as Cgroups, are a Linux kernel feature that provides a framework for organizing and managing system resources. Cgroups allow administrators to impose limits, track usage, and allocate resources such as CPU, memory, and I/O among processes or groups of processes. This hierarchical and flexible resource management system enhances system performance, ensures fair resource distribution, and facilitates the implementation of resource usage policies. By leveraging *Cgroups*, developers can achieve better control over system resources, prevent resource contention, and gain valuable insights into the resource consumption patterns of various processes, contributing to improved system stability and efficiency.

- *Cgroup SKB*: These programs specifically operate within the Socket Buffer subsystem of cgroups,
- *eXpress Data Path* (XDP): EBPF programs in XDP allow for packet processing at the earliest point within the networking stack, enhancing performance and reducing latency.
- *Linux Security Modules* (LSM): Linux Security Modules, commonly known as LSM, form an integral part of the Linux kernel's security architecture. LSM provides a modular framework that allows the implementation of various security enhancements and access control mechanisms. Through LSM, the Linux kernel supports the integration of multiple security modules, each designed to enforce specific security policies. LSM facilitates the enforcement of mandatory access controls, integrity checks, and other security measures by allowing kernel developers to integrate their security modules seamlessly. These modules can augment or replace the default Linux security mechanisms, tailoring the security posture of the system to meet specific requirements. The extensibility of LSM makes it a powerful tool for implementing diverse security policies, and its integration with EBPF allows for dynamic and programmable security controls, further fortifying the overall security of the Linux operating system.

5.1.1 XDP program

The *eXpress Data Path* (XDP) represents a groundbreaking feature of the EBPF framework, offering ultra-fast packet processing directly at the network driver layer. XDP programs are executed extremely early in the network stack, right at the point of packet reception, allowing for lightning-fast decision-making on incoming packets.

XDP programs are attached to specific network interfaces and operate as a set of rules or filters applied to incoming packets. They execute in the kernel space, offering unprecedented efficiency and speed. These programs can efficiently 1. drop 2. modify 3. redirect 4. pass packets to the networking stack based on customizable criteria defined within the program logic.

Due to the early nature of packet handling in XDP, extracting details about the source process—such as *Process Identifier* (PID) poses significant hurdles. This limitation stems from the network packets being processed before the Linux kernel associates them with specific process contexts, requiring alternative strategies to be employed to achieve the desired functionality.

Another limitation of XDP is that it can not operate for *egrees* packets.

5.1.2 TC program

The *Traffic Control* (TC) Classifier program is as a sophisticated EBPF tool for fine-grained control and manipulation of network traffic within the Linux kernel. Leveraging EBPF's capabilities, TC Classifier programs extend the traditional functionalities of Linux's Traffic Control infrastructure, allowing for intricate packet classification and processing.

The main difference with XDP is that TC programs operate on both *ingress* and egress packets, this bidirectional handling grants TC programs a comprehensive control over network traffic, enabling them to apply diverse policies and actions to both incoming and outgoing packets. Unlike XDP, which primarily focuses on packet processing at the earliest possible stage upon arrival, TC programs offer a more nuanced approach by allowing manipulation and classification at various points in the network stack.

This versatility makes TC a valuable tool for implementing the desired functionality.

The problem of getting the PID of the process that generated the packet is still present.

5.1.3 Probes

EBPF *probes* are a powerful EBPF programs, providing a flexible and non-intrusive mechanism for debugging and analysis within the Linux kernel. These *probes* serve as programmable hooks strategically positioned at crucial junctures in the kernel, allowing for the interception and observation of diverse events, system calls, and functions.

Notably, EBPF supports both user-level (UProbe) and kernel-level (KProbe) probes, each offering distinct advantages.

Leveraging *Uprobe* facilitates the interception and analysis of function calls at the user-space level, offering valuable insights into user-space applications. This capability proves advantageous when aiming to understand and monitor system behavior during specific events.

With *Kprobe* is possible to hook a kernel function call and read the function arguments. This could be useful to get the PID of the process that generated the packet. A possible approach could be to hook the `security_sock_rcv_skb` [30] kernel function, which is called when a packet is received, and read the PID of the process that generated that packet.

Once the PID is known is it possible to store it in a map, *eBPF maps* are the standard way to communicate between two EBPF programs.

5.1.4 EBPF maps

In the intricate landscape of EBPF, one of the key mechanisms facilitating efficient communication between different EBPF programs is the concept of EBPF *maps*. These *maps* serve as dynamic data structures within the kernel that enable sharing information across various EBPF programs, fostering collaboration and synchronization in a secure and controlled manner.

EBPF *maps* can be visualized as shared memory regions accessible by multiple EBPF programs, providing a means for them to exchange data seamlessly. These *maps* come in various types, each tailored to specific use cases and data requirements. Common types include *array maps*, *hash maps*, and *CPU maps*, each offering unique advantages in terms of data organization and retrieval efficiency.

Array maps

Array maps [31] in EBPF serve as straightforward, indexed storage structures accessible by multiple EBPF programs. Each element in the array is indexed by an integer, allowing for efficient and direct access to stored data. This type of map is particularly useful in scenarios where a simple numerical index suffices, making it a pragmatic choice for straightforward data organization and retrieval.

Hash maps

Hash maps [32] provide a more versatile option by allowing the storage of key-value pairs. This flexibility facilitates the association of data with specific keys, enabling EBPF programs to manage more complex relationships. *Hash maps* are advantageous in situations where the data retrieval pattern is not strictly

numeric, offering a dynamic and efficient means for EBPF programs to exchange and manipulate information.

CPU maps

CPU maps [33] are designed for scenarios where data needs to be maintained on a per-CPU basis. This type of map enhances efficiency by minimizing contention between different CPU cores. By assigning dedicated storage for each CPU, per-CPU maps enable parallel processing, reducing the need for synchronization mechanisms. This makes them particularly beneficial in scenarios where scalability and performance optimization are critical considerations for inter-program communication in EBPF.

In the context of this project, *hash maps* were leveraged to implement a robust filter mechanism, allowing for the selective interception of specific process identifiers (PID) or *process names* associated with corresponding *local port* addresses. This strategic use of *hash maps* enhances the filtering capabilities within the EBPF program, enabling the targeted interception of desired processes and contributing to the project's overarching goal of tailored and precise packet interception based on specified criteria.

5.2 Rust AYA

Rust *Aya* is a specialized EBPF library designed for perfect integration of user-supplied programs within the Linux kernel. In contrast to other eBPF libraries, *Aya* distinguishes itself by being crafted entirely in Rust, without dependencies on *libbpf* or *bcc* [34]. The library prioritizes operability and developer experience while leveraging Rust's capabilities and safety features previously discussed.

This is the library chosen for the implementation of the *transparent* mode on Mitmproxy and this decision is driven by its synergy with the existing codebase. Rust's safety features, expressive syntax, asynchronous support, cross-kernel compatibility, and deployment efficiency contribute to a seamless integration process, fostering code coherence, maintainability, and an overall positive development experience for the Mitmproxy project.

Chapter 6

Conclusions

6.1 Future improvements

6.1.1 MacOS

In the future, as the implementation for the macOS moves beyond the beta version, users can expect various enhancements in line with the major versions of Mitmproxy. These improvements will likely fix existing bugs, refining the overall functionality and user experience. In addition, performance improvements are expected to optimize the general efficiency.

6.1.2 Linux

At the moment the approach that involves the use of a *TC* program and a *Kprobe* is the most promising but still not tested enough. The goal is to complete the implementation and test it to obtain a result similar to the one obtained on macOS. Another missing implementation for Linux is an automatic certificate installer, to make the *Transparent mode* truly transparent.

6.2 Contributions

Throughout the development process, extensive collaboration with Dr. Maximilian Hils significantly influenced and enhanced the project. Dr. Hils played a pivotal role in code reviews, offering valuable feedback, and actively contributing to the improvement of various aspects. The key contributions include about 7 main pull requests:

1. The implementation of the first version of the *Redirector* application [35], which is responsible for redirecting traffic to the proxy, implemented in swift.
2. The code that handles the compilation process of the application and that copies the binaries into the wheel [36].
3. The first implementation of the communication mechanism between Rust and Swift with Protobuf [37] and a later update [38], the integration of *protocol buffer* into the codebase, both rust and swift side.
4. The contribution to the Rust wrapper of the Apple NE [39], that are used in Mitmproxy to manage the automatic installation and trust of certificates.
5. The implementation of the *certificate installer* [40], written in Rust.
6. The main roadmap for the development of the *Transparent mode* [41]
7. Tests and first implementation of the *Transparent mode* for Linux [42]

For the macOS portion, the codebase contribution amounts to an addition of more than 6,000 lines of code. In parallel, the Linux segment of the project also saw a substantial increase in its code base, with about 10,000 lines of code added.

