INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Minimal G-IKEv2 implementation for RIOT OS

Tobias Heider

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Minimal G-IKEv2 implementation for RIOT OS

Tobias Heider

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Dr. Nils gentschen Felde
	Tobias Guggemos
Abgabetermin:	28. April 2017

I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 28. April 2017

. . . .
(Unterschrift des Kandidaten)

Abstract

The Internet of Things (IoT) is one of today's fastest growing trends in technology and lead to a growing number of constrained devices connected to the internet and an increasing importance of group communication. The device's constraints present a challenge for security standards, as they cannot simply be reused for these novel systems. In order to enable secure group communication in the IoT, a group key management solution must be found that complies with the limitations arising from the use of low powered embedded systems. The G-IKEv2 protocol is found to provide a secure key exchange, even though it is not optimized for the use in IoT networks. A solution is offered by the design of a "minimal G-IKEv2 client" which reduces the proposed G-IKEv2 standard to a minimal subset of messages and payloads necessary to achieve a secure key exchange.

This work implements the "minimal G-IKEv2 client" on the IoT operating system RIOT OS. The evaluation tests the implementation in regard to memory requirements as well as CPU performance - measured by the time needed to handle the exchange - and proves the feasibility of secure group key distribution on IoT systems.

3DES	Triple Data Encryption Standard
AES	Advanced Encryption Standard
AH	Authentication Header
API	Application Programming Interface
CBC	Cipher Block Chaining
CCM	Counter with CBC-MAC
CTR	Counter Mode
DH	Diffie-Hellman
DTLS	Datagram Transport Layer Security
ECB	Electronic Code Book
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ESP	Encapsulating Security Payload
GCKS	Group Controller / Key Server
G-IKEv2	Group Internet Key Exchange version 2
GM	Group Member
GSA	Group Security Association
HMAC	Keyed-Hash Message Authentication Code
IKEv2	Internet Key Exchange version 2
IoT	Internet of Things
IP	Internet Protocol
IPsec	Security Architecture for the Internet Protocol
KEK	Key Encryption Key
LKH	Logical Key Hierarchy
MAC	Message Authentication Code
MODP	More Modular Exponential
OS	Operating System
PRF	Pseudorandom Function
RAM	Random Access Memory

ROM	Read-Only Memory
RNG	Random Number Generator
RSA	Rivest-Shamir-Adleman
SA	Security Association
SAD	Security Association Database
SHA	Secure Hash Algorithm
SPI	Security Parameter Index
TEK	Transport Encryption Key
TLS	Transport Layer Security
UDP	User Datagram Protocol
WSN	Wireless Sensor Network
XCBC	XOR CBC

Contents

1. Introduction

The “Internet of Things”(IoT) is one of today’s fastest growing trends in technology. The idea behind IoT is providing everyday objects with network capabilities, in order to gain advantage from real time data. Possible applications of such a network reach from health and body function monitoring, as implemented by fitness trackers, to large scale infrastructure or traffic control, as it is used in so called “smart cities”, to improve traffic flows, parking space and public safety. According to the swedish company Ericsson, IoT makes up for a total of 5.6 billion internet connected devices today, which is only second to mobile phones with 7.3 billion devices. It is predicted to surpass mobile phones in the year of 2018 and it is expected to grow further till 2022 where 18 billion of a total of 29 billion connected devices are forecasted to be related to IoT [1].

One of the most widespread applications are sensor networks and especially wireless sensor networks (WSNs). WSNs consist of sensor nodes arranged around a gateway node, which acts as a border router. Use cases are diverse, some of the most prominent are the monitoring of smart home solutions or industrial facilities.

The nodes in such networks are usually low power, single purpose embedded devices, provided with a number of sensors and wireless network capabilities. They may be limited in terms of processing power, memory capacity, and power usage. As they are often deployed in difficult to access places and running on battery, a low power consumption resulting in a longer lifetime is often essential.

WSN operating systems need to face challenges presented by the sensor nodes hardware heterogeneity and constraints, while trying to provide the capabilities of a full fledged OS like Linux. An abstraction of the hardware layer is desirable, for allowing the use of portable libraries and software implementations and being more developer friendly. The dominant Contiki [2] and TinyOS [3] follow an event driven approach, which allows them to efficiently handle typical tasks, but limits networking functionality. RIOT OS [4], on the contrary, is a newly developed IoT focused OS supporting real multi-threading. It was developed with the goal to provide Linux like features on heterogeneous constrained hardware platforms.

Aside from their advantages, WSNs, as all network connected computer systems, pose a great risk for data security. The use of a shared medium, like wireless access points, exposes sensor nodes to a broad attack surface. Applications like a fire protection system consisting of a net of connected smoke detectors and fire extinguishing units, as well as pressure control and monitoring system on an oil platform, may be both safety and security critical. Correct functionality has to be guaranteed at any time and a hostile actor could cause severe harm if he is able to access the systems network traffic. Because sensor nodes usually come in the form of low power embedded devices, they have special requirements for security implementations to adjust to their various constraints.

Existing protocol standards provide a number of security mechanisms enabling secure networking for the IoT. The widely used IEEE 802.15.4 [5] standard enables link-layer security via message authentication codes (MACs) providing confidentiality, integrity, access control and replay control between two hosts on the data link layer.

End-to-end security can be achieved on the network layer by using the IPsec protocol suite with it's various adaptions for the IoT like [6] and [7]. [8] describes why and how IPsec should be used to protect wireless sensor networks. Another popular end-to-end encryption technology is the Transport Layer Security (TLS) protocol and it's UDP based derivative called Datagram Transport Layer Security (DTLS).

One of the biggest challenges presented by large scale sensor networks is the cryptographic protection of group communication. Sensor nodes in WSNs often need to send their data to multiple controllers. At the same time a single controller may receive status updates from various sensors in the network. Using group key management, every member only needs to establish one secure connection with the key server and then receive and store a single set of keys, in order to communicate with all group members over a protected channel. Optionally, for larger networks, the group may be split to several smaller groups (e.G. one group for every sensor node), in order to prevent a single point of failure.

This thesis describes the implementation of a group key management protocol, adapted for the requirements of the Internet of Things. The implementation will be built using RIOT OS, because of it's support for widely used IoT platforms and embedded libraries. In addition the protocol will use the IPsec suite for end-to-end encryption because of various reasons:

- IPsec encryption is not limited to a single transport layer protocol like DTLS and TLS.
- Applications running on a node protected with IPsec do not have to be aware of existing encryption. This allows to protect applications that were not built with data security in mind.
- IPsec's modular architecture makes it easy to exchange the key management protocol while preserving other functionality

A commonly named downside of IPsec is, that it needs access to the OSs network stack and therefore needs to run in kernel space, compared to TLS, which runs entirely in user space. This is not a problem in RIOT due to it's microkernel architecture, which does not differentiate between kernel and user space.

Structure of this thesis: Chapter 2 describes the motivation for this work, including a specification of secure group key management, it's applications and an analysis of various protocols which leads to the final protocol choice for the implementation.

The background required for the further chapters is described in Chapter 3. It will give a more in depth explanation of technologies used in IoT networks, including the most important protocol standards like 6LoWPAN and IEEE 802.15.4. Section 3.2 gives an overview of RIOT OS and the features used for the implementation, especially the GNRC network stack. Section 3.3 will explain IPsec's mechanisms and how the different IPsec databases and protocols work together.

"Requirements of G-IKEv2 for RIOT" (Chapter 4) derives a number of requirements for a system implementing G-IKEv2 on RIOT OS. These include both functional requirements, like the availability of cryptographic functions, as well as non-functional requirements, describing minimal limits for necessary resources like memory or CPU power.

Chapter 5 defines a “minimal G-IKEv2 client” protocol combining features of G-IKEv2 with the adaptions for constraint devices of the “minimal IKEv2 initiator implementation”. The resulting protocol is optimized for IoT devices and compatible with all G-IKEv2 server implementations.

Chapter 6 describes the implementation of the “minimal G-IKEv2 client” for RIOT OS in detail, including the full feature scope and limitations. Section 6.1 describes the implementation of the used IPsec databases. Then the implementation’s configuration setup is explained. Section 6.3 treats RIOT’s network packet API which is used to build G-IKE messages as well as the structure of the message processing functions used in the implementation. In the following, the configuration setup, and finally the protocol handler structure are illustrated in detail. In Section 6.5 several problems encountered during the implementations including solutions are presented. Finally the used external libraries are described in Section 6.6.

A summary of the evaluation of the implementation including an analysis of the actual memory requirements and the protocol’s performance on a number of selected embedded systems is given in Chapter 7. This chapter will assess the implementations in regards to the requirements found in Chapter 4.

Chapter 8 reevaluates the findings and contributions of the thesis and provides an outlook for future work necessary to use the minimal G-IKEv2 client in a productive environment.

The result of this work has been published in the form of an academic paper called “Secure Group Key Distribution in Constrained Environments with IKEv2”. Throughout this thesis several chapters will refer to this paper. It can be found in it’s full extent in Appendix A.

2. Motivation

This section illustrates the tasks of secure group communication, how it can be achieved and when it should be used. Section 2.1 deals with the features a group key management protocol must provide to guarantee secure group communication. In Section 2.2 the applications of secure group communication and when it should be used over traditional peer-to-peer key management are explained. Finally, Section 2.3 treats the existing group key management solutions in regard to suitability for the tasks derived in Section 2.1.

2.1. Secure group communication

Secure group communication involves several aspects such as the management of the groups keys, the groups security policies as well as the data handling.

These tasks are typically implemented using a cryptographic protocol suite. Similar to secure point-to-point communication, protocol suites might use different protocols for group management and data handling. The advantage of such an architecture is the reusability of common functionality. In the case of IPsec, for example, the key exchange protocol may be exchanged for a group based approach (G-IKEv2, GDOI) while the transport protocol (e.g. ESP) can be used without the need to make adaptions.

[9] defines a framework of security services a secure group key management protocol should provide to enable secure communication. Appendix A further summarizes these to a subset of security features that is found mandatory for secure group communication - with regard to constrained networks:

Identity Management: The management of group members unique identity which is required for further management operations such as authorization management or group membership operations.

Authorization and Authentication Infrastructure (AAI): Provides a mechanism for nodes to proof their identity (Authorization), as well as the management of the nodes rights in the group (Authentication). Authorization is either achieved by a pre-shared secret only known by the group manager and a number of selected nodes, or by a public-key infrastructure.

Group Key Management: A mechanism to selectively distribute keys to authorized group members, enabling them to take part in the group communication and protect the groups traffic.

Group Management: The provision of group membership operations, like join, leave and group creation/destruction.

Security: Achieved by using cryptographic functions providing confidentiality and sender authentication. Dynamic groups may raise special security requirements allowing new group members to only access future communication (backward secrecy) and not allowing former group members access to future communication (forward secrecy).

2.2. Use Cases

Constrained devices are often used in complex network environments, a typical example being WSNs. Network nodes may be divided into several groups, either defined by a common function, local proximity, or by their level of authorization. The following scenarios highlight common cases where group key management may be used and how it is superior to a point-to-point key management approach.

2.2.1. Group key protected unicast communication

WSNs commonly consist of a single central controller and several constrained nodes like sensors, periodically communicating their current status to the controller. The communication uses traditional IP unicast, all nodes only communicate with the controller not with each other.

A group key solution may be preferred over 1-to-1 key management to reduce memory usage and complexity for the controller. The sensors and the controller may form a protected group and communicate using a single set of group keys k. In contrast to the use of a 1-to-1 key management the controller will only have to store k, instead of $m * k$ where m is the number of sensor nodes reporting their status to the controller. A disadvantage of the group key solution is that a single compromised node enables an attacker to decrypt the whole groups communication.

2.2.2. Multicast communication

In dynamic WSNs single nodes may need to send generated data, for example sensory input, to various receivers that will further process the input. This can be achieved by forming a multicast group. Receiver nodes can “subscribe” to the sensors output by joining the corresponding multicast group.

To limit access and provide security for the groups communication the multicast group members additionally form a protected group securing the multicast groups communication. Hostile nodes are still able to join the multicast group and receive encrypted group messages. However, they won't be allowed group access by the group key management server and are thus unable to decrypt received messages.

2.3. G-IKEv2 protocol choice

A comprehensive analysis of existing group key management approaches and their applicability to constrained systems has been done as part of the paper “Secure Group Key Distribution in Constrained Environments with IKEv2” which can be found in Appendix A This section is an excerpt from section II of this paper.

