



SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN

Networks Final Project Report

Team WAN

	Alex Wang Wei Jie
	Dev Bahl
	Jose Johnson Emerson Raja
	Li Yiwen
	Shang Zewen

Table of Content

Table of Content	2
Abstract	3
Glossary	4
Problem Statement	5
References	6
Methodology	7
Experiment & Learning Points	9
Experiment Infrastructure	9
Experiment 1: Throughput Analysis with iPerf	10
Experiment 1.1: iPerf Default	10
Experiment 1.2: iPerf with Wondershaper	12
Experiment 2: CPU Usage with Speedtest	14
Experiment 3: Packet Analysis with Wireshark	16
Experiment 3.1: Instantiation Analysis	17
Experiment 3.2: Wireshark with Speedtest	19
Experiment 3.3: Network Interface analysis with wget	21
Experiment 4: Censorship Bypass & Traffic Obfuscation	24
Internet Censorship via Packet Inspection	24
Deep Packet Inspection with ntopng	25
Traffic Obfuscation	26
Conclusion & Application	28
Appendix	29
Experiment 1.1: - iPerf	29
Experiment 1.2: - iPerf with Wondershaper	30
Experiment 2: - Speedtest	30
Experiment 3: - Packet Analysis with Wireshark	31

Abstract

Even before the COVID-19 pandemic, there was a significant increase in remote working, as technological advancements allowed teams to successfully communicate and work remotely. With this shift to remote working, many organizations and commercial companies need to transfer data between locations in a secure manner. While this was easy to ensure with on-premises intranet infrastructures, with remote working, companies now sought a solution that was effective over the internet as well. This has led many companies to consider the adoption of Virtual Private Network (VPN) tools. VPNs secure communication, by successfully masking internet data allowing for safe remote working. WireGuard and OpenVPN are two popular examples of VPN tools. WireGuard is a relatively recent application whereas OpenVPN is a more mature solution. The design of WireGuard has accounted for some of the shortcomings of OpenVPN and for that reason, as well as the popularity of these two tools, we have decided to focus on them in our project.

In this project, we compare two popular VPN/tunnelling implementations (WireGuard, OpenVPN) and evaluate their performance by conducting comparative quantitative experiments before doing a qualitative analysis on the data gathered.

Glossary

Terminology	Explanation
VPN	A virtual private network (VPN) provides privacy and anonymity online by creating a private network from a public internet connection.
Throughput	Throughput measures the rate of data received successfully over a period of time, measured in bits per second.
OpenVPN	OpenVPN is a VPN system that implements techniques to create secure point-to-point or site-to-site connections in routed or bridged configurations and remote access facilities.
WireGuard	WireGuard is a communication protocol that implements VPN techniques to create secure point-to-point connections in routed or bridged configurations.
Wireshark	Wireshark is a network protocol analyzer that allows users to observe information related to packets transmitted and received.
iPerf3	iPerf3 is a popular tool used to measure and tune network performance. iPerf3 has client and server functionality, and can create data streams to measure the throughput between the two ends in one or both directions.
wget	wget is a software package for retrieving files using HTTP and FTP.
Wondershaper	Wondershaper is a tool to limit the network bandwidth and allow users to do traffic control easily.
Speedtest	Speedtest is a tool to test internet connection bandwidth.
ntopng	ntopng is computer software that probes a computer network to show network use. It does application-layer detection of protocols, using deep packet inspection, regardless of non-standard ports used.
obfsproxy	obfsproxy is a tool that attempts to circumvent censorship, by transforming the traffic between the client and the server through obfuscation.

Problem Statement

A comparative quantitative analysis of OpenVPN and WireGuard tunneling protocols within a framework followed by a qualitative analysis of our results to relate their strengths and weaknesses to real-world applications.

References

No.	Title	Link
1	Network performance evaluation of VPN protocols	<u>Link</u>
2	Network performance on different Operating Systems	<u>Link</u>
3	Fast, Modern, Secure VPN Tunnel - Blackhat USA 2018	<u>Link</u>
4	IP Address Blocking	<u>Link</u>
5	Deep Packet Inspection	<u>Link</u>
6	Active Probing	<u>Link</u>
7	Wireguard: Next Generation Kernel Network Tunnel	<u>Link</u>
8	Security Analysis of WireGuard	<u>Link</u>

Methodology

From the references gathered, we found one reference that resonated with our objective. We began our experimentation process with the study and emulation of this specific reference:

Evaluation of WireGuard and OpenVPN VPN Solutions

Author: Ahmad Anbarje

Author: Mohammed Sabbagh

Supervisor: Diego Perez Palacin

Link: [Link](#)

Problem Statement: Benchmark the throughput between the two types of VPN solutions–WireGuard and OpenVPN.

Approach: The evaluation is done by four different experiments to measure the maximum throughput of each of the VPN solutions, and a theoretical study on the encryption techniques that each VPN solution employs.

Findings:

- 1. Higher Throughput for WireGuard*
- 2. Problems with connectivity for WireGuard*
- 3. WireGuard gives higher packet loss*
- 4. OpenVPN uses more CPU resources*

After studying the findings from the reference in focus we decided to emulate their experiment (*Exp 1.1*) to observe the findings ourselves. We used this reference to help design the methodology for our project.

The Methodology employed in our project is as follows:

Performance Analysis:

1. Throughput
 - a. Throughput within the Internal Network
 - Tool Used: iPerf
 - b. Throughput beyond the Internal Network
 - Tool Used: Speedtest
2. Systems Resources Used
 - a. Tool Used: top
3. Packet Analysis
 - a. Tool Used: Wireshark

Censorship Bypass and Traffic Obfuscation:

1. Tool Used:
 - a. ntopng
 - b. obfsproxy

Experiment & Learning Points

Experiment Infrastructure

To emulate a VPN connection between a server and a client, we decided to set up 2 EC2 instances. The configuration of the EC2 instances are detailed below:

Account Type	AWS educate EC2
Operating System	Ubuntu 20.04.1 LTS
Instance type	t2.micro
Region	us-east-1

Table 1. AWS EC2 instance configuration

The first is the VPN server, and the second will be the VPN client. We implemented both VPNs, OpenVPN and WireGuard in the same set of EC2 instances. This way we can quickly switch between our intended VPNs when performing experiments. Each VPN connection can be established by starting their respective daemon with their respective configuration files. We would then perform the necessary commands after we have connected to the instances via Secure Shell (SSH) protocol.

One of the challenges we faced was the fact that once the client had established a VPN connection, it's IP address would change from it's original AWS allocated IP address to the internal IP address assigned by the VPN. This meant that we would lose SSH connection right after we execute the command to start the VPN connection. This was a challenge because the internal IP address (e.g. 10.0.0.2) was not a public IP address which we could use to connect from our personal machines.

Our workaround was simple. We could still connect to the server because it maintains it's public address even after establishing a VPN connection. So our solution was to connect via SSH to the server first, then connect via SSH into the client from the server SSH connection using the internal VPN IP address.

Experiment 1: Throughput Analysis with iPerf

Experiment 1.1: iPerf Default

The set-up for this experiment was emulated using the reference highlighted earlier.

