VIETNAMESE - GERMAN UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

Frankfurt university of applied sciences

Faculty 2: Computer Science and Engineering

Evaluation And Implementation of Modern Virtual Private Network Protocol for The GNRC Network Stack

By

Le Hoang Dang Nguyen

Matriculation number: 17028

BACHELOR THESIS

Submitted in partial fulfillment of the requirements for the degree of Bachelor Engineering in study program Computer Science, Vietnamese - German University, 2024

First supervisor: Prof. Dr. Oliver Hahm
Second supervisor: Dr. Tran Thi Thu Huong

Binh Duong, 2024

Declaration

I hereby declare that this thesis is a product of my own work, unless otherwise referenced. I also declare that all opinions, results, conclusions and recommendations are my own and may not represent the policies or opinions of Vietnamese – German University.

Le Hoang Dang Nguyen

Acknowledgements

I want to thank you.

Abstract

Abstract go here

Table of Contents

List of Figures

Listings

Abbreviations

6LoWPAN IPv6 over low-power wireless personal area networks.

AEAD Authenticated Encryption with Associated Data.

AES Advanced Encryption Standard.

API Application programming interface.

CPU Central Processing Unit.

CSMA/CA Carrier Sense Multiple Access with Collision Avoidance.

DH Diffie-Hellman.

ECDH Elliptic Curve Diffie-Hellman.

HKDF HMAC-based Key Derivation Function.

HMAC Hash-based Message Authentication Code.

ICMP Internet Control Message Protocol.

ICMPv6 Internet Control Message Protocol version 6.

IoT Internet of things.

IP	Internet protocol.
IRS	Interrupt Service Routine.
ISM	Industrial, Scientific, and Medical.
L2	Layer 2.
LIFO	Last In, First Out.
LoWPAN	Low-power wireless area networks.
LQI	Link Quality Indication.
MAC	Message Authentication Code.
MCU	Microcontroller Unit.
MMU	Memory Management Unit.
MPU	Memory Protection Unit.
OSI	Open Systems Interconnection.
PSK	Pre-shared Symmetric Key.
RAM	Random Access Memory.
RFID	Radio frequency identification.
ROM	Read-only Memory.
RSSI	Received Signal Strength Indicator.
RTT	Round Trip Time.

TCP	Transmission Control Protocol.
TDMA	Time-division multiple access.
TISCH	Time Slotted Channel Hopping.
UDP	User Datagram Protocol.
VPN	Virtual private network.

Chapter 1

Introduction

1.1 Motivation

The modern world is showing great interest in the Internet of things (IoT) with the expectation of a significant increase in the number of the internet-connected devices [Ori15]. In 2016, forecasts by Statista estimated that there would be more than 75 billions IoT devices in used [16]. However, this rapid, unregulated expansion also comes with serious privacy and security challenges [Taw+20]. One such vector of security attack could be the router, the machine standing between the IoT devices and the internet. In 2018, Cisco researcher found out a new malware called VPNFilter affecting 500,000 networking devices worldwide, including Mikrotic routers [Lar18]. With such a malware, the hackers may have the ability to hijack any traffic between the device and the internet outside. Thus, better security mechanisms need to be enforce to guarantee an end-to-end (E2E) secure communication of the IoT devices against such untrustworthy actors.

Virtual private network (VPN) could possibly be one of the approachs. VPN does not only ensure the encryption of network traffic, but also hides away the original addresses coming from the devices, making network communication private to only the owners. However, adapting normal VPN technology for IoT devices faces serious physical limitation of such devices. These devices are usually small, with severe constraints on power, memory, and processing resources [BEK14]. On the otherhand, existent VPN protocols like OpenVPN or IPSec are complex, fragile against faulty misconfiguration and have many known vulnerabilites [Wu19]. There has been recent effort on adapting IPSec protocol for the constrained enviroments by reducing the requirements for an implementation to the minimum [MG23] and developing a compression framework for the protocol [Mig+24].

Another consideration to introduce the VPN into the IoT infrastructure is Wireguard [Don20]. Wireguard is a modern VPN protocol designed for Linux with simplicity and security in mind. While achive multiple security properites, the Wireguard messages still has low overhead and a potential low memory footprint, making it desirable for the class of constrained devices.

The goal of this thesis is to evaluate the merits that Wireguard can bring into the Internet of Things fields and propose an implementation of the protocol for GNRC - a network stack designed specifically for the IoT enviroment.

1.2 Organization

The thesis is organized as follows:

- Chapter 2 provides the background knowlegde of IoT, Wireless embedded internet and IPv6 over low-power wireless personal area networks (6LoWPAN).
- Chapter 3 gives an overview on the GNRC network stack, and RIOT operating system that runs on top of it.
- Chapter 4 explains in details the Wireguard protocol.
- Chapter 5 describes the security features of Wireguard and evaluate the benefits they can add to the Internet of Things.
- Chapter 6 focuses on the design and important implementation apsects of Wireguard for GNRC.
- The final chapter concludes and summarizes the main works of the thesis.

Chapter 2

Wireless Embedded Internet

This chapter provide the definition and overview of the wireless embedded internet. It covers It covers architectures specifically developed for IoT applications, hightlights the 6LoWPAN protocol stack that allows the integration of the Internet protocol (IP) stack onto this type of network, and reviews 802.15.4, a common protocol in embedded context.

2.1 Overview

The Internet of Things (IoT) comprises of all embedded devices and networks that are natively IP-enabled and connected to the Internet, such as sensors, machines, and Radio frequency identification (RFID) readers, alongside the services monitoring and controlling these devices [SB09]. A subset of IoT, the Wireless Embedded Internet consists of low-power, resource-limited wireless devices connected through standards like IEEE 802.15.4. Integrating standard Internet protocols with such networks presents several challenges:

Power and duty-cycle: The IP-enabled devices should always be connecting, contradicting the low-duty-cycle nature of the battery-powered wireless devices.

Multicast: Wireless embedded radio technologies like IEEE 802.15.4, do not generally support multicast, and flooding wastes power and bandwidth in such network. However, Multicast is an important operation for many IPv6 features such as address auto-configuration [NJT07].

Limited bandwidth and frame sizes: Bandwith and frame size inside a low-power wireless embedded radio network are limited, with only 20-250 kbits/s and 40-200 bytes correspondingly. For example, the frame size for IEEE 802.15.4 standard has a 127-byte frame size, with layer-2 payload sizes as low as 72 bytes. The minimum frame size for standard IPv6 is 1280 bytes [DH17], hence fragmentation is required.

Reliability: Standard Internet Protocols are not optimized for low-power wireless networks. For example, TCP is not able to differentiate between packets dropped by congestion or lost on wireless links. Node failure, energy exhaustion, and sleep duty cycles can also incur unreliability in wireless embedded networks.

To tackle these issues, 6LoWPAN [Mon+07] was developed, enabling IPv6 and its related protocols to function effectively in wireless embedded networks. IPv6's simple header structure and hierarchical addressing make it ideal for use in these constrained environments.

2.2 The 6LoWPAN Architecture

According to Zach Shelby and Carsten Bormann, the Wireless Embedded Internet is formed by connecting islands of wireless embedded devices, with each island functioning as a stub network within the Internet [SB09, p. 13]. A stub network is one where IP packets are either sent to or received from, but it does not serve as a transit point for other networks. The 6LoWPAN architecture is illustrated in Figure 2.1. In this context, the 6LoWPAN architecture consists of low-power wireless area networks (LoWPANs), which operate as IPv6 stub networks. Each LoWPAN is a set of 6LoWPAN nodes, sharing a common IPv6 address prefix (the first 64 bits of an IPv6 address), with the interconnection between the LoWPANs achieved through the edge router. There are 3 different kinds of LoWPANs:

- An **Ad hoc LoWPAN** operates independently without the connection to the internet.
- A **Simple LoWPAN** connects to another IP network via an edger router.
- An **External LoWPAN** comprises LoWPANS of multiple edge routers along with a backbone link connecting them.

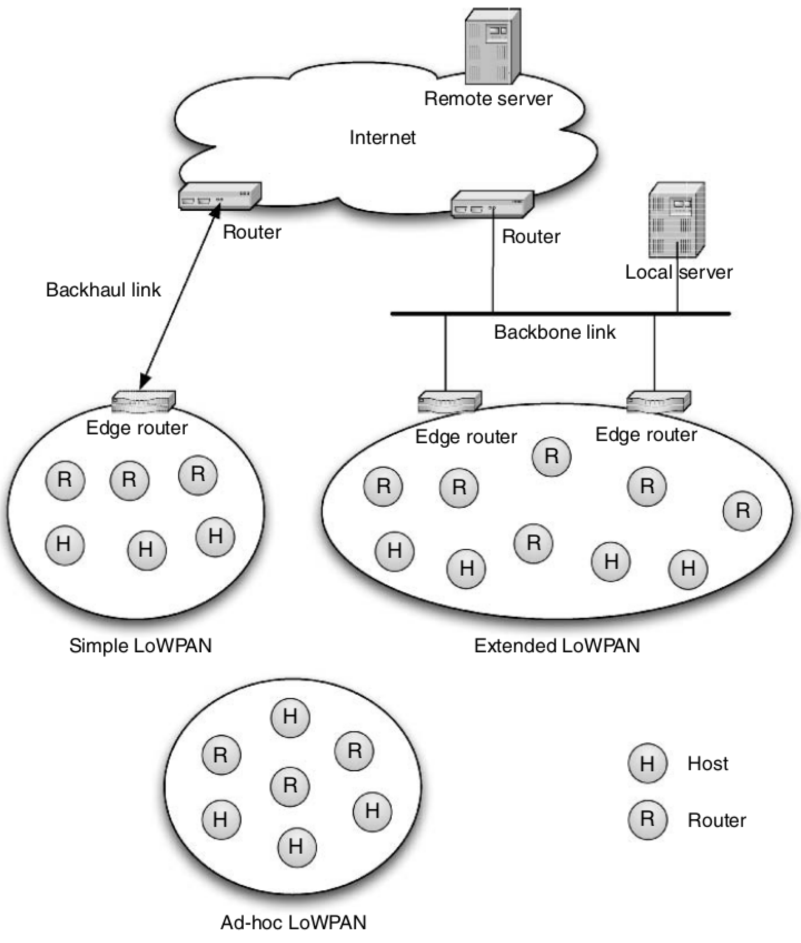


Figure 2.1: The 6LoWPAN architecture, see [SB09, p. 14]

LoWPANs are connected to other IP networks via edge routers, as illustrated in Figure 2.1. The edge router plays a key role by routing traffic to and from the LoWPAN, managing 6LoWPAN compression, handling Neighbor Discovery within the network, and other network management features. Each node in a LoWPAN could either be a host, an edge router, or a node routing between other nodes. The shared common IPv6 prefix within the LoWPAN is advertised by edge routers or is pre-configured on each node. An edge router keeps a list of registered nodes that are accessible through its network interface inside the LoWPAN.

