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

Sicherer KNX-Schlüsselspeicher auf einer Javacard

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Vincent Weigt
geboren am:
geboren in:

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

eingereicht am: .... verteidigt am: ....

Inhaltsverzeichnis

1 Einleitung 2
1.1 Initialisierung des Schlüsselspeichers . . . 3
1.2 Entschlüsselung eines Telegramms . . . 4
1.3 Inhalt der Arbeit . . . 5

2 Beschreibung des Javacard-Applets 7
2.1 Allgemeines zur Entwicklung eines Javacard-Applets . . . 7
2.2 Installation . . . 8
2.2.1 Statuswörter bei der Installation des Applets . . . 8
2.2.2 Initialisierung der PIN . . . 9
2.2.3 Weitere Initialisierungen im Konstruktor . . . 10
2.3 Verarbeitung einer APDU . . . 11
2.3.1 Bestimmung der Instruktion . . . 12
2.3.2 Übersicht über alle Instruktionen . . . 12
2.3.3 Funktionsweise der einzelnen Instruktionen . . . 15
2.3.4 Nicht enthaltene Instruktionen . . . 16

3 Beschreibung der anderen Softwarekomponente 18
3.1 Übersicht über die einzelnen Klassen . . . 18
3.2 Öffentliche Schnittstelle des Schlüsselspeichers . . . 19
3.3 KNX-Telegramm . . . 21
3.3.1 Rahmen-Format eines KNX-Telegramms . . . 21
3.3.2 Schlüssel-ID . . . 22
3.3.3 Zu verifizierende Nachricht . . . 22
3.4 PIN-Eingabe an einem Kartenleser . . . 23
3.4.1 Kontroll-Code . . . 24
3.4.2 Kommando . . . 25

4 Laufzeittests 26
4.1 Entschlüsselung und Verifikation als getrennte Instruktionen . . . 26
4.2 verschiedene Laufzeittests . . . 27
4.3 Entschlüsselung und Verifikation in einer Instruktion . . . 29
4.4 Vergleich . . . 30
4.5 Weitere Ideen zur Verbesserung der Performance . . . 31

5 Abschluss 34
5.1 Zusammenfassung . . . 34
5.2 Ausblick . . . 35

6 Anhang 37
6.1 Gegenseitige Authentisierung mit dem Secure Channel Protocol . . . 37
6.1.1 Kurzbeschreibung der gegenseitigen Authentisierung . . . 37
6.1.2 Javacard-Applet . . . 38

6.1.3 Java-Programm . . . 39
6.2 KNX-Laufzeitschlüssel aus ETS exportieren . . . 40
6.2.1 Werkzeug zum Extrahieren der Schlüssel aus einem Schlüsselbund 41
6.2.2 Kurzanleitung . . . 41
6.3 Entwicklung von Javacard-Applets unter Linux . . . 41

Literaturverzeichnis **i**

Appendix **iii**

Abkürzungen

AES	Advanced Encryption Standard
AID	Application Identifier
APCI	Application Layer Protocol Control Information
APDU	Application Protocol Data Unit
API	Application Programming Interface
CBC	Cipher Block Chaining
CCM	Counter with CBC-MAC
CMAC	Cipher-based Message Authentication Code
CTR	Counter
ECB	Electronic Code Book
ETS	Engineering Tool Software
FT	Frame Type
GCM	Galois/Counter Mode
HMAC	Hash-based Message Authentication Code
IDS	Instrusion Detection System
ISD	Issuer Security Domain
ISO	International Organization for Standardization
IV	Initialisierungsvektor
MAC	Message Authentication Code
PBKDF2	Password-Based Key Derivation Function 2
PIN	Personal Identification Number
PUK	Personal Unblocking Key
RAM	Random Access Memory
SAI	Security Algorithm Identifier
SCF	Security Control Field
SCP	Secure Channel Protocol
SHA	Secure Hash Algorithms
TPCI	Transport Layer Protocol Control Information
UTF-8	Unicode Transformation Format - 8 Bit
XML	Extensible Markup Language

1 Einleitung

KNX ist ein offener Standard für Heim- und Gebäudeautomation. Die Kommunikation zwischen den Geräten erfolgt dabei in der Regel ungesichert. Insbesondere werden die Daten weder vertraulich noch authentisch übermittelt, was sie anfällig gegenüber unberechtigtem Zugriff und Manipulation macht. Die Erweiterung „KNX Data Secure“ löst dieses Problem durch Verschlüsselung und Authentisierung.

Das langfristige Ziel ist es, ein Intrusion-Detection-System (IDS) für ein „KNX Data Secure“-Netzwerk zu entwerfen. Dieses soll den Datenverkehr überwachen und so verdächtige Aktivitäten erkennen. Das IDS muss also die Möglichkeit haben, beliebige KNX-Telegramme zu entschlüsseln und deren Authentizität mittels MAC zu prüfen. Die symmetrischen Schlüssel dafür müssen an einem sicheren Ort aufbewahrt werden.

In der Diplomarbeit von Alexander Fehr [8] werden Anforderungen an einen solchen Schlüsselspeicher gestellt und ein Konzept zu dessen Umsetzung beschrieben. Das Ziel meiner Arbeit soll es sein, einen Prototypen zu implementieren, der dieses Konzept umsetzt. Dieses Konzept möchte ich im Folgenden kurz beschreiben.

Die Schlüssel können nicht einfach im Dateisystem des IDS gespeichert werden, da ein Angreifer mit Zugriff auf das IDS alle Schlüssel erhalten würde. Eine Idee könnte sein, die Schlüssel auf einer Smartcard zu speichern. Allerdings ist der persistente Speicher einer Smartcard so klein, dass sich keine große Anzahl an Schlüsseln darauf speichern lässt. Also wird in [8] ein hybrider Ansatz zur Speicherung vorgeschlagen. Dabei wird auf der Smartcard ein Wrapping-Schlüssel erzeugt (der zu jedem Zeitpunkt auf der Smartcard verbleibt), mit dem alle im Netzwerk verwendeten KNX-Laufzeitschlüssel verschlüsselt werden. Die verschlüsselten Schlüssel können dann im Dateisystem des IDS abgelegt werden.

Wenn nun ein KNX-Telegramm entschlüsselt werden soll, wird dieses auf die Smartcard übertragen und der dazu passende (verschlüsselte) KNX-Laufzeitschlüssel in die Smartcard importiert. Auf der Smartcard wird der KNX-Laufzeitschlüssel zunächst entschlüsselt. Mit dem entschlüsselten Schlüssel kann dann auf der Smartcard das KNX-Telegramm entschlüsselt werden. Während des Betriebs befindet sich kein Schlüssel im Klartext außerhalb der Smartcard und dennoch ist das IDS in der Lage beliebige Telegramme zu entschlüsseln.

Als Smartcard wird in [8] eine Javacard empfohlen. Javacard ist eine frei programmierbare Smartcard, auf der die benötigten kryptografischen Operationen vorgenommen werden können. Im Rahmen meiner Arbeit habe ich zwei Softwarekomponenten entwickelt: Zum einen eine in Java geschriebene Programmbibliothek, welche dem IDS öffentlich aufrufbare Methoden bereitstellt. Zum anderen wurde ein Javacard-Applet entwickelt, welches die benötigte Smartcard-Funktionalität für den Schlüsselspeicher bereitstellt. Das IDS selbst ist nicht Teil dieser Arbeit.

Um den Schlüsselspeicher zu testen wurde ein echtes „KNX Data Secure“-Netzwerk aufgebaut, welches aus zwei Geräten besteht. Das Betätigen eines Schalters an einem

Abbildung 1: Vereinfachtes Komponentendiagramm. Verbindungen mit einem Kreis am Ende stehen für eine Schnittstelle, die von der entsprechenden Komponente angeboten wird. Ein Halbkreis steht dementsprechend für eine verwendete Schnittstelle. Der Schlüsselspeicher ist eine Java-Programmbibliothek, dessen öffentliche Methoden vom IDS gerufen werden können. Das Javacard-Applet wird auf einer Javacard installiert und stellt Funktionen bereit, die wiederum vom Schlüsselspeicher gerufen werden. Da das IDS nicht Teil dieser Arbeit ist, wird dieses in grau dargestellt.

Gerät sorgte dafür, dass ein verschlüsseltes und authentisiertes KNX-Telegramm über den KNX-Bus versendet wird. Diese Telegramme wurden mithilfe des Konfigurationswerkzeugs ETS (Engineering Tool Software) aufgezeichnet. Die Aufzeichnung enthält neben den verschlüsselten Telegrammen auch die entsprechenden Klartexte. Außerdem wurden die verwendeten Schlüssel aus der ETS exportiert. Ob der Schlüsselspeicher die Telegramme korrekt entschlüsselt und authentisiert konnte dann getestet werden, indem die verwendeten Schlüssel in den Schlüsselspeicher importiert und die aufgezeichneten Telegramme vom Schlüsselspeicher entschlüsselt wurden. Das Ergebnis wurde dann mit den Klartexten der Aufzeichnung verglichen.

1.1 Initialisierung des Schlüsselspeichers

Bevor der Schlüsselspeicher zur Entschlüsselung von Telegrammen verwendet werden kann, muss zuerst ein Wrapping-Schlüssel auf der Smartcard generiert werden. Dazu wird ein Smartcard-interner Zufallszahlen-Generator verwendet. Der Wrapping-Schlüssel wird also auf der Karte erstellt und verbleibt dort.

Danach müssen alle verwendeten KNX-Laufzeitschlüssel mit diesem Wrapping-Schlüssel verschlüsselt werden. Dieser Vorgang soll in dieser Arbeit als „Verpacken“ des Schlüssels (vom englischen „key wrapping“) bezeichnet werden. Um einen KNX-Laufzeitschlüssel zu verpacken, wird zunächst ein MAC über dem Klartext-Schlüssel berechnet. Dann werden der KNX-Laufzeitschlüssel und der berechnete MAC konkateniert. Das Ergebnis der Konkatenation wird dann verschlüsselt.

Die verpackten Schlüssel müssen dann auf einem persistenten Speicher abgelegt werden, auf den der Schlüsselspeicher jederzeit Zugriff hat. Dabei wird jeder verpackte Schlüssel zusammen mit einer eindeutigen Schlüssel-ID in einer Zuordnungstabelle, d.h. einer Liste von Schlüssel-Wert-Paaren, gespeichert. Diese Datenstruktur wird serialisiert und als Datei im Dateisystem abgelegt.

Dieses gesamte Vorgehen soll als Initialisierungsphase des Schlüsselspeichers bezeichnet werden. Die eigentliche Entschlüsselung und Verifikation von KNX-Telegrammen findet dagegen in der Betriebsphase statt.

Während der Initialisierungsphase müssen die KNX-Laufzeitschlüssel also im Klartext vorliegen. Aus diesem Grund muss die Initialisierung in einer sicheren Umgebung stattfinden. Es ist denkbar die Funktionen der beiden Phasen in verschiedene Programme auszulagern. So könnte der Schlüsselspeicher vom Installateur eines KNX-Netzwerks initialisiert werden. Ein potenziell unsicheres bzw. kompromittiertes IDS im Betrieb könnte dann nurnoch auf Funktionen der Betriebsphase zugreifen und nicht etwa den Wrapping-Schlüssel löschen und damit die Verfügbarkeit des Schlüsselspeichers einschränken.

In der hier vorgestellten Implementation jedoch sind die Funktionen der Initialiserungs- und der Betriebsphase in einem Programm enthalten. Um zu verhindern, dass der Wrapping-Schlüssel während des Betriebs verändert oder entfernt wird, werden diese Funktionen durch eine PIN abgesichert. Die Initialisierung des Speichers soll also nur nach der Eingabe einer korrekten PIN möglich sein. Zur Eingabe der PIN soll auch nur ein kontaktbehafteter Kartenleser mit eigener Tastatur verwendet werden dürfen. Indem während der Betriebsphase ein Kartenleser ohne Tastatur und/oder mit kontaktloser Schnittstelle verwendet wird, kann verhindert werden, dass ein Angreifer mit physischem Zugriff falsche PIN-Eingaben vornimmt und damit die PIN sperrt.

1.2 Entschlüsselung eines Telegramms

Die Entschlüsselung und Verifikation von KNX-Telegrammen wurde auf zwei verschiedene Arten implementiert. Um den Unterschied dieser beiden Methoden zu verdeutlichen, soll zunächst der Ablauf der Entschlüsselung eines KNX-Telegramms grob beschrieben werden. Es wird angenommen, dass der Schlüsselspeicher initialisiert ist. Das bedeutet, dass ein Wrapping-Schlüssel auf der Smartcard existiert, mit dem die KNX-Laufzeitschlüssel verschlüsselt wurden. Außerdem muss die Zuordnungstabelle mit den verpackten KNX-Laufzeitschlüsseln existieren und den passenden KNX-Laufzeitschlüssel enthalten.

Angenommen wir haben nun ein KNX-Telegramm, welches entschlüsselt werden soll.

1. Wir müssen zunächst herausfinden, mit welchem Schlüssel dieses Telegramm verschlüsselt wurde und diesen (verpackten) Schlüssel der Smartcard übergeben, damit diese den Schlüssel entschlüsselt.

2. Erstellen des Initialisierungsvektors nach KNX-Spezifikation.

3. Entschlüsselung des Telegramms.

4. Aus dem entschlüsselten KNX-Telegramm wird jetzt die Nachricht konstruiert, über welcher der Sender der Nachricht den MAC berechnet hat.

5. Der MAC wird berechnet und mit dem entschlüsseltem MAC verglichen. Im Erfolgsfall kennen wir jetzt den Klartext der Nutzdaten des Telegramms.

In [8] ist vorgesehen, dass der Schlüsselspeicher den Initialisierungsvektor erstellt und die Entschlüsselung dann auf der Smartcard vorgenommen wird. Mit dem Klartext konstruiert der Schlüsselspeicher dann die Nachricht, über die auf der Smartcard der MAC berechnet wird. Der Schlüsselspeicher kann jetzt die Gültigkeit des MAC prüfen. Hier ist zu beachten, dass der Schlüsselspeicher viel Arbeit übernimmt und Funktionen der Smartcard nur verwendet werden, wenn es notwendig ist, d.h. genau dann wenn der KNX-Laufzeitschlüssel verwendet werden muss. Eine alternative Vorgehensweise wäre, dass der Schlüsselspeicher einfach das verschlüsselte Telegramm der Smartcard übergibt und alles andere dann auf der Smartcard durchgeführt wird.

Beide Methoden habe ich implementiert. Das hat zur Folge, dass einige Funktionen redundant sind. Es existieren also im Javacard-Applet mehrere Instruktionen, die eine Entschlüsselung durchführen. Auch der Schlüsselspeicher enthält mehrere Funktionen, die das gleiche Ergebnis haben, sich aber in ihrer Implementation unterscheiden. Warum es die beiden Verfahren gibt und wo jeweils die Vor- und Nachteile liegen, wird in dieser Arbeit erörtert.

1.3 Inhalt der Arbeit

In dieser Arbeit soll der von mir entwickelte Prototyp vorgestellt werden.

In Abschnitt 2 wird das Javacard-Applet beschrieben. Dabei werden zunächst allgemeine Hinweise zur Entwicklung eines Javacard-Applets gegeben. Es wird erklärt wie das Applet auf einer Javacard installiert werden kann und welche Fehler bei der Installation auftreten können. Danach folgt eine Beschreibung der verwendeten Datenstrukturen und Kryptoprimitiven, sowie aller Instruktionen, die das Applet ausführen kann. Jede Instruktion wird in ihrer Funktionsweise beschrieben.

