# IKEv2 Separation: Extraction of security critical components into a Trusted Computing Base (TCB)

Reto Bürki, Adrian-Ken Rüegsegger

February 4, 2013

University of Applied Sciences Rapperswil (HSR), Switzerland

**Abstract**

The IPsec protocol relies on the correct operation of the IKE key exchange to meet its security goals. The implementation of the IKE protocol is a non-trivial task and results in a large and complex code base. This makes it hard to gain a high degree of confidence in the correct operation of the code.

We propose a component-based approach by disaggregating the IKE key management system into trusted and untrusted components to attain a higher level of security. By formulating desired security properties and identifying the critical components of the IKE protocol, a concept to split the key management system into an untrusted and trusted part is presented. The security-critical part represents a trusted computing base (TCB) and is termed "Trusted Key Manager" (TKM). Care was taken to only extract the functionality that is absolutely necessary to ensure the desired security properties. Thus, the presented interface between the untrusted IKE processing component and TKM allows for a small and robust implementation of the TCB. The splitting of the protocol guarantees that even if the untrusted side is completely subverted by an attacker, the trusted components uphold the proposed security goals.

The viability of the design has been validated through a prototypical implementation of the presented system. The untrusted parts of the IKE daemon have been implemented by extending the existing strongSwan IKE implementation. The trusted components have been implemented from scratch using the Ada programming language, which is well suited for the development of robust software. The new Design-by-Contract feature of Ada 2012 has been used for the implementation of state machines, to augment the confidence of operation according to the specification.

# Contents

# List of Figures

# List of Tables

# Listings

# Chapter 1

# Introduction

In a system with high requirements on security, functions relevant to guarantee these requirements must be isolated from the rest of the system and consolidated in a Trusted Computing Base (TCB). To be trusted, this code must be as minimal as possible to allow formal verification of code correctness. Lampson et al. [20] define the TCB of a computer system as:

> A small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security.

It is an easier task to design a system from scratch with separation properties in mind than dividing an existing project or protocol later. This is not always possible, and more importantly, sometimes not intended. Functionality in an existing system identified as uncritical should be left as is as much as possible.

In order to isolate functionality in a TCB, critical sections of existing systems must be identified and they must be separated into a critical (trusted) and non-critical (untrusted) part. Communication mechanisms between the sections needs to be established, which must be robust and well defined. If an attacker is able to compromise the untrusted-part of the system, the security and integrity functions guaranteed by the TCB must still hold.

Figure 1.1 depicts a simple schematic of an example TCB. Components colored in red specify trusted components inside the TCB. The TCB normally consists of multiple such components which implement different, separated functionality. One or more untrusted components colored in black exchange data with the TCB over an interface. This coloring scheme is used throughout this document to label untrusted and untrusted components.

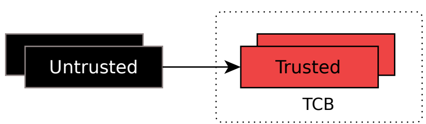


Figure 1.1: Trusted Computing Base

## 1.1 Overview

This section gives an introduction into the terminology and systems used in this project and explains the basic key concepts. Section 1.1.1 briefly outlines IPsec and the IKEv2 protocol, section 1.1.2 introduces an implementation of this protocol in the form of the strongSwan[1] project. Section 1.1.3 summarizes the most important aspects of the Ada programming language, which is used to implement the Trusted Key Manager (TKM) specified by this paper. The term TKM is explained in the following section 1.1.4.

### 1.1.1 IPsec and IKEv2

Internet Protocol Security (IPsec) provides, as the name implies, security services to the Internet Protocol (IP). This is done by encrypting and authenticating IP packets of communication sessions. The protection is transparent to the communicating applications because it is performed in the IP layer. To protect packets, cryptographic transforms are applied to them which in turn require cryptographic keys. The bundle of algorithms and data that provide the parameters necessary to operate these cryptographic transforms are called a security association (SA). For more information on the IPsec protocol suites, the reader is directed to the corresponding *"Security Architecture for the Internet Protocol"* RFC [18].

Parameters and keys needed to establish a security association are usually provided to the IPsec protocol suite by means of the Internet Key Exchange (IKE) protocol. The IKE protocol is responsible for the key establishment phase and the negotiation of the cryptographic algorithms between communicating endpoints. There are two versions of the IKE protocol: IKEv1 and IKEv2 [9, 16]. IKEv2 was designed to add new features and correct some problems found in the previous version. This project exclusively targets the newer IKEv2 protocol, IKEv1 is not considered.

To negotiate cryptographic keys, SA parameters and to perform mutual authentication, message pairs are exchanged between the participating peers. Section 2.1 explains the message exchanges of the IKEv2 protocol in detail. The service implementing the IKE protocol is normally provided by an user space application.

### 1.1.2 strongSwan

The strongSwan project is an open-source IPsec-based VPN solution for Unix-like operating systems. It provides the charon daemon, which is a feature-rich implementation of the Internet Key Exchange protocol version 2[2] (IKEv2) as specified in [16].The software is implemented using the C programming language with an object oriented (OO) approach. This allows to emulate modern programming paradigms while still using a standard C compiler and tool set[3].

By using a flexible plugin architecture, the strongSwan project can be easily extended with new features. The task of adding new features can be reduced

[1] http://www.strongswan.org/

[2] The project also implements IKE version 1 (IKEv1) but this project is only concerned with IKEv2.

[3] http://wiki.strongswan.org/projects/strongswan/wiki/ObjectOrientedC

to writing a new plugin. This architecture has proven to be very helpful in the course of this project, as very few changes were required in the upstream core strongSwan code to implement the Trusted Key Manager (TKM, 1.1.4) architecture.

### 1.1.3 Ada

Ada is a structured, strongly typed programming language. The language has initially been designed by Jean Ichbiah from Honeywell Bull in the 1970s. Ada has a very similar structure to Pascal and is often used for systems with a special demand for security and integrity.

The development of Ada was initiated by the US Department of Defense (DoD) in order to consolidate and supersede the hundreds of programming languages used in their countless projects. The new language should comply with all identified DoD requirements (dubbed “Steelman Language Requirements” [10]), which focused strongly on security and safety.

Ada was the first standardized high-level programming language [11]. The current version is Ada 2012 which supports all modern programming paradigms. It has just recently been released[4] as an ISO standard [2]. Ada 2012 adds the possibility to use contract-based programming methods (“Design by contract” [23]).

Ada compilers, before used in practice, have to pass a standardized test suite which guarantees the compliance of the compiler with the Ada standard. Since Ada provides many features which aid in the development of safety and security critical applications, it is nowadays mostly used in areas where such aspects are important. The primary industries making use of Ada are avionics, railway systems, banking, military and space technology.

The language is named after Lady Ada Lovelace[5], the daughter of Lord Byron, who is considered to be the first computer programmer.

GNAT, a free-software compiler for the Ada programming language, is available as part of the GNU Compiler Collection.

### 1.1.4 Trusted Key Manager

The Trusted Key Manager is a minimal TCB developed during this project which implements the identified security-critical functions of the IKEv2 protocol using the Ada programming language. The TKM is explained in detail in section 5.5.

The TKM uses the tkm-rpc library to communicate with the strongSwan charon daemon in the untrusted part. This library is also written in Ada and explained in section 5.3.

## 1.2 Related work

The concept of decomposing larger systems into smaller, trusted parts dates back to John Rushby in 1981 [26]. The most prominent implementations of

[4]The announcement was made on December 18, 2012: http://www.ada-europe.org/press/20121218-Ada2012.pdf

[5]Ada Lovelace - http://en.wikipedia.org/wiki/Ada_Lovelace

the concept exist in the form of microkernels (µ-kernels), which provide the foundation to separate functionality into smaller, separated parts by providing compartments for subjects running in userspace. Examples of such systems are Fiasco[6], L4Ka::Pistachio[7] and Coyotos[8]. Type-1 (bare-metal) hypervisors like Xen are intentionally excluded from the list because Xen requires a complete Linux kernel (dom0) with direct access to hardware to operate. The fact that the dom0 kernel must be accounted as part of the trusted system makes it unsuitable for in-depth review and therefore unusable as part of a TCB[9].

Even though the concept proposed by Rushby offers many advantages related to security and integrity, it has not been widely realized. Common operating systems like Windows, Linux and *BSD variants use a monolithic kernel, which itself must be trusted as a whole, even though the compromise of a device driver can corrupt the complete system.

One reason seems to be the tremendous effort needed to adapt existing software to a separation concept. In order to move critical parts into a TCB, the existing code must be studied and sensitive parts re-implemented using the corresponding APIs and methods of the underlying separation platform. Of course, the complete system could be rewritten for the dedicated secure environment, but often this is not possible and especially not desired for code deemed as untrusted. The dedicated goal is to only re-implement sensitive parts while leaving the untrusted part mostly untouched.

A different reason for the disregard of Rushby's ideas by most software vendors is the focus on extending the functionality of existing products by adding new features. This phenomenon is known as *feature creep*.

Research has been done in the formal analysis of the IKEv1 and IKEv2 protocols [7, 22], pointing out weaknesses in both standards. The separation of the sensitive part from the bulk of the IKE protocol seems to be a valuable effort to minimize the working surface of attacks. Nevertheless, the IKEv2 separation protocol described in this paper must still undergo the same rigorous verification as the original protocols to formally show the delivered security improvements compared to its monolithic ancestor.

The presented project is based on the concept of IKEv2 disaggregation described in [25], which is the result of preliminary research on the same topic.

[6] http://os.inf.tu-dresden.de/fiasco/
[7] http://www.l4ka.org/65.php
[8] http://www.coyotos.org/
[9] Concepts for Dom0 disaggregation exist [5] but they have not been implemented in Xen.

# Chapter 2

# Analysis of strongSwan

This chapter describes the current operation and the inner workings of the strongSwan charon IKEv2 daemon. A deep understanding of these mechanisms is a prerequisite for the extraction of sensitive functionality from the daemon into a minimal trusted part to achieve the requirements formalized in section 3.5 later.

The following section 2.1 will therefore provide an introduction into the IKEv2 message exchanges in general to give the reader a basic understanding of the protocol. The main aim of the section is to identify critical payloads contained in the message exchanges. Section 2.2 will then analyze the code flow inside the strongSwan charon daemon implementing the actual IKEv2 exchanges and payload handling.

## 2.1 IKEv2 protocol analysis

The following section provides a detailed analysis of the IKEv2 message exchanges (as specified by [16]), focusing on the security relevance of the transmitted data. All communication using IKE consists of a request / response pair. The analysis of the message exchanges concentrates on the role of the initiator since the responder case varies only slightly.

In the following descriptions, the message payloads are indicated by names as listed in table 2.1.

Every IKE message contains a message *ID* as part of its fixed header (*HDR*). This message *ID* is used to match up requests and responses, and to identify retransmissions of messages [16]. The fixed header does not contain security-relevant information and is therefore omitted from the discussion.

A value declared as *critical* or *sensitive* in the following sections must not be accessible by the untrusted part, i.e. it must not be present in memory or storage accessible from within the untrusted part. Other payloads (such as *AUTH*) are calculated from critical values inside the TCB but then handed to the untrusted part for further processing and transmission.

| Notation | Payload |
|---|---|
| AUTH | Authentication |
| CERT | Certificate |
| CERTREQ | Certificate Request |
| CP | Configuration |
| D | Delete |
| EAP | Extensible Authentication |
| HDR | IKE header (not a payload) |
| IDi | Identification - Initiator |
| IDr | Identification - Responder |
| KE | Key Exchange |
| Ni, Nr | Nonce |
| N | Notify |
| SA | Security Association |
| SK | Encrypted and Authenticated |
| TSi | Traffic Selector - Initiator |
| TSr | Traffic Selector - Responder |
| V | Vendor ID |

Table 2.1: IKEv2 payloads

### 2.1.1 Notation

The exchanges are presented as a communication between peers *A* and *B*. The arrows represent the direction from the source to the destination of the message. The transmitted values are listed on the right-hand side. Optional parts of the exchange are enclosed in square brackets. The notation SK { ... } indicates that the payloads listed inside the curly brackets are encrypted and integrity protected.

### 2.1.2 IKE_SA_INIT

The first pair of messages (IKE_SA_INIT) negotiate cryptographic algorithms, exchange nonces, and do a Diffie-Hellman exchange [16]:

| | | | | | |
|---|---|---|---|---|---|
| 1 | `A` | $\rightarrow$ | `B` | : | *HDR, SAi1, KEi, Ni* |
| 2 | `B` | $\rightarrow$ | `A` | : | *HDR, SAr1, KEr, Nr, [CERTREQ]* |

The *SAi1* payload states the cryptographic algorithms the initiator supports for an IKE SA. This payload is not considered critical because the TKM will only support a subset of cryptographic algorithms which are strong enough and believed to be secure. A deviation from allowed proposals would only result in a non-functional configuration since the TKM enforces the allowed algorithms of a specific connection.

Child keys are derived from the shared secret value resulting from the Diffie-Hellman exchange after the IKE_SA_INIT messages. Therefore the TKM must implement the DH protocol in the TCB and compute the public *KE* payload on behalf of the untrusted part. The peers exchange the *KE* payloads in the initial IKE_SA_INIT messages as shown above.

The nonces *Ni* and *Nr* are used as input to cryptographic functions and provide freshness to the key derivation technique used to obtain keys for the child SA. Therefore the nonce *Ni* used in the initial exchange must be randomly chosen, must be at least 128 bits in size, and must be at least half the key size of the negotiated pseudo-random function (PRF). These constraints must be enforced by the TKM. Values created by the responder can not be controlled by the TKM so these values are taken as is. This is obviously true for all IKE message exchanges.

The responder may also send a list of its trust anchors in the CERTREQ payload. This has no relevance for the TCB because it maintains a separate list of trusted root CAs.

| | |
|---|---|
| *Created by TKM* | KEi, Ni |

Table 2.2: Critical IKE_SA_INIT payloads

### 2.1.3 IKE_AUTH

After the completion of the IKE_SA_INIT exchange, each party is able to compute SKEYSEED, from which all keys are derived for that SA. The messages that follow are encrypted and integrity protected in their entirety, with the exception of the message headers. The keys used for the encryption and integrity protection are derived from SKEYSEED and are known as SK_e (encryption) and SK_a (authentication, a.k.a. integrity protection). Separate SK_e and SK_a keys are computed for each direction. The payloads marked with SK { ... } are protected using the direction's SK_e and SK_a ([16], section 1.2).

| | | | | | |
|---|---|---|---|---|---|
| 3 | A | → | B | : | *HDR, SK {IDi, [CERT,] [CERTREQ,] [IDr,] AUTH, SAi2, TSi, TSr}* |
| 4 | B | → | A | : | *HDR, SK {IDr, [CERT,] AUTH, SAr2, TSi, TSr}* |

As stated in the previous section, the DH protocol must be implemented inside the TCB. As a result, the SK_e and SK_a keys must be provided to the untrusted part. These keys are not considered critical because an attacker taking over the untrusted part is already able to extract all information protected by these keys (see the threat model section 3.1).

The initiator asserts its identity with the *IDi* payload. This value is not sensitive itself but the TKM must enforce correct identities during the authentication step to assure that only trusted peers are allowed.

The authentication payload *AUTH* contains the signature allowing the peers to verify each other's authenticity. The value inside this payload must be created by the TKM since it is signed by a private key only known to the TCB. The signature is handed to the untrusted part because the TKM assures that the PRF used to generate it (see 5.5.7.1) is strong enough.

Analogous to the IKE_SA_INIT exchange, the *SAi2*/*SAr2* payloads are not considered critical and can be configured directly in the untrusted part. The same is true for the *TS* payloads. The TKM enforces the correct algorithms and peer addresses before deriving child keys.

The initiator might also send its user certificate in a *CERT* payload and a list of its trust anchors in *CERTREQ* payload(s). If any *CERT* payloads are included, the first certificate provided must contain the public key used to verify the AUTH field [16]. These payload are uncritical since invalid certificates would result in an authentication failure.

| | |
|---|---|
| *Created by TKM* | SK, AUTH |
| *Enforced by TKM* | ID, CERT, CERTREQ, SAi, TS |

Table 2.3: Critical IKE_AUTH payloads

### 2.1.4 CREATE_CHILD_SA

The *SK* used to protect the CREATE_CHILD_SA exchange is the same as described in section 2.1.3. The *SK* is created by the TKM but handed to the untrusted part to protect the IKE exchanges from outside attackers. Attackers which have taken over the untrusted part are already able to extract all information protected by these keys.

| | | | | | |
|---|---|---|---|---|---|
| 5 | A | → | B | : | *HDR, SK {SA, Ni, [KEi], TSi, TSr}* |
| 6 | B | → | A | : | *HDR, SK {SA, Nr, [KEr], TSi, TSr}* |

The *SA* payloads used to negotiate the algorithms of the child SA are again not considered critical and can be configured directly in the untrusted part. The *TS* payloads specify the IPsec SA endpoints and are also uncritical given that the TCB maintains and enforces its own policy before installing a new child SA.

Depending on the perfect forward secrecy (PFS)[1] configuration of the connection, the CREATE_CHILD_SA request may optionally contain a *KE* payload for an additional Diffie-Hellman exchange to enable stronger guarantees of forward secrecy for the child SA. The keying material for the child SA is a function of the SK_d key created along the SK_e and SK_a keys during the establishment of the IKE SA, the nonces exchanged during the CREATE_CHILD_SA exchange, and this public Diffie-Hellman value, if present ([16], section 1.3).

Payloads created by the responder can not be controlled but the algorithms selected from *SA* and the traffic selectors selected from *TS* must be checked by the TKM.

| | |
|---|---|
| *Created by TKM* | SK, Ni, [KEi] |
| *Enforced by TKM* | SAi, TS |

Table 2.4: Critical CREATE_CHILD_SA payloads

## 2.2 Code analysis

This section illustrates the charon source code, which processes the IKEv2 message exchanges and the security relevant data as described by the previous section. Graphs are used to illustrate the code flow of a specific functionality inside the strongSwan architecture.

[1]PFS ensures that a session key derived from a long-term key will not be compromised if the long-term key is disclosed in the future.

### 2.2.1 IKE_SA_INIT

Figure 2.1 shows the code involved in the IKE SA establishment. The exchange involves an initiator and a responder which are displayed in separate blocks in the graph. During IKE_SA_INIT, two messages are exchanged which are indicated between the initiator and responder code blocks. Round labels, e.g. the label *(CD)*, are references to sub-graphs which illustrate a continuative process in detail.

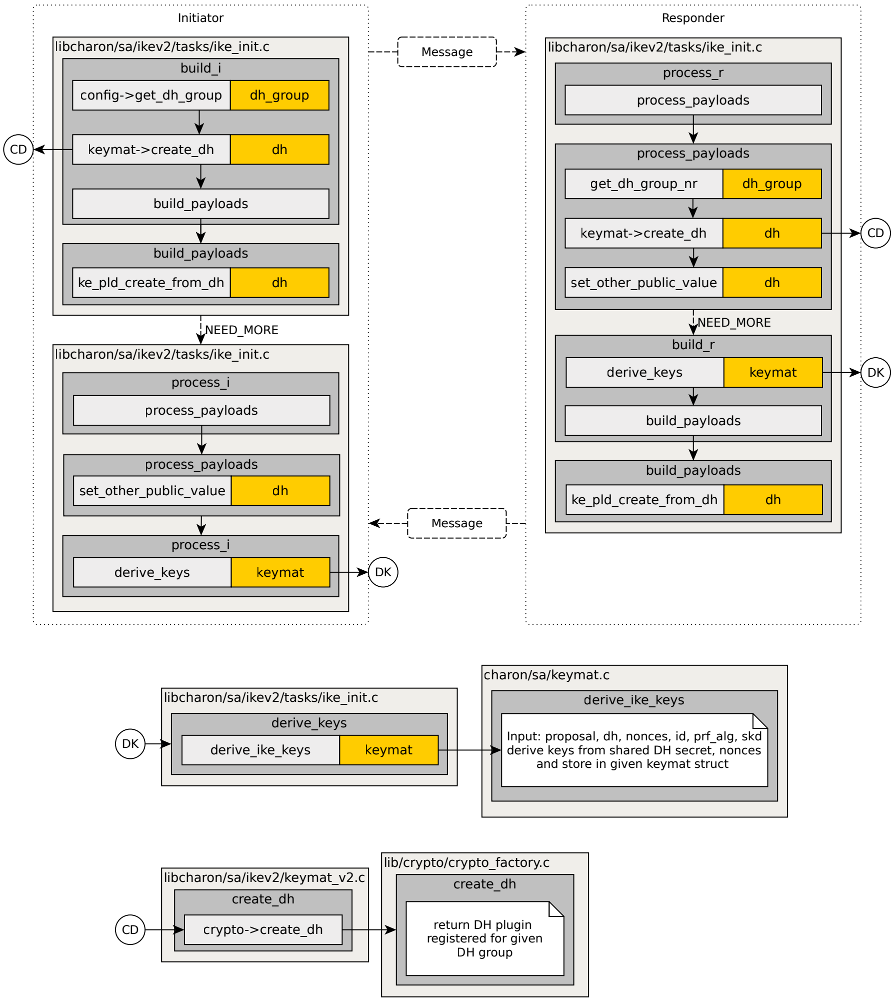


Figure 2.1: IKE SA establishment

IKE exchanges are implemented as task entities in charon and are situated in the `libcharon/sa/ikev2/tasks` directory. The IKE SA establishment process is implemented in the `ike_init.c` file in this directory. Each task represents a finite-state machine (FSM) which changes state depending on internal or external events like sent or received messages. The `NEED_MORE` state displayed in figure 2.1 indicates that the state machine responsible to establish an IKE SA is expecting more data to proceed. This state is used to separate the sending path from the receiving path inside the `build_i/process_i` and `process_r/build_r` blocks.

