Analysis of the WireGuard protocol

Wu, S.P.

Award date:
2019

Master's Thesis

Analysis of the WireGuard protocol

Peter Wu

supervised by
Prof. Dr. Tanja Lange
Jacob Appelbaum
Jason A. Donenfeld

Eindhoven University of Technology
Department of Mathematics and Computer Science

June 17, 2019

Abstract

WireGuard is a new, secure network tunneling protocol that uses modern cryptography. It aims to replace technologies such as IPsec and OpenVPN with a more secure and performant alternative. A reference C implementation is in process of being integrated into the main Linux kernel while a Go implementation provides support for various other platforms, including Microsoft Windows, Android, iOS, and macOS. Given the potential for universal adoption of this protocol, it is highly desirable to perform a careful analysis to avoid flaws in the protocol and implementations. The primary research question is evaluating whether the security claims made by WireGuard are correct and evaluating whether it is suitable for implementing a secure Virtual Private Network (VPN).

To facilitate a proper analysis, we examine the original WireGuard whitepaper, investigate various implementations, and then write an extensive protocol description including rationale for design decisions and an observation of potential weaknesses. We developed tools to aid in network traffic analysis and to test implementations for potential flaws.

We have identified weaknesses including a denial-of-service attack and weaker identity hiding properties, but overall the protocol appears to be solid and we would recommend it when a secure network tunnel is desired.

Acknowledgements

This Master's thesis would not have been possible if Jason A. Donenfeld did not come up with this neat protocol in the first place. I thank him for his relentless work on building and improving the protocol and implementations, and the helpful discussions.

I would also like to thank my supervisor, Tanja Lange, who was very enthusiastic to my thesis proposal. She is a very kind person and goes in great lengths to support me, connecting me to others, providing great feedback with eye for detail, and answering communications even at times where most people would be asleep.

Thanks for Jerry den Hartog for being on my committee.

Thanks to my advisor, Jacob Appelbaum, for providing continuous feedback and involvement in insightful discussions that resulted in the AFRICACRYPT 2019 paper.

Thanks to Daniel J. Bernstein for helping me understand ChaCha20 somewhat better by drawing it on paper.

Thanks to Vlad Krasnov for pointing out the potential amplification issue with transport messages.

Finally lots of thanks to my parents for their unconditional love, tasty and healthy food, and support throughout my life. Thanks to my siblings, family and friends for their support, distractions and fun. ☺

Contents

1 Introduction **1**

2 WireGuard protocol **6**
- 2.1 Protocol overview . . . 6
- 2.2 Noise . . . 10
  - 2.2.1 Execution of the IKpsk2 handshake pattern . . . 13
- 2.3 Protocol messages . . . 18
  - 2.3.1 Handshake messages . . . 21
  - 2.3.2 Cookie Reply message . . . 26
  - 2.3.3 Data message . . . 27
- 2.4 Timers . . . 30
- 2.5 Key rotation . . . 32
- 2.6 Roaming . . . 33
- 2.7 Handshake state machine . . . 34

3 Related work **38**

4 Software **40**
- 4.1 Wireshark Lua dissector . . . 40
  - 4.1.1 Key extraction . . . 42
- 4.2 Wireshark C dissector . . . 42
  - 4.2.1 X25519 implementation . . . 44
  - 4.2.2 Wireshark dissector features . . . 46
- 4.3 Low-level prototyping tool for WireGuard . . . 48

5 Protocol weaknesses and countermeasures **50**
- 5.1 Amplification attacks . . . 50
  - 5.1.1 Validation using a handshake initiation . . . 54

5.1.2 Validation using a token challenge . . . 55
5.1.3 Validation using a token challenge (variant 2) . . . 57
5.2 Handshake initiation collision . . . 58
5.2.1 Impact . . . 61
5.3 Identity hiding . . . 62
5.4 Sender and Receiver indices . . . 64

6 Transitional post-quantum security improvement 65
6.1 Introduction . . . 65
6.2 Quantum attack . . . 66
6.3 A brief comment on extra security options . . . 68
6.4 Blinding flows against mass surveillance . . . 69
6.5 Modified protocol costs . . . 70

7 Conclusion 72

A Informal protocol narration 81

Chapter 1

Introduction

In the current age, people take access to the Internet for granted; it is considered a commodity item. People access their email, look up information from websites or retrieve documents from a shared network folder. In the background, operating system updates are retrieved, shared folders are synchronized, and new social media posts are loaded.

All of these actions can be performed from various devices, including laptops, smartphones and tablets running various operating systems. These typically use Wi-Fi or cellular networks for connectivity.

With public Wi-Fi networks eavesdropping is trivial, allowing a shady person or company to track one's activity and alter information. Even Internet Service Providers (ISPs) are not always trustworthy, they might decide to inject HTTP headers to facilitate user tracking [36], inject advertisements, or block access to services.

To protect themselves against such untrustworthy actors, users can set up a Virtual Private Network (VPN). The user runs a client on their devices and connects to some remote server and the VPN protocol ensures that the link between these two nodes are protected. IPsec and OpenVPN are two well-known, older, established protocols for this purpose, but are understood to be complex and easily misconfigured such that they fail to protect the user. For example, OpenVPN provides many different parameters to control its security settings, and these may have known vulnerabilities [8].

It is worth noting that the term *private network* originates from Internet Protocol (IP) addressing standards. It describes a range of IP addresses that cannot be publicly routed over the Internet and contrasts with *public network*. These private addresses are typically used in residential and office

networks. VPNs were originally used to enable users to access services in their private network from remote sites. Nowadays, commercial providers have tried to redefine *private* as *privacy*, including concealment of the user's physical location and anonymity. This for example permits geographical restrictions to be bypassed, enabling people to watch videos which would otherwise have been locked to a single country. This differs from the original use case for VPNs, in this example the VPN provider is not used to access private network hosts, but merely as an intermediate to access the public Internet. Nevertheless, both examples do depend on one common piece that is being offered by VPNs: a secure pipe between the user and the VPN provider.

A pipe can be constructed in many ways, but ideally one would like a solid pipe which is not leaky or quickly broken under some pressure. This is best done by design by not including features that make the pipe bulky and prone to breakage, and instead select quality components that others have vetted.

WireGuard [23] is a new VPN protocol that fits the role of this new pipe and it looks quite promising. Note that WireGuard was originally presented at NDSS 2017 [15], but while the main concepts still apply, the protocol has slightly evolved in an incompatible way. The latest version is described in the WireGuard whitepaper [18].

WireGuard is designed using modern cryptography, aims for high performance and reduces the attack surface as a simple protocol. Unlike other protocols, no form of negotiation over cryptographic parameters is possible. Instead, it uses the following constructions and algorithms:

Noise protocol framework A collection of cryptographic handshake patterns which provide building blocks to construct new secure protocols with authenticated key agreement [47].

ChaCha20-Poly1305 The ChaCha20 stream cipher and Poly1305 authenticator, used in an Authenticated Encryption with Additional Data (AEAD) construction. This is also used in modern security protocols such as TLS 1.3 [45]. It provides authenticity and confidentiality of transported data.

X25519 An elliptic-curve-Diffie-Hellman (ECDH) function [4]. Compared to other functions, it has a rather small key size and simpler require-

ments regarding key validation. It is used in the key agreement protocol.

HKDF The HMAC-based Key Derivation Function (HKDF) is a construction to derive one or more keys from an initial secret [40]. It is used to link all pieces of the handshake state to each other, including keys and protocol messages. It is also ensures that the original key material that is involved in calculations cannot be recovered.

BLAKE2 A fast cryptographic hash function, BLAKE2s [50], is used by the HKDF and as a message authentication code (MAC).

WireGuard has two primary implementations:

- WireGuard (Linux kernel implementation) [20]. The name *WireGuard* refers to both the protocol and the primary implementation written in the C programming language. From now on we will refer to this as the *Linux kernel implementation*. This implementation has been heavily optimized for speed, part of that is accounted to a full in-kernel implementation as opposed to a user space solution such as OpenVPN. The latter requires a context switch between kernel space and user space which incurs a performance penalty. Note that the process of integrating it in the main kernel sources is still in progress, so the source code currently lives in a separate Git repository [20].

- wireguard-go [21], a user space implementation in the Go programming language. The user space program implements the protocol but relies on the operating system to expose an IP tunnel interface where network packets can be sent and retrieved from. This interface is included by default on Linux (including Android), macOS, iOS, and BSD. On Microsoft Windows, this interface is not provided by default. The OpenVPN tuntap driver is popular, but criticized for its old and complicated code that uses legacy application programming interfaces (APIs) [19]. The WireGuard author has therefore started developing a new minimal TUN driver, Wintun [24],

Apart from these primary implementations, there are also some toy and other third-party user space implementations, including:

- wireguard-rs [22], a toy implementation written in the Rust programming language. It uses a generic software library that implements the Noise protocol.

- boringtun [11], a newer implementation in Rust, developed by Cloudflare. Its current version is compatible with WireGuard although there are known issues related to key rotation [58].

- TunSafe [54], the initial Windows implementation developed by a third-party. It relies on the TAP driver from OpenVPN. This implementation also includes a TCP transport which is non-standard as WireGuard is only defined for UDP.

With an increasing number of implementations, it becomes increasingly important to have a complete and precise specification with implementation guidance and have tools available for verification purposes.

In this thesis we provide a detailed description of the WireGuard protocol including potential attacks to defend against. This description is not limited to the contents of the whitepaper, but also reflects a review of the Linux kernel implementation, and to a lesser extent, the wireguard-go and wireguard-rs implementations. It also incorporates our experiences from reviewing the boringtun implementation (Section 5.4). We also present a state machine based on an analysis of the protocol specification and protocol implementations.

We have developed several WireGuard implementations, namely Lua and C dissectors for the Wireshark protocol analyzer software, and a Python implementation that enables testing of other implementations. As part of the Wireshark dissectors, we have also developed methods to extract decryption keys from the Linux kernel, and investigated various X25519 implementations in C.

Based on our protocol description and an evaluation using our newly developed tools, we were able to discover issues such as weakened identity hiding (Section 5.3) and denial-of-service attacks due to IP spoofing (Section 5.1). We discovered potential operation issues such as a longer session setup time in presence of cookies or handshake initiation collisions (Section 2.3). We discovered some important implementation details that were not documented in the specification from the whitepaper, such as Explicit Congestion Notification (ECN, Section 2.3.3) and handshake initiation jitter (Section 5.2).

During the thesis, we wrote a related peer-reviewed paper with Appelbaum and Martindale which was accepted to AFRICACRYPT 2019 and describes a transitional post-quantum security improvement (Chapter 6). This description and previous specification issues have been communicated back to the designer of the WireGuard protocol. Our key extraction approach has been adopted for the handshake keys extraction tool in the WireGuard project. The Wireshark modifications have also been contributed and accepted by the Wireshark project. Our related pcapng format changes to store WireGuard keys have also been contributed to the official pcapng specification.

This thesis is organized as follows. Chapter 2 provides an extensive description of the protocol. Chapter 3 describes related work in the area of analysis. Chapter 4 describes the software we developed in order to support protocol analysis. Chapter 5 describes weaknesses in WireGuard based on our analysis and suggests some countermeasures. Chapter 6 discusses an improvement that ensures transitional post-quantum security. Finally, Chapter 7 summarizes our findings and our conclusions.

Chapter 2

WireGuard protocol

This chapter describe the WireGuard protocol based on our interpretation of the whitepaper and analysis of various implementations. Section 2.1 provides a high-level overview of the protocol characteristics. Section 2.2 describes the handshake protocol on which WireGuard's key exchange protocol is based. Section 2.3 incorporates that handshake protocol and defines all messages exchanged on the wire for the full WireGuard protocol. Section 2.4 describes time-related characterstics, Section 2.5 describes key rotation considerations, and Section 2.6 provides a discussion on roaming between networks.

2.1 Protocol overview

The WireGuard protocol provides a secure OSI Layer 3 [46] network tunnel between two endpoints using User Datagram Protocol (UDP) as transport for protocol messages. It uses a cryptographic handshake protocol based on the Noise Protocol Framework [47] to provide mutual authentication, key agreement, and forward secrecy. Transport data such as IP packets encapsulated in WireGuard tunnels are protected using Authenticated Encryption with Additional Data (AEAD).

Either endpoint can start a new session following transmission of a handshake message and receipt of a corresponding handshake response message. There is no single *client* or *server* role in the protocol as is common in other tunneling protocols. The endpoint that wishes to start a new handshake is referred to as an **initiator** while the peer that it tries to communicate with is called the **responder**. Figure 2.1 depicts a usual session with periodic key

rotation. Peers may change roles over time, after some inactivity from both sides, a responder might become an initiator of a new session and its peer will subsequently assume the role of a responder.

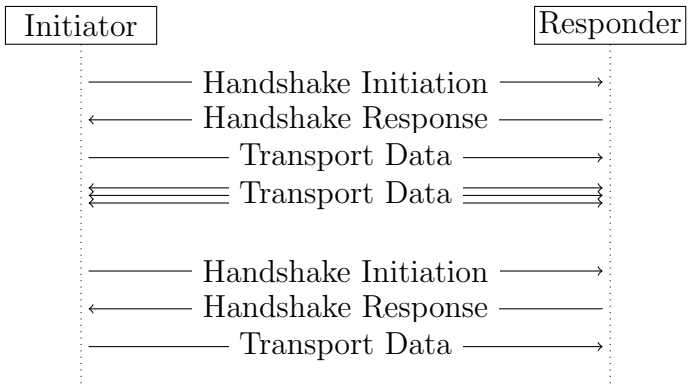


Figure 2.1: Protocol overview: after completing a 1.5-RTT handshake, both peers can send data. After some time has passed, rekeying occurs which is indistinguishable from a new handshake.

A single UDP source port is used for both the handshake and data channel. This port number is the same regardless of the initiator or responder role, and simplifies network address translation (NAT) traversal.

Endpoints are identified by a static 32-byte Curve25519 public key and will never respond to messages unless the sender proves knowledge of this public key. Adversaries can therefore not probe arbitrary systems for WireGuard using port scanning without knowing the long-term static public keys of the endpoints. However, this also makes troubleshooting of configuration issues more difficult due to the lack of feedback from the protocol itself.

A 1.5 round-trip time (1.5-RTT) handshake ensures fast session establishment. Once the initiator sends its initial handshake message and receives a handshake response, it may immediately transmit encrypted data messages. To protect against replay attacks, the responder may not send encrypted data messages until it has received an encrypted data message from the initiator that acknowledges receipt of the response handshake message. Hence this is considered a 1.5-RTT handshake protocol.

Both parties generate fresh *ephemeral* Curve25519 key pairs during the handshake. The corresponding private keys are erased after the handshake and thereby ensure a forward secret session. This session has a fixed lifetime

and a limit on the number of data messages that can be sent. Endpoints that approach the session expiration time or message limit must switch to new session keys if they wish to continue communication. Frequent, periodic key rotation ensures limited impact in the event of session key compromise and is done by starting a new handshake.

The protocol does not have an explicit shutdown signal, it only removes sessions after a fixed duration. Even if one side has removed its VPN tunnel, the peer would not be aware of this and continue forwarding data.

A pre-shared symmetric key (PSK) can optionally be configured between two parties. This provides post-quantum security as long as the PSK remains secret. A passive adversary [31] that records all network traffic today and learns the PSK later will be able to decrypt this traffic in the future using a quantum computer. If the PSK is never revealed, then this traffic cannot be decrypted, even to an attacker with a quantum computer.

To protect endpoints against denial-of-service attacks that trigger expensive handshake calculations, handshake messages are authenticated by a keyed BLAKE2s [50] message authentication code (MAC). The primary protection mechanism (*MAC1*) uses the receiver's static public key as MAC key. Handshake messages that fail this check will be ignored and silently dropped. The secondary mechanism (*MAC2*) uses a *cookie* as MAC key and is only enabled when a receiver is considered "under load". If "under load", the receiver will reject handshake messages and respond with an encrypted cookie that can only be decrypted by the original sender. This cookie can subsequently be used to authenticate new handshake messages.

WireGuard does not have additional retransmission logic for IP packets that are transmitted in data messages. If an encrypted transport data message is lost, then higher protocol layers within the tunnel are assumed to react accordingly. The handshake is a special case, if an initiator does not receive a timely response, it will periodically attempt a new handshake with new ephemeral keys until a response is received, or until the "rekey" timeout is reached.

A WireGuard network interface can be connected to multiple endpoints. To ensure that IP traffic is routed to exactly one of those endpoints, WireGuard has coined the term **Cryptokey routing** as a concept for this policy. An endpoint is associated with a set of *allowed* IP addresses, possibly covering all IPv4 or IPv6 addresses. Local traffic directed to the WireGuard network interface will be sent to an endpoint only if the destination IP address is within its set of allowed IP addresses. Inbound traffic is only accepted

if the source IP address of the decrypted IP packet is within the set of allowed IP addresses from the authenticated peer.

Do note that the operating system must still be properly configured to route traffic to the WireGuard network interface. Failure to do so could result in leaking network traffic over unprotected network links. These network routes are expected to cover the same addresses as described by the Cryptokey routing table, but are not necessarily the same. If the Cryptokey routing table does not cover an address, then traffic passed from the operating system to WireGuard will be dropped. IP packets originating from the local host will be answered with an ICMP "no route to host" error.

This mechanism can be used to implement a VPN configuration where a "client" routes all of its traffic to a "server", and where this "server" would only forward traffic to the "client" if the destination IP address matches. In WireGuard *implementations*, as opposed to the wire protocol, this can be achieved via the per-peer *Allowed IPs* setting. On the client, the setting for the server would be set to 0.0.0.0/0 which is the equivalent of accepting all traffic. On the server, the setting for the client could be set to 192.0.2.2/32 for example.

WireGuard's design choices make the protocol and its implementations more resilient to attacks that have traditionally been possible with other VPN protocols:

- Use of modern cipher constructions eliminates attacks such as Bleichenbacher (RSA encryption in TLS), Sweet32 (Birthday attacks on 64-bit block ciphers in TLS and OpenVPN) [8], and padding oracle attacks like BEAST (AES-CBC in TLS). Similarly, use of an AEAD construction avoids known problematic constructions such as MAC-then-Encrypt as used in older TLS versions and IPsec [13].

- Lack of cryptographic agility [5] due to the use of fixed key exchange and cipher algorithms eliminates downgrade attacks to insecure configurations that resulted in issues such as POODLE and FREAK in TLS. If algorithms have to be updated, a new protocol implementation has to be deployed. New algorithm names will also completely change the derived handshake and session keys, and will thus not be vulnerable to cross-protocol version attacks.

- Avoidance of shortcuts in the handshake protocol for *session resumption* as found in TLS, and IPsec with IKEv2 Session Resumption [52]

prevents issues such as the TLS Triple Handshake attack [9].

- Use of fixed message sizes during the handshake reduces the risk of vulnerabilities that result from complex message parsing requirements. This prevents issues such as Heartbleed, and parser bugs in ASN.1 or X.509 implementations.

- Remaining silent and not sending messages in response to hosts that do not know the static public key of the recipient prevents network scanners from discovering potential WireGuard installations.

2.2 Noise

Noise [47] is a framework for protocols based on the Diffie-Hellman (DH) key exchange. Protocol designers can use this to build new protocols with properties such as mutual authentication, identity hiding, and forward secrecy for derived shared symmetric keys. Instantiations of these protocols have been formally analyzed and verified [37]. While not all Noise protocols guarantee these security properties, the IKpsk2 protocol variant selected by WireGuard does provide them. The remainder of this section provides a general overview of Noise and is not specific to WireGuard.

A protocol always starts with an exchange of **handshake messages** between two parties which are used to derive symmetric keys to encrypt **transport messages**. The initial sender is called the **initiator** while the intended recipient is referred to as the **responder**.

