Studienarbeit

Eine virtualisierte Smartcardarchitektur für mobile Endgeräte

Frank Morgner, Dominik Oepen

3. Februar 2009

Institut für Informatik
Systemarchitektur

Inhaltsverzeichnis

1 Einleitung

1.1 Einleitung

Sowohl Smartcards als auch Mobiltelefone können als portable Kleinstcomputer angesehen werden. Jedoch unterscheiden sie sich in ihren Eigenschaften und Einsatzzwecken erheblich voneinander. Smartcards werden meist zur sicheren Authentifizierung genutzt. Dies ist möglich, da die Hardware von Smartcards beim aktuellen Stand der Technik als vertrauenswürdig angesehen werden kann. Das heißt, dass es zur Zeit nicht möglich ist, geschützte Speicherbereiche von Smartcards auszulesen. Somit können (synchrone oder asynchrone) Schlüssel sicher gespeichert werden. Die Schnittstellen zu Smartcards sind sowohl auf physikalischer als auch auf Anwendungsebene durch die ISO standartisiert. Die Standardfamilie ISO-7816 beschreibt kontaktbehaftete, die Familie 14443 kontaktlose Smartcards (so genannte Proximity Integrated Circuit Cards (PICC)). Die Anwendungsschicht ist bei beiden Standardfamilien zueinander kompatibel. Es existieren ausgereifte Middleware Lösungen zur Verbindung zwischen Smartcards und PCs, als Beispiel sei PC/SC genannt. Allerdings ist zur Nutzung von Smartcards in Verbindung mit einem Computer ein Smartcardterminal notwendig. Die Verbreitung dieser speziellen Hardware im Heimbereich ist jedoch recht gering, auch wenn sie, zum Beispiel beim Onlinebanking, höhere Sicherheit verspricht.

Mobiltelefone verfügen im Vergleich zu Smartcards über sehr viel mehr Rechenleistung und Speicher, sowie über ein sehr viel höheres Maß an Konnektivität. Heutzutage verfügen die meisten Handys über standartisierte Schnittstellen, wie Bluetooth, USB und teilweise auch WLAN Funktionaltät. Zusätzlich sind Mobiltelefone natürlich auch mit GSM bzw. UMTS ausgestattet. Der hohe Verbreitungsgrad macht es attraktiv, das Handy in Kombination mit anderen neuen Technologien zu benutzen. Durch diese Eigenschaften entwickelt sich das Mobiltelefon immer weiter zum mobilen Allroundgerät. Fast jeder verfügt heutzutage über ein Handy, mit dem er nicht nur telefoniert und SMS schreibt, sondern verstärkt auch im Internet surft und auf mobile Dienste zugreift. Es handelt sich also um sehr vielfältig einsetzbare Geräte, wohingegen Smartcards meist einen eingeschränkten Verwendungszweck, z.B. als Bankkarte, Versichertenkarte, etc., haben.

In der vorliegenden Studienarbeit wollen wir die Möglichkeit untersuchen die Eigenschaften dieser beiden Geräteklassen miteinander zu kombinieren. Konkret soll es ermöglicht werden die sicheren Authentifizierungs- und Authentisierungsmöglichkeiten von Smartcards mit Hilfe eines Handys zu nutzen. Die genaue Funktionsweise und die Grobarchitektur unseres Systems stellen wir in diesem Kapitel vor. Das zweite

Kapitel beschäftigt sich mit der Funktionsweise von ISO-konformen Smartcards, um ein theoretisches Fundament für das Verständnis der Arbeit zu liefern. Im dritten Kapitel stellen wir die zentrale Komponente unseres Systems, die virtuelle Smartcard, vor und im vierten Kapitel die weiteren Komponenten des Systems. Im fünften Kapitel diskutieren wir unsere Lösung kritisch und bieten einen Ausblick auf weitere mögliche Entwicklungen.

1.2 Grobarchitektur

In dem von uns vorgestellten System wird mittels eines Mobiltelefons eine Bridging Funktionalität zwischen kontaklosen oder -basierten Smartcards und ebensolchen Terminals realisiert. Hierbei ist das Handy in der Lage sowohl die Karte als auch das Terminal zu emulieren und kann dadurch gegebenenfalls die komplette Infrastruktur zur Verfügung stellen. Somit wird es möglich, Smartcard-basierte Sicherheitsmechanismen einzusetzen, ohne eine Erweiterung bestehender Infrastrukturen vorzunehmen. Es ist weder nötig neue Hardware anzuschaffen noch bestehende Rechnersysteme oder Anwendungen zu modifizieren. Falls bereits ein Terminal für drahtlose Smartcards vorhanden ist, so kann mit diesem via NFC kommuniziert werden (siehe unten). Genauso kann das Handy als Terminal für vorhandene drahtlose Smartcards genutzt werden. Bestehende Infrastruktur kann also mit unserem System kombiniert werden. Abbildung 1.1 verdeutlicht das von uns realisierte Bridging:

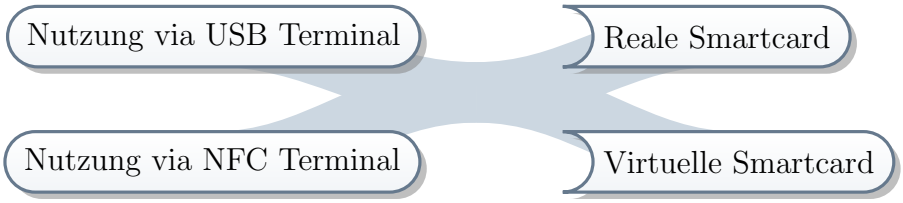

Abbildung 1.1: Bridging zwischen Typen von Smartcards und deren Nutzungsweise

Damit das Mobiltelefon auch die Funktion einer kontaktlosen Smartcard übernehmen kann, muss es natürlich über die notwendige Funktechnik verfügen. Der ISO Standard 14443 legt fest, dass PICCs im Frequenzbereich von 13,56 MHz, also im RFID Bereich, operieren. 2002 entwickelten die Firmen NXP Semiconductor und Sony den so genannten Near Field Communication (NFC) Standard. Dieser ist zu RFID kompatibel und wurde für den Datenaustausch zwischen Mobiltelefonen entwickelt. Darüber hinaus soll er Dienste, wie z.B. elektronische Tickets (beispielsweise [Deu08]) oder bargeldlose Zahlungsmöglichkeiten, realisierbar machen. Diese Technik ermöglicht es uns, mittels eines Handys, welches über NFC Technologie verfügt, sowohl mit kontaktlosen Smartcardterminals (Proximity Coupling Device - PCD) als auch mit PICCs zu kommunizieren.

Hardwareseitig verwenden wir für unser System die folgenden Komponenten:

- OpenMoko ([Ope08]): Als Mobiltelefon verwenden wir das OpenMoko Freerunner Telefon. Es bietet den Vorteil einer vollständigen Linux Systemumgebung, für die alle verwendeten Libraries bereits zur Verfügung stehen.

- OpenPICC ([Wel08]): Da das OpenMoko nicht über einen NFC Chip verfügt, muss diese Funktionalität nachgerüstet werden. Dafür verwenden wir den OpenPICC. Dieser verfügt über einen PN532 Chip von NXP, der sowohl den Betrieb als Smartcard als auch als Terminal ermöglicht. Bis jetzt ist die Integration des OpenPICC allerdings nur geplant und noch nicht durchgeführt.

Um unser System zu realisieren, müssen auf dem OpenMoko diverse Softwarekomponenten vorhanden sein. Teilweise handelt es sich dabei um bereits existierende Kompontenen, die lediglich für das OpenMoko kompiliert werden müssen, teilweise handelt es sich um Eigenentwicklungen. Folgende Software wird benötigt:

- Kern des Systems ist die virtuelle Smartcard, ein in Python geschriebenes Programm, welches eine ISO 7816 konforme Smartcard emuliert. Die genaue Architektur dieses Programms wird in Kapitel 3 beschrieben.

- Als Middleware zur Kommunikation von Smartcard und Anwendungen wird PC/SC verwendet. Damit die virtuelle Smartcard von PC/SC erkannt wird, wurde ein so genannter IFD-Handler programmiert. Dieser stellt einen Treiber für einen virtuellen Smartcard Reader dar. Er nimmt Kommandos in Form von Application Protocol Data Unit (APDU)s vom PC/SC Daemon entgegen und reicht sie an die virtuelle Smartcard weiter. Die Kommunikation findet hierbei über einen Socket statt.

- Zur Kommunikation des OpenMoko mit einem Computer via USB wurde ein CCID Treiber erstellt. Dieser Treiber bewirkt, dass ein per USB angeschlossenes OpenMoko am Computer als Smartcardterminal erkannt wird.

- Der OpenPICC wird via USB mit dem OpenMoko verbunden. Die Kommunikation zwischen der virtuellen Smartcard und dem OpenPICC läuft wiederum über PC/SC ab.

Die Kommunikation zwischen den einzelnen Kompontenten läuft dabei folgendermaßen ab: Je nachdem ob das OpenMoko via USB mit einem Computer verbunden ist oder per NFC mit einem Terminal kommuniziert werden die Datenpakete der Gegenstelle entweder von dem CCID Treiber oder vom OpenPICC empfangen. In beiden Fällen werden sie über das PC/SC API an den IFD-Handler weitergereicht. Dieser schickt sie über einen Socket an die virtuelle Smartcard weiter, welche die Abarbeitung übernimmt. Die Antwort der emulierten Smartcard nimmt den umgekehrten Weg zurück zur Gegenstelle. Abbildung 1.2 verdeutlicht die Kommunikation noch einmal grafisch.

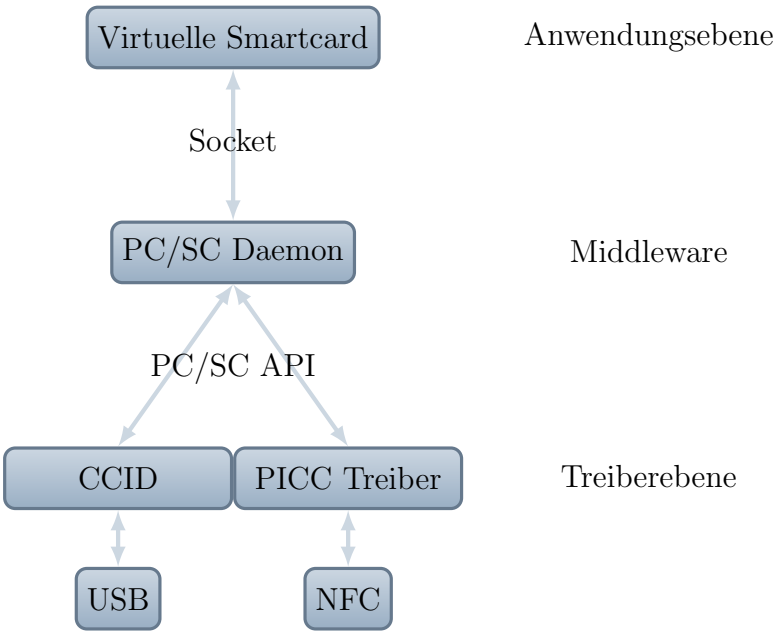

Abbildung 1.2: Kommunikation mit der virtuellen Smartcard

2 Vorgaben von ISO-7816

Bei der Implementierung unserer virtuellen Smartcard richteten wir uns nach den Vorgaben von ISO 7816, um eine möglichst hohe Kompatibilität unseres Systems mit bestehenden Anwendungen zu erreichen. Unser Fokus lag dabei auf Teil 4 der Spezifikation. Dieser Teil definiert die Anwendungsschicht und somit das Format und die Bedeutung der wichtigsten Chipkartenkommandos. Zusätzlich haben wir noch die in Teil 8 des Standards enthaltenen Kommandos für kryptografische Operationen implementiert, welche für eine Verwendung von Smartcards in sicherheitsrelevanten Bereichen wichtig sind. Außerdem wurde auch Teil 9 zum Lebenszyklus von Dateien auf der Chipkarte beachtet. Im Folgenden erklären wir die für unser Projekt wichtigen Teile der Standards. Der Abschnitt ist allerdings keinesfalls als vollständige Referenz zu verstehen, sondern soll lediglich dem Verständnis der restlichen Arbeit dienen.

2.1 Aufbau der APDUs

Die Kommunikation zwischen Terminal und Smartcard findet in Application Protocol Data Unit (APDU)s statt. Die Kommunikation wird immer durch das Terminal initiiert, welches eine Kommando APDU (Command Application Protocol Data Unit (CAPDU)) an die Karte sendet. Die Karte arbeitet den durch die CAPDU gegebenen Befehlt ab. Der Status der Abarbeitung und evtl. anfallende Antwortdaten werden in einer Response Application Protocol Data Unit (RAPDU) an das Terminal zurück geschickt.

2.1.1 Aufbau von Kommando APDUs

Abbildung 2.1 zeigt den Aufbau einer Kommando-APDU. Sie besteht aus einem obligatorischen 4 Byte langen Header und einem optionalen Datenfeld.

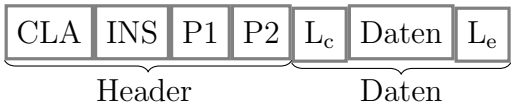

Abbildung 2.1: Aufbau einer Kommando APDU

Die einzelnen Bestandteile haben dabei folgende Bedeutung:

Class Byte (CLA) dient zur Auswahl des Befehlssatzes. So verwendet zum Beispiel GSM das Class Byte A0. Der ISO Standard verwendet Class Bytes von `0X`. Die letzten beiden Bits von X geben dabei die Nummer des verwendeten logischen Kanals an. Die ersten beiden Bits geben Auskunft, ob Secure Messaging (SM) zum Schutz der Nachrichtenintegrität und -vertraulichkeit (siehe Abschnitt 2.7) verwendet werden soll und falls ja, in welcher Form. Der genaue Aufbau des letzten Quartetts des Class Byte (CLA) ist in Tabelle 2.1 aufgelistet.

Tabelle 2.1: Bedeutung des Class Bytes einer Kommando APDU

b4	b3	b2	b1	Bedeutung
		X	X	Nummer des logischen Kanals
0	0			kein Secure Messaging
0	1			proprietäres Secure Messaging, nicht nach ISO
1	0			Secure Messaging nach ISO, Header wird nicht authentisiert
1	1			Secure Messaging nach ISO, Header wird authentisiert

Der Instruction Code (INS) gibt den durch die Smartcard auszuführenden Befehl an.

Die Parameter P1 und P2 geben Optionen für den jeweiligen Befehl an.

Das Datenfeld ist optional. Es kann aus bis zu drei Elementen bestehen: Dem L_c Feld, welches die Länge des Nutzdatenfeldes der APDU kodiert, den eigentlichen Nutzdaten und dem LE Feld, welches die Länge der Nutzdaten RAPDU enthaltenen Nutzdaten angibt. Es werden vier Fälle unterschieden:

1. Kein Datenfeld vorhanden
2. Datenfeld enthält nur das L_e Byte
3. Datenfeld enthält L_c Byte und Daten
4. Das Datenfeld enthält das L_c Byte gefolgt von Daten und dem L_e Byte

Damit Nutzdaten von mehr als 256 Bytes Länge transportiert werden können, muss die Smartcard erweiterte L_c- und L_e-Felder unterstützen. Da lange Datenfelder auch in mehrere (Antwort-) APDUs verpackt werden können, haben wir die Variante der erweiterten Feldern nicht implementiert. Die Bedeutung der Werte von L_c- und L_e sind in Punkt 2.1.1 angegeben.