To enter a 6LoWPAN, a node sends a Router Solicitation message to obtain the IPv6 prefix unless it has been statically configured. Upon receiving the prefix, the node generates a unique global IPv6 address and registers this address with the edge router of the LoWPAN. This allows the edge router to make informed routing decisions for traffic entering and exiting the LoWPAN, as well as to facilitate neighbor discovery within the 6LoWPAN. The edge router must update the list of registered nodes regularly, as addresses expire after a configurable period. A longer expiration time helps reduce a node's power consumption, while a shorter expiration time accommodates rapidly changing network structures. These processes are defined in the dedicated neighbor discovery protocol for 6LoWPAN [Bor+12]. LoWPAN nodes can travel freely within and among multiple LoWPAN networks, and they may participate in several LoWPANs simultaneously. Communication between a LoWPAN node and an external IP node occurs in an end-to-end manner, similar to interactions between standard IP nodes.

In an extended LoWPAN, multiple edge routers are integrated into the same LoWPAN, sharing the same IPv6 prefix. These edge routers are interconnected through a common backbone link. When a node moves between edge routers, it must register with the edge router it can access, but it retains its IPv6 address. Communication between edge routers regarding neighbor discovery is handled over the backbone link, which reduces messaging overhead. This extended LoW-

PAN architecture allows a single LoWPAN to cover larger areas.

An ad-hoc LoWPAN works in the same manner as a simple LoWPAN, but without the link to other IP networks. Instead of an edge router, a node will act as a simple edge router, handle unique local address generation [HH05] and provide the neighbor discovery registration feature to other nodes.

2.3 6LoWPAN Protocol Stack

Figure 2.2 depicts the IPv6 protocol stack with 6LoWPAN in comparison to a standard IP protocol stack and the corresponding five layers of the Internet model. This Internet model connects a wide range of link-layer technologies with various transport and application protocols.

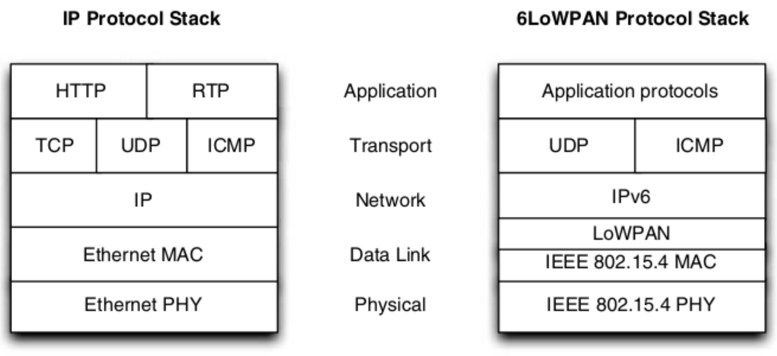


Figure 2.2: IP and 6LoWPAN stack, see [SB09, p. 16]

The IPv6 protocol stack with 6LoWPAN (sometimes called the 6LoWPAN protocol stack) is nearly identical to a conventional IP stack, with a few notable differences. Primarily, 6LoWPAN only supports IPv6, for which a small adaptation layer—the LoWPAN adaptation layer—has been defined to optimize IPv6 for IEEE 802.15.4 and similar link layers, as detailed in [Mon+07]. In practice, many implementations of the 6LoWPAN stack in embedded devices combine the LoWPAN adaptation layer with IPv6, allowing them to be displayed together as part of the network layer.

The User Datagram Protocol (UDP) [Pos80] is the most commonly used transport protocol with 6LoWPAN, and it can be compressed using the LoWPAN format. The Transmission Control Protocol (TCP) is less frequently used due to concerns about performance, efficiency, and complexity, although there have been recent efforts on guidance to use and implement TCP for the IoT [GCS21]. The Internet Control Message Protocol version 6 (ICMPv6) [GC06] is utilized for control messaging, including functions like Internet Control Message Protocol (ICMP) echo requests, destination unreachable messages, and Neighbor Discovery. While many application protocols are application-specific and in binary format, there is a growing availability of more standardized application protocols.

2.4 IEEE 802.15.4

Established by the IEEE, the IEEE 802.15.4 standards define low-power wireless radio techniques and specify the physical and media access control layers that serve as the foundation for 6LoWPAN. The IEEE 802.15.4-2011 version of the standards includes features such as access control via CSMA/CA, optional acknowledgments for retransmission of corrupted data, and 128-bit Advanced Encryption Standard (AES) encryption at the link layer. It offers addressing modes that utilize both 64-bit and 16-bit addresses with unicast and broadcast capabilities. The payload of a physical frame can reach up to 127 bytes, with 72 to 116 bytes of the usable payload after link-layer framing, depending on different addressing and security options [SB09, Appendix B.1].

Star and point-to-point network topologies are supported by IEEE 802.15.4. The MAC layer can operate with CSMA/CA in the beaconless mode. In beacon-enabled mode, TDMA/TISCH for media access is utilized. The number of nodes, the length of the transmitted messages, and the level of radio interference within the ISM band significantly influenced the average packet loss in IEEE 802.15.4 [Shu+07]. While the use of acknowledgments at the link layer enhances reliability, it complicates the estimation of packet round-trip times.

Chapter 3

RIOT-OS and The GNRC Network Stack

3.1 RIOT Operating System

RIOT, the friendly operating system for the Internet of Things, is a real-time operating system, specifically designed for low-end IoT devices with a minimal memory in order of $\approx$ 10K Byte [Bac+18]. It can run on devices with neither memory management unit (MMU) nor memory protection unit (MPU).

Under the distribution of LGPLv2.1 License, RIOT is free and open-source software, meaning it can be used and distributed by anyone. Furthermore, this license allows the linkage of RIOT with proprietary software and supports the ability to be customized by the end users.

The design objectives of RIOT focus on several key areas: optimizing resource usage such as RAM, ROM, and power consumption;

supporting a broad spectrum of configurations, from 8-bit to 32-bit MCUs, and accommodating various boards and use cases; reducing code duplication across different setups; ensuring most of the code is portable across supported hardware; offering user-friendly software platform; and enabling real-time capabilities. To realize these goals, one of the principles that the RIOT follows is modularity.

RIOT is organized into software modules that are combined at compile time, centered around a kernel offering minimal functionality. This modular approach allows the system to be built in a way that includes only the necessary modules for a given use case. As a result, both memory usage and system complexity are kept to a minimum in practical deployments. The code structure of RIOT is illustrated in Figure 3.1:

- **core** provides the kernels and basic data structures like linked lists, LIFOs, and ringbuffers.
- Four parts of hardware abstractions:
  1. **cpu** implements functionalities of microcontroller.
  2. **boards** selects, maps and configures the used CPU and drivers.
  3. **drivers** implement the device drivers.
  4. **periph** provides unified access to microcontroller peripherals and is used by device drivers.
- **sys** implements libraries beyond kernel features, such as cryptog-

raphy, networking, and file system.

- **pkg** contains third-party libraries which do not exist within the main code repository.

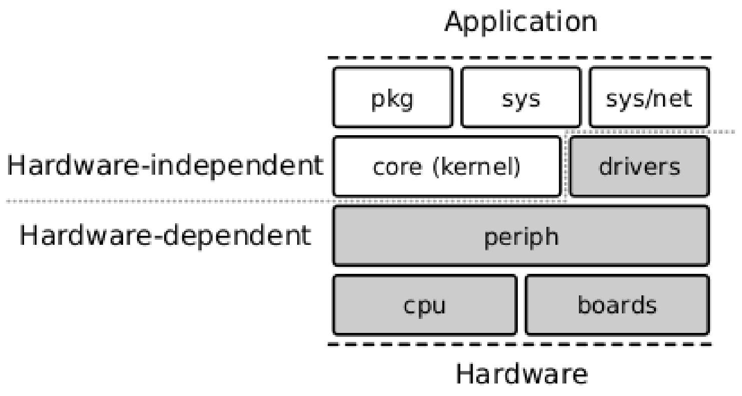


Figure 3.1: Structure elements of RIOT, see [Bac+18, p. 3]

Multi-threading is a builtin feature for RIOT to offer several benefits: (a) clear logical separation between different tasks, (b) straightforward task prioritization, and (c) easier integration of external code [Bac+18, p. 4]. Various synchronization primitives, such as mutex, semaphore, and message passing (**msg**) are provided by RIOT kernel. Multi-threading can also be optional in the case of extremely low-memory usage of the application.

RIOT's kernel employs a scheduler that uses fixed priorities and preemption with O(1) operations, enabling soft real-time capabilities. Specifically, the time required to interrupt and switch between threads is bounded by a small upper limit, as operations like context saving, selecting the next thread, and context restoring are deterministic. The system follows a class-based run-to-completion scheduling policy, where the highest-priority active thread is executed and can only be interrupted by interrupt service routines (ISRs). This scheduler allows RIOT to effectively prioritize tasks, ensuring that high-priority events can preempt lower-priority tasks as needed.

RIOT's scheduler operates in a tickless manner, meaning it does not rely on CPU time slices or periodic system timer ticks. As a result, the system remains in a low-power state unless an actual event occurs, such as an interrupt triggered by hardware. Wake-up events can be initiated by a transceiver receiving a packet, timers expiring, buttons being pressed, or similar activities. When no threads are in a running state and no interrupts are pending, the system automatically switches to the idle thread, which has the lowest priority. The idle thread, in turn, transitions the system into the most energy-efficient mode available thereby reducing energy consumption.

3.2 GNRC

3.2.1 Overview & Architecture

The GNRC (short for generic) network stack is the default network stack for RIOT [Len16]. It was designed as a replacement for the previous monolithic and unmaintainable network stack of RIOT [Bru16]. The old stack was lack of unified interfaces, making it difficult to achieve modularity, testability, and extensibility. Furthermore, as most of the networks were using their own buffers, the memory footprint was increased, creating a need for significant data copying between layers, thus leading to a reduction in overall performance. These limitations drove the creation of the "gnrc" network stack to replace the previous one.

The design objectives of the generic network stack encompass a low memory footprint, full-featured protocols, modular architecture, support for multiple network interfaces, parallel data handling and customization during compilation time.

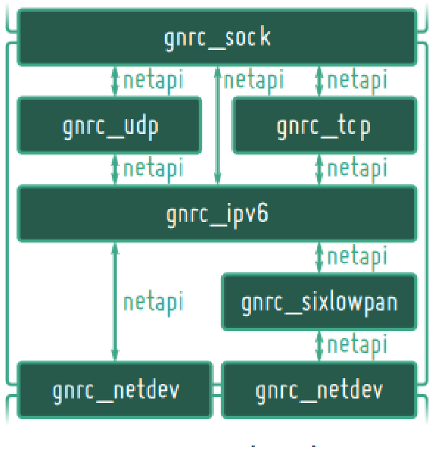


Figure 3.2: GNRC Network Stack

Figure 3.2 illustrates the architecture of the GNRC network stack. Using multi-threading features and RIOT's thread-targeted IPC, every networking module runs in its own thread, communicating with other modules via the message queue. The messages passing between the network threads follow a clearly defined format called "*netapi*" in GNRC. A network registry called "*netreg*" is used for a network thread to search for a module that is interested in receiving the next packet, using the packet's type. Modules that want to receive certain types can register with this registry. While traversing through the stack, a packet is stored in the centralized GNRC packet buffer "*pktbuf*". With this architecture, GNRC attains the required modularity and simplifies testing. It also offers an easy way to prioritize different components of the stack, by assigning different priorities to the threads running the protocols and allowing the operating system's process scheduler to handle these priorities. The trade-off of this design is a performance penalty compared to straightforward function calls, as IPC involves context switches and context saves. However, this overhead is shown to be only one order of magnitude slower than a direct function call on real IoT hardware [Pet+15, section 4.1].