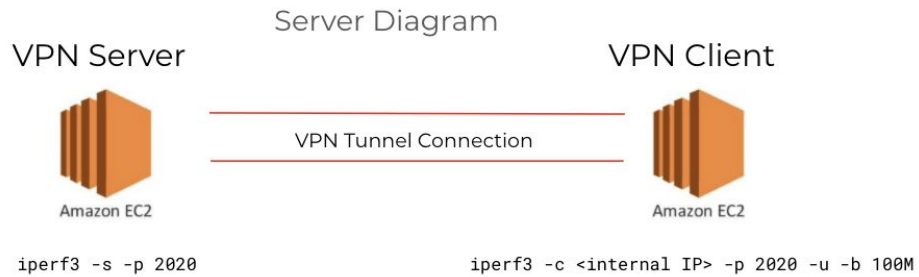


Figure 1. The infrastructure set-up for Experiment 1.1.

Server Command:

```
iperf3 -s -p 2020
```

Client Command:

```
iperf3 -c <internal IP> -p 2020 -u -b 100M
```

After completion of the instance set-up, iPerf3 was used to simulate communication between the two servers and key data was collected. The table below is the average of the data collected over various rounds of experiments. The key metrics in focus are Packet Loss, peak CPU Usage and Throughput.

	OpenVPN	WireGuard
Packet Loss (%)	0.127	0.078
CPU Usage (%)	23.7	3.9
Throughput (Mbits/s)	99.87	99.9

Table 1. Summary of Experiment 1.1 results

There are incongruencies from the data collected by us and that found in the reference. The reference detailed that OpenVPN performs worse than WireGuard with lower throughput values at the same network speed. This however was not the case in the experiment we conducted. We still believe that the experiment conducted by us was valid and decided to build on the future experiments with the data collected here.

We have two main findings from experiment 1.1. Firstly, even though the packet loss rate is different for OpenVPN and WireGuard, it is still relatively low. Secondly, the CPU usage is much lower for OpenVPN as compared to WireGuard.

After analysing the output data above, we were still curious if we could express the differences between the two VPNs to a greater extent or search for new unfound differences. We decided to experiment with the bandwidth bottleneck of our network as we hypothesised that by reducing the bandwidth bottleneck, it would be easier to observe differences in packet loss and we may also be able to observe a more expressed difference in CPU usage. We postulate that as shown in the figure below, as bandwidth increases, packet loss would increase too due to fixed queues in the network being unable to manage the increased influx in incoming packets. Therefore, instead of increasing packet loss for both VPNs we decided to decrease the bandwidth bottleneck to observe if the packet loss ratio (packet loss for OpenVPN / packet loss for WireGuard) would change.

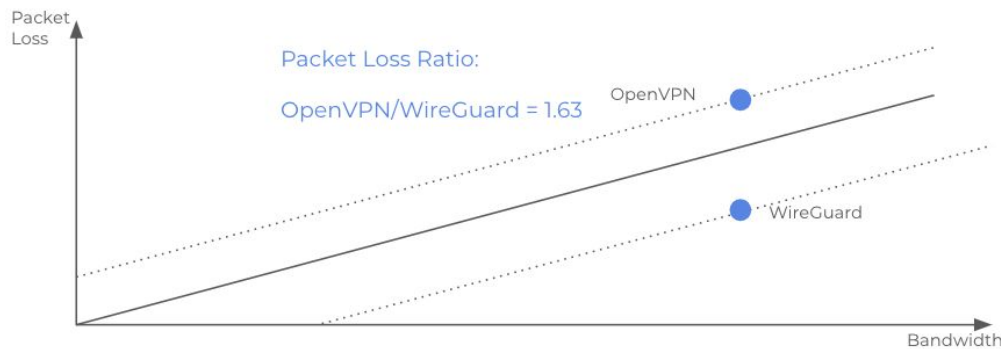


Figure 2. The hypothesis we had of the relationship of Packet Loss and Bandwidth as well as the reference lines for OpenVPN and WireGuard if there were to maintain the same Packet Loss Ratio.

Experiment 1.2: iPerf with Wondershaper

We implemented a new tool, Wondershaper, above our client server to limit the bandwidth and simulate a bottleneck.

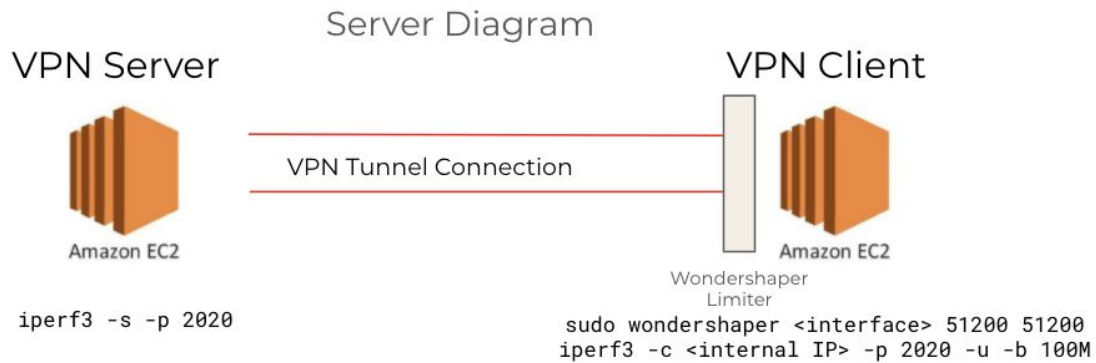


Figure 3. The infrastructure set-up for Experiment 1.2. Wondershaper was implemented on the VPN Client and limited the achievable bandwidth.

Server Command:

```
iperf3 -s -p 2020
```

Client Command:

```
sudo wondershaper <interface> 5120 5120
iperf3 -c <internal IP> -p 2020 -u -b 100M
```

	OpenVPN	WireGuard
Packet Loss (%)	3.6	0.71
Total Datagrams	49508	7388
CPU Usage (%)	11.8	2.3
Throughput (Mbits/s)	51.4	52.0

Table 3. Summary of Experiment 1.2 results

Similar to experiment 1.1, the averaged value for Packet Loss, peak CPU Usage and Throughput was noted. In addition to these values, we also took note of the Total Datagrams transmitted as we noticed there was a big difference between the total datagrams transferred between OpenVPN and WireGuard.

Based on this data, several findings can be drawn. Firstly, similar to experiment 1.1 WireGuard still has significantly lower CPU usage compared to OpenVPN. Additionally, contrary to our hypothesis, there was an increase in packet loss for both VPNs when there

was a reduced bandwidth bottleneck. What we found noteworthy was the increase in the packet loss ratio from 1.63 to 5.00, which indicated to us how packet loss management for OpenVPN may not be on par with that of WireGuard. Furthermore, the total datagrams transferred for OpenVPN is much larger than WireGuard. Upon further research, we found this to be due to the difference in Maximum Transmission Unit (MTU) of the two VPN protocols. OpenVPN has a MTU of 1500 and WireGuard has a MTU of 8921.



Figure 4. The observed Packet Loss for OpenVPN and WireGuard differed from our hypothesis and was higher than before. The Packet Loss Ratio increased tremendously.

Experiment 2: CPU Usage with Speedtest

For this experiment we tested the performance, in the form of throughput and CPU usage, of both VPNs by running Speedtest while connected to each VPN.