Tabelle 2.2: Bedeutung von kurzen L_c- und L_e-Feld

L_c	Bedeutung
abwesend	keine Nutzdaten in der APDU
`0x00`	maximale Länge der Nutzdaten in der APDU (256 Bytes)
`0x01-0xff`	1 bis 255 Bytes Nutzdaten in der APDU
L_e	Bedeutung
abwesend	keine Nutzdaten in der RAPDU
`0x00`	maximale Länge der Nutzdaten in der RAPDU (256 Bytes)
`0x01-0xff`	1 bis 255 Bytes Nutzdaten in der RAPDU

2.1.2 Aufbau von Antwort APDUs

Der Aufbau einer Antwort-APDU ist sehr einfach. Sie besteht lediglich aus einem optionalen Datenfeld und einem obligatorischen Trailer, welcher zwei Statuswörter SW1 und SW2 umfasst. Diese Statuswörter sind jeweils ein Byte lang und geben Auskunft über den Status der Abarbeitung eines Kommandos. Die erfolgreiche Abarbeitung eines Kommandos wird durch ein SW1 | SW2 von `0x9000` angezeigt. In den anderen Fällen zeigt SW1 die Klasse des Statusworts an. `0x62` und `0x63` steht für Prozess wurde mit einer Warnung abgeschlossen, bei `0x64` und `0x65` wurde der Prozess auf Grund eines Ausführungsfehlers abgebrochen. Ein SW1 von `0x67` bis `0x6F` zeigt an, dass der Prozess Aufgrund eines Prüfungsfehlers abgebrochen wurde. Der genaue Fehler innerhalb einer Klasse wird durch SW2 angezeigt. Ein Wert von `0x00` bedeutet hierbei immer, dass keine genauere Information verfügbar ist.

Das optionale Datenfeld einer RAPDU hat höchstens die im L_e Feld der CAPDU angegebene Länge. Fallen bei der Abarbeitung eines Kommandos mehr Antwortdaten an, so müssen sie mit einem nachfolgenden get response Kommando gesondert abgeholt werden (siehe Abschnitt 2.4.1).

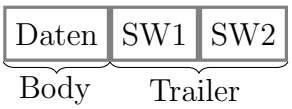

Abbildung 2.2: Aufbau einer Antwort APDU

2.2 Datenkodierung

Jedes Datenfeld, gehöre es zu einem Kommando oder bereits zu den Nutzdaten einer Datei, kann auf eine von zwei Arten strukturiert sein: Einerseits können die Daten im Binärformat vorliegen, also weitestgehend unstrukturiert sein. Andererseits ist die

Kodierung in Tag Length Value (TLV)-Tupeln möglich. Die zwei im ISO-Standard vorgesehenen Möglichkeiten der Bildung von TLV Datenobjekten möchten wir nun näher vorstellen.

2.2.1 SIMPLE-TLV Datenobjekte

Ein SIMPLE-TLV Datenobjekt besteht aus zwei oder drei Teilen:

- einem obligatorischem Tag
- einem obligatorischen Längenfeld
- und einem optionalen Datenfeld.

Das Tag besteht aus einem Byte mit einem Wert von 1 bis 254. Die Werte 0 und 255 sind für dieses Feld ungültig.

Das Längenfeld besteht entweder aus genau einem oder drei Bytes. Wenn der Wert des ersten Längenbyte *nicht* 255 ist, so kodiert er die Länge der direkt auf dieses Byte folgenden Nutzdaten. Ist das erste Byte des Längenfeldes jedoch auf 255 gesetzt, so folgen dem ersten zwei weitere Bytes, die die Länge der Nutzdaten angeben.

Die Länge der Nutzdaten liegt also bei kurzem Längenfeld zwischen 0 und 254. Bei langem Datenfeld beträgt sie zwischen 0 und 65535. Die Länge wird stets in Anzahl der Bytes angegeben. Wenn die Länge der Nutzdaten 0 ist, so fehlt das Datenfeld.

2.2.2 BER-TLV Datenobjekte

Wie bereits bei der SIMPLE-TLV Kodierung besteht ein BER-TLV Datenobjekt aus zwei oder drei Teilen:

- einem obligatorischem Tag
- einem obligatorischen Längenfeld
- und einem optionalen Datenfeld.

BER-TLV basiert auf der ASN.1 BER Kodierung in ISO/IEC 8825-1. Das Tag wird dabei mit bis zu drei Bytes festgelegt. Sind die ersten 5 Bits des Tags alle auf 1 gesetzt, so wird das Tag durch die Folgebytes definiert, andernfalls ist das Tag nur dieses eine Byte lang. Die Kodierung des ersten Bytes des Tags ist in Tabelle 2.3 abgebildet. Das achte Bit jedes Folgebytes gibt an, ob dieses das letzte Tagbyte ist. Steht es auf 0 gesetzt, folgt ein weiteres Byte zur Ermittlung des Tags, andernfalls ist es das letzte Tagbyte. Die Anzahl der Tags erhöht sich auf diese Weise auf insgesamt $2^{7+7} - 1 = 16383$.

Das 6. Bit des ersten Tagbytes gibt an, ob die Nutzdaten weitere BER-TLV Datenobjekte enthalten, d. h. ob das aktuelle Datenobjekt aus anderen konstruiert (*constructed*) wurde. Ein Beispiel hierfür sind die File Control Parameter (FCP), die

Tabelle 2.3: Bedeutung des ersten Bytes eines BER-TLV Datenobjektes

b8	b7	b6	Bits 5 bis 1	Bedeutung
0	0			Universal class (nicht definiert in ISO/IEC 7816)
0	1			Application class
1	0			Kontext-bezogene Klasse
1	1			Private class (nicht definiert in ISO/IEC 7816)
		0		Daten enthalten keine weiteren BER-TLV Objekte
		1		Daten enthalten weitere BER-TLV Objekte (*constructed*)
			nicht alle auf 1	Tag mit den Werten 1 bis 30
			alle auf 1	Tag größer als 30 und in den folgenden 2 oder 3 Bytes

aus mehreren BER-TLV-kodierten Dateieigenschaften zusammengesetzt sind (siehe Abschnitt 2.3).

Auch das Längenfeld kann nun mehr Werte tragen als bei SIMPLE-TLV, weil bis zu 5 Bytes dafür verwendet werden. Im ersten Byte wird so entweder eine Länge von bis zu 127 festgelegt oder bekannt gegeben, dass das Längenfeld aus bis zu 4 weiteren Bytes besteht. Folglich kann ein BER-TLV Objekt bis zu $2^{4 \cdot 8} = 4294967295$ Bytes (4,3 GB) Nutzdaten tragen.

2.3 Dateistrukturen auf der Smartcard

Eine Smartcard hat eine ähnliche Verzeichnisstruktur wie man sie von üblichen Festplattendateisystemen kennt. Es gibt ein Wurzel-Verzeichnis, das Master File (MF), Unterverzeichnisse (Dedicated File (DF)) und Dateien (Elementary File (EF)), die hierarchisch in mehreren Ebenen organisiert sein können.

Zu jeder Datei gehören Kontrollinformationen (FCP), die Aufschluss über strukturelle, logische oder die Sicherheit betreffende Attribute der Datei geben. Einige davon möchten wir nun näher betrachten.

2.3.1 File Identifier, DF Name und Short Identifier

Die EFs werden identifiziert über den zwei Byte langen File Identifier (FID). Diese müssen einerseits in den sie enthaltenden DFs eindeutig sein und andererseits darf keine Datei den FID des übergeordneten DFs tragen. Hinzu kommt die Begrenzung durch einige reservierte Werte, wie z. B. der FID des MF, welcher auf `0x3F00` festgelegt ist.

Ein EF kann einen optionalen Short EF Identifier (SID) besitzen, der Werte von 1 bis 30 annehmen darf, also 5 Bits lang ist. Ähnlich wie bei 1-Adressmaschinen dient

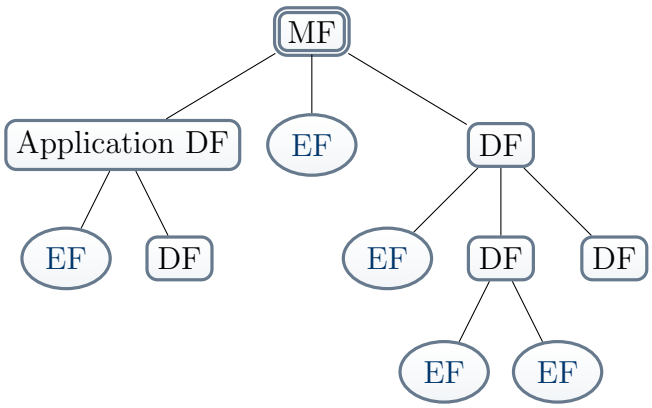

Abbildung 2.3: Typische Dateistruktur auf einer Smartcard

der SID dem Smartcard-Lesegerät dazu, zwei Befehle auf einmal auszuführen: Einen Wechsel der aktuellen Datei und ein Lese/Schreib-Kommando. So spart man bei der Kommunikation zwischen Karte und Lesegerät eine APDU.

Allein durch die Größe des FID ist die Anzahl der Dateien in einem DF bereits stark limitiert. Vorausschauend hat man daher den bis zu 16 Byte langen DF name eingeführt. Er ist, wie es der Name andeutet, DFs vorbehalten, welche nun also entweder einen DF name oder einen FID oder sowohl DF name als auch FID zur Identifikation tragen. Der FID eines DF folgt den selben Regel wie dem eines EF (s. o.). Zwei DFs dürfen nur dann denselben DF name tragen, wenn sie sich in ihrem FID unterscheiden. Der DF name kann genutzt werden, um einen Application Identifier (AID) zur weltweit eindeutigen Auswahl von Applikationen auf der Smartcard zu speichern. Über den DF name können also Applikationen gestartet werden oder Daten einer Applikation ausgelesen werden.

2.3.2 File Descriptor Byte

Das File Descriptor Byte (FDB) legt den Typ einer Datei fest. Eine bereits erwähnte grundsätzliche Unterscheidung ist die Differenzierung zwischen DF und EF. Ist im FDB das Bit für die Eigenschaft *shareable* gesetzt, ist der gleichzeitige Zugriff auf die Datei in verschiedenen logischen Kanälen erlaubt. Weiterhin gibt es spezielle EFs, deren Nutzdaten von der Smartcard interpretiert werden, wie z. B. EF.DIR, welches die Karteneigenschaften enthält. Diese Dateien sind dann als *internal* EF markiert. Nutzdaten eines *working* EF werden von der Smartcard nicht interpretiert. Bei unserem Smartcard Betriebsystem sind internal EFs nicht notwendig. Es handelt sich somit bei allen EFs um working EFs, die wir kurz EF nennen werden.

Die Struktur von einem EF wird unterteilt in transparent, record- oder TLV-basiert. Die Nutzdaten eines transparenten EFs sind direkt zugreifbar als kontinuierliche Folge von Dateneinheiten. Records hingegen sind Datenträger, die innerhalb eines EF individuell adressierbar sind. TLV-kodierte Daten können zu jeder Datei hinzugefügt werden (z. B. auch einem DF), sofern der Zugriff auf diese Datei erlaubt

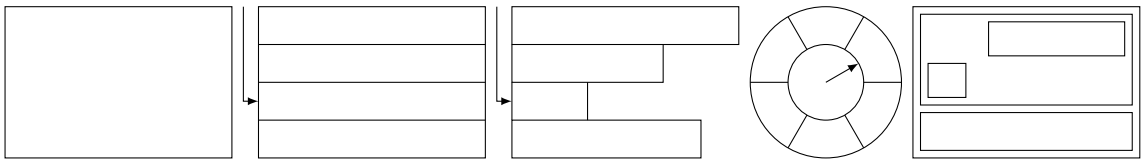

Abbildung 2.4: EF mit transparenter Struktur, linearer Struktur fester Größe, linearer Struktur variabler Größe, zyklischer Struktur und TLV Struktur (von links nach rechts)

ist. TLV-Daten sind seperiert von den Nutzdaten einer Datei. Eine speziell als TLV-basiert markierte Datei ist ein EF, der ausschließlich BER-TLV oder SIMPLE-TLV Datenobjekte speichert.

Die Adressierung eines record-basierten EFs geschieht entweder relativ zum aktuellen record oder absolut über die record number. Man unterscheidet bei diesem Typ von EF zwischen zyklisch und linear. Die record number 1 bezieht sich bei zyklischen EFs auf den zuletzt erstellten record, bei linearen ist dies hingegen der zuerst erstellte. Entsprechend ist der nächste Record bei der relativen Adressierung in einem zyklischen EF der record, welcher direkt vor dem aktuellen record erstellt wurde. Bei linearen EFs wurde der nächste record zeitlich gesehen direkt nach dem aktuellen erstellt. Bezüglich der Adressierung von records sind zyklische und lineare also genau invers. Der gravierendste Unterschied zeigt sich erst in dem Gebrauch und der Implementation von zyklischen bzw. linearen EFs. In Umgebungen, in denen beispielsweise die letzten x Aktionen von Relevanz sind, werden meist zyklische EFs benutzt, um diese zu speichern. Die Datei bestünde dann aus genau x records, wobei eine neu zu speichernde Aktion den ältesten record überschreiben würde. Zyklische EFs befinden sich aufgrund ihrer hohen Dynamik in realen Smartcards meist in Speicherbereichen, die oft geschrieben werden können. Der aktuelle record wird durch den sogenannten record pointer referenziert.

Die Nutzdaten eines records haben entweder eine vom EF vorgegebene statische Länge oder sind variabel. Weiterhin kann im FDB festgelegt werden, ob Nutzdaten der records eine TLV-Struktur besitzen oder nicht. Ist dies der Fall, so bekommt der entsprechende record den Tag seiner Nutzdaten als Identifikator. Auf diese Art kann man das Navigieren zum nächsten record eingrenzen auf alle records mit einem bestimmten Identifikator.

2.3.3 Data Coding Byte

Das data coding byte legt fest, wie zu schreibende Daten bereits vorhandene modifizieren. Es besteht die Möglichkeit, die zu schreibenden Daten durch logische Oder- oder Und-Verknüpfung der gegebenen mit den neuen Daten zu erhalten. Das intuitive Schreiben, welches die alten Daten durch die neuen ersetzt, wird als one-time write bezeichnet. Auch wenn in einem bestimmten EF, DF oder auch kartenweit ein bestimmtes data coding byte vorgegeben sein mag, kann man in manchen Fällen die

Methode des Schreibens explizit vorgeben (siehe Tabelle 2.4.2). Hier kann man für das Schreiben der Nutzdaten dieses Kommandos sogar die Option der XOR-Verknüpfung mit den vorhandenen Daten auswählen.

Weiterhin kodiert das data coding byte die kleinstmögliche Dateneinheit einer Datei (data unit size). Sie wird als Zweierpotenz von Quartets gespeichert und ist standardmäßig auf 1 festgelegt ($2^1 = 2$ Quartets, 1 Byte). Der größte Wert der data unit size ist 15 ($2^{15} = 8192$ Quartets, 16384 Bytes).

2.3.4 Life Cycle Status Byte

Jedes Objekt der Smartcard sowie die Smartcard selbst befindet sich in einem Lebenszyklus von 5 Zuständen:

Creation state Das Objekt wurde erstellt und befindet sich auf der Smartcard. Es ist jedoch von außen lediglich selektierbar und aktivierbar.

Initialisation state In diesem Zustand gibt es dieselben Einschränkungen wie im creation state. Hinzu kommt, dass das Objekt noch initialisiert werden muss oder gerade initialisiert wird.

Operational state (activated) Das Objekt ist verfügbar für beliebige Kommandos.

Operational state (deactivated) Das Objekt kann von außen zwar noch selektiert werden. Aber es ist sonst nur verfügbar für Kommandos, die das Objekt löschen, wieder aktivieren oder in den termination state überführen.

Termination state Das Objekt ist von außen nicht mehr nutzbar, obwohl es sich noch auf der Smartcard befindet. Das Selektieren des Objektes ist nur unter dem Erhalt einer Warnung möglich, da keine weiteren Aktionen erlaubt sind.

