INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Efficiently Re-Keying Multicast Groups with LKH in G-IKEv2

Marinus Enzinger

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Efficiently Re-Keying Multicast Groups with LKH in G-IKEv2

Marinus Enzinger

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Tobias Guggemos
Abgabetermin:	15. November 2019

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 15. November 2019

. . . . . .
(Unterschrift des Kandidaten)

Abstract

The growing field of the Internet of Things (IoT) has many applications which are in the need for secure communication within groups of devices, for example wireless sensor networks. Group re-keying, which means securely providing new keys to the group as a result of members joining or leaving the group, remains one of the main challenges. Several algorithms exist which provide for efficient re-keying with only one or few multicast messages. They rely upon a central instance, called the Group Controller/Key Server (GCKS) to manage the group and the keys associated with the group.

This work implements and evaluates the LKH group key management algorithm as specified for its use within the G-IKEv2 protocol and focuses on constrained clients, as those are mostly used in IoT scenarios. The GCKS part is integrated into Strongswan, an open source, multi-platform IKE daemon. The group member part is implemented for RIOT, an open source operating system for embedded devices supporting multiple execution threads. It is shown that providing keys to constrained devices with low effort while still ensuring security properties such as post compromise security (also known as forward secrecy) and backward secrecy is possible.

In addition to the implementation, a proposal is made to enhance the G-IKEv2 standard in a way that allows the LKH key distribution mechanism to provide updated keys to the group members more efficiently. With the proposal, rekey messages are significantly decreased in size, while maintaining low computational complexity for the group members to process those messages.

Contents

1 Introduction

Communication between devices in networks is typically done by using the unicast communication scheme, where one device transmits messages to exactly one destination device. However, unicast is not always the most reasonable scheme, especially if nodes want to transmit the same data to not only one, but two or more destination nodes. Not only does this impose a burden on the transmitting node as it has to send multiple messages with the same contents but different target addresses, but it also may utilize too much of the limited bandwidth of the network (especially in shared media like wireless or bus networks). Here the multicast communication scheme comes into play. It allows devices in a network to transmit a message to many other devices, a multicast group, at once. To participate in a multicast group, devices have to join it. In the context of IPv4 this is done by using the "Internet Group Management Protocol" (IGMP) or "Multicast Listener Discovery" (MLD) when using IPv6. But all IP-based multicast approaches lack security features such as ensuring confidentiality of the transmitted data. This can be resolved by using a "Group Key Management Protocol" (GKMP)[8] whose main function is to provide the missing security features (confidentiality, integrity, authorization) to the multicast group by providing the group members with cryptographic keying material they can use to communicate securely.

In centralized environments an instance called the "Group Controller/Key Server" (GCKS) is responsible for tasks such as key distribution or authorization of group members.[1] The keying material provided to the group members by the GCKS consists of a "Key Encryption Key" (KEK) which is a symmetric key used to encrypt/decrypt one or more "Traffic Encryption Keys" (TEK), which themselves are used to encrypt the actual data exchanged between the group members. Whenever the TEKs have to be changed, the GCKS sends them via a multicast message to the group members which is protected by the group KEK. Due to the nature of symmetric encryption, the keys have to be the same on all devices to ensure working communication. Consider now a scenario where, due to various reasons, existing group members have to be removed from their groups and should not be able to have access to any further data traffic from the group [2]. It is clear, that somehow the symmetric keys used for group communication have to be replaced on all but the excluded group member(s).

A naive approach would be for the GCKS to randomly generate a new KEK and provide it to the group members over the secure channel which has been established when the GM initially entered the group ("Registration SA"). But that would require the GCKS to not only keep the individual registration SAs open for each group member

[1] In IoT networks the GCKS might also act as the border router for the devices in the network.
[2] This security property is known as "post compromise security" or "forward secrecy".

but also to send a total of n unicast messages with n being the number of remaining group members. This would not only impose significant computational load and high memory usage on the GCKS but also might congest the possibly low-bandwidth network carrier.

A more efficient approach is to use a so-called "KEK management algorithm" which can handle the task of providing new keys to the remaining group members more efficiently and with less transmission overhead and/or a lower amount of messages that need to be sent. They solve the task with the help of auxiliary data structures and by taking advantage of specific cryptographic properties.

To be eligible for the use in the context of a GKMP, a KEK management algorithm needs to fulfill specific security properties:

Backward secrecy Whenever a member joins a group, its knowledge of the current group keys should not allow it to decrypt group data traffic it possibly recorded prior to joining the group.

Post compromise security (forward secrecy) Any member which was removed from a group must not be able to decrypt future group traffic.

Immunity against collusion If more than one possibly malicious group members have been excluded from the group, their combined knowledge of group keys and other parameters must not be sufficient to calculate the new group key(s).

This thesis describes the implementation of an efficient KEK management algorithm, namely LKH (Logical Key Hierarchy) [17], in the context of the group key management protocol G-IKEv2 [19] adapted for the needs of networks consisting of constrained devices. The implementation of the GM side of the KEK management algorithm will extend Tobias Heiders existing implementation of an minimal G-IKEv2 initiator on RIOT OS whereas the GCKS side will be based on Wolfgang Engelbrechts implementation of an G-IKEv2 responder in Strongswan.

Structure of this thesis

Chapter 2 describes the basic concepts of secure group communication as well as group key management in more detail, while chapter 3 describes the conceptual aspects and design decisions of the implementation. In chapter 4, concrete details on the implementation and encountered difficulties are explained, whereas in chapter 5 the result of the implementation is examined in regards to performance and functional criteria. This work is concluded in chapter 6 with a short summary and some outlook to future research.

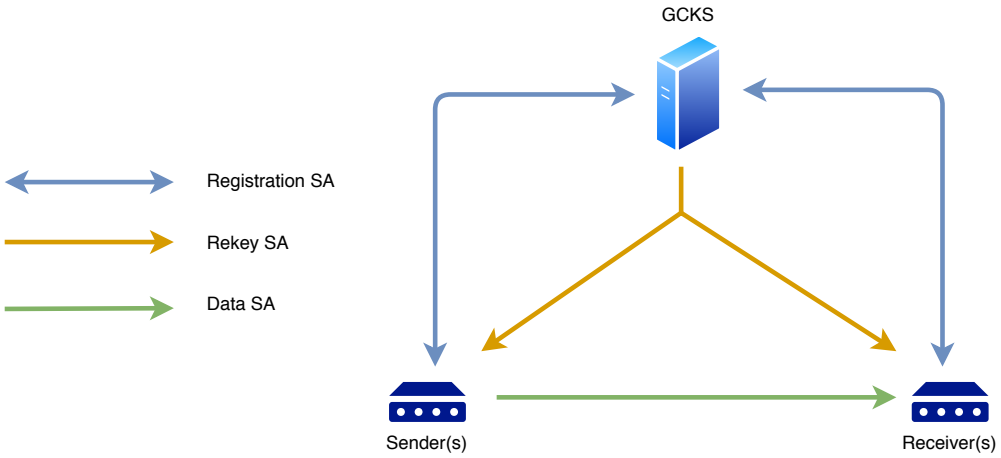

Figure 2.1: Multicast security architecture

2 Background and related work

This chapter provides an overview of the fundamental architecture and principles of modern multicast group communication scenarios. Firstly, detailed information about the different conceptual components is given, whereas afterwards a concrete implementation of the abstract architecture in form of the protocol G-IKEv2 is presented and explained. Then, an efficient group key management algorithm, namely LKH, is presented and compared to other algorithms serving the same purpose.

2.1 Secure multicast communication

The "Multicast Group Security Architecture" describes a general reference framework for secure multicast group communication scenarios in [6]. Essential elements are the Group Controller/Key Server (GCKS) and the Group Members (GMs) which can either be passive (only receive traffic) or active (send and receive). The GCKS is responsible for creating and distributing the keys needed for group communication. The basic architecture features three types of security associations (SAs),[1] which are depicted in figure 2.1 and described in more detail in the following sections.

[1] A SA is a set of parameters exchanged between network entities to allow for secure communication, like negotiated cryptographic algorithms or the actual keys needed for data exchange.

2.1.1 Registration security association

The registration SA is a unicast SA between the GCKS and a GM. It is established by the GM to securely exchange identity and authentication information with the GCKS and register to an existing group. If, during the registration exchange, a rekey SA as described in section 2.1.3 is provided to the GM by the GCKS, the registration SA is usually dropped afterwards. This is possible because future changes of group policies as well as distributing new keying material needed for data SAs and the rekey SA can be done by utilizing the established rekey SA.

2.1.2 Data security association

A data security association (data SA) is a multicast, many to many SA between the senders within a group and its receivers. Typically senders are also receivers and in some scenarios there might even be no receive-only members at all. It it used to protect data between the senders and receivers of a group by using so-called traffic encryption keys (TEKs). There might be more than one data SA in a group, which adds the possibility to separate unrelated traffic or allowing the receivers to use source origin authentication to verify that received data was sent by a selected subset of the group senders or even one specific sender.

2.1.3 Rekey security association

The rekey security association is a multicast, one-to-many SA between the GCKS responsible for a group and its GMs. It is usually created at GCKS startup and provided to new GMs at registration. No negotiation is done as solely the GCKS is entitled to define the properties which make up the rekey SA. The rekey SA serves as a secure channel over which the GCKS can distribute new TEKs to the group members. All data in the rekey SA is encrypted with another key, the group KEK (key encryption key) which is generated by the GCKS. A message sent over the rekey SA is called a rekey message and can also be used to replace the current rekey SA itself by including a new group KEK.

2.1.4 Group security association

The group security association (GSA) contains all policies and keys needed for secure multicast group communication (see [6]). On the one hand it is a superset of a security association as it contains group specific policy attributes such as the rekey scheme that is applied in the event of group members being added or removed from the group. On the other hand it is an aggregation of SAs as it is composed of multiple individual SAs. When combining the three types of SAs described beforehand, they add up to the complete GSA. Even though a rekey SA is not necessary in any scenario and is in fact an optional part of the GSA, as this work focuses on dynamic groups, a rekey SA is considered to be an essential component of a GSA.

Source origin authentication

As every group member knows the group KEK, nothing would prevent possibly hostile group members from crafting their own rekey messages and sending them over the rekey SA. In order to allow the group members to be able to determine whether a rekey message actually originated from the GCKS, rekey messages need to be authenticated using some form of digital signatures. If authentication based on shared secrets were used, the fact that a GM could verify the authentication information included in a rekey message would only prove that the sender is a member of the group [19].