The tasks access required functionality by requesting plugins from different factories. Examples of such plugins are RNGs[2] or plugins which perform a DH exchange.

The initiator creates the payloads of the initial message in the `build_i` code block during which the initial steps of the Diffie-Hellman protocol are performed. The task calls the `create_dh` function of the `keymat` object (*CD*) which internally requests a new DH plugin instance from the crypto factory and returns this instance to the calling task. A `keymat` object stores the complete IKE SA key material and is used to derive IKE and child SA keys. A `keymat` object is always associated with an IKE SA inside the IKE SA manager.

After constructing all payloads, the initiator sends the IKE_SA_INIT message to the peer and waits for a response (error handling if the peer is not answering is omitted from this discussion). The responder processes the request in the `process_r` code block and performs the DH protocol on his side. Since it already received the DH public value from the initiator, it is able to complete the DH exchange without waiting for further data. It then uses the SKEYSEED from the DH exchange to derive the IKE SA keying material (*DK*) and creates an IKE_SA_INIT response containing its DH public value to allow the initiator to complete the initial exchange on his side.

The initiator then also derives IKE SA keying material used to protect the following IKE_AUTH or CHILD_CREATE_SA exchanges (*DK*). This completes phase 1.

### 2.2.2 IKE_AUTH

Figures 2.2, 2.3 and 2.4 show the code involved during the authentication of an IKE SA. As can be deduced from the number of graphs needed to illustrate the process, this exchange is more complex than the IKE_SA_INIT exchange explained in the previous section.

The initiator begins the exchange by building its own AUTH payload used to prove its identity to the responder. This is done by creating a so called *authenticator* plugin (see the *CB* label). After that, the authenticator's `build` function illustrated by the *BA* sub-graph shown in figure 2.3 is called. To construct the signed authentication octets the authenticator plugin requests a private key (*GP*) matching a specific certificate configured for this connection. The returned private key is used to sign the AUTH octets requested from the keymat object (*A8*) of the associated IKE SA. The private key is implemented as a plugin.

The initiator then sends a message containing the constructed payloads to the responder and waits for a response message.

The responder creates a *verifier* plugin to check the AUTH payload extracted from the initiator's message. The creation of a verifier plugin is depicted in the *CV* graph. The responder processes the authentication octets of the initiator by calling the verifier's `process` function (*PA*). The authenticator requests the AUTH octets from the IKE SA keymat (*A8*) and retrieves the associated public

[2] Random number generator

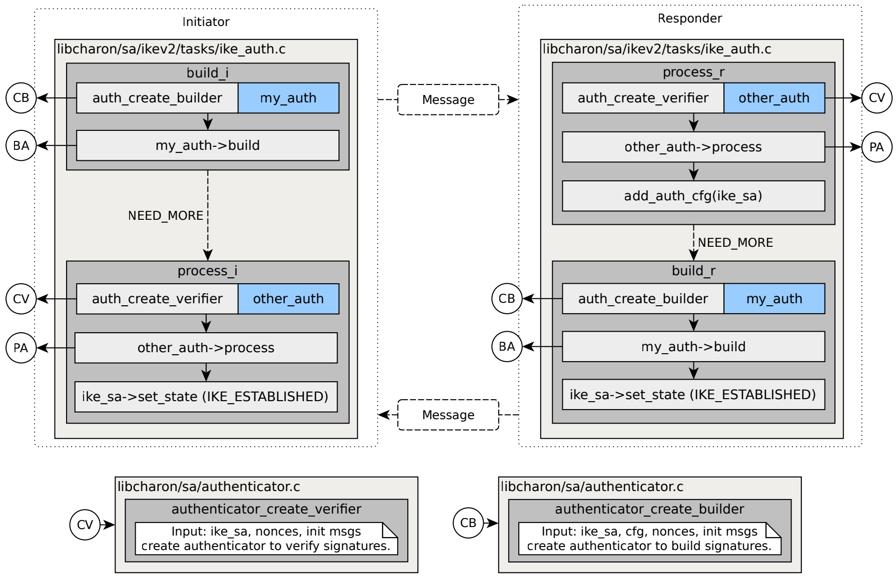


Figure 2.2: IKE SA authentication

key needed to verify the signature from the credential manager. To use the public key, its chain of trust must be verified first.

The trust chain verification process is shown in (*PU*) of figure 2.4. The credential manager verifies the signature chain of all involved certificates starting from the peer's public key until it reaches a trusted CA certificate. The details of how such signature chains are verified is explained in the implementation section 5.5.7.3.

To create the response message, the responder performs the same steps as the initiator to create its AUTH payload (*CB*, *BA*). The initiator verifies the AUTH payload of the responder using the same steps as described for the responder (*CV*, *PA*).

After the IKE SA is established, both peers normally install the first child SA.

### 2.2.3 CHILD_CREATE_SA

The CHILD_CREATE_SA exchange is implemented as a task in the `child_create.c` file and depicted in figure 2.5 on page 22. The initiator starts by collecting the traffic selectors and proposals from the configuration (not visible in the graph) and allocates a SPI by calling the `allocate_spi` function. This function dispatches into the registered kernel plugin to acquire a free SPI from the OS kernel. If the connection has PFS enabled, the initiator starts a new DH exchange and builds all required payloads. After sending the message, the task changes its state to NEED_MORE and waits for an answer.

The responder processes the received CHILD_CREATE_SA message and extracts the contained payloads. It conducts the DH exchange and then directly installs the derived child SA keying material in the kernel. The complete process

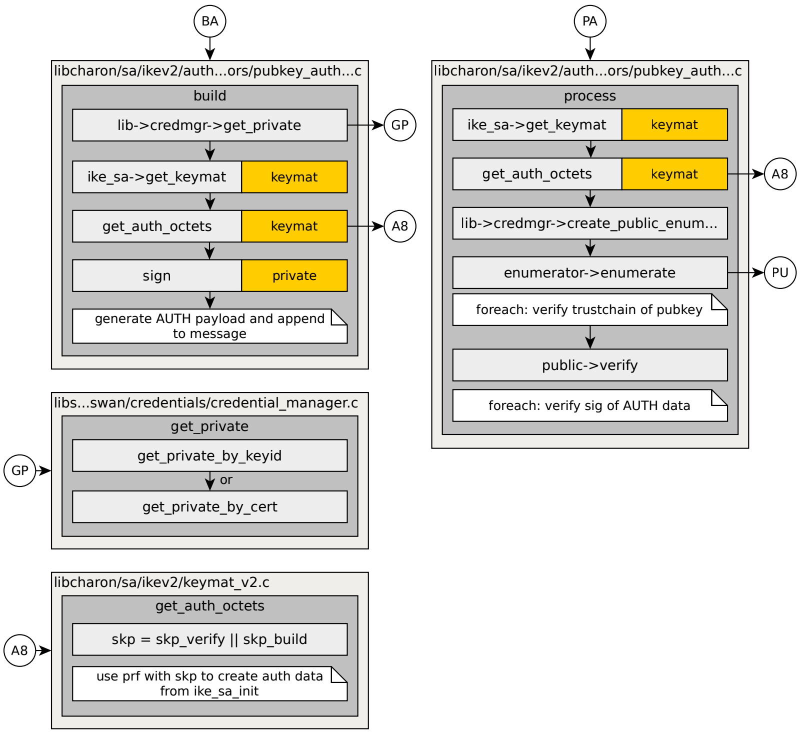


Figure 2.3: IKE public key authenticators

of deriving keys for the new child SA is depicted in *(SI)*.

First the child SA data structure associated with the task is set into the CHILD_INSTALLING state. The `derive_child_keys` function of the keymat is called to derive keying material for the child SA (*DC*). The kernel plugin `add_policy` (*IP*) and `add_sa` (*IS*) functions are used to install the new policy and state into the kernel's SPD and SAD databases. If no errors occurred, the state of the child SA is set to CHILD_INSTALLED and it is attached to the associated IKE SA object.

The responder then builds the payloads of the response message and sends the message back to the initiator. The initiator processes the message and calls the `select_and_install` function to derive child keying material after extracting the payloads. It then installs the new policy and state in the kernel.

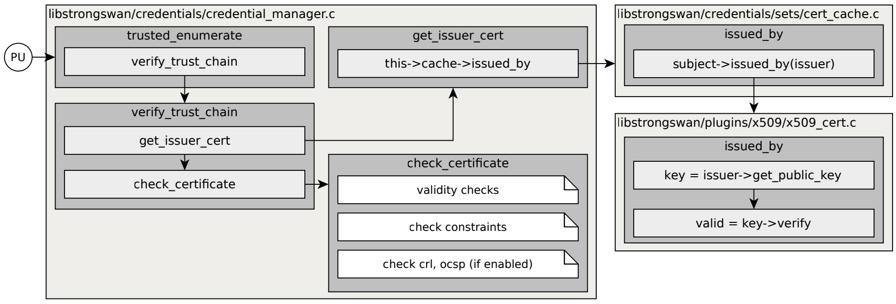


Figure 2.4: IKE Certificate trust chain verification

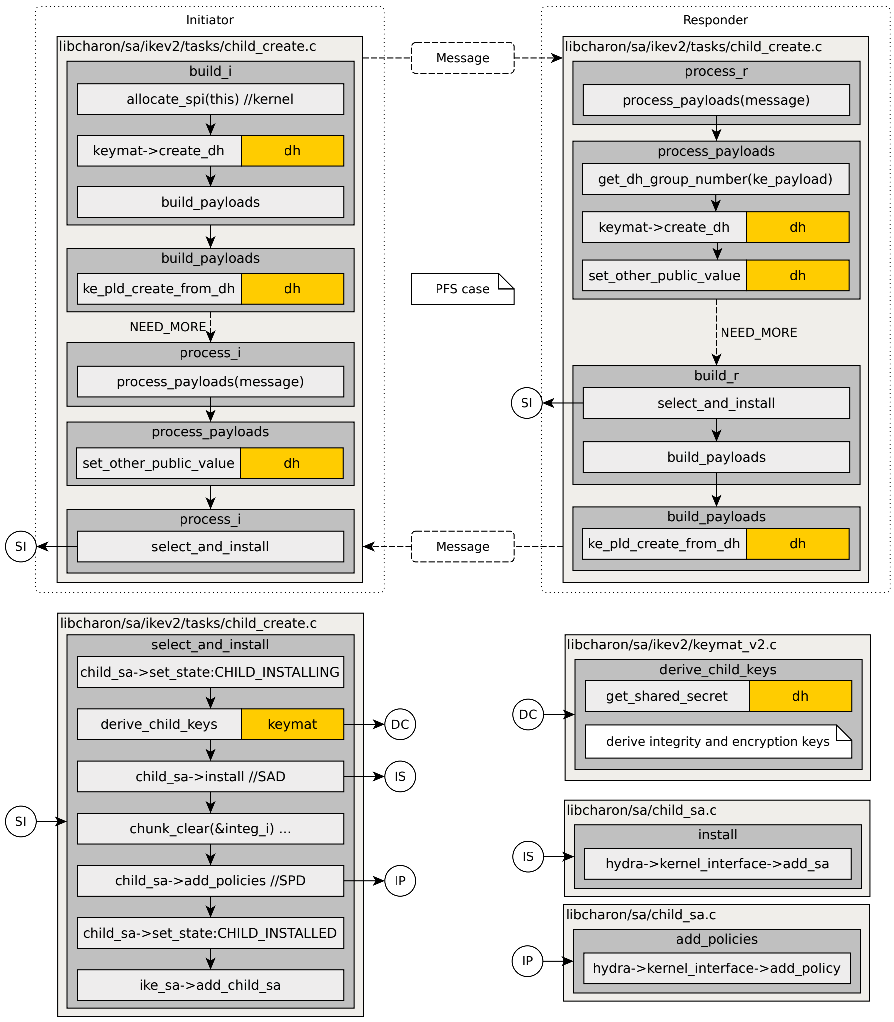


Figure 2.5: Child SA establishment

### 2.2.4 Source of randomness

Randomness is provided by requesting a random number generator plugin instance from the crypto factory of libstrongswan. This process is shown in figure 2.6, by using the nonce creation process as an example. Depending on the requested quality (`RNG_WEAK` or `RNG_STRONG`), a suitable RNG plugin providing the needed quality is created and returned to the caller by the crypto factory. The `get_bytes` or `allocate_bytes` functions can be used to retrieve random chunks from the RNG plugin.

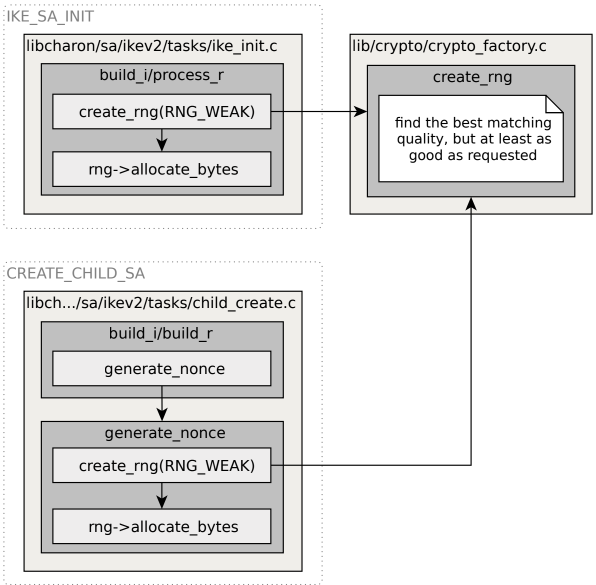


Figure 2.6: Nonce generation

### 2.2.5 Payload encryption

Figure 2.7 schematically shows the code involved in the encryption of payloads in the IKE message exchanges. If a new connection is initiated by calling the initiate function of the IKE SA, all tasks required to establish an IKE SA and the associated child SA are created and run by the task manager. The tasks then call back the IKE SA `generate_message` function to create the appropriate message sent to the peer in their exchange.

The `generate_message` function calls the `generate` function of the message which in turn checks if the message is required to be encrypted. If encryption is enabled, an encrypted payload is created by accessing the key material of the IKE SA's keymat object. The actual encryption is done by a crypter plugin which in turn uses a RNG plugin to retrieve random bytes needed for the IV[3]. The yellow "aead" blocks in figure 2.7 depict cryptographic algorithms using

[3] Initialization vector

the Authenticated Encryption with Associated Data (AEAD) mechanism to guarantee confidentiality and integrity of the IKE message payloads (see RFC 5116 [21] for details on AEAD).

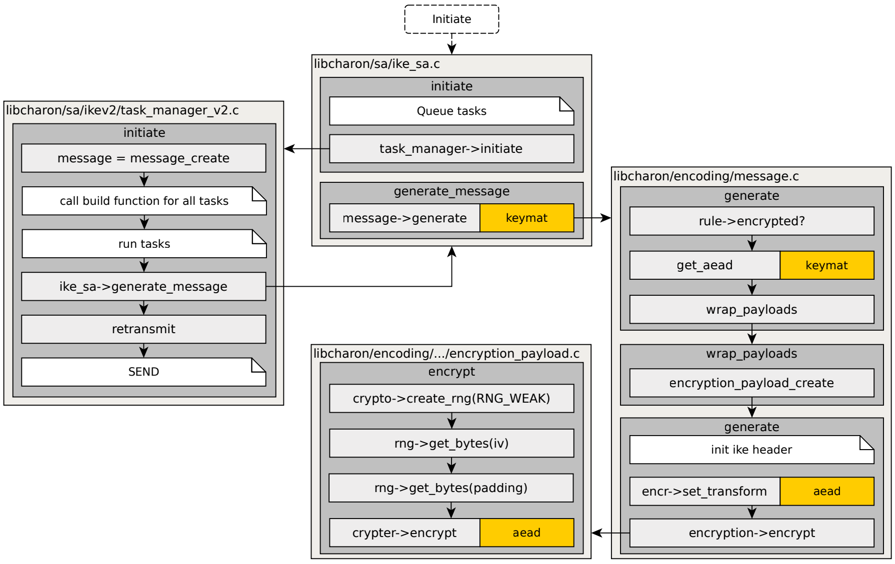


Figure 2.7: IKE SA payload encryption

### 2.2.6 Payload decryption

Figure 2.8 shows the process of payload decryption which reverses the process of payload encryption presented in chapter 2.2.5. An incoming message is processed by calling the task managers `process_message` function. This function parses the message by calling the message `parse_body` function with the keymat object from the IKE SA as function argument.

The `parse_body` function calls `decrypt_payloads`, which determines if the payloads are encrypted or not. If they are, it decrypts them by using an encryption payload object which uses the keymat's keying material to decrypt and verify the payloads.

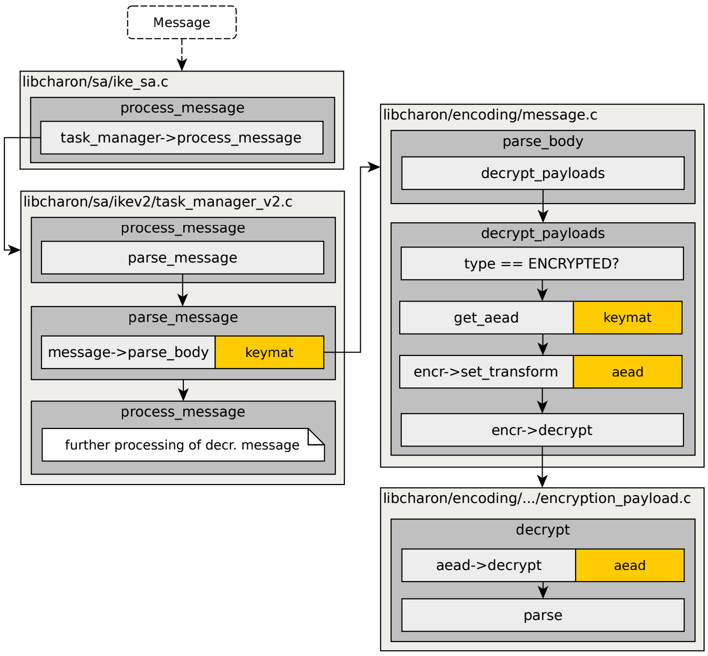


Figure 2.8: IKE SA payload decryption

# Chapter 3

# Design

The main concept is to separate the security relevant functionality from all other IKEv2 services and split the IKEv2 key management daemon into two components: a trusted and an untrusted part. The trusted part performs the critical operations, stores all relevant keying material and exposes the necessary services to the untrusted component via a well defined and minimal interface. The split of the components must guarantee the fulfillment of the security requirements defined in section 3.5.

## 3.1 Threat model

An example system separated in a trusted and untrusted component is shown in figure 1.1. This section describes the threat model used during the development of this project.

It is assumed that the strongSwan charon IKEv2 daemon, which is considered an untrusted software component in the envisioned architecture, is potentially under total control of the attacker. This means the attacker has complete access to all data available to the IKEv2 daemon and is able to execute arbitrary code with the privileges of charon. As a result of this assumption, charon is must not have access to any sensitive data. Also, intermediate computation results which are needed to create sensitive values must be protected from access by untrusted components. The following list summarizes the capabilities of an attacker:

1. The attacker is able to analyze all network traffic of the system.
2. The attacker is able to compromise the untrusted IKE daemon and read all its memory.
3. The attacker can execute arbitrary code in the untrusted component with the privileges of the IKE daemon.
4. As a result of point 2, the attacker is in possession of all data known to the IKE daemon.
5. The attacker can send arbitrary commands to the TCB (deduced from point 4).

## 3.2 TCB security properties

Even if an attacker manages to take complete control of the untrusted part of the system as described by the threat model, the TCB must guarantee the following properties:

1. The attacker has no access to the IPsec SA keying material.
2. The attacker has no means to draw conclusions about the IPsec SA keying material from sensitive intermediate values.
3. The attacker is therefore unable to decrypt recorded ESP traffic of a communication session.
4. The attacker is not able to forge authentication exchanges with unauthorized peers.
5. As a conclusion from point4, the attacker is not able to derive child keying material for an unauthorized connection.
6. The attack can only install IPsec connections, which conform to the security policy.