Each party always creates a fresh **ephemeral key pair** for every new handshake, the corresponding public key is sent to the peer. In protocols that require authentication, each party additionally possesses a long-term **static key pair** to identify themselves. The static public keys may optionally be published in handshake messages in case a party has no prior knowledge about the peer identity. These ephemeral and static key pairs are used in a series of Diffie-Hellman computations to ensure forward secrecy. To ensure a cryptographic binding of derived secrets to the handshake, the symmetric keys for handshake and transport data encryption are derived using all handshake messages and key material accumulated so far.

Protocols can also mix a 256-bit **pre-shared symmetric key** into the handshake. This can be used to provide channel binding between the Noise session and an earlier protocol-specific conversation. For example, a hybrid

post-quantum key exchange could be executed prior to the Noise handshake and the output of this prior key exchange could subsequently be transformed into a PSK. For post-quantum security, the PSK could also be fixed between different handshakes as long as the PSK is eventually forgotten before quantum computers become practical.

Each handshake message is described by a **message pattern**. Such a message pattern contains a sequence of predefined **tokens** that describe the required changes to the cipher and handshake state, and data to read or write. The cipher state consists of a key and nonce for encryption, a handshake hash, and a *chaining* key from which the cipher key is derived. The handshake state stores the local key pairs and the peer's ephemeral and static public keys.

The current set of predefined tokens in message patterns for the Noise framework are `"e"`, `"s"`, `"ee"`, `"es"`, `"se"`, `"ss"`, and `"psk"`. The meaning of these tokens is as follows:

- `"e"`: The sender generates a new ephemeral key pair, writes the public key to the output buffer and hashes the public key into the handshake hash state. In a handshake involving a PSK, the ephemeral public key is additionally mixed in the chaining key. See the description for the `"psk"` token for the motivation on this requirement.

- `"s"`: The sender writes its long-term static public key to the output buffer in encrypted form if a symmetric key has been set, and in plaintext otherwise. This output is also hashed into the handshake hash state.

- `"ee"`, `"es"`, `"se"`, `"ss"`: the sender performs a DH computation using its ephemeral or static private key (indicated by the first letter) and the receiver's ephemeral or static public key (indicated by the second letter). The resulting shared secret is hashed into the chaining key and the cipher is reinitialized with a new key derived from the chaining key.

- `"psk"`: A pre-shared symmetric key is hashed into the chaining key and handshake hash state. Since a symmetric key is directly derived from a chaining key, a nonce must be mixed into the chaining key to ensure that a randomized symmetric encryption key is derived. This prevents fatal key reuse. The ephemeral public key published by the `"e"` token acts as such a nonce. In valid Noise protocols using the `"psk"` token, encryption must therefore be preceded by the `"e"` token. For example,

`"psk, s, e"` would be illegal since `"s"` would publish the encrypted static public key using a fixed symmetric key derived from the PSK.

It follows that the order of tokens within a message pattern is important. For example, `"s, ss"` is different from `"ss, s"`. The former indicates that the long-term static public key of the sender is written in plain text (`"s"`). Subsequently, a DH computation between the static keys of the two parties (`"ss"`) occurs, and any message following this pattern will be encrypted with derived keys. The latter pattern (`"ss, s"`) implies that the long-term static public key of the sender is encrypted using keys derived from a hash that includes the DH computation between the static keys of the two parties (`"ss"`).

Prior to the handshake, protocols might already have knowledge of static and/or ephemeral public keys of the peer. This knowledge of *public* keys is captured by one of the **pre-message patterns** `"e"`, `"s"`, `"e, s"`, or empty in case no pre-messages are involved. The `"s"` pre-message pattern indicates knowledge about the static public key of the other party. The `"e"` pre-message pattern is used in a *fallback* pattern and indicates a previously shared fresh ephemeral public key. None of these public keys will be published again during the handshake, they are implicit.

Together, the pre-message patterns and message patterns form a **handshake pattern** which consists of:

- pre-message pattern for the initiator, conveying information about the responder's public keys.
- pre-message pattern for the responder, conveying information about the initiator's public keys.
- A sequence of message patterns that generate handshake messages. Every message pattern has an associated direction which affects the interpretation of tokens. For example, to the initiator, the `-> s` notation means *send the initiator's static public key* while the responder interprets it as *read the initiator's static public key.*

Without knowing the exact public keys from the pre-message patterns, one will not be able to derive the correct symmetric keys during the handshake and as a result the handshake will fail due to out of sync handshake states.

Noise defines a number of standard handshake patterns with various requirements, properties and security guarantees. User data can be sent after a handshake has completed, but they could also be included right after the output buffer for a message pattern during the handshake. However, sending such data before the handshake has completed can result in weaker authenticity and confidentiality guarantees.

A concrete Noise protocol is instantiated by selecting a **handshake pattern**, a **DH function**, a **cipher function** and a **hash function**. A **prologue** byte pattern can be set to bind the cryptographic keys to this value. WireGuard is instantiated as follows:

handshake pattern	IKpsk2
DH function	X25519
cipher function	ChaCha20-Poly1305
hash function	BLAKE2s
prologue	`WireGuard v1 zx2c4 Jason@zx2c4.com` (34 bytes)

The name of the **IKpsk2** handshake pattern is decomposed as:

- **I**: static public key for the initiator is **i**mmediately transmitted to the responder, despite reduced or absent identity hiding.
- **K**: static public key for the responder is **k**nown to the initiator.
- **psk2**: a **PSK** is used at the end of the **second** handshake message.

The **Noise protocol name** is the concatenation of the string `Noise`, the handshake pattern, and functions, separated by an underscore (_). For WireGuard, this results in: `Noise_IKpsk2_25519_ChaChaPoly_BLAKE2s`.

A description of this specific IKpsk2 pattern will be presented in Section 2.2.1. Finally note that not all potential handshake patterns are valid in Noise, additional validity rules may apply [47, Section 7.3].

2.2.1 Execution of the IKpsk2 handshake pattern

This section will describe the exact computations for the IKpsk2 handshake pattern as used by WireGuard. To ease interpretation, we will apply the notation from the WireGuard paper to refer to certain keys:

- S_i^{pub}: a 32-byte **s**tatic **pub**lic key of the **i**nitiator.
- E_r^{priv}: a 32-byte **e**phemeral **priv**ate key of the **r**esponder.
- Q: a 32-byte pre-shared symmetric key (PSK) used for post-**q**uantum security.
- T_i^{send}, T_r^{recv}: a 32-byte symmetric encryption key which the **i**nitiator can use for **send**ing **t**ransport data messages. It is obviously the same as the key used by the **r**esponder for decrypting **rec**ei**v**ed data.

The handshake and cipher state will be described by the following symbols:

- h: the 32-byte handshake hash.
- ck: the 32-byte chaining key.
- k: the 32-byte handshake cipher key.

We define the following functions which will be used in the following text:

- $\mathsf{DH}(\mathsf{sk}, \mathsf{pk})$: the X25519 [4] DH function, takes a 32-byte secret key sk, a 32-byte public key pk and returns the corresponding 32-byte shared secret.
- DHGen: generates a new uniform random private Curve25519 [4] key sk, computes the corresponding public key pk and returns the key pair $(\mathsf{sk}, \mathsf{pk})$.
- $\mathsf{aead\text{-}enc}(\mathsf{key}, \mathsf{nonce}, \mathsf{plaintext}, \mathsf{aad})$: takes a 256-bit key, a 64-bit nonce, an arbitrary length plaintext, and arbitrary length additional authenticated data. A 96-bit AEAD nonce is constructed by concatenating 32 zero bits with the little-endian encoding of the 64-bit nonce. Apply the ChaCha20-Poly1305 AEAD [45] and return the corresponding ciphertext including a 16-byte authentication tag.
- $\mathsf{aead\text{-}dec}(\mathsf{key}, \mathsf{nonce}, \mathsf{ciphertext}, \mathsf{aad})$: takes a 256-bit key, a 64-bit nonce, an arbitrary length ciphertext including a 16-byte authentication tag, and arbitrary length additional authenticated data. A 96-bit AEAD nonce is constructed by concatenating 32 zero bits with the little-endian encoding of the 64-bit nonce. Apply ChaCha20-Poly1305 AEAD [45] decryption and return the corresponding plaintext.

- $\mathsf{Hash}(\mathsf{data})$: the BLAKE2s [50] hash function, takes arbitrary length data and returns a 32-byte output.
- $\mathsf{HKDF}_n(\mathsf{salt}, \mathsf{input\_keying\_material})$: uses the HKDF construction [40] with unkeyed BLAKE2s [50] as hash function for the HMAC function to derive n keys, each of length 32 bytes.

  $\mathsf{PRK} = \mathsf{HMAC\text{-}Hash}(\mathsf{salt}, \mathsf{input\_keying\_material})$
  $T_1 = \mathsf{HMAC\text{-}Hash}(\mathsf{PRK}, 1)$ ▷ The constant 0x01 is a single byte.
  $T_2 = \mathsf{HMAC\text{-}Hash}(\mathsf{PRK}, T_1||2)$
  ...
  $T_n = \mathsf{HMAC\text{-}Hash}(\mathsf{PRK}, T_{n-1}||n)$
  Return $(T_1, \ldots, T_n)$

Figure 2.2 presents the IKpsk2 handshake pattern in a concise notation which Noise uses to summarize handshake patterns. An elaboration of this pattern including the precise calculations will follow below.

```
    IKpsk2:
0.    <- s
      ...
1.    -> e, es, s, ss
2.    <- e, ee, se, psk
```

Figure 2.2: Summary of IKpsk2 handshake pattern from Noise [47]. The dots separate pre-messages from messages.

The handshake computations start with hashing the protocol name and prologue:

1: $h_0 = \mathsf{Hash}(\texttt{"Noise\_IKpsk2\_25519\_ChaChaPoly\_BLAKE2s"})$
2: $\mathsf{ck}_0 = h_0$
3: $h_1 = \mathsf{Hash}(h_0||\texttt{"WireGuard v1 zx2c4 Jason@zx2c4.com"})$

Next, the handshake (pre-)messages are processed. As some message tokens are interpreted differently by the initiator and responder, lines will be prefixed with `I:` or `R:` to restrict them to the initiator or responder respectively. The three (pre-)messages from Figure 2.2 are interpreted as follows:

0. `<- s`: a *pre-message* from responder to initiator:
   - `"s"`: static public key S_r^{pub} was previously exchanged out-of-band.

4: $h_2 = \mathsf{Hash}(h_1 || S_r^{pub})$

1. `-> e, es, s, ss`: a message from initiator to responder:

   - `"e"`: the initiator generates an ephemeral key pair, sends ephemeral public key E_i^{pub} to the initiator. E_i^{pub} is mixed into the handshake hash, and the chaining key due to the use of a PSK.
     5: `I`: $(E_i^{priv}, E_i^{pub}) = \mathsf{DHGen}$; publish E_i^{pub}
        `R`: Read E_i^{pub}
     6: $h_3 = \mathsf{Hash}(h_2 || E_i^{pub})$
     7: $\mathsf{ck}_1 = \mathsf{HKDF}_1(\mathsf{ck}_0, E_i^{pub})$
   - `"es"`: the initiator computes $\mathsf{es} = \mathsf{DH}(E_i^{priv}, S_r^{pub})$. The responder computes the same shared secret using $\mathsf{es} = \mathsf{DH}(S_r^{priv}, E_i^{pub})$.
     8: `I`: $(\mathsf{ck}_2, k_0) = \mathsf{HKDF}_2(\mathsf{ck}_1, \mathsf{DH}(E_i^{priv}, S_r^{pub}))$
        `R`: $(\mathsf{ck}_2, k_0) = \mathsf{HKDF}_2(\mathsf{ck}_1, \mathsf{DH}(S_r^{priv}, E_i^{pub}))$
   - `"s"`: the initiator sends its encrypted static public key S_i^{pub}. If the responder fails to decrypt the result, it aborts the handshake.
     9: `I`: $\mathsf{enc\text{-}id} = \mathsf{aead\text{-}enc}(k_0, 0, S_i^{pub}, h_3)$; publish $\mathsf{enc\text{-}id}$
        `R`: $S_i^{pub} = \mathsf{aead\text{-}dec}(k_0, 0, \mathsf{enc\text{-}id}, h_3)$; abort on failure
     10: $h_4 = \mathsf{Hash}(h_3 || \mathsf{enc\text{-}id})$
   - `"ss"`: the initiator computes $\mathsf{ss} = \mathsf{DH}(S_i^{priv}, S_r^{pub})$. The responder computes the same secret using $\mathsf{ss} = \mathsf{DH}(S_r^{priv}, S_i^{pub})$.
     11: `I`: $(\mathsf{ck}_3, k_1) = \mathsf{HKDF}_2(\mathsf{ck}_2, \mathsf{DH}(S_i^{priv}, S_r^{pub}))$
        `R`: $(\mathsf{ck}_3, k_1) = \mathsf{HKDF}_2(\mathsf{ck}_2, \mathsf{DH}(S_r^{priv}, S_i^{pub}))$
   - Append an encrypted (possibly empty) payload to the handshake message buffer. If the responder fails to decrypt the result, it aborts the handshake. In WireGuard, the payload must be a monotonically increasing 96-bit unsigned integer encoded in network byte order to avoid a replay of handshake initiation messages (see Section 2.3.1), any other value is treated as failure.
     12: `I`: $\mathsf{enc\text{-}time} = \mathsf{aead\text{-}enc}(k_1, 0, \mathsf{time}, h_4)$; publish $\mathsf{enc\text{-}time}$
        `R`: $\mathsf{time} = \mathsf{aead\text{-}dec}(k_1, 0, \mathsf{enc\text{-}time}, h_4)$; abort on failure
     13: $h_5 = \mathsf{Hash}(h_4 || \mathsf{enc\text{-}time})$

2. `<- e, ee, se, psk`: a message from responder to initiator:

- `"e"`: the responder generates an ephemeral key pair and sends ephemeral public key E_r^{pub} to the initiator. E_r^{pub} is mixed into the handshake hash, and the chaining key due to the use of a PSK.

  14: `R`: $(E_r^{priv}, E_r^{pub}) = \mathsf{DHGen}$; publish E_r^{pub}
  `I`: Read E_r^{pub}
  15: $h_6 = \mathsf{Hash}(h_5 || E_r^{pub})$
  16: $\mathsf{ck}_4 = \mathsf{HKDF}_1(\mathsf{ck}_3, E_r^{pub})$

- `"ee"`: the responder computes $\mathsf{ee} = \mathsf{DH}(E_r^{priv}, E_i^{pub})$. The initiator computes the same secret using $\mathsf{ee} = \mathsf{DH}(E_i^{priv}, E_r^{pub})$.

  17: `R`: $\mathsf{ck}_5 = \mathsf{HKDF}_1(\mathsf{ck}_4, \mathsf{DH}(E_r^{priv}, E_i^{pub}))$
  `I`: $\mathsf{ck}_5 = \mathsf{HKDF}_1(\mathsf{ck}_4, \mathsf{DH}(E_i^{priv}, E_r^{pub}))$

- `"se"`: the responder computes $\mathsf{se} = \mathsf{DH}(E_r^{priv}, S_i^{pub})$. The initiator computes the same secret using $\mathsf{se} = \mathsf{DH}(S_i^{priv}, E_r^{pub})$.

  18: `R`: $\mathsf{ck}_6 = \mathsf{HKDF}_1(\mathsf{ck}_5, \mathsf{DH}(E_r^{priv}, S_i^{pub}))$
  `I`: $\mathsf{ck}_6 = \mathsf{HKDF}_1(\mathsf{ck}_5, \mathsf{DH}(S_i^{priv}, E_r^{pub}))$

- `"psk"`: incorporate the pre-shared symmetric key Q in both the chaining key and the handshake hash.

  19: $(\mathsf{ck}_7, \tau, k_2) = \mathsf{HKDF}_3(\mathsf{ck}_6, Q)$
  20: $h_7 = \mathsf{Hash}(h_6 || \tau)$

- Append an encrypted (possibly empty) payload to the handshake message buffer. If the responder fails to decrypt the result, it aborts the handshake. For WireGuard, the payload is always empty, any other value is treated as failure.

  21: `R`: $\mathsf{enc\text{-}empty} = \mathsf{aead\text{-}enc}(k_2, 0, \mathsf{empty}, h_7)$; publish $\mathsf{enc\text{-}empty}$
  `I`: $\mathsf{empty} = \mathsf{aead\text{-}dec}(k_2, 0, \mathsf{enc\text{-}empty}, h_7)$; abort on failure
  22: $h_8 = \mathsf{Hash}(h_7 || \mathsf{enc\text{-}empty})$

Finally the handshake is finished by deriving symmetric keys for transport data protection. This is done by hashing the chaining key once more with an empty string ϵ:

23: `I`: $(T_i^{send}, T_i^{recv}) = \mathsf{HKDF}_2(\mathsf{ck}_7, \epsilon)$
`R`: $(T_r^{recv}, T_r^{send}) = \mathsf{HKDF}_2(\mathsf{ck}_7, \epsilon)$

At this point, the ephemeral keys must be erased.

Summarizing the serialization of handshake messages in this protocol:

1. Initiator to responder:

- 32 bytes for E_i^{pub}, see Line 5.
- 48 bytes for the encrypted S_i^{pub}, 32 bytes for the ciphertext, 16 bytes for the authentication tag. See Line 9.
- $x + 16$ bytes for the encrypted first payload, x bytes for the ciphertext, 16 bytes for the authentication tag. See Line 12. In case of WireGuard $x = 12$ which implies a field size of 28 bytes.

2. Responder to initiator:

   - 32 bytes for E_r^{pub}, see Line 14.
   - $x + 16$ bytes for the encrypted first payload, x bytes for the ciphertext, 16 bytes for the authentication tag. See Line 21. In case of WireGuard $x = 0$ which implies a field size of 16 bytes.

This concludes the generic overview of a Noise IKpsk2 protocol. WireGuard directly encapsulates this cryptographic handshake with certain fixed choices for the handshake payloads. This adaptation will be described in Section 2.3.

2.3 Protocol messages

As stated before, the WireGuard protocol runs on top of UDP and can thus not rely on in-order, guaranteed delivery of protocol messages. It must be able to cope with packet loss, handle out-of-order messages, associate messages with a communication channel between two peers, defend itself against IP spoofing attackers, and so on. This section will define the protocol message types and provide a motivation to support the aforementioned requirements.

All messages begin with a single **type** byte and must be followed by three zero bytes which are reserved for future use. If the reserved bytes are not zero, the message must be ignored. Current implementations take advantage of this by reading the first four bytes as little-endian integer and interpreting this as the message type. The four message types that are defined for the current protocol version can be found in Table 2.1.

Type	Message Name	Length	Message format
1	Handshake Initiation	148 bytes	See Figure 2.5, Section 2.3.1
2	Handshake Response	92 bytes	See Figure 2.6, Section 2.3.1
3	Cookie Reply	64 bytes	See Figure 2.7, Section 2.3.2
4	Transport Data	$\geq$ 32 bytes	See Figure 2.8, Section 2.3.3

Table 2.1: Summary of message types. In the linked figures, the numbers on the left indicate the starting byte offset for the first column in that row.

In order to link a message to a local handshake or session, a 32-bit index is maintained by each peer. The Handshake Initiation and Handshake Response messages include a **sender index** which advertises their locally chosen 32-bit identifier to the peer. All messages except for the Handshake Initiation include a **receiver index** that echoes the sender index from the peer.

The first two messages, Handshake Initiation and Handshake Response, encapsulate the Noise IKpsk2 handshake messages as described earlier in Section 2.2.1. Following this handshake, the initiator can immediately transfer data in Transport Data messages, while the responder must wait for a Transport Data message from the initiator that implicitly acknowledges receipt of the Handshake Response message. After the responder receives the initial data message, both parties will be able to transmit Transport Data messages. This basic 1.5-RTT handshake is visualized in Figure 2.3 while an informal protocol narration is included in Appendix A. An explanation for all fields in the protocol messages will be provided in later sections.