Speedtest has the options to connect to the many servers it has available around the world. We chose a server located at Windstream-Ashburn, Vancouver, Canada, with the server ID 17383.

Client (with VPN running) Commands:

```
speedtest -s 17383 -f csv >> ov17383.csv && cat ov17383.csv
```

Experiment results:

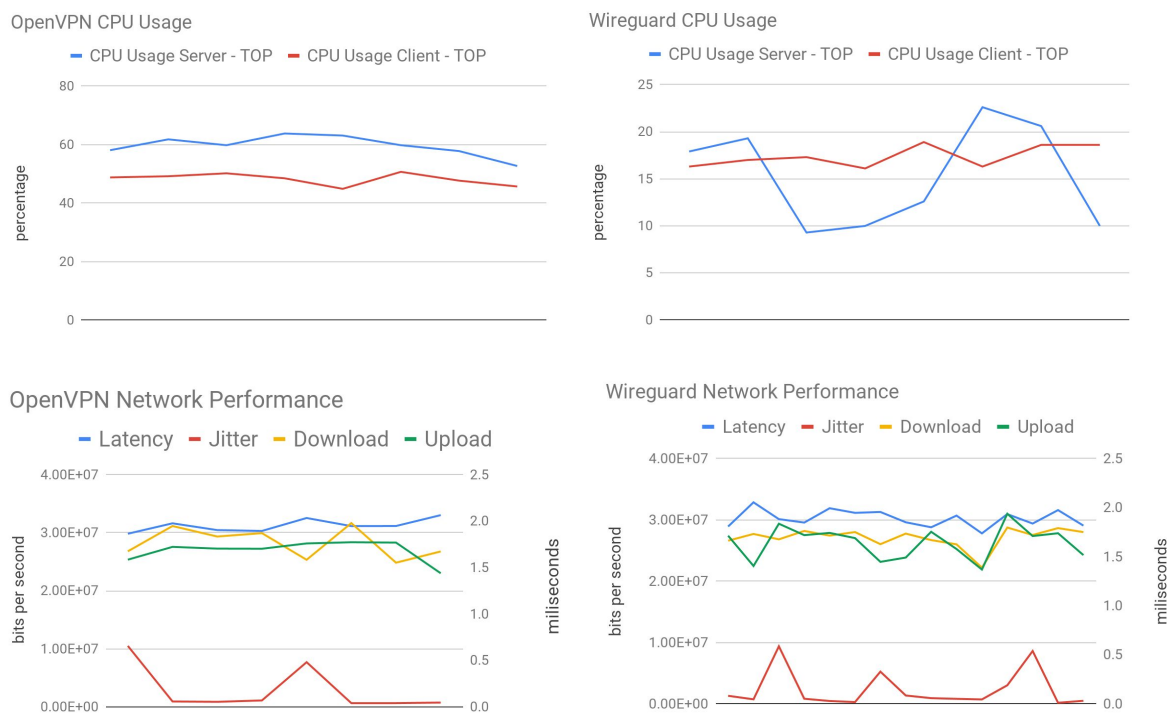


Figure 5. The graphs above contain the plots for the data collected for Experiment 2

After running many rounds of experiments, the data seen above was collected. We observed that the throughput collected for both VPNs was relatively stable however there was a large difference in CPU Usage. The peak CPU utilisation for WireGuard was far less than that for OpenVPN. We attribute this to the reduced complexity of WireGuard, due to its greatly reduced lines of code (4,000 vs 500,000), and the fact the WireGuard's mode of operation remains predominantly in the Kernel space.

We also noted that this experiment was a relatively accurate indicator of the real world as users tend to be mostly concerned about throughput. That said, we feel it is important to notice the variance in CPU usage.

After conducting this experiment, we reflected that Speedtest may not have been a necessarily good tool to use to test the performance of VPNs as it tends to be adaptive to the infrastructure it runs on. For example, if there is a bottleneck, it will not throttle the network tremendously but would instead just return a lower result. Therefore, for following experiments, we explored the use of alternative tools instead.

Experiment 3: Packet Analysis with Wireshark

EC2 instances do not come with a Graphical User Interface, but with a Command Line Interface. This means that we would not be able to run the GUI version of Wireshark. Running Wireshark on CLI is pretty straightforward using a tool called **tshark** that analyzes the network traffic according to the packets being transferred into and out of a machine.

```
sudo tshark
```

The challenge we faced was saving and exporting the results from tshark and analysing interesting packets on the GUI version of Wireshark. We found that Wireshark is able to read files that have the extension .pcapng. Therefore, what we were able to do was change the output format of tshark and save these results in a pcapng file.

After reading through the documentation and a couple of attempts, we realised that tshark would not create the output file, but would only write to an already existing file, with the right privileges. Therefore, we had to create an output file with the right privileges before running tshark.

```
touch <*.pcapng>  
chmod o=rw <*.pcapng>
```

Knowing how to save the results from our tshark runs, we were able to start experimentation proper with the command below.

```
sudo tshark -w <*.pcapng> -F pcapng
```


Experiment 3.1: Instantiation Analysis

For this experiment, we analysed the instantiation process for both VPN tools. We used the tool Wireshark as mentioned above and ran it on both the Client and Server instances to observe the handshake process for OpenVPN and WireGuard.

4	0.220700615	54.198.107.88	172.31.85.159	OpenVPN	96	MessageType: P_CONTROL_HARD_RESET_CLIENT_V2
5	0.220959980	172.31.85.159	54.198.107.88	OpenVPN	108	MessageType: P_CONTROL_HARD_RESET_SERVER_V2
6	0.222379637	54.198.107.88	172.31.85.159	OpenVPN	104	MessageType: P_ACK_V1
7	0.222422408	54.198.107.88	172.31.85.159	SSL	329	Continuation Data
8	0.222619167	172.31.85.159	54.198.107.88	SSL	207	Continuation Data
9	0.223767026	54.198.107.88	172.31.85.159	SSL	380	Continuation Data
10	0.224658717	172.31.85.159	54.198.107.88	SSL	1170	Continuation Data
11	0.224683553	172.31.85.159	54.198.107.88	SSL	552	Continuation Data
12	0.225895580	54.198.107.88	172.31.85.159	OpenVPN	104	MessageType: P_ACK_V1
13	0.227114633	54.198.107.88	172.31.85.159	SSL	1170	Continuation Data
14	0.227154781	54.198.107.88	172.31.85.159	SSL	603	Continuation Data
15	0.228158358	172.31.85.159	54.198.107.88	OpenVPN	104	MessageType: P_ACK_V1
16	0.229493090	172.31.85.159	54.198.107.88	SSL	266	Continuation Data
17	0.229541584	172.31.85.159	54.198.107.88	SSL	324	Continuation Data
18	0.230604870	54.198.107.88	172.31.85.159	OpenVPN	104	MessageType: P_ACK_V1
19	0.230629859	54.198.107.88	172.31.85.159	OpenVPN	104	MessageType: P_ACK_V1
25	1.486353416	54.198.107.88	172.31.85.159	SSL	131	Continuation Data
26	1.486651371	172.31.85.159	54.198.107.88	OpenVPN	104	MessageType: P_ACK_V1
27	1.486684800	172.31.85.159	54.198.107.88	SSL	336	Continuation Data
29	1.505633678	54.198.107.88	172.31.85.159	OpenVPN	104	MessageType: P_ACK_V1

Figure 6. Packets observed on server instance during handshake for OpenVPN

16	3.194940069	54.198.107.88	172.31.85.159	WireGua...	190	Handshake Initiation, sender=0xAEDDB68A
17	3.195301041	172.31.85.159	54.198.107.88	WireGua...	134	Handshake Response, sender=0x2A3C5E88, receiver=0xAEDDB68A
18	3.196396085	54.198.107.88	172.31.85.159	WireGua...	170	Transport Data, receiver=0x2A3C5E88, counter=0, datalen=96

Figure 7. Packets observed on server instance during handshake for WireGuard

As can be seen above, the handshake for OpenVPN required a significant number of packets and also employed SSL whereas WireGuard was able to establish a handshake in 3 packets.

