Performance comparison of VPN implementations WireGuard, strongSwan, and OpenVPN in a 1 Gbit/s environment

Erik Dekker
University of Amsterdam
erik.dekker@os3.nl

Patrick Spaans
University of Amsterdam
patrick.spaans@os3.nl

ABSTRACT

The goal of this research was to gain insight into how the VPN implementations strongSwan, OpenVPN, the WireGuard kernel implementation (WireGuard-C), and the WireGuard Go implementation (WireGuard-Go) compare to each other in terms of performance when a maximum throughput of 1 Gbit/s can be achieved. We did this by measuring the UDP and TCP goodput, latency, CPU utilization, and connection initiation time for each implementation. As OpenVPN and strongSwan are both configurable in terms of ciphers suites, we measured multiple cipher suites for these implementations. Our results show that strongSwan with an AES-GCM cipher in general achieves the highest goodput, lowest latency, and lowest CPU utilization. WireGuard-C achieved the lowest connection initiation time and OpenVPN the highest. Furthermore, our results show that WireGuard-Go consistently has the highest CPU utilization and it also achieved the highest median latency.

1 INTRODUCTION

The division between the traditional untrusted internet and trusted intranet is becoming less relevant as more organizations host internal services for their customers and employees [1]. As a consequence, these internal services often need to be reached over the internet. To ensure data privacy, this data needs to be secured. One frequently used solution to accomplish this is a Virtual Private Network (VPN).

Well-known VPN implementations include strongSwan and OpenVPN, which enable two endpoints to create a secure connection between each other. However, these implementations are often acknowledged as complex and can be easily misconfigured [2]. In addition, both implementations are based on standardized protocols. While standardized protocols certainly have their benefits, standardization takes time and such implementations are often forced to support obsolete options, such as insecure cryptographic algorithms [1]. This could lead to cryptanalytic attacks [3] and software flaws [4].

WireGuard is a new VPN protocol that aims to be simpler, faster, and leaner than IPsec and better performing than TLS based VPN solutions such as OpenVPN [5]. WireGuard has got a lot of traction recently due to being integrated in Linux kernel version 5.6 [6, 7]. Moreover, US senator Ron Wyden recommended WireGuard to the National Institute of Standards and Technology (NIST) to use it as the default government VPN solution [8].

In order to research these claims, we want to analyse the performance of WireGuard and compare it to existing VPN implementations. Hence, we will measure the encapsulated UDP and TCP goodput, latency, connection initiation time, and CPU utilization of two WireGuard implementations. One WireGuard implementation is written in C and is included in Linux kernel version 5.6, therefore we will call this version *WireGuard-C*. The other implementation is written in Go, therefore we will refer to this as *WireGuard-Go*. In addition, we will also measure the aforementioned metrics of a popular IPsec implementation, namely strongSwan. Furthermore, we will also analyse the open source OpenVPN implementation, which is called OpenVPN Community. From now on we will refer to this implementation as *OpenVPN*.

2 RELATED WORK

Some research into the performance of WireGuard has already been conducted in the past. In 2018, Pudelko published a paper regarding the performance of VPN gateways [9]. For their research, they created their own VPN sandbox *MoonWire*, which allowed them to easily alter their VPN configurations. They created and analysed three different *MoonWire* configurations, these being based on OpenVPN, IPsec, and a hybrid approach of the two. The performance of these configurations on 10 and 40 Gbit/s network connections were measured and compared to those of existing implementations of OpenVPN, IPsec and WireGuard. The analysed WireGuard version was an early version of WireGuard-C. They concluded that none of these existing VPN implementations were fully able to utilize the faster network connections, due to being limited by the kernel. Their own MoonWire implementations bypassed the kernel and therefore obtained better performance results. WireGuard did not scale nearly as well as its competitors in fast data environments due to it spending a large amount of time on locking system resources, which halts the encryption process. In their results, WireGuard turned out to be the worst-performing VPN implementation based on packet processing rates.

Lackorzynski et al. published a paper in 2019, discussing their research into VPN implementations able to transmit Layer 2 Ethernet Frames over the Layer 3 IP protocol [10]. In their research, they analysed which implementation was best-suited for factory environments based on performance and security. They compared numerous implementations among which OpenVPN, IPsec, MACsec, and WireGuard. While MACsec is not a VPN protocol, Lackorzynski et al. believed it could lead to interesting results. Besides this, they also analysed the best performing hardware platform for each protocol, as well as how MACsec could be combined with a Layer 3 VPN implementation. From their research, they concluded that WireGuard is the VPN solution able to reach the highest throughput values, with MACsec obtaining the best latency results.

A similar research was conducted by Mackey et al. in 2020. In this research they compared the performance between the implementations OpenVPN and WireGuard [11]. They do not specify which WireGuard implementation was used in their research. Their

purpose was to analyse the claims made by WireGuard that an unoptimized version of WireGuard should outperform OpenVPN. They conducted this research on virtual machines, both in a cloud environment and on their own physical hardware, in order to test both CPU limits and network interface controller capacity. Their findings show that WireGuard consistently outperforms OpenVPN in each testing setup. They reason that this is the case due to WireGuards use of multithreading.

Another paper discussing the performance of WireGuard was published by Osswald et al. in 2020 [12]. For their research, they connected two virtual servers with a link supporting a traffic speed of 10 Gbit/s. This setup was used to compare the VPN implementations of OpenVPN, strongSwan, and a beta version of WireGuard-C. For each of these implementations network performance for TCP traffic was measured. Osswald et al. conclude that the three different implementations all have their own use cases, with WireGuard and strongSwan both obtaining good performance results in specific use cases.

As for papers not directly involving the WireGuard protocol, Kotuliak et al. published their findings in 2011 on the performance differences between IPsec and TLS based VPN technologies [13]. They concluded that both technologies have their benefits, with IPsec being slightly faster performance-wise and OpenVPN being easier to configure. In 2015, Conjaah et al. published a paper in which they describe the performance differences for OpenVPN when using a TCP or UDP tunnel [14]. They concluded that for OpenVPN, a TCP tunnel led to a more stable connection, whereas a UDP tunnel led to a faster connection at the cost of stability.

3 RESEARCH QUESTIONS

As described earlier, Pudelko, Lackorzynski et al., Mackey et al. and Osswald et al. already researched the performance of WireGuard. However, there is room for improvement. Firstly, Mackey et al. only compared OpenVPN with WireGuard and left the IPsec solutions untouched, while IPsec solutions are expected to be better performing [13]. Secondly, OpenVPN supports a multitude of cipher suites. However, Mackey et al. are unclear as to what cipher suite they investigated. As Osswald et al. had shown, the choice of cipher suite is of importance when analysing performance [12]. Thirdly, Osswald et al., Lackorzynski et al., and Pudelko did research WireGuard-C. However, WireGuard-C was at that time not integrated into the kernel. In addition, Mackey et al. and Osswald et al. mention as future work that it would be interesting to measure the performance again when WireGuard is integrated into the Linux kernel, as is now the case. Fourthly, Osswald et al. and Lackorzynski et al. only measured TCP throughput and there were no results about other metrics like UDP throughput. Fifthly, goodput, instead of throughput, might be more interesting to measure, as this refers to the usable application data that could be send through the VPN tunnel. Finally, none of the mentioned authors conducted research into the AES Galois/Counter Mode (GCM) cipher suite for OpenVPN. Because of these arguments, we think that it is useful to measure WireGuard and the other implementations once again. This leads us to the following main question:

How do the VPN implementations WireGuard-C, WireGuard-Go, strongSwan and OpenVPN compare in terms of performance when the maximum link throughput is 1 Gbit/s between two VPN endpoints?

The main reason we chose to do the measurements in a 1 Gbit/s environment is that we think that VPNs are primarily utilized over the internet. At the time of writing, it is rare to see an internet connection above 1 Gbit/s.

This main question was then divided into multiple sub-questions, each detailing an aspect related to the performance of the implementations. The sub-questions are as follows:

- How do the implementations compare in terms of UDP goodput?
- How do the implementations compare in terms of TCP goodput?
- How do the implementations compare in terms of latency?
- How do the implementations compare in terms of connection initiation time?
- How do the implementations compare in terms of CPU efficiency?

Note that by UDP and TCP goodput we mean the UDP and TCP payload through the VPN tunnel, which, in this case, is additionally encapsulated by UDP for OpenVPN and WireGuard and Encapsulation Security Payload (ESP) for strongSwan.

4 BACKGROUND

This paper measures the performance of four VPN implementations. To provide an overview of the related technologies, we will first explain VPNs in general. Then we will go more in-depth into the specific VPN implementations strongSwan, OpenVPN and WireGuard. In these sections we will cover whether the implementations support multithreading, what key exchange they use, and what encryption ciphers and integrity algorithms they support. In addition, we will cover whether the implementations operate in user or kernel-space, how they encapsulate packets and in which language they are written. An overview of this can be seen in Table 1.

4.1 Virtual Private Networks

Through the use of VPN solutions, clients are able to remotely connect to applications or services without endangering the security of the private network. When initiating a VPN connection, a network tunnel is created between two endpoints of the VPN,

Table 1: Main differences between the VPN implementations. For the item marked with a star: While strongSwan does support multithreading, Linux kernel version 5.6.17 does not.

	strongSwan	**OpenVPN**	**WireGuard**
Multithreaded	Yes*	No	Yes
Key exchange	IKEv1/IKEv2	SSL/TLS	WG
Cipher	Configurable	Configurable	ChaCha20
Integrity	Configurable	Configurable	Poly1305
User/kernel-space	Kernel*	User	Kernel
Encapsulation	ESP	UDP + OVPN	UDP + WG
Language	C	C	C/Go

over which all VPN traffic will be transmitted. VPN implementations can offers data confidentiality as well as integrity through the use of encryption and hashing, which prevents any captured packet from being interpreted or altered. In addition, VPNs can offer message authentication and sender non-repudiation, ensuring that each message is sent from and to the correct source and destination, with the sender unable to deny sending the message. Different VPN implementations make use of different encryption and integrity algorithms, with most implementations supporting multiple options to allow the user to decide what algorithms will be used. These options are not always limited to secure algorithms, with often insecure algorithms still being supported as well. While the use of these insecure algorithms is not recommended, they are still supported due to legacy reasons [15].