In der Praxis findet die Komplexität des Lebenszyklus meist keine Anwendung. Meist genügt es, eine Datei zu erstellen, die dann sofort für beliebige Kommandos zur Verfügung steht bzw. aktiviert ist. Ebenso geschieht das Löschen meist sofort endgültig, so dass der terminierte oder deaktivierte Zustand nicht benötigt werden.

2.4 Kommandos an die Smartcard

2.4.1 Allgemeine Kommandos

Get response

Falls das Ergebnis eines Kommandos zu lang für das Datenfeld der dazugehörigen response APDU gewesen ist, so wurde in den Statusbytes eine Warnung kodiert inklusive der Anzahl der Bytes, die noch nicht übermittelt werden konnten. Das Kartenlesegerät kann nun die fehlenden Daten mit einem get response Kommando abholen. Die beiden Parameterbytes sollen dabei auf 0 gesetzt werden.

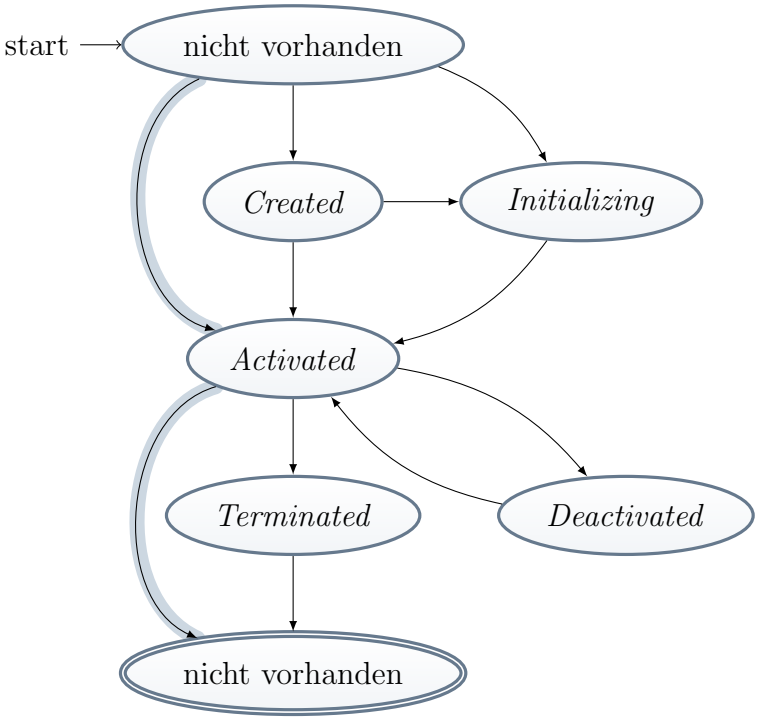

Abbildung 2.5: Lebenszyklus einer Datei. In der Praxis wird meist nur der aktivierte Zustand genutzt.

2.4.2 Dateibezogene Kommandos

Select Kommando

Nach dem Aufbau der Verbindung zwischen Karte und Lesegerät ist implizit das MF als aktuelle Datei selektiert. Die Dateiselektion muss die Smartcard als inneren Zustand speichern, da die meisten Kommandos die aktuelle Datei referenzieren können. Die aktuelle Datei ist vom Typ EF oder DF. Wenn sich ein Kommando auf das aktuelle Verzeichnis bezieht und ein EF als aktuelle Datei markiert ist, so ist das DF gemeint, welches dieses EF enthält. Das select Kommando wählt nun eine beliebige Datei aus dem Smartcard-Dateisystem und markiert diese als aktuelle Datei.

Im Smartcard Betriebssystem wird intern ein Sicherheitsstatus gespeichert, der angibt, ob die aktuelle Datei oder die übergeordneten Verzeichnisse bereits freigegeben wurden (z. B. durch ein external authenticate, siehe Punkt 2.5). Beim Wechsel der aktuellen Datei bleibt der Sicherheitsstatus erhalten, wenn ein übergeordnetes DF selektiert wird. Wird ein DF im aktuellen Verzeichnis gewählt, bleibt der Sicherheitsstatus ebenfalls erhalten, sofern für das untergeordnete DF keine weiteren Beschränkungen existieren. Tritt ein Fehler auf oder die zu selektierende Datei wurde nicht gefunden, geht der aktuelle Sicherheitsstatus verloren.

Das erste Parameterbyte gibt die Selektionsmethode vor. Möglich sind:

- Eine kontextbezogene Selektierung (`0x00`), bei der der Datenteil des Komman-

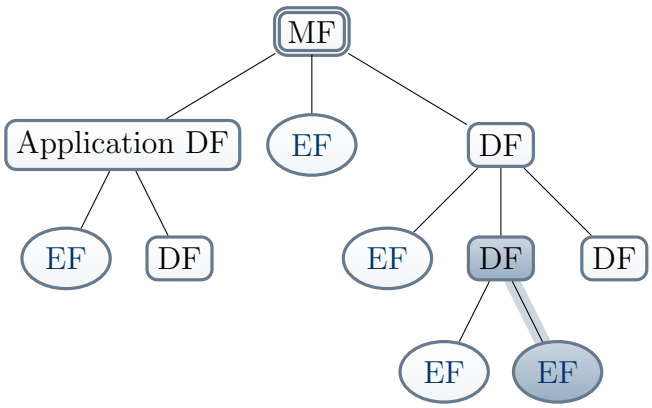

Abbildung 2.6: Selektion eines EF vom aktuellen DF

dos abwesend ist oder einen FID enthält. Ist ein FID vorgegeben, so wird entweder das MF (mit dem FID `0x3F00`), das aktuelle DF oder eine Datei im aktuellen Verzeichnis selektiert. Nach den Regeln zum FID ist der im Datenbereich enthaltene FID eindeutig (siehe Abschnitt 2.3.1).

- Eine Selektierung, bei der explizit der Typ der Datei gegeben ist (EF `0x01` oder DF`0x02`, Abschnitt 2.4.2). Im Datenfeld ist der FID der gesuchten Datei gegeben.
- Eine explizite Selektierung des DF, welches dem aktuellen Verzeichnis übergeordnet ist (`0x03`).
- Eine Selektion nach DF name (`0x04`, Abbildung 2.7), bei der im Datenfeld der entsprechende Binärstring gegeben ist. Bei dem DF name kann es sich auch um ein application identifier zum Starten einer Applikation auf der Smartcard handeln. Der DF name kann, wenn die Karte diese Funktion unterstützt, auch um einige Bytes gekürzt werden. Über das zweite Parameterbyte wird in diesem Fall festgelegt, welches DF selektiert werden soll, wenn mehrere zu dem abgeschnittenen DF name passen. Es ist möglich das erste, das letzte oder das nächste oder vorige DF (relativ zum aktuellen) zu wählen. Sinnvoll ist die Selektion über den gekürzten DF name, wenn damit ein application identifier kodiert wird. Dann ist es möglich, das AID um die Versionsnummer zu kürzen und zunächst irgendeine Version der Applikation zu selektieren bzw. starten.
- Eine Pfadselektion beginnend bei dem MF (`0x08`, Abbildung 2.8) oder dem aktuellen DF (`0x09`). Das Datenfeld enthält in diesem Fall eine Sequenz von mehreren FIDs, die zur gesuchten Datei führt (exklusive dem FID des startenden Verzeichnisses).

Das zweite Parameterbyte gibt an, welche Informationen in der response APDU enthalten sein sollen. Unterschieden wird hier zwischen den FCP, die allgemeine Dateiinformationen enthalten (siehe Abschnitt 2.3), den File Management Data (FMD),

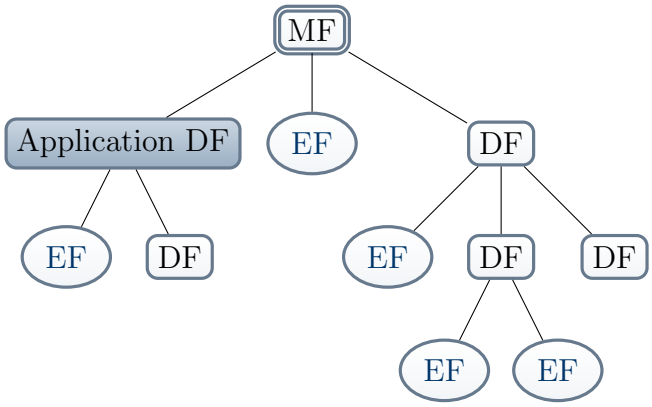

Abbildung 2.7: Selektion nach DF name bzw. AID

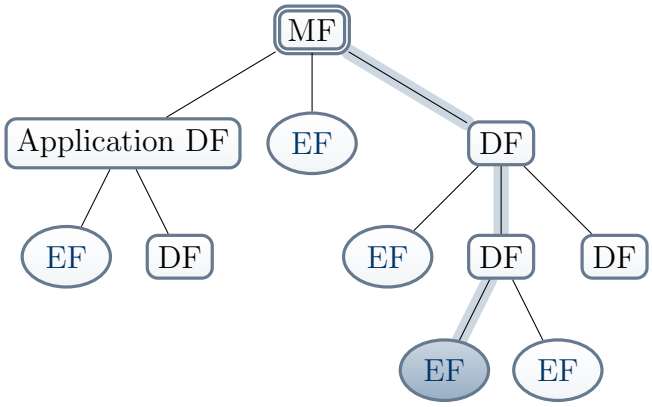

Abbildung 2.8: Selektion nach Pfad beginnend bei MF

die z. B. applikationsspezifische Daten enthalten, und dem File Control Information (FCI) template, welches sowohl Informationen zu FCP als auch FMD enthält.

Dateioperationen für transparente EFs

Read, write, update, search und erase binary Kommando geben entweder Teile der Daten eines transparenten EFs zurück oder modifizieren diese. Generell existiert jedes dieser Kommandos in zwei Varianten:

1. Bei geradem Instruction Code (INS) werden P1 und P2 entweder interpretiert als SID des betroffenen EF und/oder als offset der zu lesenden bzw. schreibenden Daten. Ist das Bit 8 von P1 gleich 0, so identifizieren die letzten 5 Bits von P1 den SID des transparenten EF und P2 den offset von 0 bis 255 Dateneinheiten (siehe Abschnitt 2.3.3). Bit 7 und 6 von P1 sollen dabei 0 sein. Ein SID von 0 meint den aktuellen EF. Ist Bit 8 von P1 jedoch auf 1 gesetzt, so kodieren die 15 verbleibenden Bits von P1 und P2 einen offset von 0 bis 32767.

2. Bei ungeradem INS werden P1 und P2 interpretiert als FID oder SID. Sind die ersten 11 Bits von P1 und P2 auf 0 gesetzt, so steht in den letzten 5 Bits von P2 ein SID. Ein SID von 0 meint den aktuellen EF. Sind die ersten 11 Bits nicht alle 0, so kodieren P1 und P2 einen FID. Bei ungeradem INS werden offsets von Dateneinheiten (siehe Abschnitt 2.3.3) und Datenfragmente sowohl in der response APDU als auch der command APDU mit BER-TLV als Offsetobjekt oder diskretes Datenobjekt kodiert. Mindestens ein Offsetobjekt soll sich in dem Datenfeld der command APDU befinden.

Read binary liest an den gegebenen Offsets die Daten eines transparenten EF aus und gibt diese im Datenfeld der response APDU zurück.

Write binary schreibt die gegebenen Datenfragmente an die gegebenen Offsets. Die Reihenfolge des Auftauchens legt fest, welches Datenfragment zu welchem Offset gehört. Wie die gegebenen Daten in die Datei geschrieben werden, wird durch das data coding byte festgelegt (siehe Abschnitt 2.3.3).

Update binary schreibt ebenso wie write binary Datenfragmente an die gegebenen Offsets des EF. Die vorhandenen Bits in den Daten des EF werden jedoch stets aktualisiert, was einem *one-time-write* entspricht.

Search binary sucht ein oder mehrere Datenfragmente in dem EF und gibt deren Offsets in der RAPDU zurück. Das Datenfeld ist leer, falls das Datenfragment nicht gefunden wurde.

Tabelle 2.4: Bedeutung von P2 bei Kommandos an record-basierte EFs. P2 legt mit dem 3. Bit die Bedeutung von P1 fest.

Bits 8 bis 4	b3	b2	b1	Bedeutung
0 0 0 0 0				aktuell selektierter EF
nicht alle auf 1				SID des zu selektierenden EF
	0			P1 ist record identifier
	0	0	0	wähle ersten passenden record
	0	0	1	wähle letzten passenden record
	0	1	0	wähle nächsten passenden record
	0	1	1	wähle vorigen passenden record
	1			P1 ist record number
	1	0	0	wähle genau einen record
	1	0	1	wähle alle records von P1 bis zum letzten
	1	1	0	wähle alle records vom letzten bis P1

Erase binary löscht einen Teil der Daten des EF. Sowohl bei geradem als auch ungeradem Instruction Code (INS) ist stets ein Offset gegeben. Dieser markiert die erste zur Löschung vorgesehene Dateneinheit. Ist zusätzlich ein zweiter Offset gegeben (bei geradem INS binär, bei ungeradem INS BER-TLV-kodiert im Datenfeld der APDU), so markiert dieser die letzte zu löschende Dateneinheit. Sollte kein zweiter Offset gegeben werden, so wird bis zum Ende des EF gelöscht.

Dateioperation für record-basierte EFs

Wie bereits oben erwähnt (Abschnitt 2.3.2), werden die records eines EFs entweder über die record number identifiziert, welche durch die Reihenfolge des Kreierens festgelegt wird, oder über den record identifier, welcher beliebig vergeben werden kann.

In einem Kommando an einen record-basierten EF legt Bit 3 des Parameterbyte P2 fest, ob die Adressierung über den record identifier (0) oder die record number (1) durchgeführt wird. Die Spezialfälle, die über die zwei Folgebits festgelegt werden, sind in Tabelle 2.4 ausgeführt. Die ersten 5 Bits bestimmen dabei den SID des EF, der angesprochen werden soll. Ein Wert von 0 meint wieder das aktuell selektierte EF. Wird ein EF über einen SID selektiert, wird dieses als aktuelle Datei markiert und sein record pointer wird zurückgesetzt.

Wieder können Kommandos mit ungeradem INS in ihrem Datenfeld einen oder mehrere BER-TLV-kodierte Offsets enthalten und ebenso kodierte Datenfragmente. Die Datenfelder der RAPDU werden dann ebenfalls BER-TLV kodiert. Bei geradem INS jedoch stehen je nach Befehl ein binärer Offset oder ein binäres Datenfragment im Datenfeld. Die RAPDU ist dann ebenfalls ein binärer Datenstrom.

Read record gibt Daten ein oder mehrerer records zurück. Bei geradem INS soll das Datenfeld abwesend sein. Bei ungeradem INS soll das Datenfeld einen BER-TLV-kodierten Offset enthalten.

Write record schreibt die gegebenen Datenfragmente an die Stelle der gegebenen Offsets (0 wenn nicht vorhanden) eines records. Bei der Adressierung der records dürfen nur die Modi verwendet werden, durch welche genau ein record angesprochen wird (die letzten 3 Bits von P2 bilden einen Wert kleiner 5). Beim Schreiben soll das data coding byte beachtet werden (siehe Abschnitt 2.3.3). Wenn die Adressierung des vorigen record gewählt wird und der EF zyklisch ist, so soll ein append record ausgeführt werden.

Update record verhält sich bei geradem INS wie write record, jedoch soll beim Schreiben ein *one-time-write* (siehe Abschnitt 2.3.3) angewendet werden. Als weiterer Unterschied zu write record soll das Kommando abgebrochen werden, wenn die Länge der gegebenen Daten ungleich der bereits vorhandenen Daten in dem record ist.

Bei ungeradem INS legen die letzten 3 Bits von P2 *nicht* die Adressierung des record fest, sondern es bestimmt wie ein Datenfragment der command APDU mit den vorhandenen Daten verknüpft werden soll. Möglich sind das Schreiben per *one-time-write*, logischem Und, logischem Oder und neu per logischem XOR. Das erste Parameterbyte ist dann immer eine record number.