OpenVPN's handshake can be understood by exploring the crypto-agility of OpenVPN. Crypto-agility refers to the capacity to adopt an alternative to the original encryption method or a cryptographic primitive method without having to significantly change the system infrastructure. OpenVPN uses SSL and TLS and supports a variety of ciphers that can be used for cipher negotiation between client and server nodes. The string of packets observed during the handshake process is the cipher negotiation process between our client and server instances.

```
[...]  
BF-CBC 128 bit default key (variable)  
BF-CFB 128 bit default key (variable) (TLS client/server mode)  
BF-OFB 128 bit default key (variable) (TLS client/server mode)  
[...]  
AES-128-CBC 128 bit default key (fixed)  
AES-128-OFB 128 bit default key (fixed) (TLS client/server mode)  
AES-128-CFB 128 bit default key (fixed) (TLS client/server mode)  
AES-192-CBC 192 bit default key (fixed)  
AES-192-OFB 192 bit default key (fixed) (TLS client/server mode)  
AES-192-CFB 192 bit default key (fixed) (TLS client/server mode)  
AES-256-CBC 256 bit default key (fixed)  
AES-256-OFB 256 bit default key (fixed) (TLS client/server mode)  
AES-256-CFB 256 bit default key (fixed) (TLS client/server mode)  
[...]
```

Figure 8. The variety of ciphers supported by OpenVPN.

Alternatively, WireGuard ditches crypto-agility for crypto-versioning. In crypto-versioning, if one of the existing ciphers becomes obsolete, all WireGuard does is release a new version to upgrade and leave the vulnerability behind in its entirety. After the release of the upgrade, the client and server will only need to drop the previous version for the new one and continue. An example of a cipher used by WireGuard is ChaCha20Poly1305.

WireGuard also runs the Elliptic Curve Diffie-Hellman key exchange, authenticated using the static public/private keys, to establish symmetric keys which are used to encrypt and authenticate data messages in the session. However, there is a security caveat to this protocol. The handshake responder cannot assume the connection is authentic until they have received at least one valid data packet; otherwise, they are vulnerable to key-compromise impersonation (KCI) attacks. Therefore, the handshake initiation packet itself is not sufficient proof of authenticity and thus the WireGuard initiator always sends at least one, possibly-empty packet, immediately after a session is created. Thus in our experiment we observe three packets used to establish a valid connection.

Experiment 3.2: Wireshark with Speedtest

This experiment was the very first experiment we conducted with Wireshark to analyse packets being sent by the respective VPNs. The procedure to conduct this experiment was to start **tshark**, to begin capturing packets, followed by running **speedtest** to completion. Once speedtest had completed, we would stop **tshark**.

Time	Source	Destination	Protocol	Length	Info
0.0000000000	172.31.39.104	54.85.98.114	OpenVPN	378	MessageType: P_DATA_V2
0.000081351	172.31.39.104	54.85.98.114	OpenVPN	322	MessageType: P_DATA_V2
0.000166977	172.31.39.104	54.85.98.114	OpenVPN	354	MessageType: P_DATA_V2
0.001128498	54.85.98.114	172.31.39.104	OpenVPN	118	MessageType: P_DATA_V2
0.001147329	54.85.98.114	172.31.39.104	OpenVPN	118	MessageType: P_DATA_V2
0.001164371	54.85.98.114	172.31.39.104	OpenVPN	118	MessageType: P_DATA_V2
0.386208002	54.85.98.114	172.31.39.104	OpenVPN	154	MessageType: P_DATA_V2

Figure 9. Screenshot of packets collected on OpenVPN

Time	Source	Destination	Protocol	Length	Info
2.602217400	10.0.0.1	10.0.0.2	TCP	52	43058 → 22 [ACK] Seq=141 Ack=19237 Wi
2.602263734	10.0.0.1	10.0.0.2	TCP	52	43056 → 22 [ACK] Seq=1 Ack=11201 Win=
2.602296016	10.0.0.1	10.0.0.2	TCP	52	43054 → 22 [ACK] Seq=1 Ack=17749 Win=
2.603016202	10.0.0.1	10.0.0.2	TCP	52	43056 → 22 [ACK] Seq=1 Ack=13373 Win=
2.603039258	10.0.0.1	10.0.0.2	TCP	52	43054 → 22 [ACK] Seq=1 Ack=19985 Win=
2.603043712	10.0.0.1	10.0.0.2	TCP	52	43058 → 22 [ACK] Seq=141 Ack=21473 Wi
2.603068234	10.0.0.1	10.0.0.2	TCP	52	43056 → 22 [ACK] Seq=1 Ack=13473 Win=
2.603082344	10.0.0.1	10.0.0.2	TCP	52	43058 → 22 [ACK] Seq=141 Ack=21573 Wi

Figure 10. Screenshot of packets collected on WireGuard

For packets collected on OpenVPN, our results were as expected. We were able to capture packets that were explicitly labeled OpenVPN. But for WireGuard, we were expecting to find WireGuard packets and not TCP packets.

Upon further investigation, we found that we made a mistake while conducting the experiment. For WireGuard, we did not correctly configure Wireshark to capture packets on the Ethernet network interface (eth0), and thus it defaulted to capturing packets on the VPN network interface (wg0-client-aws). Thus, we refined our tshark commands:

```
sudo tshark -w <*.pcapng> -F pcapng -i <intended_interface>
```

Not only were we able to refine our experimentation methods, this unexpected finding prompted us to research further into network interfaces and how VPN network interfaces interact with the Ethernet network interface. Our research allowed us to come up with the diagram below. ‘Internet’ includes all hosts that are in a VPN connection with the client and the server.

