Department of Computer Science
University of Innsbruck

Bachelor Thesis

Implementing a WireGuard frontend for mitmproxy

Supervisor: Dr. Maximilian Hils

Fabio Valentini
MN 01018782

January 5, 2023

Abstract

Mitmproxy is "a free and open source interactive HTTPS proxy" [Cor+10]. The currently available option for transparently proxying traffic through mitmproxy (i.e. "transparent" mode [1]) is difficult to use correctly, and requires complicated steps for setup – on both the mitmproxy host and client device:

On the host, using "transparent" mode requires configuring custom IP routing and firewall rules – which requires administrative privileges, and permanently changes the operating system's networking configuration. On the client device, connecting to a mitmproxy instance running in "transparent" mode requires manual changes to a network interface's default routing configuration, i.e. setting the mitmproxy host as default gateway.

The implementation of a mode based on the WireGuard protocol provides functionality that is similar to the existing "transparent" mode, but which is easy to use, and does not require any manual changes to the network configuration on either the mitmproxy host or the client device. It runs entirely in user space, requires no firewall configuration changes when using default settings, and does not require administrative privileges to run or set up.

Running mitmproxy in this mode is officially supported on Windows, Linux, and macOS, and since WireGuard client applications are available for almost all platforms, this mode is both easy to use ***and*** applicable to a wide range of devices, applications, and use cases.

[1] `https://docs.mitmproxy.org/stable/concepts-modes/#transparent-proxy`

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The mitmproxy project provides a free and open source interactive man-in-the-middle proxy. It is a widely used tool for debugging, testing, privacy and security research, and reverse engineering of network protocols.

Authors of many recent publications that reference mitmproxy ([Hue+21; Ngu+21; Lei21; KKW20; Liu+21]) apparently used it for proxying network traffic at the IP protocol level (i.e. used mitmproxy in "transparent" mode) to route all network traffic of a device through the proxy.

This existing "transparent" mode is more powerful than other modes that are supported by mitmproxy, as it allows proxying arbitrary IP traffic. However, this mode is cumbersome to set up on the mitmproxy host, and is not always easy or possible to set up on client devices.

Additionally, setting it up on the host machine requires administrative privileges to modify firewall settings and change the operating system's network configuration to accept IP packets with arbitrary destination addresses (i.e. enable AnyIP support for IPv4).

Implementing an alternative to "transparent" mode that is still powerful, but much easier to set up and use, makes mitmproxy a more approachable and useful tool for all existing and potential users.

Building this alternative on top of the open, standardized WireGuard protocol made it possible to leverage official WireGuard clients on Microsoft Windows, macOS, Linux, Android, and iOS, and other platforms.

1.2 Limitations and advantages of the new approach

Usage of the new "wireguard" mode should not be limited by platform support, as all relevant host operating systems are supported, and official WireGuard clients exist for most platforms. However, due to the nature of WireGuard as an encrypted protocol, there is a small but noticeable overhead compared compared to plain IP traffic.

By combining several existing technologies – a user space WireGuard protocol implementation, a user space TCP/IP and network stack – in a novel way, the resulting implementation provides similar features, but is easier to set up and use than the existing "transparent" mode.

The performance impact of using an encrypted protocol should not be noticeable. In most circumstances, network transfer speed and latency between the mitmproxy host machine and the client device will be the bottleneck, not the processing of WireGuard packets.

1.3 Use of new technologies

The *WireGuard* protocol [Don20] is one of the newest (stable release in 2019) and most efficient protocols that can be used to implement VPNs. A performance comparison between a VPN based on WireGuard and a VPN based on the OpenVPN protocol showed a smaller overhead [Mac+20].

While mitmproxy is written in Python, the core functionality of "wireguard" mode is written in *Rust* [KN18], which is a modern systems programming language that enables "fearless concurrency" and prevents several types of memory safety issues with its language features. There is also a growing ecosystem of networking-related open source libraries that are implemented in Rust, some of which were used to build mitmproxy-wireguard: *tokio* (asynchronous runtime), *smoltcp* (user space network stack), *boringtun* (user space WireGuard implementation). The *PyO3* Rust libraries also provide a convenient way to implement native Python modules in Rust, which was used to implement the core functionality of "wireguard" mode – the mitmproxy-wireguard Python package.

By combining these technologies, traffic between the client device and the mitmproxy host is implemented in a higher-level network protocol (UDP instead of IP), which eliminates the need for most – if not all – manual routing setup and firewall configuration changes. This makes the implementation and set up of mitmproxy easier, without losing any features compared to "transparent" mode.

1.4 Architecture overview

The implementation of the user space WireGuard server in mitmproxy-wireguard and its integration in mitmproxy are split into several loosely coupled components (Figure 1.1).

The WireGuard server itself is implemented as four asynchronously running sub-tasks:

- a task which implements a user space *WireGuard* server,
- a task that implements a *Network* (i.e. TCP/IP) stack in user space,
- a task for handling *interoperability with the Python runtime* (not shown), and
- a task that is responsible for clean server *shutdown* and error handling (not shown).

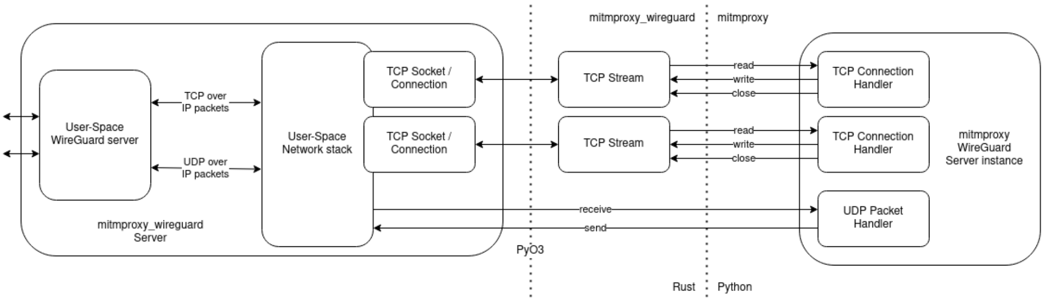

Figure 1.1: Architecture of the user space WireGuard server, network stack, Python interface, and integration with mitmproxy.

All sub-tasks of the WireGuard server operate fully asynchronously, and exchange data by sending and receiving messages over asynchronous channels or sockets (indicated by arrows):

- WireGuard UDP packets are sent and received by the WireGuard task on a standard UDP socket.
- Events for incoming and outgoing IP packets are exchanged between the WireGuard task and the network stack.
- Events for sending and receiving UDP packets and for TCP socket operations are exchanged between the Python interface and the network stack.

This event-based, modular architecture results in low CPU usage when idle, and will enable future extensions and optimizations.

1.5 Achievements

The “wireguard” mode as an alternative to the existing “transparent” mode achieves its goal of being easier to use – requiring no administrative privileges for set up on the mitmproxy host, and requires no additional configuration steps other than importing the WireGuard configuration provided by mitmproxy on the client device.

Connecting clients is straightforward, since existing official WireGuard client applications can be used.

1.6 Installation and Setup

As of version 9.0 of mitmproxy, the “wireguard” mode is generally available and included by default, and is expected to work out-of-the-box without additional installation steps (other than the usual steps for installing and setting up mitmproxy).

1.7 Evaluation

To evaluate the new functionality, mitmproxy in “wireguard” mode was tested running on different host operating systems (Windows and Linux), and with different client devices (Windows, Linux, Android).

Additionally, unit tests are used to verify the expected behaviour of the network stack implementation in mitmproxy-wireguard, and integration tests in mitmproxy ensure the basic functionality of “wireguard” mode is working as expected (by simulating incoming WireGuard traffic), similar to how tests for the existing “transparent” mode are implemented.