2.3.1. Related Work

During the past decade, several research activities on key distribution in different areas have been carried out [10]. It started with the key distribution for mobile networks (e. g. UMTS, GPRS, etc.), followed by activities to share keys in wireless sensor networks, while considering constrained environments. Salgarelli et al. [11] provide a solution for a wireless key exchange and authentication protocol, which has been compared to current approaches such as the *Extensible Authentication Protocol* (EAP [12]) and *Transport Layer Security* (TLS [13]). Although this work focuses on wireless networks and mobility, it still requires 12 messages to distribute a pair of keys. Additionally, it is not designed for multicast key distribution, which is essential when talking about group communication.

Group key distribution itself has been studied in [14], which resulted in a couple of standardization activities. Rafaeli et al. [14] survey a set of approaches for secure group key distribution (GKD). According to their analysis, there are three different types of GKDs: centralized, decentralized and distributed GKD protocols. Most of the protocols considered are rather mathematical schemes than networking protocols, but nevertheless some of them are included in actual group management protocols, such as the *Group Domain of Interpretation* (GDOI) [15]. Although all of these approaches specify the possibility of secure (group) key distribution (also in constrained networks), none of them had been properly implemented, distributed or evaluated against the requirements in constrained environments such as 1) highly resource constraint devices, 2) highly distributed and 3) globally connected.

In the following, recent activities with the goal of a DTLS-based multicast solution for low latency networks [16] are analyzed and the most commonly used protocol (GDOI) and its potential replacement G-IKEv2 are explained (more on G-IKEv2 can be found in section 3.3.6). Additionally, the concept behind hierarchical and distributed GKMs and their applicability in constrained networks is detailed.

Centralized Group Key Management

Centralized key distribution is the most obvious way of managing group keys as it leaves the complexity and trust to a single system. Most of the commonly used protocols (DTLS, GDOI, Kerberos, etc.) are designed on top of centralized systems. However, the concept of centralization inherits some natural difficulties, such as weak scalability, especially when only one server manages a geographically distributed network. In general, the larger a group gets, the more complex becomes its management and the resulting operations. This makes many centralized group key management concepts not feasible for constrained environments.

Decentralized Group Key Management

Decentralized key distribution still sticks to the concept of a central server, but splits the group in administrative domains for both management operations and key exchanges. Additionally, it distributes the workload over more devices and thus eases the key calculations on the client side. On the other hand, the decentralized concept requires strong trust relationships, which is a showstopper for many use cases where administrative domains can change frequently and devices are potentially physically accessible by arbitrary persons. As exemplary standards, GDOI and G-IKEv2 (see below) are able to use decentralized schemes for group key derivation.

Table 2.1.: Comparison of existing group key exchange mechanisms

Feature / Requirement	**DTLS**	**GDOI**	**G-IKEv2**	**Decentralized**	**Distributed**
Group Management					
↪ Join Unicast	✘	✔	✔	✔	✔
↪ Join Multicast	✔	✔	✔	✔	✔
↪ Leave Unicast	✘	(✔)[a]	(✔)[a]	✔	✔
↪ Leave Multicast	✘	(✔)[a]	(✔)[a]	✔	✔
Security					
↪ *in general*	(✔)	(✔)[b]	✔	✔	✔
↪ Forward Secrecy	✘	✔[c]	✔[c]	(✔)	✔
↪ Backward Secrecy	✘	✔[c]	✔[c]	✔	✔
Applicability					
↪ Link Local	✘	✘	✔	✘	✘
↪ Broadcast Domain	✘	✘	✔	✔	✔
↪ LAN	✔	✔	✔	✔	✔
↪ WAN	✔	✔	✔	(✔)	✘
Lightweight:					
↪ Implementation	✔	✘	✔	✘	✘
↪ Memory/Storage	✔	(✔)	✔	✘	✘
↪ Networking	(✔)	✘	✔	(✔)	✘
↪ Standardized for IoT	✔	✘	(✔)	✘	✘

legend: ✔ addressed by design (✔) partially addressed ✘ not addressed by design

[a] *Leave* is only supported by the GKM server
[b] based on obsoleted IKEv1
[c] with logical key hierarchy (LKH)

Distributed Group Key Management

Distributed concepts remove any kind of management server, thus, every member of the group holds the complete state of the group. This approach produces additional network load and compute operations for re-keying every time a group management operation is performed. Thus, these concepts work well in closed and "stable" environments such as wireless sensor networks, but they obviously do not scale to a large extent.

DTLS-based

The IETF internet draft on *Security for Low-Latency Group Communication* [16] describes secure multicasting by using DTLS in low latency networks. The focus is the distribution of symmetric keys, which are also used for authentication and thus limiting the achievable level of security. Nevertheless, the draft designs a system including a key distribution center and authorization server, but it does not consider issues of forward and backward secrecy and re-keying. Additionally, no group management operations are considered, despite *joinGroup*-operations. Due to the design choice to use DTLS, the applicability is limited to local and wide area networks, but excludes possibility to be used on lower ISO/OSI-Layers. On the other hand, with RFC 7925 [17] DTLS is adjusted to comply with the needs and constraints of IoT devices.

IPsec based (GDOI and G-IKEv2)

With multicast usually being an IP specific use case, the choice of IPsec for security seems natural. GDOI [15] was the first standardized protocol for securing multicast. It is a centralized protocol, but with the possibility of using hierarchical key distributions mechanism such as logical key hierarchy (LKH). However, due to the network overhead during key exchanges (7 messages) and using the obsolete IKEv1 as the underlying protocol, a replacement is inevitable. That said, G-IKEv2 [18] is currently proposed as a new standard for group key management. Using IKEv2 reduces the number of messages during key exchange to only 4, making it easier adoptable for constrained devices. Additionally, with RFC 7815 IKEv2 is specified for the use in IoT [6] and recommended for the key exchange in IEEE 802.15.4 (*Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)*) networks.

2.3.2. Summary and choice of a protocol

Table 2.1 shows the findings of this section and evaluates related work against four categories of requirements:

Group Management describes the ability to manage groups for multicast and unicast, both of which are considered subsets of $n : m$-communication patterns.

Security evaluates on the security of a given solution in general and adds a special focus on forward and backward secrecy.

Applicability considers if a solution can be used in link local networks (e. g. radio networks) or broadcast domains (e. g. in self organizing WSNs) – both being ISO/OSI layer II networks – or local area networks (LAN, environments requiring local routing) or wide area networks (WAN) where global routing is required.

Lightweight shows if a solution can be found suitable for constrained devices in terms of their limitations (complexity of the implementation, limited CPU power, memory / storage capacities, network resources). Additionally, it is assessed on the availability of a specification specific to constrained devices and environments.

To conclude, G-IKEv2 is suited best for the area of application of this work. With its underlying protocol IKEv2 already being adopted (RFC 7815 [6]) and used in constrained environments and its flexibility to be extended by concepts of decentralized and distributed group key distribution schemes, it represents a good choice.

3. Background

This chapters purpose is to introduce the reader to the necessary prerequisites for this work. The first section will explain the IoT and more specifically sensor network environment in both hardware and software. The second part will introduce the basic functionalities and architecture of RIOT OS, on which this work's implementation depends. IPsec and it's protocols will be treated in the last section. The focus will be the IKEv2 protocol and it's derivative, the “minimal IKEv2 initiator implementation”, which this work heavily depends upon.

3.1. Internet of Things

As there is no single clear definition for the IoT, for this work it is understood as a network of interconnected objects, sensors and smart devices with the ability to exchange information using network capabilities. The size of an IoT network can span from small “personal networks”, containing smart devices like trackers and mobile phones, up to big wireless sensor networks monitoring power plants, hospitals or cities. The Applications of this kind of sensor networks include energy management, infrastructure management, medical usage and industrial manufacturing in the so called Industry 4.0.

3.1.1. Network standards

A number of protocols have been proposed by a cooperation [19] of the IEEE 802 and the IETF, in an effort to standardize IoT networking. Figure 3.1 shows a comparison of a traditional wireless IP network stack and a the typical IoT stack.

The leading physical and link layer protocol is the IEEE 802.15.4 [5] standard. It defines low-rate wireless personal area networks (LR-WPANs), that, in contrast to the much faster IEEE 802.11 Wireless LAN standard [20], sacrifice speed, at the gain of power efficiency. This makes it the ideal protocol for battery powered devices. Another popular wireless technology used in IoT networks is the Bluetooth Low-Energy standard.

Regarding network layer encapsulation IP based approaches are gaining popularity compared to older proprietary non-IP standards like ZigBee [21] or Z-wave [22]. The dominant IP based protocols are the 6LoWPAN [23] and 6TiSCH [24] standards, proposed by the corresponding IETF working groups. 6LoWPAN defines efficient transmission of IPv6 packets over LR-WPANs, as defined by the 802.15.4 standard. Efficiency is achieved by applying header compression to transmitted IPv6 packets. 6TiSCH defines an IPv6 implementation based on the Timeslotted Channel Hopping mode (TSCH) added in the IEEE 802.15.4e [25] MAC amendment. IPv6 is preferred over IPv4 because of it's better scalability and the consequent redundancy of NAT solutions.

Network layer routing is commonly handled by the Routing Protocol for Low-Power and Lossy Networks (RPL) [26]. On the Transport layer UDP is preferred over TCP because it does not require a handshake which adds network overhead.

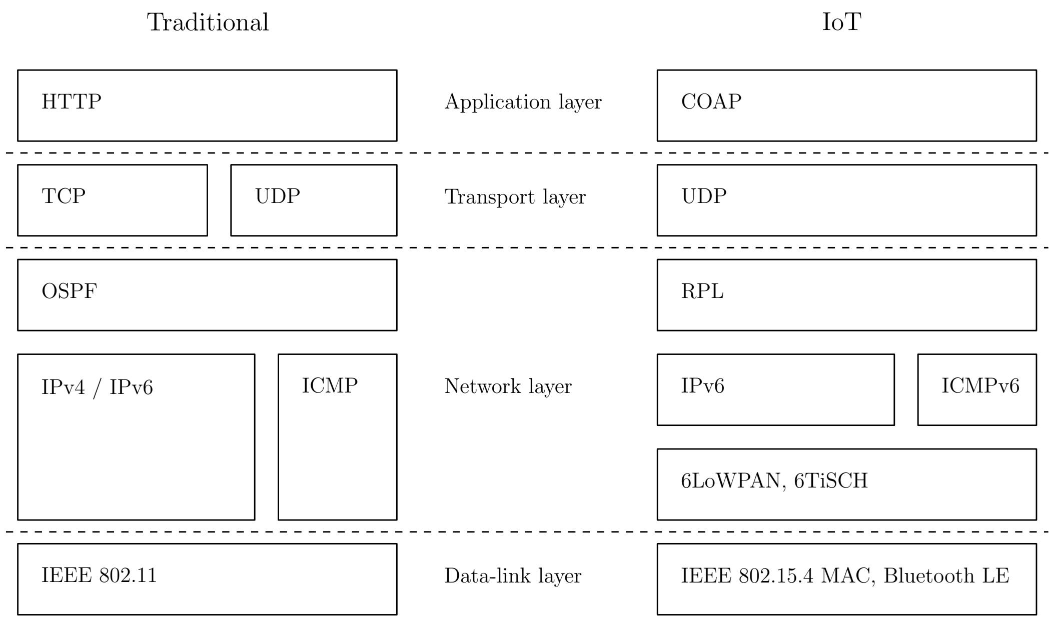


Figure 3.1.: A traditional IP protocol stack compared to the IoT stack

3.1.2. Multicast

Another technology used to reduce the overall network traffic and the nodes energy consumption is the IP multicast. Multicast is a form of group communication and has several advantages over the traditional IP unicast in environments where a single message addresses several nodes. Contrary to communication via unicast messages a multicast message sent to more than one host only needs to pass each link between two nodes once and thus saves bandwidth and battery. [27] shows that the use of multicast in WSNs using the 6LowPAN protocol may reduce the nodes energy consumption drastically compared to using unicast messages. However while multicast works with groups and defines several group management operations it does not provide a way to limit membership and assure confidentiality of communication because any node in a network may join any multicast group using the ICMPv6 protocol. As there is no native authorization/authentication management, confidentiality of the groups communication can only be achieved using an external cryptographic suite like IPsec, to encrypt and protect the groups traffic, with keys, which are selectively distributed to authenticated group members.