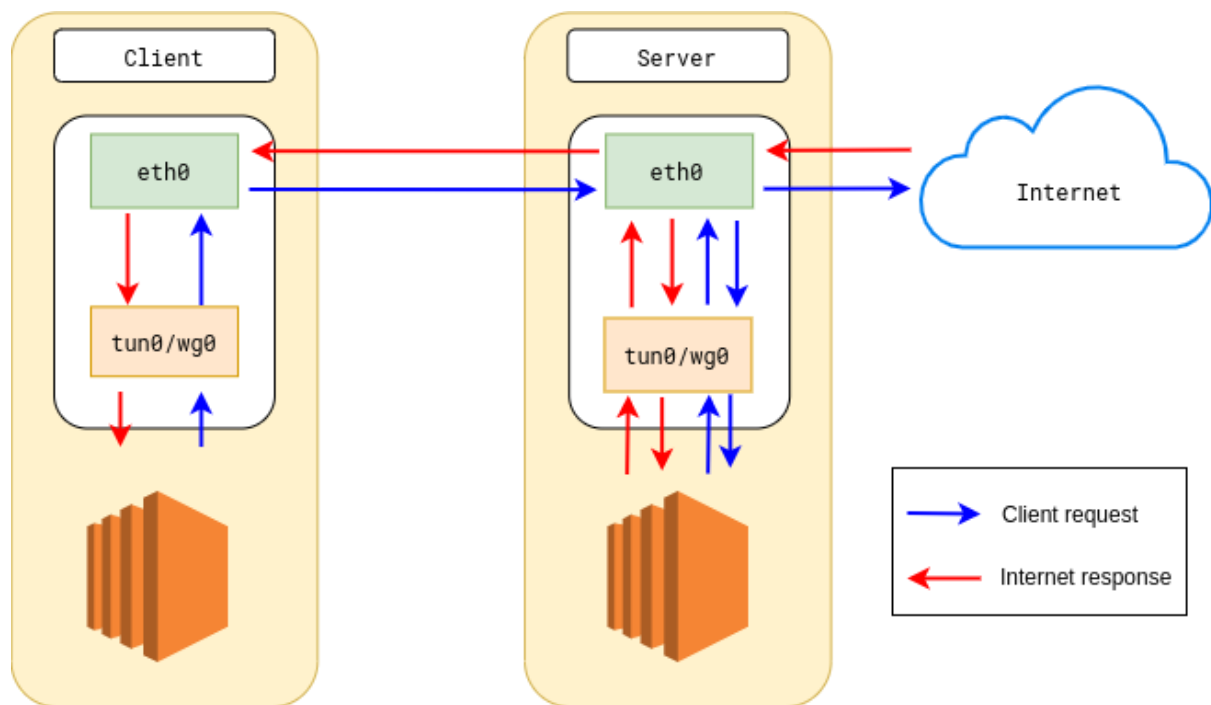


Figure 11. Interactions between the 2 network interfaces for accessing Internet resource

In the diagram above, we see that the network interface responsible for sending out packets to other hosts is the Ethernet network interface (eth0). The role or responsibility of the VPN network interface on the other hand is to perform data encryption and packet management. Packet management involves the assigning of source and destination IP address on top of the information provided by the processing requesting for the internet resource. The VPN network interface does not send out any packets to other hosts.

Information passing through the VPN network interface is encrypted on keys that were agreed upon during the instantiation handshake. These are the keys that the VPN interface uses to encrypt outgoing packets and decrypt incoming packets.

There are 2 main takeaways we can obtain from this diagram. First, the client does not directly interact with the internet. This means that the client IP address is never revealed to the Internet. The Internet is only able to see the IP address of the server. Second, the packets that run between the client and the server are encrypted. This means that regardless of the medium through which the packets are being transmitted, nefarious users will only be able to decipher the message if they can break the encryption keys, which is extremely difficult.

Since the network interfaces have differing roles in a VPN connection, it is justified that the packets they receive would have differing contents. That is the focus for our next experiment.

Experiment 3.3: Network Interface analysis with wget

The aim of this experiment is to study the identity of packets in the Ethernet network interface and the VPN network interfaces. To accomplish this, we made a slight modification to the procedure we carried out in experiment 3.2.

Instead of capturing packets for a **Speedtest**, we decided to use the **wget** command to obtain packets from around the world. Our rationale for using international URLs was to ensure that no form of cached data would affect the packets we received. The table below summarises the different URLs used for this experiment:

URL	IPv4 address	Country
devbahl.com	23.236.62.147	United State of America
https://www.u-tokyo.ac.jp/en/	210.152.243.234	Japan
https://en.snu.ac.kr/	147.46.10.129	Korea

Table 4. Details of different URLs used for experiment 3.3.

All of these URLs gave the same expected results. Snippets of the experiment run for the Korean URL are below.

Time	Source	Destination	Protocol	Length	Info
4.451057774	54.85.98.114	172.31.39.104	OpenVPN	118	MessageType: P_DATA_V2
4.451083171	54.85.98.114	172.31.39.104	OpenVPN	118	MessageType: P_DATA_V2
4.451149195	172.31.39.104	54.85.98.114	OpenVPN	722	MessageType: P_DATA_V2
4.451448721	54.85.98.114	172.31.39.104	OpenVPN	118	MessageType: P_DATA_V2
4.451467372	54.85.98.114	172.31.39.104	OpenVPN	118	MessageType: P_DATA_V2
4.451487910	54.85.98.114	172.31.39.104	OpenVPN	118	MessageType: P_DATA_V2
4.451746437	54.85.98.114	172.31.39.104	OpenVPN	118	MessageType: P_DATA_V2
4.451765971	54.85.98.114	172.31.39.104	OpenVPN	118	MessageType: P_DATA_V2
4.451808585	54.85.98.114	172.31.39.104	OpenVPN	118	MessageType: P_DATA_V2
4.452068318	54.85.98.114	172.31.39.104	OpenVPN	118	MessageType: P_DATA_V2
4.452086806	54.85.98.114	172.31.39.104	OpenVPN	118	MessageType: P_DATA_V2
4.452128901	54.85.98.114	172.31.39.104	OpenVPN	118	MessageType: P_DATA_V2
4.454446581	172.31.39.104	54.85.98.114	OpenVPN	202	MessageType: P_DATA_V2
4.454463843	172.31.39.104	54.85.98.114	OpenVPN	202	MessageType: P_DATA_V2
4.454475681	172.31.39.104	54.85.98.114	OpenVPN	202	MessageType: P_DATA_V2
4.454859617	172.31.39.104	54.85.98.114	OpenVPN	1464	MessageType: P_DATA_V2
4.454876089	172.31.39.104	54.85.98.114	OpenVPN	392	MessageType: P_DATA_V2
4.454884291	172.31.39.104	54.85.98.114	OpenVPN	1464	MessageType: P_DATA_V2
4.454891112	172.31.39.104	54.85.98.114	OpenVPN	344	MessageType: P_DATA_V2

Frame 248: 135 bytes on wire (1080 bits), 135 bytes captured (1080 bits) on interface eth0, id 0

Ethernet II, Src: 0e:87:ab:c9:0b:27 (0e:87:ab:c9:0b:27), Dst: 0e:c4:51:73:0d:0f (0e:c4:51:73:0d:0f)