The main advantage of VPNs is the fact that physical infrastructure for a private network is no longer needed, with instead the private network being created on top of a public network through the use of tunneling. This solution is not only cheaper, but also more flexible in regards to changes due to the infrastructure being virtual. This advantage comes at a cost, as the usage of security measures affects the performance of the VPNs. Take for example the encryption process. Besides being computationally expensive, it also prevents transmitted data from being compressed, hindering the throughput of the network [16].

These days, popular VPN implementations include OpenVPN and strongSwan, which are a TLS based and an IPsec based VPN implementation respectively. Both of these implementations support secure algorithms for encryption and integrity, and are widely used in the field. As of a couple of years ago, the WireGuard protocol was introduced, which could become a serious contender in the VPN market.

4.2 StrongSwan

StrongSwan is an IPsec implementation released in 2005 that provides encryption and authentication to servers and clients [17]. StrongSwan handles the the keying that is required to setup such a connection. Whereby the Internet Key Exchange (IKE) protocol is used to establish security associations (SA) between two peers. The component that handles this keying in strongSwan is called charon. Charon installs the negotiated SAs into the kernel. The actual traffic encryption and decryption is handled by the IPsec stack of the operating system itself [17]. For simplicity, we will also refer to this as strongSwan in this paper. The traffic that is send through the tunnel is encapsulated in ESP packets.

At the time of writing, the latest strongSwan version is 5.8.4 and is written in C. This version does support multithreading [18]. However, it is important to note that the native Linux kernel IPsec implementation does not fully support multithreading. Instead, IPsec is ran on the same thread as where the hardware interrupt arrives [19].

Since IPsec is highly scalable, strongSwan supports many cipher suites and integrity protocols. The recommended cipher suites being aes128-sha256-modp3072, aes128gcm16-prfsha256-ecp256 and aes256gcm16-prfsha384-ecp384 [20]. For convenience sake, we will refer to these ciphers as *AES-128-CBC*, *AES-128-GCM* and *AES-256-GCM* respectively.

StrongSwan does support both IPsec tunnel and transport mode. The main difference between the two being that transport mode only encapsulates the original transport layer protocol. Tunnel mode encapsulates the original IP header and its payload. While transport mode can be slightly more efficient, since it has no original IP header overhead [21], tunnel mode is more versatile. Tunnel mode can both be used between two endpoints and it can act as a VPN gateway [21, 22]. In addition, it supports NAT traversal. Because of the versatility of tunnel mode, we deem this mode more interesting and therefore chose to research it in our paper.

The keying material that is necessary to setup a SA is handled by IKE, which uses the Diffie-Hellman (DH) algorithm to exchange the keying material. IKE also supports perfect forward secrecy (PFS). IKE exists in two versions, IKEv1 and IKEv2 respectively. Since IKEv2 is used in this research, we will give a brief description of its working. IKEv2 always begins with *IKE_SA_INIT* and *IKE_AUTH* exchanges (a request and response message pair), which normally would consist of four separate messages [23]. The *IKE_SA_INIT* exchange has the function to negotiate cryptographic algorithms, exchange nonces, and to exchange the keying material through the DH exchange. After the *IKE_SA_INIT* exchange, parts of the next exchange, the *IKE_AUTH* messages, are encrypted and integrity protected. The *IKE_AUTH* messages have the function to exchange identities and certificates, as well as to establish the first child SA. The child SA is the controlling entity that holds the state of a IPsec connection. It includes elements such as the encryption algorithm, its keys, authentication algorithm and sequence numbers. All messages that are followed by the *IKE_AUTH* exchange are protected with the cryptographic algorithms and keys negotiated in the *IKE_SA_INIT* exchange.

4.3 OpenVPN

In 2001, the TLS based VPN OpenVPN was released. This VPN implementation makes use of TLS for its key exchange, which allows it to configure a secure connection between endpoints. Unlike IPsec, it is easy to configure, mainly due to the fact that no extensive configuration is needed on the client-side. Two versions of OpenVPN exist, these being the open-source community edition as well as the OpenVPN Access Server. The latter is based on the community edition, but offers additional paid features. For this research, only the community edition was examined [24].

At the time of writing, the latest OpenVPN version is 2.4.9. A beta version of OpenVPN 3 is also available, but was excluded from this research due to being unfinished. OpenVPN 2.4.9 does not support multithreading, which might cause scalability issues when trying to send packets more frequently [11]. Multithreading will be supported in OpenVPN 3, which would potentially impact its performance.

OpenVPN makes use of OpenSSL in order to provide encryption and integrity, and therefore supports a multitude of cipher suites and hashing algorithms. Not only secure algorithms are supported, as legacy algorithms are also provided. By default, when configuring a OpenVPN server, a 256 bit version of AES, combined with the GCM mode of operation is used. As AES-GCM is an Authenticated Encryption with Additional Data (AEAD) cipher, it also provides integrity, preventing the need of a separate integrity algorithm such

as SHA. It makes use of the integrity function GHASH, which takes place simultaneously with the encryption process. In the past, the default encryption algorithm was instead 256 bit AES with Cipher Block Chaining (CBC). In the official manual for OpenVPN, GCM as well as CBC are listed as recommended modes of operation [25]. In this research, both AES-256 as well as AES-128 will be investigated, making the four different OpenVPN cipher suites analysed AES-128-CBC, AES-256-CBC, AES-128-GCM and AES-256-GCM.

An advantage OpenVPN has over the other VPN implementations researched in this paper is the fact that OpenVPN allows for both TCP and UDP tunneling. Using a TCP tunnel leads to a a better stability at the cost of goodput [14]. In this paper, only the UDP tunneling option is investigated in order to fairly compare the different VPN implementations.

OpenVPN can perform at both the *Network* Layer and the *Data Link Layer*. Which layer is used depends on whether *tun* or *tap* is configured. The former is used to provide a network tunnel, while the latter is used in order to bridge two different networks. In our research, we have chosen to use the *tun* option, not only due to it performing in the same layer as the WireGuard protocol, but also due to it being more commonly used for this use case [24].

The most common configuration of OpenVPN involves certificates. In order to confirm whether these certificates are valid, a key exchange is performed between the client and server. A brief overview of this key exchange is as follows. First, the client transmits an initiation request, containing the initial data of the client and a request to the server. This request contains an indicator indicating that TLS will be utilized. A similar request is then returned by the server, to which the client will respond with an acknowledgement as well as the start of the TLS handshake. This handshake, as well as the session key creation, are performed through the use of OpenSSL. After this process has been completed, this key will be acknowledged and a tunnel is created [26, 27].

Besides the standard certificate-based OpenVPN configuration, OpenVPN also supports a peer-to-peer mode where secret keys are used instead. For this mode, the secret keys need to be pre-shared, and no handshake takes place during this mode [24].

4.4 WireGuard

The WireGuard protocol has been in the works since 2015, and has spawned two different implementations, namely a Go and a kernel implementation. At the time of writing, both versions are still in active development, with the kernel implementation having received its first official release early 2020. Both implementations only support the use of UDP tunnels. However, an additional WireGuard header is added during the usage of this tunnel.

WireGuard makes use of a multitude of protocols in order to provide privacy and integrity. The AEAD cipher ChaCha20 combined with Poly1305 provide these security measures. It also prevents the need of hashing the message after already having encrypted it, this is unlike when an algorithm such as SHA is used. In addition, the Noise Protocol Framework is used during the handshake process to further provide anonymity, authentication, and PFS. In order to derive a key from the handshake process, HMAC-based Key Derivation Function (HKDF) is used, combined with BLAKE2 to safely hash these handshake messages. Lastly, Curve25519 is used to generate a public and private key pair, which allows for fast encryption and decryption of messages. During a session, a new session key is generated, which is deleted after a session is closed, thus providing PFS [2].

Both implementations of WireGuard make use of only the aforementioned ciphers, with no room for customization. Due to this, as well as being written efficiently, the codebase of the kernel implementation is less than 4000 lines of code in size. Both the lack of freedom in terms of ciphers and the small codebase lead to improved security. Unlike customization providing VPN implementations, it is not possible to accidentally make a configuration mistake that leads to an insecure implementation. The limited codebase makes the code easier to audit, which makes security easier to be guaranteed [28].

Both implementations of WireGuard are only able to configure a peer to peer connection, and no certificate-based option is involved. This allows for less calculations to be performed, as there are no certificates that need to be checked for their validity. In general, the handshake consists of three separate messages. First, the initiator sends a message to the responder to initiate a connection, to which the responder sends a reply. At this point, the initiator has set up a connection, and is able to transmit data. After the responder receives its first packet of data, it knows that a connection has been set up, and now it is able to finish its initiation as well [2].

5 METHODOLOGY

This section outlines our methodology. First we will describe our lab setup, then we will cover which VPN configurations were analysed. In addition, we will describe the four different experiments conducted in our research, these experiments measuring goodput, latency, initiation time and CPU utilization respectively.

5.1 Lab setup

To perform our experiments, a test environment was created where all network link speeds were 1 Gbit/s. This environment can be seen in Figure 1. The two servers were connected to each other through a HP 6600-48G-4XG Layer 3 switch, with a link speed of 1 Gbit/s, and a MTU of 1500 bytes. A list of the hardware specifications of the servers can be found in Appendix A.