## 3.3 Assumptions

- The TCB security properties stated in the previous section 3.2 can only be guaranteed if the separation of the components itself withstands an attack, i.e. an attacker is unable to subvert the TCB in any way. In this project it is assumed that the separation mechanism in use is designed as such that this requirement holds. Possible solutions to this problem are elaborated in section 6.2.8.
- The untrusted IKE daemon and the trusted component can only exchange messages via the well defined interface and are otherwise completely isolated from each other. In a real system this is very difficult to achieve since there are many possibilities for side channels, which have been demonstrated to work, see for example [1, 4, 27].
- Denial-of-Service attacks (DoS) are not considered security critical because an attacker taking over the untrusted part and making all communication with the TCB impossible is still unable to access sensitive material.

## 3.4 Split of IKE

The charon software design is based on a plugin architecture. Almost every functional part of the daemon is implemented as a plugin. This provides the flexibility to extend or exchange specific parts of the system by providing a suitable plugin implementation. As outlined in the code analysis section 2.2, most security critical operations and values are already encapsulated in plugins. The changes needed to allow complete separation of the critical parts from the charon daemon are limited. Therefore, the architecture depicted in figure 3.1 is proposed.

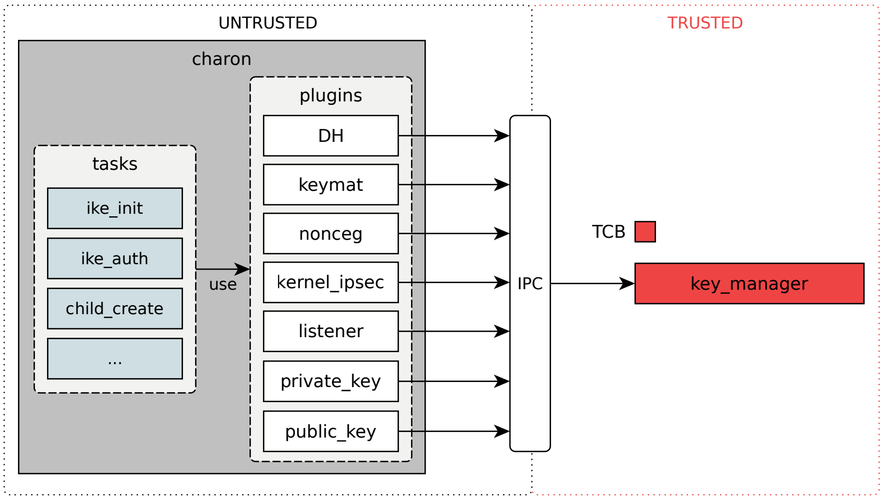


Figure 3.1: Split of IKE into trusted and untrusted parts

By implementing custom plugins which act as proxy between the trusted and untrusted parts of the component, it is possible to move the key material and related operations into the TCB. This ensures that the untrusted part has no direct access to security relevant data. The critical parts extracted from charon are implemented by the Trusted Key Manager which is part of the TCB.

### 3.4.1 Contexts and identifiers

By using a well-defined interface, the internal functionality of the TCB's key manager is completely hidden from the charon plugins. The plugins reference the data (and their associated state) needed for processing via context identifiers (IDs). These identifiers are positive integers and can be interpreted as index values into an array of contexts stored in the TCB.

This mechanism enables plugins to instruct the key manager to perform actions on specific contexts without needing access to the actual data. Only uncritical results of operations are returned to the caller plugin (e.g. the public value of a DH exchange). This architecture allows the trusted part to be minimal while the bulk of the charon code can be used as is, in the untrusted part to handle the vast majority of IKEv2 processing.

To simplify the implementation of the TCB, the management of context IDs is done in the untrusted part since it is not security-critical. The trusted part supports a limited number of contexts. These limits can be inquired from the TCB by the untrusted components by using a dedicated exchange. Usage of context IDs outside the supported numeric range is refused by the TCB.

## 3.5 Requirements

This section outlines the identified requirements of the separated system in detail. These requirements specify the properties the TCB must enforce even in the event of a complete compromise of the untrusted part of the system. The properties are derived from the threat model described in section 3.1 and the more abstract description of TCB security properties in section 3.2.

### 3.5.1 TCB robustness

The systems comprising the TCB must be robust and reliable. The TCB must be as simple as possible and at the same time small in size.

### 3.5.2 Separation

The IKEv2 component must be separated into a trusted and untrusted part in such a way that the size and complexity of the TCB are minimal.

### 3.5.3 Communication

The communication protocol between the trusted and untrusted parts must be simple, robust and well-defined to allow a verifiable implementation.

### 3.5.4 Separation of key material

The untrusted part of the IKEv2 component must not have access to generated key material that is used for authentication of peers, encryption and integrity protection of user data (i.e. child SA keys). This also includes critical intermediate values, which may result from the key agreement, generation and derivation process.

Excluded from the critical material are keys used to protect the IKE SA. As defined by the threat model, an attacker might be able to compromise the untrusted IKE daemon and read all its memory (point 2). Defeating IKE message decryption by protecting the IKE SA keys is unnecessary since the attacker is already in possession of all data these keys are intended to protect.

This implies that the procedure used to create the IKE SA keys must be cryptographically secure and exhibit the properties of a one-way function to prevent the deduction of the underlying shared secret from the keying material.

### 3.5.5 Cryptographic operations

All relevant cryptographic operations must be performed by the trusted computing base (TCB) to assure the correctness of the resulting values. Since cryptographic operations require keying material, this is also a consequence of the requirement specified in the previous section 3.5.4.

Data and intermediate values used for cryptographic operations must follow a strict life-cycle and it must be guaranteed, that such values are not used more than once. Additionally generation of pathological cryptographic keys (e.g. 0) must be detected and their usage prevented.

### 3.5.6 Authentication

The IKEv2 component must only allow IPsec SAs to be established for peers that have successfully been authenticated. The authentication must be performed by the TCB to assure the correctness of the process and foil man-in-the-middle (MitM)[1] attacks. The authentication state in the TCB must always be unambiguously associated with the corresponding SA.

### 3.5.7 Integrity

The security of the IKEv2 component must solely depend on the correct operation of the trusted part. The security operation of the system must not be violated by a misbehaving untrusted part.

### 3.5.8 Availability

The resulting system must be freely available to guarantee broader review and allow it to be extended by other interested parties. The TKM-specific changes and plugins should be integrated into the upstream strongSwan project. Also, integration tests must be provided for the TKM use-case.

[1]MitM is a form of active eavesdropping where an attacker maintains independent connections between the victims (e.g. peer and TKM) and relaying their messages while making them believe they talk directly to each other.

# Chapter 4

# TKM interface

This chapter specifies the interface between the trusted and the untrusted parts of the system. In a first step an overview of the communication between IKE and TKM is given by describing how the main operations of IKE are achieved through the usage of the services provided by the interface. After the abstract illustration of the protocol, the data types and constants are specified. These are the building blocks of the message exchanges which are described in section 4.3.

## 4.1 Protocol overview

This section gives an overview of the main IKE operations: creation and rekeying of IKE and Child SAs. The description presents the success case and specifies which parameters are passed back and forth between IKE and the TKM using the exchanges specified in the chapter 4.3.1.

In the illustrated negotiation of SAs with the peer, IKE is assuming the role of the initiator of the exchanges. As mentioned before the responder case varies only slightly and is thus not presented here. Where necessary the exchanges provide a parameter called *"initiator"* which is used to specify whether IKE is the initiator or responder of an IKEv2 message exchange with the remote peer.

Note that child SA and ESP[1] SA are used interchangeably.

### 4.1.1 Notation

The protocol is presented as an exchange of messages between the untrusted component IKE and the Trusted Key Manager TKM. The name of the operation is displayed on the left while the communicating entities are separated by an arrow which is directed from the source to the destination. Transmitted data is specified on the right-hand side.

For some exchanges only a status code of the performed operation is returned to IKE. In such cases the response is simply omitted for the sake of brevity.

Since exchanges operate on contexts that contain data and maintain associated state, these must be referenced when performing operations. This is done

[1] Encapsulating Security Payload is part of the IPsec protocol suite and provides authenticity, integrity and confidentiality of data packets.

using context IDs. For example the transmitted parameter *nc_id* identifies the nonce context to operate on. The rationale and further explanations of context IDs are given in section 3.4.1.

To distinguish local and remote values, *_loc* and *_rem* suffixes are used.

### 4.1.2 Creation of an IKE SA

In a first step the client gets a nonce and a Diffie-Hellman public value from the TKM using the `nc_create` and `dh_create` operations:

| | | | | | |
|---|---|---|---|---|---|
| nc_create | `IKE` | → | `TKM` | : | *nc_id* |
| | `TKM` | → | `IKE` | : | *Ni* |
| dh_create | `IKE` | → | `TKM` | : | *dh_id, dh_group* |
| | `TKM` | → | `IKE` | : | *KEi* |

The IKE daemon then initiates an IKE SA exchange with the remote peer. Upon receipt of the peer's response the Diffie-Hellman shared secret can be calculated. Thus IKE issues the `dh_generate_key` operation:

| | | | | | |
|---|---|---|---|---|---|
| dh_generate_key | `IKE` | → | `TKM` | : | *dh_id, KEr* |

TKM performs the calculation and stores the DH key for future consumption. No data other than the status code of the operation is passed back to IKE.

Using the previously created nonce and Diffie-Hellman value plus the nonce (*Nr*) received from the remote peer, a new IKE SA is created:

| | | | | | |
|---|---|---|---|---|---|
| isa_create | `IKE` | → | `TKM` | : | *isa_id, ae_id, ia_id, dh_id, nc_id, Nr, init, spi_loc, spi_rem* |
| | `TKM` | → | `IKE` | : | *sk_ai, sk_ar, sk_ei, sk_er* |

The returned encryption and integrity protection keys can now be used by the IKE daemon to send encrypted and integrity protected IKEv2 messages to the remote peer. For a consideration of why these keys can be handed out by TKM to the untrusted side, please refer to section 3.5.4.

To authenticate itself to the remote peer the IKE daemon requests signed local authentication data from TKM using the `isa_sign` exchange:

| | | | | | |
|---|---|---|---|---|---|
| isa_sign | `IKE` | → | `TKM` | : | *isa_id, lc_id, init_message* |
| | `TKM` | → | `IKE` | : | *AUTH_loc* |

In possession of the necessary data and keys, the IKE_AUTH protocol step is performed with the remote peer.

Upon reception of the peer's response the IKE daemon starts to validate the certificate chain of the remote peer certificate *CERT*:

| | | | | | |
|---|---|---|---|---|---|
| cc_set_user_certificate | `IKE` | → | `TKM` | : | *cc_id, ri_id, autha_id, CERT* |

Each certificate in the chain is added by issuing the `cc_add_certificate` operation:

| | | | | | |
|---|---|---|---|---|---|
| cc_add_certificate | IKE | → | TKM | : | *cc_id, autha_id, CERT* |

Once the root of the certificate chain is reached it must be asserted that the CA is trusted. This is done using the `cc_check_ca` exchange:

| | | | | | |
|---|---|---|---|---|---|
| cc_check_ca | IKE | → | TKM | : | *cc_id, ca_id* |

After successful verification of the remote certificate, IKE can authenticate the peer:

| | | | | | |
|---|---|---|---|---|---|
| isa_auth | IKE | → | TKM | : | *isa_id, cc_id, init_message, AUTH_rem* |

As a final step the first child SA can be created issuing the `esa_create_first` exchange:

| | | | | | |
|---|---|---|---|---|---|
| esa_create_first | IKE | → | TKM | : | *esa_id, isa_id, sp_id, ea_id, esp_spi_loc, esp_spi_rem* |

With this exchange processed successfully by the TKM, IKE has established an IKE and one ESP SA which can be used to encrypt application data according to the associated security policy identified by *sp_id*.

### 4.1.3 Creation of a Child SA

Creating a child SA is quite similar to creating an IKE SA. All steps related to peer authentication can be omitted since the remote identity has already been authenticated.

To create a new child SA with perfect forward secrecy (PFS), a fresh nonce and Diffie-Hellman value must be created:

| | | | | | |
|---|---|---|---|---|---|
| nc_create | IKE | → | TKM | : | *nc_id* |
| | TKM | → | IKE | : | *Ni* |
| dh_create | IKE | → | TKM | : | *dh_id, dh_group* |
| | TKM | → | IKE | : | *KEi* |

The IKE daemon then initiates a CREATE_CHILD_SA exchange with the remote peer (see section 2.1.4). Upon receipt of the peer's response the Diffie-Hellman shared secret is calculated by issuing the `dh_generate_key` operation:

| | | | | | |
|---|---|---|---|---|---|
| dh_generate_key | IKE | → | TKM | : | *dh_id, KEr* |

TKM performs the calculation and stores the DH key for future consumption. Only the status code of the operation is passed back to IKE.

Finally the child SA can be created using the `esa_create` operation:

| | | | | | |
|---|---|---|---|---|---|
| esa_create | IKE | → | TKM | : | *esa_id, isa_id, sp_id, ea_id, dh_id, nc_id, Nr, initiator, esp_spi_loc, esp_spi_rem* |

After this final step the IKE daemon has successfully established a new child SA.

### 4.1.4 Rekeying of an IKE SA

An IKE SA is rekeyed by replacing it with a new IKE SA. For this purpose a fresh nonce and a DH public value is needed:

| | | | | | |
|---|---|---|---|---|---|
| nc_create | IKE | → | TKM | : | *nc_id* |
| | TKM | → | IKE | : | *Ni* |
| dh_create | IKE | → | TKM | : | *dh_id, dh_group* |
| | TKM | → | IKE | : | *KEi* |

The IKE daemon then initiates a CREATE_CHILD_SA exchange to rekey the existing IKE SA with the peer. Upon receipt of the peers response the Diffie-Hellman shared secret can be calculated:

| | | | | | |
|---|---|---|---|---|---|
| dh_generate_key | IKE | → | TKM | : | *dh_id, KEr* |

Rekeying of the IKE SA, identified by *parent_isa_id*, is performed using the `isa_create_child` operation:

| | | | | | |
|---|---|---|---|---|---|
| isa_create_child | IKE | → | TKM | : | *isa_id, parent_isa_id, ia_id, dh_id, nc_id, Nr, initiator, spi_loc, spi_rem* |
| | TKM | → | IKE | : | *sk_ai, sk_ar, sk_ei, sk_er* |

TKM returns the new encryption and integrity keys of the new IKE SA, which from this point on is used to exchange IKEv2 messages with the remote peer.

To effectively complete the rekeying operation, the superseded IKE SA must be reset:

| | | | | | |
|---|---|---|---|---|---|
| isa_reset | IKE | → | TKM | : | $isa\_id_{old}$ |

Note that $isa\_id_{old}$ is the same as the *parent_isa_id* used in the `isa_create_child` operation.

### 4.1.5 Rekeying of a child SA

A child SA is rekeyed by replacing it with a new child SA. In order to achieve this, the steps described in section 4.1.3 must be performed. After the new child SA has been established it must be selected to make it the active SA for ESP encryption:

| | | | | | |
|---|---|---|---|---|---|
| esa_select | IKE | → | TKM | : | *esa_id* |

The only thing left to do is to reset the old, rekeyed child SA:

| | | | | | |
|---|---|---|---|---|---|
| esa_reset | IKE | → | TKM | : | $esa\_id_{old}$ |

## 4.2 Data types and constants

This section presents the data types and constants that are used in the specification of the TKM interface. They are referenced in the description of the interface exchanges, which follows in section 4.3.

### 4.2.1 Integer types

These types are numeric integers. Their `size` is specified in bytes, which is also the amount of memory an object of such a type consumes.

Table 4.1: Integer types

| Name | Size | Description |
|---|---|---|
| `operation_type` | 8 | *This type identifies the interface operation. Each exchange has a corresponding constant of this type.* |
| `request_id_type` | 8 | *Identifier of a request which is part of an exchange. This allows communicating parties to associate corresponding request and response messages.* |
| `result_type` | 8 | *Status of a processed exchange (e.g. success or failure).* |
| `version_type` | 8 | *Version of an interface implementation* |
| `active_requests_type` | 8 | *Number of concurrently active requests* |
| `authag_id_type` | 8 | *Authentication algorithms group handle* |
| `cag_id_type` | 8 | *Certificate Authority group handle* |
| `li_id_type` | 8 | *Local identity handle* |
| `ri_id_type` | 8 | *Remote identity handle* |
| `iag_id_type` | 8 | *IKE algorithm group handle* |
| `eag_id_type` | 8 | *ESP algorithm group handle* |
| `dhag_id_type` | 8 | *Diffie-Hellman algorithm group handle* |
| `sp_id_type` | 8 | *Security Policy handle* |
| `authp_id_type` | 8 | *Authentication parameter handle* |
| `dhp_id_type` | 8 | *Diffie-Hellman parameter handle* |
| `autha_id_type` | 8 | *Authentication algorithm handle* |
| `ca_id_type` | 8 | *Certificate Authority handle* |
| `lc_id_type` | 8 | *Local certificate handle* |
| `ia_id_type` | 8 | *IKE algorithms handle* |
| `ea_id_type` | 8 | *ESP algorithms handle* |
| `dha_id_type` | 8 | *Diffie-Hellman algorithm handle* |
| `nc_id_type` | 8 | *Nonce context handle* |
| `dh_id_type` | 8 | *Diffie-Hellman context handle* |
| `cc_id_type` | 8 | *Certificate chain context handle* |
| `ae_id_type` | 8 | *Authenticated endpoint context handle* |
| `isa_id_type` | 8 | *IKE SA context handle* |
| `esa_id_type` | 8 | *ESP SA context handle* |
| `esp_enc_id_type` | 8 | *ESP encryptor handle* |
| `esp_dec_id_type` | 8 | *ESP decryptor handle* |
| `esp_map_id_type` | 8 | *ESP map entry handle* |

| | | |
|---|---|---|
| abs_time_type | 8 | *Absolute time in seconds since unix epoch* |
| rel_time_type | 8 | *Relative time in seconds* |
| duration_type | 8 | *Duration timespan in seconds* |
| counter_type | 8 | *Generic counter type* |
| pfs_flag_type | 8 | *Perfect-Forward secrecy flag* |
| cc_time_flag_type | 8 | *Certificate chain time flag* |
| expiry_flag_type | 1 | *Expiration flag* |
| auth_algorithm_type | 8 | *Authentication algorithm identifier* |
| dh_algorithm_type | 2 | *Diffie-Hellman algorithm group IDs (IANA)* |
| iprf_algorithm_type | 2 | *IKE Pseudo-random function algorithm IDs (IANA)* |
| iint_algorithm_type | 2 | *IKE Integrity algorithm IDs (IANA)* |
| ienc_algorithm_type | 2 | *IKE Encryption algorithm IDs (IANA)* |
| eprf_algorithm_type | 2 | *ESP Pseudo-random function algorithm IDs (IANA)* |
| eint_algorithm_type | 2 | *ESP Integrity algorithm IDs (IANA)* |
| eenc_algorithm_type | 2 | *ESP Encryption algorithm IDs (IANA)* |
| key_length_bits_type | 8 | *Length of cryptographic keys in bits* |
| block_length_bits_type | 8 | *Length of block in bits* |
| protocol_type | 4 | *Protocol numbers (IANA)* |
| init_type | 8 | *Initiator role flag* |
| ike_spi_type | 8 | *IKE SPI in network byte order* |
| esp_spi_type | 4 | *ESP SPI in network byte order* |
| nonce_length_type | 8 | *Length of nonce* |

### 4.2.2 Variable octet types

These types are octet sequences of variable size. `Data` is the maximum number of data bytes that can be stored in the octet sequence, while `size` is the number of bytes an object of this type occupies in memory.

Table 4.2: Variable octet sequence types

| Name | Data | Size | Description |
|---|---|---|---|
| init_message_type | 1500 | 1504 | *IKE init message* |
| certificate_type | 1500 | 1504 | *ASN.1/DER encoded X.509 certificate* |
| nonce_type | 256 | 260 | *Nonce value* |
| dh_pubvalue_type | 512 | 516 | *Diffie-Hellman public value* |
| dh_priv_type | 512 | 516 | *Diffie-Hellman private value* |
| dh_key_type | 512 | 516 | *Diffie-Hellman shared secret value* |
| key_type | 64 | 68 | *Cryptographic key* |
| identity_type | 64 | 68 | *Base type for remote and local identity* |
| signature_type | 256 | 260 | *Cryptographic signature* |
| auth_parameter_type | 1024 | 1028 | *Authentication parameter* |
| dh_parameter_type | 1024 | 1028 | *Diffie-Hellman parameter* |

### 4.2.3 Constants

The TKM interface specifies various numeric constants, which can be referenced by the IKE daemon or the TKM. All constants are typed, which restricts their range of valid values. All constants are given in hexadecimal form.

#### 4.2.3.1 result_type constants

Status of a processed exchange (e.g. success or failure).