2.2 G-IKEv2

"Group Key Management using IKEv2", abbreviated as G-IKEv2, is a protocol extension for the common IKEv2 protocol which adds support for centralized multicast group communication scenarios as specified in [6]. It is standardized in [19] and has "Internet Draft" status at the time of writing. It is the successor of the GDOI (Group Domain Of Interpretation [18]) protocol which was based based on the outdated IKEv1 [7]. The protocol shares the same initial exchange `IKE_SA_INIT` with the IKEv2 protocol. The IKE SA established by that exchange serves as the registration SA for the GM. G-IKEv2 also supports a way to update data SAs over the registration SA by the `GSA_INBAND_REKEY` exchange. As updating the keys for a group with n members would require n transmission and this document primarily focuses on constrained devices/networks, this option is not described further.

2.2.1 GSA_AUTH

The two-way `GSA_AUTH` exchange follows the `IKE_SA_INIT` exchange and is used to authenticate the previously established IKE SA. Similar to the `IKE_AUTH` exchange used in IKEv2, its contents is encrypted using a key derived from the Diffie Hellman exchange completed by the `IKE_SA_INIT` exchange. The GM acts as initiator and sends its identity as well as its authentication data to the GCKS which in response authenticates itself against the GM and also provides the GM with initial information and keying material associated with the data SA(s) as well as the rekey SA. An exemplary group communication sequence in the context of a small group with only two members is show in figure 2.2. Both members one after another register at the group controller using the `GSA_AUTH` message exchange described in this section. After some time, the group controller sends new keys to the group using the `GSA_REKEY` exchange whose description follows in the next section.

2.2.2 GSA_REKEY

The `GSA_REKEY` exchange is initiated by the GCKS and renews or updates the data SA(s) and/or rekey SA of the communication group. It is sent in a secure way via multicast to the group members by means of the previously established rekey SA. The `GSA_REKEY`

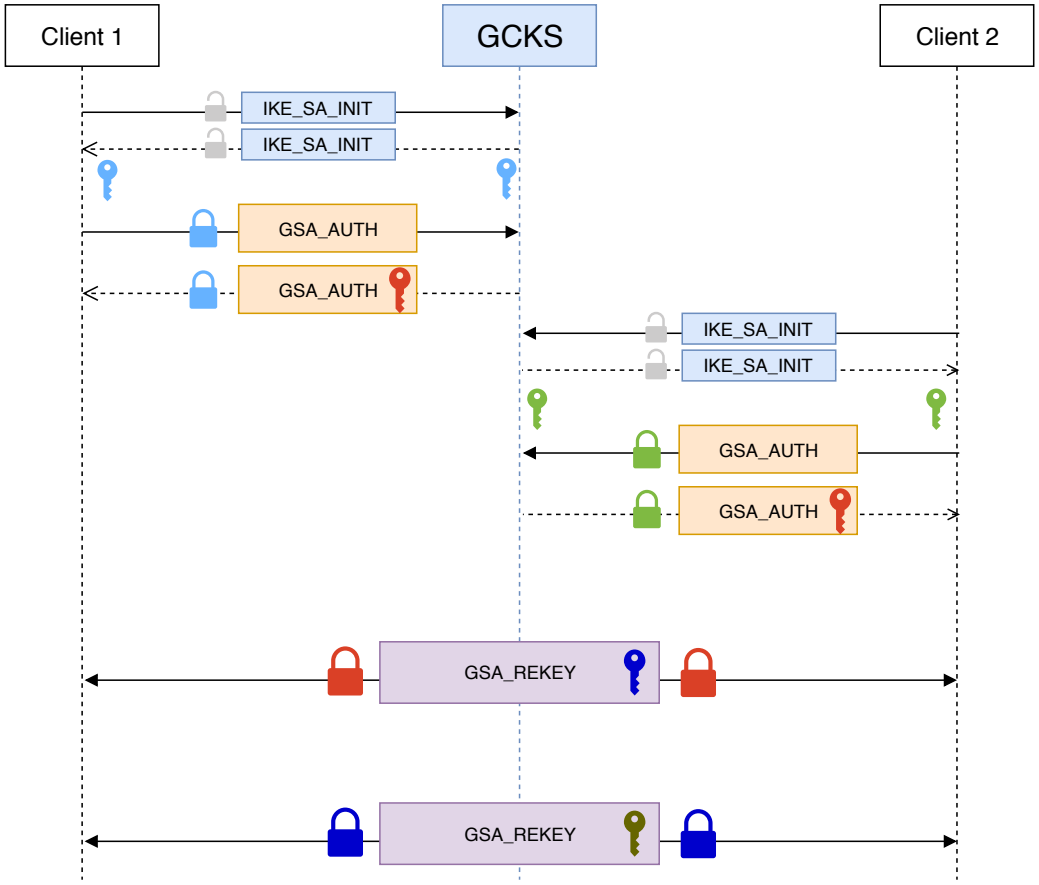

Figure 2.2: G-IKEv2 communication sequence

message is not acknowledged by the group members. Due to the missing acknowledgment it is considered an unreliable exchange in that the GCKS cannot decide whether all group members have received and processed the message. The G-IKEv2 draft therefore suggests that the GCKS should send `GSA_REKEY` messages repeatedly within a small period of time to account for messages not reaching all GMs.

2.2.3 KEK management algorithm

G-IKEv2 defines its KEK management method in a very generic manner and encourages the use of a sophisticated KEK management algorithm. Those can be integrated into G-IKEv2 by specifying payload formats for both initial key distribution used when a group member registers at the GCKS as well as the embedding of rekeying information into a multicast `GSA_REKEY` message. However, the only concrete algorithm that is yet standardized for its use within G-IKEv2 is LKH, which is described in the next section.

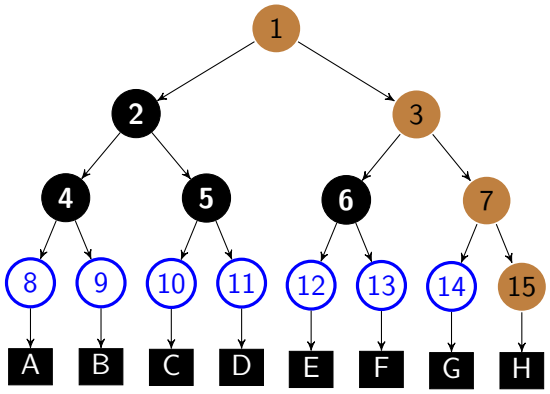

Figure 2.3: Keys provided to H at group join

2.3 Logical key hierarchy

The logical key hierarchy (LKH), described in [17], is a centralized, efficient and robust algorithm for multicast key management. It provides backward and forward secrecy and is secure against collusion of excluded members. For a group G, a binary tree is generated by the GCKS with at least as many leaves as there are possible group members in G. The key associated with the root node is called the group KEK. Each node in the tree is assigned a freshly generated symmetric key, called intermediate KEK.

2.3.1 Member join

When a member m joins G, a leaf node x_m not yet associated with any group member is selected. Every key starting from x_m along the path to the root of the tree (the group KEK) is provided to m. That leads to all members at least having the group KEK in common, which is why it is used to protect the rekey SA of the group. Figure 2.3 shows an example LKH tree with leaf nodes 8 to 15, which are associated with group members A to H. Marked in brown are those keys which the GCKS provides to member H when it registers at the group.

2.3.2 Member leave

When a group member m leaves G or is excluded from the group, the GCKS replaces every key that m possesses (from x_m until the group KEK) with a newly generated key. To allow the remaining group members to get access to the replaced keys, a multicast rekey message, encrypted with the old group KEK, is sent to the group. The message contains a list of replaced keys, starting from the parent of x_m until the group KEK. Each replaced key is contained twice, once encrypted with the corresponding left child key and once encrypted with its right child key. Thereby, each of the remaining group members is able to replace their affected keys including the group KEK. Figure 2.4 shows in light blue the new keys 1', 3', 6' and 13' which had to be generated in order to replace

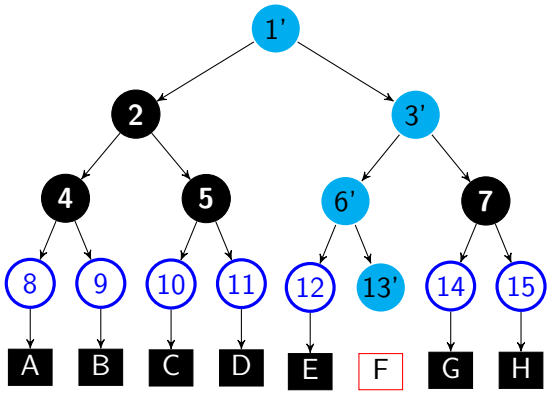

Figure 2.4: LKH key tree after exclusion of F

the existing keys when member F left the group.

2.3.3 Security properties

LKH fulfills a set of security properties which are characterized in this section. The list of properties is not exhaustive, but describes fundamental features of a secure architecture.

Backward secrecy

Backward secrecy with LKH is accomplished by replacing the group KEK before adding a group member. That is, before a member wants to join, one rekey message is sent to the group replacing the group KEK. Afterwards, the GCKS adds the member by using the registration protocol and provides it with the new group KEK.

Post compromise security

The way LKH rekeys the remaining group members with a single rekey message when a member is excluded implicitly provides forward secrecy to the group. This is because the excluded group member, although being able to decrypt the rekey message itself, cannot replace/decrypt any of the contained keys because none of them is sent encrypted with any key that the group member possesses. It is important that the encrypted rekey message may not include any unencrypted new TEK(s). The GM would otherwise be able to use these to decrypt future data traffic. The new TEK(s) might for example be sent to the group in a rekey message immediately following the one that replaces the group KEK, while being encrypted with the new group KEK.

Protection against collusion

LKH provides protection against collusion of an arbitrary amount of excluded hostile group members. After the group is rekeyed, neither of the group members possesses a key that could be used to decrypt the new group KEK. Also, as the keys are unrelated,

the combined knowledge of all their keys does not disclose any information about the new group KEK.

2.4 Related Work

Other than LKH, there exist many more group key mangagement algorithms for centralized environments. All of them have individual properties, so depending on the scenario and its particular requirements, a different algorithm may be best suited.