Append record fügt der Datei einen record hinzu. Bei zyklischen wird der neue record den älteren vorangestellt, bei linearen EFs ist der neue record der letzte. Hat die Datei ihre Maximalzahl von records erreicht, wird das Kommando bei linearen EFs abgebrochen. Bei zyklischen EFs wird dann der älteste record durch den neuen ersetzt. Enthalten die records TLV-kodierte Daten (per FDB), so ist das gegebene Tag der Daten gleichzeitig der identifier des neuen record.

Search record durchsucht die Datei nach einem record, der einen Suchstring enthält. Das Kommando kann mit einem zusätzlichen Offset und der Suchrichtung sowie mit dem zu startenden record modifiziert werden. Bei Erfolg setzt die Smartcard den record pointer auf den gefundenen record.

Erase record löscht Records aus dem EF. Die letzten 3 Bits von P2 legen *nicht* die Adressierung des record fest sondern bestimmen, ob lediglich der record mit der record number in P1 gelöscht werden soll oder gleich von P1 bis zum letzten.

Kommandos bezüglich TLV Datenobjekte

Wie bereits oben erwähnt, kann jede Datei TLV-Daten tragen (siehe Abschnitt 2.3.2). Man unterscheidet hier zwischen den Kodierungen BER- und SIMPLE-TLV. Sie können via get und put data ausgelesen und geschrieben werden.

Bei ungeradem INS werden die beiden Parameterbytes als FID oder SID der betroffenen Datei interpretiert. Sind P1 und die letzten drei Bits von P2 gleich 0, so beinhalten die letzten 5 Bits von P2 den SID. Ein SID von 0 meint dabei wieder die aktuell selektierte Datei, wenn nicht im Datenfeld der command APDU ein file reference object gegeben ist. Andernfalls handelt es sich um den FID der gesuchten Datei. Ein FID von $3FFF$ meint das aktuelle DF. Die identifizierte Datei wird als aktuelle Datei markiert.

Bei geradem INS spezifizieren P1 und P2 entweder ein Tag der SIMPLE- oder ein BER-TLV-kodierten Daten. Die auszulesende Datei ist die aktuell selektierte Datei.

Get data gibt die BER- oder SIMPLE-TLV Daten einer Datei zurück. Über eine tag, header oder extended header list[ISO05b] werden bei ungerader INS alle oder tag-selektierte Daten konkateniert und TLV-kodiert zurückgegeben. Bei gerader INS wird mit P1 und P2 das Tag spezifiziert und nur Tag, Länge und Wert dieses Tags in das Datenfeld der Antwort geschrieben.

Put data legt BER- oder SIMPLE-TLV Daten bei einer Datei ab. Bei ungeradem INS enthält das Datenfeld der command APDU konkatenierte BER-TLV Daten. Bei geradem INS enthält das Datenfeld lediglich die Datenbytes der TLV Kodierung (P1 und P2 tragen das Tag und die Länge ist implizit gegeben).

2.5 Authentisierungsmechanismen

Im ISO Standard 7816-4 sind verschiedene Mechanismen zur Authentisierung der an der Smarcardkommunikation beteiligten Entitäten spezifiziert. Bei diesen Entitäten handelt es sich um:

1. Den Benutzer der Smartcard, welcher beweisen muss, dass er wirklich der legitime Besitzer der Karte ist.

2. Die Smartcard selbst

3. Das Terminal muss sich ebenfalls als vertrauenswürdig ausweisen, um Angriffe mit gefälschten Terminals zu erschweren. Es sei angemerkt, dass wir mit Terminal immer die Kombination aus Lesegerät und einem System, welches die geheimen Schlüssel von Smartcards kennt, bezeichnen. Das Manipulieren von Lesegeräten allein ist durchaus möglich und wird durch die unten vorgestellten Maßnahmen nicht verhindert. Die Vertraulichkeit und Integrität der Nachrichtenübertragung muss daher durch zusätzliche Mechanismen (wie etwa Secure Messaging (siehe Abschnitt 2.7) gewährleistet werden.

Verify Das verify Kommando wird verwendet um festzustellen, ob der Anwender auch der legitime Besitzer der Smartcard ist. Hierzu wird ein von der Karte und dem Besitzer geteiltes Geheimnis, die Personal Identification Number, kurz PIN, verwendet. Der Benutzer gibt die PIN, oftmals auch als Card Holder Value (CHV) bezeichnet, an einem Terminal ein. Sie wird dann im Datenfeld der verify APDU im Klartext an die Smartcard übermittelt. Dort wird sie mit der in der Smartcard gespeicherten PIN verglichen. Bei einer Übereinstimmung ist der Nutzer erflogreich authentisiert. Wird eine falsche PIN an die Karte gesendet, so wird ein Fehlversuchszähler dekrementiert. Erreicht dieser Zähler den Wert Null, so ist die Karte gesperrt. Sie muss dann mit dem Personal Unblocking Key (PUK) entsperrt werden. Dieser ist meist länger als die PIN selbst und wird in einer unblock CHV APDU an die Karte übertragen.

Außer dem Benutzer müssen auch Chipkarte und Terminal authentisiert werden. Bei dem Terminal ist dies wichtig, um Man-in-the-middle Angriffe mit gefälschten oder manipulierten Terminals zu erschweren. Bei der Smartcard kommt es darauf an, den Einsatz von selbst erstellten oder gefälschten Smartcards zu verhindern. Auch bei den Authentisierungsverfahren für Terminals und Smartcards wird ein pre-shared secret verwendet, nämlich ein gemeinsamer Schlüssel, der dem Terminal (oder einem mit dem Terminal verbundenen System) und der Karte bekannt sein muss. Im Gegensatz zum verify Kommando wird die Authentizitätsprüfung in diesem Fall jedoch nicht über einen simplen Vergleich des im Klartext übermittelten Geheimnisses sondern über ein challenge-response Verfahren realisiert. Da diese Protokolle meist mit Zufallszahlen, so genannten Noncen, arbeiten, wird noch ein zusätzliches Kommando namens get challenge benötigt. Dieses Kommando veranlasst die Karte eine Zufallszahl zu generieren und an das Terminal zu übermitteln. Diese Zufallszahl fließt daraufhin in die eigentlichen Authentisierungsprozeduren ein. Dadurch dass bei jeder Authentisierung eine andere Zufallszahl verwendet wird, sollen Replay Attacken,also das gezielte Wiedereinspielen in der Vergangenheit beobachteter Nachrichten, verhindert werden.

Internal authenticate Das internal authenticate Kommando dient der Authentisierung der Chipkarte gegenüber dem Terminal. Im Datenfeld der APDU wird eine Nonce an die Karte übermittelt. Die Karte verschlüsselt diese Nonce und sendet das Resultat an das Terminal. Das Terminal verschlüsselt seinerseits ebenfalls die Nonce und vergleicht sein Ergebnis mit der Antwort der Karte. Stimmen die beiden Zahlen überein, so weiß das Terminal, dass die Karte über den geheimen Schlüssel verfügt und somit vertrauenswürdig ist.

External authenticate Dieses Kommando ist das Gegenstück zu internal authenticate und dient somit der Authentisierung des Terminals gegenüber der Karte. Da grundsätzlich alle Kommunikation vom Terminal ausgeht, muss dieses zunächst eine Nonce mit dem oben erwähnten get challenge Kommando von der Karte anfordern.

Diese Nonce verschlüsselt das Terminal und sendet das Ergebnis im Datenfeld von external authenticate an die Karte, welche daraufhin die verschlüsselte Challenge verifiziert.

Mutual authenticate Auf der Basis von internal authenticate und external authenticate kann man nun ein Protokoll zur wechselseitigen Authentisierung von Karte und Terminal entwerfen. In dieses Protokoll fließen die Zufallszahlen N_S und N_T von beiden Parteien ein. Das Protokoll wird wieder vom Smartcardterminal initiiert. Dieses muss mittels get challenge die erforderliche Nonce von der Smartcard abfragen. Anschließend wird im Terminal die Konkatenation der beiden Zufallszahlen (also $N_S \mid N_T$) gebildet. Diese wird mit dem Schlüssel K_S verschlüsselt und im Datenfeld der mutual authenticate APDU an die Karte gesendet. Die Karte entschlüsselt das Datum und überprüft, ob die von ihr generierte Challenge N_S korrekt enthalten ist. Ist dies der Fall, so vertauscht sie die Reihenfolge der beiden Noncen und erhält somit das Datum $N_T \mid N_S$. Dieses wird wiederum verschlüsselt und an das Terminal übermittelt. Nach der Entschlüsselung kann das Terminal die empfangenen Daten verifizieren. Nach diesem Schritt können sich beide Parteien sicher sein, dass ihr jeweiliger Gegenüber im Besitz des korrekten Schlüssels ist und somit als vertauenswürdig betrachtet werden kann.

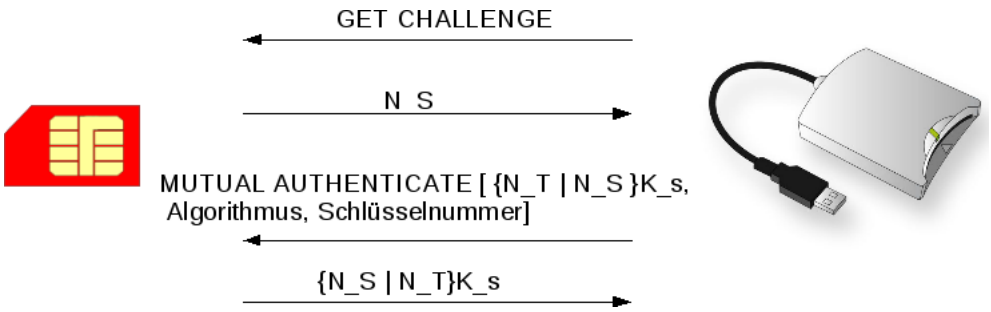

Abbildung 2.9: Ablauf des mutual authenticate Kommandos

Es bleibt zu erwähnen, dass für internal authenticate und external authenticate der selbe INS (`0x82`) verwendet wird. Dadurch kann auf einer Karte entweder nur eines der Kommandos implementiert werden, oder es muss aus dem Anwendungskontext heraus klar sein, welches Kommando zu verwenden ist.

Bei den vier oben erklärten Kommandos (verify, internal, external und mutual authenticate) dient der Parameter P1 zur Codierung des zu verwendenden Verschlüsselungsalgorithmus. Ein Wert von `0x00` zeigt hierbei an, dass keine weiteren Informationen übermittelt werden, der Algorithmus also implizit bekannt ist. Der ISO Standard macht keine Angaben dazu, wie Algorithmen zu referenzieren sind. Die Festlegungen, welche wir in unserer Implementierung getroffen haben, sind auf Seite 47 zu finden. Paramter P2 referenziert den zu verwendenden Schlüssel. Wieder bedeutet der Wert `0x00`, dass keine weiteren Informationen gegeben sind. Die genaue

Codierung von P2 zeigt Tabelle 2.5 auf:

Tabelle 2.5: Bedeutung von P2 bei Authentisierungskommandos

b8	b7 b6	Bits 5 bis 1	Bedeutung
0	0 0	0 0 0 0 0	Keine weiteren Informationen
0			Globale Referenz (MF spezifisch)
1			Lokale Referenz (DF spezifisch)
	0 0		andere Werte sind nicht erlaubt
		nicht alle 0	Nummer des referenzierten Schlüssel, bzw. Geheimnisses

2.6 Security Environment

Die Security Environment (SE) ist eine Datenstruktur, die zum Abspeichern der Betriebsparameter für alle sicherheitsrelevanten Vorgänge innerhalb der Chipkarte dient. Es handelt sich hierbei um TLV-kodierte Daten, eingebettet in ein interindustry SE template. Dieses template hat das Tag `0x7B` und kann folgende TLV Objekte enthalten:

- ein Security Environment Identifier (SEID) Byte mit dem Tag `0x80`
- ein optionales LCS Byte mit dem Tag `0x8A` (siehe auch Abschnitt 2.3.4)
- ein optionales cryptographic mechanism identifier template, wie es auch im Secure Messaging verwendet wird (siehe Abschnitt 2.7), mit dem Tag `0xAC`
- ein oder mehrere Control Reference Template (CRT)

Das SEID Byte dient zur eindeutigen Identifizierung einer Security Environment. Der Wert `0x00` deutet dabei eine leere Umgebung an, der Wert `0xFF`, dass keine Operationen durchgeführt werden können. Der Wert `0x01` wird üblicherweise für die Standard SE, welche immer vorhanden ist, verwendet und der Wert `0xEF` ist für den zukünftigen Gebrauch reserviert.

Die in einer SE abgespeicherten CRTs definieren die Betriebsparameter für die durch die Karte verwendeten kryptographischen Operationen. Jedes CRT ist abermals eine TLV Datenstruktur, die so genannte control reference data objects enthält. Es existieren sechs Typen von CRTs:

- Authentication Template (AT), Tag `0xA4`
- Key Agreement Template (KAT), Tag `0xA6`
- Hash-code Template (HT), Tag `0xAA`

- Cryptographic Checksum Template (CCT), Tag `0xB4`
- Digital Signature Template (DST), Tag `0xB6`
- Confidentiality Template (CT), wobei es hier ein Template für symmetrische Algorithmen (CT-sym) und eines für asymmetrische Algorithmen (CT-asym) gibt. Als Tag wird `0xB8` verwendet.

Es existiert eine Vielzahl von control reference data objects. Allerdings kann nicht jedes Objekt in jedem Typ von template vorkommen. Im Folgenden betrachten wir die einzelnen Objekte etwas genauer und zeigen auf in welchen templates sie gültig sind.

Das grundlegenste CRT Objekt hat das Tag `0x80` und dient zur Angabe des zu verwendenden Algorithmus. Zunächst einmal existieren verschiedene control reference data objects, um Dateien oder Schlüssel zu referenzieren (siehe Tabelle 2.6).

Tabelle 2.6: Control reference data objects zum Datei oder Schlüsselzugriff

Tag	Bedeutung	AT	KT	HT	CCT	DST	CT-asym	CT-sym
`0x81`	File reference	x	x	x	x	x	x	x
`0x82`	DF name	x	x	x	x	x	x	x
`0x83`	Reference of a secret key (for direct use)	x	x	x	x			x
`0x84`	Reference for computing a session key	x	x		x			x
`0xA3`	Key usage template	x	x	x	x	x	x	x

Die zweite Art von control reference data objects dient zur Spezifikation von Initialisierungsvektoren. Bei diesen Objekten ist eventuell kein Längenfeld vorhanden, da der Inhalt bereits aus dem Tag ersichtlich wird. In diesem Fall ist das Längen-Byte ebenfalls auf Null gesetzt (in Tabelle 2.7 mit $L = 0$ gekennzeichnet). Falls kein Wert für einen Initialisierungsvektor angegeben ist, so sieht der ISO Standard die Verwendung eines Nullvektors vor.

Tabelle 2.8 zeigt die sogenannten *auxiliary data elements* und in welchen CRTs sie vorkommen können. Diese Objekte dienen dazu Initialiserungsdaten für die verschiedenen Operationen zur Verfügung zu stellen.

Drei weitere Objekte sollen noch betrachtet werden:

- Das usage qualifier Byte mit dem Tag `0x95`. Dieses Objekt kann in allen CRTs bis auf HT vorkommen. Von der Bedeutung her entspricht es dem Parameter P1 im manage security environment Kommando (siehe Tabelle 2.9) und deutet somit an, in welcher Komponente einer Security Environment das Control Reference Template (CRT) verwendet werden darf. Zusätzlich kann durch das Setzen des dritten oder vierten Bit festgelegt werden, ob eine Passwort-

Tabelle 2.7: Control reference data objects für Initialisierungsvektoren

Tag	Bedeutung	AT	KT	HT	CCT	DST	CT-asym	CT-sym
`0x85`	Null Block ($L = 0$)			x	x			x
`0x86`	Chaining block ($L = 0$)			x	x			x
`0x87`	IV im Datenfeld ($L \neq 0$)			x	x			
`0x87`	Vorheriger IV plus 1 ($L = 0$)				x			x

oder eine Biometrie-basierte Nutzerauthentifizierung im authentication template verwendet werden soll.