3.2. RIOT

RIOT is a real time operating systems designed for use in IoT and sensor networks. It was originally developed by a cooperation of Freie Universität Berlin, HAW Hamburg and INRIA under the LGPLv2.1 License. RIOT implements a modular microkernel architecture, supports native multi-threading and a standard ANSI-C and C++ API. Table 1 shows a comparison of RIOT's features with similar OSs used in sensor networks in terms of Memory requirement, C support, Multi-threading capabilities and Real-Time support.

OS	Min RAM	Min ROM	C Support	C++ Support	Multi-Threading	Modularity	Real-Time
Contiki	<2kB	<30kB	●	×	●	●	●
Tiny OS	<1kB	<4kB	×	×	●	×	×
Linux	~1MB	~1MB	✓	✓	✓	●	●
RIOT	~1.5KB	~5KB	✓	✓	✓	✓	✓

Table 3.1.: (✓)Full Support, (●)Partial Support, (×)No Support, Source: [4]

3.2.1. Reasons for choosing RIOT

Developing for RIOT brings several advantages compared to it's competitors:

- Microkernel Architecture: The microkernel architecture makes RIOT a very favourable choice especially for an IPsec implementation. One of the most prominent arguments against using IPsec in favor of TLS is IPsecs need to operate on the kernel level in order to maintain control on layer 3. This is not a problem in RIOT's microkernel architecture where the whole GNRC network stack runs in user space.
- Small memory footprint: Despite being having a lot of capabilities and a powerful scheduler RIOT has a low memory footprint. It requires less than 5KB of ROM and less than 1.5KB of RAM for a basic application.
- API compatibility with existing Libraries: Because of RIOTs standard C and C++ compatibility it allows the use of powerful external libraries. The implementation presented in chapter 6 uses this feature for external cryptographic libraries like *micro-ecc*.
- Native port: The native port is a virtual hardware platform that uses system calls and signals to emulate hardware at the API level. A RIOT stack compiled to native can be run as a normal process in *NIX environments which allows for most of the development to be done on a Linux system before porting it to the target Hardware.
- Hardware Support: RIOT comes with hardware support for many platforms, including MSP430, ARM7, Cortex-M and x86 architectures. Further RIOT has builtin driver support for a number of radio transceivers, sensors and other peripherals.

3.2.2. Generic (GNRC) Network Stack

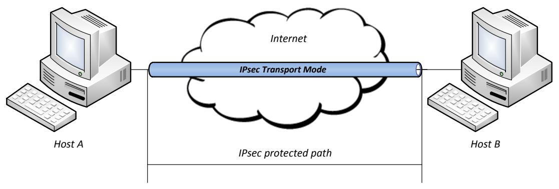
RIOT's network stack Generic (GNRC) Network Stack implements a full featured IoT protocol stack. It's modular architecture makes different protocol implementations easily interchangeable while it has a low memory footprint and is optimized for low power consumption. In order to provide this modularity, GNRC runs every module in a single thread with a own message queue. These threads communicate over RIOTs built in Inter Process Communication (IPC) module. The communication interface used between GNRC's modules is called "netapi". In order to receive packets from the network device, GNRC modules use the network registry (*netreg*), a database that manages which module receives which packet type. *netreg* allows modules to subscribe to a certain packet type for outgoing as well as incoming packets. A module will then handle the received package by for example appending the corresponding header and then passing it back to the *netreg* with it's updated type.

Most IoT network standard protocols (see Section 3.1.1) are already implemented in GNRC itself, others can be included in the form of external kernel modules.

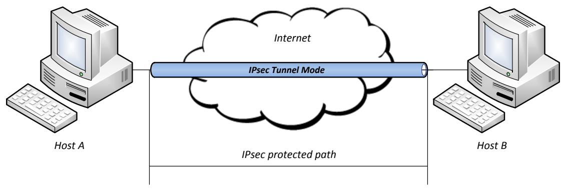
3.3. IKEv2 and IPsec

The “Security Architecture for the Internet Protocol” (IPsec) [28] is a security protocol suite designed to protect IP traffic on the network layer. It consists of two traffic security protocols: Authentication Header (AH) [29]. and Encapsulating Security Payload (ESP) [30]. Both Protocols work with so called Security Associations (SAs), which are established and maintained by a key management protocols like Internet Key Exchange [31] and stored in the security association database (SAD).

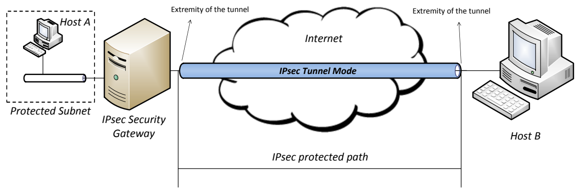
3.3.1. Modes of operation



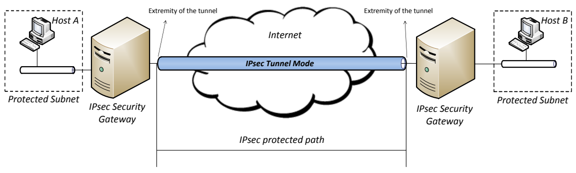
(a) Scenario for IPsec in Transport Mode.



(b) Endpoint to Endpoint Tunnel



(c) Endpoint to Gateway Tunnel



(d) Gateway to Gateway Tunnel

Figure 3.2.: Scenarios for IPsec Modes [7]

IPsec supports two modes of Operation: Transport Mode and Tunnel Mode. Both can be used by AH as well as ESP.

The Transport Mode (Figure 3.2a) is used for Endpoint-to-Endpoint or Endpoint-to-Gateway encryption. The original IP packet header stays unchanged with the exception of the next payload field, which is changed to AH or ESP. The IP payload is replaced by the corresponding encrypted packet. The original protocol stack stays intact.

Tunnel Mode is designed for Gateway-to-Gateway (Figure 3.2d) or Endpoint-to-Gateway (Figure 3.2c) security, but can also be used for Endpoint-To-Endpoint (Figure 3.2b) communication. The original IP packet is not modified in any way, and AH or ESP are constructed around it. A new IP header is prepended, containing the IPsec packet as payload. The receiving Gateway can unpack the full original IP packet and route it to the receiving endpoint.

3.3.2. Transport protocols

IPsec provides two transport security protocols: Authentication Header (AH) and Encapsulating Security Payload (ESP). Both may be used alone, combined or in a nested fashion. Authentication Header is used to provide connection-less integrity and data origin authentication for IP datagrams. It can optionally provide replay protection using the sliding window technique. AH protects as much of the IP header as possible, with the only exception being mutable fields that might be altered in transit like the flags and fragment offset fields. Encapsulating Security Payload can be used to provide confidentiality, data origin authentication, connection-less integrity, replay protection and traffic flow confidentiality. While ESP offers an “integrity only” and a “confidentiality only” mode, it typically employs both of those services. The anti-replay service can only be used if the integrity service is active. Traffic flow confidentiality will, opposed to the one offered by AH, only work in modes where the ultimate source or destination addresses are hidden (e.G. Tunnel mode).

3.3.3. Security Associations

IPsec uses the concept of Security Associations (SA) to describes how outgoing and incoming packets need to be processed, which encryption/authentication algorithms to use and which keys to use. Each SA describes the negotiated security features for a single connection protected with either AH, ESP or IKE.

Security Associations are typically established using a key management protocol like IKE and are saved in the Security Association Database (SAD). A SA contains a 32 bit Security Parameter Index (SPI) which serves as a unique identifier for the SA, the source and destination IP addresses, an identifier for the corresponding protocol (AH/ESP), the current sequence number, a number of algorithms and keys for authentication, integrity and encryption, the SAs lifetime and the used IPsec mode. A special kind of SA is the IKEv2 SA. It differs from the transport protocol SAs, as it describes not a simplex, but a duplex connection and it’s SPI is a 16 Byte value. An IKE SA has to store twice the keys an AH or ESP SA, as it needs to provide one set of keys for each direction of traffic.

3.3.4. IPsec Databases

IPsec specifies two databases to store control information, the Security Policy Database (SPD) stores Security Policies stating which and how security services should be provided to IP packets. A security policy is defined by a Traffic Selector (TS). Each TS may contain a source and destination IP address, a domain name and the used Transport Layer Protocol. It may also include a Port which allows to specify IPsec protection for a specific application running on a host. In addition to the traffic selector each Security Policy contains the IPsec protocol used on the connection, the used protocol mode, additional parameters like the policies lifetime and an action (discard, secure or bypass) which defines how incoming packets are treated. The second database used is the Security Association Database (SAD) that stores negotiated SAs. Each entry in the SAD is associated to an entry in the SPD. A ESP or AH SA holds the specific information necessary to secure a single simplex connection. Figure 3.3 shows how SA, SP and TS work together. Due to a difference in format, systems providing an IKEv2 implementation require an additional SAD for internally used IKE SAs.

3.3.5. IKEv2

In order to protect their communications both participants need to have a common SA. Only if both peers use the same algorithms with the same keys they are able to decrypt each others traffic. The used SA may be statically configured for both hosts, or negotiated by a key management protocol.

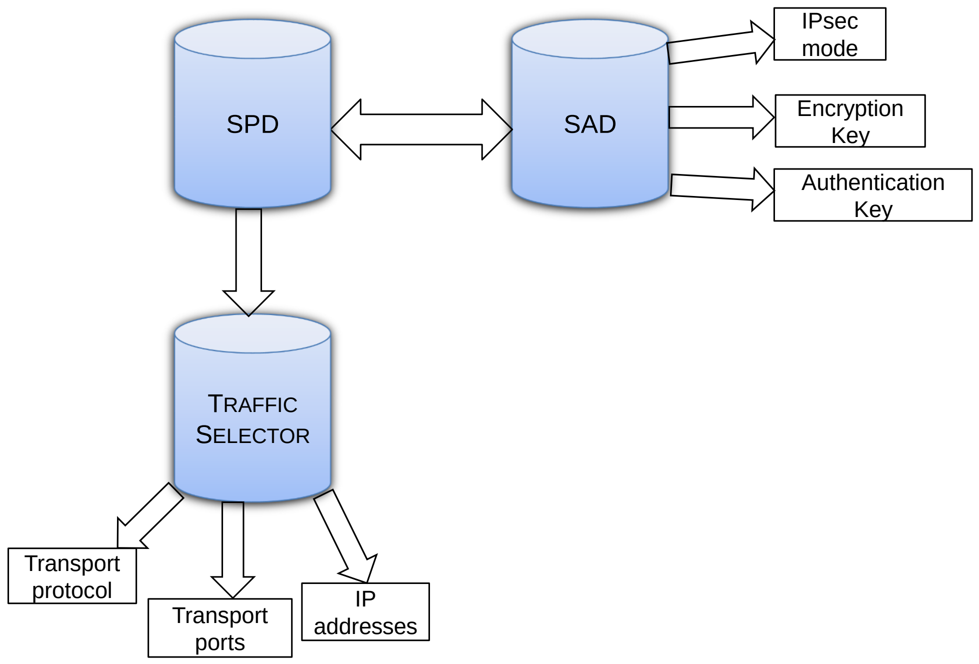


Figure 3.3.: IPsec Databases [7]

The Protocol commonly used to establish SAs is called Internet Key Exchange (IKE) [31] in it's current version referred to as IKEv2. It performs mutual authentication between two hosts and establishes an IKE SA for secure communication between these two. This SA can be used to derive ESP and AH security associations in order to provide end-to-end encryption. In the IKEv2 protocol all communication consists of pairs of messages, a request and a response, named Exchanges. The first two exchanges IKE_SA_INIT and

IKE_AUTH are used to establish the IKE SA and are therefore mandatory. They have to be performed before any other exchange takes place. IKE_SA_INIT initiates the secure channel, negotiates cryptographic algorithms, exchanges nonces and Diffie-Hellman values and allows the responder to request a certificate for authentication. After the first exchange both parties derive cryptographic keys used to protect their following IKE communication. The IKE_AUTH exchange authenticates initiator and responder by either shared secret or asymmetric cryptography via certificates, integrity protects the first exchange, and optionally allows the negotiation of a child SA.

After a secure authenticated channel is established both peers may exchange control information with the Informational exchange or derive new child SAs with the CREATE_CHILD_SA exchange.

[6] describes a minimal subset of the IKEv2 protocol used by the initiator, designed for the use in constrained nodes such as embedded devices. The minimal node acts as the initiator

and supports only the IKE_SA_INIT and IKE_AUTH exchanges and their required payloads. Authentication is accomplished using a preshared key. The child SA must be established during the AUTH exchange. The minimal IKEv2 implementation is able to interoperate with all IKEv2 responder implementations that are compliant with [31].

