VIETNAMESE - GERMAN UNIVERSITY

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

FACULTY OF ENGINEERING

COMPUTER SCIENCE DEPARTMENT

BACHELOR THESIS

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

By

Le Hoang Dang Nguyen

Matriculation number: 17028

Assessor:

1. Prof. Dr. Oliver Hahm
2. Dr. Tran Thi Thu Huong

Binh Duong, 2024

This page intentionally left blank.

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

Contents

Chapter 1

Introduction

1.1 Motivation

As VPN is mentioned in many papers as not suitable for constrained small IoT Devices, I start to question, is it really the true for in this modern area. As a modern network protocols called wireguard show up, with an aim to replace the old, sound but complex protocols like IPSec, we here consider adapting the wireguard protocols onto this constrained IoT environment, to see whether this VPN protocol is a good fit for the IoT world or not.

1.2 Organization

Chapter 2 discusses about the background knowlegde of IoT, RIOT Operating system, and the 6LoWPAN network stack.

Chapter 3 is about the underlying network stack that powers RIOT - GNRC.

Chapter 4 is a thorough explaination of the wireguard protocol.

Chapter 5 gives an evaluation of wireguard security, and how it help the IoT world.

Chapter 6 gives some related works of other compressed VPN protocols, and other evaluation of wireguard for constrained devices.

Chapter 7 discusses the design and implementation of wireguard.

Chapter 8 explains the testing of the wireguard.

Chapter 9: conclusion and future works of this 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 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 sev-

eral 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.

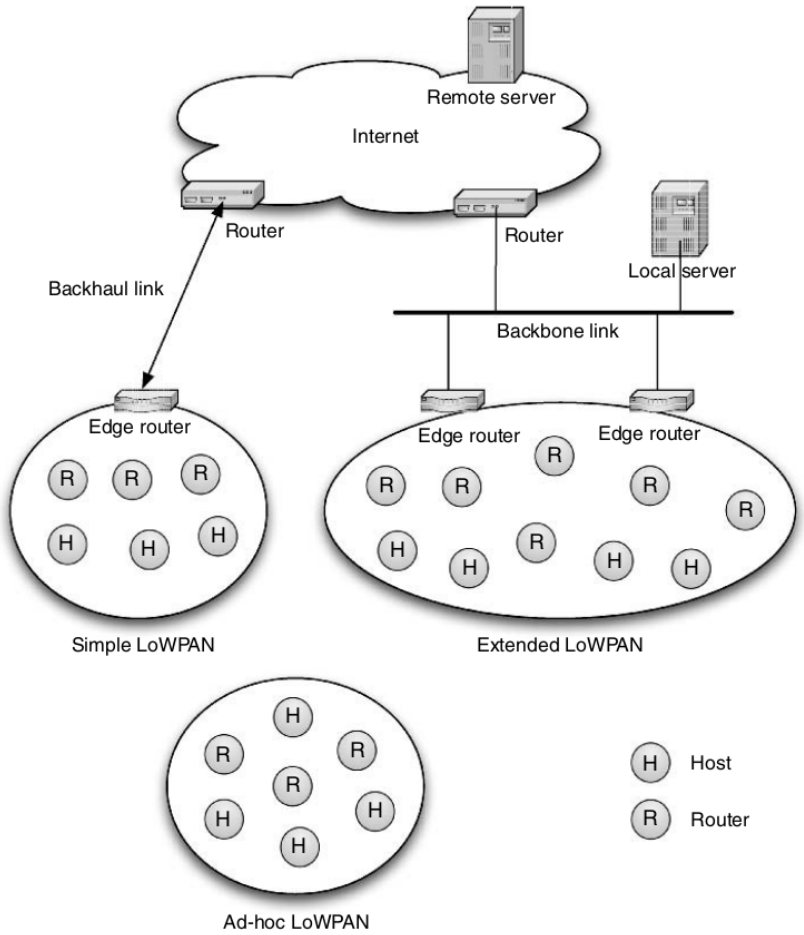

Figure 2.1: The 6LoWPAN architecture, see [SB09, p. 14]

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.

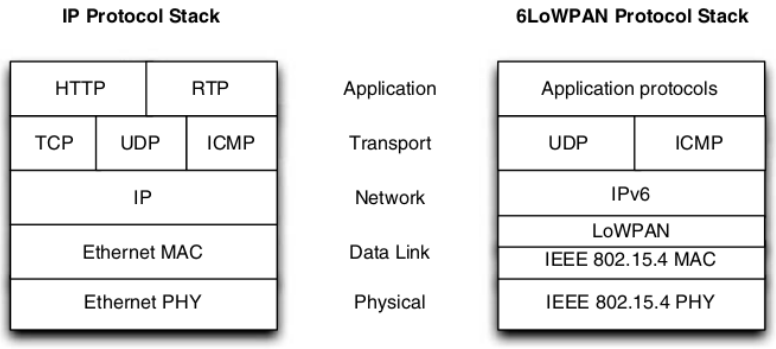

Figure 2.2: IP and 6LoWPAN stack, see [SB09, p. 16]

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 adapta-

tion 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 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 re-

transmission of corrupted data, and 128-bit 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 ≈ 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.

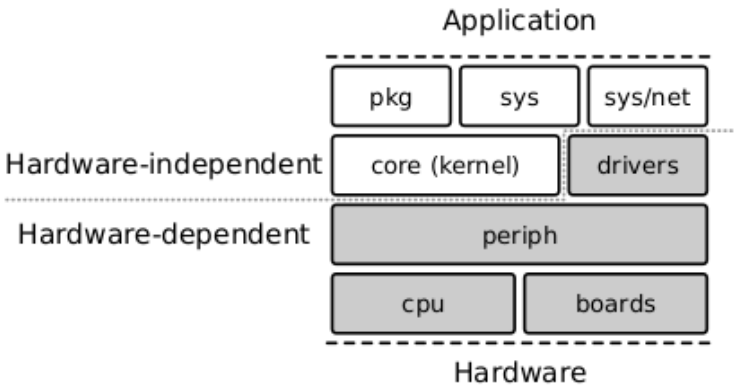

Figure 3.1: Structure elements of RIOT, see [Bac+18, p. 3]

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

- Design of the network stack

  - Northbound API with sock for application

  - Southbound API for network interface

  - Each network stack runs on its stack, communicate with each other through IPC api.

3.2.2 The Packet Buffer - pktbuf

- The copy on write nature

  - Central packet buffer to reduce packet copy.

3.2.3 GNRC's Module Registry - netreg

- What it is?

3.2.4 A Common Inter-Modular API - netapi

- Does it still exist?

3.2.5 Network Interfaces and Device Drivers

- Netif and how it works

Chapter 4

Concepts of Wireguard

4.1 Protocol & Cryptography

4.1.1 Overview

4.1.2 Cryptokey Routing

4.1.3 Handshake

4.2 Noise

4.3 Wireguard Messages

4.4 Cookies and Denial of Services Attack

4.5 Timers

4.6 Key Rotation

Chapter 5

Evaluation of Wireguard

Chapter 6

Related Work

Chapter 7

Design and Implementation of Wireguard for GNRC

Chapter 8

Testing

8.1 Methodology

8.2 Environment

8.3 Scenario 1

8.4 Scenario 2

8.5 Scenario 3

8.6 Scenario 4

8.7 Memory Usage

Chapter 9

Conclusion and Future Work

References

[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.

[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.

[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.

[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.

[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.

[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.

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

[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.