Université Libre de Bruxelles

ELEC-H504 - Network security

Exam paper - Wireguard
(cryptographic protocols)

DEJAEGHERE Jules, JANIAK Beata, RIVAS Chin, THYS Killian

Group ID: 10

June 2021

1 Introduction

WireGuard takes a new approach to the creation of encrypted tunnels for Linux. It tries to address the major drawbacks of IPSec and OpenVPN with a solution that is easy to configure, cryptographically secure and performant.

Unlike IPSec, where a strong layering is present with the key exchange separated from the transport encryption, all the WireGuard operations are performed in the layer 3. Instead of the complexity of the IPSec solution and the Linux transform layer, WireGuard exposes a virtual interface that appear to be stateless and can be configured with usual command line tools. The interface provided by WireGuard has to be configured with a private key and the public keys of the peers with whom the host will communicate. Once this configuration is done, all the prior steps needed to establish an encrypted tunnel are done behind the scene.

Another common solution to create encrypted tunnels is to use OpenVPN, a solution based on a user space TUN/TAP that uses TLS. Because OpenVPN lives in user space, the performance of that solution is quite poor due to the fact that the packets have to be copied multiple times between kernel and user space. Furthermore, OpenVPN is far from being stateless to the user. WireGuard tries to address those issues with a solution living in the kernel space and a network interface that appears to be stateless to the user.

For the key distribution, WireGuard, like OpenSSH, is agnostic: the peers exchange their public keys in some unspecified way. The attitude of WireGuard is to claim that this is not the right place to address that specific consideration. Even if both IPSec and OpenVPN are right to be implemented the way they are, WireGuard takes a different approach and starts by violating the layering approach. Then it uses practical engineering solution and cryptography to solve the issues caused by the flawed layering approach.

2 Sending and receiving flows

2.1 Cryptokey routing table

The identification of the peers in the WireGuard is done strictly based on their public key - a 32-byte Curve25519 point. In the result there is a mapping done between the public keys and a set of allowed IP addresses. It is stored in a so called **cryptokey routing table**. In such a table there are Interface Public and Private Keys stored. There is also a UDP port stored on which the Interface listens. In the second part of the table the peers public keys are listed and the allowed source IPs that are associated with them.

WireGuard uses an wg0 interface and when outgoing packet is being transmitted on it, the cryptokey routing table is consulted to determine which public key will be used for the encryption. By analogy, when there is an incoming packet arriving it will be analysed only if the source IP matches the key. In the result the main idea is close to the organisation of the firewall rules and it can be realised because WireGuard is a secure network tunnel operating only at layer 3.

On arrival of correctly authenticated packet from a peer, WireGuard will have to determine the endpoint. It can be done either using the outer external source or the information can be simply stored in the same cryptokey routing table. In the first case identifying the remote endpoint of a peer enables peers to change external IPs during the connection (for example changing mobile network). The tables can be updated by the host after receiving every single packet. When

Interface Public Key	Interface Private Key	Listening UDP Port
HIgo...8ykw	yAnz...fBmk	41414
Peer Public Key	**Allowed Source IPs**	**Internet Endpoint**
xTIB...p8Dg	10.192.122.3/32, 10.192.124.0/24	
TrMv...WXX0	10.192.122.4/32, 192.168.0.0/16	
gN65...z6EA	10.10.10.230/32	192.95.5.64:21841

Figure 1: Example configuration of cryptokey routing table from [1]

receiving the packet from the peer, the host will simply update its IP basing on the source IP of received packet.

There is also another rule adding more simplicity, it ensures that the listen port of peers and the source port of packets sent are always the same. Moreover it is convenient when the NAT is used.