Notable features that were enabled were Intel Advanced Encryption Standard New Instructions (AES-NI), Generic Segmentation Offload (GSO), TCP Segmentation Offload (TSO) and Intel Turbo boost. Kernel version 5.6.17 was installed on the servers, since this was the latest stable available kernel version at the time of writing, which included WireGuard-C. On each of the servers, the different VPN implementations were configured. These being strongSwan v5.8.4,

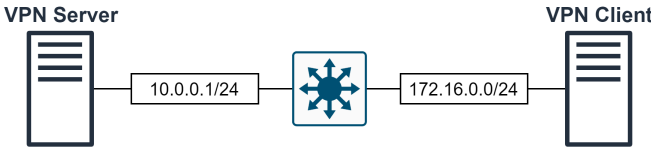

Figure 1: The lab setup used in this paper. A VPN server and client were configured on two different servers, connected to each other through a network switch.

Table 2: Encryption and integrity algorithms that were measured in this research.

VPN Solution	Encryption	Integrity
strongSwan	AES-128-CBC	SHA256
	AES-128-GCM	GHASH
	AES-256-GCM	GHASH
	ChaCha20	Poly1305
OpenVPN	AES-128-CBC	SHA256
	AES-128-GCM	GHASH
	AES-256-CBC	SHA256
	AES-256-GCM	GHASH
WireGuard-C	ChaCha20	Poly1305
WireGuard-Go	ChaCha20	Poly1305

OpenVPN v2.4.9, WireGuard-C v1.0.20200513 and WireGuard-Go v0.0.20200320, which were the latest stable versions at the time of writing.

5.2 VPN configurations

Both OpenVPN and strongSwan support many cipher suites, authentication mechanisms, key lengths, and integrity algorithms, from now on referred to as *VPN parameters*. Since many of these VPN parameters are outdated or experimental, we only experimented with the recommended VPN parameters. StrongSwan solely provides a list of recommended VPN cipher suites [20], with OpenVPN only recommending to use GCM or CBC modes [25]. This only allowed us to use Camellia, ARIA and AES. Out of these cipher suites, only AES is approved by NIST [29]. Additionally, since we expect AES to be more commonly used, and the fact that features such as the performance improving AES-NI were created, only the performance of AES was investigated for OpenVPN. AES is also recommended by strongSwan, and therefore we deem it more fair to compare the AES cipher for OpenVPN. OpenVPN was tested in UDP tunnel mode, since earlier research showed that it is better performing [10, 14].

Both WireGuard implementations only support one set of VPN parameters, which is ChaCha20Poly1305. This VPN parameter set is also supported by strongSwan, thus we will also research it for strongSwan. OpenVPN does not support ChaCha20Poly1305 at the time of writing. The complete list of VPN parameters explored in this research can be seen in Table 2.

5.3 Experiments

In our research, three main experiments were performed. Namely, goodput, latency, and initiation time. In addition, whilst performing the goodput measurements the CPU utilization was also measured. The experiments are described in more detail below.

5.3.1 Goodput. For the goodput experiments, we based our methodology on RFC 2544 [30]. This means that we did goodput measurements for two transport protocols, these being UDP and TCP (CUBIC), for several different packet sizes. The packet sizes being 64, 512, and 1024 bytes. In addition, we also did measurements with the maximum packet size, which is different for each VPN solution, due to their varying overhead. To discover the maximum packet size, we did preliminary experiments to discover when the packets were not fragmented. These maximum packet sizes without fragmentation are shown in Table 3 and are based on the Ethernet MTU of 1500 bytes.

Table 3: Maximum payload sizes for each VPN parameter set based on a maximum MTU of 1500.

VPN Solution	Encryption	UDP payload	TCP payload
strongSwan	AES-CBC	1410	1386
strongSwan	AES-GCM	1418	1394
strongSwan	ChaCha20	1418	1394
OpenVPN	AES-CBC	1375	1351
OpenVPN	AES-GCM	1420	1396
WireGuard	ChaCha20	1392	1368
Baseline	ChaCha20	1472	1448

It is important to note that by packet sizes we mean the size of an IP packet including the payload and its transport protocol. Thus, in the case of a 64-byte packet and using UDP as transport protocol, it exists of a 20 bytes IP header, 8 bytes UDP header and 36 bytes of payload. For TCP this is also 20 bytes for the IP header, 32 bytes for the TCP header and 12 bytes for the payload. It should be pointed out that iPerf generated a TCP header of 32 bytes during this research.

IPsec adds 34 bytes of ESP overhead when using a GCM or ChaCha20Poly1305. As IPsec also encapsulates the original IP and UDP/TCP header this adds an additional 28 or 52 bytes. Another IP header of 20 bytes is added. The aforementioned amount of bytes with the payloads of 1418 and 1394 sums up to 1500 bytes for UDP and TCP respectively. For CBC mode the same principles apply. However, the payload should always be a multiple of 16 bytes and is otherwise padded.

For OpenVPN it is more complicated. The OpenVPN header is 24 bytes when GCM mode is used [31]. In the case of encapsulating a TCP segment, this means that we should deduct the UDP (8), OpenVPN (24), IP (20), TCP (32) and another IP (20) header. This equates to 104 bytes, leaving 1396 bytes for the payload. With UDP 1420 bytes of payload can be sent, since its has 24 bytes less overhead when compared to TCP. In CBC mode the OpenVPN header is a bit different. Another difference is that multiples of 16 bytes are encrypted, otherwise, padding is used to fill it up. In CBC mode the header consists of an HMAC, Initialization Vector (IV), packet ID, type and Peer ID [31]. In our case, this header is 57 bytes. For TCP, when adding up all other overhead (80 bytes) and the payload of 1351 bytes, this totals to 1486. Since multiples of 16 are used, and our maximum MTU is 1500 bytes, it cannot have 1502 bytes as its maximum payload but rather 1486 bytes. The same principles apply for encapsulated UDP traffic, except for 24 bytes less overhead. Important to note is that by default OpenVPN has a Maximum Segment Size (MSS) of 1450 configured [32]. We changed this to the maximum value to allow it to send more bytes in one TCP segment.

Note the smaller payload sizes of WireGuard compared to IPsec. This is the result of the default MTU size of 1420 bytes that is configured on the WireGuard interface. Minus the UDP and IP header, this leaves 1392 bytes for the payload for UDP and 1368

for TCP. We experimented with increasing the MTU size to 1440, since the WireGuard header is 16 bytes and each packet also includes a 16-byte authentication tag [33], combined with the IP and UDP header this equates to 60 bytes. This should allow more room for additional payload. However, this yielded no performance improvements. Therefore, we decided to use the default 1420 MTU instead.

All VPN parameter sets that were analysed are listed in Table 2. The experiments were performed with iPerf v2.0.10, which was used for the UDP and TCP traffic generation. We also performed baseline goodput measurements to discover the maximum link goodput. After this, we measured the UDP and TCP goodput for all VPN implementations. All measurements had a duration of 60 seconds, as is recommended by RFC 2544 [30], and were repeated one hundred times.

Important to note is that with the 64-byte packet sizes experiment, iPerf was initiated with two parallel threads. This means that two iPerf threads were used to generate packets. When we used iPerf with one thread in this scenario, we observed it became the bottleneck.

We did not experiment with jumbo frames, as we think most VPN connections will be created over the internet, and therefore do not deem it likely that jumbo frames can be utilized in most scenarios.

5.3.2 Latency. The latency tests were performed by the ping utility for each implementation. In addition, this experiment was also conducted without the use of a VPN implementation in order to obtain a baseline measurement. In total, a million ICMP requests were sent with an interval of a thousand per second. This was done for every VPN parameter set as listed in Table 2. The ping output was in turn saved to a text file for further analysis. Through the ping utility, the round trip time (RTT) between two endpoints was obtained.

5.3.3 Connection Initiation time. Each VPN implementation allows for logging, and from these logs, the status of each connection can be obtained. By measuring the time between the initiation and the moment the logs mention the initiation to be complete, the initiation time can be extracted. In addition, the packets required for the setup were collected with tcpdump. With this information, the difference between timestamps between the first and last packet needed to set up a connection was used to calculate the initiation time in milliseconds.

It should be noted that interactions with this log file are not instantaneous, as it not only takes time to write the message into the logs, but reading it also expends time. In order to improve the accuracy of the initiation times obtained, a test experiment was conducted. An event was inserted into the log file, and read by the script, while measuring the time elapsed. By extracting the mean time needed to interact with the log files from the obtained initiation times, more accurate results were obtained.

Two different metrics were obtained. First, the total initiation time, including both the handshake and the preprocessing. Secondly, the time it takes between the two handshakes. This experiment was repeated one thousand times for each of the four VPN implementations, with a break of five seconds after starting or stopping the VPN connection. This prevents a potential strain on resources from interfering with the results. For this experiment, we considered WireGuard-C and WireGuard-Go to be two different implementations.

For strongSwan, IKEv2 was used. Ed25519 elliptic curve Diffie–Hellman (ECDH) was configured for the key exchange with SHA256 as its integrity algorithm. WireGuard only has one option for the key exchange, which is also Ed25519 in combination with BLAKE2 for the integrity. OpenVPN relies on TLS for its key exchange. We configured OpenVPN to use TLS version 1.3 with Ed25519 as key exchange algorithm and SHA256 as its integrity algorithm.

As the WireGuard implementations do not make use of certificate sharing, and instead make use of a pre-shared key, we deemed this comparison to be unfair. For this reason, secondary configurations for both OpenVPN and strongSwan were tested, with both

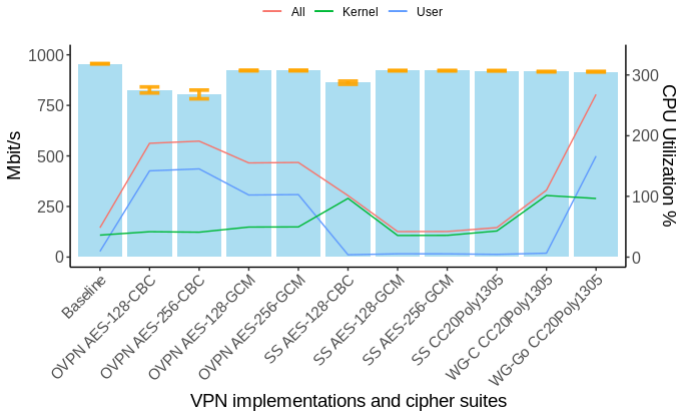

Figure 2: The goodput and CPU utilization results for UDP with the maximum packet size. The bars indicate the goodput while the lines showcase the CPU utilization.