3.2.2 The Packet Buffer - pktbuf

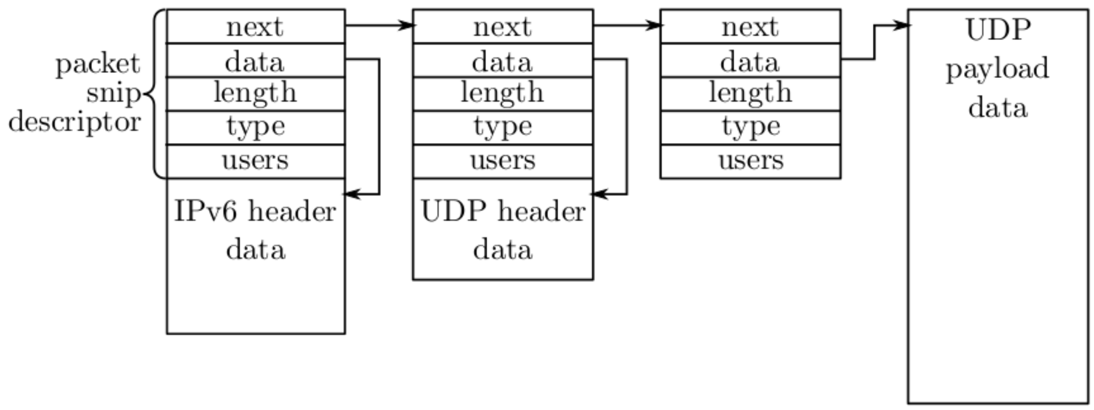


Figure 3.3: An example of a packet in transmission in the GNRC packet buffer, see [Len16, p. 22]

Packets in GNRC are stored in a packet buffer called *pktbuf*, consisting of variable-length segments known as "packet snips" (see Figure 3.3). These snips are typically used to distinguish between different headers and payloads. The packet data, along with the structures describing the packet snips, is stored in arrays allocated in static memory, although the API can theoretically support dynamic allocation on the memory heap. Packet snips can be labeled with a type to indicate the content of the packet. To create a packet snip in pktbuf, a user can use the function `gnrc_pktbuf_add()`, which takes all user-controllable fields of a packet snip (`next`, `data`, `length`, and `type`) as arguments and returns the new snip.

With minimal packet duplication across the stack, data is being copied or moved as infrequently as possible from the network interface to the application, ideally only once at each end-point of the stack and vice versa. This approach allows the structures that describe the packet (referred to as the "packet snip descriptor" in Figure 3.3) to be stored independently of the actual data, enabling the marking of a packet header without requiring to move data around.

To ensure safety in concurrency, a reference counter (`user`) is maintained in a packet snip to keep track of the using threads. The APIs that handle the increment and the decrement of the counter are `gnrc_pktbuf_hold()` and `gnrc_pktbuf_release()`. The packet is removed from the packet buffer when the counter hits 0.

When a thread starts to write to a packet and the reference counter is greater than one, the packet snip and all its next pointer will be cloned using a copy-on-write approach, resulting in the decrement of the original snip's reference counter by one. For minimal duplication, the pointers reaching the current snip will be copied, creating a tree structure for the packet within the packet buffer, hence reversing the order of packet snips for a received packet. The packet snips now starts with the payload and end with the header of the lowest layer, whilst a packet in transmission retains the order in which it will be sent, reducing the number of pointers needed for the tree structure. Nevertheless, it's crucial to note that data will be typically kept in the order it was received by the packets in reception. The only thing to be reversed is the packet descriptor list that marks the data. When a packet (i.e., the first snip in a packet snip list) is released by a thread, its own snips will be removed from the packet buffer, with any of its copy snips still persist. The implementation of this feature is implemented in the `gnrc_pktbuf_start_write()` function.

This certain way of storing packets brings the merits of parallel data handling and low-memory footprint to GNRC, while avoiding duplication across the network stack.

3.2.3 GNRC's Module Registry - netreg

The *netreg* API provides a central directory to the generic network stack. When the thead of a network module is created, it also registered in *netreg* using its thread PID and "NETTYPE" - the kind of information that it is interested in. For example, an IPv6 module registers with its PID and type "NETTYPE_IPV6". If the UDP module wants to dispatch the packet down the stack, *netreg* would be used to search for threads registered with type “NETTYPE_IPV6”. Using *netapi*, the module then send a pointer to the pktbuf-allocated packet to every interested thread.

3.2.4 Network Interfaces

In GNRC, network interfaces run in their own threads and communicate with the network layer via *netapi.* Within a network interface thread, MAC protocol are implemented and interact directy with network device drivers through the *netdev* API. This direct access are required since some link-layer protocols, such as TDMA-based MAC, need minimal delay when acessing the device. To reduce the overhead of dealing with L2 headers, the header formats of each network interface's L2 protocol are converted into netif *netif* header - a general interface header format. Source and destination L2 addresses, header's length, along side with additional link metrics utilized for routing protocol, such LQI and RSSI are included is this *netif* header. A unified conversion API for popular L2 protocols ensures that no redundant porting efforts are necessary.

Chapter 4

Concepts of Wireguard

This chapter provide the core concepts of Wireguard protocol, especially in its handshake and timer state machine. Section 4.1 gives a high-level overview of the traits of the protocols, and the cryptography primitives that they build upon. Section 4.2 details the fondational handshake protocol that Wireguard's key exchange protocol builds upon. Section 4.3 defined all messages types of Wireguard and section 4.4 descibes the several timers constraint of Wireguard.

4.1 Protocol & Cryptography

4.1.1 Overview

Wireguard works as an encrypted IP network tunnel, resides at the layer 3 of the OSI layer, and uses UDP as its transport protocol. The establishment of a secure session before any transmission of data is via 1-RTT key-exchange handshake protocols. This handshake protocol is designed based on a Trevor Perin's noise handshake pattern [Per].

After the handshake, the transported IP payload are protected using ChaCha20-Poly1305 Authenticated Encryption with Associated Data (AEAD) [NL18].

In Wireguard, the endpoints of communication have no role of server and client, but works in a peer-to-peer style. At any point in time, a peer can have a role of an **initator**, start to send a handshake initiation message to create a secure channel with a **responder**. If the secure channel is not active for some amount of time, the initiator peer can change to a responder in the event of a the previous responder tries to initiate a new handshake, leading to the establishment of a new session between 2 peers. Figure 4.1 illustrates this handshake flow with periodic key rotation.

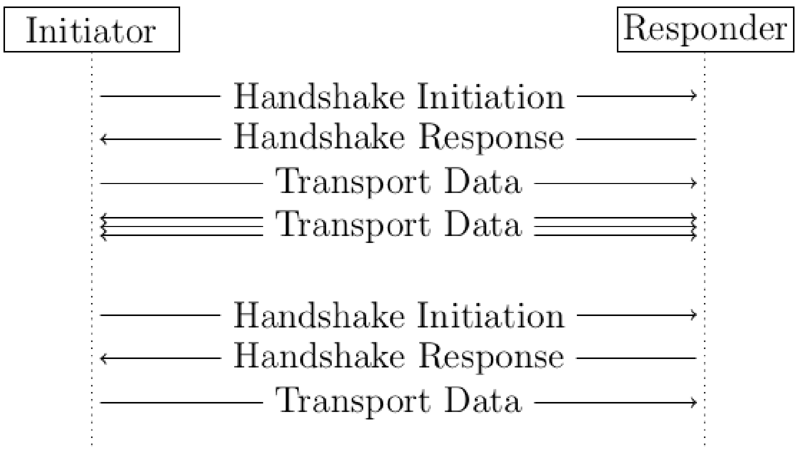


Figure 4.1: After the handshake, both peers can send data towards each other. After a duration, rekeying occurs to create a new session. See [Wu19, p. 7]

A static 32-byte Curve25519 [Ber06] public key are used to identified an endpoint inside the tunnel. Without the proof of sender's knowledge on this public key, an endpoint will never respond. Thus, adversaries can't probe any system protected by Wireguard, using port scanning if they do not know the long-term static public key.

The Wireguard handshake protocol can be considered as a 1.5 round-trip time (1.5-RTT) handshake [Wu19]. Upon the reception of the handshake response after sending a handshake initiation, the initiator can promplty begin delivering encrypted payload. To safeguard against replay attacks, the responder must wait to send encrypted data messages until it has received an encrypted data message from the initiator, which serves as an acknowledgment of the handshake response.

During a handshake, new ephemeral key pairs will be generated by both parties, with the private key being wiped after the handshake, assuring the forward secrecy of the session. Each session is bound by fixed lifetime and the number of data messages that can be sent. When either of the limits is reached, the derivation of new session keys are required if the peers want to extend the communication. The impact of compromised session key is also mitigated by frequent periodic key rotation.

There is no direct shutdown signal within the protocol, only session removal after a fixed duration. Even if one side has terminated its VPN tunnel, the peer would remain unaware and keep forwarding data.

Wireguard also provides an optional pre-shared symmetric key (PSK) mode, where any pairs of peer pre-share a 256-bit symmetric encryption key between themselves, to assure the post-quantum security as long as the PSK never leaks out. This protection is against the idea that if adversaries may be recording all the traffic for a long time, until the quantum computer exists. With PSK, despite the fact they maybe break all Curve25519-encrypted traffic, but not the ones that include a PSK.

Another adversary model that Wireguard mititage is the denial of service through CPU exhaustion denial of services attack [Hen11, p. 268]. This mechanism is achieved via an encrypted cookie, when a peer is currently under load. This will be explained more in section 5.3.

At last, Wireguard is cryptographically opinionated. The protocol intentionally does not provide any form of flexibility when it comes to choosing the cryptography suite, hence a negation for cryptography algorithms does not exist. Specifically, Wireguard uses the following modern cryptography constructions:

Noise protocol framework A set of cryptographic handshake patterns that serve as building blocks for creating new secure protocols with authenticated key agreement.

Elliptic Curve Diffie-Hellman (ECDH) A Curve25519-based key-argeement protocol with small key size and simple requirements regarding key validation.

ChaCha20-Poly1305 a modern AEAD combining ChaCha20 [Ber08] stream cipher and Poly1305 [Ber05] authenticator to achieve authenticity and confidentiality.

XChaCha-Poly1305 an extended version of chacha20-poly1305 that supports 192-bit nonce. This is used for cookie encryption [Arc20].

HKDF The HMAC-based Key Derivation Function [KE10] to derive the encryption and decryption keys from the handshake state, keys and protocol messages.

BLAKE2 A fast and cryptographic hash function for message authentication code and is used by the HKDF [SA15].