2.4.1 OFT

Like LKH, the oneway function tree (OFT) is a centralized multicast group key management protocol originally proposed in [15]. It is also based on hierarchical key tree and has the same storage requirements as LKH on both GCKS and group member. However, it is able to reduce the number of keys needed to be sent in rekey messages from $2\log_2 n$ to $\log_2 n + 1$. This is achieved by computing the keys associated to the tree nodes from their children's keys instead of just attributing randomly generated keys to the nodes.

Notation

In the following, let $h(x)$ be a one-way function and $f(x, y)$ a mixing function. The key associated with node x is denoted as k_x. The left child node of x is denoted as $left(x)$ and the right child note is denoted as $right(x)$. If a symmetric key i is encrypted with another key j, this is denoted as $encr(i, j)$.

Construction

In a binary tree, for a node x, the set of nodes

$$A(x) := \{x, p_1, p_2, \ldots, p_n | parent(x) = p_1 \wedge parent(p_i) = p_{i+1}\ \forall i \in 1 \ldots n\}$$

is called the ancestor set of x. The set of nodes

$$S(x) := \{p | p \notin A(x) \wedge (\exists i \in A(x) \mid parent(i) = parent(p))\}$$

is called the sibling set of x and resembles the siblings of x or one of its ancestors within the tree. For any node x, its key is calculated as

$$k_x = f(h(k_{left(x)}), h(k_{right(x)}))$$

The application of the oneway function h on the key of node x, which is $h(k_x)$, is called the "blinded" key of node x.

Member join

When a member joins, similarly to LKH it is assigned to a leaf node x not yet associated with another group member. In contrast to LKH, the keys of all nodes in the sibling set of x are sent blinded to the group member, that is, in addition to its own individual leaf key, the group member receives

$$\{h(k_i) \mid i \in A(x)\}$$

As per the definition of how the key of the parent node is constructed, the group member is able to compute all keys for all nodes in its ancestor set.

Member eviction

When a member associated with node x leaves the group or is evicted from it, the GCKS creates a new key for x and recomputes all keys until the root (group KEK). Then, in the rekey message a list of encrypted keys is sent to the remaining group members.

$$\{\ encr(h(k_j), k_i) \mid i \in A(x)\ \wedge\ j \in S(x)\ \wedge\ parent(i) = parent(j)\ \}$$

That is, the blinded key of any recomputed node is sent encrypted with the key of its respective sibling. By possessing one of those keys used to encrypt its blinded sibling key, any remaining group member can decrypt the blinded sibling key and thus compute all needed parent keys until the group KEK.
OFT is however not yet standardized for its use within G-IKEv2 and thus not further discussed in this document. Also, it has been found that OFT in its original form is susceptible against various collusion attacks. Jing and Bo proposed an altered version, called HOFT, which has the same communication overhead as the original form of OFT [12].

2.4.2 Secure Lock

The Secure Lock method, originally presented in [4], uses properties of the Chinese Remainder Theorem (CRT) to efficiently deliver encrypted messages to subsets of group members. This specifically allows for efficient handling of mass-entry or mass-exclusion of multiple members at once.

Construction

Let G denote a group with n members. The GCKS generates symmetric cryptographic keys $\{k_1, \ldots, k_n\}$ as well as positive, pairwise relatively prime integers $\{N_1, \ldots, N_n\}$. At group registration of member i, the GCKS securely provides the tuple (k_i, N_i) to the group member. When the GCKS wants to send a message M to a nonempty subgroup of G, with q members $\{m_1, \ldots, m_q\}$, in a way such that only the members of the subgroup can decrypt M, it constructs a message as follows:

- It generates a random session key d and encrypts M with d.

- It generates the Secure Lock X as specified in [4] and uses it to encrypt d, prepends the result of this to the generated ciphertext of M.

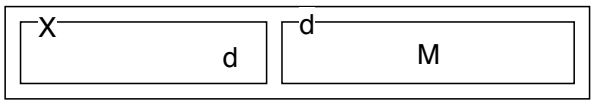

Figure 2.5: Secure Lock message construction

For X, the following conditions hold (E denotes the encryption operation):

$$X \equiv E_{k_1}(d) \ mod \ N_1$$
$$\vdots$$
$$X \equiv E_{k_q}(d) \ mod \ N_q$$

The group members in Q are then able to retrieve the value $E_{k_i}(d)$ by calculating $X \ mod \ N_i$ and as they are in possession of k_i, they can decrypt the session key d and therefore also the message M. In the case of mass-exclusion of members, the GCKS simply generates a new group KEK and distributes it to the remaining group members using the method described just now. A serious disadvantage of the Secure Lock method is the high computational effort needed to compute the CRT based congruence system, especially if group exclusion operations have to be performed frequently.

2.4.3 CAKE

The "Central Authorized Key Extension" (CAKE), originally presented in [11], is a hybrid group key management protocol which combines the hierarchical approach of LKH and the Secure Lock technique. Comparably to LKH, a hierarchical tree structure (the original authors propose a ternary tree structure) is built. However, opposing to LKH each node is associated with not only a single cryptographic key, but a tuple (k_i, N_i) as in the Secure Lock method. For every node, its immediate children form a Secure Lock subdomain, that is, when one of the child keys need to be replaced, it can be sent to its remaining siblings in the same way as the original Secure Lock system provided a new group KEK to remaining group members. The advantage is that the calculation complexity reduces significantly, as the number of nodes included in the CRT calculations compared to the initial Secure Lock method is significantly smaller. But as with LKH, when a member is excluded from the group, still the keys for all tree nodes along to the root key have to be renewed.

Figure 2.6 depicts the ecxlusion of member $mD32$. All keys along the path to the root (dark nodes) have to be replaced and are not included in the CRT calculation. For each level of the tree, only the remaining siblings of the excluded node (shaded nodes) are included in the CRT calculation. This way, the networking overhead of LKH is reduced and the computational effort of the original Secure Lock system is reduced as well. In

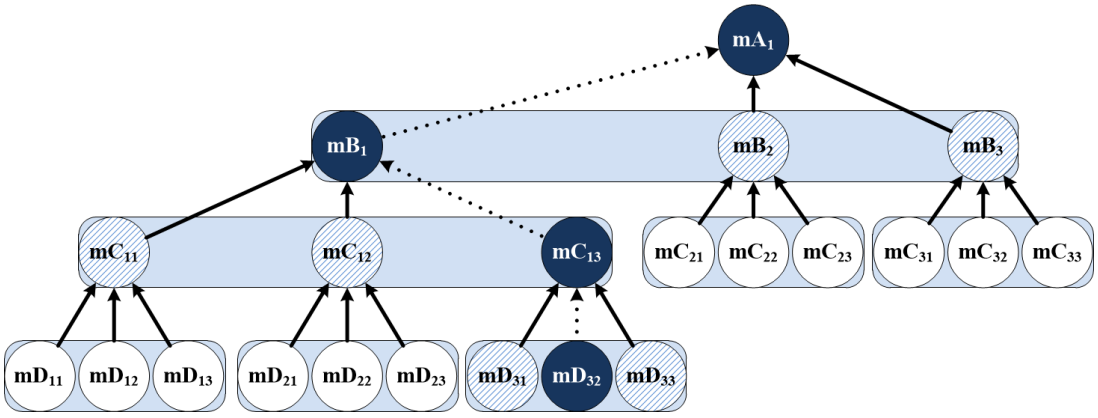

Figure 2.6: From [11]: Ternary tree structure to manage the keys and to reduce the calculation effort by withdrawal

[5], an academic proposal for the integration of CAKE as a group key management algorithm into the G-IKEv2 protocol was made.

3 Concept

In this chapter, firstly the requirements an implementation of the key management algorithm LKH within the G-IKEv2 protocol must fulfill are described. Based on those requirements, the basic architecture and the structure of the implementation is explained. In addition, the principles upon which the Strongswan and RIOT projects are designed and built, are described. Their influence on the planned implementations is discussed and it is shown how those principles are considered during the integration of the needed features into the existing projects.

3.1 Requirements analysis

An implementation of rekeying functionality for RIOT OS and Strongswan needs to fulfill several requirements in order to provide needed functionality and be able to be extended further in the future. Those requirements can be split into functional requirements on the one hand and non-functional requirements on the other hand.

3.1.1 Functional requirements

Functional requirements declare features and properties which must be provided by the implementations in order to operate correctly. The following functional requirements are defined:

Protocol conformity The parts of the implementation which are responsible for generating and parsing the payloads as well as sending and receiving rekey-related group messages should conform to the G-IKEv2 specification. This is, the defined payload formats should be strictly followed and also the semantics of the respective fields should be honored and handled correctly.

Multicast messaging on IPv6 In order to be able to send and receive messages over the rekey SA, network-level multicast messaging with IPv6 need to be used on both the GCKS part (Strongswan) and the GM part (RIOT).

LKH key tree generation and management The GCKS needs to generate a LKH key tree with fresh, independent symmetric keys at group initialization. Whenever a member is excluded from the group, specific keys must be replaced.

LKH key array storage and management The GM needs a facility to store an array of symmetric cryptographic keys and functionality for updating/replacing existing keys by decrypting other keys must be available.

Replay protection Possible replay attacks by multiply sending previously recorded rekey messages need to be prevented.

Member eviction user interface It should be possible for the operator of a GCKS to interactively or programmatically evict members from their groups. Thus, an user interface providing this exact functionality should be implemented.

3.1.2 Non-functional requirements

Non-functional requirements define properties in regards to code quality, resource consumption, memory efficiency and scalability which must be fulfilled. The following non-functional requirements are defined:

Efficient GM RAM usage As the primarily targeted platforms of RIOT OS are constrained microcontroller units, RAM usage by the G-IKEv2 and LKH implementation should be as low as possible.

Low computation effort The constrained computing capacity of microcontroller units needs an efficient implementation so that the responsiveness of other running applications is not affected.

Scalability Multiple group memberships for a single device should be possible. Also if needed large group sizes should be supported.

Extensibility Both implementations should fulfill code quality metrics in terms of modularity, abstraction and encapsulation of modules. This is needed if later support for protocol extensions or different group key management algorithms should be added.

3.2 Architecture

In a centralized group communications scheme utilizing LKH as its rekeying mechanism, the GCKS has the responsibility of creating and maintaining the LKH key tree. Upon group initialization at the GCKS, a fresh key tree has to be generated from secure random data, where the root key represents the first group KEK. Whenever a group member enters the group, an unused leaf key needs to be associated with the new group member. The key chain up to the root key has to be provided to the group member within the `GSA_AUTH` response. Whenever a group member is removed from the group, its leaf key and all the keys on the path along to the root key need to be replaced with fresh keys.