Table 4.3: result_type constants

| Name | Hexvalue | Description |
|---|---|---|
| `OK` | 0x0000000000000000 | *Request was processed successfully* |
| `Invalid_Operation` | 0x0000000000000101 | *The requested operation is invalid* |
| `Invalid_ID` | 0x0000000000000102 | *The given identifier is invalid* |
| `Invalid_State` | 0x0000000000000103 | *TKM is in an invalid state to process the given request* |
| `Invalid_Parameter` | 0x0000000000000104 | *Invalid value given as request parameter* |
| `Random_Failure` | 0x0000000000000201 | *The random number generator is inoperable* |
| `Sign_Failure` | 0x0000000000000202 | *Signature could not be generated* |
| `Aborted` | 0x0000000000000301 | *Processing of request was aborted* |
| `Math_Error` | 0x0000000000000401 | *Mathematical computation error* |

#### 4.2.3.2 version_type constants

Version of an interface implementation

Table 4.4: version_type constants

| Name | Hexvalue | Description |
|---|---|---|
| `CFG_Version` | 0x0000000000000000 | *Version of CFG interface* |
| `EES_Version` | 0x0000000000000000 | *Version of EES interface* |
| `IKE_Version` | 0x0000000000000000 | *Version of IKE interface* |

#### 4.2.3.3 dh_algorithm_type constants

Diffie-Hellman algorithm group IDs (IANA)

Table 4.5: dh_algorithm_type constants

| Name | Hexvalue | Description |
|---|---|---|
| `Modp_3072` | 0x000000000000000f | *3072-bit MODP Group (RFC 3526, section 4)* |

| | | |
|---|---|---|
| `Modp_4096` | 0x0000000000000010 | *4096-bit MODP Group (RFC 3526, section 5)* |

#### 4.2.3.4 protocol_type constants

Protocol numbers (IANA)

Table 4.6: protocol_type constants

| **Name** | **Hexvalue** | **Description** |
|---|---|---|
| `Proto_ESP` | 0x32 | *Encap Security Payload* |
| `Proto_AH` | 0x33 | *Authentication Header* |

## 4.3 Exchanges

This section describes all exchanges of the different TKM interfaces. The interface is comprised of two service-specific parts: IKE and EES (ESP Event Service).

Communication is seen as an exchange of request and response message pairs between a client and a server. In the concrete implementation, which is presented in section 5.4, the untrusted charon daemon takes the role of the client while TKM is the server of the IKE interface. Contrary charon acts as a server of the EES interface, described in section 5.4.12, while the xfrm-proxy (see section 5.6) implements the client side.

Exchanges are identified by numeric values (`operation_type` defined in section 4.2.1) which are unique on a per-interface basis.

Requests contain an identifier (`request_id`) which is chosen by the client of an exchange. The server must set the `request_id` of the corresponding response to be identical. This enables the client to match responses to their requests and handle multiple pending exchanges with possible *out-of-order* arrival of responses.

The basic layout of a request and response object is show in figure 4.1.

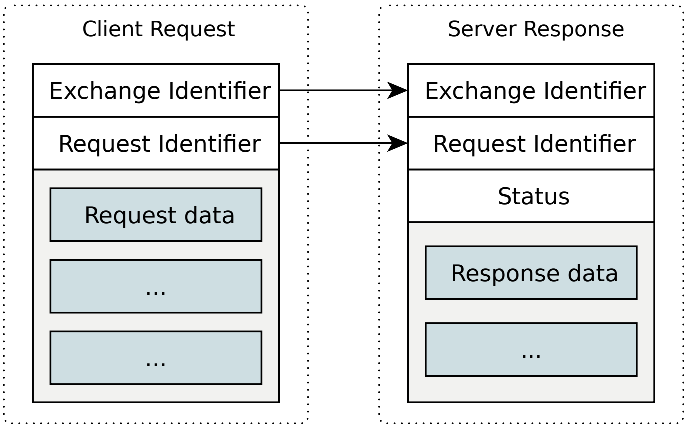


Figure 4.1: Request and response structure

### 4.3.1 IKE Exchanges

All the following exchanges are used by IKE to communicate with the TKM and perform operations related to IKE or ESP SA establishment.

#### 4.3.1.1 nc_create

Creates and returns a nonce of a given length.

**Exchange identifier** 0x0101

**Request**

Table 4.7: nc_create request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0101* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| nc_id | nc_id_type | *Handle of nonce* |
| nonce_length | nonce_length_type | *Length of nonce in bytes* |

**Response**

Table 4.8: nc_create response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0101* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |
| nonce | nonce_type | *Generated nonce* |

#### 4.3.1.2 nc_reset

Resets a NC context to its initial nc_clean state.

**Exchange identifier** 0x0100

**Request**

Table 4.9: nc_reset request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0100* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| nc_id | nc_id_type | *Handle of nonce context to reset* |

**Response**

Table 4.10: nc_reset response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0100* |
| request_id | request_id_type | *Request ID, returned identically* |

| | | |
|---|---|---|
| result | result_type | *Status code* |

#### 4.3.1.3 dh_create

Creates a new Diffie-Hellman (DH) secret value for a given algorithm and returns its public value, using the DH context specified by id.

**Exchange identifier** 0x0201

**Request**

Table 4.11: dh_create request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0201* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| dh_id | dh_id_type | *Handle of Diffie-Hellman context* |
| dha_id | dha_id_type | *Id of Diffie-Hellman algorithm/group* |

**Response**

Table 4.12: dh_create response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0201* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |
| pubvalue | dh_pubvalue_type | *Diffie-Hellman public value* |

#### 4.3.1.4 dh_generate_key

Calculate a DH shared secret based on the given remote public value and the private value stored in the specified DH context.

**Exchange identifier** 0x0202

**Request**

Table 4.13: dh_generate_key request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0202* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |

| | | |
|---|---|---|
| dh_id | dh_id_type | *Handle of Diffie-Hellman context holding private value* |
| pubvalue | dh_pubvalue_type | *Public value of remote* |

**Response**

Table 4.14: dh_generate_key response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0202* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |

#### 4.3.1.5 dh_reset

Resets a Diffie-Hellman context to its initial dh_clean state.

**Exchange identifier** 0x0200

**Request**

Table 4.15: dh_reset request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0200* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| dh_id | dh_id_type | *Handle of DH context to reset* |

**Response**

Table 4.16: dh_reset response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0200* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |

#### 4.3.1.6 cc_set_user_certificate

Sets the user certificate of a specified, clean certificate chain context. The user certificate is associated with a given remote identity and an authentication algorithm.

**Exchange identifier** 0x0301

**Request**

Table 4.17: cc_set_user_certificate request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0301* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| cc_id | cc_id_type | *Handle of certificate chain to store certificate* |
| ri_id | ri_id_type | *Handle of remote identity* |
| autha_id | autha_id_type | *Handle of authentication algorithm* |
| certificate | certificate_type | *ASN.1/DER encoded user certificate* |

**Response**

Table 4.18: cc_set_user_certificate response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0301* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |

#### 4.3.1.7 cc_add_certificate

Adds a certificate to a certificate chain context. The certificate chain remains linked if the certificate is a valid certificate representing the issuer of the previous certificate. Otherwise the chain becomes invalid. The signature of the previous certificate is verified using the specified authentication algorithm.

**Exchange identifier** 0x0302

**Request**

Table 4.19: cc_add_certificate request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0302* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| cc_id | cc_id_type | *Handle of CC context to add certificate* |
| autha_id | autha_id_type | *Id of authentication algorithm* |
| certificate | certificate_type | *ASN.1/DER encoded certificate to add to chain* |

**Response**

Table 4.20: cc_add_certificate response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0302* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |

#### 4.3.1.8 cc_check_ca

Determine whether the current root of the certificate chain stored in the identified certificate chain context is bitwise identical to the certificate of the trusted certificate authority specified by id.

**Exchange identifier** 0x0303

**Request**

Table 4.21: cc_check_ca request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0303* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| cc_id | cc_id_type | *Handle of certificate chain to check* |
| ca_id | ca_id_type | *Handle of CA to check against* |

**Response**

Table 4.22: cc_check_ca response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0303* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |

#### 4.3.1.9 cc_reset

Resets a certificate chain context to its initial cc_clean state.

**Exchange identifier** 0x0300

**Request**

Table 4.23: cc_reset request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0300* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| cc_id | cc_id_type | *Handle of certificate chain to reset* |

**Response**

Table 4.24: cc_reset response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0300* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |

#### 4.3.1.10 ae_reset

Resets an authenticated endpoint context to its initial ae_clean state. All dependent isa and esa contexts will become stale.

**Exchange identifier** 0x0800

**Request**

Table 4.25: ae_reset request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0800* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| ae_id | ae_id_type | *Handle of AE context to reset* |

**Response**

Table 4.26: ae_reset response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0800* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |

#### 4.3.1.11 isa_create

IKE uses this exchange to request derivation of IKE key material for a new IKE SA specified by isa_id. As a new authenticated endpoint is created, an ae_id has to be provided too. TKM derives keying material for the IKE SA using the shared secret stored in the DH context and the nonces performing the calculations defined in RFC 5996, sections 2.13 and 2.14. The used DH and nonce contexts are cleared after the key derivation procedure. (As the nonces are still required for isa_auth and esa_create_first, they are stored in the AE context.) nonce_rem and spi_rem are values received from the peer, spi_loc is generated by IKE. The initiator flag designates if IKE is the initiator or responder of the IKE SA. The parameter ia_id defines the algorithms to use as pseudo-random function, encryption and integrity protection of the IKE SA identified by isa_id.

**Exchange identifier** 0x0901

**Request**

Table 4.27: isa_create request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0901* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| isa_id | isa_id_type | *Handle of ISA context to create* |
| ae_id | ae_id_type | *Handle of AE context to create* |
| ia_id | ia_id_type | *Handle of IKE algorithms* |
| dh_id | dh_id_type | *Handle of DH context holding shared secret* |
| nc_loc_id | nc_id_type | *Handle of local nonce* |
| nonce_rem | nonce_type | *Nonce of peer* |
| initiator | init_type | *Flag designating initiator or responder role* |
| spi_loc | ike_spi_type | *Local IKE security policy identifier* |
| spi_rem | ike_spi_type | *Remote IKE security policy identifier* |

**Response**

Table 4.28: isa_create response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0901* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |
| sk_ai | key_type | *Integrity protection key of initiator* |
| sk_ar | key_type | *Integrity protection key of responder* |
| sk_ei | key_type | *Encryption key of initiator* |

| | | |
|---|---|---|
| sk_er | key_type | *Encryption key of responder* |

#### 4.3.1.12 isa_sign

This exchange is used by IKE to request signed authentication octets for an IKE SA identified by isa_id from TKM. TKM generates the authentication octets for the ISA context as described in RFC 5996, section 2.15 using the given IKE init message. TKM then computes the signature of the generated octets using the scheme and private key defined by the local certificate identified by lc_id.

**Exchange identifier** 0x0902

**Request**

Table 4.29: isa_sign request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0902* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| isa_id | isa_id_type | *Handle of IKE SA to sign* |
| lc_id | lc_id_type | *Handle of local identity certificate* |
| init_message | init_message_type | *IKE init message needed to create authentication octets* |

**Response**

Table 4.30: isa_sign response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0902* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |
| signature | signature_type | *Signed local authentication octets* |

#### 4.3.1.13 isa_auth

This message exchange is initiated by IKE to authenticate an IKE SA identified by isa_id. TKM reconstructs the authentication octets of the peer and verifies their signature against the (already validated) certificate of the peer, as specified in RFC 5996, section 2.15.

**Exchange identifier** 0x0903

**Request**

Table 4.31: isa_auth request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0903* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| isa_id | isa_id_type | *Handle of IKE SA to authenticate* |
| cc_id | cc_id_type | *Handle of certificate chain holding peer certificate* |
| init_message | init_message_type | *IKE init message needed to create authentication octets* |
| signature | signature_type | *Signed authentication octets from peer* |

**Response**

Table 4.32: isa_auth response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0903* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |

#### 4.3.1.14 isa_create_child

IKE uses this exchange to request derivation of IKE key material for a new IKE SA specified by isa_id in the context of an IKE SA specified by parent_isa_id. This operation can be used to rekey an existing IKE SA. TKM derives keying material for the IKE SA using the shared secret stored in the DH context and the nonces performing the calculations defined in RFC 5996, section 2.18. The used DH and nonce contexts are cleared after the key derivation procedure. nonce_rem and spi_rem are values received from the peer, spi_loc is generated by IKE. The initiator flag designates if IKE is the initiator or responder of the IKE SA. The parameter ia_id defines the algorithms to use as pseudo-random function, encryption and integrity protection of the IKE SA identified by isa_id.

**Exchange identifier** 0x0904

**Request**

Table 4.33: isa_create_child request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0904* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| isa_id | isa_id_type | *Handle of IKE SA to create* |
| parent_isa_id | isa_id_type | *Handle of parent IKE SA* |

| | | |
|---|---|---|
| ia_id | ia_id_type | *Handle of IKE algorithms* |
| dh_id | dh_id_type | *Handle of DH context holding shared secret* |
| nc_loc_id | nc_id_type | *Handle of local nonce* |
| nonce_rem | nonce_type | *Nonce of peer* |
| initiator | init_type | *Flag designating initiator or responder role* |
| spi_loc | ike_spi_type | *Local IKE security policy identifier* |
| spi_rem | ike_spi_type | *Remote IKE security policy identifier* |

**Response**

Table 4.34: isa_create_child response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0904* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |
| sk_ai | key_type | *Integrity protection key of initiator* |
| sk_ar | key_type | *Integrity protection key of responder* |
| sk_ei | key_type | *Encryption key of initiator* |
| sk_er | key_type | *Encryption key of responder* |

#### 4.3.1.15 isa_reset

Resets an IKE SA context to its initial isa_clean state.

**Exchange identifier** 0x0900

**Request**

Table 4.35: isa_reset request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0900* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| isa_id | isa_id_type | *Handle of IKE SA to reset* |

**Response**

Table 4.36: isa_reset response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0900* |
| request_id | request_id_type | *Request ID, returned identically* |

| | | |
|---|---|---|
| result | result_type | *Status code* |

#### 4.3.1.16 esa_create_first

IKE uses this exchange to activate the initial child SA for a newly authenticated IKE SA using the security policy specified by sp_id. As no explicit DH exchange is performed, the generic esa_create exchange cannot be used. The parameter ea_id defines the algorithms to use as pseudo-random function, encryption and integrity protection of the child SA. TKM derives keying material for the child SA by using the nonces and the sk_d key of the IKE SA. The key derivation algorithm is specified in RFC 5996, section 2.17.

**Exchange identifier** 0x0A03

**Request**

Table 4.37: esa_create_first request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0A03* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| esa_id | esa_id_type | *Handle of ESP SA to create* |
| isa_id | isa_id_type | *Handle of associated IKE SA* |
| sp_id | sp_id_type | *Handle of associated security policy* |
| ea_id | ea_id_type | *Id of ESP algorithms to use* |
| esp_spi_loc | esp_spi_type | *Local ESP security policy identifier* |
| esp_spi_rem | esp_spi_type | *Remote ESP security policy identifier* |

**Response**

Table 4.38: esa_create_first response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0A03* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |

#### 4.3.1.17 esa_create

IKE uses this exchange to request derivation of key material for a child SA specified by esa_id in the context of an IKE SA specified by isa_id. The security policy associated with the ESP SA is given by sp_id. The dh_id parameter specifies the DH context to use in key derivation. It must have been created using the dh_create and dh_generate exchanges. The nc_loc_id parameter specifies the nonce context to use in key derivation. It must have been created using the nc_create exchange. nonce_rem is the nonce received from peer. The initiator flag designates if IKE is the initiator or responder of the child SA. The parameter ea_id defines the algorithms to use as pseudo-random function, encryption and integrity protection of the child SA. TKM derives keying material for the child SA by using the shared secret stored in the DH context, the nonces and the sk_d key of the IKE SA. The key derivation algorithm is specified in RFC 5996, section 2.17.

**Exchange identifier** 0x0A01

**Request**

Table 4.39: esa_create request parameters

| Name | Type | Description |
|---|---|---|
| `operation` | `operation_type` | *Exchange ID: 0x0A01* |
| `request_id` | `request_id_type` | *Request ID, chosen by untrusted* |
| `esa_id` | `esa_id_type` | *Handle of ESP SA to create* |
| `isa_id` | `isa_id_type` | *Handle of associated IKE SA* |
| `sp_id` | `sp_id_type` | *Handle of associated security policy* |
| `ea_id` | `ea_id_type` | *Id of ESP algorithms to use* |
| `dh_id` | `dh_id_type` | *Handle of DH context holding shared secret* |
| `nc_loc_id` | `nc_id_type` | *Handle of local nonce* |
| `nonce_rem` | `nonce_type` | *Remote nonce of peer* |
| `initiator` | `init_type` | *Flag designating initiator or responder role* |
| `esp_spi_loc` | `esp_spi_type` | *Local ESP security policy identifier* |
| `esp_spi_rem` | `esp_spi_type` | *Remote ESP security policy identifier* |

**Response**

Table 4.40: esa_create response parameters

| Name | Type | Description |
|---|---|---|
| `operation` | `operation_type` | *Exchange ID: 0x0A01* |
| `request_id` | `request_id_type` | *Request ID, returned identically* |
| `result` | `result_type` | *Status code* |

#### 4.3.1.18 esa_create_no_pfs

IKE uses this exchange to request derivation of key material for a child SA specified by esa_id in the context of an IKE SA specified by isa_id. The security policy associated with the ESP SA is given by sp_id. The nc_loc_id parameter specifies the nonce context to use in key derivation. It must have been created using the nc_create exchange. nonce_rem is the nonce received from peer. The initiator flag designates if IKE is the initiator or responder of the child SA. The parameter ea_id defines the algorithms to use as pseudo-random function, encryption and integrity protection of the child SA. TKM derives keying material for the child SA by using the nonces and the sk_d key of the IKE SA. The key derivation algorithm is specified in RFC 5996, section 2.17.

**Exchange identifier** 0x0A02

**Request**

Table 4.41: esa_create_no_pfs request parameters

| Name | Type | Description |
|---|---|---|
| `operation` | `operation_type` | *Exchange ID: 0x0A02* |
| `request_id` | `request_id_type` | *Request ID, chosen by untrusted* |
| `esa_id` | `esa_id_type` | *Handle of ESP SA to create* |
| `isa_id` | `isa_id_type` | *Handle of associated IKE SA* |
| `sp_id` | `sp_id_type` | *Handle of associated security policy* |
| `ea_id` | `ea_id_type` | *Id of ESP algorithms to use* |
| `nc_loc_id` | `nc_id_type` | *Handle of local nonce* |
| `nonce_rem` | `nonce_type` | *Remote nonce of peer* |
| `initiator` | `init_type` | *Flag designating initiator or responder role* |
| `esp_spi_loc` | `esp_spi_type` | *Local ESP security policy identifier* |
| `esp_spi_rem` | `esp_spi_type` | *Remote ESP security policy identifier* |

**Response**

Table 4.42: esa_create_no_pfs response parameters

| Name | Type | Description |
|---|---|---|
| `operation` | `operation_type` | *Exchange ID: 0x0A02* |
| `request_id` | `request_id_type` | *Request ID, returned identically* |
| `result` | `result_type` | *Status code* |

#### 4.3.1.19 esa_select

Chooses the ESP SA identified by esa_id for outgoing traffic encryption.

**Exchange identifier** 0x0A04

**Request**

Table 4.43: esa_select request parameters

| Name | Type | Description |
|---|---|---|
| `operation` | `operation_type` | *Exchange ID: 0x0A04* |
| `request_id` | `request_id_type` | *Request ID, chosen by untrusted* |
| `esa_id` | `esa_id_type` | *Handle of ESP SA to select* |

**Response**

Table 4.44: esa_select response parameters

| Name | Type | Description |
|---|---|---|
| `operation` | `operation_type` | *Exchange ID: 0x0A04* |
| `request_id` | `request_id_type` | *Request ID, returned identically* |
| `result` | `result_type` | *Status code* |

#### 4.3.1.20 esa_reset

Resets an ESP SA context to its initial esa_clean state.

**Exchange identifier** 0x0A00

**Request**

Table 4.45: esa_reset request parameters

| Name | Type | Description |
|---|---|---|
| `operation` | `operation_type` | *Exchange ID: 0x0A00* |
| `request_id` | `request_id_type` | *Request ID, chosen by untrusted* |
| `esa_id` | `esa_id_type` | *Handle of ESP SA to reset* |

**Response**

Table 4.46: esa_reset response parameters

| Name | Type | Description |
|---|---|---|
| `operation` | `operation_type` | *Exchange ID: 0x0A00* |
| `request_id` | `request_id_type` | *Request ID, returned identically* |
| `result` | `result_type` | *Status code* |

#### 4.3.1.21 tkm_version

Returns the version of the TKM - IKE interface.