Internet Protocol Version 4, Src: 172.31.39.104, Dst: 54.85.98.114

User Datagram Protocol, Src Port: 59462, Dst Port: 1194

OpenVPN Protocol

Figure 12. Packets collected from Korea URL on eth0 network interface

Time	Source	Destination	Protocol	Length	Info
2.614928128	10.8.0.2	110.234.5.57	TLSv1.2	444	Client Hello
2.690550892	110.234.5.57	10.8.0.2	TCP	52	443 → 58960 [ACK] Seq=1 Ack=393 Win=30080
2.691845211	110.234.5.57	10.8.0.2	TLSv1.2	1398	Server Hello
2.691856441	10.8.0.2	110.234.5.57	TCP	52	58960 → 443 [ACK] Seq=393 Ack=1347 Win=64
2.691870066	110.234.5.57	10.8.0.2	TLSv1.2	841	Certificate, Server Key Exchange, Server
2.691875448	10.8.0.2	110.234.5.57	TCP	52	58960 → 443 [ACK] Seq=393 Ack=2136 Win=64
2.692480596	10.8.0.2	110.234.5.57	TLSv1.2	178	Client Key Exchange, Change Cipher Spec,
2.767849920	110.234.5.57	10.8.0.2	TLSv1.2	310	New Session Ticket, Change Cipher Spec, E
2.767867809	10.8.0.2	110.234.5.57	TCP	52	58960 → 443 [ACK] Seq=519 Ack=2394 Win=64
2.770885943	10.8.0.2	110.234.5.57	TLSv1.2	220	Application Data
2.885586866	110.234.5.57	10.8.0.2	TCP	52	443 → 58960 [ACK] Seq=2394 Ack=687 Win=31
3.458274303	110.234.5.57	10.8.0.2	TCP	1398	443 → 58960 [ACK] Seq=2394 Ack=687 Win=31
3.458296712	10.8.0.2	110.234.5.57	TCP	52	58960 → 443 [ACK] Seq=687 Ack=3740 Win=64
3.458308311	110.234.5.57	10.8.0.2	TCP	1398	443 → 58960 [ACK] Seq=3740 Ack=687 Win=31
3.458313372	10.8.0.2	110.234.5.57	TCP	52	58960 → 443 [ACK] Seq=687 Ack=5086 Win=64
3.458320937	110.234.5.57	10.8.0.2	TCP	1398	443 → 58960 [ACK] Seq=5086 Ack=687 Win=31
3.458324216	10.8.0.2	110.234.5.57	TCP	52	58960 → 443 [ACK] Seq=687 Ack=6432 Win=64
3.458331275	110.234.5.57	10.8.0.2	TCP	1398	443 → 58960 [ACK] Seq=6432 Ack=687 Win=31
3.458334610	10.8.0.2	110.234.5.57	TCP	52	58960 → 443 [ACK] Seq=687 Ack=7778 Win=60

Frame 363: 52 bytes on wire (416 bits), 52 bytes captured (416 bits) on interface tun0, id 0
 Raw packet data
 Internet Protocol Version 4, Src: 10.8.0.2, Dst: 110.234.5.57
 Transmission Control Protocol, Src Port: 58960, Dst Port: 443, Seq: 687, Ack: 14508, Len: 0

Figure 13. Packets collected from Korea URL on tun0 network interface

Time	Source	Destination	Protocol	Length	Info
2.678239222	10.0.0.2	110.234.5.57	TCP	52	41044 → 443 [ACK] Seq=1 Ack=1 Win=62208 L
2.680150867	10.0.0.2	110.234.5.57	TLSv1.2	444	Client Hello
2.756112039	110.234.5.57	10.0.0.2	TCP	52	443 → 41044 [ACK] Seq=1 Ack=393 Win=30080
2.757381697	110.234.5.57	10.0.0.2	TLSv1.2	1500	Server Hello
2.757394512	10.0.0.2	110.234.5.57	TCP	52	41044 → 443 [ACK] Seq=393 Ack=1449 Win=60
2.757410139	110.234.5.57	10.0.0.2	TLSv1.2	739	Certificate, Server Key Exchange, Server
2.757416359	10.0.0.2	110.234.5.57	TCP	52	41044 → 443 [ACK] Seq=393 Ack=2136 Win=60
2.758038521	10.0.0.2	110.234.5.57	TLSv1.2	178	Client Key Exchange, Change Cipher Spec,
2.835376360	110.234.5.57	10.0.0.2	TLSv1.2	310	New Session Ticket, Change Cipher Spec, E
2.835411194	10.0.0.2	110.234.5.57	TCP	52	41044 → 443 [ACK] Seq=519 Ack=2394 Win=58
2.837769398	10.0.0.2	110.234.5.57	TLSv1.2	220	Application Data
2.953732164	110.234.5.57	10.0.0.2	TCP	52	443 → 41044 [ACK] Seq=2394 Ack=687 Win=31
3.509603312	110.234.5.57	10.0.0.2	TCP	1500	443 → 41044 [ACK] Seq=2394 Ack=687 Win=31
3.509631339	10.0.0.2	110.234.5.57	TCP	52	41044 → 443 [ACK] Seq=687 Ack=3842 Win=58
3.509640482	110.234.5.57	10.0.0.2	TCP	1500	443 → 41044 [ACK] Seq=3842 Ack=687 Win=31
3.509646101	10.0.0.2	110.234.5.57	TCP	52	41044 → 443 [ACK] Seq=687 Ack=5290 Win=57
3.509650911	110.234.5.57	10.0.0.2	TCP	1500	443 → 41044 [ACK] Seq=5290 Ack=687 Win=31
3.509654473	10.0.0.2	110.234.5.57	TCP	52	41044 → 443 [ACK] Seq=687 Ack=6738 Win=56
3.509659163	110.234.5.57	10.0.0.2	TCP	1500	443 → 41044 [ACK] Seq=6738 Ack=687 Win=31

Frame 245: 52 bytes on wire (416 bits), 52 bytes captured (416 bits) on interface wg0-client-aws, id 0
 Raw packet data
 Internet Protocol Version 4, Src: 10.0.0.2, Dst: 110.234.5.57
 Transmission Control Protocol, Src Port: 41044, Dst Port: 443, Seq: 687, Ack: 18322, Len: 0

Figure 14. Packets collected from Korea URL on wg0-client-aws network interface

All of these packets were collected on the client. There are three interesting observations to be pointed out.

Firstly, from Figure 12, it is observed that the OpenVPN packets are encapsulated in User Datagram Protocol (UDP). This means that the communication between the server and the client is done in UDP. This is an expected result, the OpenVPN connection was set-up to use UDP instead of TCP. This was done to make a fair analysis between WireGuard and OpenVPN, since WireGuard only runs on UDP.

Secondly, also from Figure 12, the source and destination of these packets are the public IP address of the server and the private IP address of the client, interchangeably. This means that the client is only exchanging packets with the server and no other host. This is also expected, as we have discussed previously the client is only in communication with the server, and never with any other host.

Finally, from Figure 13 and 14, in both OpenVPN and WireGuard network interfaces, the packets that are captured do not have the client's IP address, but rather the IP address of the server or the IP address of the Internet resource requested interchangeably. This shows that the client receives packets from the Internet resource through the server, where the server forwards the packets it received from the Internet resource to the client via the VPN connection. This interaction is highlighted by the red arrows on Figure 11.

These three observations show 2 main things. Firstly, the packets that travel between the hosts in a VPN connection are encrypted and cannot be deciphered without the encryption keys, which only the server and client possess. This means that when connected to a VPN, the security of the connection link does not need to be very secure, since the packets that are being transmitted are already encrypted.

Secondly, the VPN server is the only host in a VPN connection that is responsible for requesting and retrieving packets from Internet resources. It then just forwards the packets it receives to the respective clients, encrypted as a VPN packet.

Experiment 4: Censorship Bypass & Traffic Obfuscation

Internet Censorship via Packet Inspection

Internet censorship works by disallowing internet users from accessing content that is considered unauthorised by authorities. One of the ways to enforce the censorship is via packet inspection. There are mainly 3 types of packet inspections:

Shallow Packet Inspection (SPI) checks for Internet and Link layers' source and destination IP addresses. Basic firewalls employ this method to block unwanted connections from the blacklisted IP addresses.