The group member implementation, on the other hand, needs a way to store the chain of keys it received and functions to replace an arbitrary subset of its stored keys with new keys, which also need to be decrypted using preceding keys.

3.3 Design decisions

The GCKS part holding the LKH tree logic and sending multicast rekey messages to the group members is integrated into the Strongswan project [16]. Strongswan is a multi platform open source IKE daemon written in the C programming language and encouraging a highly object-oriented programming paradigm. The GM part responsible for receiving and processing the rekey messages is integrated into RIOT OS [1], which is a modern open source operating system for low-end microcontrollers written also in the C programming language. RIOT OS and Strongswan were chosen because existing implementations of the basic G-IKEv2 registration procedure were available for both: Wolfgang Engelbrechts G-IKEv2 GCKS implementation for Strongswan [2] on the one hand and Tobias Heiders G-IKEv2 group member implementation [10] on the other hand.

3.4 RIOT

RIOT is an operating system mainly targeting low-end microcontrollers and specifically designed for the use in IoT and sensor devices with low energy consumption. It can be programmed using the C or C++ languages and contains an efficient, full-fledged network stack called GNRC, which is further described in [14]. Out of the box, modules for various cryptographic applications are provided as well as a sophisticated interprocess communcation (IPC) framework. It runs on a multitude of different hardware architectures and provides drivers for lots of different sensors, network communication chips and other hardware.

3.4.1 Memory management

As RIOT primarily targets microcontrollers with very limited resources in terms of RAM, ROM and computing speed, efficient memory management is a key factor to consider. In particular, memory usage by the G-IKEv2 and LKH implementation should be:

Predictable No unnecessary dynamic memory allocations should happen at program runtime, as the risk of program crashes due to running out of memory increases.

Efficient As the general amount of available memory on microcontrollers is very low (i. e. in the range of multiple KiB), the implementation should be as efficient as possible.

Typically, microcontrollers or embedded systems are used in more static, long-term applications. Dynamic behavior due to user interaction, for example, is often not required. Consider the use case of group communication between constrained devices in infrastructure components of a smart city. In this setup the needed network configuration and authentication information including pre-shared keys or certificates would likely be deployed on the nodes before they are brought to their destination point of operation. Therefore it is sufficient for the RIOT implementation to support only a static

number of security associations defined at compile time. However, the number of supported security associations should be adjustable, because in some scenarios there might be a need for multiple group memberships.

GNRC Packet Buffer

The RIOT GNRC network stack relies upon the so called packet buffer to store received packets and ones which are about to be sent. Its default implementation consists of a large block of statically allocated memory where the actual payloads and also the control structures are stored and an API to add, manipulate and remove packets in the packet buffer is provided. A network packet is represented as a linked list of data chunks where each chunk normally represents the payload of a specific layer in the OSI model. This way, each layer in the protocol stack can easily handle only the header or payload chunk it is responsible for. Figure 3.1 shows how in the packet buffer the `pktsnip_t` structures are interleaved with the actual data chunks. As an example, an IPv6 packet containing a UDP payload is shown. Marked in green is the data chunk containing the actual user data. A packet snip structure of undefined data type references this data chunk. It also references the next packet snip structure of type UDP which is linked with the UDP header data (drawn in red) as well as the packet snip responsible for the IPv6 header (green). The placement of packet snip structures as well as data chunks within the GNRC packet buffer is not predictable. The packet buffer can also be used as a generic pool from which dynamically sized chunks of memory can be allocated and used in user code. However this should be used sparingly, as by misusing the packet buffer as a generic memory allocation mechanism, space actually needed for the operation of the network stack may be occupied and this can result in dropped or missed network packets.

3.5 Strongswan

Strongswan is implemented in the C programming language, however as opposed to other software projects written in C, it uses a strongly object oriented approach. With the help of preprocessor macros, the concept of ”classes” and ”interfaces” can be employed. Object methods are represented by structure members of function-pointer type. This concept should be respected and followed by any new functionality which has to be added to Strongswan in order to realize group rekeying and LKH functionality.

3.5.1 Memory management

The memory management in Strongswan is completely realized by dynamically allocated objects. Each class provides one or more factory functions which can be used to allocate storage for the requested object and initialize it adequately. As there is no garbage collection, to prevent memory leaks it has to be ensured that the memory allocated by every object is released at some point in time. This is done by using the concept of ownership, which means that some functions which take the ownership of objects given

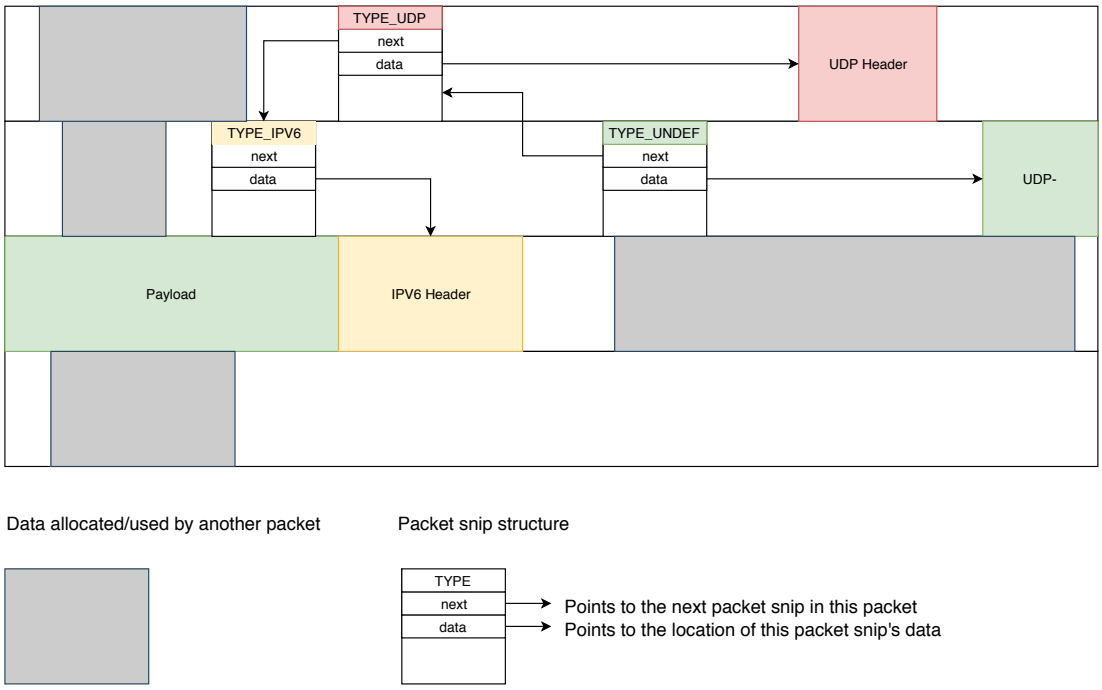

Figure 3.1: Layout of a IPv6 packet in the GNRC packet buffer

as their parameters guarantee that they will release the memory used by the provided objects. Another option is the ownership of objects by composite data structures. For example a linked list structure might be responsible for releasing its contained objects when it is destroyed/released itself. In the context of payload processing, this means that a message object is the owner of its contained payloads and will release them when the message itself is released, for example when the message has successfully been sent over the network. If this concept is applied to the tree structure of the LKH, it becomes clear that a parent node must be the owner of its child nodes. Similarly, a specific node will be the owner of its associated key.

3.5.2 Member eviction

When a member leaves the group or is excluded from it to ensure post compromise security two consecutive `GSA_REKEY` messages need to be sent. The first one renews the group KEK and the second one renews the TEK(s). This is necessary because the data within a `GSA_REKEY` message is encrypted using the current KEK. If the new TEK(s) were sent in the first `GSA_REKEY` message, the excluded GM could use the current group KEK, which it possesses, to decrypt the new TEKs. Also the required user interface which allows the operator of a GCKS to evict group members from their groups needs some consideration. To remove a group member by user interaction, which corresponds to handling externally triggered events, a communication option with the internal state of the IKE daemon is required. As there are already several command line tools available

for Strongswan, which allow for creation, manipulation or deletion of IKE SAs, member eviction should be implemented by extending those tools by the needed functionality.

3.6 Improve rekeying efficiency

The current version of the G-IKEv2 draft only one method of how the LKH algorithm can distribute updated keys to the GMs is standardized. This method simplifies parsing and processing the `GSA_REKEY` messages for the GMs but lacks efficiency as the message sizes get unnecessarily big. Suppose a dynamic multicast group is to be formed which consists of n members. The depth d of a LKH key tree sufficient to serve the group is calculated as

$$d = \lceil \log_2 n \rceil$$

Suppose now, a group member k is to be excluded from the group, with k denoting that the GMs leaf key is the kth leaf key in the tree. The GCKS will now send a `GSA_REKEY` message to the group containing LKH_UPDATE_ARRAY attributes. The first one of these contains d encrypted keys, starting with the replaced parent key of k up until the new root key of the tree, which will become the new group KEK. The second attribute contains $d-1$ keys and so on until the last attribute, which only contains one encrypted key, the new root key of the tree. Thus the total number of keys sent in one `GSA_REKEY` message can be calculated as

$$n_{keys} = \sum_{i=1}^{d} i = \frac{(d+1)d}{2} \quad \in \mathcal{O}(d^2) = \mathcal{O}((\log_2 n)^2)$$

This shows that the message size, which clearly increases linearly with the number of keys contained, grows quadratically in the depth of the LKH tree. An alternative approach which would heavily decrease message size is described as follows: Each key which is to be replaced is sent only twice, firstly encrypted with its left child key and secondly encrypted with its right child key. Replaced key(s) residing deeper in the tree hierarchy have to be ordered before keys which are closer to the root key in the `GSA_REKEY` message. This way each GM will be able to decrypt all needed keys on the path along to the root key, as the child keys needed for decryption themselves already were successfully decrypted. When excluding a GM using this alternative approach, each of the n keys from the parent key of the excluded member along to the root key only has to be sent twice, except for the lowest key which is only sent once, encrypted with the individual key of the sibling of the excluded GM. Thus the total number of keys calculates as

$$n_{keys} = 2d - 1 \quad \in \mathcal{O}(d) = \mathcal{O}(\log_2 n)$$

This is a significant improvement compared to the initial, standardized method. Table 3.1 shows the difference. 16 Byte Keys with 16 bytes of Initialization Vector (as the key itself is encrypted) and 8 Bytes of metadata are assumed, which makes each key payload a total of 40 Bytes. Already at a moderate group size of 256 members, the combined

Table 3.1: LKH rekey comparison

Tree depth	Max group size	Standardized method		Proposed method	
		n_{keys}	$s_{payload}$ (B)	n_{keys}	$s_{payload}$ (B)
3	8	6	240	5	200
5	32	15	600	9	360
8	256	36	1440	15	600
10	1024	55	2200	19	760
15	32768	120	4800	29	1160

payload size of the keys nearly exceeds the MTU of a normal Ethernet frame, let alone that constrained devices usually use network techniques with much smaller MTUs. In the proposed method even rekeying large groups with e.g. more than 30,000 members is possible without exceeding a typical Ethernet MTU.

4 Implementation

This section describes the implementation details of the integration of rekeying functionality by using LKH within the G-IKEv2 protocol into the IKE daemon Strongswan and into RIOT OS. Strongswan is used to represent the GCKS part which creates and manages the LKH key tree whereas the RIOT implementation contains the client (group member) part. The interface between the base LKH algorithm and the G-IKEv2 protocol is implemented as specified in the version 16 of the G-IKEv2 draft. As multiple updated revisions of the G-IKEv2 draft were published after their work, some parts of those implementations had to be altered or extended to be compatible to the newest revision as of the time writing this document.

4.1 Strongswan

The implementation builds on the basic G-IKEv2 implementation of Wolfgang Engelbrecht [2], which focused on the G-IKEv2 registration protocol and did not include any functionality related to G-IKEv2 rekeying of group members. As the rekey SA is generated by the GCKS at initialization, there is no negotiation of any parameters done with the group members. Referring to the Strongswan implementation, this means that all transforms related to the rekey SA are chosen from the first IKE-proposal in the group configuration structure. Although it might theoretically be possible for the GCKS to change security related parameters or policies for the rekey SA at any point in time, this is not implemented in the current version.

4.1.1 LKH

A new module is created for the LKH related functionality in Strongswan. This includes tree as well as key generation and operations which operate on an established LKH key tree like replacing a key at a specific position. It is designed to be completely independent of the G-IKEv2 protocol and could thus be used in any other context. The new module mainly consists of three classes:

lkh_tree representing a complete LKH key tree

lkh_node representing a single node within a LKH tree

lkh_key representing a symmetric key associated with a LKH node

The class diagram in figure 4.1 shows the positioning of the new LKH module within the GCKS architecture. The LKH module contains some more classes, which are not

displayed in the class diagram to simplify matters. For instance, a `lkh_util` class is implemented which aggregates functions to create random numbers (used for key generation) as well as key encryption (used for tree update functionality).

Key tree generation

An LKH key tree is generated as follows: let n be the number of group members for a specific group. A binary tree structure is generated with its depth d calculated as

$$d = \lceil \log_2 n \rceil$$

Now a chunk of random binary data representing a key for a symmetric encryption algorithm is generated for and assigned to each node in the tree. The encryption algorithm for which the keys are to be used and with it the size of the generated key is passed to the LKH module as a parameter. In the current version, the number of configured members in the Strongswan configuration determines the maximum size of the group. Thus, by generating the tree structure and the node keys at GCKS initialization, a perfectly balanced tree is constructed and the tree won't have to be rebalanced at any time, because its capacity is sufficient for all possible group members. However, when only a small amount of possible group members actually joins the group, this is a slightly inefficient solution as more keys than necessary would have to be transmitted to and stored by the group members. However, as the maximum group sizes are configured statically, a fixed size LKH tree is seen as an appropriate solution and no tree rebalancing mechanism is yet implemented.

4.1.2 Rekey SA

To add rekeying functionality to the existing GCKS implementation, a new class called `kek_policy` is introduced. It is associated with a specific group and is responsible for holding and maintaining the data needed for building `GSA_REKEY` messages. It keeps track of the `kek_message_id` counter which is incremented in every `GSA_REKEY` message and provides for protection against replay attacks. The class also maintains the integrity key used for signing the messages and provides functions to add or remove members from its group. For that it holds a reference to an interface type, `kek_mgmt_alg`, which represents a generic abstraction for all KEK management algorithms. Listing 4.1 shows an excerpt from the interface definition. When members are added to or removed from a group, the corresponding methods in `kek_policy` call their counterpart in `kek_mgmt_alg` and let the underlying implementation decide how to handle the request and in which way a new KEK is possibly created. At the moment only one implementation of this interface exists, `lkh_kek_mgmt_alg`. This class encapsulates a LKH tree and figure 4.1 shows a class diagram of the part of the GCKS implementation in Strongswan which is relevant for the rekeying functionality. The part framed in blue is result of the existing work of [2]. The part in the middle shows the structure of the rekey SA related module, as it has just been described, while visualizing its interconnections to both the existing group management module as well as the LKH module.

```
struct kek_mgmt_alg_t {
        /**
         * Get the current KEK. The caller receives a copy of the key
         */
        chunk_t (*get_kek)(kek_mgmt_alg_t *this);

        /**
         * Add a group member
         */
        kd_substructure_t *(*add_member)(kek_mgmt_alg_t *this, uint32_t
            member_id, uint64_t spi_i, uint64_t spi_r);

        /**
         * Remove a group member
         */
        bool (*remove_member)(kek_mgmt_alg_t *this, kd_substructure_t **,
            uint32_t member_id, uint64_t spi_i, uint64_t spi_r);

        /**
         * Get the KEK management algorithm type
         */
        gikev2_kek_management_type (*get_algorithm_type)();

        /**
         * Renews the current KEK without changing group membership
         */
        bool (*renew_kek)(kek_mgmt_alg_t *this, kd_substructure_t **,
            uint64_t spi_i, uint64_t spi_r);
};
```

Listing 4.1: KEK management algorithm interface

4.1.3 Initialization

Upon initialization of the Strongswan IKE daemon, the group manager, group and group member structures are built and initialized as described in [2]. Additionally, the KEK policy for each group is initialized, which in turn initializes the according KEK management algorithm. The current version assigns the LKH KEK management algorithm to each group. The KEK policy generates an KEK integrity key with respect to the chosen integrity algorithm.

4.1.4 Group security association creation

When a group member successfully authenticated itself at the GCKS and is added to the group, the `GSA_AUTH` response contains policies and keys needed by the group member in order to build up the data security SAs and the rekey SA. From the GM's point of view, its registration SA is completed into a group security association (GSA, see 2.1.4). That is, in addition to the existing registration SA, also the data SA and rekey SA get

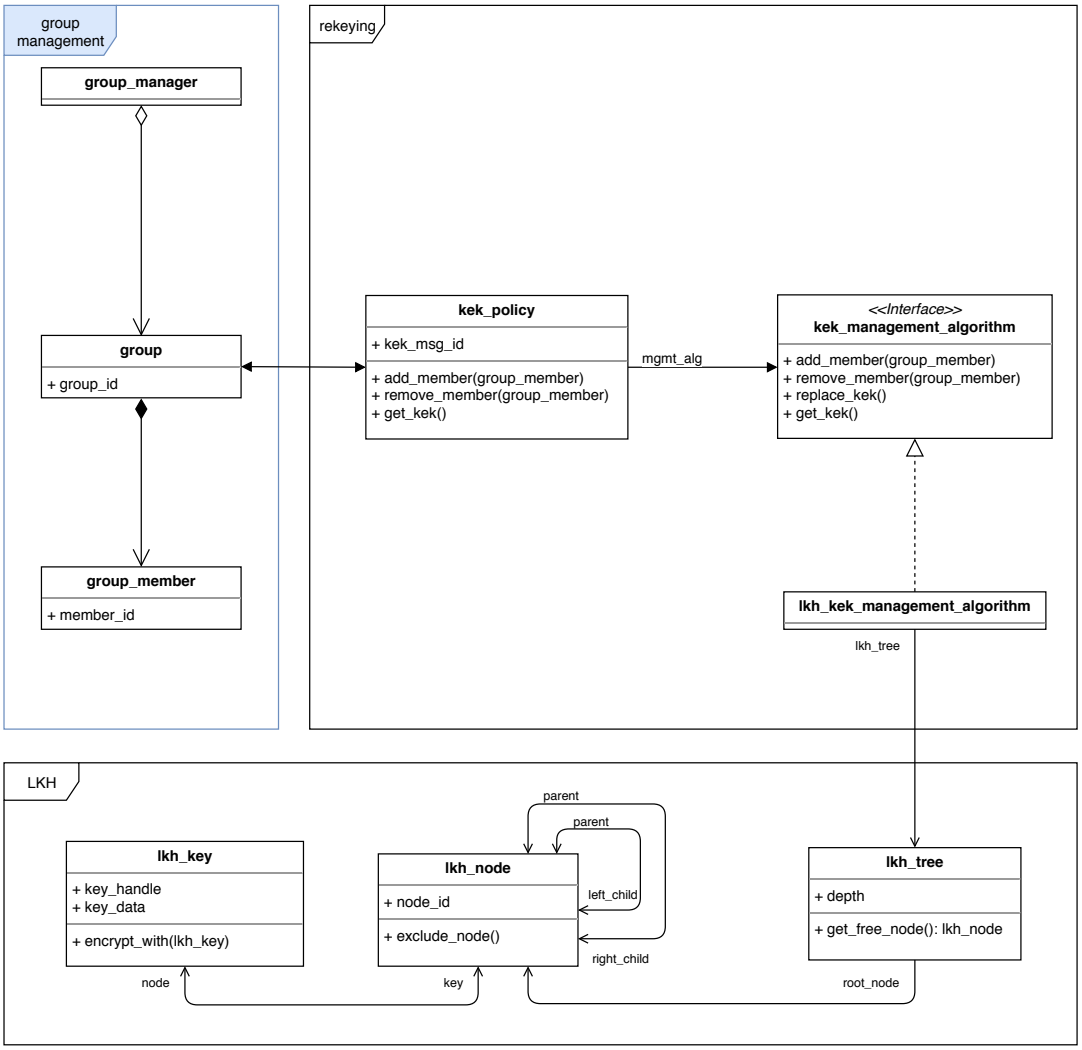

Figure 4.1: Strongswan GCKS architecture

established. From the GM's point of view, this completes its GSA and makes it able to securely communicate with the group. As this procedure differs considerably from the semantics of the `IKE_AUTH` response. Both responses authenticate the previously generated IKE SA but while `GSA_AUTH` as described completes the GSA, `IKE_AUTH` is used to also create a so called child SA, a unicast SA between the initiator and responder. Due to these notable differences, a new task is created in Strongswan, called `GSA_CREATE`. This task and the existing `CHILD_CREATE` task normally responsible for creating the first IPSec child SA (see [13] for more information) are mutually exclusive.

4.1.5 Payload generation

The G-IKEv2 draft extensively makes use of the concept of IKEv2 transform attributes, specified in [13], section 3.3.5. A transform attribute essentially specifies how a key-value pair is serialized into IKEv2 payloads. The key has a fixed length of 15 bits and the value can have any length, but if the size of the value is at most 2 bytes, an improved and more efficient serialization mechanism is used. IKEv2 specifies the use of transform attributes only in one context, namely to add additional attributes to transform substructures such as the definition of which algorithms to use within an IKE or IPSec SA. However, in G-IKEv2 transform attributes are used in various contexts and are often used to encapsulate complete payload substructures. In Strongswan, different payloads are implemented as classes which all inherit from the generic `payload` type. Each payload subtype contains a set of rules defining how its contained attributes and data should be serialized or parsed. Those rules might even contain rules which specify that some of their contained attributes should be serialized or parsed as nested payloads of a given type. The `generator` class in Strongswan is used to serialize payload structures to be able to send a message to the remote peer. The generator is given the outermost payload object and it then serializes data based on different sets of rules which are defined for each payload type. In the existing generator implementation, whenever a transform attribute is encountered, it is only able to serialize those by serializing the transform attribute header and then the value as a chunk of bytes. To be able to serialize transform attributes containing nested payload substructures, another option has been added. A flag in the transform attribute payload type determines whether its value is a nested payload substructure. The generator checks this flag and if it is set, it proceeds to serialize the nested payload by processing its payload rules. Figure 4.2 shows the adapted control flow in the `generate_payload` function. The parts drawn in green had to be added to support nested payload substructures.

4.1.6 Multicast address handling

For testing purposes the IPv6 multicast address ff02::1 which represents all nodes on a link, was used. With the standard configuration of Strongswan, it was not possible to send rekey messages from the GCKS to the group members using ff02::1 as source address. The reason for this is that the socket-default plugin used by Strongswan does not explicitly set the outbound interface which should be used for sending packets, before passing the arguments to the Linux system function `sendmsg()`. This led to `sendmsg()` fail with `EINVAL` which stands for invalid argument. After setting the configuration option `charon.plugins.socket-default.set_sourceif` to "1", sending rekey messages worked.

4.1.7 Member registration

As soon as a member joins the group (it passes the authentication step) a new group KEK is generated and sent to the group members over the existing rekey SA via a `GSA_REKEY`

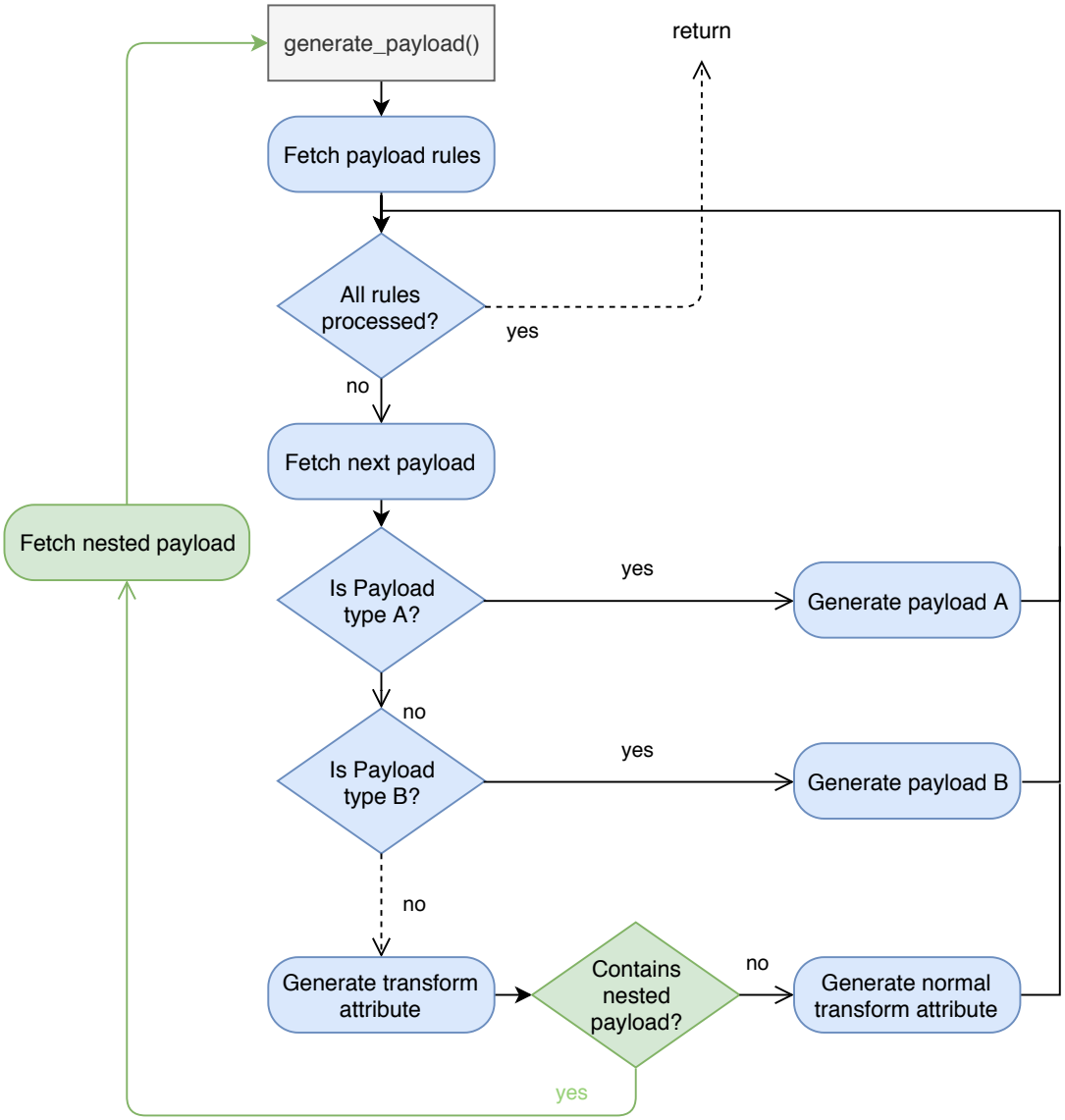

Figure 4.2: Payload generation process

message. A future version of the implementation would also renew the current TEK(s) and include those within the `GSA_REKEY` message. Afterwards the successful `GSA_AUTH` response is sent to the new GM containing the LKH key array (and including the new KEK) and would in future also include the new TEK(s). By renewing the group KEK before a group member is added, backward access control is ensured. As the new GM does not receive the old KEK, it cannot use that to decrypt rekey messages it might have recorded before and thus get access to the old TEK(s).

4.1.8 Member eviction

For the operator of the GCKS it should be possible to administratively evict hostile members from their respective group. Sending commands to Strongswan from the command line or other programs is possible by means of the Stroke plugin. This plugin presents a command line interface to the user which he can invoke by calling the `ipsec stroke` command. Available commands allow for investigating internal program state of the charon daemon, initiating or manipulating IKE SAs. To be able to evict registered members from groups, a new command, `evict`, has been implemented. It requires two parameters to be given, the first one is the ID of the respective group and the second one is the ID of the member within that group. The following figure shows the example output of a command which evicts the member with ID 6 from the group with ID 2:

```
# ipsec stroke evict 2 6
Successfully evicted member 6 from group 2
#
```

4.2 RIOT

A fully working G-IKEv2 implementation with rekey and LKH support mainly needs three different types of databases which need to be held in memory during runtime, two of which are security association databases (SADs). A SAD contains a set of SA entries, that is, security related properties and parameters such as cryptographic keys or information about the cryptographic algorithms used. The needed databases are:

- IKE SAD
- Rekey SAD
- LKH key storage

Theoretically, the LKH key storage could be embedded into the rekey SAD. But from an architectural point of view, it makes sense to keep the G-IKEv2 rekey related module separate from the LKH module. However, the LKH storage arrays are still referenced from within the rekey SA entries.

As RIOT targets mainly platforms with very small amount of RAM, all components of the G-IKEv2 rekey implementation which are dealing with persistent state that needs to be held in memory during runtime, are designed to use only statically allocated memory. That is, at runtime no dynamic memory is allocated on the heap and thus, memory allocation at runtime can be predicted very accurately. Hence, platforms with sufficient RAM for use with G-IKEv2 can be selected easily. However, the size of the rekey SAD and LKH key storage is still configurable and therefore scalability is still maintained.

4.2.1 Static memory block allocator

Tobias Heider integrated an implementation of a static memory block allocator into RIOT [9]. The implementation consists of an API to allocate and free memory blocks

of a fixed size up to a configurable maximum number of blocks. The size of one block as well as the number of available blocks are both configured at compile time. Notably both allocating and freeing blocks are executed in constant time. Also the control structure has a constant size independent of the configured number of available blocks. It works by storing a pointer to the next free block at the top of each free block itself. The first free block is pointed to by the static control structure. Allocating chunks is done by advancing the `first_free` pointer and returning the block which the pointer referenced beforehand. Freeing chunks is done by setting the `first_free` pointer to the block which is freed and store the previous `first_free` pointer into that block itself. The exact procedure is depicted in the left column of figure 4.3. In summary the implementation provides a lightweight and fast solution for use cases where dynamic allocating and freeing of fixed-size memory chunks is needed. However, it is not possible to iterate over the used blocks e.g. for searching a specific block as the used blocks don't reference each other. Thus, an improved version of the static memory block allocator has been implemented. In addition to storing pointers to the free nodes, each used block is prepended with a pointer to the next used block. Essentially, a second linked list structure is created which contains all used blocks. Figure 4.3 shows the differences between the existing implementation and the improved version.