In Abschnitt 3 soll der Teil des Schlüsselspeichers beschrieben werden, der nicht auf der Smartcard läuft. Dazu wird zunächst auf die Architektur des Schlüsselspeichers eingegangen und die einzelnen Komponenten beschrieben. Es folgt die Beschreibung der öffentlichen Schnittstelle, die zum Beispiel vom IDS verwendet werden kann. Es soll in diesem Abschnitt nicht jedes Implementationsdetail erklärt werden. Manches könnte aber auch für andere Projekte interessant sein. Darum wird im Detail darauf eingegangen, wie ein KNX-Telegramm aufgebaut ist und wie die PIN-Eingabe am Kartenleser funktioniert.

In Abschnitt 4 werden Ergebnisse verschiedener Laufzeittests vorgestellt. Eine wichtige Metrik zur Bewertung des Schlüsselspeichers ist sein Durchsatz, d.h. wieviele Telegramme pro Sekunde entschlüsselt werden können. Beide im vorigen Abschnitt beschriebenen Verfahren zur Entschlüsselung von Telegrammen wurden gemessen und miteinander verglichen. Darüberhinaus wurden weitere Laufzeittests gemacht, um zu bestimmen wieviel Zeit verschiedene Operationen auf der Smartcard in Anspruch

nehmen. Diese Ergebnisse werden verwendet, um die Laufzeit des Schlüsselspeichers zu erklären und den Flaschenhals zu identifizieren. Zum Schluss werden hier weitere Ideen zur Verbesserung der Laufzeit genannt, die jedoch nicht implementiert wurden.

In Abschnitt 5 wird das Erreichte zusammengefasst und es werden Hinweise gegeben, wie der Schlüsselspeicher noch verbessert werden kann. Dabei geht es insbesondere auch um Maßnahmen, welche die Sicherheit in der Betriebsphase betreffen.

Abschnitt 6 beinhaltet Themen, die nicht zum Schlüsselspeicher gehören, aber im Rahmen dieser Arbeit entstanden sind. Der erste Unterabschnitt beschreibt, wie man die gegenseitige Authentifizerung eines Javacard-Applets mit einer anderen Softwarekomponente mithilfe des Secure Channel Protokolls durchführt. Ein weiterer Abschnitt erklärt, wie man die KNX-Laufzeitschlüssel eines echten KNX-Netzwerks mithilfe der Konfigurationssoftware ETS exportiert. Schließlich wird beschrieben, wie Javacard-Applets unter Linux gebaut und mit einer virtuellen Javacard getestet werden können.

Die Quelltexte aller erstellten Programme sind verfügbar in einem Git-Repository unter `https://gitlab.informatik.hu-berlin.de/weigtvin/keystore`.

2 Beschreibung des Javacard-Applets

In diesem Abschnitt wird die Funktionsweise des Javacard-Applets beschrieben. Die hier verwendeten Klassen- und Methodennamen sind dokumentiert in der *Javacard-API* [19].

2.1 Allgemeines zur Entwicklung eines Javacard-Applets

Jedes Javacard-Applet muss von der abstrakten Klasse `javacard.framework.Applet` erben. Dazu ist es notwendig, die abstrakte Methode der Basisklasse `process()` zu implementieren. Zusätzlich muss die Methode `install()` existieren, in der die Methode `register()` gerufen wird. Ein minimales, kompilierbares Applet könnte folgendermaßen aussehen:

```
package main;

import javacard.framework.APDU;
import javacard.framework.Applet;

public class Smartcard extends Applet {

    public static void install(byte[] bArray, short bOffset, byte bLength) {
        new Smartcard();
    }

    protected Smartcard() {
        register();
    }

    public void process(APDU apdu) {
    }
}
```

Dieses Applet kann auf einer Javacard installiert werden. Bei der Installation wird einmalig die Methode `install()` von der Javacard-Laufzeitumgebung gerufen. Diese Methode wird nach erfolgreicher Installation nie mehr gerufen. Nach erfolgreicher Installation kann das Applet selektiert werden. Nachdem das Applet erfolgreich selektiert wurde, können APDUs an das Applet gesendet werden. Jedes Mal wenn eine APDU an dieses Applet gesendet wird, wird die Methode `process()` gerufen. Als Argument wird die jeweilige APDU übergeben. Wenn die Methode `process()` zurückkehrt, sendet das Applet eine Antwort APDU mit Statuswort 0x9000 zurück. Im Fehlerfall muss eine `javacard.framework.ISOException` mit einem Statuswort ungleich 0x9000 geworfen werden.

In der Klasse `javacard.framework.Applet` sind weitere Methoden, die von der Javacard-Laufzeitumgebung gerufen werden, welche aber nicht zwingend benötigt werden. So kann zum Beispiel die Methode `deselect()` implementiert werden, die gerufen wird wenn das Applet deselektiert wird.

Auf einer Javacard existieren zwei verschiedene Arten von Speicher: Flüchtiger bzw. transienter RAM, sowie persistenter Speicher (EEPROM oder Flash). Daten die im RAM gehalten werden gelten als verloren, sobald die Karte aus dem Kartenleser entfernt wird. Daten im persistenten Speicher bleiben hingegen erhalten. Schreibzugriffe auf den persistenten Speicher reduzieren die Lebensdauer des Speichers. Daher sollten Daten, die häufig verändert werden, wenn möglich im RAM gehalten werden.

Objekte werden normalerweise im persistenten Speicher angelegt. Um die Lebensdauer des Speichers zu erhöhen ist es sinnvoll, alle Objekte die das Applet jemals benötigt, nur ein einziges Mal anzulegen und später in verschiedenen Situationen wiederzuverwenden. Für diese Initialisierung eignet sich die `install()`-Methode, da diese nur ein einziges Mal ausgeführt wird. Bei der Entwicklung des Applets habe ich mich an dieses Vorgehen gehalten.

Weitere Informationen zur Entwicklung eines Javacard-Applets sind zu finden in *An Introduction to Java Card Technology* [2] Teil 1, 2 und 3, sowie ausführlicher in *Java Card Platform Runtime Environment Specification* [17].

2.2 Installation

In diesem Abschnitt wird beschrieben, was bei der Installation des Applets passiert. Die Installation umfasst den Eintritt in die `install()`-Methode bis zum Aufruf der `register()`-Methode.

Bei der Installation des Applets muss die PIN als Parameter übergeben werden. Die Installation mit PIN „123456“ kann durchgeführt werden mit:

```
$ java -jar gp.jar --install main.cap --params 010203040506
```

Dabei wird das Programm *GlobalPlatformPro* [13] verwendet. Die Installationsparameter werden in hexadezimaler Schreibweise angegeben. Ein Byte stellt eine Ziffer der PIN dar.

Hierbei wird vorausgesetzt, dass der Globalplatform-Schlüssel der Karte noch den Standardwert 0x40 ... 0x4f hat. Andernfalls muss noch der korrekte Schlüssel angegeben werden.

2.2.1 Statuswörter bei der Installation des Applets

Bei der Installation können im Fehlerfall folgende Statuswörter zurückkommen:

0x6985: Es wurde keine PIN als Installationsparameter übergeben.

0x6700: Die PIN ist zu lang oder zu kurz.

0x6901: Der PIN-Typ wird von der Karte nicht unterstützt.

0x6902: Der Zufallszahlen-Algorithmus wird von der Karte nicht unterstützt.

0x6903: Der Schlüsseltyp des Wrapping-Schlüssels wird von der Karte nicht unterstützt.

0x6904: Der Schlüsseltyp des Laufzeitschlüssels wird von der Karte nicht unterstützt.

0x6905: AES-ECB wird von der Karte nicht unterstützt.

0x6906: CBC-MAC wird von der Karte nicht unterstützt.

0x6907: AES-CTR wird von der Karte nicht unterstützt.

Die Statuswörter 0x6901 bis 0x6907 sind in *ISO 7816-4* [15] reserviert für zukünftige Nutzung. Diese Statuswörter werden dann zurückgegeben, wenn das Applet auf einer Javacard installiert wird, die einen vom Applet verwendeten Algorithmus nicht beherrscht. Damit bei der Installation ersichtlich ist, welche Funktion nicht unterstützt wird, habe ich für jeden Algorithmus ein eigenes Statuswort definiert. Da in *ISO 7816-4* [15] keine geeigneten Statuswörter dafür definiert sind, habe ich an dieser Stelle auf nicht-definierte, reservierte Statuswörter zurückgegriffen.

Auch nach der Installation, während der Ausführung von Instruktionen, werden im Fehlerfall Statuswörter zurückgegeben. Dabei werden aber ausschließlich in *ISO 7816-4* [15] definierte Statuswörter verwendet.

2.2.2 Initialisierung der PIN

Die PIN muss bei der Installation als Installationsparameter angegeben werden. Die Installationsparameter werden der Funktion `install()` als Argument übergeben. Diese befinden sich im Byte-Array `bArray`. Zusätzlich dazu werden noch ein Versatz innerhalb dieses Arrays (`bOffset`), sowie dessen Länge (`bLength`) übergeben. Die erste Anweisung in `install()` ist der Aufruf der Funktion `getPinOffset()`. Diese Funktion nimmt das Byte-Array, den Versatz und die Länge, und gibt den Versatz innerhalb des Byte-Arrays zurück, an dem die PIN steht. Informationen zum Aufbau des Byte-Arrays befinden sich in *Installation Parameters (JavaCardOS Wiki)* [14]. Ist der PIN-Versatz ermittelt, wird noch die Länge der PIN bestimmt und anschließend der Konstruktor mit diesen Informationen gerufen.

Im Konstruktor werden Funktionen gerufen, die Objekte initialisieren. Zum Schluss erfolgt noch der notwendige Aufruf von `register()`. Die erste Funktion, die gerufen wird, ist `initPin()`. Dieser Funktion wird das Byte-Array mit den Installationsparametern, sowie der PIN-Versatz und die PIN-Länge übergeben.

Zu Beginn dieser Funktion wird der Typ der PIN festgelegt. Es wurde der Typ `OWNER_PIN_X_WITH_PREDECREMENT` gewählt. Dieser unterstützt das Dekrementieren des Versuchszählers bevor die PIN geprüft wird. Wenn die korrekte PIN eingegeben wurde, wird der Zähler danach wieder auf das Maximum gesetzt. [19] Eine Alternative dazu wäre, dass der Zähler erst nach der Prüfung der PIN verändert wird. Bei diesem Verfahren ist allerdings vorstellbar, dass ein Angreifer nach der Eingabe einer

PIN durch einen Seitenkanal Informationen darüber erhält, ob die eingegebene PIN falsch ist, noch bevor der Versuchszähler dekrementiert wird. In einem solchen Fall könnte die Karte aus dem Leser entfernt werden, ohne dass dieser Zähler verändert wird, d.h. es könnten beliebig viele Versuche durchgeführt werden. Das Predekrement verhindert einen solchen Angriff. Der Nachteil dieses Verfahrens ist jedoch, dass bei jeder PIN-Eingabe der persistente Speicher der Karte beschrieben wird.

Als nächstes wird die PIN mit der Funktion `buildOwnerPIN()` initialisiert. Dabei wird neben dem Typ noch die maximale PIN-Länge, sowie die Anzahl der möglichen Fehlversuche bis zur Sperrung der PIN angegeben. Das Versuchslimit ist drei, die maximale PIN-Länge liegt bei sechs. Diese Werte sind jeweils in den Konstanten `TRY_LIMIT` und `MAX_PIN_SIZE` gespeichert und können bei Bedarf angepasst werden. Die Länge der PIN sollte 15 nicht übersteigen. (Siehe [7], bei cyberjack Kartenlesern von ReinerSCT kann es Probleme geben, wenn die PIN länger ist als 15 Ziffern. Habe ich jedoch nicht getestet.) Zum Schluss wird die PIN mit der Methode `update()` gesetzt und die Funktion `initPin()` kehrt zurück.

2.2.3 Weitere Initialisierungen im Konstruktor

initRng(): Als nächstes wird die Funktion `initRng()` gerufen, in welcher der Zufallszahlen-Generator initialisiert wird. Dieser soll verwendet werden, um die Wrapping-Schlüssel auf der Karte zu generieren. Als Typ wurde `ALG_KEYGENERATION` ausgewählt, da sich dieser zur Erzeugung von Schlüsseln eignet. [19]

Die von mir verwendete Javacard hat einen SmartMX3-P71D320-Mikroprozessor, welcher sowohl über einen AIS31-konformen Echt-Zufallszahlen-Generator, als auch über einen deterministischen Pseudo-Zufallszahlen-Generator verfügt. [26] Ob die Schlüssel auf der Karte ausschließlich durch echten Zufall erzeugt werden, oder ob der echte Zufall z.B. nur einen Seed für den Pseudo-Zufall erzeugt, ist mir nicht bekannt.

initWrappingKey(): In `initWrappingKey()` werden die Wrapping-Schlüssel initialisiert. Dabei werden lediglich Objekte erzeugt, die Schlüssel werden hier noch nicht auf einen konkreten Wert gesetzt. Dafür wird nach der Installation eine Instruktion zur Verfügung stehen. Es werden zwei Schlüssel erzeugt: Einer zum Berechnen eines Message Authentication Codes (MAC) über dem zu verpackenden Schlüssel und ein anderer zum Verschlüsseln des Schlüssels. Es sind beides AES-256-Schlüssel, die im persistenten Speicher der Karte gehalten werden.

initKeyCipher(): In `initKeyCipher()` wird der Verschlüsselungsalgorithmus für die Verschlüsselung mit dem Wrapping-Schlüssel bestimmt. Dies ist AES-ECB ohne Padding, da sowohl die zu verschlüsselnden Schlüssel, als auch die MACs ganze Blocklänge haben. Wenn eine andere MAC-Länge verwendet werden soll, muss an dieser Stelle das Padding und ggf. der Betriebsmodus angepasst werden.

initMac(): In `initMac()` wird der MAC-Typ bestimmt. Dieser soll ein CBC-MAC

der Länge 16 Bytes sein, welcher für die Verifikation der KNX-Laufzeitschlüssel verwendet wird.

Eine mögliche Alternative dazu stellt das HMAC-Verfahren dar, welches jedoch von der von mir verwendeten Javacard nicht unterstützt wird. Anstelle von CBC-MAC + Verschlüsselung könnte auch der GCM bzw. CCM-Betriebsmodus verwendet werden, die authentische Verschlüsselung anbieten. Diese Verfahren werden jedoch ebenfalls nicht von der von mir verwendeten Javacard unterstützt.

KNX-Telegramme werden ebenfalls mit dem CBC-MAC verifiziert, hierbei wird der MAC jedoch auf 4 Bytes gekürzt.

initKey(): In `initKey()` wird der Schlüssel initialisiert, mit dem während der Betriebsphase KNX-Telegramme entschlüsselt werden sollen. Wie schon bei der Initialisierung der Wrapping-Schlüssel, wird hier nur ein Objekt erzeugt, der Schlüssel wird noch nicht auf einen Wert gesetzt. Es ist ein AES-128-Schlüssel (so wie es der KNX-Standard vorsieht), welcher im transienten Speicher der Smartcard gehalten wird und bei der Deselektion des Applets gelöscht wird.

initCipher(): In `initCipher()` wird der Verschlüsselungsalgorithmus AES im Counter-Modus (AES-CTR) zur Entschlüsselung von KNX-Telegrammen initialisiert. KNX Data Secure verwendet den CCM-Betriebsmodus zur Ver- und Entschlüsselung. Das bedeutet, dass die Daten mit einem 4 Byte langen CBC-MAC verifiziert, und mit AES-CTR ver- und entschlüsselt werden. Das standardisierte CCM-Verfahren [4] beschreibt aber auch den Aufbau des Initialisierungsvektors (IV) für AES-CTR und der zu authentisierenden Nachricht. An dieser Stelle unterscheidet sich jedoch der KNX-Standard vom CCM-Verfahren. Javacard unterstützt grundsätzlich den CCM-Modus, dieser kann jedoch hier nicht verwendet werden, da der IV und die Nachricht bei der Entschlüsselung von KNX-Telegrammen anders aufgebaut werden müssen. Aus diesem Grund muss der CCM-Modus mit AES-CTR und CBC-MAC umgesetzt werden.

initBuffers(): In `initBuffers()` werden Byte-Arrays initialisiert, die während der Verarbeitung der Instruktion `INS_DECRYPT_AND_VERIFY` verwendet werden. Für diese Instruktion reicht der APDU-Puffer alleine nicht aus. Dafür wird transienter Speicher verwendet, welcher bei Deselektion des Applets gelöscht wird.

register(): Zuletzt wird `register()` gerufen und der Konstruktor sowie `install()` kehren zurück. Wurde bis zu diesem Zeitpunkt keine Exception geworfen, ist die Installation erfolgreich mit Statuswort 0x9000. Die `install()`-Funktion wird während der gesamten Lebensdauer des Applets nicht mehr gerufen.

2.3 Verarbeitung einer APDU

In diesem Abschnitt wird beschrieben, was passiert, wenn eine APDU an unser Applet weitergeleitet wird. Zunächst wird `process()` gerufen und die APDU als Argument

übergeben. Dann wird geprüft, welche Instruktion in der APDU angegeben ist. Diese wird ausgeführt. Während der Ausführung wird im Fehlerfall eine Exception geworfen und ein Statuswort ungleich 0x9000 zurückgegeben. Im Erfolgsfall kehrt `process()` zurück und das Statuswort 0x9000 wird zurückgegeben.

2.3.1 Bestimmung der Instruktion

Beim Einstieg in die Funktion `process()` wird zunächst `getBuffer()` gerufen, damit wir auf den APDU-Puffer zugreifen können. Im nächsten Schritt können wir dort an einer bestimmten Position das Instruktions-Byte lesen. Dieses gleichen wir mit den von uns implementierten Instruktionen ab und rufen die entsprechende Funktion, welche die Instruktion ausführt.

2.3.2 Übersicht über alle Instruktionen

In diesem Abschnitt wird die Schnittstelle des Javacard-Applets zum Schlüsselspeicher beschrieben. Diese Schnittstellenbeschreibung ist wichtig für die Entwicklung der Softwarekomponente, die mit dem Javacard-Applet mithilfe von APDUs kommuniziert. Zu jeder bereitgestellten Instruktion gibt es zunächst eine Kurzbeschreibung. Dann wird der Aufbau der APDU beschrieben, die an das Applet gesendet werden muss, damit die jeweilige Instruktion ausgeführt wird. Außerdem werden die Antwort-APDUs beschrieben, die bei den einzelnen Instruktionen zurückkommen können. Hierbei sind insbesondere die Statuswörter wichtig, die im Fehlerfall zurückgegeben werden.

Bei der Wahl der Statuswörter habe ich mich an *ISO 7816-4* [15] orientiert. Jedoch kann das gleiche Statuswort in verschiedenen Kontexten eine leicht andere Bedeutung haben. So kann zum Beispiel das Statuswort 0x6985 sowohl bei der Instruktion `INS_GENERATE_WRAPPING_KEY` als auch bei der Instruktion `INS_DECRYPT` vorkommen. In *ISO 7816-4* [15] ist 0x6985 definiert als „Conditions of use not satisfied", das heißt die Nutzungsbedingungen sind nicht erfüllt. Bevor `INS_GENERATE_WRAPPING_KEY` verwendet werden kann, muss sich der Benutzer authentisiert haben. Hat er dies nicht, so sind die Nutzungsbedingungen für diese Instruktion nicht erfüllt und es kommt 0x6985 zurück. Bei `INS_DECRYPT` muss vor der Nutzung ein Schlüssel initialisiert worden sein. In diesem Zusammenhang wird 0x6985 zurückgegeben, wenn kein Schlüssel initialisiert ist.

```
INS_AUTHENTICATE:
    Prüfe ob die eingegebene PIN korrekt ist.
    Nicht möglich über eine kontaktlose Schnittstelle.

    Kommando-APDU:
    0 0x20 0 0 lc PIN

    Antwort-APDU:
    0x9000: Erfolg
```

```
    0x6982: PIN ist falsch
    0x6983: Falls eine kontaktlose Schnittstelle verwendet wird
    0x6700: Daten zu lang (mehr als MAX_PIN_SIZE + 1 Byte Längeninformation)

INS_GENERATE_WRAPPING_KEY:
    Generiert ein neues Paar Wrapping-Schlüssel aus Zufallsdaten.
    Dieses Paar besteht aus einem Schlüssel zur Verschlüsselung
    und einem zur Authentisierung.
    Die Instruktion setzt Authentifikation mittels PIN voraus.

    Kommando-APDU:
    0 0x10 0 0

    Antwort-APDU:
    0x9000: Erfolg
    0x6985: Nicht authentisiert

INS_REMOVE_WRAPPING_KEY:
    Entfernt die Wrapping-Schlüssel.
    Die Instruktion setzt Authentifikation mittels PIN voraus.

    Kommando-APDU:
    0 0x30 0 0

    Antwort-APDU:
    0x9000: Erfolg
    0x6985: Nicht authentisiert

INS_WRAP_KEY:
    Nimmt einen KNX-Laufzeitschlüssel, verpackt ihn,
    und gibt den verpackten Laufzeitschlüssel zurück.

    Kommando-APDU:
    0 0x40 0 0 16 key 32

    Antwort-APDU:
    Data: Verpackter Laufzeitschlüssel
    0x9000: Erfolg
    0x6700: Die übermittelten Daten sind nicht 16 Bytes lang.
    0x6985: Der Wrapping-Schlüssel ist nicht initialisiert.
    0x6400: Es ist ein Fehler bei der Berechnung des MAC
            oder der Verschlüsselung aufgetreten.

INS_UNWRAP_KEY:
    Nimmt einen verpackten KNX-Laufzeitschlüssel und entpackt ihn.
    Dieser Schlüssel wird bei der Deselektion des Applets gelöscht.

    Kommando-APDU:
    0 0x50 0 0 32 wrappedKey

    Antwort-APDU:
    0x9000: Erfolg
```

```
    0x6700: Die übermittelten Daten sind nicht 32 Bytes lang.
    0x6985: Der Wrapping-Schlüssel ist nicht initialisiert.
    0x6400: Fehler bei der Entschlüsselung
    0x6982: MAC-Verifikation ist fehlgeschlagen

INS_DECRYPT:
    Entschlüsselt ein Kryptogramm mit AES-CTR.
    Benötigt einen 16 Byte langen Initialisierungsvektor
    und mindestens ein Byte Chiffretext.

    Kommando-APDU:
    0 0xA0 0 0 >16 iv1 ... iv16 data datalen

    Antwort-APDU:
    Data: Klartext
    0x9000: Erfolg
    0x6700: Die übermittelten Daten sind zu kurz.
    0x6985: Schlüssel nicht initialisiert
    0x6400: Fehler bei der Entschlüsselung

INS_VERIFY:
    Verifiziert eine Nachricht mittels eines vier Byte langen CBC-MAC.
    Benötigt einen 4 Byte langen MAC und mindestens ein Byte Nachrichtentext.

    Kommando-APDU:
    0 0x80 0 0 >4 mac1 ... mac4 message

    Antwort-APDU:
    0x9000: Erfolg
    0x6700: Die übermittelten Daten sind zu kurz.
    0x6985: Schlüssel nicht initialisiert
    0x6400: Fehler bei der Berechnung des MAC.
    0x6982: MAC-Verifikation ist fehlgeschlagen

INS_DECRYPT_AND_VERIFY:
    Entschlüsselt ein KNX-Telegramm mit AES-CTR
    und verifiziert den Klartext mittels eines vier Byte langen CBC-MAC.

    Kommando-APDU:
    0 0xB0 p1 0 >22 telegram_1 ... telegram_n le
    p1 == 0: Entschlüsselung und Verifikation => le = (n - 22)
    p1 != 0: Nur Verifikation => le = 0

    Antwort-APDU:
    Data: Klartext
    0x9000: Erfolg
    0x6700: Die übermittelten Daten sind zu kurz.
    0x6985: Schlüssel nicht initialisiert.
    0x6400: Fehler bei der Berechnung des MAC oder der Entschlüsselung.
    0x6982: MAC-Verifikation ist fehlgeschlagen
```

2.3.3 Funktionsweise der einzelnen Instruktionen

In diesem Abschnitt wird beschrieben, wie die einzelnen Instruktionen implementiert sind.

INS_AUTHENTICATE: Zuerst wird der Versuchszähler dekrementiert, dann wird durch Aufruf von `check()` die PIN auf Korrektheit geprüft. Vorher wird durch die Funktion `isContactless()` geprüft, ob eine kontaktlose Schnittstelle verwendet wird. Wenn das der Fall ist, wird die Operation abgebrochen. Eine Authentifikation mittels PIN kann nur über eine kontaktbehaftete Schnittstelle erfolgen.

INS_GENERATE_WRAPPING_KEY: Diese Instruktion darf nur nach erfolgreicher Authentifikation mittels PIN ausgeführt werden. Um das zu überprüfen wird `verifyPin()` gerufen. Diese Hilfsfunktion prüft, ob die PIN durch einen Aufruf der Instruktion `INS_AUTHENTICATE` korrekt eingegeben wurde. Falls ja, wird `reset()` gerufen, was den Validierungsstatus zurücksetzt, d.h. falls eine weitere Instruktion ausgeführt werden soll, für die der Benutzer authentisiert werden muss, muss die PIN vorher erneut eingegeben werden.

Nach erfolgreicher Authentifikation wird `nextBytes()` gerufen. Dabei werden mit dem karteninternen Zufallszahlen-Generator 64 Bytes generiert, d.h. zweimal 256 Bit für zwei AES-256 Schlüssel. Die Schlüssel werden dann mit `setKey()` auf die entsprechenden Werte gesetzt. Damit gelten diese Schlüssel als initialisiert und können verwendet werden.

INS_REMOVE_WRAPPING_KEY: Auch hier wird zunächst der Verifikationsstatus der PIN geprüft und im Anschluss die beiden Wrapping-Schlüssel mit `clearKey()` gelöscht.

INS_WRAP_KEY: Zuerst wird der MAC über dem zu verpackendem Schlüssel mit `sign()` berechnet. (In der *Javacard-API* [19] werden sowohl Signaturen als auch MACs unter der Klasse `javacard.security.Signature` geführt. Darum heißt die Funktion zum Berechnen des MAC `sign()`, obwohl hier nicht signiert wird.) Der MAC wird in den APDU-Puffer unmittelbar hinter den Schlüssel geschrieben. Dann wird der Schlüssel zusammen mit dem MAC mit `doFinal()` verschlüsselt. Da die Daten ganze Blocklänge haben, kann das Chiffrat genau an die Stelle im APDU-Puffer geschrieben werden, an dem der Klartext steht. Zum Schluss wird der verpackte Schlüssel mit `setOutgoingAndSend()` zurückgegeben.

INS_UNWRAP_KEY: Hier wird die Wrap-Operation umgekehrt, d.h. zuerst wird der verpackte Schlüssel mit `doFinal()` entschlüsselt, und dann wird mit `verify()` der MAC über dem entschlüsseltem Schlüssel berechnet und mit dem entschlüsselten MAC verglichen. Bei erfolgreicher Prüfung wird der Schlüssel mit `setKey()` gesetzt und kann zur Entschlüsselung von KNX-Telegrammen verwendet werden. Ein auf der Smartcard entschlüsselter Laufzeitschlüssel wird nicht nach außen gegeben.

INS_DECRYPT: Entschlüsselt eingehende Daten mit AES-128-CTR und gibt den Klartext zurück. Der Schlüssel muss vorher mit der `INS_UNWRAP_KEY`-Instruktion gesetzt werden. Wird benötigt um KNX-Telegramme zu entschlüsseln.

INS_VERIFY: Berechnet den MAC über die eingehenden Daten und vergleicht diesen mit dem übermitteltem MAC. Der Schlüssel muss vorher mit der `INS_UNWRAP_KEY`-Instruktion gesetzt werden. Wird benötigt um KNX-Telegramme zu verifizieren.

INS_DECRYPT_AND_VERIFY: Diese Instruktion nimmt ein verschlüsseltes KNX-Telegramm entgegen und gibt den Klartext zurück, sofern die Prüfung der Authentizität erfolgreich war. Ablauf:

1. Bestimmen des Initialisierungsvektors (bzw. des Counters) nach KNX-Spezifikation.
2. Entschlüsselung der Daten + MAC. (Oder nur MAC, falls das KNX-Telegramm nur authentisiert werden muss.)
3. Setzen der Nachricht, über welcher der MAC berechnet wurde.
4. Berechnung des MAC und Vergleich mit dem entschlüsseltem MAC.

Falls die Daten verschlüsselt sind wird der Klartext zurückgegeben, andernfalls wird nur Statuswort 0x9000 zurückgegeben.

Die Entschlüsselungs- und Verifikationsoperationen funktionieren ähnlich wie in den entsprechenden einzelnen Instruktionen. Hier kommt aber noch dazu, dass die Daten aus dem KNX-Telegramm gelesen und aufbereitet werden müssen. Beispielsweise steht in einem KNX-Telegramm der MAC nach den Daten, entschlüsselt werden muss aber in der Reihenfolge: Erst der MAC und dann die Daten. Im Gegensatz zu den anderen Instruktionen reicht hier der APDU-Puffer nicht mehr aus, es müssen Byte-Arrays verwendet werden. Diese wurden bei der Installation in `initBuffers()` allokiert.

Diese Instruktion ist äquivalent zur Verwendung der `INS_DECRYPT` und `INS_VERIFY` Instruktionen, wobei der IV und die zu verfizierende Nachricht dann, wie in [8] vorgeschlagen, außerhalb der Smartcard gebaut werden müssen. In Abschnitt 4.4 werden diese beiden Verfahren miteinander verglichen.

2.3.4 Nicht enthaltene Instruktionen

Export von Schlüsseln im Klartext Die Wrapping-Schlüssel werden auf der Karte erzeugt und können zum Ver- und Entpacken von KNX-Laufzeitschlüsseln verwendet werden. Es gibt keine Möglichkeit, die Wrapping-Schlüssel zu exportieren.

KNX-Laufzeitschlüssel werden auf der Karte entschlüsselt und können zur Entschlüsselung und MAC-Verifikation von KNX-Telegrammen verwendet werden. Es gibt keine Möglichkeit, die KNX-Laufzeitschlüssel im Klartext zu exportieren.

Ändern, Entsperren und Zurücksetzen der PIN Es ist momentan nicht möglich die PIN nach der Installation zu ändern. Es wäre denkbar dem Applet eine Instruktion hinzuzufügen, mit der die PIN verändert werden kann, nachdem sich der Nutzer mit der aktuell gültigen PIN authentisiert hat. Es ist auch nicht möglich, eine durch mehrfache Fehlversuche gesperrte PIN zu entsperren oder eine vergessene PIN zurückzusetzen. Diese Funktionen könnten unter Verwendung eines PUK (Personal Unblocking Key) hinzugefügt werden, der dann abgefragt wird, wenn die PIN entsperrt oder zurückgesetzt werden soll. Alternativ könnten bei der Installation auch 2 PINs vergeben werden, wobei eine PIN die PUK für die jeweils andere ist.

3 Beschreibung der anderen Softwarekomponente

Das Javacard-Applet benötigt noch ein Gegenstück, also eine Softwarekomponente, die eine öffentliche Schnittstelle bereitstellt und zur Erfüllung dieser Funktionen Instruktionen des Javacard-Applets ruft. Dafür werden Kommando-APDUs konstruiert, an das Applet gesendet und die Antwort-APDUs ausgewertet.

Es existieren Bibliotheken zur Kommunikation mit einer Smartcard für verschiedene Programmiersprachen. Eine Übersicht dazu liefert *PC/SC sample in different languages* [24]. Da ich bereits zur Entwicklung des Javacard-Applets mit Java gearbeitet habe und die `javax.smartcardio`-Bibliothek teil der „Java Runtime Environment“ ist, habe ich mich dazu entschieden, diesen Teil der Software in Java zu schreiben. Es sollte jedoch problemlos möglich sein, ein äquivalentes Programm in einer anderen Programmiersprache zu entwickeln.

In [8] wird ein Schichtenmodell als Architektur des Schlüsselspeichers vorgeschlagen. Diesem Vorschlag bin ich nicht ganz gefolgt. Zwar stellt das Javacard-Applet die Schicht dar, die in [8] als Backendschicht bezeichnet wird. Abstraktions- und Keystoreschicht werden in meiner Implementation aber nicht getrennt, das heißt die in diesem Abschnitt beschriebenen Klassen stellen sowohl die Keystore- als auch die Abstraktionsschicht dar.

Die Unterscheidung von Keystore- und Abstraktionsschicht sollte dafür sorgen, dass die Verwaltung der verpackten Schlüssel vom Rest des Programms getrennt wird. Es hat sich jedoch gezeigt, dass diese Verwaltung nicht sehr aufwendig ist. Statt dieser Trennung habe ich einzelne Funktionen in Klassen ausgelagert wo ich es für sinnvoll hielt. Diese Klassen möchte ich im Folgenden beschreiben.

3.1 Übersicht über die einzelnen Klassen

Keystore: Die Klasse `Keystore` stellt eine öffentliche Schnittstelle bereit, die es ermöglicht KNX-Telegramme zu entschlüsseln und zu verifizieren. Diese Funktionen werden auf der Smartcard ausgeführt. Dazu ruft diese Klasse Instruktionen des Javacard-Applets und wertet dessen Antworten aus. Darüberhinaus werden hier die verpackten Schlüssel verwaltet. Die Klasse enthält außerdem Benchmarks und Tests. Die Tests können ausgeführt werden, indem die `main()`-Funktion dieser Klasse mit Argument `test` gerufen wird. Dabei werden alle öffentlichen Methoden einmal ausgeführt.

KeystoreException: Diese Klasse beschreibt einen eigenen Exception-Typ. Die öffentlichen Methoden der `Keystore`-Klasse werfen im Fehlerfall eine Exception dieses Typs. Eine `KeystoreException` enthält immer einen Fehlercode, welcher den aufgetretenen Fehler genauer beschreibt. Ein Klient, der den Schlüsselspeicher verwendet, kann diese Exceptions fangen und den Fehlercode mit der öffentlichen Methode `KeystoreException.getReason()` abfragen. In Abhängigkeit von

diesem Fehlercode kann der Klient dann unterschiedlich reagieren. Welche Fehlercodes bei welchen Methoden auftreten können, ist in der Javadoc-Dokumentation der `Keystore`-Klasse beschrieben.

KnxTelegram: Diese Klasse beschreibt den Aufbau eines KNX-Telegramms und enthält Methoden, die bei der Entschlüsselung und Verifikation eines KNX-Telegramms benötigt werden.

KnxKey und WrappedKnxKey: Diese Klassen repräsentieren KNX-Laufzeitschlüssel in verpackter und unverpackter Form. Der Unterschied zwischen `KnxKey` und `WrappedKnxKey` liegt in ihrer Länge: Ein `KnxKey` ist 16 Bytes lang, da es sich um ein AES-128-Schlüssel handelt. Ein `WrappedKnxKey` ist 32 Bytes lang, da dies das Kryptogramm eines AES-128-Schlüssels mit 16 Byte MAC ist.

KnxSimulator: Mit dem KNX-Simulator können verschlüsselte und authentische KNX-Telegramme erzeugt werden. Damit wird der Durchsatz des Schlüsselspeichers gemessen.

PinVerifyStructure: Diese Klasse beschreibt die benötigte Datenstruktur, um eine PIN-Eingabe am Kartenleser zu ermöglichen.

Alle diese Klassen sind Teil des Java-Package `de.hu_berlin.keystore`. Die Klassen `Keystore` und `KeystoreException` sind die einzigen öffentlichen Klassen dieses Packages. Alle anderen sind nur innerhalb des Package sichtbar. Klienten müssen zur Verwendung des Schlüsselspeichers also nur die beiden öffentlichen Klassen importieren. Es existiert darüberhinaus noch die Klasse `KeystoreTest`, die keinem Package zugeordnet ist. Diese soll beispielhaft die Nutzung der Bibliothek von außen zeigen. Die einzelnen Klassen und ihre Abhängigkeiten sind in Abbildung 2 dargestellt.

3.2 Öffentliche Schnittstelle des Schlüsselspeichers

In diesem Abschnitt werden die öffentlichen Methoden der Klasse `Keystore` beschrieben, die einem Klienten zur Verfügung stehen, der den Schlüsselspeicher verwenden möchte.

Keystore(): Um den Schlüsselspeicher zu verwenden, müssen Klienten mit diesem Konstruktor ein Objekt vom Typ `Keystore` erstellen. Dabei wird ein Kommunikationskanal zur Smartcard etabliert und das Javacard-Applet selektiert.

Diese Methode schlägt fehl, falls keine Karte gefunden werden kann oder das Applet auf der Karte nicht installiert ist.

closeCardChannel(): Diese Methode schließt den Kanal zur Smartcard. Nach Aufruf dieser Methode sollten an dem `Keystore`-Objekt keine anderen Methoden mehr gerufen werden. Dies würde einen Laufzeitfehler verursachen.

initialize(): Diese Methode bereitet den Schlüsselspeicher zur Verwendung vor. Dazu gehört das Generieren eines Wrapping-Schlüssels auf der Smartcard, sowie das Erstellen einer Zuordnungstabelle, in der die verpackten Schlüssel gespeichert

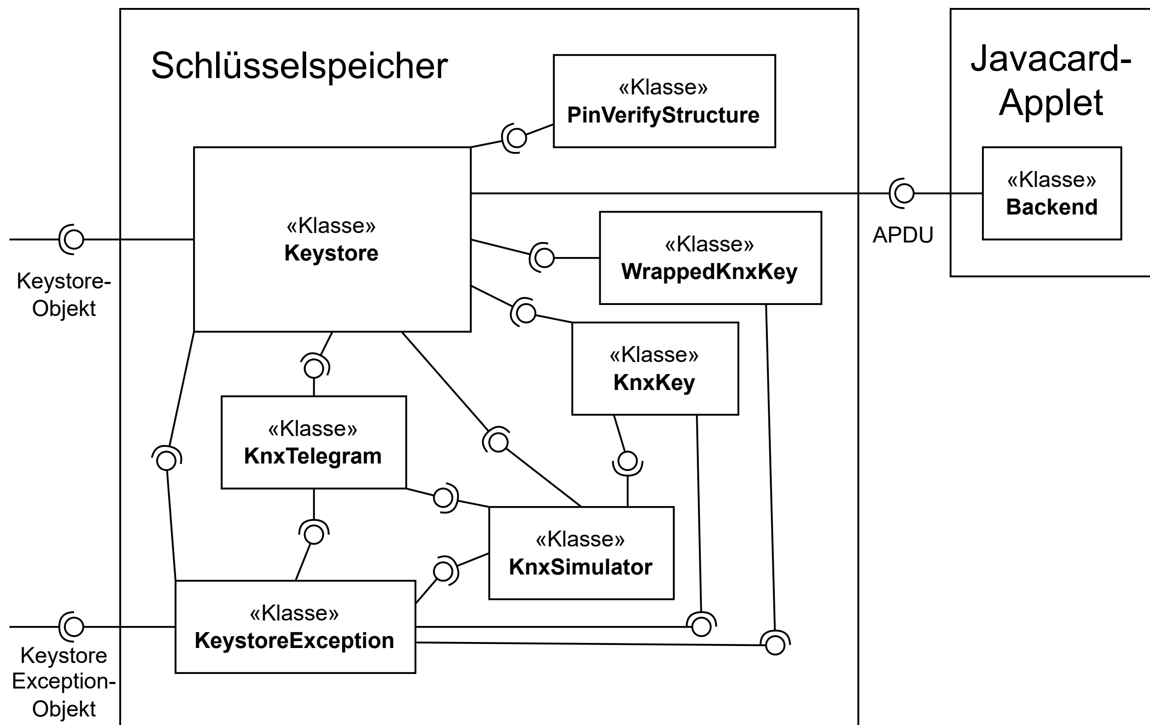

Abbildung 2: Komponentendiagramm, welches alle zum Schlüsselspeicher gehörenden Klassen darstellt. Die Verbindungen zwischen den Klassen stehen für Methodenrufe, d.h. eine Klasse zu der ein Halbkreis gehört ruft eine Methode aus einer Klasse, zu der ein Kreis gehört. Ein Programm welches den Schlüsselspeicher verwenden möchte, muss die Klassen Keystore und KeystoreException importieren.

werden. Sollte eine solche Zuordnungstabelle bereits auf der Festplatte existieren, wird diese nur geladen und kein neuer Wrapping-Schlüssel erstellt. Dies ist zum Beispiel nach einem Neustart des Systems der Fall.

Wenn ein Wrapping-Schlüssel generiert werden muss, schlägt diese Methode fehl, falls eine falsche PIN eingegeben wurde, eine kontaktlose Schnittstelle verwendet wird oder der Kartenleser eine PIN-Eingabe nicht unterstützt.

terminate(): Löscht die Zuordnungstabelle mit den verpackten Schlüsseln und entfernt den Wrapping-Schlüssel von der Smartcard. Diese Methode schlägt in den gleichen Fällen fehl, wie die `initialize()`-Methode.

installKey(): Nimmt einen KNX-Laufzeitschlüssel und dessen Schlüssel-ID, verpackt ihn und fügt den verpackten Schlüssel mit seiner ID in die Zuordnungstabelle ein. Bei dieser und allen anderen Methoden, die einen Schlüssel ent- oder verpacken, tritt ein Fehler auf, wenn kein Wrapping-Schlüssel auf der Smartcard existiert. Außerdem findet eine Längenprüfung des übergebenen Schlüssels statt.

removeKey(): Nimmt eine Schlüssel-ID und entfernt den zugehörigen Schlüssel aus

der Zuordnungstabelle. Tut nichts, falls kein Schlüssel mit der übergebenen ID existiert.

decryptAndVerify1() / decryptAndVerify2(): Diese Methoden nehmen ein verschlüsseltes und authentisiertes KNX-Telegramm, entschlüsseln dieses, prüfen den MAC, und geben im Erfolgsfall die im Telegramm enthaltenen Nutzdaten im Klartext zurück.

Die beiden Methoden unterscheiden sich nicht in ihrem Ergebnis, sondern nur in ihrer Implementation. Zunächst wird der benötigte KNX-Laufzeitschlüssel entpackt. Der Unterschied liegt nun darin, welche Instruktionen des Javacard-Applets verwendet werden. Die Methode `decryptAndVerify1()` ruft die Instruktion `INS_DECRYPT_AND_VERIFY`. Die Methode `decryptAndVerify2()` hingegen verwendet nacheinander die Instruktionen `INS_DECRYPT` und `INS_VERIFY`.

Es tritt ein Fehler auf, falls die MAC-Prüfung nicht erfolgreich war oder der passende KNX-Laufzeitschlüssel nicht in der Zuordnungstabelle gefunden werden konnte.

verify1() / verify2(): Diese Methoden nehmen ein KNX-Telegramm, welches nur authentisiert aber nicht verschlüsselt wurde, und prüft dessen MAC. Der Unterschied zwischen den beiden ist analog zu den `decryptAndVerify`-Methoden.

3.3 KNX-Telegramm

3.3.1 Rahmen-Format eines KNX-Telegramms

Die Klasse `KnxTelegram` beschreibt den Aufbau eines KNX-Telegramms. Es gibt verschiedene Rahmen-Formate für KNX-Telegramme. Ein typisches KNX-Datenpaket hat das `L_Data_Frame`-Format, dessen Aufbau in [23, 3] beschrieben ist. Beim Einsatz eines KNX nach IP-Gateways wird jedoch das `CommonEMI`-Format verwendet. [3] Der Unterschied der beiden Formate liegt darin, dass es nicht nur *ein* Kontrollbyte vor der Quelladresse gibt, sondern dort stattdessen vier Bytes stehen, wovon eines davon das Kontrollbyte ist. [5] In unserer „KNX Data Secure“-Installation wurden KNX-Telegramme im `CommonEMI`-Format verwendet. Darum verwendet dieser Schlüsselspeicher ausschließlich dieses Format.

Um herauszufinden, welches Rahmen-Format ein KNX-Telegramm hat, kann die ETS verwendet werden. Dazu muss ein Kommunikations-Log aufgezeichnet werden. Alle zur Zeit der Aufzeichnung übermittelten Telegramme werden dabei in einer XML-Datei gespeichert. Jedes Telegramm in dieser XML-Datei sollte das Attribut `FrameFormat` haben. Die in unserem Versuchsaufbau aufgezeichneten Telegramme hatten alle das Rahmen-Format `CommonEmi`.

3.3.2 Schlüssel-ID

Jeder Schlüssel soll zur Identifikation eine Schlüssel-ID bekommen. Das Format dieser Schlüssel-ID soll das Folgende sein:

`Quelladresse.Zieladresse.[0,1]`

Dabei gibt die letzte Ziffer an, ob es sich bei der Zieladresse um eine Gruppenadresse handelt. Diese Information steckt im KNX-Telegramm im vierten Byte, erstes Bit. [1] Um das für ein gegebenes KNX-Telegramm zu ermitteln, steht die Methode `KnxTelegram.getAddressType()` zur Verfügung.

Die Methode `KnxTelegram.getKeyId()` gibt die zu einem KNX-Telegramm passende Schlüssel-ID zurück. Diese Methode wird vom Schlüsselspeicher gerufen, wenn ein KNX-Telegramm entschlüsselt werden muss, um die Schlüssel-ID des Telegramms zu ermitteln. Mithilfe dieser Schlüssel-ID kann der Schlüsselspeicher dann den passenden KNX-Laufzeitschlüssel aus der persistent gespeicherten Zuordnungstabelle laden und an die Smartcard zum Entpacken weitergeben. Nachdem der Schlüssel entpackt wurde, kann der Schlüsselspeicher das verschlüsselte KNX-Telegramm auf die Smartcard übertragen und entschlüsseln lassen.