**Exchange identifier** 0x0000

**Request**

Table 4.47: tkm_version request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0000* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |

**Response**

Table 4.48: tkm_version response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0000* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |
| version | version_type | *Version of the IKE interface* |

#### 4.3.1.22 tkm_limits

Returns limits of various TKM IKE resources.

**Exchange identifier** 0x0001

**Request**

Table 4.49: tkm_limits request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0001* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |

**Response**

Table 4.50: tkm_limits response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0001* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |

| | | |
|---|---|---|
| max_active_requests | active_requests_type | *Maximum number of simultaneously active requests* |
| nc_contexts | nc_id_type | *Maximum number of nonce contexts* |
| dh_contexts | dh_id_type | *Maximum number of Diffie-Hellman contexts* |
| cc_contexts | cc_id_type | *Maximum number of certificate chain contexts* |
| ae_contexts | ae_id_type | *Maximum number of authenticated endpoint contexts* |
| isa_contexts | isa_id_type | *Maximum number of IKE SA contexts* |
| esa_contexts | esa_id_type | *Maximum number of ESP SA contexts* |

#### 4.3.1.23 tkm_reset

Reset all contexts of the TKM - IKE interface to their initial state.

**Exchange identifier** 0x0002

**Request**

Table 4.51: tkm_reset request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0002* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |

**Response**

Table 4.52: tkm_reset response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0002* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |

### 4.3.2 ESP SA Event Service (EES) Exchanges

The exchanges specified in this section are used by the xfrm-proxy to communicate with IKE. EES is used to send notifications about ESP SA events such as acquire or expire.

#### 4.3.2.1 esa_acquire

TKM uses this exchange to request the initiation of an ESP SA with associated Security Policy identified by sp_id.

**Exchange identifier** 0x0100

**Request**

Table 4.53: esa_acquire request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0100* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| sp_id | sp_id_type | *Handle of associated security policy to acquire ESP SA for* |

**Response**

Table 4.54: esa_acquire response parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0100* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |

#### 4.3.2.2 esa_expire

TKM uses this exchange to signal the expiration of an ESP SA with associated Security Policy identified by sp_id. The ESP SPI of the remote peer and the protocol number (ESP/AH) are passed as parameters, as well as a flag specifying if the SA is about to expire (soft expiry) or has expired (hard expiry).

**Exchange identifier** 0x0101

**Request**

Table 4.55: esa_expire request parameters

| Name | Type | Description |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0101* |
| request_id | request_id_type | *Request ID, chosen by untrusted* |
| sp_id | sp_id_type | *Handle of associated security policy* |
| spi_rem | esp_spi_type | *Remote ESP security policy identifier* |
| protocol | protocol_type | *Protocol of ESP SA* |

| | | |
|---|---|---|
| hard | expiry_flag_type | *Flag designating a hard or soft expiry event* |

**Response**

Table 4.56: esa_expire response parameters

| **Name** | **Type** | **Description** |
|---|---|---|
| operation | operation_type | *Exchange ID: 0x0101* |
| request_id | request_id_type | *Request ID, returned identically* |
| result | result_type | *Status code* |

## 4.4 State machines

Contexts are used to describe stateful entities within the TKM. They are finite state machines (FSM) which have a set of states and transitions between those states. The FSM is in a specific state at any given time and can only change its state by performing a transition. A transition prescribes the source state the FSM has to be in, the actions to execute and the new target state once the transition has completed.

The state machine transitions to a known failure state if an error occurs. To recover from such an error the FSM has to be reinitialized by explicitly performing a reset operation.

The state of the overall TKM system can be interpreted as the sum of the states of all FSMs and their associated data at any given time.

### 4.4.1 Notation

All states of an FSMs are listed by name and giving a short description of the state. The initial state of the state machine is marked with a *.

Transitions are given by their name, source and target states and a description explaining the actions performed during when transitioning. If a transition can be performed from multiple states, all of them are listed in the source field. A * symbol as source means that the transition can be executed from any state.

Additionally each state machine is depicted by a diagram. Transitions are drawn as directed arrows from the source to the target state with a label identifying the name of the transition.

*Reset* and *error* transitions are treated differently in order to create less cluttered graphs. These two transitions can be triggered from any state so their labels are omitted and their arrows have different styles. Reset transitions are shown using blue lines and error transitions are marked with red dashed lines.

### 4.4.2 Nonce Context (nc)

Nonce contexts provide random nonces of a specified length. In order to prevent uncontrolled reuse of values, nonce contexts are destroyed whenever a nonce context is used within TKM.

#### 4.4.2.1 States

Table 4.57: Nonce Context States

| Name | Description |
|---|---|
| `clean*` | *No nonce is present.* |
| `invalid` | *Due to an error the context is erased. It can only be reused after explicitly reseting of the context.* |
| `created` | *A nonce is available for consumption.* |

#### 4.4.2.2 Transitions

Table 4.58: Nonce Context Transitions

| Name | Source | Target | Description |
|---|---|---|---|
| `create` | clean | created | *Create new nonce.* |
| `consume` | created | clean | *Consume nonce.* |
| `invalidate` | * | invalid | *Invalidate nonce context; it can only be reused by explicitly resetting the context.* |
| `reset` | * | clean | *Reset nonce context to initial clean state.* |

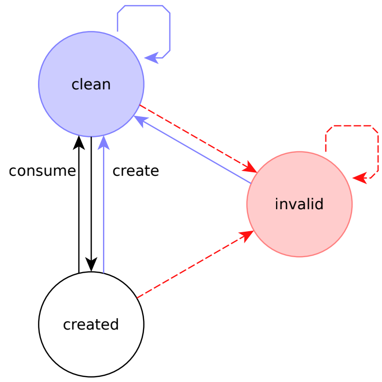


Figure 4.2: Nonce context state machine

### 4.4.3 Diffie-Hellman Context (dh)

A Diffie-Hellman context represents a Diffie-Hellman exchange with the peer.

#### 4.4.3.1 States

Table 4.59: Diffie-Hellman Context States

| Name | Description |
|---|---|
| `clean*` | *Initial clean state.* |
| `invalid` | *Error state.* |
| `created` | *Waiting for remote pubvalue.* |
| `generated` | *Diffie-Hellman shared secret has been calculated and is ready to be used.* |

#### 4.4.3.2 Transitions

Table 4.60: Diffie-Hellman Context Transitions

| Name | Source | Target | Description |
|---|---|---|---|
| `get_dha_id` | created | created | *Return DHA reference.* |
| `get_secvalue` | created | created | *Return local Diffie-Hellman secret value.* |
| `consume` | generated | clean | *Use Diffie-Hellman shared secret thus consuming it.* |
| `create` | clean | created | *Create new Diffie-Hellman context.* |
| `generate` | created | generated | *Generate the shared Diffie-Hellman secret.* |
| `invalidate` | * | invalid | *Invalidate Diffie-Hellman context; it can only be reused by explicitly resetting the context.* |
| `reset` | * | clean | *Reset Diffie-Hellman context to initial clean state.* |

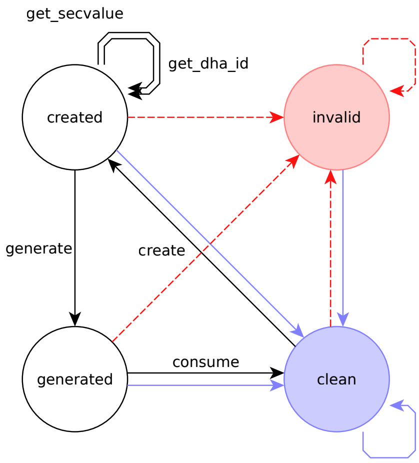


Figure 4.3: Diffie-Hellman context state machine

### 4.4.4 Certificate Chain Context (cc)

A certificate chain context is used to verify the trustchain of a user certificate by checking each certificate signature and asserting that the chain is attested by a trusted certificate authority.

#### 4.4.4.1 States

Table 4.61: Certificate Chain Context States

| Name | Description |
|---|---|
| `clean*` | *Initial clean state.* |
| `invalid` | *Error state.* |
| `linked` | *CC is linked.* |
| `checked` | *CC has been checked and verified.* |

#### 4.4.4.2 Transitions

Table 4.62: Certificate Chain Context Transitions

| Name | Source | Target | Description |
|---|---|---|---|
| `create` | clean | linked | *Create new certificate chain.* |
| `add_certificate` | linked | linked | *Add new certificate to the certificate chain.* |

| | | | |
|---|---|---|---|
| `check` | linked | checked | *Check that the current root of the CC is a trusted CA certificate.* |
| `get_last_cert` | linked | linked | *Return the last certificate which is the current root of the CC.* |
| `get_certificate` | checked | checked | *Return user certificate.* |
| `get_not_before` | linked | linked | *Return start of validity period.* |
| `get_not_after` | linked | linked | *Return end of validity period.* |
| `invalidate` | * | invalid | *Invalidate certificate chain; it can only be reused by explicitly resetting the context.* |
| `reset` | * | clean | *Reset certificate chain to initial clean state.* |

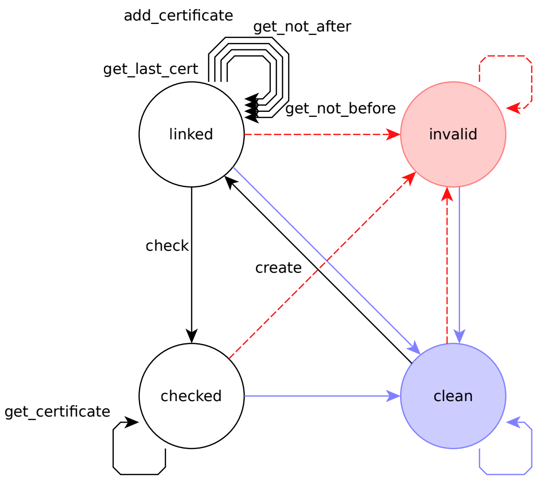


Figure 4.4: Certificate chain context state machine

### 4.4.5 Authentication Endpoint Context (ae)

Authenticated endpoints represent peers of IKE connections. Multiple IKE SAs can be established to the same authenticated endpoint.

#### 4.4.5.1 States

Table 4.63: Authentication Endpoint Context States

| **Name** | **Description** |
|---|---|
| `clean*` | *Initial clean state.* |
| `invalid` | *Error state.* |
| `unauth` | *AE context is unauthenticated.* |
| `loc_auth` | *Local identity of AE is authenticated.* |
| `authenticated` | *AE is authenticated.* |
| `active` | *AE is authenticated and in use.* |

#### 4.4.5.2 Transitions

Table 4.64: Authentication Endpoint Context Transitions

| **Name** | **Source** | **Target** | **Description** |
|---|---|---|---|
| `create` | clean | unauth | *Create new authenticated endpoint.* |
| `sign` | unauth | loc_auth | *Sign local authentication octets.* |
| `authenticate` | loc_auth | authenticated | *Verify remote authentication octets.* |
| `activate` | authenticated | active | *Use authenticated endpoint for IKE SA.* |
| `is_initiator` | authenticated | authenticated | *Return local initiator role of authenticated endpoint.* |
| `get_nonce_rem` | authenticated unauth | authenticated | *Return nonce of remote peer.* |
| `get_nonce_loc` | authenticated loc_auth | authenticated | *Return local nonce.* |
| `get_sk_ike_auth_loc` | unauth | unauth | *Return local SK_p value.* |
| `get_sk_ike_auth_rem` | loc_auth | loc_auth | *Return remote SK_p value.* |
| `reset` | * | clean | *Reset authenticated endpoint to initial clean state.* |

| | | | |
|---|---|---|---|
| invalidate | * | invalid | *Invalidate authenticated endpoint; it can only be reused by explicitly re-setting the context.* |

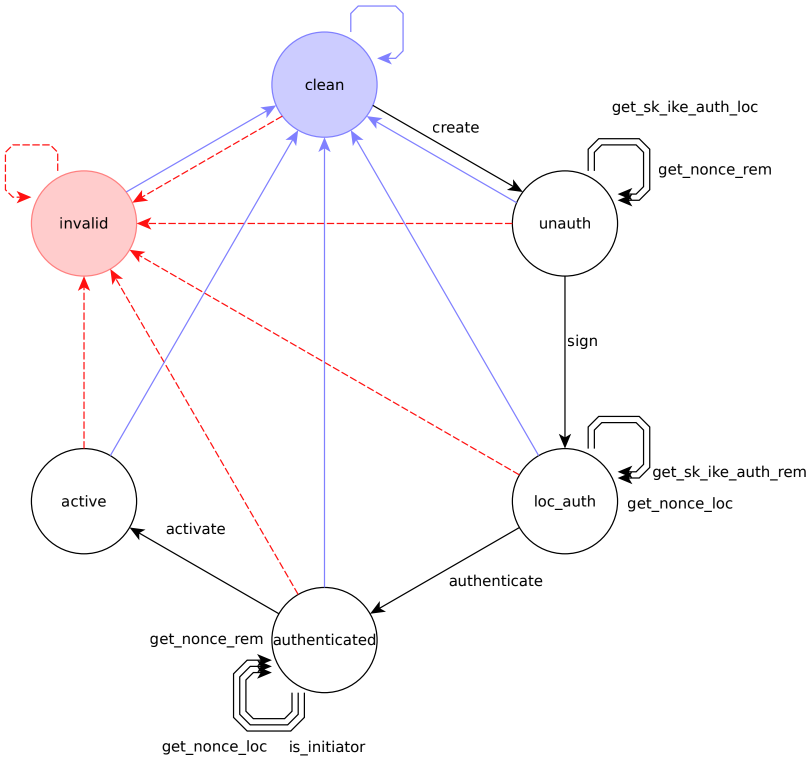


Figure 4.5: Authenticated endpoint context state machine

### 4.4.6 IKE SA Context (isa)

#### 4.4.6.1 States

Table 4.65: IKE SA Context States

| Name | Description |
|---|---|

| | |
|---|---|
| `clean*` | *Initial clean state* |
| `invalid` | *Error state* |
| `active` | *IKE SA is in active use.* |

#### 4.4.6.2 Transitions

Table 4.66: IKE SA Context Transitions

| Name | Source | Target | Description |
|---|---|---|---|
| `create` | clean | active | *Create new IKE SA.* |
| `get_sk_d` | active | active | *Return SK_D value.* |
| `get_ae_id` | active | active | *Return AE reference.* |
| `reset` | * | clean | *Reset IKE SA to initial clean state.* |
| `invalidate` | * | invalid | *Invalidate IKE SA; it can only be reused by explicitly resetting the context.* |

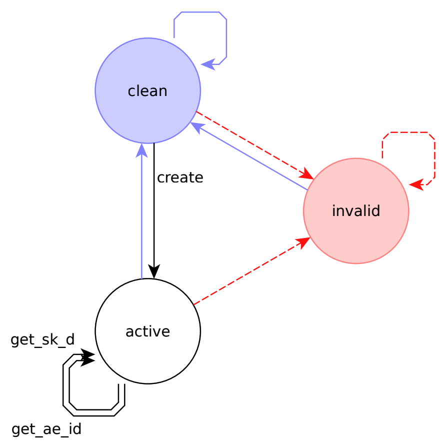


Figure 4.6: IKE SA context state machine

### 4.4.7 ESP SA Context (esa)

#### 4.4.7.1 States

Table 4.67: ESP SA Context States

| Name | Description |
|---|---|

| | |
|---|---|
| `clean*` | *Initial clean state.* |
| `invalid` | *Error state.* |
| `selected` | *ESA is selected.* |
| `active` | *ESP SA is active.* |

#### 4.4.7.2 Transitions

Table 4.68: ESP SA Context Transitions

| **Name** | **Source** | **Target** | **Description** |
|---|---|---|---|
| `create` | clean | active | *Create a new ESP SA.* |
| `select_sa` | active | selected | *Select an ESP SA to be used for outgoing traffic matching the associated security policy.* |
| `unselect_sa` | selected | active | *Unselect an ESP SA so it is not used for encryption of outgoing traffic matching the associated security policy.* |
| `invalidate` | * | invalid | *Invalidate ESP SA; it can only be reused by explicitly resetting the context.* |
| `reset` | * | clean | *Reset ESP SA to initial clean state.* |

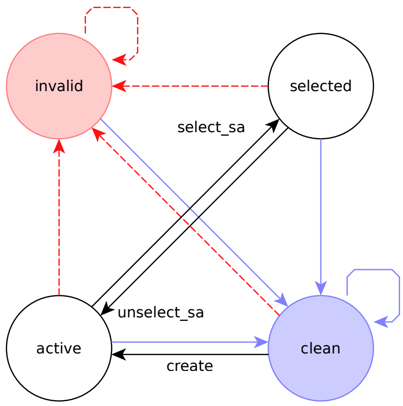


Figure 4.7: ESP SA context state machine

# Chapter 5

# Implementation

This chapter describes the implementation of the design outlined in chapter 3. The first section gives a high-level overview of the system and introduces the different components, what their purpose is and how they interact.

The next section briefly presents how the interface is described in XML, how that specification is transformed into various formats and what parts of the system are generated based on that specification. The next section then describes the remote procedure call (RPC) library which is used by various components for communication.

Section 5.4 gives an in-depth characterization of the changes to the strongSwan project, the newly implemented plugins and how the integration of Ada code into the existing project is realized. Following that the Trusted Key Manager implementation is presented.

The new component xfrm-proxy which provides ESP SA events to charon-tkm is described in section 5.6. Additional libraries that are used by either TKM or xfrm-proxy are illustrated in section 5.7. Finally limitations of the current implementation with regards to the design are listed in section 5.8 and the implementation is examined if and to what degree it meets the requirements laid out in section 3.5.

## 5.1 System Overview

The system is comprised of three distinct components:

1. charon
2. key_manager
3. xfrm-proxy

Charon provides the non-critical IKE protocol handling and is implemented by leveraging the existing strongSwan IKEv2 implementation. IKE messages with a remote peer are exchanged using a network socket. It uses the Trusted Key Manager to perform sensitive operations, such as generating key material or authenticating a remote peer. The two components communicate via the TKM interface presented in the previous chapter. The interface messages are exchanged via Unix sockets, which are abstracted by the tkm-rpc library.

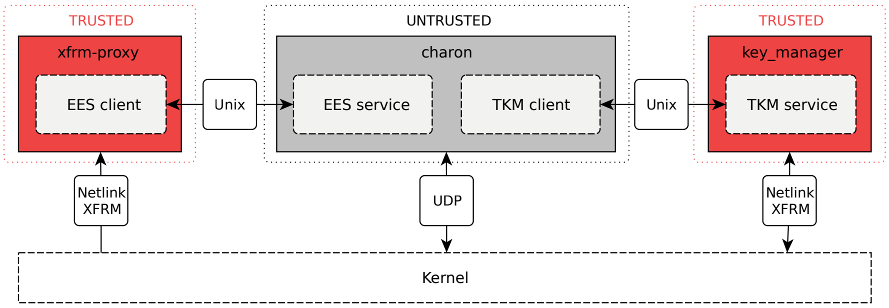


Figure 5.1: System overview

The TKM uses a Netlink/XFRM socket to install security policies and key material of an IPsec SA in the kernel.

Similarly to the communication between the charon and TKM, the trusted xfrm-proxy communicates with the charon daemon using an Unix domain socket. The xfrm-proxy handles acquire and expire events for IPsec SAs, sent by the kernel via a Netlink/XFRM socket. These events are then propagated to the charon daemon for processing.

All components and their implementation are described in detail in the rest of this chapter.

## 5.2 XML specification

The interface specification, which is the basis of the communication of system components, is done in XML. Extensible stylesheet language transformations (XSLT[1]) are used to generate many different representations of the XML document.

Automatically generating code and documentation from a single XML source assures that the created documents are always in sync and there is no mismatch between the implementation and the specification. The cost of interface change and extension is lowered considerably since the generation process is automated and no manual steps are necessary. Figure 5.2 shows the process of applying the XSL transformations to the specification and the various generated outputs.

An interesting example of such a transformation is the generation of the Ada context state machine code. Leveraging the newly added contract feature of Ada 2012, the transitions of a context state machine are translated into pre- and postconditions. Listing 5.1 shows the specification of the `nc_create` transition as an example.