- Das Objekt mit dem Tag `0x8E` dient dazu Informationen über des Inhalts eines Kryptogramms bereitzustellen. Das erste Byte des Datenfeld kodiert dabei die Information über den Inhalt. Es handelt sich hierbei um ein padding-content indicator byte, wie es in Tabelle 2.7 näher beschrieben ist.

- Objekte mit dem Tag `0xA3` referenzieren einen key usage counter. Dieser kann in jeder Art von CRT vorkommen und dient dazu, einen Zähler mit der in dem CRT referenzierten Datei und/oder Schlüssel zu verknüpfen. Dieser Zähler kann entweder als Fehlversuchszähler oder aber als Zugriffszähler verwendet werden.

Die Verwaltung der Security Environments erfolgt über das Kommando manage security environment. Das Kommando wird dazu verwendet Security Environments zu aktivieren, zu speichern und zu löschen (restore, store, erase). Darüber hinaus dient es zur Manipulation der control reference templates einer Security Environment (set). Ob ein set, store, restore oder erase ausgeführt werden soll, wird durch den Parameter P1 festgelegt. Tabelle 2.9 beschreibt die genaue Codierung. Wie man sieht besteht eine SE aus 4 Komponenten: Einer für Secure Messaging in der CAPDU, eine für SM in der RAPDU und zwei für die Kommandos aus ISO 7817-8.

Der Parameter P2 enthält im Falle eines store, restore oder erase das SEID der zu verwendenden Security Environment. Die durch P1 spezifizierte Aktion wird mit der SE, welche die in P2 angegebene SEID innehat, durchgeführt. Bei einem set entspricht der Parameter P2 dem Tag eines Control Reference Template (CRT), also `0xA4`, `0xA6`, `0xAA`, `0xB4`, `0xB6` oder `0xB8`. Das Datenfeld der manage security environment APDU enthält in diesem Fall control reference data objects, welche auf das angegebene CRT der aktuell aktivierten Security Environment angewand werden.

Tabelle 2.8: Auxiliary data elements

Tag	Bedeutung	AT	KT	HT	CCT	DST	CT-asym	CT-sym
0x88	Vorangegangene challenge plus 1 ($L = 0$)				x	x	x	x
0x88	Keine weiteren Information ($L > 0$)							
0x90	Hash Code, wird von der Karte erzeugt ($L = 0$)			x		x		
0x91	Zufallszahl, wird von der Karte genertiert ($L = 0$)		x		x	x	x	
0x91	Zufallszahl ($L > 0$)					x	x	
0x92	Zeitstempel, wird von der Karte generiert ($L = 0$)		x		x	x		
0x92	Zeitstempel ($L > 0$)				x	x		
0x93	Digital Signature Counter ($L > 0$)				x	x		
0x93	Digital Signature Counter + 1 ($L = 0$)		x		x	x		
0x94	Challenge oder Daten zur Schlüsselableitung	x	x		x	x	x	

Tabelle 2.9: Bedeutung des Parameter P1 in manage security environment

b8	b7	b6	b5	b4	b3	b2	b1	Bedeutung
			1					Secure Messaging im Datenfeld der Kommando-APDU
		1						Secure Messaging im Datenfeld der Anwtort APDU
	1							Computation, decipherment, internal authentication and key agreement
1								Verification, encipherment, external authentication and key agreement
				0	0	0	1	set
				0	0	1	0	store
				0	0	1	1	restore
				0	1	0	0	erase

2.7 Secure Messaging

Um die Kommunikation zwischen Smartcard und Terminal abzusichern, spezifiziert ISO 7816-4 das so genannte Secure Messaging. Es dient dazu, die Integriät und die Vertraulichkeit der Datenübertragung zwischen Karte und Terminal zu gewährleisten. Secure Messaging wird zur Zeit noch kaum eingesetzt, da die Kommunikation zwischen Karte und Terminal bei kontaktbasierten Karten nur sehr schwer zu belauschen bzw. zu manipulieren ist. Es ist jedoch damit zu rechnen, dass das Secure Messaging bei einer weiteren Verbreitung von kontaklosen Karten stark an Bedeutung gewinnt. Bei diesen Karten ist die Kommunikation durch die funkbasierte Datenübertragung naturgemäß sehr einfach abzuhören. Somit ist eine Absicherung der Übertragung zwingend notwendig.

Secure Messaging funktioniert durch die Kapselung der Daten (unter Umständen auch des Headers/Trailers) einer APDU in so genannte Secure Messaging data objects. Bei diesen Objekten handelt es sich um TLV-kodierte Daten. Tabelle 2.10 listet alle spezifizierten SM data objects auf:

Objekte mit geradem Tag fließen in die Berechnung von Prüfsummen mit ein, solche mit ungeradem Tag nicht. Ob der Command Header in die Berechnung mit einfließt wird durch das CLA Byte der Kommando-APDU festgelegt (siehe 2.1.1).

Welche Objekte in den Kommando und Antwort-APDUs vorhanden sein müssen, wird durch die jeweiligen Security Environments festgelegt. Des Weiteren kann in einer Kommando-APDU ein response descriptor template vorhanden sein (Tag `0xBA`, `0xBB`). Dieses template beschreibt, welche SM data objects in der Antwort APDU erwartet werden. Grundsätzlich werden zur Berechnung der SM data objects die Paramter (Schlüssel, Modi von Verschlüsselungsalgorithmen, Initialisierungsvektoren, etc.) aus den CRTs der Security Environments verwendet. Es können allerdings auch

Tabelle 2.10: Secure Messaging data objects

Tag	Bedeutung
0x80, 0x81	Klartext nicht BER-TLV-kodiert
0x82, 0x83	Kryptogramm, Klartext in BER-TLV enthält weitere SM data objects
0x84, 0x85	Kryptogramm, Klartext in BER-TLV enthält keine weiteren SM data objects
0x86, 0x87	Padding content indicator Byte (siehe 28)
0x89	Command header (CLA INS P1 P2) vier Byte lang
0x8E	Kryptographische Prüfsumme
0x90, 0x91	Hash Code
0x92, 0x93	Zertifikat
0x94, 0x95	Security Environment Identifier (SEID Byte)
0x96, 0x97	Ein oder zwei Byte die N_e des ungesicherten command-respond Paar enthalten
0x99	Status (SW1 SW2)
0x9A, 0x9B	Eingabedaten zur Berechnung einer digitalten Signatur
0x9C, 0x9D	Öffentlicher Schlüssel
0x9E	Digitale Signatur
0xA0, 0xA1	Eingabedaten für die Berechnung eines Hash Code (die Daten werden gehasht)
0xA2	Eingabedaten für die Verifikation einer kryptographischen Prüfsumme
0xA4, 0xA5	CRT für Authentifizierung (AT)
0xA6, 0xA7	CRT für Schlüsselvereinabarung (KAT)
0xA8	Eingabedaten zur Verifizierung einer digitalen Signatur
0xAA, 0xAB	CRT für Hash Codes (HT)
0xAC, 0xAD	Eingabedaten zur Berechnung einer digitalen Signatur (die konkatenierten Value Felder mehrerer dieser Objekte werden signiert)
0xAE, 0xAF	Eingabedaten zur Überprüfung einer digitalen Signatur (die konkatenierten Value Felder mehrerer dieser Objekte sind signiert)
0xB0, 0xB1	BER-TLV-kodierter Klartext, der weitere SM data objects enthält
0xB2, 0xB3	BER-TLV-kodierter Klartext, der keine weitern SM data objects enthält
0xB4, 0xB5	CRT für kryptographische Prüfsummen (CCT)
0xB6, 0xB7	CRT für digitale Signaturen (DST)
0xB8, 0xB9	CRT für Vertaulichkeit (CT)
0xBA, 0xBB	Response descriptor template
0xBC, 0xBD	Eingabedaten zur Berechnung einer digitalen Signatur (Daten sind signiert)
0xBE	Eingabedaten zur Verifikation eines Zertifikats (Zertifikat ist enthalten)

CRTs als SM data objects übertragen werden. Ist dies der Fall, so hat das CRT in der APDU Vorrang vor dem in der SE gespeicherten. Es ist für genau ein Kommando-Antwort-APDU Paar gültig und gilt ab dem Objekt nach dem CRT.

Im Folgenden werden einzelne SM data objects näher erläutert:

Kapselung von Klartextdaten Daten, die nicht BER-TLV-kodiert sind, müssen durch entsprechende SM data objects gekapselt werden. Für BER-TLV-kodierte Daten ist dies optional. SM data objects zur Kapselung von Klartext sind die Objekte mit den Tags `0xB0`, `0xB2`, `0x80`, `0x89`, `0x96` und `0x99`, bzw. ihre Pendants mit ungeraden Tags, soweit vorhanden.

Kryptogramme Zur Wahrung der Vertraulichkeit können Daten zu Kryptogrammen verschlüsselt werden. Hierfür sind die SM data objects mit den Tags `0x82` bis `0x87` vorgesehen. Die Tags `0x86` und `0x87` sind für Kryptogramme, die ein padding-content indicator Byte enthalten, vorgesehen. Bei diesen Kryptogrammen wird das verwendete Padding durch das erste Byte im Datenfeld des Objekts kodiert. Tabelle 2.11 erläutert die möglichen Bedeutung dieses Bytes.

Tabelle 2.11: Bedeutung des Padding-content indicator Byte

Tag	Bedeutung
`0x00`	Keine weiteren Informationen zu Padding oder Inhalt
`0x01`	ISO Padding
`0x02`	Kein Padding
`0x1X`	Ein bis vier Schlüssel zur Verschlüsselung von Informationen
`0x2X`	Schlüssel zur Verschlüsselung weiterer Schlüssel
`0x3X`	Privater Schlüssel eines asymmetrischen Schlüsselpaars
`0x4X`	Passwort
`0x80` bis `0x8E`	Proprietär

Das ISO Padding funktioniert dabei folgendermaßen: An das Ende der zu paddenden Daten wird das Byte `0x80` angehängt. Anschließend werden die Daten bis zum Erreichen der Blockgröße der verwendeten Chiffre mit Nullbytes aufgefüllt. Zum Entfernen des Paddings müssen die Daten lediglich von rechts nach links bis zum Auftreten von `0x80` abgeschnitten werden.

SM data objects für Datenauthentizität Zur Sicherung der Integrität der übermittelten Daten sind die SM data objects mit den Tags `0x8E` bis `0x93` und `0x9C` bis `0x9E` vorgesehen.

2.8 Kommandos nach ISO 7816-8

Der Standard ISO 7816-8 spezifiziert die ”interindustry commands for a cryptographic toolbox”. Im wesentlichen sind damit drei Kommandos gemeint:

1. Manage security environment
2. Generate public key pair
3. Perform security operation

Manage security environment wurde bereits in Tabelle 2.6 erläutert.

Generate public key pair dient zur Erzeugung von asymmterischen Schlüsselpaaren innerhalb der Karte. Alternativ kann es dazu verwendet werden auf ein zuvor generiertes Schlüsselpaar zuzugreifen. Der INS des Kommandos beträgt `0x46`. Durch P1 können verschiedene Betriebsmodi gewählt werden (siehe Tabelle 2.12). Der Parameter P2 wird immer auf `0x00` gesetzt.

Tabelle 2.12: Bedeutung des Parameter P1 in Generate Public Key Pair

Bits 8 bis 3	b2	b1	Bedeutung
0		0	Erzeuge ein Schlüsselpaar
0		1	Greife auf ein zuvor erzeugtes Schlüsselpaar zu
0	0		Der öffentliche Schlüssel wird in der Antwort-APDU zurückgegeben
0	1		Das Datenfeld der Antwort-APDU entspricht einer Extended Header List (nicht implementiert)

Falls der öffentliche Schlüssel in der Antwort-APDU zurückgegeben wird, so geschieht dies in Form eines bestimmenten interindusrty templates. Dieses template (eine BER-TLV Struktur mit dem Tag `0x7F49`) kann die in Tabelle 2.13 aufgelisteten Datenobjekte enthalten.

Perform security operation ist das dritte in ISO 7816-8 spezifiezierte Kommando. Es dient gewissermaßen als Frontend für eine ganze Reihe kryptografischer Operationen. Abhängig von den Parametern P1 und P2 löst Perform Security Operation (PSO) unterschiedliche Aktionen aus. P1 wird hierbei als Tag für das Datenfeld der Antwort-APDU und P2 als Tag für das Datenfeld der Kommando-APDU interpretiert. Die Tags sind hierbei Tags von SM data objects. Folgende Aktionen können durch ein PSO Kommando ausgelöst werden:

Tabelle 2.13: Interindustry template zur Codierung von öffentlichen Schlüsseln

Tag	Bedeutung
`0x06`	optionaler Object Identifier des verwendeten Algorithmus
`0x80`	optionale Algorithmenreferenz, analog zu den in CRTs verwendeten Referenzen
	RSA
`0x81`	Modul
`0x82`	Öffentlicher Exponent
	DSA
`0x81`	Erste Primzahl
`0x82`	Zweite Primzahl
`0x83`	Basis
`0x84`	Öffentlicher Schlüssel
	ECDSA
`0x81`	Primzahl
`0x82`	Erster Koeffizient
`0x83`	Zweiter Koeffizient
`0x84`	Generator
`0x85`	Ordnung
`0x86`	Öffentlicher Schlüssel

Compute Cryptographic Checksum veranlasst die Berechnung einer kryptographischen Prüfsumme über den Wert, der im Datenfeld der Kommando-APDU übermittelt wird. Der Parameter P1 des Kommandos muss hierbei `0x8E` sein (als das Tag für ein SM data object, das eine solche Prüfsumme enthält). P2 muss auf `0x80` gesetzt sein, womit angezeigt wird, dass die Antwort im Klartext und nicht BER-TLV-kodiert in der Antwort-APDU übertragen wird.

Compute Digital Signature führt zur Berechnung einer digitalen Signatur für die Daten der Kommando-APDU. P1 kann hierbei nur den Wert `0x9E` annehmen. P2 hingegen kann auf `0x9A`, `0xAC` oder `0xBC` gesetzt werden und deutet an, welche Art von Daten im Datenfeld übermittelt wird. Hierbei steht `0x9A` für zu signierende Rohdaten (nicht BER-TLV-kodiert), `0xAC` für BER-TLV-kodierte Datenobjekte, deren Value-Feld signiert werden soll und `0xBC` für BER-TLV-kodierte Datenobjekte, die komplett signiert werden sollen.

Hash führt zur Durchführung einer Hashoperation. Hierbei ist P1 immer auf `0x90` zu setzen, um anzuzeigen, dass die Antwort-APDU einen Hash-Wert enthält. P1 kann entweder `0x80` oder `0xA0` betragen. Ein Wert von `0x80` führt dazu, dass die Daten der Kommando-APDU direkt gehasht werden. Ein Wert von `0xA0` zeigt an, dass das Kommando Datenelemente enthält, die für das Hashing relevant sind, beispielsweise Zwischenwerte der Hashberechnung (Tag `0x90`). In diesem Fall werden weitere Datenelemente gelesen, bis alle zur Berechnung des Hashs erforderlichen Informationen vorhanden sind. Der Hash-Wert kann entweder intern gespeichert werden, um in nachfolgenden Kommandos (wie z.B. dem Berechnen einer Signatur) verwendet zu werden, oder in der RAPDU zurückgegeben werden.