3.3.6. Group-IKEv2

The G-IKEv2 protocol [18] is currently proposed as a new standard for group key management. It is strongly related to IKEv2 and is used to establish secure group communications in contrast to IKEv2 that only allows to negotiate SAs for secure point to point communication. Groups are managed by the Group Controller/Key Server (GCKS), who represents the responder in the initial IKE exchange. The initiator who wants to join a group is called Group Member (GM). The initial IKE_SA_INIT exchange and mutual key generation is the same as in IKEv2, the following GSA_AUTH exchange authenticates GM and GCKS, integrity protects the initial exchange and additionally allows the GM to request group membership to a specific group. The GCKS validates the GMs authorization to join the group and responds with the Group SA policy and the group keys. The GM can now communicate with other group members protecting its traffic with the shared keys.

4. Requirements of G-IKEv2 for RIOT

A system implementing secure group communication via G-IKEv2 on an IoT platform needs to fulfill a number of requirements in order to be able to provide the security services defined in Chapter 2 without limiting the devices main functionality. The requirements can be grouped in two categories: Functional requirements, describing features a implementation must provide, and the underlying Non-functional requirements, defined by resources a constrained system must provide for an implementation to function properly. It should be noted that a full requirements analysis is not in the scope of this work.

4.1. Functional requirements

In order to fulfill the security services introduced in Chapter 2, as well as the use cases in Section 2.2, an implementation will have to offer a number of mandatory functions. The most basic functional requirements is the ability to communicate over some form of network. IPsec's key management protocols use UDP as transport protocol, thus a network stack providing OSI layers 1 to 4 is required. These requirements can be considered fulfilled if the system runs RIOT OS and is equipped with some form of network functionality.

In order to guarantee a secure nonce as well as a secure Diffie-Hellman value the minimal client needs to have a reliable source of randomness, whether it is a hardware random number generator or a sufficiently seeded secure pseudorandom function.

4.1.1. Cryptographic requirements

Secure communication requires the use of a number of cryptographic functions and algorithms in order to guarantee confidentiality, source authentication and other desired mechanisms. A G-IKEv2 implementation must support at least one encryption algorithm, one integrity protection algorithm a random number function and a Diffie-Hellman group. An enumeration of all algorithms supported by IKEv2 and G-IKEv2 can be found in [32].

[6] defines a subset of relevant algorithms including the Advanced Encryption Standard (AES) [33], in cipher block chaining (CBC) mode or counter with CBC-mode (CCM), as encryption algorithm. HMAC-SHA1 as pseudorandom function, AES-XCBC and HMAC-SHA1 as integrity algorithms and the 1536 and 2048 MODP Diffie-Hellman groups. As an addition the ECP Diffie-Hellman groups described in [34] should also be considered due to their performance advantages and small key sizes. It is also noteworthy that since the release of [6] SHA1 has been broken by [35] and should thus be replaced by a stronger alternative like SHA256.

4.2. Non-functional requirements

Resource requirements can be divided into CPU requirements, memory requirements, which are further divided into RAM and Flash requirements, and energy requirements.

Table 4.1.: Memory need of different ciphers

Cipher-Suite	**IKE_SA_INIT**	**GSA_AUTH**	**Key length (2 per SA)**
Fixed Size	88 B	98 B	—
Encryption:			
- AES128-CBC	—	+ 16 B + 15 B[a]	16 B
- AES256-CBC	—	+ 16 B + 15 B[a]	32 B
Integrity:			
- SHA1-96	—	+ 12 B	20 B
- SHA256-160	—	+ 16 B	32 B
- AES-XCBC-96	—	+ 12 B	16 B
Combined:			
- AES128-CCM	- 8 B [b]	+ 8 B + 8 B[a]	19 B
- AES256-CCM	- 8 B[b]	+ 8 B + 8 B[a]	35 B
- AES128-GCM	- 8 B[b]	+ 8 B + 8 B[a]	19 B
- AES256-GCM	- 8 B[b]	+ 8 B + 8 B[a]	35 B
PRF:			
- SHA1-96	—	+ 20 B	—
- SHA256-160	—	+ 32 B	—
DH Group:			
- ECP256	+ 64 B	—	—
- curve25519	+ 64 B	—	—
- 2048MODP	+ 256 B	—	—

[a] a maximum of $+x\,B$ for padding need to be added

[b] packet size is reduced by the size of the Integrity Proposal

4.2.1. CPU requirements

The CPU requirements strongly depend on the negotiated cipher suite. The use of stronger cryptographic algorithms usually comes at the expense of processing power to generate cipher text. At the very minimum the CPU should be able to handle network operations and at the same time be able to calculate all cryptographic algorithms mentioned in Section 4.1.1 without breaking eventual real time guarantees. It should be noted that the encryption and integrity algorithms need to be calculated for every network packet while the pseudorandom and Diffie-Hellman calculation only happen once per initiated exchange.

[36] benchmarks various microcontrollers, as they are used in IoT, using widespread cryptographic algorithms including AES128, SHA1 and RSA. Real time guarantees are considered fulfilled if a threshold of 300ms for cryptographic operations is not exceeded. They come to the result that "[...] symmetric ciphers, for example, AES, are fast enough to be implemented into security solutions that run on constrained devices [...]" and further "security solutions based on asymmetric cryptographic operations, for example, RSA, ECDH, [...] need more time to execute".

4.2.2. Memory requirements

The flash memory needs to provide enough space for the implementation including the necessary environment providing network functionality and other necessary functions (e.G. RIOT and external cryptographic libraries). RIOT itself has a minimum requirement for 5 KB ROM. The total requirement for the implementation can not be predicted, as a number of modules and libraries will add overhead.

The biggest items in the systems RAM are the network buffer, the SAD, and the stack. The network buffer is used to store incoming and outgoing network packets before they are further processed. Every G-IKEv2 implementation must be able to handle network packets of 1280 Bytes size [18]. This is therefore the lower limit of the network buffers memory requirement, assuming that, at any point in time, only one packet is stored in the buffer. The actual packet size depends on the used cryptographic algorithms as well as their key lengths. Table 4.1 shows the impact of different cryptographic algorithms and key lengths on the size of the IKE_SA_INIT and GSA_AUTH packets. The same can be applied to the SADs size as each SA contains a set of keys. The implementation must be able to store at least one IKE SA for the initial exchange and rekeying as well as one TEK SA for the use in ESP or AH.

Further, it should be considered that some RIOT modules may add additional stacks, buffers and other forms of memory overhead.

4.2.3. Energy Requirements

Sensor nodes may be running on battery, while being required to function for several years without maintenance, in order to be feasible. Aside from the systems scheduler, one of the most severe energy consumers is the network stack. As a solution, most IoT network standards are optimized for low power consumption, which is achieved by minimizing network packets' sizes as well as their quantity. A G-IKEv2 implementation for energy constrained systems should thus sustain energy efficiency by reducing network traffic during the key exchange and group negotiation to a minimum.

4.3. Summary

Table 4.2.: Minimal requirements for a system implementing G-IKEv2

Functional Requirements
RIOT OS Support
Network Functionality
Cipher Suite
- Encryption
- Integrity
- PRF
- Diffie-Hellman
Source of Randomness
Non-functional Requirements
Flash Memory
RAM
Processor Performance
Energy Efficiency

Based on a simple analysis of the mandatory security services of a group key management protocol, combined with the common use cases, a number of requirements for a system implementing G-IKEv2 on RIOT OS have been worked out. A summery can be found in Table 4.2.

5. Minimal G-IKEv2 Protocol Design

This chapter describes a minimal subset of the Group-IKEv2 protocol adjusted to the needs and limitations of embedded systems and the IoT. The resulting protocol combines the functions of the G-IKEv2 protocol with the considerations for constrained systems of [6].

G-IKEv2, like GDOI, uses UDP port 848, because, same as IKEv1 and IKEv2, they serve the same function and can be distinguished by the version number in the IKE header. Similar to IKEv2, all unicast communications consist of a pair of messages called an "exchange". Because of GIKEv2 being a reliable protocol every request requires a response. The initiator must retransmit a request until it receives a response or deems the IKEv2 SA failed. Messages sent using multicast from the GCKS must not be replied by the GM.

The presented protocol only supports the two initial exchanges of the G-IKEv2 protocol, which are used to establish a secure channel and join a group. Additionally it can handle the GSA_REKEY messages from the server which are used to change a SAs key's or attributes. Other exchanges and messages, like the informational exchange, are omitted. Group management operations other than a group join are not supported due to the constraints of the targeted clients, which do not need to create or delete groups. The "GM registration" operation is performed during the GSA_AUTH exchange, as the separate GSA_REGISTRATION exchange is not in the scope of the minimal protocol, similar to the miminal IKEv2 where the negotiation of the transport SA must happen during the IKE_AUTH exchange. Leaving a group is handled from the group members by deleting the local SAs.

Only mandatory payloads that are explicitly mentioned in the following are part of the protocol, other payloads, like the *Notify* payload, simply are ignored. Despite the absence of a number of features, an implementation of this minimal G-IKEv2 protocol should be compatible with any implementation compliant with [31].

5.1. Secure Channel Establishment

Figure 5.1 illustrates the minimal exchanges necessary to establish a mutual IKE SA. Each message is preceded by a IKEv2 header, it contains two Security Parameter Indexes (SPIs), one for the responder and one for the initiator, a IKEv2 version number, the message ID and exchange type, and various flags. Only the SPIs are used to match an incoming Packet to it's corresponding SA, IP addresses and ports must not be used.

In order to authenticate each other, the Group Controller / Key Server (GCKS) and Group Member (GM) need to share an authentication key over a secure channel.

5.1.1. IKE_SA_INIT

The initial exchange called IKE_SA_INIT is identical with IKEv2's initial exchange. In the minimal protocol, the initiator sends a Security Association SA, a Key Exchange KE and a nonce N_i payload. The SA payload contains exactly one proposal stating the cryptographic algorithms and the Diffie-Hellman group the initiator supports for the IKE SA, securing the

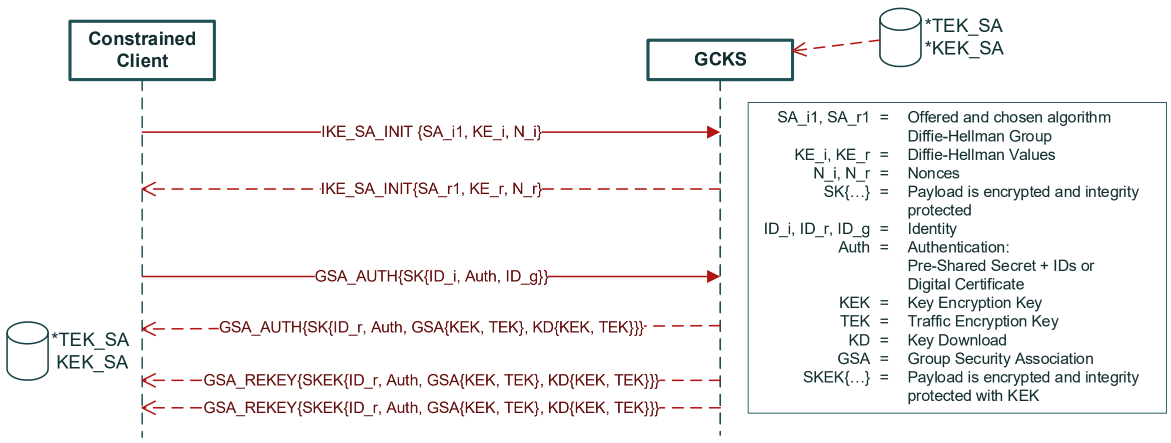

Constrained Client
GCKS
*TEK_SA
*KEK_SA
IKE_SA_INIT {SA_i1, KE_i, N_i}
IKE_SA_INIT{SA_r1, KE_r, N_r}
GSA_AUTH{SK{ID_i, Auth, ID_g}}
GSA_AUTH{SK{ID_r, Auth, GSA{KEK, TEK}, KD{KEK, TEK}}}
*TEK_SA
KEK_SA
GSA_REKEY{SKEK{ID_r, Auth, GSA{KEK, TEK}, KD{KEK, TEK}}}
GSA_REKEY{SKEK{ID_r, Auth, GSA{KEK, TEK}, KD{KEK, TEK}}}
SA_i1, SA_r1 = Offered and chosen algorithm Diffie-Hellman Group
KE_i, KE_r = Diffie-Hellman Values
N_i, N_r = Nonces
SK{...} = Payload is encrypted and integrity protected
ID_i, ID_r, ID_g = Identity
Auth = Authentication: Pre-Shared Secret + IDs or Digital Certificate
KEK = Key Encryption Key
TEK = Traffic Encryption Key
KD = Key Download
GSA = Group Security Association
SKEK{...} = Payload is encrypted and integrity protected with KEK

Figure 5.1.: Simplified G-IKEv2 exchange for minimal implementations. [37]

following exchanges. In total 4 algorithms need to be negotiated: an encryption algorithm, an integrity algorithm, a Diffie-Hellman group and a pseudorandom function (PRF). The KE payload contains a Diffie-Hellman value, used to generate a shared secret between GM and GCKS. The values group should match the group proposed in the SA payload. N_i contains a random value called nonce.

