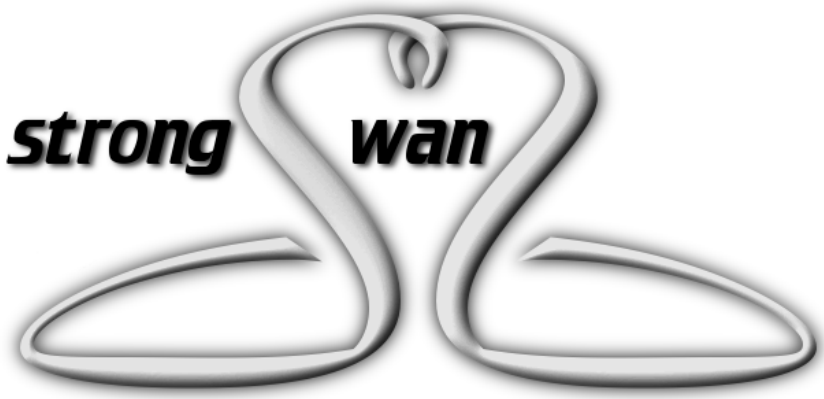
Diplomarbeit

strongSwan II
Eine IKEv2-Implementierung für Linux



Jan Hutter
Martin Willi

Betreut durch Prof. Dr. Andreas Steffen

16. Dezember 2005
Hochschule für Technik Rapperswil

Abstract

Ziel der Arbeit

Das Internet Key Exchange Protokoll (IKE) wird für den Aufbau von IPsec-Verbindungen eingesetzt. Es tauscht die dafür notwendigen Schlüssel aus und vereinbart die zu verwendenden Algorithmen. Die Version 2 des Protokolls ist definiert und wird IKEv1 in naher Zukunft ablösen. Der Meldungsaustausch wurde vereinfacht und das Protokoll flexibler aufgebaut.

Die auf Linux basierende OpenSource-Software strongSwan ist weit verbreitet und verwendet IKE in der Version 1. Die Unterstützung von IKEv2 ist erforderlich, damit strongSwan auch in Zukunft eine starke Alternative zu kommerziellen Produkten bleibt.

Mit dieser Diplomarbeit soll strongSwan um das IKEv2-Protokoll erweitert werden. Eine Integration in den bestehenden Code wäre allerdings ungünstig. Die Übersichtlichkeit würde zu stark darunter leiden, auch erscheint dessen Aufbau nicht mehr ganz zeitgemäss. Es soll daher eine neue Architektur für strongSwan erstellt und darin die IKEv2-Unterstützung implementiert werden.

Ergebnisse

Im Rahmen der Diplomarbeit ist ein Daemon für Linux entstanden, welcher IKE in der neuen Version bereits zu grossen Teilen implementiert. Die völlig neu konzipierte Architektur basiert auf einem Thread-Pool, um optimal zu skalieren. Der in C geschriebene Code nutzt objektorientierte Prinzipien, wodurch er übersichtlich und einfach erweiterbar ist.

Der Aufbau einer Security Assocation für IKE wurde implementiert. Die Authentisierung kann dabei über Shared-Secrets oder RSA-Schlüssel erfolgen. Parameter für die IPsec-Verbindungen werden ausgehandelt. Für eine resultierende Kommunikation über AH oder ESP fehlt noch die Möglichkeit zur Konfiguration des Kernels. Mit einem Proof-of-Concept der Kernel-Schnittstelle wurde hier jedoch wichtige Vorarbeit geleistet.

Mit der entstandenen Software ist eine solide Grundlage für strongSwan II geschaffen worden.

Ausblick

Als nächster Schritt zur Weiterentwicklung von strongSwan II steht die Kernel-Anbindung des Daemons im Vordergrund. Diese würde bereits den vollständige Aufbau von IPsec-Verbindungen ermöglichen.

Später soll durch die Integration einer mächtigen Konfigurations-Schnittstelle die Verwendbarkeit im normalen Betrieb ermöglicht werden.

Längerfristig ist eine komplette Kompatibilität zu IKEv2 erstrebenswert. Eine zusätzliche Abwärtskompatibliltät zu IKEv1 soll dabei nicht ausser Acht gelassen werden.

Inhaltsverzeichnis

1 Aufgabenstellung

Der nachfolgende Text entspricht der Aufgabenstellung, wie sie von Prof. Dr. Andreas Steffen formuliert wurde.

1.1 Einführung

Die Stärke der Linux OpenSource VPN Lösung `strongSwan` liegt in der Benutzer- und Host-Authentisierung mittels X.509 Zertifikaten. `pluto`, der `IKE` Dämon von `strongSwan` basiert auf Version 1 des Internet Key Exchange Protokolls (`IKE`, RFC 2409), das nun im Verlauf der nächsten Jahre durch das im Oktober 2004 fertig standardisierte `IKEv2` Protokoll (draft-ietf-ipsec-ikev2-17.txt) abgelöst werden soll.

In dieser Diplomarbeit soll mit `strongSwan II` ein `IKEv2` Dämon in Angriff genommen werden, der über die bestehende `NETKEY` Schnittstelle die eingebaute IPsec Implementation des Linux 2.6 Kernels konfigurieren und steuern kann. An der bestehenden „whack“ Benutzer-Schnittstelle soll vom Grundkonzept her festgehalten werden. Damit aber der aus dem `FreeS/WAN` Projekt stammende Code abgespeckt und modernisiert werden kann, besteht die Freiheit, sowohl die Finite-State-Machine des `IKEv2` Dämons, wie auch die internen Datenstrukturen völlig neu zu konzipieren.

Im Zentrum der Diplomarbeit soll die Peer-Authentisierung auf der Basis von RSA Signaturen und Zertifikaten stehen. Dazu sollen wenn immer möglich bestehende Krypto- und Zertifikatsfunktionen aus der `strongSwan` Distribution verwendet werden. Für Multi-Precision Operationen soll auf die `GMP` Library zurückgegriffen werden. Keine `OpenSSL` Bibliotheksfunktionen!

1.2 Aufgabenstellung

- Entwurf eines Grundkonzepts für den `IKEv2` Dämon.
- Festlegung der zu realisierenden `IKEv2` Leistungsmerkmale. Um den Erfolg der Diplomarbeit zu sichern, soll nur das für einen Proof-of-Concept absolut-notwendige Minimum von Features implementiert werden.
- Entwickeln und Testen eines `IKEv2` Minimal-Dämons in der Programmiersprache C.
- Coding Rules: `/* English comments */`, alle Variablendeklarationen sollen am Beginn eines Blocks stehen.

1.3 Links

- Internet Key Exchange (IKEv2) Protocol:
  http://www.ietf.org/internet-drafts/draft-ietf-ipsec-ikev2-17.txt
- Cryptographic Algorithms for use in IKEv2:
  http://www.ietf.org/internet-drafts/draft-ietf-ipsec-ikev2-algorithms-05.txt
- strongSwan Projekt:
  http://www.strongswan.org/
- GMP Projekt:
  http://swox.com/gmp/
- Linux IKEv2 Projekt:
  http://sourceforge.net/projects/ikev2/
- FreeBSD Racoon 2 Projekt:
  http://www.freshports.org/security/racoon2/

Rapperswil, 24. Oktober 2005

Prof. Dr. Andreas Steffen

2 Management Summary

2.1 Ausgangslage

Virtual Private Networks

Immer mehr Firmen nutzen das Internet um ihrer Standorte kostengünstig zu vernetzen. Das Schlagwort in diesem Bereich heisst VPN und steht für Virtual Private Network. Ein solches VPN wird meistens mit IPsec realisiert, welches das Internet-Protokoll (IP) um Sicherheitsfunktionen wie Vertraulichkeit und Integrität erweitert.

IKE

Um eine Verbindung über IPsec herzustellen, müssen sich die Kommunikationspartner vorgängig über diverse Details wie verwendete Schlüssel und Algorithmen einigen. Dazu wird das Internet Key Exchange-Protokoll (`IKE`) eingesetzt.

Komplex und unflexibel

In der Praxis hat sich gezeigt, dass die Grundfunktionalität von `IKE` zu unflexibel ist, woraufhin diverse proprietäre Erweiterungen entstanden sind. Leider sind diese teilweise als unsicher zu betrachten. Auch wird `IKE` als zu komplex und somit fehleranfällig beurteilt.

Version 2 behebt Fehler

Die Version 2 von `IKE` behebt die Fehler der Vorgängerversion. Unsichere Eigenschaften wurden entfernt und nützliche Funktionen hinzugefügt. Es ist zu erwarten, dass das neue `IKE`-Protokoll das alte in den nächsten Jahren ablösen wird.

strongSwan

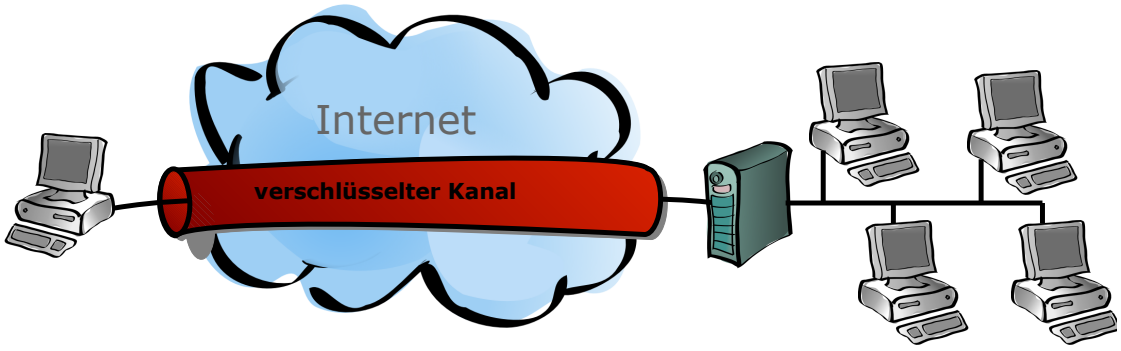
Die etablierte OpenSource-Software `strongSwan` bietet unter Linux eine `IKE`-Implementierung der Version 1. Damit `strongSwan` auch in Zukunft eine starke Alternative zu anderen Produkten bleibt ist eine Unterstützung des `IKE`-Protokolls in der Version 2 erstrebenswert.

Neue Architektur

Die Architektur von `strongSwan` ist in den vielen Jahren evolutionär gewachsen und deshalb unübersichtlich geworden. Darum soll mit dieser Diplomarbeit eine neue Architektur für `strongSwan` definiert und in der Programmiersprache C implementiert werden. Die neu aufgebaute Architektur soll einfach zu erweitern sein und die zukünftige Basis von `strongSwan II` bilden.

Unterstützung von IKEv2

Des Weiteren soll die Unterstützung für das neue `IKE`-Protokoll in der minimalsten Form implementiert werden.



IKEv2 ermöglicht den automatischen Aufbau eines verschlüsselten Kanals auch über unsichere Netzwerke wie das Internet.

2.2 Vorgehen

Einarbeitung

Die Implementierung eines Protokolls setzt detaillierte theoretische Kenntnisse der entsprechenden Technologien voraus. Die erste Phase der Arbeit wurde dazu genutzt, um sich in die gesamte Materie einzuarbeiten. Etliche Standards wurden gelesen und der bisherige strongSwan-Code analysiert.

Design der Software

Die gewonnenen Erkenntnisse ermöglichten ein erstes Design der neuen Software. Grundsatzentscheidungen wie die Wahl des Threading-Modells konnten getroffen werden. Des weiteren wurde entschieden, die neue Software unabhängig von strongSwan von Grund auf neu zu entwickeln.

Schrittweise Implementierung

Die Implementierung erfolgte schrittweise über drei Phasen hinweg. Die erste Phase hatte zum Ziel, die Grundarchitektur der Software zur erstellen. Auf dieser aufbauend konnten die geplanten Funktionalitäten von IKEv2 über zwei Phasen hinweg implementiert werden.

Betreuung

Diese Diplomarbeit wurde von Prof. Dr. Andreas Steffen betreut. In wöchentlichen Sitzungen wurde der Stand der Arbeit aufgezeigt und Probleme besprochen.

2.3 Ergebnisse

„charon“ als Grundlage

Mit „charon“ wurde die Grundlage für die IKEv2-Unterstützung von strongSwan II geschaffen. Die Linux-Software ist modular und übersichtlich aufgebaut und unterstützt das IKE–Protokoll in der Version 2 in einer minimalen Form.

Multi-Threaded

Gegenüber der single-threaded Event-Queue des alten IKE-Dämons von strongSwan basiert die Architektur auf einem Multi-Threading-Modell.

Name aus Mythologie

Sein Name hat der Dämon aus der griechischen Mythologie, in welcher Charon den Fährmann zwischen dieser und der Unterwelt verkörpert.

Einfach zu erweitern

Wiederverwendbare Algorithmen, wie beispielsweise zur Verschlüsselung, sind aus dem bestehenden strongSwan-Code übernommen und an die neue Architektur angepasst. Die dabei eingesetzten objektorientierten Prinzipien machen den Code verständlicher und lassen ihn einfach erweitern.

strongSwan II im Aufbau

Mit strongSwan II ist eine Software im Aufbau, welche hoffentlich breite Verwendung findet und von der OpenSource-Community weitergepflegt wird.

2.4 Ausblick

Funktionalitäten abschliessen

Manche Funktionalitäten konnten während der Diplomarbeit nicht vollständig implementiert werden. In einem nächsten Schritt gilt es somit, diese noch offenen Punkte fertig zu implementieren, beispielsweise in einer Studienarbeit.

IKEv2-Kompatibilität

Für eine vollständige IKEv2-Kompatibilität fehlen noch bestimmte Funktionalitäten, wie beispielsweise die Unterstützung digitaler Zertifikate. Weiterführendes Ziel sollte es daher sein, diese Features zu implementieren und damit die volle IKEv2-Kompatibilität zu erreichen.

Integration von IKEv1

Da die Ablösung des älteren IKE–Protokolls zum neuen fliessend erfolgen wird, wäre die Unterstützung von IKEv1 in strongSwan II wünschenswert. Sinnvoll wäre hier die Integration von strongSwan I.

3 Einleitung

3.1 Problemstellung

Die Zielsetzung der Diplomarbeit „strongSwan II - Eine IKEv2 Implementierung für Linux“ war es, die IPsec-Implementierung strongSwan um die Fähigkeiten des neuen IKEv2 Protokolls zu erweitern.

Das Internet Key Exchange Protocol, kurz IKE, wird verwendet, um den Schlüsselaustausch für die IPsec-Kommunikation zu automatisieren. Mit dessen Hilfe können zwei Rechner auf Basis von Shared-Secrets oder Zertifikaten einen sicheren Schlüsselaustausch realisieren. Neben den Schlüsseln müssen sich die beiden Parteien auf Algorithmen einigen und Richtlinien für die Kommunikation aushandeln. IKE setzt die Verbindungen für IPsec nicht nur auf, sondern verwaltet sie und baut sie auch wieder ab.

IKEv1 basiert auf verschiedenen Standards und ist sehr umfangreich. Es gilt zwar nicht als unsicher, wurde aber wegen seiner unnötigen Komplexität oft kritisiert. Durch die Unflexibilität des Protokolls sind zudem Erweiterungen entstanden, die unsicher sind. So ist z.B. das weit verbreitet XAUTH unter Umständen anfällig gegen „Man in the middle“-Attacken. Des weiteren ist der „Agressive-Mode“ anfällig gegen die Ausspionage von Identitäten.

IKEv2 wurde entwickelt, um die Schwächen von IKE in der Version 1 auszumerzen. Es ist in einem einzigen Dokument beschrieben und lässt dadurch viele Altlasten fallen. Wo möglich wurde es vereinfacht, wodurch es ebenfalls effizienter und logischer arbeitet. Die diversen Modi von IKEv1 wurden ersetzt. Der Aufbau der Verbindung wurde drastisch verkürzt, ohne dabei die Sicherheit zu gefährden. Um die Authentisierung über verschiedenste Varianten abzuwickeln, ist es möglich die Methoden von EAP zu verwenden. Somit steht ein flexibles Authentisierungs-system zur Verfügung, welches unsichere Methoden wie XAUTH überflüssig macht.

strongSwan ist eine schon ältere Software. Sie ist ein Abkömmling des nicht mehr fortgesetzten FreeS/WAN-Projektes. Sicherlich hat sie dadurch eine sehr hohe Stabilität erreicht, allerdings ist die Architektur im Kern single-threaded und dadurch schwer zu erweitern. Da die Änderungen am IKE-Protokoll zur Version 2 nicht unbeachtlich sind, wäre eine direkte Integration des Protokolls in den jetzigen Key-Daemon pluto weniger sinnvoll. Es würde den Code nur weiter aufblasen und absolut unübersichtlich werden lassen. Für diese Diplomarbeit wurde deshalb entschieden, IKEv2 in einer neuen Architektur aufzubauen. Diese kann auf heute üblichen Technologien aufsetzen und multi-threaded arbeiten.

Da auch IKEv2 ein umfangreiches Protokoll ist, mussten die Ziele klar abgesteckt werden. Viel wichtiger als viel Funktionalität war ein sauberes Design, welches als Basis für die nächste Generation eines Key-Daemons für IPsec verwendet werden kann. Es wurde entschieden, sich auf die grundlegende Funktionalität zu beschränken. Neben dem Aufbau einer modernen Architektur soll der Aufbau einer Security-Association für IKE realisiert werden.

3.2 Gliederung der Dokumentation

Die Gliederung der ganzen Dokumentation ist in mehrere Teile aufgeteilt. Der erste Teil der Dokumentation bietet einen Überblick. Dazu gehören neben dem Abstract das Inhaltsverzeichnis, die vom Betreuer formulierte Aufgabenstellung, eine Management-Summary sowie eine Einleitung in die Problematik. Der zweite Teil bildet den Technischen Bericht und erstreckt sich über mehrere Dokumente:

- Das Dokument „Technologien“ beschreibt eingesetzte oder für das Verständnis erforderliche Technologien.
- Im Dokument „Design“ wird auf die realisierte Architektur eingegangen. Es beschreibt detailliert den Aufbau der Software.
- Das Dokument „Test“ beschreibt die eingesetzten Testverfahren, um die Qualität der Software sicherzustellen.
- Die Resultate der Arbeit sind im Dokument „Projektstand“ beschrieben. Es gibt einen Überblick über die realisierten Funktionen und dient als Wegweiser für die weitere Entwicklung.

Im dritten Teil sind die persönlichen Erfahrungsberichte sowie Dokumente des Projektmanagements zu finden. Zu letzteren zählen Projektplan, Risikoanalyse, Programmierrichtlinien und die Protokolle der Betreuersitzungen.

Der vierte und letzte Teil enthält den kompletten Anhang. Dazu gehören diverse Verzeichnisse, ein Überblick über die abgegebene CD-ROM, sowie eine kurze Anleitung zum Übersetzen und Ausführen des Quellcodes.

4 Technologien

Dieses Dokument beschreibt Technologien, die in strongSwan II Verwendung finden oder für das allgemeine Verständnis der Problematik erforderlich sind. Im Zentrum steht dabei das IKEv2 Protokoll. Des weiteren wird IKEv1 kurz beschrieben und die Unterschiede zum neuen Protokoll dargestellt. Eine Übersicht von strongSwan und dessen Key-Daemon pluto sollen einen Einblick geben, was für die Realisierung eines IKE-Daemons zu beachten ist. Ein Vergleich mit dem Dokument „Design“ lässt somit die Unterschiede klar werden, die zwischen dem älteren pluto und dem neu entworfenen Daemon charon entstanden sind.

Zudem werden Technologien erläutert, auf welchen strongSwan II aufbaut. Dazu gehört die Threading-Schnittstelle pthreads, die intensiv verwendet wurde. Auch Netlink wird kurz angeschnitten, worüber die Kommunikation mit dem Kernel erfolgt. Auf eine detailliertere Beschreibung der gmp-Library wurde verzichtet, da diese bereits in pluto verwendet wurde und somit weder für die Entwickler, noch für das Zielpublikum Neuland darstellt.

Insgesamt ist das Verständnis der hier beschriebenen Technologien notwendig, um die im „Design“ beschriebene Architektur und daraus resultierende Entscheidungen nachzuvollziehen.

4.1 IKEv1

IKEv1 wird in strongSwan und diversen anderen Implementierungen für den automatisierten Austausch von Schlüsseln und anderen Parametern verwendet. Ein grobes Verständnis ist erforderlich, um sowohl IKEv2 besser zu verstehen als auch den Aufbau von pluto leichter nachzuvollziehen.

4.1.1 Übersicht

Ehe zwei Kommunikationspartner Daten geschützt über IPsec übertragen können, müssen sich diese über die zu verwendenden Algorithmen und Schlüssel einigen. Diese Informationen sind notwendig, um die Daten beispielsweise zu verschlüsseln und so vor unberechtigtem Zugriff zu schützen. Die für eine Verbindung ausgehandelten Algorithmen und Schlüssel werden im Begriff „Security Association“ zusammengefasst.

Eine SA einer bestimmten IPsec–Verbindung kann auf einem IPsec-Client manuell konfiguriert werden. Dies ist allerdings mit einem enormen Konfigurationsaufwand verbunden, da eine SA mehrere Schlüssel für Verschlüsselung, Signierung, etc. umfasst. Auch sollten die verwendeten Schlüssel in bestimmten Zeitabständen gewechselt werden, um so Angreifern zu verunmöglichen, durch das Knacken eines einzelnen Schlüssels den gesamten Datenverkehr zu entschlüsseln. Diese Forderung ist mit manueller Konfiguration keinem Administrator zumutbar.

Hier kommt IKEv1 ins Spiel. IKEv1 ist ein Protokoll zur automatischen Verwaltung von Security Associations für das IPsec-Protokoll. Die für eine IPsec-Verbindung notwendigen zwei Security Associations (in beide Richtungen eine) werden mit IKEv1 automatisch verwaltet. Auch das gelegentliche Wechseln der einzelnen Schlüssel einer SA erfolgt mit der Verwendung von IKEv1 automatisch.

IKEv1 ist in insgesamt drei RFCs definiert:

RFC	Titel	Inhalt
[RFC2408]	Internet Security Association and Key Management Protocol (ISAKMP)	In diesem RFC wird ISAKMP beschrieben. ISAKMP ist ein Framework. Es definiert Prozeduren und Paketformate zur Verwaltung von Security Associations. Auf die spezifischen kryptographischen Verfahren wird explizit nicht eingegangen, da es sich nur um ein Framework handelt und somit nur den Rahmen definiert. Mehr zu ISAKMP ist unter 4.1.2 zu finden.
[RFC2407]	The Internet IP Security Domain of Interpretation for ISAKMP	Um das im [RFC2408] beschriebene ISAKMP-Framework einsetzen zu können, muss dieses für eine spezifische Anwendung, auch „Domain of Interpretation“ genannt, „konfiguriert“ werden. Zu diesen „Konfigurationen“ gehört beispielsweise die Definition von Verschlüsselungs-Identifikationen oder die Definition von anwendungsspezifischen Payload-Typen. Für die „Domain of Interpretation“ IPsec, welche übrigens die einzige für ISAKMP definierte DOI ist, sind diese Angaben im [RFC2407] definiert. Mehr zu IPsec DOI siehe 4.1.3.
[RFC2409]	The Internet Key Exchange (IKE)	Dieses RFC beschreibt das IKEv1-Protokoll. Es erläutert, wie das ISAKMP-Protokoll eingesetzt und wie die Authentisierung zweier Kommunikationspartner sichergestellt wird. Dabei wird das Diffie-Hellman-Verfahren zur Authentisierung, bekannt aus dem Oakley-Protokoll, eingesetzt.

Tabelle 1: RFCs rund um IKEv1

4.1.2 ISAKMP

Wie bereits unter 4.1 kurz erläutert, handelt es sich bei ISAKMP um ein Framework. Es beschreibt Prozeduren und Paketformate zum automatischen Aushandeln, Anpassen und Löschen von Security Associations. ISAKMP wurde mit dem Ziel entwickelt, als Grundlage für Schlüsselaustausch-Protokolle unterschiedlichster „Domain of Interpretations“ eingesetzt zu werden. In der Praxis hat sich ISAKMP jedoch nie richtig durchsetzen können, so dass es nur in IPsec als IKEv1 Verwendung findet.

Da ISAKMP als Framework definiert wurde, sind die darin definierten Prozeduren und Paketformate nicht vorgeschrieben. Stattdessen können sie, wo nötig, von der entsprechenden DOI angepasst werden. Was jedoch alle „Key Exchange“-Protokolle gemeinsam haben müssen, die das ISAKMP-Framework einsetzen, ist der allgemeine Aufbau eines Paketes mit einem definierten Header. Und sie müssen die Kommunikation über UDP auf dem Port 500 unterstützen.

Der Header des ISAKMP-Protokoll hat folgenden Aufbau, der somit auch für das IKEv1-Protokoll gilt:

<table>
<tr><th>00</th><th>01</th><th>02</th><th>03</th><th>04</th><th>05</th><th>06</th><th>07</th><th>08</th><th>09</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td colspan="32">Initiator Cookie
-</td></tr>
<tr><td colspan="32">Responder Cookie
-</td></tr>
<tr><td colspan="8">Next payload</td><td colspan="8">Major version</td><td colspan="8">Minor version</td><td colspan="8">Exchange Type</td><td colspan="8">Flags</td></tr>
<tr><td colspan="32">Message ID</td></tr>
<tr><td colspan="32">Length</td></tr>
</table>

Abbildung 1: Aufbau des ISAKMP-Headers

Die folgende Tabelle beschreibt kurz die Bedeutung der einzelnen Felder. Details können dem RFC von ISAKMP [RFC2408] entnommen werden.

Feldname	Grösse	Beschreibung
Initiator Cookie	8 Byte	Dieses Cookie ist dem Initiator zugeordnet.
Responder Cookie	8 Byte	Dieses Cookie ist dem Responder zugeordnet.
Next payload	1 Byte	Bestimmt die Art des ersten Payloads, welcher nach dem Header folgt. Ein ISAKMP-Paket kann mehrere Payloads nacheinander enthalten, abhängig vom Typ des Pakets (siehe Feld „Exchange Type“). ISAKMP definiert diverse Payload-Typen inklusive dessen Kodierung. Zudem erlaubt es einer DOI seine eigenen Payload-Typen zu definieren.
Major version	4 Bit	Major Version des ISAKMP-Protokolls. Da lediglich die Version 1 von ISAKMP existiert, hat dieses Feld auch den Wert 1.
Minor version	4 Bit	Minor Version des ISAKMP-Protokolls. Dieses Feld hat den Wert 0.
Exchange Type	1 Byte	Spezifiziert den Typ eines „Exchanges“. Anhand dieses Typs werden die Payload-Typen definiert.
Flags	1 Byte	Spezielle ISAKMP-Flags.
Message ID	4 Byte	Eindeutige Identifikation um einen aktuellen Status in der Phase 2 (Details zu den Phasen folgen weiter unten) zu identifizieren. Dieser Wert wird vom Initiator zufällig generiert.
Length	4 Byte	Länge des gesamten ISAKMP-Pakets, d.h. Header und alle Payloads.

Tabelle 2: Header eines ISAKMP-Pakets

ISAKMP unterscheidet zwei Phasen bei der Aushandlung von Security Associations. In der Phase 1 wird eine SA ausgehandelt, um den eigenen ISAKMP-Verkehr zu schützen, beispielsweise durch Verschlüsselung. In der zweiten Phase wird die SA aus Phase 1 genutzt, um mit dieser weitere SAs für andere Protokolle geschützt, d.h. im Falle von IKEv1 verschlüsselt, auszuhandeln. So kann IKEv1 die ausgehandelten SAs nutzen, um damit eine IPsec-Verbindung zu konfigurieren.

4.1.3 IPsec DOI

Da ISAKMP neben IPsec auch für andere „Domain of Interpretations“ als Grundlage für ein eigenes „Key Exchange“-Protokoll eingesetzt werden kann, müssen die verschiedenen DOIs unterschieden werden. Das IPsec DOI RFC [RFC2407] definiert, wie ISAKMP in IKEv1 für die IPsec-„Domain of Interpretation“ eingesetzt wird. Es definiert beispielsweise Identifikationsnummern von Verschlüsselungsalgorithmen, welche für IPsec, also in IKEv1, Gültigkeit haben. Was alles in diesem RFC spezifiziert wird, kann diesem selbst entnommen werden und wird hier nicht weiter erläutert.

4.1.4 Oakley

Oakley ist ein vorgeschlagenes Protokoll[1] zur Aushandlung von gemeinsamen Schlüsseln zwischen authentisierten Kommunikationspartnern auf der Basis des Diffie-Hellman-Verfahrens. Oakley ist in [RFC2412] beschrieben. IKEv1 hat Mechanismen von Oakley übernommen und beschreibt sie im eigenen RFC [RFC2407].

4.1.5 Modi

IKEv1 unterscheidet verschiedene Modi. Die Modi für die ISAKMP-Phase 1 sind „Main Mode“ und „Agressive Mode“. Diese beiden Modi werden verwendet um eine ISAKMP-SA auszuhandeln, die notwendig ist, um darüber SAs für IPsec auszuhandeln. Der „Main Mode“ benötigt mindestens sechs Meldungen zum erzeugen einer ISAKMP-SA. Im Gegensatz dazu benötigt der „Agressive Mode“ nur drei Meldungen, wobei jedoch die Identitäten von Initiator und Responder im Klartext übertragen werden.

Der „Main Mode“ ist in drei Meldungsaustausche aufgeteilt. Im ersten werden die kryptographischen Suiten ausgehandelt. Der Zweite führt einen Diffie-Hellman-Austausch durch, wodurch die Parteien in den Besitz eines gemeinsamen, geheimen Schlüssels kommen. Die Meldungen des dritten Austausches authentisieren die Kommunikationspartner.

Der Modus um in ISAKMP-Phase 2 eine SA für ein anderes Protokoll auszuhandeln, wird in IKEv1 „Quick Mode“ bezeichnet. Zusätzlich kennt IKEv1 in der Phase 2 auch noch einen „New Group Mode".

Über den „Quick Mode“ werden SAs für IPsec ausgehandelt. IKEv1 kann dabei SAs für AH, ESP und IPCOMP aushandeln. Standardmässig werden im „Quick Mode“ die in Phase 1 ausgehandelten Schlüssel zur Verschlüsselung der Daten verwendet.

Weitere Details zu den verschiedenen Modi können dem RFC entnommen werden.

1 „Vorgeschlagen“ heisst, dass dieses Protokoll sich nicht zu einem Standard durchsetzen konnte.

4.2 strongSwan

In diesem Abschnitt soll der Aufbau von `strongSwan` in seinen Grundzügen sowie der Aufbau des `IKEv1`-Daemons `pluto` untersucht werden.

Folgende Grafik verdeutlicht das Zusammenspiel der Komponenten:

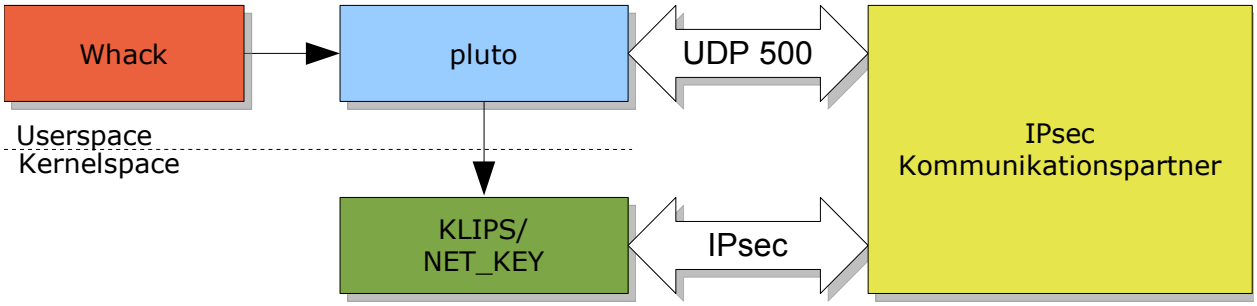


Abbildung 2: Komponenten von strongSwan und deren Zusammenspiel

Die eigentliche Kommunikation mit verschlüsselten und signierten Nutzdaten über IPsec erfolgt komplett im Kernelspace und ist in dessen IP-Stack integriert. In der Kernel-Version 2.4 kommt `KLIPS` (`kernel IPsec support`) zum Einsatz, beim neueren 2.6er nennt sich die IPsec Implementierung `NET_KEY` (auch `26sec`). Allerdings wurde `KLIPS` auch auf den 2.6er Kernel und `NET_KEY` auch auf den älteren 2.4er Kernel portiert.

Der Aufbau und die Verwaltung dieser IPsec-Verbindungen wird in einem Userspace-Daemon erledigt. In `strongSwan` wird dieser Daemon `pluto` genannt. Er verarbeitet empfangene `IKEv1`-Meldungen oder initiiert selber eine Verbindung. Die Kommunikation erfolgt über UDP auf dem Port 500. Die nötigen Informationen für die Kommunikation wie Session-Schlüssel gibt er über eine definierte Schnittstelle an den Kernel weiter.

Die Steuerung von `pluto` erfolgt über ein weiteres Programm, nämlich `whack`. Dieses kommuniziert mit einem definierten Protokoll über einen Unix-Socket mit `pluto`. Dies hat den Vorteil, dass `pluto` zur Laufzeit über diese `whack`-Schnittstelle angesprochen und gesteuert werden kann.

4.2.1 IKE-Daemon pluto

In `strongSwan` ist `IKEv1` im `pluto`-Daemon implementiert. Er übernimmt den Schlüssel-Austausch und das Aufbauen und Verwalten der SAs zwischen den Kommunikationspartnern.

Die Kern-Architektur von `pluto` ist single-threaded. Um auf mehrere Ereignisse mit nur einem Thread reagieren zu können, wird eine Event-Queue eingesetzt. In dieser Event-Queue können beliebige Events platziert werden, die dann zu gegebener Zeit verarbeitet werden.

Der Aufbau von `pluto` kann vereinfacht folgendermassen visualisiert werden:

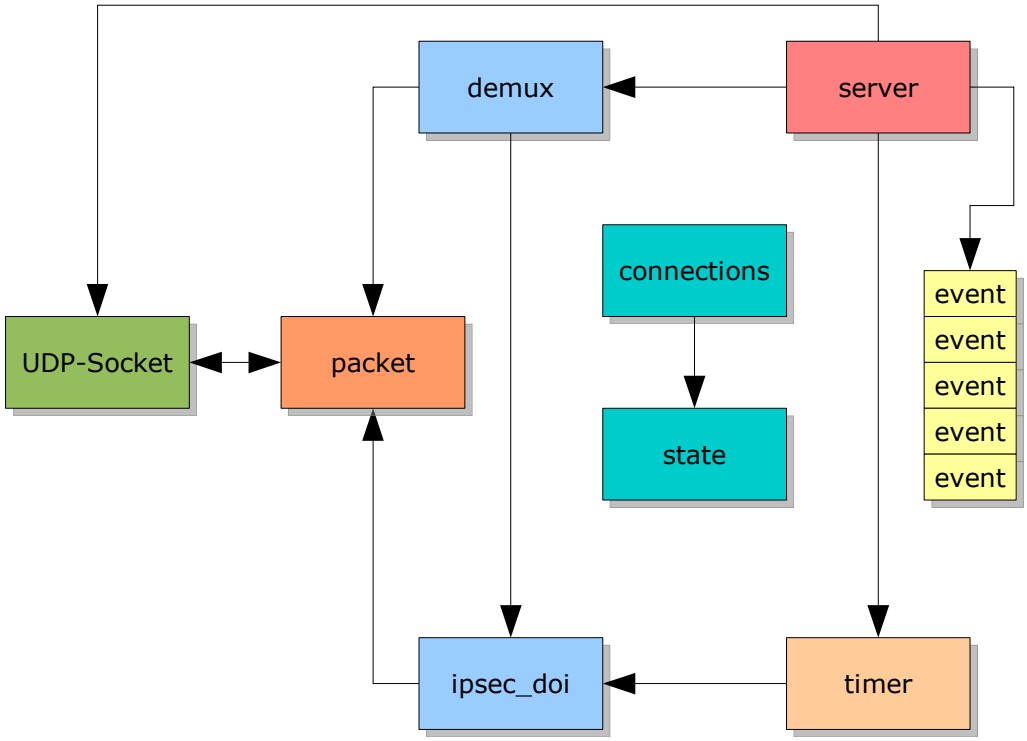


Abbildung 3: Vereinfachter Aufbau von pluto

Dabei haben die verwendeten Module folgende Aufgaben:

Modul	**Beschreibung**
server	Das Modul „server“ enthält die Hauptschlaufe von `pluto`. Diese wartet auf eingehenden Verkehr an den Sockets sowie auf die Abarbeitung von Events. Tritt ein Event ein, so wird dieser zur Abarbeitung an das Modul „timer“ weitergereicht. Wird an einem eingehenden Socket Verkehr gemeldet, ruft es Funktionen des Demuxers (Modul „demux“) auf, um die Daten einzulesen.
timer	Im „timer“-Modul sind Funktionen für die Event-Verwaltung implementiert. Es können Events in die Event-Queue eingereiht oder solche abgearbeitet werden. Das „event“-Objekt wird mit einer Ablaufzeit in die Event-Queue eingeordnet und im Modul „timer“ abgearbeitet. Es kann nur mit einem Thread auf mehrere Ereignisse gewartet werden.
demux	Das Modul „demux“ liest eingehenden Verkehr von den Sockets und verarbeitet den Verkehr entsprechend. Es enthält eine State-Machine, um auf eingehenden Verkehr reagieren zu können. Dabei kann es Antworten über das „ipsec_doi“-Modul zurück senden.
ipsec_doi	Das Modul „ipsec_doi“ enthält diverse Funktionen, welche den ausgehenden Verkehr betreffen. Es werden dort Meldungen erstellt, welche dem Kommunikationspartner zu übertragen sind.
packet	Im „packet“-Modul sind Funktionen zu finden, welche fürs (De-)Marshalling von Datenstrukturen verantwortlich sind. „demux“ verwendet es, um eingehende Pakete einzulesen, „ipsec_doi“ um solche Datenstrukturen übers Netzwerk zu verschicken.
connections	Das „connections“-Modul verwaltet eine Menge von „connections“. Es handelt sich dabei nicht um eine konkret eingegangene Verbindung, sondern vielmehr um die Beschreibung einer solchen. Sie enthält die Konfiguration, mit welcher eine konkrete SA erstellt werden kann.
state	Das Modul „state“ verwaltet die entstehenden oder bereits eingegangenen SAs. Die Datenstruktur von „state“ enthält Informationen über diese. Darin enthalten ist auch der Status, in welchem sich die SA befindet.

Tabelle 3: Module von pluto und deren Aufgaben

4.3 IKEv2

Wie IKEv1 ist auch IKEv2 ein Protokoll zur automatischen Verwaltung von Security Associations für IPsec. IKEv2 hat das Ziel, die Schwächen von IKEv1 zu beseitigen und dieses in naher Zukunft abzulösen. Zwar gilt die ältere Version von IKE keinesfalls als unsicher, allerdings ist dessen Aufbau unnötig komplex und aufwendig. So ist beispielsweise IKEv2 in einem einzigen Dokument beschrieben, wo IKEv1 drei RFCs benötigt. Auch müssen bei gleicher Sicherheit weniger Meldungen ausgetauscht werden.

IKEv2 ist noch kein Standard. Es ist jedoch zu erwarten, dass der aktuelle Draft in der Version 17 bald als Standard verabschiedet und dann als RFC zur Verfügung gestellt wird.

IKEv2 macht eine klare Unterscheidung zwischen SAs, die IKE selber verwendet, und solche, welche die effektive Nutzlastkommunikation erlauben (AH, ESP). SAs für IKE werden IKE_SA genannt. Über diese IKE_SA können SAs für AH und ESP ausgehandelt werden. Diese sind dann immer einer IKE_SA zugeordnet und werden selber CHILD_SA genannt.

Nachfolgend werden die wichtigsten Unterschiede von IKEv2 gegenüber IKEv1 aufgezeigt. Anschliessend werden die unterschiedlichen Meldungstypen von IKEv2 erläutert und mögliche Szenarien aufgezeigt.

4.3.1 Wichtige Unterschiede gegenüber IKEv1

4.3.1.1 Vereinfachungen

Der erste und wichtigste Unterschied zwischen den beiden IKE-Versionen ist die verringerte Komplexität von IKEv2 gegenüber IKEv1. Die Anzahl von unterschiedlichen Phasen und die Anzahl notwendiger Meldungen konnte signifikant verringert werden. So benötigt IKEv2 zur Authentisierung eines Partners lediglich den Austausch von vier Nachrichten, wo IKEv1 für die gleiche Sicherheit sechs Nachrichten benötigt. Des weiteren ist der Aufbau der Meldungen logischer und leichter verständlich.

Die Komplexität von IKEv1 wurde oft als seine grösste Schwäche gesehen, denn Komplexität führt zu Fehleranfälligkeit.

4.3.1.2 Erhöhte Sicherheit

Ein weiterer Kritikpunkt an IKEv1 war die Anfälligkeit gegen DoS-Attacken. Ein Angreifer kann von einer gefälschten Absender-Adresse beliebig viele Anfragen für einen Verbindungsaufbau senden und somit die Ressourcen des Servers verschwenden.

Zum Schutz gegen DoS-Attacken wurde in IKEv2 ein Cookie-Mechanismus eingeführt. Dabei weist der Server einen Verbindungsaufbau ab und sendet ein Cookie zurück, welches nur von ihm reproduzierbar ist. Der Initiator muss mit demselben Cookie den Verbindungsaufbau wiederholen. Somit ist sichergestellt, dass die Absender-Adresse gültig ist. Angriffe von einer gültigen Adresse können leicht geblockt werden, indem alle Pakete dieser Adresse verworfen werden.

4.3.1.3 Flexibilität

IKEv1 ist in einigen Punkten zu unflexibel bzw. bietet nicht die gewünschte Funktionalität. Daraufhin sind Erweiterungen entstanden, wie etwa XAUTH. XAUTH gilt als unsicher, da es unter Umständen gegen „Man in the middle“-Attacken verwundbar ist. IKEv2 führt hier die Authentisierung über das „Extensible Authentication Protocol“ ein. EAP ist ein erweiterbares Protokoll, in welchem weitere Möglichkeiten zur Authentisierung eingebaut werden können.

4.3.2 Meldungstypen

IKEv2 unterscheidet nicht mehr zwischen verschiedenen Phasen, sondern definiert folgende Austausch-Typen:

- IKE_SA_INIT
- IKE_AUTH
- CREATE_CHILD_SA
- INFORMATIONAL

Diese stehen in einem klaren Kontext zu einander. Eine Kommunikation muss mit IKE_SA_INIT beginnen und muss über IKE_AUTH fortgesetzt werden. Erst wenn der Austausch von IKE_AUTH erfolgreich abgeschlossen ist, kann die Kommunikation mit den anderen beiden Meldungstypen erfolgen. Des weiteren besteht jeder Austausch aus einem Request und einer darauf folgenden Response. Damit ist es möglich, für das erneute Senden von verlorenen Paket allein den Absender des Requests verantwortlich zu machen.

Im IKE_SA_INIT-Austausch einigen sich die Kommunikationspartner über die zu verwendenden Algorithmen und führen dabei auch einen Diffie-Hellman-Austausch durch. Im IKE_AUTH-Austausch authentisieren sich die beteiligten Kommunikationspartner und erstellen eine gemeinsame CHILD_SA. Die IKE_AUTH Kommunikation erfolgt dabei bereits verschlüsselt.

Nach erfolgreichem IKE_SA_INIT- und IKE_AUTH-Austausch haben die Kommunikationspartner eine gemeinsame IKE_SA und eine optional auch eine oder mehrere CHILD_SAs aufgebaut.

Ein weiterer wichtiger Punkt in der Terminologie von IKEv2 ist die Verteilung der Rollen. Ein Endpunkt, der einen Meldungsaustausch initiiert, wird immer als „Initiator“ bezeichnet. Derjenige, der darauf antwortet, wird „Responder“ genannt. Werden diese Bezeichnungen allerdings in einem globalen Kontext verwendet, also über einen Meldungsaustausch hinweg, so ist der „Initiator“ derjenige, der die IKE_SA initiiert hat. Oft wird dann auch von „Original Initiator“ und „Original Responder“ gesprochen. Da in den ersten Meldungen aber der „Initiator“ immer auch der „Original Initiator“ ist, wird diese Bezeichnung oft auch weggelassen.

4.3.3 IKE_SA_INIT

Der Meldungsaustausch IKE_SA_INIT ist der erste IKEv2-Austausch zwischen zwei Kommunikationspartnern und erfolgt unverschlüsselt. Ein Austausch von diesem Typ hat folgenden Zweck:

- Mit dem Diffie-Hellman-Verfahren wird ein gemeinsamer Schlüssel ausgehandelt. Dieser Schlüssel dient später als Grundlage für die Verschlüsselung von weiteren Meldungen.
- Die zu verwendenden Algorithmen für Verschlüsselung, Signierung, etc. werden ausgehandelt. Beide Kommunikationspartner müssen sich auf die gleichen Algorithmen einigen, damit der IKE_SA_INIT-Austausch erfolgreich beendet werden kann.

Der Meldungsaustausch IKE_SA_INIT besteht im Idealfall aus einer Request- und einer Response-Nachricht. Die Zahl der tatsächlich ausgetauschten Nachrichten kann jedoch höher ausfallen, da zum einen die Kommunikation über das unzuverlässige UDP-Protokoll erfolgt und zum anderen sich die Partner erst nach mehreren ausgetauschten Nachrichten über die zu verwendenden Algorithmen einigen können.

Nach erfolgreichem IKE_SA_INIT-Austausch haben sich die beteiligten Parteien auf die zu verwendenden Algorithmen geeinigt und einen gemeinsamen Diffie-Hellman-Schlüssel ausgehandelt. Beide sind so im Besitz eines gemeinsamen Schlüssels und wissen, mit welchen Algorithmen die weitere Kommunikation erfolgt.

Nachfolgend werden ein paar wichtige Szenarien eines IKE_SA_INIT-Austausches genauer betrachtet und beschrieben. Die entsprechenden Abbildungen enthalten zum Teil nicht alle möglichen Payload-Typen. Für detailliertere Informationen sei auf das [IKEv2Draft] verwiesen.

4.3.3.1 Erfolgreicher Austausch ohne Retransmit

Der Idealfall eines IKE_SA_INIT-Austausches tritt dann auf, wenn keine der IKEv2-Meldungen verloren geht und sich die beteiligten Kommunikationspartner bereits mit dem ersten Meldungsaustausch über die zu verwendenden Algorithmen einigen können. Dabei ergibt sich folgender Ablauf:

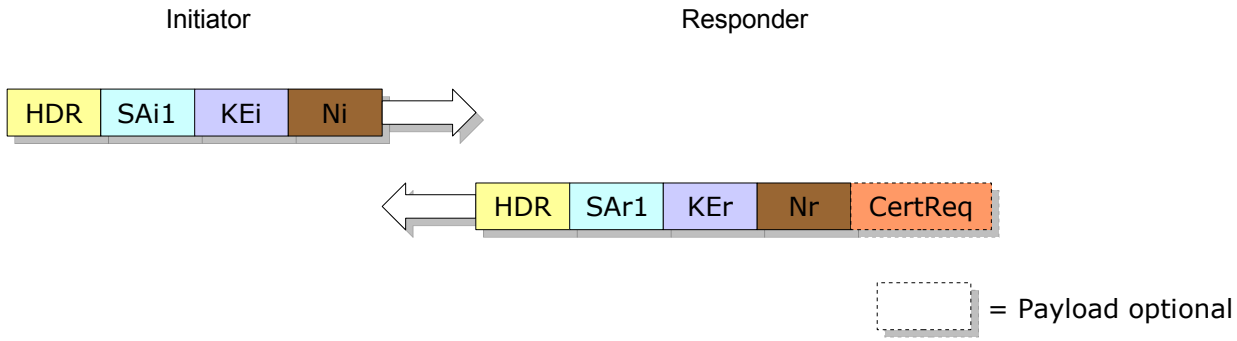


Abbildung 4: Erfolgreicher IKE_SA_INIT-Austausch ohne Retransmit

Die einzelnen Payloads haben dabei folgende Bedeutung:

Payload	Beschreibung
HDR	Der Header des IKE-Protokolls.
SAi1	Enthält ein oder mehrere Vorschläge von unterstützten kryptographischen Algorithmen (Proposals).
SAr1	Auswahl von kryptographischen Algorithmen aus dem Vorschlag vom Initiator.
KEi	Enthält den öffentlichen Diffie-Hellman-Wert des Initiators.
KEr	Enthält den öffentlichen Diffie-Hellman-Wert des Responders.
Ni	Vom Initiator zufällig gewählte Nonce.
Nr	Vom Responder zufällig gewählte Nonce.
CertReq (optional)	Ein Request für ein Zertifikat, welches der Initiator in der nächsten Meldung einbeziehen soll.

Tabelle 4: Payloads in IKE_SA_INIT

4.3.3.2 Erfolgreicher Austausch mit Retransmit

Da IKEv2 über das unzuverlässige Protokoll UDP läuft, kann sowohl ein Request als auch eine Response verloren gehen. Geht der Request oder eine Response verloren, so sendet der Initiator nach ablaufen eines bestimmten Timeout-Wertes seinen Request neu. Der Responder weiss, ob er den entsprechenden Request bereits beantwortet hat und erstellt entweder eine neue Response oder sendet seine zuletzt gesendeten Response erneut.

Der folgende Ablauf zeigt ein erfolgreicher IKE_SA_INIT-Austausch, bei welchem zwei Meldungen verloren gehen:

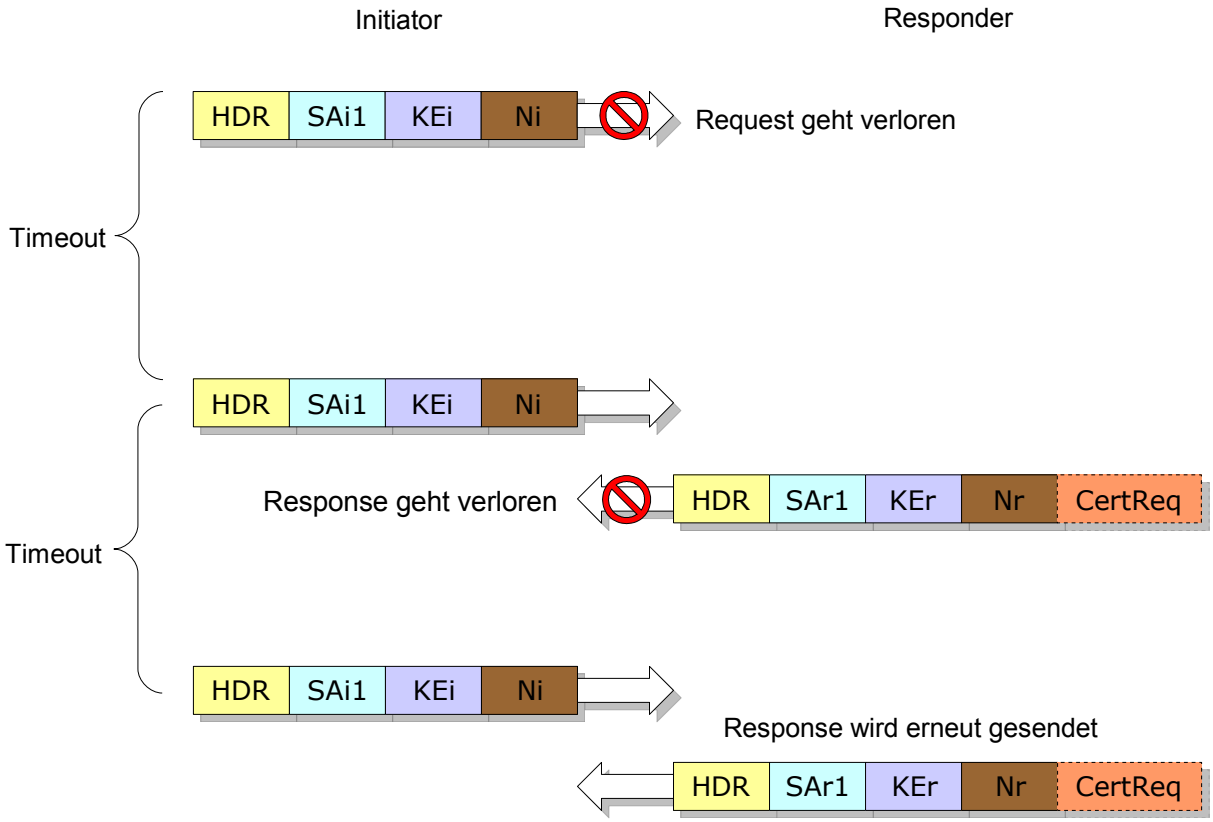


Abbildung 5: Erfolgreicher IKE_SA_INIT-Austausch mit Retransmit

Alle nachfolgenden Szenarien nehmen an, dass keine Pakete verloren gehen. Das Verhalten bei verloren gegangenen Paketen entspricht bei jedem Meldungsaustausch dem hier gezeigten.

4.3.3.3 Nicht erfolgreicher Austausch

Der Initiator muss in seinem IKE_SA_INIT-Request Vorschläge für Algorithmen zur Verschlüsselung, Signierung, etc. machen. Falls der Responder aufgrund seiner lokalen Policy keine der Vorschläge akzeptieren kann oder sonst mit irgend etwas nicht einverstanden ist, sendet dieser eine IKE_SA_INIT-Response mit einem Notify Payload an den Initiator zurück. Im Notify Payload gibt er den Grund für den nicht erfolgreichen IKE_SA_INIT-Austausch mit einem spezifischen Error-Code an.

Der folgende Ablauf zeigt einen nicht erfolgreichen IKE_SA_INIT-Austausch, bei welchem der Responder keinen der Algorithmen-Vorschläge akzeptiert:

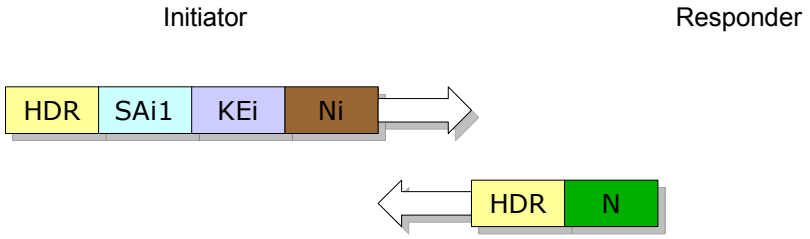


Abbildung 6: Nicht erfolgreicher IKE_SA_INIT-Austausch

Der Initiator hat nun die Möglichkeit anhand des Error-Codes im Nonce Payload (N in der Abbildung) unterschiedlich zu reagieren. Wie die Reaktionen auf solch eine Antwort aussehen können, ist dem Draft zu entnehmen.