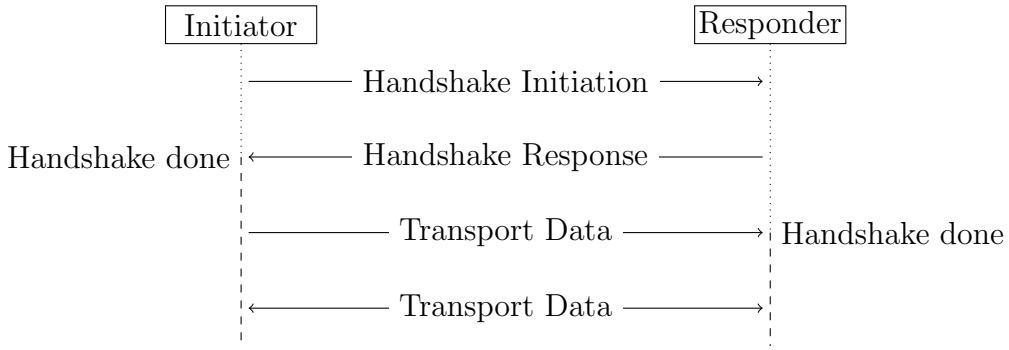


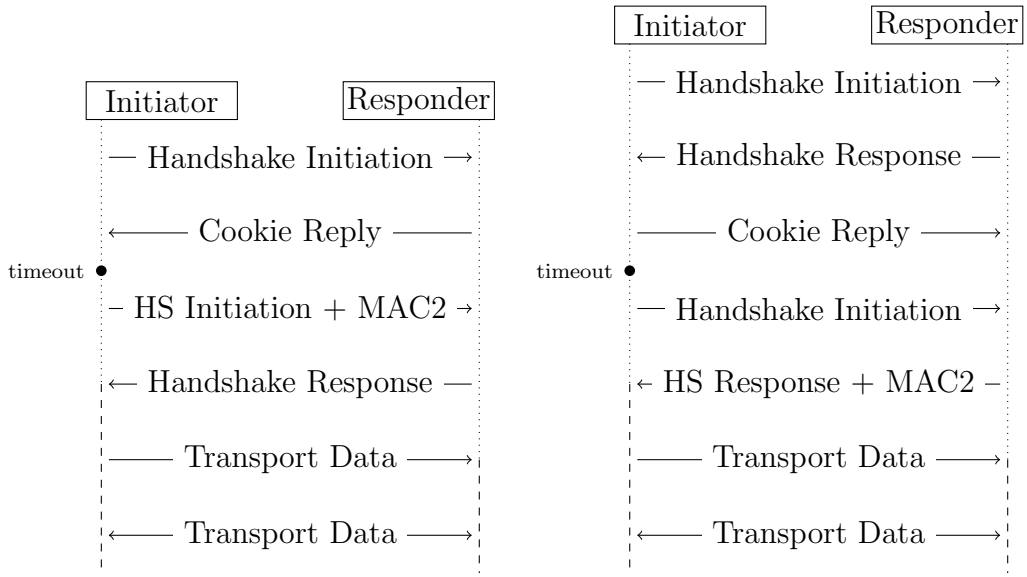
Figure 2.3: Basic protocol flow with a 1.5-RTT handshake. A handshake is only finished after receiving confirmation from the peer.

Recall that handshake initiation and response messages have a MAC2 field which uses a cookie as MAC key. The Cookie Reply message contains an encrypted cookie. The MAC2 field is usually ignored, unless a party is "under load". When either party is "under load", the handshake will become more complicated and take more time due to additional round trips and timer-based delays. The "under load" condition is not precisely specified in the whitepaper, and appears to be implementation defined. For the current Linux kernel implementation, it means that the incoming queue for unprocessed handshake messages contains at least $\frac{4096}{8} = 512$ items.[1]

Consider a responder who is under load (Figure 2.4a). After the initiator sends its Handshake Initiation message, the responder issues a Cookie Reply. The initiator decrypts the cookie and remembers it, but will **not** immediately send another handshake message with this cookie. Instead, it has to wait for a "rekey timeout" of five seconds to occur before the initiator starts a *fresh* handshake attempt, using the cookie as MAC key for the MAC2 field. This timeout will be described in more detail in Section 2.4.

A similar situation occurs when the initiator itself is "under load" (Figure 2.4b). After it receives a Handshake Response from the responder, the initiator could send a Cookie Reply instead of completing the handshake. When the responder receives this Cookie Reply, it will store the decrypted cookie. The initial handshake attempt remains unaffected. After a timeout, the Initiator will retry a handshake, this time it succeeds since the responder knows the appropriate cookie.

[1] The macro `MAX_QUEUED_INCOMING_HANDSHAKES` is defined as 4096 in the messages.h header file. This number is divided by 8 in receive.c in the "under load" check.



(a) Responder under load.

(b) Initiator under load.

Figure 2.4: Protocol flows where one party is under load and triggers a Cookie Reply. After the Cookie Reply and a delay, a new handshake is started.

An unfortunate situation could occur when both parties are under load. In that case, the first Handshake Initiation message triggers a Cookie Reply which will not complete the handshake. After the initial timeout, a new Handshake Initiation message with a valid MAC2 field is sent which will trigger a Handshake Response message without a valid MAC2 field. When the initiator receives this, it will send a Cookie Reply to the responder. The responder records this value, but won't reply with a handshake message. Only after the second timeout, both parties will have an appropriate cookie for the MAC2 field to continue the handshake. However, cookies have a limited lifetime of at most two minutes. In case the cookies have changed, then the above scenario could be repeated once more. If packets are lost, then this could take even more time as new handshake initiations can only be attempted every five seconds.

2.3.1 Handshake messages

The message format for the Handshake Initiation and Handshake response messages are displayed in Figure 2.5 and Figure 2.6 respectively. These

encapsulate handshake messages from the Noise IKpsk2 pattern which was described earlier in Section 2.2.1. This section will additionally define validity requirements for the encrypted handshake payloads.

Both handshake messages include a MAC1 and MAC2 field which use the keyed BLAKE2s MAC [50] with a 128-bit hash value as output. The input data and key depends on the recipient and field type:

MAC1 Let $S_{m'}^{pub}$ be the static public key of the receiver. The 256-bit MAC1 key is computed by hashing the concatenation of an eight-byte label with $S_{m'}^{pub}$ using an unkeyed BLAKE2s function: $\mathsf{Hash}(\texttt{"mac1----"} || S_{m'}^{pub})$. The data starts at the Type field and ends before the MAC1 field.

MAC2 If the most recent cookie was received from the peer within 120 seconds, then it will be used as 128-bit MAC2 key. If no such cookie has been received, or if it is too old, then the MAC2 *field* will be set to zeroes. The data starts at the Type field and ends before the MAC2 field.

The **initiator** creates a Noise handshake message as described in Section 2.2.1 (message from initiator to responder) with the following parameters:

- The initiator static key pair S_i^{priv}, S_i^{pub}.
- The responder static public key S_r^{pub}.
- A pre-shared symmetric key Q.
- The initial message payload: a 12-byte timestamp. The WireGuard paper recommends using the current time, encoded in the external TAI64N timestamp format [3] which has nanosecond precision. However, to avoid being used as time oracle in side-channel attacks, implementations should reduce the accuracy of the nanoseconds part by clearing some of the least significant bits. For example, the Linux kernel and Go implementations remove the 24 least significant bits, resulting in a precision of about 16.7 milliseconds.[2] Implementations can also

[2]The Linux kernel implementation limits the number of handshake initiations to 50 per second per peer as identified by their static public key. This implies that one handshake initiation is permitted every 20 milliseconds or $2^{24} + \ldots$ nanoseconds. Thus by truncating the last 24 bits, all valid handshake initiations will still contain a valid, increasing value. Note that the Go implementation has even stricter limits, even though it truncates 24 bits, it only accepts one initiation every 50 milliseconds.

choose to send any other monotonically increasing 96-bit unsigned integer value encoded as big-endian, it does not have to be a timestamp.

The Noise outputs (E_i^{pub}, encrypted S_i^{pub}, encrypted timestamp) are included in the Handshake Initiation message (Figure 2.5). A new randomly-generated local identifier is included as **Sender index**, the **MAC1** and (if necessary) **MAC2** fields are populated and the message is ready to be delivered to the responder.

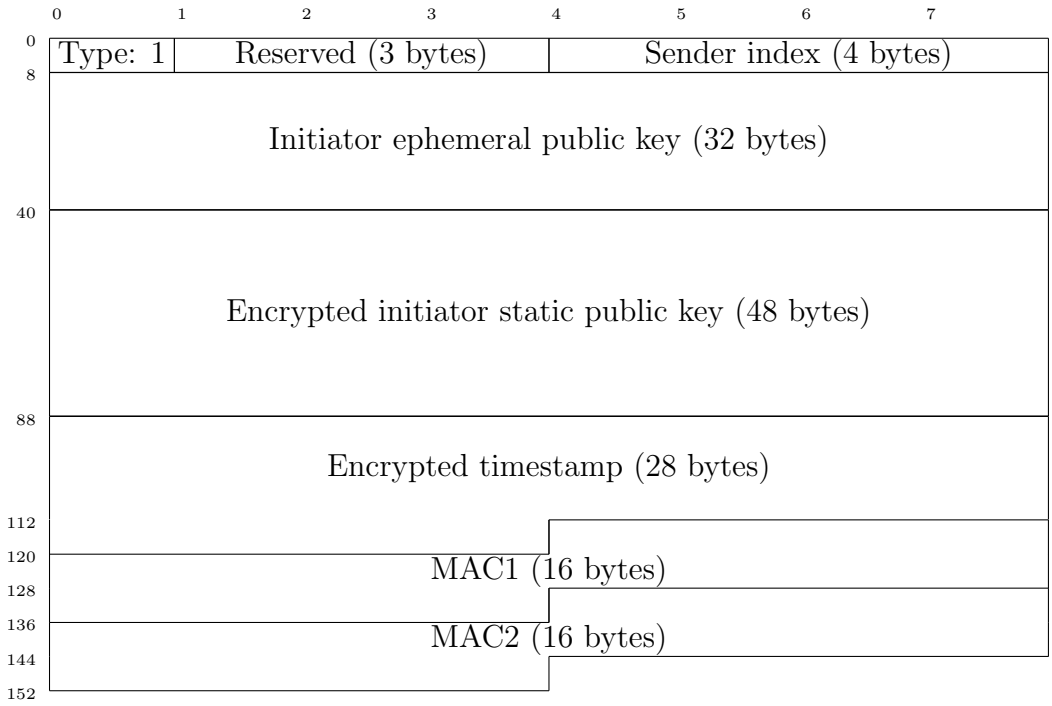


Figure 2.5: Handshake Initiation message format (148 bytes).

The **responder** receiving the Handshake Initiation message must perform the following steps and checks:

1. Verify the MAC1 field using S_r^{pub}. Abort on failure.
2. If "under load", check the MAC2 field:
    - Verify the MAC2 field. On failure, send a Cookie Reply (Section 2.3.2) and stop.

- On success, continue normal processing.

3. Process the Noise handshake message from the initiator to the responder as described in Section 2.2.1. Abort on decryption failure or if the decrypted payload, representing the timestamp, is not exactly 12 bytes. As an optimization, the Linux kernel implementation and wireguard-go perform the next two checks while processing decrypted fields and abort earlier on failure. However, implementations using a generic Noise library, such as wireguard-rs, will not be able to apply this optimization.

4. Lookup the peer configuration matching the decrypted S_i^{pub}. Abort if this identity is unknown.

5. If a previous handshake initiation for this client exists with a timestamp value larger than or equal to the decrypted timestamp, abort. In a comparison, these values must be treated as 96-bit unsigned big-endian integers since not all values can be decoded to a valid TAI64N label. For example, the nanoseconds part of a TAI64N label cannot be 10^9 [3].

The purpose of the timestamp field is to prevent attackers from disrupting an active session by replaying older sessions. If an initiator does not have a stable time reference or storage, it could potentially send a lower timestamp value after a reboot. This would constitute a protocol violation and the responder is allowed to ignore the handshake message. The responder may still accept the handshake message if it has reset its peer state, for example by rebooting. However, an attacker who has recorded previous handshake initiations with higher timestamp values could potentially prevent newer handshakes from being accepted by replaying an old handshake initiation. While the responder would send a handshake response, the initiator would not be able to process it. Any new handshake initiation from the initiator would also be dropped by the responder as long as the timestamp value is smaller than the replayed handshake initiation.

Once an initiation handshake message has successfully been processed, the responder will create a Handshake Response (Figure 2.6). It generates a Noise handshake message from the responder to the initiator as described in Section 2.2.1 and populates the **encrypted empty** field with the encrypted empty payload. A new randomly-generated local identifier is included as **Sender index** and the sender index from the Handshake Initiation message

is copied to the **Receiver index** field. Finally the **MAC1** and, if necessary, **MAC2** fields are populated as described before.

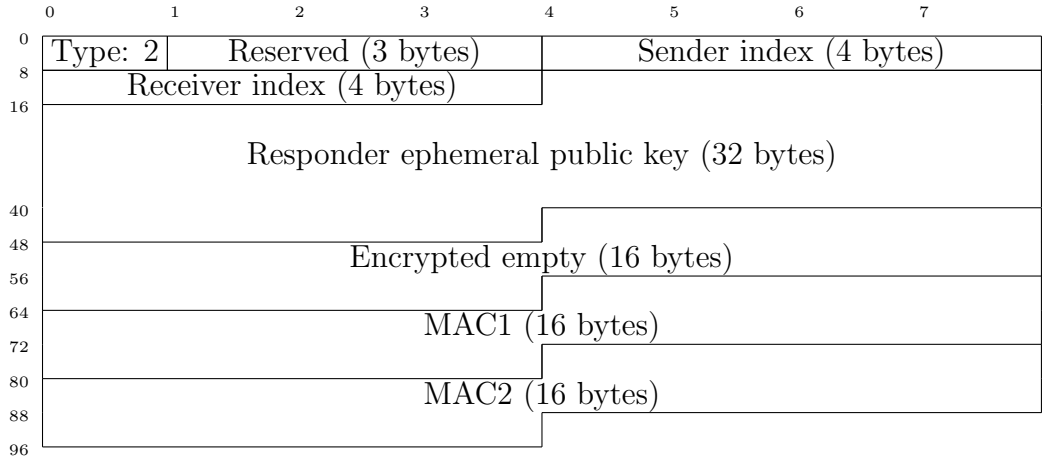


Figure 2.6: Handshake Response message format (92 bytes).

The **initiator** receiving the Handshake Response message must perform the following steps and checks:

1. Verify the MAC1 field using S_i^{pub}. Abort on failure.
2. If "under load", check the MAC2 field:
   - Verify the MAC2 field. On failure, send a Cookie Reply (Section 2.3.2) and stop.
   - On success, continue normal processing.
3. Look up the local handshake state matching the receiver index. Abort if not found or if the handshake has already been completed.
4. Process the Noise handshake message as described in Section 2.2.1 (message from responder to initiator). Abort on decryption failure or if the decrypted payload is non-empty.

At this point, the initiator has all necessary information to derive transport keys and complete the cryptographic handshake by sending an encrypted transport message to the responder.

2.3.2 Cookie Reply message

The MAC2 field described in Section 2.3.1 is only set if a recent cookie (not older than 120 seconds) is available. This cookie is transmitted in a Cookie Reply message (Figure 2.7). The Cookie Reply is sent in response to a handshake message when under load. It is smaller than the original handshake messages to avoid assisting in amplification attacks as will be explained in Section 5.1.

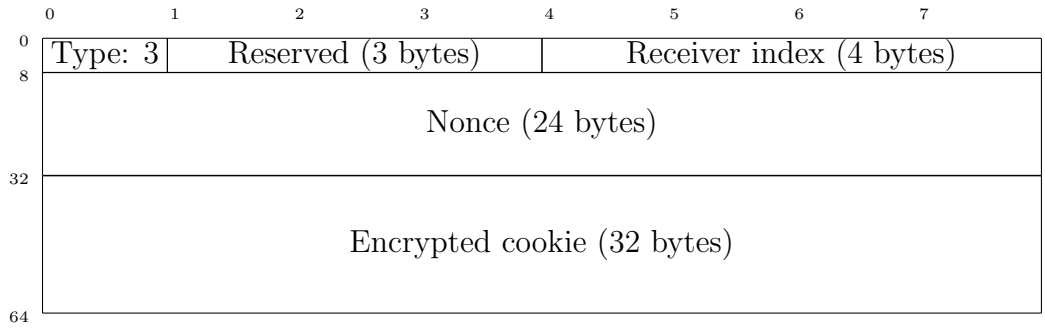


Figure 2.7: Cookie Reply message format (64 bytes).

The cookie is based on a local 256-bit secret value R_m that is changed to a new random value every two minutes, and the concatenation of the IP source address and UDP source port $A_{m'}$ of the original handshake message:

$$\mathsf{cookie} = \mathsf{BLAKE2s}(R_m, A_{m'})$$

The IP address is 32 bits for IPv4 and 128 bits for IPv6. The 16-bit port number is encoded in network byte order.

To avoid leaking the cookie in cleartext, it is encrypted using a symmetric key based on the unkeyed BLAKE2s hash over the concatenation of a 8-byte label and the local static public key S_m^{pub}:

$$\mathsf{cookie\text{-}key} = \mathsf{Hash}(\texttt{"cookie--"} || S_m^{pub})$$

Finally the cookie is encrypted with the XChaCha20Poly1305 AEAD which accepts a 32-byte key, a 24-byte nonce, arbitrary length plaintext, and arbitrary length additional authenticated data. Since the key cookie-key will be reused, a random nonce must be used. A counter could work as well, but that might leak the number of cookie replies to an attacker. The original

handshake message msg is used as additional authenticated data to protect the source against off-path attackers who cannot observe messages and who might try to send fraudulent cookies to disrupt the sender. [3] The encrypted cookie is calculated as follows:

$\mathsf{nonce} \in_R \{0,1\}^{192}$

$\mathsf{encrypted\text{-}cookie} = \mathsf{XChaCha20Poly1305}(\mathsf{cookie\text{-}key}, \mathsf{nonce}, \mathsf{cookie}, \mathsf{msg})$

The **Receiver index** is populated with the Sender index from the original handshake message. The **Nonce** and **Encrypted nonce** are as above.

The receiver of this message must look up the original handshake and associated static public key using **Receiver index**, decrypt the cookie and store the cookie in the handshake state. It must not immediately respond with a handshake message as the timers described in Section 2.4 handle retransmission of handshake messages. Future handshake messages will use the stored cookie to populate the MAC2 field.

2.3.3 Data message

Once a handshake has been completed, Transport Data messages (Figure 2.8) can be sent to transmit encrypted IP packets.

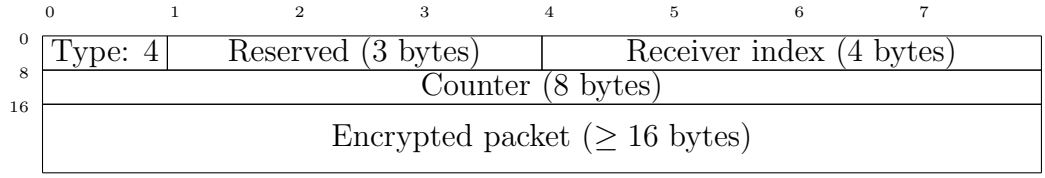


Figure 2.8: Data Message message format ($\geq$ 32 bytes).

Before encrypting an IP packet P, it is zero padded at the end to a multiple of 16 bytes, but never increasing the UDP packet length beyond the maximum transmission unit (MTU). A zero-length packet is treated as Keepalive message, one that does not transmit a IP packet but merely signals that a WireGuard tunnel must be kept active.

Every peer maintains a local 64-bit nonce N_m^{send} starting at zero right after the handshake and incrementing after encryption. The nonce is encoded as

[3] An attacker with knowledge of the static public key could observe messages and compute a fraudulent cookie regardless. An on-path attacker could simply drop traffic to cause a denial of service.

little-endian integer in the **counter** field. Messages may only be sent only if the messages limit has not been reached yet ($N_m^{send} < 2^{64} - 2^4 - 1$) and if the session has not expired yet (see Chapter 2.4).