The GCKS responds with a SA payload containing the received proposal, a KE payload containing the server's Diffie-Hellman value g^r and it's own nonce N_r. The GM validates the received proposal and then proceeds to generate the shared Diffie-Hellman secret g^{ir} with the server's value.

5.1.2. Generating Keying Material

With the PRF negotiated in the IKE_SA_INIT exchange and the nonces and Diffie-Hellman values exchanged, both parties are able to generate secure keying material, used to protect the consecutive exchanges. The minimal client supports only cryptographic algorithms using fixed-size keys where a randomly chosen value of the size can be used as key.

In the first step a quantity called SKEYSEED is computed from a combination of the nonces and the Diffie-Hellman shared secret known from the first exchange:

$$SKEYSEED = prf(N_i|N_r, g^{ir})$$

The SKEYSEED is further used to generate the keys securing the following communication. Because the generated keying material is usually longer than PRFs output, keys are read from the *prf+* function, that outputs a consecutive byte stream using PRF in cipher block chaining mode. It is defined as follows (| indicates concatenation):

$$prf+(K,S) = T1|T2|T3|T4|...$$

where:

$$\begin{aligned}T1 &= prf(K, S|0x01)\\ T2 &= prf(K, T1|S|0x02)\\ T3 &= prf(K, T2|S|0x03)\\ T4 &= prf(K, T3|S|0x04)\end{aligned}$$

The keys are consecutively read from the output like this:

$$SK_d|SK_{ai}|SK_{ar}|SK_{ei}|SK_{er}|SK_{pi}|SK_{pr} = prf+(SKEYSEED, N_i|N_r|SPI_i|SPI_r)$$

For each negotiated algorithms, two keys are derived. Each key secures only a single direction of traffic. SK_{ai} and SK_{ar} are the integrity algorithm keys, SK_{ei} and SK_{er} are used to encrypt packets and SK_{pi} and SK_{pr} are used to generate the authentication values. SK_d is used in IKEv2 to generate child SAs. It has no use in G-IKEv2, but is still derived in order to preserve compatibility.

5.1.3. GSA_AUTH

The GSA_AUTH exchange authenticates the previous messages and allows the GM to join a group. It consists of a single SK payload, containing an ID_r, AUTH, GSA and KD payload. All payloads inside the SK payload are encrypted and integrity protected by the keys derived in the previous step.

The GM asserts its identity in the ID_i payload. A identification value usually contains a IPv6 address identity, alternatively an arbitrary string may be used. The AUTH payload proves knowledge of the shared secret corresponding to ID_i and additionally integrity protects the content of the first message. The ID_g payload signals the wish to register for a group by stating the groups identity (usually a multicast group IP).

The GCKS responds with it's own identity in the ID_r payload and authentication and integrity protection of the IKE_SA_AUTH response in the AUTH payload. The minimal client usually does not need to check the ID_r, because it only knows one shared secret. A server proving knowledge of this secret can be trusted in any case. If the server authorizes the GM to be part of the requested group, it sends the GSA policy and KD payloads, providing the GM with the SAs and corresponding keying material for the group it registered for. The GSA policy contains used security algorithms and other SA attributes of at least one Data-security SA (TEK) and optionally one rekey SA (KEK), used for the groups rekeying.

At this point the GM can securely communicate with other group members using the TEK SAs and handle the servers GSA_REKEY request with the KEK SA. The established IKE SA is no longer used and may be deleted in order to save memory.

5.2. Group rekeying

The G-IKEv2 Rekey is a protected multicast message, sent by the GCKS, in order to inform group members of changed group SA attributes, keys or both. It is protected by the groups

KEK SA and contains at least a GSA, a KD and an AUTH payload. The GSA and KD payloads may contain a KEK payload updating the existing KEK and keys and a TEK payload creating a new TEK SA. The AUTH payload authenticates the GCKS. It should be generated using asymmetric cryptography like a digital signature. Unlike the AUTH value in the GSA_AUTH exchange a shared secret can not provide source authentication because the secret is known to all group members. The message is sent to multiple GMs using IP multicast and thus does not require a response. Anti-replay protection is achieved by just accepting message IDs higher than the current KEK SA message ID.

6. Implementation

The minimal G-IKEv2 client is implemented in the C programming language as an external RIOT module. It makes use of the GNRC network stack and especially GNRC's packet buffer. The module includes a working configuration using cryptographic algorithms available in RIOTs system modules, or via RIOTs packet interface, as well as a minimal implementation of a SAD that can be used to interface with eventual ESP or AH modules. The module can be seamlessly integrated into GNRC using the *netapi*. The implementation does not use any dynamic memory allocations to preserve RIOTs real-time nature and to guarantee a clearly defined memory limit.

6.1. IPsec Databases

```c
// Insert SA into database
ikev2_sa_t *ikev2_sad_insert(ikev2_sa_t *sa);

// Get SA with given SPI from database
ikev2_sa_t *ikev2_sad_get(uint64_t spi_i, uint64_t spi_r);

// Delete SA with given SPI from database
int ikev2_sad_delete(uint64_t spi_i, uint64_t spi_r);
```

Code 6.1: ikev2_sad.h

The implementation includes two Security Association Databases: an IKEv2 SAD, used only internally in the G-IKE module to store the initial and KEK SAs, and an IPsec SAD, storing the TEK SAs received from the GCKS.

The reason two different databases are used is the different structure of IKEv2 and IPsec SAs, as well as the different scope of the two. IKEv2 SAs have a SPI value of two times 8 byte, once for the initiator and once for the responder. Additionally they need to store two key sets, one for each direction. TEK SA SPIs are 4 byte in total and each type of key has to be stored only once, as they protect only a single direction of traffic.

Both SADs consist of an array of statically allocated memory who's size can be configured at compile time. The IKEv2 SAD has a pre-configured size to store two SAs, one initial IKE SA and one KEK SA. The IPsec SAD is configured to store one TEK SA by default. Each database has a simple API supporting insert, get and delete operations. Inside the database, memory is managed with the help of a linked list _first_free_sa containing free segments of data in the database. In the following the basic functions of the IKEv2 SAD API (shown in Code 6.1) are explained. The IPsec SAD API works similarly, with the exception of the different SPI format.

ikev2_sad_insert(ikev2_sa_t *sa): returns a pointer to the first element of _first_free_sa and consequently removes it from the list. If a pointer to a SA is passed in sa the corresponding structure is copied into the database, otherwise an uninitialized ikev2_sa_t is returned.

ikev2_sad_get(uint64_t spi_i, uint64_t spi_r): iterates the array until a SA sharing the SPIs is found and returns a pointer to the corresponding SA, otherwise NULL is returned.

ikev2_sad_delete(uint64_t spi_i, uint64_t spi_r): calls ikev2_sad_get() and when a SA is returned deletes it from the database by inserting it into _first_free_sa.

6.2. Configuration

The usage of the module requires some configuration to be able to build a secure channel and negotiate SAs. The configuration settings are stored in a configuration file *config.h* and are compiled into the module. Configuration at runtime is not supported.

Configuration parameters include the GCKS IP address and port, identification data used for the ID payloads, a shared secret used for authentication and the available cipher suite. The minimal implementation comes with a pre-configured cipher suite using AES128 for encryption, SHA1-96, SHA1-160 or SHA-256 for integrity and as pseudorandom function and the SECP256R1 Diffie-Hellman group. A more detailed description of available algorithms and the supporting libraries can be found in section 6.6. Additionally various other sizes, including the number of SAs in the SADs and the nonce length, are declared in the configuration file.

The cryptographic algorithms proposed in the IKE_SA_INIT request can be selected by passing the corresponding constants in the Makefile or setting them in the configuration file.

6.3. Networking

This section deals with the minimal G-IKEv2 client's networking functions. The first subsection summarizes RIOT's network packet API, the second subsection describes the basic structure of packet processing functions used throughout the implementation.

6.3.1. Packet structure

GNRCs packets are stored in a packet buffer *pktbuf*, in the form of a linked list of packet snip (gnrc_pktsnip_t) structures, where each represents either header or payload of a packet. A gnrc_pktsnip_t contains a pointer to the next element of the list, a pointer to it's data, it's length, it's *nettype* and the number of users. These structures, as well as their data, are stored in a statically allocated *pktbuf* array. An example for such a packet can be seen in Figure 6.1 It depicts a UDP packet containing an UDP payload data segment, an UDP header and an IPv6 header in a list of three packet snips.

Packet snips are initialized with a call to the gnrc_pktsnip_add() function which takes the packets next element, a data array, the packet size and the protocol type as an argument and returns a pointer to the newly allocated structure. Internally, this initialization leads to two calls of the packet buffer's allocation function. The first call returns the first segment in the buffer, big enough to hold the gnrc_pktsnip_t structure. The second call does the same for the packets data segment. Consequently RIOTs network packets are stored in no particular

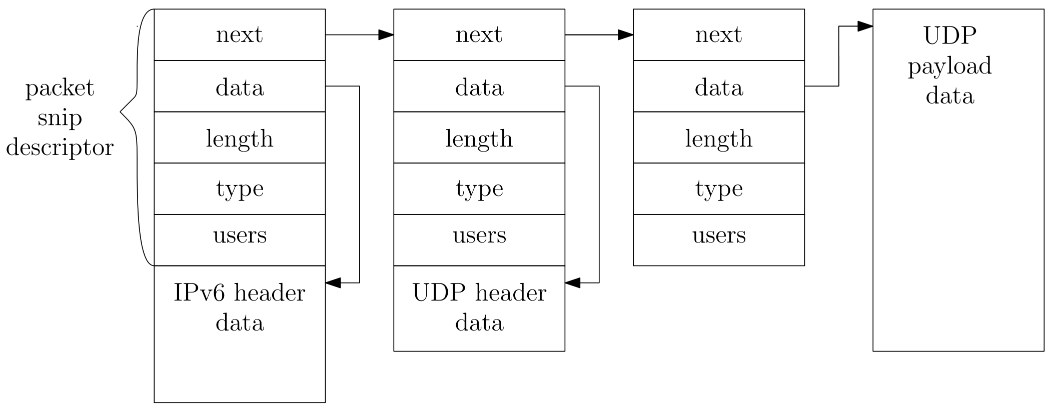
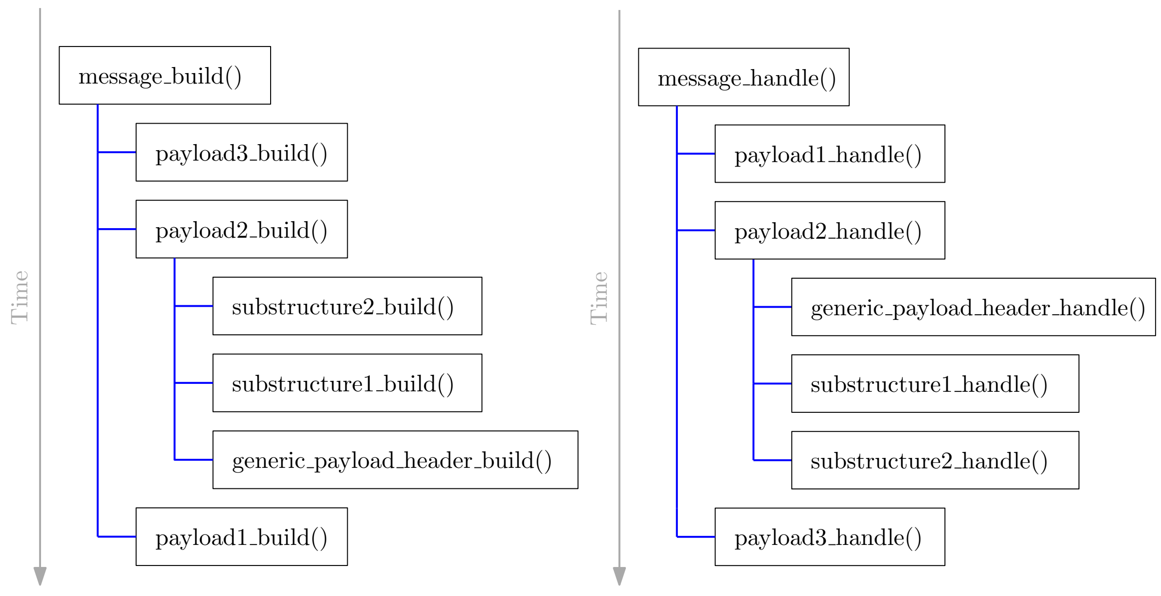


Figure 6.1.: Example of a packet in GNRCs packet buffer [38]

order in the *pktbuf*. When the packet is not needed anymore it can be freed recursively, by calling the gnrc_pktsnip_release() function, passing the head of the linked list as input.