4.3.3.4 Erfolgreicher Austausch mit Wechsel der DH-Gruppe

Der Initiator muss neben dem Vorschlag für Algorithmen zur Verschlüsselung, Signierung, etc. bereits eine Diffie-Hellman-Gruppe auswählen, die er zur Berechnung des gemeinsamen Diffie-Hellman-Schlüssels verwenden will. Falls der Responder die Gruppe nicht unterstützt, so muss der Initiator seinen öffentlichen Diffie-Hellman-Wert mit einer anderen Diffie-Hellman-Gruppe erneut berechnen.

Der Responder sendet in einem solchen Fall eine IKE_SA_INIT-Response mit einem Notify Payload vom Typ INVALID_KE_PAYLOAD an den Initiator zurück. Der Initiator weiss nun, dass seine zuletzt gewählte Diffie-Hellman-Gruppe nicht der Gruppe im ausgewählten Proposal entspricht. Er kann nun einen neuen Versuch mit einer anderen Diffie-Hellman-Gruppe starten und annehmen, dass der Responder diese akzeptiert.

Der folgende Ablauf zeigt nicht erfolgreichen IKE_SA_INIT-Austausch, bei welchem der Responder die Diffie-Hellman-Gruppe im ersten Austausch nicht akzeptiert und der Initiator daraufhin einen neuen Request mit anderer Diffie-Hellman-Gruppe startet. Es wird angenommen, dass keine Meldung verloren geht:

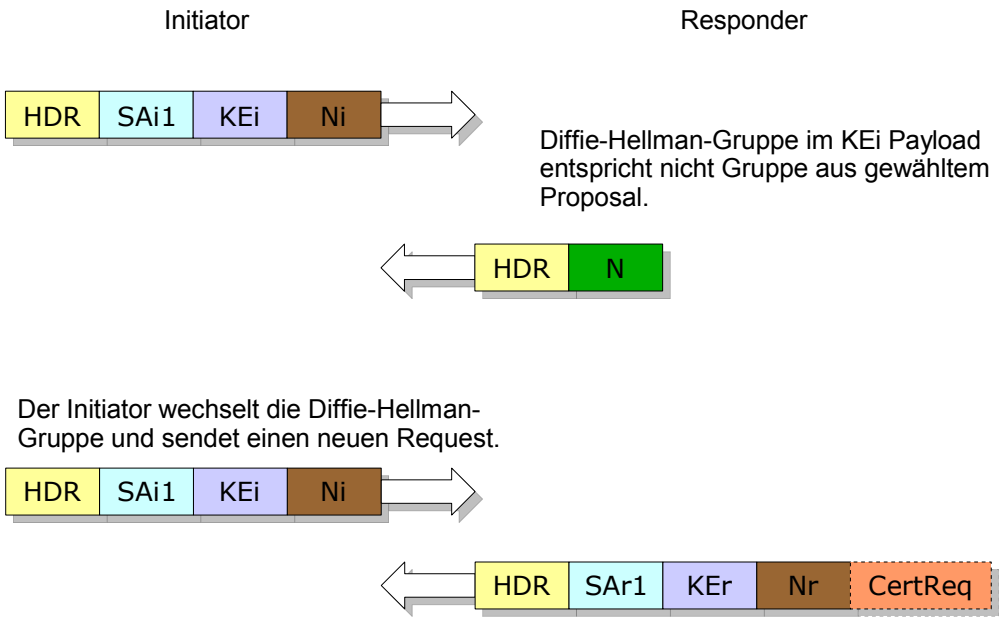


Abbildung 7: Erfolgreicher IKE_SA_INIT-Austausch mit Wechsel der DH-Gruppe

4.3.3.5 Erfolgreicher Austausch mit Cookie

Bei den bisher vorgestellten IKE_SA_INIT-Szenarien muss der Reponder jeden IKE_SA_INIT-Request parsen, bearbeiten, eine entsprechende Antwort an den Initiator zurücksenden und den Status des Verbindungsaufbaus zwischenspeichern. Dieses Verhalten erlaubt einem Angreifer eine DoS-Attacke gegen einen Server zu starten, indem er beliebig viele IKE_SA_INIT-Requests mit irgend einer Absenderadresse an den Server sendet. Der Server kann so in die Knie gezwungen werden, da er für jeden Request Ressourcen braucht.

Um dem entgegenzuwirken kommen so genannte Cookies zum Einsatz. Dabei muss der Initiator mit einem Cookie antworten, welches vom Responder ausgestellt wurde. Damit kann sichergestellt werden, dass nur für wirkliche Verbindungsversuche Ressourcen verwendet werden. Eine `IKEv2`-Implementierung entscheidet selbst, wann sie in den Cookie-Modus wechselt.

Ein erfolgreicher IKE_SA_INIT-Austausch mit der Verwendung von Cookies sieht folgendermassen aus:

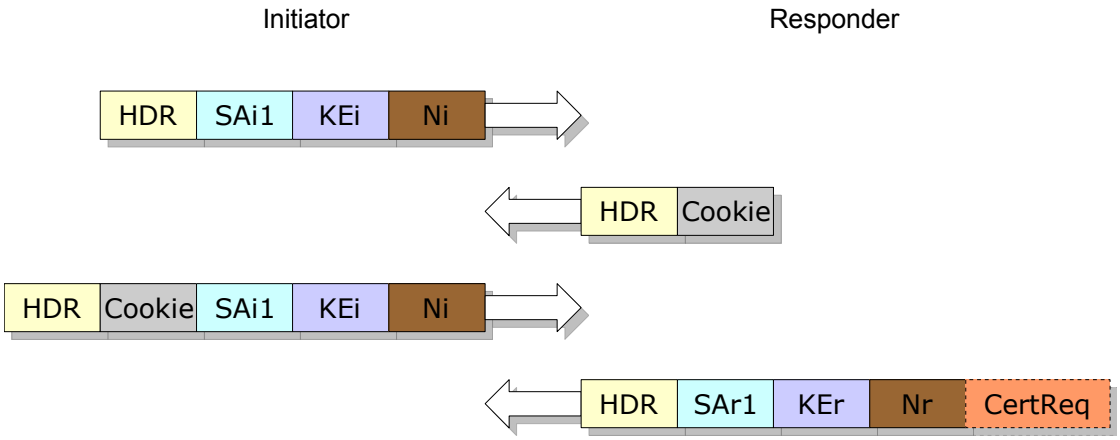


Abbildung 8: Erfolgreicher IKE_SA_INIT-Austausch mit Cookies

Die vom Initiator gesendeten Meldungen sind dabei identisch, mit der Ausnahme, dass das Cookie in die zweite Meldung einfliesst. Auf der Responder-Seite muss das Cookie nicht unbedingt zwischengespeichert werden. Im [IKEv2Draft] ist eine Möglichkeit beschrieben, wie es aus SAi, Ni und einem geheimen Schlüssel generiert werden kann. Dadurch müssen keine Zustands-Informationen für halboffene Verbindungen gehalten werden.

4.3.4 IKE_AUTH

Aus dem gewonnenen Diffie-Hellman-Schlüssel des vorhergehenden IKE_SA_INIT-Austausches leiten nun Initiator und Responder diverse Schlüssel für Verschlüsselung, Signierung, usw. ab. Die genaue Methode für die Ableitung von Schlüsseln ist unter 4.3.7.1 beschrieben. Mit den gewonnenen Schlüsseln kann nun die weitere Kommunikation verschlüsselt stattfinden. Dies bedeutet, dass alle Payloads mit Ausnahme des Headers verschlüsselt übertragen werden.

Ein Austausch von Typ IKE_AUTH hat folgenden Zweck:

- Initiator und Responder authentisieren sich gegenseitig
- Eine erste CHILD_SA für `AH` und/oder `ESP` wird ausgehandelt

Nach erfolgreichem IKE_AUTH-Austausch haben sich die beteiligten Parteien gegenseitig authentisiert und möglicherweise eine erste CHILD_SA für `AH` und/oder `ESP` ausgehandelt. Die Initialisierungsphase ist hiermit abgeschlossen. Ab diesen Zeitpunkt erfolgt die Kommunikation nur noch mit Meldungen vom Typ CREATE_CHILD_SA oder INFORMATIONAL.

Nachfolgend werden ein paar wichtige Szenarien eines IKE_AUTH-Austausches genauer betrachtet und beschrieben. Die entsprechenden Abbildungen enthalten zum Teil nicht alle möglichen Payload-Typen. Für detailliertere Informationen sei auf das [IKEv2Draft] verwiesen.

4.3.4.1 Erfolgreicher Austausch

Ein IKE_AUTH-Austausch ist dann erfolgreich, wenn sich die beteiligten Kommunikationspartner erfolgreich gegenseitig authentisieren konnten und möglicherweise eine CHILD_SA erstellt haben. Ein erfolgreicher IKE_AUTH-Austausch kann folgendermassen dargestellt werden:

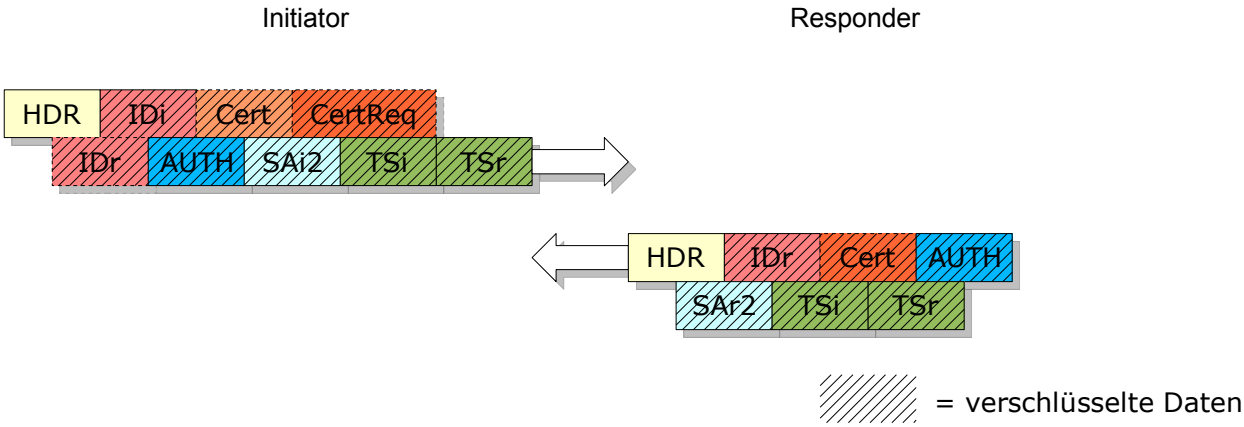


Abbildung 9: Erfolgreicher IKE_AUTH-Austausch

Die verwendeten Payloads haben dabei folgende Bedeutung:

Payload	Beschreibung
HDR	Unverschlüsselter `IKE` Header.
IDi	Identifikation des Initiators. Ist beispielsweise eine IP oder Email-Adresse.
Cert (optional)	Zertifikat für Authentisierung, falls diese von Initiator oder Responder mit Zertifikaten erfolgt. Das Zertifikat muss nicht mitgesendet werden, da der Empfänger den anderen Partner auch aufgrund der ID und lokal gespeicherten Zertifikaten authentisieren kann.
CertReq (optional)	Request für ein Zertifikat. Der Initiator kann in diesem Payload angeben, welche CA das Zertifikat des Kommunikationspartners unterschrieben haben soll.
IDr (optional bei Initiator)	Identifikation des Responders. Macht dann Sinn, wenn der Responder mehrere Identifikationen konfiguriert hat.
AUTH	Signatur durch Shared-Secret oder mitgeliefertes Zertifikat über empfangene und gesendete Daten.
SAi2	Enthält Informationen über die Eigenschaften der gewünschten SA (Algorithmen usw.).
SAr2	Enthält die ausgewählten Eigenschaften für die SA.
TSi	Gewünschte Traffic Selectors des Initiators (Adressbereiche, die durch SA geleitet werden).
TSr	Akzeptierte Traffic Selectors.

Tabelle 5: Payloads in IKE_AUTH

Durch den AUTH Payload wird die Authentisierung sichergestellt. Der Ablauf der Authentisierung ist unter 4.3.8.9 genauer beschrieben.

Die beiden Kommunikationspartner haben nach erfolgreichem IKE_AUTH-Austausch in jedem Fall die IKE_SA vollständig aufgebaut. Ob eine erste CHILD_SA aufgebaut wurde, hängt davon ab, ob auch entsprechende Proposals und Traffic Selectors vom Initiator an den Responder gesendet wurden. Die Aushandlung von Proposals und Traffic Selectors ist genauer unter 4.3.8.2 resp. unter 4.3.8.10 beschrieben.

4.3.4.2 Nicht erfolgreicher Austausch

Der Meldungsaustausch kann bei IKE_AUTH aus mehreren Gründen fehlschlagen. Unter anderem können folgende Fälle auftreten:

- Die Authentisierung des Initiators schlägt fehl
- Die Authentisierung des Responders schlägt fehl
- Es können keine übereinstimmenden Proposals gefunden werden
- Es können keine Traffic Selectors akzeptiert werden

Tritt ein Fehler beim Responder auf, so schickt er ein Notify zurück an den Initiator:

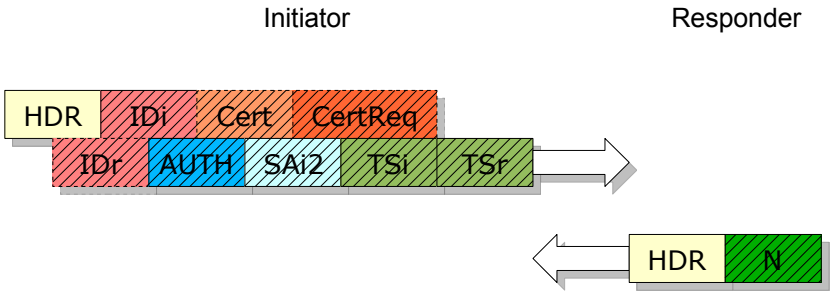


Abbildung 10: Fehlgeschlagener IKE_AUTH-Austausch

Der Responder Informiert den Initiator mit einem Notify. Dieses wird ebenfalls verschlüsselt übertragen, damit ein Aussenstehender den Austausch nicht beeinflussen kann.

Tritt aber der Authentisierungsfehler beim Initiator auf, so sendet er kein Notify. Es wird immer strikt das Request/Response Schema eingehalten, und auch hier gibt es keine Ausnahme.

4.3.5 CREATE_CHILD_SA

Mit dem Meldungsaustausch CREATE_CHILD_SA können zu einer bestehende IKE_SA zusätzliche CHILD_SAs erstellt werden. Zwar können bereits im IKE_AUTH-Austausch Security Associations für `AH` und `ESP` erstellt werden. Mit CREATE_CHILD_SA können jedoch zusätzliche SAs erstellt oder auch vorhandene einem Rekeying unterzogen werden. Der spezielle Fall, in welchem für eine IKE_SA neue Schlüssel erstellt werden, wird ebenfalls über diesen Meldungsaustausch erledigt.

Der Aufbau der Meldungen in CREATE_CHILD_SA enthält Elemente aus den Meldungen IKE_SA_INIT als auch IKE_AUTH. So werden Proposals sowie Traffic Selectors ausgetauscht, genau wie in IKE_AUTH. Zudem sind Nonces enthalten, die neue Zufälligkeit in die zu generierenden Schlüssel bringen sollen. Um zur IKE_SA völlig unabhängige Schlüssel zu erstellen, kann auch ein erneuter Diffie-Hellman-Austausch in die Meldungen miteinbezogen werden.

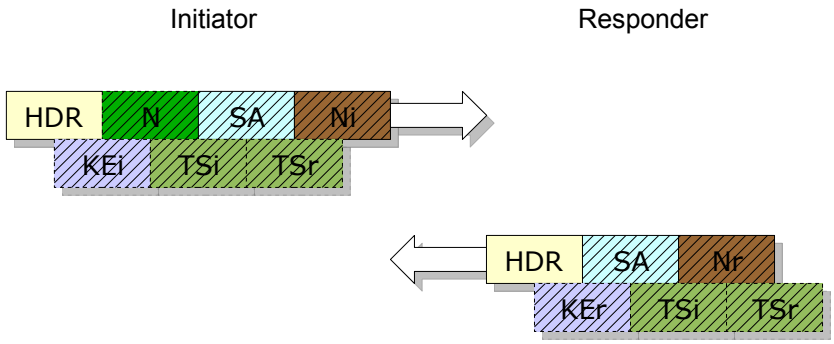


Abbildung 11: CREATE_CHILD_SA-Austausch

Der Initiator kann ein Notify vom Typ REKEY_SA miteinbeziehen. Mit diesem signalisiert er, dass für eine vorhandene SA neue Keys erstellt werden sollen.

4.3.6 INFORMATIONAL

Der INFORMATIONAL-Austausch wird für diverse Aufgaben eingesetzt, die bei einer bestehenden IKE_SA durchgeführt werden können. Dazu gehört der Austausch von weiteren Informationen, aber auch das Löschen von SAs:

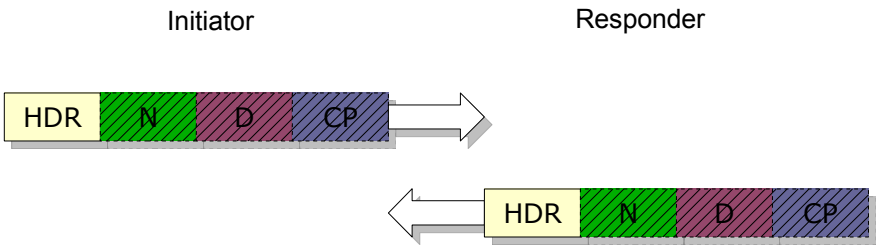


Abbildung 12: INFORMATIONAL-Austausch

Darin enthalten sind Payloads vom Typ Notify, Delete oder Configuration. Alle sind optional, können aber auch mehrfach vorkommen. Auf jeden Request wird eine Response zurück gesendet. Ein INFORMATIONAL-Austausch kann auch keine Payloads enthalten, wodurch eine Dead-Peer-Detection möglich wird.

Weitere Informationen über diesen Austausch können dem [IKEv2Draft] entnommen werden.

4.3.7 Ableiten von Schlüsselmaterial

Durch den Diffie-Hellman-Austausch entsteht ein gemeinsames Secret, welches für die weitere verschlüsselte Kommunikation verwendet werden kann. Da aber diverse Schlüssel benötigt werden und es unsicher wäre, für verschiedene Zwecke denselben Schlüssel zu verwenden, werden mehrere Schlüssel aus dem gemeinsamen Secret abgeleitet.

Diese Ableitung geschieht mit einer speziellen Pseudo-Zufalls-Funktion, im [IKEv2Draft] wird diese `prf+` genannt. Sie basiert auf der ausgehandelten Pseudo-Random-Funktion, liefert aber genügend Material für alle benötigten Schlüssel. Der Aufbau von `prf+` kann dem Draft entnommen werden.

4.3.7.1 Schlüssel für die IKE_SA

Um Schlüssel für die IKE_SA zu erzeugen, wird als erstes ein `SKEYSEED` generiert. Dazu wird die ausgehandelte PRF verwendet:

```
SKEYSEED = prf( Ni | Nr, g^ir)
```

Die PRF erhält als Schlüssel die Konkatenation der beiden Noncen, als Seed dient das Shared-Secret. Als nächstes können die Schlüssel hergeleitet werden:

```
KEYMAT = prf+(SKEYSEED, Ni | Nr | SPIi | SPIr)
```

`prf+` erhält als Schlüssel den `SKEYSEED` aus der vorhergegangenen Operation, als Seed dient die Konkatenation von den Noncen sowie den SPIs.

`KEYMAT` enthält nun genügend Schlüsselmaterial, um daraus Schlüssel in folgender Reihenfolge zu erstellen:

Schlüssel	Verwendung
SK_d	Schlüssel zur Ableitung von Schlüsseln für die CHILD_SAs (siehe 4.3.7.2)
SK_ai	Schlüssel für die Integritätsprüfung der Daten des Initiators
SK_ar	Schlüssel für die Integritätsprüfung der Daten des Responders
SK_ei	Schlüssel für die Verschlüsselung der Daten des Initiators
SK_er	Schlüssel für die Verschlüsselung der Daten des Responders
SK_pi	Schlüssel für die Authentisierungsdaten des Initiators (siehe 4.3.8.9)
SK_pr	Schlüssel für die Authentisierungsdaten des Responders

Tabelle 6: Abgeleitete Schlüssel für die IKE_SA

Die Länge der Daten, die von `KEYMAT` für einen Schlüssel entnommen werden, wird durch den ausgewählten Algorithmus resp. durch die definierte Schlüssellänge bestimmt.

4.3.7.2 Schlüssel für CHILD_SAs

Für jede CHILD_SA werden neue Schlüssel abgeleitet, um die Sicherheit zu erhöhen. Wird die CHILD_SA direkt im Meldungsaustausch von IKE_AUTH erzeugt oder der Diffie-Hellman-Austausch in einem CREATE_CHILD_SA wird ausgelassen, werden die Schlüssel folgendermassen erzeugt:

```
KEYMAT = prf+(SK_d, Ni | Nr)
```

Wird hingegen bei einem CREATE_CHILD_SA-Austausch ein neuer Diffie-Hellman-Austausch durchgeführt, wird das neue Shared-Secret miteinbezogen:

```
KEYMAT = prf+(SK_d, g^ir | Ni | Nr )
```

In welcher Reihenfolge die Schlüssel aus `KEYMAT` entnommen werden, ist im [IKEv2Draft] unter 2.17 nachzulesen.

4.3.8 Aufbau von Header, Payloads und Substrukturen

4.3.8.1 IKE Header

Der IKE Header, kompatibel zum ISAKMP Header aus IKEv1, definiert für welche SA die Meldung gilt, welche Protokollversion verwendet wird, um welchen Meldungstyp es sich handelt, usw.

Konkret enthält der Header folgende Felder:

Feldname	Grösse	Beschreibung
IKE SA Initiator's SPI	8 Byte	Vom Initiator zugeteilte SPI, identifiziert die IKE_SA für den Initiator.
IKE SA Responder's SPI	8 Byte	Vom Responder zugeteilte SPI, identifiziert die IKE_SA für den Responder.
Next payload	1 Byte	Bestimmt die Art des ersten Payloads, welcher nach dem Header folgt. Jeder folgende Payload hat ein solches Feld um den nächsten Payload anzugeben. Der Parser kann damit bestimmen, wie er eine Meldung zu verarbeiten hat.
Major version	4 Bit	Major Version des Protokolls. Für IKEv2 ist dieser Wert 2.
Minor version	4 Bit	Minor Version des Protokolls. Für IKEv2 momentan immer 0.
Exchange Type	1 Byte	Spezifiziert den Typ dieses Meldungsaustausches. Mögliche Werte sind: IKE_SA_INIT (siehe 4.3.3), IKE_AUTH(siehe 4.3.4), CREATE_CHILD_SA (siehe 4.3.5) und INFORMATIONAL (siehe 4.3.6).
Flags	1 Byte	IKEv2 definiert drei Flags, siehe dazu Tabelle 8.
Message ID	4 Byte	Eindeutige Identifikation dieses Meldungsaustausches. Erlaubt das Erkennen von verlorenen Paketen, Duplikaten und verhindert Replay-Attacken.
Length	4 Byte	Länge des gesamten Pakets, d.h. Header und alle Payloads.

Tabelle 7: Header von IKEv2

Die im Header auftauchenden Flags werden folgendermassen interpretiert:

Flag	Bit	Beschreibung
Reserviert	0-2	Reserviert, da in ISAKMP (IKEv1) verwendet.
Initiator	3	Stammt dieses Paket vom Initiator der IKE_SA, so muss dieses Flag gesetzt sein. Wenn nicht, so muss es gelöscht werden. Mit diesem Flag kann der Empfänger einer Nachricht entscheiden, welche IKE_SA SPI im Header von ihm gesetzt wurde.
Version	4	Der Absender muss dieses Flag setzen, wenn er eine höhere Major-Version des Protokolls versteht. Mit Hilfe von diesem Flag kann die höchstmögliche Version ausgehandelt werden.
Response	5	Wird gesetzt, wenn es sich bei dieser Meldung um die Response eines Meldungsaustausches handelt.
Reserviert	6-7	Reserviert für zukünftige Versionen.

Tabelle 8: Flags im Header von IKEv2

4.3.8.2 Security Association Payload

Der Security Association Payload wird benötigt, um die erforderlichen Parameter für eine SA auszuhandeln. Dabei wird dieser Payload sowohl für den Aufbau von IKE_SAs als auch von CHILD_SAs verwendet. Der Initiator listet hier kryptographische Suiten aus denen der Responder auswählen soll. In IKE werden solche Suiten Proposals genannt.

Der SA Payload enthält direkt keine Daten, sondern lediglich einen Satz von Proposals, kodiert in Proposal Substructures. Die Anzahl von diesen Strukturen ist über die Länge des Payload definiert. Die Terminologie ist etwas verwirrend. Ein Proposal wird nicht zwingend in einer einzigen Proposal Substructure definiert, sondern kann sich über mehrere solche erstrecken.

4.3.8.3 Proposal Substructure

Die Proposal Substructure ist im SA Payload enthalten. Sie enthält eine Menge von Algorithmen, in IKE werden diese Transforms genannt. Kodiert sind diese Algorithmen in Transform Substructures. Eine Proposal Substructure gilt immer für genau ein Protokoll, nämlich AH, ESP oder für IKE selber.

Der Aufbau der Struktur erlaubt eine sehr detaillierte Konfiguration der gewählten Algorithmen. So ist es möglich, Proposal Substructures zu verknüpfen, um dadurch ein komplettes Proposal darzustellen. Ein Beispiel soll dies verdeutlichen:

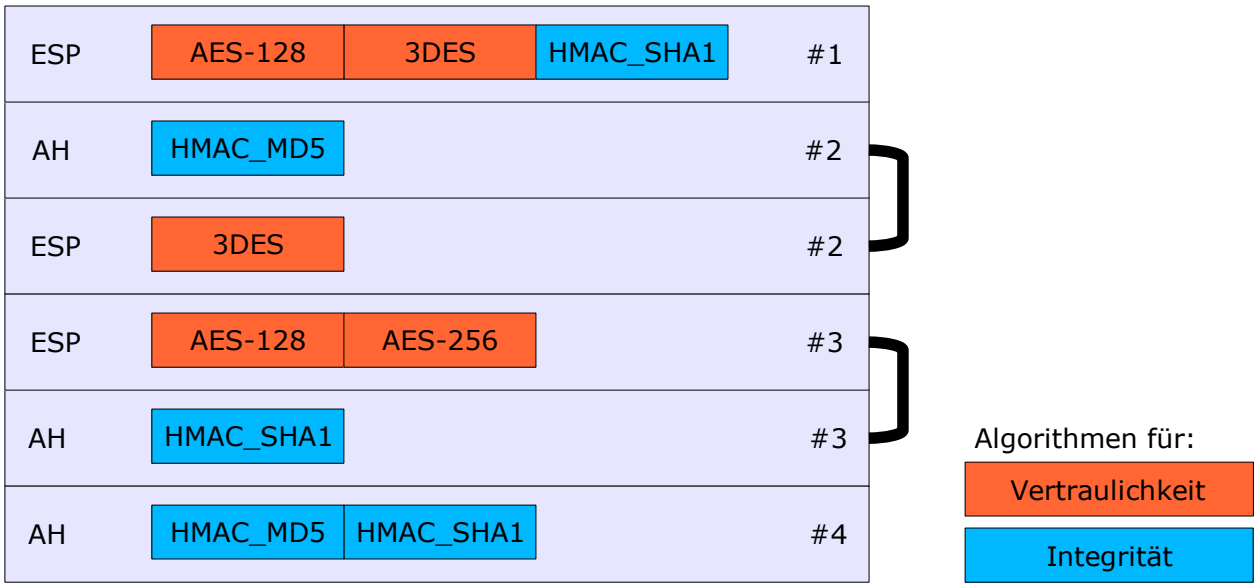


Abbildung 13: Beispiel einer Zusammenstellung von „Proposal Substructures“

Wie in der Abbildung zu erkennen ist, werden die Strukturen durchnummeriert. Die erste trägt immer die Nummer eins, jede weitere trägt dieselbe Nummer wie die vorherige, oder eine um eins höhere Nummer. Dabei gilt:

- Trägt eine Struktur dieselbe Nummer wie die vorhergehende, so gehört sie zum selben Proposal.
- Trägt eine Struktur eine höhere Nummer als die vorhergehende, so gehört sie zu einem anderen Proposal.

Der Responder, welcher eine Proposal auswählen soll, wählt genau eine Nummer und daraus je ein Transform pro Typ. Zu dieser Nummer können eventuell mehrere Strukturen gehören. Es sind also folgende Proposals wählbar:

Proposal #	ESP Verschlüsselung	ESP Integrität	AH Integrität
1	AES-128	HMAC_SHA1	Nicht verwendet
1	3DES	HMAC_SHA1	Nicht verwendet
2	3DES	Nicht verwendet	HMAC_MD5
3	AES-128	Nicht verwendet	HMAC_SHA1
3	AES-256	Nicht verwendet	HMAC_SHA1
4	Nicht verwendet	Nicht verwendet	HMAC_MD5
4	Nicht verwendet	Nicht verwendet	HMAC_SHA1

Tabelle 9: Mögliche Proposals aus den in Abbildung 13 dargestellten Proposal Substructures

4.3.8.4 Transform Substructure

In einer Transform Substructure werden verschieden kryptographische Algorithmen definiert. IKEv2 beschreibt folgende Typen für die verschiedenen Protokolle:

Bezeichnung	Schlüssellänge notwendig	Erforderlich in Protokoll	Optional in Protokoll
Encryption Algorithm	Nur für variable Schlüssel	IKE, ESP	-
Pseudo-Random Function	Ja	IKE	-
Integrity Algorithm	Ja	IKE, AH	ESP
Diffie-Hellman Group	Nein	IKE	AH, ESP
Extended Sequence Numbers	Nein	-	AH, ESP

Abbildung 14: Algorithmus-Typen und deren Verwendung in den IPsec-Protokollen

Für gewisse Algorithmen muss die Schlüssellänge definiert werden. Dies geschieht in einem angehängten Transform Attribute. Bei Verschlüsselungen ist dies nur erforderlich (und erlaubt), wenn ein Algorithmus mit variabler Schlüssellänge eingesetzt wird, wie dies bei AES der Fall ist.

4.3.8.5 Transform Attribute

Das Transform Attribute erlaubt das Hinzufügen von zusätzlichen Attributen zu einem kryptographischen Algorithmus. Momentan ist lediglich eine Kodierung für die Schlüssellänge vorgesehen. Die Struktur ist allerdings so ausgelegt, dass weitere Attribute darin untergebracht werden könnten.

4.3.8.6 Key Exchange Payload

Der Key Exchange Payload ist dafür da, Diffie-Hellman-Werte auszutauschen. Neben der verwendeten Diffie-Hellman-Gruppe wird der öffentliche Wert darin kodiert.

4.3.8.7 Nonce Payload

In der Nonce Payload werden, wie der Name vermuten lässt, Nonces übermittelt. Eine Nonce muss eine Länge im Bereich von 16 bis 255 Bytes haben und zufällig generiert worden sein.

4.3.8.8 Identification Payload

Der Identification Payload kodiert verschiedenste Typen von Identitäten. Dabei kann es sich um Identitäten von Personen als auch um solche von Rechnern handelt. Neben IP-Addresse, E-Mail-Adressen und Domainnamen sind auch speziellere Typen von Identitäten möglich. Genaueres kann dem Draft entnommen werden.

4.3.8.9 Authentication Payload

Über den Authentication Payload wird die Authentisierung des Kommunikationspartners abgewickelt, falls kein EAP eingesetzt wird. Folgende Methoden sind dabei vorgesehen:

- Erstellen eines MACs mit einem Preshared-Secret
- Erstellen einer Signatur mittels RSA
- Erstellen einer Signatur mittels DSS

Die Authentisierung erfolgt beim Meldungsaustausch IKE_AUTH. Dabei wird die Signatur[1] über das eigene Paket von IKE_SA_INIT, über die von der Gegenseite gewählte Nonce, sowie über die eigene ID erstellt:

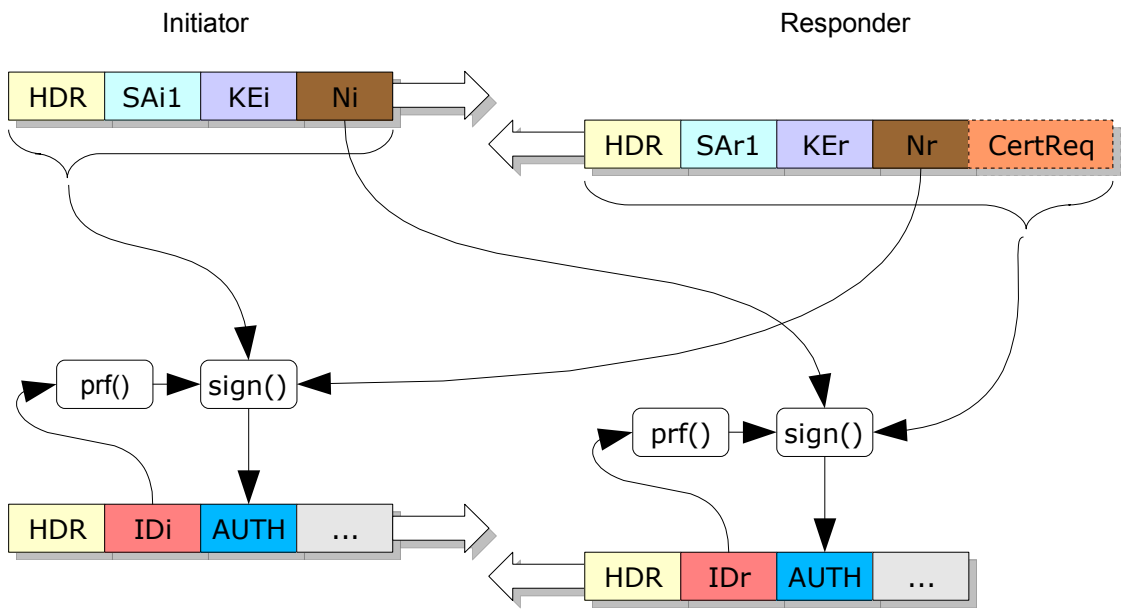


Abbildung 15: In die Authentisierung einbezogene Daten

Das Paket aus IKE_SA_INIT wird komplett mit Header und allen gesendeten Payloads einbezogen. Die Nonce wird ohne Payload, d.h. nur der Nonce-Wert selber, in die Signatur einbezogen. Vom ID Payload wird der ganze Payload, exklusiv den ersten vier Bytes, als Seed in die ausgehandelte PRF-Funktion gegeben. Als Schlüssel für die PRF dient der erzeugte Schlüssel SK_pr resp. SK_pi. Die Reihenfolge der Daten, über welche die Signatur erstellt wird, ist wie folgt:

- Eigenes Paket von IKE_SA_INIT
- Nonce des Kommunikationspartners
- Resultat, der mit der eigenen ID gefütterten PRF

1 Zur Vereinfachung wird in diesem Zusammenhang ein MAC ebenfalls als Signatur bezeichnet.

Die in Abbildung 15 dargestellte `sign()`-Operation unterscheidet sich, je nachdem welche Methode verwendet wird:

Methode	Signaturbildung
Preshared-Secret	Damit eine `IKEv2`-Implementierung das Shared-Secret nicht im Klartext speichern muss, wird das Secret zusätzlich mit einer PRF-Funktion und einem eindeutigen String bearbeitet. Insgesamt ergibt sich folgende Operation: `Auth = prf(prf(Shared Secret, "Key Pad for IKEv2"), Daten)` Der String „Key Pad for IKEv2“ wird in die PRF-Funktion mit einbezogen (ohne Null-Terminierung).
RSA	Im PKCS#1-Standard sind die Operationen von RSA beschrieben. In `IKEv2` wird eine Codierung verwendet, die sich von derjenigen in `IKEv1` unterscheidet. Die Operation für die Erstellung der Signatur ist im [RFC2437] unter 8.1.1 beschrieben, die Beschreibung der Verifizierung unter 8.1.2. Der [IKEv2Draft] sagt nichts aus über den zu verwendenden Hash-Algorithmus. [IKEv2Clarifications] empfiehlt hier SHA1 als zu verwendender Algorithmus.
DSS	Bei DSS wird ein SHA1-Hash über die Daten erstellt, wonach dieser mit DSS signiert wird.

Tabelle 10: Methoden für Signaturbildung

4.3.8.10 Traffic Selector Payload

Der Traffic Selector Payload erlaubt es, einer Secuirty Association bestimmten Netzwerkverkehr zuzuordnen. Ein Endpunkt kann so entscheiden, über welche SA ein bestimmtes IP-Paket geleitet werden soll, oder ob es überhaupt nicht durch IPsec geschützt werden muss. Traffic Selectors kommen immer paarweise vor, als TSi und TSr. Der Initiator definiert im TSi für ausgehenden Verkehr mögliche Quelladressen, für eingehenden Verkehr mögliche Zieladressen. Im TSr sind möglich Ziele für ausgehenden Verkehr definiert, für eingehende Pakete sind erlaubte Quellen gelistet. Für den Responder gelten dieselben Regeln, allerdings mit vertauschtem TSi/TSr. Ein Beispiel soll den Mechanismus verdeutlichen:

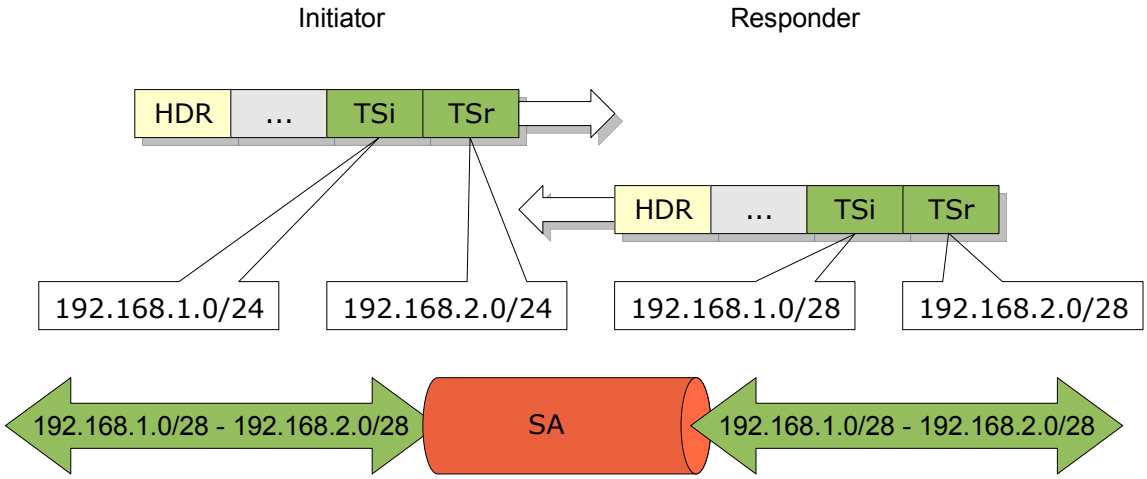


Abbildung 16: Aushandeln von "Traffic Selectors"

Der Initiator schlägt vor, den gesamten Verkehr zwischen seinem Subnetz 192.168.1.0/24 und demjenigen des anderen Endpunktes 192.168.2.0/24 über die SA zu leiten. Der Responder ist damit aber nicht ganz einverstanden, korrigiert den Vorschlag des Initiators so, dass nur Verkehr zwischen dem kleineren /28 Subnetz des Initiators zu seinem kleineren Subnetz /28 über die SA geleitet wird.

In Wirklichkeit ist das Aushandeln noch ein wenig komplexer, denn dies kann viel detaillierter als nur auf Basis von Subnetzen geschehen. Konkret enthält ein Traffic Selector Payload einen Satz von Traffic Selectors, welche diese Regeln definieren. Mehr dazu im nächsten Abschnitt.

4.3.8.11 Traffic Selector

Eine Zusammenstellung von Traffic Selector-Strukturen erlaubt es, detaillierte Konfigurationen für die Traffic Selection zusammenstellen. Ein einzelner Traffic Selector definiert einen Adress-Bereich sowie einen Port-Bereich für ein bestimmtes Protokoll. Wieder soll ein Beispiel einer Zusammenstellung von Traffic Selectors zur Verdeutlichung dienen:

Protokoll	Beginn Adress-Bereich	Ende Adress-Bereich	Begin Port-Bereich	Ende Port-Bereich
TCP	192.168.1.10	192.168.1.254	0	65535
TCP	192.168.1.7	192.168.1.7	20	21
UDP	192.168.1.122	192.168.1.125	100	200
UDP	192.168.1.3	192.168.1.3	674	674

Tabelle 11: Beispiel einer Zusammenstellung von "Traffic Selectors"

Wir nehmen an, dies ist der Vorschlag des Initiators im TSi Payload. Er möchte also den Zugriff auf einen Grossteil seiner Rechner in einem Subnetz erlauben. Auf der IP 192.168.1.7 betreibt er einen FTP-Server, worauf der Zugriff ebenfalls möglich sein soll. Darauf haben alle im TSr Payload definierten Rechner Zugriff. Des weiteren erlaubt er gewissen UDP-Verkehr.

Der Responder muss nun diesen Vorschlag untersuchen und daraus auswählen. Er kann den Vorschlag akzeptieren, in dem er die Traffic Selectors in dieser Form zurück schickt. Er könnte aber auch einen Eintrag daraus streichen oder er z.B. den Adress- und Port-Bereich für TCP-Verkehr verkleinern.

Unter gewissen Umständen kann es dazu kommen, dass eine Auswahl nicht eindeutig oder nicht komplett ausfallen kann. Diese Spezialfälle sind im [IKEv2Draft] im Abschnitt 2.9 beschrieben.

4.3.8.12 Encrypted Payload

Der Encrypted Payload wird verwendet, um in `IKEv2` andere Payloads zu verschlüsseln, aber auch um die Integrität einer ganzen Nachricht zu gewährleisten. Er hat folgenden Aufbau:

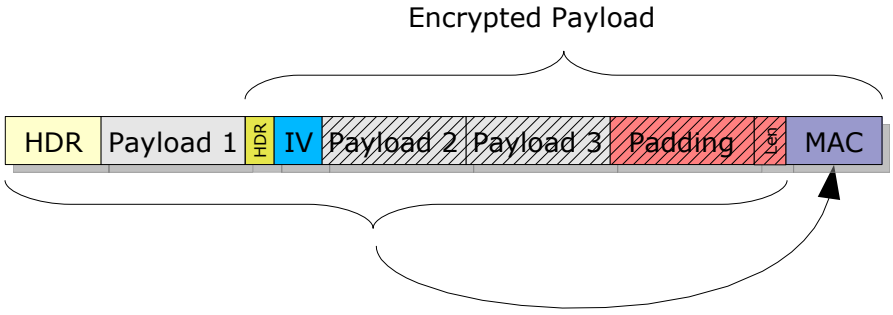


Abbildung 17: Meldung mit einer "Encrypted Payload"

Eine Meldung darf nur einen einzigen Encrypted Payload enthalten, und dieser muss immer der letzte Payload sein. Payload 1 ist unverschlüsselt, Payloads 2 und 3 sind verschlüsselt. Da meistens eine Verschlüsselung mit fester Blockgrösse eingesetzt wird, ist ein Padding der Daten nötig. Das Padding besteht aus Zufallswerten, die Länge des Paddings ist in den letzten Bytes des Paddings selber kodiert. Am Anfang des Encrypted Payloads ist wie in jedem Payload ein kurzer Header vorangehängt, welcher unter anderem die Länge definiert. Danach folgt der Initialisierungsvektor für die Verschlüsselung. Seine Länge ergibt sich aus dem verwendeten Algorithmus. Die Authentizität und Integrität wird über einen MAC sichergestellt. Dieser gehört zwar zum Encrypted Payload, erstreckt sich aber über die ganze Meldung inklusive Header und unverschlüsselten Payloads.

4.3.8.13 Notify Payload

Der Notify Payload wird verwendet, um den Kommunikationspartner über Ereignisse oder Fehler zu informieren. IKEv2 definierte verschiedenste Typen, einzelne verwenden auch zusätzliche Daten, um das Ereignis zu erläutern. Weitere Informationen sind dem [IKEv2Draft] zu entnehmen.

4.3.8.14 Weitere Payloads

IKEv2 definiert noch weitere Payloads. Diese wurden während der Arbeit nicht weiter verwendet, weshalb sie hier nur am Rande erwähnt sind:

Payload	Beschreibung
Certificate Payload	Dieser Payload wird zum Austausch von Zertifikaten und anderen Authentisierungsdaten verwendet.
Certificate Request Payload	Mit diesem Payload werden Anfragen für Zertifikate gesendet. Dazu können CA Zertifikate eingebettet werden.
Delete Payload	Durch diesen Payload wird mitgeteilt, dass eine oder mehrere SAs gelöscht wurden.
Vendor ID Payload	Ein solcher Payload lässt eine bestimmte Implementierung eindeutig erkennen.
Configuration Payload	Über diesen Payload werden Konfigurationen ausgetauscht. Zum Beispiel ist es möglich, darüber dynamische IP Adressen zu verteilen.
EAP Payload	Mit Hilfe dieses Payloads ist es möglich, die Authentisierung über die verschiedensten Möglichkeiten des Extensible Authentication Protocols abzuwickeln.

Tabelle 12: Weitere Payloads von IKEv2

4.4 Netlink

Netlink ist eine Technologie, die die Kommunikation zwischen Userspace und dem Linux-Kernel ermöglicht. Der Datenaustausch erfolgt dabei über normale Sockets.

Eingesetzt wird Netlink beispielsweise für den Datenaustausch von Routing- und Firewall-Daten, aber auch eine Schnittstelle für IPsec ist vorhanden. Darüber lassen sich SAs Verwalten und weitere Operationen ausführen. Das Protokoll für die Kommunikation mit dem IPsec-Subsystem wird XFRM genannt.

Um eine Kommunikation mit dem Kernel zu starten, wird ein Socket erstellt, wie er normal für die Netzwerkkommunikation eingesetzt wird:

```
xfrm_socket = socket(PF_NETLINK, SOCK_RAW, NETLINK_XFRM);
```

NETLINK_XFRM definiert hierbei, dass die Kommunikation über das XFRM-Protokoll stattfinden soll.

Für eine ausführlichere Beschreibung von Netlink und nützlichen Makros dazu sei an dieser Stelle auf [Horman04] verwiesen.

4.4.1 Paketaustausch

Die Kommunikation über einen Netlink-Socket erfolgt immer mit Paketen, denen ein Header vorangestellt ist.

```
struct nlmsghdr {
     __u32       nlmsg_len;        /* Länge des Paketes, mit Header */
     __u16       nlmsg_type;       /* Typ der Nachricht */
     __u16       nlmsg_flags;      /* Zusätzliche Flags */
     __u32       nlmsg_seq;        /* Sequenznummer */
     __u32       nlmsg_pid;        /* PID des Senders */
};
```

Allen Paketen wird eine Sequenznummer zugeordnet, damit eine Antwort einer Anfrage zugeordnet werden kann. Des weiteren ist im Header die Prozess-ID enthalten. Somit kann bestimmt werden, wer das Paket gesendet hat. Der Kernel sendet dabei mit der Prozess-ID Null. Der Typ definiert den Aufbau der Daten, welche nach dem Header folgenden. Diese sind spezifisch für das verwendete Protokoll (z.B. XFRM).

Das versenden von Nachrichten erfolgt über die normalen Socket-Funktionen, am besten eignet sich wohl die sendto()-Funktion:

```
struct sockaddr_nl addr;
struct nlmsghdr hdr;

/* Erstelle Header und Nachricht ... */
addr.nl_family = AF_NETLINK;
addr.nl_pid = 0;

sendto(xfrm_socket, request, &hdr, 0, (struct sockaddr*)&addr, sizeof(addr));
```

Als Empfänger ist hier der Kernel definiert. Er wird darauf Antworten. In seinem Paket wird ein Header vorangestellt sein, welcher die PID Null enthält und dieselbe Sequenznummer trägt, wie wir sie in unserem Paket vergeben haben.

Das Einlesen der Antwort wird am einfachsten mit der `recvfrom()`-Funktion abgewickelt:

```
struct sockaddr_nl addr;
socklen_t addr_len = sizeof(addr);
char buffer[1024]

recvfrom(xfrm_socket, &buffer, 1024, 0, (struct sockaddr*)&addr, &addr_len);
```

Der Buffer enthält nun zu Beginn einen Header, mit welchem der Aufbau der darauf folgenden Daten spezifiziert wird.

4.4.2 Beziehen einer SPI

Um eine SA für `AH` oder `ESP` einzurichten, kann vorzeitig vom Kernel eine SPI bezogen werden. Dies geschieht mit einer Nachricht vom Typ `XFRM_MSG_ALLOCSPI`. Ein solches Paket ist folgendermassen zusammengesetzt:

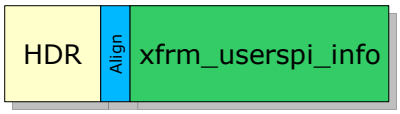


Abbildung 18: XFRM-Paketaufbau für das Beziehen einer SPI

Eventuell sind noch zusätzliche Bytes zwischen den Header und den folgenden Daten zu setzen. Dies hat den Grund, dass die folgenden Datenstrukturen auf ein gewisses Alignment gesetzt werden müssen. Die Struktur von `xfrm_userspi_info` ist im Header `<linux/xfrm.h>` definiert. Sie enthält Informationen zur Art der SPI. Ebenfalls kann darin der Bereich angegeben werden, in welcher die SPI reserviert werden soll.

4.4.3 Einrichten einer SA

Soll eine komplette SA eingerichtet werden, so wird der Aufbau des Paketes etwas komplexer. Der Kernel braucht zusätzliche Informationen, wie verwendete Algorithmen und dazugehörende Schlüssel. Der Typ der Nachricht heisst `XFRM_MSG_NEWSA`, resp. `XFRM_MSG_UPDSA`, falls eine bestehende SA bearbeitet werden soll. Ein möglicher Aufbau sieht folgendermassen aus:

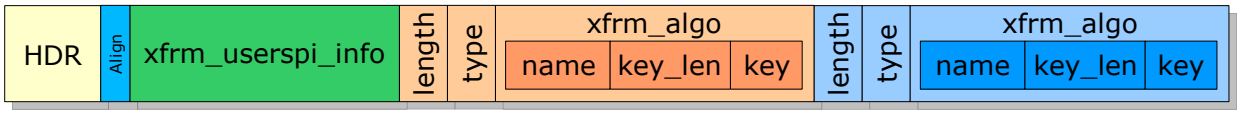


Abbildung 19: XFRM-Paketaufbau für das Einrichten einer SA

Die Struktur von `xfrm_userspi_info` enthält bei diesem Paket zusätzliche Informationen, wie z.B. die Lebensdauer einer SA. Danach folgen zu verwendende Algorithmen, sowie Schlüssel dazu. Das Feld `length` definiert die Länge der Algorithmusdefinition, welche das Längenfeld selber bis und mit der `xfrm_algo` Struktur beinhaltet. Danach folgt der Typ des Algorithmus. Die darauf folgenden `xfrm_algo` Struktur enthält einen String, der den Algorithmus identifiziert (z.B. „`aes`“). Ebenfalls darin enthalten ist die Länge des Schlüssels, sowie der konkrete Schlüssel. Danach folgt die nächste Algorithmusdefinition.

Der Aufbau von Strukturen sowie alle Konstanten sind im Header `<linux/xfrm.h>` definiert.

4.5 pthreads

pthreads ist eine standardisierte POSIX-Schnittstelle für die Verwendung von Threads. Linux unterstützt diese Schnittstelle mit mehreren Kernel-Implementierungen. Sowohl die älteren LinuxThreads als auch die neuen NPTL-Threads im 2.6er Kernel können über diese standardisierte API verwendet werden.

4.5.1 Erstellen von Threads

Ein Thread kann mit pthreads sehr trivial erstellt werden:

```
#include <pthread.h>

void *hello(void *text)
{
     printf((char*)text);
}

int main()
{
     pthread_t thread;
     if (pthread_create(&thread, NULL, hello, "Hallo!\n")) {
          printf("Konnte Thread nicht erstellen!\n");
     }
     pthread_join(thread, NULL);
}
```

Die Funktion pthread_create() erstellt ein Thread. Sie nimmt folgende Parameter entgegen:

Parameter	Beschreibung
thread [out]	Adresse für einen Thread-Identifier. Über diesen Identifikation kann der Thread nach der Erstellung manipuliert werden.
attr	Attribute für den zu erstellenden Thread. Über diese Attribute können Verhalten und Eigenschaften des Threads bereits bei der Erstellung definiert werden. NULL für Standard-Einstellungen.
start_routine	Funktion, welche der Thread nach seiner Erstellung abarbeitet. Diese hat ein Rückgabewert und ein Argument vom Typ void*.
arg	Wert, welcher der Funktion als Argument übergeben werden soll.

Tabelle 13: Parameter von pthread_create()

Damit auf das Beenden des erstellten Threads gewartet wird, ruft der Haupt-Thread pthread_join() auf. Diese Funktion wartet das Beenden des Threads ab und kann den Rückgabewert der start_routine entgegen nehmen.

4.5.2 Synchronisation

Wenn mit mehreren Threads gearbeitet wird, muss der Zugriff auf Ressourcen strikt geregelt werden. Benutzen beispielsweise mehrere Threads den gleichen Speicher, so muss der Zugriff darauf synchronisiert werden, d.h. nur ein Thread darf zu einer gegebenen Zeit auf diesen Speicher zugreifen. Diese Locks können mit einem Mutex realisiert werden.

Des weiteren muss eine Signalisation zwischen Threads möglich sein. Wartet ein Thread auf eine Ressource, welche von einem anderen Thread erst zu einem späteren Zeitpunkt zur Verfügung gestellt wird, muss der wartende Thread nach der Bereitstellung durch den anderen Thread benachrichtigt werden können. Dies kann mit einer Conditional-Variable gelöst werden.