To ensure performance of “wireguard” mode is acceptable in comparison with the existing “transparent” mode, throughput of two TCP echo servers implemented with Python asyncio and mitmproxy-wireguard was compared in both a worst-case scenario (server and client running on the same device) and the expected, average use case (server and client running on different devices on the same local network).

1.8 Contributions

The effort for implementing a “wireguard” mode for mitmproxy can be described as two core contributions:

- The development of *mitmproxy-wireguard*, a Python package implemented in Rust, which provides an interface that is modelled after the “asyncio” module from the Python standard library. Its development history is preserved in the project’s GitHub repository[1]. It is also published on the Python Package Index (PyPI)[2].
- The integration of mitmproxy-wireguard to implement a “wireguard” mode for mitmproxy. The initial code submission to the mitmproxy project was discussed – and ultimately accepted – in Pull Request #5562[3], and follow-up Pull Request #5607[4] added documentation for “wireguard” mode.

Other, minor contributions include the reporting of issues in open-source libraries which were encountered during the implementation or testing of this project; most notably, a fix for a race condition in *pyo3-asyncio*, the Rust library that was used for bridging the *tokio* and Python runtimes[5].

[1] `https://github.com/decathorpe/mitmproxy_wireguard`
[2] `https://pypi.org/project/mitmproxy-wireguard/`
[3] `https://github.com/mitmproxy/mitmproxy/pull/5562`
[4] `https://github.com/mitmproxy/mitmproxy/pull/5607`
[5] `https://github.com/awestlake87/pyo3-asyncio/issues/77`

2 Background and Motivation

2.1 Comparison with "transparent" mode

The existing "transparent" mode of mitmproxy operates on the IP protocol level, which makes it necessary to configure the mitmproxy host to accept arbitrary incoming IP packets (i.e. enable "AnyIP" mode), and to modify firewall settings to allow the necessary packet routing. It is implemented on top of the Python asyncio module, which provides the necessary functionality for handling TCP and UDP packets.

The new "wireguard" mode in mitmproxy works in essentially the same way as "transparent" mode, but replaces uses of networking-related components from the asyncio standard library module with drop-in replacements from mitmproxy-wireguard. The interfaces provided by mitmproxy-wireguard abstract over the underlying differences between the two implementations, i.e. that network traffic between server and client happens over UDP packets (over IP) instead of IP packets directly.

2.2 Problems with "transparent" mode

Using mitmproxy in "transparent" mode requires several additional steps, all of which need to be run with administrative privileges, and most of which permanently modify operating system or firewall settings, i.e. enabling IP packet forwarding, disabling ICMP redirects, and modifying routing / firewall rules.

The exact commands which need to be run also depend on the local environment, and need to be adapted for the name of the interface for the local network, and the port numbers for which incoming traffic should be redirected through mitmproxy.

Additionally, not all client devices might expose the necessary settings to set them up to direct traffic at the mitmproxy host (i.e. setting the default gateway address for a specific network connection) – in these cases, DHCP needs to be disabled entirely to override the address of the default gateway.

2.3 Assumptions

The existing modes of operation of mitmproxy are powerful and useful, but of limited availability or applicability. They are either difficult to set up, require elevated privileges, and change operating system settings permanently ("transparent" mode), or require the client device or application to support – and respect – HTTP or SOCKS5 proxy settings (corresponding to the "regular" and "socks5" modes of mitmproxy).

Providing an alternative to the existing "transparent" mode that is at least as powerful, but more easily available for potential users was assumed to be a worthwhile goal, which was also confirmed by mitmproxy developers via personal communication.

2.4 Scope of the project

The interfaces that are provided by the mitmproxy-wireguard Python package are intended to be drop-in replacements for any interfaces of the Python standard library's asyncio module that are used by mitmproxy. Only the functionality necessary for mitmproxy was implemented, and the package is not a general-purpose replacement.

The initial release was also focused on supporting IPv4 traffic over WireGuard, with only basic support for IPv6 traffic being implemented. Support for IPv6 traffic is also essentially untested, and hence not enabled in the default configuration.

Additionally, there are only limited ways to configure "wireguard" mode in this first release. For example, mitmproxy only supports one client ("peer") per WireGuard server, while mitmproxy-wireguard provides interfaces for configuring an arbitrary number of peers per connection.

2.5 Benefits of "wireguard" mode

Using mitmproxy in "wireguard" mode does not require modifying any networking configuration on the host operating system or changing firewall settings when using the default configuration (i.e. running the internal WireGuard server on port 51820).

To connect a client device, the WireGuard configuration provided by mitmproxy can be imported or used by any official WireGuard client. No manual changes to the device's network or proxy configuration are required.

3 Design and Architecture

The goal of this project was to provide an alternative to the existing “transparent” mode of mitmproxy that is at least as powerful as the existing functionality, but which works in more circumstances and is available for more users with a lower barrier to entry.

The “wireguard” mode in mitmproxy is based on code for “transparent” mode, with functionality for running a TCP server and receiving UDP packets that is provided by the “asyncio” Python module replaced by equivalent functionality from mitmproxy-wireguard, and a small amount of glue code for handling UDP packets.

3.1 Required changes to mitmproxy

Other than adding the actual code for the new mode, only minor changes to mitmproxy were required. The logic for parsing command-line arguments needed to be extended for arguments that are specific to “wireguard” mode, and small changes to the classes which abstracted over UDP transport were required to allow different underlying implementations of sending and receiving data over UDP sockets (i.e. to support implementations based on both Python asyncio and mitmproxy-wireguard).

Additionally, the custom logging framework in mitmproxy was not compatible with mitmproxy-wireguard, as the library it uses to bridge log messages between Rust and Python (*pyo3-log*) only supports the `logging` module from the Python standard library. The custom logging framework was later removed from mitmproxy and replaced by equivalent functionality based on the `logging` module, which resulted in it being able to show log messages that originate in mitmproxy-wireguard.

3.2 Component overview

The main functionality of the mitmproxy-wireguard package is the user space WireGuard server, which is implemented as loosely coupled, asynchronous tasks in an event-based architecture:

- a *WireGuard server* task, which is responsible for sending and receiving WireGuard packets over UDP, forwards IP packet payloads of valid incoming packets to the network stack, and processes outgoing IP packets;

- a task which acts as a *Network stack*, which keeps track of sockets, handles TCP connections, and implements parsing and construction of IP, TCP, and UDP packets;
- a task for *interoperating with Python* Futures and coroutines, which is responsible for launching the callback coroutines for newly established TCP connections and for received UDP packets;
- a watchdog task for *handling server shutdown*, which keeps track of the state of all other sub-tasks, responds to server shutdown requests, or triggers server shutdown in case any of the sub-tasks failed unexpectedly, and logs error messages accordingly.

Additionally, the public interface of the package includes a coroutine which sets up and initializes these tasks, and some small utility functions for generating new WireGuard encryption keys.

3.3 Architectural Constraints

Transmitting data over a WireGuard tunnel is slightly less efficient than sending plain IP packets due to the smaller MTU (Maximum Transmission Unit). The size of WireGuard packet headers is 60 bytes for IPv4 packets and 80 bytes for IPv6 packets, so the payload size for WireGuard packets is limited to 1420 bytes in mitmproxy-wireguard.

Due to the way the WireGuard server task is implemented, it might be a bottleneck for packet throughput. Encryption and decryption of WireGuard packets are handled on the task's main loop, even though this could possibly be handled asynchronously to not block the thread that sends and receives WireGuard packets.

Another bottleneck might be the network task, where multiple, possibly unrelated channels are polled for events on the task's main loop, which also handles updating socket states. Splitting the processing of events from different sources and updating socket states into separate sub-tasks might be a way to improve throughput.

Additionally, passing packet payloads from the network stack to the Python callback functions for TCP connections and UDP packets as Python `bytes` objects involves copying all data. Initialization of a new `bytes` object always requires copying the entire block of memory due to how this class is implemented in the Python interpreter.

4 Implementation

4.1 Basis for validation

Different methods were used to monitor implementation progress and validate expected behaviour of project components:

- *user space WireGuard server*: Clients can successfully connect and subsequently transmit and receive data.
- *user space network stack*: Task responds correctly to incoming network events and requests.
- *compatibility with Python*: Functions, classes, methods, and coroutines / Futures implemented in Rust work as expected when called from Python.
- *performance*: Throughput of a TCP server implemented on top of mitmproxy-wireguard is not substantially worse than an equivalent implementation based on the Python asyncio module.

4.2 Use of existing software projects

The Rust programming language [KN18] was chosen to implement the stand-alone mitmproxy-wireguard package for multiple reasons. The memory safety guarantees provided by the language itself make it possible to implement highly concurrent and parallel programs without risking data races or invalid memory accesses. Additionally, some of the functionality that was needed to implement this project was available in existing open-source libraries, which are also written in Rust:

1. *tokio* [PCc]: an asynchronous runtime for Rust, which also provides implementations of asynchronous channels, synchronization primitives, network sockets;
2. *boringtun* [Clo]: a user space implementation of the WireGuard protocol;
3. *smoltcp* [PCb]: an efficient, zero-allocation user space implementation of a network stack, which also includes functionality for parsing and constructing packets for various network protocols;
4. *PyO3* [PCa]: Rust bindings for the C API of the Python interpreter and other functionality for implementing “native” Python modules in Rust;

5. *maturin* (part of the PyO3 project): a tool for building "native" Python packages that are implemented with Rust and PyO3, and for publishing them to the Python Package Index (PyPI).

Additionally, the process for publishing the mitmproxy-wireguard Python package to the Python Package Index (PyPI) with *maturin* was automated with GitHub Actions[1].

4.3 Resource Usage

4.3.1 Size on disk

The size of the pre-compiled "native" Python packages ("binary wheels") for mitmproxy-wireguard is between 700 kB and 1.7 MB, depending on the target architecture (Table 4.1; file sizes as reported by PyPI for version 0.1.18 of mitmproxy-wireguard[2]).

Distribution	Size
Binary wheel for Windows / `x86_64`	694.0 kB
Binary wheel for Linux / `x86_64`	1.2 MB
Binary wheel for Linux / `aarch64`	1.2 MB
Binary wheel for macOS / `x86_64`	855.3 kB
Binary wheel for macOS / Universal2 (`x86_64` and `aarch64`)	1.7 MB
Source distribution	28.8 kB

Table 4.1: Size of mitmproxy-wireguard distributables on PyPI as of version 0.1.18.

The inclusion of these binary wheels in mitmproxy distributables appears to have contributed to their general increase in size between version 8.1.1 (last version without "wireguard" mode) and version 9.0.1 (version 9.0 was the first version that shipped with "wireguard" mode).

Download links for all supported platforms and for older versions of mitmproxy are still available from the project's download archive[3], which is the source of file sizes listed in table 4.2.

Even though file sizes increased by at least 12 MB, the inclusion of mitmproxy-wireguard module should only account for at most 2 MB of this change – so other factors must have contributed to this significant file size increase.

[1] `https://docs.github.com/en/actions`
[2] `https://pypi.org/project/mitmproxy-wireguard/0.1.18/#files`
[3] `https://snapshots.mitmproxy.org`

Distributable	Size (mitmproxy 8.1.1)	Size (mitmproxy 9.0.1)
Windows (standalone binaries)	55.6 MB	71.6 MB
Windows (installer)	34.3 MB	48.8 MB
Linux (standalone binaries)	86.2 MB	110.1 MB
macOS (standalone binaries)	43.1 MB	55.7 MB

Table 4.2: File sizes of mitmproxy distributables in the project's download archive.

4.3.2 Memory usage

After importing the relevant modules and launching a simple TCP echo server that is implemented on top of either the Python asyncio module or mitmproxy-wireguard, the memory usage of the Python interpreter process was similar in both cases (10.7 MB and 12.2 MB memory use, respectively; tested on Fedora Linux 37, with Python 3.11.0 and mitmproxy-wireguard 0.1.18, running on an `x86_64` system).

However, passing large amounts of data through the server implemented on top of mitmproxy-wireguard resulted in growing memory usage, which appears to indicate a memory leak.

4.3.3 CPU usage

The user space WireGuard server implemented with mitmproxy-wireguard has very low CPU usage at idle. Introspection of the asynchronous runtime confirmed that no tasks are "busy waiting".

When processing data, the CPU usage of a server based on mitmproxy-wireguard is slightly lower than that of a server based on Python asyncio, while also achieving lower throughput. This probably indicates a bottleneck in how mitmproxy-wireguard processes packets, and might be a possible target for future optimizations.

4.4 Implementation of individual components

The functionality of mitmproxy-wireguard package was implemented as several separate tasks that operate asynchronously and exchange events over message channels. This approach made it possible to split unrelated functionality into separate, loosely coupled components, whose behaviour could be developed and tested separately.

4.4.1 Integration with mitmproxy

The interface provided by mitmproxy-wireguard is designed to be a drop-in replacement for the TCP server implementation and other related functionality from the asyncio module in the Python standard library. Due to this approach, integrating its functionality as a new mode in mitmproxy was relatively straightforward.

The class which represents a mitmproxy server in “wireguard” mode built on top of mitmproxy-wireguard is based on existing code for “transparent” mode, and was modified to use methods from mitmproxy-wireguard instead of Python asyncio, needing only minor adaptations to account for differences in how packets and connections need to be handled.

Integrating support for handling UDP packets required the addition of a simple adapter class, similar to an existing class in mitmproxy that already wrapped the UDP functionality from the Python standard library.

For the first release, only limited WireGuard configuration options were implemented. For now, only one WireGuard client (“peer”) per server is supported – but multiple proxy servers can be launched on different ports. Adding support for configuring multiple peers per server in the future would be straightforward, since this functionality is already present in mitmproxy-wireguard, but just not exposed to users of mitmproxy.

4.4.2 Public Python interface

The public interface of the mitmproxy-wireguard Python package was modelled after the functionality of the asyncio module from the Python standard library – in particular, the `TcpStream` class implements methods that are identical to those provided by “StreamReader” and “StreamWriter”, merged into a single class instead of separate classes for reader and writer. These implementations are intended to be drop-in replacements, but only functionality that is actually used by mitmproxy was implemented.

The classes, methods, and functions that are part of the public interface of the mitmproxy-wireguard package are implemented as Rust structs and functions, but transformed into code that provides the expected C ABI for native Python extension modules at compile time – by using macro-based metaprogramming functionality of *PyO3*.

Server setup

The function for initializing the WireGuard server and spawning its sub-tasks is similar to the `start_server` function in the Python asyncio module. It accepts parameters for general server setup (i.e. which host address and port to bind to), WireGuard encryption keys (private key of the server, and public keys of the configured peers), and callbacks for handling TCP connections and incoming UDP packets.

At runtime, the callback coroutine for handling TCP connections is called with a `TcpStream` object corresponding to newly established TCP connections, and the callback function for handling UDP packets is called with the incoming packet’s payload, its source socket address, and its destination socket address.

Server initialization

Calling the `start_server` function binds a UDP socket to the specified address, starts listening on the configured UDP socket for incoming WireGuard packets, initializes channels for communicating events between sub-tasks, sets up all configured WireGuard peers, and launches the sub-tasks for the WireGuard server, the network stack, and for interoperability with the Python asynchronous runtime.

Methods on the returned `Server` object can be used to request shutdown of the server (`close()`), awaiting shutdown (`wait_closed()`), and for sending UDP packets over the WireGuard tunnel (`send_datagram(data, src_addr, dst_addr)`).

TCP stream interface

The `TcpStream` objects which are passed to the TCP connection handler callback coroutine are initialized in the user space network task. The following methods are implemented for objects of this class:

- `read(n)`: This method sends a `ReadData` event to the network stack, which contains a limit on how many bytes should be read (`n`), and a sender for an ad-hoc channel for returning the read data to the caller once it is ready. It returns a Python `Future`, which yields its result once the network task has processed the requested read operation and has sent the data over the ad-hoc channel.
- `write(data)`: This method sends a `WriteData` event to the network stack, which includes the data itself. It is non-blocking and returns immediately.
- `drain`: This method sends a `DrainWriter` event to the network stack, which contains a sender for an ad-hoc channel. This method returns a Python `Future`, which yields after receiving a message from the network stack which indicates that the "drain" operation has completed.
- `write_eof`: This method sends a `CloseConnection` event to the network stack that indicates that the TCP connection should be closed, but pending data should still be sent. Calls to this method are non-blocking and return immediately.
- `close`: This method works the same as the `write_eof` method, except that the `CloseConnection` event it sends indicates that the TCP connection should be closed immediately and pending data should be dropped.
- `is_closing`: This method returns a boolean that indicates whether the `write_eof` or `close` method had been called on this `TcpStream` instance.
- `get_extra_info`: This method can be used to query for details of the underlying TCP connection, i.e. the source and destination addresses of the connection (inside the WireGuard tunnel), or the original source and destination addresses.

4.4.3 WireGuard server

The WireGuard server task is initialized with a pair of channels to communicate with the task responsible for the user space network, and gets passed the handle to the UDP socket which was initialized during server initialization.

The main loop of the WireGuard server task waits for three different types of events:

1. *shutdown notification*: This triggers the WireGuard server to flush its buffers and then shut down.
2. *incoming WireGuard UDP packet*: Incoming packets are passed to the user space WireGuard implementation to process handshakes and connection establishment. Once a session is established, incoming data packets are decrypted and their payload (an IP packet) and the original source address are sent to the network stack as a "ReceivePacket" event.
3. *event for outgoing traffic*: Outgoing IP packets are associated with the correct peer according to their destination address, encrypted with the peer's public key, and sent as a WireGuard packet to the peer's address over the UDP socket.

4.4.4 User space networking

The task for running a user space network stack is initialized with two pairs of channels – one pair for exchanging events with the WireGuard server task, and one pair for exchanging events with the Python interoperability task.

At start-up, a virtual network device is initialized with default routes and explicit forwarding for incoming packets with arbitrary destination IP ("AnyIP" support).

The main loop of this task consists of five phases:

1. Check the state of the virtual network device, channel capacities, TCP timeouts, and acquire a permit for sending an event to the Python runtime interoperability task.
2. Await events from up to six different sources (depending on the state of the virtual network device and channel capacities):

   a. *shutdown notification*: This triggers the task to drop pending events and shut down.

   b. *timeout*: If there is a pending timeout from any established TCP connection, the task will never sleep longer than this timeout duration.

   c. *incoming IP packet*: If the channel for sending events to the Python interoperability task is already full, this is skipped to apply backpressure.

   d. *outgoing traffic event*: If the channel for sending events to the WireGuard task is already full, this is skipped, as well.

e. *regain channel capacity*: Wait until either of the channels for sending events are no longer full (skipped if channels are not full).

3. Poll the network device to process received data and update internal TCP connection states.

4. Iterate over all open TCP sockets and process pending requests:

   a. If the `read` method was called on the socket, pass data from the socket's buffer, if available; otherwise, indicate to the reader that the socket has no data or was closed.

   b. If the socket's send buffer is not empty, drain data from the buffer and send an event for outgoing network traffic to the WireGuard task.

   c. If `drain` method was called on the socket, or if the socket is being closed, drain data from its buffer, send it as an event for outgoing network traffic to the WireGuard task, and send an event that indicates to the caller that the requested "drain" operation is done.

   d. If the `close` method was called on the socket, drain the data from its buffers, and mark the socket as pending closure by the virtual network device.

   e. If the socket is marked as closed by the virtual network device, remove it from the list of currently open connections.

5. Poll the network device to process received data and update internal TCP connection states.

4.4.5 Python runtime interoperability

The task for communicating between the user space network stack and the Python runtime is initialized with a pair of channels for exchanging events, a callback coroutine which will be run when a new TCP connection is established, and a callback function which is called for every received UDP packet.

In contrast to other channels in mitmproxy-wireguard, the channel for receiving commands from the Python interface is unbounded, to match behaviour of the TCP server in the Python asyncio module, which provides non-blocking writes. The main loop of this task waits for events from two sources:

1. *shutdown notification*: This triggers the task to drop any pending events and shut down.

2. *incoming network event*:

   a. For new TCP connections, a new `TcpStream` object is instantiated (including a copy of the sending end of the channel for sending events to the network device task). The callback coroutine for new TCP connections is then scheduled to run asynchronously, and is passed this object as an argument.

b. For received UDP packets, the provided callback for new UDP packets is called with the packet's payload and its source and destination address as arguments.

The actual interaction with the `TcpStream` object (i.e. reading from, writing to, and draining or closing the connection) causes events to be sent to the network device stack directly – these events contain ad-hoc channels to facilitate sending data back from the network device task to the `TcpStream` after the request has been processed.

Similarly, sending UDP packets is accomplished by calling a method on a "Server" object directly, and does not involve the Python interoperability task, since this is a blocking function call.

4.4.6 Watchdog / Shutdown handler

The task that handles clean server shutdown also handles cases where any server task might fail early, and will trigger a clean shutdown of the remaining tasks in this case.

It is initialized with "join handles" of all other running tasks, and waits for any of them to return; if any task fails early (i.e. server shutdown had not been requested), this task also triggers shutdown of all other tasks. It attempts to joining all pending tasks to collect and log any error messages that are returned by them.

At this point, the task also sends a notification that results in the `Server.wait_closed` coroutine to yield, if it is being awaited anywhere.

4.5 Implemented optimizations

Early versions of the network stack implementation only consumed one network event or request per iteration of its main loop, and only one event was processed before sockets on the virtual network device were polled. In recent versions, the implementation was changed to consume incoming events in batches, so long capacities of outgoing channels allow for it.

5 Evaluation

5.1 Throughput comparison with Python asyncio

Since the interface provided by mitmproxy-wireguard intentionally mirrors the one provided by the Python `asyncio` module and is intended as a drop-in replacement, they are easy to compare.

As a substitute for an actual workload, TCP echo servers (i.e. servers that immediately return any received data without modifications) were implemented on top of both `mitmproxy-wireguard.start_server` and `asyncio.start_server`, and the time it took to send and receive between 1000 and 100,000 packets – each with 1000 bytes of payload – was measured to compare server throughput.

These benchmarks were run both against a server running on the same host (unsupported worst-case scenario) and over a local network (the expected average use case). The complete benchmark environment and configuration are listed in Appendix A.

5.1.1 Resource usage

During the "worst-case" benchmark run, the average CPU usage of the server based on mitmproxy-wireguard was considerably higher (117%) than of the server based on Python asyncio (77%). In the "average case" scenario, the difference was smaller (16% and 6%). In both scenarios, the server based on mitmproxy-wireguard allocated significantly more memory than its counterpart (ca. 2.5 GB vs. 20 MB), which might point to a memory leak.

5.1.2 Test results

In the worst-case scenario – server and client running on the same device, no traffic sent over a physical network – throughput of a server based on mitmproxy-wireguard was significantly lower than the implementation using a Python TCP server. When sending packets with 1000 B payloads, the average throughput of the server based on Python asyncio was 9500 packets/s, but throughput of the server based on mitmproxy-wireguard was only about 3730 packets/s (Fig. 5.1; full results in Appendix B: Tables 9.1 and 9.2, raw measurements in Appendix C: Tables 9.5 and 9.6).

This scenario only represents a control case, since running mitmproxy in "wireguard" mode is currently not supported in this configuration.

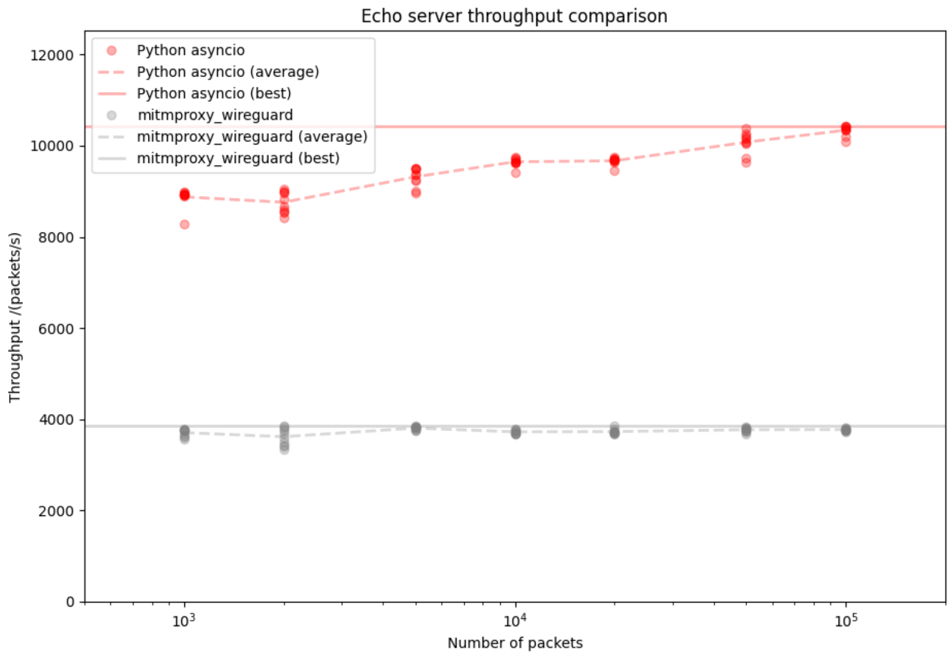

Figure 5.1: TCP echo server throughput comparison (local, 1000 bytes per packet)

In the scenario that more closely matches the expected use case (i.e. server and client being connected over a local network), both absolute and relative differences of packet throughput between the two implementations were smaller, with the server based on the Python asyncio module achieving an average throughput of 343 packets/s, and the server based on mitmproxy-wireguard achieving 274 packets/s (Fig. 5.2; full results in Appendix B: Tables 9.3 and 9.4, raw measurements in Appendix C: Tables 9.7 and 9.8).

In general, variance of results was similar between the two implementations. However, when using a small number of packets, the results for the server based on Python asyncio varied more strongly – possibly due to the additional variance introduced by garbage collection in parts of the TCP server implementation (compared to the non-garbage-collected TCP server implementation in mitmproxy-wireguard).

In contrast, results for the server based on mitmproxy-wireguard varied more strongly when using large amounts of packets, which was probably caused by WireGuard session timeouts and subsequent renegotiations, which are more likely during longer runtimes. Observation of log messages emitted by the server during the benchmark runs seemed to support this.

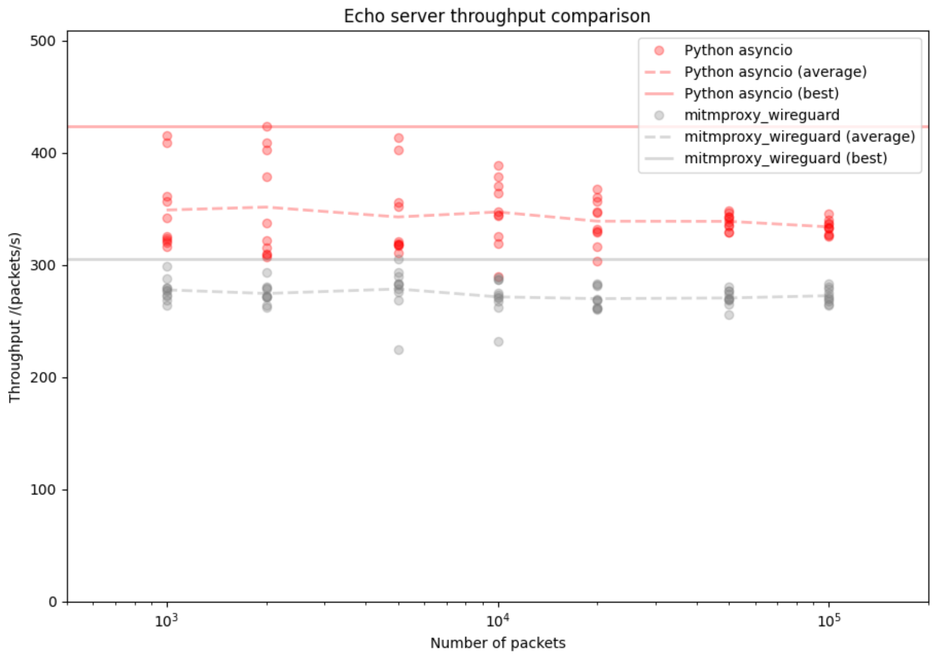

Figure 5.2: TCP echo server throughput comparison (networked, 1000 bytes per packet)

5.1.3 Limitations

The workload for these benchmarks does not necessarily reflect usage patterns of mitmproxy. However, the results show that a TCP server implementation based on mitmproxy-wireguard is not prohibitively slow – in the scenario that match the expected use case, it provided packet throughput that was about 25 % lower than that of a comparable implementation based on Python asyncio. Even demanding network traffic, such as streaming video, did not cause any problems.

5.2 Automated testing

5.2.1 Unit tests in mitmproxy-wireguard

The continuous integration (CI) pipeline for mitmproxy-wireguard executes test builds and runs tests on Windows, Linux, and macOS, ensuring that the project successfully builds on all supported platforms.

The core functionality of the user space network task is covered by tests (over 90 % line-based test coverage) – validating that the task correctly handles all possible incoming events and emits the expected events in response. These tests are automatically run by the mitmproxy-wireguard project's CI pipeline.

However, due to the nature of mitmproxy-wireguard as a Python package implemented in Rust, the parts of its functionality that are only exposed as Python interfaces are either difficult or almost impossible to correctly invoke from Rust test code, and are hence not covered by Rust unit tests.

5.2.2 Integration tests in mitmproxy

Additional integration tests, which also cover code paths that are not executed by unit tests in mitmproxy-wireguard, are present in mitmproxy, where basic functionality of the "wireguard" mode is tested in its CI pipeline (similar to tests for "transparent" mode), which also runs tests on Windows, Linux, and macOS.

5.3 Manual testing

In addition to automated tests, manual testing of the "wireguard" mode of mitmproxy was performed in various configurations (Table 5.1).

Server OS	Client OS	WireGuard Client
Fedora Linux 37	Fedora Linux 37	WireGuard kernel module (linux v6.0.8); wireguard-tools v1.0.20210914
Fedora Linux 37	Windows 11	WireGuard Windows Client v0.5.3
Fedora Linux 37	Android 13	WireGuard Android Client v1.0.20220516
Windows 11	Fedora Linux 37	WireGuard kernel module (linux v6.0.8); wireguard-tools v1.0.20210914
Windows 11	Android 13	WireGuard Android Client v1.0.20220516

Table 5.1: Test configurations for manual user testing of mitmproxy in "wireguard" mode

Due to limited availability of hardware for testing, no test scenarios involving macOS, iOS, or the combination of Windows running on both server *and* client device were included.

5.3.1 Test workload

The workload for manual user testing involved the typical steps that mitmproxy users would take to start capturing network traffic:

1. Download and / or install the latest version of mitmproxy (currently: version 9.0.1).
2. Start "mitmweb" (the mitmproxy web interface) in "wireguard" mode according to the documentation for the current operating system.

3. Download and / or install official WireGuard client software on the “client” device.
4. Import the WireGuard configuration that is provided by mitmweb on the client device (i.e. scan the QR code on Android; import configuration file on Windows and Linux).
5. Enable the newly configured WireGuard tunnel on the client device.
6. Verify that device traffic is being routed through mitmproxy and shows up in the mitmweb interface.
7. Verify that using a browser to open the special “mitm.it” address on the client device works correctly (i.e. the page contains instructions for downloading and importing a self-signed TLS certificate, and not the “ If you can see this, traffic is not passing through mitmproxy.” warning).

5.3.2 Resource usage

Enabling a WireGuard tunnel on client devices did not result in any obvious increase of used resources. However, since the overhead caused by encrypting and decrypting packets scales with the amount of data that is transmitted, any additional resource usage will depend on the pattern of network use on the connected client device.

5.3.3 Test results

In all tested combinations, all steps of the test workload were successful and yielded the expected results, though one additional step was required in two cases:

When launching the WireGuard client application for Windows for the first time, it automatically requested permissions for network access through the Windows firewall. Only a confirmation of this network access request was required, and it was only required once.

On Android, connecting to a WireGuard tunnel (or any VPN service) for the first time on a given device caused a generic warning dialogue to be shown by the Android user interface. Only simple manual confirmation by the user was required, and this warning is shown only once.

5.3.4 Limitations

No macOS or iOS devices were available for manual testing. However, initial user feedback indicates that “wireguard” mode indeed works as expected on all supported platforms.

Importing the self-signed certificate provided by mitmproxy and intercepting HTTPS traffic were not tested, since these features should not depend on the current mode of operation.

6 Limitations

Proxying network traffic from the same device that mitmproxy is running on is not supported in “wireguard” mode, since there is no way to reliably differentiate between traffic sent by mitmproxy (which should not pass through mitmproxy a second time) and traffic that should be sent through mitmproxy.

Support for proxying IPv6 traffic is still considered experimental. This feature is not yet complete and not enabled by default, but it can easily be enabled for testing purposes by modifying the provided WireGuard client configuration to allow IPv6 traffic.

7 Related Work

Other MITM or HTTP / HTTPS proxy software, like Charles[1], Telerik Fiddler[2], ZAP[3], or Requestly[4] all appear to only operate as HTTP(S) and / or SOCKS5 proxies, at least according to publicly available documentation.

Since many applications do not include support for manual proxy configuration, their usefulness is limited. Most operating systems support setting system-wide proxy settings, but not all applications will honour these settings. On the other hand, it is rare for application traffic to bypass system-wide VPN connections (which is often considered a security risk), but this is usually limited to some core system applications [Lun20].

[1] `https://www.charlesproxy.com/`
[2] `https://www.telerik.com/fiddler`
[3] `https://www.zaproxy.org/`
[4] `https://requestly.io/`

8 Future Work

The current version of mitmproxy-wireguard and its integration in mitmproxy will likely be extended and further built upon in the future.

Enabling support for proxying IPv6 traffic by default still requires implementing support for some IPv6-specific features and packet types – in addition to broader testing of this feature. Some aspects of the current implementation are not yet optimized with respect to performance and resource usage, which are also a possible target of future work.

Some parts of mitmproxy-wireguard – in particular, the user space network stack implementation – seem to be useful in general. It is possible to add implementations of other “frontends” to cover more use cases without requiring a major rewrite of the project (for example. more user-friendly implementations of “transparent” mode).

9 Conclusion

Intercepting and / or manipulating network traffic is necessary for many tasks, like debugging, testing, and reverse engineering of network protocols. While mitmproxy supports intercepting arbitrary IP traffic in “transparent” mode, setup and configuration on both the mitmproxy host and the client device are complicated and error-prone.

Falling back to using an HTTP or SOCKS5 proxy is often not possible either, since support for setting network proxy settings is often missing from target applications – and some applications do not even respect system-wide network proxy settings.

By implementing a proxy mode on top of a VPN protocol instead (in this case, WireGuard), most drawbacks of both approaches are eliminated. Compared to “transparent” mode, the setup for correctly routing packets is no longer necessary, as that is handled by the underlying protocol. Setting up a client device to route traffic through mitmproxy also becomes straightforward – all relevant platforms support setting up system-wide VPN connections, and official WireGuard clients are almost universally available.

As a result, launching mitmproxy in “wireguard” mode requires no additional setup, and connecting a client device becomes as simple as importing the provided configuration in an officially supported WireGuard client, and enabling the network tunnel.

While there is some overhead involved with sending traffic over an encrypted protocol, i.e. encryption and decryption of packet payloads and a slightly smaller maximum payload size, it is small enough that it should not be noticeable in most use cases.

The modular architecture and implementation of mitmproxy-wireguard also make it possible for the project to be extended to cover other use cases, which could benefit from a similar setup – like using a TUN device on Linux (or similar functionality on other operating systems) instead of a user space WireGuard server.

References

[Clo] Cloudflare. *Boringtun*. Version 0.5.2. URL: `https://github.com/cloudflare/boringtun`.

[Cor+10] Aldo Cortesi et al. *mitmproxy: A free and open source interactive HTTPS proxy*. [Version 9.0]. 2010. URL: `https://mitmproxy.org/`.

[Don20] Jason A. Donenfeld. *WireGuard: Next Generation Kernel Network Tunnel*. Ed. by Jason A. Donenfeld. Version e2da747. June 1, 2020. URL: `https://www.wireguard.com/protocol/`.

[Hue+21] Man Hong Hue et al. "All Your Credentials Are Belong to Us: On Insecure WPA2-Enterprise Configurations". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 1100–1117. ISBN: 9781450384544. DOI: `10.1145/3460120.3484569`.

[KKW20] Sina Keshvadi, Mehdi Karamollahi, and Carey Williamson. "Traffic Characterization of Instant Messaging Apps: A Campus-Level View". In: *2020 IEEE 45th Conference on Local Computer Networks (LCN)*. 2020, pp. 225–232. DOI: `10.1109/LCN48667.2020.9314799`.

[KN18] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. USA: No Starch Press, 2018. ISBN: 1593278284.

[Lei21] Douglas J. Leith. "Web Browser Privacy: What Do Browsers Say When They Phone Home?" In: *IEEE Access* 9 (2021), pp. 41615–41627. DOI: `10.1109/ACCESS.2021.3065243`.

[Liu+21] Kaizheng Liu et al. "On Manually Reverse Engineering Communication Protocols of Linux-Based IoT Systems". In: *IEEE Internet of Things Journal* 8.8 (2021), pp. 6815–6827. DOI: `10.1109/JIOT.2020.3036232`.

[Lun20] Anders Lundberg. *Apples own programs bypass firewalls and VPNs in Big Sur*. [Online; accessed on December 8, 2022]. Nov. 18, 2020. URL: `https://www.macworld.com/article/675671/apples-own-programs-bypass-firewalls-and-vpns-in-big-sur.html`.

[Mac+20] Steven Mackey et al. "A Performance Comparison of WireGuard and OpenVPN". In: *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*. CODASPY '20. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 162–164. ISBN: 9781450371070. DOI: `10.1145/3374664.3379532`.

[Ngu+21] Trung Tin Nguyen et al. “Share First, Ask Later (or Never?) - Studying Violations of GDPR’s Explicit Consent in Android Apps”. In: *USENIX Security Symposium*. Aug. 2021. URL: `https://publications.cispa.saarland/3400/`.

[PCa] PyO3 Project and Contributors. *PyO3*. Version 0.17.3. URL: `https://github.com/PyO3/pyo3`.

[PCb] Smoltcp Project and Contributors. *Smoltcp*. Version 0.8.1. URL: `https://github.com/smoltcp-rs/smoltcp`.

[PCc] Tokio Project and Contributors. *Tokio*. Version 1.22.0. URL: `https://github.com/tokio-rs/tokio`.

Appendix

Appendix A: Benchmark configuration and environment

The “server” for both benchmark scenarios was a PC equipped with an AMD Ryzen 7 5800X 8-Core processor and 32 GB of DDR4 RAM. In the “local system” scenario, it was also the “client”. The “client” device for the “local network” scenario was a Laptop equipped with an Intel Core i7-8550M 4-Core processor and 16 GB of LPDDR3 RAM.

Both devices were running Fedora Workstation 37 on top of Linux 6.0.8, version 3.11.0 of the CPython Python interpreter, and version 0.1.18 of mitmproxy-wireguard. For the “local network” scenario, the “server” was connected to the local network over Gigabit Ethernet, and the “client” was connected over 5 GHz Wifi (802.11ac).

Appendix B: TCP echo server throughput benchmark results

Number of packets	Mean Throughput	SD	RSD	Worst	Best
1000	8880/s	190/s	2.2 %	8290/s	8990/s
2000	8760/s	210/s	2.5 %	8430/s	9050/s
5000	9320/s	190/s	2.1 %	8970/s	9510/s
10 000	9650/s	90/s	0.9 %	9410/s	9760/s
20 000	9670/s	70/s	0.8 %	9470/s	9760/s
50 000	10 080/s	220/s	2.2 %	9630/s	10 390/s
100 000	10 340/s	100/s	1.0 %	10 100/s	10 430/s
total	9500/s	600/s	5.9 %	8300/s	10 400/s

Table 9.1: TCP echo server throughput (Python asyncio, local system, 1000 bytes per packet, N=10)

Number of packets	Mean Throughput	SD	RSD	Worst	Best
1000	3700/s	80/s	2.0 %	3560/s	3790/s
2000	3610/s	180/s	4.9 %	3340/s	3860/s
5000	3800/s	30/s	0.9 %	3740/s	3850/s
10 000	3720/s	40/s	1.1 %	3680/s	3790/s
20 000	3730/s	50/s	1.4 %	3670/s	3850/s
50 000	3770/s	40/s	1.2 %	3680/s	3840/s
100 000	3771/s	26/s	0.7 %	3732/s	3817/s
total	3730/s	100/s	2.6 %	3350/s	3860/s

Table 9.2: TCP echo server throughput (mitmproxy-wireguard, local system, 1000 bytes per packet, N=10)

Number of packets	Mean Throughput	SD	RSD	Worst	Best
1000	350/s	30/s	10 %	320/s	420/s
2000	350/s	40/s	13 %	310/s	420/s
5000	340/s	30/s	10 %	310/s	410/s
10 000	350/s	30/s	8 %	290/s	390/s
20 000	339/s	19/s	5 %	300/s	367/s
50 000	339/s	6/s	1.9 %	329/s	349/s
100 000	334/s	6/s	1.8 %	326/s	345/s
total	343/s	29/s	8 %	290/s	424/s

Table 9.3: TCP echo server throughput (Python asyncio, local network, 1000 bytes per packet, N=10)

Number of packets	Mean Throughput	SD	RSD	Worst	Best
1000	278/s	9/s	3.4 %	264/s	299/s
2000	274/s	8/s	3.1 %	262/s	293/s
5000	279/s	20/s	7 %	225/s	305/s
10 000	272/s	16/s	5 %	232/s	289/s
20 000	270/s	9/s	3.3 %	260/s	283/s
50 000	271/s	7/s	2.5 %	255/s	281/s
100 000	273/s	6/s	2.3 %	264/s	283/s
total	274/s	12/s	4.5 %	225/s	305/s

Table 9.4: TCP echo server throughput (mitmproxy-wireguard, local network, 1000 bytes per packet, N=10)

Appendix C: Raw TCP echo server benchmark results

Packets	Runtime /s	Packets	Runtime /s
1000	0.120 640 951 001 405 48	10 000	1.040 612 925 000 459 7
1000	0.111 306 323 000 462 73	10 000	1.036 462 058 000 324 8
1000	0.111 193 533 000 914 61	10 000	1.037 505 932 999 920 3
1000	0.111 778 165 002 760 94	10 000	1.032 057 480 999 356
1000	0.111 756 385 002 081 52	10 000	1.030 005 922 002 601 4
1000	0.111 747 383 998 590 52	20 000	2.065 217 046 001 635
1000	0.111 971 076 003 101 19	20 000	2.065 487 927 000 504
1000	0.112 157 577 001 198 66	20 000	2.066 903 472 001 286
1000	0.112 058 226 000 954 16	20 000	2.112 898 129 998 939
1000	0.112 321 477 001 387 4	20 000	2.068 773 539 998 801 4
2000	0.222 138 244 000 234 4	20 000	2.072 927 417 000 755 7
2000	0.226 736 042 997 799 8	20 000	2.062 540 574 999 73
2000	0.222 907 236 999 162 7	20 000	2.061 026 308 001 601
2000	0.223 137 318 000 226 51	20 000	2.058 243 317 998 858 4
2000	0.221 081 359 999 516 28	20 000	2.050 128 144 001 064 4
2000	0.230 737 029 000 010 8	50 000	5.192 086 221 999 489
2000	0.232 569 627 001 794 3	50 000	4.910 976 798 000 775
2000	0.234 150 202 999 444 45	50 000	4.968 672 982 999 124
2000	0.233 842 452 002 136 27	50 000	4.976 011 863 000 167
2000	0.237 318 485 000 287 18	50 000	5.140 922 353 999 485 5
5000	0.542 253 887 000 697 4	50 000	4.871 109 296 000 213
5000	0.557 518 920 002 621 6	50 000	4.925 201 506 001 031
5000	0.533 930 092 999 071 3	50 000	4.951 085 470 998 805
5000	0.555 858 561 998 320 5	50 000	4.892 460 333 001 509
5000	0.533 457 450 997 957 5	50 000	4.813 745 742 001 629
5000	0.540 369 499 998 632 8	100 000	9.646 146 889 997 908
5000	0.527 345 766 000 507 9	100 000	9.585 295 853 001 298
5000	0.525 745 340 000 867 1	100 000	9.597 965 475 000 820
5000	0.526 397 182 002 256 2	100 000	9.799 774 664 999 859
5000	0.525 754 810 001 672	100 000	9.619 604 712 999 717
10 000	1.062 276 803 000 713 7	100 000	9.663 213 489 999 18
10 000	1.027 517 462 000 105 3	100 000	9.645 961 419 999 367
10 000	1.024 950 041 999 545 6	100 000	9.590 124 143 000 14
10 000	1.037 818 404 001 882 3	100 000	9.657 840 357 998 793
10 000	1.036 780 170 001 293 3	100 000	9.900 315 185 001 09

Table 9.5: Benchmark runtimes (Python asyncio, local system, 1000 bytes per packet)

Packets	Runtime /s	Packets	Runtime /s
1000	0.264 975 004 000 007 2	10 000	2.651 280 314 999 894 5
1000	0.267 321 018 000 075 13	10 000	2.652 627 221 999 864 6
1000	0.280 631 667 999 841 74	10 000	2.716 451 704 000 064 6
1000	0.272 695 786 999 975 1	10 000	2.703 110 614 002 071
1000	0.267 565 737 998 666	10 000	2.711 571 309 002 465 6
1000	0.277 454 562 001 366 87	20 000	5.386 473 500 999 273
1000	0.274 554 645 999 160 13	20 000	5.265 557 869 999 611
1000	0.264 056 780 000 828 44	20 000	5.386 713 730 997 144
1000	0.265 799 308 999 703 5	20 000	5.343 137 036 998 087
1000	0.264 801 003 999 309 6	20 000	5.431 827 074 000 466
2000	0.518 065 825 999 656 2	20 000	5.405 627 173 000 539
2000	0.570 858 339 000 551 5	20 000	5.390 988 753 999 409
2000	0.528 718 462 002 871 1	20 000	5.386 078 308 001 743
2000	0.588 411 423 999 787 1	20 000	5.452 584 265 996 847 5
2000	0.582 577 511 999 261 3	20 000	5.199 394 284 998 561 5
2000	0.597 772 474 000 521 5	50 000	13.127 972 212 001 623
2000	0.556 758 733 000 606 3	50 000	13.112 231 078 001 06
2000	0.535 026 976 998 779	50 000	13.227 987 500 999 006
2000	0.544 425 897 001 929 1	50 000	13.208 480 075 001 717
2000	0.522 405 329 000 321 2	50 000	13.343 738 932 999 258
5000	1.302 161 520 998 197 4	50 000	13.439 049 514 003 273
5000	1.310 512 535 997 986 6	50 000	13.592 552 680 998 779
5000	1.308 456 934 999 412 6	50 000	13.342 658 530 000 335
5000	1.307 413 678 998 273	50 000	13.033 756 085 998 903
5000	1.299 983 690 001 681 7	50 000	13.246 800 770 997 652
5000	1.307 366 578 999 790 4	100 000	26.452 402 930 997 778
5000	1.323 922 187 999 414 7	100 000	26.520 914 748 998 62
5000	1.338 103 993 999 539	100 000	26.545 697 263 001 784
5000	1.328 483 872 999 640 9	100 000	26.640 788 923 999 935
5000	1.314 490 157 001 273 5	100 000	26.684 191 517 997 533
10 000	2.713 525 969 997 135 7	100 000	26.557 492 915 999 32
10 000	2.717 679 110 999 597 5	100 000	26.195 931 261 998 34
10 000	2.690 974 908 000 498 5	100 000	26.522 350 662 002 28
10 000	2.638 393 725 999 776 5	100 000	26.213 136 045 000 283
10 000	2.671 876 385 000 359 7	100 000	26.798 035 271 000 117

Table 9.6: Benchmark runtimes (mitmproxy-wireguard, local system, 1000 bytes per packet)

Packets	Runtime /s	Packets	Runtime /s
1000	3.162 707 505 999 989 6	10 000	30.757 553 628 999 972
1000	2.443 998 925 999 992	10 000	28.985 757 314 000 01
1000	2.407 320 584 999 993 7	10 000	27.489 316 504 999 92
1000	2.805 142 768 999 999 7	10 000	31.293 547 040 000 02
1000	2.771 073 156	10 000	26.386 314 405 999 997
1000	2.926 086 337 999 990 4	20 000	63.210 990 025 999 99
1000	3.091 881 399 000 002	20 000	60.706 880 544
1000	3.068 396 515 000 003 4	20 000	56.019 443 185
1000	3.128 955 382 000 001	20 000	54.448 480 169 000 05
1000	3.110 921 271 999 999	20 000	60.609 655 994 000 036
2000	6.218 189 545 000 001	20 000	65.972 269 167 000 04
2000	6.332 153 851 000 001	20 000	57.682 432 017 999 986
2000	5.925 691 771 000 004	20 000	60.169 552 698 999 95
2000	4.964 154 350 999 991	20 000	57.493 081 655 999 96
2000	4.718 228 914 000 008	20 000	55.528 621 991 000 136
2000	4.888 197 665 999 996	50 000	151.683 606 400 999 9
2000	5.275 927 448 999 994	50 000	144.241 473 366 999 8
2000	6.453 565 776 999 994	50 000	145.942 042 396 000 03
2000	6.505 509 601 999 989	50 000	145.811 670 006 999 98
2000	6.479 473 431 999 992	50 000	147.855 065 374 999 87
5000	15.715 430 780 000 02	50 000	149.009 609 959 000 27
5000	15.734 753 167 999 997	50 000	152.059 588 805 999 74
5000	14.035 682 269 999 995	50 000	143.467 750 231 000 08
5000	12.417 623 493 999 997	50 000	146.425 219 064 999 96
5000	15.560 091 289 000 013	50 000	149.367 332 179 000 2
5000	15.653 612 096 000 018	100 000	305.618 528 255
5000	15.769 737 119 000 013	100 000	307.185 749 201 999 9
5000	16.106 333 910 000 046	100 000	289.509 102 952 999 9
5000	14.213 877 252 999 964	100 000	293.930 143 942 000 1
5000	12.098 061 182 999 97	100 000	296.391 752 162 999 64
10 000	34.492 101 002 000 03	100 000	296.874 763 198 000 44
10 000	25.730 634 544 999 987	100 000	300.020 756 760 000 04
10 000	28.779 093 071 000 034	100 000	299.274 625 333 999 2
10 000	29.091 104 072 000 007	100 000	300.003 072 491 000 24
10 000	26.981 896 785 000 004	100 000	306.330 804 889 999 85

Table 9.7: Benchmark runtimes (Python asyncio, local network, 1000 bytes per packet)

Packets	Runtime /s	Packets	Runtime /s
1000	3.793 304 534 999 947 5	10 000	34.821 278 410 999 87
1000	3.574 513 163 000 006	10 000	43.052 497 463 000 01
1000	3.342 341 664 000 059 6	10 000	38.115 639 179 000 03
1000	3.575 248 643 000 009	10 000	37.387 160 265 000 15
1000	3.478 244 340 999 936	10 000	36.318 693 413 000 12
1000	3.615 890 495 000 030 6	20 000	74.471 766 127 000 14
1000	3.676 162 473 999 966	20 000	76.924 743 232 000 08
1000	3.596 971 521 999 99	20 000	76.523 460 254 000 16
1000	3.727 407 027 999 902 3	20 000	70.616 685 084 999 97
1000	3.655 263 969 999 964 5	20 000	74.384 014 528 999 84
2000	7.567 333 302 999 941	20 000	70.714 998 097 000 03
2000	7.374 637 612 000 015	20 000	76.614 985 817 999 92
2000	7.378 119 307 999 896	20 000	70.957 038 56
2000	7.640 110 106 999 941	20 000	74.103 587 180 999 97
2000	7.176 956 469 999 936	20 000	76.282 277 960 000 1
2000	7.332 760 238 999 981	50 000	185.435 577 822 000 14
2000	6.817 267 241 000 081	50 000	180.754 717 535 000 05
2000	7.159 158 885 000 124	50 000	185.759 349 949 999 9
2000	7.119 312 927 000 01	50 000	182.555 185 958 000 04
2000	7.342 415 024 000 047	50 000	180.618 083 385 000 32
5000	18.595 441 986 999 96	50 000	195.718 769 439 999 96
5000	17.678 711 755 999 984	50 000	188.516 070 656
5000	16.395 253 162 000 017	50 000	178.051 132 372 999 75
5000	18.087 470 578 999 955	50 000	186.433 696 061 999 85
5000	17.962 535 997 000 032	50 000	184.966 574 347
5000	17.637 772 671 999 983	100 000	370.097 150 001 000 05
5000	22.247 207 706 999 916	100 000	356.311 271 839 999 8
5000	17.262 015 693 999 956	100 000	377.923 454 432 000 65
5000	17.034 029 701 999 998	100 000	372.978 172 971 000 3
5000	17.650 403 137 000 012	100 000	367.018 512 445 999 5
10 000	34.588 406 609	100 000	353.496 507 779 000 24
10 000	36.565 368 881 000 04	100 000	378.326 236 187 000 2
10 000	36.887 099 323 000 03	100 000	358.620 071 204 999 5
10 000	36.922 760 474 000 14	100 000	371.457 644 431 000 5
10 000	34.887 330 880 000 036	100 000	363.931 632 387 999 34

Table 9.8: Benchmark runtimes (mitmproxy-wireguard, local network, 1000 bytes per packet)