Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Sicherer Schlüsselspeicher für KNX Data Secure

Diplomarbeit

zur Erlangung des akademischen Grades
Diplominformatiker

eingereicht von: Alexander Fehr
geboren am:
geboren in:

Gutachter/innen: Prof. Dr. Jens-Peter Redlich
Prof. Dr. Ernst-Günter Giessmann

eingereicht am: .....

Inhaltsverzeichnis

1 Einleitung **6**

2 Vorbetrachtungen **7**
2.1 KNX . . . 7
2.1.1 KNX-Telegrammformat . . . 8
2.2 KNX Data Secure . . . 9
2.2.1 KNX Data Secure-Telegrammformat . . . 9
2.2.2 AES-128 CCM in KNX Data Secure . . . 10
2.3 Raspberry Pi und Pigator-Modul . . . 13

3 Anforderungen **16**
3.1 Funktionale Anforderungen . . . 16
3.2 Finanzielle Anforderungen . . . 17
3.3 Sicherheitsanforderungen . . . 17
3.3.1 Schutzziele . . . 17
3.3.2 Angreifermodell . . . 17
3.3.3 Bedrohungsanalyse . . . 18

4 Vergleich von Speicherlösungen **19**
4.1 Speicherung im Dateisystem . . . 19
4.2 Speicherung auf einer Smartcard . . . 19
4.3 Speicherung in einem HSM . . . 20
4.4 Speicherung in einem TPM . . . 21
4.5 Speicherung in der ARM TrustZone . . . 21

5 Konzeptionelles Modell des Schlüsselspeichers **22**
5.1 Schichtenarchitektur . . . 22
5.1.1 IDS . . . 22
5.1.2 Keystoreschicht . . . 23
5.1.3 Abstraktionsschicht . . . 23
5.1.4 Backendschicht . . . 23
5.2 Lebenszyklus . . . 24
5.2.1 Initialisierung . . . 24
5.2.2 Betrieb . . . 25
5.2.3 Terminierung . . . 25
5.3 Schlüssel-Wrapping . . . 25
5.4 Schlüssel-Speicherung im Dateisystem . . . 26
5.5 Benutzerauthentifizierung . . . 26
5.6 Gegenseitige Authentisierung und Secure Messaging . . . 27

6 Schnittstellenspezifikation und Funktionsbeschreibung 28
6.1 Keystore . . . 28
6.1.1 Schnittstelle . . . 28
6.1.2 Funktionen . . . 29
6.2 Abstraction . . . 31
6.2.1 Schnittstelle . . . 31
6.2.2 Funktionen . . . 32

7 Smartcards als Backend für den Schlüsselspeicher 35
7.1 Anforderungen an die Smartcard . . . 35
7.1.1 Format und Schnittstelle . . . 35
7.1.2 Kryptographische Algorithmen . . . 36
7.1.3 Schlüssel-Management . . . 36
7.1.4 Authentifizierung und Secure Messaging . . . 36
7.2 ISO 7816-Kommandos . . . 37
7.2.1 Kryptographische Operationen . . . 37
7.2.2 Schlüssel-Wrapping . . . 37
7.2.3 Authentifizierung . . . 38
7.2.4 Terminierung . . . 38
7.3 Smartcardmodelle mit festem Befehlssatz . . . 39
7.3.1 CryptoMate Nano . . . 39
7.3.2 Nitrokey Pro 2 . . . 40
7.3.3 TCOS-Smartcard . . . 41
7.3.4 SmartCard-HSM . . . 41
7.3.5 Zusammenfassung . . . 42

8 Java Card 44
8.1 Einführung in Java Card . . . 44
8.2 GlobalPlatform Card Specification . . . 45
8.3 Auswahl von Java Cards . . . 45
8.4 Java Card Applet für den KNX-Schlüsselspeicher . . . 46
8.4.1 Kryptographische Operationen . . . 46
8.4.2 Schlüssel-Wrapping . . . 47
8.4.3 Authentifizierung . . . 47
8.4.4 Terminierung . . . 47

9 Cryptographic Service Provider 48
9.1 CSP als Backend für den Schlüsselspeicher . . . 48
9.1.1 Kryptographische Operationen . . . 48
9.1.2 Schlüssel-Wrapping . . . 49
9.1.3 Authentifizierung . . . 49
9.1.4 Terminierung . . . 49
9.1.5 Sonstiges . . . 49
9.1.6 Bewertung . . . 50

10 Ausblick **51**

11 Fazit **52**

12 Literatur **53**

Abbildungsverzeichnis

1 Verbindung von Geräten über den KNX-Bus 7
2 KNX-Telegramm auf dem TP1-Medium 8
3 KNX Data Secure-Telegramm (Authentisierung) 10
4 KNX Data Secure-Telegramm (Verschlüsselung und Authentisierung) . . 10
5 Blockdiagramm für CBC-MAC 11
6 Blockdiagramm für AES-CTR Verschlüsselung 12
7 Blockdiagramm für AES-CTR Entschlüsselung 13
8 Raspberry Pi 3 Model B+ 14
9 Pigator mit KNX-Modul 15
10 Schichtenmodell des Schlüsselspeichers 22
11 Gegenseitige Authentisierung mit Schlüsselaushandlung 27
12 CryptoMate Nano USB-Token 39
13 Nitrokey Pro 2 USB-Token 40
14 SmartCard-HSM 4K USB-Token 42
15 Ablauf bei der Java Card Appletentwicklung 44

1 Einleitung

Smart-Home-Technologien haben mittlerweile Einzug in viele Haushalte gehalten. Die Verbreitung der intelligenten Vernetzung von Haustechnik wie Lampen, Heizungen und Rollläden wird auch zukünftig weiter zunehmen. Eine schon lange am Markt etablierte Lösung für die Verbindung von Smart-Home-Geräten ist der KNX-Standard. Dabei erfolgt die Kommunikation zwischen den Geräten auf dem KNX-Bus jedoch ungesichert, was sie zu einem potenziellen Angriffsziel macht. Erst in den letzten Jahren wurde die Spezifikation mit KNX Data Secure um Mechanismen erweitert, welche die Vertraulichkeit und Datenintegrität der KNX-Kommunikation gewährleisten sollen.

Ziel dieser Arbeit ist es, Anforderungen an einen Schlüsselspeicher für KNX Data Secure zu spezifizieren und ein Konzept zur deren Umsetzung zu erstellen. Weiterhin soll evaluiert werden, inwieweit sich auf dem Markt befindliche Smartcards dazu eignen, als Grundlage für den Schlüsselspeicher zu dienen. Der Schlüsselspeicher soll auf einem Raspberry Pi laufen und dort die Basis für ein zukünftiges KNX Intrusion Detection System bilden. Der Sicherheit des KNX-Schlüsselspeichers kommt dabei eine besondere Bedeutung zu, da er im Gegensatz zu einzelnen KNX Data Secure-Geräten viele oder sogar alle Schlüssel einer KNX Data Secure-Installation enthält. Eine Verletzung der Vertraulichkeit dieser Schlüssel hätte weitreichende Konsequenzen für die Sicherheit der gesamten KNX-Installation.

Die Struktur der vorliegenden Arbeit ergibt sich wie folgt: Zunächst werden in Kapitel 2 einige Grundlagen zu KNX und dem Raspberry Pi vermittelt. Anschließend werden in Kapitel 3 die Anforderungen an den Schlüsselspeicher definiert, bevor dann in Kapitel 4 unterschiedliche Lösungen zur Speicherung der Schlüssel kurz erklärt werden. In Kapitel 5 wird ein mögliches Konzept des Schlüsselspeichers vorgestellt, dessen Schnittstelle dann in Kapitel 6 spezifiziert wird. Kapitel 7 beschäftigt sich damit, ob marktübliche Smartcards als Basis für den Schlüsselspeicher geeignet sind und Kapitel 8 und 9 stellen mit der Java Card und dem Cryptographic Service Provider weitere Möglichkeiten dafür vor. Kapitel 10 gibt schließlich einen kurzen Überblick, wie das Thema sinnvoll fortgesetzt werden kann.

2 Vorbetrachtungen

Bevor näher auf den KNX-Schlüsselspeicher eingegangen wird, sollen erst mal einige in diesem Kontext wichtige Grundlagen vorgestellt werden. Einerseits geht es um KNX und dessen Absicherung KNX Data Secure und andererseits um den Raspberry Pi inklusive dessen Anbindung an den KNX-Bus.

2.1 KNX

Bei KNX [1] handelt es sich um einen offenen Standard für Heim- und Gebäudeautomation, dessen Spezifikationen von der KNX Association [2] entwickelt und publiziert werden. KNX-Installationen steuern unter anderem Heizungs-, Beleuchtungs- und Zugangskontrollsysteme und man findet diese sowohl in Einfamilienhäusern als auch in Büro- oder Industriegebäuden.

Ein Feldbus verbindet verschiedene KNX-Geräte miteinander (Abbildung 1), wobei dieser sogenannte KNX-Bus auch unabhängig vom Stromnetz verlegt werden kann. Neben kabelgeführten Twisted-Pair-Bussen (TP0, TP1), gibt es auch IP-basierte (KNXnet) und funkbasierte Lösungen (KNX-RF). Eine KNX-Installation kann maximal 15 Bereiche mit jeweils 15 Linien und 256 Bus-Teilnehmern pro Linie beinhalten. Die Palette an möglichen KNX-Geräten reicht dabei von Systemgeräten wie Kopplern zum Verbinden von Busabschnitten über Sensoren wie Bewegungsmeldern bis hin zu Aktoren wie Lichtdimmern. Die Geräte werden durch die KNX Association zertifiziert, was deren Interoperabilität in einer Installation ermöglicht.

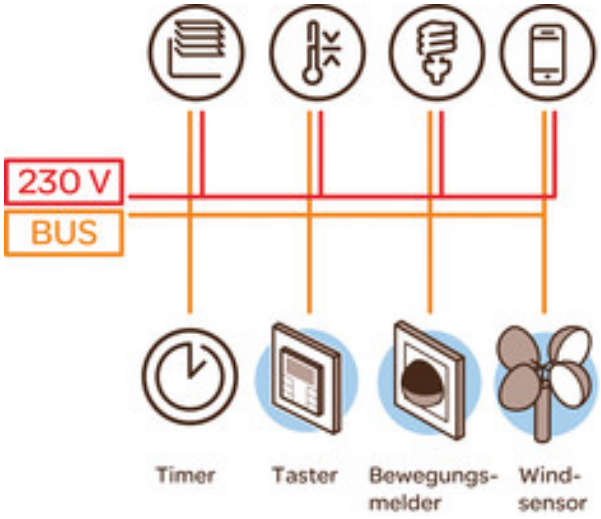

Abbildung 1: Verbindung von Geräten über den KNX-Bus (Quelle: [3])

Die Konfiguration der KNX-Installation wird mit Hilfe der Windows-basierten Software ETS (Engineering Tool Software) [4] durchgeführt. Die aktuell in Version 5 vorliegende Software ist in verschiedenen Varianten erhältlich, welche sich durch die maximale Anzahl der konfigurierbaren Geräte und entsprechend auch im Preis unterscheiden.

2.1.1 KNX-Telegrammformat

Die Datenpakete auf dem KNX-Bus werden in Form von Telegrammen versendet. Abbildung 2 zeigt ein solches Telegramm für das Twisted-Pair-1-Medium [5]. Das Kontrollbyte enthält unter anderem den Telegramm-Typ, also die Information ob es sich um ein Standard- oder erweitertes Paket handelt. Die maximale Länge eines Standard-Telegramms beträgt 23 Oktetts und bei erweiterten Frames sogar 263 Oktetts. Die jeweilige Quell- und Zieladresse eines Telegramms wird in der Form <Bereich>.<Linie>.<Teilnehmer> angegeben. Außerdem können zusammengehörige Geräte auch gruppiert werden und eine entsprechende Gruppenadresse hat dann die Form <Hauptgruppe>.<Mittelgruppe>.<Untergruppe>. Das DRL-Byte enthält den Typ der Zieladresse (Unicast-Adresse oder Gruppenadresse) und gibt die Länge der folgenden Nutzdaten als 4 Bit-Information an. Bei erweiterten Paketen wird das gesamte Oktett für die Längeninformation genutzt. Die restlichen Informationen des DRL-Bytes werden dabei in ein erweitertes Kontrollbyte direkt nach dem Standard-Kontrollbyte verlagert. Die TPCI (Transport Layer Protocol Control Information) beschreibt den Zweck des Telegramms (Datenpaket oder Kontrolldaten) und enthält eine optionale Sequenznummer der KNX-Transportschicht. Die APCI (Application Layer Protocol Control Information) ist ein 4 oder 10 Bits langer Service-Identifier, welcher die jeweiligen Nutzdaten charakterisiert. Nach den Nutzdaten folgt noch eine Checksumme, welche die invertierte, bitweise XOR-Verknüpfung aller Bytes des Telegramms darstellt.

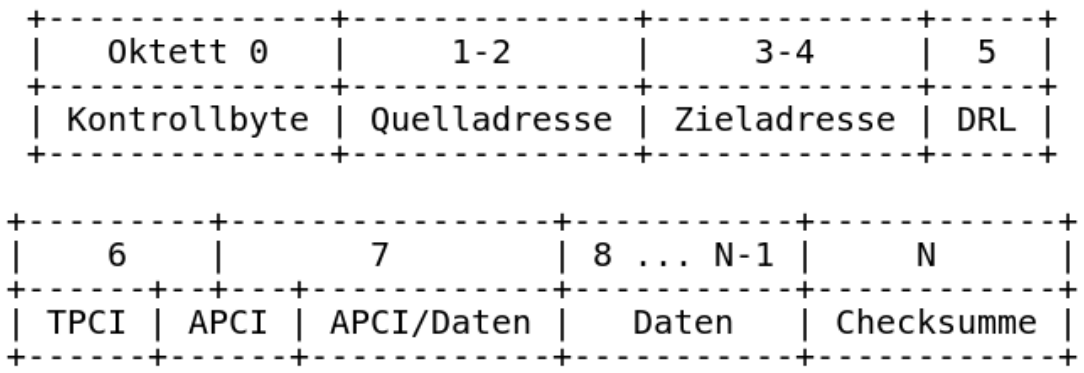

Abbildung 2: KNX-Telegramm auf dem TP1-Medium

2.2 KNX Data Secure

Da auf dem klassischen KNX-Bus Daten unverschlüsselt und nicht authentifiziert übertragen werden, stellen besonders Bus-Leitungen die ins Freie führen, wie bei Außensensoren, ein erhebliches Sicherheitsrisiko dar. Speziell sicherheitsrelevante Systeme, wie beispielsweise Alarmanlagen oder Torsteuerungen, können so mögliche Ziele für einen Angreifer werden. Um das zu verhindern und die Laufzeitkommunikation auf dem KNX-Bus abzusichern, wurde mit KNX Data Secure [6] von der KNX Association eine Lösung geschaffen.

KNX Data Secure schützt die Nutzdaten durch Verschlüsselung und Authentisierung vor unberechtigtem Zugriff und Manipulation. Dabei werden Datenauthentizität und -integrität (Authentifikationscodes), Vertraulichkeit (Verschlüsselung) und Freshness (Sequenznummern) sichergestellt. KNX Data Secure nutzt dafür den AES-128-CCM-Algorithmus, das bedeutet den Counter Mode zur Verschlüsselung und den CBC-MAC zur Integritätssicherung.

Die AES-Schlüssel für die Laufzeitkommunikation werden in einem mehrstufigen Prozess erzeugt. Zuerst wird der mit jedem KNX Data Secure-fähigem Gerät ausgelieferte gerätespezifische FDSK (Factory Device Setup Key) in die ETS eingegeben. Anschließend wird von der ETS ein gerätespezifischer Werkzeugschlüssel erzeugt, welcher mit dem FDSK verschlüsselt und authentifiziert über den KNX-Bus an das zu konfigurierende Gerät gesendet wird. Ab diesem Punkt wird vom Gerät nur noch der Werkzeugschlüssel für die weitere Kommunikation mit der ETS akzeptiert. Schließlich erzeugt die ETS die benötigten Laufzeitschlüssel und sendet diese mit den entsprechenden Werkzeugschlüsseln verschlüsselt und authentifiziert an die jeweiligen Geräte. Dabei können pro kommunizierendem Gerätepaar eindeutige Laufzeitschlüssel auftreten, aber es können auch Schlüssel mehrfach Verwendung finden. AES-128 CCM verwendet den gleichen Schlüssel für die Integritätssicherung und Verschlüsselung.

2.2.1 KNX Data Secure-Telegrammformat

Das für KNX Data Secure verwendete Telegrammformat [7] entspricht bis Oktett 5 dem ungesicherten Telegramm und enthält ab Oktett 6 die gesicherte APDU. Die Abbildungen 3 und 4 zeigen ein solches Paket für die Fälle „nur Authentisierung“ und „Verschlüsselung und Authentisierung“. Die Secure APCI zeigt hier an, dass es sich um eine Secure APDU handelt. Das SCF (Security Control Field) gibt Auskunft darüber welcher Verschlüsselungsalgorithmus und Betriebsmodus verwendet wird, wobei im Moment nur AES-128 CCM zugelassen ist. Außerdem wird hier angezeigt, ob neben der Integritätssicherung auch die Verschlüsselung Anwendung findet. Die Sequenznummer sorgt für Daten-Freshness und schützt das System so vor Replay-Attacken. Bei Verschlüs-

selung enthält die Cipher-APDU den Ciphertext der APCI und Nutzdaten, ansonsten findet man hier die Plain-APDU. Beide werden jeweils noch durch den MAC ergänzt.

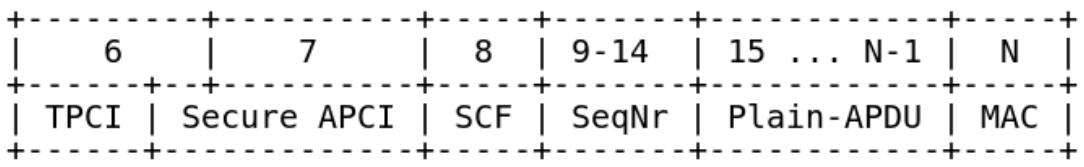

Abbildung 3: KNX Data Secure-Telegramm (Authentisierung)

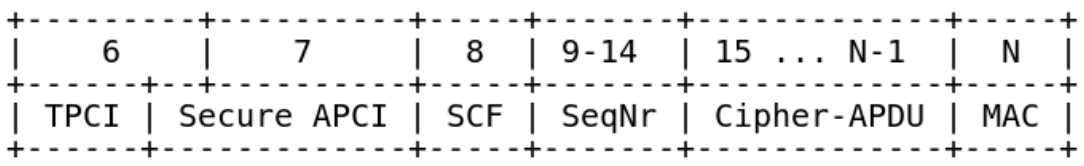

Abbildung 4: KNX Data Secure-Telegramm (Verschlüsselung und Authentisierung)

2.2.2 AES-128 CCM in KNX Data Secure

Nachfolgend soll dargestellt werden, wie der AES-128 CCM KNX-spezifisch abläuft, um den MAC und optional die Cipher-APDU zu berechnen [7].

Falls nur der MAC berechnet wird, bleibt die Payload P leer und die Plain APDU wird in die zugehörigen Daten A eingefügt.

A = Secure APCI | SCF | SeqNr | Plain APDU (= APCI + Nutzdaten)
P = $\emptyset$

Falls das Paket zusätzlich noch verschlüsselt werden soll, enthält die Payload die Plain APDU.

A = Secure APCI | SCF | SeqNr
P = Plain APDU

A und P werden durch Null-Padding auf Blocklänge verlängert und daraus ergeben sich dann die Nachrichten-Blöcke B_1 bis B_n. Die Länge von A in Oktetten wird als 16 Bit-Wert a angegeben.

B_1 bis B_n = a | A_{padded} | P_{padded}

Der Block B_0 beinhalten Felder aus dem KNX-Telegramm (SeqNr, Quelladresse, Zieladresse, FT, AT und TPCI), damit Veränderungen an diesen erkannt werden können. Q ist die Länge der Payload in Oktetten als 8 Bit-Wert.

B_0 = SeqNr | Quelladresse | Zieladresse | FT | AT | TPCI | 00h | 00h | Q

Beim CBC-MAC wird als Initialisierungsvektor der Nullvektor gewählt. Der Schlüssel k ist ein AES-128-Schlüssel.

Der erste Ausgabeblock Y_0 ergibt sich durch AES-Verschlüsselung mit dem Schlüssel k aus dem Eingabeblock B_0.

$Y_0 = AES_k(B_0)$

Daraus folgen dann schrittweise die weiteren Ausgabeblöcke (i = 1 ... n).

$Y_i = AES_k(Y_{i-1} \oplus B_i)$

Die 32 höchstwertigen Bits von Y_n ergeben den MAC.

$MAC = MSB_{32}(Y_n)$

Abbildung 5 zeigt das entsprechende Blockdiagramm für den CBC-MAC.

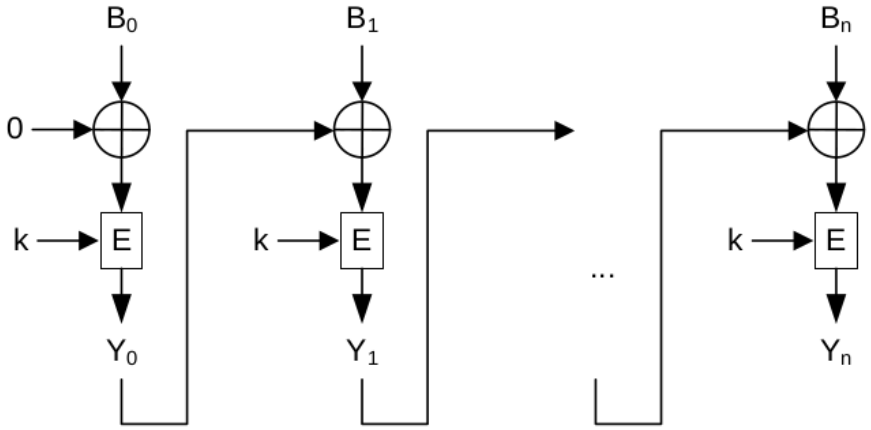

Abbildung 5: Blockdiagramm für CBC-MAC (Quelle: [7])

Der Blockcounter Ctr_j (j = 0 bis n) für die Verschlüsselung nach dem Counter-Modus enthält mehrere Felder aus dem KNX-Telegramm. Für jeden weiteren Block wird dieser um jeweils eins inkrementiert.

Ctr_j = SeqNr | Quelladresse | Zieladresse | 0000000001h | j

Die Ausgabeblöcke S_0 bis S_n (j = 0 bis n) ergeben sich durch AES-Verschlüsselung mit dem Schlüssel k aus den jeweiligen Counterwerten.

$S_j = AES_k(Ctr_j)$

Der Stream-Block S ergibt sich dann aus den einzelnen Ausgabeblöcken.

$S = S_1 \mid S_2 \mid ... \mid S_n$

Abschließend resultiert der Ciphertext C aus einem bitweisen Exklusiv-Oder vom Stream-Block S und der Payload P und dem Anhängen des verschlüsselten MAC. P_{len} ist dabei die Länge der Payload P in Bits.

$C = (P \oplus MSB_{P_{len}}(S)) \mid (MSB_{32}(Y_n) \oplus MSB_{32}(S_0))$

Abbildung 6 zeigt das entsprechende Blockdiagramm für die Verschlüsselung im Counter-Modus.

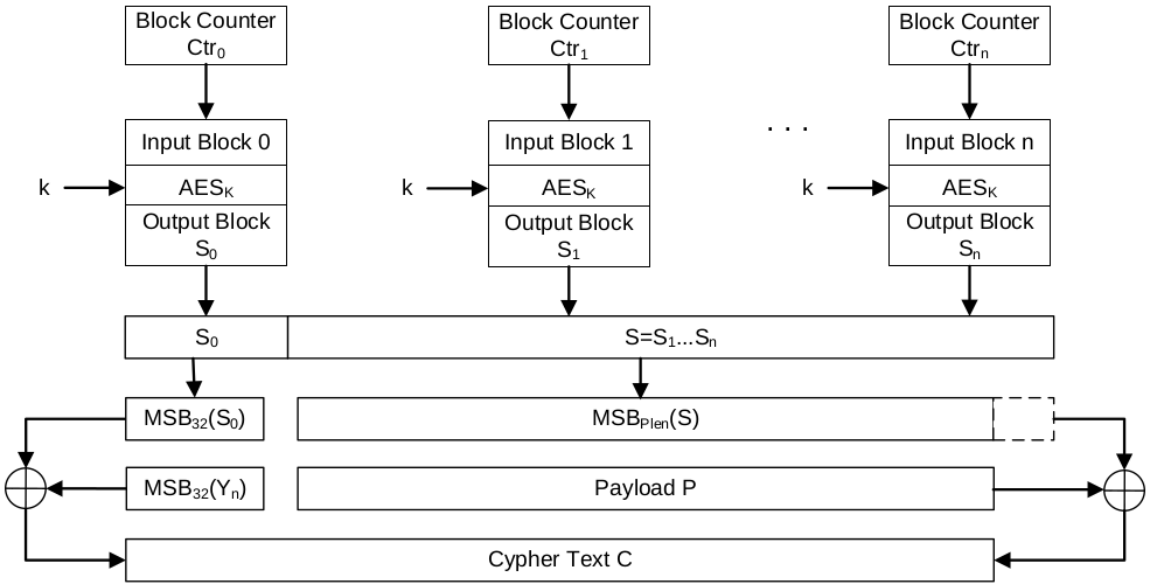

Abbildung 6: Blockdiagramm für AES-CTR Verschlüsselung (Quelle: [7])

Die Telegramm-Entschlüsselung und Verifizierung erfolgt nach folgendem Verfahren.

Als erstes wird die erhaltene Nachricht entschlüsselt und so die Payload wieder hergestellt. Ctr_j, S_0 und S ergeben sich wie bereits oben beschrieben. C_{len} ist die Länge des Ciphertextes in Bits.

$P = MSB_{C_{len}-32}(C) \oplus MSB_{C_{len}-32}(S)$

Abbildung 7 zeigt das entsprechende Blockdiagramm für die Entschlüsselung im Counter-Modus.

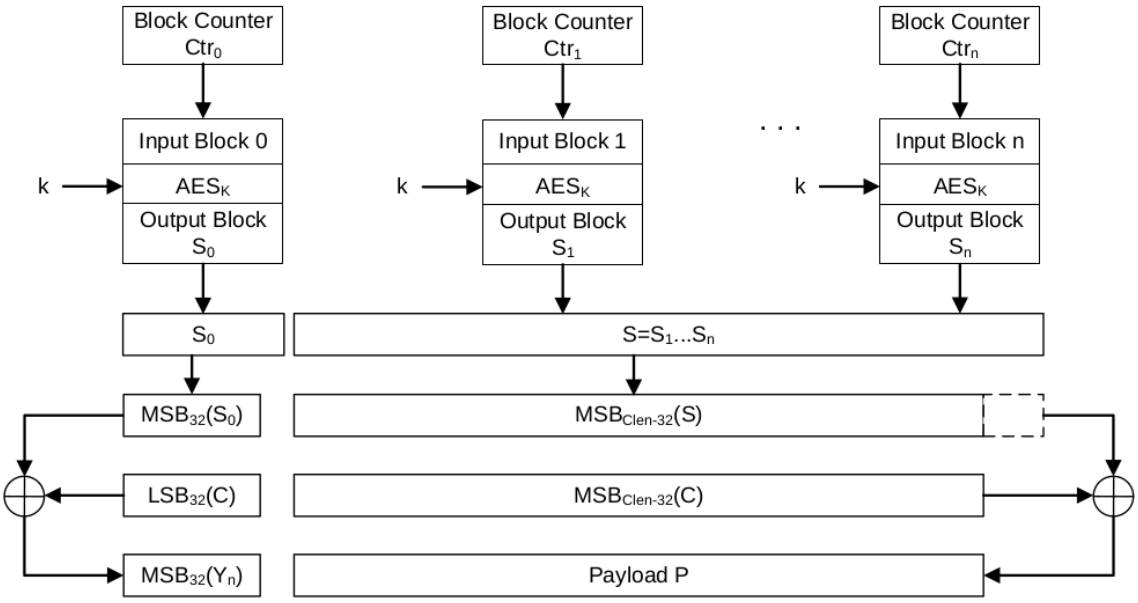

Abbildung 7: Blockdiagramm für AES-CTR Entschlüsselung (Quelle: [7])

Anschließend wird der empfangene MAC T_r rekonstruiert.

$T_r = LSB_{32}(C) \oplus MSB_{32}\ (S_0)$

Zum Vergleich wird der MAC der empfangenen Nachricht berechnet. Dabei ergeben sich B_0 und B_1 bis B_n wie oben beschrieben und auch Y_0 bis Y_n werden genauso berechnet.

$MAC = MSB_{32}(Y_n)$

Das empfangene Telegramm wird nur dann als gültig anerkannt, wenn der berechnete MAC $MSB_{32}(Y_n)$ dem empfangenen MAC T_r entspricht.

2.3 Raspberry Pi und Pigator-Modul

Die Basis für das Intrusion Detection System und den darunterliegenden Schlüsselspeicher bildet ein Raspberry Pi [8]. Dabei handelt es sich um einen kreditkartengroßen ARM-basierten Einplatinencomputer, welcher seit 2012 von der Raspberry Pi Foundation [9] entwickelt und vertrieben wird. Ursprüngliche Zielsetzung des Projekts war es, Jugendlichen einen günstigen Computer zum Erwerb von Programmier- und Hardwarekenntnissen zur Verfügung zu stellen und somit die digitale Bildung zu fördern. Daneben entstanden mit der Zeit vielfältige Soft- und Hardwareerweiterungen, die den Raspberry Pi für ein breites Anwendungsspektrum, wie beispielsweise Multimedia Center, Robotersteuerung aber auch für Smart Home einsetzbar machen.

Das aktuelle Modell Raspberry Pi 3 B+ (Abbildung 8) kostet etwa 35 EUR und hat einen 1,4 GHz 64-Bit-Vierkernprozessor, 1 GB RAM, Gigabit Ethernet, Wireless LAN, Bluetooth, einen HDMI-Anschluss sowie vier USB 2.0 Ports. Weiterhin verfügt der Raspberry Pi über eine frei programmierbare 40-Pin GPIO-Schnittstelle (General Purpose Input/Output), über welche zahlreiche Erweiterungen wie LEDs oder Sensoren angesteuert werden können. Die vorhandenen GPIO-Pins können dabei wahlweise als SPI-, I2C- oder UART-Schnittstelle verwendet werden.

Abbildung 8: Raspberry Pi 3 Model B+ (Quelle: [9])

Das Betriebssystem des Raspberry Pi kann von einer wechselbaren SD-Speicherkarte oder bei neueren Modellen auch von einem USB-Speichergerät gebootet werden. Dabei kann neben verschiedenen Linux-Distributionen auch eine spezielle Version von Windows 10 zum Einsatz kommen. Ein weit verbreitetes und von der Raspberry Pi Foundation empfohlenes Betriebssystem ist Raspbian, ein auf Debian GNU/Linux basierendes System. Einen Vorteil von GNU/Linux als Betriebssystem für den Raspberry Pi stellt der bereits vorhandene vollständige und leistungsstarke Softwarestack dar.

Um den Raspberry Pi mit dem KNX-Bus zu verbinden, kommt der Pigator [10] von Busware zur Anwendung. Dabei handelt es sich um einen Modulträger für die GPIO-Schnittstelle des Raspberry Pi, welcher um verschiedene Module, unter anderem ein KNX-Modul (Abbildung 9), erweitert werden kann. Die Preise für den Modulträger und das zugehörige Pigator-Modul betragen jeweils 39 EUR. Softwareseitig wird das Pigator KNX-Modul von knxd [11] unterstützt, einer unter Linux lauffähigen Software, welche es ermöglicht, ein preiswertes KNX/IP-Gateway aufzubauen, dass eine Alternative zu den von der KNX Association zertifizierten Gateways darstellt.

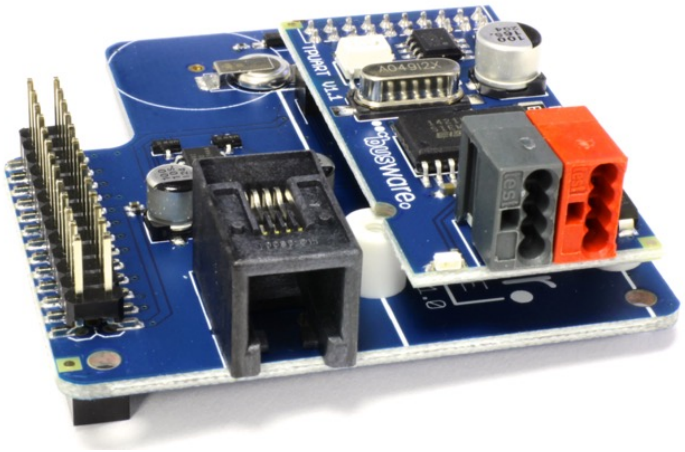

Abbildung 9: Pigator mit KNX-Modul (Quelle: [10])

3 Anforderungen

Im Folgenden werden die Anforderungen an den geplanten KNX Data Secure-Schlüsselspeicher beschrieben. Es wird dabei auf funktionale, finanzielle und die Sicherheit betreffende Aspekte eingegangen.

3.1 Funktionale Anforderungen

Das KNX-Intrusion Detection System inklusive dem Schlüsselspeicher soll auf einem Raspberry Pi laufen und an den KNX-Bus angebunden sein. Vom Bus erhält das IDS KNX-Pakete, welche es weiter analysiert, um so potenzielle Angriffe zu erkennen.

Die KNX Data Secure-Pakete werden vom IDS an den Schlüsselspeicher übergeben, damit dieser sie entschlüsselt und anschließend deren Integrität und Authentizität prüfen kann. Dazu muss der Schlüsselspeicher AES-128 im CCM-Betriebsmodus beherrschen. Da keine eigenen KNX-Telegramme erzeugt werden, reicht die Entschlüsselung im Counter-Modus und die Verifizierung durch CBC-MAC.

Für die AES-Operationen müssen die zugehörigen Schlüssel durch Import im System hinterlegt werden. Der konzipierte Schlüsselspeicher soll in der Lage sein, bis zu 3000 KNX Data Secure-Laufzeitschlüssel persistent zu speichern. Diese Anzahl an Schlüsseln wurde so gewählt, damit das System auch in großen KNX-Installationen eingesetzt werden kann, in welchen verschiedene Data Secure-Geräte und auch deren Gruppen jeweils eindeutige Schlüssel für die Kommunikation untereinander verwenden.

Während der Betriebsphase des IDS soll der Zugriff auf die kryptografischen Operationen des Schlüsselspeichers ohne Nutzerinteraktion möglich sein, da das IDS durchgehend läuft. Wünschenswert wäre diese Funktionalität auch direkt nach einem Neustart des Systems, wie er beispielsweise nach einem Stromausfall auftreten kann, wobei es das aber aus Sicherheitssicht durchaus kritisch abzuwägen gilt.

Die vom IDS durchgeführte Überwachung der Telegramme sollte möglichst lückenlos und falls gewünscht in Echtzeit erfolgen, damit keine Pakete verworfen werden müssen. Hierzu sollte die Geschwindigkeit der entsprechenden kryptografischen Operationen für die Datenmenge auf dem KNX-Bus ausreichend hoch sein.

Es wäre außerdem von Vorteil, wenn der Schlüsselspeicher portabel wäre, also nicht an einen bestimmten Raspberry Pi gebunden ist. Er könnte so bei Bedarf flexibel an verschiedenen Stellen in der KNX-Installation eingesetzt werden. Auch könnte der Schlüsselspeicher nach einem Angriff unter Ausnutzung einer Betriebssystem-Schwachstelle

an einem „sauberen“ Raspberry Pi weiter verwendet werden. Das gilt natürlich nur, falls die Vertraulichkeit der Laufzeitschlüssel nicht verletzt wurde.

3.2 Finanzielle Anforderungen

Die Kosten für das Gesamtsystem sollen sich im niedrigen dreistelligen Eurobereich bewegen. Das bedeutet, dass sich der Preis für mögliches Soft- und Hardwarezubehör für den Schlüsselspeicher im Rahmen des Raspberry Pi und der Pigator-Module bewegen sollte.

3.3 Sicherheitsanforderungen

3.3.1 Schutzziele

Folgende Schutzziele der Informationssicherheit [12] sollen durch den sicheren Schlüsselspeicher umgesetzt werden.

Authentizität Es soll sichergestellt sein, dass die im Schlüsselspeicher aufbewahrten KNX-Laufzeitschlüssel denen von der ETS erstellten entsprechen. Außerdem sollen nur authentifizierte Benutzer Zugriff auf bestimmte Funktionen des Schlüsselspeichers haben.

Integrität Die gespeicherten Laufzeitschlüssel sollen nicht unautorisiert und unbemerkt veränderbar sein.

Vertraulichkeit Es darf kein unautorisierter Lesezugriff auf die gespeicherten Laufzeitschlüssel erfolgen.

Verfügbarkeit Ein befugter Zugriff auf die Funktionen des Schlüsselspeichers soll nicht beeinträchtigt sein. Dazu gehört auch die vorgesehene Nutzung der entsprechenden Laufzeitschlüssel durch autorisierte kryptographische Operationen.

3.3.2 Angreifermodell

Da der Schlüsselspeicher die gesamten Laufzeitschlüssel einer KNX-Installation enthalten kann, erscheint ein Angriff auf diesen lohnenswert, auch weil er einen nachgelagerten

Angriff auf weitere KNX-Komponenten ermöglicht. KNX-Geräte wie Torsteuerungen oder Alarmanlagen könnten so ein mögliches Ziel von Angriffen durch Einbrecher werden.

Ein potenzieller Angreifer hat im vorgegebenen Szenario sowohl Zugriff übers Netz als auch physischen Zugriff auf den Raspberry Pi, auf welchem die IDS-Installation inklusive Schlüsselspeicher läuft. Der Netz-Zugriff umfasst sowohl den KNX-Bus als auch das Ethernet und kann das Abgreifen oder Senden beliebiger Datenpakete beinhalten. Der physische Zugriff auf den Raspberry Pi ermöglicht etwa das Kopieren der SD-Karte oder auch das Manipulieren des Raspberry oder seiner Zusatzhardware.

Bekannte Schwachstellen im Betriebssystem oder in den Diensten des Raspberry Pi, wie beispielsweise Buffer-Overflows, könnten von Angreifern ausgenutzt werden, um so möglicherweise Root-Rechte und somit volle Systemkontrolle zu erlangen. Auch die Installation von Malware, wie etwa Keyloggern um damit Tastatureingaben zu protokollieren, ist denkbar.

3.3.3 Bedrohungsanalyse

Der Schlüsselspeicher kann durch mögliche Ausnutzung von Schwachstellen und daraus resultierender Angriffe mehreren Bedrohungen ausgesetzt sein, welche verschieden schwere Gefährdungen darstellen.

Missbrauch der Entschlüsselungs- und Verifikationsfunktionen Der Angreifer erlangt Zugriff auf die Entschlüsselungs- und Verifikationsfunktionen des Schlüsselspeichers und kann diese für eigene Zwecke missbrauchen.

Verändern/Löschen der Laufzeitschlüssel Der Angreifer verändert oder löscht die gespeicherten Laufzeitschlüssel. Das führt möglicherweise zu einem Betriebsausfall des IDS-Systems (Denial of Service).

Kopieren der Laufzeitschlüssel Der Angreifer kommt in den Besitz einiger oder aller Laufzeitschlüssel. Diese Bedrohung ist sehr schwerwiegend und das entsprechende Szenario sollte unbedingt verhindert werden. Die Erneuerung sämtlicher Laufzeitschlüssel einer KNX-Installation ist sehr aufwendig und teuer und kann sogar temporär zu einem Betriebsausfall der KNX-Installation führen.

4 Vergleich von Speicherlösungen

In meiner Studienarbeit [13] habe ich verschiedene Möglichkeiten der Schlüsselspeicherung auf einem Raspberry Pi evaluiert. Nachfolgend sollen die Ergebnisse nochmal kurz zusammengefasst werden.

4.1 Speicherung im Dateisystem

Die einfachste und kostengünstigste Möglichkeit ist die Speicherung der Laufzeitschlüssel im Dateisystem der SD-Karte des Raspberry Pi. Da SD-Karten 256 GB und mehr Speicherkapazität haben, stellt eine Speicherung von 3000 128-Bit-AES-Schlüsseln kein Problem dar. Das Speichern der Schlüssel im Dateisystem ist allerdings nicht sicher. So kann beispielsweise Malware mit Root-Zugriff die Schlüssel auslesen und die SD-Karte kann bei physischem Zugriff auf den Raspberry Pi leicht kopiert werden.

Eine Alternative dazu stellt die Speicherung der Laufzeitschlüssel im Dateisystem verschlüsselt mit Passwort dar. Der für die Verschlüsselung nötige Schlüssel wird mit Hilfe eines kryptographischen Schlüsselableitungsverfahrens aus dem Passwort abgeleitet. Der Vorteil der verschlüsselten Speicherung besteht darin, das einfaches Kopieren der SD-Karte nicht direkt zum Besitz der Schlüssel führt. Allerdings kann das Passwort durch einen Keylogger oder durch einen Brute-Force-Angriff ermittelt werden. Ein weiteres Problem liegt darin, das die Schlüssel zur Laufzeit im Speicher des Raspberry Pi im Klartext vorliegen. Falls die direkte Verfügbarkeit der Schlüssel ohne Nutzerinteraktion gewünscht ist, muss das zugehörige Passwort allerdings in einer Datei hinterlegt werden, was das Sicherheitsniveau auf das der obigen Lösung senkt.

4.2 Speicherung auf einer Smartcard

Smartcards [14] sind Plastikkarten, die einen Prozessor, ROM und nichtflüchtigen EEPROM enthalten. Neben den Karten im Kreditkartenformat, gibt es auch andere Formfaktoren wie etwa USB-Token. Smartcards besitzen kryptografische Fähigkeiten und auf ihnen gespeicherte Daten sind gegen unerwünschte Zugriffe und gegen Manipulation geschützt. Ihre wesentlichen Eigenschaften werden in der ISO-Norm 7816 festgelegt. Die Anzahl der im EEPROM einer Smartcard speicherbaren Schlüssel hängt unter anderem von dessen Kapazität ab. Dabei kann die Bandbreite je nach Produkt und dessen Funktionalität von einem bis mehreren Hundert Schlüsseln reichen. Auch wenn es nicht ausgeschlossen ist, dürfte es schwer werden, eine Smartcard in der gewünschten Preisklasse zu finden, welche 3000 Schlüssel speichern kann.

Der Zugriff auf eine Smartcard kann durch eine PIN geschützt werden. Diese könnte zwar durch einen Keylogger abgegriffen werden, allerdings hat der Angreifer damit trotzdem nicht die Laufzeitschlüssel, welche unter der Annahme einer sicheren Smartcard nicht aus der Karte ausgelesen werden können. Brute-Force-Angriffe auf die PIN lassen sich durch einen Fehleingabezähler verhindern. Für die direkte Verfügbarkeit des Systems nach einem Neustart, müsste die PIN in einer Datei hinterlegt werden, was wiederum einen negativen Einfluss auf das Sicherheitsniveau hat. Ein Angreifer mit physischem Zugriff auf den Raspberry Pi kann die Smartcard mitnehmen und dadurch das System unbrauchbar machen, was aber für alle Speicherlösungen gilt. Auch hier hat er aber keinen direkten Zugriff auf die Schlüssel, da die Smartcard vor entsprechenden Manipulationen geschützt ist.

Falls eine größere Anzahl an Laufzeitschlüsseln in einer Smartcard gespeichert werden sollen, als in deren EEPROM-Speicher passen, kann ein hybrider Ansatz zur Speicherung gewählt werden. Dabei werden die Schlüssel auf der Karte verschlüsselt und dann in verschlüsselter Form ins Dateisystem der SD-Karte ausgelagert (Wrapping). Vor Benutzung der Schlüssel müssen diese wieder in die Smartcard importiert und intern entschlüsselt werden. Verglichen mit der reinen Smartcard-Lösung, ist aufgrund der zusätzlichen Kryptooperationen die Geschwindigkeit geringer. Des Weiteren ergibt sich ein zusätzlicher Angriffsvektor daraus, dass die Schlüssel im Dateisystem der SD-Karte liegen und ein Angreifer versuchen könnte deren Verschlüsselung offline zu brechen.

4.3 Speicherung in einem HSM

Ein Hardware-Sicherheitsmodul (HSM) [15] ist ein internes (PCI Express) oder externes (Netzwerkanbindung) Peripheriegerät, welches die effiziente und sichere Ausführung kryptographischer Operationen ermöglicht und einen umfangreichen Schutz der Schlüssel gegen verschiedene Arten von Angriffen bietet. Allerdings bewegt sich der Preis für ein solches Gerät im bis zu fünfstelligen Eurobereich, was preislich weit außerhalb des geforderten Rahmens liegt, weshalb stattdessen der Zymkey [16] vorgestellt wird.

Der Zymkey ist ein Sicherheitsmodul für den Raspberry Pi, welches in der Version 5 AES beherrscht und einen manipulationssicheren Schlüsselspeicher für 32 Schlüssel enthält. Der Zymkey wird auf die GPIO-Schnittstelle des Raspberry Pi gesteckt und dann über APIs für C, C++ und Python angesprochen. Der Preis des Zymkey liegt bei etwa 50 USD und das Sicherheitsniveau ist in etwa mit dem einer Smartcard vergleichbar, weshalb er für das geplante Szenario keinen relevanten Mehrwert gegenüber dieser bietet.

4.4 Speicherung in einem TPM

Ein Trusted Platform Module (TPM) [17][12] ist ein Chip, welcher der Spezifikation der Trusted Computing Group [18] folgt und grundlegende kryptografische Dienste implementiert. Ein TPM ist an eine konkrete Hardware-Instanz gebunden, welche dadurch eineindeutig identifizierbar wird. Die Speicherressourcen eines TPM sind beschränkt, wobei es nur etwa fünf bis zehn Schlüssel-Speicherslots [19] hat. Allerdings ist es durch Schlüssel-Wrapping möglich, eine nahezu unbegrenzte Anzahl verschlüsselter Schlüssel außerhalb des TPM abzulegen. Das Sicherheitsniveau von TPMs entspricht in etwa dem von Smartcards.

Ein TPM für den Raspberry Pi stellt das LetsTrust TPM von pi3g [20] dar. Dabei handelt es sich um ein GPIO-Aufsteckmodul, welches 25 EUR kostet. Das LetsTrust TPM basiert auf dem Infineon SLB 9670VQ2.0, welcher laut Spezifikation [21] kein AES implementiert. Damit scheidet es leider für den geplanten Einsatzzweck aus.

4.5 Speicherung in der ARM TrustZone

Die ARM TrustZone [22] ist eine Technologie zur Realisierung eines Trusted Execution Environment (TEE) [23]. Es existieren zahlreiche TEE-Implementationen, wobei es sich um eine Art vertrauenswürdiges Betriebssystem handelt, welches parallel zum Standardbetriebssystem läuft. Ein Beispiel für ein solches System für die ARM TrustZone ist OP-TEE (Open Portable Trusted Execution Environment), welches von der Linaro-Initiative weiterentwickelt wird und auch auf den Raspberry Pi 3 portiert wurde [24].

OP-TEE bietet die Möglichkeit Schlüssel vertrauenswürdig in einem Standard-Dateisystem zu speichern [25]. Dabei werden die Schlüssel vorher mit einem File Encryption Key (FEK) verschlüsselt. Mit dieser Methode ist es somit möglich, eine sehr große Anzahl an Laufzeitschlüsseln zu speichern. Das Sicherheitsniveau von TEEs ist ähnlich dem der hybriden Speicherung auf Smartcards oder im TPM. Im Vergleich zu den bisher vorgestellten Lösungen, ist es durch ein TEE allerdings möglich, dem Angreifer das Ausspähen eines Passworts oder einer PIN durch die Hardwareisolation zu erschweren.

Leider bietet der ARM-Prozessor des Raspberry Pi 3 keine vollständige ARM TrustZone-Implementation und der OP-TEE-Port dient lediglich Bildungs- und Testzwecken [24]. Dadurch ist der Speicherbereich für vertrauenswürdige Applikationen nicht vollständig vom restlichen Speicher isoliert. Das macht die ARM TrustZone des Raspberry Pi für das gewünschte Vorhaben unbrauchbar.

5 Konzeptionelles Modell des Schlüsselspeichers

In diesem Kapitel wird ein Konzept vorgestellt, welches die zuvor definierten Anforderungen an den sicheren Schlüsselspeicher umsetzt. Dabei wird eine zu realisierende Software-Komponente beschrieben, welche auf einem Raspberry Pi laufen soll und als Basis eine Smartcard nutzt.

5.1 Schichtenarchitektur

Die Architekturspezifikation des Schlüsselspeichers erfolgt als Schichtenmodell (Abbildung 10). Dieser besteht aus drei Schichten: Keystoreschicht, Abstraktionsschicht und Backendschicht. Darauf setzt das Intrusion Detection System auf, welches die Dienste des Schlüsselspeichers nutzt. Die dreischichtige Architektur ermöglicht es, verschiedene Hardware oder Dienste als Backend für kryptographische Operationen zu nutzen. Im hier vorgestellten Konzept bildet eine Smartcard das Backend. Da wir eine große Menge an Schlüsseln speichern wollen und das System nach oben hin skalierbar sein soll, wählen wir den im letzten Kapitel beschriebenen hybriden Ansatz zur Speicherung.

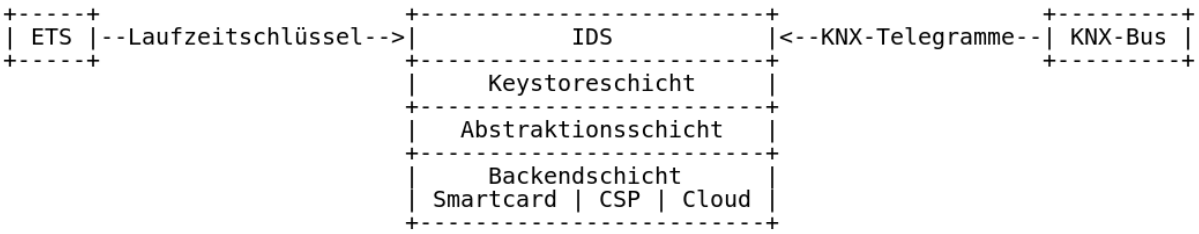

Abbildung 10: Schichtenmodell des Schlüsselspeichers

5.1.1 IDS

Das KNX-Intrusion Detection System zeichnet KNX Data Secure-Pakete vom Bus auf, analysiert diese und versucht so potenzielle Angriffsmuster zu erkennen. Bevor die Pakete weiter untersucht werden können, müssen diese entschlüsselt und verifiziert werden. Hierfür werden die Pakete an den Schlüsselspeicher weitergegeben, der diese kryptographischen Operationen als Dienst anbietet. Die Paketanalyse kann dabei sowohl in Echtzeit als auch geclustert zu einem späteren Zeitpunkt erfolgen, wobei das unter anderem von der Performance der tatsächlichen Umsetzung des Schlüsselspeichers abhängt. Da das IDS Teil einer weiteren zukünftigen Arbeit ist, wird es deshalb hier nicht weiter betrachtet.

5.1.2 Keystoreschicht

Die Keystoreschicht verwaltet die gespeicherten KNX-Laufzeitschlüssel. Es ist möglich Laufzeitschlüssel im Schlüsselspeicher zu installieren und auch wieder zu löschen. Die verschlüsselten (gewrappten) Laufzeitschlüssel werden dabei im Dateisystem gespeichert, um sie persistent verfügbar zu machen. Nähere Informationen dazu befinden sich in den Abschnitten 5.3 und 5.4.

Jedem Schlüssel ist eine Key-ID zugeordnet. Da die Laufzeitschlüssel eindeutig von der KNX-Quell- und Zieladresse abhängen, sollte die Key-ID deterministisch aus diesen abgeleitet werden. Die einfachste Methode dafür wäre beispielsweise die Konkatenation der Quell- und Zieladresse.

Die Keystoreschicht übergibt die zu entschlüsselnden und verifizierenden Pakete inklusive der richtigen Laufzeitschlüssel an die darunter liegende Schicht zur weiteren Bearbeitung. Die von dieser erhaltenen Plain-Pakete und der ermittelte Verifizierungsstatus werden an das IDS herausgegeben.

5.1.3 Abstraktionsschicht

Die Abstraktionsschicht kapselt die Backendfunktionalität, wobei eine Anpassung an die jeweilige Hardwarekomponente oder den Dienst erforderlich ist. Das Ziel dabei ist, ein einheitliches Interface gegenüber der Keystoreschicht bereitzustellen.

Eine weitere Aufgabe dieser Schicht ist es, fehlende Hardwarefunktionalität, soweit ohne Sicherheitseinschränkungen möglich, in Software zu implementieren. Ein Beispiel hierfür sind bestimmte Betriebsmodi von Blockchiffren.

Die Abstraktionsschicht ruft hardware- bzw. dienstspezifische Kommandos der Backendschicht auf. Die Ergebnisse dieser Kommandos werden wiederum an die Keystoreschicht zurückgegeben.

5.1.4 Backendschicht

Als Backend können verschiedene kryptographische Hardwarekomponenten oder Cloud-Dienste mit ihren jeweiligen Zugriffsschnittstellen dienen. Diese führen die benötigten kryptographischen Operationen wie Entschlüsselung, Verifizierung oder Wrapping in einer gesicherten Umgebung aus.

Für das hier spezifizierte Konzept wurde eine Smartcard als Backend gewählt, da diese wie zuvor gezeigt die für das geplante Szenario erforderlichen Anforderungen erfüllt. In Kapitel 9 wird zusätzlich noch auf Cryptographic Service Provider als mögliches Backend eingegangen.

Neben kryptographischer Hardware ist auch die Nutzung von entsprechenden Cloud-Diensten möglich. Allerdings bieten diese zusätzliche Angriffsvektoren und haben auch eine Reihe von Nachteilen bezüglich Verfügbarkeit und sicherem Löschen der Schlüssel. Aus diesem Grund werden Cloud-Dienste hier nicht weiter betrachtet.

5.2 Lebenszyklus

Der Lebenszyklus des Schlüsselspeichers kann in drei Phasen gegliedert werden. Es lassen sich Initialisierung, Betrieb und Terminierung unterscheiden. Diese Phasen werden nacheinander durchlaufen und es sind auch zyklischen Wiederholungen dieser möglich.

5.2.1 Initialisierung

In der Initialisierungsphase werden die KNX-Laufzeitschlüssel unter Angabe der Key-ID im Keystore installiert. Da die Laufzeitschlüssel in dieser Phase im Plaintext vorliegen, erfolgt die Initialisierung in einer isolierten, sicheren Umgebung.

Die Laufzeitschlüssel werden vom ETS generiert und anschließend in einer passwortgeschützten Datei abgelegt. Auf das konkrete Format dieser Datei und den genauen Ablauf des Transfers der Schlüssel in den Schlüsselspeicher wird nicht weiter eingegangen. Hier erfolgt erst mal die Annahme, dass sich die Laufzeitschlüssel bereits im IDS oder einer zusätzlichen Softwarekomponente befinden und von dieser an den Keystore übergeben werden.

Die übergebenen Laufzeitschlüssel werden in die Smartcard importiert, dort werden sie verschlüsselt (gewrappt) und in dieser Form wieder exportiert. Anschließend werden die gewrappten Schlüssel im Schlüsselspeicher persistent hinterlegt. Für das Wrapping wird während der Initialisierungsphase in der Smartcard ein Wrapping-Schlüssel erzeugt, welcher die Karte nie verlassen darf. Falls es zu einem späteren Zeitpunkt zur einer Konfigurationsänderung der KNX-Installation kommt (geänderte oder neue Laufzeitschlüssel), muss die gesamte Initialisierung inklusive neuem Wrapping-Schlüssel wiederholt werden.

5.2.2 Betrieb

Die Betriebsphase des Schlüsselspeichers entspricht dem regulären Betrieb des Intrusion Detection Systems auf dem Raspberry Pi. In dieser Phase werden die vom IDS übergebenen KNX-Telegramme mit AES-128 CCM entschlüsselt und verifiziert. Die dafür nötigen Laufzeitschlüssel werden aus dem Dateisystem geladen und anschließend in die Smartcard importiert und dort entschlüsselt (Unwrapping). Die so entpackten Schlüssel dürfen nicht von außen lesbar sein und nur den kryptographischen Operationen in der Smartcard zu Verfügung stehen.

5.2.3 Terminierung

Die Terminierungsphase bildet das Ende des Lebenszyklus des Schlüsselspeichers. Ziel dieser Phase ist es, dass es nicht mehr möglich sein soll, die kryptographischen Operationen unter Verwendung der Laufzeitschlüssel auszuführen. Das wird durch Löschen oder Überschreiben des Wrapping-Schlüssels in der Smartcard erreicht. Dadurch ist eine Nutzung der gewrappten KNX-Laufzeitschlüssel nicht mehr möglich. Zusätzlich können noch die gespeicherten Laufzeitschlüssel und der zuletzt in der Smartcard entpackte Schlüssel gelöscht werden. Wenn gewünscht, kann anschließend ein neuer Lebenszyklus des Schlüsselspeichers durch eine weitere Initialisierung begonnen werden.

5.3 Schlüssel-Wrapping

Wie bereits zuvor erwähnt, verwendet der Schlüsselspeicher sogenanntes Schlüssel-Wrapping, das bedeutet, dass die Laufzeitschlüssel durch Verschlüsselung „verpackt“ werden. So ist es möglich, die Schlüssel auch außerhalb der Smartcard ohne Verletzung der Vertraulichkeit abzulegen. Diese Lösung erlaubt es somit, eine sehr große Anzahl an Laufzeitschlüsseln zu speichern, welche nur durch den entsprechenden Datenträger limitiert wird.

Im vorliegenden Fall soll ein symmetrisches Kryptoverfahren für das Wrapping verwendet werden. Im Vergleich zu einem asymmetrischen Verfahren ist das symmetrische schneller und auch durch Quantensicherheit zukunftsfähiger. Außerdem erfolgt das Wrapping und Unwrapping innerhalb der Smartcard, weshalb der Vorteil von asymmetrischen Verfahren bei der Schlüsselverteilung (öffentlicher Schlüssel) hier nicht benötigt wird.

Da die Sicherheit des Gesamtsystems zu großem Teil auf der Sicherheit des Wrapping-Algorithmus beruht, soll für diesen AES-256 zur Verwendung kommen. Die Vertraulichkeit des Wrapping-Schlüssels darf zu keinem Zeitpunkt seiner Lebensdauer verletzt werden.

Deshalb wird der Wrapping-Schlüssel, unter Ausnutzung eines kryptographisch sicheren Zufallszahlengenerators, in der Smartcard generiert und darf diese auch nie verlassen.

Beim Wrapping innerhalb der Smartcard wird der Laufzeitschlüssel zusammen mit Zufallsdaten verschlüsselt. Die Zufallsdaten dienen der Erhöhung der Entropie. Optional kann während des Wrappings ein MAC über den Laufzeitschlüssel und die zugehörige Key-ID berechnet werden. Der MAC dient der Erkennung von unautorisierten Veränderungen der gespeicherten Schlüssel, wobei die Veränderungen bei entsprechendem Angriff dadurch nicht verhindert werden können.

5.4 Schlüssel-Speicherung im Dateisystem

Die durch das Wrapping verschlüsselten Laufzeitschlüssel werden in der Initialisierungsphase ins Dateisystem der SD-Karte geschrieben. Zusätzlich zu den Schlüsseln werden noch die zugehörigen Key-IDs und falls vorhanden der jeweilige MAC gespeichert. Eine mögliche Form der Speicherung ist, die Datenstruktur im Hauptspeicher welche alle Schlüssel enthält (z. B. Python-Dictionary) zu serialisieren und in eine Datei zu schreiben.

In der Betriebsphase wird die Datenstruktur mit den Schlüsseln aus der während der Initialisierung gespeicherten Datei eingelesen. Die 3000 Laufzeitschlüssel mit Key-ID und MAC sollten dabei problemlos in den Hauptspeicher des Raspberry Pi passen. Da während der Betriebsphase per Definition keine neuen Schlüssel hinzukommen, muss die Schlüsseldatei auch nur einmal zu Beginn eingelesen werden.

5.5 Benutzerauthentifizierung

Da die Initialisierung und Terminierung des Schlüsselspeichers kritisch für die Systemsicherheit sind, sollten diese nur unter Nutzeranwesenheit stattfinden. Das gilt besonders für das Generieren und Löschen des Wrapping-Schlüssels. Dafür muss sich der Benutzer gegenüber der Smartcard entsprechend authentisieren. Die Authentisierung erfolgt durch eine mindestens sechsstellige Persönliche Identifikationsnummer (PIN). Diese PIN wird während der Betriebsphase nie eingegeben.

Da in der operativen Phase der Betrieb des Schlüsselspeichers (auch nach einem Neustart) ohne Nutzerinteraktion möglich sein soll, wäre eine Authentisierung per PIN an dieser Stelle nicht zielführend, da die PIN sowieso im Dateisystem hinterlegt werden müsste. Auch würde eine dauerhaft automatisch ablaufende PIN-Eingabe viele Schreibvorgänge des Fehlbedienungszählers im EEPROM verursachen, was dessen Lebensdauer nicht zuträglich

wäre. Deshalb erfolgt in der Betriebsphase eine Authentisierung mit Schlüsselaushandlung fürs Secure Messaging durch asymmetrische kryptographische Algorithmen.

5.6 Gegenseitige Authentisierung und Secure Messaging

Da die Betriebsphase in einer potenziell unsicheren Umgebung abläuft und eine Authentisierung durch PIN wie oben erklärt nicht ausreicht, wird eine Authentisierung durch asymmetrische kryptographische Verfahren und der Einsatz von Secure Messaging angestrebt. Das nachfolgend vorgestellte Konzept setzt die gewünschte Funktionalität um, ist aber stark von der zum Einsatz kommenden Smartcard abhängig.

Die gegenseitige Authentisierung zwischen der Smartcard und der Raspberry-Software soll durch ein Challenge-Response-Verfahren auf Basis elliptischer Kurven, wie Elliptic Curve DSA (ECDSA) erfolgen. Der Vorteil im Vergleich zu RSA liegt in den geringeren Parameterlängen und dem daraus resultierenden Performance-Gewinn innerhalb der Smartcard. Die für die Authentisierung nötigen Schlüsselpaare können in der Initialisierungsphase von der Smartcard und der Raspberry-Software generiert werden. Da die Initialisierung in einer sicheren Umgebung stattfindet, können die jeweiligen öffentlichen Schlüssel anschließend ausgetauscht und für die Betriebsphase hinterlegt werden.

In den Authentisierungsprozess ist ein Mechanismus zur Schlüsselaushandlung integriert (Abbildung 11). Durch die Kombination ist es möglich, die Identität der jeweiligen Instanz bei der Aushandlung nachzuweisen und somit Man-in-the-Middle-Angriffe zu verhindern. Als Verfahren zur Aushandlung der Schlüsseldaten soll hier Ephemeral Elliptic Curve Diffie-Hellmann (ECDHE) genutzt werden. Anschließend werden aus den Aushandlungsdaten die Session-Schlüssel für die Absicherung der Kommunikation zwischen Smartcard und Raspberry-Software durch Secure Messaging abgeleitet.

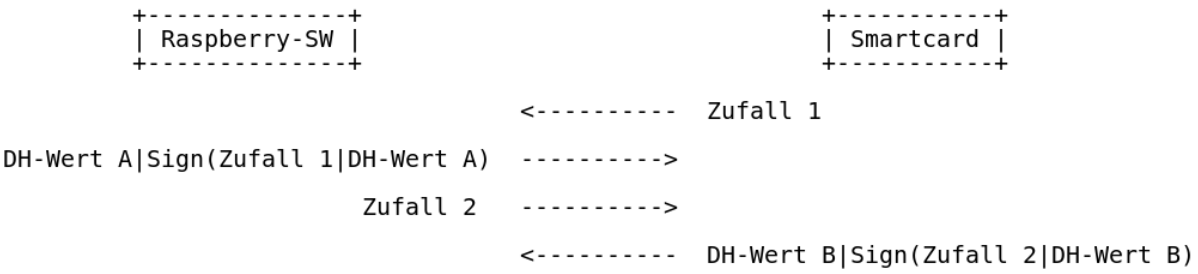

Abbildung 11: Gegenseitige Authentisierung mit Schlüsselaushandlung

6 Schnittstellenspezifikation und Funktionsbeschreibung

In diesem Kapitel sollen die Schnittstellen und Funktionen des zuvor konzeptionell beschriebenen Schlüsselspeichers definiert werden. Dabei erfolgt die Spezifikation der Keystore- und Abstraktionsschicht in einem Python-ähnlichen Pseudocode. Vieles ist im Vergleich zu einer tatsächlichen Implementation vereinfacht dargestellt, um die Lesbarkeit zu erhöhen und den Fokus auf das Wesentliche zu lenken. So fehlt beispielsweise eine Fehlerbehandlung, aber auch funktionelle Aspekte wie die gegenseitige Authentisierung zwischen Smartcard und Raspberry-Software wurden ausgelassen. An einigen Stellen beschreiben entsprechende Kommentare mögliche Erweiterungen oder Alternativen.

6.1 Keystore

6.1.1 Schnittstelle

```
# Initialisierung der Keystoreschicht
#
# initialization = True:  Initialisierungsphase
# initialization = False: Betriebsphase
#
Keystore.init(initialization)

# Importiert einen KNX-Laufzeitschlüssel in den Keystore
#
# key:    KNX-Laufzeitschlüssel
# key_id: Key-ID des Laufzeitschlüssels
#
Keystore.install_key(key, key_id)

# Entfernt einen KNX-Laufzeitschlüssel aus dem Keystore
#
# key_id: Key-ID des Laufzeitschlüssels
#
Keystore.remove_key(key_id)

# Entschlüsselt das KNX-Telegramm und prüft dessen MAC
#
# knx_telegram: Felder des KNX-Telegramms
# key_id:       Key-ID des Laufzeitschlüssels
#
# plaintext:    Plain-APDU
# mac_validity: Gültigkeit des MAC
#
```

```
Keystore.decrypt_and_verify(knx_telegram, key_id)
-> plaintext, mac_validity

# Prüft den MAC des KNX-Telegramms
#
# knx_telegram: Felder des KNX-Telegramms
# key_id:       Key-ID des Laufzeitschlüssels
#
# mac_validity: Gültigkeit des MAC
#
Keystore.verify(knx_telegram, key_id)
-> mac_validity

# Terminierung des Schlüsselspeichers
# (Löschung des Wrapping-Schlüssels)
#
Keystore.terminate()
```

6.1.2 Funktionen

```
Keystore.init(initialization)
  if not initialization and exists('keystore.dat'):
    # Betriebsphase

    # Schlüssel-Dictionary aus dem Dateisystem laden
    key_dict = load('keystore.dat')
  else:
    # Initialisierungsphase
    # (oder keystore.dat nicht vorhanden)
    initialization = True

    # leeres Schlüssel-Dictionary anlegen
    #   {<key_id>: <wrapped_key>}
    key_dict = {}

  # Initialisierung der Abstraktionsschicht
  Abstraction.init(initialization)


Keystore.install_key(key, key_id)
  # Schlüssel-Wrapping
  #  falls beim Wrapping ein MAC auch über die Key-ID
  #  berechnet werden soll, dann müsste diese hier
  #  auch mit übergeben werden
  wrapped_key = Abstraction.wrap_key(key)
```

```
  # im Schlüssel-Dictionary ablegen
  key_dict[key_id] = wrapped_key

  # Schlüssel-Dictionary ins Dateisystem schreiben
  #  besser wäre ein Schreiben erst am Ende der gesamten
  #  Initialisierung, damit die Datei nicht 3000 mal
  #  geschrieben werden muss
  dump(key_dict, 'keystore.dat')


Keystore.remove_key(key_id)
  # Entferne Schlüssel aus dem Schlüssel-Dictionary
  del key_dict[key_id]

  # Schlüssel-Dictionary ins Dateisystem schreiben
  dump(key_dict, 'keystore.dat')


Keystore.decrypt_and_verify(knx_telegram, key_id)
  # Schlüssel aus Schlüssel-Dictionary holen
  wrapped_key = key_dict[key_id]

  # Weiterreichen an Abstraktionsschicht
  plaintext, mac_validity =
  Abstraction.unwrap_decrypt_and_verify(knx_telegram, wrapped_key)

  return plaintext, mac_validity


Keystore.verify(knx_telegram, key_id)
  # Schlüssel aus Schlüssel-Dictionary holen
  wrapped_key = key_dict[key_id]

  # Weiterreichen an Abstraktionsschicht
  mac_validity =
  Abstraction.unwrap_and_verify(knx_telegram, wrapped_key)

  return mac_validity


Keystore.terminate()
  # lösche Schlüssel-Dictionary
  del key_dict

  # lösche Schlüssel-Datei
  remove('keystore.dat')
```

```
  # Terminierung in der Abstraktionsschicht
  Abstraction.terminate()
```

6.2 Abstraction

Die Abstraktionsschicht wird am Beispiel einer Smartcard spezifiziert. Bei einer wirklichen Implementation müssten die Funktionsaufrufe in entsprechende APDUs umgesetzt werden, welche sich nach dem jeweiligen Funktionsumfang der Karte richten.

6.2.1 Schnittstelle

```
# Initialisierung der Abstraktionsschicht
#
# initialization = True:  Initialisierungsphase
# initialization = False: Betriebsphase
#
Abstraction.init(initialization)

# Schlüssel-Wrapping
#
# key: KNX-Laufzeitschlüssel
#
# wrapped_key: verschlüsselter Laufzeitschlüssel
#
Abstraction.wrap_key(key)
-> wrapped_key

# Entschlüsselt das KNX-Telegramm und prüft dessen MAC
#
# knx_telegram: Felder des KNX-Telegramms
# wrapped_key:  verschlüsselter Laufzeitschlüssel
#
# plaintext:    Plain-APDU
# mac_validity: Gültigkeit des MAC
#
Abstraction.unwrap_decrypt_and_verify(knx_telegram, wrapped_key)
-> plaintext, mac_validity

# Prüft den MAC des KNX-Telegramms
#
# knx_telegram: Felder des KNX-Telegramms
# wrapped_key:  verschlüsselter Laufzeitschlüssel
#
```

```
# mac_validity: Gültigkeit des MAC
#
Abstraction.unwrap_and_verify(knx_telegram, wrapped_key)
-> mac_validity

# Terminierung des Schlüsselspeichers
# (Löschung des Wrapping-Schlüssels)
#
Abstraction.terminate()
```

6.2.2 Funktionen

```
Abstraction.init(initialization)
  if initialization:
    # Nutzerauthentisierung
    Smartcard.authenticate('Admin')

    # Wrapping-Schlüssel in der Smartcard erzeugen
    #  dieser bleibt in der Karte und kann nicht exportiert werden
    Smartcard.generate_wrapping_key()


Abstraction.wrap_key(key)
  # Wrapping innerhalb der Karte mit internem Wrapping-Schlüssel
  Smartcard.import_key(key, aes_keyslot)
  wrapped_key = Smartcard.wrap_key(aes_keyslot)

  return wrapped_key


Abstraction.unwrap_decrypt_and_verify(knx_telegram, wrapped_key)
  # benötigter Schlüssel wird in Smartcard installiert
  Smartcard.unwrap_key(wrapped_key, aes_keyslot)

  # erstelle den Counter nach KNX-Spezifikation
  counter = build_counter(knx_telegram)

  # Entschlüsselung im Counter-Modus
  #  abhängig von der Smartcard-Funktionalität muss hier eventuell
  #  der Counter-Modus in Software nachgebaut werden (durch ECB)
  plaintext = Smartcard.decrypt(knx_telegram.ciphertext,
                                counter + 1, aes_keyslot)

  # der MAC muss vor dem Prüfen erst entschlüsselt werden
  #  den tatsächlichen KNX-MAC bilden nur die 32 MSB
  #  (hier nicht umgesetzt)
```

```
  mac = Smartcard.decrypt(knx_telegram.mac, counter, aes_keyslot)

  # erstelle die Message nach KNX-Spezifikation
  message = build_message(knx_telegram, plaintext)

  # CBC-MAC-Prüfung
  #  hier muss der CBC-MAC eventuell in Software nachgebaut werden
  #  (durch CBC oder durch ECB)
  mac_validity = Smartcard.verify(message, mac, aes_keyslot)

  # falls der MAC nicht gültig ist,
  # gibt es auch keinen gültigen Plaintext
  if not mac_validity:
    plaintext = None

  return plaintext, mac_validity


Abstraction.unwrap_and_verify(knx_telegram, wrapped_key)
  # benötigter Schlüssel wird in Smartcard installiert
  Smartcard.unwrap_key(wrapped_key, aes_keyslot)

  # erstelle den Counter nach KNX-Spezifikation
  counter = build_counter(knx_telegram)

  # Entschlüsselung des MAC im Counter-Modus
  #  abhängig von der Smartcard-Funktionalität muss hier eventuell
  #  der Counter-Modus in Software nachgebaut werden (durch ECB)
  #  den tatsächlichen KNX-MAC bilden nur die 32 MSB
  #  (hier nicht umgesetzt)
  mac = Smartcard.decrypt(knx_telegram.mac, counter, aes_keyslot)

  # erstelle die Message nach KNX-Spezifikation
  message = build_message(knx_telegram)

  # CBC-MAC-Prüfung
  #  hier muss der CBC-MAC eventuell in Software nachgebaut werden
  #  (durch CBC oder durch ECB)
  mac_validity = Smartcard.verify(message, mac, aes_keyslot)

  return mac_validity


Abstraction.terminate()
  # Nutzerauthentisierung
  Smartcard.authenticate('Admin')
```

```
# Wrapping-Schlüssel in der Smartcard löschen
Smartcard.remove_wrapping_key()

# zuletzt verwendeten Laufzeitschlüssel löschen
Smartcard.remove_key(aes_keyslot)
```

7 Smartcards als Backend für den Schlüsselspeicher

In diesem Kapitel soll untersucht werden, inwieweit Smartcards mit einem festen Befehlssatz geeignet sind, als Backend für den Schlüsselspeicher zu dienen. Dabei wird einerseits auf die ISO 7816-Kommandos eingegangen und andererseits werden einige am Markt erhältliche Karten betrachtet.

7.1 Anforderungen an die Smartcard

Folgende Anforderungen ergeben sich an eine Smartcard, falls diese im Kontext des KNX-Schlüsselspeichers genutzt werden soll, so wie es in den letzten zwei Kapiteln spezifiziert wurde.

7.1.1 Format und Schnittstelle

Es soll eine kontaktbehaftete Smartcard eingesetzt werden, da eine kontaktlose Karte weitere Sicherheitsprobleme schaffen könnte und der zusätzliche Komfort außerdem nicht benötigt wird. Ein im Kontext des Raspberry Pi passender Formfaktor wäre ein USB-Token, aber auch andere Formate wie Mini-SIM oder auch das Scheckkarten-Format mit entsprechendem USB-Smartcard-Leser sind akzeptabel. Das Token oder der Leser sollten vom freien CCID-Treiber [26] unterstützt werden, damit sie unter GNU/Linux und somit auf dem Raspberry Pi lauffähig sind.

Als Software-Schnittstelle soll PC/SC für den Datenaustausch mit der Smartcard unter Verwendung von APDUs sorgen. Die Kommando- und Antwort-APDUs der jeweiligen Karte inklusive aller Parameter sollten dabei vollständig dokumentiert sein. Eine extended APDU-Unterstützung der Karte ist nicht zwingend erforderlich, wäre aber wünschenswert, da erweiterte KNX-Frames maximaler Länge (263 Byte) sonst nicht auf einmal übergeben werden können.

Alternativ ist auch die Verwendung von PKCS#11 oder einer anderen High-Level API für den Zugriff auf die Smartcard möglich. Allerdings sollte die zugehörige Programmbibliothek auf dem Raspberry Pi laufen und auch ausreichend dokumentiert sein. Viele Hersteller von Smartcards bieten zwar PKCS#11-Module an, diese sind aber oft proprietär und nur unter Windows lauffähig. Das Open-Source-Programmpaket OpenSC [27] bietet Bibliotheken, Werkzeuge und PKCS#11-Module für den Zugriff auf verschiedene Smartcards.

7.1.2 Kryptographische Algorithmen

Die Smartcard muss natürlich AES-128 beherrschen. Optimal wäre der CCM-Betriebsmodus, also Counter-Modus und CBC-MAC. Es ist aber auch möglich diese Modi durch ECB und CBC nachzubilden.

Fürs Schlüssel-Wrapping wäre AES-256 wünschenswert. Außerdem werden ECDSA für die gegenseitige Authentisierung und ECDHE für die Schlüsselaushandlung benötigt. Diese können aber auch durch andere, adäquate Algorithmen ersetzt werden.

7.1.3 Schlüssel-Management

Es muss möglich sein, mindestens einen AES-128-Schlüssel auf der Smartcard so abzulegen, dass dieser für die kryptographischen Operationen nutzbar ist, er aber von außen nicht gelesen oder unverschlüsselt exportiert werden kann. Für das Schlüssel-Wrapping ist es notwendig, dass der AES-128-Schlüssel zunächst im Plaintext in die Karte importiert werden kann.

Für das geplante Szenario ist eine Unterstützung des Schlüssel-Wrappings essenziell. Dazu muss ein Wrapping-Schlüssel innerhalb der Karte generiert werden können, welcher dauerhaft geschützt auf dieser verbleibt. Anschließend kann ein auf der Karte befindlicher AES-128-Schlüssel mit dem Wrapping-Schlüssel verschlüsselt und dann exportiert werden. Beim Unwrapping dagegen wird der verschlüsselte AES-128-Schlüssel wieder in die Karte importiert, dort entschlüsselt und abgelegt, ohne dass er nach außen gereicht wird.

Für die gegenseitige Authentisierung durch asymmetrische Kryptographie muss die Smartcard in der Lage sein, den dafür nötigen Signaturschlüssel intern zu generieren und den öffentlichen Teil davon zu exportieren. Der öffentliche Schlüssel der Gegenseite muss wiederum in die Karte importiert werden können.

7.1.4 Authentifizierung und Secure Messaging

Die Smartcard sollte verschiedene Rollen und davon abhängige Zugriffsrechte auf Kommandos und Files der Karte unterstützen. Das wird insbesondere für die Absicherung der Erzeugung und Löschung des Wrapping-Schlüssels benötigt. Die entsprechende Authentisierung sollte durch eine PIN erfolgen.

Für die Betriebsphase wird eine gegenseitige Authentisierung zwischen der Smartcard und der Raspberry-Software durch asymmetrische kryptographische Algorithmen benötigt. Dieser Prozess sollte auch eine Schlüsselaushandlung für das Secure Messaging beinhalten.

Die Karte sollte folglich Secure Messaging beherrschen, damit die Laufzeitkommunikation in der operativen Phase abgesichert werden kann.

7.2 ISO 7816-Kommandos

Der ISO/IEC 7816 ist ein mehrteiliger Standard, der wesentliche Eigenschaften von Chipkarten spezifiziert. Darin werden unter anderem auch einheitliche Chipkarten-Kommandos definiert. Ein Großteil der Kommandos wird dabei im ISO 7816-4 [28] festgelegt. Zusätzlich werden noch Kommandos für kryptographische Funktionen im ISO 7816-8 [29] und Kommandos zur Dateiverwaltung im ISO 7816-9 [30] beschrieben. Es soll hier untersucht werden, inwieweit diese Kommandos den geplanten Einsatzzweck abdecken. Selbst wenn eine Smartcard ISO 7816-konform ist, bedeutet das aber nicht, dass sie alle hier vorgestellten Kommandos mit den gewünschten Parametern vollständig unterstützt.

7.2.1 Kryptographische Operationen

Entschlüsselung und MAC-Verifizierung stellen für die Standard-Kommandos kein Problem dar. *PERFORM SECURITY OPERATION* führt mit den Optionen *DECIPHER* und *VERIFY CRYPTOGRAPHIC CHECKSUM* entsprechende kryptographische Funktionen aus. Vor dem Ausführen dieser und auch sämtlicher anderer kryptographischer Operationen, wie dem Secure Messaging, können die jeweiligen Schlüssel und weiteren Parameter mit *MANAGE SECURITY ENVIRONMENT* festgelegt werden.

7.2.2 Schlüssel-Wrapping

Das Schlüssel-Wrapping wird durch die ISO-Kommandos leider nicht abgedeckt. Es existiert kein Kommando, um den symmetrischen Wrapping-Schlüssel direkt in der Smartcard zu generieren. Es gibt zwar *GENERATE ASYMMETRIC KEY PAIR* für asymmetrische Schlüssel, aber ein symmetrisches Äquivalent wird nicht definiert. Der Plaintext-Import des Laufzeitschlüssels vor dem Wrappen ließe sich eventuell, je nach interner Kartenstruktur und Zugriffsrechten, mit entsprechenden Schreib-Kommandos wie beispielsweise *WRITE BINARY* (transparente Datenstruktur), *UPDATE RECORD* (lineare oder zyklische Recordstruktur) oder *PUT DATA* (Datenobjekte) umsetzten. Allerdings fehlen Kommandos für das Wrapping und Unwrapping der Schlüssel, was den gesamten Vorgang somit nicht umsetzbar macht.

7.2.3 Authentifizierung

Benutzer können sich mit dem Kommando *VERIFY* durch eine PIN gegenüber der Karte authentisieren.

Das Kommando *GENERATE ASYMMETRIC KEY PAIR* generiert ein asymmetrisches Schlüsselpaar in der Smartcard, welches für die Authentisierung der Karte gegenüber der Raspberry-Software genutzt werden kann. Die Antwort-APDU zu diesem Kommando kann den zugehörigen öffentlichen Schlüssel beinhalten. Der Import des externen öffentlichen Authentifizierungsschlüssels in die Karte ist mit dem Kommando *MANAGE SECURITY ENVIRONMENT* und dessen Option *STORE PUBLIC KEY* möglich. Diese Option wird zwar im ISO 7816-4 nicht explizit erwähnt, allerdings wird sie von einigen Karten unterstützt. Alternativ könnte, wie oben beschrieben, der Schlüsselimport auch mit einem Schreib-Kommando wie *WRITE RECORD* oder *PUT DATA* erfolgen, falls das bei der genutzten Karte möglich ist.

Der eigentliche Authentifizierungsvorgang zwischen der Smartcard und der Raspberry-Software wird über die Kommandos *GET CHALLENGE* (Karte erzeugt eine Zufallszahl), *INTERNAL AUTHENTICATE* (Authentisierung der Karte) und *EXTERNAL AUTHENTICATE* (Authentisierung der externen Instanz) realisiert. In diesem Prozess können, je nach Kartenfunktionalität, auch die Schlüsselaushandlung und anschließende Schlüsselableitung für das Secure Messaging integriert sein, in dem sie über entsprechende Kommandoparameter gesteuert werden. Eine Alternative zu *INTERNAL AUTHENTICATE* und *EXTERNAL AUTHENTICATE* ist dessen Zusammenfassung in einem Kommando *MUTUAL AUTHENTICATE* für die gegenseitige Authentisierung.

7.2.4 Terminierung

Für das Löschen des Wrapping-Schlüssels bei der Terminierung des Schlüsselspeichers gibt es kein explizites Kommando. Falls es ein Kommando für die Generierung des Schlüssel geben würde, könnte man dieses nochmal aufrufen und so den alten Schlüssel einfach überschreiben. Auch hier gilt, dass je nach interner Kartenstruktur und Zugriffsrechten das File oder Datenobjekt welches den Schlüssel enthält eventuell mit den entsprechenden Schreib-Kommandos überschrieben werden kann. Eine weitere Möglichkeit wären hier auch Löschkommandos wie *ERASE BINARY*, *ERASE RECORD* oder wenn verfügbar sogar *DELETE FILE*. Es ist auch denkbar mit *TERMINATE EF* das File oder mit *TERMINATE CARD USAGE* sogar die gesamte Karte in den Terminierungszustand zu versetzen. Diese Kommandos sind aber nicht umkehrbar und führen damit voraussichtlich zur Unbrauchbarkeit der Smartcard für weitere Initialisierungen des Schlüsselspeichers.

7.3 Smartcardmodelle mit festem Befehlssatz

Nachfolgend sollen einige konkrete Smartcardmodelle mit festem Befehlssatz kurz vorgestellt und ihre Eignung für den Schlüsselspeicher geprüft werden. Diese Smartcards bilden nur eine kleine Auswahl derer, welche im Kontext dieser Arbeit recherchiert wurden.

7.3.1 CryptoMate Nano

Der CryptoMate Nano (Abbildung 12) von Advanced Card Systems [31] ist ein ISO 7816-konformes kryptographisches USB-Token. Er kostet 18,00 EUR und besitzt 64 kB EEPROM. Der CryptoMate beherrscht DES, Triple DES und auch AES-128/192/256 im ECB- und CBC-Betriebsmodus. Daneben unterstützt er RSA mit bis zu 4096 Bit Schlüssellänge, aber leider keine Elliptische-Kurven-Kryptographie. Die gegenseitige Authentisierung zwischen Token und externer Instanz erfolgt mit Triple DES unter Aushandlung eines Session-Schlüssels für das anschließende Secure Messaging. Der CryptoMate Nano wird vom CCID-Treiber unterstützt und läuft deshalb auch unter GNU/Linux.

Abbildung 12: CryptoMate Nano USB-Token (Quelle: [31])

Schlüssel-Wrapping wird scheinbar nicht angeboten, da es in der Funktionalitätenliste nicht auftaucht. Leider sind dieses und auch andere Merkmale nicht genauer nachprüfbar, da es trotz mehrmaligem Anschreiben des Herstellers nicht möglich war, an die Referenzanleitung des CryptoMate Nano zu kommen. Weiterhin ist ein CryptoMate Nano Client Kit für 59,00 EUR erhältlich, welches ein PKCS#11-Modul enthält, das aber nur für Windows verfügbar ist. OpenSC-Unterstützung für den CryptoMate Nano gibt es nicht.

7.3.2 Nitrokey Pro 2

Der Nitrokey Pro 2 (Abbildung 13) ist ein USB-Token, dessen Hardware und auch Software Open Source sind [32]. Er kostet 49,00 EUR und bietet vielfältige Funktionen wie sicheres Login durch Einmalpasswörter oder E-Mail-Verschlüsselung mittels OpenPGP oder S/MIME. Der Nitrokey enthält eine manipulationssichere ISO 7816-konforme Smartcard, welche auf der OpenPGP Card in Version 3.3 [33] basiert. Somit steht mit der OpenPGP Card-Spezifikation eine ausführliche APDU-Dokumentation zu Verfügung. Auch der Nitrokey Pro 2 wird vom CCID-Treiber unterstützt und ist deshalb unter GNU/Linux auf dem Raspberry Pi nutzbar. Es existiert auch entsprechende OpenSC-Unterstützung über das PKCS#11-Modul für die OpenPGP Card, wobei OpenSC hier aber noch nicht mit AES umgehen kann.

Abbildung 13: Nitrokey Pro 2 USB-Token (Quelle: [32])

An asymmetrischen kryptographischen Algorithmen werden RSA mit Schlüsseln bis 4096 Bit sowie ECDSA und ECDH mit NIST- und Brainpool-Kurven und Schlüssellängen bis 521 Bit unterstützt. Es können entweder drei RSA-Schlüsselpaare oder drei ECC-Schlüsselpaare im Token gespeichert werden, welche vorher intern generiert oder importiert wurden. Der Nitrokey unterstützt auch AES-128 und AES-256 nach der OpenPGP Card-Spezifikation. Das bedeutet, dass genau ein AES-Schlüssel unter Angabe des Admin-Passwortes in das entsprechende Data Object geschrieben werden kann und anschließend AES-Operationen im CBC-Modus ausgeführt werden können. Ein Schlüsselwrapping ist in der OpenPGP Card-Spezifikation nicht vorgesehen und wird somit von der Karte auch nicht implementiert. Eine Authentisierung der Karte gegenüber einer externen Instanz ist mit einem Challenge-Response-Verfahren mit RSA oder ECDSA möglich. Das Secure Messaging erfolgt mit AES und vorher festgelegten statischen Schlüsseln.

7.3.3 TCOS-Smartcard

Beim TeleSec Chipcard Operating System (TCOS) [34] handelt es sich um ein ISO 7816-konformes Chipkartenbetriebssystem der T-Systems International GmbH [35]. Das vom Bundesamt für Sicherheit in der Informationstechnik (BSI) zertifizierte Kartenbetriebssystem liegt aktuell in der Version 3.0 vor. TCOS kommt beispielsweise beim neuen deutschen Personalausweis und beim deutschen elektronischen Reisepass zum Einsatz. TCOS-Chipkarten richten sich eher an Großkunden als an Privatanwender, was auch der Verkauf von Paketen mit mindestens dreistelligen Stückzahlen zeigt. Außerdem bietet T-Systems auch die Möglichkeit einer kundenindividuellen Anpassung der Smartcards.

Kontaktbehaftete TCOS-Chipkarten sind sowohl im Scheckkarten-Format als auch im Mini-SIM-Format erhältlich. Somit ist für die Kartenanbindung über PC/SC unter GNU/Linux ein kompatibler Kartenleser erforderlich. Das TCOS-Referenzhandbuch mit der vollständigen APDU-Dokumentation ist nicht öffentlich verfügbar, aber für Systementwickler zugänglich. T-Systems stellt proprietäre PKCS#11-Module für GNU/Linux zur Verfügung, welche allerdings nicht auf dem Raspberry Pi laufen. Es ist eine eingeschränkte OpenSC-Unterstützung verfügbar, wobei teilweise kein Schreibzugriff auf die Karte möglich ist und vom Secure Messaging abhängige Funktionalitäten nicht genutzt werden können.

Je nach Variante kann eine TCOS-Smartcard beispielsweise über 80 kB EEPROM verfügen. Laut TCOS-Leistungsbeschreibung [36] werden an symmetrischen Algorithmen DES, Triple DES und IDEA unterstützt. AES taucht in der Liste nicht auf, wobei eventuell neuere Kartenvarianten diesen beherrschen, was aber mangels aktueller Referenzanleitung nicht nachgeprüft werden konnte. Genauso scheint auch das Schlüssel-Wrapping nicht verfügbar zu sein. Was asymmetrische Algorithmen anbelangt, so gibt es hier RSA und je nach Kartentyp auch ECC. Die beidseitige Authentisierung mit Schlüsselaushandlung ist mit symmetrischen und asymmetrischen Algorithmen möglich, genauso wie auch das Secure Messaging.

7.3.4 SmartCard-HSM

Das SmartCard-HSM [37] der CardContact Systems GmbH ist ein leichtgewichtiges Hardware-Sicherheitsmodul auf Chipkarten-Basis, welches der sicheren Erzeugung, Speicherung und Nutzung von Schlüsseln dient. Dem ISO 7816-konformen SmartCard-HSM liegt ein Java Card-Applet für die NXP JCOP-Plattform (siehe nächstes Kapitel) zugrunde. Das aktuelle Modell SmartCard-HSM 4K ist im Chipkarten-Format (mit und ohne Dual-Interface) und als USB-Token (Abbildung 14) erhältlich. Die Preise bewegen sich dabei zwischen 34 und 59 EUR. Das USB-Token wird vom CCID-Treiber unterstützt und läuft somit auch unter GNU/Linux. Für die Variante im Karten-Format ist ein Kartenleser

mit Extended Length APDU-Unterstützung erforderlich. Die Kommando-Dokumentation ist für registrierte Entwickler im CardContact Developer Network erhältlich, was es möglich macht, das SmartCard-HSM auf APDU-Basis anzusprechen. Zusätzlich steht noch eine PKCS#11-Schnittstelle direkt über die OpenSC-Middleware zur Verfügung.

Abbildung 14: SmartCard-HSM 4K USB-Token (Quelle: [37])

Das SmartCard-HSM kann je nach Schlüsseltyp bis zu 300 Schlüsselpaare gleichzeitig speichern. Dabei werden RSA-Schlüssel mit bis zu 4096 Bit und ECC-Schlüssel mit bis zu 521 Bit Länge sowie die dazugehörigen kryptographischen Algorithmen unterstützt. Des Weiteren lassen sich auch AES-Schlüssel mit 128, 192 und 256 Bit Länge erzeugen und speichern. Das SmartCard-HSM hat als eine von wenigen Smartcards Schlüssel-Wrapping-Funktionalität. Dabei können auf der Karte befindliche Schlüssel mit einem AES-256 Device Key Encryption Key (DKEK) intern verschlüsselt und dann ausgelagert werden. Diese Schlüssel können anschließend wieder in das SmartCard-HSM importiert werden. Zusätzlich zur Verschlüsselung wird dabei noch ein MAC über den jeweiligen Schlüssel und dessen Metadaten berechnet. Leider wird das Schlüssel-Wrapping nur für in der Karte erzeugte Schlüssel angeboten, da das SmartCard-HSM aus Sicherheitsgründen keinen Plaintext-Schlüssel-Import besitzt. Die beidseitige Authentisierung und Schlüsselaushandlung erfolgen durch asymmetrische Algorithmen. Dabei werden symmetrische Schlüssel für das Secure Messaging etabliert.

7.3.5 Zusammenfassung

Wie sich gezeigt hat, war es nicht möglich eine passende Smartcard mit festen Befehlssatz zu finden, welche unsere Anforderungen vollständig erfüllt. Oft scheitert es daran, dass eine Kommando-Referenzdokumentation nicht öffentlich zugänglich ist, was es sehr schwierig gestaltet, die Karte über die korrekten APDUs anzusprechen. Das gilt insbesondere

deshalb, da sich viele Smartcards nicht komplett an die ISO 7816-Kommandos halten. Die PKCS#11-Schnittstelle ist hier auch keine Lösung, da die angebotenen Module häufig proprietär sind und nur für Windows oder GNU/Linux auf der Intel-Architektur vorliegen, was eine Verwendung auf dem Raspberry Pi unmöglich macht. Des Weiteren kommt hinzu, dass die AES-Unterstützung in OpenSC faktisch nicht vorhanden ist.

Asymmetrische Kryptoverfahren wie RSA und ECC scheinen bei den Kartenherstellern mehr im Vordergrund zu stehen. Diese werden oft zum Signieren, Verschlüsseln und Authentifizieren mit einer Identität zugeordneten Schlüsseln genutzt. Oftmals sind zu AES, den unterstützten Betriebsarten und deren Nutzung keine weitergehenden Informationen zu finden. Das größte Problem stellt allerdings das Schlüssel-Wrapping dar, welches, bis auf wenige Ausnahmen, als Funktionalität nicht vorhanden ist. So ist es beispielsweise häufig nicht möglich, auf der Smartcard einen symmetrischen Wrapping-Schlüssel zu generieren. Zudem kann ein verschlüsselter Schlüssel nicht in die Karte importiert und dort entschlüsselt werden, ohne den Schlüssel dabei im Klartext nach außen zu geben. Selbst wenn das Schlüssel-Wrapping unterstützt wird, so gilt das nur für intern erzeugte Schlüssel und ein Import von Plaintext-Schlüsseln wird in diesem Fall nicht angeboten. Das ist zwar aus Gründen der Schlüsselqualität (interner starker Zufallszahlengenerator) und Vertraulichkeit sinnvoll, macht aber die jeweiligen Smartcards für den geplanten Einsatzzweck ungeeignet. Aus diesen Gründen ist es für die Realisierung des KNX-Schlüsselspeichers notwendig, auf frei programmierbare Chipkarten zurückzugreifen, welche im nächsten Kapitel am Beispiel der Java Card vorgestellt werden.

8 Java Card

Wie sich im letzten Kapitel gezeigt hat, ist aufgrund der speziellen Anforderungen an die Smartcard eine frei programmierbare Chipkarte für den KNX-Schlüsselspeicher notwendig. Eine weit verbreitete Multiapplikations-Chipkarte ist dabei die Java Card, welche im Folgenden vorgestellt werden soll.

8.1 Einführung in Java Card

Bei Java Card [14] [38] handelt es sich um eine Technologie, welche es erlaubt Java Applets auf Chipkarten mit entsprechender Virtual Machine auszuführen. Seit Einführung wurden weltweit bereits mehrere Milliarden Java Cards in Umlauf gebracht, wobei sich viele davon in SIM-Karten oder Kreditkarten befinden. Die Java Card Platform Specification und das zugehörige Java Card Development Kit werden von Oracle [39] weiterentwickelt und veröffentlicht, wobei momentan Version 3.1 aktuell ist. Genauer betrachtet, handelt es sich bei Java Card um eine Untermenge der Programmiersprache Java. So fehlen etwa bestimmte Datentypen und auch die Java Klassenbibliothek ist stark reduziert.

Java Card Applets sind von der konkreten Chipkarten-Hardware unabhängig, was es einem Applet erlaubt auf verschiedenen Smartcards zu laufen. Diese Portabilität kann aber teilweise durch die Speichergröße, Performance oder Kryptoalgorithmenunterstützung der jeweiligen Smartcard eingeschränkt sein. Auf einer Karte können gleichzeitig mehrere Applets installiert sein, wobei diese durch eine Applet Firewall voneinander getrennt sind. Das gerade aktive Applet erhält die eingehenden Kommando-APDUs, arbeitet diese ab und erzeugt so entsprechende Antwort-APDUs. Java Card unterstützt sowohl symmetrische (z. B. DES, Triple DES, AES) wie auch asymmetrische (z. B. RSA, ECC) kryptographische Algorithmen. Besonders hervorzuheben ist hier die Unterstützung von AES im CCM-Modus.

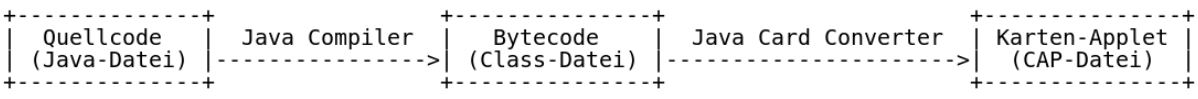

Abbildung 15: Ablauf bei der Java Card Appletentwicklung

Abbildung 15 zeigt den Ablauf bei der Entwicklung von Java Card Applets. Zunächst muss der entsprechende Quellcode mit dem Java Compiler in maschinenunabhängigen Bytecode kompiliert werden. Danach wird aus diesem mit dem Java Card Converter eine CAP-Datei (Card Application File) erzeugt. Die CAP-Datei kann bei Bedarf noch signiert werden und ist anschließend bereit um in die Chipkarte geladen zu werden.

8.2 GlobalPlatform Card Specification

Die GlobalPlatform Card Specification [14] ist ein internationaler Standard für das Anwendungsmanagement auf Multiapplikations-Chipkarten. Die Spezifikation ist frei zugänglich und liegt aktuell in Version 2.3.1 [40] vor. Obwohl die GP-Spezifikation unabhängig von einem bestimmten Betriebssystem ist, hat sie sich jedoch in der Praxis vor allem bei Java Card als De-facto-Standard zur Verwaltung von Java-basierten Anwendungen etabliert.

Die zentrale Komponente einer GlobalPlatform-Karte ist der Card Manager. Er stellt Anwendungen interne Schnittstellen für seine Verwaltungsdienste durch die GlobalPlatform API und externe durch APDU-Kommandos zur Verfügung. Anwendungen können über in der GP-Spezifikation festgelegte Kommandos in die Chipkarte geladen, installiert und wieder entfernt werden. Diese Vorgänge werden durch kryptographische Verfahren abgesichert. Nach der Installation muss ein Applet über einen entsprechenden Application Identifier (AID) selektiert werden, bevor es verwendet werden kann.

Um ein Applet auf eine Chipkarte zu laden, werden entsprechende GlobalPlatform-konforme Software-Werkzeuge benötigt. Kartenhersteller stellen diese kostenpflichtig zur Verfügung. Es existieren aber auch eine Reihe von Open Source Lösungen, wovon zwei hier kurz vorgestellt werden. Die GPShell [41] ist ein Kommandozeilenwerkzeug für die Applet-Verwaltung auf einer GP-Chipkarte. Es setzt auf einer zum Projekt gehörenden C-Bibliothek auf und gilt als Quasistandard im Bereich Open Source GlobalPlatform Implementation. Allerdings stammt das letzte Release von 2010, weshalb neuere GlobalPlatform Versionen ab 2.2 nicht vollständig unterstützt werden. Eine Alternative für das Applet-Management stellt das auf Einfachheit und Benutzerfreundlichkeit ausgelegte GlobalPlatformPro [42] dar. Es ist vollständig in Java geschrieben und bietet sowohl ein Kommandozeilenwerkzeug als auch eine Java API für die Integration in eigene Projekte an. GlobalPlatformPro wird regelmäßig weiterentwickelt und unterstützt auch neuere Versionen der GlobalPlatform Spezifikation.

8.3 Auswahl von Java Cards

Bei der Auswahl konkreter Java Card-Modelle muss einiges beachtet werden [43]. Zunächst sollte es sich bei der genutzten Java Card um eine offene generische Blanko-Karte handeln, die das Laden und Installieren von eigenen Applets erlaubt. Die Java Card Version sollte mindestens 3.0.1 betragen, damit neuere kryptographische Verfahren unterstützt werden. Hierbei ist zu beachten, dass Anbieter von Java Cards manchmal nur einen Teil der spezifizierten Kryptoalgorithmen implementieren. Andersherum kann es auch sein, dass bestimmte Karten proprietäre API-Erweiterungen bereitstellen, für die dann ein herstellerspezifisches und kostenpflichtiges SDK notwendig ist. Wenn man sich hier aber an die Funktionalitäten der Java Card Platform Specification hält, dann sollte das Oracle

Development Kit ausreichen. Bei der Version der GlobalPlatform Card Specification sind 2.1.1 und 2.2.1 üblich. Ein sehr wichtiger Punkt ist auch die Kenntnis des Secure Channel-Schlüssels für das Laden der Applikation auf die Karte. Dieser sollte dem GlobalPlatform Default-Schlüssel (`404142434445464748494A4B4C4D4E4F`) entsprechen oder anderweitig bekannt sein, da es sonst nicht möglich ist, die Applets auf die Karte zu bringen. Auch sollte bei Verwendung von Schlüsseldiversifizierung das genaue Verfahren (z. B. EMV oder VISA2) bekannt sein. Die Größe des EEPROM der Chipkarte richtet sich nach den jeweiligen Anforderungen der Applikation. So gibt es beispielsweise Java Cards mit 64, 72, 128 oder 144 kB EEPROM.

Java Cards gibt es wie auch andere Smartcards im Kreditkarten- oder SIM-Kartenformat, aber auch als USB-Token. Die Preise können sich je nach Formfaktor, Interface und Version zwischen 10 und 60 EUR bewegen. Prominente Vertreter sind hier beispielsweise die JCOP-Karten von NXP Semiconductors. Die Java Card OpenPlatform (JCOP) [44] ist ein ursprünglich von IBM entwickeltes Smartcard-Betriebssystem für die Java Card-Plattform. Daneben ist JCOP auch GlobalPlatform-konform. Obwohl es inzwischen Version 3.1 von JCOP gibt, sind nach wie vor viele Karten mit 2.4.1 oder 2.4.2 erhältlich. Eine Alternative zu den JCOP-Karten stellt die SmartCafe Expert Serie von Giesecke + Devrient dar. Auch hier gibt es Modelle mit unterschiedlichen Formfaktoren sowie verschiedenen Java Card- und GlobalPlatform-Versionen. Weitere Hersteller von Java Cards sind Gemalto, Idemia und Infineon.

8.4 Java Card Applet für den KNX-Schlüsselspeicher

Nachfolgend soll die Kommando-Schnittstelle eines entsprechenden Java Card Applets für die Nutzung im KNX-Schlüsselspeicher konzipiert werden. Neben den in Abschnitt 7.2 vorgestellten ISO-Kommandos, werden weitere nicht im ISO 7816-Standard spezifizierte Kommandos benötigt. Diese sollen über herstellerspezifische ISO 7816-strukturierte Kommandos umgesetzt werden. Ob ein Kommando herstellerspezifisch ist, kann durch ein entsprechendes Bit im Class-Byte der jeweiligen APDU gekennzeichnet werden. Bei einer Implementation des Applets müssten die ISO 7816-konformen Kommandos genauso wie die herstellerspezifischen umgesetzt werden, da es hier keine vorgefertigten Klassen für die Standard-Kommandos in Java Card gibt.

8.4.1 Kryptographische Operationen

Wie bereits beschrieben, können die Entschlüsselung und MAC-Verifizierung über die Standard-ISO-Kommandos *PERFORM SECURITY OPERATION* und *MANAGE SECURITY ENVIRONMENT* umgesetzt werden. Ein Vorteil ist, dass die Java Card

Plattform AES im CCM-Modus unterstützt. Somit muss dieser nicht durch andere Betriebsarten nachgebildet werden.

8.4.2 Schlüssel-Wrapping

Für das Schlüssel-Wrapping müssen mehrere eigene Kommandos definiert werden. Der symmetrische Wrapping-Schlüssel kann intern unter Angabe der entsprechenden Parameter mit *GENERATE SYMMETRIC KEY* erzeugt werden. Dafür wird der Zufallszahlengenerator der Karte genutzt. Der Zugriff auf dieses und auch auf alle weiteren in der Initialisierungs- und Terminierungsphase benötigten Kommandos sollte erst nach Eingabe der Admin-PIN möglich sein. Für den Plaintext-Import der Laufzeitschlüssel vor dem Wrappen wird *IMPORT KEY* definiert. Dieses Kommando legt den übergebenen Schlüssel unter Angabe einer Key-ID in der Karte ab. Das Wrappen und Unwrappen der Schlüssel erfolgt über *WRAP KEY* und *UNWRAP KEY*. Als Parameter wird diesen Kommandos unter anderem die Key-ID des jeweiligen Laufzeitschlüssels übergeben.

8.4.3 Authentifizierung

Die PIN-Authentisierung des Benutzers erfolgt über das Standard-ISO-Kommando *VERIFY*. Die Erzeugung des Schlüsselpaares für die Authentisierung der Karte gegenüber der Raspberry-Software und der Export des zugehörigen öffentlichen Schlüssels werden wie beschrieben über *GENERATE ASYMMETRIC KEY PAIR* realisiert. Der Import des externen öffentlichen Authentifizierungsschlüssels in die Karte kann über das im letzten Abschnitt definierte *IMPORT KEY* mit entsprechender Parametrisierung erfolgen. Der Authentifizierungsvorgang mit Schlüsselaushandlung zwischen der Smartcard und der Raspberry-Software durch ein Challenge-Response-Verfahren kann durch die beschriebenen ISO 7816-Kommandos *GET CHALLENGE*, *INTERNAL AUTHENTICATE* und *EXTERNAL AUTHENTICATE* bzw. *MUTUAL AUTHENTICATE* umgesetzt werden.

8.4.4 Terminierung

Für das Löschen des Wrapping-Schlüssels und der in der Karte befindlichen Laufzeitschlüssel wird das Kommando *DELETE KEY* definiert. Es löscht unter Angabe einer Key-ID vorher erzeugte oder importierte Schlüssel unwiederbringlich von der Karte.

9 Cryptographic Service Provider

Der Cryptographic Service Provider (CSP) [45] ist eine universelle Komponente, die kryptographische Dienste für den Schutz der Vertraulichkeit und Integrität von Daten und für die Authentisierung von Instanzen bereitstellt. Ein CSP besteht aus Hardware, Firmware und Software und bietet eine definierte externe Schnittstelle für Applikationen. Er kann beispielsweise als Smartcard oder HSM realisiert sein, aber auch weitere Ausprägungen sind vorstellbar. Anbieter von CSPs können diese nach Common Criteria Schutzprofil PP-CSP [45] vom Bundesamt für Sicherheit in der Informationstechnik zertifizieren lassen. Ein Cryptographic Service Provider ist generisch genug konzipiert, damit er in verschiedenen Anwendungsszenarien zum Einsatz kommen kann, in denen sich bestimmte Sicherheitsanforderungen anwendungsübergreifend wiederholen und nicht nur allein in Software umgesetzt werden können. Eine erste Anwendung bilden Registrierkassen [46], in denen der CSP als Teil eines Sicherheitsmoduls der Umsetzung der Kassensicherungsverordnung dient.

9.1 CSP als Backend für den Schlüsselspeicher

Nachfolgend soll geprüft werden, ob der Cryptographic Service Provider geeignet ist, als Backend für den Schlüsselspeicher zu dienen. Damit würde er eine weniger aufwändige Alternative zur Java Card darstellen. Dazu werden ausgewählte Sicherheitsanforderungen aus dem CSP-Schutzprofil mit den Backend-Anforderungen des Schlüsselspeichers abgeglichen, um zu ermitteln, inwieweit der Schlüsselspeicher mit einem CSP im Hintergrund umsetzbar wäre.

9.1.1 Kryptographische Operationen

Der CSP ermöglicht die AES-128-Entschlüsselung im Counter-Modus (*FCS_COP.1/ED Cryptographic operation Data encryption and decryption*). Bei der MAC-Verifizierung werden laut Schutzprofil nur CMAC und GMAC angeboten, aber kein CBC-MAC (*FCS_COP.1/MAC Cryptographic operation MAC using AES*). Allerdings wird im Kontext der hybriden Entschlüsselung und MAC-Verifizierung in Kombination mit einem asymmetrischen Algorithmus der CCM-Modus explizit genannt (*FCS_COP.1/HDM Cryptographic operation Hybrid data decryption and MAC verification*), was diesen somit prinzipiell möglich macht. Ansonsten könnte, wie auch schon bei den Smartcards, der CBC-MAC durch andere Modi nachgebildet werden.

9.1.2 Schlüssel-Wrapping

Der CSP unterstützt das Schlüssel-Wrapping und Unwrapping mit AES-256 (*FCS_COP.1/KW Cryptographic operation Key wrap*, *FCS_COP.1/KU Cryptographic operation Key unwrap*). Der dazu nötige Wrapping-Schlüssel kann problemlos erzeugt werden (*FCS_CKM.1/AES Cryptographic key generation AES key*). Auch der Import der Laufzeitschlüssel scheint möglich zu sein (*FPT_ISA.1/CK Import of TSF data with security attributes Cryptographic keys*), wobei hier nicht ganz klar ist, ob sich das auch auf Klartext-Schlüssel bezieht.

9.1.3 Authentifizierung

Der CSP bietet verschiedene Authentisierungsmechanismen, wie etwa Authentisierung durch Passwort, PACE, Terminal Authentication oder Chip Authentication (*FIA_UAU.5 Multiple authentication mechanisms* , *FIA_API.1/PACE Authentication Proof of Identity PACE authentication to Application component*, *FIA_API.1/CA Authentication Proof of Identity Chip authentication to user*). Auch die Schlüsselaushandlung für die Etablierung eines sicheren Kanals ist dabei möglich (*FCS_CKM.1/PACE Cryptographic key generation Key agreement for trusted channel PACE*, *FCS_CKM.1/TCAP Cryptographic key generation Key agreement by Terminal and Chip authentication protocols*).

9.1.4 Terminierung

Der CSP gestattet das Löschen des Wrapping-Schlüssels und der im CSP befindlichen Laufzeitschlüssel (*FCS_CKM.4 Cryptographic key destruction*).

9.1.5 Sonstiges

Neben den oben behandelten CSP-Sicherheitsanforderungen gibt es noch eine ganze Reihe weiterer im Schutzprofil, welche direkt oder indirekt für den geplanten Anwendungsfall relevant sind. Alle hier zu erwähnen würde den Rahmen dieser Arbeit übersteigen, trotzdem sollen einige wenige beispielhaft genannt werden: Der CSP integriert einen Zufallszahlengenerator (*FCS_RNG.1 Random number generation*), welcher etwa bei der Schlüsselerzeugung genutzt wird. Beim Schlüssel-Wrapping sollte die Vertraulichkeit und Integrität der Schlüssel gewährleistet sein (*FPT_TCT.1/CK TSF data confidentiality transfer protection Cryptographic keys*, *FPT_TIT.1/CK TSF data integrity transfer protection Cryptographic keys*). Außerdem sollte der CSP auch Schutz gegen physische Attacken bieten (*FPT_PHP.3 Resistance to physical attack*).

9.1.6 Bewertung

Auf den ersten Blick scheint eine Umsetzung des Schlüsselspeichers mit dem Cryptographic Service Provider möglich. Zwar gibt es noch ein paar kleine Minuspunkte, wie der „fehlende“ CBC-MAC oder die unklare Situation beim Import von Klartext-Schlüsseln, hier sollte aber erst mal das Erscheinen der ersten zertifizierten CSP-Produkte abgewartet werden, um dann genauere Aussagen treffen zu können. Genauso ist auch noch nicht bekannt, in welchem preislichen Rahmen sich derartige Produkte bewegen werden. Es ist aber davon auszugehen, dass sie aufgrund der Zertifizierung teurer als beispielsweise Java Cards sein werden.

10 Ausblick

Der nächste logische Schritt wäre die Implementierung eines Prototypen, der das in der vorliegenden Arbeit beschriebene Konzept des KNX-Schlüsselspeichers umsetzt. Dazu gehört einerseits die Softwarekomponente, die auf einem Raspberry Pi läuft und eine von der IDS-Software nutzbare Schnittstelle anbietet und andererseits das Java Card Applet, das die benötigte Smartcard-Funktionalität für den Schlüsselspeicher bereitstellt.

Es wäre auch interessant den implementierten Schlüsselspeicher in einer realen KNX Data Secure-Installation zu testen. Dafür müsste der Import der KNX-Laufzeitschlüssel aus dem ETS in den Schlüsselspeicher, welcher in dieser Arbeit vorausgesetzt wurde, im Detail geklärt werden. Zu diesem Zweck müsste unter anderem das genaue Format der ETS-Schlüsseldatei analysiert werden, um so an die entsprechenden Laufzeitschlüssel zu gelangen.

Ein weiterer wichtiger Punkt ist die Performanz des Schlüsselspeichers. Hier wäre es interessant, die Geschwindigkeit des Systems im Betrieb zu messen, um zu ermitteln wie viele KNX-Telegramme pro Zeiteinheit verarbeitet werden können. Schließlich müssen pro übergebenem Paket mehrere AES-Operationen (Schlüssel-Unwrapping, Paketentschlüsselung und -verifizierung, Secure Messaging) ausgeführt werden. Es ist auch zu vermuten, dass die USB-Anbindung der Smartcard eventuell einen Flaschenhals darstellen könnte. Eine mögliche Strategie zur Erhöhung der Performanz wäre, wenn das IDS nicht alle Pakete an den Schlüsselspeicher weitergibt, sondern nur bestimmte ausgewählte. Auch die Lastverteilung der IDS-Anfragen auf mehrere parallel arbeitende Schlüsselspeicher wäre denkbar. Zusätzlich könnten auch mehrere Laufzeitschlüssel gleichzeitig in der Smartcard gespeichert werden und eine zu wählende Verdrängungsstrategie (z. B. LFU, LRU) entscheidet dann darüber, welche Schlüssel beim Import überschrieben werden.

Eine mögliche Erweiterung, welche die Sicherheit und Flexibilität bei der Authentisierung zwischen der Smartcard und der Raspberry-Software erhöhen würde, wäre die Identität beider öffentlicher Authentifizierungsschlüssel mit Hilfe von Zertifikaten zu prüfen, anstatt sie aufgrund der sicheren Initialisierungsumgebung als gegeben anzunehmen. Dazu müssten das Applet und die Raspberry-Software jeweils ein CV-Zertifikat vorhalten, um diese dann in der Betriebsphase gegenseitig zu validieren.

KNX Data Secure-fähige Geräte sind nach wie vor ziemlich teuer und nicht so weit verbreitet. Deshalb könnte versucht werden, das hier vorgestellte Konzept des Schlüsselspeichers auch auf andere Szenarien im IoT-Bereich anzuwenden. Auch außerhalb dessen, wie etwa zum Schutz der privaten Signaturschlüssel für Kryptowährungen, ist eine Anwendung eines angepassten Schlüsselspeichers vorstellbar.

11 Fazit

Die vorliegende Arbeit beschäftigte sich mit einem sicheren Schlüsselspeicher im Kontext eines KNX Data Secure Intrusion Detection Systems. Dieser bietet die Möglichkeit KNX-Laufzeitschlüssel zu importieren und diese im Zusammenhang mit der Entschlüsselung und Verifizierung von Data Secure-Telegrammen zu nutzen. Es wurden zunächst Anforderungen an den Schlüsselspeicher definiert, wobei auf funktionale, finanzielle und vor allem sicherheitsrelevante Aspekte eingegangen wurde. Anschließend folgte ein Konzept zur Umsetzung dieser Anforderungen, welches es durch eine mehrschichtige Architektur ermöglicht, verschiedene kryptographische Hardwarekomponenten oder auch Cloud-Dienste als Basis für den Schlüsselspeicher zu nutzen. Im vorliegenden Fall wurde die Smartcard als mögliches Backend gewählt, da sich diese bzgl. der Anforderungen als besonders geeignet erwies. Im Mittelpunkt des vorgestellten Konzepts steht das Schlüssel-Wrapping, welches es ermöglicht, dass die präsentierte Lösung auch für eine große Anzahl von zu speichernden Schlüsseln skaliert. Aufbauend auf dem konzeptionellen Modell des Schlüsselspeichers wurde eine entsprechende Schnittstelle zur Nutzung durch ein zukünftiges Intrusion Detection System spezifiziert.

Der zweite Teil der Arbeit befasst sich mit Smartcards als Backend für den Schlüsselspeicher. Dabei wurden Anforderungen an potenzielle Chipkarten definiert und es wurde geprüft, inwieweit diese von am Markt befindlichen Modellen mit festem Befehlssatz erfüllt werden. Es stellte sich jedoch heraus, dass wohl keine der erhältlichen Karten vollständig für den geplanten Einsatzzweck geeignet ist. Oft scheitert es an der fehlenden Wrapping- und Unwrapping-Funktionalität und auch der benötigte Import von symmetrischen Klartext-Schlüsseln ist vielfach nicht vorhanden. Ein weiteres Problem ist die häufig fehlende bzw. nicht öffentlich zugängliche Dokumentation der Kartenschnittstelle, was deren Nutzung massiv erschwert. Aus diesem Grund wurde als Lösung ein individuelles Java Card Applet vorgeschlagen, welches die nötigen Anforderungen entsprechend umsetzt. Dabei nutzt das Applet soweit möglich Standard ISO 7816-Kommandos und greift, dort wo es notwendig ist, auf eigens definierte herstellerspezifische Kommandos zurück. Mit dieser vorgeschlagenen Lösung scheint eine Umsetzung für den vorliegenden Anwendungsfall prinzipiell möglich zu sein. Zusätzlich könnte bald mit dem Cryptographic Service Provider eine mögliche Alternative zur Java Card als Backend des Schlüsselspeichers bereitstehen.

12 Literatur

[1] Wikipedia-Autoren: *KNX-Standard — Wikipedia*. https://de.wikipedia.org/wiki/KNX-Standard, besucht: 2018-09-10.

[2] KNX Association: *Offizielle Webseite der KNX Association*. https://www.knx.org, besucht: 2018-09-10.

[3] Merten: *Wie funktioniert KNX*. https://www.merten.de/Wie-funktioniert-KNX.1347.0.html, besucht: 2018-09-10.

[4] KNX Association: *ETS5 Professional*. https://www.knx.org/knx-en/for-professionals/software/ets-5-professional/index.php, besucht: 2018-09-10.

[5] KNX Association: *KNX System Specifications, Communication Media Twisted Pair 1, Version 01.02.02*, 2013.

[6] KNX Association: *KNX Sicherheit — KNX Positionspapier zu Datensicherheit und Datenschutz*, 2014.

[7] KNX Association: *Application Note 158 — KNX Data Security*, 2013.

[8] Wikipedia-Autoren: *Raspberry Pi — Wikipedia*. https://de.wikipedia.org/wiki/Raspberry_Pi, besucht: 2018-09-12.

[9] Raspberry Pi Foundation: *Webseite der Raspberry Pi Foundation*. https://www.raspberrypi.org/, besucht: 2018-09-12.

[10] Busware: *Pigator*. http://busware.de/tiki-index.php?page=PIGATOR, besucht: 2018-09-12.

[11] Matthias Urlichs: *Pigator — GitHub*. https://github.com/knxd/knxd, besucht: 2018-09-12.

[12] Claudia Eckert: *IT-Sicherheit: Konzepte — Verfahren — Protokolle*. Oldenbourg, München, 9. Auflage, 2014.

[13] Alexander Fehr: *Sichere Speicherung geheimer Schlüssel auf einem Raspberry Pi*. Studienarbeit, Humboldt-Universität zu Berlin, 2018. SAR-PR-2018-05.

[14] Wolfgang Rankl und Wolfgang Effing: *Handbuch der Chipkarten — Aufbau, Funktionsweise, Einsatz von Smart Cards*. Hanser, München, 4. Auflage, 2002.

[15] Wikipedia-Autoren: *Hardware-Sicherheitsmodul — Wikipedia*. https://de.wikipedia.org/wiki/Hardware-Sicherheitsmodul, besucht: 2018-09-17.

[16] Zymbit: *Zymkey — Security Module for Raspberry Pi*. https://www.zymbit.com/zymkey/, besucht: 2018-09-17.

[17] Wikipedia-Autoren: *Trusted Platform Module — Wikipedia*. https://de.wikipedia.org/wiki/Trusted_Platform_Module, besucht: 2018-09-18.

[18] Trusted Computing Group: *Trusted Platform Module Library Specification, Family 2.0, Level 00, Revision 01.16*, 2014.

[19] Will Arthur, David Challener und Kenneth Goldmann: *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, 2015.

[20] buyzero: *LetsTrust TPM für den Raspberry Pi*. https://buyzero.de/products/letstrust-hardware-tpm-trusted-platform-module, besucht: 2018-09-18.

[21] Infineon Technologies: *Infineon SLB 9670VQ2.0*. https://www.infineon.com/cms/de/product/security-smart-card-solutions/optiga-embedded-security-solutions/optiga-tpm/slb-9670vq2.0/, besucht: 2018-09-18.

[22] Arm Developer: *Arm TrustZone*. https://developer.arm.com/technologies/trustzone, besucht: 2018-09-18.

[23] Wikipedia-Autoren: *Trusted Execution Environment — Wikipedia*. https://de.wikipedia.org/wiki/Trusted_Execution_Environment, besucht: 2018-09-18.

[24] Linaro-Initiative: *OP-TEE on Raspberry Pi 3*. https://www.op-tee.org/docs/rpi3/, besucht: 2018-09-18.

[25] Linaro-Initiative: *Secure Storage in OP-TEE — GitHub*. https://github.com/OP-TEE/optee_os/blob/master/documentation/secure_storage.md, besucht: 2018-09-18.

[26] Ludovic Rousseau: *CCID free software driver*. https://ccid.apdu.fr/, besucht: 2019-04-06.

[27] OpenSC-Entwickler: *OpenSC — GitHub*. https://github.com/OpenSC/OpenSC, besucht: 2019-04-07.

[28] ISO/IEC Draft International Standard 7816: *Identification cards — Integrated circuit cards — Part 4: Organization, security and commands for interchange*, 2011.

[29] ISO/IEC Draft International Standard 7816: *Identification cards Integrated circuit cards Part 8: Commands for security operations*, 2012.

[30] ISO/IEC Draft International Standard 7816: *Identification cards Integrated circuit cards Part 9: Interindustry commands for card and file management*, 2002.

[31] Advanced Card Systems Ltd.: *CryptoMate Nano Cryptographic USB Token*. https://www.acs.com.hk/en/products/414/cryptomate-nano-cryptographic-usb-tokens/, besucht: 2018-09-08.

[32] Nitrokey UG: *Nitrokey*. https://www.nitrokey.com/de, besucht: 2019-04-10.

[33] Achim Pietig: *Functional Specification of the OpenPGP application on ISO Smart Card Operating Systems, Version 3.3.1*, 2017.

[34] Wikipedia-Autoren: *TCOS Wikipedia*. https://de.wikipedia.org/wiki/TCOS, besucht: 2019-04-13.

[35] T Systems International GmbH: *TCOS-Smartcards*. https://www.telesec.de/de/tcos, besucht: 2019-04-13.

[36] T Systems International GmbH: *Leistungsbeschreibung TeleSec TCOS 3.0 Smart Card OS, Version 1.0*, 2010.

[37] CardContact Systems GmbH: *SmartCard-HSM*. https://www.smartcard-hsm.com/, besucht: 2019-04-15.

[38] Wikipedia-Autoren: *Java Card Wikipedia*. https://en.wikipedia.org/wiki/Java_Card, besucht: 2019-04-18.

[39] Oracle Corporation: *Java Card Technology*. https://www.oracle.com/technetwork/java/embedded/javacard/overview/index.html, besucht: 2019-04-18.

[40] GlobalPlatform Inc.: *GlobalPlatform Technology Card Specification, Version 2.3.1*, 2018.

[41] Karsten Ohme: *GlobalPlatform SourceForge.net*. https://sourceforge.net/projects/globalplatform/, besucht: 2019-04-19.

[42] Martin Paljak: *GlobalPlatformPro GitHub*. https://github.com/martinpaljak/GlobalPlatformPro, besucht: 2019-04-19.

[43] Martin Paljak: *JavaCard Buyer's Guide of 2018*. https://github.com/martinpaljak/GlobalPlatformPro/tree/master/docs/JavaCardBuyersGuide, besucht: 2019-04-19.

[44] Wikipedia-Autoren: *Java Card OpenPlatform — Wikipedia*. https://en.wikipedia.org/wiki/Java_Card_OpenPlatform, besucht: 2019-04-19.

[45] Bundesamt für Sicherheit in der Informationstechnik: *Common Criteria Protection Profile — Cryptographic Service Provider — BSI-CC-PP-0104-2019*, 2019.

[46] Bundesamt für Sicherheit in der Informationstechnik: *Technische Richtlinie BSI TR-03153 — Technische Sicherheitseinrichtung für elektronische Aufzeichnungssysteme*, 2018.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 23. April 2019 .....