About 30% of Mitmproxy users, out of an estimated total of 500,000 users, use the macOS version of the software. This user estimate is derived from the number of downloads registered on PyPI [43], which provides an indicator of the adoption and reach of the software.

The percentage of users using the Linux version, on the other hand, is much more significant and covers about 60% of the Mitmproxy user community, this is because the software is also widely used on the server side. As with the macOS component, the estimated Linux user base is informed by PyPI download statistics.

Aside from these main contributions I have been and still am involved in maintaining the entire repository, as the *mitmproxy_rs* development leader for both Linux and macOS, I am involved in code review, bug fixes, and answering questions and special requests.

Appendix A

MacOS Certificate Truster with system functions

To make the proxy truly transparent, the functionality to automatically install certificates within macOS systems has been added.

On macOS, the system certificates are stored in the keychain. The keychain is a password management system that is used to store passwords and other sensitive data like certificates, private keys, and so on.

Mitmproxy uses the system keychain to store the certificate which allow HTTPS interception and private key to correctly compile the Redirector app.

Keychain system is part of the Security framework, the security framework is a set of functions that allow to manage security aspects of the macOS systems.

Rust provides a wrapper for the Security framework [44], but some additional functions are needed to correctly install certificates: `SecCertificateAddToKeychain` and `SecTrustSettingsSetTrustSettings`.

SecCertificateAddToKeychain

`SecCertificateAddToKeychain` [45] is a function that allows to add a custom certificate to the chosen keychain. This function is a system function and for this reason it has been added to the *security_framework_sys* crate, the crate that

provides the bindings directly to the system functions.

Its purpose is much wider than the one needed by Mitmproxy, but it has been added to the crate to make it available to everyone.

For Mitmproxy scope, a caller function has been added to the higher level crate *security_framework*, under the *certificates* crate: `add_to_keychain`, a new implementation function for `SecCertificate` that accepts the desired target `SecKeychain` as argument and returns a `Result` indicating the outcome of the operation.

SecTrustSettingsSetTrustSettings

`SecTRustSettingsSetTrustSettings` [46] is a function that allows to set the trust settings to a specific certificate.

As for `SecCertificateAddToKeychain`, this function is a system function, it has been added to the *security_framework_sys* crate and the related caller function `set_trust_settings_always` has been added to the *security_framework* crate, it accepts the desired target `SecCertificate` as argument and returns a `Result` with the outcome of the operation.

Appendix B

CI/CD

CI/CD is a practice that allows to automate the process of building, testing and deploying. Mitmproxy, and in particular *mitmproxy_rs* uses GitHub Actions [47] as CI/CD platform and performs *auto-fixing*, *building* and *docs*.

Autofix

Autofix process is triggered by a pull request and it checks the code for formatting errors. The code is formatted using *rustfmt* [48] and *clippy* [49], but the auto-fix process is also responsible for *protocol buffer* files compilation.

Auto-fix is performed for each OS, in fact each OS has its own *auto-fix* job and when auto-fix is completed, the result is automatically sent to the codebase with a new commit.

Build

Build process is the most important one, it is triggered by a pull request and, for each OS, it launches tests and builds the binaries. After compilation process, the binaries are sent to the *artifacts* section of the pull request.

Artifacts section is a special section of the pull request that allows to upload files (in this case the binaries), that can be downloaded by the user but also used

by other jobs.

Docs

This process is triggered by a pull request too and its purpose is to build the documentation and upload the static HTML files as an artifact.

Appendix C

Windows Named Pipes

Named pipes are a typical inter-process communication mechanism. On Windows, *named pipes* are considered like usual files, and they are managed with same functions used for files. The first difference between Windows named pipes and Unix pipes is that Windows named pipes are bidirectional, while Unix pipes are unidirectional.

In addition, since on Windows, pipes have a persistent file system name, they can be written and read by different processes at different times.

Acknowledgements

To Elena, my unwavering support, trusted shoulder and kindred spirit.

To my family, for their continuous encouragement.