Encryption uses the same ChaCha20-Poly1305 AEAD as used for handshake encryption. The transport key was established in the handshake, and the additional authenticated data is empty:

$P_{padded} = P \mid\mid 0^{16 \cdot \lceil \frac{\|P\|}{16} \rceil - \|P\|}$

$\mathsf{encrypted\text{-}packet} = \mathsf{aead\text{-}enc}(T_m^{send}, N_m^{send}, P_{padded}, \epsilon)$

Since UDP does not preserve order nor does it guarantee delivery, receivers of the Transport Data message must proceed carefully. They must keep track of previously received counters and drop messages with reused counter values.

The transport data message is variable length and at minimum 32 bytes. The practical maximum length of tunneled IP packets over the public Internet is 1420 bytes as will be described later in Section 2.3.3.

Congestion control prevents networks from becoming overwhelmed by reducing packet rate. WireGuard has no form of congestion control, this is a responsibility for upper layers. Protocols such as TCP and QUIC do rely on the IP layer to provide an Explicit Congestion Notification (ECN) [28] through two bits. If a transport protocol supports ECN, it will set one of the two bits. Routers that detect congestion may then set forward packets and set both bits to signal Congestion Encountered (CE) instead of dropping the packet. This mechanism enables protocols to explicitly detect congestion rather than observing it as a result of packet loss. While the wireguard-go implementation does not support ECN, the Linux kernel implementation does propagate the CE signal from the outer IP packet to the inner IP packet for ingress traffic.

Maximum packet lengths

The theoretical maximum sizes are dependent on technical limits such as the UDP header size and support for Jumbograms. Jumbograms increase the maximum possible length of UDP datagrams by replacing the 16-bit length field in the UDP header by a 32-bit Jumbo Payload Length field in the IPv6 Hop-by-Hop Options header. In Table 2.2, the theoretical maximum values are displayed. In practice, however, the MTU of the physical network link is much smaller. Additionally, platforms might restrict the theoretical limits

too. For example, Linux and FreeBSD do not support IPv6 Jumbograms [27, 29].

Payload size	Protocol description
$2^{16} - 1 - 8$	IPv4 with fragmentation (limited by UDP Length)
$2^{16} - 1 - 20 - 8$	IPv4 without frag. nor IP options (limited by Total Length)
$2^{16} - 1 - 8$	IPv6 without extension headers (limited by Payload Length)
$2^{32} - 1 - 8$	IPv6 with Jumbograms (limited by Jumbo Payload Length)

Table 2.2: Theoretical maximum sizes for UDP datagram payloads without the IP or the 8-byte UDP headers, based on transport header field limits [1].

Aside from MTU restrictions on the directly attached network link, additional limits may apply along the path between two peers. This is called the path maximum transmission unit (PMTU). One of the reasons for a lower PMTU are tunneling protocols that add extra protocol headers such as WireGuard.

As for practical minimum message lengths on the Internet, IPv6 requires a minimum link MTU of 1280 bytes without Path MTU Discovery (PMTUD) [12, Section 5]. Assuming an IPv6 header without extension headers, this implies a minimum UDP payload size of 1232 bytes. Thus, WireGuard should usually be able to send IP packets of at least 1200 bytes due to an overhead of 80 bytes:

- 40 bytes for an IPv6 header without extension headers (20 bytes for IPv4 without options).
- 8 bytes for the UDP header.
- 32 bytes for the WireGuard message header: 4 bytes for the type and a reserved field, 4 bytes for the receiver index, 8 bytes for the message counter, and finally 16 bytes for the authentication tag.

The exact limit depends on whether WireGuard is encapsulated in other tunneling protocols. If the chosen MTU results in too large IP packets, any hop between two peers might try to fragment it into smaller IP packets. The minimum supported reassembled IP packet size is 1500, but relying on fragmentation is “discouraged in any application that is able to adjust its

packets to fit the measured path MTU" [12]. However as many links on the public Internet use Ethernet or similar, a MTU 1500 is generally safe [44] which implies a MTU of 1420 bytes inside a WireGuard tunnel.

Selection of an appropriate MTU is important, a too low value would incur more per-packet overhead and might reduce throughput. An overly high value would prevent packets from being delivered across some links or cause unpredictable behavior due to IP fragmentation. Neither the WireGuard protocol nor the Linux kernel implementation have a built-in mechanism to automatically configure an appropriate MTU. This is a manual process, but scripts such as wg-quick do select a default MTU based on the link MTU.

2.4 Timers

The WireGuard protocol operation is affected by several time-related constraints. The involved constants are summarized below:

Symbol	Value
Rekey-After-Messages	$2^{64} - 2^{16} - 1$ messages
Reject-After-Messages	$2^{64} - 2^{4} - 1$ messages
Rekey-After-Time	120 seconds
Reject-After-Time	180 seconds
Rekey-Attempt-Time	90 seconds
Rekey-Timeout	5 seconds
Keepalive-Timeout	10 seconds

A single host could be configured to accept multiple other peers. All of the following rules apply on a per-peer basis related to handshake initiation:

1. A handshake initiation may only be sent once every Rekey-Timeout. In the following rules, queuing a handshake initiation means sending a handshake initiation message if none was sent in the last Rekey-Timeout.

2. A handshake initiation which has not been matched with a valid handshake response will be retried with fresh ephemeral keys until Rekey-Attempt-Time has passed. This timer will be reset when new transport data is added to the outbound queue. After that, the handshake is considered failed and the outbound queue is flushed.

3. The age of a secure session is measured from the moment when the handshake has completed. For the initiator, this means when the handshake response has been processed and transport keys have been derived. For the responder, it means when the first valid transport data message from the initiator has been received.

4. Timer expiration on its own must not trigger a new handshake initiation, only transport data can trigger it. This ensures that a tunnel is not unnecessarily kept alive.

5. If a session is older than Reject-After-Time, it must not be used.

6. After sending Rekey-After-Messages, a handshake initiation message may be queued.

7. If more than Reject-After-Messages have been sent in the current session, it must no longer be used. Similarly, a receiver must reject all transport data messages with a counter matching or exceeding this value.

8. If an **initiator** sends data and Rekey-After-Time has passed, a new handshake initiation may be queued.

9. If an **initiator** receives data and Rekey-After-Time − Keepalive-Timeout − Rekey-Timeout has passed, a new handshake initiation may be queued.

10. If a **responder** receives data and Rekey-After-Time − Keepalive-Timeout has passed, a new handshake initiation may be queued.

Only the initiator may trigger rekeying while a session is valid to avoid a "thundering herd" [18] problem where the both the initiator and responder try to start a new session at the same time.

In order to quickly identify whether a tunnel is still alive, WireGuard implements a *Passive Keepalive* mechanism:

1. If a transport data message is received, and no data has been sent in the past Keepalive-Timeout, send a keepalive message if either:

   (a) The received transport data message is not a Keepalive.

   (b) The received transport data message is a Keepalive and the most recently sent transport data message is not a Keepalive.

Otherwise, if the most recently sent two messages were Keepalives, no passive keepalive message should be sent.

2. If a transport data or keepalive message was sent, but no data has been received in the past Keepalive-Timeout + Rekey-Timeout, consider the link dead and start a new handshake initiation. The session itself is not yet destroyed, WireGuard session keys are only destroyed after a fixed time to permit decryption of out-of-order packets.

Aside from the passive keepalive mechanism, the Linux kernel and wireguard-go implementations can also be configured to send a *Persistent Keepalive*. This will result in a Keepalive message whenever a tunnel has been idle for a configurable amount of time.

Network Address Translation (NAT) is a mechanism where routers between a private and public network change addresses of IP packets. As a single public IP address is typically shared between the many private IP addresses, the UDP source port will also change to avoid collisions. As router memory is constrained, these mappings are typically only maintained for a limited time. WireGuard's keepalive mechanism is intended to keep NAT sessions active to avoid NAT rebinding, a phenomenon where a host behind a NAT router becomes unreachable from the outside, and where this host changes the source port once it start transmitting data again. We observe that while this setting can be configured in the Linux kernel to 65535 seconds (the maximum of a 16-bit integer), it will not keep the tunnel alive in this full time period. Since the maximum session age is 180 seconds, any higher persistent keepalive setting will result in a window where the tunnel is not active.

2.5 Key rotation

Approximately every Rekey-After-Time, a new session is established which changes the session keys for transport data. While old sessions must be destroyed to ensure forward secrecy, it is possible that existing transport data messages were already in transit while a handshake is being completed. These messages must be decipherable, and therefore old sessions cannot immediately be erased. To facilitate this, WireGuard maintains three session slots per peer.

At any point in time, at most three session key slots are available, the "previous", "current", and "next" session. Initially no session exists between two peers and thus all slots are empty. Outbound transport data messages will always be encrypted with keys from the "current" key slot. Inbound transport data messages can be decrypted with keys from either the "current" or "previous" slot as identified by the receiver index.

When an initiator sends a handshake response, it has completed the handshake and can derive session keys which will be stored in "current". The previous recent session (either "next" or "current") will be moved to "previous" and "next" will be emptied. Note that these slots store *session keys* and not incomplete sessions which are instantiated by handshake initiations.

When a responder receives a handshake response, it still needs to wait for a confirmation even if it can derive session keys. These keys will be stored in "next". Once a handshake has been confirmed by receipt of the first transport data message, slots rotate: the "next" session moves to "current", and the "current" slot moves to the "previous" slot. If the "previous" slot was non-empty before rotation, it must be properly destroyed. Likewise, sessions that reach Reject-After-Time must be removed from its slot and destroyed.

In the unfortunate event that two peers decide to start a handshake with each other at exactly the same time, this behavior is not defined in the current whitepaper. In the current Linux kernel implementation, each peer has a single handshake state. When one side begins an initiation, it initializes this handshake state for the initiator role and only accepts a responder message. However, when it receives a valid handshake initiation message from the peer, it will unconditionally overwrite its own pending handshake initiation state with a new state based on the received handshake initiation. If the same happens to the other side, then both peers end up with an unusable handshake state which cannot be completed since they lack their own original handshake initiation state. We will explore this scenario in Section 5.2.

2.6 Roaming

The external source IP address is used by WireGuard to identify the endpoint address to which messages should be sent. This address can change for various reasons, including moving between Wi-Fi and mobile networks. WireGuard supports maintaining an active session even under these conditions. It does so by updating the endpoint address of the peer from which

the change was observed. It does not restart the handshake after roaming, presumably because that would increase latency before data can be transmitted.

The external source IP address and UDP port for the most *recent* authenticated handshake or transport message is always used by the receiver to determine the destination for future responses. At least the Linux kernel implementation always uses the most recently seen source address, disregarding the counter in transport data messages for example. Changes to the source address do not have other side-effects such as triggering a new handshake.

We observe that cookie reply messages are not consistently handled like other messages. The source address of cookie replies are ignored, at least in the Linux kernel implementation. Peers that receive a cookie reply should try to avoid updating their own source address for the next handshake message as the cookie within the cookie reply is bound to the source IP address of the original message.

While the current mechanism to support roaming is fast, it does contain a denial-of-service vulnerability which will be described later in Section 5.1.

2.7 Handshake state machine

In this section, we will describe a state machine that we identified based on the protocol description and the Linux and Go implementations.

Recall that a single WireGuard device is uniquely identified by a static public key, and that multiple peers with different public keys might be attached to this device. Due to key rotation (Section 2.5), a single peer on a device might have multiple sessions active at any time. Each WireGuard device maintains one set of receiver indices (Section 2.3) which is a bijective mapping to some session state.

The session state includes the handshake state, the transport data counters, transport data anti-replay counters, various timer-related timestamps, and so on. We will describe a state machine that covers the handshake state for one such session. We assume that the receiver index does not change for the lifetime of a session, a handshake message will always be interpreted by one handshake state machine only.

The state machine in Figure 2.9 defines the following states:

Idle The initial state where no handshake initiation has been exchanged yet for this receiver index. Therefore no session is present.

ISent The handshake initiation message was previously sent to the peer, so the local party assumes the initiator role.

IRcvd A handshake initiation message was previously received from the peer, so the local party assumes the responder role.

RSent A handshake response message has been sent to the initiator and is now pending confirmation from the initiator.

RRcvd The local party is the initiator and receives a handshake response message.

Alive The handshake process is complete, transport data may be sent and received.

Dead The terminal state where the session has exceeded its maximum age and can be destroyed.

Transitions are generally channel-related events, *recv Init* means that a handshake initiation message was received while *send Resp* means that a handshake response message is sent. The *expires* transition is a time-related event and indicates that the completed handshake state is older than Reject-After-Time.

In this state machine, we assume that the messages are properly formed. In particular, we assume that counters do not increase past Reject-After-Messages, AEAD decryption passes, MAC1 and MAC2 are valid, the handshake initiation timestamp has been validated, and so on. Since cookie reply messages do not directly affect the handshake state, and only affect the session state, cookie replies are not included as transition either.

Data messages are omitted from earlier states as decryption is not possible without a completed handshake. In those cases, implementations could either discard them or buffer messages if they would like to account for out-of-order packets. The implementation is assumed to be valid, so we omit transitions that send messages from unexpected states.

Obviously the handshake response transitions are only possible if their receiver index matches the sender index of the original handshake initiation. Otherwise a receiver would not be able to find a matching handshake state and the handshake response message would be discarded with no transitions. Likewise for data messages, a receiver would not be able to find matching session keys and thus these messages would be dropped.

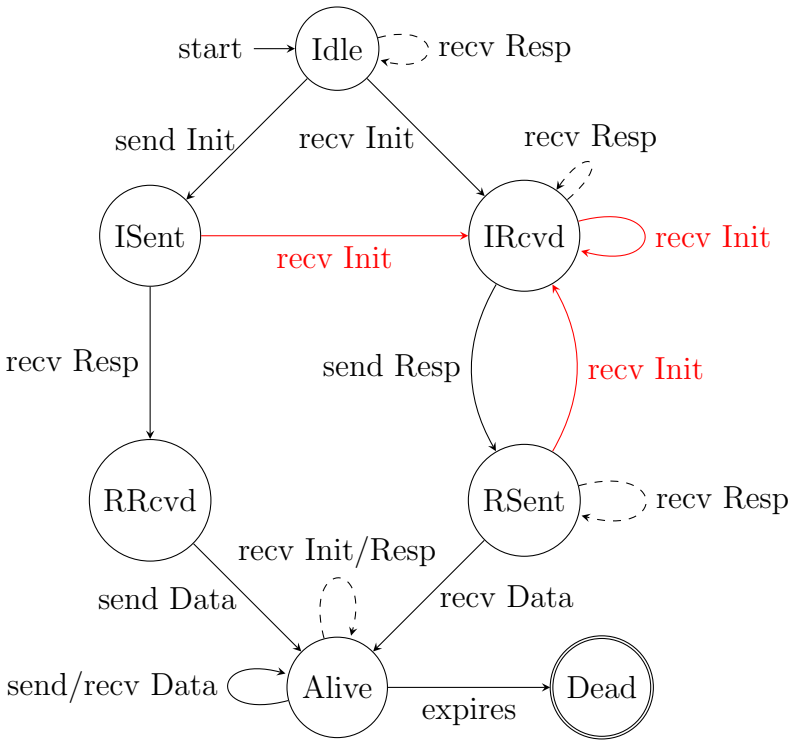


Figure 2.9: Handshake state machine. Dashed transitions are illegal, but implementations must properly handle them by discarding the messages. Red transitions are not defined in the whitepaper, but exist in the Linux and Go implementations.

An implementation must be prepared to receive a handshake message at any time and properly handle it. Responders must ignore any handshake response for an existing session, this is depicted with a dashed arrow. We observed that the whitepaper does not document the expected behavior when a peer receives multiple handshake initiation messages, for example when a handshake retransmission occurs. Since a handshake initiation can be identified by its unique sender index, a peer could in theory maintain multiple pending handshake states for both the initiator or responder role. In practice, both the Linux and wireguard-go implementations only maintain the most recent handshake initiation (see also Section 5.2 for a side-effect). In the state machine, this is depicted with a red receive Initiation (*recv Init*) transition.

Data can only be sent when the handshake is confirmed. Hence no *send*

Data transition exists in the Response Sent (*RSent*) state. Likewise, a Receive Data (*recv Data*) transition is forbidden from the Response Received (*RRcvd*) state.

Finally, when the session is alive for Reject-After-Time, it must no longer be used (it enters the *Dead* state). That marks the end of one handshake state.

Chapter 3

Related work

Various aspects of WireGuard have previously been studied with formal analysis. In this chapter we will describe their scope and summarize their results.

Noise Explorer by Kobeissi and Bhargavan [37] enables automated symbolic modelling and verification for Noise protocols. Their symbolic security analysis is based on ProVerif and covers 57 different Noise handshake patterns including the IKpsk2 handshake pattern used by WireGuard. Specifically for this IKpsk2 pattern, the following security properties from the Noise specification [47] were formally verified:

- Authentication: "Sender and receiver authentication resistant to key-compromise impersonation (KCI)". A protocol is secure against KCI attacks if an attacker who has compromised the victim's key cannot impersonate any other party to the victm [41].

- Confidentiality: "Encryption to a known recipient, strong forward secrecy". The symmetric encryption key is derived from an ECDH computation involving the recipient's static key pair as well as an ECDH computation using ephemeral key pairs from both parties.

Donenfeld and Milner [25] have developed a symbolic model of the WireGuard handshake protocol using Tamarin and have proven properties such as key agreement and correctness, key secrecy, forward secrecy, session uniqueness, and identity hiding. Identity hiding will be discussed later in Section 5.3. Since session keys are derived from identity keys, it is not possible to compute the same session key between different peers and therefore unknown key-share attack resistance is also proven.

Donenfeld and Milner [25] also briefly describe resistance against KCI attacks, we will elaborate on that. In WireGuard, knowledge of the initiator static private key does not allow an attacker to impersonate any responder to the initiator since the attacker would need either the responder static private key or initiator ephemeral private key to derive session keys (see `"es"` in Section 2.2.1). Knowledge of the responder static private key permits an attacker to create a handshake initiation message that can impersonate any initiator to the responder, but the attacker will not be able to compute the actual session keys as that requires either the initiator static private key or the responder ephemeral private key (see `"se"` in Section 2.2.1). Since the initial handshake message can be crafted by an attacker, the responder must only consider a handshake authenticated when it receives a message that proves the initiator's ability to derive session keys. This is done in WireGuard via the initial transport data message from the initiator to the responder. Hence WireGuard is secure against KCI attacks if both static private keys are not compromised at the same time.

Dowling and Paterson [26] describe a computational proof on a modified version of WireGuard. They were not able to prove the original WireGuard protocol secure with respect to key indistinguishability since session keys are no longer indistinguishable from random. To remedy this, they could either model the handshake and data transport as a monolithic whole, or modify the protocol to be proven. They have opted for the latter and inserted an additional message at the end of the handshake that proves authenticity without requiring a data message that uses session keys. With this modification, they were able to prove key indistinguishability security in their security model in addition to properties such as perfect forward secrecy and resilience against KCI attacks. This analysis excludes cookie messages and key rotation.

Lipp [42] presented a mechanised computational proof using CryptoVerif covering properties such as correctness, message secrecy, forward secrecy, mutual authentication, resistance against KCI, resistance against unknown key-share attacks, and resistance against replay of the first protocol message.

All these related works have focused on a theoretical model of the WireGuard protocol. Rather than replicating the work on symbolic and computational models, we have instead focused on developing a complete specification of the WireGuard *protocol* suitable for implementers (Chapter 2). In addition, we include an analysis on practical aspects of WireGuard *implementations*.

Chapter 4

Software

In order to facilitate protocol analysis we have developed several tools. These will be described in this chapter.

4.1 Wireshark Lua dissector

Wireshark [30] is an open-source network protocol analyzer which can capture network traffic and dissect the individual bytes. We created a Lua dissector plugin [56] to support analysis of the WireGuard protocol.