Received network packets come in the form of a single element linked list containing the full packet data. A packet handler will use the gnrc_pktbuf_mark() function, which marks a number of bytes in a packet and appends it as a new element to the linked list. This way the packet is split into headers and payloads, for further processing.

6.3.2. Message processing



(a) msg_build() structure (b) msg_handle() structure

Figure 6.2.: Handle and build function structure for an example message

The message processing is done in a hierarchical way. A message processing function calls the underlying payload processor, which eventually call substructure processors. This structure helps minimizing the code base and promotes reusability. Common patterns, for example the generic payload header, might be used in more than one message or payload. Handle and build functions for a generic example packet are shown in Figure 6.2. Functions are called in order, from top to bottom. Indentation visualizes the hierarchical call structure.

The message build function start building the last payload in the message in a new packet snip and appends every further payload as new head to the list. Lastly the generic G-IKE message header is appended. The payload build functions work the same way as the message build function, only on a lower level in the payload hierarchy. An example for a build function is shown in Section 6.4.2, where the sa_init_build() function is explained step by step.

Message handlers work contrary to the build functions. The message is parsed from the first to the last payload, using the gnrc_pktbuf_mark() function to split the single packet snip into a linked list of headers and payloads which are then parsed and further processed.

6.4. G-IKEv2 Client Structure

In the following the clients structure, including setup, sending of the initial message and the overall protocol handling, is explained in procedural order.

6.4.1. Initial setup

```
void gikev2_run() {
  /* Initialize SADs */
  ipsec_sad_init();
  ikev2_sad_init();

  /* Initialize Diffie-Hellman Context */
  dh_context_t *dh_ctxt;
  dh_ctxt = init_dh_ctxt();

  /* Initialize SA */
  ikev2_sa_t *sa = ikev2_sad_insert(NULL);
  ikev2_sa_init(sa, config, dh_context);

  /* Generate AUTH key*/
  sha1_context auth_key_ctxt;
  uint8_t auth_key[PRF_DIGEST_SIZE];
  sha1_init_hmac(&auth_key_ctxt, SHARED_SECRET, strlen(SHARED_SECRET));
  sha1_update(&auth_key_ctxt, KEYPAD, strlen(KEYPAD));
  sha1_final_hmac(&auth_key_ctxt, auth_key);

  /*Initialize AUTH values with AUTH key*/
  sha1_init_hmac(&sa->auth_i.auth_ctxt, auth_key, PRF_DIGEST_SIZE);
  sha1_init_hmac(&sa->auth_r.auth_ctxt, auth_key, PRF_DIGEST_SIZE);
```

Code 6.2: gikev2_run(): Security Association setup

Before starting the protocol exchange, various components need to be set up. Code 6.2 shows the initialization steps necessary before starting the protocol handler. First the two SADs need to be initialized by creating the linked list of free elements. After a random Diffie-Hellman key is generated the first IKEv2 SA is inserted into the IKEv2 SAD containing the

```
25  // Initialize message handling
26  static msg_t _msg_q[MES_QUEUE_SIZE];
27  msg_init_queue(_msg_q, MES_QUEUE_SIZE);
28  gnrc_netreg_entry me_reg = { .demux_ctx = GIKEV2_PORT,
29                               .pid = thread_getpid()};
30  gnrc_netreg_register(GNRC_NETTYPE_UDP, &me_reg);
31
32  // Build and send IKE_SA_INIT
33  gnrc_pktsnip_t* pkt;
34  sa_init_build(pkt, sa);
35  gikev2_pkt_send(pkt);
36
37  // Handle incoming messages
38  while (1) {
39    pkt = gikev2_msg_recv();
40    gikev2_message_handle(pkt);
41  }
```

Code 6.3: gikev2_run(): Network setup

newly created Diffie-Hellman values and the identities and cipher suite read from *config.h*. After the SA is set up, the authentication key is generated from the shared secret, set in the configuration file and the fixed KEYPAD. The key is used to initialize the authentication value hashes for initiator and responder.

The GIKEv2 module is integrated into GNRC via the network registry (*netreg*) module (Code 6.3). Network messages received from the *netreg* are transmitted via the kernels IPC message interface, which requires a message queue to be set up (lines 26-27).

In order to receive network messages from the *netreg* module, the GIKEv2 module needs to register it's thread to the *netreg* database. This is done creating a *netreg* entry containing the threads process ID *pid* and the *demux* context. The *demux* context is a rule used to distinguish packets of the same *nettype*. UDP packets are distinguished by their ports, thus GIKEv2s *netreg* entry *demux* context is the GIKEV2_PORT.

6.4.2. IKE_SA_INIT request

After the IKEv2 SA and networking are initiated, the IKE_SA_INIT is built from the configured cipher suite. Packets are built from back to forth, as described in Section 6.3.2.

Code 6.4 shows the sa_init_build() function. After the random nonce is generated the packets payloads are built from back to forth. The nonce and KE payloads are straightforward as they only consist of a byte string containing the nonce or Diffie-Hellman value and a fixed header. The sa_payload_build() function iteratively builds the proposal substructure from the cipher suite defined in *config.h*. Finally the fixed IKEv2 header is attached containing the SAs SPI, the exchange type (IKE_SA_INIT) and the overall length of the packet.

6.4.3. Protocol handler loop

Code 6.5 shows the simplified structure of the gikev2_message_handle() function. After the message header is parsed, the corresponding SA is looked up from the database using the

```c
gnrc_pktsnip_t *sa_init_build(gnrc_pktsnip_t *pkt, ikev2_sa_t *sa)
{
    prng_read(sa->nonce_i, NONCE_LENGTH);

    pkt = nonce_payload_build(pkt, PL_NONE sa->nonce_i, NONCE_LENGTH);
    pkt = ke_payload_build(pkt, PL_NONCE, sa->dh_ctxt);
    pkt = sa_payload_build(pkt, PL_KE, sa->proposals, sa->proposal_num);
    pkt = gikev2_hdr_build(pkt, sa->spi_i, sa->spi_r, PL_SA,
                           IKE_SA_INIT, false, false,
                           true, 0, gnrc_pkt_len(pkt)+IKE_HDR_SIZE);
    return pkt;
}
```

Code 6.4: sa_init_build()

```c
int gikev2_message_handle(gnrc_pktsnip_t *buf) {
    ikev2_hdr_t hdr;
    size_t len = buf->size;

    /*Parse generic G-IKE header*/
    _gikev2_hdr_handle(buf, &hdr);

    ikev2_sa_t *sa = ikev2_sad_get( byteorder_ntohll(hdr.i_spi),
                                    byteorder_ntohll(hdr.r_spi)
                                  );

    /* Handle messages */
    if (hdr.ex_type == IKE_SA_INIT && sa->state == IKE_SA_INIT) {
        if(sa_init_handle(buf, sa)) return EXIT_ERROR;
        gnrc_pktsnip_t *req;
        gsa_auth_payloads_build(req, sa);
        gikev2_pkt_send(pkt)
    } else if (hdr.ex_type == GSA_AUTH && sa->state == GSA_AUTH) {
        gsa_auth_handle(buf, sa);
    } else if (hdr.ex_type == GSA_REKEY && sa->state == GSA_REKEY) {
        gsa_auth_handle(buf, sa);
    } else {
        return EXIT_UNKNOWN_PAYLOAD;
    }
    return EXIT_SUCCESS;
}
```

Code 6.5: Message Handler

```c
int cipher_encrypt_cbc( cipher_t *cipher,
                        uint8_t iv[IV_LEN],
                        uint8_t *input,
                        size_t input_len,
                        uint8_t *output )
```

Code 6.6: cipher_encrypt_cbc() definition

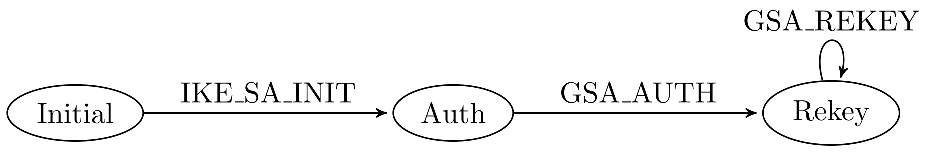


Figure 6.3.: Security association state machine

SPIs from the message header. Next the message is passed to a handler function, depending on the SAs state and the exchange type set in the header. Handler functions follow the scheme described in Section 6.3.2. The SAs states are defined by the expected response message. Figure 6.3 shows the SAs states in a simple state machine. In total three states can be distinguished corresponding to the IKE_SA_INIT, GSA_AUTH and GSA_REKEY exchanges. States are changed by successfully handling the expected response. After the response was handled, the client builds and sends the following request. The initial IKE SA starts in the *Initial* state and reaches the *Rekey* state after the GSA_AUTH was successfully handled. When the final SA state is reached, the implementation is caught in a loop, only handling GSA_REKEY Messages. KEK SAs received from the server always are in the *Rekey* state.

6.5. Complications

This section deals with complications and hardships found during the implementation process. Several changes to RIOT have been necessary in order to function properly.

6.5.1. Encrypting packets in the packet buffer

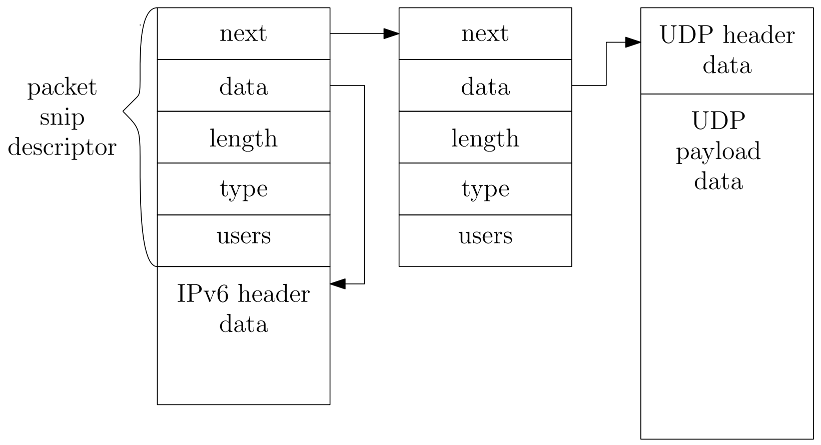


Figure 6.4.: Packet from Figure 6.1 after the udp header packet snip was merged

The packet buffer's structure has several advantages in terms of memory efficiency as it reduces packet duplication and removes the necessity to move or copy data in the packet buffer. A main downside of this, however, is, that it is not compatible with most cryptographic library APIs, including RIOTs own *Crypto* module. Code 6.6 shows the definition of the *Crypto* modules cipher_encrypt_cbc() encryption function. The function expects an arbitrary length byte array as input data and an array of the same length for the output as argument. Consequently, encrypting a GNRC packet is not possible, as long as it contains more than one packet snip. It can be worked around, without changes to RIOT, by aggregating the linked list's data segments in one consecutive buffer outside the *pktbuf*. This would, however, in the worst case require a second buffer the size of *pktbuf* and thus destroy all previous efforts of memory efficiency and render the packet buffer itself useless.

A solution is the introduction of a new packet buffer API function gnrc_pktbuf_merge() , which merges a linked list of packet snips to a single packet snip pointing to a newly allocated *pktbuf* segment, containing the former list's data in a continuous memory segment. The downside is a loss of control information as there is no way to keep the information, stored in the lost packet snip descriptors. When used for encrypting or hashing the packet this should not be a problem as the encrypted or hashed packets rarely have to be modified afterwards. Moreover this solution still requires a *pktbuf* of twice the size of the largest packet, but it does, in contrast to an external memory segment, integrate well into the GNRC API. An example of a "merged" packet can be seen in Figure 6.4. It shows the UDP packet from Figure 6.1 after pktbuf_merge() was applied to the second element of the list. The packet contains the same data as in it's original form, the UDP header and data segments are stored in a single *gnrc_pktsnip_t*.

6.5.2. Stack size

The implementation was mostly built using the virtual RIOT environment described in Section 3.2. Deploying the finished build on hardware systems turned out to lead to several problems. RIOT's default stack size for Cortex-M0 CPUs of 1024 Byte is too small for the implementation and results in randomly seeming RIOT kernel panics. As a workaround the stack size is increased to 2048 Byte.

6.6. Supporting modules and libraries

In order to provide a compatible cipher suite the implementation uses several RIOT modules, as well as external cryptographic libraries. In the following, various compatible libraries for the "minimal G-IKEv2 client" are described:

Hashes [39] RIOT's internal *hashes* module defines several cryptographic hash functions including SHA1 and SHA256. Both are preconfigured and can be used in the implementation by setting the corresponding flags in the *Makefile*.

Crypto [40] RIOT's crypto module provides an AES-128 and 3DES implementation for the use in several cipher modes, including ECB, CTR and CBC. The "minimal G-IKEv2 client" is configured to use the AES-128 in CBC mode. Counter based modes can not be used in the protocol as sharing the counter would require an additional payload to be handled.

micro-ecc [41] As RIOT's internal modules do not provide any support for the Diffie-Hellman exchange the implementation by default uses the *micro-ecc* library, using RIOT's package interface. It provides a number of elliptic curve Diffie-Hellman groups allowed for the use in IKEv2. The default configuration uses the *secp256r1* group.

In addition to the currently used libraries, a number of other compatible libraries can be used to provide further cryptographic functions, with some adaptions in the code. Notable libraries providing more features include:

tweetnacl [42] This library is available over the package interface and should be considered for providing the *curve25519* [43] Diffie-Hellman group, which is well suited for the use in constraint systems, because it's low processing requirements and short keys.

wolfssl [44] A multi-purpose embedded cryptographic engine that is currently being ported to RIOT OS . It provides several elliptic curve Diffie-Hellman groups, hash functions, encryption functions and hash based random number generators.

7. Evaluation

In order to evaluate the implementation, a testbed, containing different embedded systems with varying levels of constraints, is set up in Section 7.1. The most limiting sizes are measured, including memory requirements of the implementation as well as the time needed to process the key exchange. Finally the findings are evaluated against the requirements found in Chapter 4 The evaluation of the implementation has been published in the form of a research paper. The paper is, in it's full form, appended in Appendix A.

7.1. Test setup

The implementation was tested on three popular hardware platforms with different levels of constraints. Table 7.1 shows a technical overview of the chosen platforms. All platforms are supported by RIOT OS and can be equipped with a compatible network device.

Arduino	Architecture	Clock Speed	Flash Memory	SRAM
UNO	ATmega328	16 MHz	32 KB	2 KB
M0 Pro	ARM Cortex-M0+	48 MHz	256 KB	32 KB
Due	ARM Cortex-M3	84 MHz	512 KB	96 KB

Table 7.1.: Available Arduino boards in testbed

In the used testbed the three Group Members (GMs) need to negotiate group membership for a pre configured group with a Group Controller / Key Server (GCKS). An overview of the setup is presented in Figure 7.1 The performed exchange is explained in detail in chapter 5. The devices are authenticated with the pre-shared secret method, which is pre-configured on all devices. The Arduino boards are equipped with Espressif ESP8266[1] WiFi modules.

The cryptographic cipher suite proposed by the clients consists of the following algorithms provided either by RIOT or by packets available over the package interface (modules are RIOT modules, libraries are external):

- AES128-CBC for encryption (*Crypto* module)
- HMAC-SHA256 for integrity (*Hashes* module)
- HMAC-SHA1 as pseudorandom function (*Hashes* module)
- SECP256R1 Diffie-Hellman group (*micro-ecc* library)

The nonces have a size of 32 Byte and are generated with the *periph_hwrng* module on the Arduino Due, which is equipped with a hardware random number generator (RNG), and using the tinymt32 [45] pseudorandom number generation algorithm on the other two devices.

[1]https://espressif.com/en/products/hardware/esp8266ex/overview

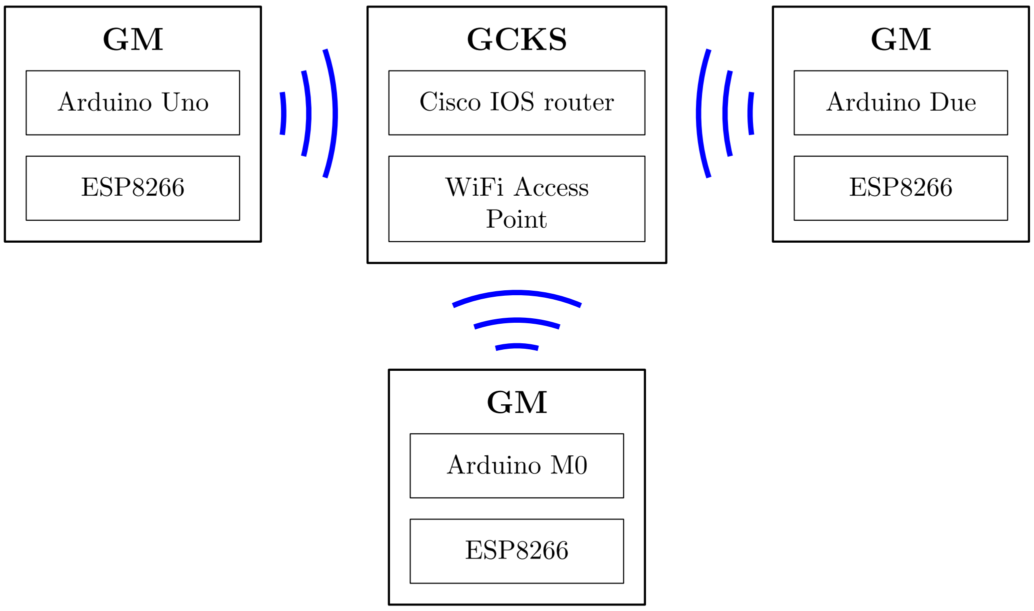


Figure 7.1.: Test setup

7.2. Results

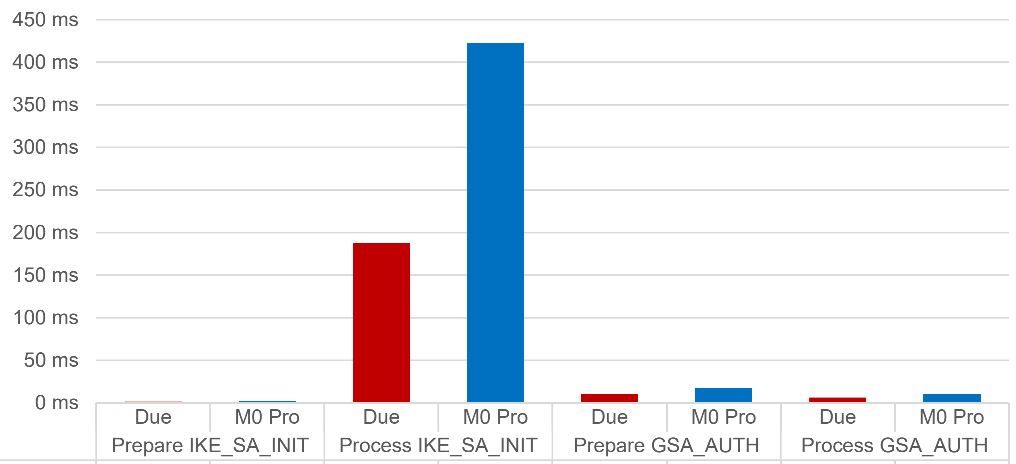
The most relevant results from the tests - in regards to the practicability of secure group key management on the chosen systems - are the memory requirement and the CPU performance for the minimal G-IKEv2 exchange. The use of only static memory in the implementation allows for a very precise analysis of the minimal memory requirement. The implementations performance is measured by the time necessary to process the performed exchanges.

7.2.1. Memory

Table 7.2.: Memory required for the minimal G-IKEv2 client

Feature	**Required Memory**
RIOT kernel (incl. stack)	2,560 Byte
RIOT IPv6 stack	1,024 Byte
RIOT UDP stack	1,024 Byte
RIOT net cache	928 Byte
RIOT packet buffer	1,280 Byte
IKE SA	~ 210 Byte
SAD for 1 group membership	~ 100 Byte
SPD for 1 group membership	40 Byte
$\sum$	6,142 Byte

The total minimum RAM requirement of RIOT, including necessary network modules and the implementation itself, is found to be 6142 Byte. This includes a RIOT (plus G-IKEv2 module) size of 2560 Byte. 1024 Bytes each for the IPv6 and UDP stacks, a packet buffer of 1280 Byte, and 350 Byte in total for the different SADs. A more precise list of the



	Due	M0 Pro	Due	M0 Pro	Due	M0 Pro	Due	M0 Pro
	Prepare IKE_SA_INIT		Process IKE_SA_INIT		Prepare GSA_AUTH		Process GSA_AUTH	
avg [ms]	1.62	2.62	187.92	421.94	10.29	17.41	6.32	10.53
std. dev. [ms]	0.00	0.00	0.11	0.13	0.00	0.00	0.40	0.00
min [ms]	1.62	2.62	187.72	421.71	10.29	17.41	6.15	10.52
max [ms]	1.62	2.62	188.11	422.18	10.29	17.41	7.26	10.53

Figure 7.2.: Processing time for G-IKEv2 packets on Arduino M0 and Due [37]

implementations memory requirements can be found in Table 7.2.

This exceeds the Arduino UNO's RAM capacity by more than double the available RAM and consequently makes it unusable for the further tests using this setup.

The Arduino M0 as well as the Arduino Due provide 32 Byte and 96 Byte RAM respectively and thus have no problems fitting the minimal configuration.

7.2.2. Performance

The performance analysis could only be performed on the two more powerful boards Arduino M0 and Arduino Due. Four different steps in the protocol were measured: 1) prepare the IKE_SA_INIT in the packet buffer , 2) process of the servers response, measured from the moment the incoming packet is received, 3) build the GSA_AUTH request until it is sent and finally 4) process the GSA_AUTH response. Due to the necessity of manual intervention during the tests, which makes an automated test setup impossible, 20 measurements have been made for each device. This is considered an appropriate number as the standard deviation is negligible in all steps. The results of the measurements can be seen in Figure 7.2.

The most time intensive operation of the exchange is the processing of the IKE_SA_INIT request, which contains the calculation of the shared Diffie-Hellman value. This is no surprise as asymmetric cryptographic operations are known to be more CPU expensive than symmetric ones [46]. In total the Diffie-Hellman calculation makes up for almost 99% of the time measured for this operation. The Arduino Due needs about 44.5% the time the M0 needs, which is in accordance with their processor's clock rate difference, shown in table 7.1.

Even though the processing of the IKE_SA_INIT may violate an eventual real-time threshold, it should be noted that this is a one time step during the connection establishment. Further messages will rather compare to the GSA_AUTH exchange which takes $\approx 17\ ms$ on the stronger Arduino Due and barely above $\approx 28\ ms$ on the Arduino Zero. Both boards can thus be considered fulfilling the performance requirement of handling the cryptographic algorithms without limiting the devices overall functionality.

7.3. Summary

Two of the three tested devices fulfilled all requirements collected in Chapter 4. An overview of the tested systems in regards to the requirements is given in Table 7.3. All three devices are theoretically able to run RIOT, which includes a working network stack. RIOT provides a compatible cipher suite, which can be further extended using additional libraries available via RIOT's package interface. The only board failing the memory requirements, for both Flash and RAM, is the Arduino Uno. An even more minimal implementation could outsource the network stack's software requirements entirely to the network module and thus reduce the memory requirements even further. Common embedded WiFi modules, like the ESP8266, theoretically allow this by providing a own IP and UDP implementation. This way even the Arduino Uno might be able to handle the protocol. However, this has not yet been practically proven. The CPU requirement can be fulfilled by both, Due and M0, as only the asymmetric Diffie-Hellman operation, which only takes place once for each group registration, adds a notable time overhead. The Arduino Uno could not be tested in regards to CPU performance. Energy requirements are fulfilled by using the "minimal G-IKEv2 client" protocol which limits network traffic to the minimum by nature.

Table 7.3.: Comparison of the tested devices in regards to the requirements

Functional Requirements	Arduino Uno	Arduino M0	Arduino Due
RIOT OS Support	✔	✔	✔
Network Functionality	ESP8266	ESP8266	ESP8266
Cipher Suite	✔	✔	✔
Encryption	AES128	AES128	AES128
Integrity	HMAC-SHA256	HMAC-SHA256	HMAC-SHA256
PRF	HMAC-SHA1-96	HMAC-SHA1-96	HMAC-SHA1-96
Diffie-Hellman	SECP256R1	SECP256R1	SECP256R1
Source of Randomness	tinymt	tinymt	Hardware-RNG
Non-functional Requirements			
Flash Memory ($\geq 57\ KB$)	32 KB	256 KB	512 KB
RAM ($\geq 6.2\ KB$)	2 KB	32 KB	96 KB
Processor Performance	●	✔	✔
Energy Efficiency	✔[a]	✔[a]	✔[a]
Total	✘	✔	✔

legend: (✔)Full Support, (●)Not tested, (✘)No Support

[a] Fulfilled by using the "minimal G-IKEv2 client" protocol

8. Conclusion

With the growing number of Internet of Things (IoT) devices being connected to the internet, a new kind of networks called wireless sensor networks (WSNs) increasingly gain importance. Secure group communication brings various requirements compared to traditional point-to-point communication.

Various secure group key management solutions have been standardized, however, none of them has been adapted and tested for the use in constrained environments like WSNs.

A list of functional and non-functional requirements for a system implementing the G-IKEv2 protocol was collected, including cryptographic requirements for securing the communication as well as the underlying resource requirements.

The "minimal G-IKEv2 client" protocol was specified by defining a minimal subset of the G-IKEv2 protocol for the use in constrained environments. It is compatible with server implementations compliant to the proposed G-IKEv2 standard and is optimized to support only the minimal number of messages required to join a group and maintain IPsec security associations, in order to save processing power and memory.

The protocol was implemented for RIOT OS, an aspiring IoT operating system. It was deployed and tested on three constrained target systems in order to evaluate the implementation's performance and memory usage in regards to the derived requirements. The tests showed that two out of three tested devices are able to handle the protocol reliably and without noteworthy time delays. Of these, the most drastic delays are caused by the shared Diffie-Hellman value, which only needs to be calculated once, for each established connection. The following exchanges have been performed below a time threshold of 20 ms with a negligible maximum standard deviation of 0.4 ms. The tests prove that the "minimal G-IKEv2 client" protocol can be a viable choice for secure group management in sensor networks.

Future Work While the protocol provides the basic group key management functions necessary for secure communication, it can not be used productively to secure large scale wireless sensor networks, without further additions. One of the biggest limitations of the current implementation is the lack of an efficient authentication method. The pre-shared key authentication used for the tests in chapter 7 is only sufficient for small groups of systems that provide a secure side channel to exchange secrets. This method lacks source authentication in the GSA_REKEY exchange, as all group members share one secret. A secure identity management, as well as an authorization and authentication infrastructure will be necessary.

Another requirement that arises especially with the use in dynamic groups is the need for forward and backward secrecy. The G-IKEv2 protocol supports this by using a logical key hierarchy (LKH) and thereby being able to exclude members from the rekey multicast.

A. Secure Group Key Distribution in Constrained Environments with IKEv2

List of Figures

Bibliography

[1] W. OBILE, "Ericsson mobility report," Nov. 2016.

[2] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, IEEE, 2004, pp. 455–462.

[3] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, *et al.*, "Tinyos: An operating system for sensor networks," in *Ambient intelligence*, Springer, 2005, pp. 115–148.

[4] E. Baccelli, O. Hahm, M. Gunes, M. Wahlisch, and T. C. Schmidt, "Riot os: Towards an os for the internet of things," in *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, IEEE, 2013, pp. 79–80.

[5] *The IEEE802.15.4 website*, 2005. [Online]. Available: `http://standards.ieee.org/about/get/802/802.15.html` (visited on 04/11/2017).

[6] T. Kivinen, "Minimal internet key exchange version 2 (ikev2) initiator implementation," 2016.

[7] D. Migault, T. Guggemos, and C. Bormann, "Diet-ESP: a flexible and compressed format for IPsec/ESP," Internet Engineering Task Force, Internet-Draft draft-mglt-6lo-diet-esp-02, Jul. 2016, Work in Progress, 31 pp. [Online]. Available: `https://datatracker.ietf.org/doc/html/draft-mglt-6lo-diet-esp-02`.

[8] J. Granjal, R. Silva, E. Monteiro, J. S. Silva, and F. Boavida, "Why is ipsec a viable option for wireless sensor networks," in *Mobile Ad Hoc and Sensor Systems, 2008. MASS 2008. 5th IEEE International Conference on*, IEEE, 2008, pp. 802–807.

[9] T. Hardjono and B. Weis, "The multicast group security architecture", rfc 3740," 2004.

[10] S. A. Camtepe and B. Yener, "Key distribution mechanisms for wireless sensor networks: A survey," Tech. Rep., 2005. [Online]. Available: `http://www.cs.rpi.edu/research/pdf/05-07.pdf`.

[11] L. Salgarelli, M. Buddhikot, J. Garay, S. Patel, and S. Miller, "Efficient authentication and key distribution in wireless IP networks," *IEEE WIRELESS COMMUNICATIONS*, vol. 10, no. 6, pp. 52–61, 2003.

[12] B. Aboba, D. Simon, and P. Eronen, *Extensible Authentication Protocol (EAP) Key Management Framework*, RFC 5247, Aug. 2008. DOI: `10.17487/rfc5247`. [Online]. Available: `https://rfc-editor.org/rfc/rfc5247.txt`.

[13] T. Dierks, *The Transport Layer Security (TLS) Protocol Version 1.2*, RFC 5246, Aug. 2008. DOI: `10.17487/rfc5246`. [Online]. Available: `https://rfc-editor.org/rfc/rfc5246.txt`.

[14] S. Rafaeli and D. Hutchison, "A survey of key management for secure group communication," *ACM Comput. Surv.*, vol. 35, no. 3, pp. 309–329, Sep. 2003, ISSN: 0360-0300. DOI: `10.1145/937503.937506`. [Online]. Available: `http://doi.acm.org/10.1145/937503.937506`.

[15] B. Weis, S. Rowles, and T. Hardjono, "The group domain of interpretation," Tech. Rep., 2011.

[16] W. Werner, A. Somaraju, S. S. Kumar, and H. Tschofenig, "Security for Low-Latency Group Communication," Internet Engineering Task Force, Internet-Draft draft-somaraju-ace-multicast-02, Oct. 2016, Work in Progress, 27 pp. [Online]. Available: `https://tools.ietf.org/html/draft-somaraju-ace-multicast-02`.

[17] T. Fossati and H. Tschofenig, *Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things*, RFC 7925, Jul. 2016. DOI: `10.17487/rfc7925`. [Online]. Available: `https://rfc-editor.org/rfc/rfc7925.txt`.

[18] B. Weis, V. Smyslov, and Y. Nir, "Group Key Management using IKEv2," Internet Engineering Task Force, Internet-Draft draft-yeung-g-ikev2-11, Mar. 2017, Work in Progress, 42 pp. [Online]. Available: `https://tools.ietf.org/html/draft-yeung-g-ikev2-11`.

[19] B. Aboba, S. Dawkins, D. Romascanu, and P. Thaler, "The ieee 802/ietf relationship," 2014.

[20] *The IEEE802.11 website*, 2012. [Online]. Available: `http://standards.ieee.org/about/get/802/802.11.html` (visited on 04/11/2017).

[21] *The ZigBee website*, 2005. [Online]. Available: `http://www.zigbee.org/` (visited on 04/11/2017).

[22] *The Z-Wave website*. [Online]. Available: `http://www.z-wave.com` (visited on 04/14/2017).

[23] N. Kushalnagar, G. Montenegro, and C. Schumacher, "Ipv6 over low-power wireless personal area networks (6lowpans): Overview, assumptions, problem statement, and goals," RFC Editor, RFC 4919, Aug. 2007, `http://www.rfc-editor.org/rfc/rfc4919.txt`. [Online]. Available: `http://www.rfc-editor.org/rfc/rfc4919.txt`.

[24] P. Thubert, R. Assimiti, and T. Watteyne, "An architecture for ipv6 over the tsch mode of ieee 802.15. 4," *Draft-ietf-6tisch-architecture-08 (work in progress)*, 2015.

[25] *The IEEE802.15.4e group*, 2012. [Online]. Available: `http://www.ieee802.org/15/pub/TG4e.html` (visited on 04/14/2017).

[26] T. Winter, "Routing protocol for low-power and lossy networks," *RFC 6550, 6551, 6552. IETF, Tech. Rep.*, 2012.

[27] R. Silva, J. S. Silva, M. Simek, and F. Boavida, "Why should multicast be used in wsns," in *Wireless Communication Systems. 2008. ISWCS'08. IEEE International Symposium on*, IEEE, 2008, pp. 598–602.

[28] S. Kent and K. Seo, "Security architecture for the internet protocol," RFC Editor, RFC 4301, Dec. 2005, `http://www.rfc-editor.org/rfc/rfc4301.txt`. [Online]. Available: `http://www.rfc-editor.org/rfc/rfc4301.txt`.

[29] S. Kent, "Ip authentication header," 2005.

[30] S. Kent, "Ip encapsulating security payload (esp)," RFC Editor, RFC 4303, Dec. 2005, `http://www.rfc-editor.org/rfc/rfc4303.txt`. [Online]. Available: `http://www.rfc-editor.org/rfc/rfc4303.txt`.

[31] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen, "Internet key exchange protocol version 2 (ikev2)," RFC Editor, STD 79, Oct. 2014, `http://www.rfc-editor.org/rfc/rfc7296.txt`. [Online]. Available: `http://www.rfc-editor.org/rfc/rfc7296.txt`.

[32] *The IANA IKE site.* [Online]. Available: `http://www.iana.org/assignments/ipsec-registry/ipsec-registry.xhtml` (visited on 04/22/2017).

[33] N. F. Pub, "197: Advanced encryption standard (aes)," *Federal Information Processing Standards Publication*, vol. 197, no. 441, p. 0311, 2001.

[34] D. Fu and J. Solinas, "Elliptic curve groups modulo a prime (ecp groups) for ike and ikev2," 2010.

[35] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full sha-1," 2017.

[36] L. Malina, J. Hajny, R. Fujdiak, and J. Hosek, "On perspective of security and privacy-preserving solutions in the internet of things," *Computer Networks*, vol. 102, pp. 83–95, 2016.

[37] N. gentschen Felde, T. Guggemos, T. Heider, and D. Kranzlmüller, "Secure group key distribution in constrained environments with ikev2," 2017.

[38] M. Lenders, *Analysis and comparison of embedded network stacks*, 2016.

[39] *The riot hashes module.* [Online]. Available: `http://riot-os.org/api/group__sys__hashes.html` (visited on 04/22/2017).

[40] *The riot crypto module.* [Online]. Available: `http://riot-os.org/api/group__sys__crypto.html` (visited on 04/22/2017).

[41] *The micro-ecc website.* [Online]. Available: `http://kmackay.ca/micro-ecc/` (visited on 04/22/2017).

[42] *The tweetnacl website.* [Online]. Available: `https://tweetnacl.cr.yp.to/` (visited on 04/22/2017).

[43] D. J. Bernstein, "Curve25519: New diffie-hellman speed records," in *International Workshop on Public Key Cryptography*, Springer, 2006, pp. 207–228.

[44] *The wolfssl website.* [Online]. Available: `https://www.wolfssl.com/wolfSSL/Home.html` (visited on 04/22/2017).

[45] M. Saito and M. M. Matsumoto, "Tiny mersenne twister (tinymt): A small-sized variant of mersenne twister," *Cited on*, p. 44, 2011.

[46] M. Sethi, J. Arkko, A. Keranen, and H.-M. Back, "Practical considerations and implementation experiences in securing smart object networks," Internet Engineering Task Force, Internet-Draft draft-ietf-lwig-crypto-sensors-02, Feb. 2017, Work in Progress, 32 pp. [Online]. Available: `https://datatracker.ietf.org/doc/html/draft-ietf-lwig-crypto-sensors-02`.