The generated Ada code is shown in listing 5.2. The preconditions state that the nonce context with the given ID must be in the "clean" state. This corresponds to the source state element of the XML specification. Transitioning to the target state "created" is assured by the postcondition. If a violation of a pre- or postcondition occurs a *System.Assertions.Assert_Failure* exception is raised by the Ada runtime. This assures that only transitions conforming to the

[1]XSLT is a language standardized by the W3C (World Wide Web Consortium) for transforming XML documents

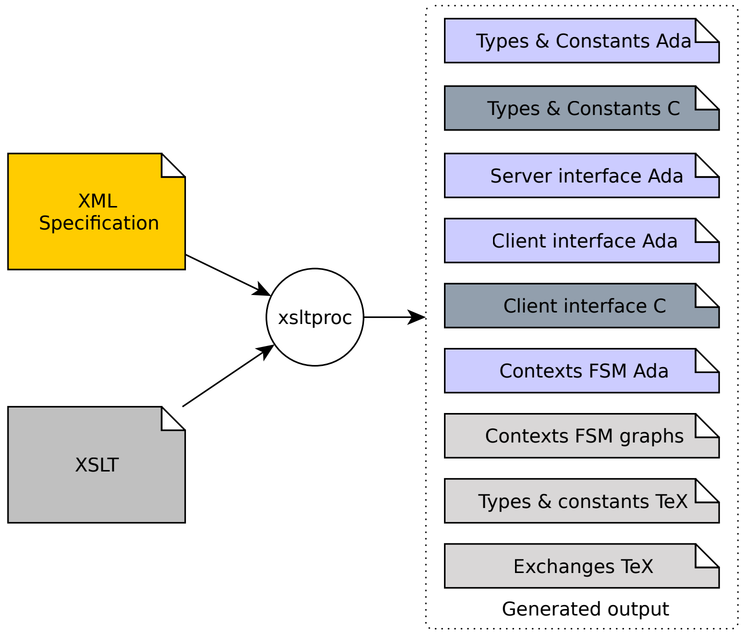


Figure 5.2: XSL Transformation of XML specification

specification are possible. Confidence that the code implements the specification can be raised further by applying the GNATprove[2] tool [3] to the source. The XSL code generation process provides support to run GNATprove automatically, after the sources have been created.

Another example of generated output are the state machine diagrams show in section 4.4.

The following list enumerates the main XSLT output that is generated from the specification:

- *Types*: Ada and C type definitions
- *Constants*: Ada and C constant definitions
- *RPC*: Ada RPC library with exported C functions, includes request/response marshaling and server-side exchange ID to service procedure dispatching
- *Contexts*: Ada context state machines including Ada 2012 contracts
- *Documentation*: Types, constants and exchange description as well as state machine diagrams

[2]GNATprove is a formal verification tool for Ada 2012 contracts. It can prove that subprograms honor their preconditions and postconditions.

```
<transition name="create">
    <descr>Create new nonce.</descr>
    <source_states>
        <state name="clean"/>
    </source_states>
    <target>
        <state name="created"/>
        <field name="nonce"> nonce </field>
    </target>
</transition>
```

Listing 5.1: Specification of nonce create transition

```
procedure create (Id : Types.nc_id_type;
                  nonce : Types.nonce_type)
with
  Pre  => Is_Valid (Id) and then
         (Has_State (Id, clean)),
  Post => Has_State (Id, created) and
          Has_nonce (Id, nonce);
```

Listing 5.2: Generated Ada nonce create procedure

## 5.3 RPC library: tkm-rpc

Since the main objective of this project is to separate security-critical functionality from untrusted software components and extract it into a TCB, the need for a communication mechanism between the disjointed parts arises. The communication layer is abstracted into a self-contained library called tkm-rpc. It allows the untrusted and trusted side to exchange well-formed messages, so called request and responses, as defined by the interface specification.

At the core of an exchange are the request and response data types. Clients send a request object to a server and the server responds by sending back a corresponding response object. Section 5.3.1 describes the general operation of the tkm-rpc library.

To make use of the library clients simply include the necessary project or header files, which contain the type, constant definitions and procedure or function specifications. How the library is intended to be used by clients is described in section 5.3.3.

Server-side components are expected to provide an implementation of interface specific procedures. How the server processing is done is illustrated in section 5.3.4.

When appropriate, the concrete implementation is illustrated using the `nc_create` exchange, which is specified in section 4.3.1.1.

### 5.3.1 Basic operation

The tkm-rpc library provides an RPC (remote procedure call) interface that uses a data transmission channel to pass client requests to a server and responses back to the client. The basic layout of request and response data types are shown in figure 4.1. The operation type of a request or response specifies what exchange it is part of. Requests are matched to their corresponding responses using the request_id field. However, this is currently not implemented (see also the limitation section 5.8). Support for multiple simultaneous exchanges and asynchronous request processing can be implemented using the request_id matching. Currently a call to the tkm-rpc library blocks the client until the server's response is received.

Most of the library code is automatically generated based on the XML specification. Only the transport-specific parts of sending and receiving requests and responses using a particular communication method is implemented manually. The exchange of data is performed using Unix domain sockets. The necessary networking functionality is provided by the Anet library, which is described in section 5.7.1.

The round trip of an exchange is illustrated by figure 5.3.

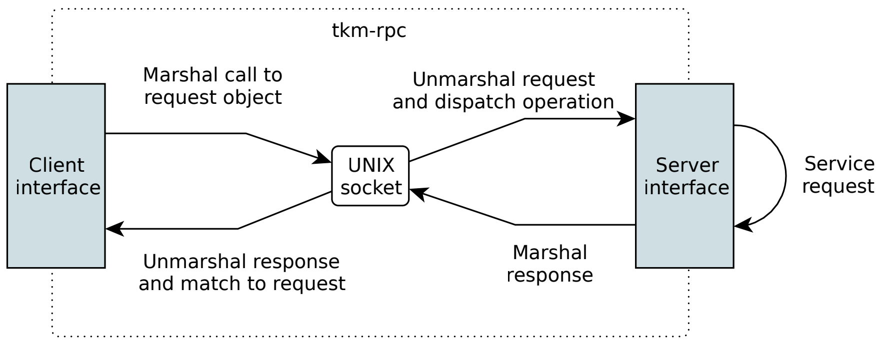


Figure 5.3: Basic IPC operation

A client calls a function or procedure that is specified by the TKM interface. That call is translated into a request object with the operation set to the corresponding exchange id. Any parameters are marshaled into data fields of the request object. The request is then transmitted to the server via a Unix domain socket[3] [12].

On the server side of the socket, the request object is unmarshaled. The operation is dispatched according to the exchange ID and the parameters of the exchange are extracted from the request object. The call is then forwarded to the server passing it the necessary arguments sent by the client. At this point the server performs all necessary actions to service the requested operation. After the server has finished handling the request, it returns result data. A response data object is created with the same exchange and request IDs as the request object. The response parameters are then marshaled into the corresponding response data fields and the response object is sent back to the client via the Unix socket.

Back on the client side the response is unmarshaled and any return parameters are extracted from the response object. These values are then passed back to the client thus completing the exchange.

[3]Unix domain sockets are a standard IPC mechanism and are part of the POSIX socket API

```
1  type Data_Type is record
2     Nc_Id         : Types.Nc_Id_Type;
3     Nonce_Length : Types.Nonce_Length_Type;
4  end record;
5
6  for Data_Type use record
7     Nc_Id        at 0 range 0 .. (8 * 8) - 1;
8     Nonce_Length at 8 range 0 .. (8 * 8) - 1;
9  end record;
10 for Data_Type'Size use Data_Size * 8;
11
12 type Request_Type is record
13    Header  : Request.Header_Type;
14    Data    : Data_Type;
15    Padding : Padding_Type;
16 end record;
```

Listing 5.3: nc_create request-specific data type

A complete list of all IKE exchanges is given in section 4.3.1.

### 5.3.2 Request and Response types

Each exchange has a specific request and response type. These are generated from the XML specification and their basic structure is depicted in figure 4.1. Every request has a header which contains the exchange and the request identifier. Responses contain the same header information plus an additional status code. The result code signals success or error conditions to the caller using the constant values specified in section 4.2.3.

Exchange specific data is stored in additional record fields after the header. Listing 5.3 shows the generated data type of the `nc_create` exchange, consisting of the header and request-specific data.

As is apparent, the requests parameters as specified in section 4.3.1.1 have a corresponding record field in the exchange-specific data type. All requests like all response types are of the same size. Requests that are smaller than the required length are padded with zeros. Responses are constructed following the same idiom.

The exact memory layout of the record is specified using an Ada record representation clause (lines 6-9). The clause specifies that the *Nc_Id* record field starts at byte offset 0. Starting at that offset the range occupied is from 0 up to bit 63. For a more detailed explanation of the representation clause, the reader is directed to [2], section 13.5.1.

### 5.3.3 Client-side usage

The purpose of an RPC library is to hide the complicated exchange and transport details from the user. It must be very easy to use and remote calls should look like local procedure or function calls to the client. As previously mentioned the majority of the RPC client library is automatically generated from the XML

```
with Tkmrpc.Request;
with Tkmrpc.Response;

package Tkmrpc.Transport.Client is

   procedure Connect (Address : String);
   --  Connect to the RPC server given by socket address.

   procedure Send (Data : Request.Data_Type);
   --  Send request data to RPC server.

   procedure Receive (Data : out Response.Data_Type);
   --  Receive response data from RPC server.

end Tkmrpc.Transport.Client;
```

Listing 5.4: Client Tkmrpc transport abstraction

specification. An exception is the transport layer. The next section explains the motivation and the operation of the transport layer abstraction.

Section 5.3.3.2 illustrates how clients use the RPC library and how the internal processing works.

#### 5.3.3.1 Transport mechanism abstraction

The transport layer constitutes the lowest level of the RPC library. To ease the usage of different communication mechanisms, all necessary functionality is encapsulated in the `Tkmrpc.Transport.Client` package. The current implementation employs stream-oriented Unix sockets using the functionality provided by Anet (see section 5.7.1). To run the TKM daemon on a different physical machine, switching to a TCP socket implementation and connecting to an IP address and port is all that is necessary from the client's point of view.

The interface, which is automatically generated, is rather simple and only three procedures must be implemented, see listing 5.4.

Before a client can do remote procedure calls using tkm-rpc it must connect to the remote server component specifying the filename of the Unix socket, where the server is listening for exchanges. The `Send` procedure is used to transmit request objects to the connected RPC server while the `Receive` procedure returns a response object received from the server. Section 5.3.3.2 explains how the two procedures are used to implement request and response handling.

#### 5.3.3.2 Request handling

Based on the XML exchange description Ada procedure definitions are generated. Since the exchanges are specified on a per-interface basis (e.g. IKE or EES), procedures belonging together are put in the same package, e.g. `Tkmrpc.Clients.Ike`. Listing 5.5 shows the generated procedure declaration for the `nc_create` exchange.

The export pragmas make the procedures callable from the C programming language. To enable the use of the library in C, a header file containing cor-

```
procedure Nc_Create
  (Result        : out Results.Result_Type;
   Nc_Id         : Types.Nc_Id_Type;
   Nonce_Length  : Types.Nonce_Length_Type;
   Nonce         : out Types.Nonce_Type);
  pragma Export (C, Nc_Create, "ike_nc_create");
  pragma Export_Valued_Procedure
    (Nc_Create,
     Mechanism => (Nc_Id => Value, Nonce_Length => Value));
--  Create a nonce.
```

Listing 5.5: Nc_Create procedure declaration (client-side)

```
/**
 * Create a nonce.
 */
extern result_type ike_nc_create(const nc_id_type nc_id,
                const nonce_length_type nonce_length,
                nonce_type *nonce);
```

Listing 5.6: ike_nc_create function declaration

responding C function declarations for each exchange is also generated. Since the C language has no notion of packages and has one global namespace, all procedures are prefixed with the name of the interface they belong to. Thus the exchange to create a nonce is called `Nc_Create` in Ada and `ike_nc_create` in C. Listing 5.6 shows the C function declaration equivalent to the Ada procedure presented in listing 5.5.

When a client calls the Nc_Create procedure a request object is created, filling in the passed parameters. Next the object is transmitted using the Send procedure described in section 5.3.3.1. Afterwards the Receive procedure is used to get a response from the server. The result parameters are extracted from the response data type and returned to the client depending on the function signature.

### 5.3.4 Server-side processing

RPC servers are passive components which respond to requests sent by clients. The main focus of server-side processing is automatic mapping of requests to concrete exchanges as specified by the interface. A server implementation should not be burdened with the details of exchange and request ID handling but concentrate on the implementation of the functionality prescribed by the exchange.

Much like the client part of the RPC library, most of the code is automatically generated. Section 5.3.4.1 describes how incoming requests are dispatched to their corresponding exchange handlers. After that a description of error handling is given in section 5.3.4.2.

```
procedure Nc_Create
  (Result        : out Results.Result_Type;
   Nc_Id         : Types.Nc_Id_Type;
   Nonce_Length  : Types.Nonce_Length_Type;
   Nonce         : out Types.Nonce_Type);
--  Create a nonce.
```

Listing 5.7: Nc_Create procedure declaration (server-side)

```
procedure Dispatch
  (Req : Request.Data_Type;
   Res : out Response.Data_Type);
--  Dispatch IKE request to concrete operation handler.
```

Listing 5.8: Ike request dispatcher

#### 5.3.4.1 Operation dispatching

All operations exposed to the client via the tkm-rpc library must be implemented by a RPC server. To ensure this, an Ada package containing procedure declarations is generated for each interface described in the XML specification. As can be seen by comparing listing 5.7 to listing 5.5, the client and server side procedure declarations are almost identical. The procedure is not exported since all processing is done in Ada and the procedure is not meant to be called from C code.

A server implementing the *ike* interface must provide a package body implementing the `Tkmrpc.Servers.Ike` package.

A dispatcher which takes a request data object as input and calls the corresponding procedure according to the exchange identifier is generated also. This takes the burden of mapping an exchange ID to the correct operation handler from the server implementation. It also avoids possible errors such as typos, which can be hard to detect. Additionally the generated code guarantees that all specified exchanges are handled and unknown exchanges are answered by returning an Invalid_Operation status code via a response data object. Listing 5.8 shows the procedure declaration of the *ike* dispatcher, which is located in the (generated) `Tkmrpc.Dispatchers.Ike` package.

Since data received from the client via the Unix socket is just a sequence of octets, a method to translate the binary data into request types and passing them to the presented dispatcher is needed. The different parts are brought together by the `Tkmrpc.Process_Stream` generic. Listing 5.9 shows the declaration of the generic procedure.

To instantiate the generic, a Dispatch procedure matching the given signature must be provided. Optionally an exception handler can also be specified. The generic `Process_Stream` procedure automatically converts stream data to Tkmrpc request/response objects and passes them on to the given dispatch procedure. The exception handler is called when the specified dispatching procedure raises an exception.

```
generic
   with procedure Dispatch
     (Req :      Request.Data_Type;
      Res : out Response.Data_Type);

   with procedure Exception_Handler
     (Ex : Ada.Exceptions.Exception_Occurrence) is null;

procedure Tkmrpc.Process_Stream
  (Recv_Data :     Ada.Streams.Stream_Element_Array;
   Send_Data : out Ada.Streams.Stream_Element_Array;
   Send_Last : out Ada.Streams.Stream_Element_Offset);
```

Listing 5.9: Process stream generic

#### 5.3.4.2 Error handling

The intended way for indicating errors during processing of client requests is by raising exceptions. Such an exception propagates all the way up to the `Process_Stream` generic's exception block. There the result code of the response is set to failure to indicate an error to the client.

This mechanism works well in combination with the automatically generated context state machines because violation of pre- and postconditions raise an *System.Assertions.Assert_Failure* exception (see section 5.2). These are then properly processed by the `Process_Stream` generic to relieve the server implementation of the burden of dealing with all possible error cases.

The current implementation returns Invalid_Operation if an error occurs and does not translate exceptions to their corresponding error codes, see also section 5.8.

How potential exceptions are handled on the client side is outlined in section 5.4.13.

## 5.4 charon-tkm

The untrusted IKEv2 component used in conjunction with the Trusted Key Manager infrastructure is implemented as a separate charon "instance" located in its own directory below the strongSwan top-level source directory (`src/charon-tkm`). This has the advantage that the TKM code is contained and does not mix with other strongSwan files. The charon-tkm binary startup code works like the already existing charon-nm instance, a special charon daemon variant to be used with the GNOME NetworkManager project[4]. The only difference is the registration of custom TKM plugins as the final step of the startup phase. The charon-tkm daemon does not rely on the dynamic plugin loading mechanism for its core plugins, they are statically registered before entering the main processing loop.

Since the charon-tkm code uses the tkm-rpc library written in Ada, the daemon has to be built using an Ada-aware toolchain. This integration of Ada code into the strongSwan codebase is explained in section 5.4.1. Apart from the

[4] http://projects.gnome.org/NetworkManager/

tkm-rpc library explained in section 5.3, the ESP SA event service and a special exception handler component are directly written in Ada inside the charon-tkm project itself. These subsystems are outlined in sections 5.4.12 and 5.4.13.

### 5.4.1 Ada integration

As explained in section 5.3, the tkm-rpc library is written in Ada and uses the export feature of the language (`pragma` Export) to make procedures available to the charon-tkm C code. To call Ada code from C requires an initialized Ada runtime. To that end the special `adainit` and `adafinal` procedures must be called before and after Ada code is used. Setup and teardown of the Ada runtime is transparently handled by the tkm-rpc library (in the `tkmlib_init` and `tkmlib_final` functions), but the charon-tkm code must still be compiled with an Ada-aware tool chain to correctly compile, bind and link the daemon binary.

strongSwan uses the GNU build system, also known as the Autotools[5], to configure, compile and install the project. Ada projects using the GNAT Ada compiler usually use gnatmake or gprbuild[6] to build projects. It is common practice in Ada projects to mix these two concepts by calling the respective GNAT project manager from inside a Makefile for example. Therefore the charon-tkm project provides a Makefile.am file which describes how to build the charon-tkm daemon binary with gprbuild. The project uses the more advanced gprbuild manager because it provides superior support for mixed language projects (C and Ada in this case).

### 5.4.2 Initialization

The entry point of the untrusted component is the `main` function located in the file `charon-tkm.c`. Before entering the main loop, the charon-tkm daemon calls the `tkm_init` function which initializes the tkm-rpc library explained in section 5.3 and starts the exception handler (5.4.13) used to catch Ada exceptions on the client side.

It then calls the `ike_init` function to connect to the IKE interface of the TKM. After that the ESP SA event service is started which accepts ESA acquire and expire events from clients (5.4.12). If no error occurred (which would result in the termination of the daemon), the initialization code instructs the TKM inside the TCB to reset itself by calling the `ike_tkm_reset` remote procedure call.

Since the TKM supports a static number of contexts (see section 3.4.1), the upper limit of context IDs is requested from the TKM. This limit configuration is then passed on to the TKM ID manager which is initialized in the final step along with the TKM chunk map. The daemon enters the main loop and waits for external events.

### 5.4.3 ID manager

The TKM ID manager implemented in files `tkm/tkm_id_manager.[h|c]` handles the management of the different context ID kinds. Its interface is very

[5] https://en.wikipedia.org/wiki/GNU_build_system
[6] GNAT's Project Manager

simple. The `acquire_id` function can be used to acquire (reserve) a new ID for a given context (e.g. `TKM_CTX_DH` for a new DH context ID). The `release_id` function releases an already reserved ID. If no ID can be acquired, the `acquire_id` function indicates this error by returning zero. The first valid ID of a given context always starts at number one.

### 5.4.4 Data passing

The TKM code uses two main techniques to pass on information from one plugin to another for cases where the strongSwan interface is not prepared to handle the use case. These two techniques allowed to implement the required TKM functionality without being too invasive to the upstream strongSwan codebase. This is especially true for situations which are only relevant for the TKM project, with no benefit for the project as a whole.

One of these mechanisms use the chunk map explained in the next section and the other is explained in section 5.4.4.2.

#### 5.4.4.1 Chunk map

The chunk map can be used to store mappings of chunks[7] to context IDs.

The mapping mechanism is illustrated using the nonce allocation process. The nonce plugin allocates a fresh nonce in a new context and stores this relation in the chunk map. This is necessary since such IDs cannot be passed along using the existing strongSwan interfaces and are only used inside the TKM code. Listing 5.10 shows how the described functionality is implemented in the nonce plugin.

```
*chunk = chunk_alloc(size);
if (get_nonce(this, chunk->len, chunk->ptr))
{
    tkm->chunk_map->insert(tkm->chunk_map, chunk,
                           this->context_id);
...
```

Listing 5.10: Nonce ID insertion

The keymat plugin receives the nonce chunk as function parameter. It needs the corresponding nonce context ID to tell the TKM which nonce to use for processing. The associated context ID is retrieved from the chunk map, as shown by listing 5.11.

```
/* Acquire nonce context id */
uint64_t id = tkm->chunk_map->get_id(tkm->chunk_map, nonce);
```

Listing 5.11: Nonce ID retrieval

[7]Chunks are strongSwan's notion of binary data containing e.g. nonces or cryptographic keys

#### 5.4.4.2 Piggybacking

Another method of passing TKM specific information over plugin borders uses a piggybacking technique to store informational structs inside chunk objects. strongSwan often treats such chunks as opaque values while passing them between plugins. This allows to store TKM-specific information in these chunks for plugins which use it to initiate an action with the TKM.