This dissector is able to show the basic structure of protocol messages and display the contents of decrypted fields when AEAD secrets are provided via a *keylog file*. An example of its operation is shown in Figure 4.1.

The *AEAD secrets* secrets required for decryption refer to a pair consisting of a 32-byte symmetric key and 32-byte authenticated additional data. One advantage of the Lua dissector accepting symmetric encryption secrets is that keys can be selectively shared. For example, one could share keys to decrypt the handshake while maintaining confidentiality of the transport data messages. Or alternatively, one could share transport data keys without revealing their identity through the static public key.

Our Lua dissector has over 300 source lines of code (SLOC). This includes code to parse the key log file, dissection of protocol structures, and glue code to support decryption. The source code and documentation for the dissector (wg.lua), sample capture files, and keylog files can be found in the author's Git repository [56].

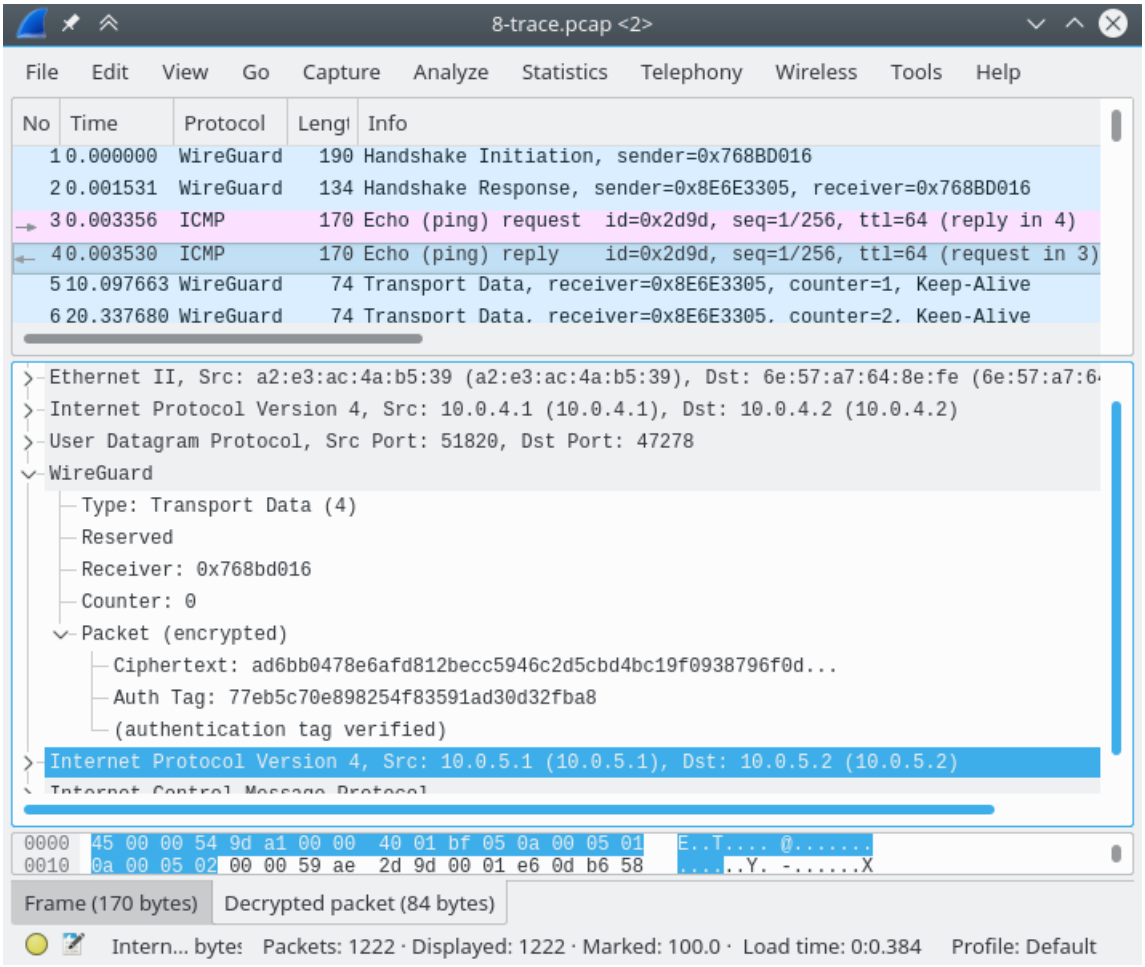


Figure 4.1: A screenshot of Wireshark, showing the WireGuard Lua dissector.

4.1.1 Key extraction

In order to extract the AEAD secrets from a Linux system, one could patch the WireGuard source code and expose the keys. However, that could increase the risk of key exposure and crashes depending on the implementation. It would also not work on a system where the kernel module cannot be replaced, or where doing so requires a reboot. To enable key extraction on a live system, we used kprobe-based event tracing [33] to dump kernel memory when certain carefully chosen functions are entered. This mechanism makes it possible to write one line with register and memory contents to a special file (`/sys/kernel/debug/tracing/trace_pipe`).

We ended up hooking into the `chacha20poly1305_encrypt` and `chacha20poly1305_decrypt` functions in order to dump the key and AAD for fields in handshake messages. Luckily, transport data messages do not use these functions and thus we can prevent the log from growing fast when there are many data messages. Recall that the symmetric keys for transport data are derived using the HKDF function which involves multiple calls to a HMAC function. While HKDF is called multiple times, only the final call uses an empty input parameter. Therefore we hooked into the `blake2s_hmac` function and extracted the parameters passed to the HKDF function. To identify the final call, we distinguished calls based on the input size. To link keys to a session, we dumped the sender and receiver indexes by hooking into the function that inserted new sessions into a hash table using the sender index as hash table key. To match the initiator with the responder, we hook into the handshake response processing function. Finally a post-processing script (key-extract.py) correlates all logged information and produces a key log file which can be fed to Wireshark to enable decryption.

4.2 Wireshark C dissector

While this initial Lua implementation was able to fully decrypt and interpret every field, except for cookies, it would not have been able to verify implementations for protocol conformance as decryption is completely isolated from the handshake. Therefore we took a new approach when developing a new C implementation [57]. Instead of relying on external sources to provide symmetric keys to enable decryption, we now follow the calculations required for the Noise protocol and derive the symmetric keys. This new WireGuard

dissector has been integrated in Wireshark 3.0 and thus Wireshark now recognizes the WireGuard protocol out-of-the-box.

Deriving symmetric keys using the Noise protocol requires static and ephemeral private keys, and an optional PSK. We extended Wireshark such that these keys can be provided via either a *key log file* as configured in the protocol preferences, or via a decryption secrets block within the pcapng file [59]. The original kprobe method extracted symmetric keys that required a session identifier for linking. As asymmetric keys can be linked to a public key, the kprobe post-processing script can be greatly simplified. The simplified script to extract keys from the Linux kernel is *extract-handshakes.sh* [17].

The earlier Lua dissector looks up decryption secrets based on the sender or receiver indices. This is stateless in the sense that individual packets can be processed independently, but it is vulnerable to receiver index collisions. The new C dissector instead has to perform the full handshake computations which requires linking messages to a session. A naive implementation could use the IP addresses and port numbers of the outer IP packet, but this information could change due to roaming. For this reason, we maintain state to link handshake, cookie replies, and data messages to a session. Connection tracking survives IP address changes and depends on the received message. The dissector recognizes the following message types:

- Handshake Initiation: create a new unlinked session, storing the source address and port as initiator address, while storing the destination address and port to quickly identify potential responders. The new session is added to a list and identified by the sender index.

- Handshake Response: look up a previous unlinked session based on the receiver index. If the session was not previously linked to a handshake, and the source address of this response matches the destination address of the initiation message, the handshake computations will complete and data decryption will be possible from this point on.

- Cookie Reply: similar to the Handshake Response, the corresponding session is looked up based on the receiver index.

- Transport Data: look up the most recent session based on the receiver index. If the handshake has been completed, decryption can be enabled using the corresponding handshake secrets.

Actual implementations such as the Linux kernel implementation typically use the receiver index to identify the associated session. Since Wireshark is not a real initiator or responder, it does not know the exact session that belongs to a receiver or sender index. As implementations could generate colliding indices, potentially due to malice, it is important that a passive observer such as Wireshark can accurately model the session associations. Hence it makes assumptions such as a match between the handshake response destination address and the handshake initiation source address.

Unlike for the Lua dissector, a full implementation of the WireGuard handshake is required in order to compute the appropriate decryption keys for the static initiator public key, the timestamp field, the empty responder confirmation field, and transport data. For this we mostly relied on Libgcrypt [39], a general purpose cryptographic library. This includes implementations for BLAKE2s, ChaCha20-Poly1305, HMAC, and HKDF via existing glue code in Wireshark. Since X25519 is not readily available in Libgcrypt, we had to create our own implementation, see Section 4.2.1.

Our Wireshark C dissector (packet-wireguard.c) has over 1300 SLOC. This includes code for dissection of protocol structures, connection tracking, an implementation of the WireGuard handshake protocol, code to parse the key log file, an implementation of our identity hiding attack (Section 5.3), and code to support decryption. Our X25519 glue code has 66 SLOC while the test suite gained about 200 SLOC. The related changes can be found in the Wireshark bug tracker [57].

4.2.1 X25519 implementation

Wireshark already uses Libgcrypt for its cryptographic needs, so it would have been nice if this library could be reused for the X25519 DH function which is necessary to derive keys during the handshake. Unfortunately, the current version of Libgcrypt (1.8.4) does not provide a convenient interface. A new interface is however planned for the next 1.9.0 release [38].

Given the lack of a standard Libgcrypt interface, we did consider several others to provide the simple $\mathsf{X25519}(\mathsf{sk}, \mathsf{pk})$ interface:

- Sodium [14]: a portable fork of NaCl [6]. The `crypto_scalarmult` function directly implements the required interface using optimized assembly code. While the interface is great, it would add a new dependency for Wireshark which increases packaging requirements and needs

a justification.

- TweetNaCl [7]: a small reimplementation of NaCl fitting in 100 tweets of 140 characters. The `crypto_scalarmult` function is implemented in C as opposed to optimized assembly code. While fuzzing this function using LibFuzzer [48] and Undefined Behavior Sanitizer (UBSan), we found undefined behavior (left shift of a signed negative number) in the internal car25519 function affecting the last line, `o[i]-=c<<16`. For us, this is a reason not to use this implementation in Wireshark. See below for more discussion. The full TweetNaCl C code has about 700 SLOC. A stripped down subset with just the `crypto_scalarmult` implementation provided by Jason A. Donenfeld has about 130 SLOC. We did not end up including either code with Wireshark since the project has been trying to move away from bundled crypto code, and instead reuse existing external cryptographic libraries.

- Libgcrypt [39]: although no direct implementation exists, we found three ways to indirectly compute X25519:

  - High-level gcry_pk_decrypt function.
  - High-level gcry_pk_encrypt function.
  - Low-level multi-precision-integers (MPI) API.

  Both high-level functions require significant preprocessing of the inputs (Curve25519 secret and public keys) and outputs (the shared secret), and were the slowest among the tested implementations. For these reasons, we stopped considering these two variants. The low-level MPI API still requires conversion of the inputs, but is the best alternative to the high-level variants. Internally this API uses generic elliptic curve point routines which are not specific to Curve25519.

We evaluated the above implementations [55] and found that the Sodium implementation was the fastest which was unsurprising due to its optimized assembly implementation. The TweetNaCl implementation was 18x slower at optimization level -O3, and 38x slower at optimization level -O2. The MPI-based implementation is slightly slower than TweetNaCl at -O2.

The final Wireshark implementation (wsutil/curve25519.c) uses the MPI-based implementation which has 66 SLOC and has no new library dependencies aside from Libgcrypt 1.7.0.

All of the above X25519 implementations have been fuzz-tested using LibFuzzer. Address Sanitizer (ASAN) was enabled to check for memory safety issues and UBSan was enabled to catch potential undefined behavior. Only TweetNaCl triggered UBSan issues as described above. The C standard declares left shift of a signed negative integer as undefined behavior, and since compilers could turn code affected by undefined behavior into defective programs [10], we have adopted a policy to avoid such code. This makes it easier to reason about correctness of a program on various platforms supported by Wireshark.

The final implementation in Wireshark has also been tested against the reference implementation using LibFuzzer. LibFuzzer is an in-process, coverage-guided, evolutionary fuzzing engine [48]. It calls a target function, `LLVMFuzzerTestOneInput`, with random input and mutates this input in order to maximize code coverage. In our target function, we partitioned the input into a 32-byte private key and a 32-byte public key. The fuzzer ran for a day, and we observed that all outputs from our `crypto_scalarmult_curve25519` and `crypto_scalarmult_curve25519_base` functions produce exactly the same output as corresponding reference implementations in Sodium. This increases the confidence in the correctness of our X25519 implementation.

4.2.2 Wireshark dissector features

Our new Wireshark dissector has several features, including:

Heuristics dissection Wireshark is able to detect the WireGuard protocol with little effort due to its unique format. The current heuristics assume WireGuard when the first byte is 1, 2, 3, or 4, and when the next three reserved bytes are zero. If this turns out to yield false positives, then the UDP payload length could be used as an additional selector. At least one commercial vendor has also added WireGuard detection to their deep packet inspection (DPI) software library [34], so we speculate that some networks could block WireGuard in the future.

Identity confirmation As will be described later in Section 5.3, receiver identities for honest senders can be trivially confirmed by a passive observer. When static public keys are configured in Wireshark, it will use these to probe the MAC1 field and display the matching identity. See Figure 4.2.

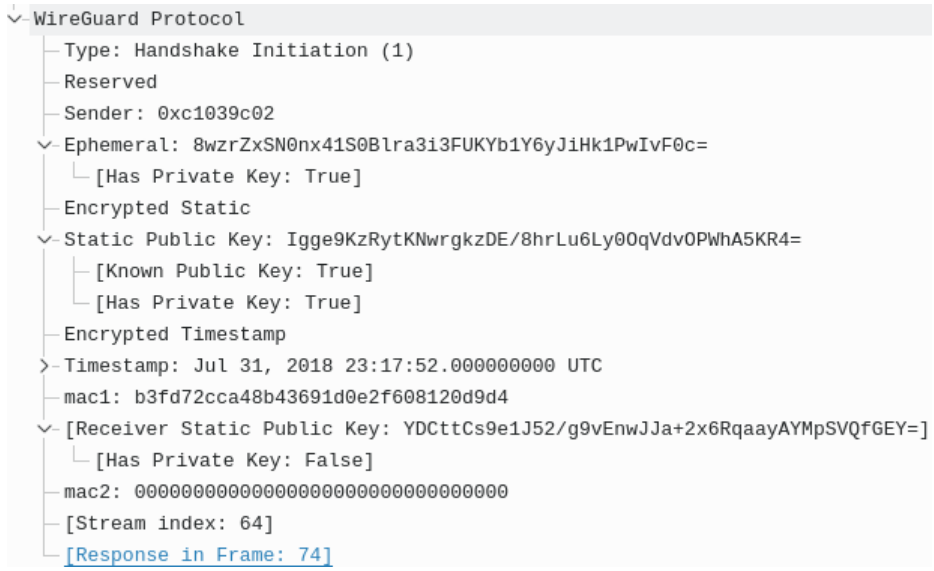


Figure 4.2: A screenshot of Wireshark, revealing successful decryption of the initiator static key and the timestamp. The receiver public key was also identified.

Key matching In conventional WireGuard implementations, the static public and private keys must be configured for the right endpoint or interface section. As a passive observer however, Wireshark can read the ephemeral public key and decrypted initiator static public key from the wire and match them against the set of known keys. It does not need an additional identifier, such as the sender index in the Lua dissector.

Decryption All handshake and transport messages can be decrypted when appropriate keys are extracted from the endpoint. The dissector automatically derives all symmetric keys. If decryption fails, Wireshark will report this. This greatly helps in development of new WireGuard implementations.

Decryption using embedded keys The traditional pcap format as produced by the tcpdump program can only store packet data. The pcapng packet capture format can additionally store information such as name resolution results, interface statistics, packet comments, and decryption secrets. We have extended the pcapng specification [59] with support for embedding WireGuard decryption secrets and extended Wireshark

to recognize this new format. This enables decryption of WireGuard packet captures without additional configuration.

Keepalive messages Keepalive messages are sent in transport data messages, but since their fixed length acts as selector, Wireshark is able to report a Keepalive message as opposed to a transport data message. Additionally, decryption failures will also be reported when detected.

Connection tracking Wireshark usually tracks protocol flows based on a match of source and destination addresses and ports. In WireGuard, the sender and receiver indices provide a way to track address changes. Together with confirmation via successful decryption, Wireshark is able to reliably track flows. In Figure 4.2, one can recognize the *Stream index* field which provides a unique identifier to track a session. Additionally, the *Response in Frame* field links to a legitimate handshake response that successfully completed the handshake.

4.3 Low-level prototyping tool for WireGuard

To enable experimentation with the WireGuard protocol without being constrained by implementation choices that increase security or conformance to the protocol, we have developed a WireGuard implementation for low-level prototyping, wgll [60]. This tool is written in the Python 3 programming language and implements the full WireGuard handshake. It is able to establish a new session, send encrypted data, and decrypt received data.

Conventional implementations usually generate ephemeral keys using a random number source and enforce a particular messages order. Our wgll tool instead provides granular primitives to build, send, receive, and parse one of the four protocol messages. These primitives are state transitions that take an immutable state and produces the next state. For example, a *Process and initiation message* action takes the 148-byte initiation handshake message and a structure with three keys, and produces a new state with the decrypted fields and session keys for transport data encryption.

We limited states to externally observable behavior. For example, while handshake initiation processing could be split off after decryption results are written to the wire, we felt that this level of detail was unnecessary.

In our implementation, the ephemeral private key could be fixed as opposed to random. Semantics of decrypted fields are not validated. For ex-

ample, the decrypted timestamp field is not checked against replay which is required for conforming implementations. This kind of freedom ensures that a reproducible *test scenario* can be written to test real implementations against edge cases. It is also a reason why wgll is not suitable for production use, it should only be used for testing purposes.

The wgll tool consists of a core component that defines state and manages state transitions (states.py). This includes computations for the cryptographic handshake protocol (noise_wg.py). This library can be used to develop new tests in Python, but for more concise test specifications we have developed a small domain-specific language (DSL) that is interpreted by the wgll.py script.

We have used wgll to verify a specification and implementation issue related to key rotation (Section 2.5). Our tool could potentially be adapted to create protocol conformance tests in the future, for example to verify anti-replay functionality.

Our low-level prototyping tool for WireGuard consists of 585 SLOC. The source code and example scenarios can be found in the author's Git repository [60].

Chapter 5

Protocol weaknesses and countermeasures

Based on the analysis of the protocol description and implementation in the previous chapters, this chapter will describe some potential issues and suggestions for countermeasures.

5.1 Amplification attacks

An amplification attack is a denial-of-service attack against a victim where an attacker sends a small packet to an unsuspecting third party in order to elicit a larger response to the victim. UDP protocols are especially vulnerable due to the connectionless nature of UDP. In networks where IP spoofing is possible, an attacker would send UDP datagrams to a third party server with a forged IP source address matching the address of the victim. This server would typically send a response back to the forged source address, the victim. If the size of the response is significantly larger than the attacker's original message, then the attacker could overload a victim with less effort than sending a message flood directly. Thus, it is important that the source address of UDP datagrams are validated before sending a larger message back.

As previously described in Section 2.3, handshake responses are smaller than handshake initiations, and cookie replies are smaller than both handshake messages. Thus, these cannot be used for amplification attacks.

Transport data messages on the other hand are potentially very large,

at minimum 32 bytes, but potentially 1232 bytes according to Section 2.3.3. Unlike protocols such as QUIC [35], WireGuard performs no validation of the source IP address during connection migration. This weakness makes WireGuard VPN providers complicit in denial-of-service attacks against other hosts.

Consider an attacker with the following capabilities:

- Access to one or more WireGuard VPN providers with large bandwidth.
- Ability to send spoofed IP packets to the VPN provider.