Verify Cryptographic Checksum erlaubt es, eine gegebene kryptographische Prüfsumme durch die Karte überprüfen zu lassen. P2 beträgt in diesem Fall immer `0xA2`. Im Datenfeld müssen alle notwendigen SM data objects vorhanden sein (also Klartext und zu verifizierende Prüfsumme). In der Antwort APDU werden keine Daten übertragen, daher ist P1 auf `0x00` zu setzen. Ob die Checksumme korrekt ist, wird durch den Statuscode der Antwort-APDU angezeigt. `0x9000` steht für eine korrekte Prüfsumme, `0x6987` für fehlende, `0x6988` für inkorrekte SM data objects.

Verify Digital Signature arbeitet analog zur Verify Cryptographic Checksum. P1 beträgt ebenfalls `0x00`, es werden keine Daten in der Antwort-APDU zurückgegeben. P2 wird auf `0xA8` gesetzt. Alle relevanten Daten müssen als SM data objects im Datenfeld der Kommando-APDU enthalten sein.

Verify certificate verifiziert das im Datenfeld übertragene Zertifikat. Der ISO Standard macht keine genauen Angaben zum Format des Zertifikats. Das Datenfeld muss die digitale Signatur als Secure Messaging data object enthalten. Um den richtigen

öffentlichen Schlüssel zur Verifikation des Zertifikats auszuwählen gibt es zwei Möglichkeiten:

1. Der zu verwendende öffentliche schlüssel wird durch ein Tag im Zertifikat referenziert (so genannte selbstbeschreibende Zertifikate). In diesem Fall wird P2 auf `0xBE` gesetzt.

2. Das Zertifikat ist nicht selbstbeschreibend. In diesem Fall muss der zu verwendende Schlüssel entweder implizit bekannt sein, oder explizit in einer header list angegeben werden. In diesem Fall hat P2 den Wert `0xAE`.

Der durch ein verify certificate Kommando ausgewählte öffentliche Schlüssel wird zum Standardschlüssel für nachfolgende verify digital signature Kommandos.

Encipher veranlasst die Verschlüsselung von an die Karte gesendeten Klartextdaten (P2 ist `0x80`). Drei verschiedene Formate für die Antwortdaten sind möglich. Ist P1 auf `0x82` gesetzt, so wird dadurch angezeigt, dass die verschlüsselten Daten SM data objects enthalten. Bei `0x84` ist dies nicht der Fall. Ein P1 von `0x86` veranlasst eine Kodierung der Antwortdaten gemäß den Vorgaben zum Padding Indicator Byte (siehe Tabelle 2.7). In allen drei Fällen wird die Antwort BER-TLV-kodiert.

Decipher verhält sich analog zu encipher. Es wird immer nicht BER-TLV-kodierter Klartext zurückgegeben (P1 = `0x80`). Die Daten der Kommando APDU müssen den Daten von SM data objects mit den Tags `0x82`, `0x84` oder `0x86` entsprechen.

3 Implementation

Die Implementation der virtuellen Smartcard besteht im Wesentlichen aus drei Teilen:

- Die eigentliche virtuelle Chipkarte, bestehend aus dem Betriebssystem `SmartcardOS` und der Klasse `virtualICC` zur Kommunikation mit dem IFD-Handler.
- Dem Dateisystem
- Der Sicherheitsarchitektur, bestehend aus Secure Access Module (SAM), Card-Container und SMhandler.

3.1 Die virtuelle Smartcard

3.1.1 VirtualICC

Die Klasse `VirtualICC` dient zur Verknüpfung von der virtuellen Smartcard mit dem IFD-Handler. Die Kommunikation findet dabei über einen Socket statt. Die wichtigste Methode der Klasse ist `run`. Sie realisiert die Hauptprogrammschleife, welche Anweisungen via Socket vom IFD-Handler empfängt, an das virtuelle Smartcard Betriebsystem weiterleitet. Die Antworten werden wiederum an den IFD-Handler versendet. Das Protokoll, mit dem die beiden Parteien über den TCP-Socket kommunizieren, ist simpel gehalten und wird in Abschnitt 4.1 näher erläutert.

3.1.2 SmartcardOS

Das virtuelle Smartcard Betriebsystem sorgt für eine Interpretation der erhaltenen APDUs. Weiterhin verbindet das `SmartcardOS` das Dateisystems und das Sicherheits-Modul.

Die zentrale Methode der Klasse ist `execute`. Sie wird vom `virtualICC` gerufen, um die Abarbeitung einer APDU anzustoßen. Hierzu wird zunächst das CLA Byte der APDU verarbeitet, um festzustellen, ob Secure Messaging verwendet wird oder nicht. Ist dies der Fall, so wird die APDU zunächst vom `SMhandler` in eine ungeschützte APDU umgewandelt. Anschließend wird die APDU an das für die Abarbeitung zuständige Untermodul weitergereicht. Das Dictionary `ins2handler` übernimmt die Zuordnung eines INS zu der behandelnden Routine. Jeder Eintrag zeigt auf eine Funktion, welche ein festgelegtes Interface haben muss. Als Parameter müssen jeweils P1, P2 und die Daten der APDU entgegen genommen werden. Zurück gegeben wird eine Liste, die das Statuswort und die Daten der RAPDU enthält, die gegebenenfalls

einer leeren Zeichenkette entsprechen. Kommt es bei der Abarbeitung zu einem Fehler, so wird eine Exception des Typs `SwError` geworfen. Diese nimmt als Parameter einen String entgegen, welcher in dem Dictionary `SW` in das entsprechende hexadezimale Statuswort umgewandelt wird. Die `execute`-Funktion fängt diese Exceptions ab und generiert eine entsprechende RAPDU aus den Fehlercodes. `SwError` und `SW` sind in dem Hilfsmodul `SWutils.py` definiert. Dieses Modul enthält auch das Dictionary `SW_MESSAGES`, welches den Statuswörtern menschlich lesbare Fehlermeldungen zuordnet.

SmartcardOS

`ins2handler` ordnet über den INS jeder APDU eine Behandlungsroutine zu.

`mf` MF des Dateisystem der Smartcard

`SAM` Sicherheitsmodul der Smartcard

`atr` Answer To Reset (ATR) der Smartcard

`filename` Name der Datei, welche die Karte speichert

`execute(self, msg)` löst die Abarbeitung einer APDU (`msg`) aus. Rückgabewert ist die Zeichenkette der RAPDU.

`formatResult(self, le, data, sw, sm)` Formatiert die RAPDU als Zeichenkette und speichert eventuell auftretende Datenüberhänge.

`makeATR(**args).` fügt die verschiedenen Teile eines Answer To Reset (ATR)s zu einem die Karte repräsentierenden String zusammen und gibt diesen zurück.

`load(self,filename,password)` lädt die Kartendaten aus der Datei `filename`. `password` wird zu deren Entschlüsselung verwendet. Die Funktion setzt das Sicherheits- und Dateimodul der Smartcard auf die erhaltenen Daten.

`save(self)` speichert den Zustand der Smartcard. Dateiname und zu verwendendes Passwort müssen bereits in der Smartcard registriert sein.

Abbildung 3.1: Die Klasse `SmartcardOS` mit ihren wichtigsten Attributen und Funktionen.

Die Funktion `format_result` generiert einen RAPDU-String. Diese Zeichenkette trägt die Statusfelder sowie die Nutzdaten der Antwort. Sie ist abhängig von dem L_e-Feld der APDU, welche eine bestimmte Länge der RAPDU vorschreibt, sowie vom CLA, welches einen kryptografischen Schutz der Antwort vorgeben kann. Offensichtlich kann es dazu kommen, dass einige Nutzdaten die Größe der RAPDU überschreiten. Der überstehende Teil der Daten wird abgeschnitten und in `lastCommandOffcut`

Tabelle 3.1: Karteneigenschaften in den historical characters

Selektionsmethoden (first software function table)	Smartcard Dateisystem
Selektion mit vollem DF name	ja
Selektion mit gekürztem DF name	ja
Selektion mit Pfad von FIDs	ja
Implizite Selektion eines DF	ja
Unterstützung von short identifier	ja
Unterstützung von record number	ja
Unterstützung von record identifier	ja
Datenkodierung (second software function table)	**Smartcard Dateisystem**
EFs mit TLV Unterstützung	ja
Default für Schreibverhalten	one-time-write (siehe Abschnitt 2.3.3)
Kleinste Dateneinheit in Quartetts	2 = 1 Byte
Gültigkeit von `0xFF` des ersten BER-TLV Bytes	ungültig
third software function table	**Virtuelle Smartcard**
command chaining	nein
erweiterte APDU-Größe	nein
Zuweisung der logischen Kanäle	keine logischen Kanäle
Maximale Anzahl der logischen Kanäle	0

gespeichert. In diesem Fall ändert sich auch das Statuswort, welches nun die Länge der noch in der Smartcard gespeicherten Antwortdaten angibt (siehe Abschnitt 2.4.1). Das Statuswort der Abarbeitung der APDU wird zum späteren versenden in `lastCommandSW` gesichert.

Das Answer To Reset (ATR) wird bei der virtuellen Smartcard nur zum Kodieren der Fähigkeiten der Smartcard benutzt. Prinzipiell ist die Funktion `makeATR` in der Lage beliebige ATRs aus dem format character, den interface characters, dem Übertragungsprotokoll sowie den historischen Zusatzdaten zu konstruieren. In unserem Fall jedoch bestimmen wir kein physikalisches Verhalten der Karte und setzen daher den format character auf direct convention und betrachten nicht die interface character [ISO05a]. Für uns interessant sind die historical character, welche wir im Kompakt-TLV-Format kodieren. Dem kompakten Header mit Tag `0x7` folgt ein weiteres Quartett mit der Länge der Daten über die Fähigkeiten der Karte. Welche Eigenschaften bekanntgegeben werden und ob sie von uns umgesetzt wurden, sind in Tabelle 3.1 aufgelistet:

3.2 Smartcard Dateisystem

Alle Klassen, die für das Dateisystem der Smartcard definiert wurden, befinden sich im Paket `SmartcardFilesystem`. Die Klassen sind hierarchisch strukturiert damit sowohl Dateieigenschaften als auch Dateifunktionen an abgeleitete Klassen vererbt werden (Abbildung 3.2). So geht aus den Vorgaben des ISO-Standards beispielsweise hervor, dass sich jede Datei in einem Zustand des Lebenszyklus befindet, ein FDB besitzt und sowohl BER- als auch SIMPLE-TLV Datenobjekte speichern kann (siehe Abschnitt 2.3.2, Abschnitt 2.3.4, Abschnitt 2.3.2). Dateifunktionen, die das Zugreifen auf die TLV-Daten oder das Ver- und Entschlüsseln von Dateiattributen ermöglichen, müssen also allen Dateiobjekten zugänglich sein.

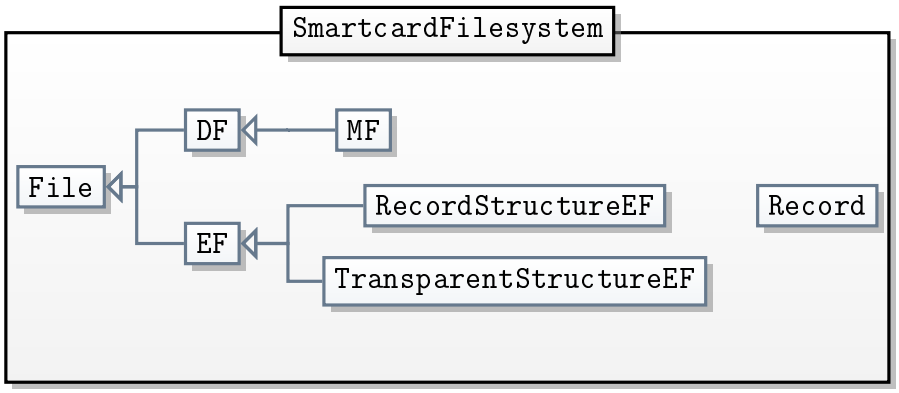

Abbildung 3.2: Klassen und Abhängigkeiten im Paket `SmartcardFilesystem`

Die Klasse `MF` stellt sowohl das Wurzel-Verzeichnis als auch die Schnittstelle des Smartcard Betriebssystem zum Dateisystem dar. Bevor wir zeigen wie über das `MF` die ISO-konformen Dateikommandos durchgeführt werden können, möchten wir einige Hilfsfunktionen sowie die Dateiklassen vorstellen.

3.2.1 Hilfsfunktionen

Das Schreiben von Daten mittels `write(old, newlist, offsets, datacoding , maxsize=None)` gibt einen String zurück, der durch das Modifizieren des Binärstrings `old` entsteht. Als Parameter werden zwei gleich lange Listen übergeben, die die zu schreibenden Datenfragmente als Binärstrings (`newlist`) und deren offsets (`offsets`) enthalten. Der Parameter `maxsize` bewirkt, dass Ergebnisse ab einer bestimmten Länge abgeschnitten werden.

Weiterhin soll das anzuwendende data coding byte gegeben werden. Mögliche Werte für `datacoding` sind (siehe auch Abschnitt 2.3.3):

`DCB["ONETIMEWRITE"]` spezifiziert das ersetzende Schreiben.

`DCB["WRITEOR"]` Das Ergebnis der logischen Oder-Verknüpfung von neuen und alten Daten soll gespeichert werden.

`DCB["WRITEAND"]` Das Ergebnis der logischen Und-Verknüpfung von neun und alten Daten soll gespeichert werden.

`DCB["PROPRIETARY"]` ist im ISO-Standard ein reservierter Wert. In unserer Implementierung wird die logische XOR-Verknüpfung von neuen und alten Daten durchgeführt. Im Befehl update record (Tabelle 2.4.2) entspricht die Spezifizierung des XOR-Schreibens genau dem Wert `DCB["PROPRIETARY"].write` sorgt also bei allen validen Schreibkommandos für das gewünschte Ergebnis.

Die Pfadselektion mittels `walk(start, path)` gibt die durch den Binärstring von FIDs spezifizierte Datei zurück. `start` ist dabei eine Datei vom Typ `DF`. Entspricht der Pfad einem leeren String, so wird `start` als Ergebnis geliefert.

Die Selektion mittels reference data object mittels `getfile_byrefdataobj(mf, refdataobjs)` gibt ähnlich wie `walk` mehrere Dateien als Ergebnis zurück. Diese werden aber mit je einem identifiziert. `refdataobjs` ist eine Liste von entsprechenden 3-Tupeln, die eine Datei entweder durch einen FID, einen SID, einen relativen oder absoluten Pfad bestimmt. Zur genauen Erklärung von reference data objects siehe [ISO05b].

3.2.2 Die Dateiklassen

`File` ist die Implementation einer Datei mit TLV-Struktur (siehe Abschnitt 2.3.2). Gleichzeitig sind alle anderen Dateitypen von `File` abgeleitet.

Von ISO-7816 ist die Funktionalität von `getenc` und `setdec` nicht vorgeschrieben, jedoch sind sie notwendig, um die Vertraulichkeit der Daten jeder einzelnen Datei zu gewährleisten. Mit diesen Funktionen kann man sicherstellen, dass das Entschlüsseln der Daten eines Dateisystems in höchstem Grade feingranular verläuft. Sowohl das Verschlüsselungsverfahren als auch die Verwaltung von Schlüsseln wird damit nicht vorgeschrieben. Es ist z. B. denkbar, dass jede Datei bzw. jeder Kontext separat verschlüsselt wird.

Der Typ `EF` dient lediglich als Vorlage für die Klassen `TransparentStructureEF` und `RecordStructureEF` und ist nicht zur eigentlichen Instanziierung gedacht. Ein EF, der einen fortlaufenden Strom von Daten speichert, d. h. von transparenter Struktur ist, wird durch ein Objekt des Typs `TransparentStructureEF` repräsentiert. Eine record-basierte Datei wird durch `RecordStructureEF` realisiert.

Jede der genannten Dateiklassen verschlüsselt vertrauliche Daten:

- In `File` und durch Vererbung auch in allen anderen Dateiklassen werden BER- und SIMPLE-TLV Datenobjekte verschlüsselt.
- In `RecordStructureEF` und `TransparentStructureEF` werden die Daten der Datei verschlüsselt, d. h. entweder das Attribut `records` oder `data`.