Listing 5.12 shows the `isa_info_t` informational structure used to transfer ISA information from the keymat of a parent SA to the keymat of the new IKE SA during a rekeying operation.

```c
struct isa_info_t {
	/**
	 * Parent isa context id.
	 */
	isa_id_type parent_isa_id;

	/**
	 * Authenticated endpoint context id.
	 */
	ae_id_type ae_id;
};
```

Listing 5.12: isa_info_t struct

In this case the sk_d data chunk returned by the `get_skd` function is used to transport the `isa_into_t` informational structure. This is possible since the sk_d chunk is treated as an opaque value and handed to the `derive_ike_keys` procedure of the new keymat as-is without any processing. The information is stored in the sk_d chunk as shown by listing 5.13.

```c
isa_info_t *isa_info;
INIT(isa_info,
	.parent_isa_id = this->isa_ctx_id,
	.ae_id = this->ae_ctx_id,
);
*skd = chunk_create((u_char *)isa_info, sizeof(isa_info_t));
```

Listing 5.13: Piggybacking

This method is simple and does not require a global data structure accessible to the involved plugins thus avoiding the problem of synchronization.

### 5.4.5 Nonce generation plugin

Nonce generation plugins are a new feature of strongSwan introduced during this project. A nonce generation plugin is responsible to create new nonces needed in the IKE_SA_INIT and CHILD_CREATE_SA exchanges (see sections 2.1.2 and 2.1.4). In case of the TKM, the nonce generation plugin requests a new nonce from the TKM by calling the `ike_nc_create` RPC and then registers the nonce in the chunk map to store the nonce to context ID mapping. This mapping is used by other plugins which need to pass on a nonce context to the TKM for key derivation purposes.

### 5.4.6 Diffie-Hellman plugin

The TKM Diffie-Hellman plugin instructs the TKM to perform the DH protocol on its behalf. On creation, the plugin calls the `ike_dh_create` RPC with a new context ID acquired from the ID manager. This initiates the initial steps of the Diffie-Hellman protocol in the TKM. The plugin completes the DH exchange by calling the `ike_dh_generate_key` function as soon it receives the public value when its `set_other_public_value` function is called, as illustrated by figure 2.1 and 2.5. No secret values leave the TCB at any time but the DH context stored in the TKM can be referenced later for key derivation by using the correct DH context ID.

### 5.4.7 Keymat plugin

The charon-tkm code uses the new keymat registration facility developed during this project to register a special TKM keymat variant, which acts as proxy for the remote keying material stored in the TKM. A keymat instance is constructed together with its corresponding IKE SA and stays active for the lifetime of this SA.

Upon construction, the TKM keymat plugin acquires an ISA context ID (`TKM_CTX_ISA`) from the ID manager. It then behaves like the standard IKEv2 keymat, except that it does not store or receive any critical data. Calls to `derive_ike_keys` and `derive_child_keys` are dispatched into the TCB by using context IDs. The keys used to protect the IKE SA are returned to the keymat after the `ike_isa_create` or `ike_isa_create_child` remote procedure call returns because they are not classified as critical (see section 3.5.4).

The keymat plugin uses the piggybacking mechanism described in 5.4.4.2 to forward information to plugins or to extract required information from other sources. For example the `derive_child_keys` function does nothing more than use the encryption key chunks to store information needed by the kernel IPsec plugin. The actual child key derivation is postponed until the registered kernel plugin's `add_sa` function is called by the task which takes care of child creation, see figure 2.5 on page 22, labels (*SI*) and (*IS*).

### 5.4.8 Kernel IPsec plugin

After keying material for a new child SA has been derived in the TKM, the child SA state must be established using a kernel IPsec plugin. In case of the TKM, where no child keying material leaves the TCB and child SA policy handling is completely done by the TKM, the kernel plugin can be kept very simple. It only provides a custom `add_sa` function used to instruct the TKM to derive child keys and install a new ESA (ESP SA) state inside the TCB's encrypter component. This is of course only possible if all preconditions for this operation are met.

### 5.4.9 Private key plugin

The TKM private key plugin instructs the TKM to create and return the authentication octet signature for a given ISA context. Since the code flow of the signature creation process involves two different plugins, namely the keymat and the private key plugin, information must be passed between these plugins. The AUTH octet chunk returned by the keymat's `get_auth_octets` function is piggybacked in this case. See section 5.4.4.2 for an explanation of the piggybacking mechanism. The TKM keymat stores the associated ISA context ID and the initial message in the chunk and returns it to the caller, which is a pubkey authenticator in this case (see figure 2.3). The public key authenticator then calls the sign operation of the private key plugin. The private key code extracts the stored data and calls the `ike_isa_sign` operation to create the AUTH octet signature. The signature is then returned to the caller.

In its current implementation, the TKM private key plugin is hard-coded to a specific key pair (*alice@strongswan.org* used in the strongSwan integration test suite). The reason for this limitation lies in the way the code is searching for a matching private key to authenticate a connection. It uses the key fingerprint (which is encoded from the key's modulus and public exponent values) of a public key contained in the user certificate configured for a connection to find the corresponding private key. Since no real private key exists in the TKM-case, because the private key never leaves the TCB, the private key plugin must imitate a key fingerprint to be found.

### 5.4.10 Public key plugin

Figure 2.3 shows how the AUTH octet signature received from a peer is verified. Since the verification is done in the TKM, a dummy public key plugin must be provided which fakes the verification process in the untrusted part.

To make sure charon always uses the TKM public key plugin implementation for public key processing, it is registered first during daemon startup.

### 5.4.11 Bus listener plugin

The strongSwan architecture provides an internal bus which can be used to subscribe for specific events. To inform charon about the IKE SA authorization result from the TKM, a mechanism called authorization hooks is used. The TKM bus listener plugin registers itself as listener for IKE messages and the corresponding IKE authorization events to make sure it is consulted in the final authorization round for an IKE SA.

The message hook in the TKM listener is needed to extract the authorization payload from the peer's incoming IKE_AUTH message. The extracted authorization payload is stored in the keymat in the IKE SA corresponding to the exchange in progress. This is done by calling the custom TKM keymat function `set_auth_payload`. Later this payload is used in the authorize hook of the bus listener hook to instruct the TKM to perform the authentication process in the TCB.

The authorize hook, called by charon as last step in authorization rounds, retrieves the keymat by using the associated IKE SA object received as function argument. It then allocates a new certificate chain context ID and calls the internal `build_cert_chain` function to construct the certificate trust chain of the received peer certificate. The peer's user certificate stored in the authentication configuration of the associated IKE SA is set as user certificate for this CC context in the TKM by calling the `ike_cc_set_user_certificate` function. This is the certificate for which trust must be established. For all intermediate certificates, the `build_cert_chain` function calls the TKM `ike_cc_add_certificate` RPC. The TKM verifies the trust chain. At the end the CA certificate of the chain in question is passed on to the TKM. This certificate must be bit-wise identical to the one the TKM trusts[8]. If the trust chain could not be verified, the authorize hook returns failure and the authentication of the IKE SA does not succeed.

The the trust chain verification is successful, the authorize hook retrieves the authentication payload stored by the message hook from the keymat and passes it to the TKM by using the `ike_isa_auth` RPC. The TKM uses the given certificate context which contains the now trusted peer public key to verify the signature.

### 5.4.12 ESP SA event service (EES)

The ESP SA event service exports the EES interface specified in section 4.3.2. The service is written in Ada as a subsystem of the charon-tkm daemon and is located in the `ees` sub-directory. It uses the tkm-rpc library outlined in section 5.3 to implement its RPC interface.

The EES component accepts ESA acquire and expire events from clients and dispatches them to the charon C code by using callbacks. The callbacks use the strongSwan hydra kernel interface to initiate an acquire or expire event the same way it is used if events are received from the Linux kernel directly. The ESP SA service is used by the xfrm-proxy component outlined in section 5.6 to relay messages from the kernel's XFRM subsystem to charon. This is needed since charon, in this separation scenario, is no longer allowed to talk to the kernels IPsec SAD database directly since it contains sensitive child SA keys.

### 5.4.13 Exception handler (EH)

The charon TKM code located in the `ehandler` sub-directory provides a special exception handler which implements the functionality to log exception messages from within Ada code into the daemon's log file. This mechanism is implemented using the *Exceptions_Actions* framework of the GNAT Ada runtime. An Ada procedure with the correct signature can be registered as handler for any exception occurring in the runtime[9].

The registered exception handler calls the imported C function `Charon_Terminate` which logs the exception message into the daemon's log file and instructs it to terminate.

[8] In its current implementation, the TKM only trusts one CA

[9] As a side note, this also includes internal exceptions which are normally not seen by user code.

## 5.5 TKM

The TKM component implements a minimal Trusted Key Manager as depicted in figure 3.1 on page 28. It provides the critical functionality extracted from the strongSwan code base. The TKM is written in the Ada programming language and uses the tkm-rpc library described in section 5.7.2 to provide the IKE interface (see 4.3.1) via remote procedure calls.

The dispatching of incoming calls is done by providing a custom IKE server implementation (`Tkmrpc.Servers.Ike`) as explained in section 5.3.4.1. From there, calls are forwarded to the appropriate subsystems explained in the following sections.

### 5.5.1 Client communication

Exchanges between charon-tkm and the TKM daemon are transferred using a Unix domain socket. The TKM implementation instantiates the `Process_Stream` generic described in section 5.9 with the automatically generated IKE dispatcher and a logging procedure as exception handler. The procedure is used in conjunction with an Anet stream receiver to perform the request and response processing.

### 5.5.2 Nonce generation

Nonces are used to guarantee freshness in the cryptographic operations when deriving key material. Hence nonce values must be random and must not be predictable. The nonce handling is implemented in the `Tkm.Servers.Ike.Nonce` package.

Currently, `/dev/urandom` is used as random source inside the TKM. The quality of randomness provided by this source is considered strong enough for the current initial iteration. The design is such that the implementation could be easily replaced by a stronger random source at a later time.

The TKM guarantees that nonces are consumed once and can not be reused, as specified by requirement 3.5.5. This is assured by using auto-generated nonce FSM as explained in the state-machines section 4.4. Each created nonce is an instantiation of a nonce FSM. If the client requests to create a new context with an already taken nonce ID, an assertion exception is raised and an error status is returned to the requester.

### 5.5.3 Diffie-Hellman

Keying material used to protect a child SA is derived from the shared secret computed by a Diffie-Hellman exchange. This keying material is considered the most sensitive and must therefore reside in the TCB only. From this requirement follows that the TKM must implement the Diffie-Hellman protocol to perform the exchange on behalf of clients like the untrusted charon-tkm daemon.

Currently the TKM provides a Diffie-Hellman implementation for the 3072-bit and 4096-bit MODP Diffie-Hellman groups specified in RFC 3526 [19]. The GNU Multiple Precision Arithmetic Library[10] is used in the implementation

[10] http://gmplib.org/

```
package Tkm.Crypto.Hmac_Sha512 is new Tkm.Crypto.Hmac
  (Hash_Block_Size => 128,
   Hash_Length     => 64,
   Hash_Ctx_Type   => GNAT.SHA512.Context,
   Initial_Ctx     => GNAT.SHA512.Initial_Context,
   Update          => GNAT.SHA512.Update,
   Digest          => GNAT.SHA512.Digest);
```

Listing 5.14: TKM HMAC SHA-512

since an Ada binding exists[11].

An active DH exchange is stored in the DH FSM introduced in section 4.4. The FSM's pre- and postconditions assure that only valid states and transitions are allowed during an exchange. If the protocol specified by the DH FSM is violated, an assertion exception is raised and the requester is informed about the violation. DH contexts can only be consumed if they are in *generated* state as shown by the corresponding state machine diagram in section 4.4.

### 5.5.4 Key derivation

The TKM implements the procedures needed to derive IKE and child keys as described by the following subsections.

#### 5.5.4.1 IKE SA keys

The IKE SA (ISA) key derivation functionality in the TKM implements the mechanism described in RFC 5996 [16], section 2.14. To derive keys for an IKE SA, the derivation function first retrieves the associated DH and nonce contexts which must be in the correct state, otherwise an exception is raised. It then instantiates a pseudo-random function (PRF) needed to generate the SKEYSEED value as shown by formula 5.1.

$$SKEYSEED = prf(Ni|Nr, shared\,secret) \tag{5.1}$$

The TKM provides a PRF which uses a hash-based message authentication code (HMAC) as base. The HMAC functionality is implemented as a flexible Ada generic which can be instantiated using different hash functions. The TKM currently does not implement its own hash functions but instead re-uses the ones provided by the GNAT Ada compiler. The HMAC generic is instantiated as shown by listing 5.14.

To derive the IKE SA keys, the $prf+$ function as specified in RFC 5996 [16], section 2.13 is required. This functionality is again provided by an Ada generic, which can be instantiated using different PRF contexts matching the required signature. The $prf+$ function outputs a pseudo-random stream used for IKE SA encryption and integrity keys. The keys are returned to the untrusted caller as they are not considered critical itself. This is true under the assumption that the PRF function used to generate the keys is strong enough to make it impossible to reverse the process[12].

[11] http://mtn-host.prjek.net/projects/libgmpada/

[12] The TKM currently uses PRF-HMAC-SHA512 as PRF for the $prf+$ function

An authentication context is created alongside the ISA context after the IKE SA keying material has been successfully derived. This AE context must first be authenticated properly until child SA keys can be derived (see section 5.5.7.2).

#### 5.5.4.2 Child SA keys

The process of deriving keying material for a child SA is described in RFC 5996 [16], section 2.17. The TKM only allows the derivation of child keys if the associated authentication context (AE) is in the *authenticated* state:

```
pragma Precondition (Tkmrpc.Contexts.ae.Has_State
   (Id     => Tkmrpc.Contexts.isa.get_ae_id (Id => Isa_Id),
    State  => Tkmrpc.Contexts.ae.authenticated));
```

Listing 5.15: Create_Esa precondition

The actual keying material for the child SA is derived using the *prf+* function. Currently the TKM only supports PRF_HMAC_SHA512 as base for the *prf+*, so the untrusted charon-tkm counterpart and the remote peer involved in the connection must be configured accordingly. The keys derived are pushed into the kernel's SA database (SAD) using functionality provided by the xfrm-ada project described in section 5.7.2.

The TKM supports different configurations for ESA creation only differing in the way related nonce and DH contexts are consumed. The first child SA of a connection does not depend on nonce or DH contexts at all, because it is derived in conjunction with its IKE SA. Then there is the configuration where no PFS is desired, so no new DH context must be created beforehand.

### 5.5.5 Private key

The TKM only supports authentication schemes based on asymmetric cryptography. To create a signature using such a scheme, a private key is needed. The key to use can be specified on the command line using the `-k` option. The TKM expects the key to be a RSA PCKS#1 [15] private key in DER [14] encoding and is loaded into the `Tkm.Private_Key` package where it can be retrieved using a getter function. The functionality to load and parse the private key is provided by the x509-ada project described in section 5.7.3.

### 5.5.6 CA certificate

To establish assurance in a user certificate provided by a remote peer, the trust chain of this certificate must be verified (see the following section 5.5.7.3). Hence the TKM needs a trust anchor which is embodied in a certificate authority (CA). Currently the TKM only trusts one CA certificate which can be specified on the command line using the `-c` option. The CA certificate in X.509 [6] format is loaded into the `Tkm.Ca_Cert` package using the x509-ada (5.7.3) project. The `Load` procedure of the package checks the validity of the CA certificate and raises an exception if it is not valid.

### 5.5.7 Authentication

As dictated by the requirement described in section 3.5.6, the authentication process must be performed by the TKM to assure correctness. The following sections outline the implemented mechanisms in detail.

#### 5.5.7.1 Signature generation

The TKM implements the RSASSA-PKCS1-v1_5 signature scheme with appendix as specified by RFC 3447 [15], section 8.2. The functionality is provided as an Ada generic, allowing the instantiation with different hashing algorithms. Pre-instantiated instances are provided for SHA-1 and SHA-256 algorithms. Listing 5.16 shows how to create a signature using the PKCS#1 private key given on the command line.

```
1  declare
2     use X509.Keys;
3
4     package RSA renames Crypto.Rsa_Pkcs1_Sha1;
5
6     Signer  : RSA.Signer_Type;
7     Privkey : constant RSA_Private_Key_Type
8        := Private_Key.Get;
9     Chunk   : Tkmrpc.Types.Byte_Sequence (1 .. 5)
10       := (others => 10);
11 begin
12    RSA.Init
13     (Ctx   => Signer,
14      N     => Get_Modulus (Key => Privkey),
15      E     => Get_Pub_Exponent (Key => Privkey),
16      D     => Get_Priv_Exponent (Key => Privkey),
17      P     => Get_Prime_P (Key => Privkey),
18      Q     => Get_Prime_Q (Key => Privkey),
19      Exp1  => Get_Exponent1 (Key => Privkey),
20      Exp2  => Get_Exponent2 (Key => Privkey),
21      Coeff => Get_Coefficient (Key => Privkey));
22    declare
23       Sig : constant Tkmrpc.Types.Byte_Sequence
24          := RSA.Generate (Ctx  => Signer,
25                           Data => Octets);
26    begin
27       --  Do something with the signature
28    end;
29 end;
```

Listing 5.16: Signature generation

On line 4 a RSA signer is instantiated. Line 5 retrieves the private key stored in the `Tkm.Private_Key` package and uses the parameters of this key to initialize the RSA signer on line 10. Finally, on line 21 the signature over the given data chunk is created using the `Generate` procedure of the RSA package.

The same code is used to create a signature over the local authentication octets during the IKE_AUTH exchange (see section 2.2.2). The charon IKEv2 daemon currently only supports AUTH octet signatures based on the SHA-1 hash algorithm, this must be improved in a future iteration so that other hash algorithms are possible.

#### 5.5.7.2 Signature verification

Similar to the signature generation outlined in the previous section 5.5.7.1, the TKM provides an Ada generic to verify RSASSA-PKCS1-v1_5 signatures [15]. To perform a verification, a `RSA.Verifier_Type` must be initialized using a public key extracted from a trusted certificate. The process of trust chain verification is explained in detail in section 5.5.7.3.

During the IKE_AUTH exchange, the identity of a remote peer must be asserted. This is done by verifying the signature of the authentication octets. If the signature validates, the authentication context (AE) of the IKE SA in question is set into the *authenticated* state, meaning that it is now possible to establish child SAs (ESA) under this IKE SA (ISA).

#### 5.5.7.3 Certificate chain validation

Figure 5.4 provides an overview of the steps performed to establish trust in the user certificate provided by a peer during the IKE_AUTH exchange. The chosen example involves three certificates: the user certificate *A*, the intermediate CA certificate *B* and the trusted *CA*. The goal of the process is to link the user's X.509 [6] certificate *A* to the *CA* trusted by the TKM. This is done by verifying the chain of certificate signatures, starting at the bottom with the peer certificate *A* and moving upwards to the root of trust, the *CA* certificate.

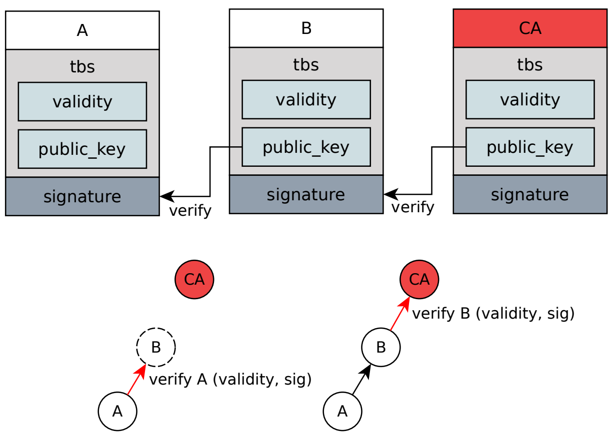


Figure 5.4: TKM trust chain validation overview

In order to to this, the user certificate depicted as certificate *A* in figure 5.4 must be validated using the intermediate CA certificate *B*, and the intermediate certificate *B* must be validated using the trusted *CA* certificate. Validation in the context of a certificate trust chain means to perform the following steps:

1. Checking the validity period of the certificate: The current time (`Now`) must be within this period as illustrated by listing 5.17.

2. Verify the signature stored in the certificate by using the public key of the subsequent certificate (the issuer certificate).

3. Perform additional checks as suggested by [6, 8, 24]. These checks are not yet implemented in the current state of the project.

```
function Is_Valid (V : Validity_Type) return Boolean
is
   use Ada.Calendar;
   Now : constant Time := Clock;
begin
   return V.Not_Before <= Now and then Now <= V.Not_After;
end Is_Valid;
```

Listing 5.17: Certificate validity check

To initiate the trust chain validation process in the TKM, a new CC context must be instantiated by calling the `Cc_Set_User_Certificate` remote procedure call as illustrated by figure 5.5. This call stores the user certificate in the CC for which trust must be established. Before storing the user certificate in the context, the validity is checked.

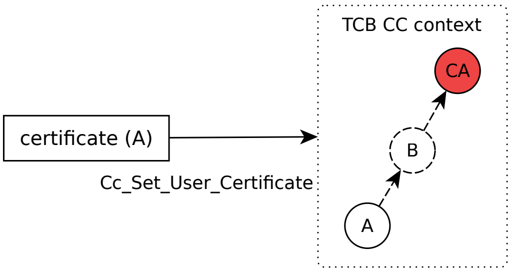


Figure 5.5: TKM trust chain set user certificate

Intermediate CAs and the final CA are added to the CC context by calling the `Cc_Add_Certificate` remote procedure call as shown by figure 5.6.