The attacker does not necessarily have the ability to observe arbitrary network traffic from a VPN provider. Neither does the attacker require visibility into network traffic to the victim.

We present the following attack scenario:

- The attacker completes a handshake as usual with a VPN provider.
- The attacker connects to a website through the VPN tunnel. This website could potentially be located on the same host or network to ensure greater bandwidth for attacking the victim.
- The attacker sends a small HTTP GET request through the VPN tunnel in order to download a **large** file. It sets the source address of the outer IP packet to the address of a victim.
- The VPN server receives the WireGuard transport data message and verifies its authenticity. As it detects a change of the source IP address of the outer packet, it will deliver future transport data messages to the victim.
- The VPN server forwards the HTTP GET request to the website which will start returning TCP segments with response data.
- The attacker continues sending **small** TCP ACK segments through the tunnel with a forged IP address in order to avoid the TCP window from closing down. This keeps the download stream active. At the WireGuard session level, this data message also ensures that the receiver considers the peer as reachable.

- The attacker can periodically restore its original IP address to check the status of the TCP stream and synchronize sequence numbers if needed.

This attack can continue until the session expires after about 3 minutes, but an attacker can always start a new handshake and continue exploitation. The attacker could also use multiple providers to increase the load to the victim.

The above attack scenario works best if the website that acts as traffic generator is able to produce a stable packet stream at a predictable rate and with predictable segment sizes. A possible mechanism to maximize throughput is to probe the TCP congestion avoidance algorithm of the receiver, measure the latency, and model the behavior of the TCP stack of the traffic generator. Once these details are known, the attacker can try to maximize the receive window assumed by the traffic generator by emitting TCP ACK segments at fixed intervals, even if those segments have not been received yet. Note that this attack does not require a large bandwidth from the attacker. Low latency is not necessary either since the attacker does not actually have to receive the payload. Low jitter on the other hand is important to avoid the traffic generator from filling up the assumed TCP receive window which would reduce the throughput.

While HTTP over TCP is a commonly available service that could be abused in an attack, UDP-based protocols could be even more dangerous if they require no acknowledgement messages. For example, iPerf3 is a protocol for testing network throughput. It consists of a TCP control channel and a UDP test stream [43]. While an iPerf3 client usually floods a server, this role can be reversed such that the server floods the client instead. The protocol operates as follows. The iPerf3 client connects to the iPerf3 server over TCP and exchanges parameters, including the reverse mode setting. Once confirmed, the client sends a single UDP datagram to the server. The server will then continue flooding UDP datagrams to the originating UDP port until a stop signal is sent over the TCP control channel. In an attack, one could perform the initial iPerf3 setup over TCP, then send the single iPerf3 UDP message in a WireGuard data message with a spoofed outer IP source address. If suitable iPerf3 servers are available, then this attack is more powerful than the TCP/HTTP variant since no further messages are necessary after the initial setup. We tested one iPerf3 server using the `iperf3 -u -t 3 -c ping.online.net -p5208 -R` command and confirmed that the pro-

tocol is indeed a viable amplification method. This particular server does impose rate limits however.

The author of the WireGuard protocol has previously reported a potential issue due to the lack of authenticity for the IP address [16]. In their scenario, an attacker could force itself into a temporary man-in-the-middle position after observing and forwarding one packet with a modified IP source address. The attacker could then selectively drop packets, perform correlation attacks, and so on. The author of the WireGuard protocol did not consider this to problematic enough to warrant inclusion of extra implementation options to disable roaming. However, our amplification attack was not considered before and it arguably needs further action. Either the option to disable roaming should be added, or the protocol should be repaired.

To deter these attacks, we recommend modifying the WireGuard protocol specification by applying the connection migration considerations from QUIC [35, Section 9]:

- When an external source address change is detected, endpoints may start sending packets to the new peer address as is done now, but must initiate address validation when they do. The immediate migration should limit the amount of lost data in case of legitimate network changes, but exposes endpoints to data loss in case a passive observer forwards transport data messages with a modified source address.

- Addresses that have been validated before might not trigger another address validation. This is for efficiency reasons.

- QUIC supports a *disable migration* option which can be advertised during the handshake and forbids address changes during a session. WireGuard implementations could potentially offer a similar mechanism to implement this policy as an additional layer of protection for site-to-site VPNs over fixed non-mobile networks where address changes are rare. For simplicity, this policy could be shared out-of-band rather than being advertised during the handshake.

- Data to unverified peers must be rate-limited. QUIC bases this limit on the estimated round-trip time and congestion window, but WireGuard currently does not have access to either number. At the very least, a fixed maximum number could be imposed, but the exact value has yet

to be determined. Lower numbers could result in data loss on high-latency networks since the handshake takes more time, higher numbers could increase the exploitation window.

- Address changes should only be considered when the transport data counter has advanced, otherwise out-of-order packets could trigger spurious connection migrations.

- Endpoints must not change their addresses during the handshake. Address validation works by sending a challenge to the peer that is being verified. They must return a response from the same address to prove that they are able to receive messages from said IP address. If there is a mismatch between addresses, it could indicate a rogue peer who was able to complete the handshake but subsequently directs traffic to a third-party victim. We note that this property is already required for when MAC2 validation is enabled as cookies are cryptographically bound to the original address. However current implementations, including Linux and wireguard-go, do not enforce this for regular handshake messages, any accepted handshake message will implicitly update the endpoint address without further validation.

The following sections will analyze some potential mechanisms to perform address validation and implement connection migration. All solutions require the party detecting the address change to issue some form of unpredictable challenge which must be returned from the same new address. Care should be taken not to turn address validation into another form of denial-of-service mechanism to the peers themselves.

5.1.1 Validation using a handshake initiation

As handshake response messages can only be created by a responder who knows the handshake initiation message, it could potentially serve as address validation mechanism. This mechanism is compatible with the existing protocol so a modified implementation could interoperate with an original, unprotected implementation.

The naive implementation does have some potential drawbacks:

- New handshakes are usually created as part of key rotation (Section 2.5). Overloading this mechanism for address validation could result in colli-

sions with regular key rotations and thus loss of data due to expiration of old session keys.

- Handshake initiations can occur only once every 5 seconds which limits the potential validations that can be triggered. This could increase latency for session establishment when a user rapidly switches between networks, for example from Wi-Fi to mobile and back.

- Handshakes are rather expensive, a passive attacker could mount a denial-of-service attack against a peer by copying and forwarding messages with many different source IP addresses. Each of these could potentially trigger address validation, and thus a handshake initiation (subject to the rate limit.)

- Handshake initiation messages are larger than the smallest data message (148 bytes versus 32 bytes), so there is still potential for an amplification attack.

5.1.2 Validation using a token challenge

While the previously suggested handshake mechanism is compatible with existing implementations, it has several drawbacks. Therefore we would like to propose a new message type which stores an unpredictable challenge which can only be answered by the receiver of this message. The token is encrypted to avoid passive observers from trivially bypassing address validation. The fields are as follows:

- Type (1 byte): challenge token or challenge response.

- Reserved (3 bytes): reserved.

- Receiver (4 bytes): receiver index, same as transport data message. Allows the message to be mapped to a peer.

- Counter (8 bytes): AEAD counter, for simplicity reusing the transport data one.

- Encrypted token (8 + 16 bytes): a 64-bit token, encrypted using the same transport key, again with no authenticated additional data.

This message is 40 bytes which is 1.25 times larger than the smallest, a keepalive message, but still smaller than a data message with an actual IP packet. To avoid an amplification attack to one target, we limit the number of challenges per address per time unit (Challenge-Timeout). Similarly to handshake initiation messages, a retransmission mechanism is necessary to ensure that the challenge request reaches the peer. A challenge token message would thus be retransmitted for up to Challenge-Attempt-Time. Note that both timers are tracked on a per-peer basis rather than per-address under the assumption that legitimate peers are typically active on only one address at a time, and only their most recent address would have to be confirmed. While handshakes are expensive, challenges are not, so we suggest lower parameter values:

Symbol	Value
Challenge-Attempt-Time	30 seconds
Challenge-Timeout	2 seconds

The **challenger** performs these steps when address validation is requested:

1. If the challenged address matches the current endpoint address, or previously validated addresses, complete validation with success.

2. If the address was not previously challenged, create a new random token and remember the address and token pair for this peer. This pair must be stored per peer and not shared between different peers. Otherwise that could be used as a side-channel to confirm use of a certain IP address by other users at the same VPN provider.

3. If this address has recently been challenged (within the past Challenge-Timeout), defer validation.

4. If no valid response has been received within Challenge-Attempt-Time, fail validation. This situation could occur for several reasons:

    - The peer is an older implementation that does not support this mechanism. If support for older peers is important, the validation failure must unfortunately be ignored.
    - An attacker has been trying to redirect traffic.

- The peer has become unreachable again on the new network, for example due to switching networks or shutting down.

5. Otherwise reset the Challenge-Attempt-Time timer and send an challenge token message to the address, encrypted similar to transport data messages (Section 2.3.3).

As long as no valid response has been received, the challenger will retransmit the same token in a new challenge message at most every Challenge-Timeout, until Challenge-Attempt-Time has passed since validation was requested.

A **challenge responder** receiving a challenge token message must decrypt it similarly to data messages (avoiding counter reuse, etc.) and on success, it must perform the following steps:

1. If a challenge was previously received from this peer with a higher counter, ignore it.

2. Otherwise immediately transmit a challenge response message with the same token and remember the counter for this peer.

In either case, the considerations for keepalive data messages apply. In particular, counter values must not exceed Reject-After-Messages and the session age must not exceed Reject-After-Time. Like keepalive messages, the challenge messages will not not trigger more keepalive messages.

5.1.3 Validation using a token challenge (variant 2)

The previous address validation mechanism (Section 5.1.2) unfortunately requires existing protocol implementations to be modified in an incompatible way. The same technique can however be implemented using regular IP packets inside a WireGuard tunnel as opposed to modifying the outer UDP messages, but requires additional addressing information. This method can be used by a modified challenger without requiring further implementation changes from challenge responders.

Assuming that the challenger knows the primary *internal* IP address of the peer and itself, it can "ping" the peer and wait for its reply. It does so by creating an ICMP Echo request message with the token in its payload field. Assuming that the internal IP addresses were valid, and the firewall does not block ICMP messages, the responder will send an ICMP Echo reply with the

same token in its payload field. The payload fields of both Echo messages can contain arbitrary data and can store up to approximately 2^{16} bytes, and is thus a perfect fit for the 64-bit token.

While this is compatible with existing deployed implementations, it does have drawbacks when compared to the previous solution:

- It requires a challenger to distinguish regular transport data from token-related ICMP Echo reply messages.
- It requires access to internal addressing information, either IPv4 or IPv6 addresses.
- It requires the peer to pass ICMP traffic through its firewall.

On the other hand, it might offer slightly more privacy since a passive observer is not able to immediately recognize token messages based on the type field.

This compatible mechanism likely requires a considerable amount of extra complexity in the implementations. Additionally, the unpatched peer implementations would still remain vulnerable to the attack that was supposed to be countered. Thus, while this mechanism could potentially be implemented by VPN providers, a separate challenge message would still be preferable for simplicity.

5.2 Handshake initiation collision

In Section 2.5, we described the expected implementation behavior regarding key rotation. We discovered that the protocol specification in the WireGuard whitepaper did not define the expected behavior when both peers attempt to start a handshake at exactly the same time.

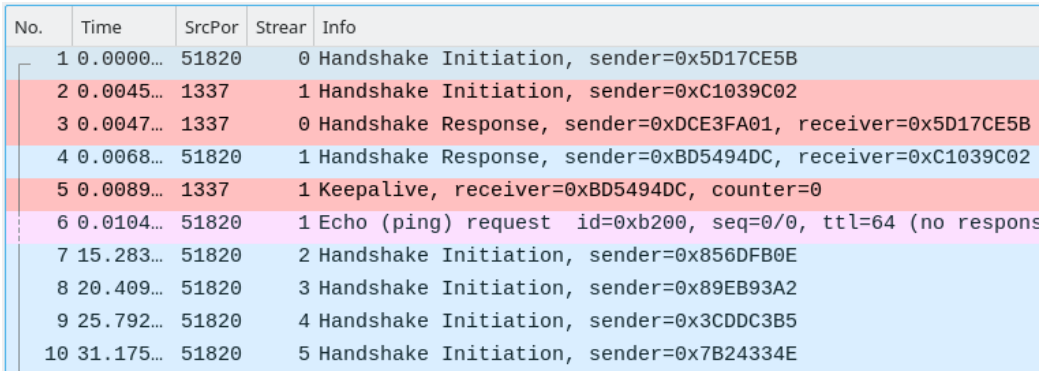
Through a manual code review of the Linux kernel implementation, we discovered that a peer always maintains a single handshake state for both outgoing and incoming handshake initiations. If each peer starts an initiation at the same time and subsequently receives a handshake initiation from the other party, it would overwrite its own original handshake state with a new state based on the received message and reply with a handshake response message. However, since each peer has overwritten its original handshake

initiation, it would no longer be able to complete the handshake. Effectively, the peers have ended up with two disjoint sessions and are unable to communicate with each other until a new handshake is attempted.

We were able to confirm this theory through experimental evaluation using our low-level WireGuard prototyping tool (wgll) (Section 4.3). Consider this test scenario:

1. WireGuard sends an initiation message to wgll, triggered by executing the `ping -c1 $wgllInternalIP` command.
2. wgll immediately sends a new handshake initiation.
3. wgll immediately sends a handshake response matching WireGuard's initiation from step 1.
4. We observe a handshake response from WireGuard matching wgll's initiation from step 2.
5. wgll sends a data message, encrypted using keys matching its own initiated session from step 2.
6. We observe a data message from WireGuard with an ICMP echo-request, encrypted using keys matching wgll's initiated session from step 2.
7. After a delay of 15 seconds (Rekey-Timeout+Keepalive-Timeout), we observe that WireGuard sends new Handshake initiation attempts every 5 seconds (Rekey-Timeout).

Figure 5.1 visualizes our empirical results using a packet capture between wgll and WireGuard. The above steps match with the frame numbers.



No.	Time	SrcPor	Strear	Info
1	0.0000…	51820	0	Handshake Initiation, sender=0x5D17CE5B
2	0.0045…	1337	1	Handshake Initiation, sender=0xC1039C02
3	0.0047…	1337	0	Handshake Response, sender=0xDCE3FA01, receiver=0x5D17CE5B
4	0.0068…	51820	1	Handshake Response, sender=0xBD5494DC, receiver=0xC1039C02
5	0.0089…	1337	1	Keepalive, receiver=0xBD5494DC, counter=0
6	0.0104…	51820	1	Echo (ping) request id=0xb200, seq=0/0, ttl=64 (no respons
7	15.283…	51820	2	Handshake Initiation, sender=0x856DFB0E
8	20.409…	51820	3	Handshake Initiation, sender=0x89EB93A2
9	25.792…	51820	4	Handshake Initiation, sender=0x3CDDC3B5
10	31.175…	51820	5	Handshake Initiation, sender=0x7B24334E

Figure 5.1: The packet list in Wireshark matching the handshake initiation collision scenario. From left to right, the displayed column cover the frame number, the relative time in seconds, the UDP source port, the WireGuard session number, and a summary of the packet.

The kernel logs from WireGuard that correspond to these events are:

```
01.675562 Sending handshake initiation to peer 4 (10.0.2.2:1337)
01.682011 Receiving handshake initiation from peer 4 (10.0.2.2:1337)
01.682044 Sending handshake response to peer 4 (10.0.2.2:1337)
01.682920 Keypair 6 created for peer 4
01.683179 Invalid handshake response from 10.0.2.2:1337
01.685935 Receiving keepalive packet from peer 4 (10.0.2.2:1337)
16.958589 Retrying handshake with peer 4 (10.0.2.2:1337) because
    we stopped hearing back after 15 seconds
16.958708 Sending handshake initiation to peer 4 (10.0.2.2:1337)
22.085231 Handshake for peer 4 (10.0.2.2:1337) did not complete
    after 5 seconds, retrying (try 2)
22.085369 Sending handshake initiation to peer 4 (10.0.2.2:1337)
27.468017 Handshake for peer 4 (10.0.2.2:1337) did not complete
    after 5 seconds, retrying (try 3)
```

Note the *Invalid handshake response* line, it indicates that WireGuard was not able to decrypt the handshake response message based on its current pending handshake state. However as we can see in the Wireshark capture (Figure 5.2), Frame 3 can in fact be linked to the handshake initiation from Frame 1 and Wireshark was able to verify the encrypted payload. As the Keepalive message in Frame 5 was successfully decrypted, we can confirm

that WireGuard's current session is based on the handshake initiation from wgll in Frame 2.

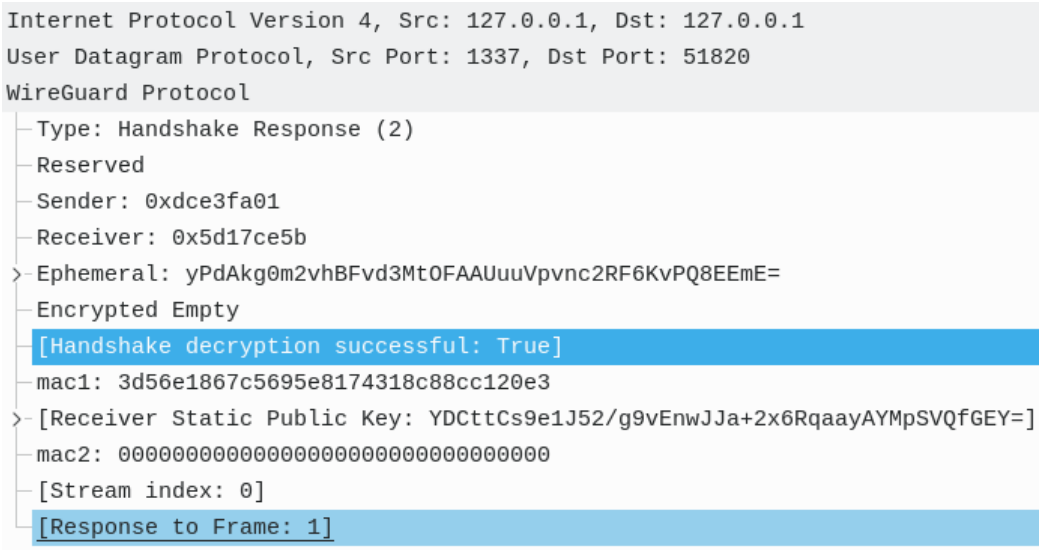


Figure 5.2: .

While our wgll implementation is able to send an encrypted data message that matches the received handshake response message, a real implementation would not be able to do so since it cannot derive session keys based on its original handshake initiation.

Note that the Linux kernel implementation tries to avoid this collision issue by adding a random jitter to the handshake retransmission timer. This jitter is up to $\frac{1}{3}$ second, see the `REKEY_TIMEOUT_JITTER_MAX_JIFFIES` constant defined in src/messages.h and used in src/timers.c. The whitepaper initially did not mention this jitter, but when we privately discussed the issue with Donenfeld, he confirmed that this was an omission.

5.2.1 Impact

While the protocol specification and implementations now require random jitter to reduce the probability of accidental handshake initiation collisions, it does not completely prevent the issue where both peers are unable to send encrypted data.

Consider the following denial-of-service attack against two peers, Alice and Bob, from an adversary, Mallory, with interception and suppression capabilities:

1. Mallory intercepts a handshake initiation message from Alice to Bob and ensures that Bob does not receive it.

2. Mallory intercepts a handshake initiation message from Bob to Alice and ensures that Alice does not receive it.

3. Mallory forwards Alice's handshake initiation message to Bob, and Bob's handshake initiation message to Alice. Effectively, the messages have been stalled.

4. Alice and Bob overwrite their original handshake state and are no longer able to complete the handshake.

While both parties are following the protocol, they will still not be able to establish a mutual session in our attacker model. The practical impact of this issue might be limited as an attacker with such capabilities could also suppress all traffic without maintaining state.