File
`bertlv_data` speichert in einer Liste von 3-Tupeln, die mit BER-TLV-kodierten Datenobjekte der Datei (siehe Abschnitt 2.3.2). Das Attribut ist verschlüsselt. `simple_data` speichert in einer Liste von 3-Tupeln, die mit SIMPLE-TLV-kodierten Datenobjekte der Datei (siehe Abschnitt 2.3.2). Das Attribut ist verschlüsselt. `lifecycle` enthält den Zustand des Lebenszyklus, in dem sich die Datei befindet (siehe Abschnitt 2.3.4). `parent` speichert das übergeordnete DF. Dieses Dateiattribut ist nicht von ISO-7816 vorgegeben. Es erleichtert jedoch erheblich die Organisation der Dateien. Durch ein Abfragen dieses Attributs kann man z. B. auf einfache Art das aktuelle DF ermitteln, wenn ein EF die aktuelle Datei ist oder rekursiv den Pfad zu einer Datei ermitteln. `fid` ist der FID der Datei (siehe Abschnitt 2.3.1). `filedescriptor` speichert den FDB der Datei (siehe Abschnitt 2.3.2).
`getdata(self, isSimpleTlv, requestedTL)` gibt abhängig von dem Bool `isSimpleTlv` entweder SIMPLE- oder BER-TLV-kodierte Datenobjekte zurück. Die Liste `requestedTL` spezifiziert dabei die geforderten Tags und die maximale Länge je Datenobjekt. `putdata(self, isSimpleTlv, newtlvlist)` speichert die Liste der gegebenen 3-Tupel `newtlvlist` abhängig von dem Bool `isSimpleTlv` entweder in den SIMPLE- oder BER-TLV Datenobjekten. `getpath(self)` gibt den Pfad von FIDs zu der Datei zurück. Diese Funktion wird von ISO-7816 nicht zwingend benötigt. Zur genauen Lokalisierung und Identifikation im Dateisystem, insbesondere für die Verschlüsselung, ist diese Funktion dennoch sinnvoll. `setdec(self, attribute, data)` serialisiert die gegebenen Daten und verschlüsselt sie. Anschließend wird dem Attribut `attribute` des Objekts die Chiffre zugewiesen. `getenc(self, attribute)` liefert den entschlüsselten Wert des Objektattributs `attribute`.

Abbildung 3.3: Die Klasse `File` mit ihren wichtigsten Attributen und Funktionen.

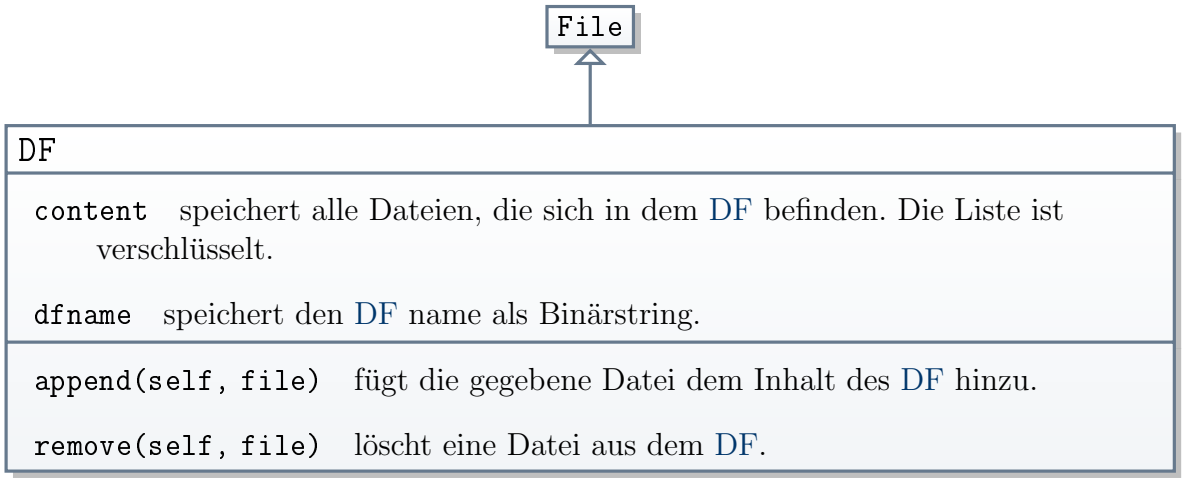

Abbildung 3.4: Die Klasse `DF` mit ihren wichtigsten Attributen und Funktionen.

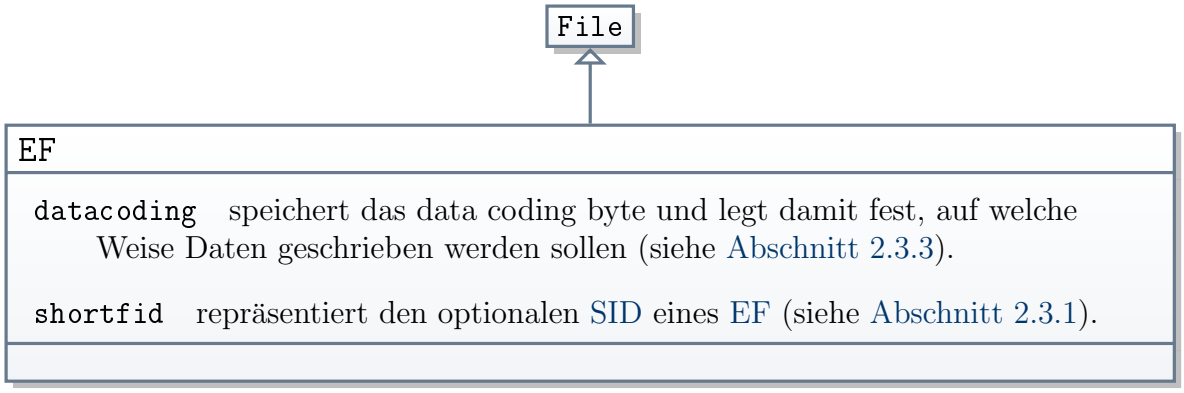

Abbildung 3.5: Die Klasse `EF` mit ihren wichtigsten Attributen und Funktionen.

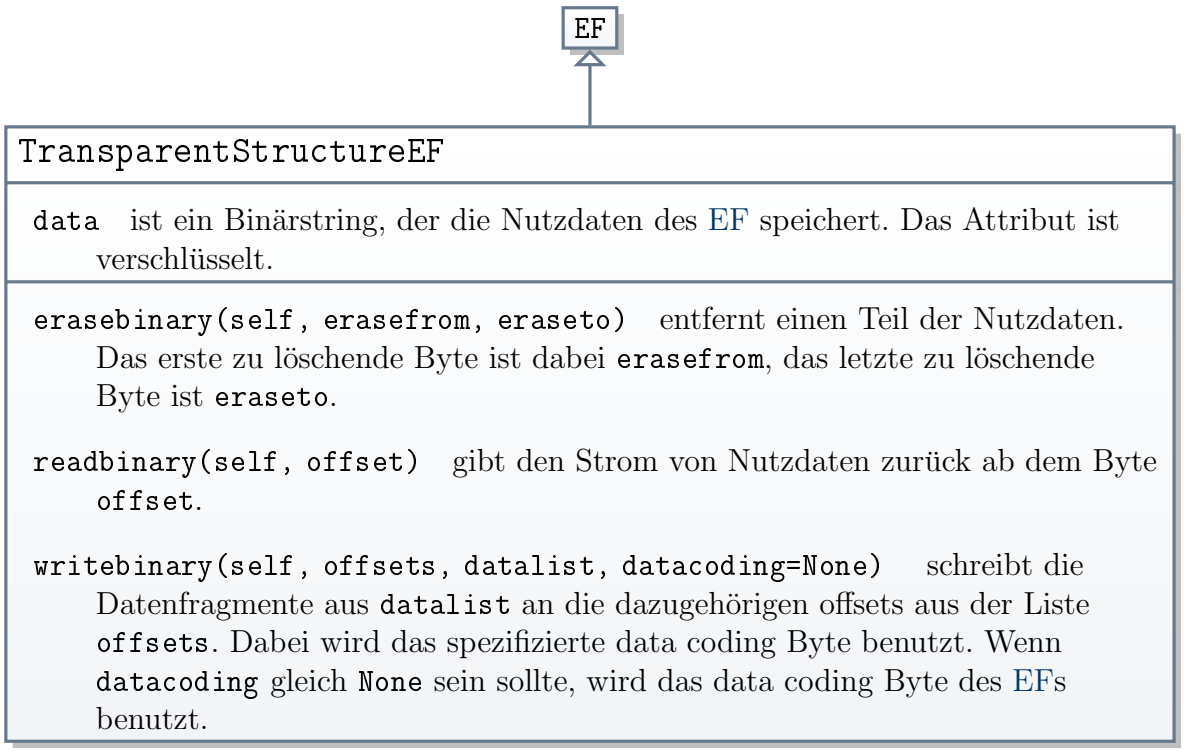

Abbildung 3.6: Die Klasse `TransparentStructureEF` mit ihren wichtigsten Attributen und Funktionen.

- In einem `DF` wird der Inhalt des Verzeichnisses verschlüsselt, so dass beispielsweise eine Anzeige des Verzeichnisinhalts ohne Freigabe des DFs nicht möglich ist.

Es ist also potentiell möglich, durch ein geeignetes Schlüsselmanagement flexible und zugleich hohe Sicherheitsrichtlinien zu gewährleisten.

3.3 Interface zum Dateisystem nach ISO-7816

Das MF ist bezüglich seiner Dateieigenschaften weniger interessant. Im Wesentlichen ist es ein DF mit dem FID `0x3F00`. In meiner Implementation jedoch stellt die Klasse `MF` gleichzeitig das Interface zum Dateisystem der Smartcard dar.

Jede Funktion der ISO-Schnittstelle wird aufgerufen mit zwei Parameterbytes und dem binären Datenstring der command APDU. Die Funktionen geben dann einen Ganzzahlwert von zwei Bytes Länge, die Statusbytes, und einen String mit binären Antwortdaten zurück. Im Falle eines Fehlers wird ein `SwError` geworfen, der einen Fehlercode in Form entsprechender Statusbytes und die dazugehörige Fehlermeldung enthält.

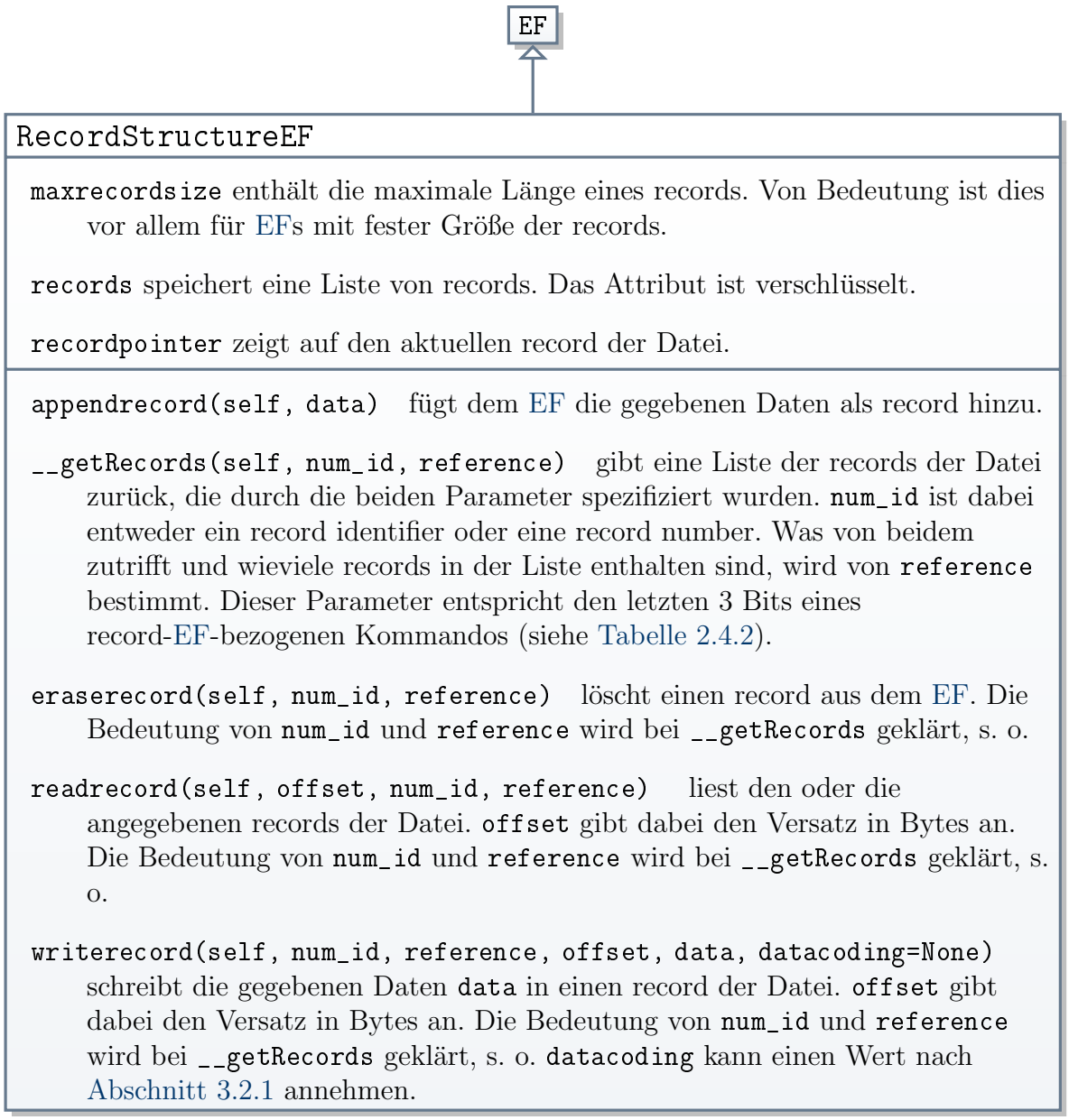

Abbildung 3.7: Die Klasse `RecordStructureEF` mit ihren wichtigsten Attributen und Funktionen.