Medium Packet Inspection (MPI) is usually implemented as a proxy to analyse packet headers and packet types based on data format.

Deep Packet Inspection (DPI) does pattern matching on the packet as a whole to identify and classify encrypted packets.

All three methods will drop packets if needed to enforce censorship laws.

The following depicts a typical Internet censorship network architecture. DPI is often implemented on ISP level as a DPI box and scans all incoming and outgoing traffic.

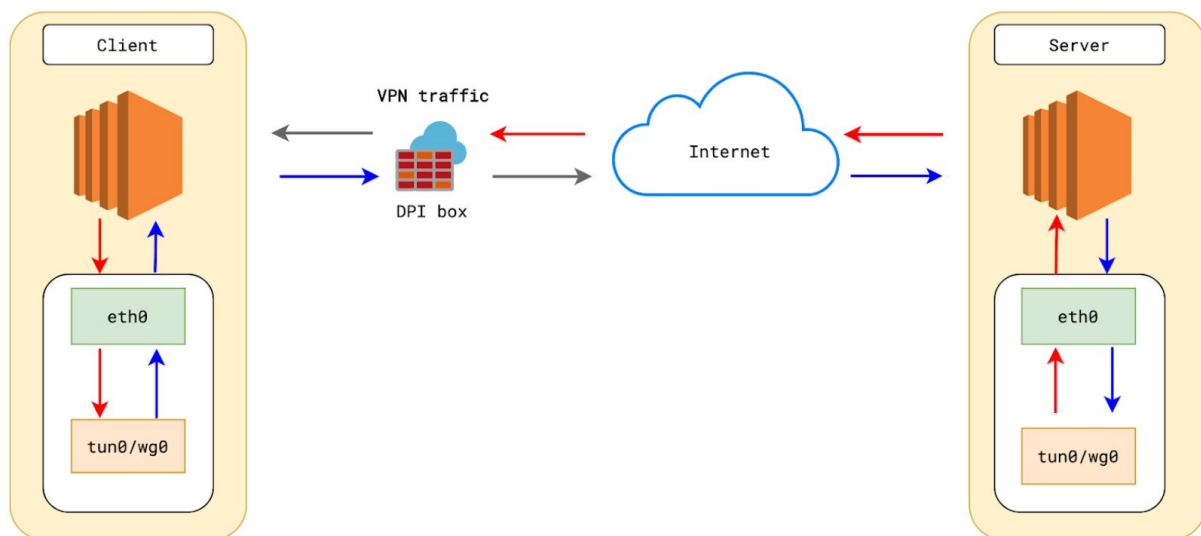


Figure 16: Diagram showing role of DPI box in obfuscation

When the DPI box detects VPN traffic originating from its client, the packets will be dropped, as shown by the grey arrow. Likewise, when VPN traffic from the Internet tries to reach its clients, the incoming packets will also be dropped - deterring clients from accessing censored content.

Deep Packet Inspection with ntopng

In our experiment, we used the tool, **ntop**, to study DPI and how traffic obfuscation is used to circumvent censorship attempts.

ntopng uses nDPI, which is a superset of OpenDPI library, does application-layer detection of protocols, regardless of non-standard ports used. The list of supported protocols can be found on <https://www.ntop.org/products/deep-packet-inspection/ndpi/>.

Operating System	Ubuntu 20.04.1 LTS
Tools installed	OpenVPN WireGuard ntopng obfsproxy

Table 5. Ubuntu instance configuration

The local instance connects to our EC2 VPN via OpenVPN and WireGuard separately. The traffic is then captured using ntopng and the results are as follows:


Flow Peers [Client / Server]	ubuntu 🇺🇸:51949 [00:0C:29:4D:58:F7] ↔ 54.85.98.114 🇺🇸:1194 [00:50:56:E8:46:9A]	
Protocol / Application	UDP / OpenVPN.Amazon (Web) 👍	
First / Last Seen	20/11/2020 02:24:05 [01:40 ago]	20/11/2020 02:25:43 [00:02 ago]
Total Traffic	Total: 21.5 MB ↑	Goodput: 20.5 MB (95.3 %) ↑
	Client → Server: 12,453 Pkts / 9.4 MB ↑	Client ← Server: 12,656 Pkts / 12.1 MB ↑
		

Figure 17. Results from ntopng via OpenVPN connection


Flow Peers [Client / Server]	ubuntu 🇺🇸:54712 [00:0C:29:4D:58:F7] ↔ 54.85.98.114 🇺🇸:51900 [00:50:56:E8:46:9A]	
Protocol / Application	UDP / WireGuard.Amazon (VPN) 👍	
First / Last Seen	20/11/2020 02:25:55 [00:48 ago]	20/11/2020 02:26:43 [< 1 sec ago]
Total Traffic	Total: 50.3 MB ↑	Goodput: 48.1 MB (95.7 %) ↑
	Client → Server: 25,292 Pkts / 21.3 MB ↑	Client ← Server: 28,828 Pkts / 28.9 MB ↑
		

Figure 18. Results from ntopng via WireGuardconnection

On both instances, DPI is able to accurately detect the local instance running a VPN protocol to tunnel its traffic. Given the ease of detecting VPN traffic, the packets can be easily dropped and the connection blocked.

Traffic Obfuscation

Traffic obfuscation works by adding another layer of complexity by scrambling packet data to hide the original VPN signature that is used by DPI to identify the underlying protocol.

The aim is to disguise VPN traffic as inconspicuous web traffic to circumvent DPI detection.

OpenVPN is often used together with obfsproxy, a traffic obfuscator originally created for TOR. We used obfs2 protocol for obfuscation.

Below is the network architecture diagram showing **obfsproxy** in action:

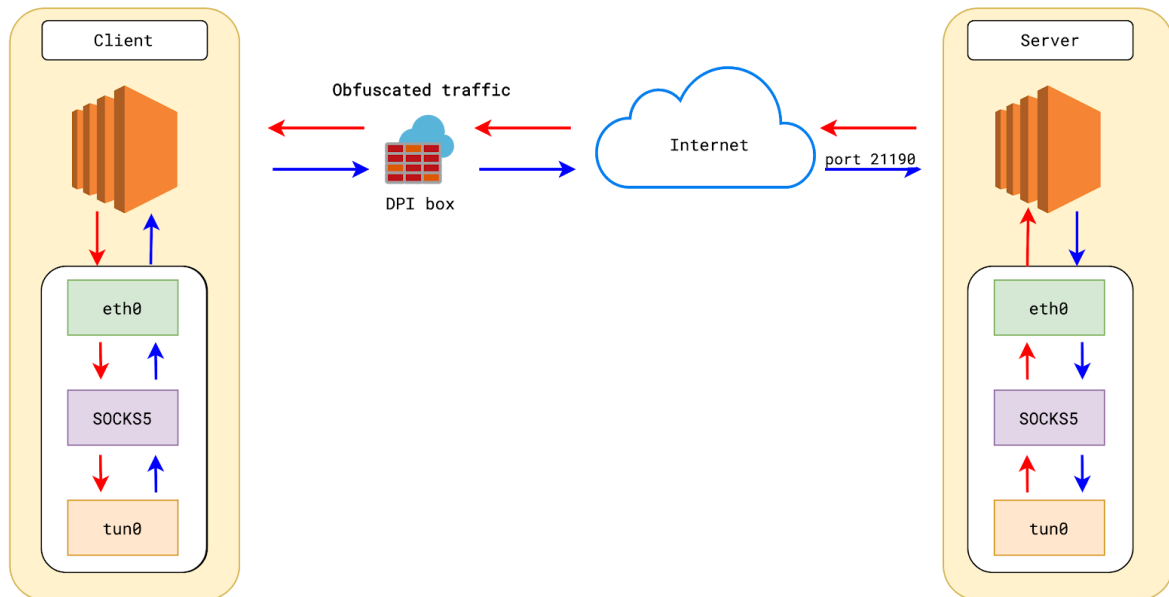


Figure 19. Diagram showing role of SOCKS5 proxy in thwarting DPI box

obfsproxy works by creating a local SOCKS5 proxy that accepts all tun0 packets for OpenVPN, scrambles the data and delivers to the Internet via eth0. On the server end, it runs the same SOCKS5 session with a symmetric key to decrypt and forward the traffic.

DPI should not be able to recognise that the obfuscated traffic contains hidden VPN traffic.

We set up **obfsproxy** as well, and doing ntopng again on the local traffic, the results are:

Flow Peers [Client / Server]	localhost:46674 ↔ localhost:10194	
Protocol / Application	TCP / ? Unknown (Unspecified)	
First / Last Seen	25/11/2020 22:08:48 [02:24 ago]	25/11/2020 22:11:06 [00:06 ago]
Total Traffic	Total: 26 MB —	Goodput: 25.4 MB (98.0 %) ↑
	Client → Server: 3,638 Pkts / 12.6 MB —	Client ← Server: 4,670 Pkts / 13.3 MB —
	localhost:46674	localhost:10194

Figure 20. Result from ntopng on traffic between local ports

Flow Peers [Client / Server]	ubuntu 🇧🇪:54026 [00:0C:29:4D:58:F7] ⇄ 34.228.167.77 🇺🇸:21190 [00:50:56:E8:46:9A]	
Protocol / Application	TCP / Amazon (Web) 🍷	
First / Last Seen	25/11/2020 22:08:48 [03:40 ago]	25/11/2020 22:12:26 [00:02 ago]
Total Traffic	Total: 13.6 MB ↑	Goodput: 12.7 MB (93.5 %) ↑
	Client → Server: 6,267 Pkts / 6.5 MB ↑	Client ← Server: 10,242 Pkts / 7.1 MB ↑
	<div> <div>ubuntu:54026</div> <div>34.228.167.77:21190</div> </div>	

Figure 21. Result form ntopng on traffic via OpenVPN connection

This shows **obfsproxy** is effective in masking the VPN traffic, as **ntopng** fails to identify OpenVPN protocol, instead showing TCP web traffic, which is the desired behaviour.

Conclusion & Application

After conducting our series of experiments, we have gained new and valuable insights into the operation and potential of both VPN tools.

We have found OpenVPN to be the more mature protocol. Being first released in 2001, it has garnered and supported a large user base. This large base of users have successfully created additional tools that are built on top of OpenVPN. An example of this would be obfsproxy, which made it easy to integrate obfuscation.

On the other hand, WireGuard, which is a relatively younger tool, only created since 2016, has its merits in the fact that it uses state-of-the-art technologies to achieve faster and lighter operation. This can be seen by its relatively greater speed as well as better CPU efficiency.

The strengths of OpenVPN and WireGuard can be summarised as follows:

Protocol	OpenVPN	WireGuard
Strengths	Matured Compatible with obfsproxy	CPU efficient Fast network performance Faster encryption Quick handshake

Table 6. Summary of strengths of OpenVPN and Wireguard

With these points in mind, we would like to extend these recommendations for various use cases:

Application	Censorship Circumvention	Access Company Resources	Secure Browsing
Protocol	OpenVPN	WireGuard	WireGuard
Reason	Compatible with obfsproxy	Fast and efficient operation	Fast and efficient operation

Table 7. Application of both VPNs

Appendix

General Commands:

SSH:

- Server:

```
ssh ubuntu@54.85.98.114 -i aws_educate.pem
```

- Client:

```
ssh ubuntu@54.198.107.88 -i aws_educate.pem
```

Aliases:

- Server:

```
alias sshov="ssh ubuntu@10.8.0.2 -i aws_educate.pem"  
alias sshwg="ssh ubuntu@10.0.0.2 -i aws_educate.pem"
```

- Client:

```
alias connectwg="sudo wg-quick up wg0-client-aws.conf"  
alias connectov="openvpn --config aws-openvpn.ovpn"
```

```
alias killwg="sudo wg-quick down wg0-client-aws.conf"  
alias killov="sudo pkill ovpn"
```

Experiment 1.1: - iPerf

- Client:
 - WireGuard

```
iperf3 -c 10.0.0.1 -p 2020 -u -b 100M
```

- OpenVPN

```
iperf3 -c 10.8.0.1 -p 2020 -u -b 100M
```

- Server:

```
iperf3 -s -p 2020
```

Experiment 1.2: - iPerf with Wondershaper

- Client:

```
sudo wondershaper <interface> 51200 51200
```

- WireGuard

```
iperf3 -c 10.0.0.1 -p 2020 -u -b 100M
```

- OpenVPN

```
iperf3 -c 10.8.0.1 -p 2020 -u -b 100M
```

- Server:

```
iperf3 -s -p 2020
```

Experiment 2: - Speedtest

- Client:

```
top  
speedtest -s 17383 -f csv >> wg17383.csv && cat wg17383.csv
```

Experiment 3: - Packet Analysis with Wireshark

Experiment 3.1: - Instantiation Analysis

- Client:

```
touch <*.pcapng>  
chmod o=rw <*.pcapng>  
sudo tshark -w <*.pcapng> -F pcapng -i <interface_name>
```

- Connect to VPN
 - See commands above

- Server:

- Before starting VPN

```
touch <*.pcapng>  
chmod o=rw <*.pcapng>  
sudo tshark -w <*.pcapng> -F pcapng -i <interface_name>
```

- SSH into client
 - See commands above

- For Client:

```
sudo pkill tshark
```

- For Server:

```
sudo pkill tshark
```

- Kill VPN connection
- Export <*.pcapng> file using cyberduck

Experiment 3.2: - Speedtest

- Client:
Connect to VPN

- Server:
 - SSH into client
 - See commands above

```
touch <*.pcapng>
sudo tshark -w <*.pcapng> -F pcapng -i <interface_name>
speedtest -s 17383
sudo pkill tshark
```

- Kill VPN connection
- Export <*.pcapng> file using cyberduck

Experiment 3.3: - wget from around the world

- Client:
 - Connect to VPN
 - See commands above
- Server:
 - SSH into client
 - See commands above

```
touch <*.pcapng>
chmod o=rw <*.pcapng>
sudo tshark -w <*.pcapng> -F pcapng -i <interface_name>
wget <index.html from external sites>
sudo pkill tshark
```

- Kill VPN connection
- Export <*.pcapng> file using cyberduck