4.1.2 Cryptokey Routing

The binding between peers and the allowed source IP addresses is the fundamental principle to build a secure VPN. As mentioned above, in Wireguard, peer must be identified by the 32-byte Curve25519 public key, allowing an association between the public key and the set of allowed IP addreses on the peer. This simple mapping is the concept behind the cryptokey routing table of Wireguard. A transmission of outbound packet will consult this table to search for the approriate public key, using the IP destination address of the packet. With inbound packets, after decryption and authentication, the source IP address need to be resolved to the same peer that have the same public key used in the secure session for decrypting the packet. In cryptokey routing table, each peer may optionally pre-specify an outer external IP address and UDP port of that peer's endpoint. If the endpoint is not specified, the external source IP of the machine will be used to determine the endpoint . This design also allow the peers to roam freely between different external IP addresses as the public is the main identification of the peer.

Combining the roaming and the cryptokey routing table, when the network interface needs to send the data out, the flow start with the inspection of the IP destination of the packet to find the corresponding peer and its public key (An ICMP "no route to host" will be returned in case of no peer). The packet is then encrypted with mentioned AEAD, prepended with extra headers and sent as a UDP packet to the internet. On the receiving flow, when a UDP packet is received, Wireguard finds the matching peer, decrypts the packet and updates the peers' endpoint. If the packet is either not an IP packet or there is no entry for source IP address of the packet inside the cryptokey routing table, the packet is droped. Otherwise, the packet will be dispatched to the layer above.

4.2 Noise

Noise [Per] is a framework to design secure protocol based on Diffie-Hellman(DH) key exchange [DH76]. With Noise, designers can construct new protocols that offer features such as identity hiding, mutual authentication, and forward secrecy for deriving shared symmetric keys. Furthermore, this framework has gone through formal analysis and verification [KNB18]. Although not all Noise protocols ensure these security guarantees, the IKpsk2 form used by WireGuard does. The rest of this section gives a general introduction to Noise, without focusing specifically on WireGuard.

A Noise protocol begins with the exchange of **handshake messages** between two parties to generate a shared secret key for the encryption of **transport messages**. The first one to send the handshake message is refered to as the **initiator**, while the recipient of this message is called the **responder**.

Each handshake in the Noise framework involves a long-term **static key pair** and/or **ephemeral key pair**. Every new handshake requires a fresh creation of the ephemeral keys. The static keys may be used for the purpose of authentication, when a peer has little information on the other's identity. Within the handshake, 256-bit **pre-shared symmetric key** can also be mixed in to bind the Noise session to a previous protocol-specific communication. Both the ephemeral and static keys are used in a chain of Diffie-Hellman computation to guarantee the forward secrecy. A cryptographic binding between the final shared secret and the handshake are achieved by the fact that all handshake messages and key material are used to derive the symmetric keys for handshake and the encryption of transport data.

In Noise, the sequential exchange that comprises a handshake message is called a **handshake pattern**. Each handshake pattern consists of **message pattern**, which is arranged from a set of pre-defined **tokens**. The tokens describes the modification to the cipher and handshake state, and data to read or write. The handshake state holds both local key pairs as well as the remote ephemeral and static public keys of peer. The cipher state is made of a key and a counter-based nonce for encryption, a handshake hash and a chaining key to derive the final symmetric key. Noise framework defines the following tokens for the message patterns:

- **e**: The sender creates a new ephemeral key pair, writes the public key into the message buffer, and hashes this key into the handshake hash. In a PSK handshake, the chaining key is mixed with the ephemeral public key.
- **s**: The sender encrypts the static public key and write the result to message buffer. if the handshake symmetric key exists, otherwise the plaintext of the public key will be written. The result will also be hashed into the handshake hash.
- **ee**, **es**, **se**, **ss**: DH computation is performed using the sender's ephemeral or static private key (determined by the first letter) and the receiver's ephemeral or static public key (determined by the second letter). The output is hashed into the chaining key, reseting the cipher state with a new key derived from the chaining key and a zero nonce.
- **psk**: A pre-shared symmetric key is hashed both into the chaining key and the handshake hash state. As the chaining key derives the symmetric key directly, it's essential to mix the nonce into

the chaining key for the generation of a randomized symmetric encryption key. This mitigates the key resuse vulnerability. The ephemeral public key provided by the "e" token serves as this nonce. Consequently, in valid Noise protocols that utilize the **psk** token, encryption must be preceded by the **e** token. An example of invalid pattern is "**psk**, **s**, **e**". Here, **s** establishes the encrypted static public key with a fixed symmetric key derived from the PSK.

The order of tokens within a message pattern is crucial. For instance, "**s**, **ss**" differs from "**ss**, **s**". In the former, the sender's long-term static public key is presented in plain text ("**s**"), followed by a DH computation between the static keys of both parties ("**ss**"). Any subsequent messages following this pattern will be encrypted with derived keys. In contrast, the latter pattern ("**ss**, **s**") indicates that the sender's long-term static public key is encrypted, using keys derived from a hash that incorporates the DH computation between the two parties' static keys ("**ss**").

Before the handshake, protocols may already possess information about the peer's static and/or ephemeral public keys. This knowledge is represented by one of the **pre-message patterns**: "**e**," "**s**," "**e**, **s**," or none if there are no pre-messages. The "**s**" pre-message pattern signifies knowledge of the other party's static public key, while the "**e**" pre-message pattern, used in a fallback scenario, indicates a previously shared fresh ephemeral public key. None of these public keys will be published again during the handshake; they remain implicit. The pre-message patterns and message patterns together create a **handshake pattern** that includes:

- A pre-message pattern for the initiator, which conveys information about the responder's public keys.
- A pre-message pattern for the responder, which conveys information about the initiator's public keys.
- A chain of message patterns the form the handshake messages. Each message pattern has a specific direction that influences how the tokens are interpreted. For instance, for the initiator, the $\rightarrow$ **s** notation signifies sending the initiator's static public key, whereas the responder interprets it as reading the initiator's static public key.

Without knowledge of the exact public keys from the pre-message patterns, it will be impossible to derive the correct symmetric keys during the handshake. Consequently, this will lead to a failure of the handshake due to mismatched handshake states.

Noise specifies several standard handshake patterns, each with different requirements, properties, and security guarantees. Transmission of user data can happen after a completed handshake or inside the buffer of message pattern within a handshake. Nevertheless, sending such data beforehand can lead to reduced authenticity and confidentiality.

A instantiation of a concrete Noise protocol requires a choosing of **handshake pattern**, a **DH function**, a **cipher function**, and a **hash function**. A prologue byte pattern can be established to associate the cryptographic keys with this value. Following is the instantiation of Wireguard:

- **handshake pattern** IKpsk2
- **DH function** X25519
- **cipher function** ChaCha20-Poly1305
- **hash function** BLAKE2s
- **prologue** WireGuard v1 zx2c4 Jason@zx2c4.com

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

- **I**: static public key is **i**mmediately transmitted to the responder, despite absent or reduced 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 name for the Noise protocol is concatenated from the string Noise, the handshake pattern, and functions, separated by an underscore (_). Hence, for Wireguard, the full Noise protocol name is: Noise_IKpsk2_25519_ChaChaPoly_BLAKE2s. The details of the IKpsk2 is presented in section 4.2.1.

4.2.1 IKpsk2 Handshake Pattern

The steps to execute the IKpsk2 handshake pattern used in Wireguard will be elaborated in this section. Following is the explaination for several operators, functions and symbols:

- S_i^{pub}: the initiator's 32-byte static public key.
- E_r^{priv}: the responder's 32-byte ephemeral private key.
- Q: optional 32-byte pre-shared symmetric key.
- T_i^{send}, T_r^{recv}: the initator's 32-byte transport data symmetric key for sending, and the responder's 32-byte transport data symmetric key for receiving.

The symbols for handshake and cipher state:

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

The functions and constants that will be used:

- **ECDH(private key, public key)**: the Elliptic Curve Diffie-Hellman function [Ber06], returning the 32-byte shared secret from 32-byte public key and 32-byte private key.
- **ECDH_GEN()**: returns a key pairs consisting of a random Curve25519 private key and the related public key.

- **AEAD_ENC(cleartext, aad, key, nonce)**: applies ChaCha20-Poly1305 AEAD encryption on a variable-length cleartext message, using a 256-bit key, a 96-bit counter after prepending the 64-bit nonce with 4 bytes of zero and arbitary length additional authenticated data to return the ciphertext including a 16-byte authentication tag.

- **AEAD_DEC(ciphertext, aad, key, nonce)**: the reverse of **AEAD_ENC** that authenticates and decrypt the ciphertext to return the original plaintext.

- **Hash(data)**: hashes the variable-length data with BLAKE2s function to return 32 bytes of output.

- **KDF$_n$(key, input)**: derives the n 32-bytes keys with HKDF, using unkeyed BLAKE2s as the hash function for the HMAC [PP09, section 13.2]. The construction of HKDF is as follow:

$$\begin{aligned}
PRK &= HMAC(key, input) \\
T_1 &= HMAC(PRK, 0x1) \\
T_2 &= HMAC(PRK, T_1||0x2) \\
&\dots \\
T_n &= HMAC(PRK, T_{n-1}||n)
\end{aligned}$$

The IKpsk2 handshake pattern in compact notation are shown Figure 4.2. The details of the computation follows below, as a reiteration in [Wu19, p. 15].

IKpsk2:

1. $\leftarrow s$

   $\ldots$

2. $\rightarrow e, es, s, ss$
3. $\leftarrow e, ee, se, psk$

Figure 4.2: Compact notation for IKpsk2. The dots distinguish between pre-messages and messages. See [Per, p. 36]

The hashing of protocol name and prologue begins the handshake computation.

1. $h_0 = \textbf{Hash}(\text{“Noise IKpsk2 25519 ChaChaPoly BLAKE2s”})$

2. $ck_0 = h_0$

3. $h_1 = \textbf{Hash}(h_0 \parallel \text{“WireGuard v1 zx2c4 Jason@zx2c4.com”})$

Next, the handshake messages and pre-messages will be processed. Since there are message tokens that have different interpretation depending on the initator and the responder, I: and R: will be prefixed before the line to constratin them to the initator or responder correspondingly. Following are how to interpret the three (pre-)messages from Figure 4.2:

0. ← s: a pre-message from responder to initiator, with “s” as the previously out-of-band exchanged static public key S_r^{pub}.

4. $h_2 = \textbf{Hash}(h_1 \parallel S_r^{pub})$

1. → e, es, s, ss: the initiator’s message to the responder:

- “e”: the initator creates an ephemeral key pair, transmits the public key E_i^{pub} to the responder. E_i^{pub} is mixed into the handshake hash and the chaining key because of the PSK.

  5. I: $(E_i^{priv}, E_r^{pub}) = \textbf{ECDH\_GEN()}$; write E_i^{pub}

     R: read E_i^{pub}

  6. $h_3 = \textbf{Hash}(h_2 \parallel E_i^{pub})$

  7. $ck_1 = \textbf{KDF}_1(ck_0, E_i^{pub})$

- “es”: the responder calculates the $es = \textbf{ECDH}(S_r^{priv}, E_i^{pub})$. The same shared secret is calculated as $es = \textbf{ECDH}(E_i^{priv}, S_r^{pub})$ for the initiator.

  8. I: $(ck_2, k_0) = \textbf{KDF}_2(ck_1, \textbf{ECDH}(E_i^{priv}, S_r^{pub}))$

     R: $(ck_2, k_0) = \textbf{KDF}_2(ck_1, \textbf{ECDH}(S_r^{priv}, E_i^{pub}))$

- “s”: the initiator transmits its encrypted static public key S_i^{pub}. On failure of decryption, the handshake is aborted.

  9. I: msg = $\textbf{AEAD\_ENC}(S_i^{pub}, h_3, k_0, 0)$; write msg

     R: $S_i^{pub} = \textbf{AEAD\_DEC}(\text{msg}, h_3, k_0, 0)$; abort on failure

  10. $h_4 = \textbf{Hash}(h_3 \parallel msg)$

- “ss”: the responder calculates the $ss = \textbf{ECDH}(S_r^{priv}, S_i^{pub})$. The same shared secret is calculated as $ss = \textbf{ECDH}(S_i^{priv}, S_r^{pub})$ for the initiator.

  11. I: $(ck_3, k_1) = \textbf{KDF}_2(ck_2, \textbf{ECDH}(S_i^{priv}, S_r^{pub}))$

      R: $(ck_3, k_1) = \textbf{KDF}_2(ck_2, \textbf{ECDH}(S_r^{priv}, S_i^{pub}))$

- Add an encrypted (possibly) empty payload to the message buffer in the handshake. If the decryption fails for the responder, it stops the handshake. For Wireguard, the payload is the current timestamp.

12. I: c = **AEAD_ENC**(time, h_4, k_1, 0); write c

    R: time = **AEAD_DEC**(c, h_4, k_1, 0); abort on failure

13. $h_5 = \textbf{Hash}(h_4 \parallel m)$

2. ← e, ee, se, psk: the responder's message to the initiator:

- "e": the responder creates an ephemeral key pair, transmits the public key E_r^{pub} to the initator. E_r^{pub} is mixed into the handshake hash and the chaining key because of the PSK.

  14. R: $(E_r^{priv}, E_i^{pub}) = \textbf{ECDH\_GEN()}$; write E_r^{pub}

      I: read E_r^{pub}

  15. $h_6 = \textbf{Hash}(h_5 \parallel E_r^{pub})$

  16. $ck_4 = \textbf{KDF}_1(ck_3, E_r^{pub})$

- "ee": the responder calculates the $ee = \textbf{ECDH}(E_r^{priv}, E_i^{pub})$. The same secret is calculated as $es = \textbf{ECDH}(E_i^{priv}, E_r^{pub})$ for the initiator.

  17. R: $ck_5 = \textbf{KDF}_1(ck_4, \textbf{ECDH}(E_r^{priv}, E_i^{pub}))$

      I: $ck_5 = \textbf{KDF}_1(ck_4, \textbf{ECDH}(E_i^{priv}, E_r^{pub}))$

- "se": the responder calculates the $se = \textbf{ECDH}(E_r^{priv}, S_i^{pub})$. The same secret is calculated as $se = \textbf{ECDH}(S_i^{priv}, E_r^{pub})$ for the initiator.

  18. R: $ck_6 = \textbf{KDF}_1(ck_5, \textbf{ECDH}(E_r^{priv}, S_i^{pub}))$

      I: $ck_6 = \textbf{KDF}_1(ck_5, \textbf{ECDH}(S_i^{priv}, E_r^{pub}))$

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

19. $(ck_7, \tau, k_2) = \mathbf{KDF}_3(ck_6, Q)$

20. $h_7 = \mathbf{Hash}(h_6 \parallel \tau)$

- Add an encrypted (possibly) empty payload to the message buffer in the handshake. If the decryption fails for the responder, it aborts the handshake.

21. R: c = **AEAD_ENC**(m, h_7, k_2, 0); write c
    I: m = **AEAD_DEC**(c, h_7, k_2, 0); abort on failure

22. $h_8 = \mathbf{Hash}(h_7 \parallel m)$

At last, the handshake ends with the derivation of the symmetric keys for transport data encryption, using HKDF with the chaining key and a empty string ϵ.

23. I: $(T_i^{send}, T_i^{recv}) = \mathbf{KDF}_2(ck_2, \epsilon)$
    R: $(T_r^{send}, T_r^{recv}) = \mathbf{KDF}_2(ck_2, \epsilon)$

This wraps up the execution and overview of the Noise IKpsk2 protocol. Next section will delve into how Wireguard uses this pattern for its payload.

4.3 Wireguard Messages

This sections details the underlying Wireguard messages that realizes the Noise handshake protocol, protects again denial of service attacks and encapsulates the IP packet in a robust tunnel. In Wireguard, there are four kinds of message, all of which start with a single type byte, followed by three zero reserved bytes. This layout allows the implementation to read the first 4 bytes of the message as a little-endian integer.

type := 0x1 (1 byte)	reserved := 0^3 (3 bytes)
sender := I_i (4 bytes)	
ephemeral (32 bytes)	
static ($\widehat{32}$ bytes)	
timestamp ($\widehat{12}$ bytes)	
mac1 (16 bytes)	mac2 (16 bytes)

(a) Handshake Initiation

type := 0x2 (1 byte)	reserved := 0^3 (3 bytes)
sender := I_r (4 bytes)	receiver := I_i (4 bytes)
ephemeral (32 bytes)	
empty ($\widehat{0}$ bytes)	
mac1 (16 bytes)	mac2 (16 bytes)

(b) Handshake Response

Figure 4.3: Handshake messages, see [Don20, p. 10].

Figure 4.3 show first 2 types of Wireguard messages: **Handshake initiation** and **Handshake response**. These 2 messages are the adaptation of Wireguard for Noise handshake. Both of messages contains a **sender index** to indicate a 32-bit index that locally represents the sending peer. The **receiver index** of handshake response is the received **sender index** from the initiator. The ephemeral fields are the ephemeral public keys for key exchange, while the static field of the **handshake initiation** is the encrypted static public key of the initator after the DH computation. The initiation message also includes an encrypted TAI64N [Ber] timestamp to protect against replay attack. The motivation for the final MAC fields are explained in section 5.3.

These MAC fields are computed using the keyed BLAKE2s MAC with a 16-byte hash value as output. Depending on the recipient and field type, the input and key for each MAC field will be different. To be specific:

MAC1 Using $S_{m'}^{pub}$ as the receiver static public key, the key of the MAC computation is the value: **Hash**(“mac1----” $\|$ $S_{m'}^{pub}$). The input of the MAC is all the bytes of the message prior to MAC1 field.

MAC2 Given that the latest cookie was received within 120 seconds, this cookie would be the MAC2 key. Otherwise, if the key is too old or there is no such cookie, the MAC2 field will be zeroed out. The data for MAC2 is all the bytes prior to the MAC2 field.

The third type of message is the **Cookie Reply Message**, which is sent when the current peer is under load. The exact condition to transmit this cookie message is after the reception of a handshake message with valid MAC1 but invalid/expired MAC2 and the peer is under load. The format of the message is as follow:

type := 0x3 (1 byte)	reserved := 0^3 (3 bytes)
receiver := $I_{m'}$ (4 bytes)	
nonce := ρ^{24} (24 bytes)	
cookie ($\widehat{16}$ bytes)	

Figure 4.4: Cookie Reply Message, see [Don20, p. 13].

The receiver is the sender from the previous handshake message.

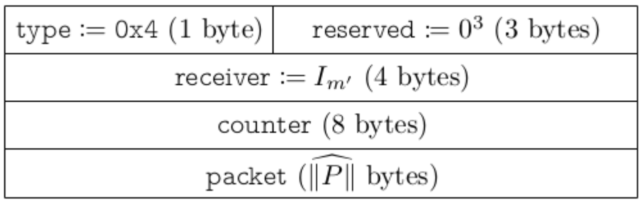
With a random secret value R_m that changes every two minutes, the concatenation of the last message's external IP source address and its UDP source port, and MAC1 as M, the encrypted cookie is created like below:

$$\tau = \mathbf{BLAKE2s}(R_m, A_{m'})$$

$$cookie = \mathbf{XAEAD}(\tau, M, \mathbf{Hash}(''cookie\text{--}''\|S_m^{pub}), nonce)$$

By using M as the additional authenticated data field, the cookie reply is securely bound to the corresponding message, preventing peers from being targeted by fraudulent cookie reply attacks. Additionally, this message is smaller than both the handshake initiation and handshake response messages, reducing the risk of amplification attacks.

The final type of messages is the Transport Data Message - the encrypted encapsulated payload for secure communication. A counter is included in the packet to use as the nonce for AEAD and a tool to prevent replay attack. The inner IP packet will padded with zero bytes until its length is a multiple of 16 to achieve a better address alignment, thus improving the performance on many CPU architectures.



type := 0x4 (1 byte)	reserved := 0^3 (3 bytes)
receiver := $I_{m'}$ (4 bytes)	
counter (8 bytes)	
packet ($\|\widehat{P}\|$ bytes)	

Figure 4.5: Transport Data Message, see [Don20, p. 12].

4.4 Timers

The Wireguard protocol leverages a simple timer state machine behind the scenes for the maintenance of session states, perfect forward secrecy and handshakes. The involved constants for the timers are as below:

Symbol	Value
REKEY-AFER-MESSAGES	2^{60} messages
REJECT-AFTER-MESSAGES	$2^{64} - 2^{13} - 1$ messages
REKEY-AFTER-TIME	120 seconds
REJECT-AFTER-TIME	180 seconds
REKEY-ATTEMPT-TIME	90 seconds
REKEY-TIMEOUT	5 seconds
KEEPALIVE-TIMEOUT	10 seconds

With multiple peers can be configured to be accepted on an interface, the following are the constraints applied on a per-peer basis for the handshake initiation:

1. There can only be one handshake initiation sent every REKEY-TIMEOUT.

2. If there is no handshake response reception of a handshake initiation after REKEY-TIMEOUT, a new handshake initiation will be constructed and transmitted again. This attempt happens for REKEY-ATTEMPT-TIME seconds. This timer will reset when a transport data message is ready to be sent.

3. The handshake completion marks the starting of the age of a secure session. For the initator, this age begins when the processing of handshake reponse is done and the symmetric keys have been derived. For the responder, it is when the first valid transport data is received from the initiator.

4. A timeout alone will not trigger a new handshake unless it is accompanied with the transport data.

5. A session that is older than REJECT-AFTER-TIME or has sent more than REJECT-AFTER-MESSAGES is not usable.

6. A handshake initiation may be sent after sending REKEY-AFTER-MESSAGES.

7. After the initator sends data and the REKEY-AFTER-TIME has passed, a new handshake initiation will be sent.

8. After the initator receives data and the REJECT-AFTER-TIME - KEEPALIVE-TIMEOUT - REKEY-TIMEOUT has passed, a new handshake initiation will be sent.

To mitigate the “thundering herd” problem, where 2 peers try to renew the session at the same time, the initiator is the only one to trigger the rekeying during a valid session. Another mechanism to reduce the probability of this handshake collision, a jitter (1/3 second in the linux-implementation of Wireguard) is added to the expiration time for sending the handshake initation message.

Wireguard also provides a passive keepalive mechanism to determine the activeness of the connection. After an authenticated transport data is received and the no packet can be sent back for KEEPALIVE-TIMEOUT seconds, a keepalive message is sent. The encapsulated payload of this message is just a zero-length buffer, which help distinguish with actual IP payload, using its size. A session can be considered broken when no transport data has been received for (KEEPALIVE-TIMEOUT + REKEY-TIMEOUT) seconds. This will lead to the resending of the handshake initation message every REKEY-TIMEOUT seconds.

About every REKEY-AFTER-TIME, the new session keys for the transport data will created, raising the problems of discarding an in-transit encrypted packet from old session. To tackle this, for each peer, Wireguard maintains 3 symmetric keys for the previous, the current and the next session. The sending transport data messages will always be encrypted with the current session key. For the decryption of recieved messages, depending on the receiver index, either the current or the previous session key will be used. After the initatior completes the handshake, new derived keys will be stored in “current”, while the previous “current” or the previous “next” session keys will be slided to the previous slot. For the responder, its new derived keys will be kept in the “next” slot, waiting to rotate to the “current” when a transport data message is received after the handshake. Before rotation, if the “previous” session keys is not empty, it must be completely wiped out.

Chapter 5

Wireguard Security and the Internet of Things

This chapter discusses in details security properties that Wireguard provides and both the benefits and drawback of adapting these features for the Internet of Things. Section 5.1 is about the silence nature of Wireguard. Section 5.2 show the resilience of WireGuard against other VPN and security protocols at the transport layer. Finally, section 5.3 is about the details of cookie mechanism.

5.1 Virtuous Silence

Preventing storing any state before the authentication and transmitting responses to dubious packets is one of the design objectives of Wireguard. With this specific approach, Wireguard is unreachable from the standpoint of unauthenticated peers and network scanners.

The threat of mirai botnet can also be eliminated using a correct configuration of Wireguard interface on the device. Mirai works by first scanning on the IoT device for the TCP port 23 or 2323, then brute-force logining with a random list of credentials [Ant+17]. With silence nature of Wireguard, if all device's traffic is configured to go through the Wireguard interface except for the external UDP port, mirai won't be able to hijack the device for malicious reasons.

Wireguard handshake messages are implementated without dynamic memory allocation, which is desirable for network protocol on constrained devices. However, this property demands the first initiator's message to be authenticated from the responder, opening a risk for replay attack. An adversary could trick the responder into re-creating its ephemeral key by replay the initial handshake initiation, thus nullifying the session of the valid initiator. To tackle this problem, Wireguard includes an encrypted 12-byte TAI64n [Ber] timestamp in the first message. The responder will now discard handshake packets with timestamp that is lesser than or equal to the peer's greatest received timestamp. However, a drawback of this is that the increasing in size of the packet causes the fragmentation of the message.

5.2 Cryptography Resilience

Adopting new VPN protocols the for IoT should be cryptographically sound, where existing vulnerabilites in previous protocols should not exist within the implementation. The followings are examples of Wireguard being resistant compare to other security protocols:

- Usage of modern cryptography algorithms prevents cryptographic attacks on TLS such as the Sweet32 birthday attacks [BL16], the Bleichenbacher threat on RSA encryption [Ble98], and the padding oracle attack like BEAST [Duo11]. Furthermore, problems with using constructions like MAC-then-Encrypt in IPsec [DP10] are eliminated with AEAD in Wireguard.

- Wireguard lacks of cryptography agility [Ber15] with fixed key exchange and cipher primitives, thus preventing the downgrade attack aiming at insecure older version of the protocol like Lucky Thirteen [AP13] in DTLS/TLS. Given that the underlying algorithm has to be updated, the newest version must be deployed on the endpoints.

- The risk of vulnerabilites from complex message parsing like Heartbleed [Sul14] or parser bugs in X.509 implementations [Bai23] is mitigated due to the fixed message sizes during the handshake.

- With the frequent re-transmission of handshake to re-generate the ephemeral key pairs, attacks taking advantage of key reusing for DH exchange like Racoon attack [Mer+21] would be impossible to launch against the protocol. Even stronger post-quantum security can be enforced with the use of the pre-shared symmetric key, guaranteeing the forward secrecy of the protocol.

5.3 Cookies and Denial of Services Attack

Curve25519 point multiplication is CPU extensive [Don20] and could lead to the exhaustion of battery of constrained devices despite being fast on some processor. In order to protect against this kind of battery-exhaustion attack, a responder could reject handshake message (either initiation or response from the initiator) in an under-load condition, and instead sending back a cookie reply message with a cookie on it. This cookie will be sent in the next handshake initation message from the initiator in the mac2 field.

A cookie is an output of a MAC of the initiator's source IP address with a secret random value that changes every two minutes as a key. For the retransmission, a MAC of its message using the cookie as the MAC key will be sent. As silence is one of the Wireguard virtue, the responder should not send back any cookie reply message when under load for any unauthenticated packets. Thus, every handshake packet incorporates a MAC in the mac1 field with the responder's public key at its MAC key. This at least ensures the sending peer know who it is talking to. The MAC must always be valid regardless of the under-load condition.

To prevent a man-in-the-middle capturing the plaintext cookie and creating fraudulent messages from it, the sending cookie is encrypted using an AEAD with extended randomize nonce [Arc20] and the responder's public key.

A final challenge to tackle is a denial-of-service attack on the initiator, where fraudulent cookie is sent to it, making the MAC computing of its message fail. Wireguard solves the problem by using the mac1 field of the previous handshake initiation as the additonal data field of AEAD for cookie encryption. This effectively binds the cookie reply to the initiation message, assuring flow of invalid cookie replies could not stop the initiator to authenticate the correct cookie. These three mechanism work together to mitigate a battery-exhaustion attack on the Wireguard-running constrained device.

Chapter 6

Design and Implementation

This chapter covers the specific requirements for the Wireguard adaptation for GNRC, the design details to meet the requirements and important implementation aspects inside the Wireguard source code for RIOT.

6.1 Requirements

- **Static memory allocation**: Dynamic memory allocation is prohibited within RIOT core components to maintain a predictable runtime memory usage and real-time performance. With Wireguard being at the layer 3 of the network stack, the protocol must use static memory allocation exclusively.

- **Security**: The GNRC implementation needs to be align to the security properties that the original Linux implementation holds. This includes the resistant to replay attack and side-channel attack [Sta10], perfect forward secrecy, low probability of handshake collision [Wu19] ,cryptographic soundness, and safety from concurrency bugs.

- **Interconnectiviy with other Wireguard implementation**: The RIOT adaptation must be able to communicate with other RIOT Wireguard node and obviously the Linux implementation.
- **Integration to RIOT network interfaces API**: Similiar to the Linux implementation, Wireguard will be an network interface, receive IPv6 packets from the layer above, and hande the all the encryption and dispatching.
- **6LoWPAN compatiblility**: The Wireguard implementation only supports IPv6 packet to be compatible with the 6LoWPAN layer.

6.2 Event-driven architecture

Taking inspiration from BoringTun [Kra19], the Wireguard implementation handles packets and timers in an event-oriented manner. Three different kinds of event can occur within a Wireguard session:

- Transmitting data from the IP layer.
- Reception of a packet from any peer.
- The timeout event related to the Wireguard timers.

Figure 6.1 illustrates the overview of the architecture of the GNRC Wireguard implementation. The Wireguard driver is the core compo-

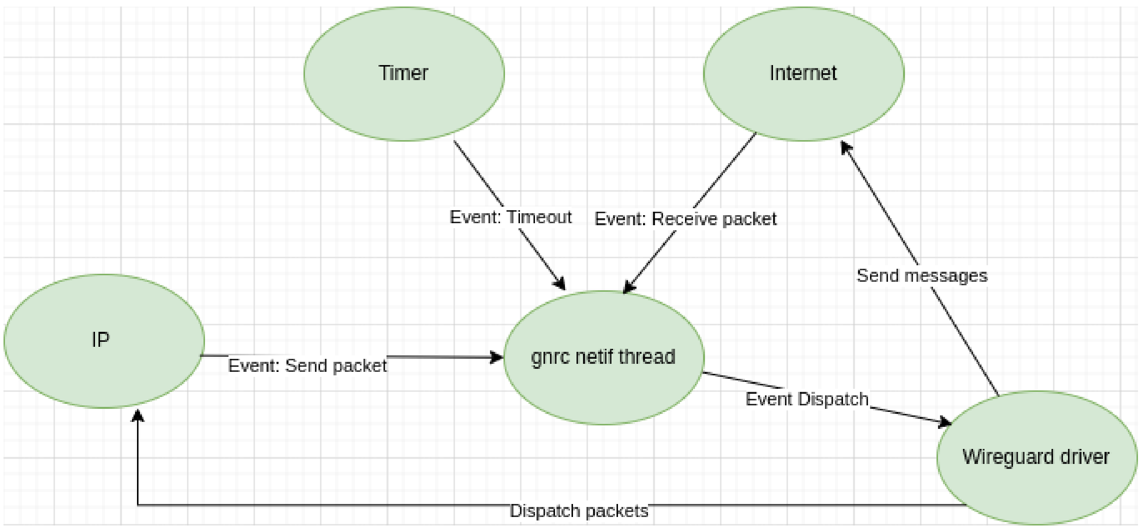


Figure 6.1: GNRC Wireguard Software Architecture

nent where it executes the Noise handshake, handles the functions triggered by the timeout event, encrypts, decrypts and transmits packet to the IP layer or the internet outside. To make use of the existing RIOT system, the GNRC netif's event queue is used as the central event listener for all 3 types of event.

Whenever, a packet is received on the UDP port of Wireguard, new event to handle the reception will be pushed at the netif's event queue, letting Wireguard make the decision whether to responde to the peer if the packet is a handshake message, dispatch the packet directly to the IP layer if it is authenticated or discard if the packet is invalid.

An IP layer will mostly be unaware of the inner working of Wireguard, as it will only transmit the packet to the specified netif and receives the pre-processed IPv6 packet from the driver. One thing to note is that the Linux implementation of Wireguard will queue the packets for another thread to process. The GNRC also follows that design, but with smaller packet queue to match the constrained environment.

Every expiration of a timer, like timer for retransmission of the handshake or passing the keepalive, the event will also be pushed on top of the event queue for the driver to handle later.

The design reduces the complexity of concurrency, as every event is handled sequentially, making it impossible for other threads to meddle with the inner cryptography state of the driver. Although it could make the netif thread a bottleneck of performance, the thread would not be overload in a normal circumstance of IoT environment.

6.3 Network Interfaces API

The interface to connect to the Wireguard driver is exposed as a set of functions interacting with the **gnrc_netif_t** object. The current API is not finalized yet, and future works will involve abstracting the **netif** part away, representing as a generic Wireguard tunnel. Code listing 6.1 shows overview of used functions to communicate with the driver.

```
int gnrc_netif_wireguard_create(
  gnrc_netif_t *netif,
  char *stack,
  int stacksize,
  char priority,
  char *name,
  netdev_t *dev
);
inline int gnrc_netif_wireguard_add_ipv6_addr(
  gnrc_netif_t *netif,
  ipv6_addr_t *addr,
    unsigned int pfx_len
  );
  void gnrc_netif_wireguard_peer_init(
    wireguard_netif_peer_t *peer
  );
  int gnrc_netif_wireguard_add_peer(
    gnrc_netif_t *netif,
    wireguard_netif_peer_t *peer,
    uint8_t *peer_idx
  );
  int gnrc_netif_wireguard_connect(
    gnrc_netif_t *netif,
    uint8_t peer_idx
  );
  int gnrc_netif_wireguard_disconnect(
    gnrc_netif_t *netif,
    uint8_t peer_idx
  );

```

Listing 6.1: Wireguard netif API

The API was largely based on the API of Wireguard for the lwIP stack [Hop], with a bit of modification to be compatible with the RIOT API. It includes a functions to spawn the running **netif** thread to handle sending, receiving of packet and a function to configure the IPv6 address on the network interface.

The **gnrc_netif_wireguard_peer_init** initializes the memory of the struct **wireguard_netif_peer_t** with default values. This netif peer represents the configuration of a peer objects residing on the Wireguard. The corresponding **gnrc_netif_wireguard_add_peer** finds the free region and populates the actual peer objects on the Wireguard's list. Code listing 6.2 shows the structure of the peer configuration. The object comprises of the pubic key and the pre-shared key for encryption decryption, the list of allowed ips that the peer will accept, the communication endpoint and the option to turn on/off for persistent keepalive feature.

```c
typedef struct {
  ipv6_addr_t addr;
  unsigned int pfx_len;
} wireguard_netif_allowed_ip_t;

typedef struct wireguard_netif_peer {
  const char *public_key;
  const uint8_t *preshared_key;
  wireguard_netif_allowed_ip_t *allowed_ips;
  int allowed_ips_len;
  sock_udp_ep_t endpoint;
  int persistent_keepalive;
} wireguard_netif_peer_t;

```

Listing 6.2: netif peer configuration

The API also support the user to actively connect or disconnect to a peer. The connect functions simply register a handshake initiation transmission event, and the disconnect will wipe out all the symmetric keys plus any paritial handshake on a peer. The intention for the connect function also to reduce the number of time for queuing the transport packets, as the sending packets need to be queued if the session keys have not been created.

Later API will include shutting down all the Wireguard state gracefully, more configurations, and an tunnel abstraction over the netif details.

6.4 Wireguard device drivers

The core component of Wireguard is implemented using netdev API of RIOT. Following the specification, the initialization process of the driver divides into 2 steps: **wireguard_setup** to prepare the configuration, and **wireguard_init** to actually initialize the driver.

```
typedef struct wireguard_device {
  netdev_t netdev;
  wireguard_params_t *params;
  uint16_t listen_port;
  sock_udp_t udp;
  gnrc_netif_t *netif;
  event_queue_t *evq;
  struct noise_static_identity static_identity;
  struct wireguard_peers peers;
  struct cookie_checker cookie_checker;
  bool valid;
} wireguard_t;


```

Listing 6.3: Wireguard driver

Listing 6.3 shows the data structure of Wireguard. The **netdev** is the parent struct to interop with the RIOT structure. The **params** field contains the initial configuration. The **udp** is the UDP socket connection with the listening port **listen_port**. The peers is a thread-safe list of peer. A cookie checker with the **static_identity** as the static private/public keypair are also included. A pointer to the original network interfaces and the validity of the driver are required for the driver. The final important detail is the **event_queue**.

Instead of spawning a separate thread that runs on an event loop, to satisfy the memory constraint, the design here is to reuse the existing event queue of **netif** to handle both reception of message and timeout. However, this design has a drawback of making the sending always after the event handler, due to the **netif** thread processes the awaiting events before dispatching the packets inside the mesage queue. Listing 6.4 illustrates this logic.

```
 static void _process_events_await_msg(gnrc_netif_t *netif,
msg_t *msg)
 {
     while (1) {
         /* Using messages for external IPC, and events for
internal events */

         /* First drain the queues before blocking the thread */
         /* Events will be handled before messages */
         DEBUG("gnrc_netif: handling events\n");
         event_t *evp;
         /* We can not use event_loop() or event_wait() because
then we would not
          * wake up when a message arrives */
         while ((evp = _gnrc_netif_fetch_event(netif))) {
             DEBUG("gnrc_netif: event %p\n", (void *)evp);
             if (evp->handler) {
                 evp->handler(evp);
             }
         }
         /* non-blocking msg check */
```

```
            int msg_waiting = msg_try_receive(msg);
            if (msg_waiting > 0) {
                return;
            }
            DEBUG("gnrc_netif: waiting for events\n");
            /* Block the thread until something interesting happens */
            thread_flags_wait_any(
              THREAD_FLAG_MSG_WAITING | THREAD_FLAG_EVENT
            );
        }
    }

```

Listing 6.4: Netif event handler

Another design decision is to not implement the **send** and **recv** of the driver, but instead customize an api for sending and receiving packets, where the send will handle the write-enabled **pktsnip**, and the receving will take the buffer directly from the UDP socket and dispatch the packet up to the IP layer if it is valid.

The event queue also handle the triggered timer event. Each peer will maintain its own list of timers, and push the callback to the event queue when the timer reach their timeout. To trigger the timeout, the **event_timeout_t** is utilized with the **ZTIMER_MSEC** objects. This means the time unit for the timers will be in milliseconds, allowing low-power sleep modes on many platforms.

6.5 Memory allocation & optmization

To reduce the memory consumption of Wireguard, 3 main points has been employed. Firstly, the default number of peers that a Wireguard driver can maintain is 2 peers. Hence, the user of the network interface API has the responsibility to manage the memory of Wireguard. As of the time of writing, the feature to change the number of peers have not been actualized yet. Instead of maintaining a dynamic hash map like the linux implementation to store the key pairs and the indices or a radix trie for the search of allowed IP address, every needed states are in the peer object, a lookup is simply a linear search throughout the list of peers to find the required peer. An example of a look up for allowed IP addresses is in Listing 6.5.

```
struct wireguard_peer *
wireguard_peer_lookup_by_allowed_ip(
  struct wireguard_peers *peers,
  const ipv6_addr_t *addr)
{
  struct wireguard_peer *peer = NULL;
  struct wireguard_peer *tmp;
  struct wireguard_allowed_ip *allowed;
  uint8_t best_match = 0;
  uint8_t match;
  size_t i;
  size_t j;

  mutex_lock(&peers->mtx);
  for (i = 0; i < MAX_PEERS_PER_DEVICE; ++i) {
    tmp = &peers->peers[i];
    if (!tmp->valid) {
      continue;
    }
    /* perform longest prefix match to find the dest ip */
    for (j = 0; j < MAX_SRC_IPS; ++j) {
      allowed = &tmp->allowed_source_ips[i];
      if (!allowed->valid)
        continue;
      match = ipv6_addr_match_prefix(&allowed->ip, addr);
      if (match < allowed->pfx_len)
        continue;
      /* the 0 case is when the destination allowed ip is an unspecified IPv6
      * address */
      if (match > best_match || best_match == 0) {
        peer = tmp;
        best_match = match;
      }
      /* found the exact match on the ipv6 address destination */
      if (best_match == IPV6_ADDR_BIT_LEN) {
        mutex_unlock(&peers->mtx);
        return peer;
      }
    }
  }
  mutex_unlock(&peers->mtx);
  return peer;
}

```

Listing 6.5: Allowed IP address lookup

Secondly, the driver has a global static queue for the sending packets if the symmetric session keys have not been derived. The maxium number of packets that the queue can hold is number of peers multiplied by two. The reason stems from the assumption that the period for the application to send the consecutive packets should be close to the **REKEY_TIMEOUT** limit, allowing a peer to retry transmitting the handshake while reserving the original packet in the condition of lossy network. If a handshake session can not be completed, Wireguard simply discard the queuing packets belong to the peer.

Finally, as the transport data required to be zero padded, reallocation of the orginal payload are required. With the design of the **gnrc_pktsnip_t**, a packet is represented as a linked list of header with the next pointer pointing to next header or payload. To reduce the number of merging and reallocation, before sending the IPv6 transport payload, a special prepare function will compute the total required space for the all encrypted payload including the zero-padded trailer and the transport header, copy all header and payload into the reallocate data, release the hold **pktsnip**, and move the IPv6 header into the corect region. This is applied to the normal transport data only. For a keepalive message, the payload is allocated on the stack and encrypted in the same way of the transport data. The goal of this function is to reduce the memory fragmentation for allocation as much as it possibly can.

```
static int wireguard_prepare_transport_snip(
  gnrc_pktsnip_t *pkt
)
{
  assert(pkt != NULL);
  size_t offset;
```

```
    size_t ipheader_offset;
    size_t plaintext_len;
    size_t trailer_len;
    size_t packet_len;
    uint8_t *buf;
    uint8_t *ip_end;
    int res = 0;

    offset = pkt->size;
    ipheader_offset = offset - 1;
    plaintext_len = gnrc_pkt_len(pkt);
    /* routnd to 16 bytes boundary */
    trailer_len =
        align_mask(plaintext_len, 16) - plaintext_len + noise_encrypted_len(0);
    packet_len = TRANSPORT_DATA_HEADER_LEN + plaintext_len + trailer_len;
    /* Re-allocate data to have enough buffer for:
    * transport_header + ipv6 packet + payload + padded 0 + auth tag*/
    res = gnrc_pktbuf_realloc_data(pkt, packet_len);
    if (res != 0) {
      DEBUG("[wireguard] send: realloc data failed");
      return res;
    }

    /* Copy data to new buffer */
    buf = ((uint8_t *)pkt->data) + TRANSPORT_DATA_HEADER_LEN;
    for (gnrc_pktsnip_t *ptr = pkt->next; ptr != NULL; ptr = ptr->next) {
      memcpy(buf + offset, ptr->data, ptr->size);
      offset += ptr->size;
    }
    /* reverse copying the ipv6 header since the transport header is shorter */
    ip_end = ((uint8_t *)pkt->data);
    /* Chapter 5.6: Mitigation Strategies for integer overflow in
  a reverse for
    * loop - Secure Coding in C and C++ */
    for (size_t i = ipheader_offset; i != SIZE_MAX; i--) {
      buf[i] = ip_end[i];
    }

    /* Release old pktsnips and data*/
    gnrc_pktbuf_release(pkt->next);
    pkt->next = NULL;

    return 0;
  }

```

Listing 6.6: The flow of transport data preparation

6.6 Security Requirements & Implementations

The implementation of the cryptography Noise handshake matches closely to the linux implementation. to ensure the correctness. All the cryptography primitives make use of the API that RIOT provides. To generate the private key and cookie secretly with low probability to predict, SHA256 [H311] pseudo random generator ared used. The ECDH function also include an additional check for zero bytes to prevent the zero bytes shared argeed key [Duo15].

Following the silence nature, an invalid packet is silently dropped, but no ICMP packet will be sent back to the interface. To achieve perfect forward secrecy and protect against various potential leaks, crypto_secure_wipe and crypto_equals are used to clean up all the cryptographic state after used and safe comparision between the keys.

Regarding the under load condition to prevent the DoS, a simple check on whether the number of events on the event queue is greater than 12 events is sufficient (three quarters of the limitation of the number of queued events on a thread, which is 16). The sequence space window algorithm [AK98, appendix C] is employed to protect against the transport data replay attack.

6.7 Implementation Status

The development of Wireguard for GNRC progressed into a functional prototype with the capability to communicate smoothly with the Linux implementation. The prototype needs more thorough testing and verfication and benchmarking to optimize the memory footprint and performance. The current implementation faces 2 important problems:

- The source code of the IP layer to allow the dispatching of IPv6 packet to the wireguard network interfaces.
- A direct installation of neighbor cache entry for a socket endpoint is required to send the UDP packet out.