4.2.2 IKE SAD

The IKE SAD holds information about the IKE SA which serves as the registration SA in G-IKEv2. It is implemented by means of the static memory block allocator described in 4.2.1. As for incoming (G-)IKEv2 packets the SAD needs to be searched for entries corresponding to the SPIs specified in the packet header, the improved version of the allocator as described in 4.2.1 is used to be able to iterate over all allocated SAD entries. The linked list structure being used for that is shown in listing 4.3. Most parts of the IKE SAD already existed as a result of Tobias Heiders preliminary work. However, to add proper support for the threaded architecture of the implementation, which is described in 4.2.5, a few changes have been introduced. Firstly, the current state which a SA is in at a given moment is stored in a newly introduced field. The allowed states are described in listing 4.2. This allows for correctly identifying which action has to be taken to correctly establish and authenticate the IKE SA. A newly allocated IKE SA starts in state `NEW`, as soon as the `IKE_SA_INIT` request is sent, the state is changed to `CONNECTING`. After the response to the initial exchange has been processed and a `GSA_AUTH` request has been sent, the state is set to `WAIT_AUTH` and a correctly established and authenticated IKE SA finally is in state `ESTABLISHED`.

4.2.3 Rekey SAD

The rekey SAD is implemented in a similar way as the IKE SAD. An entry in the rekey SAD is represented by a structure described in listing 4.4. This structure contains the SPIs identifying the SA, information about the used encryption and integrity algorithms as well as the message ID of the last received multicast `GSA_REKEY` message for protection

Figure 4.3: Improved static memory block allocator

```
typedef enum {
    NEW,
    CONNECTING,
    WAIT_AUTH,
    ESTABLISHED,
    DELETING
} ikev2_sa_state_t;
```

Listing 4.2: IKE SA possible states

against replay attacks. Additionally, the used KEK management type is stored as well as some general purpose storage which can be used by the KEK management algorithm to point to its internal structures (as done with the LKH management algorithm) or can be used to store the group KEK itself, if no KEK management algorithm is used.

```
typedef struct sad_entry {
    struct sad_entry *next;
    ikev2_sa_t sa;
} sad_entry_t;
```

Listing 4.3: IKE SAD entry structure

```
typedef struct {
    uint64_t spi_i;
    uint64_t spi_r;
    uint16_t kek_mgmt_type;
    uint16_t integrity_algorithm;
    uint16_t encryption_algorithm;
    uint16_t encryption_keylen_bytes;
    uint8_t kek_integrity_key[INTEGRITY_KEY_MAXLEN];

    uint16_t kek_msg_id;
    union {
        uint8_t data[REKEY_SA_PRIVATE_BYTES];
        void *ptr;
    } private;
} rekey_sa_t;
```

Listing 4.4: G-IKEv2 Rekey SA structure

4.2.4 LKH database

The LKH database introduces the concept of a keystore. A keystore is a chunk of memory together with functions to add, delete or replace keys within that chunk. The API is kept generic and not restricted to be used with G-IKEv2 only. It is possible to use multiple keystores simultaneously, for example to handle multiple rekey SAs of multiple group memberships, the process of allocating or freeing keystores as needed is implemented by the means of the static memory block allocator. As the associated rekey SAs store pointers to the LKH keystore they use, an iteration mechanism over all currently allocated keystores is not needed. Thus the existing, basic version of the allocator is used. The number of LKH keystores which can be used as well as the size of each keystore are configured statically by means of preprocessor directives.

The API is kept simple and contains functions to allocate a new, empty keystore as well as deallocating a keystore which won't be used anymore. Functions to insert new keys into a keystore, retrieve a key from the keystore as well as replace a key in the keystore have been implemented. As the LKH algorithm strongly suggests replacing existing keys is done by decrypting newly received keys, a function that replaces a key within a keystore by decrypting a ciphertext given as parameter with another key in that keystore is introduced. The overview of available functions is show in listing 4.5.

```
lkh_keystore_t *lkh_keystore_get(
    uint16_t key_size,
    uint8_t key_metadata_size
);

void lkh_keystore_free(
    lkh_keystore_t *
);

int lkh_key_get(
    lkh_keystore_t *,
    uint8_t index,
    lkh_key_t *key
);

int lkh_key_add(
    lkh_keystore_t *,
    uint16_t key_id,
    void *key_metadata,
    uint8_t *key_data,
    lkh_key_t *target_key
);

int lkh_key_update_at_index(
    lkh_keystore_t *,
    uint8_t index,
    uint8_t decryptor_index,
    uint16_t encr_alg,
    uint16_t encr_keylen_bytes,
    uint8_t *new_key_ciphertext
);
```

Listing 4.5: RIOT LKH functions

4.2.5 G-IKEv2 Thread

Due to the GCKS-initiated `GSA_REKEY` exchanges which can arrive at any point in time and also to support proper handling of other exchanges initiated by remote peers, a new threaded architecture was set up in contrast to the synchronous program flow in the existing implementation. Since RIOT OS supports creating multiple execution threads, a dedicated thread for (G-)IKEv2 related processing and communication is created, in the following denoted as G-IKEv2 thread. The main component of the thread is an event loop which uses the standard RIOT interprocess communication mechanism to process control requests or incoming data. There are three types of events which will be processed by the G-IKEv2 thread.

Network packets Those are incoming network messages destined to the G-IKEv2 port 848.

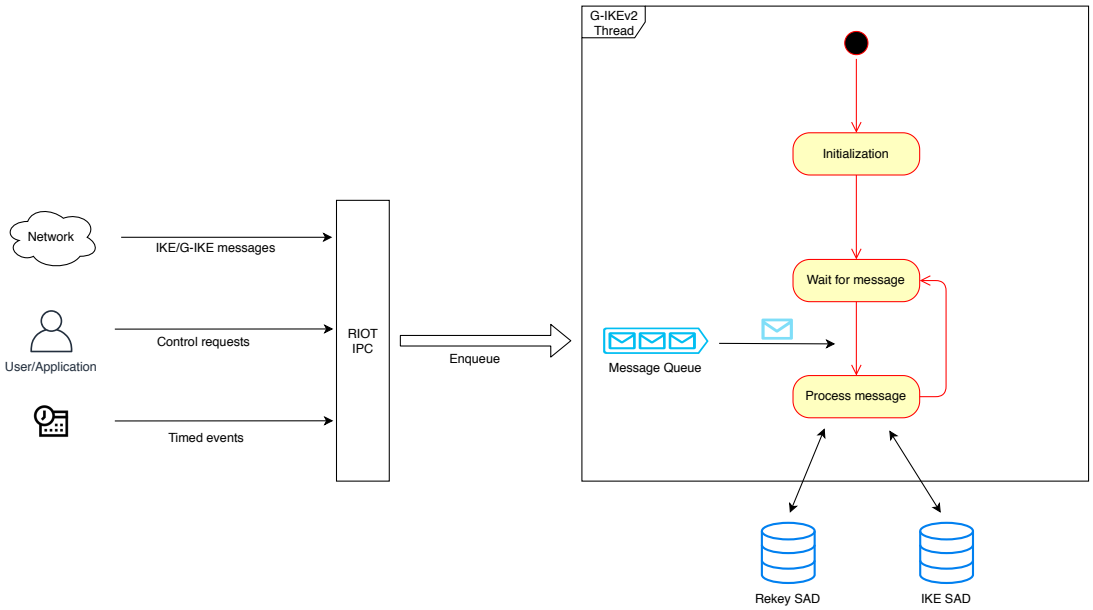

Figure 4.4: G-IKEv2 thread structure

User/application control requests Those are commands issued interactively by the user or by an application running on the target. Might be used to establish/delete a GSA or show information about the current entries in the IKE or rekey SAD.

Timed events This category stands for events, which can be used to delete half open SAs after a timeout was reached without receiving a response from the GCKS. However, in the current version those are not yet implemented.

As soon as a RIOT application wants to initiate a G-IKEv2 connection, it calls a library function and passes IKE related parameters to it. The function starts the G-IKE thread, if it is not yet running and then delivers a connection initiation request message to it, using RIOTs IPC mechanism. The G-IKEv2 thread then builds the payloads needed for the `IKE_SA_INIT` request, allocates a new IKE SA in the SAD, sends the `IKE_SA_INIT` request and sets the SA's state to `CONNECTING`, as described in 4.2.2. Thereafter, the thread returns to an idle state, waiting for either control requests or a response from the GCKS. When a `GSA_AUTH` response is received from the GCKS, it is parsed and processed. When the client registered successfully at the GCKS, the corresponding entry in the rekey SAD is populated. When a multicast `GSA_REKEY` message is received from the GCKS, its corresponding rekey SAD entry is searched and used for updating the respective keys. The thread itself functions completely stateless, all state is preserved in the IKE SAD and the Rekey SAD. Figure 4.4 depicts the control flow in the G-IKEv2 thread. Its structure makes it easy to extend its functionality and for example add support for other protocols such as normal IKEv2.

5 Evaluation

A test scenario is set up on the basis of which the implementation is then evaluated against the previously defined requirements, especially the non-functional requirements such as computing time or memory usage characteristics. The evaluation focuses on the GM part implemented in RIOT, as the platforms on which it is designed to be used have strong constraints and thus need more sophisticated analysis.

5.1 Test scenario

The implementation was tested on the very popular hardware platform Arduino Due. The Arduino Due is equipped with an ARM Cortex-M3 microcontroller which operates at a clock speed of 84 MHz. It has 96 KiB of available memory and 512 KiB of available Flash memory. Network interconnection is realized with a Keyestudio Ethernet Shield which supports Ethernet with a maximum data rate of 100 MBit/s. IPv6 fragmentation support in RIOT is enabled by including the module `gnrc_ipv6_ext_frag`. This module is used to support `GSA_REKEY` messages larger than an Ethernet MTU (1500 Bytes). The GCKS is running as a daemon on a Linux based virtual machine (VM). AES in CBC mode with a key length of 128 bits is used for all cryptographic operations as specified in [3]. The used Strongswan configuration can be found in Appendix A.

5.2 Results

This section shows the results of the measurements performed and evaluates them in relation to the requirements.

5.2.1 Memory

The memory usage of the G-IKEv2 implementation together with LKH is composed of its individual modules. There is also memory usage which can only indirectly be attributed to the G-IKEv2 implementation itself but is still necessary for it to work correctly, namely the GNRC packet buffer. Due to the inefficient specification of how LKH keys are updated via `GSA_REKEY` messages (see 3.6 for more information), those messages can become quite big. [1] However, those message still need to be stored in the GNRC packet buffer. Due to a slight inefficiency in the current implementation when the encrypted `GSA_REKEY` message is decrypted, the decrypted contents is put in a buffer

[1] An Ethernet frame size of 1490 Bytes has been observed during evaluation for a `GSA_REKEY` message replacing keys for a LKH tree of depth 8.