4.5.2.1 Mutex

Durch einen Mutex kann der Zugriff auf gemeinsame Ressourcen geregelt werden. Mit ihm wird ein kritischer Bereich definiert, in welchem nur immer ein einziger Thread Zugriff hat. Dazu muss der Mutex beim Eintreten in diesen Bereich gesperrt und beim Austreten wieder freigegeben werden.

```
...

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void critical_function()
{
     pthread_mutex_lock(&mutex);

     /* Zugriff auf Ressource */

     pthread_mutex_unlock(&mutex);
}
```

`phtread_mutex_lock()` blockiert, falls ein anderer Thread bereits den Lock besitzt. Der wartende Thread wird erst wieder geweckt, wenn `pthread_mutex_unlock()` aufgerufen wird.

Als Parameter erwarten beide Funktionen die Adresse des Mutex. Dieser muss vor der Verwendung unbedingt initialisiert werden. Dies geschieht mit der Zuweisung von `PTHREAD_MUTEX_INITIALIZER` oder alternativ mit der Funktion `pthread_mutex_init()`. Bei der Initialisierung kann dem Mutex auch ein anderes Verhalten zugeordnet werden, z.B. dass ein rekursives Sperren eines Mutex möglich ist.

4.5.2.2 Conditional-Variable

Das zweite wichtige Konstrukt für das Programmieren von Multi-Threaded-Applikationen ist die Conditional-Variable. Zusammen mit einem Mutex ermöglicht sie, stillgelegte Threads wieder aufzuwecken. Dies wird notwendig, wenn auf ein bestimmtes Ereignis gewartet werden muss und dieses Ereignis durch einen andere Thread ausgelöst wird. So kann dieser den wartenden Thread wecken.

Wie erwähnt kommt die Conditional-Variable immer im Zusammenspiel mit einem Mutex zum Einsatz. Die Conditional-Variable wird in einem kritischen Bereich verwendet. Beginnt der Thread in diesem kritischen Bereich mittels Conditional-Variable zu warten, so wird der kritische Bereich wieder freigegeben, d.h. ein zweiter Thread kann den kritischen Bereich betreten! Wird der wartende Thread geweckt, so sperrt er zuerst wieder den Mutex, damit sich kein anderer Thread im kritischen Bereich befinden kann. Somit kann es zwar vorkommen, dass mehrere Threads im kritischen Bereich sind, allerdings kann nur immer einer ausserhalb der Conditional-Variable sein.

```
...
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condvar = PTHREAD_COND_INITIALIZER;

int outstanding_events = 0;

void blocking_function()
{
     pthread_mutex_lock(&mutex);

     while (outstanding_events == 0)
     {
           pthread_cond_wait(&condvar, &mutex);
     }
     process_event();
     outstanding_events--;

     pthread_mutex_unlock(&mutex);
}
```

Als Parameter nimmt die Funktion `ptread_cond_wait()` eine Conditional-Variable als auch einen Mutex entgegen. Beim Mutex muss es sich um denselben handeln, mit welchem dieser kritische Bereich gesperrt wurde.

Wichtig zu beachten ist, dass das Prüfen des erwarteten Events in einer `while`-Schlaufe erfolgen muss. Eine einfach `if`-Abfage würde nicht genügen. Dies, weil es gut möglich ist, dass mehrere Threads auf den Event warten. Werden die Threads nun geweckt, so könnte ein anderer den Event bereits abgearbeitet haben. Deshalb ist eine erneute Prüfung notwendig.

Um in einer Conditional-Variable zu warten, steht neben `pthread_cond_wait()` noch eine weitere Funktion zur Verfügung. Bei `pthread_cond_timedwait()` kann zusätzlich noch ein Zeitpunkt angegeben werden. Wird der Thread innerhalb dieser Zeitspanne nicht geweckt, wacht er selber auf und arbeitet den kritischen Bereich weiter ab. Der Rückgabewert von `pthread_cond_timedwait()` ist in diesem Fall `ETIMEDOUT`.

Hat ein anderer Thread den Event ausgelöst, so kann er einen wartenden Thread aufwecken:

```
...
void event_builder()
{
     create_event();
     outstanding_events++;
     pthread_cond_signal(&condvar);
}
```

`pthread_cond_signal()` weckt einen schlafenden Thread. Dieser nimmt die Arbeit wieder auf, sieht dass ein Event ansteht und kann ihn nun abarbeiten. Alternativ können mittels `pthread_cond_broadcast()` alle Threads in einer Conditional-Variable geweckt werden. In diesem Beispiel macht dies allerdings keinen Sinn.

4.5.3 Beenden von Threads

Das schwierigste bei der Verwendung von Threads ist wohl deren Beendigung. Ein Thread sollte nicht einfach zerstört werden, da es sonst zu folgenden Situationen kommen kann:

- Der Thread ist mitten in einer Verarbeitung und verändert darin Daten. Wird er einfach beendet, können die Daten unter Umständen in einem inkonsistenten Zustand verbleiben.
- Der Thread befindet sich in einem kritischen Bereich. Wird er beendet, gibt er den Mutex nicht wieder frei. Es kann nun kein anderer Thread den kritischen Bereich mehr betreten und es kommt zum Deadlock.

Um diese Probleme zu lösen, bieten die `pthreads` zwei Hilfsmittel. Zum einen kann ein Thread für die späteren Terminierung markiert werden, dies nennt man „deferred cancellation“. Zum anderen können „Cleanup-Handlers“ installiert werden.

4.5.3.1 Deferred cancellation

Wird ein Thread mit Standard-Attributen initialisiert, so arbeitet er automatisch mit „deferred cancellation“. Das Verhalten kann mit dem Befehl `pthread_setcancelstate()` auch so modifiziert werden, dass er Anfragen für seine Terminierung schlicht ignoriert. In diesem Zustand arbeitet er die Anforderungen nicht ab, merkt sie sich aber. Er arbeitet sie ab, sobald sein Verhalten wieder auf das Ursprüngliche geändert wird.

Mit dem Befehl `pthread_setcanceltype()` kann das verhalten bei der Terminierung gewechselt werden, so dass er direkt terminiert. Dies ist aber aus oben genannten Gründen nur in speziellen Fällen empfehlenswert.

Bei der „deferred cancellation“ wird der Thread dazu aufgefordert, zu terminieren. Dieser leistet dem allerdings nicht direkt folge, sondern wartet, bis er in einem konsistenten Zustand in seiner Verarbeitung angelangt ist. Einen solchen Zustand wird „cancellation point“ genannt. Erreicht er einen solchen, terminiert er.

Ein „cancellation point“ ist eine blockierende Funktion. Beim Ausführen folgender Funktionen wird ein zur Beendigung markierter Thread beendet:

- `pthread_join()`
- `pthread_cond_wait()`
- `pthread_cond_timedwait()`
- `pthread_testcancel()`

`pthread_testcancel()` ist eine spezielle Funktion, die genau nur diesen Zweck erfüllt. Sie prüft, ob der Thread zur Beendigung markiert ist. In einem solche Fall wird der Thread terminiert.

Zusätzlich zu den genannten Funktionen kommen je nach POSIX-Kompatibilität weitere hinzu. In einem vollkommen kompatiblen Betriebssystem stellt jede blockierende Funktion ein „cancellation point“ dar.

Tests haben gezeigt, dass unter Linux diese Unterstützung breit vorhanden ist. So gilt beispielsweise ein Aufruf von `printf()` bereits als „cancellation point“.

Ein Beispiel zu „deferred cancellation“ ist im nächsten Abschnitt beschrieben.

4.5.3.2 Cleanup-Handlers

Beim Beenden von Threads kann es nützlich sein, zuvor noch Aufräumarbeiten durchzuführen. Dazu können in einem Thread so genannte „Cleanup-Handlers“ registriert werden. Diese werden aufgerufen, sobald der Thread beendet wird. Dies ist bei `pthread_exit()` sowie auch bei jeder Variante von „cancellation“ der Fall.

Die Handler sind in einem Stack aufgebaut, was einem erlaubt, mehrere solcher Handlers aneinander zu hängen.

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condvar = PTHREAD_COND_INITIALIZER;

void cleanup(void *mutex)
{
     pthread_mutex_unlock((pthread_mutex_t*)mutex);
}

void *loop(void *nix)
{
     for (;;) { /* nur durch cancellation zu verlassen */
           pthread_mutex_lock(&mutex);
           pthread_cleanup_push(cleanup, &mutex);
           pthread_cond_wait(&condvar, &mutex);
           pthread_cleanup_pop(0);
           pthread_mutex_unlock(&mutex);
     }
}

int main()
{
     pthread_t thread;

     pthread_create(&thread, NULL, loop, NULL);
     sleep(1);
     pthread_cancel(thread);

     pthread_join(thread, NULL);
}
```

In diesem Beispiel wird ein Thread gestartet, welcher eine Abarbeitung übernimmt. Zuerst registriert er mittels `pthread_cleanup_push()` einen „Cleanup-Handler“. Danach wartet er in einer Conditional-Variable. Der Haupt-Thread sendet nun mit `pthread_cancel()` eine Anforderung zur Terminierung an den zweiten Thread. Da ein Cleanup-Handler registriert wurde, wird der Mutex wieder freigegeben.

Wenn der Thread aus der Conditional-Variable geweckt wird, muss er den Handler unbedingt wieder löschen. Der Parameter von `pthread_cleanup_pop()` gibt an, ob der Handler abgearbeitet werden soll, bevor er gelöscht wird. Man hätte in diesem Fall also den Parameter auf 1 setzen, dafür auf das `pthread_mutex_unlock()` verzichten können.

5 Design

Das Dokument „Design“ soll einen Überblick über den Aufbau des entwickelten `IKEv2`-Daemons geben. Es soll Entscheidungen die zur Wahl des Designs führten nachvollziehbar machen und die Architektur soweit erklären, dass der Einstieg für eine Weiterentwicklung so einfach wie möglich wird.

Zu Beginn werden die Prinzipien erläutert, nach denen sich das Design richtet. Des weiteren wird auf mögliche Threading-Lösungen eingegangen und die realisierte Lösung aufgezeigt. Die Architektur mit all ihren Elementen soll die Funktionalität im Einzelnen erklären, wobei Ablaufdiagramme das komplexe Zusammenspiel dieser Komponenten visualisieren.

5.1 Design-Prinzipien

Das Software-Design wird unter anderem mit UML-Diagrammen beschrieben. UML-Diagramme eignen sich besonders für das Design auf Basis objektorientierter Programmiersprachen. Da die Programmiersprache C keine Unterstützung für Klassen und Objekte anbietet, müssen die in diesem Dokument verwendeten UML-Diagramme irgendwie in die sequenzielle Programmiersprache C umgesetzt werden.

Der „Xine hacker's guide“ [XineHG] beschreibt eine Möglichkeit, objektorientierte Ansätze in C umzusetzen. Wir haben uns entschlossen, in unserer `IKEv2`-Implementierung einen ähnlichen Ansatz bei der Umsetzung des Designs zu verfolgen. Die Gründe, die für diese Entscheidung sprechen, sind unter 5.1.4 zusammengefasst.

Die für unsere Implementierung gewählte Methode zur Umsetzung objektorientierter Ansätze beruht auf Interfaces und deren konkreten Implementierungen. Um die Übersichtlichkeit des Designs zu erhöhen, unterscheiden wir nachfolgend folgende zwei Typen von Interfaces:

- Interfaces, für welche nur eine konkrete Implementierung vorhanden ist
- Interfaces, für welche mehrere konkrete Implementierungen vorhanden sind

Diese beiden Typen werden in unserem Design als unterschiedliche UML-Diagramme dargestellt, obwohl, die entsprechende Implementierung gleich aufgebaut ist. Die Details dazu sind nachfolgend beschrieben.

Neben dem objektorientierten Ansatz verwenden wir zudem das Package-Prinzip um die Übersichtlichkeit der Software weiter zu erhöhen. Mehr dazu ist unter 5.1.5 zu erfahren.

5.1.1 Interface mit einer Implementierung

Funktionen, die nur in einer einzigen Klasse implementiert werden, würden in Java nicht in einem Interface zusammengefasst werden. Stattdessen würde man in Java nur die konkrete Klasse mit den entsprechenden Funktionen implementieren.

Unser Methode zur Umsetzung objektorientierter Ansätze in C lässt diese Unterscheidung von Klassen und Interfaces nicht zu. Stattdessen wird immer ein Interface in Form eines C-`structs` definiert, auch wenn dieses Interface nur einmal implementiert wird. Um die UML-Diagramme nicht unnötig zu überladen, verzichten wir auf die Darstellung des Interfaces, wenn nur eine einzige Klasse dieses Interface implementiert und zeichnen stattdessen eine einfache Klasse.

Ein Beispiel für ein Interface mit nur einer Implementierung ist die Send-Queue, implementiert in der Klasse `send_queue_t` (siehe 5.5.12.3):

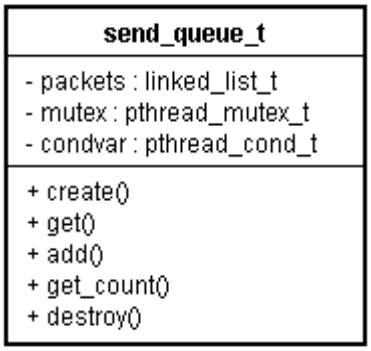


Abbildung 20: UML-Diagramm für ein Interface mit nur einer Implementierung

Wie bereits erwähnt, ist die Send-Queue als Klasse `send_queue_t` modelliert und besteht aus privaten Variablen und mehreren öffentlichen Funktionen. Wie eine Klasse nach obiger UML-Schreibweise in C umgesetzt wird, kann anhand der Anleitung unter 5.1.3 nachvollzogen werden.

5.1.2 Interface mit mehreren Implementierungen

Die für unsere Implementierung gewählte Methode zur Umsetzung objektorientierter Ansätze in C lässt das Definieren von Interfaces zu, die dann in den eigentlichen Klassen implementiert werden.

Als Beispiel soll hier das Interface `crypter_t` dienen, welches zur symmetrischen Ver- und Entschlüsselung verwendet werden kann. Pro Verschlüsselungsalgorithmus wird eine Klasse erstellt, die das `crypter_t`-Interface implementiert. Das Interface `crypter_t` sieht in UML-Schreibweise folgendermassen aus:

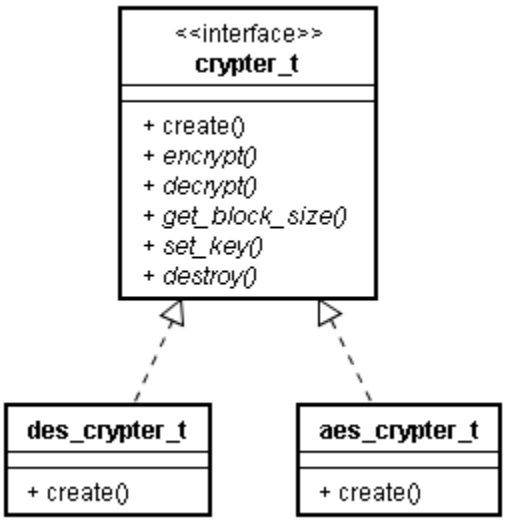


Abbildung 21: UML-Diagramm für ein Interface mit mehreren Implementierungen

Das obige Beispiel zeigt das Interface `crypter_t` mit einer Funktion zur Verschlüsselung und einer Funktion zur Entschlüsslung. Die eigentliche Klassen `des_crypter_t` und `aes_crypter_t` implementieren dieses Interface für die symmetrischen Verschlüsselungsalgorithmen DES und AES. Beide Klassen speichern den spezifischen Schlüssel zur Ver- und Entschlüsselung in einer privaten Variable. Am Beispiel `crypter_t` wird nachfolgend die Umsetzung in C beschrieben.

Speziell zu vermerken ist hier, dass das Interface einen Konstruktor besitzt. Über diesen generischen Konstruktor kann eine spezifische Instanz erstellt werden, in dem der Typ spezifiziert wird. Dies ist sehr nützlich, wenn zur Laufzeit dynamisch Instanzen von `crypter_t` erstellt werden sollen. Intern verwendet der Konstruktor von `crypter_t` die spezifischen Konstruktoren der implementierenden Klassen.

5.1.3 Umsetzung der UML-Diagramme in C

Die unter 5.1.1 und 5.1.2 vorgestellten Modellierungen von Klassen und Interfaces in UML müssen nun irgendwie in C umgesetzt werden. Dabei ist das Vorgehen für beide Modellierungs-Arten prinzipiell das selbe.

Falls ein Interface nur einmal implementiert wird, so kann die Implementierung direkt zu diesem Interface erfolgen. Wird es jedoch von mehreren Klassen implementiert, so macht es Sinn, das Interface zuerst abzuleiten. So können dem abgeleiteten Interface weitere öffentliche Methoden zugeordnet werden. Dieses abgeleitete Interface kann dann mit den zusätzlichen Methoden implementiert werden.

5.1.3.1 Definition einer Klasse

Als Beispiel wird nun das Interface `crypter_t` und die implementierende Klasse `aes_crypter_t` in die Programmiersprache C umgesetzt. Es wird zwischen C- und Header-Files unterschieden. Im Header-File werden alle öffentlichen Variablen und Funktionen deklariert. Das C-File beinhaltet die Funktionsdefinitionen und private Deklarationen. Die Umsetzung erfolgt folgendermassen:

1. Das Interface mit den öffentlichen Methoden wird im Header-File `crypter.h` als C-`struct` deklariert. Zu den öffentlichen Funktionen gehört auch die Funktion `destroy()`, die ein Objekt löscht, sowie der bereits erwähnte generische Konstruktor:

```
typedef struct crypter_t crypter_t;

struct crypter_t {
  void (*encrypt) (crypter_t *this, chunk_t iv, chunk_t data,
                   chunk_t *encrypted);
  void (*decrypt) (crypter_t *this, chunk_t iv, chunk_t encrypted,
                   chunk_t *data);
  void (*destroy) (crypter_t *this);
};

crypter_t *crypter_create(encryption_algorithm_t algorithm, chunk_t key);
```

2. Für jede Implementierung wird ein eigenes C-`struct` und ein eigener Konstruktur zum Erstellen eines spezifischen `crypter_t`'s verwendet. Die Deklaration der Klasse `aes_crypter_t` sieht dabei folgendermassen aus und kommt ins Header-File `aes_crypter.h`:

```
typedef struct aes_crypter_t aes_crypter_t;

struct aes_crypter_t {

  /* Vererbung des crypter_t interfaces */
  crypter_t crypter_interface;

  /* Hier könnten noch öffentliche Methoden für AES deklariert werden. */
};

aes_crypter_t *aes_crypter_create(chunk_t aes_key);
```

   Der Konstruktor `aes_crypter_create()` kann zwar auch selbst aufgerufen werden, ist aber prinzipiell dazu gedacht, von generischen Konstruktor `crypter_create()` aufgerufen zu werden.

3. Jede Implementierung des Interfaces `crypter_t` kann nun private Variablen und Methoden deklarieren. Dazu wird ein spezielles C-`struct` in die C-Datei der jeweiligen Implementierung eingefügt. Die `crypter_t`-Implementierung `aes_crypter_t` speichert beispielsweise den AES-Schlüssel als private Variable vom Typ chunk_t. Der folgende Code kommt in die Datei `aes_crypter.c`:

```
typedef struct private_aes_crypter_t private_aes_crypter_t;

struct private_aes_crypter_t{

   /* öffentlicher Teil (inkl. crypter_t-Interface) */
   aes_crypter_t public;

   /* Private Variablen */
   chunk_t aes_key;
};
```

4. Jede Methode des Interfaces wird als `static`-Methode implementiert und in die C-Datei der entsprechenden Implementierung eingefügt. Dies, damit der globale Namespace nicht verschmutzt wird. Als Beispiel soll hier die Implementierung der `crypter_t`-Methode `encrypt` der Klasse `aes_crypter_t` dienen:

```
static void encrypt (crypter_t *this, chunk_t iv, chunk_t data,
                         chunk_t *encrypted)
{
  /* Spezifische AES-Entschlüsselungsmethode aufrufen */

  return SUCCESS;
}
```

5. Zum Schluss wird der Konstruktor und Destruktor der entsprechenden Klasse geschrieben und auch in die C-Datei übernommen. Konstruktor und Destruktor von `aes_crypter_t` haben den folgenden Aufbau.

```
void aes_crypter_destroy(private_aes_crypter_t *this){

  /* Speicher des AES-Schlüssels freigeben */
  allocator_free_chunk(&(this->aes_key));

  /* Speicher für dieses Objekt freigeben */
  allocator_free(this);
}

aes_crypter_t *aes_crypter_create(chunk_t aes_key) {
  private_aes_crypter_t *this;

  /* Memory allozieren */
  this = allocator_alloc_thing(private_aes_crypter_t);

  /* Statische Funktionen als Objekt-Methoden angeben */
  this->public.crypter_interface.encrypt = encrypt;
  this->public.crypter_interface.decrypt = decrypt;
  this->public.crypter_interface.destroy = destroy;
```

```
  /* Datenfelder initiieren */
  this->aes_key.ptr = allocator_alloc(aes_key.len);
  this->aes_key.len = chunk_t.len;
  memcpy(this->aes_key.ptr,aes_key.ptr,aes_key.len);

  return &(this->public);
}
```

Die Klasse `aes_crypter_t` kann nun verwendet werden um Daten mit AES zu ver- und entschlüsseln. Wie diese Klasse verwendet wird, ist im nächsten Abschnitt beschrieben.

5.1.3.2 Verwendung der Klasse im Code

Die erstellte Klasse `aes_crypter_t` kann nun beinahe wie eine Klasse unter C++ verwendet werden. Der folgende Code erstellt ein `aes_crypter_t`-Objekt, verschlüsselt damit einen Datenbereich und entschlüsselt das Resultat danach wieder:

```
crypter_t *crypter;
chunk_t data;
chunk_t encrypted, decrypted;
chunk_t iv, key;

/* Wir nehmen an, dass iv, key, und data gültige Werte enthalten... */

/* Erstellen eines AES-Crypters */
crypter = crypter_create(AES, mein_key);

/* Daten verschlüsseln */
crypter->encrypt(crypter, iv, data, &encrypted);

/* wieder entschlüseln */
crypter->decrypt(crypter, iv, encrypted, &decrypted);

/* zum Schluss wird das crypter_t-Objekt wieder zerstört */
crypter->destroy(crypter);
```

5.1.4 Gründe für die Wahl des objektorientierten Ansatzes

Für den Betrachter stellt sich vermutlich die Frage, weshalb der Code beinahe objektorientiert geschrieben und nicht direkt C++ als Programmiersprache eingesetzt wird. Wir sehen folgende Vorteile, die für den objektorientierten Ansatz in C sprechen und uns auch für dessen Einsatz überzeugt haben:

- In einer Software wie `strongSwan` finden viele Algorithmen Verwendung, welche prinzipiell die gleiche Aufgabe wahrnehmen, aber verschieden implementiert werden. Als Paradebeispiel gilt die Verschlüsselung mit verschiedenen Algorithmen. Durch das Ansprechen dieser Algorithmen über eine einheitliche Schnittstelle ist die Software sehr einfach um weitere Algorithmen erweiterbar. Dies trifft nicht nur auf die symmetrische Verschlüsselung zu, auch auf asymmetrische, Hash-Algorithmen, Pseudo-Random-Funktionen und weitere.
- Beim Programmieren in C besteht die Gefahr, dass der Code durch unsauberes Design die Funktionalität nicht genügend an die dazugehörigen Daten kapselt. Wir wollen dem entgegenwirken, indem wir diese Kapselung in Klassen erzwingen. Wir glauben mit diesen Prinzipien alle Vorteile des objektorientierten Programmierens mit der stärke und Flexibilität der Programmiersprache C kombinieren zu können.
- Ein C-Entwickler muss beim gewählten objektorientierten Ansatz keine neue Programmiersprache erlernen, sondern nur die Thematik von Interfaces und Klassen verstehen. Umgekehrt fühlt sich auch ein C++-Entwickler wohl, indem er bekannte objektorientierte Ansätze so auch in C einsetzten kann.
- Durch diese Kapselung wird das Verwenden einer Klasse viel einfacher. Die interessanten Methoden sind ersichtlich und deren Implementierung bleibt verborgen. Dadurch wird die Übersichtlichkeit stark erhöht, da man sich lediglich um relevanten Code kümmern muss. Hat man das Prinzip verstanden, wirkt der Code einfacher und verständlicher.
- Bestandteile der Software sind leichter wiederzuverwenden. Ein sehr gutes Beispiel ist hier die Klasse `linked_list_t`, die beinahe in allen Klassen eingesetzt wird, die Dinge in einer Liste dynamisch speichern müssen.
- Die Objekt-Methoden sind im Code sehr einfach der entsprechenden Klasse zuzuordnen und es sind keine Prefix-Bezeichnungen notwendig. Beispielsweise haben die Klassen `send_queue_t` und `job_queue_t` je eine Funktion mit der Bezeichnung `add()`. Ohne den objektorientierten Ansatz müsste man diese Methoden `send_queue_add()` und `job_queue_add()` taufen, was den Code nicht gerade übersichtlicher machen würde.
- In einer Applikation mit mehreren Threads müssen Zugriffe auf gemeinsame Daten geregelt werden. Am besten wird der Zugriff auf Daten mit Methoden gekapselt, die das Locking selber realisieren. Bei der Verwendung von Objekten muss sich der Aufrufer nicht mehr selbst um die Synchronisation kümmern, was das Programmieren vereinfacht und Fehler vermeidet.

Natürlich hat der gewählte Ansatz auch seine negativen Seiten. So muss beispielsweise für jede Objektmethode Speicher alloziert und im Konstruktor initialisiert werden. Auch die Performance ist leicht schlechter, da Funktionspointer dereferenziert werden müssen und mehr Speicher dynamisch alloziert und dealloziert wird. Die Vorteile dieses Ansatzes haben uns schlussendlich überzeugen können, da wir die Performance-Einbusse als minimal betrachten und die dynamische Speicherallokation für heutige Linux-Systeme kein Problem mehr darstellt.

5.1.5 Packages

Um noch eine grössere Übersichtlichkeit unserer IKEv2-Implementierung zu erreichen, sind die verwendeten Klassen und Interfaces in so genannten Packages zusammengefasst. Ein solches Package besteht dabei aus Klassen und Interfaces mit ähnlichen Funktionalitäten.

Bei Programmiersprachen wie beispielsweise Java oder C# ist das Prinzip von Packages ein fester Teil der Programmiersprache – C bietet keine Unterstützung dafür. Damit in unserer IKEv2-Implementierung dieses Prinzip trotzdem eingesetzt werden kann, sind zusammengehörende Klassen in einem separaten Verzeichnis gespeichert, wobei das Verzeichnis selbst dem entsprechenden Package entspricht. Dieses Verzeichnis kann weitere Unterverzeichnisse enthalten, die einem Sub-Package entsprechen.

Eine Klasse kann aus irgend einer Sourcedatei mit folgenden include-Anweisungen eingebunden werden:

```
#include <package/klasse1.h>
#include <package/subpackage/klasse2.h>
```

Damit der Compiler die entsprechenden Header-Dateien auch findet, ist der include-Pfad auf das Wurzelverzeichnis des Quellcodes zu setzen.

5.2 Threading-Modell

Für die Implementierung von IKEv2 ist der Einsatz mehrerer Threads nicht zwingend notwendig, aber durchaus denkbar. Für die Architektur der Software ist die Wahl des Threading-Modells entscheidend. Mögliche Threading-Modelle werden nachfolgend diskutiert und deren Vor- und Nachteile aufgelistet. Abschliessend wird eines dieser Threading-Modelle ausgewählt, welches in der strongSwan IKEv2-Implementierung zum Einsatz kommen soll.

5.2.1 Single-Threaded

Eine Implementierung von IKEv2 als Single-Threaded-Applikation würde dem jetzigen Threading-Modell von pluto entsprechen. Dabei existiert nur ein einziger Thread, der alle anstehenden Aufgaben in einer Schlaufe nacheinander abarbeitet. Eine Single-Threaded-Implementierung hat Vorteile aber auch Nachteile gegenüber einer Multi-Threaded-Applikation. Die Vor- und Nachteile sind nachfolgend zusammengefasst.

5.2.1.1 Vorteile

- pluto ist als Single-Threaded-Applikation implementiert und kann dadurch als Referenz für die Architektur herangezogen werden.
- Die genutzten Speicherbereiche sind nicht durch spezielle Locks vor gleichzeitigem Zugriff zu schützen, da nur ein einziger Thread darauf zugreift.
- Es besteht keine Gefahr für Deadlocks aufgrund falsch eingesetzter Locks.

5.2.1.2 Nachteile

- Erweiterungen, wie beispielsweise eine OCSP-Unterstützung, die blockierende Aufrufe machen, sind in einer Single-Threaded-Applikation schwierig zu implementieren. Blockierende Aufrufe sollten den Gesamtablauf möglichst nicht stören.
- Eine Single-Threaded-Implementierung skaliert gegenüber einer Multi-Threaded-Implementierung schlechter, da auch unabhängige IKE_SAs sequenziell abgearbeitet werden.
- Timer-Events, welche zu einem gewissen Zeitpunkt ausgeführt werden sollten, können aufgrund eines blockierenden Aufrufs im Haupt-Thread erst viel später ausgeführt werden.
- Die vorhandene Unterstützung des Betriebssystems für die parallele Verarbeitung wird nicht in Anspruch genommen.

5.2.2 Ein Thread pro IKE_SA

Durch die Multi-Threading-Unterstützung von Linux ist eine `IKEv2`-Implementierung denkbar, welche für jede IKE_SA einen eigenen Thread erstellt und dieser dann für eine einzige IKE_SA mit mehreren CHILD_SAs zuständig ist. Bei diesem Modell wäre ein einzelner Thread einer IKE_SA zugeordnet und würde nur Aufgaben ausführen, die für diese IKE_SA bestimmt sind.

Dieses Modell hat keine Analogie zum jetzigen Single-Threaded-Modell von `pluto`. Die Vor- und Nachteile sind nachfolgend zusammengefasst.

5.2.2.1 Vorteile

- Die Unterstützung des Betriebssystems für Multi-Threaded-Applikationen wird voll in Anspruch genommen.
- Alle IKE_SAs können parallel bearbeitet und verwaltet werden.
- Vom Verständnis her ist dieses Threading-Modell einfacher zu verstehen als andere: Für jede IKE_SA ist ein „Mitarbeiter“ zuständig, welcher alle anfallenden Aufgaben für diese IKE_SA bearbeitet.
- Erweiterungen, wie beispielsweise eine OCSP-Unterstützung, die blockierende Aufrufe machen, sind einfacher zu implementieren. Blockierende Aufrufe betreffen jeweils nur eine einzelne IKE_SA und haben keinen Einfluss auf andere IKE_SAs und deren Threads.

5.2.2.2 Nachteile

- Dieses Threading-Modell skaliert relativ schlecht. Für jede IKE_SA muss ein Thread erstellt und verwaltet werden. In Gateways mit mehreren hundert Clients können die verfügbaren Ressourcen mit diesem Threading-Modell schnell erschöpfen.
- Neben der Erschöpfung der Ressourcen hat das Betriebssystem auch ein Limit bezüglich maximaler Anzahl Threads. Tests haben gezeigt, dass auf einem durchschnittlichen Linux-System maximal 512 Threads erstellt werden können. Eine Erhöhung dieser Limit kann nur durch ein Neukompilieren des Kernels erreicht werden.
- Die Erzeugung eines einzelnen Threads pro IKE_SA kann von einem potentiellen Angreifer ausgenutzt werden, um die Ressourcen eines Systems auszuschöpfen, indem dieser Anfragen zum Erstellen unterschiedlicher IKE_SAs an das betroffene System sendet.
- Von den Threads gemeinsam genutzte Speicherbereiche müssen durch Locks vom gleichzeitigen Zugriff geschützt werden. Werden diese Locks falsch implementiert, besteht beispielsweise die Gefahr eines Datenverlust.
- Der falsche Einsatz von Locks zum Schützen gemeinsamer Speicherbereiche kann zu Deadlock-Situationen führen.
- Dieses Threading-Modell hat keine grosse Gemeinsamkeiten mit dem derzeitigen Single-Threaded-Modell von `pluto`. `pluto` kann somit weniger gut als Referenz bei der Implementierung von `IKEv2` dienen.

5.2.3 Thread-Pool

Neben dem unter 5.2.2 beschriebenen Multi-Threading-Modell ist auch eine andere Lösung mit mehreren Threads denkbar. Diese basiert auf einem so genannten Thread-Pool. Ein solcher Thread-Pool besteht aus einer Anzahl Worker-Threads, welchen Aufgaben zugeordnet werden können, die dann abgearbeitet werden. Hat ein Worker-Thread seine Arbeit erledigt, wartet dieser so lange, bis wieder Arbeit ansteht und diese ihm zugeordnet wird.

Auch dieses Modell hat keine Analogie zum jetzigen Single-Threaded-Modell von `pluto`. Die Vor- und Nachteile sind nachfolgend zusammengefasst.

5.2.3.1 Vorteile

- Die Unterstützung des Betriebssystems für Multi-Threaded-Applikationen wird voll in Anspruch genommen.
- Mehrere IKE_SAs können parallel bearbeitet und verwaltet werden.
- Erweiterungen, wie beispielsweise eine OCSP-Unterstützung, die blockierende Aufrufe machen, sind einfacher zu implementieren. Blockierende Aufrufe betreffen jeweils nur eine einzelne IKE_SA und haben keinen Einfluss auf andere IKE_SAs und deren Threads.
- Dieses Multi-Threading-Modell skaliert sehr gut. Der Ressourcen-Verbrauch kann durch die Grösse des Thread-Pools geregelt werden. Die Ressourcen-Probleme des vorherigen Threading-Modells werden eliminiert.

5.2.3.2 Nachteile

- Von den Threads gemeinsam genutzte Speicherbereiche müssen durch Locks vom gleichzeitigen Zugriff geschützt werden.
- Der falsche Einsatz von Locks zum Schützen gemeinsamer Speicherbereiche kann zu Deadlock-Situationen führen.
- Da die Anzahl der Worker-Threads durch die Grösse des Thread-Pools begrenzt ist, können nicht alle IKE_SA parallel bearbeitet werden.
- Dieses Threading-Modell hat keine grosse Gemeinsamkeiten mit dem derzeitigen, Single-Threaded-Modell von `pluto` mehr.

5.2.4 Wahl des Threading-Modells

Nach genauem Abwägen von Vor- und Nachteilen der einzelnen Threading-Modelle haben wir uns für eine IKEv2-Implementierung auf Basis eines Thread-Pools entschieden. Folgende Gründe sprechen aus unserer Sicht für den Einsatz eines Thread-Pools:

- Anstehende Aufgaben können parallel abgearbeitet werden.
- Blockierende Aufrufe, wie beispielsweise das Abfragen eines Zertifikats über HTTP, haben keinen Einfluss auf parallel ablaufenden Threads. Erweiterungen die blockierende Aufrufe voraussetzen sind somit leicht zu implementieren.
- Der Ressourcen-Verbrauch kann durch die Grösse des Thread-Pools geregelt werden. Die Anzahl der tätigen Threads kann so nicht unendlich wachsen und dadurch die System-Ressourcen erschöpfen.

Natürlich hat auch dieses Threading-Modelle seine Nachteile, welche aber mit geeigneten Vorkehrungen minimiert werden können. So ist der gemeinsame Zugriff auf Speicherbereiche mit geeigneten Locks kontrollierbar. Und trotz des unterschiedlichen Threading-Modells von pluto können Teile des Codes als Referenz verwendet werden.

Die Implementation erfolgt über pthreads. Diese POSIX-Schnittstelle erlaubt die Verwendung der älteren LinuxThreads als auch den neueren NPTL-Threads. Mehr zu pthreads ist dem Dokument „Technologien“ zu entnehmen.

5.3 Threading-Architektur

Wie unter 5.2.4 zu erfahren ist, wird als Threading-Modell ein Thread-Pool eingesetzt. Neben den Worker-Threads des Thread-Pools sind zusätzlich weitere Threads vorgesehen, welche spezielle Aufgaben wahrnehmen. Die verschiedenen Thread-Typen und deren Aufgaben können folgender Tabelle entnommen werden.

Thread	Aufgabe
Worker	Ist Teil des Thread-Pools und erledigt die in einer Job-Queue anstehenden Arbeiten. Die Anzahl der Threads im Thread-Pool ist nicht vorgeschrieben. Denkbar sind ca. 10 bis 20 Worker-Threads.
Receiver	Nimmt eingehende UDP-Pakete entgegen und erstellt daraus Jobs, welche in die Job-Queue eingereiht werden. Kann zusätzliche Funktionen wie das Drosseln der Paket-Entgegennahme übernehmen, was beispielsweise unter zu hoher Systemlast sinnvoll erscheint.
Sender	Koordiniert das Absenden von UDP-Paketen. Entnimmt Pakete aus der Send-Queue und sendet diese via UDP-Socket an den Empfänger.
Scheduler	Verwaltet die Abarbeitung der Timer-Events in der Event-Queue. Erstellt aus eintreffenden Events entsprechende Jobs, die in die Job-Queue eingereiht werden. Somit werden Events schlussendlich von einem Worker-Thread abgearbeitet.
Kernel-Schnittstelle	Die Kernel-Schnittstelle besteht aus einem Thread, welcher Meldungen vom Kernel entgegen nimmt und diese verarbeitet. Diese Schnittstelle ist noch nicht vollständig implementiert, besteht jedoch bereits aus einem Thread der Meldungen des Kernels entgegen nehmen kann.

Tabelle 14: Thread-Typen und deren Aufgaben

Es sind weitere spezielle Threads denkbar. Zum Beispiel könnte ein Thread Befehle eines Steuerungsprogrammes (`whack`) entgegen nehmen und daraus Jobs erstellen und in die Job-Queue einreihen. Ein solcher Thread zur Entgegennahme von Steuerungskommandos wird während dieser Diplomarbeit nicht implementiert. Die Konfiguration des Daemons für diese Arbeit erfolgt hardcodiert im Quellcode. Unsere `IKEv2`-Implementierung wird jedoch so Entwickelt, das eine spätere Erweiterung mit einem solchen Steuerungs-Thread leicht möglich ist.

Die eigentliche Arbeit übernehmen die Worker-Threads. Sie entnehmen der Job-Queue anfallende Arbeitspakete und arbeiten diese ab. Da meist ein Job etwas mit einer bestimmten IKE_SA zu tun hat, wird die entsprechende IKE_SA in einem solchen Fall exklusiv gelockt. Mit diesem Vorgehen kann nur ein Thread gleichzeitig mit einer bestimmten IKE_SA arbeiten und Konflikte können vermieden werden.

Die nachfolgende Grafik soll die Realisierung mittels Queues und die Zuständigkeit der Threads verdeutlichen. Sie ist nicht komplett und dient nur dem Überblick. Die Pfeile stellen dabei den Datenfluss zwischen den einzelnen Queues, Objekten und Threads dar.

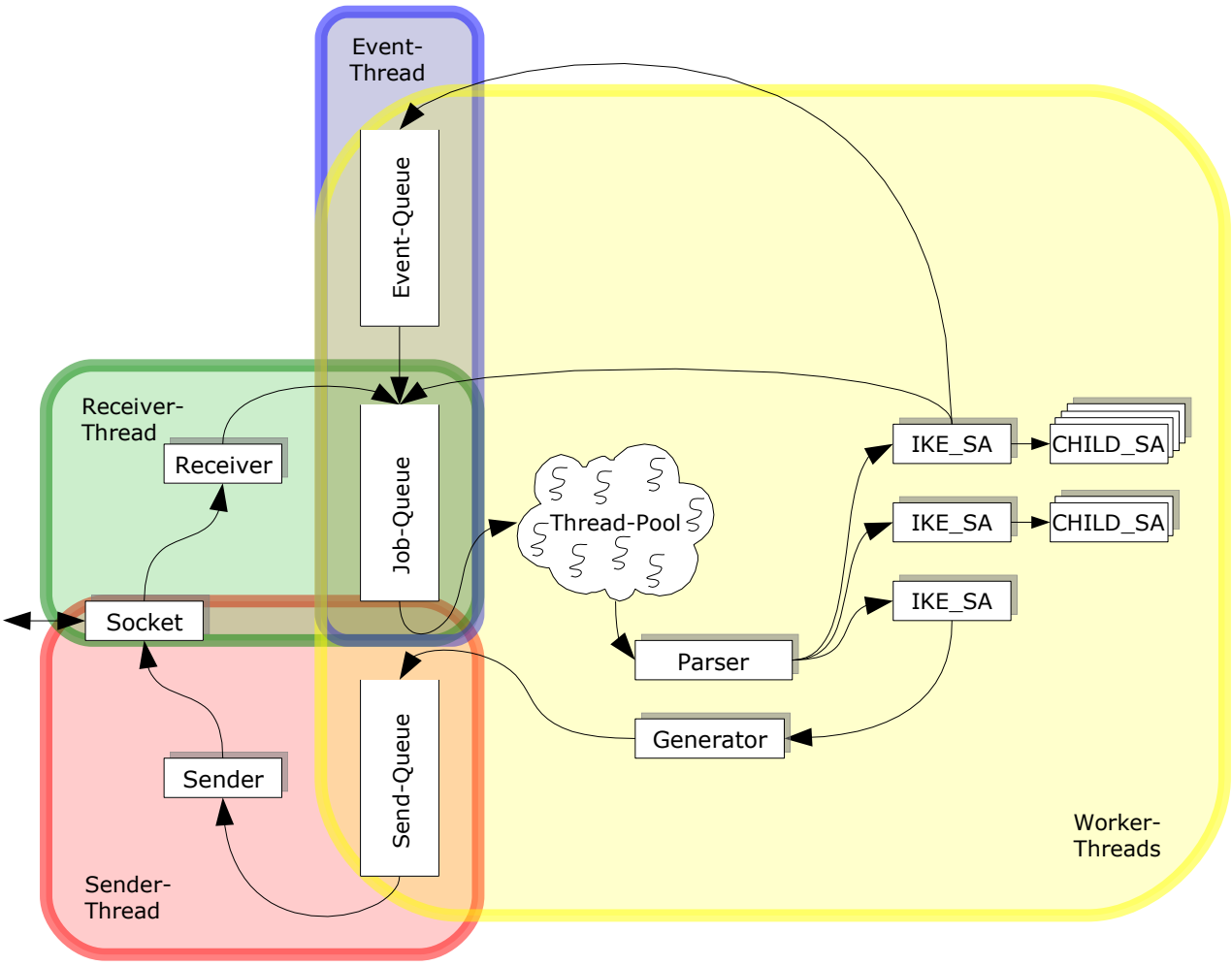


Abbildung 22: Threading-Architektur

5.4 Gesamtarchitektur

Die Gesamtarchitektur des Daemons kann am einfachsten mit einer Grafik beschrieben werden. Alle Objekte in folgender Abbildung sind während der ganzen Laufzeit des Daemons vorhanden.

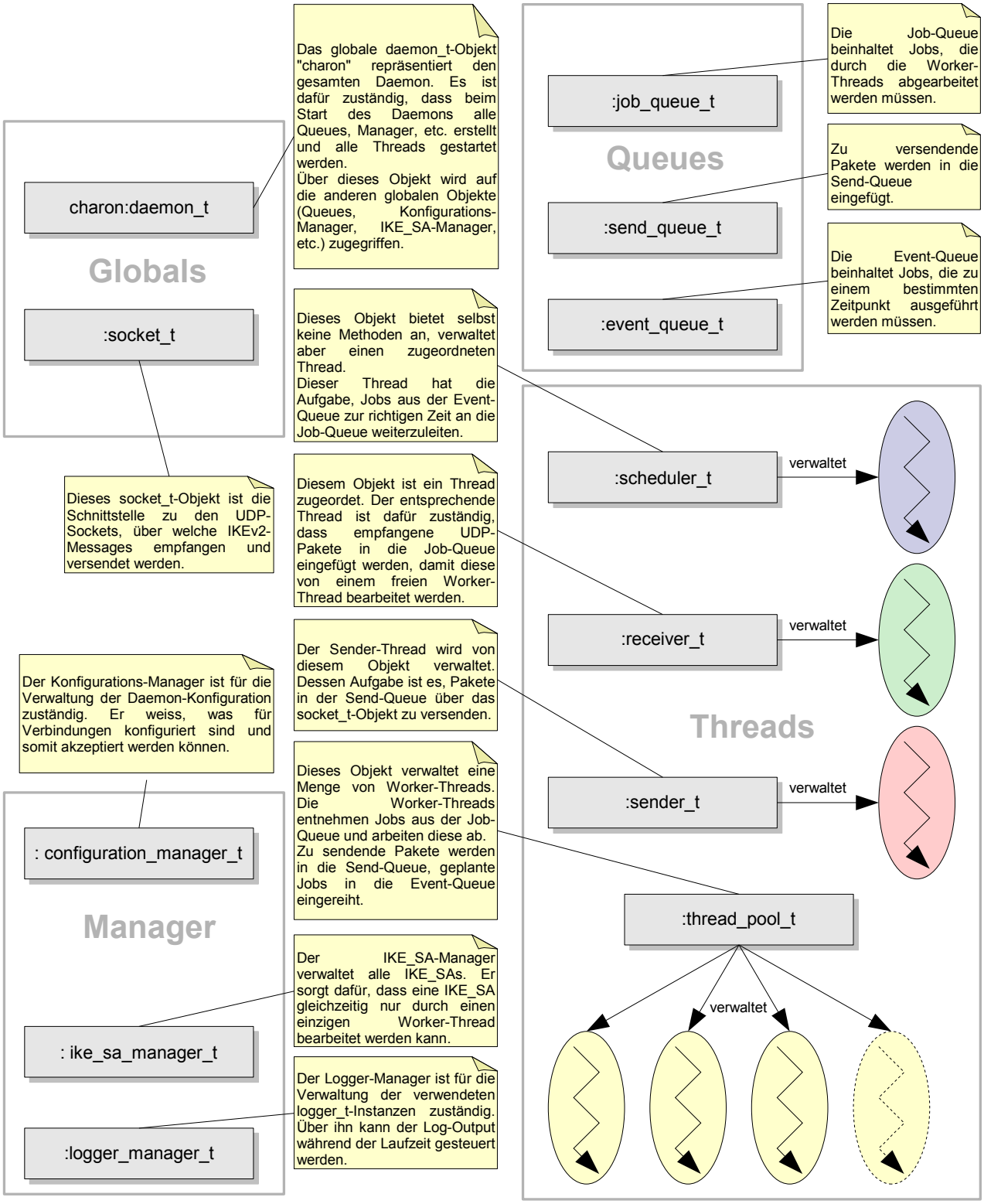


Abbildung 23: Gesamtarchitektur des Daemon

Das Herz des Daemons bildet das globale `daemon_t`-Objekt `charon`. Dieses wird beim Starten des Daemons initialisiert und ist verantwortlich für das Erstellen aller anderen globalen Objekte. So werden nicht nur alle Queues von diesem Objekt erstellt, sondern auch die einzelnen Manager und Thread-Objekte. Auch der spätere Zugriff auf die globalen Objekte geht über das Objekt `charon`. Dazu bietet es öffentliche Variablen, über welche der Zugriff auf die Objekte erlangt werden kann.

Die Kernel-Schnittstelle beinhaltet auch einen Thread, ist aber in obiger Grafik nicht eingezeichnet. Dies, weil die Kernel-Schnittstelle sich erst im Aufbau befindet und noch nicht richtig eingesetzt werden kann.

5.5 Packages

Wie bereits unter 5.1.5 beschrieben, ist diese IKEv2-Implementierung in Packages strukturiert. Aus der Architektur ergibt sich folgendes Package-Diagramm:

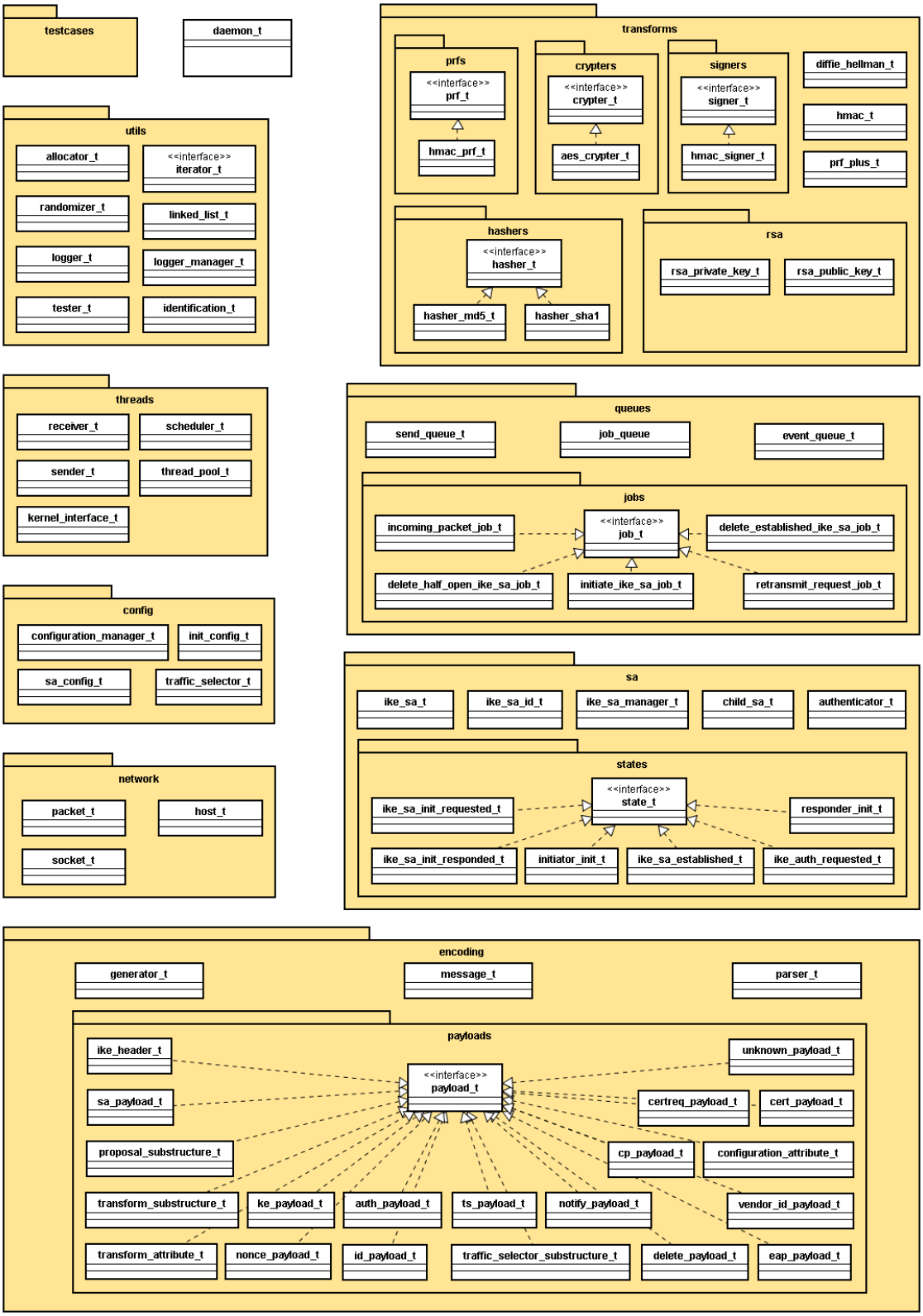


Abbildung 24: Architektur als Package-Diagramm

Jedes Package fasst Klassen und Interfaces mit ähnlicher Funktionalität zusammen. Neben den oben eingezeichneten Klassen und Interfaces beinhalten manche Packages weitere Dateien. Diese umfassen beispielsweise Typendefinitionen und sind aus Gründen der Übersichtlichkeit nicht im Diagramm abgebildet.

Um die Zusammenhänge besser zu verstehen, sind die Packages und deren Sub-Packages in nachfolgender Tabelle kurz beschrieben. Die detaillierten Beschreibungen der Packages folgen im Anschluss an diese Tabelle.

Package	Inhalt
utils	Beinhaltet allgemein nützliche Klassen, die in diversen anderen Packages eingesetzt werden.
threads	Beinhaltet Klassen zur Verwaltung der Threads, die für den Betrieb des Daemons verwendet werden. Die Klasse thread_pool_t ist für die Verwaltung aller Worker-Threads zuständig. Die einzelnen Threads werden beim Start des Daemons initialisiert und gestartet.
config	Dieses Package besteht aus Klassen, die sich um die Konfiguration des Daemons kümmern.
sa	Das Package sa fasst Funktionalitäten zusammen, die sich um die Verwaltung der Security Associations (SAs) kümmern.
sa/states	Eine IKE_SA kann sich in verschiedenen Zuständen befinden. Diese unterschiedlichen States werden durch eigene Klassen repräsentiert, die in diesem Sub-Package zusammengefasst sind.
queues	Besteht aus Queues zur asynchronen Kommunikation der eingesetzten Threads. Die Queues basieren intern auf einer Linked-List und sind vor gleichzeitigem Zugriff geschützt.
queues/jobs	Ein Job wie beispielsweise ein zu verarbeitendes UDP-Paket wird intern als Job-Objekt repräsentiert, welches in die Event- oder Job-Queue eingefügt werden kann. Die unterschiedlichen Job-Typen sind in diesem Sub-Package definiert.
transforms	Dieses Package beinhaltet Klassen, die einen so genannten „Transform“ darstellen. Im IKEv2-Draft werden ausgehandelte Algorithmen und Funktionen als „Transforms“ bezeichnet. So werden beispielsweise Verschlüsselungsalgorithmen als auch Pseudo-Random-Funktionen als Transforms zusammengefasst.
transforms/crypters	In diesem Sub-Package sind „Transforms“ enthalten, die für die symmetrische Verschlüsselung verwendet werden können.
transforms/hashers	In diesem Sub-Package sind „Transforms“ enthalten, die für das Hashing verwendet werden können.
transforms/prfs	In diesem Sub-Package sind „Transforms“ enthalten, die eine Pseudo-Random-Funktion darstellen.
transforms/signers	In diesem Sub-Package sind „Transforms“ enthalten, die zur Erstellung von symmetrischen Signaturen (MACs) verwendet werden können.
transforms/rsa	In diesem Sub-Package sind Klassen enthalten, welche Funktionen des RSA-Algorithmus implementieren.
testcases	Aus dem obigen Package-Diagramm entsteht der Eindruck, dass es sich bei diesem Package um ein leeres Package handelt. Dies ist natürlich nicht der Fall. Stattdessen enthält dieses Package alle Modultests. Da die einzelnen Modultest nur aus separaten Testfunktionen bestehen und nicht als Klassen modelliert sind, sind diese im Package-Diagramm nicht ersichtlich.
network	Wie der Name vermuten lässt, beinhaltet dieses Package Klassen rund um das Thema Netzwerk.
encoding	Das Package encoding befasst sich mit der Umwandlung von IKEv2-Messages in interne Datenstrukturen und daraus wieder in Messages. So bilden Parser und Generator auch die Hauptbestandteile dieses Packages.
encoding/payloads	Die einzelnen Payloads einer Message werden als Objekte vom Typ payload_t abstrahiert. Jedes der unterstützten Payloads ist in diesem Sub-Package definiert.

Tabelle 15: Kurzbeschreibung der Packages

5.5.1 network

Das Package `network` enthält Klassen, die für das Versenden und Empfangen von UDP-Paketen erforderlich sind.

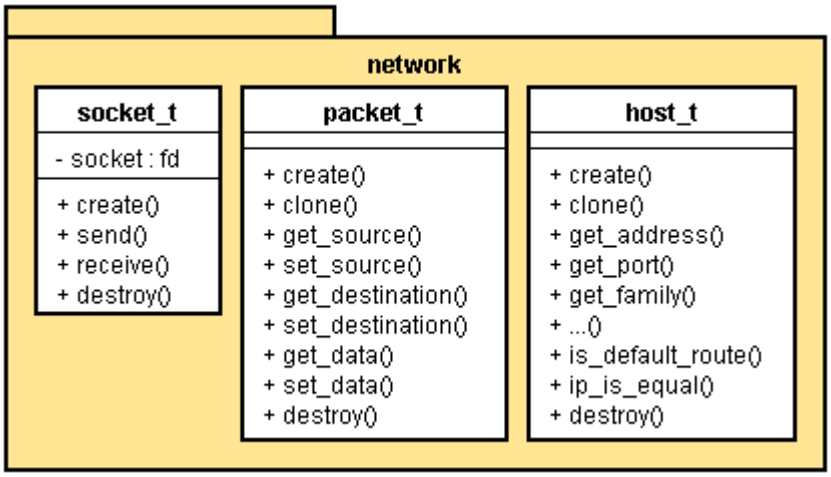


Abbildung 25: Klassen des Packages "network"

5.5.1.1 socket_t

Die Socket-Klasse `socket_t` verwaltet und abstrahiert einzelne Sockets. Über sie können Sende- und Empfangsmethoden verwendet werden, die auf die darunter liegenden UDP-Sockets abgebildet sind.

In der jetzigen Implementierung ist lediglich ein einziger IPv4 Socket vorhanden und unterstützt, der auf alle Interfaces gebunden wird. In einer Erweiterung könnten weitere Sockets für spezifische Interfaces, mehrere Ports oder auch andere Protokolle (IPv6) integriert werden. Die Schnittstelle zur Socket-Klasse müsste im Falle einer solchen Erweiterung nur geringfügig angepasst werden.

Der Konstruktor erwartet als Parameter den UDP-Port, auf welchem der Socket gebunden werden soll. Die Sendemethode nimmt ein `packet_t`-Objekt entgegen, welches eine Zieladresse in Form eines `host_t`-Objekts enthalten muss. Die Empfangsmethode liefert ein `packet_t`-Objekt zurück, welches neben den eigentlichen Paketdaten auch die Absenderadresse in Form eines `host_t`-Objekts beinhaltet.

Das globale `daemon_t`-Objekt `charon` verwaltet ein `socket_t`-Objekt, auf welches vom Receiver- und vom Sender-Thread zugegriffen wird.

5.5.1.2 packet_t

Die Klasse `packet_t` stellt eine Abstraktion eines UDP-Paketes dar. Es enthält Absender- sowie Empfangsadresse in Form von `host_t`-Objekten. Die eigentlichen Paketdaten sind intern in einem `chunk_t` gespeichert. Für den Zugriff auf diese Daten sind mehrere Methoden definiert. Zudem bietet die Klasse eine `clone`-Methode, welche ein `packet_t`-Objekt dupliziert.

5.5.1.3 host_t

`host_t` vereinfacht den Umgang mit der, unter Linux verwendeten, `sockaddr`-Struktur für Internet-Addressen. Die Klasse `host_t` kapselt eine `sockaddr`-Struktur und ist mit einfachen Methoden versehen. Der Konstruktor nimmt die Adressfamilie entgegen (derzeit lediglich IPv4), einen String mit der Adresse sowie den Port. Adresse und Port können wieder über entsprechende Methoden ausgelesen werden. Zudem sind noch weitere, spezifische Methoden für die Konvertierung der Adressdaten implementiert.

Die Klasse `host_t` ist so aufgebaut, dass eine IPv6-Unterstützung leicht eingebaut werden kann.

5.5.2 threads

Im Package `threads` sind Klassen zusammengefasst, die mit eigenen Threads gewisse Aufgaben erfüllen. Dazu gehört der Thread-Pool mit mehreren Worker-Threads, als auch Klassen die selbstständig das Versenden und Empfangen von Paketen erledigen.

Von jeder Klasse dieses Packages wird beim Starten des Daemons eine Instanz erstellt, die während der ganzen Lebenszeit erhalten bleibt.

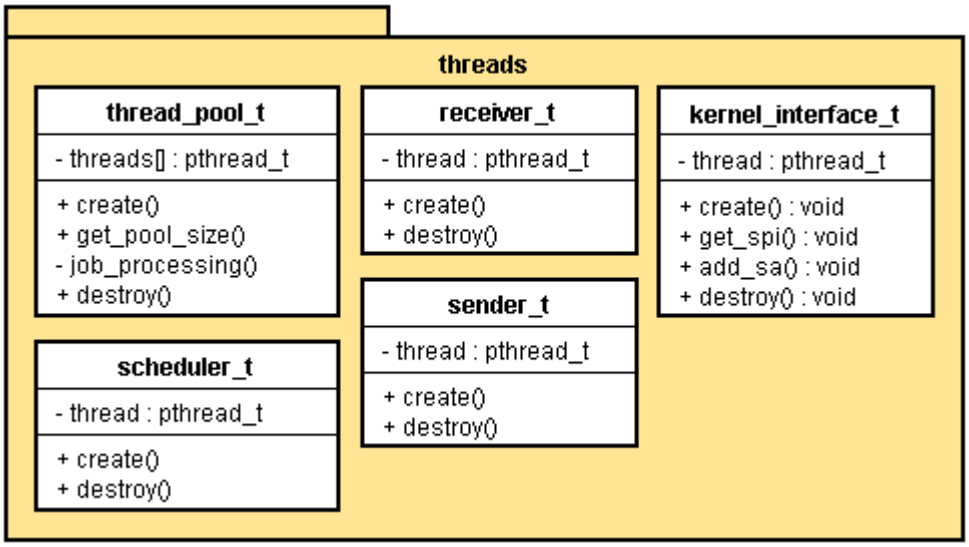


Abbildung 26: Klassen des Packages "threads"

5.5.2.1 thread_pool_t

Der Thread-Pool, implementiert in der Klasse `thread_pool_t`, ist für die Verwaltung aller Worker-Threads verantwortlich. Er instanziert alle Worker-Threads und terminiert diese bei Bedarf auch wieder. Der grobe Programmablauf der Worker-Threads ist ebenfalls im Thread-Pool definiert. Dieser Ablauf beinhaltet:

- Beziehen eines Jobs von der Job-Queue
- Auschecken bzw. Kreieren der zum Job gehörenden IKE_SA über den IKE_SA-Manager
- Verarbeiten des Jobs durch die entsprechende IKE_SA
- Einchecken der IKE_SA über den IKE_SA-Manager

Die Schnittstelle des Thread-Pools beinhaltet Methoden zum Kreieren bzw. Zerstören eines Pools. In der jetzigen Implementierung wird eine bestimmte Anzahl Threads instanziert. Diese nehmen sofort die Abarbeitung der anstehenden Jobs auf. Ein komplexerer Lösungsansatz für eine Verwaltung der eingesetzten Worker-Threads wäre denk- und auch leicht implementierbar. So könnte der Thread-Pool bei Bedarf die Threads dynamisch erstellen und terminieren.

Damit der Thread-Pool Threads terminieren kann, müssen gewisse Bedingungen gegeben sein. Im Ablauf eines Worker-Threads müssen „cancellation points“ definiert werden, in welchen der Thread sicher beendet werden kann. Wenn der Thread-Pool eine Anforderung zur Beendigung an den Thread sendet, muss dieser beim nächsten „cancellation point“ beenden. Damit ein möglicher „cancellation point“ vom Thread benutzt wird, muss vor diesem der „cancellation state“ eingeschaltet, bzw. nach dem „cancellation point“ dieser Status wieder deaktiviert werden. Mehr Details zum Thema Threads und „cancellation points“ können dem Dokument „Technologien“ entnommen werden.

5.5.2.2 scheduler_t

Die Klasse `scheduler_t` implementiert den Scheduler-Thread. Die Aufgabe dieses Threads ist es, Jobs aus der Event-Queue zum richtigen Zeitpunkt in die Job-Queue einzureihen, damit diese dann von den Worker-Threads abgearbeitet werden.

Da das eigentliche Event-Handling in der Event-Queue selbst implementiert ist, hat dieser Thread einzig die Aufgabe, aufgetretene Events weiterzuleiten.

5.5.2.3 receiver_t

Die Klasse `receiver_t` implementiert den Receiver-Thread. Ein Objekt dieser Klasse instanziert einen Thread, der das Empfangen von Paketen über die globale Instanz vom Typ `socket_t` erledigt. Die einzige Aufgabe des Receiver-Threads besteht darin, Pakete vom `socket_t`-Objekt entgegen zu nehmen, in einen Job zu verpacken und diesen anschliessend für die weitere Abarbeitung in die Job-Queue zu legen.

Eine mögliche Erweiterung der Funktionalität wäre das automatische Drosseln von erstellten Jobs bei einer nicht mehr zu bewältigenden Paketflut.

Der Ablauf beim Empfangen von Paketen ist unter 5.6.6 als Ablaufdiagramm beschrieben.

5.5.2.4 sender_t

Der in einem Objekt vom Typ `sender_t` verwaltete Thread übernimmt die umgekehrte Aufgabe wie der Thread in einem Objekt vom Typ `receiver_t`. Er entnimmt Pakete aus der Send-Queue und schickt sie über den Socket an den Empfänger.

Der Ablauf beim Versenden von Paketen ist unter 5.6.5 als Ablaufdiagramm beschrieben.

5.5.2.5 kernel_interface_t

Die Klasse `kernel_interface_t` soll die Grundlage für eine spätere Kernel-Anbindung darstellen. Mit einem Objekt von diesem Typ kann mit dem Kernel kommuniziert werden. Die Kommunikation erfolgt dabei über die `Netlink`-Schnittstelle.

Zur Zeit ist ein Thread implementiert, der Meldungen des Kernels entgegen nimmt, diese aber noch nicht verarbeitet. Auch ist es schon möglich, eine SPI vom Kernel zu beziehen sowie eine SA für `ESP` oder `AH` einzurichten.

5.5.3 sa

Das Package `sa` beinhaltet Klassen zur Verwaltung von IKE_SAs und CHILD_SAs. Dieses Klassen bilden ein zentraler Teil dieser `IKEv2`-Implementierung. Beinahe die ganze Verwaltung und Bearbeitung der unterschiedlichen SAs erfolgt durch Klassen in diesem oder untergeordneten Packages.

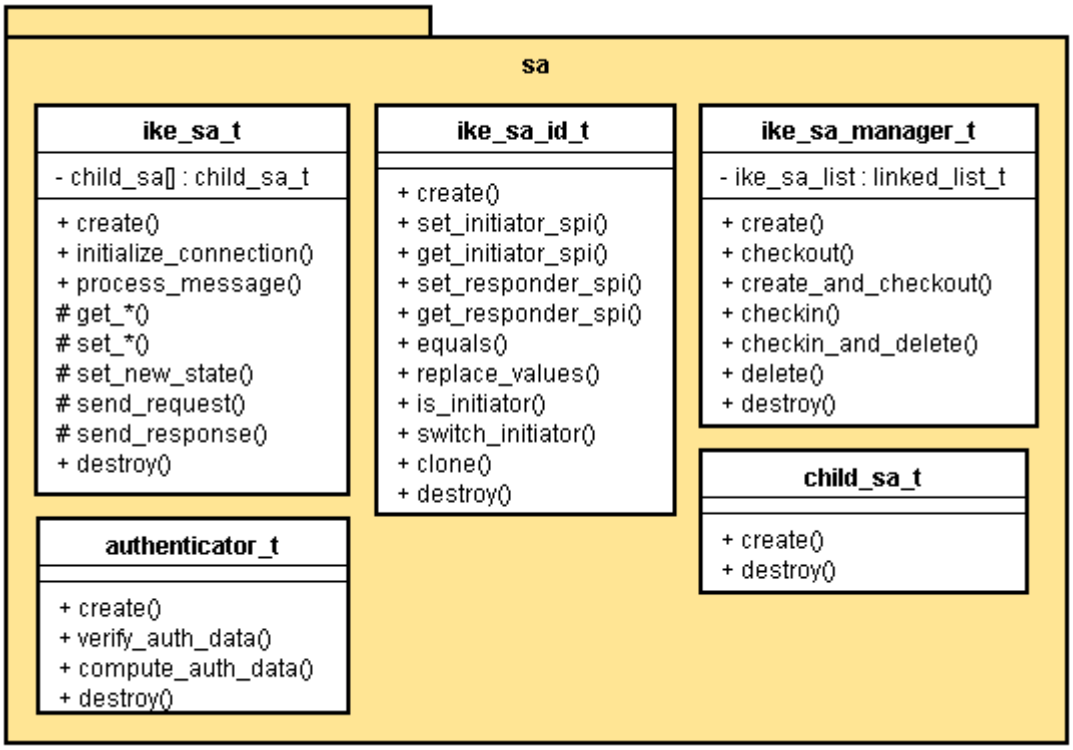


Abbildung 27: Klassen des Packages "sa"

5.5.3.1 ike_sa_t

Ein Objekt der Klasse `ike_sa_t` repräsentiert eine IKE_SA. Ein solches Objekt wird mit einem zugeordneten `ike_sa_id_t`-Objekt eindeutig identifiziert (siehe 5.5.3.2). Über eine IKE_SA müssen unter folgende Informationen gespeichert werden:

- Adressinformationen von Initiator und Responder (IPs und Ports)
- Schlüsselmaterial für Verschlüsselung, Signierung, etc.
- Aktueller Zustand der IKE_SA
- Eindeutige Identifikation der IKE_SA
- SPIs von Initiator und Responder
- Zugeordnete CHILD_SAs
- usw.

Neben diesen Informationen bietet die IKE_SA auch folgende Funktionalitäten:

- Bearbeiten einer eingehenden Message
- Aufbauen einer neuen Verbindung

Der aktuelle Zustand einer IKE_SA wird als Objekt vom Typ `state_t` realisiert. Zu bearbeitende Messages werden nicht direkt vom `ike_sa_t`-Objekt bearbeitet, sondern an das aktuell zugeordnete `state_t`-Objekt weitergeleitet. Falls eine Message erfolgreich vom `state_t`-Objekt bearbeitet wurde, wechselt dieses selber den aktuellen Zustand des `ike_sa_t`-Objektes. Es erstellt dazu das nächste `state_t`-Objekt, initialisiert dieses und setzt es als neues `state_t`-Objekt mit Hilfe der Funktion `set_new_state()`.

Mit dieser Methode zur Zustandsverwaltung ergeben sich entscheidenden Vorteile:

- Das `ike_sa_t`-Objekt muss sich nicht um die Zustandsverwaltung kümmern und somit auch keine komplizierte Zustands-Maschine implementieren. Die Klasse wird dadurch übersichtlicher und die Fehleranfälligket verringert sich.
- Die zu bearbeitende Message wird immer vom aktuellen `state_t`-Objekt bearbeitet. Da dieses Objekt stets weiss, was für ein Zustand es repräsentiert, kann eine unerlaubte Message (Beispielsweise ein CREATE_CHILD_SA-Request während der IKE_SA_INIT-Phase) einfach erkannt und verworfen werden.
- Nach der Bearbeitung einer Message weiss das aktuelle `state_t`-Objekt, in welchen Zustand die IKE_SA wechseln muss und kann diesen Zustandswechsel auch selbst durchführen.

Ein ähnliches Verhalten ergibt sich beim Aufbauen einer neuen Verbindung. Der Verbindungsaufbau wird auch wieder an das aktuelle `state_t`-Objekt weitergeleitet, welches die notwendige Message erstellt, in die Send-Queue einreiht und dann nächsten Zustand der IKE_SA erstellt und setzt.

Die Zugriffe auf ein `ike_sa_t`-Objekt sind vor gleichzeitigem Zugriff geschützt, indem eine zu bearbeitende IKE_SA immer über den IKE_SA-Manager[1] ausgecheckt und nach Gebrauch wieder dort eingecheckt werden muss. Der IKE_SA-Manager stellt den geschützten Zugriff auf die ausgecheckten IKE_SAs sicher.

Der Zugriff auf die IKE_SAs erfolgt prinzipiell nur aus den Worker-Threads.

5.5.3.2 ike_sa_id_t

Ein Objekt der Klasse `ike_sa_id_t` abstrahiert die Identifikation einer IKE_SA. Eine IKE_SA ist durch folgende Angaben eindeutig identifizierbar:

- SPI des Initiators (8 Byte)
- SPI des Responders (8 Byte)
- Rolle des laufenden Daemons für diese IKE_SA (Initiator oder Responder)

Als Initiator gilt derjenige Kommunikationspartner, welcher den IKE_SA_INIT-Austausch initiiert hat. Diese Rolle ändert sich für die entsprechende IKE_SA nur, wenn eine IKE_SA durch einen späteren CREATE_CHILD_SA-Request vom Responder aufgefrischt wird.

Objekte vom Typ `ike_sa_id_t` können einfach verglichen, geändert oder geklont werden. Der globale IKE_SA-Manager speichert intern für jede IKE_SA ein zugeordnetes `ike_sa_id_t`-Objekt und nutzt die Vergleichsfunktionen dieser Klasse, um eine bestimmte IKE_SA aufzufinden.

1 Objekt vom Typ ike_sa_manager_t.

5.5.3.3 ike_sa_manager_t

Wie schon erwähnt, repräsentiert ein Objekt vom Typ `ike_sa_t` eine IKE_SA. Alle Instanzen von `ike_sa_t` müssen in irgend einer Weise zentral verwaltet werden können. In einer Single-Threaded-Implementierung könnte dies durch eine einfache globale Liste erreicht werden. Diese Multi-Threading-Implementierung stellt diverse zusätzliche Anforderungen an eine zentrale IKE_SA-Verwaltung, die nicht mehr mit einer simplen Liste erfüllt werden können:

- Die Liste der verwalteten `ike_sa_t`-Objekte darf nicht von mehreren Threads gleichzeitig geändert werden.
- Ein `ike_sa_t`-Objekt darf zu einem Zeitpunkt nur durch einen Thread[1] bearbeitet werden.

In dieser `IKEv2`-Implementierung existiert ein Objekt vom Typ `ike_sa_manager_t`, welches alle `ike_sa_t`-Objekte zentral verwaltet. Dieses Objekt ist über die gobale `daemon_t`–Instanz erreichbar. Gegenüber einer einfachen Liste bietet dieses `ike_sa_manager_t`-Objekt folgende zusätzliche Funktionalitäten, mit denen die oben beschriebenen Anforderungen erfüllt werden:

- Eine IKE_SA kann anhand der Identifikation vom Typ `ike_sa_id_t` gesucht und ausgecheckt werden.
- Der Zugriff auf ein `ike_sa_t`-Objekt wird nur dann erlaubt, wenn kein anderer Thread dieses bearbeitet. Falls bereits ein Thread eine bestimmte IKE_SA bearbeitet, werden weitere Threads so lange blockiert, bis sie den exklusiven Zugriff auf dieses `ike_sa_t`-Objekt erhalten.
- Eine IKE_SA wird erst gelöscht, wenn kein Thread mehr die entsprechende IKE_SA bearbeitet oder auf den exklusiven Zugriff darauf wartet.
- Die Liste der verwalteten `ike_sa_t`-Objekte kann gleichzeitig nur von einem einzigen Thread bearbeitet oder durchsucht werden.

Diese beschriebenen Funktionalitäten erfordern die Verwendung von Locks (Mutexe) in Zusammenarbeit mit Conditional-Variablen.

Die Klasse `ike_sa_manager_t` speichert die verwalteten `ike_sa_t`-Objekte intern in einer Linked-List. Diese interne Liste besteht aus einzelnen Einträgen vom Typ `ike_sa_entry_t`. Die gesamte Liste wird beim Zugriff (IKE_SA beziehen, IKE_SA löschen, IKE_SA zurückgeben) exklusiv gelockt.

Damit eine IKE_SA jeweils nur von einem einzigen Thread gleichzeitig bearbeitet werden kann, wird ein Checkin/Checkout-System eingesetzt. Eine IKE_SA muss so zur Bearbeitung vorher beim globalen IKE_SA-Manager ausgecheckt und nachher zwingend wieder eingecheckt werden.

Wie der Checkin/Checkout-Mechanismus realisiert wird, ist in den Ablaufszenarien 5.6.7 und 5.6.8 genauer ersichtlich.

1 Hier ist mit Thread ein Worker-Thread gemeint.

5.5.3.4 authenticator_t

Im IKE_AUTH-Austausch authentisieren sich Initiator und Responder gegenseitig. `IKEv2` unterstützt diverse Mechanismen, wie diese Authentisierung durchgeführt werden kann. So ist es beispielsweise möglich, den Initiator über ein gemeinsames Passwort und den Responder über ein RSA-Schlüssel zu authentisieren. Die Klasse `authenticator_t` bietet eine unabhängige Schnittstelle, mit welcher die Authentisierung des Kommunikationspartners durchgeführt werden kann.

Die Authentisierung erfolgt bei `IKEv2` durch die Signierung eines bestimmten Datenblocks. Dazu bietet die Klasse `authenticator_t` eine Funktion zur Erzeugung und eine Funktion zur Überprüfung der Signatur.

5.5.3.5 child_sa_t

Die Klasse `child_sa_t` soll die Grundlage für eine spätere Abstraktion einer CHILD_SA darstellen. Die jetzige `child_sa_t`-Klasse implementiert noch keine Funktionalität.

5.5.4 states

`states` ist ein Sub-Package vom Package `sa` und beinhaltet die unterschiedlichen Zwischen-Zustände einer IKE_SA als `state_t`-Implementierungen.

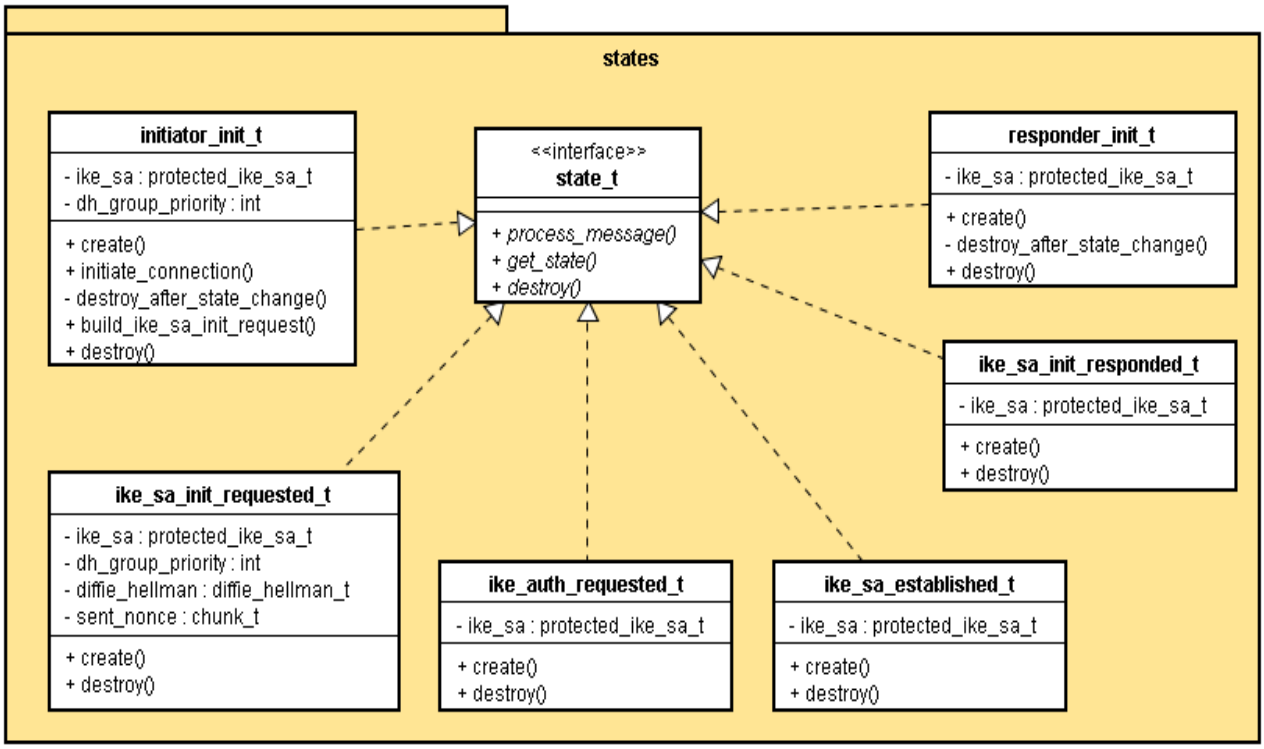


Abbildung 28: Klassen des Sub-Packages "states"

Bevor eine IKE_SA vollständig aufgebaut ist, kann sie sich in mehreren Zwischenzuständen befinden. Jeder dieser Zustände ist als Klasse im Package `states` definiert und implementiert das Interface `state_t`. Wird eine Verbindung als Intitiator aufgebaut, befindet sich die IKE_SA zu Beginn im Zustand INITIATOR_INIT, der in der Klasse `initiator_init_t` implementiert ist; Wird eine Verbindung als Responder aufgebaut, befindet sich die IKE_SA zu Beginn im Zustand RESPONDER_INIT, welcher in der Klasse `responder_init_t` implementiert ist.

Das Interface `state_t` bietet nur die Funktion `process_message()`, die von der IKE_SA für jede empfangene Message aufgerufen wird. Es ist die Aufgabe der einzelnen `state_t`-Implementierungen die entsprechenden Messages zu prüfen und zu verarbeiten.

Die Zwischenzustände können am einfachsten mit Hilfe eines Zustandsdiagramms erläutert werden:

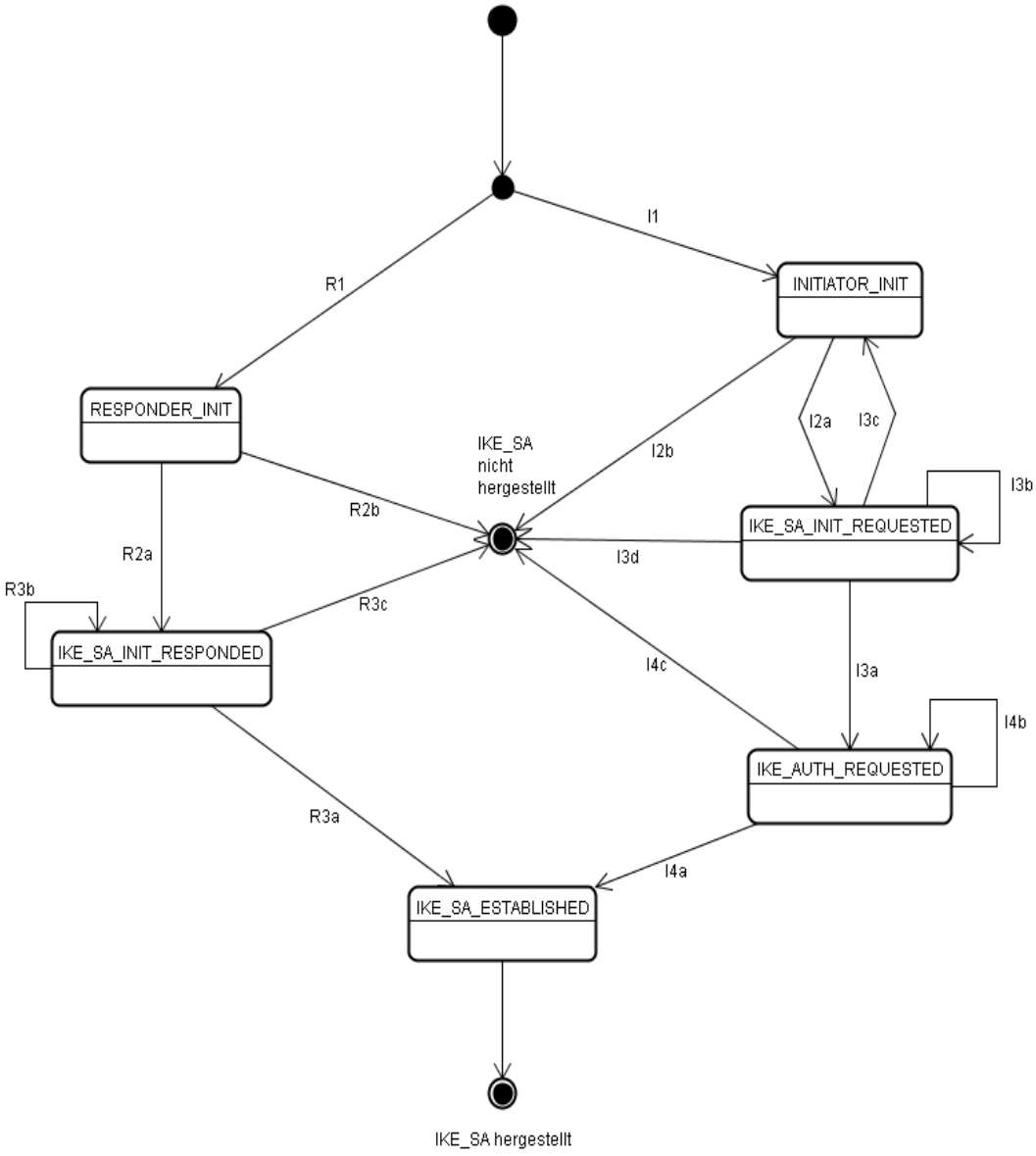


Abbildung 29: Zustandsdiagramm der IKE_SA

Die einzelnen Klassen im Sub-Package `states` entsprechen je einem Zustand aus obigem Zustandsdiagramm. Folgende Zuordnung gilt:

IKE_SA-Zustand	**Implementiert in Klasse**
INITIATOR_INIT	`initiator_init_t`
RESPONDER_INIT	`responder_init_t`
IKE_SA_INIT_RESPONDED	`ike_sa_init_responded_t`
IKE_SA_INIT_REQUESTED	`ike_sa_init_requested_t`
IKE_AUTH_REQUESTED	`ike_auth_requested_t`
IKE_SA_ESTABLISHED	`ike_sa_established_t`

Tabelle 16: IKE_SA-Zustände und deren Implementierungen

Bereits beim Erstellen eines `ike_sa_t`-Objekts entscheidet sich, was für eine Startzustand in Form eines `state_t`-Objekts gesetzt wird. Soll die entsprechende IKE_SA als Initiator aufgebaut werden, so erstellt der `ike_sa_t`-Konstruktor automatisch ein Objekt vom Typ `initiator_init_t` und setzt dieses als aktuellen Zustand (I1). Wird ein IKE_SA_INIT-Request empfangen, so erstellt der `ike_sa_t`-Konstruktor automatisch ein Objekt vom Typ `responder_init_t` und setzt dieses als aktuellen Zustand (R1).

Die Zustandsklassen werden nun genauer betrachtet. Es wird beschrieben, wie es zu den einzelnen Zustandsübergängen aus dem obigen Zustandsdiagramm kommt. Es werden nur die wichtigsten Bedingungen aufgeführt, die für einem bestimmten Zustandsübergang erfüllt sein müssen.

5.5.4.1 responder_init_t

Der erste Zustand eines `ike_sa_t`-Objekts, welches aufgrund eines empfangenen IKE_SA_INIT-Requests erstellt wurde, ist vom Typ RESPONDER_INIT und in der Klasse `responder_init_t` implementiert. Diese Klasse implementiert das Interface `state_t` und bietet somit die Funktion `process_message()`. Die Funktion `process_message()` erwartet einen Request vom Typ IKE_SA_INIT.

Beim Ausführen der Funktion `process_message()` mit der zugeordneten Message sind folgende Zustandsübergänge möglich:

Bedingungen	**Übergang**	**Auswirkung**
– Die Message ist ein Request vom Typ IKE_SA_INIT – Die bearbeitete Message konnte erfolgreich geparst und deren Aufbau verifiziert werden – Ein angebotenes Proposal kann akzeptiert werden – Die gewählte Diffie-Hellman-Gruppe kann akzeptiert werden	R2a	– Eine IKE_SA_INIT-Response wird an den Initiator gesendet – Ein neues Objekt vom Typ `ike_sa_init_responded_t` wird erstellt und als neuer Zustand der IKE_SA gesetzt – Das aktuelle Zustandsobjekt wird gelöscht
– Die Message ist kein Request vom Typ IKE_SA_INIT oder – Die Message konnte nicht erfolgreich geparst oder deren Aufbau verifiziert werden oder – Keine der angebotenen Proposals kann akzeptiert werden oder – Die gewählte Diffie-Hellman-Gruppe kann nicht akzeptiert werden	R2b	– Die IKE_SA und somit auch deren aktueller Zustand wird zerstört

Tabelle 17: Zustandsübergange aus der Klasse responder_init_t

5.5.4.2 ike_sa_init_responded_t

Die `process_message()`-Implementierung dieser Zustandsklasse erwartet einen Request vom Typ IKE_AUTH.

Aus diesem Zustand gibt es folgende Zustandsübergänge:

Bedingungen	Übergang	Auswirkung
– Die Message ist ein Request vom Typ IKE_AUTH – Die bearbeitete Message konnte erfolgreich entschlüsselt, geparst und deren Aufbau verifiziert werden – Die Identität im IKE_AUTH-Request konnte authentisiert werden	R3a	– Eine IKE_AUTH-Response wird an den Initiator gesendet – Ein neues Objekt vom Typ `ike_sa_established_t` wird erstellt und als neuer Zustand der IKE_SA gesetzt – Das aktuelle Zustandsobjekt wird gelöscht
– Die Message ist kein Request vom Typ IKE_AUTH oder – Die Message konnte nicht erfolgreich entschlüsselt, geparst oder verifiziert werden	R3b	– Womöglich wird eine IKE_SA_INIT-Response an den Initiator gesendet – Das Zustandsobjekt wird nicht geändert
– Ein Timer ist abgelaufen, der die maximale Dauer einer halb offenen IKE_SA regelt oder – Die Authentisierung der Identität des Initiators konnte nicht verifiziert werden	R3c	– Die IKE_SA und somit auch deren aktueller Zustand wird zerstört

Tabelle 18: Zustandsübergange aus der Klasse ike_sa_init_responded_t

5.5.4.3 initiator_init_t

Der erste Zustand eines `ike_sa_t`-Objekts, welches zum Aufbau einer neuen IKE_SA erstellt wurde, ist vom Typ INITIATOR_INIT und in der Klasse `initiator_init_t` implementiert. In diesem Zustand wird keine hereinkommende Message bearbeitet und die `process_message()`-Funktion returniert mit einem Fehlerstatus. Stattdessen kann die Funktion `initiate_connection()` aufgerufen werden, welche den ersten IKE_SA_INIT-Request erstellt und versendet.

Aus diesem Zustand gibt es folgende Zustandsübergänge:

Bedingungen	Übergang	Auswirkung
– Die Message kann als Request vom Typ IKE_SA_INIT erstellt werden – Für die angegebene Verbindungs-Identifikation existiert eine Konfiguration	I2a	– Ein IKE_SA_INIT-Request wird an den Initiator gesendet – Ein neues Objekt vom Typ `ike_sa_init_requested_t` wird erstellt und als neuer Zustand der IKE_SA gesetzt – Das aktuelle Zustandsobjekt wird gelöscht
– Für die angegebene Verbindungs-Identifikation existiert keine Konfiguration	I2b	– Das Zustandsobjekt wird nicht geändert, jedoch returniert die Funktion `initialize_connection()` mit einem Fehlerstatus, woraufhin die entsprechende `ike_sa_t` vom zugeordneten Worker-Thread gelöscht wird

Tabelle 19: Zustandsübergange aus der Klasse initiator_init_t

5.5.4.4 ike_sa_init_requested_t

Die `process_message()`-Implementierung dieser Zustandsklasse erwartet eine Response vom Typ IKE_SA_INIT.

Aus diesem Zustand gibt es folgende Zustandsübergänge:

Bedingungen	Übergang	Auswirkung
– Die Message ist eine Response vom Typ IKE_SA_INIT – Die bearbeitete Message konnte erfolgreich geparst und deren Aufbau verifiziert werden – Der Responder hat mindestens ein IKE_SA-Proposal ausgewählt und zurückgesendet	I3a	– Ein IKE_AUTH-Request wird an den Initiator gesendet – Ein neues Objekt vom Typ `ike_auth_requested_t` wird erstellt und als neuer Zustand der IKE_SA gesetzt – Das aktuelle Zustandsobjekt wird gelöscht
– Die Message ist keine Response vom Typ IKE_SA_INIT oder – Die Message konnte nicht erfolgreich geparst oder deren Aufbau verifiziert werden	I3b	– Das Zustandsobjekt wird nicht geändert
– Der Responder hat ein IKE_SA-Proposal ausgewählt, die nicht die Diffie-Hellman-Gruppe beinhaltet, welche im IKE_SA_INIT-Request gesendet wurde	I3c	– Ein neues Objekt vom Typ `initiator_init_t` wird erstellt und als neuer Zustand der IKE_SA gesetzt – Das aktuelle Zustandsobjekt wird gelöscht – das neue Zustandsobjekt wird beauftragt, den IKE_SA_INIT-Request mit anderer Diffie-Hellman-Gruppe zu senden
– Ein Timer ist abgelaufen, der die maximale Dauer einer halb offenen IKE_SA regelt oder – Der Responder hat einen unbekannten Fehler gemeldet	I3d	– Die IKE_SA und somit auch deren aktueller Zustand wird zerstört

Tabelle 20: Zustandsübergange aus der Klasse ike_sa_init_request_t

5.5.4.5 ike_auth_requested_t

Die `process_message()`-Implementierung dieser Zustandsklasse erwartet eine Response vom Typ IKE_AUTH.

Aus diesem Zustand gibt es folgende Zustandsübergänge:

Bedingungen	Übergang	Auswirkung
– Die Message ist eine Response vom Typ IKE_AUTH – Die bearbeitete Message konnte erfolgreich entschlüsselt, geparst und deren Aufbau verifiziert werden – Die Identität des Responders konnte verifiziert werden	I4a	– Ein neues Objekt vom Typ `ike_sa_established_t` wird erstellt und als neuer Zustand der IKE_SA gesetzt – Das aktuelle Zustandsobjekt wird gelöscht
– Die Message ist keine Response vom Typ IKE_AUTH oder – Die Message konnte nicht erfolgreich entschlüsselt, geparst oder deren Aufbau verifiziert werden	I4b	– Das Zustandsobjekt wird nicht geändert
– Ein Timer ist abgelaufen, der die maximale Dauer einer halb offenen IKE_SA regelt oder – Die Authentisierung der Identität des Responders konnte nicht verifiziert werden oder – Der Responder hat einen unbekannten Fehler gemeldet	I4d	– Die IKE_SA und somit auch deren aktueller Zustand wird zerstört

Tabelle 21: Zustandsübergange aus der Klasse ike_auth_request_t

5.5.4.6 ike_sa_established_t

Die `process_message()`-Implementierung dieser Zustandsklasse erwartet einen Request vom Typ CREATE_CHILD_SA oder INFORMATIONAL. Dieser Zustand stellt eine vollständig aufgebaute IKE_SA dar, bei welcher sich Initiator und Responder gegenseitig authentisiert haben.

Diesem Zustand folgen keine weiteren mehr. Er kann höchstens verlassen werden, wenn die IKE_SA gelöscht wird.

Momentan beherrscht die Klasse lediglich die Verarbeitung von INFORMATIONAL-Meldungen, welche die Löschung einer IKE_SA indizieren. Wird eine solche authentisierte Meldung von `process_message()` verarbeitet, wird die IKE_SA gelöscht. Alle anderen INFORMATIONAL-Requests werden mit einer leeren INFORMATIONAL-Response beantwortet.

5.5.5 utils

Im Package `utils` sind Hilfsklassen untergebracht. Sie erbringen allgemeine Funktionen und werden von diversen anderen Klassen eingesetzt.

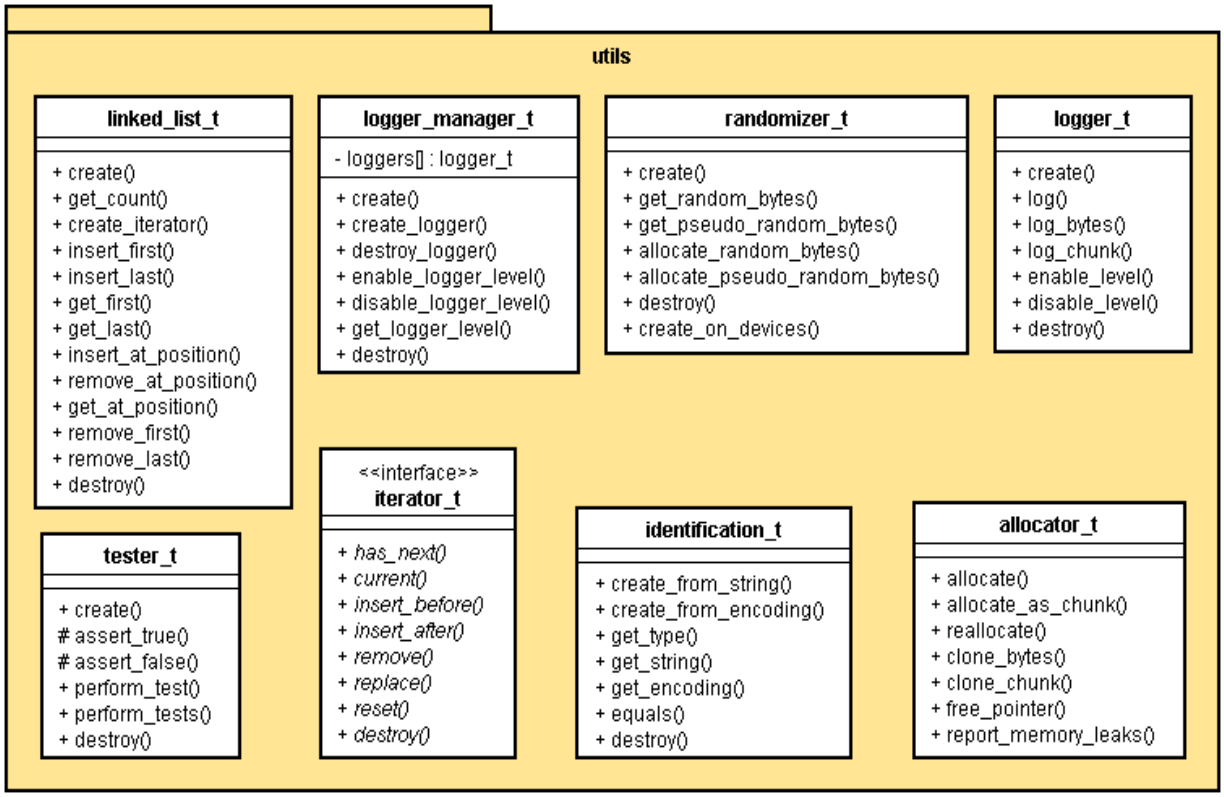


Abbildung 30: Klassen des Packages "utils"

5.5.5.1 linked_list_t

Die Klasse `linked_list_t` implementiert eine doppelt verkettete Liste, in der Objekte abgespeichert werden können. Diese Klasse bietet folgende Funktionalitäten:

- Listen-Einträge können am Anfang und am Ende eingefügt oder gelöscht werden
- Der Wert des ersten oder letzten Eintrages in der Liste kann abgefragt werden
- Die Grösse, sprich die Anzahl Einträge der Liste können abgefragt werden
- Einträge können an irgend einer Position eingefügt und herausgenommen werden
- Ein Iterator kann erstellt werden, der das `iterator_t`-Interface implementiert. Mit diesem kann durch die Liste iteriert und diese auch manipuliert werden

Die ganze Klasse ist nicht thread-save implementiert und muss gegebenenfalls über einen Wrapper thread-save gemacht werden. Dies wird beispielsweise bei den diversen Queues realisiert.

5.5.5.2 iterator_t

Das Interface `iterator_t` bietet die Möglichkeit über eine Liste zu iterieren. Implementiert wird das `iterator_t`-Interface zurzeit nur von der Klasse `linked_list_t`. Ein `iterator_t`-Objekt erlaubt folgende Methoden:

- Navigation zum nächsten Element in der Liste
- Abfrage des aktuellen Elementes in der Liste
- Einfügen eines neuen Elementes vor oder nach dem aktuellen Element
- Löschen des aktuellen Elementes
- Ersetzen des aktuellen Elementes

Das `iterator_t`-Interface erlaubt es, unabhängig der effektiven Implementierung, Listen zu verarbeiten. Es stellt zudem eine sehr universelle und einfach zu verstehende Möglichkeit zur Verfügung, in einer Liste zu suchen.

5.5.5.3 identification_t

Die Klasse `identification_t` abstrahiert die Identität eines Kommunikationsteilnehmers. Dabei werden die gebräuchlichsten Typen unterstützt. Dazu gehören:

Typ	Beschreibung
`ID_IPV4_ADDR`	Identifikation eines Rechners über seine IPv4 Adresse
`ID_FQDN`	Identifikation eines Rechners über den vollen Domainnamen
`ID_RFC822_ADDR`	Identität einer Person, beschrieben über seine E-Mail-Adresse
`ID_IPV6_ADDR`	Eine IPv6 Adresse
`ID_DER_ASN1_DN`	Ein ASN.1 kodierter Distinguished Name
`ID_DER_ASN1_GN`	Ein ASN.1 kodierter General Name
`ID_KEY_ID`	Spezielles Token, welche für die Verwendung von proprietären Kodierungen von Identifikationen verwendet werden kann

Tabelle 22: Typen, welche die Klasse identification_t beinhaltet

Die jetzige Implementierung unterstützt lediglich die Konvertierung des Typs `ID_IPV4_ADDR`. Für die anderen Typen sind sowohl der Konstruktor via String als auch das Zurückkonvertieren in einen String noch nicht möglich.

5.5.5.4 randomizer_t

Mit Hilfe der Klasse `randomizer_t` können zufällige Bytemuster generiert werden. Es stehen Funktionen zur Generierung von richtigen Zufallszahlen, als auch solche für Pseudo-Zufallszahlen zur Verfügung. Die `get()` Funktionen generieren die Bytes in bereits allozierten Speicher, die `allocate()` Funktionen allozieren den Speicher für den Aufrufer.

Die jetzige Implementierung verwendet die Quellen `/dev/random` bzw. `/dev/urandom` für die Herausgabe von Zufallszahlen.

5.5.5.5 logger_manager_t

Über eine Instanz der Klasse `logger_manager_t` können alle Logger verwaltet und gesteuert werden. Dazu führt der Logger-Manager intern eine Liste aller Logger.

Logger werden einem Kontext zugeordnet, konkret handelt es sich dabei um Logger für eine spezielle Klasse oder eine spezielle Aufgabe. Über diesen Kontext können alle dazugehörenden Logger zur Laufzeit gesteuert werden. Möchte man z.B. ein detailliertes Logging für das Parsen von Daten, so kann man den Loglevel für den Kontext `PARSER` erhöhen. Ist man hingegen nicht an den kryptographischen Einzelheiten des Session-Aufbaus interessiert, kann man den Loglevel für den Kontext `IKE_SA` reduzieren. Da im Logger-Manager alle Logger für diesen Kontext registriert sind, kann er den Loglevel jedes betroffenen Loggers anpassen.

Im Quellcode wird ein solcher Kontext `logger_context_t` genannt. Die Beschreibung der Loglevels ist in den Erläuterung zur Klasse `logger_t` zu entnehmen.

Folgende Kontexte sind derzeit definiert:

Kontext	Prefix im Log-Output	Logging für Klassen
PARSER	[PARSER]	parser_t
GENERATOR	[GENRAT]	generator_t
IKE_SA	[IKE_SA]	ike_sa_t, sowie alle state_t-Implementierungen
IKE_SA_MANAGER	[ISAMGR]	ike_sa_manager_t
MESSAGE	[MESSAG]	message_t
ENCRYPTION_PAYLOAD	[ENCPLD]	encryption_payload_t
THREAD_POOL	[THPOOL]	thread_pool_t, solange es kein Log eines Worker-Threads ist
WORKER	[WORKER]	thread_pool_t, wenn der Log von einem Worker-Thread stammt
SCHEDULER	[SCHEDU]	scheduler_t
SENDER	[SENDER]	sender_t
RECEIVER	[RECEVR]	receiver_t
SOCKET	[SOCKET]	socket_t
TESTER	[TESTER]	tester_t, sowie alle Testfunktionen
DAEMON	[DAEMON]	daemon_t
CONFIGURATION_MANAGER	[CONFIG]	configuration_manager_t

Tabelle 23: Vorhandene Logger-Kontexte

Von den Payloads besitzt lediglich `encryption_payload_t` einen Logger, da dieser der komplexeste Payload ist. Weitere Logger wären aber denkbar.

5.5.5.6 logger_t

Die `logger_t`-Klasse ermöglicht das detaillierte Logging von Ereignissen und Daten. Sie kann zur Auswertung von Geschehnissen, aber auch für das Debugging verwendet werden.

Normalerweise erfolgt die Erstellung und Zerstörung von `logger_t`-Objekten über den Logger-Manager, damit dieser die Logger verwalten kann. Ein `logger_t`-Objekt kann allerdings auch unabhängig vom Logger-Manager erstellt werden. Meldungen können über die `log`-Funktion ähnlich wie mit `printf` und Konsorten ausgegeben werden. Zudem sind Funktionen für die Ausgabe von Hex-Dumps definiert.

Jedem Logger ist ein Loglevel zugeordnet. Beim Aufruf der Logging-Funktionen wird der Log nur dann getätigt, wenn der angegebene Loglevel demjenigen des Logger entspricht. Somit kann die Ausgabeflut kontrolliert werden. Dabei sind die Loglevels in zwei Kategorien unterteilt. Die eine beschreibt die Art des Loggings, die andere den Detaillierungsgrad.

Bei der Verknüpfung der Loglevels der beiden Kategorien kann folgende Ausgabe erreicht werden:

	LEVEL0 [0]	**LEVEL1 [1]**	**LEVEL2 [2]**	**LEVEL3 [3]**
CONTROL [~]	Gröbster Kontrollfluss	Erweiterter Kontrollfluss	Detaillierterer Kontrollfluss	Ganzer Kontrollfluss
ERROR [!]	Systemfehler	Wichtige Fehler	Nebensächliche Fehler	Alle Fehler
RAW [#]	Wichtigste Datendumps	Weniger wichtige Datendumps	Unwichtigere Datendumps	Alle Datendumps
PRIVATE [?]	Wichtigste Datendumps mit sensitiven Daten	Weniger wichtige Datendumps mit sensitiven Daten	Unwichtigere Datendumps mit sensitiven Daten	Alle Datendumps mit sensitiven Daten
AUDIT [>]	Spezielle Logs, die für den eigentlichen Betrieb des Daemons interessant sind. Dazu gehören Verbindungsaufbau, -abbruch, sowie weitere wichtige Ereignisse. AUDIT macht momentan keine Unterscheidung der Levels.			

Tabelle 24: Zusammensetzung der Loglevels

Die Loglevels können mit dem Oder-Operator (|) kombiniert werden. Der spezielle Loglevel `FULL` aktiviert alle Kategorien und setzt den Level aufs Maximum.

Die Ausgabe erfolgt auf den Standard Output, kann alternativ aber auch auf den Syslogger oder direkt in eine Datei geschrieben werden. Es werden noch Zusatzinformationen ausgeben, welche den Typ dieses Logs genauer beschreiben. Hier ein Beispiel:

```
Nov 24 14:19:24 [charon] [~2] [ISAMGR] @3084584624 Waiting for all threads
```

Neben Datum, Zeit und Dienst, welche der Syslogger einträgt, wird ebenfalls der Loglevel ausgegeben. Dieser setzt sich zusammen aus der Art des Logs, sowie dem Detaillierungsgrad. Die Bedeutung der Zeichen sind der obigen Tabelle zu entnehmen. Die Nummer hinter dem „@“ bezeichnet den Thread, welcher den Log getätigt hat. Somit sind die Logs der einzelnen Threads auseinander zu halten, wenn diese verschachtelt auftreten.

5.5.5.7 allocator_t

Mit Hilfe der Klasse `allocator_t` können Memory-Leaks aufgespürt werden. Ist der Quellcode mit der Defninition `LEAK_DETECTIVE` kompiliert, so werden alle Speicherallozierungen über den Allocator erledigt und es können allfällige Leaks gemeldet werden.

Für ein Objekt vom Typ `allocator_t` existiert kein Konstruktor. Das Allozieren und Freigeben von Speicher erfolgt über Makros, welche die Aufrufe an die einzige Instanz des Allocators weiterleiten.

5.5.5.8 tester_t

Die Klasse `tester_t` ermöglicht das Testen einzelner Funktionalitäten der Software. Über `perform_test()` kann eine einzelne Testroutine gestartet werden, über `perform_tests()` gleich mehrere. Diese Testroutinen können die `assert_*()`-Funktionen des Testers verwenden, um gewisse Bedingungen zu prüfen. `tester_t` erstellt einen Report aus diesen Tests und gibt diesen aus.

Mit dieser Klasse wird das automatisierte Testen von Klassen ermöglicht. Hier ein Beispiel für den Output:

```
Start testing...


_____________________________________________________________________
Testname                                                | running time
________________________________________________________|____________
Linked List Insert and remove                           |        30 us
=====================================================================
End testing. 1 of 1 tests succeeded
=====================================================================
```

5.5.6 transforms

Das Package `transforms` enthält diverse Algorithmen, die für kryptografische Funktionen verwendet werden.

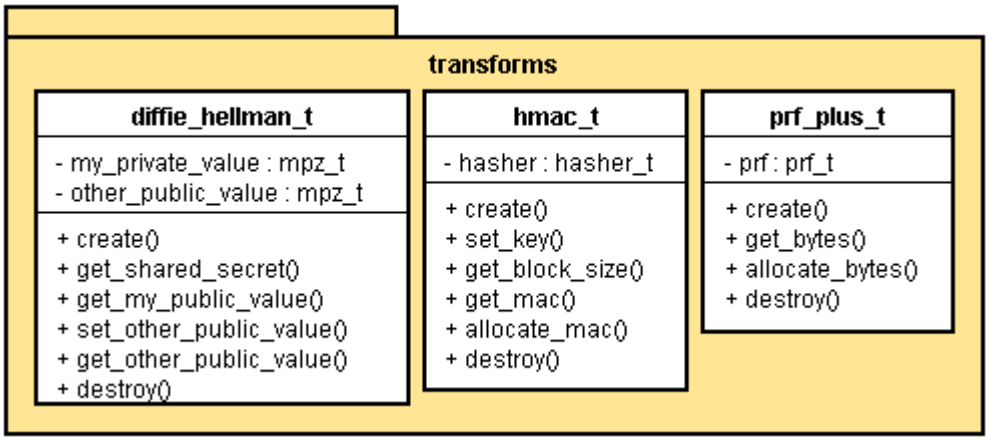


Abbildung 31: Klassen des Packages "transforms"

5.5.6.1 diffie_hellman_t

Die Klasse `diffie_hellman_t` implementiert den Diffie-Hellman-Schlüsselaustausch. Beim Erstellen des Objektes wird automatisch der eigene private Wert generiert. Der eigene öffentliche Wert kann gelesen und dem Kommunikationspartner übermittelt werden. Antwortet dieser, kann sein öffentlicher Wert gesetzt werden. Nun ist es möglich, den gemeinsamen geheimen Schlüssel abzufragen.

Die notwendigen Berechnungen werden mit Hilfe der `gmp`-Bibliothek durchgeführt.

5.5.6.2 hmac_t

`hmac_t` ist die generische Implementierung des HMAC-Algorithmus. Dem Konstruktor muss ein Hash-Algorithmus mitgegeben werden. Daraus wird intern ein Instanz von `hasher_t` erstellt, mit welcher der HMAC-Algorithmus umgesetzt wird.

Einmal instanziert können Blockgrösse (des Hash-Algorithmus) abgefragt sowie MACs generiert werden.

5.5.6.3 prf_plus_t

Die Klasse `prf_plus_t` implementiert den Algorithmus `prf+`, welcher im Draft von `IKEv2` beschrieben wird. Dem Konstruktor wird direkt ein `prf_t`-Objekt mitgegeben, mit welchem die `prf+`-Funktion realisiert werden soll.

5.5.7 prfs

Das im `transforms` enthaltene Sub-Package `prfs` enthält Pseudo-Random-Funktionen. Dazu gehört das Interface `prf_t` sowie deren Implementierungen.

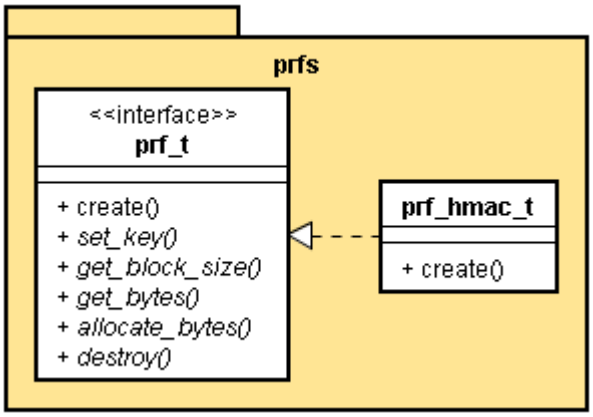


Abbildung 32: Klassen des Packages "prfs"

Momentan wird das `prf_t`-Interface lediglich durch die Klasse `prf_hmac_t` implementiert. Eine Unterstützung weiterer PRFs, wie beispielsweise über eine AES-Verschlüsselungen, ist denkbar.

5.5.7.1 prf_t

Das Interface `prf_t` definiert die Schnittstelle, welche eine PRF-Implementierung erfüllen muss. Dazu gehört eine Methode zum Setzen des Schlüssels, sowie die Abfrage der Blockgrösse der PRF-Implementierung. Pseudo-Random-Bytes können über die beiden weiteren Methoden bezogen werden. Die Anzahl Bytes entspricht dabei immer der Blockgrösse.

Zusatzlich zum Interface ist ein Konstruktor definiert, welcher je nach angegebenem Algorithmus die richtige PRF-Implementierung instanziert.

5.5.7.2 prf_hmac_t

`prf_hmac_t` ist eine `prf_t`-Implementierung, welche auf dem HMAC-Algorithmus aufbaut. Dazu verwendet es intern eine Instanz eines `hmac_t`. `prf_hmac_t` unterstützt alle Algorithmen, für welche eine `hasher_t` und somit ein `hmac_t` definiert ist.

5.5.8 hashers

Das im `transforms` enthaltene Sub-Package `hashers` beinhaltet Hash-Algorithmen. Das Interface `hasher_t` definiert dabei deren Schnittstelle.

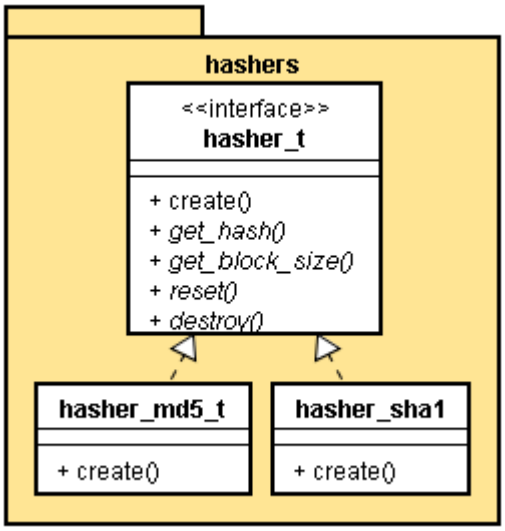


Abbildung 33: Klassen des Packages "hashers"

5.5.8.1 hasher_t

Das Interface `hasher_t` definiert Methoden, welche eine Implementierung eines Hash-Algorithmus bieten soll. Dazu gehört die Hash-Funktion selber, sowie das Abfragen der Blockgrösse und das zurücksetzen des Hashers.

Zusätzlich zum Interface ist ein generischer Konstruktor definiert. Mit diesem kann eine Instanz erstellt werden, welche den mitgegebenen Algorithmus implementiert.

5.5.8.2 hasher_md5_t

Die Klasse `hasher_md5_t` implementiert das `hash_t`-Interface, wobei MD5 als Hash-Funktion zum Einsatz kommt. Die Implementierung wurde von `strongSwan` übernommen.

5.5.8.3 hasher_sha1_t

Bei dieser Klasse wird das `hasher_t`-Interface mit dem SHA1-Algorithmus implementiert. Die Implementierung wurde ebenfalls von `strongSwan` übernommen.

5.5.9 crypters

Das im `transforms` enthaltene Sub-Package `crypters` beinhaltet symmetrische Verschlüsselungsalgorithmen. Das Interface `crypter_t` definiert dabei deren Schnittstelle.

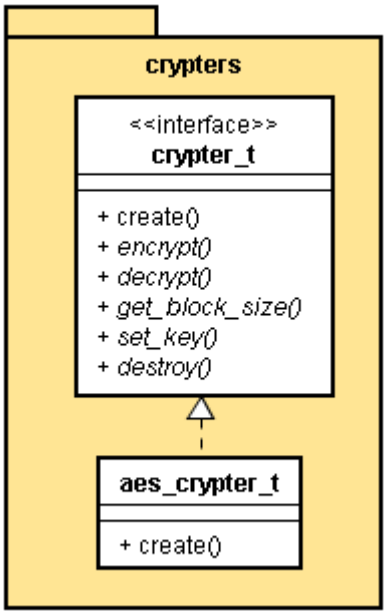


Abbildung 34: Klassen des Packages "crypters"

5.5.9.1 crypter_t

Das Interface `crypter_t` definiert Methoden, welche eine Implementierung eines symmetrischen Verschlüsselungs-Algorithmus zur Verfügung stellen muss. Dazu gehören Funktionen zum Verschlüsseln, Entschlüsseln, setzten des Schlüssels usw.

Zusätzlich zum Interface ist ein generischer Konstruktor definiert. Mit diesem kann eine Instanz erstellt werden, welche den mitgegebenen Algorithmus implementiert.

Zu verschlüsselnde und entschlüsselnde Daten müssen eine Länge haben, die einem vielfachen der Blockgrösse entspricht.

5.5.9.2 aes_crypter_t

Die Klasse `aes_crypter_t` implementiert den symmetrischen Verschlüsselungsalgorithmus AES-CBC und unterstützt dabei Schlüsselgrössen von 128, 160 und 256 Bit. Die Implementierung wurde von `strongSwan` übernommen.

5.5.10 signers

Das im `transforms` enthaltene Sub-Package `signers` beinhaltet Algorithmen zur Erstellung und Verifikation von Signaturen. Das Interface `signer_t` definiert dabei deren Schnittstelle.

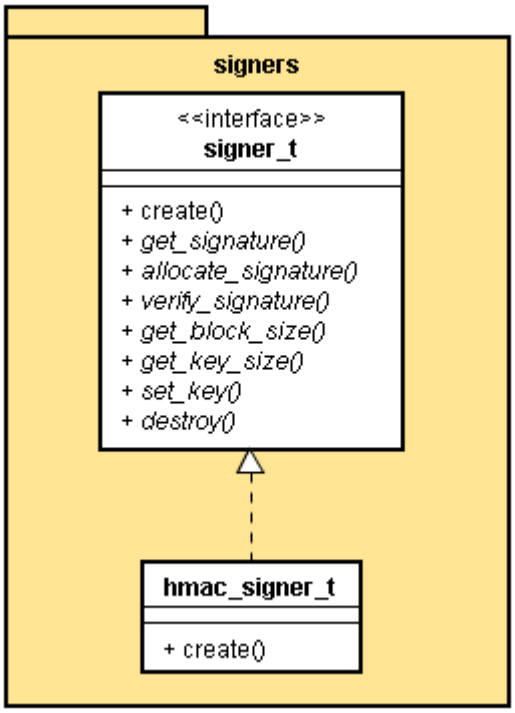


Abbildung 35: Klassen des Packages "signers"

5.5.10.1 signer_t

Das Interface `signer_t` definiert Methoden, die ein Algorithmus zum Erstellen und Prüfen von Signaturen zur Verfügung stellen muss.

Zum Interface `signer_t` gehört ebenfalls ein generischer Konstruktor, welcher das Erstellen einer Instanz von angegebenem Typ erlaubt.

5.5.10.2 hmac_signer_t

Die Klasse `hmac_signer_t` ermöglicht die Verwendung des HMAC-Algorithmus zum Erstellen und Verifizieren von Signaturen. Sie kann dabei auf alle Hash-Algorithmen zurückgreifen, welche das `hasher_t`-Interface implementieren.

5.5.11 rsa

Das im `transforms` enthaltene Sub-Package `rsa` beinhaltet Klassen, die den asymmetrischen Verschlüsselungsalgorithmus RSA umsetzen. Die Funktionen zum Laden und Speichern von Schlüsseln sind noch nicht implementiert, da die benötigten Funktionalitäten dazu noch nicht vorhanden sind (ASN.1). Die `get_key()`-/`set_key()`-Funktionen sind über ein vereinfachtes Format umgesetzt, welches die Verwendung der Klasse für Testzwecke ermöglicht.

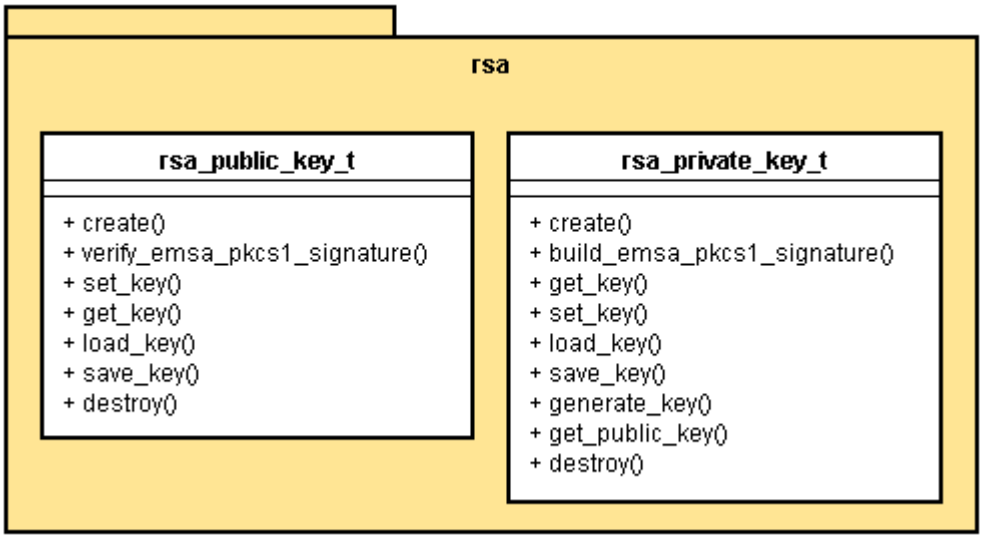


Abbildung 36: Klassen des Packages "rsa"

Es ist anzunehmen, dass beim Einbau der ASN.1-DER-Unterstützung die Funktionen für das Laden und Speichern der Schlüssel angepasst werden.

5.5.11.1 rsa_public_key_t

Die Klasse `rsa_public_key_t` abstrahiert einen öffentlichen RSA-Schlüssel. Sie beinhaltet unter anderem eine Funktion zur Verifikation einer EMSA PKCS#1-Signatur. Dabei handelt es sich um eine spezielle Kodierung, wie sie in `IKEv2` verwendet wird. Um die Signatur überprüfen zu können, muss der, bei der Signierung verwendete, Hash-Algorithmus über das `hasher_t`-Interface implementiert sein.

5.5.11.2 rsa_private_key_t

Die Klasse `rsa_private_key_t` abstrahiert einen private RSA-Schlüssel. Sie beinhaltet unter anderem eine Funktion zum Erstellen einer EMSA PKCS#1-Signatur. Der dabei verwendete Hash-Algorithmus muss wiederum via `hasher_t`-Interface implementiert sein.

Aus dem privaten Schlüssel kann der öffentliche Schlüssel vom Typ `rsa_public_key_t` erstellt werden.

5.5.12 queues

Im Package `queues` werden alle eingesetzten Queues zusammengefasst. Diese bauen intern auf einer Linked-List vom Typ `linked_list_t` auf, sind aber thread-save. Dazu verwenden alle Queues ein Mutex und eine Conditional-Variable.

Der Mutex wird dazu verwendet, dass die interne Linked-List nur von einem Thread gleichzeitig bearbeitet oder durchsucht werden kann. Die Conditional-Variable wird dazu benutzt, einen wartenden Thread aufzuwecken. Dies ist dann nützlich, wenn der entsprechende Thread aufgrund einer leeren Queue schlafen gegangen ist und nun geweckt werden soll, weil beispielsweise ein neuer Eintrag in die Queue eingefügt wurde.

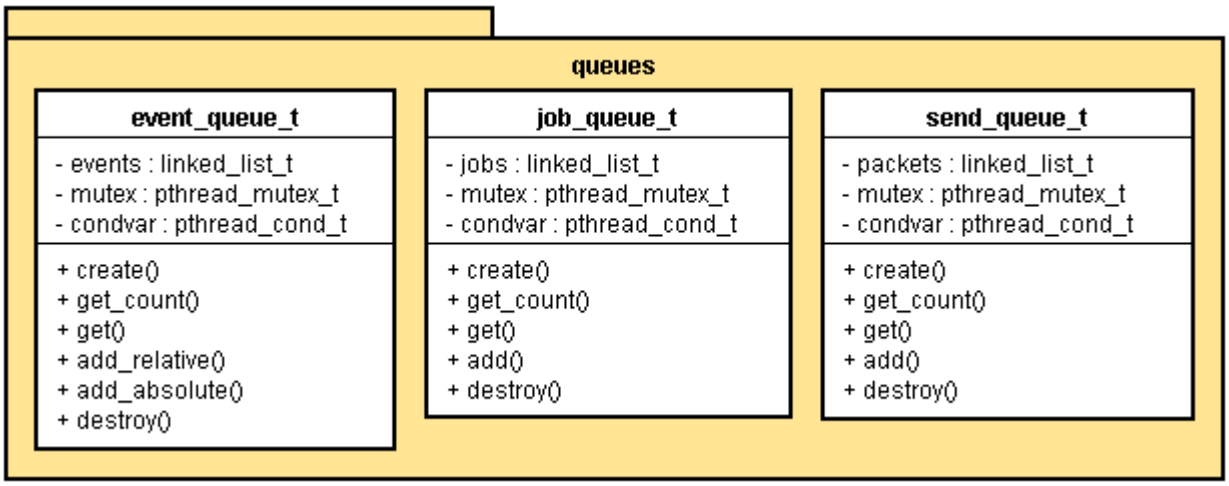


Abbildung 37: Klassen des Packages "queues"

5.5.12.1 event_queue_t

Die Klasse `event_queue_t` speichert in der internen Linked-List eine Menge von Jobs, denen jeweils eine bestimmte Abarbeitungszeit zugeordnet ist. Zum hinzufügen neuer Jobs stehen zwei Methoden zur Verfügung. Die eine nimmt eine relative, die andere eine absolute Zeitangabe als Ausführungszeitpunkt entgegen. Die Jobs sind intern nach deren Ausführungszeitpunkt sortiert, d.h. der nächste auszuführende Job ist immer am Anfang der Liste. Die Methode zum entnehmen des nächsten Jobs ist blockierend, d.h. sie blockiert solange, bis der Zeitpunkt für die Abarbeitung des frühesten Jobs erreicht ist.

Die Event-Queue ist thread-save. Es können beliebig viele Threads zur gleichen Zeit Jobs einfügen und theoretisch auch beziehen. Da der Event-Thread die herausgenommenen Threads nicht selbst arbeitet, sondern diese in die Job-Queue einfügt, bezieht in der umgesetzten Architektur nur dieser die Jobs.

5.5.12.2 job_queue_t

Mit der Klasse `job_queue_t` können Aufgaben, so genannte Jobs (siehe 5.5.13), verwaltet werden. Die eingefügten Jobs werden von den Worker-Threads entnommen und abgearbeitet. Dies geschieht nach dem FIFO-Prinzip, zuerst eingefügte Jobs werden auch wieder zuerst entnommen. Das Einfügen und Entnehmen geht gegenüber der Event-Queue viel effizienter vonstatten, da die Jobs nicht spezifisch eingeordnet werden müssen.

Auch die Job-Queue ist thread-save implementiert. Es können von beliebigen Threads Jobs eingefügt und wieder bezogen werden.

In dieser Implementierung werden Jobs von unterschiedlichen Threads in die Job-Queue eingefügt. Genauer sind dies die Worker-Threads, der Receiver-Thread und der Event-Thread. Später kommt wahrscheinlich noch der Thread der Kernel-Schnittstelle und einer für die Steuerung des Daemons hinzu (`whack`).

5.5.12.3 send_queue_t

Die Send-Queue, in der Klasse `send_queue_t` implementiert, stellt eine Warteschlange für zu versendende UDP-Pakete dar. Es können beliebig Pakete in der Form eines `packet_t`-Objekts in die Queue eingefügt werden. Ein Sender-Thread entnimmt nacheinander die `packet_t`-Objekt aus der Queue und sendet dieses an den entsprechenden Empfänger. Auch hier erfolgt die Abarbeitung strikt nach dem FIFO-Prinzip.

Die Send-Queue ist vor gleichzeitigem Zugriff durch mehrere Threads geschützt.

5.5.13 jobs

Das im Package `queues` enthaltenen Sub-Package `jobs` enthält Jobs, die sowohl für die Event-Queue als auch für die Job-Queue verwendet werden. Alle Job-Klassen implementieren das Interface `job_t`.

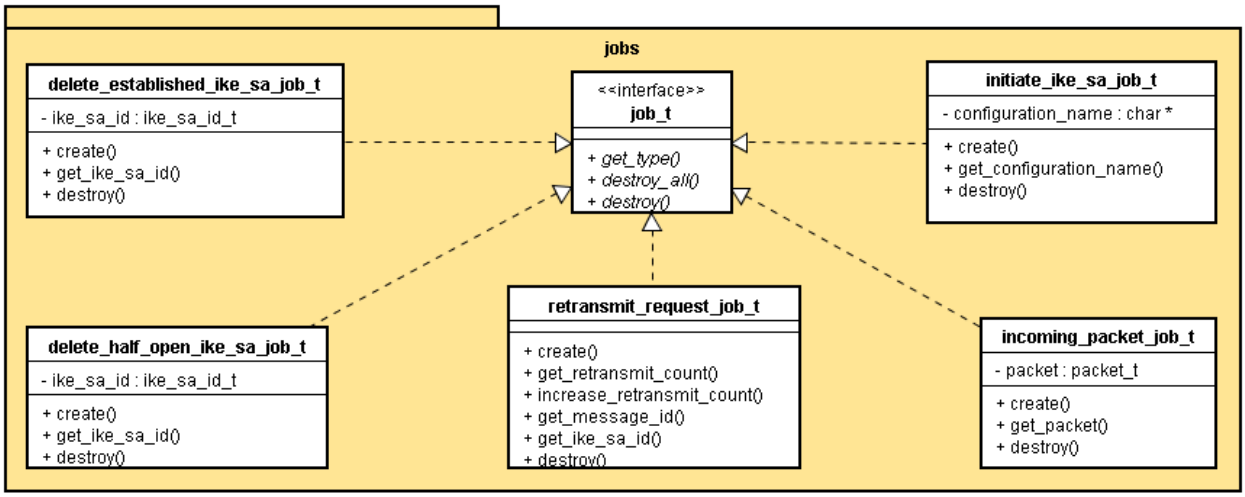


Abbildung 38: Klassen des Packages "jobs"

5.5.13.1 job_t

Das `job_t`-Interface gibt die Methoden vor, die jede Instanz eines Jobs anbieten muss. Neben einer Methode zur Bestimmung des Jobtyps sind zwei Destruktoren notwendig. Die normale Methode `destroy()` zerstört den Job, ohne die angehängten Daten freizugeben, die Methode `destroy_all()` zerstört auch die enthaltenen Daten.

Anhand des zurückgelieferten Job-Typs der Funktion `get_type()` kann der Aufrufer herausfinden um was für einen Job es sich handelt und danach das `job_t`-Objekt entsprechend casten.

5.5.13.2 delete_half_open_ike_sa_job_t

Ein Objekt vom Typ `delete_half_open_ike_sa_job_t` repräsentiert einen Auftrag zum Löschen einer IKE_SA, die noch nicht den Zustand IKE_SA_ESTABLISHED erreicht hat. Zur Identifikation der IKE_SA beinhaltet dieser Job-Typ deren Identifikation in Form eines `ike_sa_id_t`-Objektes. Der Sinn dieses Job-Typs ist es, IKE_SAs zu löschen, deren Initialisierung gestartet, jedoch nicht beendet wurde. Unterschiedliche Gründe können zu solch einer „halb offenen“ IKE_SA führen:

- Der Kommunikationspartner ist während dem IKE_SA_INIT- oder IKE_AUTH-Austausch abgestürzt.
- Die `IKEv2`-Pakete gehen aufgrund von Netzwerkproblemen verloren.
- Ein Angreifer hat eine „Denial of Service“-Attacke gestartet und sendet dabei nur IKE_SA_INIT-Anfragen.
- usw.

In dieser Implementierung ist der Worker-Thread dafür verantwortlich, dass nach dem Erstellen einer neuen IKE_SA automatisch ein Job von diesem Typ erstellt wird.

5.5.13.3 delete_established_ike_sa_job_t

Ein Objekt vom Typ `delete_established_ike_sa_job_t` repräsentiert einen Auftrag zum Löschen einer initialisierten IKE_SA. Zur Identifikation der IKE_SA beinhaltet dieser Job-Typ deren Identifikation in Form eines `ike_sa_id_t`-Objektes. Der Sinn dieses Job-Typs ist es, initialisierte IKE_SAs nach einem gewissen Zeitpunkt zu löschen, da ein Rekeying noch nicht implementiert ist.

5.5.13.4 incoming_packet_job_t

Die Klasse `incoming_packet_job_t` spezifiziert einen Job, welcher ein eingehendes Paket darstellt. Der Receiver-Thread erstellt aus jedem empfangenen UDP-Paket ein `packet_t`-Objekt, welches er in ein `incoming_packet_job_t`-Objekt einfügt. Diesen Job reiht er anschliessend in die Job-Queue ein, damit er vom nächsten freien Worker-Thread abgearbeitet wird.

5.5.13.5 initiate_ike_sa_job_t

Ein Objekt vom Typ `initiate_ike_sa_job_t` repräsentiert ein Job zur Initiierung einer neuen IKE_SA. Der Job speichert dabei den Namen der Konfiguration, die zum Aufbauen der entsprechenden IKE_SA verwendet werden soll.

5.5.13.6 retransmit_request_job_t

Ein Objekt vom Typ `retransmit_request_job_t` repräsentiert ein Job zum erneuten Senden eines Requests. Es speichert zum einen die Identifikation der IKE_SA und zum anderen die Message ID des entsprechenden Requests.

Für jeden gesendeten Request wird ein solcher Job erstellt. Dies ist notwendig, da die `IKEv2`-Nachrichten per UDP und damit über ein unzuverlässiges Protokoll versendet werden.

Der Worker-Thread, der einen solchen Job abarbeitet, beauftragt das entsprechende `ike_sa_t`-Objekt mit dem erneuten Senden des Requests. Dieses kann feststellen, ob sie bereits die Antwort auf den Request erhalten hat und somit ein Retransmit überflüssig ist.

5.5.14 encoding

Im Package `encoding` sind Klassen abgelegt, die für die (De-)Kodierung von Daten erforderlich sind. Dazu gehört ein Parser, ein Generator, sowie eine Klasse für die Repräsentation einer Meldung.

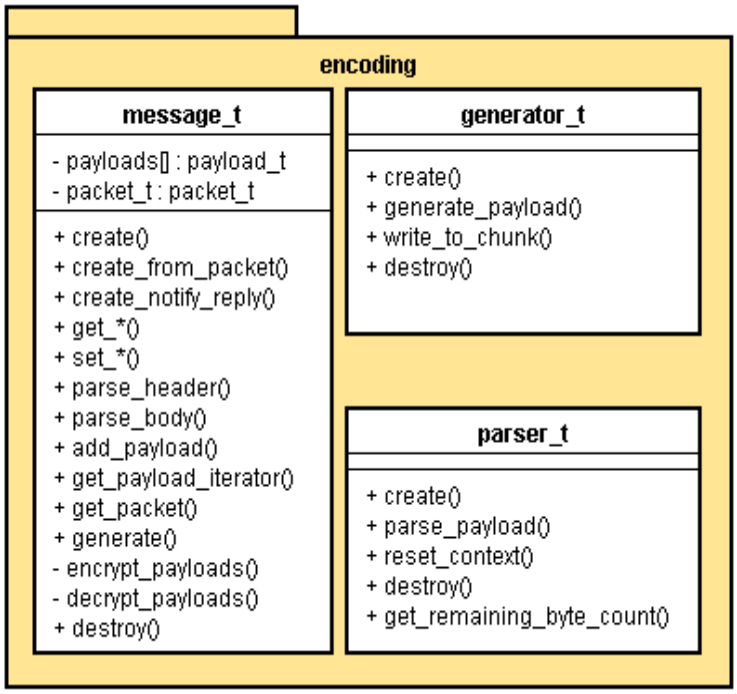


Abbildung 39: Klassen des Packages "encoding"

5.5.14.1 message_t

Ein `message_t`-Objekt stellt eine einzelne Meldung dar, wie sie zwischen `IKE`-Kommunikationspartner ausgetauscht wird. Sie kann die Daten einer Meldung auf zwei Arten speichern. Zum einen im rohen Format, wie die Nachricht über das Netzwerk ausgetauscht wird, zum anderen in einem geparsten Format, welches zur direkten Manipulation der Daten verwendet werden kann. Für die Konvertierung der beiden Formate werden die Klassen `generator_t` bzw. `parser_t` verwendet.

Wird eine Nachricht vom Netzwerk eingelesen, erstellt der Worker-Thread über den Konstruktor `create_from_packet()` ein `message_t`-Objekt. In diesem Fall liegen die Daten ungeparst im Objekt. Nach einem Aufruf von `parse_header()` kann die Meldung über diverse `get`-Methoden analysiert werden. Ist sie interessant, können alle enthaltenen Payloads mit `parse_body()` in ein lesbares Format gewandelt werden. Nun kann ein Iterator bezogen werden. Dies ermöglicht die Navigation über die Payloads sowie deren Analyse. Der genauere Ablauf für das Parsen einer Message ist unter 5.6.3 beschrieben.

Im umgekehrten Fall, wenn also eine Meldung erstellt wird, kann ein `message_t`-Objekt über den normalen Konstruktor instanziert werden. Über diverse `set`-Methoden können die Parameter der Meldung gesetzt werden. Danach werden die Payloads an die Meldung angehängt. Ein abschliessender Aufruf von `generate()` generiert die Meldung mit allen Payloads. Der genauere Ablauf für das generieren einer Message ist unter 5.6.1 beschrieben.

5.5.14.2 generator_t

Die Klasse `generator_t` bietet die Funktionalitäten zum Generieren von Daten, welche über das Netzwerk übermittelt werden sollen. Zum Generieren eines Blocks von Daten wird ein `generator_t`-Objekt instanziert. Dem Generator werden nun nacheinander Payload-Objekte übergeben, die das Interface `payload_t` implementieren. Die Funktionen des `payload_t`-Interfaces erlauben dem Generator die Regeln für die Generierung des entsprechenden Payloads abzurufen. Sind alle Payloads generiert, können die resultierenden Daten mit `write_to_chunk()` bezogen werden.

Wie die für das Generieren von Payloads notwendigen Regeln aufgebaut sind, ist unter 5.5.15.1 beschrieben.

5.5.14.3 parser_t

Der Parser, implementiert in der Klasse `parser_t`, übernimmt die gegenteilige Aufgabe des Generators. Der Konstruktor nimmt einen Block von Daten entgegen, welcher geparst werden soll. Nun können via `parse_payload()` die einzelnen Payloads geparst werden, die dann als `payload_t`-Objekt zurückgegeben werden. Mittels `reset_context()` kann der Parser zurückgesetzt werden, um mit dem Parsen von vorne zu beginnen.

Wie die für das Parsen von Payloads notwendigen Regeln aufgebaut sind, ist unter 5.5.15.1 beschrieben.

5.5.15 payloads

Im Package `encodings` ist das Sub-Package `payloads` enthalten. Es beinhaltet die in `IKEv2` verwendeten Payloads, modelliert als eigene Klassen. Zudem enthält jede dieser Klassen die Regeln, die zum Parsen und Generieren des repräsentierten Payloads notwendig sind.

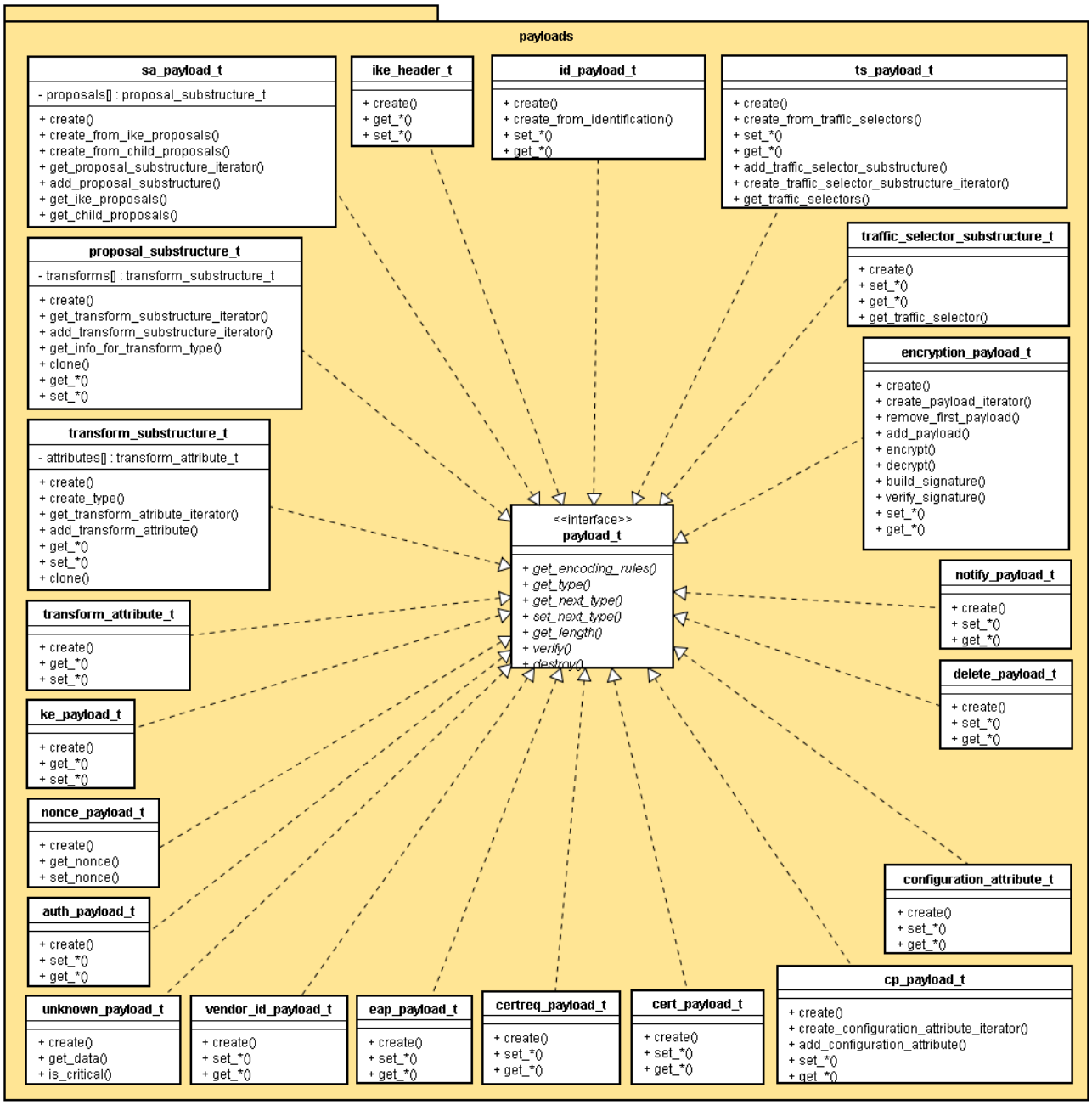


Abbildung 40: Klassen des Packages "payloads"

5.5.15.1 payload_t

Das Interface `payload_t` definiert Methoden, die von jeder Implementierung eines spezifischen Payloads angeboten werden müssen. So muss jeder Payload einen Satz von Kodierungsregeln liefern können, mit welchen der Parser bzw. der Generator diesen Payload spezifisch parsen bzw. kodieren kann.

Eine solche Kodierungsregel beinhaltet ein Typ, welcher die Art der Konvertierung beschreibt. Des weiteren enthält sie einen Offset, der auf das interne Feld in der jeweiligen Payload-Datenstruktur zeigt. So kann der Parser die gelesenen Informationen an diesem Offset speichern bzw. der Generator daraus lesen:

```
typedef struct {
     encoding_t type;
     u_int_32 offset;
} encoding_rule_t;
```

Eine Kodierungsregel beschreibt meist nur die Kodierung eines einzelnen Feldes. Die Regeln eines ganzen Payloads werden in einer Liste zusammengefasst. So sieht diese Liste für den `IKE`-Header folgendermassen aus:

```
encoding_rule_t ike_header_encodings[] = {
     { IKE_SPI,        offsetof(private_ike_header_t, initiator_spi)   },
     { IKE_SPI,        offsetof(private_ike_header_t, responder_spi)   },
     { U_INT_8,        offsetof(private_ike_header_t, next_payload)    },
     { U_INT_4,        offsetof(private_ike_header_t, maj_version)     },
     { U_INT_4,        offsetof(private_ike_header_t, min_version)     },
     { U_INT_8,        offsetof(private_ike_header_t, exchange_type)   },
     { RESERVED_BIT,   0                                               },
     { RESERVED_BIT,   0                                               },
     { FLAG,           offsetof(private_ike_header_t, flags.response)  },
     { FLAG,           offsetof(private_ike_header_t, flags.version)   },
     { FLAG,           offsetof(private_ike_header_t, flags.initiator)},
     { RESERVED_BIT,   0                                               },
     { RESERVED_BIT,   0                                               },
     { RESERVED_BIT,   0                                               },
     { U_INT_32,       offsetof(private_ike_header_t, message_id)      },
     { HEADER_LENGTH,  offsetof(private_ike_header_t, length)          }
};
```

Für das bestimmen des Offsets wird das Makro `offsetof` der C-Bibliothek verwendet.

Parser und Generator müssen gewisse Typen kennen, um sie umsetzen zu können. Zum Beispiel muss für ein Integer eventuell eine Little-/Big-Endian Konvertierung durchgeführt werden.

Folgende Tabelle bietet einen Auszug aus diesen Typen:

Typ	Rohdaten-Länge	C-Typ[1]	Bemerkungen
U_INT_4	4 Bit	u_int8_t	C kennt keine 4 Bit Integer
U_INT_8	1 Byte	u_int8_t	Direkte Übernahme des Wertes
U_INT_16	2 Byte	u_int16_t	Evtl. Little-/Big-Endian Konvertierung
U_INT_32	4 Byte	u_int32_t	Evtl. Little-/Big-Endian Konvertierung
RESERVED_BIT	1 Bit	-	Beim Parsen ignorieren, beim Generieren 0-Bit einfügen
RESERVED_BYTE	1 Byte	-	Beim Parsen ignorieren, beim Generieren 0-Byte einfügen
FLAG	1 Bit	bool	Konversion von 1-Bit-Flag zu gebräuchlichem boolschen Typ
PAYLOAD_LENGTH	2 Byte	u_int16_t	Spezieller Integer, der markiert, dass hier Längeninformationen für den folgenden Teil geschrieben oder gelesen werden müssen.

Tabelle 25: Auszug aus den Typen für die Konvertierung von und zu Rohdaten

Basis-Typen können direkt umgesetzt werden. Für komplexe Typen, zum Beispiel einer Struktur in einem Payload, ruft sich der Parser bzw. der Generator rekursiv auf.

Zudem bietet das Interface `payload_t` Methoden für das Auslesen des Payload-Typs, sowie das Lesen und Setzten des nachfolgenden Payload-Typs.

Die `verify()`-Methode verifiziert die Gültigkeit eines Payloads. In der konkreten Implementierung können Werte und Eigenschaften des Payloads überprüft werden. Mit dieser Funktionalität können falsch aufgebaute Payloads bereits beim Parsen erkannt werden.

5.5.15.2 ike_header_t

Auch die Klasse `ike_header_t` implementiert das Interface `payload_t`. Streng genommen handelt es sich nicht um ein Payload im Sinne von `IKEv2`, da aber das Parsen und Generieren nach dem gleichen Prinzip abläuft, erscheint es sinnvoll, den Header gleich zu behandeln.

Neben den `payload_t`-Methoden bietet diese Klasse diverse Methoden zum Auslesen und Setzen der einzelnen Felder des `IKE`-Headers.

5.5.15.3 sa_payload_t

`sa_payload_t` stellt ein `IKEv2`-Payload vom Typ Security Association dar. Ein solcher Payload enthält mehrere Proposal Substructures. Beim Erstellen eines SA Payloads können `proposal_substructure_t`-Objekte angehängt werden. Für das Auslesen eines solchen Payloads kann ein Iterator für die Substrukturen bezogen werden.

5.5.15.4 proposal_substructure_t

Obwohl eine Proposal Substructure im Sinne von `IKEv2` kein Payload ist, implementiert `proposal_substructure_t`, genau gleich wie `ike_header_t`, das `payload_t`-Interface. Zusätzlich können über diverse Methoden die Felder dieses Payloads gelesen und gesetzt werden. Da eine Proposal Substructure mehrere `transform_substructure_t`-Objekte enthält, können solche angehängt bzw. ein Iterator für diese bezogen werden.

1 C-Typ bzw. eine Ableitungen davon.

5.5.15.5 transform_substructure_t

Neben den Methoden von payload_t bietet transform_substructure_t spezifische Methoden für den Zugriff auf die Felder dieses Payloads. Für die enthaltenen transform_attribute_t-Objekte kann ein Iterator bezogen, bzw. können solche an die Substruktur angehängt werden.

5.5.15.6 transform_attribute_t

Die letzte Stufe der Verschachtelung eines SA Payloads stellt die Klasse transform_attribute_t dar. Über die gegebenen Methoden können die Daten eines solchen Attributs gelesen bzw. modifiziert werden.

5.5.15.7 ke_payload_t

Bei ke_payload_t handelt es sich um die Repräsentation des IKEv2-Payloads KE. Über spezifische Methoden kann auf die Daten des Diffie-Hellman-Austauschs zugegriffen werden.

5.5.15.8 nonce_payload_t

Der Nonce Payload wird auf die Klasse nonce_payload_t abgebildet. Über die definierten Methoden kann die Nonce gelesen oder gesetzt werden.

5.5.15.9 auth_payload_t

Die Klasse auth_payload_t repräsentiert den IKEv2-Payload AUTH. Ein AUTH Payload beinhaltet eine Authentisierungsmethode zusammen mit den Authentisierungsdaten, beispielsweise eine RSA-Signatur. Diese Klassen bietet Funktionen zum Lesen und Setzen von Authentisierungsmethode und Authentisierungsdaten.

5.5.15.10 id_payload_t

Die Payloads IDi und IDr werden beide durch die Klasse id_payload_t repräsentiert. Diese Klasse bietet Methoden für die Umwandlung eines solchen Payloads in ein Objekt vom Typ identification_t und umgekehrt.

5.5.15.11 ts_payload_t

ts_payload_t stellt ein Payload vom Typ TS dar. Ein solcher Payload enthält mehrere Traffic Selectors, also Objekte vom Typ traffic_selector_substructure_t. Beim Erstellen eines TS Payloads können traffic_selector_substructure_t-Objekte angehängt werden. Für das Auslesen eines solchen Payloads kann ein Iterator für die Substrukturen bezogen werden.

5.5.15.12 traffic_selector_substructure_t

Die Klasse traffic_selector_substructure_t stellt einen Traffic Selector des TS Payloads dar. Die Klasse besitzt Methoden, um direkt mit der Klasse traffic_selector_t zu interagieren, welche die Abstraktion des eigentlichen Traffic Selectors darstellt.

5.5.15.13 encryption_payload_t

Die Klasse `encryption_payload_t` repräsentiert den Encrypted Payload. Über die Methode `add_payload()` können beliebige `payload_t`-Objekte dem Payload hinzugefügt werden. Durch den Aufruf von `encrypt()` werden die enthaltenen Payloads verschlüsselt und intern in einer Datenstruktur gespeichert.

Der Aufruf von `decrypt()` entschlüsselt die interne Datenstruktur und erstellt eine Liste der entschlüsselten Payloads vom Typ `payload_t`. Über einen Iterator oder die Methode `remove_first_payload()` können die entschlüsselten Payloads ausgelesen und bearbeitet werden.

Die Funktion `verify_signature()` überprüft mit Hilfe eines kompletten Paketes, ob die darin enthaltene Signatur gültig ist. Die Funktion `build_signature()` erstellt umgekehrt anhand eines Paketes eine Signatur, die sie ans Ende des Paketes schreibt.

Wie dieser Payload zur Anwendung kommt, ist unter 5.6.2 beschrieben.

5.5.15.14 notify_payload_t

Der Notify Payload wird auf die Klasse `notify_payload_t` abgebildet. Die Klasse bietet Methoden zum Lesen und Setzen der spezifischen Notify-Daten.

5.5.15.15 delete_payload_t

Der Payload DELETE wird durch die Klasse `delete_payload_t` repräsentiert. Diese Klasse bietet Methoden zum Lesen und Setzen der zu löschenden SPIs.

5.5.15.16 cp_payload_t

Die Klasse `cp_payload_t` repräsentiert den Configuration Payload. Ein CP Payload besteht aus einem oder mehreren Attributen. So bietet die Klasse Methoden zum Lesen und Einfügen von `configuration_attribute_t`-Objekten.

5.5.15.17 configuration_attribute_t

Diese Klasse repräsentiert ein einzelnes Attribut eines CP Payloads. Dabei stellt die Klasse kein eigener `IKEv2`-Payload dar und wird nur innerhalb der Klasse `cp_payload_t` eingesetzt. `configuration_attribute_t` bietet Methoden zum Lesen und Setzen von Typ und Wert des Attributs.

5.5.15.18 cert_payload_t

Der Payload CERT wird durch die Klasse `cert_payload_t` repräsentiert. Diese Klasse bietet Methoden zum Lesen und Setzen der Felder des CERT Payloads.

Als Erweiterungen dieser Klasse wären Methoden zum Lesen und Setzen von eigenen Zertifikaten denkbar. Dies setzt jedoch eine geeignete Klasse zur Repräsentation eines Zertifikats voraus.

5.5.15.19 certreq_payload_t

Die Klasse `certreq_payload_t` repräsentiert den `IKEv2`-Payload CERTREQ.

Auch hier wäre das direkte Lesen und Setzen von CA-Zertifikaten sinnvoll. Dies setzt jedoch wiederum eine geeignete Klasse zur Repräsentation eines Zertifikats voraus.

5.5.15.20 eap_payload_t

Der Payload EAP wird durch die Klasse `eap_payload_t` repräsentiert. Sie bietet erst rudimentäre Funktionen, da EAP noch nicht unterstützt wird.

5.5.15.21 vendor_id_payload_t

Der Vendor ID Payload von `IKEv2` wird auf die Klasse `vendor_id_payload_t` abgebildet. Die Klasse bietet Methoden zum Lesen und Setzten der Vendor ID-Daten.

5.5.15.22 unknown_payload_t

Alle unbekannten Payload-Typen werden als Objekt der Klasse `unknown_payload_t` abstrahiert. Diese Klasse bietet primär eine Funktion zum Prüfen, ob der unbekannte Payload als Critical markiert ist oder nicht. Anhand dessen kann entschieden werden, wie dieser, nicht weiter verarbeitbare, Payload gehandhabt werden soll.

5.5.16 config

Im Package `config` ist alles Relevante über die Konfiguration des Daemons enthalten. Neben Verbindungskonfigurationen sind hier unter anderem auch Schlüssel oder Zertifikate zu beziehen.

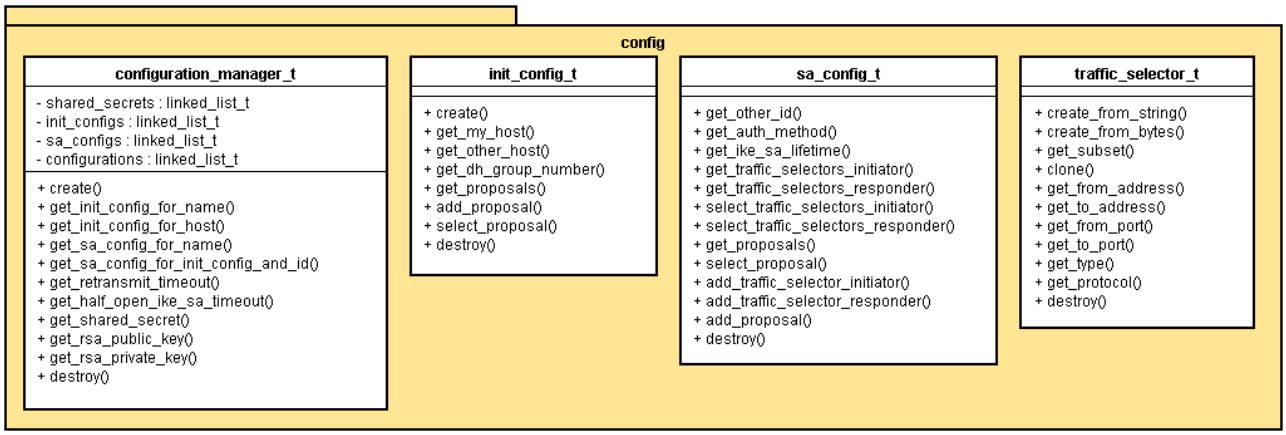


Abbildung 41: Klassen des Packages "config"

5.5.16.1 configuration_manager_t

Die Klasse `configuration_manager_t` liefert verschiedenste Informationen betreffend der Konfiguration. Abhängig davon, ob man eine IKE_SA als Initiator aufbaut oder als Responder agiert werden Konfigurationen unterschiedlich bezogen. Als Initiator baut man eine Verbindung mit der Angabe eines Namens auf, wobei diesem die Konfiguration zugeordnet ist. Als Responder muss die Konfiguration mit Angabe von Host und ID-Informationen bezogen werden. Der Konfigurations-Manager unterscheidet INIT- und SA-Konfigurationen, die in den Klassen `init_config_t` und `sa_config_t` implementiert sind. Eine INIT-Konfiguration beinhaltet alle Details für den IKE_SA_INIT-Austausch. Alle Details für den IKE_AUTH-Austausch und folgenden CREATE_CHILD- und INFORMATIONAL-Austausche sind in der SA-Konfiguration enthalten.

Neben den Konfigurationen für die einzelnen Verbindungen verwaltet der Konfigurations-Manager auch geheime und öffentliche Schlüssel.

5.5.16.2 init_config_t

Ein Objekt der Klasse `init_config_t` repräsentiert eine Konfiguration für den IKE_SA_INIT-Austausch. Der Initiator bezieht beim Konfigurations-Manager mit der Angabe des Konfigurationsnamens das `init_config_t`-Objekt. Der Responder bezieht die Konfiguration mit der Angabe des Hosts. Beide können nun über dieses die notwendigen Informationen für den IKE_SA_INIT-Austausch abfragen.

Neben den Host-Informationen der Kommunikationspartner liefert diese Klasse auch die zu verwendende Diffie-Hellman-Gruppe oder selektiert ein vorgeschlagenes `IKE`-Proposal. Ein einzelnes Proposal wird dabei über die Datenstruktur `ike_proposal_t` definiert. Diese besitzt aber keine Methode und wird deshalb nicht als Klasse angesehen.

5.5.16.3 sa_config_t

Ein Objekt der Klasse `sa_config_t` repräsentiert die Konfiguration, die ab dem IKE_AUTH-Austausch benötigt wird. Der Initiator bezieht beim Konfigurations-Manager mit der Angabe des Konfigurationsnamens das `sa_config_t`-Objekt. Der Responder bezieht das `sa_config_t`-Objekt mit der Angabe des vorherigen `init_config_t`-Objekts und der Identität des Initiators.

Beide können nun über dieses die notwendigen Informationen für den IKE_AUTH-Austausch und die folgenden CREATE_CHILD- und INFORMATIONAL-Austausche abfragen.

Diese Klasse liefert alle Informationen, die zum Aufbau einer CHILD_SA notwendig sind. Die Handhabung von Traffic Selectors funktioniert dabei über die Klasse `traffic_selector_t`. Die notwendigen Proposals werden als Datenstruktur vom Typ `child_proposal_t` dargestellt, welche aber wiederum keine eigentliche Klasse darstellen.

5.5.16.4 traffic_selector_t

Die beim Aufbau einer CHILD_SA benötigten Traffic Selectors werden in der Klasse `sa_config_t` als Objekte vom Typ `traffic_selector_t` gespeichert und verwaltet.

Eine `traffic_selector_t`-Instanz definiert einen Adress- sowie Port-Bereich für ein spezifisches Protokoll. Die Methode `get_subset()` erlaubt das Vergleichen zweier Objekte diesen Typs, um daraus den grössten gemeinsamen Adress- und Port-Bereich zu ermitteln. Über diese Methode kann sehr einfach das Auswählen von Traffic Selectors realisiert werden.

5.5.17 Klasse daemon_t

`daemon_t` ist die einzige Klasse, welche keinem Package zugeordnet ist. Sie repräsentiert den gesamten Daemon und ist für die Verwaltung aller globalen Objekte zuständig. Über öffentliche Variablen wird auf diese Objekte zugegriffen. Die einzige öffentliche Methode ist `kill()`, welche alle globalen Objekte zerstört und die Threads beendet.

Der Konstruktor ist nicht öffentlich. Das einzige `daemon_t`-Objekt heisst `charon` und wird beim Starten des Daemons durch die `main()`-Funktion instanziert. Dies ist möglich da die entsprechende C-Datei `daemon.c` die `main()`-Funktion enthält und somit den Eintrittspunkt des Daemons bildet, aber auch Zugriff auf den privaten Konstruktor hat. Das Folgende Klassendiagramm zeigt den Aufbau von `daemon_t`:

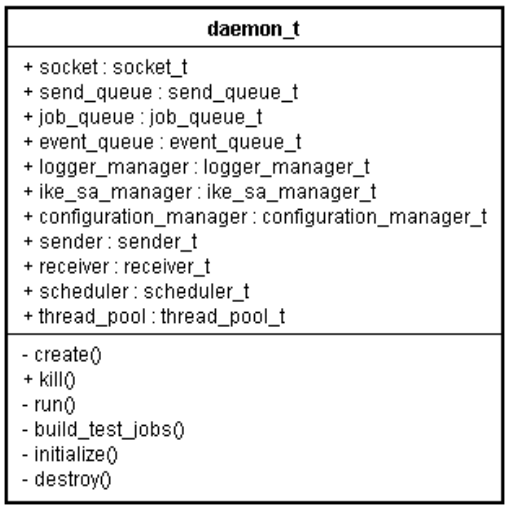


Abbildung 42: Die Klasse "daemon_t"

5.6 Ablaufszenarien

Um den Zusammenhang der gesamten Architektur besser zu verdeutlichen, sind nachfolgend wichtige Ablaufszenarien beschrieben.

Die Funktionen entsprechen meist den Bezeichnungen, wie sie im Quellcode verwendet werden. Zur höheren Übersichtlichkeit sind die Funktionsaufrufe jedoch vereinfacht dargestellt. Auch ist nur der Erfolgsfall beschrieben.

Ein Aktor in den Sequenzdiagrammen entspricht jeweils einem Thread.

Zum Verständnis der Sequenzdiagramme empfiehlt es sich, den Quellcode als Referenz zu nehmen, da einige Abläufe auch mit dem Diagramm immer noch sehr komplex sind.

5.6.1 IKEv2-Message generieren

Wie aus dem Dokument „Technologien“ entnommen werden kann, hat eine `IKEv2`-Message stets den gleichen Aufbau: Ein Header und ein oder mehrere Payloads. In dieser Implementierung ist eine Message als Objekt der Klasse `message_t` abstrahiert. Das folgende Sequenzdiagramm zeigt, wie eine solche `message_t`-Objekt erstellt und daraus eine `IKEv2`-Message in binärer Form generiert wird:

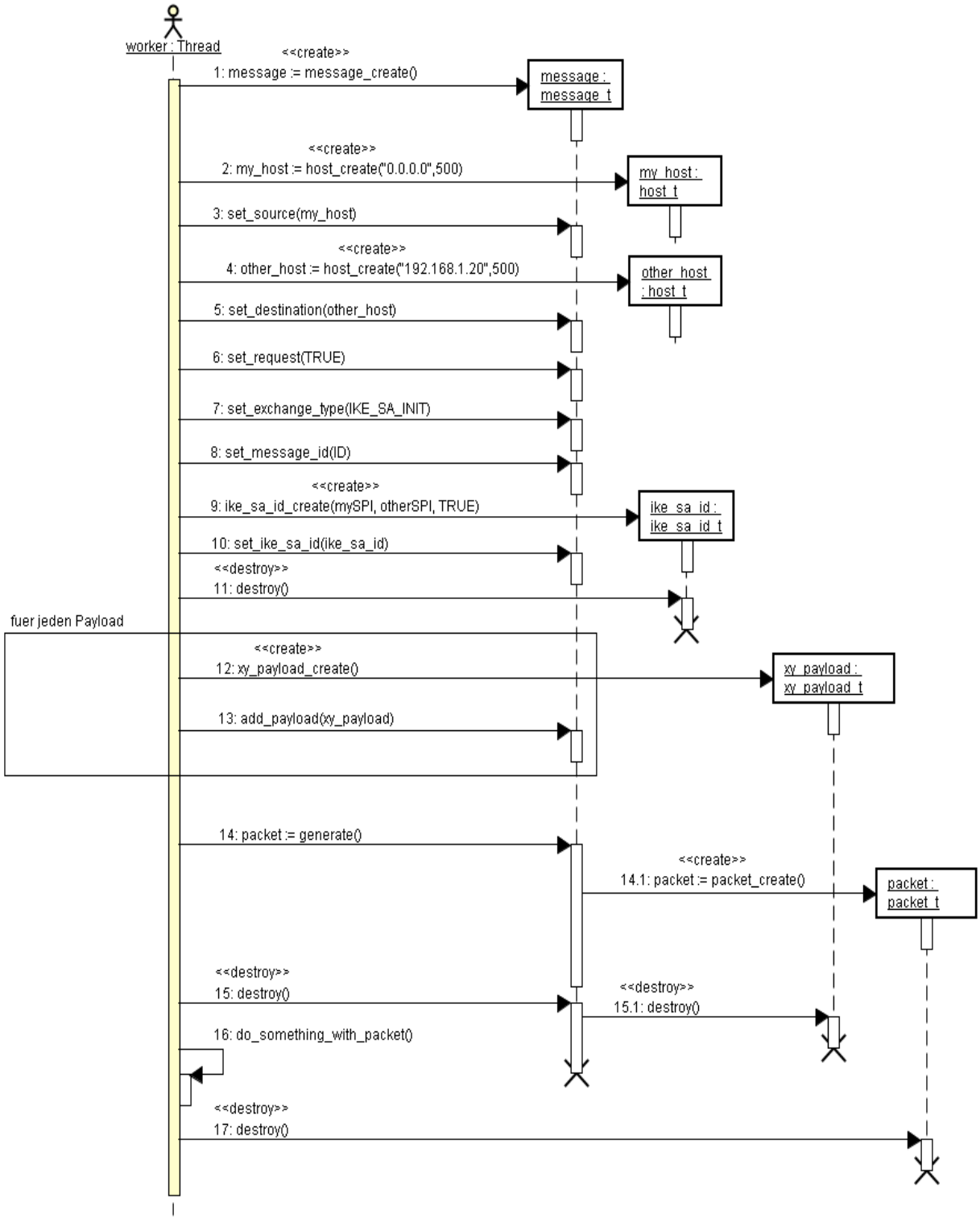


Abbildung 43: Ablauf beim Generieren einer IKEv2-Message

Die Schritte aus dem obigen Sequenzdiagramm sind folgendermassen zu interpretieren:

1. Das Objekt `message` vom Typ `message_t` wird erstellt.
2. Das Objekt `my_host` vom Typ `host_t` wird erstellt. Dieses soll den eigenen Computer repräsentieren und wird darum mit der Default-Route IP-Adresse `0.0.0.0` erstellt. Der Port wird auf den Standard `IKE`-Port `500` gesetzt.
3. Die Absender-Adresse des `message_t`-Objekts wird gesetzt.
4. Das Objekt `other_host` vom Typ `host_t` wird erstellt. Dieses soll den Empfänger repräsentieren und wird darum mit der IP-Adresse `192.168.1.20` erstellt. Der Port wird auch hier auf den Standard `IKE`-Port `500` gesetzt.
5. Die Empfänger-Adresse des `message_t`-Objekts wird gesetzt.
6. Die Message ist ein `IKEv2`-Request.
7. Die Message ist vom Typ IKE_SA_INIT.
8. Die Message-ID wird gesetzt.
9. Das Objekt `ike_sa_id` vom Typ `ike_sa_id_t` wird erstellt. Es repräsentiert eine eindeutige Identifikation einer IKE-SA.
10. Die IKE_SA-ID der Message wird gesetzt.
11. Das `ike_sa_id_t`-Objekt kann zerstört werden. Das Objekt `message` hat im vorherigen Schritt eine interne Kopie von `ike_sa_id` angelegt.
12. Ein Payload vom Typ XY wird erstellt. Abhängig vom Typ der `IKEv2`-Message sind verschiedene Payloads zu erstellen.
13. Der erstellte Payload wird zur Message hinzugefügt. Von nun an ist das Objekt `message` für die Zerstörung des hinzugefügten Payloads zuständig.
14. Die `IKEv2`-Message kann generiert werden. Es wird ein Objekt vom Typ `packet_t` erstellt, welches ein UDP-Paket abstrahiert.
15. Das Objekt `message` wird nicht mehr benötigt und kann zerstört werden. Dieses stellt sicher, dass auch alle enthaltenen Payloads zerstört werden.
16. Über das Objekt `packet` kann auf die binäre Darstellung der Message zugegriffen werden.
17. Zum Schluss muss das erstellte `packet` zerstört werden.

Beim Generieren wird ein Objekt vom Typ `packet_t` erstellt. Dieses Objekt beinhaltet neben der binären Message zusätzlich die Informationen von Absender und Empfänger und kann so über die Send-Queue direkt an den Sender-Thread weitergeleitet werden.

Der Ablauf beim Versenden eines generierten Pakets ist unter 5.6.5 genauer beschrieben.

5.6.2 Verschlüsselte IKEv2-Message generieren

Der Ablauf beim Generieren einer verschlüsselten Message ist für den Aufrufer transparent und läuft somit wie unter 5.6.1 beschrieben ab. Im `message_t`-Objekt selbst ergibt sich aber ein relativ komplexer Ablauf. Das folgende Sequenzdiagramm zeigt den Ablauf beim Generieren einer verschlüsselten `IKEv2`-Message. Es wird angenommen, dass bereits alle Payloads mit `add_payload()` zum entsprechenden `message_t`-Objekt hinzugefügt wurden:

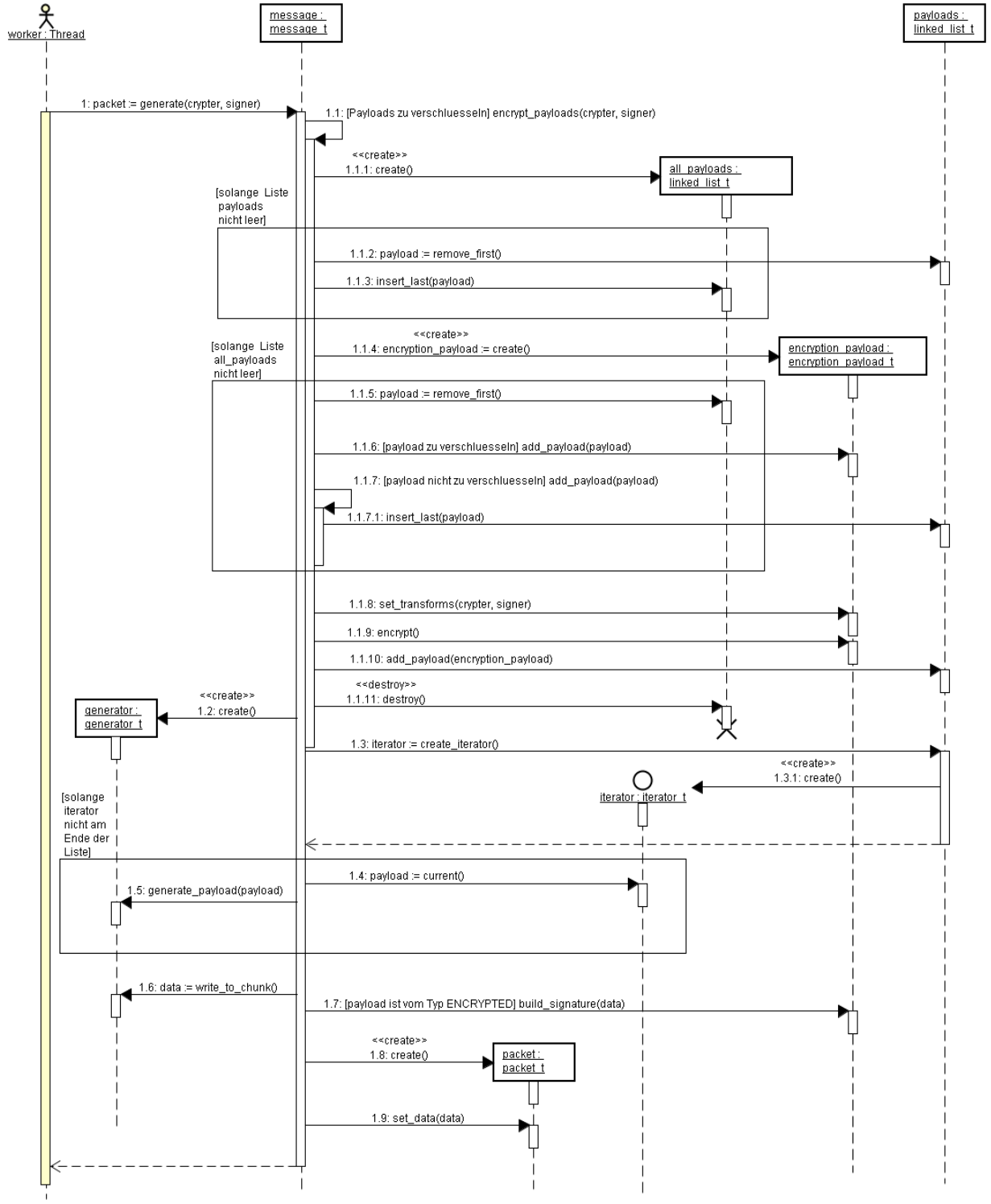


Abbildung 44: Ablauf beim Generieren einer verschlüsselten IKEv2-Message

Die Schritte aus dem obigen Sequenzdiagramm werden nun genauer erläutert. Die Ausgangslage ist, dass der Worker-Thread bereits mehrere Payloads einem `message_t`-Objekt hinzugefügt hat und nun aus dieser Message ein `packet_t`-Objekt erstellen möchte.

1. Der Worker-Thread ruft die `generate()`-Funktion des `message_t`-Objekts auf. Als Parameter gibt er ein `crypter_t`-und ein `signer_t`-Objekt mit, um damit Payloads zu verschlüsseln und zu signieren.
   1.1. Das `message_t`-Objekt ruft nun die private Funktion `encrypt_payloads()` auf. Diese Funktion ist zuständig für eine allfällige Verschlüsselung der Payloads.
      1.1.1. Eine temporäre Linked-List wird erstellt.
      1.1.2. Ein Payload wird aus der internen Payload-Liste entnommen.
      1.1.3. Der entnommene Payload wird in die temporäre Liste eingefügt.
      1.1.4. Ein neues Objekt vom Typ `encryption_payload_t` erstellt.
      1.1.5. Der nächste Payload wird aus der temporären Liste entnommen.
      1.1.6. Falls der entnommen Payload verschlüsselt werden muss, wird dieser dem vorher erstellten Objekt `encryption_payload` hinzugefügt.
      1.1.7. Falls der entnommen Payload nicht verschlüsselt werden muss, wird dieser wieder in die interne Payload-Liste eingefügt.
      1.1.8. Das `crypter_t`- und `signer_t`-Objekt wird dem erstellten `encryption_payload_t`-Objekt mitgeteilt. Diese beiden Objekte werden zur Verschlüsselung und Signierung verwendet.
      1.1.9. Das `encryption_payload_t`-Objekt wird beauftragt alle darin enthaltenen Payloads zu verschlüsseln.
      1.1.10. Das `encryption_payload_t`-Objekt beinhaltet jetzt die verschlüsselte Darstellung der Payloads und kann der internen Payload-Liste hinzugefügt werden.
      1.1.11. Die temporäre Liste enthält keine Payloads mehr und kann gelöscht werden.
   1.2. Ein `generator_t`-Objekt wird erstellt.
   1.3. Ein Iterator über alle, in der internen Liste gespeicherten, Payloads wird erstellt.
   1.4. Der Payload bei der aktuellen Iterator-Position wird gelesen.
   1.5. Der Generator wird beauftragt den Payload zu generieren.
   1.6. Die generierten Daten werden gelesen.
   1.7. Falls der letzte generierte Payload vom Typ ENCRYPTED war, wird die Signatur über die generierten Daten gebildet.
   1.8. Das `packet_t`-Objekt wird erstellt.
   1.9. Die generierte und signierten Daten werden gesetzt.

Der Worker-Thread kann nun das generierte Paket in die Send-Queue legen, damit dieses so schnell als möglich versendet wird.

Das Diagramm zeigt, dass sich der Worker-Thread beim Generieren der `IKEv2`-Message nicht darum kümmern muss, welche Payloads verschlüsselt werden müssen und welche nicht. Diesen Job erledigt das `message_t`-Objekt, indem es intern eine Liste mit diesen Informationen für jeden Message-Typ führt.

5.6.3 IKEv2-Message parsen

Unter 5.6.1 ist der Ablauf beim Generieren einer `IKEv2`-Message beschrieben. Auch beim Parsen einer solchen Message spielt die Klasse `message_t` eine zentrale Rolle. Das folgende Sequenzdiagramm zeigt, wie eine vorhandene binäre `IKEv2`-Message geparst und daraus ein `message_t`-Objekt erstellt wird:

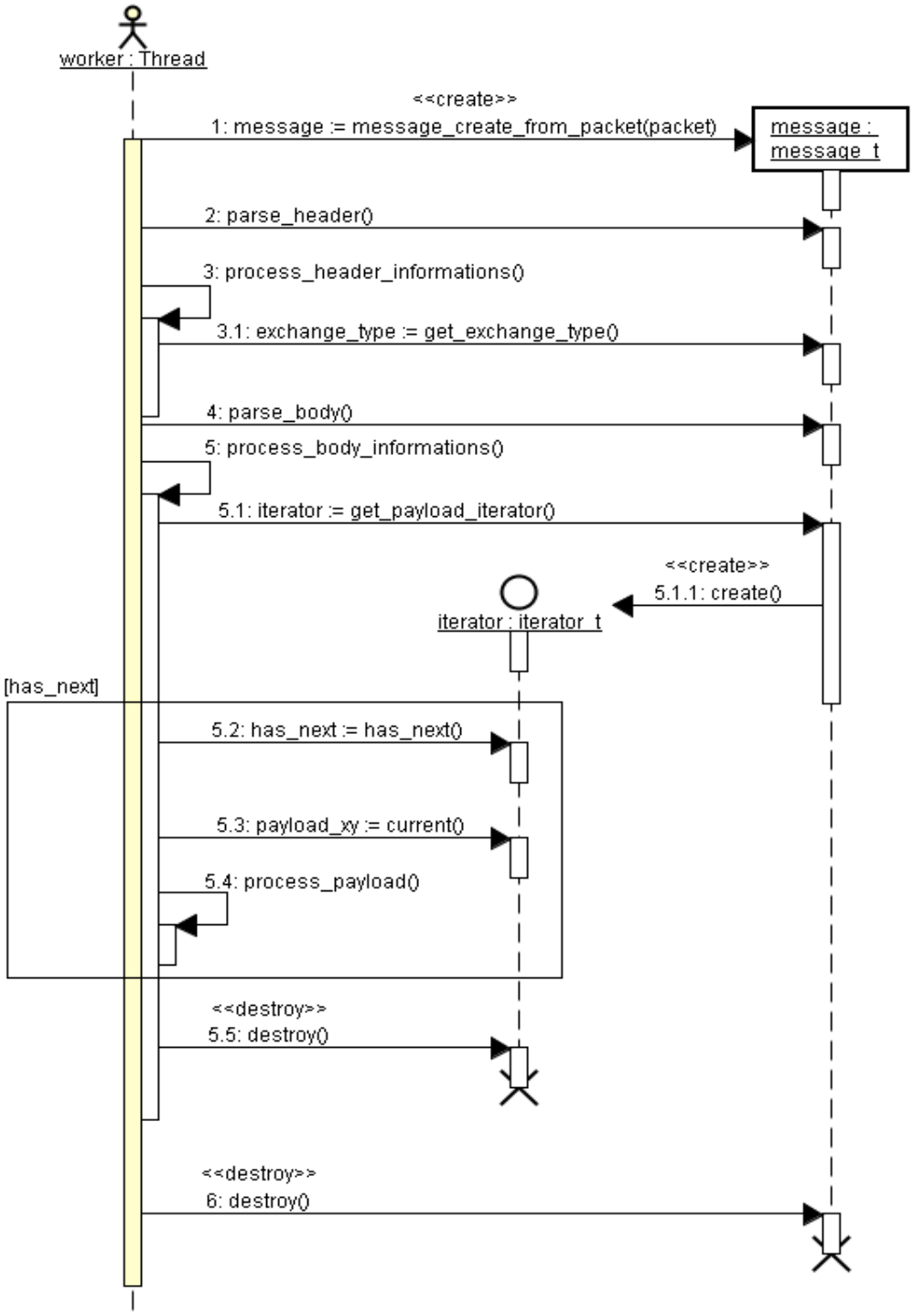


Abbildung 45: Ablauf beim Parsen einer IKEv2-Message

Die Schritte aus dem obigen Sequenzdiagramm werden nun genauer erläutert. Dabei wird davon ausgegangen, dass bereits ein Objekt `packet` vom Typ `packet_t` vorhanden ist und dieses die binäre Form einer Message enthält:

1. Das Objekt `message` vom Typ `message_t` wird mit der Angabe vom Objekt `packet` erstellt. Der Konstruktor speichert das übergebene `packet_t`-Objekt. Von nun an ist das Objekt `message` für die Zerstörung des angegebenen `packet_t`-Objektes zuständig.
2. Das `message_t`-Objekt wird beauftragt den Header des Pakets zu parsen. Tritt dabei ein Fehler auf, hat das empfangene Paket nicht die korrekte `IKEv2`-Header Struktur.
3. Nach erfolgreichem Parsen des Headers können die darin enthaltenen Informationen abgerufen werden.
   3.1. Als Beispiel wird der Typ der `IKEv2`-Message ausgelesen.
4. Jetzt kann der Body der Message geparst werden. Dabei wird neben dem korrekten Aufbau der Message auch geprüft, ob enthaltenen Payload-Typen für einen bestimmten Message-Typ überhaupt erlaubt sind und ob verwendete Wertebereiche, z.B. bei der Angabe eines Algorithmus, eingehalten werden.
5. Nach erfolgreichem Parsen aller Payloads können diese verarbeitet werden.
   5.1. Ein Iterator für die Payloads wird erstellt.
   5.2. Es wird geprüft, ob noch weitere Payloads vorhanden sind.
   5.3. Der aktuelle Payload wird ausgelesen.
   5.4. Der Payload wird bearbeitet.
   5.5. Nach Bearbeitung aller Payloads wird der Iterator vom Typ `iterator_t` zerstört.
6. Zum Schluss muss das erstellte `message_t`-Objekt `message` zerstört werden.

Wie bereits erwähnt, wird im obigen Diagramm davon ausgegangen, dass ein Objekt vom Typ `packet_t` bereits existiert. Der Ablauf beim Empfangen eines UDP-Pakets und Erstellen eines solchen Objekts ist unter 5.6.6 genauer beschrieben.

5.6.4 Verschlüsselte IKEv2-Message parsen

Der Ablauf beim Parsen einer verschlüsselten `IKEv2`-Message ist für den Aufrufer transparent und läuft somit wie unter 5.6.3 beschrieben ab. Im `message_t`-Objekt selbst ergibt sich aber ein komplexer Ablauf. Das folgende Sequenzdiagramm zeigt den stark vereinfachten Ablauf beim Parsen einer verschlüsselten `IKEv2`-Message. Es wird angenommen, dass bereits ein `message_t`-Objekt aus einem `packet_t`-Objekt erstellt und der `IKEv2`-Header mit `parse_header()` geparst wurde:

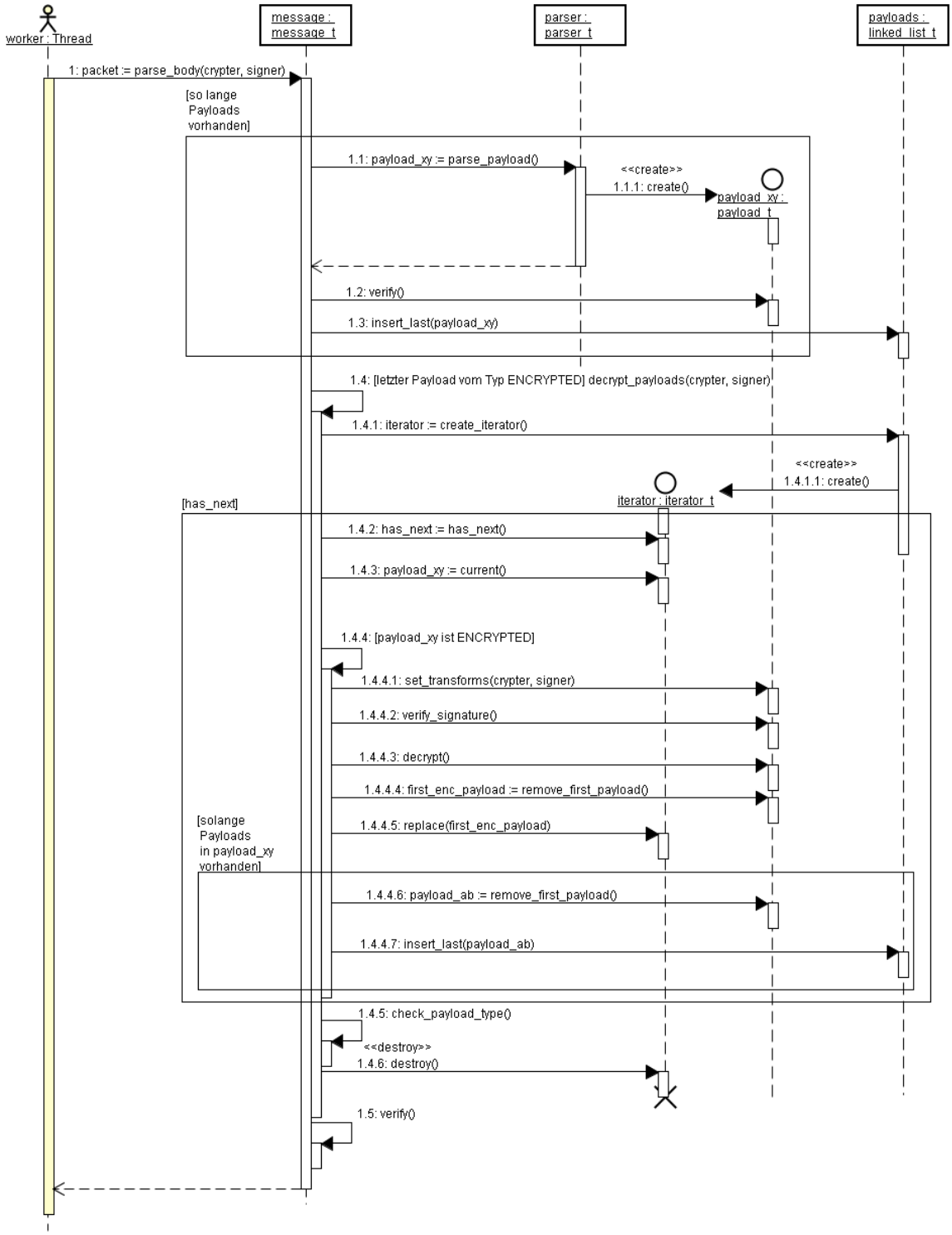


Abbildung 46: Ablauf beim Parsen einer verschlüsselten IKEv2-Message

Die Schritte aus dem obigen Sequenzdiagramm werden nun genauer erläutert.

1. Der Worker-Thread ruft die `parse_body()`-Funktion des `message_t`-Objekts auf. Als Parameter gibt er ein `crypter_t`- und ein `signer_t`-Objekt mit, um damit Payloads zu entschlüsseln und die Signatur zu überprüfen.
   1.1. Der nächste Payload wird geparst. Den Typ des ersten Payloads wurde aus den bereits geparsten Header-Daten entnommen.
      1.1.1. Der Parser erstellt ein neues `payload_t`-Objekt und füllt dieses mit den geparsten Daten ab. Dazu benutzt er die Regeln des entsprechenden Payloads.
   1.2. Der Payload-Inhalt wird überprüft. Dabei wird geprüft, ob die Felder auch Werte aus dem erlaubten Wertebereich enthalten.
   1.3. Der geparste Payload wird in die interne Liste eingefügt.
   1.4. Falls der letzte Payload vom Typ ENCRYPTED ist, wird die interne Funktion `decrypt_payload()` aufgerufen. Sie ist zuständig für das Entschlüsseln der enthaltenen Payloads.
      1.4.1. Ein Iterator über die interne Payload-Liste wird erstellt.
      1.4.2. Es wird geprüft ob noch Einträge in der Liste sind.
      1.4.3. Der nächste Payload wird gelesen.
      1.4.4. Falls der Payload vom Typ ENCRYPTED ist, muss er entschlüsselt werden.
         1.4.4.1. Die Objekte für Entschlüsslung und Prüfung der Signatur werden gesetzt.
         1.4.4.2. Die Signatur wird geprüft.
         1.4.4.3. Der Payload wird entschlüsselt.
         1.4.4.4. Der erste Payload wird aus dem nun entschlüsselten Payload entnommen.
         1.4.4.5. Der Payload an der aktuellen Iterator-Position wird durch den ersten diesen entnommenen Payload ersetzt. Dadurch enthält die interne Liste keinen Payload mehr vom Typ ENCRYPTED.
         1.4.4.6. Der nächste Payload wird aus dem nun entschlüsselten Payload entnommen.
         1.4.4.7. Der entnommene Payload wird am Ende der internen Liste eingefügt.
      1.4.5. Es wird geprüft, ob der Payload überhaupt erlaubt ist für den aktuellen Message-Typ und falls ja, ob dieser verschlüsselt sein muss oder nicht.
      1.4.6. Der Iterator kann zerstört werden.
   1.5. Die gesamte Message wird überprüft. Dabei wird geschaut, ob alle Payloads für einen bestimmten Message-Typ vorhanden sind und ob deren Anzahl stimmt.

Das Diagramm zeigt, dass sich der Worker-Thread beim Parsen der `IKEv2`-Message nicht darum kümmern muss, welche Payloads entschlüsselt werden müssen und welche nicht. Diesen Job erledigt das `message_t`-Objekt, indem es intern eine Liste mit diesen Informationen für jeden Message-Typ führt.

5.6.5 Paket versenden

Nachdem eine `IKEv2`-Message erstellt und generiert wurde, muss das erhaltene Paket in Form eines `packet_t`-Objektes versendet werden. Dazu wird dieses `packet_t`-Objekt in die Send-Queue eingereiht. Der Sender-Thread entnimmt das `packet_t`-Objekt aus der Send-Queue und sendet dieses an den entsprechenden Empfänger. Daraus ergibt sich folgendes Sequenzdiagramm:

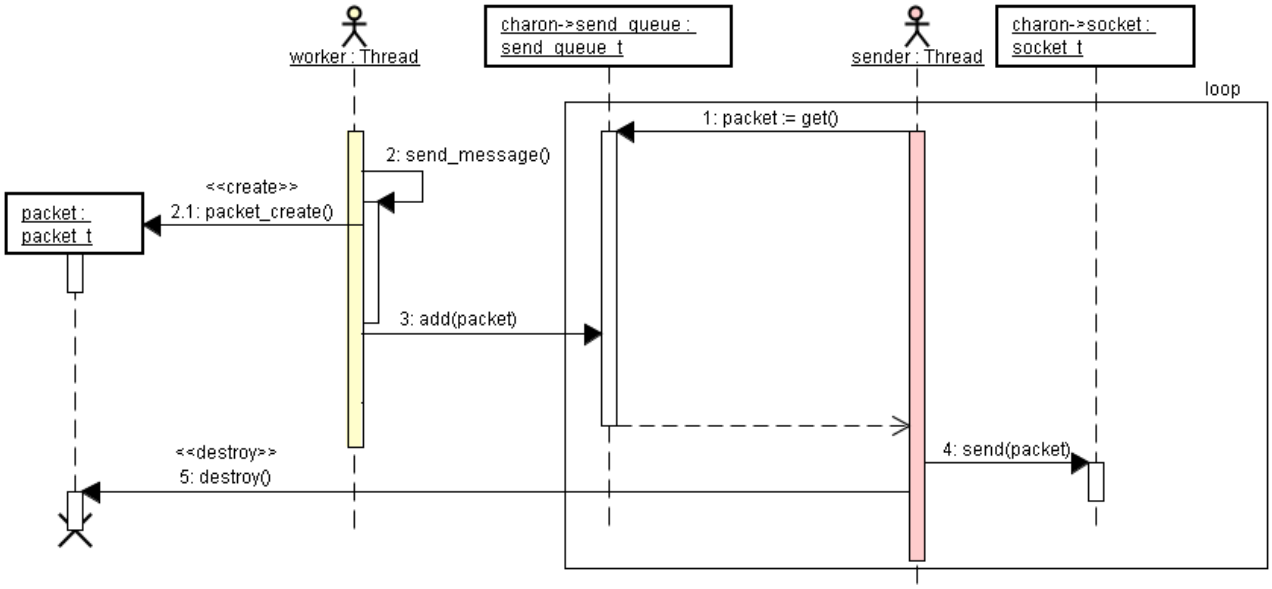


Abbildung 47: Ablauf beim Versenden eines Pakets

Die Schritte aus dem obigen Sequenzdiagramm werden nun genauer erläutert. Dabei wird davon ausgegangen, dass bereits ein Objekt `packet` vom Typ `packet_t` generiert wurde und dieses die binäre Form der `IKEv2`-Message enthält (siehe 5.6.1):

1. Der Sender-Thread holt aus der Send-Queue das nächste zu versendende Paket vom Typ `packet_t`. Falls die Send-Queue leer ist, blockiert diese Funktion so lange, bis ein Paket eingefügt wird. Den Zugriff auf die Send-Queue vom Typ `send_queue_t` bekommt der Thread über die globale `daemon_t`-Instanz `charon`.
2. Ein Worker-Thread erstellt eine Message und generiert anschliessend ein Paket daraus.
3. Er fügt das Paket vom Typ `packet_t` in die Send-Queue. Nun kann sich der Worker-Thread um eine andere Arbeit kümmern. Er kann sich darauf verlassen, dass das Paket so schnell als möglich vom Sender-Thread versendet wird.
4. Der Sender-Thread sendet das Paket über den globalen Socket vom Typ `socket_t`. Den Zugriff auf das `socket_t`-Objekt bekommt er über das globale `daemon_t`-Objekt `charon`. Er entnimmt alle Informationen (Absender, Empfänger und binäre Paketdaten) aus dem Objekt vom Typ `packet_t`.
5. Sobald das Paket versendet werden konnte, wird das Objekt `packet` zerstört.

5.6.6 Paket empfangen

Eingehende UDP-Pakete werden vom Receiver-Thread empfangen und zur Bearbeitung an einen Worker-Thread weitergeleitet. Wie bereits erwähnt, wird ein UDP-Paket als Objekt vom Typ `packet_t` abstrahiert. Beim Empfangen von Paketen wird diese Kapselung vom globalen `socket_t`-Objekt vorgenommen. Für das Empfangen von Paketen ergibt sich folgender Ablauf:

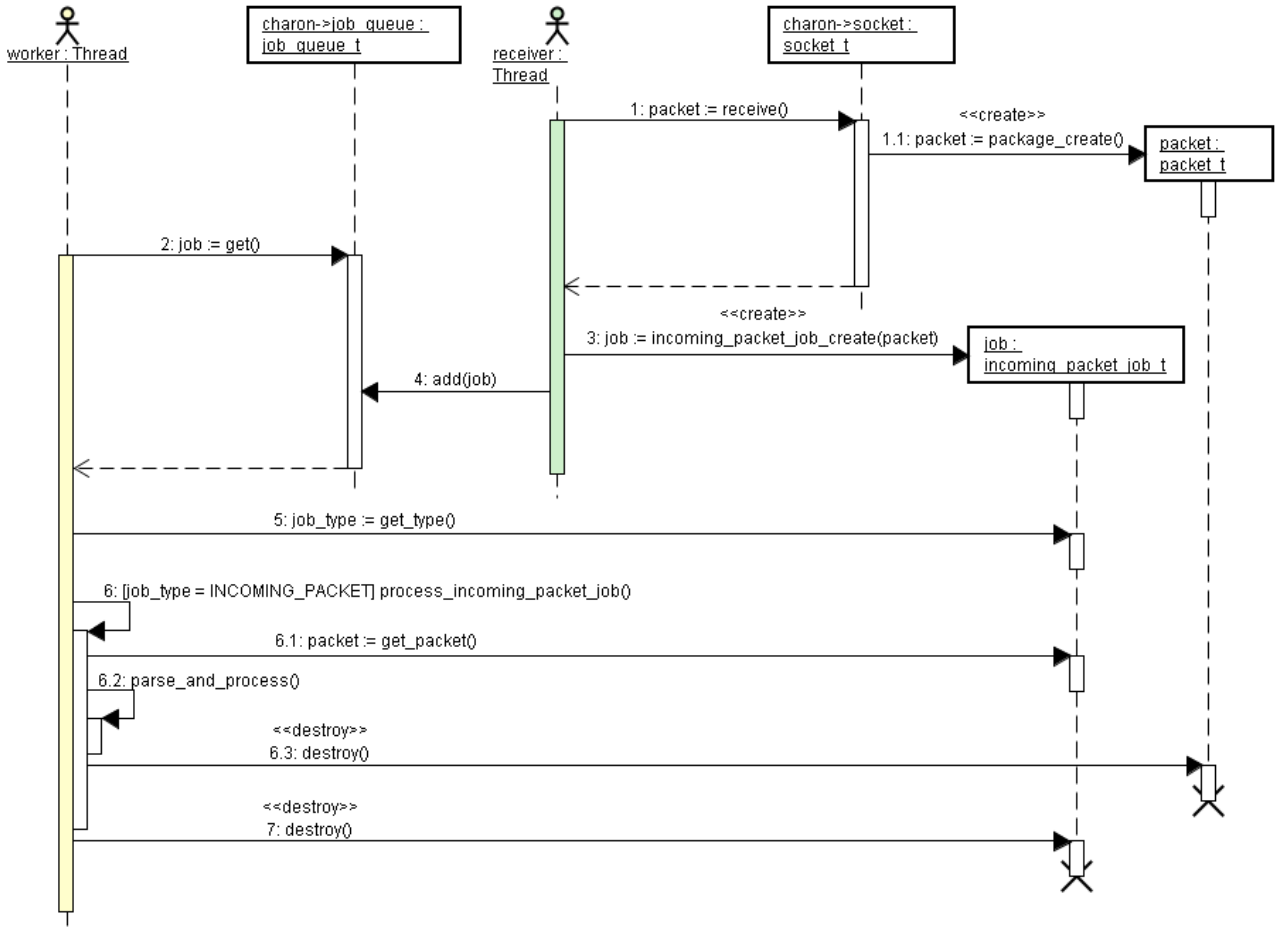


Abbildung 48: Ablauf beim Empfangen eines Pakets

Die Schritte aus dem obigen Sequenzdiagramm werden nun genauer erläutert:

1. Der Receiver-Thread möchte ein UDP-Paket empfangen und ruft dazu die Funktion `receive()` des vom globalen `daemon_t`-Objekt `charon` verwalteten `socket_t`-Objekts auf. Die Funktion blockiert so lange, bis ein UDP-Paket über die verwalteten Sockets empfangen wurde.
   1.1. Hat das `socket_t`-Objekt ein eingegangenes UDP-Paket registriert, erstellt es ein Objekt vom Typ `packet_t` und füllt dieses mit den empfangenen Daten ab.
2. In der Zwischenzeit hat ein Worker-Thread seine vorherige Aufgabe beendet und möchte nun einen nächsten Job ausführen. Dazu ruft dieser die `get()`-Funktion der globalen Job-Queue auf. Diese Funktion blockiert so lange, bis dem entsprechenden Worker-Thread ein pendenter Job zurückgegeben werden kann.
3. Der Receiver-Thread erstellt einen Job vom Typ `incoming_packet_job_t`.
4. Der neue Job wird in die globale Job-Queue eingefügt. Der Receiver-Thread beginnt nun wieder bei Schritt 1.
5. Da jeder Job in der globalen Job-Queue das Interface `job_t` implementiert, kann der Worker-Thread herausfinden, um was für ein Typ Job es sich handelt.
6. Abhängig vom Typ des Jobs wird dieser unterschiedlich bearbeitet. Im obigen Diagramm wird angenommen, dass es sich um ein Job vom Typ INCOMING_PACKET handelt. Die entsprechende Bearbeitungsfunktion wird aufgerufen.
   6.1. Das zum Job gehörende Paket wird vom Worker-Thread geholt.
   6.2. Das Paket wird geparst und bearbeitet.
   6.3. Sobald das Paket nicht mehr benötigt wird, kann es zerstört werden.
7. Zum Schluss wird der Job zerstört und der Worker-Thread beginnt wieder bei Schritt 2.

Die weitere Verarbeitung ist abhängig vom Typ des eingelesenen Paketes. Ist es für eine bestehende IKE_SA bestimmt, so verläuft die Abarbeitung gemäss 5.6.8.

5.6.7 Aufbauen einer Verbindung

Dem Dokument „Technologien“ ist zu entnehmen, dass eine IKE_SA mit einem IKE_SA_INIT- und anschliessenden IKE_AUTH-Austausch aufgebaut wird. Um in unsere Implementierung den Aufbau einer IKE_SA für eine bestimmte Verbindung zu initialisieren, muss ein spezieller Job erstellt werden, welcher dann von einem Worker-Thread bearbeitet wird. Daraus ergibt sich folgendes Sequenzdiagramm:

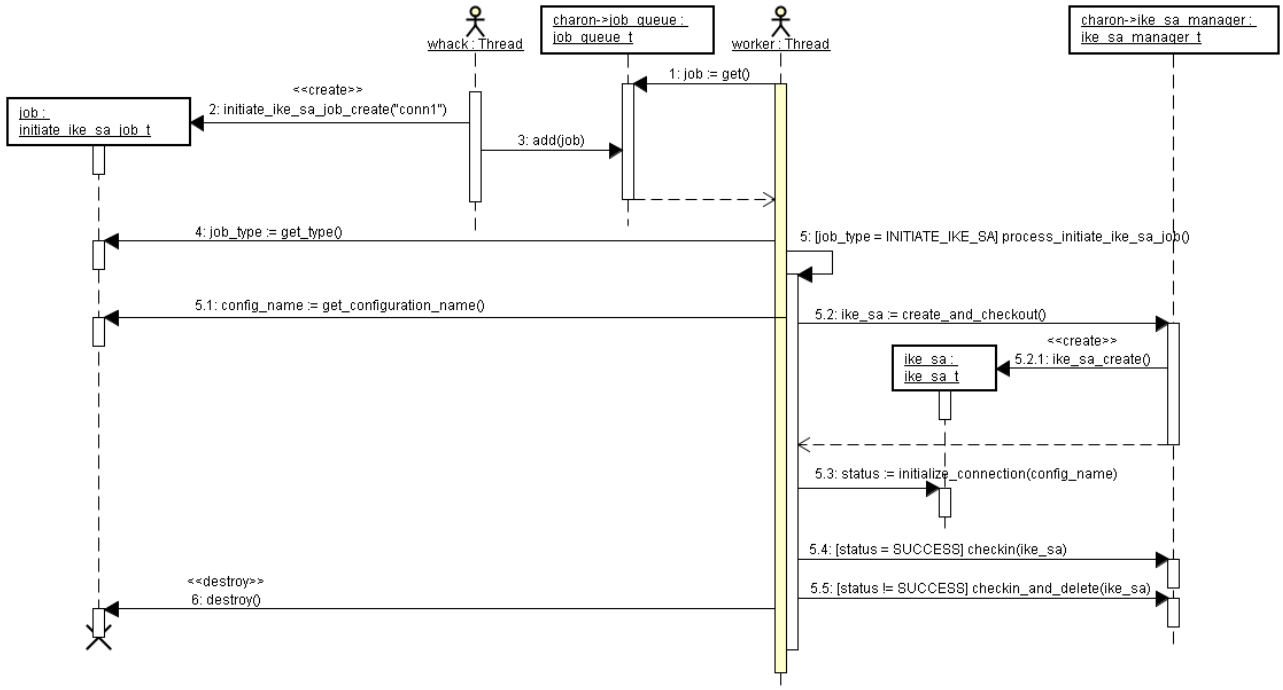


Abbildung 49: Ablauf beim Aufbauen einer Verbindung

Im Sequenzdiagramm ist ein Whack-Thread eingezeichnet. Dieser existiert zur Zeit noch nicht. Er soll lediglich eine Schnittstelle symbolisieren, über welche zur Laufzeit mit dem Daemon kommuniziert werden kann. Mögliche Trigger zum Aufbau einer IKE_SA sind beispielsweise Befehle eines Administrators, der über die Konsole eingegeben wurden oder ein Signal des Kernels.

Die Schritte aus dem obigen Sequenzdiagramm haben folgende Funktionen:

1. Ein Worker-Thread hat keine Arbeit und wartet deshalb so lange, bis ihm die globale Job-Queue einen Job zuspielt.
2. Über einen Trigger wird dem Daemon mitgeteilt, dass dieser eine bestimmte Verbindung starten soll. Dazu wird ein Job vom Typ `initiate_ike_sa_job_t` erstellt und dem Konstruktor die Identifikation der bestimmten Verbindung angegeben.
3. Der neue Job wird nun in die Job-Queue eingereiht und somit für die Bearbeitung durch einen Worker-Thread freigegeben.
4. Der bearbeitende Worker-Thread informiert sich, was für eine Art Job er erhalten hat.
5. Da es sich um ein Job zur Initiierung einer Verbindung handelt wird die interne Funktion zur Abarbeitung dieses Job-Typs aufgerufen.
   5.1. Vom `job_t`-Objekt wird die Identifikation der aufzubauenden Verbindung abgerufen.
   5.2. Beim globalen IKE_SA-Manager wird eine neues Objekt vom Typ `ike_sa_t` ausgecheckt, welches eine IKE_SA abstrahiert.
   5.3. Das ausgecheckte `ike_sa_t`-Objekt wird nun angewiesen die entsprechende Verbindung zu initiieren. Das Objekt kümmert sich nun darum, dass der erste IKE_SA_INIT-Request an den richtigen Empfänger gesendet wird.
   5.4. Konnte die Initiierung erfolgreich durchgeführt und damit das erste IKE_SA_INIT-Paket gesendet werden, wird die IKE_SA wieder eingecheckt.
   5.5. Bei erfolgloser Initiierung der Verbindung aufgrund einer falschen Identifikation oder eines anderen Problems wird die IKE_SA eingecheckt und anschliessend vom IKE_SA-Manager zerstört.
6. Der Worker-Thread hat den Job bearbeitet und kann diesen nun zerstören.

Im Sequenzdiagramm sieht man, dass das neue `ike_sa_t`-Objekt damit beauftragt wird, eine Verbindung aufzubauen.

5.6.8 IKEv2-Message verarbeiten

Empfangene IKEv2-Messages werden durch einen Worker-Thread abgearbeitet. Dieser erledigt die Arbeit nicht direkt, sondern über die zuständige IKE_SA. Das nachfolgende Sequenzdiagramm zeigt den Ablauf beim Verarbeiten einer empfangenen Nachricht. Es wird angenommen, dass der Receiver-Thread bereits ein Job vom Typ INCOMING_PACKET erstellt und in die Job-Queue eingereiht hat. Auch wird angenommen, dass bereits eine IKE_SA existiert, um die entsprechende Message zu verarbeiten.

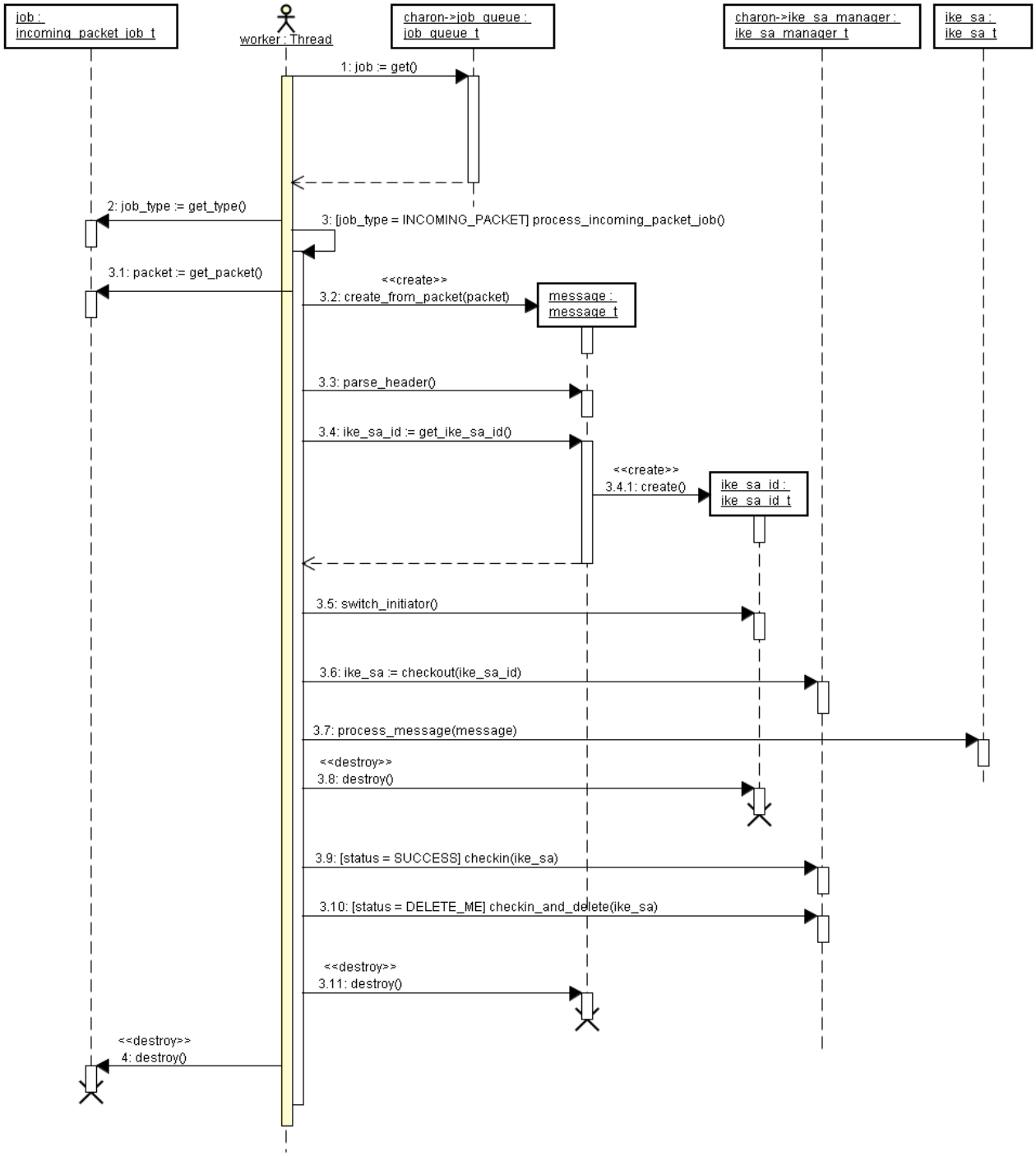


Abbildung 50: Ablauf beim Verarbeiten einer IKEv2-Message

Die Schritte aus dem obigen Sequenzdiagramm werden nun genauer erläutert:

1. Ein Worker-Thread hat keine Arbeit und wartet deshalb so lange, bis ihm die globale Job-Queue einen Job zuspielt.
2. Der bearbeitende Worker-Thread informiert sich, was für eine Art Job er zur Verarbeitung erhalten hat.
3. Da es sich um ein Job zur Verarbeitung eines empfangenen Pakets handelt, wird die interne Funktion zur Abarbeitung dieses Job-Typs aufgerufen.
   3.1. Das dem Job angehängte `packet_t`-Objekt wird gelesen.
   3.2. Aus dem `packet_t`-Objekt wird ein `message_t`-Objekt erzeugt.
   3.3. Der Header der Message wird geparst.
   3.4. Die Message hat nach dem Parsen des Headers genügend Informationen um daraus ein Objekt vom Typ `ike_sa_id_t` zu erstellen. Dieses Identifiziert die dazugehörige IKE_SA.
   3.5. Das „Initiator“-Flag wird gewechselt, da lokal alle IKE_SA-Identifikationen aus Sicht des Daemons selbst gespeichert werden.
   3.6. Mit der Identifikation wird die dazugehörige IKE_SA ausgecheckt. Die Funktion blockiert so lange, bis die entsprechende IKE_SA durch den aktuellen Worker-Thread exklusiv verwendet werden kann.
   3.7. Die ausgecheckte IKE_SA wird mit dem Abarbeiten der Message beauftragt.
   3.8. Das Identifikations-Objekt wird nicht mehr benötigt.
   3.9. Falls die Verarbeitung der Message erfolgreich war, liefert die zuständige IKE_SA den Status `SUCCESS` zurück. In diesem Fall wird die IKE_SA beim Manager eingecheckt.
   3.10. Falls die Verarbeitung der Message nicht erfolgreich war, und die zuständige IKE_SA aus diesem Grund nicht mehr weiter existieren soll, liefert sie den Status `DELETE_ME` zurück. In diesem Fall wird die IKE_SA beim Manager eingecheckt und gelöscht.
   3.11. Das `message_t`-Objekt wird nicht mehr benötigt.
4. Der Job ist abgearbeitet und kann nun gelöscht werden. Der Worker-Thread beginnt jetzt wieder bei Schritt 1.

Das Sequenzdiagramm zeigt, dass eine empfangen Message immer durch die zuständige IKE_SA verarbeitet wird. Dies hat den Vorteil, dass der Worker-Thread überschaubar bleibt und die Verantwortlichkeiten abgegrenzt sind.

6 Tests

6.1 Modultests

Um die Funktionalität der einzelnen Klassen zu gewährleisten, werden diese automatisiert getestet. Für jede wichtigere Klasse wird eine oder mehrere Testfunktionen definiert, welche das Verhalten der Klasse überprüfen. Dazu wird die dafür entwickelte Klasse `tester_t` verwendet. Diese prüft definierte Bedingungen und fasst die ermittelten Resultate zusammen.

Eine Testfunktion hat dabei folgenden Aufbau:

```
void test_example(protected_tester_t *tester)
{
     u_int8_t input, result, excepted;

     /* Erstelle Objekt für zu testende Klasse */
     example_t *example = example_create();

     /* Definiere Test- und Referenz-Werte */
     excepted = 7;
     input = 3;

     /* Führe Operation durch, deren Verhalten getestet werden soll */
     result = example->operation(example, input);

     /* Ob das Verhalten der Methode „operation“ richtig ist, wird über den
      * Tester geprüft. Er sammelt die Resultate aller tests und kann diese
      * auswerten und daraus einen Bericht erstellen.
      */
     tester->assert_true(tester, (result == excepted), „operation() using 3“);

     /* Damit über den Allocator Memory-Leaks erkannt werden, wird
      * sauber augeräumt. So kann jede Klasse auch auf Memory-Leaks getestet
      * werden.
      */
     example->destroy(example);
}
```

Dieses Beispiel zeigt eine Funktion, welche die Methode `operation()` der Klasse `example_t` testet. Einer Instanz der Klasse `tester_t` können beliebige solche Funktionen übergeben werden. `tester_t` ruft diese Funktionen der Reihe nach auf und übergibt sich selber als Parameter. So kann die Testfunktion auf die Methoden `assert_*()` zugreifen, um Bedingungen zu prüfen.

Die Instanz von `tester_t` fasst in einem Bericht zusammen, welche Tests erfolgreich waren und welche nicht.

Alle Testfunktionen sind im Verzeichnis `testcases` zu finden. Die darin enthaltene Datei `testcases.c` fasst alle Tests zusammen und führt diese mit Hilfe der Klasse `tester_t` durch. Das Makefile erstellt ein ausführbares Programm `run_tests` im Verzeichnis `bin`, welches die Tests abarbeitet.

6.2 Systemtests

Modultests testen nur das Verhalten einzelner Klassen. Um jedoch das Verhalten des kompletten Daemons zu testen, sind zusätzlich Systemtests notwendig. Die nachfolgenden Systemtests entsprechen typischen Situationen, die während des Betriebes des Daemons auftreten. Zur besseren Übersichtlichkeit wird zwischen den beiden Austauschen IKE_SA_INIT und IKE_AUTH unterschieden. Unter 6.2.2.5 sind weitere Szenarien aufgelistet, welche getestet wurden, aber hier nicht detailliert beschrieben sind.

Die Tests wurden für Initiator und Responder mit der gleichen Version von `charon` durchgeführt. Die hier beschriebenen Tests sind auf einer einzigen Maschine durchführbar indem für den Zielhost die IP-Adresse `127.0.0.1` angegeben wird. Die entsprechende Konfiguration muss im Quellcode der Datei `config/configuration_manager.c` erfolgen, da noch keine Konfigurations-Schnittstelle implementiert ist. In dieser Datei sind bereits Konfigurationen enthalten, von denen einige zum Selbsttest verwendet werden können. Für die einzelnen Systemtests ist jeweils vermerkt, was für eine Konfiguration zum Selbsttest verwendet werden kann.

Um mit einer spezifischen Konfiguration eine Verbindung zu starten, muss der Daemon mit dem Namen der Verbindung als Parameter gestartet werden. Diese wird anschliessend nach zwei Sekunden aufgebaut. Folgender Befehl startet beispielsweise die Verbindung `localhost-rsa`:

```
bin/charon localhost-rsa
```

Die Log-Ausgabe erfolgt auf der Konsole. Der verwendete Loglevel kann in der Datei `daemon.h` angepasst werden, um beispielsweise die ausgehandelten Schlüssel anzuzeigen.

6.2.1 Meldungsaustausch IKE_SA_INIT

6.2.1.1 Erfolgreicher Austausch

Beschreibung	Host A initiiert eine IKE_SA und sendet dazu ein IKE_SA_INIT-Request an Host B. Dieser wählt ein Proposal aus und sendet die IKE_SA_INIT-Response mit entsprechendem Proposal an Host A zurück.
Erwartete Resultate	– Beide Hosts haben den gleichen gemeinsamen Schlüssel aus den Diffie-Hellman-Werten berechnet. – Beide Hosts haben die gleichen Schlüssel für Verschlüsselung, Signierung, etc. abgeleitet. – Host A hat einen IKE_AUTH-Request an Host B gesendet.
Durchführung	– Auf beiden Hosts eine Konfiguration für den anderen Host mit gleichen Proposals erstellen. – Auf Host B den Daemon ohne Parameter starten. – Auf Host A den Daemon mit Angabe des Konfigurationsnamens für Host B starten.
Resultate	– Aus den Logs von Host A und B können die gleichen Schlüssel entnommen werden. – Mit dem Netzwerksniffer ist zu erkennen, dass Host A einen IKE_AUTH-Request an Host B gesendet hat.
Selbstest	Dieser Test kann durch starten des Daemons mit dem Parameter `localhost-rsa` auch auf der eigenen Maschine durchgeführt werden. Die Konfiguration `localhost-rsa` erlaubt den kompletten Aufbau einer IKE_SA mit der Verwendung von RSA-Schlüsseln.

Tabelle 26: Systemtest "IKE_SA_INIT: Erfolgreicher Austausch"

6.2.1.2 Falsche Diffie-Hellman-Gruppe

Beschreibung	Host A initiiert eine IKE_SA und sendet dazu ein IKE_SA_INIT-Request an Host B. Dieser wählt ein Proposal aus, welches jedoch eine andere Diffie-Hellman-Gruppe beinhaltet wie die vom Initiator im IKE_SA_INIT-Request gewählte.
Erwartete Resultate	– Host A sendet einen erneuten IKE_SA_INIT-Request mit anderer Diffie-Hellman-Gruppe. – Host B wählt beim zweiten Mal ein Proposal mit der gleichen Diffie-Hellman-Gruppe wie im IKE_SA_INIT-Request. – Beide Hosts haben den gleichen geimeinsamen Schlüssel aus den Diffie-Hellman-Werten berechnet. – Beide Hosts haben die gleichen Schlüssel für Verschlüsselung, Signierung, etc. abgeleitet. – Host A hat einen IKE_AUTH-Request an Host B gesendet.
Durchführung	– Auf Host A werden zwei Proposals mit unterschiedlicher Diffie-Hellman-Gruppe konfiguriert. – Auf Host B wird nur ein Proposal konfiguriert, welches aber dem zweiten Proposal von Host A entspricht. – Auf Host B den Daemon ohne Parameter starten. – Auf Host A den Daemon mit Angabe des Konfigurationsnamens für Host B starten.
Resultate	– Aus den Logs ist zu entnehmen, dass der Erste IKE_SA_INIT-Request mit einer Notify-Response zurückgesendet wird und Host A daraufhin ein neuer IKE_SA_INIT-Request startet. – Dieses Verhalten ist auch mit dem Netzwerksniffer nachzuvollziehen. Insgesamt kommt es zu 4 Meldungen vom Typ IKE_SA_INIT. – Aus den Logs von Host A und B können die gleichen Schlüssel entnommen werden. – Mit dem Netzwerksniffer ist zu erkennen, dass Host A einen IKE_AUTH-Request an Host B gesendet hat.
Selbstest	Dieser Test kann durch starten des Daemons mit dem Parameter `localhost-bad_dh_group` auch auf der eigenen Maschine durchgeführt werden. Diese Konfiguration erlaubt den kompletten Aufbau einer IKE_SA mit Aushandeln der Diffie-Hellman-Gruppe in der IKE_SA_INIT-Phase.

Tabelle 27: Systemtest "IKE_SA_INIT: Falsche Diffie-Hellman-Gruppe"

6.2.1.3 Paketverlust

Beschreibung	Host A initiiert eine IKE_SA und sendet dazu ein IKE_SA_INIT-Request an Host B. Dieser Request geht jedoch verloren und muss deshalb neu gesendet werden.
Erwartete Resultate	– Host A sendet den IKE_SA_INIT-Request erneut. – Beide Hosts haben das gleiche Shared-Secret aus den Diffie-Hellman-Werten berechnet. – Beide Hosts haben die gleichen Schlüssel für Verschlüsselung, Signierung, etc. abgeleitet. – Host A hat einen IKE_AUTH-Request an Host B gesendet.
Durchführung	– Auf beiden Hosts eine Konfiguration für den anderen Host mit gleichen Proposals erstellen. – Auf Host A den Daemon mit Angabe des Konfigurationsnamens für Host B starten. – Nach ein paar Sekunden auf Host B den Daemon ohne Parameter starten.
Resultate	– Host A sendet den IKE_SA_INIT-Request nach einem bestimmte Timeout erneut. Bei jedem erneuten Senden verdoppelt sich der Timeout. – Aus den Logs von Host A und B können die gleichen Schlüssel entnommen werden. – Mit dem Netzwerksniffer ist zu erkennen, dass Host A einen IKE_AUTH-Request an Host B gesendet hat.
Selbstest	Dieser Test kann nur auf derselben Maschine durchgeführt werden, falls zwei Daemons mit unterschiedlichem UDP Port kompiliert und die entsprechenden Konfigurationen angepasst sind.

Tabelle 28: Systemtest "IKE_SA_INIT: Paketverlust"

6.2.2 Meldungsaustausch IKE_AUTH

6.2.2.1 Shared-Secret

Beschreibung	Host A initiiert eine IKE_SA und sendet nach erfolgreichem IKE_SA_INIT-Austausch ein IKE_AUTH-Request an Host B. Die beiden Kommunikationspartner authentisieren sich gegenseitig mit einem gemeinsamen Passwort.
Erwartete Resultate	– Host A sendet nach erfolgreichem IKE_SA_INIT-Austausch ein IKE_AUTH-Request an Host B. – Host B authentisiert Host A und sendet eine IKE_AUTH-Response an Host A zurück. – Host A authentisiert Host B. – Beide haben eine IKE_SA aufgebaut und sich gegenseitig authentisiert.
Durchführung	– Auf beiden Hosts eine Konfiguration für den anderen Host mit gleichen CHILD_SA-Proposals erstellen. – ID des Kommunikationspartners konfigurieren. – Shared-Secret für entsprechende ID konfigurieren. – SHARED_KEY_MESSAGE_INTEGRITY_CODE als eigene Authentisierungsmethode konfigurieren. – Auf Host B den Daemon ohne Parameter starten. – Auf Host A den Daemon mit Angabe des Konfigurationsnamens für Host B starten.
Resultate	– Aus den Logs von Host A und B ist zu entnehmen, dass die Authentisierung funktioniert hat. – Mit dem Netzwerksniffer ist zu erkennen, dass der IKE_AUTH-Austausch verschlüsselt abläuft.
Selbstest	Dieser Test kann durch starten des Daemons mit dem Parameter `localhost-shared` auch auf der eigenen Maschine durchgeführt werden. Die Konfiguration `localhost-shared` erlaubt den kompletten Aufbau einer IKE_SA mit der Verwendung eines gemeinsamen Passworts.

Tabelle 29: Systemtest "IKE_AUTH: Shared-Secret"

6.2.2.2 Falsches Shared-Secret

Beschreibung	Host A initiiert eine IKE_SA und sendet nach erfolgreichem IKE_SA_INIT-Austausch ein IKE_AUTH-Request an Host B. Die beiden Kommunikationspartner können sich nicht authentisieren, da das gemeinsame Passwort nicht übereinstimmt.
Erwartete Resultate	– Host A sendet nach erfolgreichem IKE_SA_INIT-Austausch ein IKE_AUTH-Request an Host B. – Entweder kann Host A oder Host B nicht authentisiert werden. – Falls Host A nicht authentisiert werden konnte, sollte Host B eine IKE_AUTH-Response mit Notify-Payload vom Typ AUTHENTICATION_FAILED zurücksenden.
Durchführung	– Auf beiden Hosts eine Konfiguration für den anderen Host mit gleichen CHILD_SA-Proposals erstellen. – ID des Kommunikationspartners konfigurieren. – Shared-Secret für entsprechende ID falsch konfigurieren. – SHARED_KEY_MESSAGE_INTEGRITY_CODE als eigene Authentisierungsmethode konfigurieren. – Auf Host B den Daemon ohne Parameter starten. – Auf Host A den Daemon mit Angabe des Konfigurationsnamens für Host B starten.
Resultate	– Aus den Logs von Host A und B ist zu entnehmen, dass die Authentisierung nicht funktioniert hat. – Mit dem Netzwerksniffer ist zu erkennen, dass der IKE_AUTH-Austausch verschlüsselt abläuft.
Selbstest	Für diesen Test gibt es keine Konfiguration zum Selbsttest.

Tabelle 30: Systemtest "IKE_AUTH: Falsches Shared-Secret"

6.2.2.3 RSA

Beschreibung	Host A initiiert eine IKE_SA und sendet nach erfolgreichem IKE_SA_INIT-Austausch ein IKE_AUTH-Request an Host B. Der initiierende Endpunkt verwendet RSA für die Authentisierung, der andere verwendet ein Shared-Secret.
Erwartete Resultate	– Host A sendet nach erfolgreichem IKE_SA_INIT-Austausch ein IKE_AUTH-Request an Host B. – Host B authentisiert Host A und sendet eine IKE_AUTH-Response an Host A zurück. – Host A authentisiert Host B. – Beide haben eine IKE_SA aufgebaut und sich gegenseitig authentisiert.
Durchführung	– Auf beiden Hosts eine Konfiguration für den anderen Host mit gleichen CHILD_SA-Proposals erstellen. – ID des Kommunikationspartners konfigurieren. – RSA-Schlüssel für entsprechende ID konfigurieren. – RSA_DIGITAL_SIGNATURE als eigene Authentisierungsmethode konfigurieren. – Auf Host B den Daemon ohne Parameter starten. – Auf Host A den Daemon mit Angabe des Konfigurationsnamens für Host B starten.
Resultate	– Aus den Logs von Host A und B ist zu entnehmen, dass die Authentisierung funktioniert hat. – Mit dem Netzwerksniffer ist zu erkennen, dass der IKE_AUTH-Austausch verschlüsselt abläuft.
Selbstest	Dieser Test kann durch starten des Daemons mit dem Parameter `localhost-rsa` auch auf der eigenen Maschine durchgeführt werden. Die Konfiguration `localhost-rsa` erlaubt den kompletten Aufbau einer IKE_SA mit der Verwendung eines RSA-Schlüssels zur Authentisierung.

Tabelle 31: Systemtest "IKE_AUTH: RSA"

6.2.2.4 Falscher RSA-Schlüssel

Beschreibung	Host A initiiert eine IKE_SA und sendet nach erfolgreichem IKE_SA_INIT-Austausch ein IKE_AUTH-Request an Host B. Ein Endpunkt kann nicht erfolgreich authentisiert werden, da der Schlüssel falsch ist.
Erwartete Resultate	– Host A sendet nach erfolgreichem IKE_SA_INIT-Austausch ein IKE_AUTH-Request an Host B. – Entweder kann Host A oder Host B nicht authentisiert werden. – Falls Host A nicht authentisiert werden konnte, sollte Host B eine IKE_AUTH-Response mit Notify-Payload vom Typ AUTHENTICATION_FAILED zurücksenden.
Durchführung	– Auf beiden Hosts eine Konfiguration für den anderen Host mit gleichen CHILD_SA-Proposals erstellen. – ID des Kommunikationspartners konfigurieren. – RSA-Schlüssel für entsprechende ID falsch konfigurieren. – RSA_DIGITAL_SIGNATURE als eigene Authentisierungsmethode konfigurieren. – Auf Host B den Daemon ohne Parameter starten. – Auf Host A den Daemon mit Angabe des Konfigurationsnamens für Host B starten.
Resultate	– Aus den Logs von Host A und B ist zu entnehmen, dass die Authentisierung nicht funktioniert hat. – Mit dem Netzwerksniffer ist zu erkennen, dass der IKE_AUTH-Austausch verschlüsselt abläuft.
Selbstest	Für diesen Test gibt es keine Konfiguration zum Selbsttest.

Tabelle 32: Systemtest "IKE_AUTH: Falscher RSA-Schlüssel"

6.2.2.5 Weitere getestete Szenarien

Neben den beschriebenen Systemtests wurden natürlich weit mehr Szenarien getestet. Die folgende Auflistung ist nicht abschliessend und soll eine Vorstellung geben, was für Szenarien sonst noch getestet wurden:

- Initiieren einer IKE_SA mit falscher Versionsnummer
- Antworten mit Proposals, welche nicht vorgeschlagen wurden
- Senden von unverschlüsselten IKE_AUTH-Meldungen
- Vorschlagen von nicht konfigurierten Proposals
- Verwenden von nicht konfigurierten Identitäten
- Senden einer Response, obwohl kein Request versendet wurde
- Verarbeiten einer IKE_SA_INIT-Anfrage eines nicht konfigurierten Hosts
- usw.

6.3 Kompatibilitätstest

Um Fehlinterpretationen des Drafts von IKEv2 zu erkennen ist es unumgänglich, Kompatibilitätstests zu bestehenden Implementierungen durchzuführen. Folgende frei zugängliche Projekte befassen sich zurzeit mit IKEv2:

- IKEv2 (siehe [IKEv2Linux]): IKEv2-Daemon für Linux und andere Unix.
- Racoon2 (siehe [Racoon2]): Nachfolger von Racoon, Key-Daemon des KAME Projektes für IKEv2.

Ersteres ist noch in der Anfangsphase und noch instabil. Für Kompatibilitätstests ist es allerdings geeignet, da die Installation und Konfiguration unter Linux nicht allzu schwierig ist.

Racoon2 scheint etwas weiter zu sein, läuft allerdings unter BSD. Die Installation und Konfiguration hat sich als zu komplex und zeitaufwendig erwiesen. Deshalb wurde entschieden, auf Kompatibilitätstests mit diesem Daemon zu verzichten.

6.3.1 Meldungsaustausch IKE_SA_INIT

Um den ersten Meldungsaustausch erfolgreich abschliessen zu können, sind folgende Kriterien zu erfüllen:

Kriterium	Status
Generieren und Parsen von: Header, SA Payload, Nonce Payload, KE Payload	Durch die Logs ist ersichtlich, dass die andere Implementierung unsere Payloads erfolgreich parsen kann. Ebenso funktioniert dies umgekehrt einwandfrei.
Zusammenstellung und Auswählen von Proposals	Durch Sniffen des Netzwerkverkehrs wird ersichtlich, dass das Zusammenstellen sowie das Auswählen der Proposals funktioniert. Durch die Limitation, dass unsere Implementierung nur einen Algorithmus pro Typ und Protokoll in einem Proposal unterstützt, muss die Konfiguration der anderen Implementierung darauf abgestimmt sein (mehr zu dieser Limitation ist dem Dokument „Projektstand“ zu entnehmen).

Tabelle 33: Kompatibilitätstest "Meldungsaustausch IKE_SA_INIT"

Mit den gegebenen Einschränkungen funktioniert der Austausch soweit, dass darauf der nächste Austausch IKE_AUTH erfolgen kann.

6.3.2 Meldungsaustausch IKE_AUTH

Der Austausch des zweiten Meldungspaares IKE_AUTH bringt folgende zusätzliche Kriterien mit sich:

Kriterium	Status
Herleitung des gemeinsamen Schlüssels aus Diffie-Hellman-Austausch	Aus den Logs ist ersichtlich, dass der Schlüssel identisch ist.
Ableiten der Schlüssel	In den Logs ist zu erkennen, dass die Schlüsselgenerierung mit PRF_HMAC_SHA1 sowie PRF_HMAC_MD5 auf beiden Implementierungen erfolgreich abläuft.
Encrypted Payload ver-/ entschlüsseln, signieren/verifizieren	Aus den Logs ist ersichtlich, dass unsere Verschlüsselung mit AES entschlüsselt werden kann, ebenso können wir verschlüsselte Nachrichten des Anderen entschlüsseln. Das Signieren und Verifizieren funktioniert mit AUTH_HMAC_MD5_96 als auch mit AUTH_HMAC_SHA1_96.
Generieren und parsen von: ID Payload, AUTH Payload, TS Payload	Durch die Logs ist ersichtlich, dass die andere Implementierung unsere Payloads erfolgreich parsen kann. Ebenso funktioniert dies umgekehrt einwandfrei.
Zusammenstellen und Auswählen von Proposals	Wie beim Meldungsaustausch IKE_SA_INIT funktioniert das Zusammenstellen und Auswählen der Proposals. Allerdings unterliegt dies derselben Limitation, so dass die Konfigurationen noch aufeinander abgestimmt werden müssen.
Authentisierung über Shared-Secret	Die Authentisierung über Shared-Secret funktioniert nicht ohne weiteres. Dies liegt daran, dass die Gegenseite falsche Daten zur Signaturbildung verwendet. Die dazu Verwendete ID Payload muss gemäss [IKEv2Draft] 2.15 unserer Auffassung nach die reservierten Bytes ebenfalls enthalten. Entfernen wir hingegen diese Bytes aus der Signaturbildung, so funktioniert die Authentisierung beidseitig. Die andere Implementierung zeigt allerdings danach ein Fehlverhalten, welches zum Absturz der Software führt. Die näheren Gründe dazu wurden nicht ermittelt, da dies nicht ein Problem unserer Seite zu sein scheint.
Authentisierung über RSA	RSA konnte nicht mit der Referenz-Implementierung getestet werden, da diese RSA derzeit nicht unterstützt.

Tabelle 34: Kompatibilitätstest "Meldungsaustausch IKE_AUTH"

Bei Beachtung der erwähnten Kriterien ist der Austausch möglich, so dass eine authentisierte IKE_SA erstellt werden kann.

7 Projektstand

In der Aufgabenstellung dieser Diplomarbeit wurden keine bestimmten Leistungsmerkmale festgelegt, die es zu erreichen galt. Ziel war es, vor allem eine saubere Architektur zu Implementieren, auf welcher weiter aufgebaut werden kann.

Um trotzdem den Rahmen der Arbeit zu definieren, wurde die Unterstützung der ersten beiden Meldungsaustausche angestrebt. Dies ermöglicht es, das IKEv2-Protokoll in einer absolut minimalen Form zu erfüllen. Die minimalen Anforderungen an eine IKEv2-Implementierung sind im Draft jeweils mit einem MUST-Keyword vermerkt. Zusätzlich existiert im Draft ein separater Abschnitt „Conformance Requirements“, welcher weitere Bedingungen für eine IKEv2-Unterstützung listet.

Dieses Dokument fasst den Stand der entwickelten IKEv2-Implementierung zusammen.

- Im Abschnitt 7.1 werden die erreichten Resultate mit den minimalen Anforderungen an eine IKEv2-Implementierungen verglichen.
- Der Abschnitt 7.2 beschreibt die nächsten Schritte, die dringend realisiert werden sollten.
- Im Abschnitt 7.3 ist ein Vorschlag einer Roadmap für strongSwan II enthalten. Sie zeigt die Entwicklung während der Diplomarbeit auf und gibt einen Weg für die Weiterentwicklung des Projektes vor, welcher uns sinnvoll erscheint.
- Der Abschnitt 7.4 zieht die Schlüsse aus dem Erreichten.

7.1 IKEv2-Kompatibilität

Die nachfolgende Tabelle listet die Features, die von einer IKEv2-Implementierung im Minimum unterstützt werden müssen, damit sie als IKEv2-konform gilt. Sie stammen aus dem [IKEv2Draft] und entsprechen entweder Anforderungen mit dem MUST-Keyword[1] oder sind dem Abschnitt „Conformance Requirements“ entnommen.

Anforderung	Status	Bemerkung
Alle Payload-Typen müssen geparst werden können. Nicht unterstützte Payloads dürfen übersprungen werden, so lange das „critical“-Flag nicht gesetzt ist. Bei nicht unterstützten Payloads mit gesetztem „critical“-Flag muss die entsprechende Message abgelehnt werden.	unterstützt	Payloads mit einem Typ aus dem PRIVATE USE-Space werden als Objekt vom Typ unknown_payload_t erstellt und auch geparst. Falls dabei das „critical“-Flag gesetzt ist, bricht der Parser ab. Alle IKEv2-Payloads werden durch den Parser unterstützt, können aber je nach Message-Typ trotzdem verworfen werden. So ist beispielsweise im IKE_SA_INIT-Austausch kein Payload vom Typ AUTH möglich.
Alle Implementierungen müssen die ersten zwei Austausche IKE_SA_INIT und IKE_AUTH unterstützen und dabei zwei SAs aufbauen (eine für IKE und eine für ESP und/oder AH). Eine Implementierung kann dabei nur die Rolle als Responder oder Requester unterstützen.	teilweise unterstützt	Diese Implementierung unterstützt die Austausche IKE_SA_INIT und IKE_AUTH als Initiator und als Responder. Am Ende des IKE_AUTH-Austausches ist die IKE_SA komplett erstellt. Aufgrund der noch fehlenden Kernelanbindung können noch keine CHILD_SAs eingerichtet werden.
Alle Implementierungen müssen fähig sein, auf ein INFORMATIONAL-Request zu antworten. Eine minimale Implementierung kann diese Anforderung erfüllen, indem sie immer eine leere Response zurück sendet (Leer heisst hier, dass der verschlüsselte Payload keine Payloads enthält).	unterstützt	Die Unterstützung wurde in minimaler Form umgesetzt. Verarbeitet wird lediglich ein INFORMATIONAL-Austausch für das Löschen von IKE_SAs. Für alle anderen Meldungen vom Typ INFORMATIONAL wird keine Aktion ausgeführt und eine leere Response zurück gesendet.
Unterstützung von PKIX-Zertifikaten, welche mit RSA-Schlüsseln einer Länge von 1024 oder 2048 Bit unterschrieben wurden. Dabei müssen die ID-Typen ID_KEY_ID, ID_FQDN, ID_RFC822_ADDR oder ID_DER_ASN1_DN möglich sein.	nicht unterstützt	Zertifikate werden noch nicht unterstützt. Die Authentisierung über RSA ist jedoch bereits unterstützt, solange die Gegenseite im Besitz des Public-Keys ist.
Unterstützung der Shared-Secret Authentisierung wobei die ID vom Typ ID_KEY_ID, ID_FQDN oder ID_RFC822_ADDR ist.	unterstützt	Die Shared-Secret Authentisierung ist in die Implementierung eingebaut. Unterstützt werden dabei alle Arten von IDs wobei jedoch nur der ID-Typ ID_IPV4_ADDR aus einem String erstellt werden kann.
Unterschiedliche Authentisierung der Kommunikationspartner, wobei der Responder mit einem PKIX-Zertifikat und der Initiator über ein Shared-Secret authentisiert wird.	unterstützt	Die unterschiedliche Authentisierung der Kommunikationspartner ist unterstützt, wobei einer mit Shared-Secret und der andere mit rohem RSA-Key authentisieren kann.
Die Reihenfolge der Meldungstypen muss stets eingehalten werden. Ein IKE_AUTH-Austausch erfolgt immer nach einem IKE_SA_INIT-Austausch und die Austausche INFORMATIONAL und CREATE_CHILD_SA erfolgen erst nach erfolgreichem IKE_AUTH-Austausch.	unterstützt	Die Reihenfolge der Austausche wird eingehalten. Ein nicht erlaubter Message-Typ wird nicht verarbeitet und verworfen.
Der Empfänger einer IKE_AUTH-Message muss überprüfen, dass alle Signaturen oder MACs gültig sind und die dazu verwendeten Schlüssel mit dem entsprechenden ID Payload übereinstimmen.	unterstützt	Bei Verwendung eines Shared-Secrets wird der MAC überprüft, bei RSA die Signatur.
Falls eine IKE_SA gelöscht wird, müssen auch die zugehörigen CHILD_SAs gelöscht werden.	nicht unterstützt	Da noch keine CHILD_SAs eingerichtet werden, können diese auch nicht gelöscht werden.

1 Es sind nicht alle Anforderungen hier gelistet, welche im Draft das MUST-Keyword beinhalten.

Anforderung	Status	Bemerkung
Alle Implementierungen müssen fähig sein, UDP-Pakete bis 1280 Byte zu empfangen oder zu senden.	unterstützt	Die Implementierung stellt kein Limit und lässt auch die Verarbeitung von längeren Paketen zu (beim Parsen und beim Generieren). Das Limit hängt von den Einstellungen des Betriebssystems ab.
Der Initiator muss einen gesendeten Request zwischenspeichern bis er die Antwort darauf erhalten hat.	unterstützt	Der letzte Request wird gespeichert und nach Ablauf des Retransmit-Timers nochmals gesendet. Das Paket wird dabei nicht neu generiert und entspricht so immer dem Original.
Der Responder muss eine gesendete Response zwischenspeichern bis er ein gültiger Request mit höherer Message-ID erhalten hat.	unterstützt	Die letzte Response wird gespeichert und beim Erhalt einer Message mit unveränderter ID nochmals gesendet. Das Paket wird dabei nicht neu generiert und entspricht so immer dem Original.
Der Initiator muss ein Request neu senden, falls keine Response nach einem bestimmten Timeout erhalten wurde.	unterstützt	Der letzte Request wird gespeichert und nach Ablauf des Retransmit-Timers nochmals gesendet.
Die Retransmit-Zeiten müssen exponential zunehmen, um das Netzwerk nicht unnötig zu belasten.	unterstützt	Der Timeout verdoppelt sich nach jedem Retransmit.
Messages mit nicht unterstützter Versionsnummer müssen abgelehnt werden.	unterstützt	Jede Message mit einer anderen `IKE`-Version als 2.0 wird abgelehnt. Handelt es sich um ein IKE_SA_INIT-Request, wird eine Response mit Notify vom Typ INVALID_MAJOR_VERSION zurückgesendet.
Reserved Bits und Bytes müssen beim Senden auf 0 gesetzt und beim empfangen ignoriert werden.	unterstützt	Wird durch Parser und Generator sichergestellt.
Wird eine IKE_SA_INIT-Response mit Cookie-Notify erhalten, muss der Request mit diesem Cookie neu gestartet werden.	nicht unterstützt	Der Aufbau der entsprechenden IKE_SA wird abgebrochen.
Vorgeschlagene Proposals müssen als gesamte akzeptiert oder verworfen werden.	teilweise unterstützt	Die Implementierung unterstützt zur Zeit lediglich Proposals bei denen nur ein Transform pro Transform-Typ enthalten ist. Die Unterstützung von mehreren Transforms des gleichen Typs ist als Erweiterung geplant.
Falls der Responder ein Proposal mit anderer Diffie-Hellman-Gruppe auswählt als jene des Initiators, muss der Initiator den Request mit einer neuen Gruppe starten.	unterstützt	Der Responder sendet in einem solchen Fall ein Notify vom Typ INVALID_KE_PAYLOAD. Der Initiator startet daraufhin einen Request mit neuer DH-Gruppe. Die zurückgelieferte Angabe der DH-Gruppe im Notify-Payload wird zur Zeit nicht verwertet.
Nach überschreiten eines gewissen Timeouts darf eine SA nicht mehr benutzt werden.	unterstützt	Eine aufgebaute IKE_SA wird nach einem gewissen Timeout gelöscht und kann ab da nicht mehr verwendet werden.
Die Nonce-Werte müssen mindestens 16 Byte Länge haben und zufällig gewählt sein. Auch müssen sie mindestens die Hälfte der PRF-Keylänge haben.	unterstützt	Die Länge der Nonce ist als `#define` fest im Code auf 16 Byte definiert, kann aber selbstverständlich erhöht werden.
Requests müssen auch akzeptiert werden, falls diese nicht vom UDP Port 500 oder 4500 stammen.	unterstützt	Der Responder sendet die Response an den Port zurück, von welchem der IKE_SA_INIT-Request gestartet wurde.
Shared-Secrets müssen ASCII-Strings bis zu einer Länge von 64 Byte erlauben.	unterstützt	Die Implementierung macht hier keine Begrenzung.
Der ausgehandelte Schlüssel im `IKE`-Austausch darf nicht für andere Zwecke wiederverwendet werden.	unterstützt	Der ausgehandelte Schlüssel wird nur für die im Draft definierten Zwecke verwendet.
Meldungen müssen vom UDP Port 500 gesendet werden können.	unterstützt	Der Standard-Port ist 500. Zur Zeit wird nur dieser eine Port unterstützt.

Anforderung	Status	Bemerkung
Ein verschlüsselter Payload muss der letzte Payload einer Message sein.	unterstützt	Wird beim Parsen geprüft und beim Generieren sichergestellt.
Die Implementierung muss eine Möglichkeit anbieten, die unterstützten Algorithmen zu managen.	unterstützt	Muss noch im Quellcode des Konfigurations-Managers erfolgen.
Implementierungen müssen die Möglichkeit anbieten eine ID vom Typ ID_IPV4_ADDR, ID_FQDN, ID_RFC822_ADDR oder ID_KEY_ID als Sender zu konfigurieren. Alle diese ID-Typen müssen akzeptiert werden können.	unterstützt	Alle Typen werden akzeptiert. Die Konfiguration mit einem String ist jedoch nur für den Typ ID_IPV4_ADDR möglich.
Konfiguration des IPsec-Stack über eine Kernel-Schnittstelle.	teilweise unterstützt	Die Grundlage der Kernel-Schnittstelle ist vorhanden.

Tabelle 35: Minimalanforderung an eine IKEv2-Implementierung

Die aufgelisteten minimalen Anforderungen an eine IKEv2-Implementierung konnten nicht komplett erfüllt werden. Unsere Implementierung ist somit noch nicht ganz IKEv2-konform. Primär fehlt das Einrichten der CHILD_SAs, welches eine Kommunikation mit dem Kernel erfordert. Des weiteren fehlt eine Unterstützung des Cookie-Mechanismus sowie von PKIX-Zertifikaten.

7.2 Nächste Schritte

Dieser Abschnitt beschreibt die nächsten Schritte, die als dringlich betrachtet werden. Es handelt sich dabei um Funktionen, die aus Zeitmangel nicht mehr während der Diplomarbeit realisiert werden konnten:

Schritt	Beschreibung	Aufwand ca.
Proposal-Wahl	Die Auswahl von Proposals ist vereinfacht implementiert. Es können zwar Proposals ausgewählt werden, allerdings wird nur ein Transform pro Transform-Typ unterstützt. Um diese Limitierung aufzuheben, müssen die Datenstrukturen für Proposals (`ike_proposal_t` und `child_proposal_t`) angepasst werden, damit sie mehrere Transforms pro Typ erlauben. Allenfalls sind diese als Klasse mit Funktionalität zu Implementieren.	4 Manntage
Schlüssel für CHILD_SA	Das Ableiten der Schlüssel für die CHILD_SAs muss Implementiert werden. Dies ist ähnlich gelöst wie für die IKE_SA. Es ist jedoch die Reihenfolge der abzuleitenden Schlüssel entscheidend. Dabei werden die Schlüssel in der Reihenfolge verwendet, wie die Transforms im ausgewählten Proposal auftauchen. Dies wurde beim Design der Datenstruktur `child_proposal_t` vernachlässigt, wodurch deren Implementierung unzureichend ist. Es ist nicht möglich, auf die ursprüngliche Reihenfolge im Payload zu schliessen. Dies ist beim neuen Design der Datenstruktur zu berücksichtigen (siehe Schritt „Proposal-Wahl“).	2 Manntage, sofern Schritt „Proposal-Wahl“ umgesetzt
Kernel-Interface	Damit „charon“ überhaupt eingesetzt werden kann, fehlt noch die Anbindung zum Kernel. Ein Rumpf ist bereits implementiert, in welchem ein Thread eingehende Nachrichten einliest. Ebenfalls sind Funktionen für das Beziehen von SPIs sowie für das Einrichten einer SA als Proof-of-Concept vorhanden.	5 Manntage

Tabelle 36: Nächste Schritte in der Entwicklung von strongSwan II

Die hier genannten nächsten Schritte sehen wir als dringend nötig. Eine mögliche Roadmap über einen grösseren Zeithorizont hinweg ist im nächsten Abschnitt beschrieben.

7.3 Roadmap

Eine Roadmap für `strongSwan II` könnte folgendermassen aussehen:

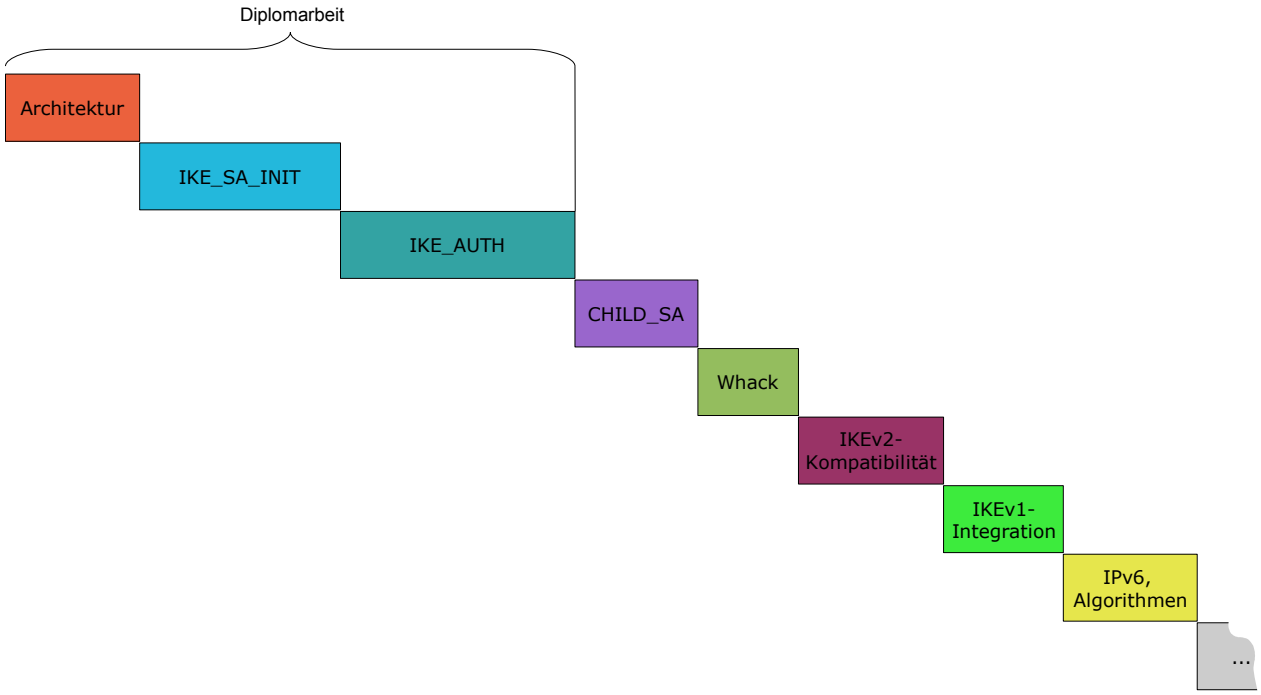


Abbildung 51: Mögliche Roadmap für strongSwan II

Die ersten drei Phasen sind im Rahmen der Diplomarbeit realisiert worden. Dazu gehört das Design der neuen Architektur sowie der Austausch der ersten vier Meldungen. Die weiteren Phasen bestehen aus folgenden Arbeiten:

Phase	Bestandteile
CHILD_SA	In dieser Phase soll die Implementierung von „Nächste Schritte“ (siehe 7.2) erfolgen. Am Ende dieser Phase sollte es möglich sein, erste CHILD_SAs aufzubauen.
Whack	Danach ist es irgendwann notwendig, eine Schnittstelle für die Steuerung von `charon` zu Implementieren. Denkbar wäre die Übernahme des Steuerungstools `whack`. Ebenfalls soll dieses Tool das Lesen von Konfigurationen erlauben.
`IKEv2`-Kompatibilität	In dieser Phase soll die Implementierung der notwendigen Features erfolgen, um `IKEv2`-Kompatibilität zu erreichen (siehe 7.1). Hohe Priorität hat sicher die Unterstützung von X.509-Zertifikaten, eine Stärke von `strongSwan`.
`IKEv1`-Integration	Eine Integration von `pluto` wäre wünschenswert, um auch `IKE` in der Version 1 zu unterstützen. Diese könnte durch eine Anbindung des `pluto`-Daemons über einen speziellen Socket erfolgen.
IPv6, Algorithmen	Des weiteren wäre die Implementation des IPv6-Supports wünschenswert. Beim Entwurf der Architektur wurde stark auf diese Erweiterung Rücksicht genommen, so dass dies nicht allzu umfangreich ausfallen sollte. Ebenfalls sinnvoll wäre die Unterstützung weiterer Algorithmen (siehe [IKEv2Algs]).

Tabelle 37: Phasen einer möglichen Roadmap von strongSwan II

7.4 Schlussfolgerungen

Mit dem neuen Daemon `charon` wurde eine Grundlage für einen modernen `IKEv2`-Daemon als Nachfolger von `strongSwan` geschaffen. Seine Architektur ist übersichtlich und flexibel. Diese Eigenschaften werden durch ein sauberes Threading-Modell und dem objektorientierten Aufbau des Quellcodes erreicht.

Neben der Entwicklung der Architektur wurde auch das Ziel angestrebt, das `IKEv2`-Protokoll in einer minimalsten Form zu unterstützen. Um dies zu erreichen, muss eine Security Association für `IKE` über die ersten vier Meldungen des Protokolls ausgehandelt werden. In diesen Meldungen ist es auch möglich, bereits eine Security Association für IPsec einzurichten. Diese Anforderung kann `charon` momentan noch nicht erfüllen, da unter anderem die Anbindung zum Kernel fehlt. Der Aufbau der SA für `IKE` funktioniert allerdings einwandfrei, wobei bereits die Authentisierung über RSA als auch über ein Shared-Secret möglich ist.

Obwohl das hoch gesteckte Ziel einer `IKEv2`-Kompatibilität nicht ganz erreicht werden konnte, darf die Arbeit als Erfolg gewertet werden. Während der Arbeit hat sich gezeigt, dass die Architektur sauber und zuverlässig ist. Zählt man die Anzahl Funktionen, so kann `charon` gegen andere `IKEv2`-Projekte noch nicht auftrumpfen. Unsere Erfahrungen mit anderen Implementierungen zeigten aber, dass in Sachen Stabilität und Codequalität unsere Software eine Nase voraus ist. Und dies, obwohl in der kurzen Diplomarbeit sehr viel Code entwickelt wurde.

Die Grundlage für `strongSwan II` ist nun vorhanden. Jetzt heisst es, diese mit Funktionalitäten zu erweitern und schlussendlich einen würdigen Nachfolger von `strongSwan I` zu schaffen.

8 Projektmanagement

8.1 Projektplan

8.1.1 Prozessmodell

Für die Implementierung von IKEv2 wird nicht nach einem bestimmten Prozessmodell wie etwa RUP vorgegangen. Die Phasen sind so geplant, wie man es aus dem Wasserfallmodell kennt, indem im Laufe der Arbeit IKEv2-Features hinzukommen und implementiert werden. Am Ende jeder Phase soll der Stand der Arbeit anhand eines Ergebnisses (Dokumentation oder Software) überprüft werden können. In den einzelnen Phasen jedoch wird iterativ gearbeitet, wie es der RUP vorsieht. So ist es möglich, das Software-Design auch noch mitten in der Diplomarbeit anzupassen. Die Nachteile bei der Verwendung eines reinen Wasserfallmodells können so zum grössten Teil vermieden werden. Natürlich sollen während dem ganzen Projekt die Fortschritte nachvollzogen und Risiken früh eingeschätzt und erkannt werden können (siehe 8.1.4).

8.1.2 Planung von Projektphasen

Um die Diplomarbeit überschaubar zu halten, wird sie in Phasen aufgeteilt. Eine Phase soll sich über Tage bis wenige Wochen erstrecken und wird am Ende mit einem Meilenstein abgeschlossen. Klar definierte Ziele der Projektphase sollen beim erreichen eines Meilensteins überprüft werden können und so Aufschluss über den aktuellen Fortschritt im ganzen Projekt geben. Die Phasen selbst sind nicht iterativ geplant, die Zielsetzungen sollen jedoch wenn möglich iterativ erarbeitet werden.

Die Aufteilung der Arbeit in die einzelnen Phasen wird bereits in der ersten Woche als Soll-Planung abgeschlossen. Diese Soll-Planung umfasst die einzelnen Projektphasen und eine grobe Definition der einzelnen Arbeitspakete (siehe 8.1.3). Muss diese Aufteilung aufgrund von geänderten Umständen angepasst oder verfeinert werden, so wird dies im Ist-Projektverlauf im Abschnitt 8.1.6.1 nachgeführt. Durch die Aufteilung in Soll-Planung und tatsächlich umgesetzter Planung kann die Arbeit am Ende besser beurteilt werden.

Allfällige Entscheide aufgrund der erreichten Resultate, welche Einfluss auf die Planung der Projektphasen haben, werden in Form eines Protokolls festgehalten.

8.1.3 Planung von Arbeitspaketen

Die Planung von Arbeitspaketen erfolgt zu Beginn des Projektes nur ganz grob und wird in die Soll-Planung der Projektphasen miteinbezogen (siehe 8.1.2). Es soll eine Aufwandschätzung der einzelnen Phasen erfolgen. Eine detailliertere Definition der Arbeitspakete einer Phase erfolgt jeweils gegen Ende der vorangehenden Phase und wird in der Zeiterfassung[1] eingetragen. Für diese detaillierten Arbeitspakete wird keine Aufwandschätzung gemacht, sondern nur die tatsächlich aufgewendeten Zeiten in die Zeiterfassung eingetragen.

Die Wochenplanung, sprich die genaue Einteilung der Arbeitspakete auf die Wochentage, erfolgt jeweils Anfangs Woche und wird dabei in einem Wiki[2] eingetragen. Diese Tagesaufgaben werden dabei in Form einer Todo-Liste geführt, wobei abgeschlossene Aufgaben als solche markiert werden können.

1 Die Zeiterfassung wird während der Diplomarbeit über eine Weboberfläche nachgeführt. Die Resultate aus dieser Zeiterfassung werden zum Schluss der Arbeit in diese Dokumentation übernommen und auf der CD abgegeben.

2 Das Wiki-System wurde vor der Diplomarbeit von den Diplomanden eingerichtet. Die Inhalte des Wikis werden am Ende der Diplomarbeit auf der abgegebenen CD enthalten sein.

8.1.4 Risikoanalyse

Eine Risikoanalyse soll Risiken des Projektes aufzeigen und auf mögliche Probleme hinweisen. Mit Hilfe der Risikoanalyse sollte es möglich sein, einzelne Risiken zu erkennen und genauer zu untersuchen. Die Risiken sollen so früh wie irgendwie möglich durch Studien und Tests eliminiert werden können. Allenfalls muss die Planung des Projektes angepasst werden.

Für jedes Risiko wird dessen Auftrittswahrscheinlichkeit und die Schwere des Schadens im Auftretensfall festgehalten. Zur Minimierung des Risikos sind Massnahmen zur Vorbeugung und Behebung zu definieren.

Die wichtigsten Risiken sollten nach der ersten Woche bestimmt sein. Neu auftretende Risiken während des Projekts sind in die Risikoanalyse aufzunehmen.

Die definierten Risiken werden während des Projektverlaufs verfolgt und deren Stand in der Risikoanalyse festgehalten. Die Massnahmen sind stets an die aktuelle Situation anzupassen. So kann der Status eines Risikos während dem Projekt beurteilt werden.

8.1.5 Soll-Planung

Die Soll-Planung entspricht dem Planungsstand am Ende der ersten Diplomarbeitswoche und wird danach nicht mehr verändert. Diese Planung besteht, wie bereits in 8.1.2 beschrieben, aus der Aufteilung des Projekts in Phasen und dazugehörenden groben Arbeitspaketen. Pro Woche wird mit einem durchschnittlichen Arbeitsaufwand von 45 Stunden gerechnet.

8.1.5.1 Projektverlauf

Die Soll-Planung ergibt folgenden Projektverlauf:

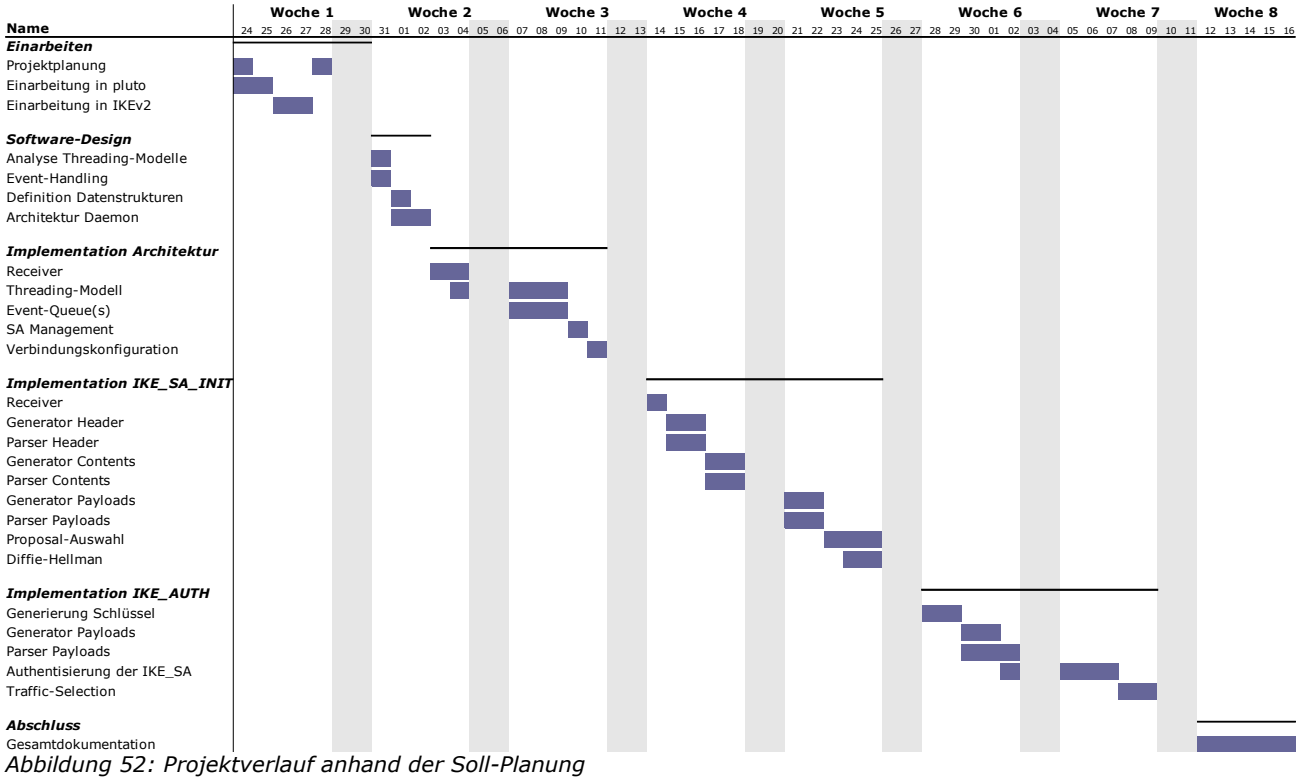


Abbildung 52: Projektverlauf anhand der Soll-Planung

Der obigen Abbildung ist zu entnehmen, dass in der Soll-Planung die acht Wochen der Diplomarbeit in insgesamt sechs Phasen aufgeteilt sind. Ein Meilensteine ist jeweils am Ende des letzten Werktages einer Phase. So ist beispielsweise der Meilenstein nach der ersten Phase „Einarbeiten“ am Freitagabend, dem 28. Oktober.

8.1.5.2 Projektphasen

Auf die einzelnen Projektphasen und Arbeitspakete aus dem Projektverlauf unter 8.1.5.1 wird nachfolgend genauer eingegangen. Änderungen an den Phasen und deren Arbeitspaketen sind im Abschnitt 8.1.6 ersichtlich. Immer wieder auftretende Tasks, wie beispielsweise administrative Aufgaben sind nicht speziell vermerkt.

8.1.5.2.1 Einarbeiten

In dieser Phase wird die Projektplanung vorgenommen und sich in die Thematiken `pluto`, `IKEv1` und `IKEv2` eingearbeitet. Die Diplomanden haben sich am Ende dieser Phase einen groben Überblick über die bevorstehenden Aufgaben verschafft und können den voraussichtlichen Aufwand ungefähr abschätzen. Genauer beinhaltet dies eine erste Einarbeitung in vorhandene Standards sowie das genauere Kennenlernen von `pluto`, dem `strongSwan IKEv1`-Daemon.

Arbeitspaket	**Ergebnisse am Ende der Phase**
Projektplanung	– Projektaufwand abgeschätzt und Soll-Projektplanung abgeschlossen – Erste Risikoanalyse erstellt
Einarbeitung in `pluto`	– Grobe Funktionsweise von `IKEv1` verstanden und beschrieben – Grober Aufbau von `pluto` verstanden und beschrieben – Ideen für neues Software-Design aufgeschnappt
Einarbeitung in `IKEv2`	– Grobe Funktionsweise von `IKEv2` verstanden und beschrieben

Tabelle 38: Soll-Planung der Phase "Einarbeiten"

8.1.5.2.2 Software-Design

In der zweiten Phase wird das Software-Design für die `IKEv2`-Implementierung definiert. Mögliche Design-Ideen konnten bereits beim Überblicken von `pluto` während der ersten Phase gesammelt werden. Verschiedene Design-Ansätze sollen berücksichtigt und analysiert werden. Am Ende dieser Phase soll das Architektur-Design definiert sein.

Arbeitspaket	**Ergebnisse am Ende der Phase**
Analyse Threading-Modelle	– Verschiedene Ansätze für Multi-Threading sind analysiert und beurteilt – Ein Threading-Modell ist gewählt
Architektur Daemon	– Der generelle Aufbau des Daemons ist definiert – Zuständigkeiten der Module sind klar geregelt – Kenntnisse aus der Analyse der Threading-Modelle sind in die Architektur übernommen
Definition Datenstrukturen	– Die Datenstrukturen für beispielsweise Verbindungsinformationen sind grob definiert und in die Module aufgeteilt
Event-Handling	– Das Design des Event-Handlings ist erstellt

Tabelle 39: Soll-Planung der Phase "Software-Design"

8.1.5.2.3 Implementierung Architektur

In der Phase „Implementierung Architektur“ wird die, im Software-Design definierte, Grundarchitektur für den neuen IKEv2-Daemon implementiert. Für eine spätere Weiterentwicklung soll eine saubere und solide Grundlage geschaffen werden. Das Design wird während der Implementierung ständig aktuell gehalten und Design-Anpassungen dokumentiert.

Arbeitspaket	Ergebnisse am Ende der Phase
Receiver	– Die einkommenden UDP-Pakete können eingelesen und weitergeleitet werden (Der Header wird noch nicht geparst und dadurch die Weiterleitung statisch konfiguriert) – Gemäss gewähltem Threading-Modell veranlasst der Receiver die weitere Verarbeitung
Threading-Modell	– Gemäss gewähltem Threading-Modell ist die Infrastruktur implementiert (Beispielsweise Thread-Pool, Event-Scheduler, Locks, usw.)
Verbindungskonfiguration	– Für die Einstellungen einer bestimmten IPsec-Verbindung sind Datenstrukturen definiert
SA Management	– Datenstrukturen zum festhalten von IKE_SA- und CHILD_SA-Informationen und Stati sind grob definiert (ohne detaillierte Datenfelder)
Event-Queue(s)	– Die Funktionalität zum Einspeisen und Abarbeiten von Events ist vorhanden

Tabelle 40: Soll-Planung der Phase "Implementierung Architektur"

8.1.5.2.4 Implementierung IKE_SA_INIT

In dieser Phase wird die minimale Unterstützung für den IKEv2-Exchange IKE_SA_INIT in den IKEv2-Daemon integriert. Nach dem Ende dieser Phase sollte der Exchange vom Typ IKE_SA_INIT zwischen zwei unterschiedlichen Hosts funktionieren.

Arbeitspaket	Ergebnisse am Ende der Phase
Receiver	– Entgegennahme von UDP-Paketen und Einreihung in eine Art Job-Queue
Diffie-Hellman	– Generierung von Diffie-Hellman-Werten und des gemeinsamen Schlüssels
Generator Payloads	– Generieren der in IKE_SA_INIT verwendeten Payloads
Parser Payloads	– Parsen der in IKE_SA_INIT verwendeten Payloads
Proposal-Auswahl	– Zusammenstellen und Auswählen von Suiten für eine SA
Generator Contents	– Zusammensetzen der Payloads
Parser Contents	– Auslesen der Payloads einer Message
Generator Header	– Generierung des IKEv2-Headers
Parser Header	– Parsen des IKEv2-Headers

Tabelle 41: Soll-Planung der Phase "Implementierung IKE_SA_INIT"

8.1.5.2.5 Implementierung IKE_AUTH

In dieser Phase wird die minimale Unterstützung für den IKEv2-Exchange IKE_AUTH in den IKEv2-Daemon integriert. Nach dem Ende dieser Phase sollte der Exchange vom Typ IKE_AUTH zwischen zwei unterschiedlichen Hosts funktionieren und eine erste SA vorhanden sein. Dafür notwendige Keys, Zertifikate, etc. dürfen hardcodiert im Code vorhanden sein.

Arbeitspaket	Ergebnisse am Ende der Phase
Generierung Schlüssel	– Pseudo-Random-Funktion prf+ implementieren – Schlüssel für Signierung, Verschlüsselung, etc. aus den Nonce-Informationen und DH-Werten des IKE_SA_INIT-Austauschs generieren
Generator Payloads	– Generieren der in IKE_AUTH verwendeten Payloads
Parser Payloads	– Parsen der in IKE_AUTH verwendeten Payloads
Authentisierung der IKE_SA	– Authentisierung der IKE_SA, beispielsweise mit Pre-Shared-Key
Traffic-Selection	– Angabe der akzeptierten Hosts und Subnetze

Tabelle 42: Soll-Planung der Phase "Implementierung IKE_AUTH"

8.1.5.2.6 Abschluss

In der letzten Phase wird die Diplomarbeit abgeschlossen. In dieser Phase werden keine weitere Features in den Code eingebaut. Die während der Arbeit begonnenen Dokumentationen sind am Ende dieser Phase abgeschlossen.

Arbeitspaket	Ergebnisse am Ende der Phase
Plakat	– A0-Plakat der Diplomarbeit
Gesamtdokumentation	– Komplette Dokumentation der Diplomarbeit in gebundener Ausgabe
CD	– Diplomarbeit zur Abgabe auf CD

Tabelle 43: Soll-Planung der Phase "Abschluss"

8.1.6 Umgesetzung

Die tatsächlich umgesetzte Planung ist diesem Abschnitt zu entnehmen. Die Daten stammen aus der Zeiterfassung, welche als Offline-Version auf der CD-ROM abrufbar ist. Abweichungen von der Soll-Planung sollen anhand dieser Umsetzung nachvollzogen werden können.

8.1.6.1 Projektverlauf

Die während der Diplomarbeit erfassten Arbeitsstunden ergeben folgenden Projektverlauf:

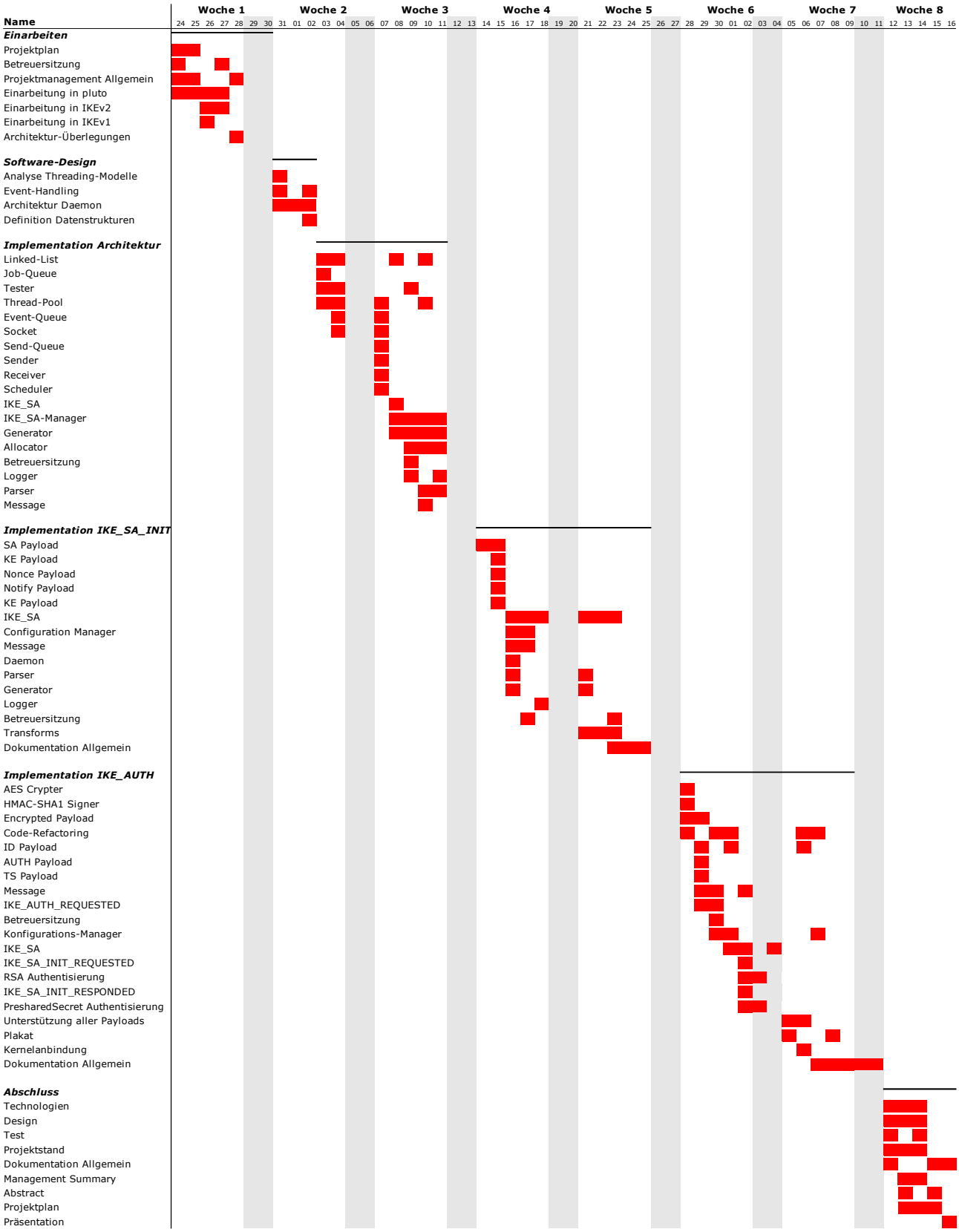


Abbildung 53: Tatsächlicher Projektverlauf

Die Projektphasen konnten alle gemäss der Soll-Planung eingehalten werden. Auch wurden die gesetzten Ziele erreicht. Im Vergleich zum Projektverlauf der Soll-Planung zeigt sich der tatsächliche Projektverlauf jedoch sehr chaotisch. Dieser Unterschied hat folgende Gründe:

- Die Soll-Planung wurde bereits Ende der ersten Woche eingefroren. Die zu jenem Zeitpunkt definierten Arbeitspakete wurden primär aufgrund der Analyse des bestehenden `strongSwan`-Codes bestimmt, welcher auf einer Single-Threaded-Architektur beruht.
- Erst in der Design-Phase kristallisierte sich der grobe Aufbau der Software-Architektur. Die in der Soll-Planung definierten Arbeitspakete zeigten sich schnell als zu grob und mussten verfeinert werden.
- Die Namen der Arbeitspakete im obigen Projektverlauf entsprechen den Bezeichnungen, wie Sie auch in der Zeiterfassung verwendet wurden. Diese wiederum können meistens einer erstellten Klassen oder einem erstellten Interface zugeordnet werden.
- Auch wenn an einem Arbeitspaket nur eine halbe Stunde gearbeitet wurde, so ist dieses im obigen Projektverlauf am entsprechenden Tag markiert und da an einem Tag meist an mehreren Arbeitspaketen gearbeitet wurde ergibt sich die teils chaotische Darstellung.

8.1.6.2 Auswertung

Die erfassten Arbeitsstunden ergeben folgendes Diagramm:

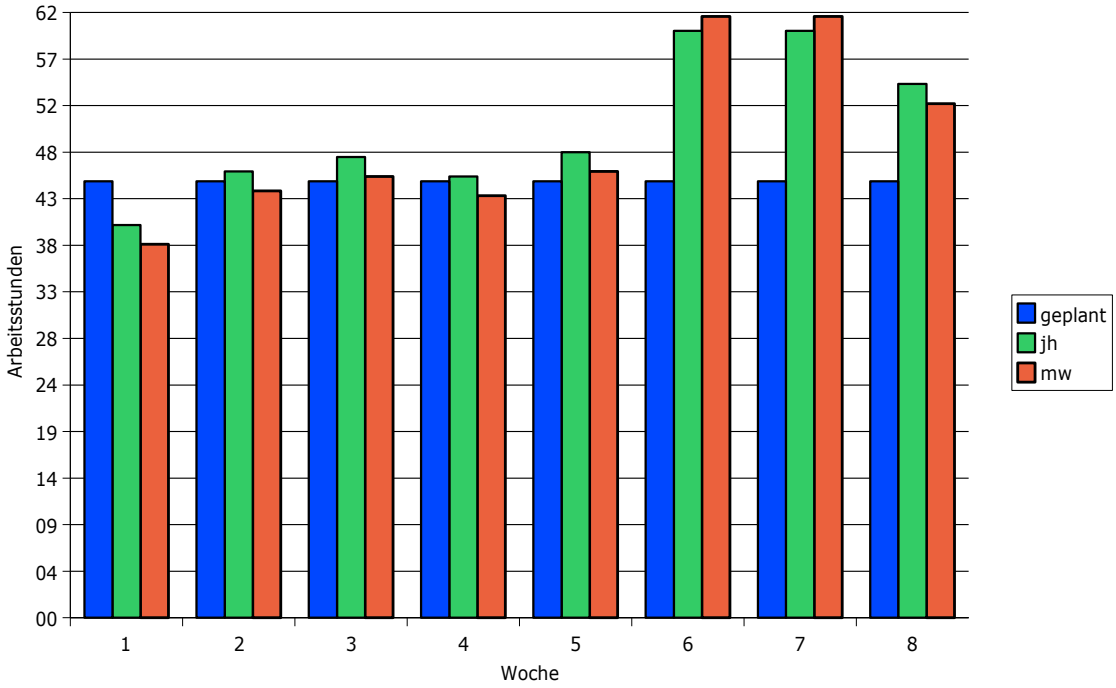


Abbildung 54: Diagramm der erfassten Arbeitsstunden

Durchschnittlich wurde mit einer Wochenarbeitsdauer von 45 Stunden gerechnet, was 9 Stunden pro Tag entspricht. In den ersten fünf Wochen entsprachen die tatsächlich geleisteten Stunden beider Diplomanden ziemlich genau diesen gerechneten 45 Stunden. In den nächsten beiden Wochen ist jedoch eine Erhöhung der Arbeitsdauer auszumachen. Die Gründe für diesen Anstieg sind nicht auf eine Fehlplanung zurückzuführen. Den Diplomanden war von Anfang an klar, dass in den letzten Wochen mehr Zeit investieren werden. Alle Phasen konnten in der geplanten Zeit durchgeführt werden. Der höhere Arbeitsaufwand in der Entwicklungsphase des IKE_AUTH-Austauschs ist auf die Tatsache zurückzuführen, dass sowohl die Authentisierung mit RSA-Schlüsseln als auch mit Shared-Secret implementiert wurde. Auch wurde bereits mit der Kernelanbindung begonnen. Diese Dinge hätten gemäss Planung nicht eingebaut werden müssen, wurde aber aus Ehrgeiz trotzdem implementiert. Der Mehraufwand in der letzten Woche ist auf die intensive und langwierige Korrektur der kompletten Dokumentation zurückzuführen.

8.2 Risikoanalyse

8.2.1 Geplante Ziele aus Zeitmangel nicht erreichbar

Risiko		Die geplanten Ziele können sich während dem Projektverlauf als zu umfangreich herausstellen. Aus Zeitmangel können so nicht alle geplanten Features von `IKEv2` implementiert werden.
Auftrittswahrscheinlichkeit		40%
Schwere des Schadens		gering bis mittel
Massnahmen zur Vorbeugung		– Implementierungsaufwand anhand der vorhandenen `IKEv1`-Implementierung `pluto` abschätzen. – Sich bereits frühzeitig in den `IKEv2`-Standard einarbeiten. – Während Implementierung keine Prioritäten auf „Nice to have“-Features setzen.
Massnahmen zur Behebung		– Falls während der Implementierung klar wird, dass die geplanten Ziele nicht erreicht werden können, müssen diese mit dem Betreuer angepasst werden. Es muss darauf geachtet werden, dass auf jeden Fall die Architektur der Software solide implementiert wird und als gute Grundlage für eine Weiterentwicklung verwendet werden kann.
Status	**25.10.2005**	– Risiko definiert.
	28.10.2005	– Die ersten beiden Massnahmen zur Vorbeugung dieses Risikos wurden bereits getroffen: Die `IKEv1`-Implementierung `pluto` wurde detailliert überblickt und dabei der Aufwand für die `IKEv2`-Implementierung abgeschätzt. Auch hat man sich bereits in den `IKEv2`-Standard eingearbeitet.
	13.11.2005	– Geplante Ziele der Phase 2 grösstenteils erreicht. Ziele der Phase 3 aufgrund gewonnener Kenntnisse als machbar beurteilt.
	23.11.2005	– Geplante Ziele der Phase 3 erreicht. Ziele der Phase 4 aufgrund gewonnener Kenntnisse als machbar beurteilt.
	09.12.2005	– Geplante Ziele der Phase 4 konnten grösstenteils erreicht werden. Es sind zwar nicht alle Funktionen für eine völlige `IKEv2`-Kompatibilität implementiert, es fehlen aber nur leicht nachrüstbare Elemente. Der genaue Stand und die Kompatibilität ist im Dokument „Projektstatus“ beschrieben.

Tabelle 44: Risiko „Geplante Ziele aus Zeitmangel nicht erreichbar"

8.2.2 Deadlocks aufgrund des Threading-Modells

Risiko		Bei der Implementierung von `IKEv2` mit Hilfe mehrerer Threads besteht das Risiko, dass sich die Threads gegenseitig blockieren und es dadurch zu so genannten Deadlocks kommt.
Auftrittswahrscheinlichkeit		30%
Schwere des Schadens		mittel
Massnahmen zur Vorbeugung		– Threading-Modell sehr klar spezifizieren. – Aufgaben der Threads klar eingrenzen. – Sequenzielle Locks in umgekehrter Reihenfolge wieder freigeben.
Massnahmen zur Behebung		– Falls der Fehler nicht gefunden wird, auf ein Thread reduzieren.
Status	**28.10.2005**	– Risiko definiert.
	23.11.2005	– In den bisherigen Tests sind keine Deadlock-Situationen aufgetreten. Die gemachten Tests wurden mit extrem vielen Threads durchgeführt, um Probleme zu provozieren.
	05.12.2005	– Ein Test, welcher mit 100 Threads parallel 500 IKE_SAs aufbaut verläuft problemlos. Dies ist ein Indiz dafür, dass es auch in normalen Betrieb keine Deadlocks geben wird.
	09.12.2005	– Es konnten im Projektverlauf keine Probleme mit Deadlocks festgestellt werden. Die Massnahmen zur Vorbegung scheinen gewirkt zu haben. Die Threading-Architektur scheint ihren Dienst einwandfrei zu leisten.

Tabelle 45: Risiko „Deadlocks aufgrund des Threading-Modells"

8.2.3 Korrupte Daten aufgrund des Threading-Modells

Risiko		Werden Locks ungenügend oder fehlerhaft eingesetzt kann es in der multi-threaded Implementierung zu korrupten Datenzuständen führen.
Auftrittswahrscheinlichkeit		30%
Schwere des Schadens		mittel
Massnahmen zur Vorbeugung		– Threading-Modell sehr klar spezifizieren. – Daten in Methoden kapseln, die Locks verwenden. – Gemeinsam verwendete Ressourcen mit Locks schützen.
Massnahmen zur Behebung		– Falls der Fehler nicht gefunden wird, auf ein Thread reduzieren.
Status	**28.10.2005**	– Risiko definiert.
	23.11.2005	– Der Zugriff auf gemeinsame Daten erfolgt über synchronisierte Methoden. Threads können sich dadurch nicht gegenseitig in die Quere kommen. Locks können durch die Kapselung der Datenzugriffe fast nicht vergessen werden.
	09.12.2005	– Im ganzen Projektverlauf sind nie Probleme mit korrupten Daten entstanden, die durch ungenügend geschützte Ressourcen herbeigeführt wurden. Die Kapselung der Daten in Klassen und die Verwendung von Locks darin scheint sich ausbezahlt zu haben.

Tabelle 46: Risiko „Korrupte Daten aufgrund des Threading-Modells"

8.2.4 Kompatibilität kann nicht getestet werden

Risiko		Um die IKEv2-Kompatibilität garantieren zu können, muss gegen eine bereits bestehende Implementierung getestet werden. Kann aus Zeitmangel oder aus Komplexität einer anderen Implementierung keine solche in Betrieb genommen werden, ist ein Testen nicht möglich.
Auftrittswahrscheinlichkeit		50%
Schwere des Schadens		mittel
Massnahmen zur Vorbeugung		– Andere Implementierungen in einer frühen Phase überblicken und Installationsaufwand abschätzen. – Eine andere Implementierung bereits in der ersten Entwicklungsphase kompilieren und installieren.
Massnahmen zur Behebung		– Auf das Testen gegen eine andere Implementierung wird verzichtet, da dies in einem solch frühen Stadium der Software keine grosse Priorität hat.
Status	**28.10.2005**	– Risiko definiert.
	17.11.2005	– Die IKEv2-Implementierung [IKEv2Linux] konnte erfolgreich kompiliert werden. – Auf das Testen gegen [Racoon2] wird verzichtet, da der Konfigurationsaufwand zu gross ist. – Das Testen gegen [IKEv2Linux] wird als ausreichend beurteilt. Risiko eliminiert.

Tabelle 47: Risiko „Kompatibilität kann nicht getestet werden"

8.2.5 Kompatibilität ist nicht gegeben

Risiko		Beim Testen gegen eine andere IKEv2-Implementierung erweist sich diese als nicht kompatibel mit der Unsrigen.
Auftrittswahrscheinlichkeit		15%
Schwere des Schadens		gering
Massnahmen zur Vorbeugung		– Bei der Entwicklung wird sich strikt ans RFC gehalten. – Die übertragenen Pakete werden, so weit möglich, mit einem Netzwerksniffer überprüft. – Nach jedem Entwicklungsschritt wird die Software gegen die andere Implementierung getestet.
Massnahmen zur Behebung		– Kompatibilitätsprobleme, die sich bis zum Schluss der Arbeit nicht eliminieren lassen, gut Dokumentieren.
Status	**28.10.2005**	– Risiko definiert.
	17.11.2005	– Erster Kompatibilitätstest der Parser- und Generator-Funktionalität konnte erfolgreich durchgeführt werden.
	23.11.2005	– Das Aushandeln der gemeinsamen Schlüssel einer IKE_SA konnte erfolgreich getestet werden.
	06.12.2005	– [IKEv2Linux] scheint sich bei der Erstellung der Authentisierungsdaten nicht strikt an den Draft zu halten. – Durch ein Workaround kann der Aufbau einer kompletten IKE_SA mit [IKEv2Linux] aufgebaut werden. Unsere Implementierung ist dabei Initiator oder Responder.

Tabelle 48: Risiko „Kompatibilität ist nicht gegeben"

8.3 Programmierrichtlinien

8.3.1 Quellcode-Dokumentation

Die Dokumentation des C Quellcodes erfolgt mit `Doxygen`. Dabei handelt sich um ein Tool ähnlich zu `JavaDoc`, welches aus dem Quelltext spezielle Kommentare extrahiert. Die daraus generierbaren HTML-Seiten sollen als Programmier-Referenz eingesetzt werden können.

Um ein Element im Quelltext zu dokumentieren, wird ein spezieller `Doxygen`-Kommentar eingefügt. Wir verwenden dazu die beiden folgenden Syntaxen:

```
/** Einzeiliger Doxygen-Kommentar */

/**
 * Mehr-
 * zeiliger
 * Doxygen-
 * Kommentar
 */
```

Für normale Kommentare, die nicht mit `Doxygen` erfasst werden sollen, werden die folgenden Syntaxen verwendet:

```
/* Normaler C-Kommentar */

/*
 * Mehr-
 * zeiliger
 * C-Kommentar
 */
```

Die Verwendung dieser Kommentare erfolgt meist nur in Funktionen um beispielsweise den Ablauf zu erläutern. Die Restlichen Kommentare sollen mit `Doxygen` erfolgen.

Spezielle Tags ermöglichen das Generieren von erweiterten Kommentaren, wobei wir uns auf die folgenden beschränken[1]:

Tag	Beschreibung
@file	Definiert den Dateinamen der Quelltext-Datei
@brief	Kurzbeschreibung des Elementes
@param	Beschreibung eines Funktionsparameters
@return	Beschreibung eines Rückgabewertes
@todo	Arbeit, die noch erledigt werden muss
@see	Verweis auf ein anderes Element im Quelltext
@ingroup	Zuweisen eines Elementes zu einem Package
@defgroup	Definition eines Packages

Tabelle 49: Verwendete Doxygen-Tags

1 Weitere, von `Doxygen` unterstützte Tags, können dem `Doxygen`-Manual [DoxygenManual] entnommen werden.

8.3.2 Quellcode-Konventionen

8.3.2.1 Code-Strukturierung

Für die Strukturierung des Quellcodes gelten folgende Regeln:

- Geschweifte Klammern bei Funktionen und Flusskontroll-Konstrukten auf eine neue Zeile
- Geschweifte Klammern bei `structs`/`unions`/`enums` und initialisierten Arrays auf derselben Zeile
- Operatoren/Variablen/Zahlen/Argumente immer mit Leerzeichen trennen
- Einrücken mit Tabs, Länge der Tabs: Vier Zeichen
- Maximale Breite des Quelltextes: 100 Zeichen

8.3.2.2 Namensgebung

Für die Namensgebung von Elementen sind folgende Regeln einzuhalten:

- Konstantennamen in Grossbuchstaben, Teilwörter mit „_“ getrennt
- Alles andere klein geschrieben, zusammengesetzte Wörter mit „_“ zusammengehängt
- Typen werden immer mit endendem „`_t`“ definiert
- Arrays, welche String-Repräsentationen für `enum`-Werte enthalten, enden mit „`_m`“

8.3.3 Klassen-Aufbau

Wie im Dokument „Design“ beschrieben, werden Funktionalitäten in Klassen gekapselt. Folgendes Beispiel soll die Strukturierung und Dokumentation einer Klassendefinition beschreiben:

```
typedef struct example_t example_t;

/**
 * @brief Kurzbeschreibung der Klasse.
 *
 * Details der Klasse und deren Implementierung.
 *
 * @b Constructors:
 * - example_create()
 *
 * @todo Beschreibung des Todo
 *
 * @ingroup examples
 */
struct example_t {
    /**
     * @brief Kurzbeschreibung der Methode.
     *
     * Zusatzinformationen zu Methode.
     *
     * @param value          Beschreibung des Argumentes value
     * @param[out] out_value Beschreibung des Out-Argumentes out_value
     * @return
     *                       - SUCCESS
     *                       - INVALID_ARG if value invalid
     *
     * @todo Beschreibung Todo
     */
    status_t (*do_it) (example_t *this, u_int8_t value, u_int8_t* out_value);
};

/**
 * @brief Beschreibung Konstruktor.
 *
 * Detailliertere Beschreibung.
 *
 * @param value              Beschreibung Konstruktor-Argument
 * @return                   example_t object
 *
 * @ingroup examples
 */
example_t *example_create(u_int8_t value);
```

Dabei gilt:

- Für eine Struktur bzw. Klasse wird immer ein Typ erstellt, welcher denselben Namen trägt
- Konstruktoren tragen immer das Prefix *klassenname_create*
- Argumente, welche einen Wert zurückliefern, werden mit `[out]` markiert
- Gibt es mehrere Varianten als Rückgabewert einer Methode, werden sie als Liste dargestellt

8.4 Protokolle

8.4.1 Sitzungsprotokoll vom 24.10.2005

8.4.1.1 Allgemeines

Ort: HSR

Zeit: 10:00 Uhr

Teilnehmer: as, jh, mw

Protokollant: mw

8.4.1.2 Besprochene Punkte

Die Aufgabenstellung wurde mit dem Betreuer besprochen:

- Es soll wirklich nur ein Minimum implementiert werden, im Umfang nicht übernehmen

Gemeinsam wurde der Quellcode von `pluto` untersucht und die Files erläutert:

- `connection.c` Verbindungsverwaltung mit den Endpunkten „this“ und „that“
- `constants.h` Alle Konstanten, welche in `IKEv1` definiert sind
- `crypto.c` Standard-Algorithmen
- `dbops.c` Unterstützung von Proposals
- `defs.c` Utility-Definitionen (Chunks usw.)
- `demux.c` Demuxen von Paketen, mit State-Machine
- `ikealg.c` Verwaltung von Algorithmen
- `kernel.c` Generische Schnittstelle zum Kernel
- `kernel_netlink.c` Spezifische Schnittstelle über `Netlink`
- `keys.c` Verwaltung von Secrets
- `packet.c` (De-)Marshalling von Paketen, Paketdefinitionen
- `plutomain.c` Kommandozeilen-Parsing
- `rcv_whack.c` Empfängt Kommandos von `whack`
- `whack.c` Steuerungsprogramm für `pluto`
- `rnd.c` Pseudo-Random-Funtionen
- `spdb.c` Datenbank von SAs
- `state.c` State-Machine
- `timer.c` Event-Abarbeitung

8.4.1.3 Nächste Sitzung

Donnerstag, 27.10.2005, 10.00 Uhr

8.4.2 Sitzungsprotokoll vom 27.10.2005

8.4.2.1 Allgemeines

Ort: HSR

Zeit: 10:00 Uhr

Teilnehmer: as, jh, mw

Protokollant: jh

8.4.2.2 Besprochene Punkte

- Der Aufbau eines connection-`structs`, definiert in `plutos connection.h`, wurde näher betrachtet:
  - Connections werden zwischen „template“ und „instantiated“ unterschieden
  - „Template“-connections entsprechen connection-Definitionen aus dem `ipsec.conf`
  - Eine Connection kann sich während Phase 1 oder 2 noch ändern
- Bei der Wahl der Architektur für die `IKEv2`-Implementierung wurde empfohlen, ein Thread-Testprogramm zu schreiben, um die Tauglichkeit von Threads zu überprüfen
- Auf `whack` und `kernel`-Schnittstelle wird für diese Arbeit verzichtet
- Es soll versucht verden, die für `IKEv2` vorgeschriebenen Mindestanforderungen zu erfüllen

8.4.2.3 Nächste Sitzung

Montag, 31.10.2005, am späteren Nachmittag

8.4.3 Sitzungsprotokoll vom 10.11.2005

8.4.3.1 Allgemeines

Ort: HSR

Zeit: 13:00 Uhr

Teilnehmer: as, jh, mw

Protokollant: jh

8.4.3.2 Besprochene Punkte

- Der `IKEv2`-Daemon soll möglichst unabhängig vom `pluto` programmiert werden
- Ein neues Logging-System ist wünschenswert, deshalb wird ein eigener Logger geschrieben
- Wiederverwendbare Funktionen dürfen aus `pluto` entnommen und in Klassen gekapselt werden. Eine 1:1 Übernahme der Funktionen ist nicht vorgesehen
- `PFKEY` ist die normierte Kernel-Schnittstelle für den IPsec-Stack
- Dateien zu `NET_KEY` in `pluto` enthalten `structs` für das `PFKEY`-Format
- Operationen auf den Kernel werden über eine Art Interface durchgeführt. Das Interface heisst `kernel_ops` und nutzt entweder `KLIPS` oder `Netkey`
- Die Konfiguration des Daemons kann an das `connection`-Format von `pluto` angelehnt werden

8.4.3.3 Nächste Sitzung

Donnerstag, 17.11.2005, 09:00 Uhr

8.4.4 Sitzungsprotokoll vom 17.11.2005

8.4.4.1 Allgemeines

Ort: HSR

Zeit: 14:00 Uhr

Teilnehmer: as, jh, mw

Protokollant: mw

8.4.4.2 Besprochene Punkte

- Vorführung:
  - Kompatibilitätstest gegen `IKEv2`-Implementierung von sourceforge.org.
  - Parsen einer von uns generierten Meldung möglich
  - Umgekehrtes wird wahrscheinlich morgen folgen
- Konfiguration:
  - Aufbau der Konfiguration besprochen
  - Keine Herausgabe kompletter Konfigurationen, um Problem mit mehreren Konfigurationen zu vermeiden
  - Herausgabe von einzelnen Konfigurationsparametern. Diese werden von einem Konfigurations-Manager herausgegeben

8.4.4.3 Nächste Sitzung

Mittwoch, 23.11.2005, 16:00 Uhr

8.4.5 Sitzungsprotokoll vom 23.11.2005

8.4.5.1 Allgemeines

Ort: HSR

Zeit: 16:00 Uhr

Teilnehmer: as, jh, mw

Protokollant: jh

8.4.5.2 Besprochene Punkte

- Vorführung:
  - IKE_SA_INIT-Austausch bis zum Generieren des gemeinsamen Diffie-Hellman-Schlüssels
- Der Zeitplan für die Phase 4 kann voraussichtlich eingehalten werden
- Der IKE_SA_INIT-Austausch sollte bis zum Freitag, 25.11.2005 vollständig durchgeführt werden können

8.4.5.3 Nächste Sitzung

Mittwoch, 30.11.2005, im Verlaufe des Nachmittags

8.4.6 Sitzungsprotokoll vom 30.11.2005

8.4.6.1 Allgemeines

Ort: HSR

Zeit: 14:00 Uhr

Teilnehmer: as, jh, mw

Protokollant: jh

8.4.6.2 Besprochene Punkte

- Vorführung:
  - Die Pakete des IKE_AUTH-Austausches werden bereits verschlüsselt
  - Kompatibilität der Verschlüsselung mit der anderen `IKEv2`-Implementierung ist vorhanden
- Konfiguration:
  - Von einer atomaren Konfiguration wurde Abschied genommen
  - Konfiguration ist neu unterteilt in zwei Bereiche: INIT- und SA-Konfiguration
- Der IKE_AUTH-Austausch sollte bis Mitte nächster Woche vollständig durchgeführt werden können

8.4.6.3 Nächste Sitzung

Mittwoch, 07.12.2005, im Verlaufe des Nachmittags

8.4.7 Sitzungsprotokoll vom 07.12.2005

8.4.7.1 Allgemeines

Ort: HSR

Zeit: 15:00 Uhr

Teilnehmer: as, jh, mw

Protokollant: jh

8.4.7.2 Besprochene Punkte

- Vorführung:
  - Ein Aufbau einer IKE_SA wurde demonstriert
- Erste Version der Kernel-Schnittstelle konnte zum laufen gebracht werden
- Interpretationsfehler: Für den Diffie-Hellman-Austausch ist keine Berechnung von Primzahlen notwendig
- Die Dokumentation soll für Experte und Betreuer doppelseitig ausgedruckt werden

9 Erfahrungsberichte

9.1 Jan Hutter

Diese Diplomarbeit war nun schon die dritte Arbeit, die wir in der Programmiersprache C verwirklicht haben. Anfangs hatte ich ernsthafte Bedenken, dass ich meine objektorientierten Kenntnisse verlernen könnte. So hat mich die früh getroffene Entscheidung, objektorientierte Ansätze auch in C anzuwenden, sehr motiviert und die meisten Bedenken auszuräumen vermocht.

Die gute Atmosphäre im Team hat dazu beigetragen, auch schwierige Situationen mit Humor zu meistern. Eine solch gute Zusammenarbeit ist nicht selbstverständlich und ich bin froh, dass wir nie Probleme untereinander hatten.

Zum ersten Mal darf ich feststellen, dass die Zeitplanung nicht völlig von der Realität entfernt war. Der geplante Programmieraufwand, der in den bisherigen Studienarbeiten stets falsch eingeschätzt wurde, stimmte dieses Mal relativ genau mit dem tatsächlich investierten Aufwand überein. Die Aussage vieler alter Hasen, dass die Planung eine Erfahrungssache ist, hat sich bestätigt.

Gemäss dem RUP wird eine Applikation bereits in den ersten Phasen zu einem grossen Teil entworfen und das daraus entstehende Design während der Programmierphase nur noch wenig angepasst. Diesen Ansatz haben wir versucht durchzusetzen. Schnell mussten wir aber erkennen, dass das detailliere Design erst während der Programmierung festgelegt werden konnte. Die Komplexität der Anforderungen hat eine frühe Festlegung des detaillierten Designs verunmöglicht.

Die Betreuung durch Prof. Dr. Steffen empfand ich als sehr gut. Durch sein gutes Fachwissen konnte er uns immer wieder Tipps geben. Dass für Diffie-Hellman keine Primzahlen-Berechnung notwendig ist, vergesse ich sicher nicht mehr :-).

Die Arbeit hat mir viel Spass bereitet, ist doch die Internet-Sicherheit eines meiner Lieblingsgebiete. Ich denke, wir haben eine brauchbare Grundlage geschaffen, auf der die zukünftige Version von `strongSwan` aufbauen kann. Ich hoffe sehr, dass unser „charon“ auch seinen Einsatz finden wird.

9.2 Martin Willi

In meinen Badeferien in der Türkei war ich ab Mitte Woche leider dazu verdammt, den Tag im Liegestuhl zu verbringen. Befund: Armbruch beim Beach-Volleyball. Zum Glück hatte ich vorgesorgt und mir den Draft von IKEv2 ausgedruckt und in den Koffer gepackt. So konnte ich meine Zeit wenigstens mit etwas nützlichem verbringen.

Der Draft ist sehr angenehm zu lesen, leicht verständlich und logisch strukturiert. Als ich zurück in der Schweiz war und die Woche darauf mit der Diplomarbeit begann, fühlte ich mich gut gerüstet. Den Draft einigermassen verstanden, dachte ich.

Ich und mein Diplomarbeitskollege waren sehr motiviert. Ich hätte keine Arbeit lieber gemacht, als diejenige, die wir ausgeführt haben. Hochinteressantes Thema, mit sicherlich genügend Herausforderungen, sowie und die Möglichkeit, etwas zu schaffen, das bei Erfolg auch Verwendung findet. Als nach der ersten Sitzung mit dem Betreuer klar war, dass wir auch auf "der grünen Wiese" IKEv2 implementieren dürfen, war ich jedoch ein wenig erleichtert. Denn so durften wir unsere eigenen Ideen viel eher ausleben, als wir das bei einer Implementation in pluto hätten machen können. Eine vielseitige und erweiterbare Architektur zu entwickeln war sehr interessant.

Als wir dann in der ca. vierten Woche mit der Implementierung des ersten Meldungsaustausches begannen, wurde mir langsam bewusst, wie komplex die Umsetzung eines solchen Protokolls eigentlich ist. Wenn man den Draft so liest, ist er nicht all zu schwer verständlich. Wenn man aber vor dem konkreten Problem steht, werden einem die Schwierigkeiten bewusst: Der Zustand eines Objektes ist nicht immer der richtige. Es gibt nicht nur immer einen Initiator oder einen Responder, es können zig sein. Der Ablauf verläuft auch nicht immer nach Schema X (bzw. "Schema Draft"), sondern jede Eventualität muss durchdacht werden. Bei der Entwicklung einer Software, bei der die Stabilität und Sicherheit die wesentlichen Faktoren sind, sind diese Eventualfälle nahezu das wichtigste. Und diese immer richtig zu behandeln ist nicht nur ungemein wichtig, sondern auch nach dreifachem durchlesen des Drafts nicht immer ganz klar. Es gibt wohl einfach zu viele.

Schlussendlich hat mir die Arbeit aber sehr viel Spass gemacht. Wir haben alle Probleme lösen können, auch wenn dazu manchmal mehr als ein Lösungsvorschlag da war: So gab es zwischen mir und meinem Arbeitskollegen oft heftige Diskussionen, wie ein Problem nun zu lösen sei. Dies war sehr interessant, auch wenn teilweise fast die Atmosphäre eines Streites herrschte. Nicht immer war es einfach, dem anderen seine Idee klar zu machen; meistens scheiterte es daran, seine komplexen Gedanken zu Wort oder Papier zu bringen. Ich muss aber sagen, dass ich alles in allem mit Jan sehr gut klar kam und wir eine tolle Teamatmosphäre geniessen durften.

Ich hatte sehr viel Freude daran, die Grundlage für einen IKE-Daemon zu schaffen. Auch wenn das Resultat jetzt noch keinen praktischen Nutzen finden wird, so glaube ich ein gute Grundlage geschaffen zu haben. Ich hoffe, unsere Design-Prinzipien finden auch bei eingefleischten C-Cracks Anklang und die Software findet Anhänger. Ich selbst bin aber vom Konzept der ganzen Software sehr überzeugt und werde ganz sicher ein Fan davon bleiben. Hoffentlich kann ich ab und zu noch ein paar weitere Zeilen dazu beisteuern...

10 Anhang

10.1 CD-Struktur

Der gedruckten Dokumentation der Diplomarbeit liegt eine CD-ROM bei. Sie enthält alle Dokumente in digitaler Form, den Quellcode des Daemons, verwendete Quellen und weiteres. Die nachfolgende Auflistung gibt einen Überblick zum Inhalt der CD-ROM.

Pfad	**Inhalt/Beschreibung**
Codedokumentation/	Aus dem Sourcecode extrahierte Dokumentation im HTML-Format, generiert mit `Doxygen`. Kann als Referenz verwendet werden, um den Quellcode zu studieren und zu erweitern.
Dokumentation/	Komplette Dokumentation im PDF- sowie im OpenDocument-Format. Details dazu sind dem Dokumentenverzeichnis zu entnehmen.
Sourcecode/	Quellcode des `IKEv2`-Daemons `charon` und den dazugehörenden Modultests.
Quellen/	Im Quellenverzeichnis angegebene Quellen, welche für die Diplomarbeit verwendet wurden.
Wiki/	Offline-Version des während der Arbeit eingesetzten Wikis. Dieses wurde verwendet, um die Wochen zu planen und Notizen zu erstellen. Ebenfalls wurden Informationen darin gesammelt, die nicht direkt in die Dokumentation geflossen sind.
Zeiterfassung/	Offline-Version der Zeiterfassung. Die Zeiterfassung wurde über eine Web-Plattform geführt.

10.2 Inbetriebnahme

10.2.1 Quellcode kompilieren

Um den Daemon aus dem Sourcecode zu übersetzen, müssen folgende Bedingungen erfüllt sein:

- Sourcecode der CD auf die Harddisk kopiert
- `gmp`-Library [libgmp] ab Version 4.1 muss installiert sein (`gmp.h` im Standard-`include`)

Sind diese erfüllt, kann der Sourcecode übersetzt werden:

```
cd /to/source/code
make
```

Beim Kompilieren wird ein Unterverzeichnis `bin/` erstellt, welches die ausführbaren Programme `charon` und `run_tests` beinhaltet.

10.2.2 Testlauf durchführen

Ein Testlauf des Daemons kann folgendermassen gestartet werden (`root`-Rechte erforderlich):

```
bin/charon localhost-rsa
```

`localhost_rsa` stellt eine Konfiguration dar, welche mit dem lokalen Rechner eine Verbindung aufbaut und über RSA authentisiert. Das Testen weiterer Konfigurationen ist im Dokument „Tests“ unter „Systemtests“ beschreiben.

10.2.3 Modultests durchführen

Nachdem der Sourcecode übersetzt wurde, können auch die Modultests durchgeführt werden:

```
bin/run_tests
```

Der Aufruf von `run_tests` führt alle Modultests durch. Die Statusausgabe erfolgt dabei auf die Konsole.

10.2.4 Codedokumentation generieren

Um die Codedokumentation neu zu übersetzten, steht ebenfalls ein `make`-Target zur Verfügung[1]:

```
make doxygen
```

Es wird ein Unterverzeichnis `doc/` erstellt, welches die Dokumentation des Quellcodes in HTML beinhaltet.

Die Darstellung der erzeugten Dokumentation kann in der Datei `Doxyfile` angepasst werden.

1 Setzt die Installation von `doxygen` voraus.

10.3 Glossar

Ausdruck	Definition und Erklärung
3DES	Triple Data Encryption Standard. Dreifache DES-Verschlüsselung mit zwei oder drei unterschiedlichen Schlüsseln.
AES	Advanced Encryption Standard. Symmetrisches Block-verschlüsselungsverfahren. Gilt als Nachfolger von DES und 3DES.
AH	Authentication Header. Erweitert ein IP Paket um Integrität und Authentizität.
ASN.1	Abstract Syntax Notation One. Standard zur abstrakten Beschreibung von Datentypen.
Asymmetrische Verschlüsselung	Die asymmetrische Verschlüsselung ist die Verschlüsselung mit einem Schlüssel bestehend aus Public- und Private-Teil. Daten, die mit dem Public-Teil verschlüsselt wurden, können nur mit dem Private-Teil entschlüsselt werden. Ein Beispiel eines asymmetrischen Verschlüsselungsverfahrens ist RSA.
BER	Basic Encoding Rules. Definiert eine Kodierung von ASN.1 Datentypen.
CA	Certification Authority. Zertifizierungsstelle für digitale Zertifikate. Gibt Zertifikate heraus.
Charon	Charon ist der Name des entwickelten IKEv2-Daemons. Sein Name hat der Daemon aus der griechischen Mythologie, in welcher Charon den Fährmann zwischen dieser und der Unterwelt verkörpert.
CHILD_SA	Der Begriff CHILD_SA wird in IKEv2 für SAs der Protokolle AH oder ESP verwendet.
Conditional-Variable	Betriebssystem-Objekt, mit welchem schlafende Threads geweckt werden können.
Daemon/Dämon	Als Daemon wird ein Programm unter Unix-Systemen bezeichnet, welches ohne Benutzerinteraktion läuft. In der deutschen Sprache wird Daemon oft mit Dämon übersetzt. Unter Windows wird ein solches Programm als Dienst bezeichnet.
DER	Distinguished Encoding Rules. Definiert eine Kodierung von ASN.1 Datentypen. DER ist ein Subset von BER.
Diffie-Hellman	Diffie-Hellman ist ein Verfahren, um Schlüssel sicher über einen unsicheren Kanal auszuhandeln.
Distinguished Name	Eindeutige Bezeichnung eines Knotens in einem Baum.

Ausdruck	Definition und Erklärung
DOI	Domain of Interpretation. Anwendungsspezifische Eigenschaften.
DoS	Denial of Service. Eine DoS-Attacke bezeichnet den Versuch, einen angebotenen Dienst zu überfordern und somit dessen Verfügbarkeit zu unterbrechen.
EAP	Extensible Authentication Protocol. Authentifizierungs-Framework.
ESP	Encapsulated Security Payload. Erweitert ein IP Paket um Integrität, Authentizität und Vertraulichkeit.
gmp	GNU Multi Precision Arithmetic Library. Bibliothek für die Verwendung von grossen Zahlen.
Hash-Funktion	Eine Hash-Funktion erstellt aus einem beliebigen Dateninput eine eindeutige Repräsentation dieser Daten, ein so genannter Digest oder Hashwert. Dabei muss das Ergebnis für den selben Input stets gleich ausfallen. Eine Hash-Funktion ist nicht umkehrbar. Beispiele für Hash-Funktionen sind MD5 und SHA1.
HMAC	Hashing for Message Authentication. Erstellt MACs mit Hilfe von einem Schlüssel und einer Hash-Funktion.
ID	Identifikation.
IKE_SA	Der Begriff IKE_SA wird in IKEv2 für SAs verwendet, welche genutzt werden um CHILD_SAs aufzubauen.
Initiator	Als Initiator wird derjenige Kommunikationspartner bezeichnet, welcher eine Anfrage initiiert hat.
IPsec	IP Security. Sicherheitsarchitektur für IP. Umfasst ESP, AH und IKE.
IPv6	Internet Protocol in der Version 6.
ISAKMP	Internet Security Association and Key Management Protocol. Ein Framework, welches Prozeduren und Paketformate zur Verwaltung von Security Associations definiert.
KAME	Das KAME Projekt ist ein Zusammenschluss sechs japanischer Firmen, welche sich zum Ziel gesetzt haben, einen freien IPv6 und IPsec Stack für BSD zur Verfügung zu stellen.
KLIPS	Kernel IPsec support. IPsec Implementierung im 2.4er Linux Kernel. Wurde auch auf Linux 2.6 portiert.
MAC	Message Authentication Code. Kryptographisch gesicherte Prüfsumme, welche die Authentizität und Integrität von Daten sicherstellt.

Ausdruck	Definition und Erklärung
Man in the middle	Bei einer „Man in the middle“-Attacke wird der Datenverkehr von durch eine dritte Person abgehört und verändert. Die Kommunikationspartner merken dabei nicht, dass sie nicht direkt miteinander sprechen.
MD5	Message Digest Algorithm 5. Hash-Algorithmus, welcher aus einem beliebigen Dateninput eine eindeutige 128 Bit-Repräsentation erstellt.
Mutex	Betriebssystem-Objekt, mit welchem eine Ressource vor gleichzeitigem Zugriff geschützt werden kann.
NET_KEY	Auch 26sec. IPsec Implementierung im 2.6er Linux Kernel. Wurde auch auf Linux 2.4 zurück portiert.
Netlink	Schnittstelle zum Linux-Kernel, welche Sockets für die Kommunikation zwischen User- und Kernel-Space einsetzt.
Nonce	Number used once. Diese Bezeichnung wird vor allem in der Kryptographie verwendet und beschreibt eine Zahl, welche nur einmal benutzt wird. Sie dient dazu um Replay-Attacken zu verhindern.
NPTL	Native POSIX Thread Library. Sehr effiziente POSIX-kompatible Thread-Schnittstelle unter Linux.
Oakley	Oakley ist ein vorgeschlagenes Protokoll zur Aushandlung von gemeinsamen Schlüsseln zwischen authentisierten Kommunikationspartnern auf der Basis des Diffie-Hellman-Austausches.
OpenSSL	Kryptographische Bibliothek für BSD- und Linux-Systeme.
OSCP	Online Certificate Status Protocol. Protokoll zur Status-Abfrage von X.509-Zertifikaten.
Padding	Als Padding werden Füllzeichen bezeichnet.
PKCS#1	Dieser Standard beschreibt ein Public-Key Verfahren auf der Grundlage des RSA-Algorithmus.
PKI	Public-Key Infrastruktur.
PKIX	Public-Key Infrastructure X.509 group. Arbeitsgruppe mit dem Ziel RFCs und andere Standards rund um das Thema PKI und X.509-Zertifikate zu entwickeln.
Pluto	IKE-Daemon der Linux OpenSource Software strongSwan. Der Name stammt aus der griechischen Mythologie.

Ausdruck	Definition und Erklärung
Preshared-Secret	Gemeinsames Geheimnis zweier Parteien. Ein Passwort ist ein Beispiel eines Preshared-Secrets.
PRF	Pseudo random function.
pthreads	POSIX Threads. Standardisierte POSIX-Schnittstelle für die Verwendung von Threads.
Responder	Als Responder wird derjenige Kommunikationspartner bezeichnet, welcher auf eine Anfrage antwortet.
RFC	Request for Comments. Vorschlag eines offenen Standards/Protokolls inklusive dessen genauer Beschreibung.
RSA	Asymmetrisches Kryptosystem benannt nach seinen Erfindern Ronald L. Rivest, Adi Shamir und Leonard Adleman.
SA	Security Association. Eine ausgehandelte „Session" für sichere IPsec-Kommunikation. Fasst Algorithmen und Schlüssel zusammen.
Seed	Ein Seed wird zur Initialisierung einer Pseudo-Random-Funktion verwendet.
SHA1	Secure Hash Algorithm 1. SHA1 ist wie MD5 ein Hash-Algorithmus. Er erstellt aus einem beliebigen Dateninput eine 160-Bit Repräsentation, welche eindeutig ist.
Signatur	Mit einer elektronischen Signatur wird die Integrität und Authentizität elektronischer Daten sichergestellt.
Symmetrische Verschlüsselung	Unter symmetrischer Verschlüsselung wird eine Verschlüsselung mit einem einzigen Schlüssel verstanden. Mit diesem Schlüssel werden die Daten verschlüsselt und können auch wieder entschlüsselt werden. Im Gegensatz zu asymmetrischen Verfahren ist die symmetrische Verschlüsselung um Faktoren schneller. Beispiele dafür sind DES, 3DES und AES.
UML	Unified Modeling Language. Standardisierte Sprache zur Modellierung von Software.
VPN	Virtual Private Network. Ein über ein unsicheres Netzwerk gelegtes virtuelles Netzwerk, welches kryptographisch gesichert ist. Wird oft mit IPsec realisiert.
whack	In strongSwan verwendete Schnittstelle zum IKE-Daemon. Die Kommunikation verläuft dabei über einen Socket.

Ausdruck	Definition und Erklärung
Wiki	Unter einem Wiki versteht man eine Webseite, die von den Benutzern aktiv bearbeitet und mitgestaltet werden kann.
X.509	Der X.509-Standard ist ein Standard der ITU-T für PKIs. Unter anderem wird darin das Format digitaler Zertifikate beschrieben.
XAUTH	IKE Extended Authentication. XAUTH ist eine Erweiterung von IKEv1. Sie erlaubt eine Authentisierung über zwei Phasen und unterstützt unter anderem die weit verbreitete Authentisierung über Benutzername/Passwort.

10.4 Quellenverzeichnis

Kürzel	Information zur Quelle
[Doxygen]	„Doxygen“ http://www.doxygen.org
[DoxygenManual]	Dimitri van Heesch „Doxygen Manual for version 1.4.5“
[Horman04]	Neil Horman „Understanding And Programming With Netlink Sockets“ Version 0.3, Dezember 2004
[IKEAnalysis]	Radia Perlman, Charlie Kaufman „Key Exchange in IPSec: Analysis of IKE“ Dezember 2000
[IKEv2Algs]	Jeffrey I. Schiller „Cryptographic Algorithms for use in the Internet Key Exchange Version 2“ Draft Version 5, April 2004
[IKEv2Clarifications]	P. Eronen, P. Hoffman „IKEv2 Clarifications and Implementation Guidelines“ Draft Version 6, Oktober 2005
[IKEv2Draft]	Charlie Kaufman „Internet Key Exchange (IKEv2) Protocol“ Draft Version 17, September 2004
[IKEv2Linux]	„An IKEv2 implementation for Linux“ http://sourceforge.net/projects/ikev2
[IPsecArch]	S. Kent, K. Keo „Security Architecture for the Internet Protocol“ Draft Version 6, März 2005
[libgmp]	„GNU Multi Precision Arithmetic Library“ http://www.swox.com/gmp
[PKCS1v2.1]	RSA Laboratories „PKCS #1 v2.1 RSA Cryptography Standard“ Version 2.1, 14. Juni 2002
[PThreadLibraries]	„YoLinux Tutorial: POSIX thread (pthread) libraries“ http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

Kürzel	Information zur Quelle
[QuantitativeAnalyses]	H. Soussi, M. Hussain, H. Afifi, D. Seret „IKEv1 and IKEv2: A Quantitative Analyses“
[Racoon2]	„Kame Key Daemon“ ftp://ftp.kame.net/pub/racoon2/
[RFC2104]	H. Krawczyk, M. Bellare, R. Canetti „HMAC: Keyed-Hashing for Message Authentication“ RFC 2104, February 1997
[RFC2407]	D. Piper „The Internet IP Security Domain of Interpretation for ISAKMP“ RFC 2407, November 1998
[RFC2408]	D. Maughan, D. Maughan, M. Schneider, J. Turner „Internet Security Association and Key Management Protocol (ISAKMP)“ RFC 2408, November 1998
[RFC2409]	D. Harkins, D. Carrel „The Internet Key Exchange (IKE)“ RFC 2409, November 1998
[RFC2412]	H. Orman „The OAKLEY Key Determination Protocol“ RFC 2412, November 1998
[RFC2437]	B. Kaliski, J. Staddon „PKCS #1: RSA Cryptography Specifications Version 2.0“ RFC 2437, Oktober 1998
[RFC3526]	T. Kivinen, M. Kojo „More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)“ RFC 3526, Mai 2003
[Thomas03]	Wolfgang Thomas „IPSec Architektur und Protokolle, Internet Key Exchange (IKE)“ Februar 2003
[XineHG]	G. Bartsch, H. Schäfer, R Wareham, M. Freitas, J. Courtier-Dutton,S. Langauf, M. Zühlke, M. Melanson, M. Roitzsch „The xine hacker's guide“ http://xinehq.de/index.php/hackersguide

10.5 Dokumentenverzeichnis

Dokument	Datei auf CD-ROM	Inhalt
Abstract	Abstract.odt	Enthält das Abstract, welches die Diplomarbeit auf eine Seite zusammenfasst.
Aufgabenstellung	Aufgabenstellung.odt	Enthält die Aufgabenstellung, so wie sie vom Betreuer formuliert wurde.
CD-Struktur	CD-Struktur.odt[1]	Beschreibt die Strukturierung und Inhalte der beiliegenden CD-ROM.
Design	Design.odt	Enthält Architektur- und Implementierungs-Details des Daemons `charon`.
Dokumenten-verzeichnis	Dokumentenverzeichnis.odt[1]	Dieses Verzeichnis über die erstellten Dokumente.
Einleitung	Einleitung.odt	Einleitung zur Diplomarbeit. Richtet sich an Ingenieure, welche einen Überblick über die Problemstellung und den Aufbau der Dokumentation gewinnen möchten.
Erfahrungsberichte	Erfahrungsberichte.odt	Beinhaltet die persönlichen Erfahrungsberichte der Diplomanden.
Glossar	Glossar.odt[1]	Beinhaltet das Glossar.
Inbetriebnahme	Inbetriebnahme.odt[1]	Beschreibt in kurzen Worten die Übersetzung und Inbetriebnahme des Daemons.
Management-Summary	ManagementSummary.odt	Enthält das Management-Summary, welches sich an Personen mit sehr wenig Fachkenntnissen richtet.
Programmier-richtlinien	Programmierrichtlinien.odt	Beinhaltet Richtlinien, nach denen der Code programmiert und dokumentiert wurde.
Projektplan	Projektplan.odt	Enthält eine Beschreibung des Vorgehens während der Arbeit und liefert ein Fazit betreffend Projektmanagement.
Projektstand	Projektstand.odt	Beschreibt den Stand des Projektes, erreichte Resultate sowie das weitere Vorgehen. Darin enthalten sind auch die Schlussfolgerungen.
Protokoll vom XY	Protokoll_XY.odt	Protokolle der Sitzungen mit dem Betreuer.
Quellenverzeichnis	Quellenverzeichnis.odt[1]	Auflistung von verwendeten und referenzierten Quellen.
Risikoanalyse	Risikoanalyse.odt	Risikoanalyse, welche die wichtigsten Risiken beschreibt und deren Handhabung erläutert.
Technologien	Technologien.odt	Beschreibt Technologien, die für die Diplomarbeit studiert und verwendet wurden.
Tests	Tests.odt	Beschreibungen der Testverfahren, wie sie im Projekt angewendet wurden.

1 Das Dokument ist Teil des Anhangs.

10.6 Tabellenverzeichnis

10.7 Abbildungsverzeichnis