Finally, integrate Wireguard into **ifconfig** tool of the interactive RIOT shell is also an important missing feature for a smoother configuration of the interface.

Chapter 7

Conclusion

The main contribution of the thesis is an adaptation of the modern VPN protocol - Wireguard for GNRC, a network stack designed for embedded devices. The thesis firstly started with building up the background for the 6LoWPAN network protocol. Backing up by the knowledge of network environment of the Internet of Things, the thesis continued by introducing the RIOT operating system and the design of the generic network stack GNRC. Then, it explained in details the Wireguard protocols, especially in the mechansim of Noise handshake to achieve desirable perfect forward secrecy and post-quantum security. With the previous foundation, the thesis delved into the security features of Wireguard and the benefits it could bring into the IoT field. Finally the thesis provided the design and implementation details of Wireguard for RIOT network stack. While the current implementation is functional, further performance benchmarking against other security protocols and thorough testing are required to identify the flaws and improve the quality of the project.

References

[16] *Internet of Things - number of connected devices worldwide 2015-2025.* Statista Research Department, Nov. 2016. URL: `https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/`.

[AK98] Ran Atkinson and Stephen Kent. *Security Architecture for the Internet Protocol.* RFC 2401. Nov. 1998. DOI: `10.17487/RFC2401`. URL: `https://www.rfc-editor.org/info/rfc2401`.

[Ant+17] Manos Antonakakis et al. “Understanding the Mirai Botnet”. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1093–1110. ISBN: 978-1-931971-40-9. URL: `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis`.

[AP13] Nadhem J. Al Fardan and Kenneth G. Paterson. “Lucky Thirteen: Breaking the TLS and DTLS Record Protocols”. In: *2013 IEEE Symposium on Security and Privacy.* 2013, pp. 526–540. DOI: `10.1109/SP.2013.42`.

[Arc20] Scott Arciszewski. *XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305.* Internet-Draft draft-irtf-cfrg-xchacha-03. Work in Progress. Internet Engineering Task Force, Jan. 2020. 18 pp. URL: `https://datatracker.ietf.org/doc/draft-irtf-cfrg-xchacha/03/`.

[Bac+18] Emmanuel Baccelli et al. "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT". In: *IEEE Internet of Things Journal* 5.6 (2018), pp. 4428–4440. DOI: `10.1109/JIOT.2018.2815038`.

[Bai23] Sandrine Bailleux. *CVE-2022-47630 Trusted Firmware-A - Out-of-bounds read in X.509 parser.* Jan. 2023. URL: `https://www.openwall.com/lists/oss-security/2023/01/16/8`.

[BEK14] Carsten Bormann, Mehmet Ersue, and Ari Keränen. *Terminology for Constrained-Node Networks.* RFC 7228. May 2014. DOI: `10.17487/RFC7228`. URL: `https://www.rfc-editor.org/info/rfc7228`.

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

[Ber05] Daniel J. Bernstein. *The Poly1305-AES message-authentication code.* 2005. URL: `https://cr.yp.to/mac/poly1305-20050329.pdf`.

[Ber06] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *Public Key Cryptography - PKC 2006.* Ed. by Moti Yung et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228. ISBN: 978-3-540-33852-9.

[Ber08] Daniel J. Bernstein. *ChaCha, a variant of Salsa20.* 2008. URL: `https://cr.yp.to/chacha/chacha-20080120.pdf`.

[Ber15] Daniel J. Bernstein. *Boring crypto.* Oct. 2015. URL: `https://cr.yp.to/talks.html#2015.10.05`.

[BL16] Karthikeyan Bhargavan and Gaëtan Leurent. *On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN.* Cryptology ePrint Archive, Paper 2016/798. 2016. DOI: `10.1145/2976749.2978423`. URL: `https://eprint.iacr.org/2016/798`.

[Ble98] Daniel Bleichenbacher. *Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1.* 1998. URL: `https://archiv.infsec.ethz.ch/education/fs08/secsem/bleichenbacher98.pdf`.

[Bor+12] Carsten Bormann et al. *Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs).* RFC 6775. Nov. 2012. DOI: `10.17487/RFC6775`. URL: `https://www.rfc-editor.org/info/rfc6775`.

[Bru16] Simon Brummer. "Concept and Implementation of TCP for the RIOT Operating System and Evaluation of Common TCP-Extensions for the Internet of Things". In: (2016).

[DH17] Dr. Steve E. Deering and Bob Hinden. *Internet Protocol, Version 6 (IPv6) Specification.* RFC 8200. July 2017. DOI: `10.17487/RFC8200`. URL: `https://www.rfc-editor.org/info/rfc8200`.

[DH76] W. Diffie and M. Hellman. "New directions in cryptography". In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. DOI: `10.1109/TIT.1976.1055638`.

[Don20] Jason A. Donenfeld. *WireGuard: Next Generation Kernel Network Tunnel.* 2020. URL: `https://wireguard.com/papers/wireguard.pdf`.

[DP10] Jean Paul Degabriele and Kenneth G. Paterson. "On the (in)security of IPsec in MAC-then-encrypt configurations". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security.* CCS '10. Chicago, Illinois, USA: Association for Computing Machinery, 2010, pp. 493–504. ISBN: 9781450302456. DOI: `10.1145/1866307.1866363`. URL: `https://doi.org/10.1145/1866307.1866363`.

[Duo11] Thai Duong. *BEAST.* 2011. URL: `https://vnhacker.blogspot.com/2011/09/beast.html`.

[Duo15] Thai Duong. *Why not validate Curve25519 public keys could be harmful.* 2015. URL: https://vnhacker.blogspot.com/2015/09/why-not-validating-curve25519-public.html.

[GC06] Mukesh Gupta and Alex Conta. *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification.* RFC 4443. Mar. 2006. DOI: 10.17487/RFC4443. URL: https://www.rfc-editor.org/info/rfc4443.

[GCS21] Carles Gomez, Jon Crowcroft, and Michael Scharf. *TCP Usage Guidance in the Internet of Things (IoT).* RFC 9006. Mar. 2021. DOI: 10.17487/RFC9006. URL: https://www.rfc-editor.org/info/rfc9006.

[H311] Tony Hansen and Donald E. Eastlake 3rd. *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF).* RFC 6234. May 2011. DOI: 10.17487/RFC6234. URL: https://www.rfc-editor.org/info/rfc6234.

[Hen11] Sushil Jajodia Henk C. A. van Tilborg, ed. *Encyclopedia of Cryptography and Security.* 2nd ed. Springer New York, NY, 2011. DOI: https://doi.org/10.1007/978-1-4419-5906-5.

[HH05] Brian Haberman and Bob Hinden. *Unique Local IPv6 Unicast Addresses.* RFC 4193. Oct. 2005. DOI: 10.17487/RFC4193. URL: https://www.rfc-editor.org/info/rfc4193.

[Hop] Daniel Hope. *WireGuard Implementation for lwIP.* URL: https://github.com/smartalock/wireguard-lwip.

[KE10] Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF).* RFC 5869. May 2010. DOI: 10.17487/RFC5869. URL: https://www.rfc-editor.org/info/rfc5869.

[KNB18] Nadim Kobeissi, Georgio Nicolas, and Karthikeyan Bhargavan. *Noise Explorer: Fully Automated Modeling and Verification for Arbitrary Noise*

Protocols. Cryptology ePrint Archive, Paper 2018/766. 2018. URL: https://eprint.iacr.org/2018/766.

[Kra19] Vlad Krasnov. *BoringTun, a userspace WireGuard implementation in Rust*. Mar. 2019. URL: https://blog.cloudflare.com/boringtun-userspace-wireguard-rust/.

[Lar18] William Largent. *New VPNFilter malware targets at least 500K networking devices worldwide*. 2018. URL: https://blog.talosintelligence.com/vpnfilter/.

[Len16] M. Lenders. "Analysis and Comparison of Embedded Network Stacks". MA thesis. Freien Universität Berlin, 2016.

[Mer+21] Robert Merget et al. "Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E)". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 213–230. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/merget.

[MG23] Daniel Migault and Tobias Guggemos. *Minimal IP Encapsulating Security Payload (ESP)*. RFC 9333. Jan. 2023. DOI: 10.17487/RFC9333. URL: https://www.rfc-editor.org/info/rfc9333.

[Mig+24] Daniel Migault et al. *ESP Header Compression Profile*. Internet-Draft draft-ietf-ipsecme-diet-esp-01. Work in Progress. Internet Engineering Task Force, July 2024. 35 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-ipsecme-diet-esp/01/.

[Mon+07] Gabriel Montenegro et al. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944. Sept. 2007. DOI: 10.17487/RFC4944. URL: https://www.rfc-editor.org/info/rfc4944.

[NJT07] Dr. Thomas Narten, Tatsuya Jinmei, and Dr. Susan Thomson. *IPv6 Stateless Address Autoconfiguration*. RFC 4862. Sept. 2007. DOI: 10.17487/RFC4862. URL: https://www.rfc-editor.org/info/rfc4862.

[NL18] Yoav Nir and Adam Langley. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 8439. June 2018. DOI: 10.17487/RFC8439. URL: https://www.rfc-editor.org/info/rfc8439.

[Ori15] Edewede Oriwoh. "'Things' in the Internet of Things: Towards a Definition". In: *International Journal of Internet of Things* 4 (Mar. 2015). DOI: 10.5923/j.ijit.20150401.01.

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

[Pet+15] Hauke Petersen et al. *Old Wine in New Skins? Revisiting the Software Architecture for IP Network Stacks on Constrained IoT Devices*. 2015. arXiv: 1502.01968 [cs.NI]. URL: https://arxiv.org/abs/1502.01968.

[Pos80] J. Postel. *User Datagram Protocol*. RFC 768. 1980. DOI: 10.17487/RFC0768. URL: https://www.rfc-editor.org/info/rfc768.

[PP09] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 3642041000.

[SA15] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*. RFC 7693. Nov. 2015. DOI: 10.17487/RFC7693. URL: https://www.rfc-editor.org/info/rfc7693.

[SB09] Zack Shelby and Carsten Bormann. *6LoWPAN The Wireless Embedded Internet*. WILEY, 2009.

[Shu+07] Feng Shu et al. "Packet loss analysis of the IEEE 802.15.4 MAC without acknowledgements". In: *IEEE Communications Letters* 11.1 (2007), pp. 79–81. DOI: 10.1109/LCOMM.2007.061295.

[Sta10] Fran¸cois-Xavier Standaert. *Introduction to Side-Channel Attacks*. Jan. 2010.

[Sul14] Nick Sullivan. *Staying ahead of OpenSSL vulnerabilities.* 2014. URL: https://blog.cloudflare.com/staying-ahead-of-openssl-vulnerabilities/.

[Taw+20] Lo'ai Tawalbeh et al. "IoT Privacy and Security: Challenges and Solutions". In: *Applied Sciences* 10.12 (2020). ISSN: 2076-3417. DOI: 10.3390/app10124102. URL: https://www.mdpi.com/2076-3417/10/12/4102.

[Wu19] Peter Wu. "Analysis of the WireGuard protocol". MA thesis. Eindhoven University of Technology, 2019.