5.3 Identity hiding

Both the formal verification [25] and the WireGuard whitepaper [18] claim identity hiding properties. Recent versions of the whitepaper clarify that the MAC1 field weakens this property: "It is worth noting that this first MAC allows a passive attacker to make guesses about for which public key the packet is intended, slightly weakening identity hiding properties, though a correct guess would not constitute cryptographic proof since no private material was used in generating the MAC." [18].

The formal verification paper uses Tamarin to build a symbolic proof for various properties, including identity hiding. In their model, the MAC1 field is a public variable which is not related to any other field. The proof in Section 4.3.4 [25] shows that the encrypted static public key cannot be revealed through the cryptographic handshake unless some of the ephemeral or static keys are revealed. It does not prove that the identity cannot be confirmed in other ways which the authors indeed acknowledge by stating

that the MAC1 field enables adversaries to confirm the destination public key of messages from honest senders.

So while the Noise IKpsk2 protocol pattern provides a form of identity hiding, WireGuard weakens this property due to changes outside the cryptographic handshake which is based on Noise. The cryptographic handshake does not leak the identity, but the MAC1 field trivially allows passive observers to confirm the identity of a receiver using their static public key, assuming honest senders. Based on these observations, we have extended our Wireshark dissector (Section 4.2) to identify the receiver in a linear number of operations (linear based on the set of known static public keys).

When considering a quantum adversary who knows the receiver's public key, then the initiator identity can also be learned from the handshake initiation message as will be described in Chapter 6.

Implementations that would like to maintain the stronger identity hiding property of Noise could ignore or remove the MAC1 field, but this has some disadvantages:

- It is not compatible with the WireGuard protocol, both the sender and receiver would require modifications. Even if the wire format remains compatible (for example, by using all zeroes for the MAC1 field), no handshake can be completed if only one party ignores the field. If only the initiator is modified, then the responder will drop the message due to an invalid MAC1 value. If only the responder is modified, then the initiator will still reveal the responder's identity and will still require the responder to include a valid MAC1 field in the response.

- When a receiver of a message is "under load", and the MAC2 field does not pass validation, it may skip expensive handshake computations and respond with a Cookie Reply message instead, but only if the MAC1 field passes validation. If the MAC1 field is ignored, then an attacker can induce a reply to any destination address.

- The receiver of an Initiation message has to perform more expensive computations even for a flood of messages from an adversary who does not know the static public key of the recipient. Computing MAC1 requires either one hash function calculation, or two if the MAC1 key is not pre-computed, while learning the identity from the Initiation message requires calculation of at least two KDFs, one hash function, and one DH function.

- Receivers of Responder messages can be forced to perform even more computations under some circumstances. When the receiver has outstanding, unanswered initiation messages as identified by a unique 32-bit identifier, then an attacker could use this 32-bit identifier to force computation of at minimum four KDFs, two hash functions, two DH functions, and one AEAD decryption. These computations are required to derive the keys that are necessary to verify the encrypted payload in a response message. This attack does not even require knowledge of any public key, passively observing initiation messages is sufficient to learn the 32-bit identifier. Actively blocking responder messages from a legitimate responder will also extend the time period in which the receiver (initiator) can be attacked.

5.4 Sender and Receiver indices

The 32-bit index used to link messages to a local session is described in the WireGuard paper [18] as "analogous to IPsec's SPI" (Security Parameter Index). Obviously this can be used to link messages, but unlinkability does not seem to be a stated goal of either Noise or WireGuard, so nothing is lost there.

While the WireGuard paper suggests using a *random* index, the SPI is only required to be *unique*. The current version of boringtun [11] uses a predictable identifier, a concatenation of a 24-bit *peer index* and an 8-bit *cycling session index*. The first value is an index starting at zero but incremented for every peer that is added to the configuration. The second value also starts at zero, but is incremented after rotating to a new key.

Based on this particular implementation (which does *not* affect the main WireGuard implementations), we can learn the following:

- **The number of configured peers on this endpoint.** For typical client applications, this will probably be one while server applications could leak the number of peer additions or changes over time, and thereby leak an estimate on the number of users.

- **The estimated number of uses.** Usually a new handshake occurs every three minutes, so based on this number one can estimate the activity period of a user.

Other than the information leak, the impact appears to be minimal.

Chapter 6

Transitional post-quantum security improvement

This chapter includes the main results from *Tiny WireGuard Tweak* by Jacob Appelbaum, Chloe Martindale, and the author as published and presented at AFRICACRYPT 2019 [1]. It has been edited to reduce duplicate information that has already been treated elsewhere in this thesis. The original function and variable names were maintained in references to the handshake details in Algorithm 1. For the full, original version, see [1].

6.1 Introduction

WireGuard optionally allows peers to fix a pairwise-unique static symmetric value known as a Pre-Shared Key (PSK). WireGuard does not require, nor use a PSK by default. A protocol is post-quantum transitionally secure when it is secure against a passive adversary with a quantum computer [51]. If this transitionally secure protocol is used today, it is not possible for a quantum attacker to decrypt today's network traffic, *tomorrow*.

If a future adversary has access to a quantum computer, historic network traffic protected by WireGuard, and knowledge of *one* WireGuard user's long-term static public key, this threatens the security of the protocol for all related WireGuard users, as explained in Section 6.2. In this chapter we propose a tiny tweak to the WireGuard protocol that makes WireGuard traffic flows secure against such an adversary; if our alteration is incorporated into the WireGuard protocol, a user's historic traffic will not be decryptable

by such an adversary if users do not release their long-term static public key to the network, as explained in Section 6.4. We accomplish this with both extremely minimal costs and minimal changes to the original protocol, as detailed in Section 6.5.

6.2 Quantum attack

Consider an attacker capable [1] of running Shor's algorithm [53]. Shor's algorithm breaks the discrete logarithm problem in any group in time polynomial in the size of the group; observe that this includes elliptic curve groups. Suppose that the long-term static public key of some WireGuard user U_0 is known to an adversary. We show in Algorithm 2 and Algorithm 3 that in this situation, Shor's algorithm will apply to users of the WireGuard protocol, as given in Algorithm 1.

An adversary can detect when a handshake takes place between U_0 and any other WireGuard user. We describe in Algorithm 2 how to extract the long-term static secret key of any initiator with a quantum computer when U_0 is the responder.

Of course after computing the ephemeral keys, an adversary who has access to the static secret and public keys of both the initiator and the responder of a WireGuard handshake can completely break the protocol (assuming the responder U_0 and the initiator use the default WireGuard settings, i.e. no PSK).

Now suppose an adversary wishes to attack some user U_n. Suppose also that there exists a *traceable path* from U_0 to U_n, that is, if by analyzing the traffic flow the adversary can find users $U_1, \ldots, U_{n-1}$ for which every pair of 'adjacent' users U_i and U_{i+1} have performed a WireGuard handshake. We show in Algorithm 3 how the adversary can then compute U_n's long-term static key pair. The initiator and responder role may change over time, so eventually a responder might turn into an initiator and leak its static public key. Recall from Section 5.3 that the information of which pairs of users have performed a WireGuard handshake is freely available; if such a path exists then an adversary can easily find it.

An important remark on this attack: if two WireGuard users do not publish their static public keys, and *both* users do not interact with any

[1] See [49] for a recent estimate of the resources needed by an attacker to carry out such an attack using Shor's algorithm.

Algorithm 1 Simplified WireGuard key agreement process

Public Input: Curve25519 $E/\mathbb{F}_p$, base point $P \in E(\mathbb{F}_p)$, hash function H, an empty string ϵ, key derivation function KDF_n returning n derived values indexed by n, and a MAC function Poly1305.

Secret Input (Laura): secret key $\mathsf{sk}_L \in \mathbb{Z}$, public key $\mathsf{pk}_L = \mathsf{sk}_L \cdot P \in E(\mathbb{F}_p)$, Julian's pre-shared public key $\mathsf{pk}_J \in E(\mathbb{F}_p)$, shared secret $\mathsf{s} = \mathsf{DH}(\mathsf{sk}_L, \mathsf{pk}_J)$, message time, PSK $Q \in \{0,1\}^{256}$; $Q = 0^{256}$ by default.

Secret Input (Julian): secret key $\mathsf{sk}_J \in \mathbb{Z}$, public key $\mathsf{pk}_J = \mathsf{sk}_J \cdot P \in E(\mathbb{F}_p)$, Laura's pre-shared public key $\mathsf{pk}_L \in E(\mathbb{F}_p)$, shared secret $\mathsf{s} = \mathsf{DH}(\mathsf{sk}_J, \mathsf{pk}_L)$, PSK $Q \in \{0,1\}^{256}$; $Q = 0^{256}$ by default.

Output: Session keys.

1: Both parties choose ephemeral secrets: $\mathsf{esk}_L \in \mathbb{Z}$ for Laura, $\mathsf{esk}_J \in \mathbb{Z}$ for Julian.
2: Laura publishes $\mathsf{epk}_L \leftarrow \mathsf{esk}_L \cdot P$.
3: Laura computes $\mathsf{se}_{JL} \leftarrow \mathsf{esk}_L \cdot \mathsf{pk}_J$; Julian computes $\mathsf{se}_{JL} \leftarrow \mathsf{sk}_J \cdot \mathsf{epk}_L$.
4: Both parties compute $(\mathsf{ck}_1, \mathsf{k}_1) \leftarrow \mathsf{KDF}_2(\mathsf{epk}_L, \mathsf{se}_{JL})$.
5: Laura computes $\mathsf{h}_1 \leftarrow H(\mathsf{pk}_J || \mathsf{epk}_L)$.
6: Laura computes and transmits $\mathsf{enc\text{-}id} \leftarrow \mathsf{aead\text{-}enc}(\mathsf{k}_1, 0, \mathsf{pk}_L, \mathsf{h}_1)$.
7: Julian decrypts enc-id with $\mathsf{aead\text{-}dec}(\mathsf{k}_1, 0, \mathsf{enc\text{-}id}, \mathsf{h}_1)$ and verifies that the resulting value (pk_L) is valid user's public key; aborts on failure.
8: Both parties compute $(\mathsf{ck}_2, \mathsf{k}_2) = \mathsf{KDF}_2(\mathsf{ck}_1, \mathsf{s})$.
9: Laura computes $\mathsf{h}_2 \leftarrow H(\mathsf{h}_1 || \mathsf{enc\text{-}id})$.
10: Laura computes and transmits $\mathsf{enc\text{-}time} \leftarrow \mathsf{aead\text{-}enc}(\mathsf{k}_2, 0, \mathsf{time}, \mathsf{h}_2)$.
11: Both parties compute $\mathsf{pkt} \leftarrow \mathsf{epk}_L || \mathsf{enc\text{-}id} || \mathsf{enc\text{-}time}$.
12: Laura computes and transmits $\mathsf{mac1} \leftarrow \mathsf{MAC}(\mathsf{pk}_J, \mathsf{pkt})$.
13: Julian verifies that $\mathsf{mac1} = \mathsf{MAC}(\mathsf{pk}_J, \mathsf{pkt})$; aborts on failure.
14: Julian computes $\mathsf{time} = \mathsf{aead\text{-}dec}(\mathsf{k}_2, 0, \mathsf{enc\text{-}time}, \mathsf{h}_2)$; aborts on failure.
15: Julian transmits $\mathsf{epk}_J \leftarrow \mathsf{esk}_J \cdot P$.
16: Laura computes $\mathsf{se}_{LJ} \leftarrow \mathsf{sk}_L \cdot \mathsf{epk}_J$; Julian computes $\mathsf{se}_{LJ} \leftarrow \mathsf{esk}_J \cdot \mathsf{pk}_L$.
17: Laura computes $\mathsf{ee} \leftarrow \mathsf{esk}_L \cdot \mathsf{epk}_J$; Julian computes $\mathsf{ee} \leftarrow \mathsf{esk}_J \cdot \mathsf{epk}_L$.
18: Both parties compute $(\mathsf{ck}_3, \mathsf{t}, \mathsf{k}_3) \leftarrow \mathsf{KDF}_3(\mathsf{ck}_2 || \mathsf{epk}_J || \mathsf{ee} || \mathsf{se}_{LJ}, Q)$.
19: Julian computes $\mathsf{h}_3 \leftarrow H(\mathsf{h}_2 || \mathsf{enc\text{-}time} || \mathsf{epk}_J || \mathsf{t})$.
20: Julian computes and transmits $\mathsf{enc\text{-}e} \leftarrow \mathsf{aead\text{-}enc}(\mathsf{k}_3, 0, \epsilon, \mathsf{h}_3)$.
21: Laura verifies that $\epsilon = \mathsf{aead\text{-}dec}(\mathsf{k}_3, 0, \mathsf{enc\text{-}e}, \mathsf{h}_3)$.
22: Both parties compute shared secrets $(T_i, T_r) \leftarrow \mathsf{KDF}_2(\mathsf{ck}_3, \epsilon)$.
23: **return** (T_i, T_r).

other WireGuard users, then this attack does not apply to those two users.

Algorithm 2 Extract Initiator's Long-term Static Key Pair

Input: Long-term static public key pk_J of the responder; Ephemeral public key epk_L of the initiator (transmitted over the wire in Step 2 of Algorithm 1); $\mathsf{enc\text{-}id}$ as sent over the wire by the initiator in Step 6 of Algorithm 1.

Output: Long-term static key pair $\mathsf{sk}_L, \mathsf{pk}_L$ of the initiator.

1: Using Shor's algorithm, compute esk_L from epk_L.
2: Compute k_1 and h_1 as in Steps 4 and Steps 5 respectively of Algorithm 1.
3: Compute $\mathsf{pk}_L = \mathsf{aead\text{-}dec}(\mathsf{k}_1, 0, \mathsf{enc\text{-}id}, \mathsf{h}_1)$.
4: Compute sk_L from pk_L using Shor's algorithm.
return $\mathsf{sk}_L, \mathsf{pk}_L$.

Algorithm 3 Extract User U_n's Long-term Static Key Pair

Input: Long-term static public key of some WireGuard User U_0; A traceable path from U_0 to WireGuard User of interest U_n.

Output: Long-term static key pair of WireGuard User U_n.

1: **for** $i := 0, \ldots, n-1$ **do**
2: $U_i \leftarrow$ Responder (without loss of generality).
3: $U_{i+1} \leftarrow$ Initiator (also without loss of generality).
4: Compute long-term static key pair of U_{i+1} using Algorithm 2.
5: **end for**
return Long-term static key pair of U_n.

6.3 A brief comment on extra security options

In Section 6.2 we analyzed the *default* use of the WireGuard protocol. There is an option open to WireGuard users to also preshare another secret key, i.e., to use a PSK Q as an additional input for the KDF in Step 18 of Algorithm 1. If the user does not configure a PSK, the default ($Q = 0^{256}$) will be used.

Use of a secret PSK will not prevent a quantum adversary from computing $\mathsf{sk}_L, \mathsf{pk}_L$ using the method described in Section 6.2. It does however

prevent compromise of session keys T_i and T_r in Step 22 of Algorithm 1 as the adversary no longer has enough information to compute ck_3 in Step 18 of Algorithm 1.

A prudent user may still be concerned about an adversary stealing their PSK; the tiny protocol tweak presented in Section 6.4 addresses this concern as well as protecting those who use the default mode of the protocol.

Of course our tweak cannot protect against an adversary who steals the static long-term public key of both the initiator and the responder in a WireGuard handshake.

6.4 Blinding flows against mass surveillance

We propose a tiny tweak to the WireGuard handshake which thwarts the quantum attack outlined in the previous section: In Step 6 and Step 7 of Algorithm 1, replace pk_L by $H(\mathsf{pk}_L)$. We suggest to use BLAKE2s as the hash function H as it is already used elsewhere in WireGuard. Naturally, the unhashed static public key pk_L of the initiator has still been exchanged out-of-band, so the responder can still perform Diffie-Hellman operations with the initiator's static public key pk_L, and is able to compute the hash $H(\mathsf{pk}_L)$. In Step 7 and Step 16 of Algorithm 1, the responder will use the decrypted value $H(\mathsf{pk}_L)$ to look up the corresponding key pk_L.

The hashing process conceals the algebraic structure of the static public key of the initiator and replaces it with a deterministic, predictable identifier. This requires no extra configuration information for either of the peers. BLAKE2s is a one-way hash function and a quantum adversary would first need to run a Grover attack [32] for 2^{128} quantum operations to even recover a candidate preimage.

An attacker as described in Section 6.2 may confirm a guess of a known long-term static public key. If the guess is correct, they may carry out the attack as in the unchanged WireGuard protocol. However, the tweak protects sessions where the public keys are not known.

We claim only *transitional security* with this alteration. That is, that a future quantum adversary will not be able to decrypt messages sent before the advent of practical quantum computers, if the messages are encrypted via an updated version of WireGuard that includes our proposed tweak. The tweaked protocol is not secure against active quantum attacks with knowledge of both long-term static public keys and a known PSK value.

With knowledge of zero or only one long-term static public key, the protocol remains secure. A redesign of the WireGuard protocol to achieve full post-quantum security is still needed.

There are of course other choices of values to replace the static public key in Step 6 and Step 7 of Algorithm 1 to increase security. One alternative choice of value is an empty string, as in the case with the message sent in response to initiator packets by the responder. This would change the number of trial decryptions for the responder for initiator messages to $O(n)$ where n is the number of configured peers. This change would allow any would-be attacker to force the responder to perform many more expensive calculations. It would improve identity hiding immensely but at a cost that simply suggests using a different Noise pattern in the first place. A second alternative choice of value is a random string which is linked to a user at configuration time, similar to a username or a numbered account, which is common in OpenVPN and similar deployments. This provides $O(1)$ efficiency in lookups of session structures but with a major loss in ease of use and configuration. It would also add a second identifier for the peer which does not improve identity hiding. Both alternative choices have drawbacks. The first method would create an attack vector for unauthenticated consumption of responder resources and the second method would require additional configuration. Both weaken the channel binding property of Noise [47, Chapter 14] as the encrypted public key of the initiator is no longer hashed in the handshake hash. The major advantage of our proposed choice is that it does not complicate configuration, nor does it require a wire format change for the WireGuard protocol. Assuming collision-resistance of the hash function, the channel binding property is also preserved. Our proposal concretely improves the confidentiality of the protocol without increasing the computation in any handshake. It increases the computation for peer configuration by only a single hash function for each configured public key.

This change does not prevent linkability of flows as it exchanges one static identifier for another, and it does preclude sharing that identifier in a known vulnerable context.

6.5 Modified protocol costs

Our modification obviously requires implementation changes. We study the effect on the proposed Linux kernel implementation as outlined in the Wire-

Guard paper [18] as well as the effect on the alternative implementations.

The hash function input (the initiator's static public key) and the output have an identical length, thus the wire format and internal message structure definitions do not need to change to accommodate the extra hash operation.

Initiators only have a single additional computational cost, computing the hash of their own static public key. This could be done during each handshake at no additional memory cost, or during device configuration which only requires an additional 32 bytes of memory in the device configuration data structure to store the hash of the peer's long-term static public key.

Responders must be able to find the peer configuration based on the initiation handshake message since it includes the peer's static public key, optional PSK, permitted addresses, and so on. In the unmodified protocol, a hash table could be used to enable efficient lookups using the static public key as table key. At insertion time, a hash would be computed over the table key. The Linux kernel implementation uses SipHash2-4 [2] as hash function for this table key [18, Section 7.4]. Our modification increases the size of the per-peer data structure by 32 bytes and requires a single additional hash computation per long-term static public key at device configuration time. There are no additional memory or computational costs during the handshake.

The wireguard-go [21, device/device.go] implementation uses a standard map data type using the static public key as map key. Again, a single additional hash computation is required at configuration time with no additional memory usage.

Recall that WireGuard is based on the Noise protocol framework. Our modification is not compatible with the current version of this framework, and thus implementations that rely on a Noise library to create and process handshake messages must be changed to use an alternative Noise implementation. This affects the Rust implementation [22].

Chapter 7

Conclusion

The primary research question was evaluating whether the security claims made by WireGuard are correct and evaluating whether it is suitable for implementing a secure Virtual Private Network (VPN).

Based on the analysis of the protocol specification and implementations, we provided a detailed protocol description (Chapter 2). This description includes details that were omitted in the original whitepaper, but present in actual implementations, such as ECN support (Section 2.3.3) and handshake initiation jitter (Section 5.2).

Based on our analysis and the results in related work (Chapter 3), we can confirm that the security claims are correct. However, the claimed identity hiding property is actually weaker than the original guarantee provided by the Noise protocol (Section 5.3). We demonstrated the feasibility of exploitation by developing a new Wireshark protocol dissector for WireGuard that reveals the identities of the peers (Section 4.2).

We created several tools to study WireGuard implementations, an initial prototype of a Wireshark dissector in Lua (Section 4.1), an improved Wireshark dissector in C (Section 4.2), and the *wgll* WireGuard implementation in Python for testing other implementations (Section 4.3). To enable decryption of network traffic in Wireshark dissectors, we developed mechanisms to extract secrets from kernel memory (Section 4.1.1) and store these in packet capture files [59]. We investigated various X25519 implementations to support decryption using these secrets (Section 4.2.1).

We described several potential improvements, such as protection against a denial-of-service attack against third parties (Section 5.1) and transitional post-quantum security (Chapter 6). The latter improvement has been incor-

porated in our peer-reviewed paper that was published at AFRICACRYPT 2019.

Our wgll tool was able to confirm a potential issue in WireGuard implementations that could result in failure to establish a communication channel in rare circumstances (Section 5.2). An attacker could potentially use this implementation detail to prevent sessions from being established. However, in most cases it would be more effective to simply drop messages to achieve the same denial-of-service attack against peers.

Our evaluation of the boringtun implementation also revealed that the original specification was not clear enough, resulting in linkability concerns for sessions (Section 5.4). However as WireGuard does not claim to possess this property, it should not considered to be problematic. A more severe issue with the boringtun implementation is improper implementation of key rotation [58]. Our protocol specification and state machine description (Section 2.7) could potentially help third-party implementations to avoid such issues.

We conclude that WireGuard is a secure building block to build secure VPN software, but do observe that it can be used in a denial-of-service attack against third parties. Aside from that issue, we think that WireGuard is suitable for mass-adoption.

Bibliography

[1] Jacob Appelbaum, Chloe Martindale, and Peter Wu. Tiny WireGuard Tweak. In *AFRICACRYPT 2019*, volume 11627 of *Lecture Notes in Computer Science*. Springer, 2019. URL https://eprint.iacr.org/2019/482. To appear.

[2] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A fast short-input PRF. In *INDOCRYPT 2012*, volume 7668 of *Lecture Notes in Computer Science*, pages 489–508. Springer, 2012. URL https://eprint.iacr.org/2012/351.

[3] Daniel J. Bernstein. TAI64, TAI64N, and TAI64NA, 1997. URL https://cr.yp.to/libtai/tai64.html.

[4] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006. URL https://cr.yp.to/papers.html#curve25519.

[5] Daniel J. Bernstein. Boring crypto, October 2015. URL https://cr.yp.to/talks.html#2015.10.05.

[6] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The Security Impact of a New Cryptographic Library. In *LATINCRYPT 2012*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer, 2012. URL https://eprint.iacr.org/2011/646.

[7] Daniel J. Bernstein, Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. TweetNaCl: A Crypto Library in 100 Tweets. In *LATINCRYPT 2014*, volume 8895 of *Lecture Notes in Computer Science*, pages 64–83. Springer, 2014. URL https://tweetnacl.cr.yp.to/papers.html.

[8] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (In-)Security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *ACM Conference on Computer and Communications Security*, pages 456–467. ACM, 2016. URL https://eprint.iacr.org/2016/798.

[9] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security and Privacy*, pages 98–113. IEEE Computer Society, 2014. URL https://mitls.org/pages/attacks/3SHAKE.

[10] CERT. C compilers may silently discard some wraparound checks, April 2008. URL https://www.kb.cert.org/vuls/id/162289/.

[11] Cloudflare. Userspace WireGuard® implementation in Rust, May 2019. URL https://github.com/cloudflare/boringtun. commit 37be47416119.

[12] Dr. Steve E. Deering and Bob Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 8200, July 2017. URL https://rfc-editor.org/rfc/rfc8200.txt.

[13] Jean Paul Degabriele and Kenneth G. Paterson. On the (in)security of IPsec in MAC-then-encrypt configurations. In *ACM Conference on Computer and Communications Security*, pages 493–504. ACM, 2010.

[14] Frank Denis. Sodium: A modern, portable, easy to use crypto library, 2019. URL https://libsodium.org/. version 1.0.16.

[15] Jason A. Donenfeld. WireGuard: Next generation kernel network tunnel. In *NDSS 2017*. The Internet Society, 2017. URL https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/wireguard-next-generation-kernel-network-tunnel/.

[16] Jason A. Donenfeld. Roaming Mischief, November 2017. URL https://lists.zx2c4.com/pipermail/wireguard/2017-November/001957.html.

[17] Jason A. Donenfeld. extract-handshakes.sh: Handshake extractor, 2018. URL https://git.zx2c4.com/WireGuard/tree/contrib/examples/extract-handshakes/README.

[18] Jason A. Donenfeld. WireGuard: Next generation kernel network tunnel, 2018. URL https://www.wireguard.com/papers/wireguard.pdf. version 416d63b 2018-06-30.

[19] Jason A. Donenfeld. [ANNOUNCE] Wintun: Layer 3 TUN Driver for Windows, March 2019. URL https://lists.zx2c4.com/pipermail/wireguard/2019-March/004038.html.

[20] Jason A. Donenfeld. WireGuard Linux kernel source, 2019. URL https://git.zx2c4.com/WireGuard. tag 0.0.20190406, commit 91b0a211861d.

[21] Jason A. Donenfeld. Source code for the Go implementation of WireGuard, May 2019. URL https://git.zx2c4.com/wireguard-go. tag 0.0.20190409, commit 18fa27047265.

[22] Jason A. Donenfeld. Source code for the Rust implementation of WireGuard, Jan 2019. URL https://git.zx2c4.com/wireguard-rs. commit a7a2e5231571.

[23] Jason A. Donenfeld. WireGuard: fast, modern, secure VPN tunnel, 2019. URL https://www.wireguard.com/.

[24] Jason A. Donenfeld. Wintun – Layer 3 TUN Driver for Windows, April 2019. URL https://www.wintun.net/.

[25] Jason A. Donenfeld and Kevin Milner. Formal verification of the WireGuard protocol, 2018. URL https://www.wireguard.com/papers/wireguard-formal-verification.pdf. version b956944 2018-06-07.

[26] Benjamin Dowling and Kenneth G. Paterson. A cryptographic analysis of the WireGuard protocol. In *ACNS 2018*, volume 10892 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2018. URL https://eprint.iacr.org/2018/080.

[27] Eric Dumazet. Linux kernel patch: ipv6: Limit mtu to 65575 bytes, 2014. URL https://git.kernel.org/linus/30f78d8ebf7f.

[28] Sally Floyd, Dr. K. K. Ramakrishnan, and David L. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, September 2001. URL https://rfc-editor.org/rfc/rfc3168.txt.

[29] FreeBSD. Chapter 8. IPv6 Internals – Jumbo Payload, 2019. URL https://www.freebsd.org/doc/en/books/developers-handbook/ipv6.html#ipv6-jumbo.

[30] Gerald Combs, et. al. Wireshark: Network protocol analyzer, 1998–2019. URL https://www.wireshark.org/.

[31] Glenn Greenwald. XKeyscore: NSA tool collects 'nearly everything a user does on the internet', July 2013. URL https://www.theguardian.com/world/2013/jul/31/nsa-top-secret-program-online-data.

[32] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *STOC*, pages 212–219. ACM, 1996.

[33] Masami Hiramatsu. Kprobe-based event tracing, 2009. URL https://www.kernel.org/doc/html/latest/trace/kprobetrace.html.

[34] ipoque GmbH. Rohde & Schwarz Adds Emerging WireGuard VPN Protocol to its Deep Packet Inspection (DPI) Software Library, January 2019. URL https://ipoque.com/news-media/press-releases/2019/rohde-schwarz-adds-emerging-wireguard-vpn-protocol-its-deep-packet.

[35] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. Internet-Draft draft-ietf-quic-transport-20, Internet Engineering Task Force, April 2019. URL https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-20. Work in Progress.

[36] Jacob Hoffman-Andrews. Verizon injecting perma-cookies to track mobile customers, bypassing privacy controls, November 2014. URL https://www.eff.org/deeplinks/2014/11/verizon-x-uidh.

[37] Nadim Kobeissi and Karthikeyan Bhargavan. Noise Explorer: Fully automated modeling and verification for arbitrary Noise protocols, June 2019. URL https://eprint.iacr.org/2018/766. To appear.

[38] Werner Koch. Add dedicated X25519 function to Libcgrypt, December 2018. URL https://dev.gnupg.org/T4293.

[39] Werner Koch. Libgcrypt website, 2019. URL https://gnupg.org/software/libgcrypt/. version 1.8.4.

[40] Dr. Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010. URL https://rfc-editor.org/rfc/rfc5869.txt.

[41] Hugo Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. In *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 546–566. Springer, 2005. URL https://eprint.iacr.org/2005/176.

[42] Benjamin Lipp. A Mechanised Computational Analysis of the WireGuard Virtual Private Network Protocol. Master's thesis, Karlsruhe Institute of Technology, May 2018. URL https://benjaminlipp.de/master-thesis.

[43] Bruce A. Mah. The Iperf3 Protocol and State Machine, February 2014. URL https://github.com/esnet/iperf/wiki/IperfProtocolStates.

[44] Marek Majkowski. Broken packets: IP fragmentation is flawed, Aug 2017. URL https://blog.cloudflare.com/ip-fragmentation-is-broken/.

[45] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, June 2018. URL https://rfc-editor.org/rfc/rfc8439.txt.

[46] Telecommunication Standardization Sector of ITU. ITU-T Rec. X.200 (07/94): Information technology – Open Systems Interconnection – Basic Reference Model: The basic model, 1994. URL https://www.itu.int/rec/T-REC-X.200-199407-I.

[47] Trevor Perrin. The Noise Protocol Framework, 2018. URL https://noiseprotocol.org/noise.html. revision 34 2018-07-11.

[48] LLVM Project. libFuzzer – a library for coverage-guided fuzz testing, 2019. URL https://llvm.org/docs/LibFuzzer.html. version 8.0.0.

[49] Martin Roetteler, Michael Naehrig, Krysta M. Svore, and Kristin E. Lauter. Quantum resource estimates for computing elliptic curve discrete logarithms. In *ASIACRYPT (2)*, volume 10625 of *Lecture Notes in Computer Science*, pages 241–270. Springer, 2017. URL https://eprint.iacr.org/2017/598.

[50] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC). RFC 7693, November 2015. URL https://rfc-editor.org/rfc/rfc7693.txt.

[51] John M. Schanck, William Whyte, and Zhenfei Zhang. Circuit-extension handshakes for Tor achieving forward secrecy in a quantum world. *Proceedings on Privacy Enhancing Technologies*, 4:219–236, 2016. URL https://eprint.iacr.org/2015/287.pdf.

[52] Yaron Sheffer and Hannes Tschofenig. Internet Key Exchange Protocol Version 2 (IKEv2) Session Resumption. RFC 5723, January 2010. URL https://rfc-editor.org/rfc/rfc5723.txt.

[53] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *FOCS*, pages 124–134. IEEE Computer Society, 1994.

[54] Ludvig Strigeus. Source code of the TunSafe client, Dec 2018. URL https://github.com/TunSafe/TunSafe. commit 85a871c1d226.

[55] Peter Wu. 25519 usability and performance in Libgcrypt, Jul 2018. URL https://lists.gnupg.org/pipermail/gcrypt-devel/2018-July/004532.html.

[56] Peter Wu. Wireshark dissector (written in Lua) for dissecting the WireGuard tunneling protocol., Aug 2018. URL https://github.com/Lekensteyn/wireguard-dissector. commit 240f0ab22232.

[57] Peter Wu. Bug 15011 – Support for WireGuard VPN protocol, 2018. URL https://bugs.wireshark.org/bugzilla/show_bug.cgi?id=15011.

[58] Peter Wu. Incomplete session lifetime management (key rotation) – boringtun, Mar 2019. URL https://github.com/cloudflare/boringtun/issues/50. Bug report about an implementation issue.

[59] Peter Wu. Add DSB type for WireGuard (0x57474b4c) – pcapng, May 2019. URL https://github.com/pcapng/pcapng/pull/62.

[60] Peter Wu. Low-level prototyping tool for WireGuard, May 2019. URL https://github.com/Lekensteyn/wgll. commit 11ac7925687b.

Appendix A

Informal protocol narration

This appendix lists the protocol narration with a focus on the most important data flow in the handshake computations as described in Section 2.3. For brevity, steps 4 and 5 have been simplified by removing hashing.

1. Initial handshake message creation

1 :	**Initiator**	**Responder**
2 :	$\mathsf{pk}_I, \mathsf{sk}_I, \mathsf{tsp}$, initial data P	$\mathsf{pk}_R, \mathsf{sk}_R$
	..... Out-of-band key exchange: $\mathsf{pk}_I, \mathsf{pk}_R, Q$	
3 :	$\mathsf{epk}_I, \mathsf{esk}_I = \mathsf{DHGen}()$	
4 :	$\mathsf{ck}_1, k_1 = \mathsf{HKDF}_2(\mathsf{epk}_I, \mathsf{DH}(\mathsf{esk}_I, \mathsf{pk}_R))$	
5 :	$h_1 = H(\mathsf{pk}_R \| \mathsf{epk}_I)$	
6 :	$\mathsf{enc\text{-}id} = \mathsf{aead\text{-}enc}(k_1, 0, h_1, \mathsf{pk}_I)$	
7 :	$\mathsf{ck}_2, k_2 = \mathsf{HKDF}_2(\mathsf{ck}_1, \mathsf{DH}(\mathsf{sk}_I, \mathsf{pk}_R))$	
8 :	$h_2 = H(h_1 \| \text{enc-id})$	
9 :	$\mathsf{enc\text{-}tsp} = \mathsf{aead\text{-}enc}(k_2, 0, h_2, \mathsf{tsp})$	
10 :	$\mathsf{pkt} = \mathsf{epk}_I \| \mathsf{enc\text{-}id} \| \mathsf{enc\text{-}tsp}$	
11 :	$\mathsf{mac1} = \mathsf{BLAKE2s}(\mathsf{pk}_R, \mathsf{pkt})$	
12 :		$\xrightarrow[\text{Initiator packet}]{\mathsf{epk}_I, \mathsf{enc\text{-}id}, \mathsf{enc\text{-}tsp}, \mathsf{mac1}}$

2. Initial handshake message processing

12 : $\underset{\text{Initiator packet}}{\xrightarrow{\mathsf{epk}_I, \mathsf{enc\text{-}id}, \mathsf{enc\text{-}tsp}, \mathsf{mac1}}}$

.....Responder receives initiator packet.....

13 : $\mathsf{pkt'} = \mathsf{epk}_I || \mathsf{enc\text{-}id} || \mathsf{enc\text{-}tsp}$

14 : Verify $\mathsf{BLAKE2s}(\mathsf{pk}_R, \mathsf{pkt'}, \mathrm{mac1})$

15 : $\mathsf{ck}'_1, k'_1 = \mathsf{HKDF}_2(\mathsf{epk}_I, \mathsf{DH}(\mathsf{sk}_R, \mathsf{epk}_I))$

16 : $\mathsf{id} = \mathsf{aead\text{-}dec}(k'_1, 0, h'_1, \mathsf{enc\text{-}id})$

17 : $\mathsf{pk}_I, Q = \mathsf{LookupPeerKeys}(\mathsf{id})$

18 : $\mathsf{ck}'_2, k'_2 = \mathsf{HKDF}_2(\mathsf{ck}'_1, \mathsf{DH}(\mathsf{sk}_R, \mathsf{pk}_I))$

19 : $h'_2 = H(h'_1 || \mathsf{enc\text{-}id})$

20 : $\mathsf{tsp}' = \mathsf{aead\text{-}dec}(\mathsf{k}'_2, 0, \mathsf{h}'_2, \mathsf{enc\text{-}tsp})$

21 : $\mathsf{VerifyAntiReplay}(\mathsf{tsp}')$

3. Response handshake message creation

22 : $\mathsf{epk}_R, \mathsf{esk}_R = \mathsf{DHGen}()$

23 : $\mathsf{ck}_3 = \mathsf{HKDF}_1(\mathsf{ck}'_2, \mathsf{epk}_R)$

24 : $\mathsf{ck}_4 = \mathsf{HKDF}_1(\mathsf{ck}_3, \mathsf{DH}(\mathsf{esk}_R, \mathsf{epk}_I))$

25 : $\mathsf{ck}_5 = \mathsf{HKDF}_1(\mathsf{ck}_4, \mathsf{DH}(\mathsf{esk}_R, \mathsf{pk}_I))$

26 : $\mathsf{ck}_6, t, k_3 = \mathsf{HKDF}_3(\mathsf{ck}_5, Q)$

27 : $h_3 = H(h'_2 || t)$

28 : $\mathsf{enc\text{-}e} = \mathsf{aead\text{-}enc}(k_3, 0, h_3, \epsilon)$

29 : $\mathsf{mac1}_r = \mathsf{BLAKE2s}(\mathsf{pk}_I, \mathsf{epk}_R || \mathsf{enc\text{-}e})$

30 : $\underset{\text{Responder packet}}{\xleftarrow{\mathsf{epk}_R, \mathsf{enc\text{-}e}, \mathsf{mac1_r}}}$

4. Response handshake message processing

30 : $\xleftarrow[\text{Responder packet}]{\mathsf{epk}_R, \mathsf{enc\text{-}e}, \mathsf{mac1_r}}$

..... Initiator receives responder packet .....

31 : Verify $\mathsf{BLAKE2s}(\mathsf{pk}_I, \mathsf{epk}_R || \mathsf{enc\text{-}e}, \mathsf{mac1}_r)$

32 : $\mathsf{ck}'_3 = \mathsf{HKDF}_1(\mathsf{ck}_2, \mathsf{epk}_R)$

33 : $\mathsf{ck}'_4 = \mathsf{HKDF}_1(\mathsf{ck}'_3, \mathsf{DH}(\mathsf{esk}_I, \mathsf{epk}_R))$

34 : $\mathsf{ck}'_5 = \mathsf{HKDF}_1(\mathsf{ck}'_4, \mathsf{DH}(\mathsf{sk}_I, \mathsf{epk}_R))$

35 : $\mathsf{ck}'_6, t', k'_3 = \mathsf{HKDF}_3(\mathsf{ck}'_5, Q)$

36 : $h'_3 = H(h_2 || t')$

37 : Verify $\mathsf{aead\text{-}dec}(k'_3, 0, h'_3, \mathsf{enc\text{-}e}) = \epsilon$

5. Handshake confirmation

38 : $T_i^{send}, T_i^{recv} = \mathsf{HKDF}_2(\mathsf{ck}'_6, \epsilon)$

39 : $\mathsf{ctr}_i = 0$

40 : $\mathsf{enc\text{-}P} = \mathsf{aead\text{-}enc}(T_i^{send}, \mathsf{ctr}_i, \epsilon, \mathsf{P})$

41 : $\xrightarrow[\text{Initial data packet}]{\mathsf{ctr}_i,\ \mathsf{enc\text{-}P}}$

..... Responder receives confirmation .....

42 : $T_r^{recv}, T_r^{send} = \mathsf{HKDF}_2(\mathsf{ck}_6, \epsilon)$

43 : $\mathsf{P'} = \mathsf{aead\text{-}dec}(T_r^{recv}, \mathsf{ctr}_i, \epsilon, \mathsf{enc\text{-}P})$

..... Session established .....

44 :