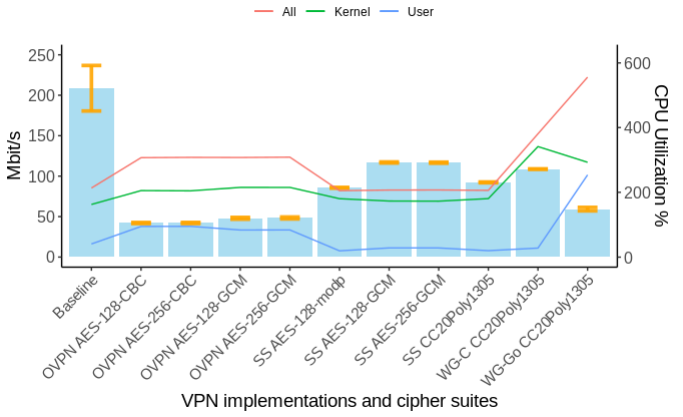

Figure 3: The goodput and CPU utilization results for UDP with a packet size of 64 bytes. The bars indicate the goodput while the lines showcase the CPU utilization.

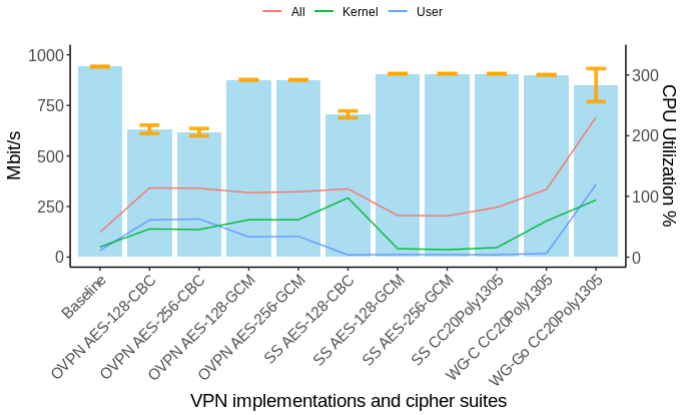

Figure 4: The goodput and CPU utilization results for TCP with the maximum packet size. The bars indicate the goodput while the lines showcase the CPU utilization.

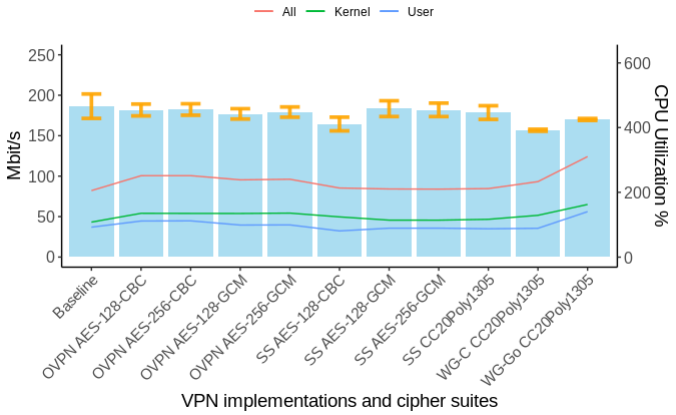

Figure 5: The goodput and CPU utilization results for TCP with a packet size of 64 bytes. The bars indicate the goodput while the lines showcase the CPU utilization.

implementations making use of pre-shared keys as well. This allows us to not only compare WireGuard fairly to OpenVPN and strongSwan, but also allows us to compare the connection initiation times of these different OpenVPN and strongSwan configurations. It is important to note that OpenVPN configured with a pre-shared key does not perform a handshake to initiate the connection. The certificate for strongSwan was based on Ed25519. OpenVPN does not support Ed25519 for its certificate authentication. Instead, we chose another elliptic curve algorithm that is also NIST approved, namely secp384r1 [34].

5.3.4 CPU utilization. Whilst measuring the goodput, the CPU utilization was measured with the tool mpstat v11.6.1. This allowed us to measure kernel- and user-level CPU utilization.

6 RESULTS

In this section, the results of our experiments are described and illustrated. For the different VPN implementations, the goodput, CPU utilization, latency and initiation time results are shown.

6.1 Goodput

In the following graphs, the goodput in Mbit/s as well as the CPU utilization percentage can be seen for each investigated packet size for both UDP and TCP. Each graph showcases the results for all investigated VPN implementations and their tested encryption and integrity algorithms. The implementations are abbreviated in the graphs as OpenVPN (OVPN), strongSwan (SS), and WireGuard (WG) while the ChaCha20Poly1305 cipher suite is abbreviated as CC20Poly1305. In this section, only the results for the maximum and 64-byte packet sizes are shown. Results for packet sizes of 512 and 1024 bytes were obtained, but due to the lack of insight that could be gained from them, they can be found in Appendix B instead.

The goodput and CPU utilization results for UDP traffic with a maximum packet size can be seen in Figure 2. The highest goodput was obtained by the baseline, reaching a mean of 956 Mbit/s. The AES-GCM ciphers of OpenVPN perform similar to the three best performing strongSwan ciphers, all ranging from 921 to 922 Mbit/s. Both versions of WireGuard are slightly behind at 916 Mbit/s each. While OpenVPN's AES-256-CBC cipher obtains the lowest goodput, it still managed to obtain a goodput of 824 Mbit/s. The standard deviation values for OpenVPN and strongSwan are all lower than 0.4 Mbit/s, which is smaller than the difference between that of OpenVPN and strongSwan. In regards to CPU utilization, both WireGuard-Go and OpenVPN require more resources than strongSwan and WireGuard-C. The different OpenVPN ciphers range from 155 to 206% utilization, whereas WireGuard-Go utilizes 268%. It is interesting to note that the strongSwan's AES-GCM ciphers have a lower CPU utilization rate than the baseline. Their utilization rates are around 42%, with that of the baseline being 48%. The utilization rate is measured as the amount of utilization on each core of the processor, from which the total is combined into one metric. As our client-side server was equipped with a quad-core processor, each core having two threads, the maximum utilization rate possible would be 800%. Besides the sum of the CPU utilization, the kernel and user-space utilization can also be seen. OpenVPN and WireGuard-Go, the implementations not implemented in the kernel, have a higher user-space utilization compared to the kernel-based implementations. The value of *All* does not simply contain the kernel- and user-space utilization percentages, but also a handful of other utilization factors such as hardware interrupts, which were excluded from this graph.

Next up, we have the goodput and CPU utilization results for UDP with the packet size of 64 bytes, as shown in Figure 3. The baseline greatly outperforms the VPN implementations in this context, with it reaching a mean of 209 Mbit/s. The next best results are obtained by strongSwan's AES-GCM ciphers, which both obtained a mean goodput of 116 Mbit/s. OpenVPN performs the worst out of all implementations, its best performing algorithm being AES-256-GCM. This cipher suite only manages to obtain a mean goodput value of 48 Mbit/s. In regards to CPU utilization, OpenVPN

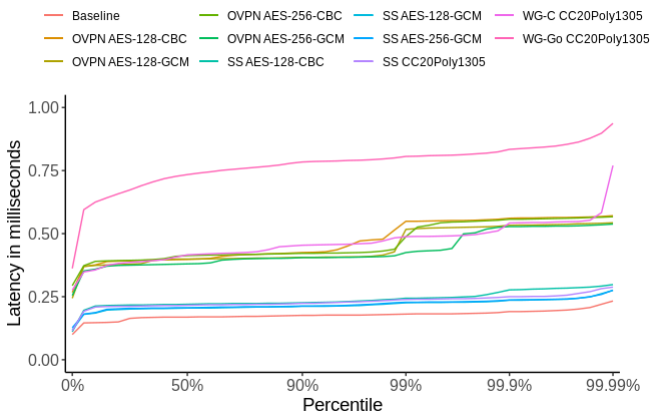

Figure 6: A percentile graph of the resulting latencies for the different VPN implementations. The latencies listed here consist of the RTT between the VPN client and server.

Table 4: The initiation time for the different VPN implementations. The results for each implementation are split into the total amount of time elapsed as well as the time between the first and last message of the handshake. Besides the mean, the 50 and 99 percentile values are also shown.

VPN implementation	Mean	50%	99%
OpenVPN certificates (Total)	1153.7	1151.8	1285.5
OpenVPN certificates (Handshake)	1144.9	1144.9	1279.1
OpenVPN pre-shared key (Total)	954.9	954.3	968.4
OpenVPN pre-shared key (Handshake)	—	—	—
strongSwan certificates (Total)	33.6	33.7	35.5
strongSwan certificates(Handshake)	4.6	4.6	5.1
strongSwan pre-shared key (Total)	31.8	31.9	34.2
strongSwan pre-shared key (Handshake)	3.4	3.4	3.9
WireGuard-C (Total)	6.9	7.8	8.0
WireGuard-C (Handshake)	0.7	0.7	1.1
WireGuard-Go (Total)	10.6	10.6	10.9
WireGuard-Go (Handshake)	1.0	1.0	1.1

and strongSwan remain consistent between cipher suites. OpenVPN achieves a mean utilization rate of 308%, whereas strongSwan achieves 206%. WireGuard-Go managed to obtain a goodput of 59 Mbit/s, but in return had a utilization rate of 556%.

In Figure 4 the results can be seen for the maximum packet size when transmitting TCP traffic. The baseline outperforms the VPN implementations with a goodput of 941 Mbit/s. Out of the VPN implementations, strongSwan is the best-performing, with its AES-GCM and ChaCha20Poly1305 cipher suites all obtaining goodput results rounded down to 906 Mbit/s. WireGuard-C follows behind with a goodput of 901 Mbit/s. OpenVPN's CBC mode leads to the lowest goodput results, with AES-CBC-128 and 256 obtaining mean goodput values of 528 and 510 Mbit/s respectively. Regarding CPU utilization, OpenVPN reaches consistent mean utilization values of around 103%, whereas the three best performing VPNs in regards to goodput here all have low mean utilization values, utilizing around 70-80% of the CPU. WireGuard-Go obtains the highest CPU utilization rate, utilizing 230% of the CPU.

Lastly, the results for TCP with a packet size of 64 bytes can be seen in Figure 5. The results here are similar between different implementations, with both goodput and CPU utilization not differing as much as in other graphs. The baseline reaches a mean goodput of 186 Mbit/s. All four OpenVPN ciphers follow suit with mean goodput values between 176 and 182 Mbit/s. StrongSwan's AES-GCM and ChaCha20Poly1305 ciphers have goodput results ranging between 178 and 183 Mbit/s. WireGuard-C falls behind here, with a goodput rate of 156 Mbit/s. Regarding utilization, the baseline sits at 205%, which is similar to strongSwan's 210%. OpenVPN is a bit more CPU intensive with mean utilization values ranging between 238% and 252%. WireGuard-Go requires once again the most resources with 311% utilization, from which it gains a goodput of 170 Mbit/s.

6.2 Latency

The results of the latency experiment can be seen in Figure 6. In this percentile graph, the median can be seen, as well as the latencies at higher percentile values. The graph showcases that for most strongSwan data-points obtained, the latency was consistent, as indicated by the nearly straight line. For the different strongSwan cipher suites tested, both AES-GCM cipher suites had a median latency of 0.21 milliseconds, with ChaCha20Poly1305 and CBC both having latencies of 0.22. Higher latency values were obtained by OpenVPN. OpenVPN's best-performing cipher suite is AES-256-GCM, with a median latency of 0.38 milliseconds. AES-128-GCM, AES-128-CBC and AES-256-CBC obtained median latencies of 0.40, 0.40 and 0.41 respectively. WireGuard-C performs similarly to the CBC ciphers, with a median latency of 0.42 milliseconds. A median of 0.73 milliseconds was obtained by WireGuard-Go.

When observing the results themselves, it can be seen that the latencies for all implementations except for WireGuard-Go remain stable 99% of the time. While there certainly are a few lower values at first, and a slow rise in latencies can be seen, the latency values at 99% do not differ too greatly from those at the 50% mark. After the 99% mark, while strongSwan's results remain stable until the very end with only a very minor increase, a larger increase can be seen in the other implementations. The latencies for OpenVPN and WireGuard-C clearly start to increase from the 99% mark onward, showcasing that while they remain stable most of the time, in a few exceptional cases a larger latency can be seen. WireGuard-Go already begins with a higher latency value, which steadily increases over time. Both WireGuard-C and WireGuard-Go suffer from a peak in latencies at the 99.99% mark, indicating that in a few cases, the latency becomes even worse. From these results, it can be seen that the most consistent latencies are obtained by strongSwan, with WireGuard-C and especially WireGuard-Go being less consistent.

6.3 Initiation time

The connection initiation times for the different VPN implementations can be seen in Table 4, with each VPN implementation separated into two rows, named *Total* and *Handshake*. The row

indicating the total shows the elapsed time in ms from the initiation of the VPN until a initiation complete message could be seen in the log. Whereas the row indicating the handshake shows the time elapsed by performing the handshake, measured by taking the elapsed time between the first and last packet sent for the initiation.

It can be seen that both of the initiation tests for OpenVPN require more time to initiate a connection than the other VPN implementations. Even without the use of certificates, the initiation time is nearly thirty times higher compared to strongSwan. While lower initiation time values were obtained by both strongSwan experiments, it is outperformed by each of the WireGuard implementations. For all implementations except for OpenVPN, only a small amount of time was spent on the handshake itself. OpenVPN configured with a pre-shared key does not perform a handshake, and therefore no handshake time could be obtained.

The results for the total initiation time were tweaked in order to accommodate for the interactions with the logging file itself. The time required to write the initialization complete message to the log as well as the time taken by the script to read this message has been subtracted from the total results. The mean time for this action to be performed was 1.9 milliseconds.

7 DISCUSSION

In this section our results are discussed. We will first discuss the goodput, latency, CPU efficiency, and connection initiation time results respectively. In addition, we discuss how our results compare to earlier work. We conclude this section with the limitations this research had.

7.1 Goodput

In general, the AES-GCM ciphers for OpenVPN and strongSwan performed well in terms of goodput, with strongSwan's AES-GCM ciphers being among the best performing cipher suites in every goodput experiment. These results can be contributed to two different causes. Firstly, in our test environment the feature AES-NI was enabled. This feature was designed to improve the performance of AES encryption and would, therefore, have a positive effect on our results. Secondly, GCM mode has an advantage over CBC mode in the fact that it has integrity built-in. For the CBC ciphers, an additional SHA256 hash was performed to provide integrity, which requires additional computations and decreases its goodput.

Pudelko claimed in his paper that disabling the SHA256 hashing, which is required for CBC based ciphers in order to provide integrity, reduces the computing cost by around 40%. The reason for this is that SHA256 functions as a second pass of the data with its hashing not happening simultaneously with the encryption process [9]. GCM mode has integrity built-in and can, therefore, provide for integrity in a single pass, which leads to an increase in performance. This allows it to outperform its CBC counterpart.

For both UDP and TCP, regardless of the packet size, WireGuard-C was never the implementation with the highest goodput. While its performance was certainly not lacking, it could never match the mean goodput values observed for strongSwan's AES-GCM ciphers. In the context of maximum packet size, this is partly explained due to the differences in payload. As WireGuard-C suffers from the largest amount of overhead, it is unable to send payloads as large as both strongSwan and OpenVPN. The gap between WireGuard-C and strongSwan's AES-GCM cipher suites for maximum packet size would not have been nearly as large were the payloads to be equal, causing the overhead of WireGuard-C to be a bottleneck at these packet sizes.

When comparing the maximum packet size goodput results for TCP with those of UDP, one can observe that OpenVPN and WireGuard-Go perform slightly worse in a TCP environment as compared to strongSwan. A probable explanation for this is related to packet loss. By analyzing our iPerf output files for the UDP experiments, we observe that strongSwan and WireGuard-C have a very low amount of packet loss. This is not the case for OpenVPN and WireGuard-Go. While for the first two implementations the packet loss can be rounded down to 0%, the latter two suffer from at least a 10% packet loss rate in every individual iPerf test. While the packet loss rates for TCP are unavailable to us, it is likely that these protocols suffer from a higher packet loss rate for TCP as well. While the amount of packet loss is irrelevant for UDP, this is not the case for TCP. Lost packets will cause TCP CUBIC to enter the fast recovery state, at which point the congestion window is temporarily decreased [35]. This decrease happening more often for OpenVPN and WireGuard-Go might explain why lower goodput rates are observed.

When considering smaller packet sizes, a clear contrast between UDP and TCP can be seen. Whereas for UDP a very low mean goodput was obtained when using a small packet size, this is not the case with TCP. Where the VPN implementations perform almost as well as the baseline for TCP, the baseline is able to obtain almost double the goodput for UDP compared to the best of the VPN implementations. This is caused by the fact that TCP makes use of Nagle's algorithm. This algorithm allows for the aggregation of TCP packets in order to limit the number of small packets being transmitted over the wire [36]. While every single 64-byte packet is transmitted for UDP, for TCP the use of Nagle's algorithm allows it to instead aggregate these into bigger packets, which reduces the strain on the resources. In this case, the kernel has to perform fewer header related functions, such as route look-ups. In addition, more goodput can be sent over the link, as fewer headers are required. This allows TCP to outperform UDP in this scenario.

It is interesting to note that for TCP with packets of 64 bytes, WireGuard-C is being outperformed by WireGuard-Go. Prior work by Pudelko et al. stated that bottlenecks caused by locking system resources hindered WireGuard-Go when a large amount of packets is being sent, as is the case with 64 bytes packets [9]. While this explains why both WireGuard-C and WireGuard-Go do not perform nearly as well compared to strongSwan at smaller packet sizes, it does not explain why WireGuard-Go outperforms WireGuard-C in terms of goodput. As we are unable to find a logical explanation for this observation, future research is required to identify as to why this is the case.

When considering the CPU utilization, it can be observed that in general the kernel-space based implementations strongSwan and WireGuard-C have a lower user-space utilization compared to OpenVPN and WireGuard-Go. It can also be seen that while the user-space utilization for the kernel-based implementations is rather low during the maximum packet size experiments, it is a lot higher during the 64-byte packet experiments. This is caused

by the tool iPerf, which is used to generate packets. Due to the larger amount of packets needed to be generated by having a small packet size, more resources are needed by iPerf. Since iPerf runs in user-space, the user-space utilization is therefore much higher [37].

In our lab setup, a singular VPN tunnel was used for every experiment. As this tunnel was instantiated on a 1 Gbit/s link, the CPUs of our servers were more than able to almost completely utilize this link. Were these experiments to be repeated in a 10 Gbit/s environment, with multiple concurrent VPN tunnels, WireGuard-C would likely see improved goodput results. Out of all the VPN implementations, WireGuard is the only implementation to fully support multithreading. While strongSwan supports multithreading, the underlying IPsec Linux kernel stack does not, which would lead to bottlenecks when multithreading would be utilized for strongSwan. While strongSwan is able to achieve a high goodput in our experiments, it does so with a single core of the CPU. When multiple cores could be utilized, which would be the case when a worse CPU would be used to fill a 1 Gbit/s link or when a 10 Gbit/s link were to used established, WireGuard-C could likely outperform strongSwan. This was also shown by Osswald et al [12].

7.2 Latency

The lackluster latency for WireGuard-C is an interesting finding. Due to both WireGuard-C and strongSwan being implemented in the kernel, a similar latency between the two implementations was expected. OpenVPN needs to perform additional computations in order to manage a VPN connection in the user-space, yet obtains similar latency results as WireGuard-C. However, WireGuard-C does outperform its user-space counterpart WireGuard-Go, the difference of which could be attributed due to WireGuard-C operating in kernel-space. As WireGuard-C has a higher CPU utilization rate than strongSwan while reaching almost the the same goodput rate, one can imply that more CPU cycles are needed per packet. If this were to be the case, it could explain the observed differences in latencies for the two different implementations, as these additional cycles could extend the latency. AES-NI could be a factor that lowers the latency values of strongSwan, as it optimizes the amount of AES related CPU instructions and thus lowering CPU cycles per packet. However, the ChaCha20Poly1305 cipher of strongSwan achieved better latency results than its WireGuard-C counterpart. Therefore, we do not expect AES-NI to be the sole reason for these differences.

7.3 Connection Initiation Time

On the topic of connection initiation time, both WireGuard implementations clearly outperform the other two VPN implementations. It is especially interesting to note that WireGuard-Go, while being outperformed by WireGuard-C, is performing much better than strongSwan. This does match the claims that the WireGuard protocol ensures fast connection establishment [2]. A probable reason for this is the fact that WireGuard handshake requires less messages and makes use of the Blake2 algorithm, which Aumasson et al. shows to outperform SHA256 [38].

The WireGuard protocol only supports initiating a connection through the use of pre-shared keys, unlike OpenVPN and strongSwan. Pre-shared keys are easier to use, but according to IBM this comes at the cost of security [39]. The disadvantage of using pre-shared keys is the lack of PFS, which may be desirable. For the WireGuard protocol, additional measures have been taken to provide for PFS [2]. Since both options for OpenVPN and strongSwan have their advantages and disadvantages, both were analysed. For both OpenVPN and strongSwan, using pre-shared keys instead of certificates had a positive influence on the initiation time, indicating that besides being easier to use, a slight speedup could also be gained from implementing pre-shared keys.

While very low initiation times were obtained from both the WireGuard implementations and strongSwan, the same cannot be said about OpenVPN. Even while using the faster to initiate pre-shared key configuration, the resulting initiation time still had a mean of nearly a second, more than thirty times as large as its strongSwan counterpart. Through analysing the packets transmitted during the handshake for both certificate-based and pre-shared key based OpenVPN, a long gap in between packets can be seen. As this gap is nearly one second in length, most of the initiation time is spent here. The gap is slightly longer when certificates are used, which could be explained due to the fact that the certificates are being approved during this time frame. It can be concluded that OpenVPN's initiation is not as efficient as those of the other VPN implementations.

7.4 CPU efficiency

The last of the results to be discussed is the CPU efficiency. When observing each of the goodput and CPU utilization graphs, it immediately becomes clear that WireGuard-Go is always the implementation that utilizes the most of the CPU. This happens for two different reasons. First, WireGuard-Go takes place in user-space, and in order to transmit traffic over a VPN implementation from the user-space, additional computations are required in order to perform the kernel processes required. For this same reason, OpenVPN also suffers from a high CPU utilization. The other reason is the fact that WireGuard-Go is written in a different language than the other implementations, namely Go. Compared to Go, C is a lower-level programming language, which makes it more efficient at the cost of programming complexity. For this reason, WireGuard-Go requires even more CPU resources than OpenVPN, as it requires more processing time as well as computing power to obtain the same mean goodput values [40].

As for the CPU utilization of the other implementations, one can clearly see the limits as to what amount of CPU resources can be utilized by OpenVPN and strongSwan. For the more expensive to compute packet size of 64 bytes, higher utilization percentages can be observed. For UDP, one can clearly see that similar utilization percentages are obtained between different cipher suites of OpenVPN and strongSwan, something which was not as apparent for the maximum packet size.

7.5 Comparison with existing papers

In the paper by Lackorzynski et al., they concluded that the WireGuard protocol was the best-suited protocol for use in a factory environment due to it obtaining the highest throughput values [10]. While their findings differ from ours, it can be attributed to the fact that their research involved 10 Gbit/s links. Since no mention of

AES-NI was made in the paper, these reasons might explain why their results differ from ours. The latter was also the case for the paper by Mackey et al. as no mention was made of AES-NI either [11].

The paper by Osswald et al. clearly showcases the difference when using AES-NI compared to not using it, with the same ciphers obtaining more than thrice as low throughput value when AES-NI is disabled [12]. Therefore, it is to little surprise that the differences between VPN implementations were not nearly as significant as in the papers of Lackorzynski et al. and Mackey et al., as WireGuard is not outperforming its competition nearly as much when AES-NI is enabled. In addition, none of these papers consider the GCM mode ciphers. In this research these ciphers are the best performing ciphers, and it is therefore not too odd that WireGuard, which in prior work has always seemed to be better performing, is being outperformed.

7.6 Limitations

This research had several limitations. Firstly, iPerf was used in this research. While being used in earlier work that also specifically researched VPN implementations [11, 41, 42, 43], iPerf is kernel driver based and there are faster alternatives based on the Data Plane Development Kit (DPDK) [44]. Examples of these are MoonGen, pktgen and WARP. Pudelko also mentions that MoonGen is better performing than iPerf [9]. However, all these aforementioned tools require DPDK support. In our case, the NIC driver did not support DPDK which is why we could not use it in our environment [45].

Secondly, we only looked at the client-side for the CPU utilization. While we do not expect much difference between the server and client in this regard, it might be interesting to look at the server-side CPU utilization.

Thirdly, we did not specifically optimize our physical servers for multithreaded packet forwarding. Instead, we relied on the kernel for this aspect. Kernel optimizations like Receive Side Scaling (RSS), Receive Packet Steering (RPS), Receive Flow Steering (RFS), and Transmit Packet Steering (TPS) could improve parallelism and performance for multiprocessor systems [46]. We looked at these options and did preliminary experiments with it, although it did not result in any better results. However, we are unsure if this was due to a configuration error or that these were legitimate results, as we would expect better results. Thus, we think this should receive more attention in further research.

8 CONCLUSION

The goal of this research was to measure the performance differences between the VPN implementations strongSwan, OpenVPN, WireGuard kernel implementation (WireGuard-C), and the WireGuard Go implementation (WireGuard-Go) in a 1 Gbit/s environment. In order to measure this we created a lab setup in which we connected two physical servers to each other through a 1 Gbit/s switch. Here we configured OpenVPN and strongSwan with the AES-CBC and AES-GCM mode cipher suites. In addition, we also configured strongSwan with a ChaCha20Poly1305 cipher suite. For WireGuard-C and WireGuard-Go ChaCha20Poly1305 was configured as well, due to it being the only available cipher suite. In this environment, we measured the difference in UDP and TCP goodput, latency, connection initiation time, and CPU utilization. The goodput measurements were conducted with iPerf, where we performed measurements with the packet sizes of 64, 512, 1024 bytes, and the maximum packet size. The latency experiments were done by sending one million ICMP echo requests through the VPN tunnels with an interval of a thousand per second. The connection initiation time was measured with a Python script, while the CPU utilization was measured with mpstat.

In our environment, when using maximum packet sizes, OpenVPN and strongSwan perform the best in terms of UDP goodput. The mean value of the OpenVPN AES-GCM ciphers were 922 Mbit/s, while strongSwan's AES-GCM implementations had a mean of 921 Mbit/s. As the standard deviation may allow their positions to change, we conclude that these implementations with these cipher suites perform equally well in this scenario. Next up is WireGuard-C, which achieved a mean goodput of 917 Mbit/s, with WireGuard-Go following with 916 Mbit/s. The AES-CBC ciphers perform worse than their AES-GCM counterparts on every occasion except for TCP with a packet size of 64 bytes, where they perform equally well.

When a packet size of 64 bytes is used with UDP being its transport layer protocol, strongSwan with an AES-GCM cipher performs the best with 117 Mbit/s. After this follows WireGuard-C with 109 Mbit/s. Hereafter comes strongSwan with its ChaCha20Poly1305 cipher suite and finally WireGuard-Go and all OpenVPN configurations.

When TCP with maximum packet sizes are used, strongSwan with an AES-GCM cipher achieves the highest goodput with 906 Mbit/s. Hereafter WireGuard-C achieves the highest goodput with a mean of 901 Mbit/s. After this comes WireGuard-Go. OpenVPN in AES-CBC mode achieved the least TCP goodput.

The AES-GCM cipher suites of OpenVPN and strongSwan achieve the highest goodput with TCP when iPerf is generating packet sizes of 64 bytes, with a mean goodput of 179 and 178 Mbit/s respectively. The CBC modes of OpenVPN and the ChaCha20Poly1305 cipher suite of strongSwan both perform better than the WireGuard implementations. With WireGuard-Go performing at a goodput rate of 170 Mbit/s and WireGuard-C at 156 Mbit/s.

In terms of latency, all of the tested strongSwan ciphers perform better than OpenVPN and WireGuard, with median values of 0.22 ms for CBC and ChaCha20Poly1305 and 0.21 ms for the AES-GCM ciphers. For OpenVPN, both the AES-CBC (0.40 ms) and AES-GCM (0.39 ms) ciphers of OpenVPN show less latency than WireGuard-C (0.42 ms). WireGuard-Go achieved the highest latency with a median value 0.73 ms. At the 99th percentile mark, while the strongSwan ciphers have a similar latency to the median, all other implementations see an increase in latency values.