2.2 Sending flow

- The original plaintext packet is first transmitted to the Wireguard wg0 interface.
- The cryptokey routing table is inspected in order to find a match with the destination IP address of the packet. If no peer matches the destination IP address, the packet is then dropped. In the latter case, the sender is informed with a standard ICMP "no route to host" and a returning -ENOKEY to user space.
- If a match is found, the plaintext packet is then encrypted using ChaCha20Poly1305 with the associated symmetric encryption key and nonce counter.
- Once the packet is encrypted, a header is prepended.
- The header and encrypted packet are then sent as payload in a UDP packet to the destination IP address's associated UDP/IP endpoint. If no peer match is found, the packet is dropped and the sender is informed with an ICMP message and a returning -EHOSTUNREACH to user space.

2.3 Receiving flow

- The UDP/IP packet is received in the correct listening port.
- Wireguard determines which peer is associated by using the header. Then it checks the validity of the message counter and attempts to authenticate and decrypt it using the receiving symmetric key of the secure session. If no peer match is found or if the authentication fails, the packet is dropped.
- Once the packet is correctly authenticated, the source IP address is then used to update the UDP/IP endpoint to its corresponding entry in the cryptokey routing table.
- The packet payload is decrypted. WireGuard checks if the source IP address of the plaintext packet routes according to the cryptokey routing table, otherwise the packet is dropped.
- If packet is not dropped, it is then put into the receiving queue of the WireGuard interface wg0.

3 Cryptographic choices

The protocols and the ciphers used are rigid. In other words, the developers have voluntarily chosen to support only a few ciphers and protocols as it reduces the complexity of update in the context of the discovery of a new vulnerability.

The chosen ciphers are:

- **Curve25519** with a public key size of 32 bytes, for the Diffie-Hellman using Elliptic curves (ECDH).
- **HKDF** for the key derivation function (the function allowing to convert the shared secrets from the Diffie-Hellman to a usable key).
- **ChaCha20**, accordingly to the construction of the RFC7539 and **Poly1305** for the authentication.
- **BLAKE2** for the hashing process.

3.1 Curve25519

Curve25519 is an elliptic-curve-Diffie-Hellman function essentially used due to its high computation speed [2].

A few talks about Curve25519 to conferences such as ECC 2005 or PKC 2006 were organised in the past where this function was discussed under various aspects. Its design decisions, security and computation are explained by the author in *"Curve25519: New Diffie-Hellman speed records"*, published in PKC 2006's journal.

This elliptic curve uses 32-byte public and private keys. The shared secret is also of 32 bytes long. The Curve25519 is one of the most popular methods used in Key-exchange protocols, it uses the Montgomery curve $y^2 = x^3 + 486662x^2 + x$ with the prime number $p = 2^{255} - 19$ and the base point $G = 9$. The function is $E(F_{p^2})$, where F_p is the prime field:

$$\frac{Z}{p} = \frac{Z}{2^{255} - 19}$$

In the mathematical expression above, the exponent 2 is not a square exponent but means that the expression is $\frac{Z}{2^{255}-19}[\sqrt{2}]$.

This algorithm is used in WireGuard in order to share a secret. Indeed, it is a Diffie-Hellman function meaning that it is used as a way to share privately a secret to generate a symmetric key that will be used later to encrypt the data between the peers. In WireGuard, all the data exchanged between the peers rely on the security of the Curve25519, making it a central element of the software.

This function has multiple advantages from multiple points of view. The following performance and security examples are taken from the author's implementation of Curve25519 in [2] to illustrate the different possibilities of this curve.

3.1.1 Performance

According to the author of [2], his software can compute a curve in:

- 832457 cycles on a Pentium III
- 957904 cycles on a Pentium 4
- 640838 cycles on the Pentium M
- 624786 cycles on an Athlon CPU

According to the author, those cycle counts are speed records for the Diffie-Hellman functions.

The public and private keys are short: 32 bytes. In comparison, the standard ECDH functions use 64-byte public keys and can reach a 32 bytes size but they must be compressed and this process takes time and is not really worth it.

ECDH public keys must be validated before using it and this process can take some time as it requires some computation. With Curve25519, any 32-byte string is a valid public key.

The author's software is so small (16kb) that it can completely fit in the instruction cache of a CPU, ensuring the most optimal speed to the software.

3.1.2 Security

The author's software is protected against timing attacks, hyper-threading attacks and cache-timing attacks without altering the computation speed of the curve.