Table 5.1: Memory usage

Component	Memory usage (Bytes)
LKH key storage	194
IKE SAD	1,036
Rekey SAD	140
G-IKEv2 thread stack + message queue	2,574
GNRC packet buffer	8,192
$\sum$	12,136

separately allocated from the packet buffer, which increases the needed space. Future versions could easily switch to in-place decryption and thus save space on the packet buffer.

Complete `GSA_REKEY` parsing support using a tree depth of 6 (supporting up to 64 members) has been confirmed working with the configuration displayed in table 5.1. Both the Rekey SAD and the IKE SAD were configured to support the storage of one SA or one group membership, respectively. As the RIOT kernel itself, specific modules and the different network stack protocols (UDP/IPv6) also consume memory, in practice a microcontroller needs more memory than the calculated total of 12 KiB. This specific configuration described in the table has a total memory consumption of 18 KiB and as such should be possible to run on other microcontrollers equipped with at least that amount of memory.

5.2.2 Computation effort

Table 5.2 shows an overview of the performance metrics measured on the previously described Arduino Due platform. Each value in the table is the computed average of five samples measured. Only the average is given, since each individual measurement did not deviate from the average by more than 0.3%. Thus, only providing the average value is sufficient enough, as its significance is very high. The following metrics are measured and evaluated:

GSA_AUTH The time needed to completely process a `GSA_AUTH` response is listed in this column.

GSA_REKEY - worst case The values in this column quantify the time the device needed for completely processing a `GSA_REKEY` message in the worst possible case. That is, the rekey message was sent as a result of excluding the direct sibling group member. In that case, the GM has to replace each key in its LKH storage except for the leaf node.

GSA_REKEY - best case These values show the time the device needed for processing a `GSA_REKEY` message in the best case. That is, within the binary LKH key tree, the group member does not reside in the half in which the excluded group member

Table 5.2: Computation time measurements

Tree depth	Max group size	GSA_AUTH (μs)	GSA_REKEY (μs)	
			worst case	best case
4	16	4,333	3,475	2,133
5	32	4,526	4,627	2,131
6	64	4,697	6,199	2,148
7	128	4,863	7,917	2,145
8	256	5,083	10,050	2,147
9	512	5,200	12,252	2,153
10	1024	5,454	14,930	2,152

was positioned. In this case, only one additional decryption operation is needed, which is decrypting the root key with the respective child key the group member possesses.

5.2.3 Interpretation

Overall it can be stated that the group member LKH implementation on RIOT OS is very efficient for even large groups supporting more than 1000 group members. The time needed for processing the GSA_AUTH response only slightly increases with a higher tree depth. The reason for this is that other than decrypting the response itself, no more cryptographic operations need to be performed on the provided keys. They just need to be copied into an appropriate place within the LKH module. The best case scenario when handling a GSA_REKEY message is clearly not depending on the tree depth, as the computational effort always stays the same, that is in any case only the new root key has to be decrypted using the corresponding left or right child key. As expected, the most expensive operation is the worst case GSA_REKEY message handling. There the group member needs to perform $d - 1$ additional cryptographic operations for a LKH tree of depth d. Every key along the path to the root key needs to be decrypted using the respective preceding key. However, even with a LKH tree depth of 10, which supports a large group with 1024 possible members, the time needed to process a GSA_REKEY message is just about 15ms. Thus, even large dynamic groups with multiple members joining or leaving the group every second, can be supported on a low end platform like the Arduino Due.

5.3 Summary

Table 5.3 compares the result of the implementation against the previously defined functional and non-functional requirements. Nearly all of them could be fulfilled, but two remarks have to be made. Firstly, the LKH key tree generation and management mechanism as implemented currently is not very sophisticated. The tree is initialized with a fixed depth and the tree is fully populated with freshly generated keys. Group members

Table 5.3: Summary of the requirements

Functional requirements	
Protocol conformity	✓[a]
Multicast messaging	✓
LKH key tree generation and management	✓
LKH array storage and management	✓
Replay protection	✓[b]
Member eviction user interface	✓
Non-functional requirements	
low computation effort	✓
efficient RAM usage	✓
scalability	✓
extensibility	✓

[a] Could not be verified due to the absence of another LKH implementation in G-IKEv2
[b] Implicitly fulfilled since specified in G-IKEv2

respectively their keys can be replaced easily, but the tree structure is fixed after initialization. Improvements could be made in terms of lazy generation of the keys just as they are needed or by implementing re-balancing mechanisms which could serve a sparsely populated group more efficiently. Although being already quite efficient, also the RAM usage of the GM implementation still has room left for improvements. As stated above, several decryption operations could be shifted towards in-place decryption which would render the need for an additional buffer allocation on the packet buffer obsolete.

6 Conclusion and future work

This work showed that an implementation of a multicast group rekeying mechanism based on the G-IKEv2 protocol and the hierarchical group key management algorithm LKH is capable of being used even on lower end devices such as an Arduino Due with less than 100MHz of clock speed and less than 100 KiB of memory.

As described in section 3.6 and also noted in [5] the way replaced LKH keys are sent to the group members, as standardized in G-IKEv2, is quite inefficient. By integrating the alternative approach which greatly reduces message sizes into the G-IKEv2 specification, the implementations for both Strongswan and RIOT could adopt and provide for more efficient group rekeying.

Future iterations of the implementation could add improvements in the way the LKH key tree is managed in the Strongswan implementation. Currently, the tree is created with a fixed depth at the startup of the Strongswan daemon. It supports the maximum number of group members but performs poorly in sparsely populated groups, as for those groups to improve efficiency, the height of the tree could be reduced.

Currently, LKH is the only group key management algorithm which is officially included in the standardization of G-IKEv2 protocol. But there exist more algorithms with even lower networking overhead than LKH has, such as OFT and CAKE. The adoption of the G-IKEv2 protocol would greatly benefit from adding a standardization of those algorithms.

Appendix A: Strongswan configuration sample

Listing 1: ipsec.conf

```
# ipsec.conf - strongSwan IPsec configuration file

# Set a default IKE proposal
conn %default
       ike=aes128-sha1-prfsha1-ecp256

# GROUP-IKE GROUP 1 DEFINITION
conn GIKE-G000001
       left=ff02::1
       right=ff02::1
       authby=psk
       auto=add

# GROUP-IKE GROUP 1 MEMBER DEFINITIONS
conn GIKE-G000001MEMBER000001
       left=fe80::1
       # GM address
       right=fe80::b863:d4ff:fe1a:65a2
       authby=psk
       auto=add
       # GM identification
       rightid=keyid:12345678

conn GIKE-G000001MEMBER000002
       left=fe80::1
       # GM address
       right=fe80::626e:ff:fe77:1e21
       authby=psk
       auto=add
       # GM identification
       rightid=keyid:abcdefgh
```

Listing 2: ipsec.secrets

```
# ipsec.secrets - strongSwan IPsec secrets file
#
keyid:abcdefgh : PSK "supersecret"
keyid:12345678 : PSK "supersecret"
```

List of Figures

Bibliography

[1] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch. RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. *IEEE Internet of Things Journal*, 5:4428–4440, 2018.

[2] W. Engelbrecht. Group Key Management with Strongswan. `http://www.mnm-team.org/pub/Fopras/enge18/`, 2018. retrieved on October 08, 2019.

[3] S. Frankel, S. Kelly, and R. Glenn. The AES-CBC Cipher Algorithm and Its Use with IPsec. `https://tools.ietf.org/html/rfc3602`. Accessed: 2019-11-12.

[4] C. Guang-Huei and C. Wen-Tsuen. Secure Broadcasting Using the Secure Lock, 1989.

[5] T. Guggemos, K. Streit, M. Knüpfer, N. Felde, and P. Hillmann. No Cookies, just CAKE: CRT based Key Hierarchy for Efficient Key Management in Dynamic Groups. In *International Conference for Internet Technology and Secured Transactions*, pages 25–32, 12 2018.

[6] T. Hardjono and B. Weis. Multicast Group Security Architecture. `https://tools.ietf.org/html/rfc3740`, 2004. retrieved on September 22, 2019.

[7] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). `https://tools.ietf.org/html/rfc2409`, 2011. retrieved on November 13, 2019.

[8] H. Harney and C. Muckenhirn. Group Key Management Protocol (GKMP) Architecture. `https://tools.ietf.org/html/rfc2094`, 1997. retrieved on April 4, 2019.

[9] T. Heider. memory array allocator. `https://riot-os.org/api/group__sys__memarray.html`. retrieved on November 13, 2019.

[10] T. Heider. Minimal G-IKEv2 implementation for RIOT OS. `http://www.mnm-team.org/pub/Fopras/heid17`, 2017. retrieved on October 08, 2019.

[11] P. Hillmann, M. Knüpfer, and G. Dreo Rodosek. CAKE: Hybrides Gruppen-Schlüssel-Management Verfahren. *10. DFN-Forum Kommunikationstechnologien, Lecture Notes in Informatics (LNI)*, 2017.

[12] L. Jing and Y. Bo. Collusion-Resistant Multicast Key Distribution Based on Homomorphic One-Way Function Trees, 2011.

[13] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen. Internet Key Exchange Protocol Version 2 (IKEv2). `https://tools.ietf.org/html/rfc7296`, 2014. retrieved on October 30, 2019.

[14] M. Lenders. Analysis and Comparison of Embedded Network Stacks, 2016.

[15] D. A. McGrew and A. T. Sherman. Key Establishment in Large Dynamic Groups Using One-Way Function Trees, 1998.

[16] A. Steffen. strongSwan: The new IKEv2 VPN Solution. `https://www.strongswan.org/docs/LinuxTag2007-strongSwan.pdf`, 2007.

[17] D. M. Wallner, E. J. Harder, and R. C. Agee. Key Management for Multicast: Issues and Architectures. `https://tools.ietf.org/html/rfc2627`, 1999. retrieved on April 4, 2019.

[18] B. Weis, S. Rowles, and T. Hardjono. The Group Domain of Interpretation. `https://tools.ietf.org/html/rfc6407`, 2011. retrieved on November 13, 2019.

[19] B. Weis and V. Smyslov. Group Key Management using IKEv2. `https://tools.ietf.org/html/draft-yeung-g-ikev2-16`, 2019. retrieved on November 13, 2019.