3.3.3 Zu verifizierende Nachricht

Nachdem ein KNX-Telegramm entschlüsselt wurde, muss dessen Authentizität geprüft werden. Das wird erreicht, indem ein MAC berechnet wird, der mit dem im entschlüsselten KNX-Telegramm enthaltenen MAC verglichen wird. Der MAC wird aber nicht einfach über dem KNX-Telegramm selber berechnet. Stattdessen wird mit Informationen aus dem KNX-Telegramm eine Nachricht konstruiert, über die dann der MAC berechnet wird. Mit dem Wort „Nachricht“ ist in diesem Abschnitt also explizit nicht das KNX-Telegramm gemeint.

Die Methode `KnxTelegram.buildMessage()` konstruiert aus einem KNX-Telegramm eine solche Nachricht. Die Konstruktion ist in [8] und *KNX Data Security: Application Note 158/13 v02* [22] beschrieben. Diese Informationen scheinen jedoch nicht korrekt zu sein. Zur Verifikation unserer KNX-Telegramme musste die Nachricht anders aufgebaut werden. Hinweise dazu habe ich in *KNX Data Secure Implementation (Commit 2aecc21)* [21, siehe insbesondere Zeile 494-506] gefunden.

Die Größe der Nachricht hat immer ganze Blocklänge und ist abhängig von der Länge der Nutzdaten. Im ersten Block B_0 steht zunächst die Sequenznummer, die Quell- und die Zieladresse. Das nächste Byte (in [22] mit FT gekennzeichnet) ist immer 0. Danach kommt der Address-Typ. Dieser ist 0x80, falls die Zieladresse eine Gruppenadresse ist. Andernfalls ist er wahrscheinlich 0, das habe ich aber nicht getestet. Dann folgen zwei Bytes mit TPCI- und Secure-APCI-Informationen. Das vorletzte Byte dieses Blocks ist immer 0. Das letzte Byte ist die Länge der Nutzdaten.

Im nächsten Block steht zunächst die Länge der zugehörigen Daten in Bytes. Das sind

die Daten im KNX-Telegramm, die nicht verschlüsselt, sondern lediglich authentisiert werden müssen. Diese Länge ist immer 1 im Fall eines Daten-Pakets [21, siehe Zeile 947]. Diese Information ist zwei Bytes lang, in Byte-Reihenfolge Big-Endian. Die Daten bestehen dann nur aus dem SCF-Byte (Security Control Field) und kommen direkt nach der Längenangabe.

Das SCF-Byte ist in *KNX Data Security: Application Note 158/13 v02* [22, Seite 15] nicht korrekt beschrieben. Stattdessen ist es folgendermaßen aufgebaut [21, siehe Zeile 788 bis 792]:

- Das erste Bit im SCF-Byte ist das ToolAccess-Bit.
- Die nächsten drei Bits sind SAI-Bits (Security Algorithm Identifier), und nehmen entweder die Werte 000 für nur Authentifizerung, oder 001 für Authentifizierung und Vertraulichkeit an.
- Das fünfte Bit steht für System Broadcast,
- und die letzten drei Bits sind immer Null bei Datenpaketen.

Das SCF-Byte sollte bei Datenpaketen also entweder den Wert 0x10 oder 0x00 annehmen. Bei unseren aufgezeichneten KNX-Telegrammen, deren Nutzdaten verschlüsselt wurden, hatte das SCF-Byte immer den Wert 0x10.

Neben Datenpaketen (`S-A_Data-PDU`) scheint es noch die Typen `S-A_Sync_Req-PDU` und `S-A_Sync_Res-PDU` zu geben. Für diese Art von Telegrammen ist die Länge der zugehörigen Daten gleich sieben [21, siehe Zeile 947]. Außerdem nehmen die letzten drei Bits des SCF-Feldes bei diesen Telegrammen die Werte 010 bzw. 011 an [21, siehe Zeile 792]. Diese Typen werden jedoch von dieser Applikation nicht unterstützt.

Im Gegensatz zu den Beschreibungen in [22, Seite 91] und [8, Seite 10], werden die zugehörigen Daten jetzt nicht durch Nullpadding auf ganze Blocklänge gebracht. Stattdessen werden die Nutzdaten im Klartext direkt an die zugehörigen Daten angehängt. Erst danach wird die Nachricht durch Nullpadding auf ganze Blocklänge aufgefüllt.

3.4 PIN-Eingabe an einem Kartenleser

Das Generieren bzw. Löschen des Wrapping-Schlüssels setzt eine Authentifikation mittels PIN voraus. Dabei darf die PIN-Eingabe nur auf der Tastatur eines Kartenlesegeräts erfolgen. Die Funktion `Keystore.insAuthenticate()` aktiviert die PIN-Eingabe an einem Kartenleser. In diesem Abschnitt wird beschrieben, wie dieser Vorgang funktioniert.

In der Klasse `Keystore` gibt es private Funktionen, deren Namen mit dem Präfix `ins` beginnen. Diese Funktionen kommunizieren mit dem Javacard-Applet, indem sie APDUs konstruieren und an die Karte senden und damit Instruktionen der Javacard rufen. Dafür wird, wie bereits erwähnt, die SmartcardIO-Bibliothek verwendet, die Teil der Java-Standardbibliothek ist. Diese stellt die Methode `CardChannel.transmit()`

zur Verfügung. Diese nimmt ein Kommando-APDU als Argument, sendet es an die Smartcard und gibt die Antwort-APDU der Smartcard als Rückgabewert zurück. Die mit `ins` beginnenden Funktionen erstellen also eine Kommando-APDU, rufen dann die Methode `CardChannel.transmit()` und kehren dann zurück bzw. geben im Erfolgsfall die in der Antwort-APDU enthaltenen Daten zurück.

So funktioniert der Aufruf aller Javacard-Instruktionen, außer der für die Instruktion `INS_AUTHENTICATE`. Das Ziel hier ist es nämlich, dass die PIN vom Benutzer am Kartenleser eingegeben wird. Das heißt, unser Programm kann die APDU nicht konstruieren, da es die PIN, die der Nutzer eingeben wird, nicht kennt. In Folge dessen muss der Kartenleser die APDU konstruieren und an unser Applet weitergeben.

Um dieses Verhalten zu aktivieren, müssen wir ein Kontroll-Kommando an den Kartenleser senden. Dafür verwenden wir die Methode `Card.transmitControlCommand()`, die ebenfalls von der SmartcardIO-Bibliothek zur Verfügung gestellt wird. Diese Methode nimmt zwei Argumente: Einen Kontroll-Code und ein Kommando.

3.4.1 Kontroll-Code

Der Kontroll-Code für die PIN-Eingabe an einem „Reiner SCT cyberJack RFID standard“ lautet `0x42000db2`. Bei einem anderen Kartenleser könnte es aber ein anderer sein. In diesem Abschnitt soll erklärt werden, wie man herausfindet, wie der Kontroll-Code für die PIN-Eingabe lautet.

In *PCSC-Spezifikation, Teil 10* [25] ist ein Kontroll-Code festgelegt, der an einen Kartenleser als Kontroll-Kommando gesendet werden kann, der alle von diesem Kartenleser unterstützten Kontroll-Codes zurückgibt. Dieser Kontroll-Code lautet `0x42000d48` (= `0x42000000 + 3400`).

Zurück kommt ein Array von Bytes, dessen Länge ein Vielfaches von sechs ist. Ein Block von sechs Bytes ist wie folgt aufgebaut: Das erste Byte gibt an, um welches Kommando es sich handelt. Die Kommandos sind in *PCSC-Spezifikation, Teil 10* [25] festgelegt und werden dort „Features“ genannt. Wir suchen das Kommando `FEATURE_VERIFY_PIN_DIRECT`, welches durch Byte `0x06` gekennzeichnet ist. Das zweite Byte gibt die Länge des Kontroll-Codes an, dieses ist immer `0x04`. Die nächsten vier Bytes geben dann den Kontroll-Code an, der an den Kartenleser gesendet werden muss, um das entsprechende Kommando auszuführen.

Nachdem wir also das Kommando an unseren Kartenleser gesendet haben, suchen wir in der Antwort nach dem Byte `0x06` (aber nur in jedem sechsten Byte, welche die Kommandos angeben, nicht in den Bytes die die Kontroll-Codes angeben). Sollte dies nicht vorhanden sein, wird das Kommando vom Kartenleser nicht unterstützt. Andernfalls können wir in den nachfolgenden Bytes den Kontroll-Code ablesen. Dieses Verfahren habe ich in der Funktion `Keystore.getVerifyPinControlCode()` implementiert.

3.4.2 Kommando

Nachdem wir den Kontroll-Code herausgefunden haben, müssen wir das Kommando konstruieren. In *PCSC-Spezifikation, Teil 10* [25] ist die Datenstruktur angegeben, die vom `FEATURE_VERIFY_PIN_DIRECT` erwartet wird. Ich habe die Java Klasse `PinVerifyStructure` geschrieben, mit dessen Hilfe sich diese Datenstruktur leicht konstruieren lässt. Der Quellcode dieser Klasse enthält viele Kommentare, die den Aufbau der Datenstruktur und der APDU beschreiben und die Benutzung der Klasse an einem Beispiel demonstrieren.

4 Laufzeittests

Der KNX-Bus nach Spezifikation TP-1 hat eine Geschwindigkeit von 9600 Baud [23, 3], also 9600 Bits pro Sekunde. „KNX Data Secure“-Telegramme im `CommonEmi`-Format haben eine Länge von (18 + Länge der Nutzdaten + 4) Bytes. Bei unseren aufgezeichneten KNX-Telegrammen, deren Nutzdaten zwei Bytes lang sind, beträgt die Länge somit 24 Bytes. Jedem Byte, welches über den KNX-Bus übertragen wird, werden drei Prüfbits angehängt. [29] Damit wächst ein 24 Byte langes Telegramm von $24 \cdot 8 = 192$ Bits auf $24 \cdot 11 = 264$ Bits an. Bei 9600 Bits pro Sekunde können dann also maximal 36 Telegramme pro Sekunde übertragen werden.

Um zu testen wieviele Telegramme das Javacard-Applet pro Sekunde entschlüsseln kann, habe ich Benchmark-Funktionen geschrieben. Diese befinden sich in der Klasse `Keystore` und lassen sich ausführen, indem die `main()`-Funktion dieser Klasse mit Argument `bench` und einer Anzahl Ausführungen gerufen wird. Die Telegramme, die während der Benchmarks entschlüsselt werden, werden von der Klasse `KnxSimulator` erzeugt. In den folgenden Tests habe ich immer Telegramme mit 5 Bytes Nutzdaten verwendet.

Alle folgenden Messungen wurden mit dem gleichen Rechner durchgeführt. Das System besitzt einen Intel-Prozessor vom Typ i7-980X und 24 GB Hauptspeicher. Die Varianz der Ergebnisse mehrfacher Ausführungen ist vernachlässigbar klein. Die Werte sind gerundet. Auf einem anderen System mit einem Intel-Prozessor vom Typ i5-2500 und 16 GB Hauptspeicher unterscheiden sich die Werte hingegen stark von den hier angegebenen. Mir ist die Ursache für diese Laufzeitunterschiede nicht bekannt. In dem in [8] beschriebenem Konzept soll der Schlüsselspeicher zusammen mit dem IDS auf einem Raspberry Pi laufen. Es ist nicht klar, ob der hier gemessene Durchsatz auch auf einem Raspberry Pi erreicht werden kann. Als Kartenleser wurde ein SCL011 von SCM Microsystems verwendet.

4.1 Entschlüsselung und Verifikation als getrennte Instruktionen

Wie in Abschnitt 1.2 bereits beschrieben, wurde die Entschlüsselung von KNX-Telegrammen auf zwei verschiedene Arten implementiert. In diesem Abschnitt soll der Durchsatz gemessen werden, wenn zur Entschlüsselung und Verifikation zwei verschiedene Smartcard-Instruktionen gerufen werden. Das ist das in [8] beschriebene Verhalten.

In einem ersten Test habe ich den Laufzeitschlüssel für jedes Telegramm neu entpackt. Das ist das normale Verhalten des Applets. Dieser Benchmark ist implementiert in der Methode `benchmark2Unwrap()`.

Anzahl der Telegramme:	1000
Vergangene Zeit in Sekunden:	60
Telegramme pro Sekunde:	16,7

In diesem Fall scheint das Javacard-Applet nicht auszureichen, um alle KNX-Telegramme

bei einem voll ausgelasteten KNX-Bus zu entschlüsseln.

Im zweiten Test wurde der benötigte KNX-Laufzeitschlüssel einmal gesetzt und für alle KNX-Telegramme verwendet. Das heißt das Entpacken des Schlüssels musste nicht durchgeführt werden. Dieser Benchmark ist implementiert in der Methode `benchmark2NoUnwrap()`.

Anzahl der Telegramme:	1000
Vergangene Zeit in Sekunden:	30
Telegramme pro Sekunde:	33,3

Die vergangene Zeit hat sich ungefähr halbiert. Daraus lässt sich schließen, dass das Entpacken eines Schlüssels ungefähr so viel Zeit in Anspruch nimmt, wie das Entschlüsseln und Verifizieren eines KNX-Telegramms. Das ist auch plausibel: Beim Entpacken eines Schlüssels wird zunächst der Chiffretext entschlüsselt und dann der MAC verifiziert. So ähnlich funktioniert es auch beim KNX-Telegramm, nur dass hier eine andere Schlüssellänge und ein anderer Betriebsmodus zum Einsatz kommen.

In einem dritten Test habe ich zusätzlich noch auf die Verifikation des MAC verzichtet, um das Verhältnis der Laufzeiten der Entschlüsselungs- und der Verifikationsoperation zu ermitteln. Dieser Benchmark ist implementiert in der Methode `benchmark2NoUnwrapNoVerify()`.

Anzahl der Telegramme:	1000
Vergangene Zeit in Sekunden:	15
Telegramme pro Sekunde:	66,7

Die vergangene Zeit hat sich nocheinmal halbiert. d.h. das Entschlüsseln dauert genauso lange wie das Verifizieren. Auch das ist plausibel: Die Berechnung des MAC ist eigentlich nur eine Verschlüsselung mit AES-CBC.

4.2 verschiedene Laufzeittests

Im Folgenden habe ich dann nicht mehr mit den Benchmark-Funktionen getestet, sondern das Javacard-Applet mit *GlobalPlatformPro* [13] direkt angesprochen. Mit

```
java -jar gp.jar --debug --applet 0102030405A1 --apdu 0010000002010202
```

wird das Applet selektiert, die APDU an das Applet gesendet und auf der Standardausgabe eine Ablaufverfolgung der gesendeten und empfangenen APDUs ausgegeben, inklusive der Dauer der Verarbeitung für jede APDU in Millisekunden. Damit wollte ich prüfen, ob die Verwendung der SmartcardIO-Bibliothek messbare Laufzeitkosten verursacht. Dafür habe ich zunächst einen Block mit AES-CTR entschlüsselt. Dies dauerte 15 ms, also genauso lang wie unter Verwendung der SmartcardIO-Bibliothek. Das lässt vermuten, dass es keinen Unterschied in der Laufzeit machen sollte, ob APDUs über das Programm oder über *GlobalPlatformPro* [13] direkt gesendet werden.

Danach habe ich die Nachrichtenlänge immer um einen Block erhöht. Die Dauer der

Entschlüsselung erhöhte sich dabei um ungefähr eine Millisekunde pro weiterem Block. Daraus schließe ich, dass nicht die AES-Operationen den Großteil der Zeit ausmachen.

Das Entpacken eines Schlüssels hat im Test nur 24 ms, statt wie erwartet 30 ms gedauert.

Außerdem wollte ich herausfinden, ob die Laufzeit abhängig vom verwendeten Kartenleser ist. Dafür habe ich ein Echo-Applet geschrieben, welches nichts anderes macht als jede APDU sofort wieder zurückzusenden. Als Kartenleser habe ich zum einen den SCL011 von SCM Microsystems verwendet, welcher nur eine kontaktlose Schnittstelle anbietet. Zum anderen habe ich den „cyberJack RFID standard“ von ReinerSCT getestet, welcher sowohl eine kontaktlose, als auch eine kontaktbehaftete Schnittstelle besitzt.

Dem Echo-Applet habe ich APDUs zugeführt, die jeweils die in der linken Spalte der nachfolgenden Tabelle stehende Anzahl Bytes Nutzdaten hatten, zusätzlich zu den benötigten Header-Bytes. Die Zeiten sind jeweils in Millisekunden angegeben.

Daten in Bytes	SCL011	RFID standard kontaktbeh.	RFID standard kontaktlos
0	5	5	10
$4 \cdot 16$	10	14	12
$8 \cdot 16$	15	22	13

Diese Ergebnisse sind auch in Abbildung 3 dargestellt.

Die kontaktbehaftete Schnittstelle ist deutlich langsamer als die kontaktlose. Daher bietet es sich an, während der Betriebsphase des Applets eine kontaktlose Schnittstelle zu verwenden. Es gibt signifikante Laufzeitunterschiede zwischen den beiden getesteten kontaktlosen Kartenlesern. Dennoch ist keiner der beiden in allen getesteten Datenmengen dem anderen überlegen. Vielleicht lässt sich der Durchsatz allein durch die Wahl eines geeigneten Kartenlesers noch steigern.

Ich wollte noch wissen, welchen Einfluss die Länge der APDU auf die Laufzeit hat. Dafür habe ich den SCL011-Kartenleser verwendet. Für diesen Test habe ich zunächst ein leeres Applet geschrieben, ähnlich dem in Abschnitt 2.1 gezeigten. Dieses nimmt eine APDU entgegen, aber tut nichts damit und sendet nur Statuswort 0x9000 zurück.

Daten in Bytes	Zeit in ms
0	5
$4 \cdot 16$	8
$8 \cdot 16$	10

Das Hinzufügen des Aufrufs der Methode `setOutgoingAndSend()` der Klasse APDU sorgt dafür, dass die gesendeten APDUs nun wieder zurückgesendet werden, d.h. das Verhalten des Applets gleicht nun dem des Echo-Applets. Hier habe ich dann auch die gleichen Laufzeiten erhalten.

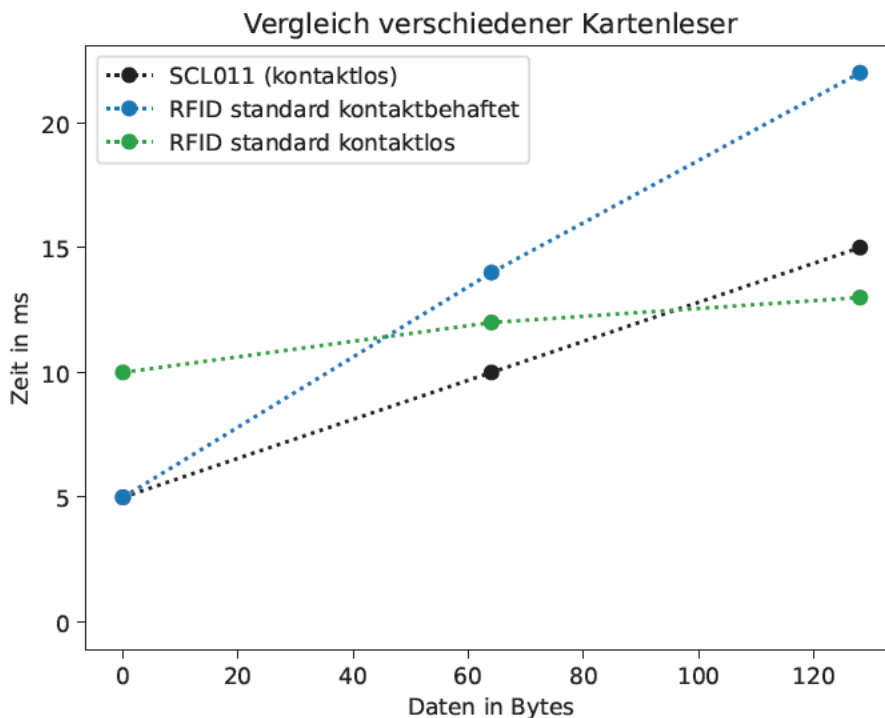

Abbildung 3: Laufzeiten eines Javacard-Applets unter Verwendung verschiedener Kartenleser. Das Applet nimmt eine beliebige Kommando-APDU entgegen und sendet diese sofort wieder als Antwort-APDU zurück. Die kontaktbehaftete Schnittstelle (blau) ist deutlich langsamer als die beiden kontaktlosen. Die Laufzeiten der beiden kontaktlosen Kartenleser unterscheiden sich ebenfalls deutlich voneinander, jedoch ist keine der beiden der jeweils anderen bei allen Datenmengen überlegen.

Daten in Bytes	Zeit in ms
0	5
$4 \cdot 16$	10
$8 \cdot 16$	15

Aus diesen Ergebnissen schließe ich, dass die Laufzeit zu einem großen Teil aus dem Senden und Empfangen von Daten besteht und die kryptografischen Operationen auf der Karte vergleichsweise schnell durchgeführt werden.

4.3 Entschlüsselung und Verifikation in einer Instruktion

Die im vorigen Abschnitt gewonnen Erkenntnisse legen nahe, dass sich der Durchsatz des Schlüsselspeichers erhöhen könnte, wenn die Anzahl an gerufenen Smartcard-Instruktionen reduziert wird. Das Applet ist so konstruiert, dass es drei Instruktionen

braucht, um ein KNX-Telegramm zu entschlüsseln:

1. Entpacken des Schlüssels (`INS_UNWRAP`)
2. Entschlüsselung (`INS_DECRYPT`)
3. MAC-Verifikation (`INS_VERIFY`)

Ich habe eine Instruktion geschrieben, die entschlüsselt *und* verifiziert. Dann werden nurnoch zwei Instruktionen benötigt, um ein KNX-Telegramm zu entschlüsseln:

1. Entpacken des Schlüssels (`INS_UNWRAP`)
2. Entschlüsselung und MAC-Verifikation (`INS_DECRYPT_AND_VERIFY`)

Dann habe ich die Methoden `benchmark1Unwrap()` und `benchmark1NoUnwrap()` geschrieben, die diese Instruktion verwenden und erneut Laufzeittest durchgeführt. Dabei wurden wieder KNX-Telegramme mit Nutzdatenlänge 5 Bytes verwendet, die von der Klasse `KnxSimulator` generiert wurden.

Durchsatz, wenn der Schlüssel bei jedem Telegramm neu entpackt werden muss:

Anzahl der Telegramme:	1000
Vergangene Zeit in Sekunden:	54
Telegramme pro Sekunde:	18,5

Durchsatz, wenn der Schlüssel nie entpackt werden muss:

Anzahl der Telegramme:	1000
Vergangene Zeit in Sekunden:	25
Telegramme pro Sekunde:	40

4.4 Vergleich

Wenn der KNX-Laufzeitschlüssel für jedes Paket entpackt werden muss, dauert die Entschlüsselung eines KNX-Telegramms mit 5 Byte Nutzdaten unter Verwendung getrennter Instruktionen ungefähr 60 ms (siehe auch Abbildung 4). Unter Verwendung einer einzelnen Instruktion verringert sich diese Zeit um 5 ms auf ungefähr 55 ms. Statt 16 Telegramme pro Sekunde, können so etwa 18 Telegramme pro Sekunde entschlüsselt werden. In Abschnitt 4.2 wurde gezeigt, dass die Ausführung einer leeren Instruktion mindestens 5 ms dauert. (Unter Verwendung dieses Kartenlesers.) Insofern ist es nachvollziehbar, dass die Reduktion der Anzahl ausgeführter Instruktionen um 1 einen Zeitgewinn von 5 ms mit sich bringt.

Neben der Ausführungsgeschwindigkeit könnte ein weiterer Vorteil sein, dass es unter Verwendung einer einzelnen Instruktion nicht möglich ist, beliebige Chiffretexte zu entschlüsseln. Die Instruktion `INS_DECRYPT` nimmt einen IV und einen beliebigen Chiffretext und entschlüsselt diesen mit dem momentan geladenen Schlüssel. Die `INS_DECRYPT_AND_VERIFY`-Instruktion gibt jedoch nur einen Klartext zurück, wenn

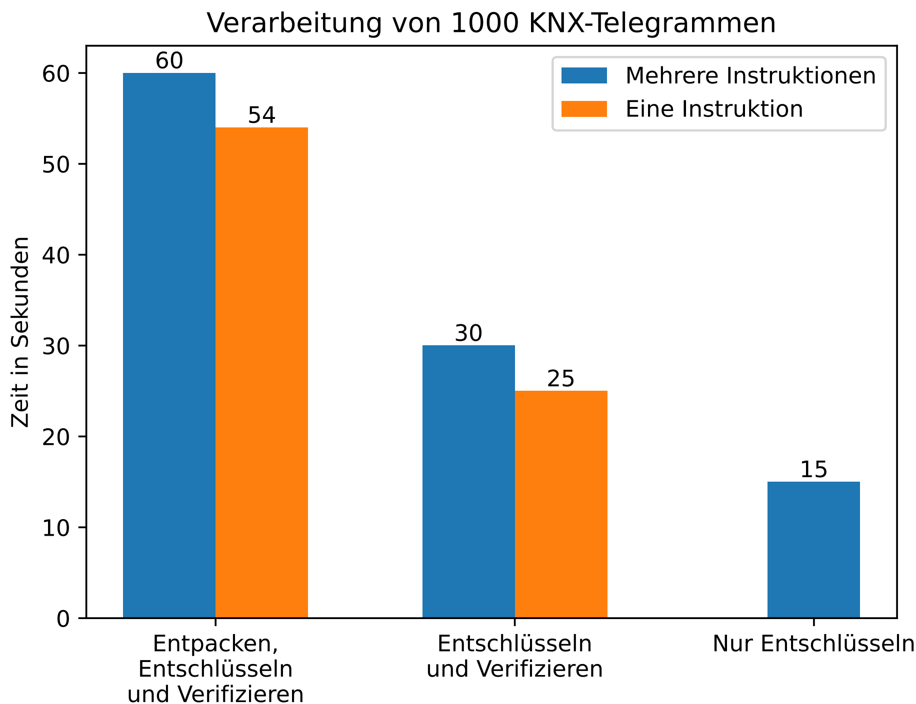

Abbildung 4: Grafische Darstellung der Laufzeiten des Schlüsselspeichers beim Entschlüsseln von KNX-Telegrammen. In Blau dargestellt sind die Ergebnisse aus Abschnitt 4.1, d.h. zum Entschlüsseln und Verifizieren werden zwei verschiedene Smartcard-Instruktionen verwendet. In Orange dargestellt sind die Ergebnisse aus Abschnitt 4.3, d.h. es wird lediglich eine Instruktion zum Entschlüsseln und Verifizieren verwendet. Das Einsparen einer Instruktion führt zu einer Verringerung der Laufzeit um ungefähr fünf Millisekunden.

auch die MAC-Prüfung erfolgreich war. Da diese KNX-spezifisch ist (siehe Abschnitt 3.3.3), können also mit dieser Instruktion nur KNX-Telegramme entschlüsselt werden.

Es könnte ein Nachteil sein, dass der IV und die zu verifizierende Nachricht nun auf der Javacard konstruiert werden müssen. Dabei werden zwar keine laufzeitintensiven Operationen ausgeführt, aber es werden Byte-Arrays als Puffer benötigt. Bei den getrennten Instruktionen reicht der APDU-Puffer aus.

4.5 Weitere Ideen zur Verbesserung der Performance

Mehrfachverwendung des gleichen Schlüssels Wenn hintereinander mehrere Telegramme entschlüsselt werden müssen, die mit dem gleichen Schlüssel verschlüsselt wurden, ist es unnötig, den KNX-Laufzeitschlüssel für jedes einzelne Telegramm neu zu

entpacken. Der letzte verwendete Schlüssel bleibt solange auf der Javacard gespeichert, bis das Applet deselektiert wird. (Oder die Karte aus dem Leser entfernt wird.) Da das Applet im Einsatz nicht deselektiert werden sollte, könnte in diesem Fall also auf die Instruktion zum Entpacken verzichtet werden. Wie in diesem Abschnitt gesehen, macht das Entpacken des Schlüssels die Hälfte der Laufzeit aus. Insofern wäre das eine sinnvolle Methode zur Verbesserung der Laufzeit.

Ein einfacher Weg dieses Verhalten zu implementieren ist es, sich immer die Schlüssel-ID des zuletzt verwendeten Schlüssels zu merken und mit der aktuellen Schlüssel-ID zu vergleichen. Sollten beide übereinstimmen, kann auf die Operationen zum Entpacken des Schlüssels verzichtet werden. Zusätzlich dazu könnte eine Vorsortierung der KNX-Telegramme vorgenommen werden, sodass Pakete mit gleichem Schlüssel hintereinander kommen und damit der Schlüssel so selten wie möglich entpackt werden muss. Diese Vorsortierung könnte bereits im IDS vorgenommen werden.

Speicherung mehrerer Schlüssel auf der Smartcard Aufgrund der geringen Speicherkapazität einer Smartcard können nicht einfach alle KNX-Laufzeitschlüssel auf dieser dauerhaft gespeichert werden. Deswegen werden die Schlüssel verpackt und auf einen externen Speicher geschrieben. In meiner Implementation befindet sich immer nur ein KNX-Laufzeitschlüssel gleichzeitig auf der Karte. Es ist jedoch denkbar, mehrere Schlüssel auf der Karte zu halten. Das könnten zum Beispiel die n zuletzt verwendeten oder die am häufigsten verwendeten sein. Auch so könnte die Anzahl der Operationen zum Entpacken reduziert werden.

Zusammenfassung der Entpack-, Entschlüsselungs- und Verifikations-Instruktionen Wie in diesem Abschnitt gesehen, kann die Laufzeit reduziert werden, indem die Anzahl der gerufenen Instruktionen reduziert wird. Das habe ich beispielhaft an der Zusammenfassung der Entschlüsselungs- und Verifikations-Instruktionen gezeigt. Eventuell kann es eine gute Idee sein, auch auf die Instruktion zum Entpacken eines Schlüssels zu verzichten und stattdessen eine `INS_UNWRAP_DECRYPT_AND_VERIFY`-Instruktion zu verwenden, wobei der Javacard dann der Schlüssel und das KNX-Telegramm übergeben wird. Dabei ist eine Reduktion der Entschlüsselungsdauer eines KNX-Telegramms um weitere 5 ms zu erwarten. Ein Nachteil dabei könnte sein, dass dann die Logik, die entscheidet ob der Schlüssel tatsächlich entpackt werden muss, im Javacard-Applet implementiert werden muss.

Mehrere Smartcards im Parallelbetrieb Vielleicht lässt sich die Applikation dahingehend erweitern, dass mehrere Smartcards unterstützt werden. Man könnte während der Initialisierungsphase entweder einen Wrapping-Schlüssel generieren, der dann auf alle Karten übertragen wird. Das setzt dann eine weitere Instruktion im Javacard-Applet voraus, die den Import eines Wrapping-Schlüssels erlaubt. Oder es wird auf allen Karten

ein eigener Wrapping-Schlüssel generiert und alle KNX-Laufzeitschlüssel müssen mit jedem Wrapping-Schlüssel verpackt werden.

Wie in diesem Abschnitt gesehen, kann eine Smartcard 16 Telegramme pro Sekunde entschlüsseln, selbst wenn jedes Mal der KNX-Schlüssel entpackt werden muss. Außerdem wurde festgestellt, dass maximal 36 Pakete pro Sekunde übertragen werden können. Das bedeutet, dass drei Smartcards ausreichen sollten, um auf einem ausgelasteten KNX-Bus alle Telegramme in Echtzeit zu entschlüsseln und verifizieren.

5 Abschluss

5.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde eine Programmbibliothek entwickelt, die das in [8] beschriebene Konzept eines sicheren KNX-Schlüsselspeichers umsetzt. Von besonderer Bedeutung war dabei ein Javacard-Applet, also ein Programm welches auf einer frei programmierbaren Smartcard läuft und dem Schlüsselspeicher kryptografische Operationen zur Verfügung stellt, die auf einer Smartcard ausgeführt werden.

In dieser Arbeit wurden allgemeine Hinweise zur Entwicklung eines Javacard-Applets gegeben und Referenzen genannt, die den Einstieg in die Javacard-Programmierung erleichern. Es wurde erklärt, wie das entwickelte Applet auf einer Javacard installiert werden kann und welche Fehler bei der Installation auftreten können. Es existiert eine Übersicht über alle Instruktionen, die als Beschreibung der Schnittstelle zwischen Javacard-Applet und Programmbibliothek dienen soll. Das erleichtert eine Reimplementation der Programmbibliothek, beispielsweise in einer anderen Programmiersprache. Außerdem wird das Applet im Detail beschrieben, was bei einer zukünftigen Modifikation des Applets behilflich sein kann.

Bei der Beschreibung der Programmbibliothek stand die öffentliche Schnittstelle im Vordergrund, sodass es nicht schwer sein sollte, einen Klienten zu programmieren, der diese Bibliothek verwendet. Es wurde auch ein Klient implementiert, der beispielhaft die öffentlichen Methoden ruft und die Fehlerbehandlung demonstriert. Detaillierter waren die Beschreibung eines KNX-Telegramms und wie die PIN-Eingabe an einem Kartenleser funktioniert, da diese Informationen eventuell auch für andere Projekte, die nicht unmittelbar mit dem Schlüsselspeicher zu tun haben, von Interesse sein könnte.

Um den Schlüsselspeicher zu testen wurde eine reale „KNX Data Secure“-Installation aufgebaut. Diese Arbeit beschreibt, wie die KNX-Laufzeitschlüssel aus einer solchen Installation extrahiert werden können. Dieses Verfahren wurde auch als Programm implementiert. Die extrahierten Schlüssel wurden in den Schlüsselspeicher geladen um dessen Korrektheit mit echten KNX-Telegrammen zu testen. Außerdem wurde durch Laufzeittests ermittelt, wieviele KNX-Telegramme pro Sekunde entschlüsselt werden können. Zusätzlich dazu wurden verschiedene andere Laufzeittests mit der Javacard durchgeführt, um einen Hinweis darauf zu erhalten, welche Smartcard-Operationen bezüglich der Laufzeit besonders ins Gewicht fallen. Außerdem wurden einige Ideen gesammelt, wie der Durchsatz des Schlüsselspeichers noch weiter gesteigert werden kann.

Zu guter Letzt beschreibt die Arbeit noch, wie eine gegenseitige Authentisierung zwischen dem Javacard-Applet und der Programmbibliothek hergestellt werden kann. Siehe dazu den folgenden Abschnitt, in dem Ansätze genannt werden, wie der Schlüsselspeicher noch verbessert werden kann.

5.2 Ausblick

Secure Messaging Das Javacard-Applet nimmt Kommando-APDUs von beliebeiger Quelle entgegen, das heißt es wird nicht geprüft, ob eingehende APDUs tatsächlich vom Schlüsselspeicher kommen. Umgekehrt prüft der Schlüsselspeicher nicht, ob Antwort-APDUs tatsächlich von unserem Javacard-Applet stammen. Außerdem sind APDUs weder vertraulich noch authentisch, das heißt sie können mitgelesen oder unbemerkt verändert werden.

Das hat zur Folge, dass ein Angreifer möglicherweise Zugriff auf die Entschlüsselungs- und Verifikationsfunktionen der Smartcard erlangen kann. Dabei würde der zuletzt verwendete, noch auf der Smartcard befindliche, KNX-Laufzeitschlüssel verwendet werden.

Das „Secure Channel Protokoll“ ermöglicht gegenseitige Authentisierung, Integrität, sowie Vertraulichkeit von APDUs. Der Schlüsselspeicher implementiert keine dieser Möglichkeiten, aber ich habe ein Javacard-Applet und eine dazu passende Java-Anwendung geschrieben, die sich zumindest gegenseitig authentisieren. Das ist in Abschnitt 6.1 beschrieben. Ich gehe davon aus, dass sich das dort Erreichte auch auf den Schlüsselspeicher anwenden lässt.

Es ist mir aus Zeitgründen nicht gelungen, Integrität und Vertraulichkeit der ausgetauschten Nachrichten sicherzustellen. Hinweise dazu finden sich in den gleichen Dokumenten, die ich schon für die gegenseitige Authentisierung verwendet habe. [12, 9, 10]

GlobalPlatform Schlüssel Um auf der Javacard ein Applet zu installieren oder zu deinstallieren wird ein Schlüssel benötigt. Der Standard-Wert für diesen Schlüssel lautet `404142434445464748494A4B4C4D4E4F`. Um zu verhindern, dass ein Angreifer mit physischem Zugriff auf die Karte Veränderungen an dieser vornimmt, indem er beispielsweise unser Javacard-Applet durch ein eigenes ersetzt, kann ein anderer Schlüssel gesetzt werden. Dafür gibt es im Programm *GlobalPlatformPro* [13] die Option `--lock`. Das habe ich aber nicht getestet. Außerdem ist unklar was passiert, wenn wiederholt versucht wird mit einem falschen Schlüssel auf die Karte zuzugreifen.

Verändern oder Löschen der Laufzeitschlüssel Es gibt keinen Schutz vor Manipulation der gespeicherten verpackten KNX-Laufzeitschlüssel, da diese einfach im Dateisystem des Schlüsselspeichers liegen. Mit modifizierten Schlüsseln sollte es nicht möglich sein, KNX-Telegramme zu entschlüsseln, da die MAC-Prüfung in einem solchen Fall fehlschlagen würde. Dennoch wäre die Verfügbarkeit des Schlüsselspeichers dadurch eingeschränkt.

Maximale Länge eines „KNX Data Secure“-Telegramms Es nicht nicht klar, wie lang ein verschlüsseltes KNX-Telegramm maximal sein kann. In [6] ist angegeben,

dass ein Standard-Frame acht bis 23 Bytes und ein Extended-Frame neun bis 263 Bytes lang sein kann. Jedoch ist nicht klar, ob diese Angabe genauso für „KNX Data Secure“ gilt. Da die Länge der Daten, die eine APDU enthält, durch nur ein Byte angegeben wird, kann ein 263 Byte langes Telegramm nicht mit einer Standard-APDU übertragen werden. In diesem Fall müsste das Javacard-Applet modifiziert werden, sodass Extended-APDUs unterstützt werden. Informationen dazu können gefunden werden in *Java Card 3 Platform Programming Notes* [16].

Implementation der installKey()-Methode Die Implementation der Methode `installKey()` ist nicht optimal: Jedes Mal wenn ein Schlüssel mit dieser Funktion dem Schlüsselspeicher hinzugefügt werden soll, wird der Schlüsselspeicher durch einen Aufruf der Funktion `writeDictToFile()` neu geschrieben. Besser wäre es, wenn gleich mehrere Schlüssel hinzugefügt werden können, sodass der Aufruf nur einmal am Ende gemacht wird.

6 Anhang

6.1 Gegenseitige Authentisierung mit dem Secure Channel Protocol

Die von mir verwendete Smartcard unterstützt das „Secure Channel Protocol“ (SCP) in der Version 3. Dieses ermöglicht eine vertrauliche und authentisierte Übertragung von APDUs. Dazu muss zuerst ein sicherer Kanal zur Karte aufgebaut werden. Das von mir entwickelte Javacard-Applet verwendet kein SCP, ich habe jedoch ein anderes Javacard-Applet und ein dazugehöriges Java-Programm geschrieben, zwischen denen zumindest eine gegenseitige Authentisierung stattfindet. In diesem Abschnitt möchte ich beschreiben, welche Schritte dazu notwendig sind. Die Quelltexte dazu befinden sich im Verzeichnis `secure_channel`.

6.1.1 Kurzbeschreibung der gegenseitigen Authentisierung

Damit ein sicherer Kanal aufgebaut werden kann, müssen sich die Smartcard und das Java-Programm gegenseitig authentisieren. Dieser Vorgang ist in *GlobalPlatform Secure Channel Protocol 3* [12, Abschnitt 5.2] beschrieben und soll in diesem Abschnitt kurz wiedergegeben werden. Die einzelnen Schritte werden später im Detail beschrieben.

Zuerst generiert das Java-Programm eine sogenannte Host-Challenge. Diese besteht aus 8 zufällig gewählten Bytes. Dann wird die `"INITIALIZE UPDATE"`-Instruktion gerufen, in dem eine Kommando-APDU an die Karte übermittelt wird, welche die Host-Challenge enthält. Das Javacard-Applet kann diese APDU mit einem Funktionsruf aus der *GlobalPlaform-API* [10] an die „Issuer Security Domain“ (ISD) weiterleiten. Das ISD ist ein Javacard-Applet, welches bereits vom Hersteller der Karte auf dieser installiert wurde. Das ISD berechnet eine Card-Challenge, leitet Sitzungsschlüssel ab und berechnet damit dann ein Karten-Kryptogramm. Das Javacard-Applet muss diese Schritte also nicht selber implementieren. Die Card-Challenge und das Kryptogramm werden als Antwort-APDU an das Java-Programm gegeben.

Nun kann das Java-Programm die Karte authentisieren. Zunächst werden die Sitzungsschlüssel abgeleitet. Damit kann das Karten-Kryptogramm berechnet werden und mit dem von der Karte erhaltenen Kryptogramm verglichen werden. Stimmen beide überein, gilt die Karte als authentisiert. Nun wird noch ein Host-Kryptogramm berechnet, welche in einer Kommando-APDU an die Karte übermittelt wird, das die `"EXTERNAL AUTHENTICATE"`-Instruktion ruft. Über diese APDU wird mit den abgeleiteten Sitzungsschlüsseln ein MAC berechnet, welcher der APDU angehängt wird. Das Javacard-Applet übergibt diese APDU wieder an das ISD, welches den MAC und das Host-Kryptogramm verifiziert. Im Erfolgsfall hat die Karte das Java-Programm authentisiert. Damit ist die gegensetige Authentisierung abgeschlossen.

6.1.2 Javacard-Applet

Wie im vorigen Abschnitt beschrieben, muss das Javacard-Applet nichts weiter machen, als APDUs, die entweder die `"INITIALIZE UPDATE"`- oder die `"EXTERNAL AUTHENTICATE"`-Instruktion rufen, an das ISD weiterleiten. Dafür werden Funktionen aus der *GlobalPlaform-API* [10] verwendet. Um diese nutzen zu können sind die nachfolgend beschriebenen Schritte notwendig.

Zuerst müssen wir herausfinden, welche GlobalPlatform-Version die Karte unterstützt. Dafür kann man das Programm *GlobalPlatformPro* [13] verwenden:

```
$ java -jar gp.jar --info | grep "GP Version"
```

Die GlobalPlatform-Version meiner Karte ist 2.3.

Dann müssen wir uns die Java-Archiv-Datei der Bibliothek und die dazugehörige Export-Datei beschaffen. Ich habe ein Github-Repository gefunden, welches diese bereitstellt. [11] In der Datei `README.md` befindet sich eine Tabelle, die zu einer GlobalPlatform-Version die dazu passende Versionsnummer der GlobalPlatform-API angibt. In unserem Fall benötigen wir die API mit der Version 1.6. Die entsprechende Java-Archiv-Datei und die Export-Datei können dann von dem Repository heruntergeladen werden. Es ist vielleicht sinnvoll herauszufinden, ob man diese Dateien auch von GlobalPlatform direkt beziehen kann.

Nun wird die Java-Archiv-Datei dem Eclipse-Projekt hinzugefügt. Dazu Rechtsklick auf das Projekt, Properties, nach „Java Build Path“ suchen und auswählen, im Reiter „Libraries“ die Schaltfläche „Add External JARs...“ auswählen und die Java-Archiv-Datei auswählen. Im Javacard-Applet können nun Klassen und Interfaces aus der *GlobalPlaform-API* [10] importiert werden. In unserem Fall benötigen wir:

```java
import org.globalplatform.SecureChannel;
import org.globalplatform.GPSystem;
```

Damit das Applet gebaut werden kann, muss dem Projekt die Export-Datei hinzugefügt werden. Falls man dies nicht tut und eine Funktion aus der *GlobalPlaform-API* [10] ruft, bricht der Kompilationsprozess mit der Meldung

```
error: export file globalplatform.exp of package org.globalplatform not found.
```

ab. Um diesen Fehler zu beheben, erstellen wir im Projektverzeichnis des Eclipse-Projekts das Verzeichnis `exports/org/globalplatform/javacard/` und legen die Export-Datei in diesem Verzeichnis ab. Dann Rechtsklick auf das Projekt in Eclipse, Javacard, CAP Files Settings. Im neuen Fenster den Eintrag in der Tabelle auswählen und die Schaltfläche „Edit“ betätigen. Es öffnet sich ein weiteres Fenster. Hier die Schaltfläche „Next“ wählen. Im Reiter „Export Path“ die Schaltfläche „Add“ betätigen und das von uns erstellte Verzeichnis „exports“ im Projektverzeichnis auswählen.

Die Datei `SecureChannelTest.java` enthält den Quellcode des Javacard-Applets, welches ich zum Testen verwendet habe. In der `process()`-Methode wird `GPSystem.getSecureChannel()` gerufen. Diese Methode gibt ein Handle zum

`SecureChannel`-Interface zurück. Dann wird mit der `isSecurityRelated()`-Methode geprüft, ob die APDU, die momentan verarbeitet wird, eine der `"INITIALIZE UPDATE"`- oder `"EXTERNAL AUTHENTICATE"`-Instruktion ruft. Ist das der Fall, wird die APDU wie oben bereits beschrieben an das ISD weitergegeben. Dies erfolgt mit dem Aufruf der Methode `sc.processSecurity()`. Das ISD schreibt Antwort-Daten direkt in den APDU-Puffer. Das Applet kann diese Daten mit `apdu.setOutgoingAndSend()` als Antwort-APDU an das Java-Programm zurücksenden.

Falls die Instruktion der APDU den Wert 0x10 hat, so wird die Methode `sc.getSecurityLevel()` gerufen. Der Rückgabewert gibt unter anderem an, ob bereits eine gegenseitige Authentisierung erfolgt ist. Dieser Wert wird in der Antwort-APDU zurückgegeben. Diese Instruktion kann verwendet werden um zu testen, ob die gegenseitige Authentisierung erfolgreich war.

In allen anderen Fällen gibt das Applet die erhaltene APDU einfach wieder zurück.

Eine Besonderheit gibt es noch: Die `deselect()`-Methode der Klasse `javacard.framework.Applet` ist implementiert. Hier wird die Methode `sc.resetSecurity()` gerufen. Dies bewirkt, dass alle Informationen bezüglich des sicheren Kanals zurückgesetzt werden, wenn das Applet deselektiert wird.

6.1.3 Java-Programm

Neben dem Applet gibt es noch das Programm `SecureChannelClient.java`, welches mit dem Applet einen sicheren Kanal etablieren soll. Hierbei ist vorallem die Methode `mutualAuthenticate()` von Interesse, welche die gegenseitige Authentisierung implementiert und in diesem Abschnitt beschrieben werden soll.

Zunächst benötigen wir den statischen Verschlüsselungsschlüssel und den statischen MAC-Schlüssel. (Siehe [12, Abschnitt 6]) Diese sind bei unserer Javacard auf den Standardwert 0x40 ... 0x4f gesetzt. Dieser Wert wird als Byte-Array in der Variable `key` gespeichert.

Dann wird mit der Methode `getHostChallenge()` die Host-Challenge erzeugt. Hier werden acht zufällige Bytes zurückgegeben. Mit der `initializeUpdate()`-Methode wird dann die entsprechende Instruktion auf der Javacard gerufen. Der Aufbau der APDU ist in [12, Abschnitt 7.1.1] beschrieben. Dort ist auch die Struktur der Antwort-Daten beschrieben. Dafür habe ich die Klasse `InitializeUpdateResponse` geschrieben, welche diese Struktur beschreibt und Zugriff auf einzelne Elemente ermöglicht. Die Methode `initializeUpdate()` gibt ein Objekt dieses Typs zurück.

In der Antwort ist die Card-Challenge enthalten, mit dessen Hilfe wir nun die Sitzungsschlüssel ableiten können. Die Ableitung ist in [12, Abschnitt 4.1.5] beschrieben und in der Methode `derive()` implementiert. Dazu wird zunächst eine 32 Byte lange Nachricht konstruiert, die abhängig ist vom Typ des Schlüssels bzw. des Kryptogramms, welches abgeleitet werden soll. Dann wird der CMAC dieser Nachricht berechnet und

entweder vollständig oder gekürzt zurückgegeben. Der Schlüssel für den CMAC ist abhängig von dem Schlüssel, der abgeleitet werden soll. Zur Berechnung des CMAC habe ich die Klasse `CMAC` geschrieben, welche die Spezifikation NIST 800-38B [27] implementiert, da die Java-Standardbibliothek nur HMAC anbietet.

Die abgeleiteten Sitzungsschlüssel werden in den Variablen `sEnc` und `sMac` gespeichert. Mit diesen kann das Karten-Kryptogramm berechnet werden und mit dem Kryptogramm verglichen werden, welches die Smartcard zurückgegeben hat. Außerdem wird noch das Host-Kryptogramm berechnet. [12, Abschnitt 6.2.2] Dieses soll mit einer APDU an die Karte übermittelt werden, welche die `"EXTERNAL AUTHENTICATE"`-Instruktion ruft.

Die Methdode `externalAuthenticate()` konstruiert und überträgt diese APDU. Laut [12, Abschnitt 6.2.3] muss über diese APDU allerdings ein MAC berechnet und mit übertragen werden. Daher wird vorher die Methode `getMacForExternalAuthenticate()` gerufen, die genau das implementiert. Das Verfahren ist in [12, Abschnitt 6.2.3 und 6.2.4] beschrieben. Wenn die `externalAuthenticate()`-Methode ohne Fehler zurückkommt, war die gegenseitige Authentisierung erfolgreich. Dies kann auch getestet werden: In der `main()`-Methode der Klasse `SecureChannelClient` wird nach Ausführung der `mutualAuthenticate()`-Methode noch `getSecurityLevel()` gerufen. Ist das höchstwertigste Bit dieses Wertes 1, so war die Authentisierung erfolgreich. [9, Abschnitt 10.6] Das gewünschte Sicherheitslevel kann der `mutualAuthenticate()`-Methode als Argument übergeben werden. Wird also zum Beispiel nur Authentisierung der Kommando-APDUs gewünscht, so sollte der Funktion `mutualAuthenticate()` das Argument 1 übergeben werden. Nach erfolgreichem Aufbau des sicheren Kanals sollte das Sicherheitslevel, welches `getSecurityLevel()` zurückgibt, den Wert 0x81 haben.

6.2 KNX-Laufzeitschlüssel aus ETS exportieren

In diesem Abschnitt wird beschrieben, wie die KNX-Laufzeitschlüssel aus der ETS (Version 6) exportiert werden können. Verwendete Ressourcen: [20, 28]

Die ETS bietet die Möglichkeit, alle Schlüssel eines Projekts als Schlüsselbund zu exportieren. Ein Schlüsselbund ist eine Textdatei im XML-Format, in der die KNX-Laufzeitschlüssel in verschlüsselter Form enthalten sind. Als Verschlüsselungsalgorithmus wird AES-128-CBC verwendet. Beim Export muss ein Passwort angegeben werden, aus dem mit einer Schlüsselableitungsfunktion der AES-Schlüssel erzeugt wird.

Die verwendete Schlüsselableitungsfunktion ist PBKDF2 mit den folgenden Eigenschaften:

- Hashfunktion: SHA-256
- Passwort: Das beim Export gesetzte Passwort, UTF-8 kodiert.
- Salt: UTF-8 kodierte Zeichenkette `1.keyring.ets.knx.org`

- Anzahl Iterationen: 65.536
- Schlüssellänge: 16 Bytes

Für den CBC-Betriebsmodus wird ein Initialisierungsvektor benötigt. Dieser lässt sich folgendermaßen bestimmen: Das äußere XML-Tag des Schlüsselbunds ist das `Keyring`-Tag. In diesem gibt es das Attribut `Created`. Nimm den Wert dieses Attributs als UTF-8 kodierte Zeichenkette und berechne darüber den SHA-256-Hash. Die ersten 16 Bytes des Hashs bilden den Initialisierungsvektor.

In `Group`-Tags gibt es das Attribut `Key`. Hier werden Laufzeitschlüssel von Gruppenadressen in Base64-Kodierung gespeichert. Diese können einfach dekodiert werden und mit dem ermittelten Schlüssel und Initialisierungsvektor entschlüsselt werden.

6.2.1 Werkzeug zum Extrahieren der Schlüssel aus einem Schlüsselbund

Es existiert ein Python-Skript `extractKeys.py`, welches ein Schlüsselbund und das zugehörige Passwort entgegennimmt und alle Laufzeitschlüssel von Gruppenaddressen im Klartext ausgibt.

Das Skript ist allerdings noch nicht fertig. Es sollte zu den Schlüsseln auch noch die zugehörige Schlüssel-ID ausgeben.

Außerdem wird das Passwort nicht auf Korrektheit geprüft. Das `Keyring`-Tag enthält ein Attribut `Signature` mit dessen Hilfe das Passwort auf Korrektheit geprüft werden kann. Momentan werden bei Eingabe eines falschen Passwortes falsche Schlüssel ausgegeben.

6.2.2 Kurzanleitung

1. Menüleiste: Arbeitsbereich > Projekt Details > Sicherheit > Backup Schlüsselbund
2. Der Schlüsselbund muss mit einem Passwort geschützt werden.
3. Es wird eine .knxkeys-Datei gespeichert.
4. Aufruf `./extractKeys.py FILE.knxkeys PASSWORT`

6.3 Entwicklung von Javacard-Applets unter Linux

Die von Oracle angebotenen Werkzeuge zur Entwicklung von Javacard-Applets sind auf den ersten Blick nur unter Windows lauffähig. Die `bin`-Verzeichnisse enthalten lediglich Batch-Skripte. Diese führen aber letztlich nur Java-Archive aus. Durch Übersetzung der Batch-Skripte in Shell-Skripte lassen sich die Werkzeuge auch unter Linux nutzen.

Im Verzeichnis `tools_linux` befinden sich die übersetzten Shell-Skripte. Außerdem ist dort ein beispielhaftes Makefile, welches diese Skripte ruft um das Applet zu bauen und es im Simulator auszuführen. Der Simulator wird mit wine ausgeführt.

Es müssen drei Umgebungsvariablen gesetzt sein:

- `JC_HOME_TOOLS`: Verzeichnis in dem die „Javacard Development Kit Tools“ liegen.
- `JC_HOME_SIMULATOR`: Verzeichnis in dem der „Javacard Development Kit Simulator“ liegt.
- `TOOLS_LINUX`: Verzeichnis in dem die übersetzten Shell-Skripte liegen.

Im Makefile müssen folgende Dinge angepasst werden:

- `CLASS`-Variable (Name der Klasse)
- `PACKAGE`-Variable (Name des Packages)
- `TESTFILE`-Variable (Name des APDU-Testfiles, welches im Simulator ausgeführt werden soll.)
- ggf. javac Source- und Target-Version [siehe 18]
- Applet- und Package-AIDs (unter converter.sh)
- Javacard-Version (unter javac, converter.sh und verifycap.sh)

Das Makefile beschreibt den Weg vom Quelltext zum fertigen Applet.

Ich empfehle zur Entwicklung von Javacard-Applets die Verwendung der Eclipse-IDE mit dem Javacard-Eclipse-Plugin von Oracle. (Welches ich nur unter Windows installieren konnte.) Dies hat mehrere Vorteile:

1. Einfache Javacard-Projekterstellung: Es geht sehr schnell ein neues Javacard-Applet zu erstellen und zu bauen.
2. Der Simulator schließt sich nicht nach Ausführung des APDU-Skripts, es bleibt geöffnet um interaktiv APDUs eintippen und ausführen zu können, oder um weitere APDU-Skripte auszuführen.
3. Der Eclipse-Debugger kann verwendet werden. Sehr hilfreich, da es in Javacard kein `printf()` oder Ähnliches gibt. Es ist denkbar, den Javacard-Simulator mit dem Java-Debugger jdb zu verwenden, der auch unter Linux läuft. Allerdings habe ich das nicht ausprobiert.

Literaturverzeichnis

[1] *Address Type*. URL: https://support.knx.org/hc/en-us/articles/115003188509-Address-Type-Hop-Count-and-Length (besucht am 01. 11. 2023).

[2] *An Introduction to Java Card Technology*. URL: https://www.oracle.com/java/technologies/java-card/javacard1.html (besucht am 04. 10. 2023).

[3] Jan Baudis. *Sicherheit in KNX-Netzen*. Jan. 2017.

[4] *CCM Mode for Authentication and Confidentiality*. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38c.pdf (besucht am 09. 10. 2023).

[5] *CommonEMI Frame Format (Commit 5451235)*. URL: https://github.com/thelsing/knx/blob/master/src/knx/cemi_frame.cpp (besucht am 01. 11. 2023).

[6] *Control Field*. URL: https://support.knx.org/hc/en-us/articles/115003188429-Control-Field (besucht am 01. 11. 2023).

[7] *cyberjack pinpad error*. URL: https://github.com/OpenSC/OpenSC/issues/1051 (besucht am 25. 10. 2023).

[8] Alexander Fehr. *Sicherer Schlüsselspeicher für KNX Data Secure*. Mai 2020.

[9] *GlobalPlaform Card Specification*. URL: https://globalplatform.org/wp-content/uploads/2018/06/GPC_Specification-2.2.1.pdf (besucht am 04. 12. 2023).

[10] *GlobalPlaform-API*. URL: https://www.win.tue.nl/pinpasjc/docs/apis/gp22/ (besucht am 04. 12. 2023).

[11] *GlobalPlatform Bibliothek und Export-File*. URL: https://github.com/OpenJavaCard/globalplatform-exports (besucht am 04. 12. 2023).

[12] *GlobalPlatform Secure Channel Protocol 3*. URL: https://globalplatform.org/wp-content/uploads/2014/07/GPC_2.3_D_SCP03_v1.1.2_PublicRelease.pdf (besucht am 04. 12. 2023).

[13] *GlobalPlatformPro*. URL: https://github.com/martinpaljak/GlobalPlatformPro (besucht am 02. 10. 2023).

[14] *Installation Parameters (JavaCardOS Wiki)*. URL: https://www.javacardos.com/wiki/javacard/jcre/11.2.1_installation_parameters (besucht am 18. 10. 2023).

[15] *ISO 7816-4*.

[16] *Java Card 3 Platform Programming Notes*. URL: https://docs.oracle.com/javacard/3.0.5/prognotes/title.htm (besucht am 04. 10. 2023).

[17] *Java Card Platform Runtime Environment Specification*. URL: https://docs.oracle.com/javacard/3.1/related-docs/JCCRE/JCCRE.pdf (besucht am 04. 10. 2023).

[18] *JavaCard SDK and JDK version compatibility.* URL: https://github.com/martinpaljak/ant-javacard/wiki/JavaCard-SDK-and-JDK-version-compatibility (besucht am 02. 10. 2023).

[19] *Javacard-API.* URL: https://docs.oracle.com/javacard/3.0.5/api/index.html (besucht am 29. 09. 2023).

[20] *Keyring class for loading and decrypting knxkeys files.* URL: https://github.com/XKNX/xknx/blob/main/xknx/secure/keyring.py (besucht am 08. 11. 2023).

[21] *KNX Data Secure Implementation (Commit 2aecc21).* URL: https://github.com/thelsing/knx/blob/master/src/knx/secure_application_layer.cpp (besucht am 01. 11. 2023).

[22] *KNX Data Security: Application Note 158/13 v02.* 14. Mai 2013.

[23] *KNX System Specifications - Communication Media - Twisted Pair 1.*

[24] *PC/SC sample in different languages.* URL: https://ludovicrousseau.blogspot.com/2010/04/pcsc-sample-in-different-languages.html (besucht am 25. 10. 2023).

[25] *PCSC-Spezifikation, Teil 10.* URL: https://pcscworkgroup.com/Download/Specifications/pcsc10_v2.02.09.pdf (besucht am 25. 10. 2023).

[26] *SmartMX3 P71D320 Mikroprozessor.* URL: https://www.cardlogix.com/downloads/support/SmartMX3-family-P71D320-datasheet.pdf (besucht am 27. 11. 2023).

[27] *The CMAC Mode for Authentication (NIST 800-38B).* URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38b.pdf (besucht am 04. 12. 2023).

[28] *Use keyring outside ETS.* URL: https://support.knx.org/hc/en-us/articles/360001582259-Use-keyring-outside-ETS-Falcon-SDK (besucht am 08. 11. 2023).

[29] *Wie schnell ist der KNX Bus?* URL: https://www.konnekting.de/2017/04/27/wie-schnell-ist-der-knx-bus/ (besucht am 08. 01. 2024).