To Professors Torchiano, for his guidance and advice.

To Maximilian Hils, a big thanks for his mentoring on this journey, for his expertise, kindness and patience.

Bibliography

[1] Aldo Cortesi. *Mitmproxy docs v10*. Introduction paragraph. 2010. URL: https://docs.mitmproxy.org/archive/v10/ (cit. on pp. ii, 3).

[2] *What is eBPF?* URL: https://ebpf.io/what-is-ebpf/#what-is-ebpf (cit. on p. iii).

[3] S. Kalarani and G. V. Uma. «Improving the efficiency of retrieved result through transparent proxy cache server». In: *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*. July 2013, pp. 1–8. DOI: 10.1109/ICCCNT.2013.6726674 (cit. on p. 1).

[4] Muhammad Ikram, Narseo Vallina-Rodriguez, Suranga Seneviratne, Mohamed Ali Kaafar, and Vern Paxson. «An Analysis of the Privacy and Security Risks of Android VPN Permission-Enabled Apps». In: *Proceedings of the 2016 Internet Measurement Conference*. IMC '16. Santa Monica, California, USA: Association for Computing Machinery, 2016, pp. 349–364. ISBN: 9781450345262. DOI: 10.1145/2987443.2987471. URL: https://doi.org/10.1145/2987443.2987471 (cit. on p. 1).

[5] Maximilian Hils and Rainer Bohme. «Watching the Weak Link into Your Home: An Inspection and Monitoring Toolkit for TR-069». In: *CoRR* abs/2001.02564 (2020). arXiv: 2001.02564. URL: http://arxiv.org/abs/2001.02564 (cit. on p. 1).

[6] Aldo Cortesi. *Mitmproxy docs v10*. 2018. URL: https://docs.mitmproxy.org/archive/v10/ (cit. on p. 4).

[7] *Official mitmproxy repository*. URL: https://github.com/mitmproxy/mitmproxy (cit. on p. 7).

[8] *Tokio crate documentation*. URL: https://docs.rs/tokio/latest/tokio/ (cit. on p. 13).

[9] *Smoltcp crate documentation*. URL: https://docs.rs/smoltcp/latest/smoltcp/ (cit. on pp. 13, 16).

[10] *Tokio sync module documentation*. URL: https://docs.rs/tokio/latest/tokio/sync (cit. on p. 15).

[11] *Pyo3 crate documentation*. URL: https://docs.rs/pyo3/latest/pyo3/ (cit. on p. 18).

[12] *Global interpreter lock*. URL: https://docs.python.org/3/glossary.html#term-global-interpreter-lock (cit. on p. 18).

[13] *Cpython definition*. URL: https://docs.python.org/3/glossary.html#term-CPython (cit. on p. 18).

[14] *System info crate documentation*. URL: https://docs.rs/sysinfo/latest/sysinfo/ (cit. on p. 19).

[15] *proc_pidbsdinfo struct*. URL: https://opensource.apple.com/source/xnu/xnu-2050.24.15/bsd/kern/proc_info.c (cit. on p. 19).

[16] *Apple documentation: core grpahics framework*. URL: https://developer.apple.com/documentation/coregraphics?language=objc (cit. on p. 20).

[17] *Core graphics crate documentation*. URL: https://docs.rs/core-graphics/latest/core_graphics/ (cit. on p. 20).

[18] *Image crate documentation*. URL: https://docs.rs/image/latest/image/ (cit. on p. 20).

[19] *Criterion crate documentation*. URL: https://bheisler.github.io/criterion.rs/book/getting_started.html (cit. on p. 20).

[20] Maxim Krasnyansky, Maksim Yevmenkin, and Florian Thiel. *Universal TUN/TAP device driver*. SPDX-License-Identifier: GPL-2.0, include:: <isonum.txt>.
Copyright 1999-2000
Maxim Krasnyansky <max_mk@yahoo.com>, Linux, Solaris drivers;
Copyright 1999-2000
Maksim Yevmenkin <m_evmenkin@yahoo.com>, FreeBSD TAP driver;
Revision of this document 2002
by Florian Thiel <florian.thiel@gmx.net>. 1999-2000 (cit. on p. 21).

[21] *Universal Tun Tap driver documentation*. URL: https://vtun.sourceforge.net/tun/faq.html (cit. on p. 21).

[22] *Apple documentation: Network Extension*. URL: https://developer.apple.com/documentation/networkextension (cit. on p. 22).

[23] *Packet tunnel provider*. URL: https://developer.apple.com/documentation/networkextension/packet_tunnel_provider (cit. on p. 22).

[24] *App proxy provider*. URL: https://developer.apple.com/documentation/networkextension/app_proxy_provider (cit. on p. 22).

[25] *Apple Documentation: NETunnelProviderManager*. URL: https://developer.apple.com/documentation/networkextension/netunnelprovidermanager (cit. on p. 22).

[26] *Apple Documentation: NETransparentProxyProvider*. URL: https://developer.apple.com/documentation/networkextension/netransparentproxyprovider (cit. on p. 23).

[27] *Protocol buffers documentation*. URL: https://developers.google.com/protocol-buffers (cit. on pp. 23, 24).

[28] *Inter Process Communication benchmark*. URL: https://github.com/goldsborough/ipc-bench/blob/589146a3f14f7675c2224ed47e414093bba13a69/README.md (cit. on p. 25).

[29] *Berkeley Packet Filter*. URL: https://en.wikipedia.org/wiki/Berkeley_Packet_Filter (cit. on p. 26).

[30] *security_sock_rcv_skb kernel function signature*. URL: https://elixir.bootlin.com/linux/v4.2/source/include/linux/security.h#L1129 (cit. on p. 30).

[31] *Array map in ebpf*. URL: https://docs.kernel.org/bpf/map_array.html (cit. on p. 30).

[32] *Hash map in ebpf*. URL: https://docs.kernel.org/bpf/map_hash.html (cit. on p. 30).

[33] *CPU map in ebpf*. URL: https://docs.kernel.org/bpf/map_cpumap.html (cit. on p. 31).

[34] *Tools for BPF-based Linux IO analysis, networking, monitoring, and more*. URL: https://github.com/iovisor/bcc (cit. on p. 31).

[35] *macos redirector app + CI*. URL: https://github.com/mitmproxy/mitmproxy_rs/pull/66 (cit. on p. 33).

[36] *Build app into wheel*. URL: https://github.com/mitmproxy/mitmproxy_rs/pull/67 (cit. on p. 33).

[37] *Protobuf integration*. URL: https://github.com/mitmproxy/mitmproxy_rs/pull/68 (cit. on p. 33).

[38] *Swift proto update*. URL: https://github.com/mitmproxy/mitmproxy_rs/pull/82 (cit. on p. 33).

[39] *Add SecTrustSettingsSetTrustSettings and SecCertificateAddToKeychain*. URL: https://github.com/kornelski/rust-security-framework/pull/181 (cit. on p. 33).

[40] *Add remove certificate*. URL: https://github.com/mitmproxy/mitmproxy_rs/pull/70 (cit. on p. 33).

[41] *Transparent Proxy Roadmap*. URL: https://github.com/mitmproxy/mitmproxy/issues/6531 (cit. on p. 33).

[42] *Mitmporxy transparent mode implementation and tests for linux*. URL: https://github.com/emanuele-em/mitmproxy_linux (cit. on p. 33).

[43] *PyPi metrics*. URL: https://pypistats.org/packages/mitmproxy (cit. on p. 33).

[44] *Rust Security Framework*. URL: https://github.com/kornelski/rust-security-framework/tree/54d905027e50ec4f0969824efa8e34581922a1f4 (cit. on p. 36).

[45] *SecCertificateAddToKeychain: Apple documentation*. URL: https://developer.apple.com/documentation/security/1396090-seccertificateaddtokeychain (cit. on p. 36).

[46] *SecTrustSettingsSetTrustSettings: Apple documentation*. URL: https://developer.apple.com/documentation/security/1399119-sectrustsettingssettrustsettings (cit. on p. 37).

[47] *Github actions*. URL: https://resources.github.com/ci-cd/ (cit. on p. 38).

[48] *Run rustfmt*. URL: https://github.com/rust-lang/rustfmt/tree/2174e6052dcb1802417d686140fe0ab7cbef0df2 (cit. on p. 38).

[49] *Run clippy*. URL: https://github.com/rust-lang/rust-clippy/tree/87aed038dad5f528f21a6b7baab7039e184386c1 (cit. on p. 38).