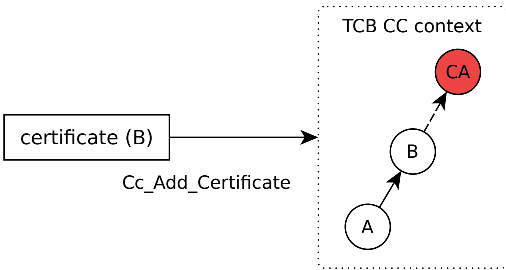


Figure 5.6: TKM trust chain add certificates

The TKM checks the validity of the intermediate CA (certificate $B$ in this example) and performs a signature verification of the signature stored in the user certificate $A$ using the public key of $B$. The signature is checked using a RSA verifier. If the signature verifies, the intermediate certificate $B$ is stored in the context along with the user certificate $A$.

The `Cc_Add_Certificate` procedure must be called multiple times for all intermediate CAs in the trust chain and also for the final root CA. The ordering of certificates delivered to the TKM is performed by the charon-tkm bus listener plugin. If the ordering is incorrect, the verification of the chain fails and the IKE SA can not be authenticated.

The next step is to link the intermediate certificate $B$ with the certificate $CA$, which is also handed to the trusted part by charon-tkm using `Cc_Add_Certificate`. The signature contained in certificate $B$ must be validated using the public key stored in the received $CA$ certificate. If the verification is successful, the last step is to check that the top-level certificate matches the trusted root $CA$, this is done by calling the `Cc_Check_Ca` RPC as shown by figure 5.7. The last certificate added by the `Cc_Add_Certificate` must be bit-wise identical to the CA trusted by the TKM. If this check succeeds, the CC context is set into the *checked* state and the context can be used to verify signatures.

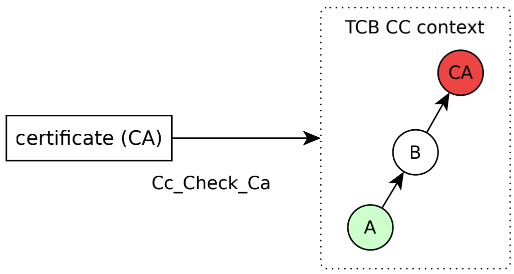


Figure 5.7: TKM trust chain check CA

### 5.5.8 Kernel SPD/SAD management

Since the Linux kernel stores sensitive keying material in its security-association database, the untrusted part is not allowed to access these databases. This must be assured by security mechanisms which are outside of the scope of this document. But as a result, the TKM must manage the kernel's security-policy (SPD) and security-association (SAD) databases itself.

The xfrm-ada project (5.7.2), which has been developed during this TKM project, is used to install security policies on TKM startup and also to manage SA states.

## 5.6 xfrm-proxy

The xfrm-proxy component uses the xfrm-ada library (5.7.2) to communicate with charon's EES service (5.4.12). See figure 5.8 for an overview of the proxy architecture.

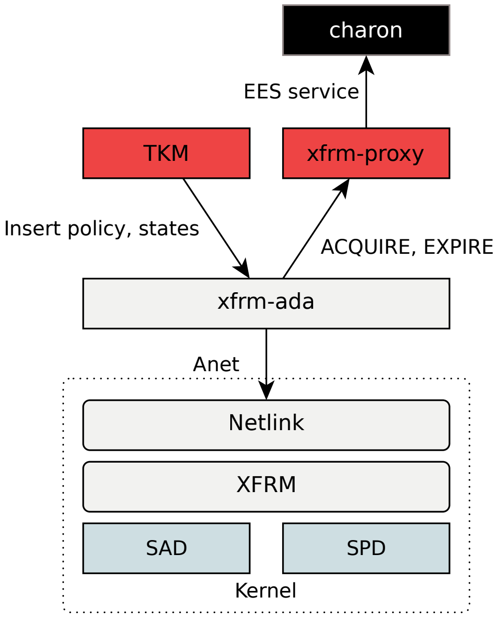


Figure 5.8: XFRM proxy architecture

As stated before, the kernel stores critical IPsec policies and SA states, therefore the charon daemon is no longer allowed to communicate with the kernel XFRM subsystem.

To make rekeying work in such a scenario, kernel XFRM acquire and expire messages must be delivered to charon by other means. The xfrm-proxy component subscribes to the kernel's XFRM subsystem acquire and expire multicast groups to receive events and delivers them to charon using the EES service. Charon then starts create or rekeying jobs for the IPsec policy or SA in question as usual.

## 5.7 Additional components

Certain functionality which was needed for the implementation of the TKM has been realized in self-contained software projects or as extension to existing libraries.

### 5.7.1 Anet

Anet is a networking library for the Ada programming language. It is used by the Trusted Key Manager and xfrm-proxy to open or connect to Unix sockets and communicate with charon-tkm.

Anet has been released as open-source software under the GMGPL[13] license and is available at http://git.codelabs.ch/?p=anet.git.

[13] GNAT Modified General Public License

### 5.7.2 xfrm-ada

This project is an Ada binding to Linux's XFRM kernel[14] interface. It provides the functionality required to add and delete XFRM policies and states.

The XFRM framework is used to manage the IPsec protocol suite in the Linux kernel. The XFRM states operate on the Security Association Database (SAD) and the XFRM policies operate on the Security Policy Database (SPD). Among other features, it provides ESP [17] payload encryption with the key material provided by an userspace application.

The TKM uses the XFRM interface via the xfrm-ada library to manage the SPD and SAD and provide keys for ESP encryption to the kernel.

xfrm-ada has been released as open-source software under the GMGPL license and is available at `http://git.codelabs.ch/?p=xfrm-ada.git`.

### 5.7.3 x509-Ada

This project is an Ada PKIX X.509 [6] library. It provides functionality to process ASN.1/DER-encoded [13, 14] certificates and private keys.

x509-Ada has been released as open-source software under the GMGPL license and is available at `http://git.codelabs.ch/?p=x509-ada.git`.

## 5.8 Limitations

This section describes the limitations of the current realization of the design outlined in chapter 3. The main reason for these limitations is the lack of time to fully implement the envisioned functionality and are not due to inadequate or deficient design.

### 5.8.1 Cryptographic algorithms

Currently only a selected set of algorithms are implemented. Table 5.1 lists the implemented cryptographic transforms:

| **Usage** | **Algorithm name** | **IANA ID** |
|---|---|---|
| Authentication method | RSA-PKCS1-SHA1 | 1 |
| Certificate chain verification | RSA-PKCS1-SHA256 | 1 |
| Encryption Algorithm | AES-256-CBC | 12 |
| Pseudo-random Function | HMAC-SHA512 | 7 |
| Integrity Algorithm | HMAC-SHA512 | 14 |
| Diffie-Hellman | 3072-bit MODP Group | 15 |
| Diffie-Hellman | 4096-bit MODP Group | 16 |

Table 5.1: Implemented cryptographic algorithms

There is no inherent limitation of usable cryptographic transforms, it is simply a question of implementing the desired methods. Algorithm agility is ensured by the design through the use of numeric algorithm identifiers and avoidance of hard-coded cryptographic mechanisms.

[14] `http://www.kernel.org/`

### 5.8.2 Identity handling

The local and peer identities are currently limited to specific, hard-coded identities. The peer subject name must be "bob@strongswan.org" and the local subject name must be "alice@strongswan.org". This is caused by the static configurability of the TKM daemon and the current private key handling of charon. In order to allow the use of arbitrary identities the configuration mechanism of TKM and charon-tkm needs to be fully implemented and a TKM credential set must be implemented.

### 5.8.3 Certificates and keys

Currently only a single CA certificate is supported for certificate chain validation. Similarly only one private key is supported for authenticating the local identity to the peer. Akin to the constraints with regards to identity handling, the cause for this is also the incomplete implementation of the configuration interface.

Additionally the currently implemented validity checks of certificates are only rudimentary.

### 5.8.4 Certificate chain context reuse

A certificate chain that has been verified, is potentially usable until the end of its validity period. Currently this fact is disregarded and verified certificate chain contexts are not reused and must be constructed anew when authenticating a peer.

### 5.8.5 Source of randomness

The nonce generation in the TKM is implemented by reading a sequence of bytes from Linux's random device node `/dev/urandom`. The source of the random data is currently not configurable. This may not be regarded as a limitation per-se but because the issue of random number generation is paramount to any system constructing cryptographic keys the authors feel compelled to explicitly mention it.

### 5.8.6 Exception mapping

If a processing error on the server-side occurs the status code of the reply message is always set to Invalid_Operation. To provide the client with more specific information about the error exceptions should be inspected and mapped to their corresponding failure code.

## 5.9 Conformance to requirements

This section describes how the implementation meets the design requirements defined in section 3.5.

- The requirement 3.5.1 demands that code running in the TCB must be as minimal and robust as possible. This has been addressed by applying the following measures:

  - Use of the Ada programming language and avoidance of problematic language constructs (like type extensions, dynamic memory allocation etc.).
  - Use of agile development methods, i.e. test-driven development, pair programming and code reviews.
  - Automatic generation of interface code from XML specification, avoiding implementation errors by verifying the generated code.
  - Use of Ada 2012 contracts to confine generated context state machine code.

- The separation and communication requirements 3.5.2, 3.5.3 demand that the untrusted and trusted parts of the system are separated and communication is only possible over a well-defined, minimal interface. These requirements are guaranteed by automatically creating the interface code from an XML-specification as described in section 5.2 and by using a simple library providing RPC services by exporting the generated interface over Unix domain sockets (see section 5.3).

- Requirements 3.5.4 and 3.5.5 require that the untrusted part must not have access to critical keying material and that the cryptographic operations using this material must be implemented in the TCB to guarantee proper operation. These requirements are fulfilled in the design by implementing plugins which act as proxy objects between the untrusted charon-tkm daemon and the TKM. These plugins operate with references to the real, sensitive data and are kept very simple. No sensitive data leaves the TCB. This directly demands that critical cryptographic operations used to either create sensitive material or operating on sensitive material must be implemented in the TCB as well. The following TKM-specific strongSwan plugins are responsible to achieve the desired degree of separation:

  - Nonce generation plugin (5.4.5)
  - DH plugin (5.4.6)
  - Keymat plugin (5.4.7)
  - Kernel IPsec plugin (5.4.8)

- Requirement 3.5.6 requires that the TCB must enforce proper authentication. The system supports strong authentication methods based on public-key cryptography only. The secret private key required to create valid signatures and the trusted CA certificate used to verify the peer's authentication data must reside in the TCB. To make this separation possible, the following TKM-specific strongSwan plugins are implemented:

  - Private key plugin (5.4.9)
  - Public key plugin (5.4.10)
  - Bus listener plugin (5.4.11)

- Requirement 3.5.7 demands that a misbehaving untrusted part is not able to violate the security properties guaranteed by the TCB. As a formal analysis of the proposed IKEv2 separation protocol has not been performed, this property is only assumed but not formally proven, see also section 5.8.

# Chapter 6

# Conclusion

This chapter provides a summary of the contributions and an outlook on possible future work.

## 6.1 Contributions

This section discusses the main results of this work which are the analysis and splitting of the IKE protocol and demonstrating the viability of the concept through the prototypical implementation of the envisioned system.

### 6.1.1 IKE protocol split

After formulating desired security properties and identifying the critical components of the IKE protocol a concept to split the key management system into an untrusted and trusted part has been proposed. Care was taken to only extract the functionality that is absolutely necessary from the untrusted IKE processing. Thus, the presented interface between IKE and TKM facilitates the implementation of a small and robust trusted component. This interface has been specified in an XML document which is used as a basis for the implementation.

The splitting of the protocol guarantees that even if the untrusted side is completely subverted by an attacker the TCB upholds the proposed security goals.

### 6.1.2 Prototype implementation

The IKEv2 separation design proposed in this paper has been implemented and demonstrated to be a viable solution to attain a higher level of security. The untrusted parts of the IKE daemon are implemented on top of the existing strongSwan IKE implementation while the trusted components have been implemented from scratch using the Ada programming language.

Leveraging the XML specification of the interface and using it to automatically generate code for the IKE and TKM, errors in the transformation process from the specification to the code are avoided. This mechanism enables changes to the interface at a low cost with a significantly smaller potential for errors compared to a manual translation of the specification into code. Since the implementation spans multiple programming languages (Ada and C) this takes even more burden off the implementer.

Generating Ada 2012 contracts from the specification of the code used in the TCB, the conformance of these parts of the TKM implementation are checked against the specification at runtime. Additionally these checks can also be formally verified by the GNATprove tool (see section 5.2).

## 6.2 Future work

This section outlines planned and possible future steps to improve upon the foundation of the current TKM implementation. In the first part of this section concrete work items are discussed which are planned to be implemented soon. These steps directly address the limitations presented in section 5.8.

The latter part discusses broader issues which aim to address the correct enforcement of assumptions formulated in section 3.3.

### 6.2.1 Credential set

The private key handling of charon-tkm must be extended with a TKM specific credential set to allow the usage of private keys with different subject than alice@strongswan.org. The set should provide an own implementation of a private key enumeration function (`create_private_enumerator` of `credential_set_t`). This way a configured private key could be fetched and installed in the credential manager on demand.

### 6.2.2 Exception mapping

As described in section 5.3.4.2, exceptions which are raised during processing of a client request are handled by the Process_Stream generic. Exceptions should be mapped to their corresponding failure code (see `result_type` constants specified in section 4.2.3) and the status code of the response set accordingly. This gives the client more information about what kind of processing error occurred.

### 6.2.3 Additional checks for generated key material

The sanity checks for generated Diffie-Hellman values and cryptographic keys should be augmented to avoid the usage of problematic key material undermining the employed encryption or integrity protection mechanisms.

### 6.2.4 Validation of certificates

Additional checks outlined in [6, 8, 24] must be implemented to more accurately verify the validity of certificates and certificate chains.

### 6.2.5 Configuration subsystem

Most deficiencies enumerated in section 5.8 can be rectified by making the current implementation more dynamically configurable. This would allow the usage of TKM in many more scenarios which would considerably broaden the applicability of the presented solution. It is expected that this will be implemented in the near future.

### 6.2.6 Automated tests

Even though the whole TKM system has been developed following the test-driven development[1] methodology, no automated integration test suit has been built. This was partly because the system is built-up by many different components and partly because of some deficiencies in the current testing framework of strongSwan, which impeded the addition of automated test cases.

In the meantime the infrastructure for the automated test of strongSwan has been improved. Once these changes are finalized, TKM-specific test cases will be added to allow the automated and reproducible testing of the whole TKM system. This ensures that changes to parts of the system are detectable and clearly indicated by failing tests.

### 6.2.7 Cryptanalytic review

A formal and rigorous cryptographic analysis of the "Splitting" and the communication between IKE and TKM is highly desirable. It is assumed that an adversary cannot somehow obtain or deduce key material or other sensitive information (see section 3.1) either using the data freely available to IKE or performing exchanges as specified by the interface to extract additional information from the TKM. Furthermore Man-in-the-Middle attacks must also be prevented.

In this context, a critical operation is the `isa_sign` exchange specified in section 4.3.1.11, since it is used to sign data (authentication octets) with a private key to assert the local identity to the peer. Parts of the input data for the signature are private to the TKM while other elements are fixed but known to the untrusted side. Yet another portion of the input can be chosen arbitrarily by an adversary assuming the role of IKE.

By repeatedly performing the aforementioned `isa_sign` exchange a malicious entity can abuse the TKM as a random oracle and mount an *adaptive chosen-plaintext attack*. Employing a signature algorithm which is resistant against such attacks should ensure the desired security properties but since a successful attack would nullify the security properties of the TKM system this issue must be analyzed with great care.

### 6.2.8 Platform integration

The basic premise for the extraction of security critical functionality into a TCB is, as stated in section 3.2, that the untrusted charon-tkm daemon can only interact with the trusted TKM using the exchanges specified in the interface description (see section 4.3.1). For high assurance systems, the process isolation mechanism of a standard Linux system is not adequate and does not provide the necessary level of separation, as was already mentioned in section 3.3. Additionally, in an ordinary operating system the amount of code that has

[1] TDD is a software development process which employs on short development cycles with a focus on writing good unit tests.

to be counted to the trusted computing base is in the range of hundred thousand or possibly even millions of lines of code. This stands in stark contrast to the demand that the TCB should be *minimal*.

Integrating the implemented system into an environment that offers superior isolation mechanisms and a smaller TCB size is thus a requirement to actually attain a higher level of security. Table 6.1 lists potential technologies or mechanisms which could be used to complement the separated IKE system.

| **Name** | **Reference** |
|---|---|
| Physical separation | none |
| Linux Containers | http://lxc.sourceforge.net/ |
| SELinux | http://selinuxproject.org/ |
| Separation Kernel | [26] |

Table 6.1: Possible target IKE/TKM platforms

Putting charon-tkm and the TKM daemon on physically distinct hosts is appealing because it is apparent, that charon-tkm and TKM can only exchange information via the intended communication channels. Additionally, such a system is expected to be fairly straight forward to implement thanks to the transport layer abstraction in the tkm-rpc library described in section 5.3.3.1. The need for additional hardware is a major drawback.

The same level of separation could be achieved by porting the charon-tkm and TKM daemon components to a separation kernel (SK). Unfortunately there are no freely available SKs at the time of this writing. Some commercial products exist but they are only available to paying customers.

Linux containers and SELinux are another possible solution to secure the trusted from the untrusted part. The degree of isolation they offer might be enough for certain usage scenarios. Using these mechanisms would however not address the issue of having a large TCB.

# Index

# Bibliography

[1] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, ASIACCS '07, pages 312–320, New York, NY, USA, 2007. ACM.

[2] Ada Rapporteur Group (ARG). *Ada Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652:2012 (E)*. ISO, 2012. http://www.ada-auth.org/standards/ada12.html.

[3] AdaCore. Project Hi-Lite / GNATprove. http://www.open-do.org/projects/hi-lite/gnatprove/, 2012. [Online; accessed 04-December-2012].

[4] Endre Bangerter, David Gullasch, and Stephan Krenn. Cache Games - Bringing Access Based Cache Attacks on AES to Practice. Cryptology ePrint Archive, Report 2010/594, 2010. http://eprint.iacr.org/2010/594.

[5] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 189–202, New York, NY, USA, 2011. ACM.

[6] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008.

[7] Cas Cremers. Key exchange in IPsec revisited: formal analysis of IKEv1 and IKEv2. In *Proceedings of the 16th European conference on Research in computer security*, ESORICS'11, pages 315–334, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *ACM Conference on Computer and Communications Security*, pages 38–49, 2012.

[9] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409 (Proposed Standard), November 1998. Obsoleted by RFC 4306, updated by RFC 4109.

[10] High Order Language Working Group, Department of Defense. Department of Defense Requirements for High Order Computer Programming Languages: Steelman. Technical report, United States Department of Defense, June 1978.

[11] Jean Ichbiah. Reference Manual for the Ada Programming Language. *ANSI/MIL-STD-1815A-1983*, 1983.

[12] IEEE. *IEEE 1003.1-2008 - IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R))*. IEEE, December 2008.

[13] International Telecommunication Union ITU. Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation. Series x: Data networks, open system communications and security directory, International Telecommunication Union, Geneva, Switzerland, oct 2011. ITU-T Recommendation X.680 (2011) - Technical Corrigendum 1.

[14] International Telecommunication Union ITU. Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). Series x: Data networks, open system communications and security directory, International Telecommunication Union, Geneva, Switzerland, oct 2011. ITU-T Recommendation X.690 (2011) - Technical Corrigendum 1.

[15] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), February 2003.

[16] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996 (Proposed Standard), September 2010. Updated by RFC 5998.

[17] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303 (Proposed Standard), December 2005.

[18] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005. Updated by RFC 6040.

[19] T. Kivinen and M. Kojo. More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE). RFC 3526 (Proposed Standard), May 2003.

[20] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. *SIGOPS Oper. Syst. Rev.*, 25(5):165–182, September 1991.

[21] D. McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116 (Proposed Standard), January 2008.

[22] Catherine Meadows. Analysis of the Internet Key Exchange Protocol using the NRL Protocol Analyzer. In *IEEE Symposium on Security and Privacy*, pages 216–231, 1999.

[23] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, October 1992.

[24] E. Rescorla. HTTP Over TLS. RFC 2818 (Informational), May 2000. Updated by RFC 5785.

[25] Reto Buerki, Robert Dorn, Adrian-Ken Rueegsegger. Split of IKEv2 Services into a Trusted and a Semi-Trusted Component. http://www.secunet.com/, 2011.

[26] J. M. Rushby. Design and verification of secure systems. *SIGOPS Oper. Syst. Rev.*, 15(5):12–21, December 1981.

[27] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM.