HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Institute for
Internet Technologies
and Applications

Bachelorarbeit

Userland IPsec für Android 4.0

Autoren:
Giuliano Grassi
Ralf Sager

Betreuer:
Prof. Dr. Andreas Steffen
Tobias Brunner

Hochschule für Technik Rapperswil
Abteilung Informatik

Frühlingssemester 2012

Abstract

IPsec ist normalerweise direkt im Kernel implementiert, während ein Daemon im Userspace für den Schlüsselaustausch über das Internet Key Exchange Protokoll (IKE) zuständig ist. Eine Konsequenz dieser Architektur ist, dass zumindest ein Teil des IKE-Daemons mit Root-Privilegien ausgeführt werden muss, um mit dem IPsec Stack kommunizieren zu können. Dies erschwert den Einsatz von IPsec auf dem Linux-basierten Betriebssystem Android, da Benutzeranwendungen auf vielen Android-Geräten per Default nur eingeschränkte Privilegien haben.

Um dieses Problem zu lösen, wurde im Rahmen dieser Arbeit eine minimale Userspace IPsec-Implementation für Linux und Android als Teil des strongSwan Projektes entwickelt. Diese unterstützt den IP Encapsulating Security Payload (ESP) im Tunnel-Modus zusammen mit den gängigsten kryptographischen Algorithmen. Die für NAT-Umgebungen unabdingbare ESP-in-UDP Encapsulation wird ebenfalls unterstützt. Im Gegensatz zum Linux IPsec Stack verwendet die Userland-Library virtuelle Netzwerk-Interfaces (TUN-Devices) sowie das normale Kernel-Routing zur Steuerung des durch einen Tunnel geschützten Traffics. Des Weiteren wurde ein strongSwan VPN Client für Android entwickelt. Dieser verwendet IKEv2 und erlaubt das Verwalten von mehreren VPN-Profilen. Durch den Einsatz der eigenen IPsec-Implementation sowie einer neuen API in Android 4.0, welche gewisse VPN-bezogene Funktionen zugänglich macht, benötigt dieser IPsec-Client keine erhöhten Privilegien mehr.

Durch die Entwicklung einer Userspace IPsec-Implementation ist es gelungen, die Hürden zum Einsatz von IPsec-basierten VPNs auf Android zu überwinden. Die entstandene Applikation kann ohne Root-Privilegien benutzt werden und funktioniert daher auf allen Android 4-Geräten.

Institute for Internet Technologies and Applications

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz

Bachelorarbeit 2012

Userland IPsec für Android 4.0

Studenten: Giuliano Grassi & Ralf Sager

Betreuer: Prof. Dr. Andreas Steffen

Ausgabe: Montag, 20. Februar 2012

Abgabe: Freitag, 15. Juni 2012

Einführung

Das Institut für Internet Technologien und Anwendungen (ITA) der HSR entwickelt und betreut seit mehreren Jahren die Open Source VPN Lösung *strongSwan* für die Linux, Android, FreeBSD und Mac OS X Betriebssysteme. Normalerweise benötigt die *strongSwan* Anwendung Root-Rechte, damit sie mit dem IPsec Stack des Betriebssystems kommunizieren kann. Dies bedingt, dass Android Geräte „gerootet“ werden müssen, was für den durchschnittlichen Anwender eine ziemlich grosse Hürde bei der Installation darstellt und dadurch die weite Verbreitung von *strongSwan* unter Android stark behindert wird.

Die Einführung einer normierten, socket-basierten Android 4.0 VPN Schnittstelle durch Google gab den Anstoss für diese Bachelorarbeit. Sie hat als Ziel die Erstellung einer Android 4.0 Java App, welch die *strongSwan* libcharon C Library einbindet, um damit IPsec ESP Pakete im Userland ausgehend zu verschlüsseln und eingehend zu entschlüsseln. Damit kann das lästige "Rooten" des Android Gerätes vermieden werden und der Android *strongSwan VPN Client* kann problemlos via App Store bezogen und installiert werden.

Aufgabenstellung

- Einarbeiten in die App-Entwicklung unter Android 4.0 und Kennenlernen der Android VPN Service Schnittstelle.
- Einarbeiten in die strongSwan IKEv2 Softwarearchitektur.
- Erstellen einer strongSwan libipsec Library für Linux und Android, welche die ESP Verschlüsselung und Entschlüsselung im Userland vornimmt. Zu unterstützende Kryptoalgorithmen: AES 128/256 Bit mit SHA-1/SHA-256 HMAC.
- Es soll eine IPsec SA im Tunnelmodus mit ESP-in-UDP Encapsulation in einer NAT-Umgebung unterstützt werden.
- Erstellen einer Java-basierten App, welche die strongSwan C Libraries via JNI einbindet und einer GUI, welche die Eingabe der IKEv2 Verbindungsparameter ermöglicht.
- Die IKEv2 Authentisierung soll eine X.509 Zertifikatskette für die Serverseite und ein EAP-MD5/MSCHAPv2 Passwort für die Clientseite unterstützen.
- Für den Client soll eine virtuelle IP Adresse angefordert werden, während serverseitig das Subnetz 0.0.0.0/0 angenommen werden soll (Kein Split-Tunneling).

Links

- Android 4.0 VPN Interface Definition
  http://developer.android.com/reference/android/net/VpnService.html
- strongSwan Projekt
  http://www.strongswan.org/

Rapperswil, 20. Februar 2012

Prof. Dr. Andreas Steffen

Inhaltsverzeichnis

1. Einleitung

IPsec [22] ist normalerweise im Kernel des Betriebssystems als Teil des IP-Stacks implementiert. Dies erleichtert vor allem die Implementation von IPsec Policies, während zudem eine hohe Performance erreicht werden kann. Ein Nachteil dieser Architektur ist jedoch, dass zumindest ein Teil des Internet Key Exchange (IKE)-Daemons, der im Userspace ausgeführt wird, Root-Privilegien benötigt um den IPsec-Stack im Kernel konfigurieren zu können. Dies erschwert den Einsatz von IPsec auf dem Linux-basierten Smartphone-Betriebssystem Android. Auf vielen Android-basierten Geräten haben Benutzeranwendungen nur eingeschränkte Privilegien. Der Besitzer des Gerätes hat oft keine einfache Möglichkeit, Root-Privilegien zu erhalten.

Um IPsec trotz dieser Einschränkung auf Android verwenden zu können, wurde im Rahmen dieser Arbeit die minimale Userspace IPsec-Implementation `libipsec` entwickelt. `Libipsec` ist Teil des strongSwan-Projektes [1] und kann sowohl unter Linux als auch unter Android verwendet werden. Die Userland-Library ersetzt die Funktion des IPsec Stacks im Kernel. Die Verarbeitung des ESP-Traffics geschieht dabei in separaten Threads im Adressraum des IKEv2-Daemons. Dank einer neuen API unter Android 4.0 [6], welche einige protokollunabhängige, VPN-bezogene Funktionen zugänglich macht, die normalerweise ebenfalls Root-Rechte erfordern, benötigt diese IPsec-Implementation zusammen mit dem strongSwan IKEv2-Daemon auf Android keine Root-Privilegien mehr.

Gleichzeitig wurde ein IPsec VPN Client für Android entwickelt, welche den strongSwan IKEv2-Daemon sowie die Userland IPsec-Implementation verwendet. Diese Android-Applikation ("App") erlaubt es, mehrere IKEv2-Profile über ein Touch User Interface bequem zu verwalten. Da die Applikation keine Root-Rechte benötigt, kann sie problemlos auf allen Android 4-Geräten eingesetzt werden.

1.1. Features

Die folgende Liste enthält die im Rahmen dieser Arbeit umgesetzten Features.

IPsec-Implementation

Die IPsec ESP-Implementation `libipsec` unterstützt folgende Features:

- IP Encapsulating Security Payload (ESP) im Tunnel Mode [21, 22]
  - Verschlüsselung: AES-CBC mit Key-Grössen von 128, 192 oder 256 Bit [13, 27]
  - Authentisierung: HMAC [24] basierend auf Hashfunktionen der SHA-2-Familie sowie SHA-1 [19, 25, 12, 28]
    - HMAC-SHA-1-96
    - HMAC-SHA-256-128
    - HMAC-SHA-384-192
    - HMAC-SHA-512-256
  - Anti-Replay Schutz
- UDP Encapsulation von ESP-Paketen [17] für NAT-Umgebungen
- Beliebige Anzahl von installierten ESP-SAs
- Virtuelle IP-Adressen
- Überwachung der Lifetime von SAs (Zeit) und Auslösen eines Rekeyings
- Verwendet TUN-Devices und die Routing-Funktionalität des Kernels
- Split-Tunneling mittels Routen möglich
- Einbindung in strongSwan 4.6.2 unter Linux (experimentell) und Android

Zu beachten ist, dass die ESP Verschlüsselung und Authentisierung nur gemeinsam verwendet werden kann, da die NULL-Cipher nicht unterstützt wird.

Android-Applikation

Für den strongSwan VPN Client für Android wurden folgende Features implementiert:

- Verwendet die strongSwan IKEv2-Implementation `libcharon` sowie `libipsec` als ESP-Implementation
- Verwaltung von mehreren IKEv2-Profilen
- IKEv2 Authentifizierung des Gateways mittels X.509 Zertifikatsketten. Es werden alle CA-Zertifikate aus dem systemweiten Keystore verwendet. CA-Zertifikate können aber auch manuell installiert und ausgewählt werden.
- IKEv2 Client-Authentifizierung über Benutzername und Passwort (EAP-MD5/EAP-MSCHAPv2)
- Unterstützt eine VPN-Verbindung mit einer Child SA

- Split-Tunneling möglich, muss jedoch auf dem Gateway konfiguriert werden. Per Default wird vom Client ein Subnetz von 0.0.0.0/0 angenommen.
- Konfiguration der über IKE-Attribute erhaltenen DNS Nameserver-Adressen
- Log-View zum Anzeigen des Debug-Outputs von strongSwan

Die Integration von `libipsec` unter Linux ist noch unvollständig. Siehe dazu Kapitel 9, Zukünftige Erweiterungen.

2. Grundlagen

In diesem Kapitel werden die zum Verständnis dieser Arbeit essentiellen Grundlagen zu IPsec sowie der strongSwan-Architektur behandelt.

2.1. IPsec

IPsec ist eine Menge von standardisierten Protokollen, welche es ermöglichen, Netzwerkverkehr auf der IP-Schicht zu schützen [22]. Es beinhaltet die zwei Sicherheitsprotokolle Authentication Header (AH) [20] sowie Encapsulating Security Payload (ESP) [21], die dem Schutz des Nutzverkehrs durch Authentifizierung und Verschlüsselung dienen. Zudem bietet es mit Internet Key Exchange (IKEv1 [14] und IKEv2 [18]) Protokolle zur initialen, gegenseitigen Authentifizierung sowie dem automatischen Key Management.

IPsec kann verwendet werden, um Traffic zwischen zwei Hosts, zwei Gateways oder einem Host und einem Gateway zu schützen. Es gibt zwei grundsätzliche Betriebsmodi: Transport Mode und Tunnel Mode. Im Transport Mode werden direkt die auf IP aufsetzenden Protokolle wie TCP oder UDP geschützt. Im Tunnel Mode hingegen wird ESP und/oder AH auf ganze IP-Pakete angewendet. Es gibt somit zwei IP Header, den äusseren Header sowie den inneren, durch ESP und/oder AH geschützten Header. Zu beachten ist, dass AH immer auch die Integrität von Teilen des äusseren IP-Headers schützt. Bei ESP ist dies nicht der Fall. Da sich diese Arbeit nur mit ESP auseinandersetzt, wird an dieser Stelle nicht näher auf AH eingegangen. Im Tunnel Mode ist es auch möglich, dass im inneren IP-Header andere Adressen (sogenannte virtuelle IP-Adressen) verwendet werden als im äusseren Header. Dies ermöglicht auch die Verwendung von privaten IP-Adressen im inneren Header. Der Tunnel Mode wird häufig als sichere Implementation eines Virtual Private Network (VPN) eingesetzt.

2.1.1. Security Associations

Ein wichtiges Konzept von IPsec ist die Security Association (SA)[22]. Eine SA definiert für eine unidirektionale (simplex-) Verbindung das verwendete Sicherheitsprotokoll (ESP oder AH) sowie alle dazu nötigen Parameter (z.B. kryptographische Algorithmen und Schlüssel). Für typische bidirektionale Verbindungen muss für jede Richtung eine separate SA installiert werden. Jede SA hat zudem einen Security Parameters Index (SPI),

eine beliebige 32-Bit Zahl, welche zur Identifikation der SA dient. Eingehende Pakete können mithilfe der im Header enthaltenen SPI sowie der IP-Zieladresse einer installierten SA zugeordnet werden.

Alle installierten SAs werden in der Security Association Database (SAD) abgelegt. SAD-Einträge können manuell konfiguriert werden. Dazu müssen vorab die nötigen Schlüssel über einen sicheren Kanal ausgetauscht und von Zeit zu Zeit erneuert werden. Da dies umständlich ist, wird stattdessen häufig IKE für das Key-Management eingesetzt.

2.1.2. Internet Key Exchange (IKE)

IKE ist ein eigenes, UDP-basierendes Protokoll, welches zur gegenseitigen Authentifizierung sowie zum Erstellen und Verwalten von SAs dient [18]. Über IKE werden die zu verwendenden Protokolle, Algorithmen und Schlüssel ausgehandelt. Andere Aufgaben von IKE sind beispielsweise NAT Detection und Traversal, die Konfiguration von internen (virtuellen) IP-Adressen sowie von DNS-Servern. Von IKE erstellte ESP- oder AH-SAs werden Child SAs genannt.

IKE erlaubt unterschiedliche Arten der Authentifizierung, beispielsweise durch X.509 Zertifikate. Clients haben auch die Möglichkeit, sich über EAP mit Benutzername und Passwort zu authentisieren (beispielsweise mit EAP-MD5 oder EAP-MSCHAPv2).

Während die Verarbeitung des Nutzverkehrs normalerweise direkt im IP Stack des Kernels geschieht, wird IKE üblicherweise von einem gewöhnlichen, im Userland laufenden Daemon implementiert. ESP- oder AH-Child SAs können dann vom IKE-Daemon meist über eine spezielle Kernel-Schnittstelle verwaltet werden. Zur Verwendung dieser Schnittstelle benötigt der IKE-Daemon normalerweise Root-Privilegien. Unter Linux geschieht die Konfiguration des IPsec Stacks über einen Netlink-Socket [9, 30].

2.1.3. Encapsulating Security Payload (ESP)

ESP [21] kann zur Verschlüsselung und Authentisierung des IP-Traffics eingesetzt werden. Es bietet dem darin gekapselten Protokoll folgende Eigenschaften:

- *Vertraulichkeit*
- *Authentisierung*
- *Integrität*
- *Anti-Replay Schutz*

Vertraulichkeit wird dadurch gewährleistet, dass das gesamte gekapselte Paket verschlüsselt wird und somit für einen Angreifer nicht mehr lesbar ist.

Die Authentisierung sowie Integrität der Daten wird durch einen Message Authentication Code (MAC) gewährleistet, welcher über das gesamte ESP-Paket berechnet wird (ohne den äusseren IP-Header).

Da ESP direkt auf IP aufsetzt, ist es grundsätzlich verbindungslos. Das bedeutet jedoch, dass Massnahmen notwendig sind, um Replay-Attacken zu verhindern. Deshalb enthält jedes ESP-Paket eine Sequenznummer. Der Empfänger kann anhand der Sequenznummer und dem lokal nachgeführten Sliding Window (Anti-Replay Window) doppelt ankommende Pakete verwerfen.

Zu beachten ist, dass je nach Konfiguration der SA nicht all diese Eigenschaften gewährleistet werden müssen.

Wie oben bereits erwähnt setzt ESP normalerweise direkt auf IP auf, es ist jedoch auch möglich, ESP in UDP zu kapseln [17]. Wird ESP in einer NAT (Network Address Translation) Umgebung eingesetzt, ist dies unumgänglich. Dies kommt daher, dass ESP keine Ports verwendet. Wenn mehrere Hosts hinter einem NAT sind und ESP verwenden, kann der NAT-Router somit die von aussen eingehenden Pakete nicht an den korrekten Host hinter dem NAT weiterleiten. Wird ESP jedoch in UDP gekapselt, ist dieses Problem behoben.

Das ESP-Protokoll wird normalerweise direkt im IP Stack des Kernels implementiert. Somit ist es für Userland-Programme komplett transparent (Ausgenommen natürlich für den IKE-Daemon, welcher die ESP-SAs im Kernel konfiguriert).

Weitere Einzelheiten zu ESP sind in Kapitel 3 aufgeführt.

2.2. StrongSwan

StrongSwan (geschrieben als *strongSwan*) ist eine IPsec-Implementation für Linux, FreeBSD sowie Mac OS X [1]. Es wird am Institut für Internet-Technologien und -Anwendungen (ITA) an der Hochschule für Technik Rapperswil (HSR) entwickelt. StrongSwan ist freie Software, lizenziert unter den Bedingungen der GNU GPLv2.

StrongSwan ist in C geschrieben, verwendet jedoch einen objektorientierten Programmierstil basierend auf Structs und Funktionspointern [3].

2.2.1. Architekturübersicht

Im Grunde genommen besteht strongSwan aus den zwei Keying-Daemons *Pluto* (IKEv1) sowie *Charon* (IKEv2). Da in dieser Arbeit nur IKEv2 verwendet wurde, wird auf Pluto nicht näher eingegangen.

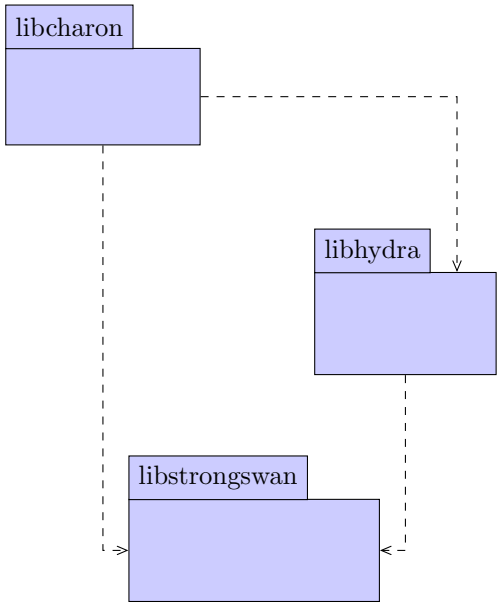

Abbildung 2.1.: Grobe Architektur der relevanten Komponenten von strongSwan

StrongSwan verfügt über ein flexibles Plugin-System mit welchem Funktionalitäten je nach Konfiguration geladen werden können. So sind beispielsweise unterschiedliche kryptographische Algorithmen (z.B. AES) als Plugins implementiert.

Abbildung 2.1 zeigt eine grobe Übersicht über die Architektur der für diese Arbeit relevanten Komponenten von strongSwan [11]:

libcharon
: Beinhaltet den Code des Charon IKEv2 Daemons

libhydra
: Enthält die von Charon verwendeten *Kernel-Interfaces*. Über diese Kernel-Interfaces wird z.B. die SA-Datenbank des Kernels verwaltet.

libstrongswan
: Bildet die Basis für einen grossen Teil des Codes. Enthält unter andrem die kryptographischen Algorithmen, einen Job Scheduler sowie zahlreiche Utilities (z.B. eine Hashtable-Implementation)

2.2.2. Architektur libhydra

Besonders wichtig für diese Arbeit ist die Architektur von `libhydra`. Abbildung 2.2 zeigt diese etwas mehr im Detail.

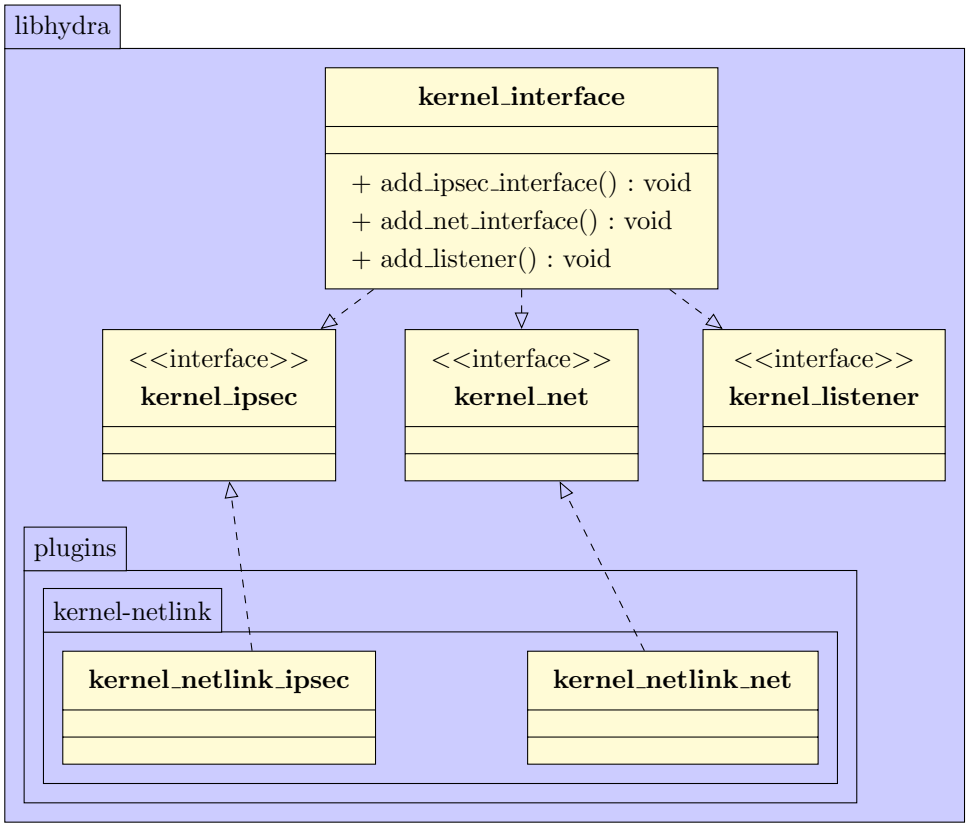

Abbildung 2.2.: Architektur von `libhydra`

kernel_interface
: Diese Klasse dient quasi als Proxy für die Kernel-Interfaces `kernel_ipsec` sowie `kernel_net`. Konkrete Implementationen dieser Interfaces können mit den Methoden `add_ipsec_interface()`/`add_net_interface()` registriert werden. Aufrufe der Methoden dieser Interfaces werden dann an die registrierte Implementation weitergeleitet. Zudem implementiert es das `kernel_listener`-Interface, welches dazu dient, höheren Layern Events vom Kernel weiterzuleiten. Solche treten zum Beispiel auf, wenn die Lifetime einer SA abläuft und der IKE-Daemon ein Rekeying initiieren muss. Der IKE-Daemon registriert dazu Callbacks mit der Methode `add_listener()`.

kernel_ipsec
: Dieses Interface deklariert Methoden zum Verwalten von SAs sowie Policies im Kernel.

kernel_net
: Dieses Interface deklariert Methoden zum Abfragen und Verwalten von Netzwerk-

Adressen, -Routen und -Interfaces.

kernel_listener

Dieses Interface deklariert Methoden welche beim Auftreten von vom Kernel generierten Events aufgerufen werden. Das Interface wird auch in `libcharon` implementiert (und bei `libhydra` registriert), damit Charon über die Events informiert wird.

Die Implementation der `kernel_ipsec` und `kernel_net`-Interfaces ist je nach verwendetem IPsec Stack und unterliegendem Betriebssystem unterschiedlich. Unter Linux beispielsweise werden die Interfaces per Default wie in Abbildung 2.2 gezeigt vom `kernel-netlink`-Plugin implementiert. Dieses Plugin konfiguriert dann den Linux IPsec Stack via einen Netlink-Socket [9, 30]. Das Abfragen der Routingtabelle wird über RT-NETLINK durchgeführt.

3. Analyse ESP-Implementation

Dieses Kapitel beschreibt die grundlegende Funktionsweise von `libipsec`. Die wichtigsten Aspekte von ESP werden analysiert und unterschiedliche Arten der Implementation besprochen. Da `libipsec` im Gegensatz zu anderen IPsec-Implementationen nicht im Kernel, sondern im Userspace ausgeführt wird, mussten einige Funktionen speziell umgesetzt oder weggelassen werden.

`Libipsec` unterstützt nur den für VPNs verwendeten Tunnel Mode [22, 21]. Somit werden komplette IP-Pakete in ESP gekapselt. Momentan unterstützt `libipsec` nur IPv4, kein IPv6. Dies hat primär den Grund, die Implementation in der ersten Phase zu vereinfachen. Die Library ist jedoch ohne Probleme für IPv6-Unterstützung erweiterbar (siehe auch Kapitel 9, Zukünftige Erweiterungen)

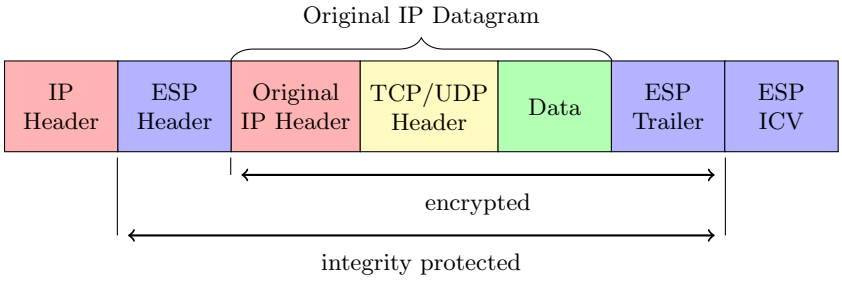

Abbildung 3.1.: ESP im Tunnel Mode

Abbildung 3.1 zeigt die einzelnen Header eines ESP-Paketes im Tunnel Mode. Das Paket hat zwei IP-Header, den originalen Header, welcher in ESP gekapselt wurde und somit den inneren Header darstellt, sowie den neuen, äusseren Header.

Das ESP Paket besteht aus Header, gekapseltem Payload, Trailer sowie einem Integrity Check Value (ICV) am Ende des Paketes. Der gesamte Payload sowie der Trailer ist dabei verschlüsselt. Der innere IP-Header, allfällige TCP/UDP Header sowie die Nutzdaten sind daher für einen Angreifer nicht lesbar. Der ICV ist ein Message Authentication Code (MAC) welcher, wie in Abbildung 3.1 gezeigt, über das gesamte ESP-Paket (ESP-Header sowie den gesamten Ciphertext) ausgenommen dem ICV selbst berechnet wird. Somit kann der Empfänger die Integrität und Authentizität des Paketes verifizieren.

UDP-Encapsulation [17] erlaubt es, ESP auch in NAT (Network Address Translation)

Umgebungen zu verwenden. Da sich die meisten WLANs und Mobilfunknetze hinter einem NAT-Gateway befinden, ist dies für die mobilen Android-Geräte äussert wichtig. Zu beachten ist, dass `libipsec`, so wie sie in strongSwan integriert ist, *ausschliesslich* in UDP gekapselte ESP-Pakete verarbeitet (siehe Abschnitt 3.5, UDP-Encapsulation).

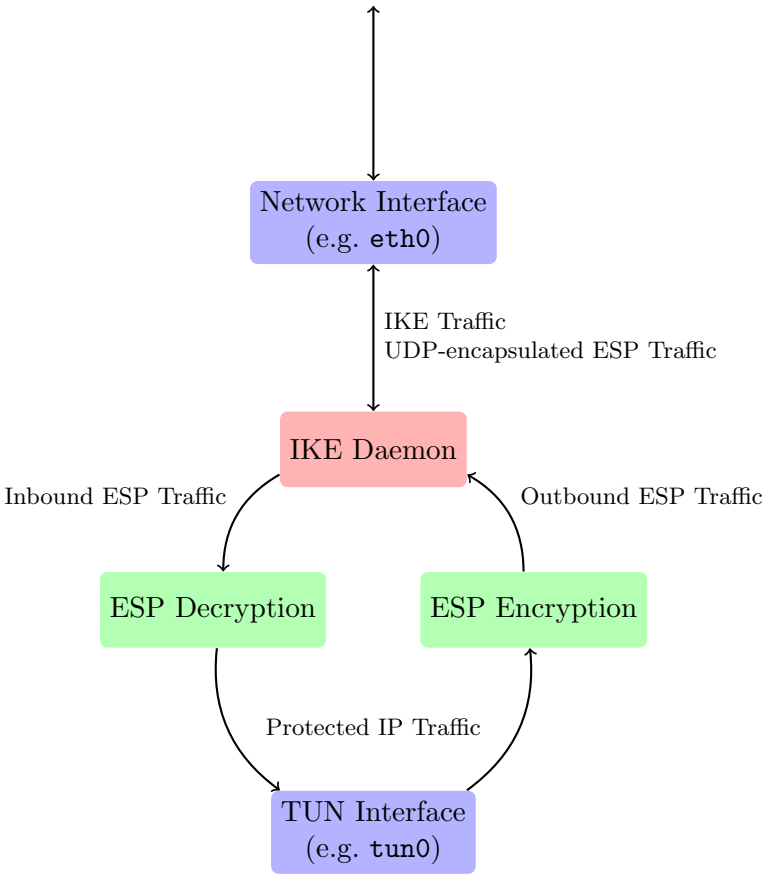

Abbildung 3.2.: Datenfluss zwischen den Komponenten

Abbildung 3.2 zeigt den Datenfluss rund um die ESP-Implementation. Eingehende, in UDP gekapselte ESP-Pakete werden vom Socket des IKE-Daemon empfangen und an `libipsec` weitergeleitet. Von `libipsec` werden die Pakete einer SA zugeordnet und entschlüsselt. Der entschlüsselte Payload, also die inneren IP-Pakete, werden danach auf das TUN-Interface geschrieben. Ausgehende Pakete, welche durch ESP geschützt werden sollen, werden über das TUN-Device geroutet. Diese vom TUN-Device *ausgehenden* Pakete werden verschlüsselt und über den Socket des IKE-Daemons versendet.

Wichtig ist, dass die IPsec-Implementation im Prozess des IKE-Daemons ausgeführt

wird. Die Verarbeitung der ESP-Pakete geschieht jedoch in separaten Threads. Die Bezeichnung "IKE-Daemon" bezieht sich also nicht auf einen separaten Prozess, sondern lediglich auf die Komponente von strongSwan, welche das IKE-Protokoll implementiert.

3.1. TUN-Interface

Da `libipsec`, anders als viele IPsec-Implementationen nicht direkt im IP-Stack des Kernels integriert ist, können Pakete, welche durch IPsec geschützt werden sollen nicht einfach abgefangen werden. Zudem können die entschlüsselten IP-Pakete nicht einfach wieder in den IP-Stack eingespeist werden.

Als Lösung für diese Probleme bietet sich der TUN/TAP-Device Treiber [23] an. Dieser erlaubt es, virtuelle Netzwerk-Interfaces (TUN/TAP Devices) zu öffnen. Ein TUN/TAP Device funktioniert wie ein normales Netzwerk-Interface, mit dem Unterschied, dass es komplett in Software implementiert ist. TAP Devices simulieren dabei ein Interface auf Layer 2 (z.B. Ethernet), während TUN Devices reine Layer 3-Interfaces sind (IPv4, IPv6, ...). Da über IPsec *ausschliesslich* IP-Pakete getunnelt werden [22, 21], wird von `libipsec` ein TUN-Device verwendet.

Userland-Programme können über `/dev/net/tun` auf den TUN/TAP Treiber zugreifen und ein Interface allozieren. Pakete, welche über das TUN/TAP-Interface gesendet werden, können dann vom Userland-Programm gelesen und verarbeitet werden. Pakete können vom Programm auch auf das Interface geschrieben werden. So können sie in den IP-Stack eingespeist werden, wo sie normal verarbeitet werden.

Ein Vorteil der TUN-Devices ist, dass diese die Verwendung der normalen Routing-Funktionalität des Kernels erlauben. So können statische Routen installiert werden, welche ein TUN-Device als ausgehendes Interface referenzieren. Abschnitt 3.7 beschreibt die Verwendung solcher Routen durch `libipsec`.

`Libipsec` verwendet jeweils ein TUN-Device für jedes vom IKE-Daemon installierte SA-Paar. Anhand des TUN-Devices, von welchem ein Paket ausgeht, kann somit direkt die dazugehörige outbound IPsec-SA ermittelt werden. Umgekehrt enthält jede inbound SA eine Referenz auf das dazugehörende TUN-Device.

3.1.1. Maximum Transmission Unit

Die Maximum Transmission Unit (MTU) eines TUN-Devices bestimmt die Länge der Pakete, die über einen Tunnel gesendet werden. Um die Fragmentation von ESP-Paketen zu verhindern, wurde eine MTU von 1400 Bytes gewählt. Aus Tabelle 3.1 ist ersichtlich, dass der maximale Overhead, der durch die ESP-Kapselung entsteht, 99 Bytes ist. Die Berechnung bezieht sich dabei auf die von `libipsec` unterstützten Algorithmen.

Beschreibung	Länge (Bytes)
Äusserer IP Header	20
UDP-Encapsulation Header	8
ESP Header	8
ESP IV (AES-CBC)	16
ESP Padding & Trailer	15
ESP ICV (HMAC-SHA-512-256)	32
Total	**99**

Tabelle 3.1.: Maximaler ESP-Overhead

Da vom TUN-Device ausgehende Pakete aufgrund der MTU auf 1400 Bytes Länge beschränkt sind, werden ausgehende, in UDP gekapselte ESP-Pakete somit nie grösser als 1499 Bytes, was die Fragmentierung bei den meisten Verbindungen knapp verhindert. Zu beachten ist, dass der Overhead typischerweise kleiner ist, da anstelle von HMAC-SHA-512-256 meist ein anderer ICV verwendet wird, beispielsweise HMAC-SHA-1-96 (12 Bytes) oder HMAC-SHA-256-128 (16 Bytes). Falls trotzdem nötig kann die MTU des TUN-Devices unter Linux auch noch während dem Betrieb geändert werden.

In Zukunft könnte durch Path MTU Discovery die MTU einer Verbindung ermittelt werden. Durch Abzug des von der SA abhängigen Overheads liesse sich die optimale MTU für das TUN-Device berechnen und konfigurieren.

3.2. Paketformat

Abbildung 3.3 zeigt das Format eines ESP-Paketes wie es in RFC 4303 [21] beschrieben ist.

Der ESP-Header besteht aus dem SPI der SA, zu welcher das ESP-Paket gehört sowie der Sequenznummer. Der SPI darf dabei nie den Wert 0 haben (siehe auch Abschnitt 3.5). Die Sequenznummer wird bei jedem ESP-Paket um eins erhöht.

Auf den Header folgt der Initialisierungsvektor, falls ein solcher vom verwendeten Verschlüsselungsverfahren benötigt wird. Da `libipsec` nur AES im CBC-Modus [13, 27] unterstützt, ist dieses Feld immer vorhanden. Die Payload-Daten enthalten im Tunnel Mode das gekapselte IP-Paket. Das Padding zusammen mit dem Pad Length- und dem Next Header-Feld bilden den ESP Trailer. Das Next Header-Feld beschreibt den Typ der Payload-Daten, beispielsweise `IPPROTO_IPIP` (4) für IPv4.

Octet	0	1	2	3
0	Security Parameters Index (SPI)			
4	Sequence Number			
...	Initialization Vector (IV, variable, optional)			
...	Payload Data (variable)			
...				
...		Padding (0-255 bytes)		
...			Pad Length	Next Header
...	Integrity Check Value (ICV, variable)			
...				

Abbildung 3.3.: ESP-Paketstruktur

Das ICV-Feld enthält den Message Authentication Code der nach der Verschlüsselung über den Rest des Paketes berechnet wird. Dieser bietet Integritätsschutz sowie Authentisierung für das ESP-Paket. Die Länge ist abhängig vom Algorithmus. Von `libipsec` unterstützt werden HMACs [24] basierend auf SHA-1 sowie der SHA-2-Familie [12, 28]. Der SHA-1 basierte HMAC wird dabei auf 96 Bits gekürzt (HMAC-SHA-1-96 [25]). Bei den SHA-2 basierten HMACs wird jeweils nur die Hälfte des Outputs verwendet: HMAC-SHA-256-128, HMAC-SHA-384-192 und HMAC-SHA-512-256 [19]).

StrongSwan hat bereits Implementationen der verwendeten kryptographischen Algorithmen für die Verschlüsselung und Authentisierung. Diese wurden auch für die ESP-Implementation benutzt.

RFC 4303 [21] erlaubt es, Verschlüsselung und Authentisierung auch einzeln zu aktivieren. So ist es zum Beispiel möglich, eine SA zu konfigurieren welche nur Authentisierung und Integritätsschutz, aber keine Vertraulichkeit bietet. `Libipsec` unterstützt dieses Verhalten nicht. Verschlüsselung und Authentisierung können nur gemeinsam verwendet werden.

3.3. Verarbeitung eingehender Pakete

Für jedes eingehende ESP-Paket wird als erstes anhand der SPI und der Zieladresse des Paketes die passende SA gesucht. Falls keine passende SA existiert, wird das Paket verworfen. Andernfalls wird danach die Sequenznummer des Paketes verifiziert, um Replay-Attacken zu verhindern. Da IP verbindungslos ist, kann es sein dass die Pakete in einer anderen Reihenfolge empfangen werden als sie gesendet wurden. Pakete können auch unterwegs verloren gehen. Ein einfacher Vergleich mit der lokal nachgeführten Sequenznummer ist deshalb nicht ausreichend. Um Pakete, die nicht der Reihe nach ankommen

verarbeiten zu können enthält jede inbound SA deshalb zusätzlich zur Sequenznummer ein Sliding Window einer bestimmten Grösse (bei `libipsec` per Default 128). Für eine genaue Beschreibung des Anti-Replay Windows siehe RFC 4303 [21].

Ist die Verifizierung der Sequenznummer erfolgreich, wird die Integrität des Paketes durch Verifizierung des ICV überprüft. Schlägt diese fehl, wird das Paket verworfen. Ist die Verifizierung erfolgreich, wird die lokale Sequenznummer und das Anti-Replay Window nachgeführt. Diese zweistufige Verifizierung hat den Vorteil, dass Pakete mit einer ungültigen Sequenznummer schneller verworfen werden können, ohne dass eine (langsame) kryptographische Operation ausgeführt werden muss.

Der Payload des Paketes wird danach entschlüsselt, allfälliges Padding überprüft und dann entfernt. Zeigt das Next Header Feld an, dass es sich beim Payload um ein IPv4-Paket handelt, wird das zur SA gehörende TUN-Device gesucht und der Payload darauf geschrieben. Dummy- oder Pakete anderer Protokolle hingegen werden verworfen.

Die ESP-Implementation selbst inspiziert die gekapselten Pakete *nicht*. Pakete mit ungültigen Headern werden jedoch durch den IP-Stack des Kernels nach dem Schreiben auf das TUN-Device automatisch verworfen.

3.4. Verarbeitung ausgehender Pakete

Wird ein ausgehendes Paket von einem TUN-Device gelesen, wird als Erstes die zum TUN-Device gehörende outbound SA gesucht. Ist diese gefunden, wird das Paket in ein ESP-Paket gekapselt.

SPI und Sequenznummer des Paketes werden von der SA bestimmt. Die Sequenznummer wird dabei für jedes Paket um eins erhöht.

Dann wird der Payload des Paketes mit dem durch die SA spezifizierten Algorithmus und Schlüssel verschlüsselt. Allenfalls muss der Payload vor der Verschlüsselung durch Padding ergänzt werden, damit die Länge einem Vielfachen der Blockgrösse des verwendeten Algorithmus entspricht.

Über das gesamte Paket wird am Ende der ICV berechnet und angehängt. Danach wird das Paket an die Zieladresse der SA versendet.

3.5. UDP-Encapsulation

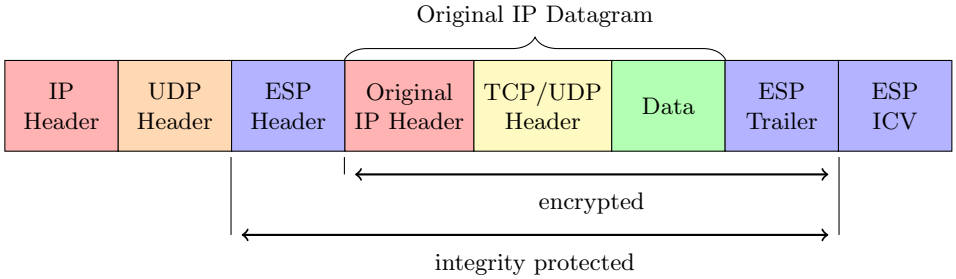

Abbildung 3.4.: ESP im Tunnel Mode, in UDP gekapselt

Wie bereits erwähnt ist die Kapselung von ESP in UDP [17] nötig, damit IPsec auch in NAT-Umgebungen verwendet werden kann. Da ESP selbst keine Ports verwendet, haben NAT-Router keine Möglichkeit, eingehende Pakete einem Host hinter dem NAT zuzuordnen. Die Kapselung in UDP behebt dieses Problem.

Octet	**0**	**1**	**2**	**3**
0	Source Port		Destination Port	
4	Length		Checksum	
8	ESP Header			
...				

Abbildung 3.5.: In UDP gekapselter ESP-Header

Wie in Abbildung 3.5 gezeigt, folgt der ESP-Header direkt auf den UDP-Header. Für die ESP-Pakete werden dabei die selben Ports wie für IKE in NAT-Umgebungen verwendet (4500). Dies bedingt, dass IKE-Messages von ESP-Paketen unterschieden werden können. Dazu enthalten IKE-Messages als erstes Feld einen 4 Byte grossen Non-ESP-Marker, der aus dem Wert 0 besteht. An dieser Stelle wäre bei einem ESP-Paket der SPI. Da dieser nie 0 sein darf, kann der Traffic anhand dieses Feldes unterschieden werden.

`Libipsec` empfängt und sendet den gesamten ESP-Traffic über den UDP-Socket des IKE-Daemons. Dieser ist bereits in der Lage, ESP-Traffic zu separieren. Zudem übernimmt der IKE-Daemon auch das senden von NAT-Keepalive Paketen. Für die UDP-Encapsulation musste deshalb in dieser Arbeit nichts mehr entwickelt werden.

Octet	0	1	2	3
0	Source Port		Destination Port	
4	Length		Checksum	
8	Non-ESP Marker			
12	IKE Header			
...				

Abbildung 3.6.: IKE Header NAT-T-Port (4500) mit Non-ESP-Marker

3.6. SA Lifetime

Jede IPsec SA hat eine limitierte Lebensdauer. Läuft diese ab, muss die SA gelöscht werden. Die Lebensdauer kann (gleichzeitig) auf zwei Arten spezifiziert sein [22]: als Zeit oder als Anzahl übertragener Bytes. `Libipsec` unterstützt jedoch nur die erste Variante.

Dabei gibt es zwei Arten einer Lebensdauer: eine Soft-Lifetime und eine Hard-Lifetime. Läuft die Soft-Lifetime ab, wird dies dem IKE-Daemon mitgeteilt. Dieser führt dann ein Rekeying durch und ersetzt die alte SA durch eine neue SA. Schlägt dies fehl, wird die SA nach Ablauf der Hard-Lifetime automatisch gelöscht.

3.7. Policies und Selektoren

IPsec definiert eine Grenze zwischen geschützten und ungeschützten Interfaces [22]. Anhand von Policies wird entschieden, wie Pakete, die diese Grenze überschreiten, verarbeitet werden. Diese Policies werden in der Security Policy Database (SPD) abgelegt. Pakete können verworfen (`DISCARD`), durch IPsec (ESP oder AH) geschützt (`PROTECT`) oder ohne Veränderung weitergereicht werden (`BYPASS`). Jede Policy enthält ein oder mehrere Selektoren, die den Traffic, auf den die Policy angewendet wird, spezifizieren. Mögliche Selektoren sind beispielsweise:

- Remote IP-Adressen (Ranges)
- Lokale IP-Adressen (Ranges)
- Protokoll des nächsten Layers (z.B. TCP)
- Ports des next-Layer Protokolls

In einer Userland-Implementation von IPsec ist die Implementation aller Selektoren aufwendig. Für alle Selektor-Typen ist das Parsen des IP-Headers nötig, für Port-Selektoren auch noch das Parsen des TCP- oder UDP-Headers.

`Libipsec` beschränkt sich deshalb auf die häufig verwendeten Adress-Selektoren. Diese werden in gewöhnliche Routen umgewandelt, die dann in der Routingtabelle des Kernels eingetragen werden. Soll zum Beispiel der gesamte Traffic zum Netzwerk `192.168.1.0/24` durch ESP geschützt werden, wird auf dem TUN-Device, welches der entsprechenden Child SA zugeordnet ist, eine Route mit Ziel-Netz `192.168.1.0/24` installiert.

Der grosse Vorteil dieser Lösung ist, dass die Implementation wesentlich vereinfacht wird. Da der Kernel das Routing komplett übernimmt, muss nichts weiter gemacht werden als die Routen zu installieren. So wird das aufwendige Parsen des IP-Headers vermieden. Ein weiterer Vorteil dieser Lösung ist, dass Benutzer theoretisch nach Verbindungsaufbau selbst Routen bearbeiten oder hinzufügen können, um zu steuern, welcher Verkehr geschützt wird. Das Routing im Kernel bringt zudem einen Performance-Vorteil gegenüber einer reinen Userland-Lösung.

Ein Nachteil der Lösung ist die eingebüsste Flexibilität. Nur mit Routen können beispielsweise keine Port-basierten Policies verwendet werden. Theoretisch wäre es jedoch denkbar, dass mit Hilfe des Netfilter Paketfilter-Frameworks Regeln installiert werden, die auch das Port-Matching ermöglichen. Ein weiterer, wesentlicher Nachteil der Lösung mit TUN-Devices ist, dass die Selektoren erst aktiviert werden, wenn eine Child SA erfolgreich erstellt wurde und somit ein TUN-Device eingerichtet wird. Deshalb können SAs auch nicht erst bei Bedarf erstellt werden, wie dies mit dem IPsec Stack des Kernels möglich ist.

Explizite Discard-Policies werden von `libipsec` nicht unterstützt. Bei ausgehendem Traffic werden durch das Routing ohnehin nur Pakete auf das TUN-Interface geschrieben, wenn eine entsprechende Route installiert wurde.

3.7.1. Bypass-Policy

Eine spezielle Art der Policy ist die Bypass-Policy. Diese wird unter anderem dazu verwendet, um zu verhindern, dass vom Host generierte IKE- oder ESP-Pakete durch IPsec geschützt werden. Dies würde sonst unter Umständen zu Routing-Loops führen.

Die Bypass-Policy kann mit der Socket-Option `SO_BINDTODEVICE` [10] realisiert werden. Mit dieser Option lässt sich ein Socket an ein spezifisches Netzwerk-Interface binden, so dass Pakete, die über den Socket gesendet werden, immer über dieses Interface gesendet werden. Zudem werden nur Pakete, die auf diesem Interface empfangen werden, an den Socket weitergegeben. Somit kann der für ESP und IKE verwendete UDP-Socket vor der Installation des TUN-Devices an das momentane Default-Interface gebunden werden. Dies verhindert, dass Pakete, welche über den Socket gesendet werden, je über das TUN-Device geroutet werden, was zu dem oben erwähnten Loop führen würde.

4. Architektur ESP-Implementation

In diesem Kapitel wird die Softwarearchitektur der entwickelten Userland IPsec Implementation `libipsec` beschrieben.

`Libipsec` enthält die Komponenten der ESP-Implementation, welche nicht bereits in anderen Teilen von strongSwan implementiert wurden. Neben der Verarbeitung von ein- und ausgehenden ESP-Paketen ist die `libipsec` auch für die Verwaltung der IPsec SAs, Policies und den TUN-Devices zuständig. Da all diese zu einem beliebigen Zeitpunkt durch den IKE-Daemon installiert oder gelöscht werden können, werden sie nur loose miteinander verknüpft. Das bedeutet, dass beispielsweise ein TUN-Device nicht direkt einen Pointer auf das zugehörige SA-Paar hat. Stattdessen sind die SAs, TUN Devices und Policies über eine ID (`reqid`) verknüpft, welche vom IKE-Daemon gewählt wird. Ein SA-Paar hat jeweils dieselbe `reqid` wie alle dazugehörigen Policies und das entsprechende TUN-Device.

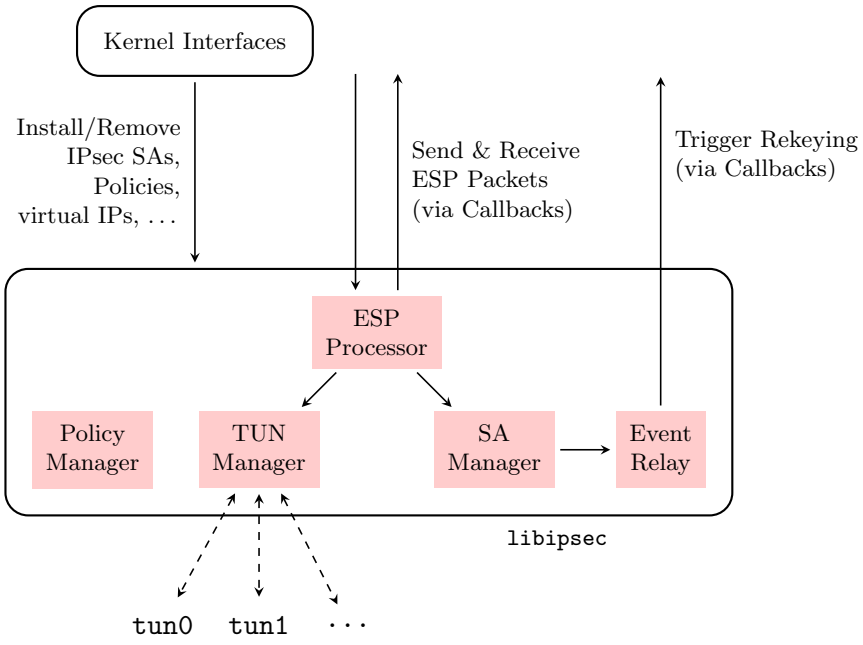

Abbildung 4.1.: Architektur von `libipsec`

Abbildung 4.1 zeigt die Architektur mit den wichtigsten Komponenten. Da Zugriffe von verschiedenen Threads erfolgen, wurden alle kritischen Bereiche der Library Thread-Safe implementiert. In den folgenden Unterkapiteln werden die einzelnen Komponenten genauer beschrieben.

4.1. ESP Processor

Der ESP-Processor ist zuständig für die Verarbeitung der ein- und ausgehenden Pakete. Dazu wird je ein Thread (genannt Job) aus dem Thread-Pool der `libstrongswan` für ein- und ausgehenden Traffic alloziert. Er enthält zudem eine synchronisierte Queue für die eingehenden ESP-Pakete.

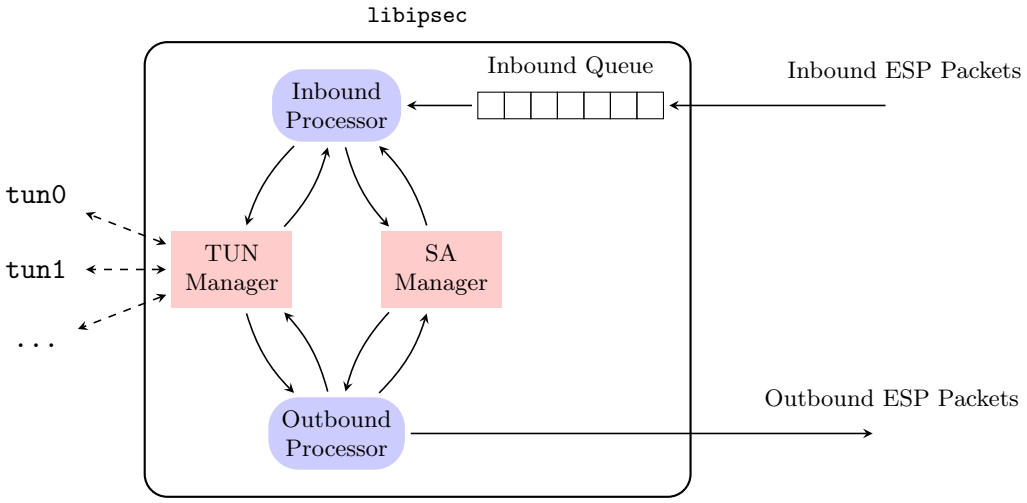

Abbildung 4.2.: Verarbeitung der ESP-Pakete in `libipsec`

Abbildung 4.2 zeigt die Paketverarbeitung in `libipsec` mit den beteiligten Komponenten und den beiden Threads des ESP-Processors (In- und Outbound Processor Thread).

Eingehende ESP-Pakete werden in die Inbound Queue geschrieben. Von dort werden sie vom *Inbound Processor*-Thread ausgelesen und verarbeitet. Die zu einem eingehenden Paket passende IPsec SA wird vom *SA Manager* ausgecheckt. Das Paket wird dann wie in Abschnitt 3.3 beschrieben verarbeitet. Wird die SA nicht mehr gebraucht, wird sie wieder eingecheckt. Nachdem ein Paket authentifiziert und entschlüsselt wurde, sucht der *TUN-Manager* das zur SA passende TUN-Device und das Paket wird darauf geschrieben.

Ausgehende Pakete werden vom *Outbound-Processor*-Thread verarbeitet. Dieser blockiert, bis er vom *TUN-Manager* ein TUN-Device erhält auf welchem gerade ein Paket versendet wurde. Das Paket wird ausgelesen und die zum TUN-Device passende IPsec SA wird vom *SA-Manager* ausgecheckt. Nachdem das Paket wie in Abschnitt 3.4 beschrieben verschlüsselt wurde, wird die SA wieder eingecheckt und das ESP-Paket versendet.

Damit Pakete gesendet und empfangen werden können, müssen zuvor zwei Callbacks registriert werden. Ein Sender Callback, der bei `libipsec` registriert werden muss, ist dafür zuständig, die ESP-Pakete über den UDP-Socket des IKE-Daemons zu versenden. Umgekehrt ist ein Receiver Callback, welcher beim IKE-Daemon registriert werden muss, dafür zuständig, die auf dem UDP-Socket eingehenden ESP-Pakete in die Inbound Queue zu schreiben. Der Grund für die Verwendung einer Queue ist, dass dieser Callback sehr schnell zurückkehren sollte und daher keine aufwendigen Operationen durchführen kann.

Bei Bedarf könnte der ESP-Processor so erweitert werden, dass mehr als zwei Threads eingesetzt werden. Da alle kritischen Bereiche bereits synchronisiert sind, müsste lediglich die Verwaltung der Threads (Start/Stop, Konfiguration der Anzahl) zusätzlich implementiert werden.

4.2. Policy Manager

Der Policy Manager dient der Verwaltung der IPsec Policies. Bei der eigentlichen Paketverarbeitung durch den ESP-Processor werden der Policy Manager und die darin gespeicherten Policies jedoch nicht direkt konsultiert. Stattdessen werden beim Aufsetzen des TUN-Devices die dazugehörigen Policies einmalig vom Manager abgefragt. Diese Policies werden dann in Routen umgewandelt und in der Routingtabelle des Kernels installiert. Somit stellt das Routing bereits sicher, dass der gewünschte Traffic über das entsprechende TUN-Device geleitet wird um durch ESP geschützt zu werden.

4.3. SA Manager

Der SA Manager ist die Implementation der Security Association Database (SAD) [22]. Er verwaltet die IPsec SAs und regelt den Zugriff auf diese. Zudem überwacht der SA Manager die Lebensdauer der SAs in einem separaten Thread. Läuft das Soft-Limit einer SA ab, wird über das Event Relay ein Rekeying ausgelöst. Der IKE-Daemon handelt dann eine neue SA aus, installiert diese und löscht die alte SA. Falls dies nicht geschieht, wird die SA beim Erreichen des Hard Limits automatisch gelöscht.

Da zu jeder Zeit von einem beliebigen Thread SAs installiert oder gelöscht werden können, muss sichergestellt sein dass Zugriffe auf den SA Manager synchronisiert werden. Ein besonders Problem stellt dabei das Löschen einer SA dar. Nach dem Entfernen der

SA aus der SAD muss sichergestellt werden, dass kein Thread mehr auf die SA zugreift bevor diese zerstört und der entsprechende Speicherbereich freigegeben werden kann.

Zur Lösung dieses Problems wurden zwei mögliche Lösungen evaluiert: *Reference Counting* für SAs sowie ein *Checkin/Checkout-System* für exklusiven Zugriff auf SAs:

Reference Counting
: Bei der *Reference Counting*-Lösung hat jede SA einen Zähler, welcher die Anzahl der Referenzen auf diese SA enthält. Wird eine SA von einem Thread vom SA-Manager geholt, wird der Zähler um eins inkrementiert. Wird eine SA nicht mehr verwendet, wird der Zähler dekrementiert. Sobald dieser Null erreicht, wird die SA zerstört und der Speicher freigegeben. Solange der SA Manager oder ein Thread, welcher Pakete verarbeitet noch eine Referenz auf die SA hat, ist sichergestellt dass diese nicht gelöscht wird.

Checkin/Checkout-System
: Das *Checkin/Checkout-System* synchronisiert Zugriffe auf gesamte SAs. Benutzer müssen eine SA zur Verwendung auschecken. Der SA-Manager stellt sicher, dass eine SA nicht mehrmals gleichzeitig ausgecheckt werden kann. Somit ist der exklusive Zugriff auf eine SA sichergestellt. Versucht ein Thread eine SA auszuchecken, solange diese noch von einem anderen Thread verwendet wird, blockiert der Aufruf bis die SA vom zweiten Thread wieder eingecheckt wurde. Soll eine SA gelöscht werden, wird allen Threads, welche auf die SA warten signalisiert, dass diese gelöscht werden soll. Diese geben dann das Warten auf. Neue Threads versuchen gar nicht erst, auf die SA zu warten. Sobald also keine Threads mehr auf die SA warten, wird diese entfernt und zerstört.

Reference Counting hat gegenüber dem Checkin/Checkout-System den Vorteil, dass grundsätzlich mehrere Threads Lesezugriff auf eine SA haben können. Zudem ist Reference Counting einfach zu implementieren. Da jedoch momentan nur je ein Thread für die Verarbeitung von ein- und ausgehenden Paketen eingesetzt wird (siehe Abschnitt 4.1), wird auf dieselbe SA nur von einem Thread aus zugegriffen und es würde kein Performance-Vorteil erreicht werden. Ausserdem müsste jede SA zusätzlich ein Lock enthalten, welcher zur Synchronisation von Zugriffen auf die kritischen veränderlichen Daten wie den Sequence Number Counter oder das Anti-Replay Window dient. Da die Verifizierung der Sequenznummer bei eingehenden SAs wie in Abschnitt 3.3 zweistufig implementiert wurde, wäre die Synchronisation einzelner Aufrufe auf der SA zudem ungenügend, d.h. die SA müsste manuell gesperrt werden, solange die kritischen Daten verwendet werden.

Das Checkin/Checkout-System hingegen hat den Vorteil, dass ein Thread exklusiven Zugriff auf eine SA erhält. Dies garantiert, dass die Veränderung kritischer Daten wie dem Anti-Replay Window in der korrekten Reihenfolge erfolgt und keine Konflikte entstehen. Ein Lock pro SA ist somit nicht notwendig, lediglich der SA-Manager benötigt ein Lock zur Synchronisation von Zugriffen. Ein Nachteil des exklusiven Zugriffs ist jedoch, dass Threads eventuell auf eine SA warten müssen. Daher sollten SAs nur möglichst kurz aus-

gecheckt werden. Langsame Operationen sollten wenn möglich nicht ausgeführt werden, während dem eine SA ausgecheckt ist. Dafür ist nach Rückkehr eines Funktionsaufrufes zum Löschen einer SA sichergestellt, dass diese auch wirklich gelöscht ist. Im Falle von Reference Counting könnte es hingegen sein, dass diese noch irgendwo verwendet wird.

Da momentan der ESP Processor wie erwähnt nur zwei Threads verwendet, ist es nicht nötig, dass zwei Threads gleichzeitig auf dieselbe SA zugreifen müssen. Zudem wäre der Performance-Vorteil der Reference-Counting Lösung vermutlich minimal, da trotzdem zeitweise exklusiver Zugriff auf eine SA benötigt wird. Zudem wäre der zukünftige Einsatz von mehr als zwei Threads vor allem dann sinnvoll, wenn viele SAs verwendet werden. In diesem Fall würden sich die Threads auch nicht gross gegenseitig behindern, da nur selten mehrere Threads Pakete derselben SA verarbeiten würden.

Aus diesen Gründen wurde der SA-Manager mit einem *Checkin/Checkout-System* zur Regelung des Zugriffs auf die IPsec SAs implementiert.

4.4. TUN Manager

Der TUN-Manager ist zuständig für die Verwaltung aller TUN-Devices. Der TUN-Manager wird vom ESP-Processor benutzt, um auf Pakete zu warten, welche über ein TUN-Device gesendet werden. Dazu benutzt der TUN-Manager ein `epoll`-Set [8] mit allen Filedeskriptoren der registrierten TUN-Devices. Zudem bietet er dem ESP-Processor die Möglichkeit, nach TUN-Devices anhand der `reqid` zu suchen. Dies wird benötigt, um die eingehenden Pakete auf das richtige TUN-Device zu schreiben.

Ähnlich wie der SA-Manager benötigt auch der TUN-Manager ein System zur Kontrolle des Lebenszyklus der TUN-Devices. Anders als beim SA-Manager wurde dies jedoch mit *Reference Counting* gelöst. Dazu wird mit einer TUN-Referenz gearbeitet, welche zusätzlich zum TUN-Device selbst noch den Referenzzähler enthält. Erreicht dieser Null, wird das TUN-Device geschlossen und der Speicherbereich freigegeben. Da mehrere Threads den Referenzzähler verändern müssen, ist der Zugriff darauf mit einem Mutex synchronisiert.

Diese Lösung hat für TUN-Devices einige wesentliche Vorteile. TUN-Devices erlauben grundsätzlich simultanen Zugriff von mehreren Threads, da sie keinerlei veränderliche Daten enthalten. Alle Schreibzugriffe wie das Einspeisen eines Paketes, das Setzen der MTU oder einer Interface-Adresse erfolgen über Systemcalls wie `write(2)` oder `ioctl(2)` und müssen daher nicht synchronisiert werden. Anders als bei den SAs, wo immer eine separate in- und outbound SA vorhanden ist, müssen die Threads des ESP-Processors beide auf dieselben TUN-Devices zugreifen. Das bedeutet dass es sehr wichtig ist, dass beide Threads gleichzeitig Pakete vom TUN Device lesen und darauf schreiben können. Dies wäre mit einem Checkin/Checkout-System wie beim SA-Manager nicht möglich.

4.5. Event Relay

Das Event Relay dient dazu, ein Rekeying durch den IKE-Daemon auszulösen, falls die Lebensdauer einer IPsec SA abgelaufen ist. Dazu können beim Event Relay Handler (Callbacks) registriert werden.

5. Integration der ESP-Implementation

Dieses Kapitel beschreibt, wie die `libipsec` ESP-Implementation in die strongSwan Architektur integriert wurde und wie die Kernel-Interfaces unter Linux und Android implementiert wurden.

Wie in Kapitel 2, Grundlagen bereits beschrieben wurde, wird die IPsec-Implementation von strongSwan über das Kernel-Interface `kernel_ipsec` angesprochen. Dieses deklariert unter anderem Funktionen zum Verwalten von SAs und Policies. Sowohl unter Android wie auch unter Linux wurde deshalb eine Implementation des `kernel_ipsec` Interfaces geschrieben, welche `libipsec` verwendet. Der Begriff "Kernel-Interface" ist in diesem Fall etwas irreführend, da sich die IPsec-Implementation ja nicht im Kernel, sondern im Userland befindet. Ebenfalls unter Linux und Android implementiert ist die Funktionalität zum Verwalten von virtuellen IP-Adressen des `kernel_net` Interfaces. Da die Implementationen der Interfaces unter Linux und Android zwar unterschiedlich, aber sehr ähnlich sind, wurde versucht, einen grossen Teil der Funktionalität nicht im Kernel-Interface, sondern direkt in `libipsec` zu implementieren. Die meisten Funktionen der Kernel-Interfaces leiten die Aufrufe deshalb direkt an `libipsec` weiter.

5.1. Integration unter Android

Android stellt ab Version 4.0 die Java-Klasse `VpnService` [6] zur Verfügung. Diese bildet eine Schnittstelle zu protokollunabhängigen VPN-bezogenen Funktionen, welche sonst nur mit Root-Rechten zugänglich sind. Diese Funktionen sind essentiell für die Integration der Userspace IPsec-Implementation unter Android.

Die Klasse enthält die Methode `protect()`, welche einen Socket mit der `SO_BINDTODEVICE`-Option an das aktuelle Default-Netzwerkinterface bindet. Mit dieser Methode kann, wie in Abschnitt 3.7.1 beschrieben, die Bypass-Policy realisiert werden. Ausserdem enthält `VpnService` eine Hilfsklasse `VpnService.Builder`, welche es Applikationen ermöglicht ein TUN-Device aufzusetzen. Bevor das TUN-Interface erstellt wird, können auf einer Instanz der `VpnService.Builder`-Klasse Interface-Adressen, Routen, DNS-Server sowie die Maximum Transmission Unit (MTU) konfiguriert werden. Zum Öffnen des TUN-Interfaces muss die `establish()`-Methode aufgerufen werden. Diese gibt den Filedeskriptor als `ParcelFileDescriptor` [5] zurück. Der native Filedeskriptor kann mit der `detachFd()`-Methode der `ParcelFileDescriptor`-Klasse losgelöst werden und danach vom nativen Code der `libipsec` verwendet werden.

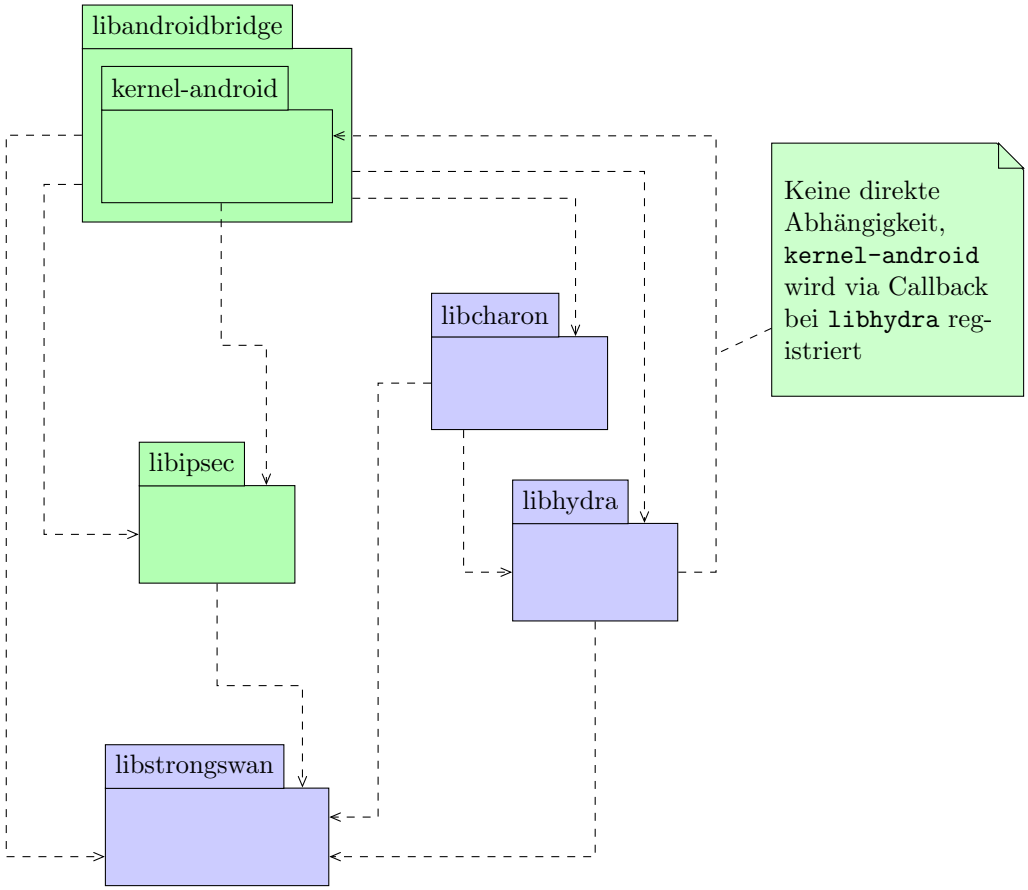

Abbildung 5.1.: Integration von `libipsec` unter Android

Abbildung 5.1 zeigt die Integration unter Android. Ausser dem `kernel_net`- und `kernel_ipsec`-Interface wurde auch noch ein Attribut-Handler zur Konfiguration allfälliger DNS-Server implementiert. Die zusätzliche Library `libandroidbridge` ist für die Initialisierung aller Libraries, Kommunikation mit der Java-Applikation und die Konfiguration der IKEv2-Daemons zuständig. Sie installiert zudem den ESP-Receiver Callback im IKEv2-Daemon Charon sowie den ESP-Sender Callback in `libipsec`.

Da das Aufsetzen des TUN-Interfaces, die Konfiguration von virtuellen IP-Adressen, Routen und DNS-Servern sowie die Installation der Bypass-Policies Zugriff auf das oben beschriebene `VpnService`-Interface erfordert, wurden diese Dinge als Teil von `libandroidbridge` implementiert. Da die Installation der Bypass-Policy über das `kernel_ipsec`-Interface geschieht, wurden die Kernel-Interfaces ebenfalls in `libandroidbridge` implementiert (`kernel-android`). So können die entsprechenden Funktionsaufrufe über das Java Native Interface (JNI) [2] an die `VpnService`-Klasse

weitergeleitet werden.

Das Öffnen eines TUN-Devices kann unter Android wie oben erwähnt nur über eine `VpnService.Builder`-Instanz geschehen. Zuvor müssen jedoch alle gewünschten Attribute wie die MTU, Interface-Adressen, Routen und DNS-Server auf dem Builder konfiguriert werden, da diese nach dem Öffnen des Interfaces nicht mehr gesetzt werden können. Dieses Verhalten erschwert die Integration in strongSwan. Da grundsätzlich nicht festgelegt ist, in welcher Reihenfolge die Installation von Policies, SAs, virtuellen IP-Adressen, etc. geschieht, muss davon ausgegangen werden, dass die entsprechenden Aufrufe auf den Kernel-Interfaces zu beliebigen Zeitpunkten in unterschiedlicher Reihenfolge erfolgen. Damit ist es nicht möglich festzustellen, wann alle Parameter einer Child SA fertig installiert wurden und das TUN-Device über den Builder aufgesetzt werden darf.

Als Lösung für dieses Problem registriert `libandroidbridge` einen Handler bei Charon, der jedes Mal aufgerufen wird, wenn eine Child SA fertig ausgehandelt wurde oder gerade zerstört wurde. Sobald der Handler nach dem erfolgreichen Aufsetzen der Child SA aufgerufen wird, ist sichergestellt, dass alle dazugehörenden Parameter bereits konfiguriert wurden. Somit kann das TUN-Device vom Handler geöffnet werden. Wird der Handler hingegen nach der Zerstörung einer Child SA aufgerufen, kann das entsprechende TUN-Device geschlossen werden. Der Nachteil dieser Lösung ist, dass das Aufsetzen des TUN-Devices in `libandroidbridge` gemacht werden muss, obwohl es rein logisch gesehen zu `libipsec` gehört.

5.2. Integration unter Linux

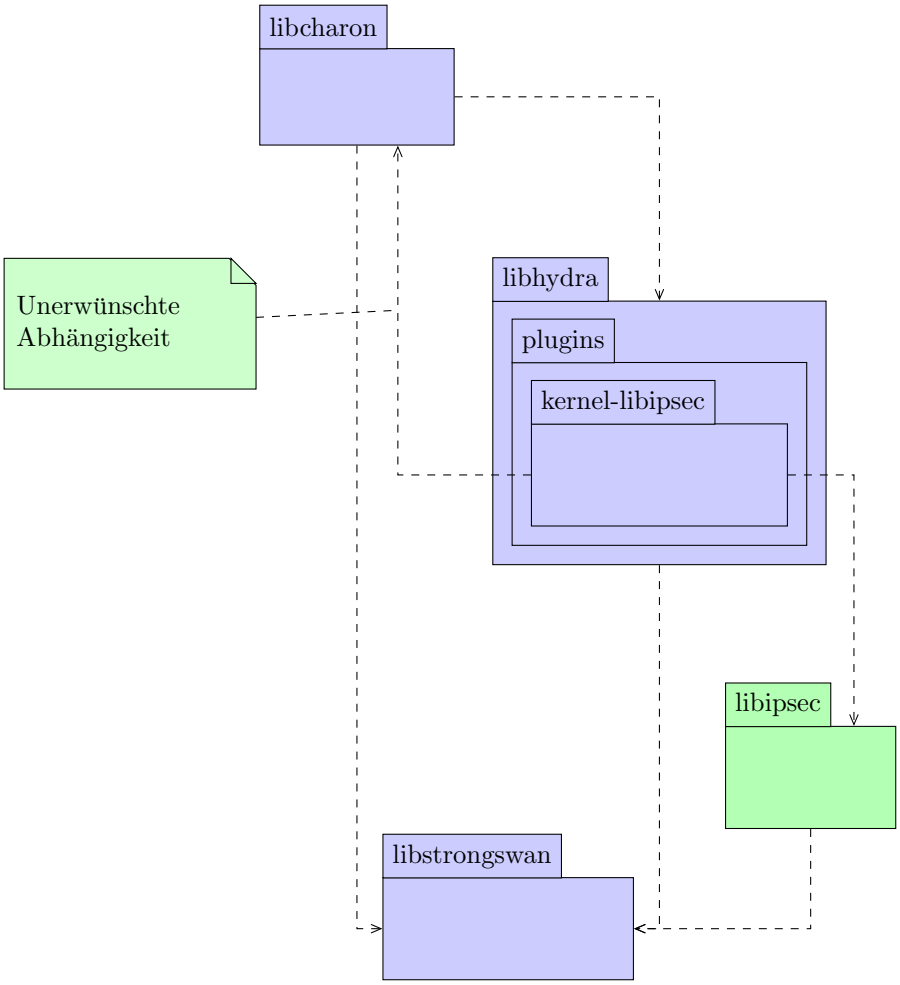

Abbildung 5.2.: Integration von `libipsec` unter Linux

Unter Linux ist die Integration wesentlich einfacher als unter Android. `Libipsec` wird über ein Plugin der `libhydra`, `kernel-libipsec`, initialisiert und angesteuert. Dieses Plugin ist die einzige Komponente, die von `libipsec` abhängig ist. Für den Rest von strongSwan ist der Einsatz anstelle des Linux IPsec-Stacks fast vollkommen transparent.

Da das `kernel-libipsec` Plugin jedoch die Sender und Receiver Callbacks bereitstellen und registrieren muss, entsteht eine eigentlich unerwünschte Abhängigkeit von `libcharon`. In Zukunft wäre es eventuell besser, wenn `libcharon` `libipsec` kennt und diese Callbacks selbst registriert, falls `libipsec` verwendet wird. Durch die Integration

unter Linux wurde im Rahmen dieser Arbeit jedoch hauptsächlich das Ziel verfolgt, `libipsec` schneller und besser entwickeln, testen und debuggen zu können. Deshalb wurden solche Änderungen noch nicht umgesetzt.

Ausserdem wurde die automatische Installation der aus dem Policies generierten Routen sowie das Einrichten der Bypass-Policy unter Linux noch nicht umgesetzt. Siehe dazu auch Abschnitt 9.3, Trennung der Kernel-Interfaces. Der Einsatz von `libipsec` unter Linux ist deshalb noch experimentell.

Obwohl die in Abschnitt 5.1 erwähnten Einschränkungen unter Linux nicht vorhanden sind, werden die TUN-Devices trotzdem erst erstellt, wenn eine Child SA fertig erstellt wurde. Dies wurde aus Kompatibilitätsgründen so entschieden.

6. Analyse Android-Applikation

6.1. Android Grundkonzepte

Dieser Abschnitt stellt kurz die für diese Arbeit wichtigen Android-Grundkonzepte vor. Detaillierte Informationen sind im Android *Developer's Guide* [7] zu finden.

Android-Applikationen werden normalerweise in Java geschrieben. Mit Hilfe des Android NDK (Native Development Kit) und dem Java Native Interface (JNI) [2] lässt sich jedoch auch nativer Code in eine Applikation einbinden. Jede Android-Applikation wird jeweils als eigener Benutzer in einem separaten Prozess und separater Java Virtual Machine (JVM) ausgeführt.

Folgende Android-Komponenten werden vom strongSwan VPN Client verwendet:

Activities
: Eine Activity repräsentiert eine Benutzeroberfläche und kann von anderen Activities aus gestartet werden.

Services
: Ein Service bezeichnet eine Komponente, welche im Hintergrund ausgeführt und nicht an die Lebensdauer von Activities gebunden ist. Services werden im selben Prozess ausgeführt wie der Rest der Applikation, falls dies nicht anders spezifiziert wird.

Fragments
: Ein Fragment ist ein Teil eines User Interfaces, wie z.B. ein einzelner Tab.

Activities und *Services* werden über *Intents* gestartet. Ein *Intent* ist eine asynchrone Nachricht, welche der angesprochenen Komponente eine Aufforderung sendet zu Starten und, falls benötigt, dem Aufrufer ein Resultat zurücksendet.

Jede Applikation benötigt ein Manifest-File (`AndroidManifest.xml`), in welchem alle verwendeten Komponenten einer Applikation deklariert werden müssen, damit das System weiss, wo sich diese Komponenten befinden und mit welchen Berechtigungen diese gestartet werden dürfen.

6.2. CA-Zertifikate

Zur Verwaltung der CA-Zertifikate für die Gateway-Authentifizierung wird die systemweite Sammlung von vertrauenswürdigen CA-Zertifikaten verwendet. Dies hat mehrere Vorteile. Die auf dem Gerät vorinstallierten CA-Zertifikate können so direkt genutzt werden. Für Verbindungen zu Gateways, welche von diesen CAs ausgestellte Zertifikate verwenden, müssen auf dem Gerät somit keine zusätzlichen Zertifikate installiert werden. Zudem gibt dies dem Benutzer eine zentrale, einheitliche Lösung zum Installieren von eigenen CA-Zertifikaten.

Da das manuelle Auswählen des korrekten CA-Zertifikates mühsam sein kann, wurde zudem eine Funktion implementiert, welche automatisch das korrekte CA-Zertifikat wählt. Dazu werden `Charon` alle CA-Zertifikate mitgegeben. Dieser sendet bei der Authentifizierung einen Zertifikats-Request für jedes installierte CA-Zertifikat. Dies bringt zwar einen gewissen Performance-Verlust mit sich, ist jedoch sehr benutzerfreundlich, da der Benutzer nicht wissen muss, welches Zertifikat benötigt wird.

6.3. User Interface

Bei der Gestaltung des User Interfaces standen vor allem die Funktionalität, Benutzerfreundlichkeit, Übersichtlichkeit sowie ein schlichtes ansprechendes Design im Vordergrund. In den folgenden Unterkapiteln werden die wichtigsten Entscheide erläutert.

Abbildung 6.1.: UI-Map

Das User Interface wurde in drei Views unterteilt: einer Übersichts-View, welche die aktuelle VPN-Verbindung sowie alle VPN-Profile anzeigt, eine Profil-Ansicht zum Bearbeiten eines Profiles sowie eine Log-View, welche den detaillierten Log-Output von strongSwan anzeigt.

Die Abbildung 6.1 zeigt die verschiedenen Views sowie die von Android automatisch generierte Notification, welche Informationen zur aktuellen VPN-Verbindung enthält.

6.3.1. Übersichts-View

Die Übersichts-View in Abbildung 6.2, welche angezeigt wird sobald die Applikation gestartet wird, ist in drei Sektionen unterteilt. Zuoberst befindet sich der Navigations-

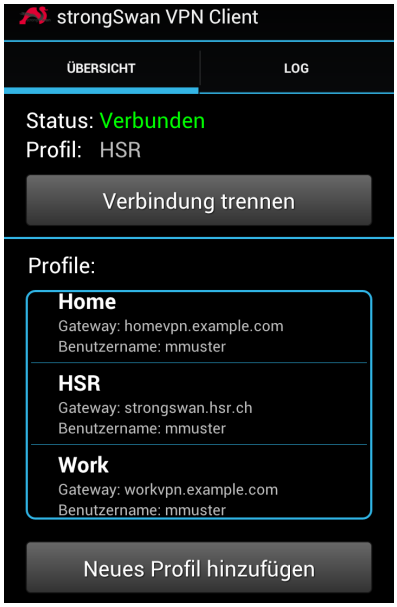

Abbildung 6.2.: Übersichts-View

bereich, welcher es erlaubt zwischen Übersicht- und Log-View zu wechseln. Darunter wird der gegenwärtige Status angezeigt. Ist zurzeit ein VPN aktiv, wird zudem der Name des aktiven Profiles sowie ein Button zum Trennen der aktuellen Verbindung angezeigt. Die gespeicherten VPN-Profile werden in einer scrollbaren Liste angezeigt. Durch einen Klick auf das gewünschte Profil kann eine VPN-Verbindung gestartet werden.

6.3.2. Profilverwaltung

Die minimalen Funktionsanforderungen beinhalten nur eine einzige View worin der Benutzer die Verbindungsparameter für eine VPN-Verbindung eingeben kann. Für Benutzer, die eine mobile VPN-Verbindung nur gelegentlich nutzen, würde diese Lösung genügen. Für Power User, die verschiedene VPN-Gateways benutzen möchten, wäre diese Lösung jedoch unbefriedigend. Daher wurde entschieden, dem Benutzer die Möglichkeit zu geben, mehrere VPN-Profile anzulegen und anschliessend auch verwalten zu können.

Abbildung 6.3 zeigt die Profil-View zum Erstellen oder Bearbeiten eines Profiles. Der Benutzer kann wählen, ob er das Passwort als Teil des Profiles speichern möchte. Falls dies jemand aus Sicherheitsgründen nicht möchte, wird bei jedem Verbindungsversuch ein Dialog angezeigt, der den Benutzer zur Eingabe des Passwortes auffordert.

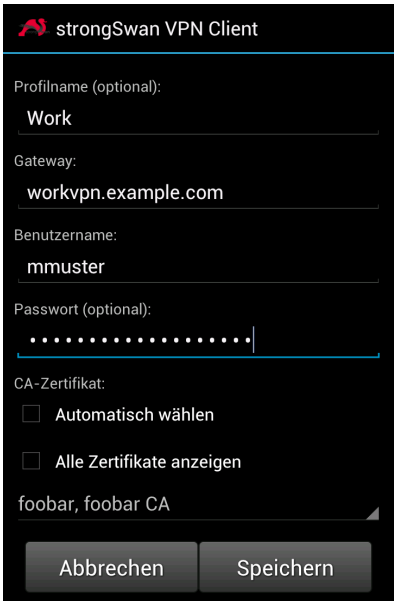

Abbildung 6.3.: Bearbeiten eines VPN-Profiles

Auswahl von CA-Zertifikaten

Der Benutzer kann das CA-Zertifikat für die Gateway-Authentifizierung wie oben erwähnt entweder manuell auswählen oder dies dem VPN Client überlassen. Um die Applikation möglichst benutzerfreundlich zu gestalten, ist per Default die automatische Zertifikatsauswahl aktiviert. Wird diese Option deaktiviert, erscheint ein Drop-down Menü (Spinner) zur Auswahl des Zertifikates. Dabei werden per Default nur die vom Benutzer manuell installierten CA-Zertifikate angezeigt. Da häufig eigene CA-Zertifikate verwendet werden, erleichtert dies die Suche nach dem richtigen Zertifikat wesentlich. Der Benutzer hat aber auch die Möglichkeit, die auf dem System vorinstallierten CA-Zertifikate anzuzeigen.

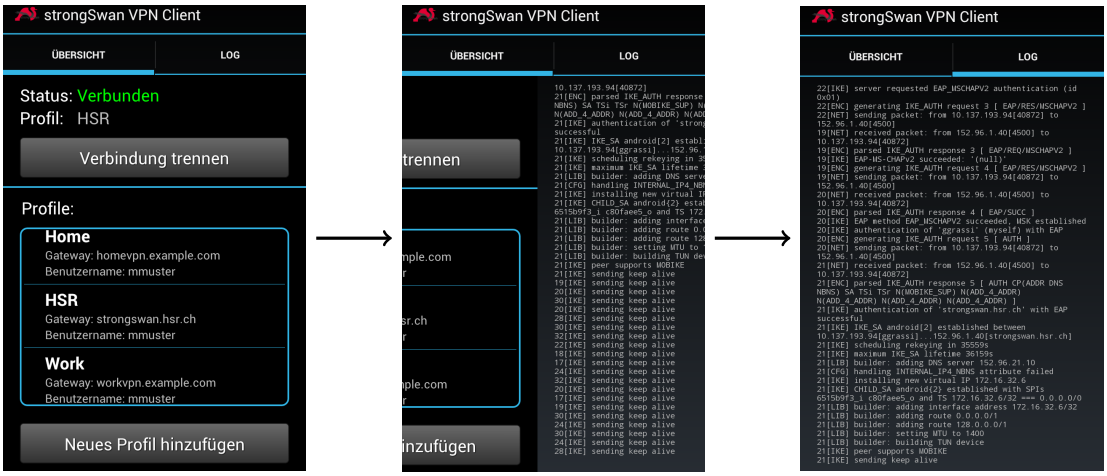

Abbildung 6.4.: Übergang von der Übersichts-View zur Log-View durch “Swipen”

6.3.3. Tabs und Log-View

Für Power User oder Netzwerkadministratoren ist es wichtig, Fehlerursachen anhand eines Logs schnell eruieren zu können. Dafür wurde eine Log-View integriert, welche dem Benutzer (oder dem Netzwerkadministrator) Informationen über den aktuellen Programmablauf oder detaillierte Informationen zu aufgetretenen Fehlern liefert. Die Logs werden vom Android eigenen Logging System `logcat` abgegriffen. Da die Logs von Android gespeichert werden, können diese auch nach einem Neustart der Applikation (oder nach einem Absturz) noch angezeigt werden.

Damit der angezeigte Log nicht zu gross wird, verwendet die Log-View einen Ringbuffer um die Anzahl angezeigter Zeilen zu limitieren. Wird die Log-View angezeigt, werden die Log-Einträge ständig (live) aktualisiert. Die Log-View verfügt zudem über eine “Autoscroll”-Funktion, d.h., es wird automatisch zum letzten Log-Eintrag gesprungen, solange der Benutzer nicht selber hoch scrollt.

Um die Navigation zwischen der Übersichts-View und der Log-View übersichtlich und benutzerfreundlich zu gestalten wurde entschieden, Tabs einzusetzen. Dazu wurden die beiden Views als Fragments implementiert, welche in einer einzelnen Activity verwendet werden. Zusätzlich zur Navigation über die Tabs, ist es auch möglich, durch “Swipen” zwischen den Views zu wechseln, wie dies in Abbildung 6.4 dargestellt ist.

6.3.4. Internationalisierung

Das gesamte User Interface ist internationalisiert und auf Deutsch, Englisch und Polnisch verfügbar. Durch das Anlegen weiterer Sprachdateien kann die Applikation ohne grossen

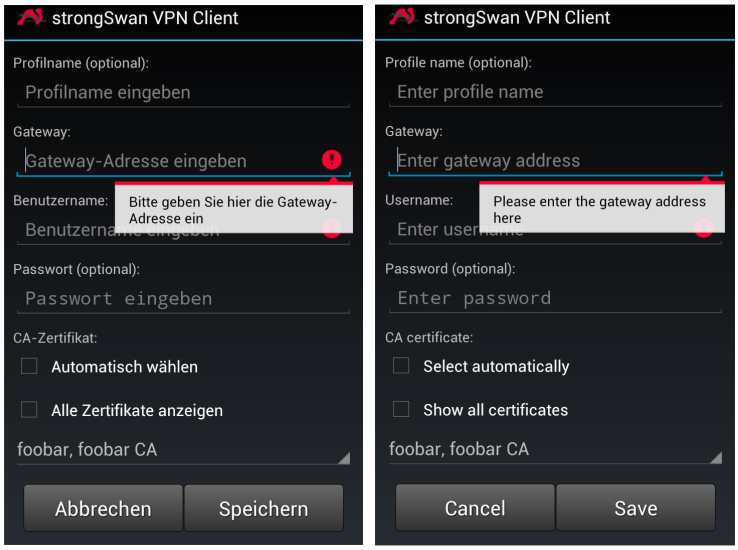

Abbildung 6.5.: Profil-Ansicht mit Fehlermeldung, auf Deutsch und Englisch

Aufwand in weitere Sprachen übersetzt werden.

6.3.5. Fehlermeldungen

Wie in Abbildung 6.5 dargestellt, wird die Vollständigkeit der Verbindungsparameter bereits beim Erstellen eines Profils überprüft. Wurde ein obligatorisches Feld nicht ausgefüllt, wird dies mit einer entsprechenden Fehlermeldung signalisiert.

Es kann jedoch vorkommen, dass ein CA-Zertifikat, welches in einem Profil konfiguriert wurde, vom Benutzer zu einem späteren Zeitpunkt vom System entfernt wird (ausserhalb des VPN-Clients). Da in der Datenbank nicht das komplette Zertifikat, sondern nur ein Alias gespeichert ist, kann es passieren, dass das Zertifikat nicht mehr vorhanden ist, wenn eine Verbindung aufgebaut werden soll. Ist das der Fall, wird automatisch die Profil-View geöffnet, wo der Benutzer dazu aufgefordert wird, ein neues Zertifikat auszuwählen.

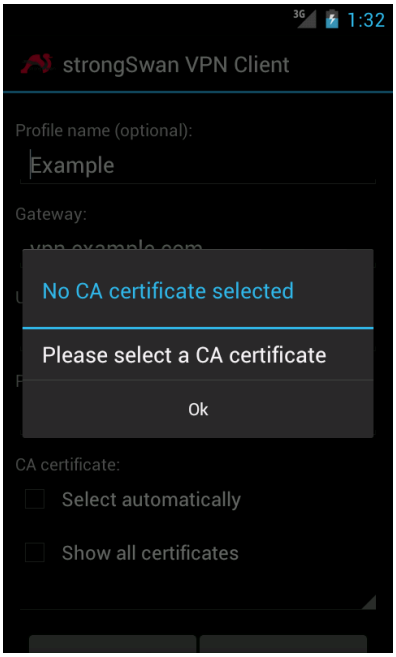

Abbildung 6.6.: Fehlermeldung bei gelöschtem Zertifikat

Tritt beim Verbindungsaufbau ein Fehler auf, wird dem Benutzer ebenfalls eine Fehlermeldung angezeigt. Abbildung 6.7 zeigt ein Beispiel für eine solche Fehlermeldung. Momentan sind die folgenden vier Fehlermeldungen implementiert:

User authentication failed
: Diese Meldung wird angezeigt, wenn ein Fehler bei der Benutzerauthentifizierung aufgetreten ist (beispielsweise weil ein falsches Passwort eingegeben wurde).

Gateway authentication failed
: Diese Fehlermeldung zeigt an, dass das Gateway nicht authentifiziert werden konnte.

Gateway address lookup failed
: Tritt auf, wenn die DNS-Auflösung der Gateway-Adresse fehlgeschlagen ist.

Gateway unreachable
: Diese Fehlermeldung wird angezeigt, wenn nach einem gewissen Timeout und mehreren Retries keine Antwort vom Gateway erhalten wurde.

Diese Meldungen decken die häufigsten Fehler beim Verbindungsaufbau ab. Um Benutzern die Möglichkeit zu geben, einen Fehler detaillierter zu analysieren, wurde die Log-View implementiert (siehe Abschnitt 6.3.3).

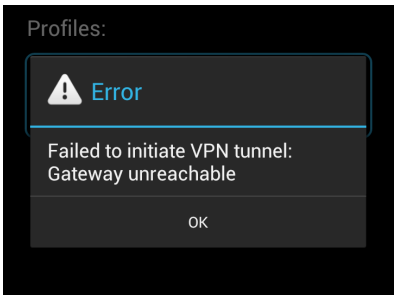

Abbildung 6.7.: Fehlermeldung bei unerreichbarem Gateway

7. Architektur Android-Applikation

Dieses Kapitel gibt einen Überblickt über die Architektur des im Rahmen dieser Arbeit entwickelten strongSwan VPN Clients für Android.

Das Android-Frontend besteht grob aus zwei Komponenten: einer Java-Applikation, welche das User Interface, Profil- und Verbindungsverwaltung und den VPN-Service enthält, sowie einer in C geschriebenen Library, `libandroidbridge`, welche `Charon` konfiguriert, `libipsec` initialisiert und die Schnittstelle zur Verwendung der Android `VpnService`-Funktionen bildet. Die beiden Komponenten kommunizieren über das Java Native Interface (JNI) [2] miteinander.

Abbildung 7.1 zeigt ein vereinfachtes Modell der Architektur der Applikation. Die gesamte Applikation wird in einem einzigen Prozess ausgeführt. Über das User Interface werden VPN-Profile aktiviert oder deaktiviert. Die VPN-Profile selbst werden wie in Abschnitt 7.5 beschrieben in einer Datenbank gespeichert.

Soll ein VPN aktiviert werden, erhält der State Manager vom User Interface ein VPN-Profil und führt alle nötigen Daten zum Initiieren der Verbindung, wie z.B. das/die im Profil konfigurierten CA-Zertifikate, zusammen. Diese Verbindungsparameter werden über einen Intent dem `CharonVpnService` übergeben. Dort werden alle benötigten nativen Libraries geladen. Über JNI ruft die `CharonVpnService`-Klasse Methoden des `charonservice` in `libandroidbridge` auf und umgekehrt.

`Charonservice` initialisiert die strongSwan-Libraries, registriert das Kernel-Interface `kernel-android` bei `libhydra` und startet den IKEv2-Daemon `Charon` (der Daemon läuft wie erwähnt im selben Prozess). `Charon` öffnet als erstes die IKE-Sockets. Für diese muss jeweils eine Bypass-Policy installiert werden (siehe Abschnitt 3.7.1). Dies geschieht über das Kernel-Interface `kernel-android`, welches die Aufrufe über JNI an `CharonVpnService` weiterleitet. Da `CharonVpnService` von der `VpnService`-Klasse [6] erbt, enthält die Klasse die benötigte Funktion zur Installation der Bypass-Policy (siehe Abschnitt 5.1, Integration unter Android für eine Beschreibung der `VpnService`-API).

`Charon` initiiert dann die IKE-Verbindung. Sobald eine Child SA erstellt wurde, wird über die `VpnService.Builder`-Klasse das TUN-Device aufgesetzt. Dies geschieht über die `vpnservice_builder` Klasse, welche über JNI mit der Java-Klasse `CharonVpnService.BuilderAdapter` benutzt. `BuilderAdapter` adaptiert `VpnService.Builder`, damit dieser über JNI einfacher zu verwenden ist.

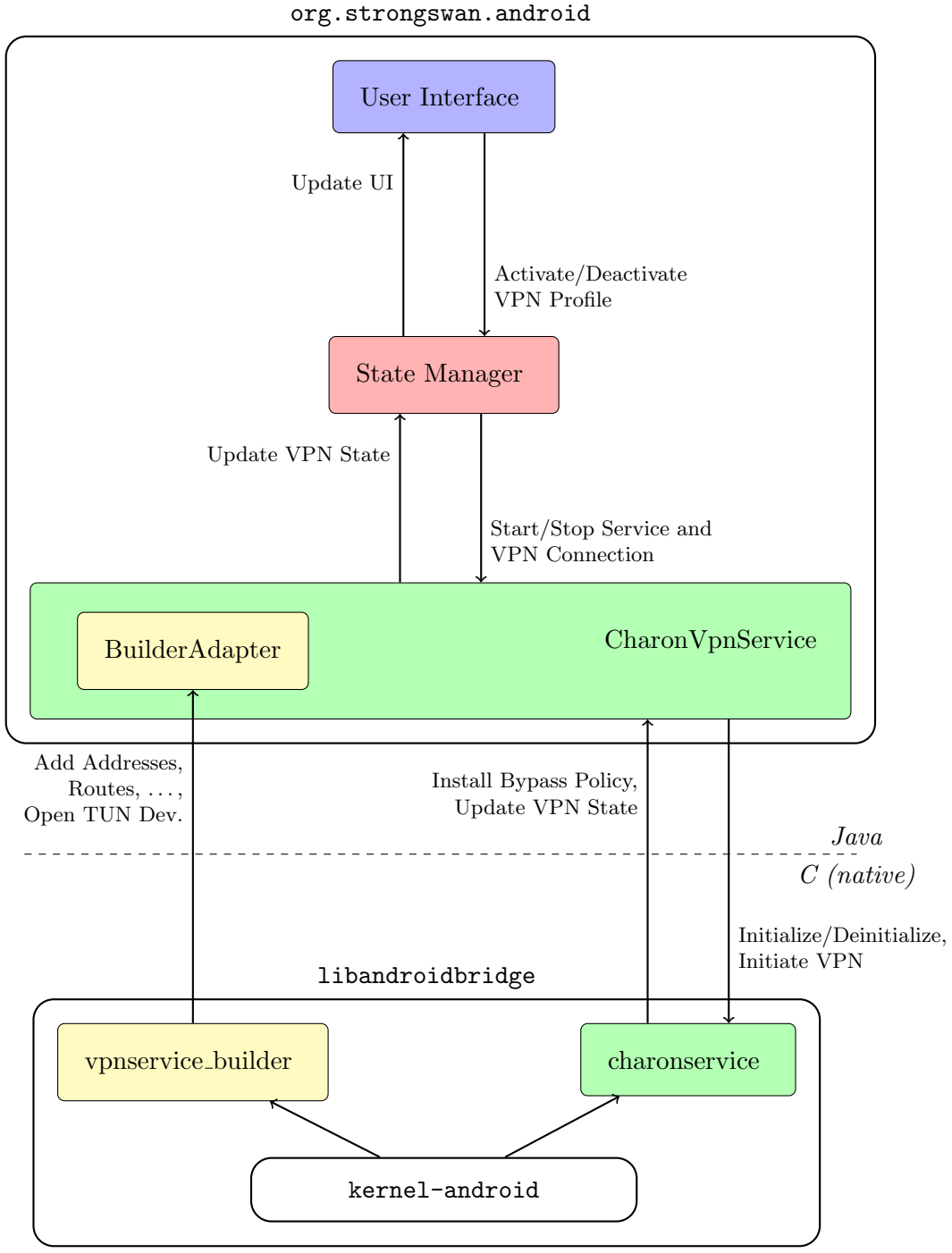

Abbildung 7.1.: Architekturübersicht der Android-Applikation

Von `libandroidbridge` aus werden Statusänderungen, wie z.B. ein erfolgreicher Verbindungsaufbau oder aufgetretene Fehler, an `CharonVpnService` gemeldet, von wo aus der Zustand des State Managers aktualisiert wird. Dieser löst dann die Aktualisierung des User Interfaces aus.

In den folgenden Abschnitten werden einzelne Komponenten etwas genauer beschrieben. In der Übersicht nicht enthalten ist der Zertifikat-Manager und die Datenbank, welche in Abschnitt 7.4 respektive Abschnitt 7.5 beschrieben werden.

7.1. Libandroidbridge

Die `libandroidbridge` bildet, wie der Name bereits sagt, eine Brücke zwischen den Java-Komponenten sowie den in C geschriebenen Komponenten. Wichtig ist, dass es sich *nicht* um eine Layer-Architektur handelt, denn die Kommunikation erfolgt bidirektional. Der Hauptgrund dafür ist, dass die nativen Komponenten Zugriff auf die `VpnService` [6] API benötigen (siehe Abschnitt 5.1).

Zudem übernimmt `libandroidbridge` die Initialisierung und anschliessende Konfiguration aller verwendeten strongSwan Libraries, insbesondere `libipsec` sowie `libcharon`, welche die IKEv2-Implementation enthält. Die Konfiguration erfolgt dabei nicht über Konfigurationsdateien wie unter Linux. Stattdessen werden die einzelnen VPN-Profile von der Java-Applikation verwaltet.

Damit Fehler beim Verbindungsaufbau festgestellt und an das User Interface weitergeleitet werden, registriert `libandroidbridge` verschiedene Listener bei `Charon`. Diese werden bei verschiedenen Events, beispielsweise wenn eine Child SA erstellt wurde oder wenn die Authentifizierung des Gateways fehlschlägt, aufgerufen. `Libandroidbridge` leitet diese Statusänderungen dann an die Java-Komponenten weiter.

7.2. Java/C Schnittstelle: CharonVpnService

Für die Kommunikation zwischen Java- und C-Komponenten wird das Java Native Interface (JNI) [2] benutzt. Das Framework erlaubt es, in C geschriebene Funktionen von Java aus aufzurufen und umgekehrt. Die von Java aus angesprochenen native-Funktionen werden in Java mittels `native` in der Methodensignatur gekennzeichnet. Vor der Nutzung dieser Methoden muss die entsprechende native-Library mit `System.loadLibrary()` geladen werden.

Ein besonderes Problem ergibt sich durch die Verwendung von native-Threads. Native-Threads sind diejenigen Threads, welche nicht von der JVM gestartet wurden sondern direkt aus dem nativen Code. Da `Charon` multithreaded ist, muss die Schnittstelle in der Lage sein, Funktionsaufaufrufe von C auf die Java-Klassen von beliebigen Threads aus zu tätigen. Dazu muss bei jedem Aufruf sichergestellt sein, dass der aufrufende Thread an

die JVM angebunden ist. Das Problem ist jedoch, dass jeder Thread sich, bevor er beendet, von der JVM lösen muss. Da der Thread-Pool jedoch nicht von `libandroidbridge` sondern von `libstrongswan` und `libcharon` verwaltet wird, stellt dies ein Problem dar. Als Lösung wird beim Anbinden an die JVM eine Thread-local Variable gesetzt. Für die Thread-local Variable lässt sich ein Destruktor definieren, der aufgerufen wird bevor der Thread terminiert wird. In diesem Destruktor kann der Thread von der JVM gelöst werden.

Ein weiteres Problem stellt die Deinitialisierung aller Libraries dar. `Libstrongswan` verlangt, dass die Initialisierung und Deinitialisierung der Library vom selben Thread durchgeführt wird. Wird die VPN-Verbindung über das normale User Interface der Applikation gestartet und beendet ist dies gewährleistet. Das Android Framework generiert jedoch zusätzlich eine Notification, sobald ein TUN-Device erstellt wurde. Über diese Notification kann der Benutzer die Verbindung ebenfalls trennen. Dazu wird automatisch auf dem betreffenden `VpnService`-Objekt eine Methode aufgerufen, welche die Verbindung beenden sollte. Dies wird jedoch nicht der gleiche Thread erledigen der auch für die Initialisierung zuständig war. Aus diesem Grund verwendet `CharonVpnService` einen zusätzlichen Handler-Thread, welcher die Aufgabe der Initialisierung und Deinitialisierung sowie das Initiieren einer Verbindung übernimmt.

7.3. State Manager

Der State Manager übernimmt die Aufgabe eines Controllers zwischen dem `CharonVpnService` und dem User Interface. Status Updates werden von einem nativen Thread ausgelöst und vom State Manager an das User Interface weitergeleitet. Ebenfalls startet und stoppt der State Manager den `CharonVpnService`.

Der State Manager verwaltet ausserdem die aktuell aktive Verbindung. Es wird unterscheiden zwischen einem VPN-Profil, welches alle Informationen über ein Profil enthält wie sie in der Datenbank abgelegt werden sowie einer VPN-Verbindung, welche alle für den Verbindungsaufbau nötigen Informationen enthält. Der grundsätzliche Unterschied besteht darin, dass das VPN-Profil nur ein Alias des gewählten CA-Zertifikat beinhaltet, wogegen die VPN-Verbindung die kompletten, DER-codierten X.509-Zertifikate enthält.

7.4. CA-Zertifikat-Manager

Der Zertifikat-Manager kapselt den Zugriff auf die auf dem Gerät installierten CA-Zertifikate. Aus Performancegründen werden die Zertifikate zudem vom Manager zwischengespeichert. Dazu werden beim Starten der Applikation in einem separaten Hintergrundthread alle auf dem Gerät vorhandenen Zertifikate ausgelesen und in einer Hashtable abgelegt. Das Auslesen aller Zertifikate zum Zeitpunkt des Gebrauchs würde die Benutzerfreundlichkeit der Applikation spürbar verringern, da der Vorgang auf dem

Gerät relativ viel Zeit benötigt (ca. 2-3 Sekunden). Bei der Erstellung eines neuen Profils oder bei einem Verbindungsaufbau werden die CA-Zertifikate spätestens benötigt. Wird eine solche Aktion ausgelöst, bevor die Zertifikate fertig ausgelesen sind, wird auf dem UI ein Fortschrittsindikator angezeigt, bis die Zertifikate verfügbar sind. Die Applikation ist währenddessen weiterhin benutzbar und blockiert nicht. Das Abspeichern des VPN-Profiles ist jedoch erst nach dem Auslesen der Zertifikate wieder aktiviert.

7.5. Datenbank

Für Android-Anwendungen steht jeweils eine private SQLite-Datenbank zur Verfügung. Die Datenbank liegt im Ordner `data` der Anwendung und ist somit durch das Dateisystem vom Zugriff von Drittanwendungen geschützt. Bei der Deinstallation der Anwendung wird der Ordner `data` mit der enthaltenden SQLite-Datenbank gelöscht. Die vom VPN-Client benötigte Datenbank wird zum Speichern der VPN-Profile verwendet und besteht nur aus einer einzigen Tabelle `vpnprofile`.

8. Testing

Sowohl die ESP-Implementation wie auch die Android-Applikation wurden intensiv getestet. Die ausführlichen Tests der ESP-Implementation wurden grösstenteils unter Linux durchgeführt. Die Android-Applikation wurde auf dem Android Emulator sowie auf einem Samsung I9250 Galaxy Nexus getestet.

8.1. Testumgebung

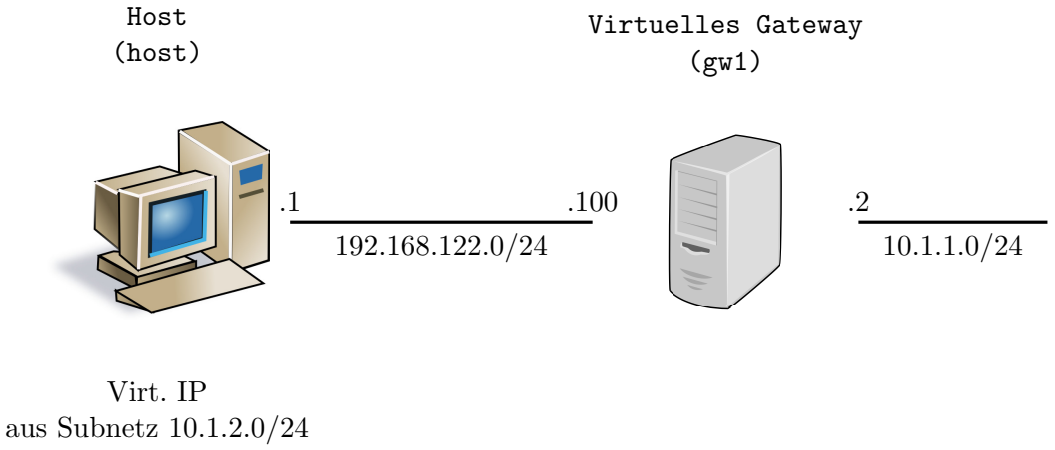

Abbildung 8.1.: Testumgebung

Um die ESP-Implementation sowie die Android-Applikation testen zu können, wurde eine virtuelle Testumgebung aufgesetzt. Abbildung 8.1 zeigt den Aufbau des Testnetzwerkes. Zur Virtualisierung wurde die Kernel-based Virtual Machine (KVM) eingesetzt, die Testumgebung liesse sich jedoch ohne Probleme auf das von strongSwan für automatisierte Tests verwendete User-mode Linux (UML) portieren.

Als Betriebssystem wurde sowohl auf dem Host-Computer sowie auf dem Gateway Debian GNU/Linux 6.0 eingesetzt. Da der von Debian per Default verwendete Kernel die

SHA-2 basierten HMACs noch nicht unterstützt, wurde auf dem Gateway ein neuerer Kernel installiert: Linux 3.2.0 (amd64). Als Testreferenz gilt deshalb dieser Kernel.

Zum Testen wurde die `libipsec`-basierte Version auf dem Host-Computer installiert und Verbindungen zum virtuellen Gateway aufgebaut. Dieses dient dabei als Referenz. Zu beachten ist, dass `libipsec` eine IPsec-Implementation für Hosts (Clients) ist und deshalb auch nur so getestet wurde.

Für Tests unter Android wurde vom Android-Emulator auf dem Host-Rechner ebenfalls Verbindungen zum virtuellen Gateway aufgebaut. Der Android-Emulator befindet sich dabei zusätzlich hinter einem NAT. Zudem wurde der VPN Client auch vom Android-Gerät aus über das Mobilfunknetz getestet. Als Gateway wurde das strongSwan-Gateway der HSR (`strongswan.hsr.ch`) verwendet.

8.2. Testkonfiguration unter Linux

```
##############################################################
# ipsec.conf - strongSwan IPsec configuration file           #
#                                                            #
# configuration for use with libipsec                        #
##############################################################

config setup
        plutostart=no
        # esp is the log group used by libipsec
        charondebug="chd 2, esp 2, knl 2"

conn %default
        keyexchange=ikev2
        mobike=no
        forceencaps=yes

conn gw1
        leftid="@host"
        left=192.168.122.1
        leftsourceip=%config
        leftcert=host_cert.der
        rightid="@gw1"
        right=192.168.122.100
        rightsubnet=10.1.1.0/24
        # cipher suite for ESP
        esp=aes128-sha256!
        auto=start
```

Listing 1: `ipsec.conf` des Host-Computers

Listing 1 zeigt die strongSwan-Konfiguration auf der Client-Seite. Wichtig ist der Parameter `forceencaps=yes`. Dieser bewirkt, dass der IKE-Daemon eine NAT-Umgebung vortäuscht um so ESP-in-UDP-Kapselung zu aktivieren. So wie `libipsec` in strongSwan integriert ist, ist dies zwingend. Ebenfalls erforderlich ist die explizite Angabe der lokalen IP-Adresse mit `left=<addr>`. Der Grund dafür ist die noch unvollständige Implementation des `kernel_net` Interfaces, welche die nötigen Funktionen zum Enumerieren der Lokalen Interfaces enthalten würde (siehe Abschnitt 9.3).

Wie bereits in Kapitel 5 erwähnt, ist die Integration von `libipsec` unter Linux noch unvollständig. Wenn Traffic über das VPN gesendet werden soll, müssen die Routen deshalb noch manuell installiert werden, nachdem eine Verbindung zum Gateway aufgebaut ist (als Root):

```
ip route add 10.1.1.0/24 dev tun0
```

`10.1.1.0/24` ist dabei das `leftsubnet` auf dem Gateway, und `tun0` das automatisch erstellte TUN-Device. Da die Bypass-Policy unter Linux noch nicht implementiert wurde (siehe Abschnitt 5.2, Integration unter Linux), muss beim Installieren der Routen darauf geachtet werden, dass keine Routing-Loops entstehen.

Der Parameter `esp=aes128-sha256!` spezifiziert die ESP-Ciphersuite. Folgende Ciphersuites werden von `libipsec` unterstützt und wurden getestet:

- `esp=aes128-sha1!`
- `esp=aes192-sha1!`
- `esp=aes256-sha1!`
- `esp=aes128-sha256!`
- `esp=aes192-sha256!`
- `esp=aes256-sha256!`
- `esp=aes128-sha384!`
- `esp=aes192-sha384!`
- `esp=aes256-sha384!`
- `esp=aes128-sha512!`
- `esp=aes192-sha512!`
- `esp=aes256-sha512!`

8.3. Systemtest

Die Android-Applikation wurde über längere Zeit auf dem Android-Emulator sowie dem Android-Gerät eingesetzt. Dabei wurden folgende Anforderungen getestet:

- Stabilität
- Verhalten in unterschiedlichen Fehlersituationen, z.B.:
  - Gateway nicht erreichbar
  - DNS-Auflösung des Gateways schlägt fehl
  - Benutzerauthentifizierung schlägt fehl
  - Gateway-Authentifizierung schlägt fehl
- Datendurchsatz

- Rekeying
- Speicherverbrauch (allfällige Memory Leaks)

Um das Auslösen des Rekeyings zu testen wurde der Code temporär so angepasst, dass Lifetime der SAs auf dem Client kürzer ist als auf dem Gateway. Somit ist für den Test sichergestellt, dass das Rekeying vom Client initiiert wird.

Die Tests haben gezeigt, dass es Probleme gibt wenn das Gerät in ein anderes Netzwerk wechselt, beispielsweise vom WLAN ins Mobilfunknetz oder umgekehrt. Dies wurde bereits von vornherein vermutet. Der Grund für die Probleme ist das Wechseln der IP-Adresse sowie des Netzwerk-Interfaces. Zur Lösung dieser Probleme wäre jedoch die vollständige Implementation des `kernel_net`-Interfaces nötig, die unter Android noch nicht verfügbar ist (siehe Kapitel 9, Zukünftige Erweiterungen).

8.4. User Interface Test

Die User Interface Tests sollen sicherstellen, dass das UI den Anforderungen entsprechend funktioniert und keine unvorhergesehene Fehler mehr auftreten. Dazu wurde eine Reihe von Test Cases manuell überprüft:

- Lösen alle Buttons die gewünschte Funktionalität aus?
- Sind alle Buttons, Überschriften und Dialoge sinngemäss beschriftet?
- Kann zwischen den Tabs durch “Swipen” und durch Tap auf die Überschrift gewechselt werden?
- Kann ein neues Profil hinzugefügt werden?
- Wird der Benutzer daran gehindert ein unvollständiges VPN-Profil abzuspeichern und wird er zweckmässig darüber informiert?
- Wird dem Benutzer bei der Eingabe der Verbindungsparameter Hilfe geboten?
- Funktioniert die automatische Zertifikatsauswahl?
- Funktioniert die explizite Zertifikatsauswahl?
- Wird das Vorhandensein des selektierten Zertifikates beim aktivieren eines VPN-Profils überprüft und eine entsprechende Meldung angezeigt, falls das Zertifikat fehlt?
- Kann ein bestehendes Profil bearbeitet werden?
- Kann ein Profil wieder gelöscht werden?
- Wird durch Tap auf das Profil die Verbindung initialisiert?
- Erscheint der Password-Dialog bei den entsprechenden Profilen?

- Wird der User entsprechend über einen laufenden Verbindungsauf- oder -abbau informiert?
- Erhält der Benutzer sinnvolle Fehlermeldungen, wenn nötig (beispielsweise bei Fehlern beim Verbindungsaufbau)?
- Werden der Verbindungsstatus und die Informationen zur Verbindung jeweils entsprechend angezeigt?
- Wird der Verbindungsstatus beim Verlassen der App und erneutem Öffnen immer noch korrekt dargestellt?
- Kann die Verbindung wieder beendet werden?
- Werden im Log entsprechende Einträge zur Verbindung angezeigt?
- Werden dem Benutzer durch die Autoscroll-Funktion die letzten Log-Einträge gezeigt?

8.4.1. Monkey-Tool

Das `monkey`-Tool, welches im Android-SDK enthalten ist, generiert pseudozufällige User Interface Events, welche dann an die zu testende Applikation gesendet werden. Es werden alle möglichen Events, welche auf einem Smartphone generiert werden können wie Klicks, Gesten und Texteingaben getestet. Der Test bricht ab, sobald die Applikation abstürzt oder eine nicht abgefangene Exception geworfen wird. Durch die hohe Frequenz der ausgelösten Events kann die Robustheit des User Interfaces gut getestet werden.

```
adb shell monkey -p org.strongswan.android -v 5000
```

Listing 2: UI-Test mit `monkey`: 5000 Events

Durch diesen Test wurden keine neuen Fehler in der Applikation gefunden.

8.4.2. Usability

Während der Planung und Entwicklung des User Interfaces wurden mehrere unabhängige Personen miteinbezogen um die UI-Abläufe zu optimieren. Nach Fertigstellung der einzelnen Elemente wurde die Applikation zudem an mehrere Testbenutzer verteilt. Durch deren Feedback wurden gewisse Mängel für die finale Version korrigiert. Einige kleinere Mängel, die entdeckt wurden, wurden noch nicht behoben. Diese wurden unter Abschnitt 9.10.4, Usability-Verbesserungen für künftige Verbesserungen dokumentiert.

9. Zukünftige Erweiterungen

Dieses Kapitel zeigt den aktuellen Projektstand und beschreibt allfällige künftige Verbesserungen sowie Erweiterungen.

9.1. Privilege Separation

Unter Linux benötigt der `charon`-Prozess noch immer Root-Privilegien. Der Grund dafür ist, dass zum Öffnen der TUN-Devices Root-Rechte nötig sind. Dies stellt ein gewisses Sicherheitsrisiko dar. Das Risiko könnte reduziert werden, indem der `charon`-Prozess in zwei separate Prozesse aufgeteilt wird: einer, der als Root ausgeführt wird und einer, der seine Privilegien abgibt und als normaler Systembenutzer ausgeführt wird. Der Umfang des Root-Prozesses könnte sehr klein (und damit besser verifizierbar und sicher) gehalten werden, während die meiste Arbeit vom unprivilegierten Prozess übernommen würde. Es könnte eine Schnittstelle definiert werden, welche die Operationen, welche Root-Rechte benötigen, abtrennt. Aufrufe dieser Operationen würden dann an den privilegierten Prozess weitergereicht. Das Sicherheitsrisiko würde so minimiert. Ein Angreifer, der eine allfällig vorhandene Sicherheitslücke im unprivilegierten Prozess ausnutzt, die es erlaubt Code auszuführen, hätte nicht gleich Root-Rechte.

`Charon` unterstützt bereits das Abgeben der Root-Privilegien nach dem Start. Es wäre auch denkbar, dass TUN-Devices für alle konfigurierten Verbindungen bereits beim Starten, vor der Abgabe der Root-Privilegien geöffnet werden. Eine solche Lösung käme ohne zwei getrennte Prozesse aus und würde weniger Änderungen an der strongSwan-Architektur erfordern.

9.2. SA-Lifetime basierend auf Datenmenge

Momentan beachtet `libipsec` nur die SA-Lebensdauer als Zeit. Unter Android ist die Lifetime so konfiguriert, dass ein Rekeying allerspätestens drei Stunden nach dem Aufbau der Child SA ausgeführt wird. Dies sollte grundsätzlich ausreichen, zumal die meisten Android-Geräte typischerweise nicht über eine sehr schnelle Netzwerkverbindung verfügen und somit in dieser Zeit nicht übermässig viele Daten über eine SA gesendet werden können. Jedoch sollte in Zukunft trotzdem auch die Lifetime basierend auf der

übertragenen Datenmenge (Anzahl Bytes) unterstützt werden. Dies wäre vor allem auch für die Linux-Version von Bedeutung.

Für die Implementation dieses Features müsste die Anzahl verarbeiteter Bytes pro SA erfasst werden. Gleichzeitig könnte auch die Funktion zum Abfragen der verarbeiteten Datenmenge für eine SA, welche im `kernel_ipsec`-Interface deklariert ist, implementiert werden.

9.3. Trennung der Kernel-Interfaces

Wie in Abschnitt 2.2.2 beschrieben, werden bei der Verwendung des Linux IPsec Stacks die `kernel_ipsec-` und `kernel_net`-Interfaces in einem einzigen Plugin, `kernel-netlink`, implementiert. Das bedeutet, dass die Implementationen der beiden Interfaces momentan nicht getrennt verwendet werden können. Jedoch wären einige Funktionen der `kernel_net` Implementation, welche über RTNETLINK die Routing-Tabelle des Kernels auslesen (und bearbeiten) können, auch bei Verwendung von `libipsec` nützlich. Beispielsweise wäre es für die Installation von Bypass-Policies mit `SO_BINDTODEVICE` (siehe Abschnitt 3.7.1) unter Linux nötig, das Netzwerkinterface zu ermitteln, über welche Pakete einer SA gesendet werden, damit der IKE- und ESP-Socket an dieses Interface gebunden werden kann (unter Android wird das Interface, an welches der Socket gebunden wird, vom Betriebssystem selbst ausgewählt). Auch die Installation der aus den Policies generierten Routen wäre unter Linux über RTNETLINK möglich. Zudem wäre es mit der `kernel_net` Implementation nicht mehr nötig, die lokale IP-Adresse im Config-File (`ipsec.conf`) explizit anzugeben. Unter Android wurde das Ermitteln der Lokalen IP-Adresse behelfsweise in Java implementiert. Falls es aber in Zukunft Android-Geräte geben sollte, auf denen gleichzeitig mehrere Netzwerk-Interfaces aktiv sein können, selektiert dieser Code nicht mehr zuverlässig die korrekte Adresse.

Ein Problem ist allerdings, dass das `kernel_net` Interface auch Funktionen zum Verwalten der virtuellen IP-Adressen enthält. Diese sind natürlich für `libipsec` anders implementiert als wenn der IPsec Stack des Kernels verwendet wird. Da nur eine Implementation des Interfaces registriert werden kann, stellt dies für die Verwendung der Netlink-Implementation des `kernel_net`-Interfaces ein Problem dar. Eine mögliche Lösung wäre das Verschieben dieser Funktionen in das `kernel_ipsec`-Interface.

9.4. Extended Sequence Number Support

ESP bietet die Möglichkeit, eine 64-Bit Extended Sequence Number (ESN) [21] als Erweiterung der Sequenznummer zu verwenden. Dabei enthält das Sequenznummer-Feld des ESP-Paketes nur die niederwertigeren 32 Bit der ESN. Die höherwertigen 32 Bit werden nicht übertragen und sowohl vom Sender als auch vom Empfänger lokal nachgeführt.

ESN ist vor allem für sehr schnelle Netzwerkverbindungen über welche in kurzer Zeit viele Pakete übertragen werden nützlich. Da Android-Geräte meist eher langsame WLAN- oder Mobilfunknetze verwenden, ist ESN nicht unbedingt nötig. Unter Linux jedoch wäre ESN-Support eventuell von Vorteil.

Zur Implementation von ESN in `libipsec` müsste in erster Linie die Verifizierung der Sequenznummer auf Empfängerseite inkl. Anti-Replay Window erweitert werden. Zudem muss die Berechnung und Verifizierung des ICV für ESN angepasst werden, da dieser bei ESN auch die höherwertigen 32 Bit der Sequenznummer umfasst, welche wie erwähnt nicht direkt Teil des ESP-Paketes sind.

9.5. Zusätzliche Ciphersuites

`Libipsec` unterstützt AES-CBC für die Verschlüsselung sowie HMAC-SHA-1,-256,-384, und -512 für die Authentisierung. In Zukunft würde die Implementation weiterer Ciphersuites Sinn machen. Beispiele sind:

- AES-CCM (Counter with CBC-MAC Mode) [31]
- AES-GCM (Galois/Counter Mode) [16]
- ...

Zusätzliche CBC-Ciphers sollten einfach zu integrieren sein. Die Implementation anderer Modi sowie die kombinierten Algorithmen wie z.B. AES-GCM erfordert einige Erweiterungen bei der Paketverarbeitung und ist daher ein wenig aufwendiger.

9.6. IPv6-Support

In Zukunft könnte die IPsec-Implementation erweitert werden, dass auch IPv6 unterstützt wird. Dazu müsste unter anderem bei outbound-Paketen festgestellt werden können, ob es sich um IPv4-Pakete oder IPv6-Pakete handelt (momentan wird immer IPv4 angenommen), um das Next Header Feld im ESP-Trailer richtig setzen zu können. Dazu müsste zumindest das Versions-Feld des IP-Headers inspiziert werden.

9.7. MOBIKE-Support

Das IKEv2 Mobility and Multihoming Protocol (MOBIKE) [29] ist eine Erweiterung zu IKEv2, welche es erlaubt, die IP-Adressen von SAs während dem Betrieb zu ändern. Dies erlaubt es, dass mobile Clients die VPN-Verbindung halten können, selbst wenn sie eine neue IP-Adresse erhalten (z.B. bei einem Wechsel vom Mobilfunknetz ins WLAN).

`Charon` unterstützt MOBIKE, `libipsec` jedoch noch nicht. Es müsste noch eine Funktion zum Aktualisieren (ändern der Adressen) einer SA implementiert werden. Zudem würde wohl eine vollständige Implementation des `kernel_net` Interfaces benötigt (siehe dazu Abschnitt 9.3). Ausserdem müsste immer dann, wenn ein anderes Netzwerkinterface verwendet werden soll (wie beim Wechsel zwischen WLAN und Mobilfunknetz), ein neuer Socket geöffnet werden. Der Grund dafür ist, dass der alte Socket aufgrund der Bypass-Policy (`SO_BINDTODEVICE`) noch an das alte Netzwerkinterface gebunden ist. Deshalb müsste dieser jeweils durch einen neuen Socket, der an das neue Netzwerk-Interface gebunden wird, ersetzt werden.

9.8. Merging der TUN-Klassen

`Libipsec` enthält zwei nur leicht unterschiedliche Klassen `tun_linux` sowie `tun_android`, welche ein TUN-Device abstrahieren. Der Grund dafür ist, dass die `ioctl(2)`-Requests beispielsweise zum Abfragen/Setzen der MTU, Setzen von IP-Adressen, etc. den Namen des Interfaces (z.B. `tun0`) als Parameter benötigen. Unter Android wird das TUN-Device jedoch mit Hilfe der `VpnService.Builder`-Klasse geöffnet, welche nur den Filedeskriptor des TUN-Devices liefert, nicht aber dessen Namen. Grundsätzlich wäre dies kein Problem, da mit dem `TUNGETIFF ioctl(2)`-Request der Name des TUN-Devices anhand des Filedeskriptors abgefragt werden kann. Dieser Request ist jedoch im Header `linux/if_tun.h` des Android-NDKs für Android API Level 14 (Android 4.0) nicht deklariert und wurde deshalb nicht verwendet. Daher wird unter Android eine separate Implementation der TUN-Klasse verwendet, in welcher die meisten Funktionen durch Stubs ersetzt wurden. Sobald der `TUNGETIFF`-Request unter Android verfügbar ist, gibt es jedoch keinen Grund mehr, nicht dieselbe TUN-Klasse wie unter Linux zu verwenden.

9.9. Statische Buffer für Paketverarbeitung

Bei der Paketverarbeitung wird dynamisches Memory-Management (`malloc(3)`, `free(3)`, ...) eingesetzt. Zwar wurde versucht, das temporäre Allozieren von Buffern sowie unnötiges Kopieren von Daten soweit möglich zu vermeiden, jedoch könnte die Performance vermutlich dennoch durch die Verwendung von statischen Buffern verbessert werden. Die Queue, welche die eingehenden Pakete enthält, könnte zudem durch eine einmalig allozierte Queue fixer Länge ersetzt werden.

9.10. Android-Applikation

In diesem Abschnitt werden künftige Erweiterungen vorgeschlagen, welche nur die Android-Applikation betreffen.

9.10.1. Authentifizierung mit User-Zertifikaten

Eine der wahrscheinlich wichtigsten künftigen Erweiterungen ist die Benutzerauthentifizierung mittels Zertifikaten. Die User-Zertifikate könnten dazu über die `KeyChain`-Klasse [4] von Android verwaltet werden.

9.10.2. Credential Storage

Das Benutzer-Passwort wird momentan, falls der Benutzer dieses mit dem Profil abspeichern möchte, im Klartext in einer SQLite-Datenbank abgelegt. Zwar können andere Applikationen nicht darauf zugreifen, jedoch wäre es wohl in Zukunft trotzdem besser, wenn stattdessen die von Android bereitgestellten APIs zum Verwalten von User Credentials verwendet werden.

9.10.3. Caching bei Verwendung aller CA-Zertifikate

Werden alle installierten CA-Zertifikate verwendet (automatische Zertifikatsauswahl), könnte zur Steigerung der Performance nach der ersten erfolgreichen Authentifizierung, eine Referenz auf das verwendete CA-Zertifikat im Profil gecached werden. Beim späteren Verbindungen müsste dann nur noch ein Zertifikats-Request für das gecachte Zertifikat gesendet werden. Allerdings müssten als Fallback wieder alle CA-Zertifikate verwendet werden, falls das Gateway ein neues Zertifikat verwendet, welches von einer anderen CA ausgestellt wurde und die Authentifizierung deshalb fehlschlägt.

9.10.4. Usability-Verbesserungen

Die folgende Liste enthält kleinere Mängel und Vorschläge zur Verbesserung der Usability des VPN-Clients, welche künftig umgesetzt werden könnten:

- Beim Menu zum Löschen oder Bearbeiten von Profilen sollten die Buttons evtl. vertikal statt horizontal angeordnet werden. Zudem sollte der Name des gewählten Profils für den Benutzer ersichtlich sein.
- Die von Android erzeugte VPN-Notification beinhaltet eine Traffic-Statistik. Eine solche Statistik sollte auch über das User Interface der Applikation selbst aufgerufen werden können.

- Wird während einer bestehenden Verbindung erneut auf ein Profil geklickt wird der Benutzer nicht gefragt, ob er die aktive Verbindung trennen möchte um eine neue zu initialisieren. Dies kann für den Benutzer verwirrend sein, wenn er versehentlich auf ein Profil klickt.
- Der Spinner (Drop-down Menü) zur Anzeige der CA-Zertifikate ist etwas überfüllt, wenn alle Zertifikate angezeigt werden. Ausserdem lassen sich einzelne Zertifikate ohne detailliertere Informationen nur schwer voneinander unterscheiden. Eine separate Listen-Ansicht könnte die Zertifikate übersichtlicher darstellen. Zudem könnte eine Detail-Ansicht implementiert werden, in welcher weitere Informationen zu einem einzelnen Zertifikat (z.B. Fingerprints) dargestellt werden könnten.
- Wenn der CA-Zertifikats-Spinner geöffnet wird sollte die virtuelle Tastatur versteckt werden.
- In der Profil-Detailansicht sollten die *Save*- und *Abort*-Buttons am unteren Rand der View fixiert werden.
- Der Status einer aktuellen Verbindung (*Connected*) könnte mit einem Icon (z.B. ein Schloss) zusätzlich visualisiert werden.
- Die Versionsnummer der Applikation sowie jene von strongSwan sollte in der Applikation direkt ersichtlich sein.

Literaturverzeichnis

[1] strongSwan, the Open Source IPsec-based VPN Solution. URL http://www.strongswan.org.

[2] *Java Native Interface Specification*, 2011. URL http://docs.oracle.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html.

[3] Object Oriented C programming style. strongSwan Developer Documentation, August 2011. URL http://wiki.strongswan.org/projects/strongswan/wiki/ObjectOrientedC.

[4] *Android 4 KeyChain Interface Documentation*, 2012. URL http://developer.android.com/reference/android/security/KeyChain.html.

[5] *Android 4 ParcelFileDescriptor Class Documentation*, 2012. URL http://developer.android.com/reference/android/os/ParcelFileDescriptor.html.

[6] *Android 4 VpnService Interface Documentation*, 2012. URL http://developer.android.com/reference/android/net/VpnService.html.

[7] Application Fundamentals. Android Basics, 2012. URL http://developer.android.com/guide/topics/fundamentals.html.

[8] epoll - I/O event notification facility. Linux Programmer's Manual, release 3.41, April 2012. URL http://man7.org/linux/man-pages/man7/epoll.7.html.

[9] netlink - Communication between kernel and userspace (AF_NETLINK). Linux Programmer's Manual, release 3.41, April 2012. URL http://man7.org/linux/man-pages/man7/netlink.7.html.

[10] socket - Linux socket interface. Linux Programmer's Manual, release 3.41, April 2012. URL http://man7.org/linux/man-pages/man7/socket.7.html.

[11] *strongSwan Developer Documentation*, June 2012. URL http://wiki.strongswan.org/projects/strongswan/wiki/DeveloperDocumentation.

[12] D. Eastlake 3rd and T. Hansen. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234, May 2011. URL http://www.ietf.org/rfc/rfc6234.txt.

[13] S. Frankel, R. Glenn, and S. Kelly. The AES-CBC Cipher Algorithm and Its Use with IPsec. RFC 3602, September 2003. URL http://www.ietf.org/rfc/rfc3602.txt.

[14] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409, November 1998. URL http://www.ietf.org/rfc/rfc2409.txt.

[15] P. Hoffman. Cryptographic Suites for IPsec. RFC 4308, December 2005. URL http://www.ietf.org/rfc/rfc4308.txt.

[16] R. Housley. Using Advanced Encryption Standard (AES) CCM Mode with IPsec Encapsulating Security Payload (ESP). RFC 4309, December 2005. URL http://www.ietf.org/rfc/rfc4309.txt.

[17] A. Huttunen, B. Swander, V. Volpe, L. DiBurro, and M. Stenberg. UDP Encapsulation of IPsec ESP Packets. RFC 3948, January 2005. URL http://www.ietf.org/rfc/rfc3948.txt.

[18] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996, September 2010. URL http://www.ietf.org/rfc/rfc5996.txt.

[19] S. Kelly and S. Frankel. Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec. RFC 4868, May 2007. URL http://www.ietf.org/rfc/rfc4868.txt.

[20] S. Kent. IP Authentication Header. RFC 4302, December 2005. URL http://www.ietf.org/rfc/rfc4302.txt.

[21] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303, December 2005. URL http://www.ietf.org/rfc/rfc4303.txt.

[22] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301, December 2005. URL http://www.ietf.org/rfc/rfc4301.txt.

[23] M. Krasnyansky and M. Yevmenkin. Universal TUN/TAP device driver, 2000. URL http://www.kernel.org/doc/Documentation/networking/tuntap.txt.

[24] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997. URL http://www.ietf.org/rfc/rfc2104.txt.

[25] C. Madson and R. Glenn. The Use of HMAC-SHA-1-96 within ESP and AH. RFC 2404, November 1998. URL http://www.ietf.org/rfc/rfc2404.txt.

[26] V. Manral. Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH). RFC 4835, April 2007. URL http://www.ietf.org/rfc/rfc4835.txt.

[27] National Institute of Standards and Technology (NIST). Announcing the Advanced Encryption Standard (AES). FIPS PUB 197, November 2001. URL http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[28] National Institute of Standards and Technology (NIST). Secure Hash Standard. FIPS PUB 180-4, March 2012. URL http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf.

[29] Ed. P. Eronen. IKEv2 Mobility and Multihoming Protocol (MOBIKE). RFC 4555, June 2006. URL http://www.ietf.org/rfc/rfc4555.txt.

[30] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux Netlink as an IP Services Protocol. RFC 3549, July 2003. URL http://www.ietf.org/rfc/rfc3549.txt.

[31] J. Viega and D. McGrew. The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP). RFC 4106, June 2005. URL http://www.ietf.org/rfc/rfc4106.txt.

Abkürzungsverzeichnis

AES Advanced Encryption Standard.

AH Authentication Header.

API Application Programming Interface.

BSD Berkeley Software Distribution.

CA Certificate Authority.

CBC Cipher Block Chaining.

DER Distinguished Encoding Rules.

DNS Domain Name System.

EAP Extensible Authentication Protocol.

ESN Extended Sequence Number.

ESP Encapsulating Security Payload.

GNU GNU's not Unix.

GPL GNU General Public License.

HMAC Hash-based Message Authentication Code.

HSR Hochschule für Technik Rapperswil.

ICV Integrity Check Value.

IKE Internet Key Exchange.

IP Internet Protocol.

IPsec Internet Protocol Security.

ITA Institute for Internet Technologies and Applications.

IV Initialization Vector.

JNI Java Native Interface.

JVM Java Virtual Machine.

KVM Kernel-based Virtual Machine.

MAC Message Authentication Code.

MTU Maximum Transmission Unit.

NAT Network Address Translation.

NDK Native (Software) Development Kit.

OS Operating System.

RFC Request For Comment.

SA Security Association.

SAD Security Association Database.

SDK Software Development Kit.

SHA Secure Hash Algorithm.

SPD Security Policy Database.

SPI Security Parameters Index.

SQL Structured Query Language.

TCP Transmission Control Protocol.

TUN Tunnel (-Device).

UDP User Datagram Protocol.

UI User Interface.

UML User-mode Linux.

VPN Virtual Private Network.

WLAN Wireless Local Area Network.

XML Extensible Markup Language.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

A. Erklärung über die eigenständige Arbeit

Wir erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde.
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben.
- dass wir keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt haben.

Rapperswil, 15. Juni 2012

Ralf Sager

Giuliano Grassi

B. Persönlicher Bericht Giuliano Grassi

Verlauf

Zu Beginn der Bachelorarbeit stand die Einarbeitung und Orientierung in der vorhandenen strongSwan Software im Vordergrund sowie das Lesen der Standards (RFCs). Der spezielle objektorientierte Programmierstil von strongSwan stellte anfänglich gewisse Hürden dar. Nachdem das grundsätzliche Konzept der Software einigermassen verstanden war, stiessen wir beim Einsatz des Java Native Interfaces auf neue Herausforderungen. Durch meine beschränkten Kenntnisse in der C-Programmiersprache fiel es mir zuerst schwer, die benötigten Funktionen dafür zu schreiben. Mit Hilfe meines darin erprobten Bachelorpartners Ralf Sager konnten diese kleinen Probleme jedoch schnell beseitigt werden. Zeitgleich stand die Programmierung des User Interfaces auf dem Plan. Meine Erfahrung mit dem Android Framework sorgte dafür, dass wir schnell zu Ergebnissen kamen und uns auf den Kern der Software konzentrieren konnten. Nach etwa 8 Wochen stand bereits eine funktionsfähiger Prototyp für das Smartphone bereit, welcher sich mit dem HSR Gateway verbinden konnte und auf den Skripte-Server zugreifen konnte. Die Anforderungen waren eigentlich erfüllt, trotzdem haben wir uns entschlossen, die Applikation so auszubauen, dass man mehrere VPN-Profile abspeichern kann, da ansonsten jedes mal die mühsame Eingabe der Login-Daten erforderlich ist. Durch diese Entscheidung mussten wir das gesamte Erscheinungsbild der Applikation neu überdenken und anpassen. Eine der grössten Schwierigkeiten bestand darin, alle im Hintergrund gestarteten Threads in der richtigen Reihenfolge zu beenden, damit eine neue Verbindung hergestellt werden kann. Zum Schluss haben wir die Applikation auf verschiedene Arten getestet und die aufgetretenen Fehler behoben.

Gelerntes

Die gesamte Thematik unserer Bachelorarbeit war mir vorher zwar theoretisch bekannt, die praktische Umsetzung jedoch ein Rätsel. Die Analyse des bestehenden Source Codes und die darin verwendeten Paradigmen gefiel mir sehr und zeigte mir einen erweiterten Einblick in die Welt der C-Programmiersprache. Diese Analyse half mir, den Knoten zu lösen und mich im strongSwan Projekt zurecht zu finden.

Das Android Framework und sein Konzept waren mir bereits geläufig, die Verbindung eines C-Programs mittels JNI an die Android App stellte sich jedoch trotzdem als Knack-

punkt dar. Die aufgetretenen Probleme mit JNI zeigten mir, dass es umso wichtiger ist, angemessene Dokumentationen und Tutorials für Tools zu erstellen, damit weitere Entwickler sich zurecht finden können. Während dem gesamten Projektverlauf konnte ich mit verschiedenen Teilen des Android-Frameworks arbeiten, was mir einen noch besseren Überblick über diese Technologie verschaffte.

Fazit

Während dem gesamten Projekt stiessen wir immer wieder auf schier unüberwindbare Hürden. Nicht nur durch Fleiss und Überlegungen konnten wir diese Probleme lösen. Einmal einen Schritt zurück setzen um die Problematik von einem neuen Gesichtspunkt anzusehen half dabei ebenfalls. Grundsätzlich zeigte mir die Bachelorarbeit, dass die Hingabe zur Erfüllung eines gesteckten Ziels enormen Einfluss auf das Team und dessen Freude an der Arbeit hat.

Ich bin mit dem erreichten Ergebnis zufrieden, leider blieb uns gegen Ende keine Zeit mehr übrig, die Applikation zu perfektionieren da die Dokumentation auch noch geschrieben werden musste.

Ich möchte mich an dieser Stelle noch bei meinem Partner, Ralf Sager für die ausgezeichnete Zusammenarbeit bedanken und wünsche ihm für seine Zukunft viel Erfolg, und hoffe wir können zu einem späteren Zeitpunkt wieder einmal ein spannendes Projekt realisieren. Ebenfalls möchte ich mich bei Prof. Dr. Andreas Steffen und Tobias Brunner für die Betreuung und kompetente Beratung während der Bachelorarbeit bedanken.

C. Persönlicher Bericht Ralf Sager

Bereits vor Beginn der Arbeit war mir klar, dass die Einarbeitung in die Architektur von strongSwan ein nicht unerheblicher Aufwand sein wird. Auch das Lesen der zahlreichen IPsec-RFCs war relativ zeitintensiv, obwohl mir die Funktionsweise der für die Arbeit relevanten Teile von IPsec bereits grösstenteils bekannt war.

Nachdem ich mir jedoch einen Überblick über strongSwan verschafft hatte und wir die Architektur unserer Software grob geplant hatten, schritt die Entwicklung zügig voran. Obwohl es nicht zwingend gefordert wurde, entschieden wir uns von Anfang an die ESP-Implementation nicht nur für Android, sondern parallel auch für "normale" GNU/Linux-Distributionen zu entwickeln. Dies stellte sich als die richtige Entscheidung heraus: die ESP-Implementation konnte so wesentlich einfacher entwickelt und getestet werden. Zudem konnten wir die Arbeit so gut untereinander aufteilen. Während ich mich vor allem mit der Implementation von ESP unter Linux beschäftigte, kümmerte sich Giuliano hauptsächlich um die Android-spezifischen Komponenten.

Obwohl ich mich schon mit vielen Protokollen auseinandergesetzt habe, war die Umsetzung von ESP etwas spezielles. Da es (normalerweise) direkt auf IP aufsetzt, unterscheidet sich sich die Implementation stark von typischen mir bekannten TCP- oder auch UDP-basierten Protokollen. Ein Beispiel ist die Verifizierung der Sequenznummern - bei TCP-basierenden Protokollen ist dafür kein Sliding-Window Mechanismus notwendig, da die TCP-Pakete vom TCP-Stack bereits geordnet werden. Dadurch, dass es sich um eine Implementation im Userland handelt, war zudem etwas an Kreativität gefragt, um die Standards richtig abzubilden. Auch die Integration in strongSwan war teilweise schwierig, da die Interfaces natürlich auf die Verwendung des IPsec-Stacks im Kernel ausgelegt sind. Gerne hätte ich die Integration unter Linux noch verbessert, jedoch wurde die Zeit knapp und da die ESP-Implementation funktionierte, war das Hauptziel nun die Android-Applikation so brauchbar wie möglich zu machen.

Die Integration unter Android stellte sich als mühsam heraus, besonders da von C aus auf die von Android bereitgestellten Java-APIs zugegriffen werden musste. Dies führte zwangsläufig zu einer relativ komplexen Softwarearchitektur.

Trotz einiger Hürden ist ein gut funktionierendes Ergebnis entstanden, mit welchem ich zufrieden bin. Gerne hätte ich die Applikation zwar noch weiter verbessert und noch mehr zusätzliche Features implementiert, doch leider blieb uns keine Zeit mehr für viele der unzähligen Erweiterungsmöglichkeiten. Die in unseren Augen wichtigsten künftigen Erweiterungen konnte wir jedoch immerhin dokumentieren.

Die Zusammenarbeit mit meinem Partner Giuliano Grassi klappte ausgezeichnet und ich möchte ihm an dieser Stelle dafür meinen Dank aussprechen. Ausserdem möchte ich mich bei Prof. Dr. Andreas Steffen und Tobias Brunner für die Unterstützung bei dieser Arbeit bedanken. Besonders freut es mich natürlich, dass die entstandene Software als Teil des strongSwan-Projektes weiterhin gepflegt und weiterentwickelt wird.