However, some attacks are still possible. The author warns the user that he still has certain responsibilities. For example, the user must generate independent uniform random bytes for the key. They also have the responsibility of keeping the secret key secret.

The user should also pay attention to the key-derivation function that he uses because some can be breakable (for example, the function could return a partially predictable result for a specific input).

It is assumed that the secret-key cryptosystem was properly selected. If the user picks a weak one, an attacker could decrypt or forge messages the way he wants.

3.2 HKDF

Hashed Message Authentication Code (HMAC)-based key derivation function (HKDF) was promoted as a building block in various protocols and applications [3].

It was officially described in May 2010 by Hugo Krawczyk and Pasi Eronen in RFC 5869. It was also presented in the *"Cryptographic Extraction and Key Derivation: The HKDF Scheme"* paper by one of its authors [4].

Hash-based Message Authentication Code, usually known as HMAC, is a kind of secure MAC that ensures both integrity and authenticity of the data. It is defined as followed.

$$K_1 = K \oplus opad$$

$$K_2 = K \oplus ipad$$

$$HMAC_K = h\left(K_1 \mid\mid h\left(K_2 || m\right)\right)$$

where *opad* and *ipad* are constants used to generate K_1 and K_2 form the secret key K and m is a message.

A key-derivation function (KDF) is another essential component of a cryptographic system. It takes as an input the *source key (SK)* - single initial keying material, *context* - unique identifier of the application and *length (l)* and produces one or more strong secret keys of the length l.

In the case when the source key was not selected randomly from the uniformed large key space the security of the output of KDF may be at risk. That is why the assumption of its high value may be dangerous. To solve this problem there is an *extract-then-expand paradigm* implemented.

In the case of HKDF in the first step - extraction - the pseudorandom key is generated using the following formula:

$$K_{new} = HMAC_{salt}(SK)$$

It means that we run HMAC using special non-secret random value called *salt* as a key and the *source key (SK)* (sometimes called IKM - Input Keying material) as a message.

The next step is about expanding. Here the input parameters are: *PRK - Pseudorandom key* - the output from the extract step (of *HashLen* octets), *CTX - context* - application specific information, *l - length*. In the result we receive a strong secret key with the length of l octets. Sometimes called the *OKM - Output Keying Material*.

The result is obtained using the following formula:

$$N = \left\lceil \frac{l}{HashLen} \right\rceil$$

$$T(0) = empty\ string$$

$$T(1) = HMAC_{PRK}(T(0) \mid\mid context \mid\mid 0x01)$$

$$T(2) = HMAC_{PRK}(T(1) \mid\mid context \mid\mid 0x02)$$

$$...$$

$$T = T(1) \mid\mid T(2) \mid\mid T(3) \mid\mid ... \mid\mid T(N)$$

And the output are the first l octets of T.

Thanks to the described extract-then-expand solution the HKDF ensure more entropy and in the result provides more security then traditional solutions. Extended security analysis is done in [4].

In the WireGuard the hash function used in HMAC is BLAKE2s. In the specification of the protocol often appears function $KDF_n(key, input)$. It is done implemented similarly to the steps mentioned above. *T0* is defined as followed

$$T0 := HMAC_{key}(input).$$

Then

$$T1 := HMAC_{T0}(0x1)$$

and

$$T_i := HMAC_{T0}(T_{i-1}||i).$$

3.3 ChaCha20 Encryption Algorithm

The ChaCha family of stream ciphers, also known as Snuffle 2008, is a variant of the Salsa20 family of stream ciphers. The ChaCha20 is a high-speed cipher designed by D.J. Bernstein [5] that uses a 256-bit key and a nonce counter.

ChaCha20 inputs:

- The 256-bit key.
- 32-bit initial counter. Usually is initialized with the value zero or one but it can be set to any number.
- 96-bit nonce or Initialization Vector.
- Plaintext.

ChaCha20 output:

- Encrypted message (same length as the input plaintext).

The first step of the algorithm is to create the key-stream which is the successive concatenation of the 64-bit key-stream blocks, produced by the **ChaCha20 block function**. The resulting key-stream is then XORed with the plaintext to produce the corresponding ciphertext. Notice that in order to encrypt the plaintext, it is necessary to produce a key-stream at least long as the plaintext. In order to output a ciphertext with the same length as the plaintext, the extra key-stream is not considered.

3.3.1 ChaCha20 block function

This function receives the 256-bit key, the 96-bit nonce, and the 32-bit block counter. All inputs are considered as a concatenation of 32-bit little-endian integers.

ChaCha20 state (16 integer numbers) is initialized as the following:

- The first 4 integers are the constants : 0x61707865, 0x3320646e, 0x79622d32, and 0x6b206574.
- The next 8 integers from the 256-bit key.
- The next integer is the block counter.
- The last 3 integers from the nonce.

The ChaCha20 state can be viewed as a 4x4 matrix:

$$\begin{bmatrix} integer00 & integer01 & integer02 & integer03 \\ integer04 & integer05 & integer06 & integer07 \\ integer08 & integer09 & integer10 & integer11 \\ integer12 & integer13 & integer14 & integer15 \end{bmatrix}$$

Chacha20 block function then runs 20 rounds, alternating *column rounds* and *diagonal rounds*. The column round consist in the following four quarter-rounds:

1. QUARTERROUND (0, 4, 8, 12)
2. QUARTERROUND (1, 5, 9, 13)
3. QUARTERROUND (2, 6, 10, 14)
4. QUARTERROUND (3, 7, 11, 15)

The diagonal round consist in the following four quarter-rounds:

5. QUARTERROUND (0, 5, 10, 15)
6. QUARTERROUND (1, 6, 11, 12)
7. QUARTERROUND (2, 7, 8, 13)
8. QUARTERROUND (3, 4, 9, 14)

At the end of each round, we add the original input words to the output words (addition modulo 2^{32}). The result is then serialized by sequencing the words one-by-one in little-endian order, which gives us an 64-bit keystream block output.

3.3.2 ChaCha Quarter Round function

The quarter round function QUARTERROUND(a,b,c,d), operates the four 32-bit unsigned integers as follows:

a += b; d $\oplus$= a; d <<<= 16
c += d; b $\oplus$= c; b <<<= 12
a += b; d $\oplus$= a; d <<<= 8
c += d; b $\oplus$= c; d <<<= 7

The + denotes addition modulo 2^{32}, $\oplus$ denotes the XOR operation, and $<<< n$ denotes n-bit left rotation.

3.4 Poly1305

Poly1305 designed by D. J. Bernstein [6] is an extremely high speed one-time authenticator that takes a 256-bit one-time key and a message, and produces a 128-bit tag.

3.4.1 Construction

The key is divided in two parts of 128 bits: (r, s). It is required that the pair (r, s) is unique and unpredictable for every message that will be authenticated. To achieve this, r needs to be modified as follows:

the 128-bit r, is treated as a 16-octet little-endian number. This value is then **clamped**, which means that the top 4 bits of r[3], r[7], r[11] and r[15] are set to zero, while the bottom 2 bits of r[4], r[8] and r[12] are set to zero.

Notice that s does not need to be modified in order to be unpredictable, because both r and s are generated each time.

We need to initialize a constant p to $2^{130} - 5$ and a variable **accumulator** (a) to zero.

Now that we have s, r clamped, p, and a; we divided the message into 16-byte blocks and then apply the following operations to each block:

- Reads the block as a little-endian number.

- Add one bit beyond. For a full 16-byte block, this means adding 2^{128}. For a shorter block, it will depends of its size.
- If the last block is not 17 bytes long, pad it with zeros.
- Calculate the new value of $a = ((\text{block}+a) * r) mod p$

Finally, we add the values of s to the accumulator a. The 128 least significant bits are serialized in little-endian order to form the 128-bit tag.

3.4.2 Using ChaCha20 to generate the key

If we assume that we have a 256-bit session key for the MAC, the one-time key used in the Poly1305 algorithm can be generated using the ChaCha20 block function:

- The ChaCha20 key is the 256-bit session key.
- The block counter is set to zero.
- The protocol will specify a 96-bit nonce.
- From the 512-bit state generated by the ChaCha20 block function, we use the first 256 bits as the one-time Poly1305 key (r, s).

3.5 AEAD_CHACHA20_POLY1305

The ChaCha20 and Poly1305 primitives can be combined to create an authenticated encryption with additional data (AD) algorithm.

3.5.1 Inputs

A 256-bit key, a 96-bit nonce, a plaintext with arbitrary length, and additional authenticated data (AAD).

Procedure

- The Poly1305 one-time key is generated with the ChaCha20 block function, the 256-bit key, and the nonce.
- The plaintext is encrypted using the ChaCha20 encryption function with the same 256-bit key and nonce.
- A new message is constructed as the concatenation of:
  - The AAD.
  - Padding1: up to 15 bytes of zeros, in order to have a total length multiple of 16.
  - The ciphertext.
  - Padding2: up to 15 bytes of zeros, in order to have a total length multiple of 16.
  - AAD length in octets (64-bit little-endian integer).

- Ciphertext length in octets (64-bit little-endian integer).

- The Poly1305 function is called using the generated one-time key and the concatenated message.

3.5.2 Output

A ciphertext with the same length as the plaintext and a 128-bit tag.

3.5.3 Decryption

Same procedure as before but this time, the ChaCha20 encryption function is used on the ciphertext to obtain the plaintext. To verify if the message is authenticated, the calculated tag should match the received tag (bitwise compared).

3.5.4 Security considerations of ChaCha20Poly1305

- The ChaCha20 encryption function is designed to provide 256-bit security only if we can ensure the uniqueness of the nonce.

- The Poly1305 authenticator is designed to be SUF-CMA(strong unforgeability against chosen-message attacks). This means that forged messages are rejected with a probability of $1 - (\frac{n}{2^{102}})$ for a 16n-byte message.

- Both ChaCha20 and Poly1305 algorithms were designed to be easily implemented in constant time to avoid side-channel vulnerabilities.

3.6 BLAKE2

BLAKE2s [7] is the hashing function used in WireGuard. BLAKE2 is a hashing function based on the SHA-3 finalist BLAKE and is designed to be optimized for speed in software. The original BLAKE hashing function proposed in the SHA-3 competition is based on the ChaCha stream cipher. BLAKE2 tries to improve that construction in order to reduce the time needed to compute the hash of large amount of data as well as the performance when hashing small amount of data.

BLAKE2 aims at providing an efficient function for speed in software. In fact, the SHA-3 competition succeeded in selecting a hash function that is faster than SHA-2 in hardware. However, SHA-3 does not perform as well as its predecessor, SHA-2, in software. As a result, many systems still use algorithms like MD5 or SHA-1 to meet the speed requirement event if they are known to be insecure.

The BLAKE2 hash function comes in to version: BLAKE2b (or just BLAKE2) optimized for 64-bit platforms and BLAKE2s (used in WireGuard) optimized for 8 to 32-bit platforms. This section will mainly focus on the BLAKE2s version as we are interested in the WireGuard case.

BLAKE2 builds on the high confidence built by BLAKE in the SHA-3 competition. In fact, the final SHA-3 report [8] stands that BLAKE has “very large security margins” and that the cryptanalysis on BLAKE at the time of the report “appears to have a great deal of depth, while

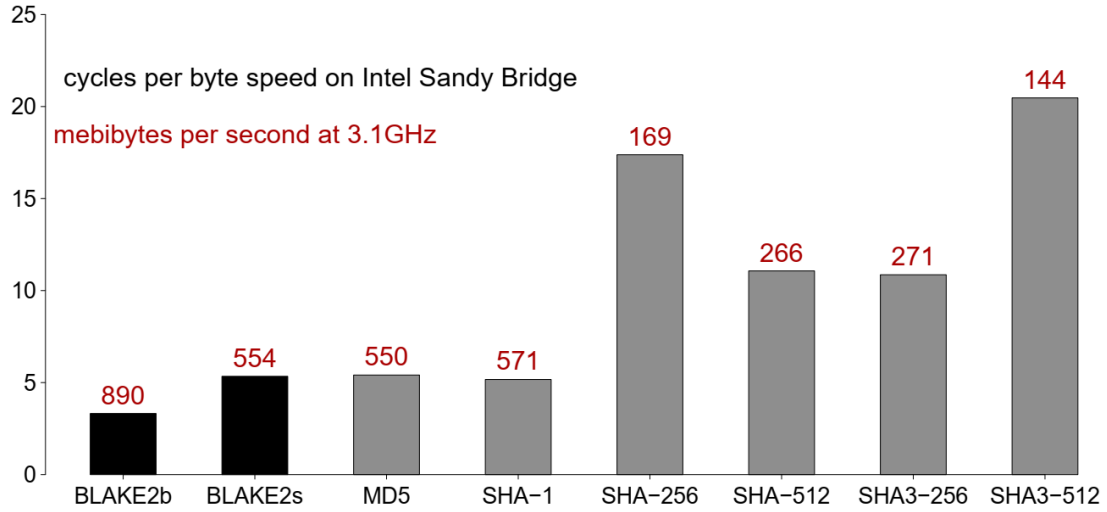

Figure 2: Speed comparison of various popular hash functions, taken from eBACS's "sandy" measurements. Some of the functions are known to be vulnerable. Illustration from [7].

the cryptanalysis on Keccak has somewhat less depth". Moreover, the NIST already mentioned that "the ARX-based algorithms, BLAKE and Skein, perform extremely well in software".

Based on BLAKE, BLAKE2s improved the speed of hashing mainly by reducing the number of rounds. In fact, the original BLAKE had 14 rounds where BLAKE2s does 10 rounds. The authors of BLAKE2 assure that, based on the analysis performed so far, it is unlikely that 14 rounds are significantly more secure than BLAKE2s' 10 rounds. This major modification allows BLAKE2s to be 29% faster than BLAKE on long data. The speed for short data hashing also improves significantly.

The memory used by BLAKE is improved with the BLAKE2s construction. BLAKE originally use a total of 24 word constants and BLAKE2 lower that number as low as 8 word. This improvement allows to save 64 ROM bytes and 64 RAM bytes for the BLAKE2s implementation.

Another change when comparing BLAKE2 to BLAKE is that the original BLAKE function is, like SHA-1 and SHA-2, big-endian but BLAKE2, like MD5, is little-endian. This choice is motivated by the fact that most target platforms are little-endian. This may provide some improvements regarding the speed of execution but also simplify the implementation of BLAKE2.

Only the major improvements are covered in this section but more improvement are described in [7].

BLAKE2 comes with keyed hashing capabilities that allowed the function to be used to compute MAC and act as PRF easily. BLAKE2s also allows parallel hashing specified by BLAKE2sp that runs 8 instances of BLAKE2s in parallel. This function is slightly different than BLAKE2s.

In WireGuard [1], BLAKE2s is used when computing a hash is needed. WireGuard also takes advantage of the BLAKE2s capability to be used to compute MAC: the keyed version of BLAKE2s is used to compute the MAC in WireGuard. Finally, the BLAKE2s function is also used in a HMAC construction when needed by WireGuard.

4 Security designs

4.1 Silence is a virtue

WireGuard avoids storing a state prior to authentication and does not send replies to unauthenticated packets. Those choices help to prevent a certain range of attacks against WireGuard. In fact, an attacker cannot exhaust the computing power of a host running WireGuard by sending unauthenticated packets as no state for those packets is stored. Not sending replies to unauthenticated packets also helps to prevent the detection of WireGuard: if WireGuard is executed on a host but invisible to the attackers, it reduces the attack surface.

In WireGuard, the very first message received by the responder must authenticate the initiator. In order to prevent replay attacks or forcing the responder to regenerate its ephemeral session key with a replayed packet, the protocol uses a 12-bytes TAI64N timestamp, encrypted and authenticated in the first message. The responder keeps track of the greatest timestamp received per peer. This design allows the responder to discard any message with timestamp equals or smaller than the one he observed. This also prevent a replay attack from resetting a ongoing session between two peers.

TAI64N has also been chosen with the implementation in mind. Indeed, due to its big-endian nature, comparisons between timestamps can directly be done using memcmp().

4.2 Denial of service mitigation and cookies

In order to determine the authenticity of a handshake message there is a computation of Curve25519 point multiplication done. Even though this process can be fast on the majority of the processors, it is still CPU intensive. This shows the potential risk of denial of service (DOS) attack.

The idea to solve this problem is rather easy. When the recipient of a message is loaded it can decide not to make a typical handshake. In that case he will send a special cookie replay message. The cookie included in it can be used later on by the initiator to resend a request.

In this cookie we will store a result of computation of MAC of the initiator's source IP address. The MAC key's will be a secret random value, generated by the responder every two minutes. When the initiator will decide to resend its message, he will send a MAC of it using the cookie as the MAC key. Depending on the correctness of this MAC, the responder will be able to decide whether he will accept or reject the connection. This idea is used to proof the IP ownership and then apply some classical IP rate limiting algorithms.

In reality there are 2 MACs used to fix some vulnerabilities, such as the fact that we would prefer not to reply to the suspicious messages and rather stay quiet, that keeping them in the plaintext could become a risk of man in the middle attack or that the initiator himself could be DOS attacked.

4.3 Quantum computing resistance

Wireguard lets the peers use an optional mode, allowing them to use a pre-shared symmetric key. This mode improves the strength of the cryptography against quantum computing. It consists of a pre-shared 256-bit symmetric key exchanged between peers to add an extra-layer of encryption. The idea is to defend against recording attacks (the attacker records the traffic and wants to

decrypt it later, when quantum computers will be able to break Curve25519). The main point of this symmetric encryption is the hope that in the future, this symmetric key was forgotten (the attackers think that it is only a Curve25519 that they need break). Furthermore, this solution also works the other way around. If quantum computers can break this pre-shared symmetric key in the near future, they will be unable to decrypt the Curve25519.

5 Conclusion

After having reviewed and discussed the main cryptographic protocols used in WireGuard protocol, we can confirm that Wireguard is a protocol that was designed to be cyber resilient. Each cryptographic choice provides a layer of security to the communication and the possible vulnerabilities were also considered and mitigated by using state of the art cryptographic functions such as Curve25519 for the Diffie-Hellman key exchange between both peers, HKDF for deriving keys in the secure manner, ChaCha20Poly1305 to encrypt and authenticate communications, and BLAKE2 for hashing needs. It is very clear that WireGuard prioritize efficiency and high speed protocols, with the purpose of provide a high quality of service to the users, while ensuring a adequate level of security.

References

[1] Jason A. Donenfeld. Wireguard: Next generation kernel network tunnel. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[2] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[3] P. Eronen Hugo Krawczyk. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). Request for Comments: 5869, 2010. `https://www.rfc-editor.org/rfc/rfc5869.txt`.

[4] Hugo Krawczyk. Cryptographic Extraction and Key Derivation: The HKDF Scheme. Cryptology ePrint Archive, Report 2010/264, 2010. `https://eprint.iacr.org/2010/264`.

[5] Daniel J Bernstein et al. Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5, 2008.

[6] Daniel J Bernstein. The poly1305-aes message-authentication code. In *International Workshop on Fast Software Encryption*, pages 32–49. Springer, 2005.

[7] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: Simpler, smaller, fast as md5. In Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *Applied Cryptography and Network Security*, pages 119–135, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[8] Shu jen Chang, Ray Perlner, William Burr, Meltem Sonmez, John Kelsey, Souradyuti Paul, and Lawrence Bassham. Third-round report of the sha-3 cryptographic hash algorithm competition, November 2012.