When looking at connection initiation time, WireGuard-C performs the best with a mean time to initiate a connection of 6.9 ms. Behind WireGuard-C comes WireGuard-Go with a mean of 10.6 ms. The initiation times for OpenVPN and strongSwan where analysed both with and without certificate authentication. With certificate authentication strongSwan has a mean of 33.6 ms, while without certificate authentication the mean is 31.8 ms. For OpenVPN this is 1152.7 and 954.9 ms respectively.

In terms of CPU efficiency, the AES-GCM ciphers of strongSwan show the least amount of CPU utilization for all TCP and UDP

measurements. WireGuard-C shows that it has a lower CPU utilization than OpenVPN in regards to the UDP and TCP maximum packet size experiments. The same applies for the experiment with TCP and sending 64-byte packets. In the experiment with UDP and sending 64-byte packets, OpenVPN shows less CPU utilization. However, it also transmits less than half of the packets that WireGuard-C sends. WireGuard-Go shows the highest CPU utilization in all experiments.

In our research, strongSwan with an AES-GCM cipher configured is the best performing VPN implementation in terms of goodput, latency, and CPU utilization. OpenVPN achieved an equal goodput rate compared to strongSwan when UDP with a maximum packet size was used. However, it does perform worse in the UDP 64-byte packet size experiment. WireGuard-C achieved the fastest connection initiation time and it also achieved better goodput results than WireGuard-Go in all but one experiment. In addition, WireGuard-Go had the worst latency and CPU utilization results.

9 FUTURE WORK

This work could be expanded upon in several ways. Firstly, as mentioned earlier in the limitations section, it is interesting to see whether RSS, RPS, RFS and TPS make a difference in the amount of packets that can be transmitted. As these technologies are made for multiprocessor systems, it is especially interesting to see if the goodput of WireGuard increases with the smaller packet sizes.

Secondly, we performed the measurements in a clean and reliable environment. However, it would be interesting to analyse how the VPN solutions perform when they are exposed to more realistic scenarios. One example of this would be to research how the implementations perform over a wireless medium or test outside of a lab environment. In addition, VPNs are not only available on servers or desktops, as there are mobile versions available as well. WireGuard, OpenVPN and strongSwan all have Android versions and it would be interesting to see how these compare in terms of performance.

Thirdly, we only performed measurements with a single tunnel between a server and client. There are scenarios were a central VPN server is used as a gateway and thus many connections to this server can be made. It would be interesting to see how the solutions compare when multiple concurrent connections are in use.

Fourthly, Pudelko and Osswald et al. have already conducted research on WireGuard in 10+ Gbit/s environments. However, as WireGuard recently has been integrated into the Linux kernel, it might be worthwhile to research this once again.

Lastly, more research into performance increasing IPsec options can also be performed. For instance, it might be worthwhile to look into ESP offloading, which can offload IPsec computations [47]. However, a lack of documentation on this topic can be observed. There are also libraries that can spread the crypto load of IPsec over multiple threads, such as pcrypt. However, documentation of Libreswan, an alternative to strongSwan, claims that this library is unstable [48]. In addition, the OpenVPN community is working on OpenVPN version 3. This version should include multithreading capabilities, yet at the time of writing only a beta version is available.

ACKNOWLEDGMENTS

This research was made possible by several individuals. First of all, we like to thank our supervisors Aristide Bouix and Mohammad Al Najar of KPMG for their guidance and creativity. Secondly, we like to thank the OS3 staff at the University of Amsterdam for lending us the required hardware and for advising us where needed.

REFERENCES

[1] Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. "A mechanised cryptographic proof of the WireGuard virtual private network protocol". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, pp. 231–246.

[2] Peter Wu. "Analysis of the WireGuard protocol". In: *Eindhoven University of Technology, Department of Mathematics and Computer Science* (2019).

[3] Karthikeyan Bhargavan and Gaëtan Leurent. "On the practical (in-) security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 456–467.

[4] Karthikeyan Bhargavan et al. "Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS". In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 98–113.

[5] Donenfeld, Jason A. *WireGuard*. 2020. URL: https://www.wireguard.com/.

[6] Salter, Jim. *WireGuard VPN makes it to 1.0.0—and into the next Linux kernel*. 2020. URL: https://arstechnica.com/gadgets/2020/03/wireguard-vpn-makes-it-to-1-0-0-and-into-the-next-linux-kernel/.

[7] Burduli, George. *WireGuard VPN protocol makes its way to Linux Kernel 5.6*. URL: https://www.xda-developers.com/wireguard-vpn-linux-kernel-5-6/.

[8] Larabel, Michael. *US Senator Recommends Open-Source WireGuard To NIST For Government VPN*. 2018. URL: https://www.phoronix.com/scan.php?page=news_item&px=WireGuard-Senator-Recommends%20.

[9] Maximilian Pudelko et al. "Performance Analysis of VPN Gateways". In: *Department of Informatics, Technical University of Munich* (2018).

[10] Tim Lackorzynski, Stefan Köpsell, and Thorsten Strufe. "A comparative study on virtual private networks for future industrial communication systems". In: *2019 15th IEEE International Workshop on Factory Communication Systems (WFCS)*. IEEE. 2019, pp. 1–8.

[11] 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*. 2020, pp. 162–164.

[12] Lukas Osswald, Marco Haeberle, and Michael Menth. "Performance Comparison of VPN Solutions". In: ().

[13] I Kotuliak, P Rybár, and P Trúchly. "Performance comparison of IPsec and TLS based VPN technologies". In: *2011 9th International Conference on Emerging eLearning Technologies and Applications (ICETA)*. IEEE. 2011, pp. 217–221.

[14] Irfaan Coonjah, Pierre Clarel Catherine, and KMS Soyjaudah. "Experimental performance comparison between TCP vs UDP tunnel using OpenVPN". In: *2015 International Conference on Computing, Communication and Security (ICCCS)*. IEEE. 2015, pp. 1–5.

[15] James Henry Carmouche. *IPsec virtual private network fundamentals*. Pearson Education India, 2007.

[16] Dale Liu et al. *Firewall policies and VPN configurations*. Elsevier, 2006.

[17] strongSwan. *Introduction to strongSwan*. 2019. URL: https://wiki.strongswan.org/projects/strongswan/wiki/IntroductionTostrongSwan.

[18] strongSwan. *strongSwan the OpenSource IPsec-based VPN Solution*. 2020. URL: https://www.strongswan.org/.

[19] Maximilian Pudelko et al. "Performance Analysis of VPN Gateways". In: *IFIP Networking 2020*. 2020.

[20] strongSwan. *Security Recommendations*. 2020. URL: https://wiki.strongswan.org/projects/strongswan/wiki/SecurityRecommendations.

[21] Jon C Snader. *VPNs Illustrated: Tunnels, VPNs, and IPsec: Tunnels, VPNs, and IPsec*. Addison-Wesley Professional, 2015.

[22] IETF. *IP Storage Security*. 2001. URL: https://www.ietf.org/proceedings/52/slides/ips-1/tsld005.htm.

[23] C Kaufman et al. *T. Kivinen," Internet Key Exchange Protocol Version 2 (IKEv2)*. Tech. rep. STD 79, RFC 7296, DOI 10.17487/RFC7296, October 2014,< http://www.rfc ..., 2014.

[24] Eric F Crist and Jan Just Keijser. *Mastering OpenVPN*. Packt Publishing Ltd, 2015.

[25] OpenVPN Inc. *GitHub - OpenVPN/openvpn at release/2.4*. 2020. URL: https://github.com/OpenVPN/openvpn/tree/release/2.4.

[26] Quarkslab. *OpenVPN 2.4.0 Security Assessment*. URL: https://ostif.org/wp-content/uploads/2017/05/OpenVPN1.2final.pdf.

[27] Tomas Novickis, Erik Poll, and Kadir Altan. "Protocol state fuzzing of an OpenVPN". PhD thesis. MS thesis, Fac. Sci. Master Kerckhoffs Comput. Secur., Radboud Univ …, 2016.
[28] Jim Salter. *WireGuard VPN review: A new type of VPN offers serious advantages*. URL: https://arstechnica.com/gadgets/2018/08/wireguard-vpn-review-fast-connections-amaze-but-windows-support-needs-to-happen/.
[29] NIST Computer Security Resource Center. *Block Cipher Techniques | CSRC*. 2020. URL: https://csrc.nist.gov/projects/block-cipher-techniques.
[30] Scott Bradner and Jim McQuaid. *RFC2544: Benchmarking Methodology for Network Interconnect Devices*. 1999.
[31] OpenVPN Inc. *Data Channel Crypto module*. URL: https://build.openvpn.net/doxygen/group__data__crypto.htmls.
[32] OpenVPN Inc. *https://openvpn.net/community-resources/reference-manual-for-openvpn-2-4/*. 2020. URL: https://openvpn.net/community-resources/reference-manual-for-openvpn-2-4/.
[33] Jason A Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel." In: *NDSS*. 2017.
[34] Sean Turner et al. "RFC 5480: Elliptic curve cryptography subject public key information". In: *Requests for Comments, Network Working Group, Tech. Rep* (2009).
[35] I Rhee et al. *R. Scheffenegger," CUBIC for Fast Long-Distance Networks*. Tech. rep. RFC 8312, DOI 10.17487/RFC8312, 2018.
[36] Greg Minshall et al. "Application performance pitfalls and TCP's Nagle algorithm". In: *ACM SIGMETRICS Performance Evaluation Review* 27.4 (2000), pp. 36–44.
[37] TNK. *iPerf testing considerations*. URL: https://www.kuncar.net/blog/2019/iperf-testing-considerations/.
[38] Jean-Philippe Aumasson et al. "BLAKE2: simpler, smaller, fast as MD5". In: *International Conference on Applied Cryptography and Network Security*. Springer. 2013, pp. 119–135.
[39] IBM. *IPsec pre-shared keys vs. certificates*. URL: https://www.ibm.com/support/pages/ipsec-pre-shared-keys-vs-certificates.
[40] Computer Language Benchmarks Game. *Go vs C gcc - Which programs are fastest?* URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/go-gcc.html.
[41] Thomas Berger. "Analysis of current VPN technologies". In: *First International Conference on Availability, Reliability and Security (ARES'06)*. IEEE. 2006, 8–pp.
[42] Berry Hoekstra, Damir Musulin, and Jan Just Keijser. "Comparing TCP performance of tunneled and non-tunneled traffic using OpenVPN". In: *Universiteit Van Amsterdam, System & Network Engineering, Amsterdam* (2011), pp. 2010–2011.
[43] Tim Lackorzynski et al. "Enabling and Optimizing MACsec for Industrial Environments". In: *2020 16th IEEE International Conference on Factory Communication Systems (WFCS)*. IEEE. 2020, pp. 1–4.
[44] Bram ter Borch. "Session based high bandwidth throughput testing". In: (2017).
[45] DPDK.org. *Supported Hardware*. URL: https://core.dpdk.org/supported/.
[46] Herbert, Tom and de Bruijn, Willem. *Scaling in the Linux Networking Stack*. URL: https://www.kernel.org/doc/Documentation/networking/scaling.txt.
[47] Shannon, Nelson. *XFRM device - offloading the IPsec computations*. URL: https://www.kernel.org/doc/Documentation/networking/xfrm_device.txt.
[48] libreswan.org. *Cryptographic Acceleration*. URL: https://libreswan.org/wiki/Cryptographic_Acceleration.

APPENDIX

A HARDWARE AND SOFTWARE USED

In Table 5, the hardware and software used in our lab setup are listed.

Table 5: The hardware and software used in our lab setup.

	VPN Client
Server hardware	Dell PowerEdge R230
Processor	Intel Xeon E3-1240L v5 @ 2.10GHz
Memory	16 GiB
NIC	NetXtreme BCM5720
SSD	Crucial CT240BX200SSD1
OS	Ubuntu 18.04
Kernel	5.6.17-050617-generic
	VPN Server
Server hardware	PowerEdge R240
Processor	Intel Xeon E-2124 CPU @ 3.30GHz
Memory	16GiB
NIC	NetXtreme BCM5720
SSD	Samsung 860
OS	Ubuntu 18.04
Kernel	5.6.17-050617-generic

B RESULTS FOR 512 AND 1024 BYTE PACKETS

In this appendix, the goodput and CPU utilization results for packet sizes of 512 and 1024 bytes can be found. They were excluded from the paper due to a lack of interesting results that could be concluded from them, but are still listed in this appendix for reference.

In Figure 7, the results for UDP traffic with a packet size of 512 are observable. Similar to most goodput experiments, the three well performing strongSwan ciphers, namely the AES-GCM ciphers and ChaCha20Poly1305, are the best performing with a goodput of 797 Mbit/s for former 796 for the latter. Closely behind are both WireGuard implementations, with WireGuard-C having a mean goodput of 793 Mbit/s and WireGuard-Go 792 Mbit/s. OpenVPN falls quite a bit behind, with its AES-128-GCM cipher having a goodput of 639 Mbit/s while its 256 bit counterpart has a goodput of 635 Mbit/s. OpenVPN's AES-128-CBC cipher is the worst performing cipher, with a goodput of 462 Mbit/s. The CPU utilization is for pair of implementation and cipher higher than that of maximum packet size, due to more packets needed to be generated and transmitted.

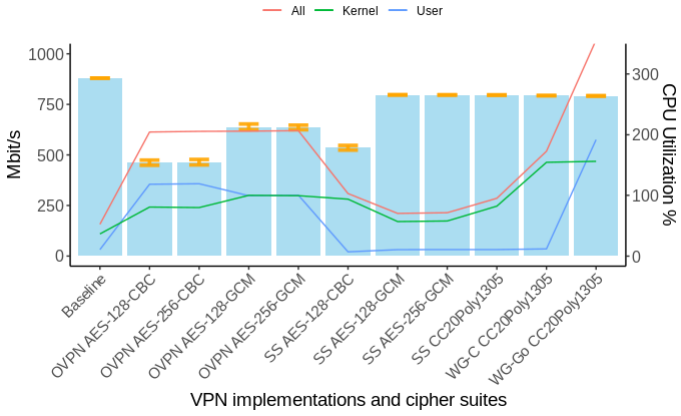

Figure 7: The goodput and CPU utilization results for UDP with a packet size of 512 bytes. The bars indicate the goodput while the lines showcase the CPU utilization.

The results of its TCP counterpart can be seen in Figure 8. The results have not seen much of a decrease compared to that of bigger packet sizes, which can be attributed due to the fact that TCP makes use of Nagle's algorithm, as explained in the Section 7.1. The best performing ciphers and algorithms are once again the three well-performing strongSwan ciphers, all having a goodput of 906 Mbit/s. WireGuard-C is closely outperformed, with a goodput of 899 Mbit/s. While OpenVPN's AES-GCM ciphers perform well with a goodput of 875 Mbit/s, it is outclassed by WireGuard-Go. WireGuard-Go has obtained a mean goodput of 892 Mbit/s, which is not that much lower than that of WireGuard-C.

The results for UDP traffic with a packet size of 1024 are similar to those of the maximum packet size, and can be seen in Figure 9. OpenVPN's AES-GCM ciphers and the three well performing strongSwan ciphers all perform about equally well int his scenario, all with means around 890 Mbit/s. Both WireGuard-C and WireGuard-Go

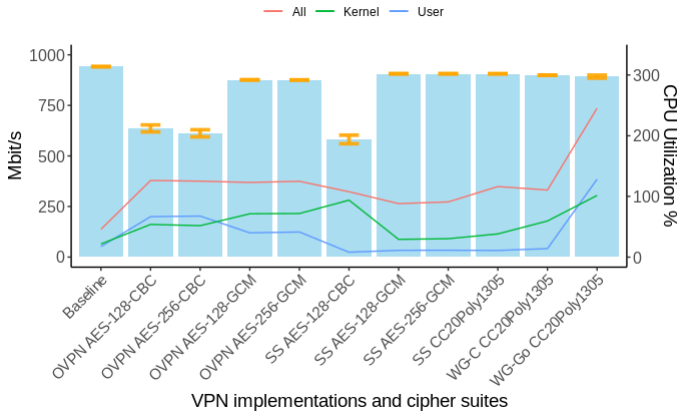

Figure 8: The goodput and CPU utilization results for TCP with a packet size of 512 bytes. The bars indicate the goodput while the lines showcase the CPU utilization.

are not far behind, with both having a goodput of 887 Mbit/s. The worst performing cipher suites, namely OpenVPN's AES-CBC ciphers as well as strongSwan's AES-CBC cipher, all perform worse than their maximum packet size counterpart, with the former two having a goodput decrease of more than 150 Mbit/s, and the latter having a decrease of nearly 100 Mbit/s. The other ciphers have not decreased more than 30 Mbit/s between the two experiments, making the contrast clearly visible.

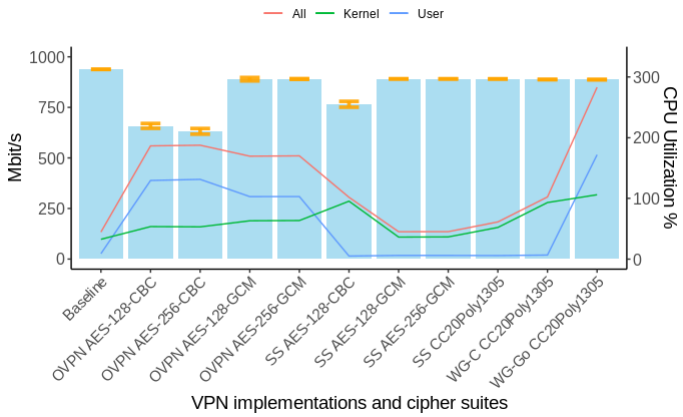

Figure 9: The goodput and CPU utilization results for UDP with a packet size of 1024 bytes. The bars indicate the goodput while the lines showcase the CPU utilization.

For the same packet size of 1024, the results of the TCP traffic can be seen in Figure 10. As with its UDP counterpart, the results are similar to that of the maximum packet size. Out of the different VPN implementations, the three well performing strongSwan ciphers again obtain the highest goodput values, all having a goodput of 906 Mbit/s. WireGuard-C follows closely behind with a goodput of 900 Mbit/s. In this scenario, OpenVPN's AES-GCM ciphers perform

differently from each other, with AES-128-GCM obtaining a goodput of 894 Mbit/s whereas its 256 bit counterpart obtained a mean goodput of 871 Mbit/s. For every implementation except OpenVPN, the CPU utilization rate has increased, with OpenVPN's utilization percentage having slightly decreased.

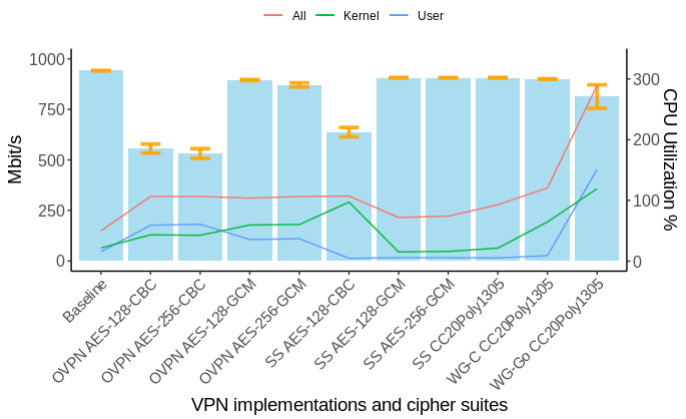

Figure 10: The goodput and CPU utilization results for TCP with a packet size of 1024 bytes. The bars indicate the goodput while the lines showcase the CPU utilization.