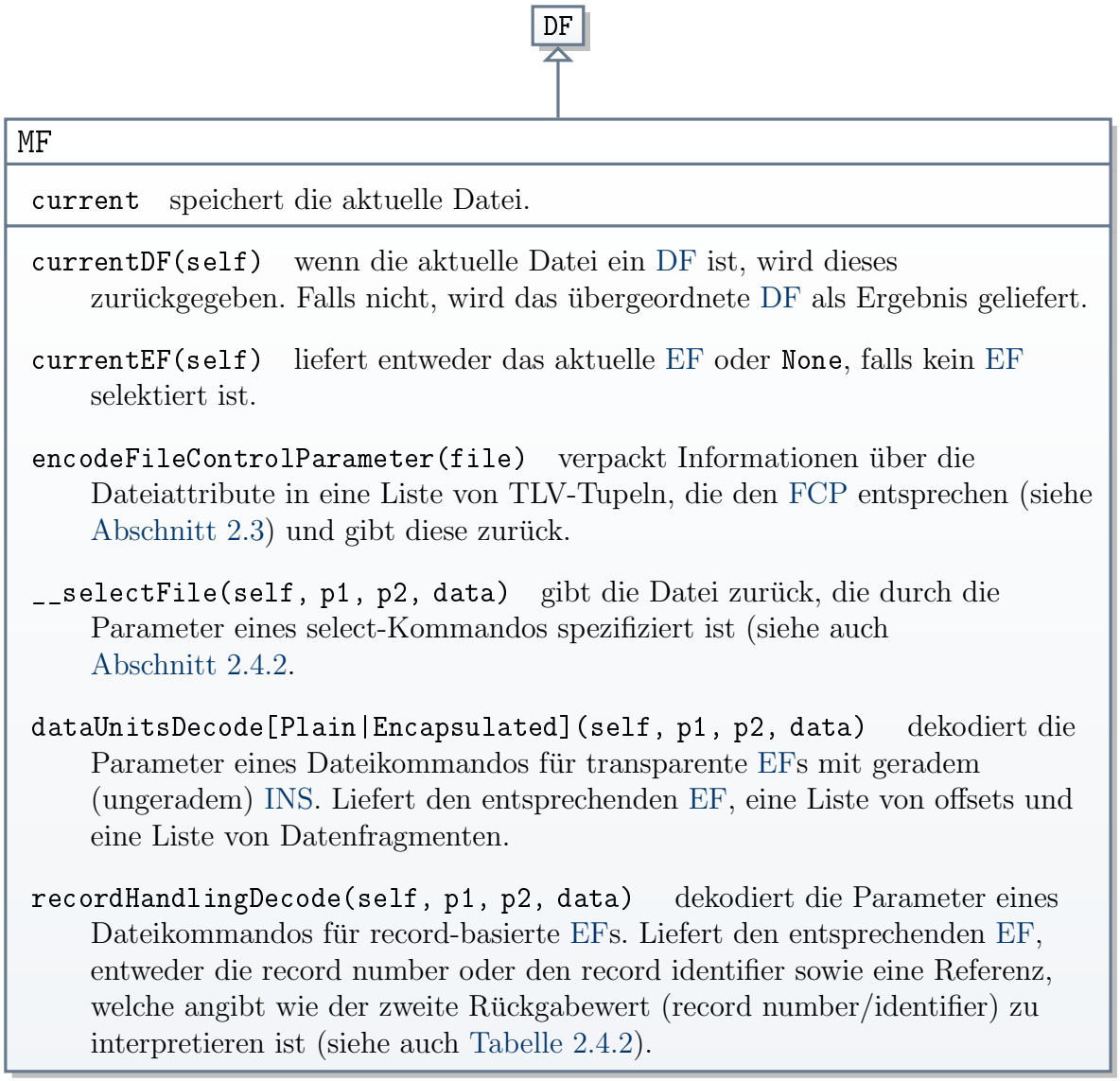

Abbildung 3.8: Die Klasse `MF` mit ihren wichtigsten Attributen und Funktionen (ausgenommen der Kommandos bzgl. ISO-7816).

Die Selektion einer Datei durch `selectFile` setzt die spezifizierte Datei als aktuelle und gibt, sofern gefordert, deren FCP in den Daten der Antwort zurück. Die Funktion implementiert das select-Kommando (vergleiche Abschnitt 2.4.2).

Zunächst wird dabei die Unterfunktion `__selectFile` aufgerufen, welche aus `p1`, `p2` und `data` die gesuchte Datei in dem MF findet und zurückgibt. `__selectFile` sucht dabei, je nach Selektierungsmethode (siehe Abschnitt 2.4.2), entweder direkt im Inhalt des MF oder rekursiv in den Inhalten der DFs des gegebenen Pfads nach der Datei. Falls gefordert, werden die Dateieigenschaften des Ergebnisses dann als FCI- oder FCP-template kodiert und in den Datenstring der Antwort geschrieben. Wenn jeder Teilschritt erfolgreich war, wird nun die gefundene Datei als aktuelle markiert und es werden entsprechende Statusbytes zurückgegeben.

Kommandos an transparente Dateien (siehe Punkt 2.4.2) werden durch folgende Funktionen realisiert

- `writeBinary[Plain|Encapsulated]`
- `updateBinary[Plain|Encapsulated]`
- `searchBinary[Plain|Encapsulated]`
- `readBinary[Plain|Encapsulated]`

Jedes dieser Kommandos richtet sich an eine transparente Datei und hat den selben schematischen Ablauf (siehe Abbildung 3.10). Aus der Spezifikation der ISO geht hervor, dass die Bedeutung der Parameter davon abhängig ist, ob die Instruktion gerade oder ungerade ist (Abbildung 2.4.2). Dementsprechend muss z. B. zwischen `readBinaryPlain` für ein gerades INS und `readBinaryEncapsulated` für ein ungerades INS unterschieden werden. Die Dekodierung der Parameterbytes und der Daten übernehmen die Hilfsfunktionen `dataUnitsDecodePlain` für Kommandos an transparente Dateien mit geradem INS und `dataUnitsDecodeEncapsulated` für diejenigen mit ungeradem INS.

Kommandos an record-basierte Dateien (siehe Punkt 2.4.2) werden durch die Funktionen `writeRecord`, `updateRecord[Plain|Encapsulated]`, `eraseRecord` und `readRecord[Plain|Encapsulated]` realisiert. Der prinzipielle Ablauf dieser Funktionen ist ähnelt den Aufrufen an transparente Datei. Wieder übernimmt eine Hilfsfunktion, `recordHandlingDecode`, das Dekodieren der Parameter, die in allen Kommandos an record-basierte Dateien gleich sind. Die records müssen natürlich auch je nach Bedarf ent- und verschlüsselt werden, um gelesen oder geschrieben werden zu können. Nur bei update und read record gibt es überhaupt die Möglichkeit einen geraden oder ungeraden Instruktionswert zu verwenden. Dieser Unterschied führt zu den je zwei Funktionen für update und read record, die die damit verbundenen Besonderheiten direkt behandeln.

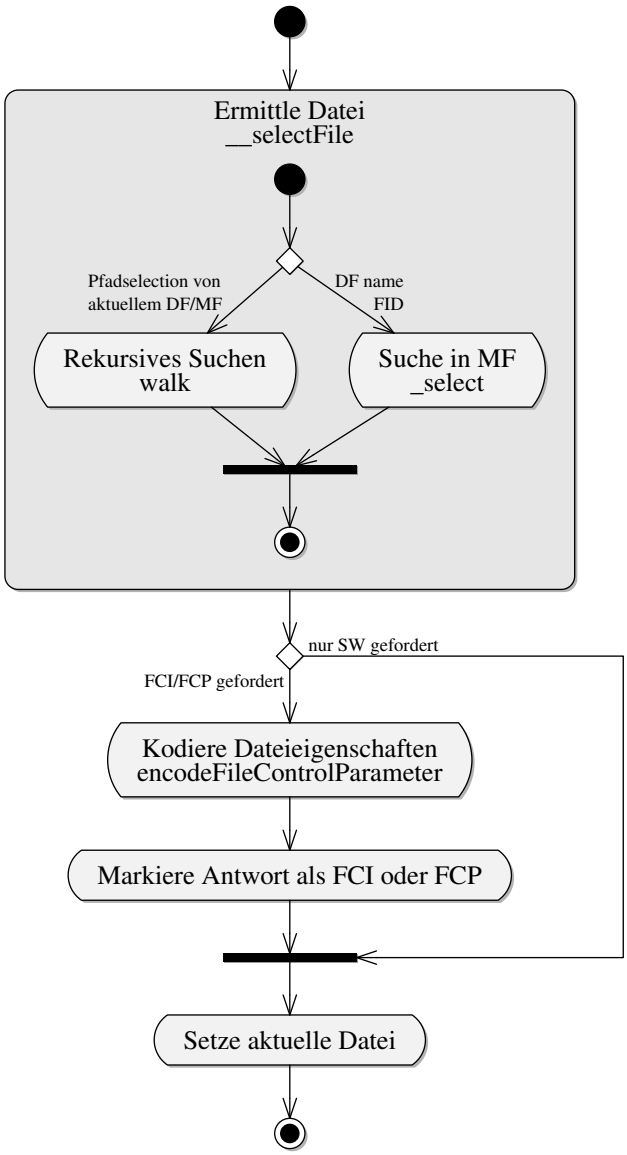

Abbildung 3.9: Schematischer Ablauf eines select-Kommandos

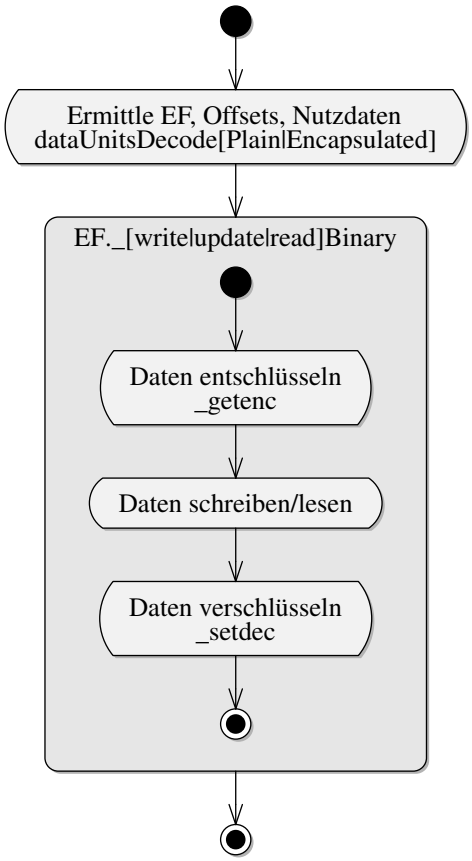

Abbildung 3.10: Ablauf von read/write/update/search binary Kommandos. Die Daten werden beim update binary Kommando als-one-time-write geschrieben. Beim write-binary-Kommando wird statt dessen das data coding byte des EFs zur Auswahl des Schreibmodus benutzt.

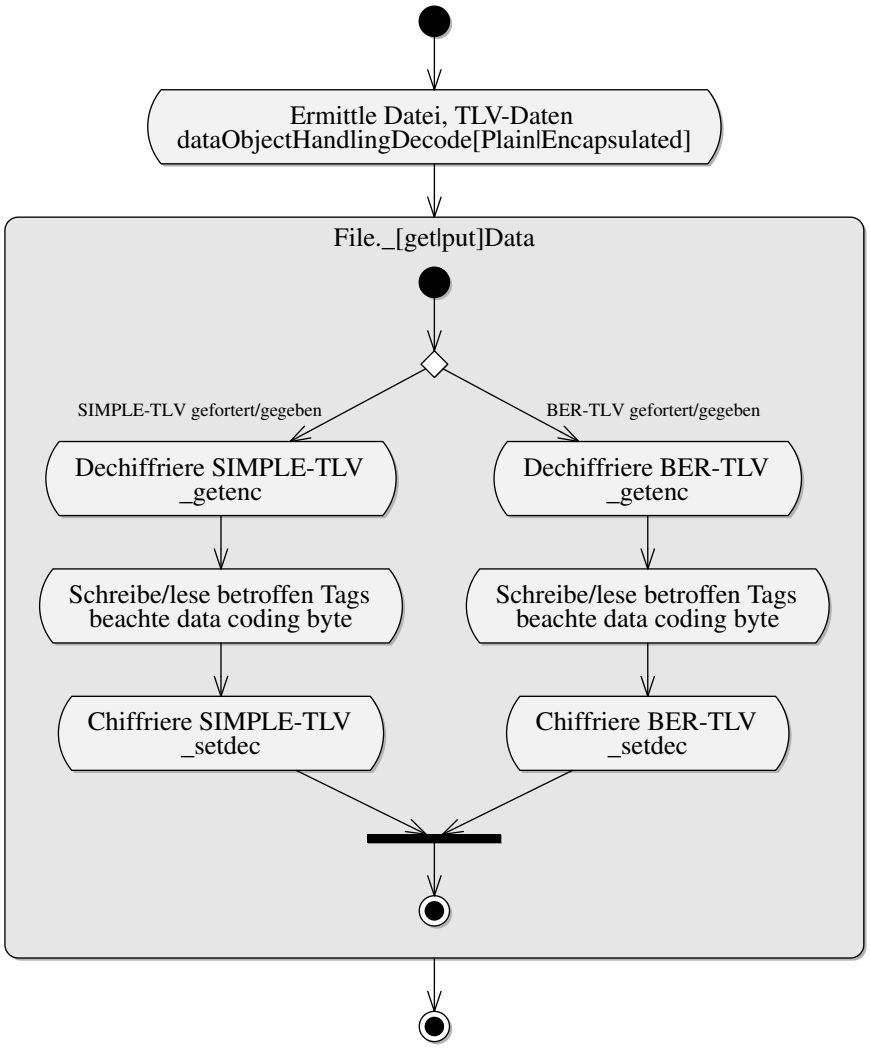

Abbildung 3.11: Ablauf von get/put data Kommandos.

Kommandos bezüglich TLV-Datenobjekte werden behandelt in den Funktionen `getData[Plain|Encapsulated]` und `putData[Plain|Encapsulated]`. Wieder wird abhängig von dem letzten Bit von INS entweder `dataObjectHandlingDecodePlain` (gerader INS) oder `dataObjectHandlingDecodeEncapsulated` (ungerader INS) zum Dekodieren der Kommandoparameter aufgerufen. In den Hilfsfunktionen wird auch ermittelt, ob es sich bei den TLV Datenobjekten um eine BER- oder SIMPLE-TLV-Kodierung handelt. Von der ermittelten Datei wird anschließend die entsprechende Methode zum Lesen oder Schreiben der TLV-Daten aufgerufen.

3.4 Sicherheitsarchitektur

Die Sicherheitsarchitektur besteht aus vier Klassen: Dem Secure Access Module Secure Access Module (SAM), dem Secure Messaging Handler, welcher auf die Klasse der Security Environment zugreift und dem `CardContainer`. Letztgenannter besitzt keine Entsprechung bei realen Smartcards sondern dient dazu Eigenschaften der Smartcard persistent zu speichern, Schlüssel auf der Karte zu verwalten und das Dateisystem der Karte im Speicher zu verschlüsseln. Das SAM dient zur Realisierung der in Abschnitt 2.5 beschriebenen Authentisierungsprozeduren, der `SMhandler` dient dazu, per Secure Messaging geschützte in ungeschütze APDUs umzuwandeln und umgekehrt. Die dafür notwendigen Parameter sind in Instanzen der Klasse `SecurityEnvironment` gespeichert. Das `SmartcardOS` muss dabei nur mit dem SAM und dem Secure Messaging Handler interagieren. Da die in Funktionalität der in ISO 7816-8 spezifizierten Kommandos auch intern für das Secure Messaging verwendet werden, wird der `SMhandler` auch zur Abarbeitung dieser Kommandos verwendet. Das SAM bietet ein Interface für den `CardContainer` an.

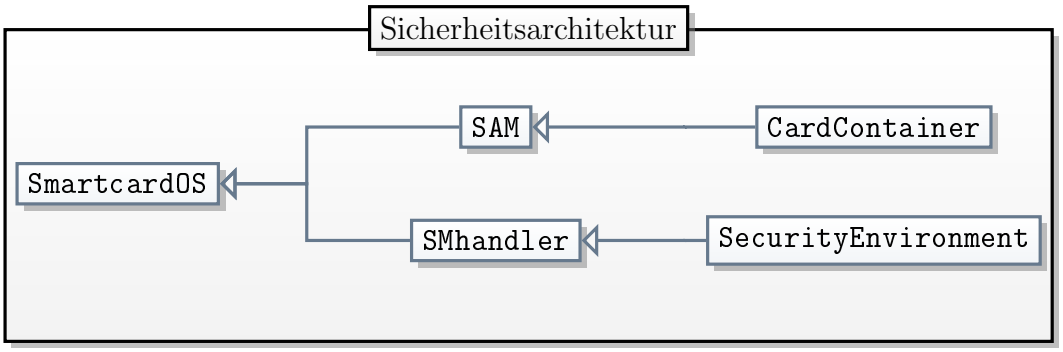

Abbildung 3.12: Klassen und Abhängigkeiten der Sicherheitsarchitektur

Nummerierung der verwendeten Algorithmen In den einzelnen Modulen der Sicherheitsarchitektur wird eine Vielzahl von kryptografischen Algorithmen verwendet. In den einzelnen Kommandos, SM data objects, und CRTs kann meist zwischen mehreren Algorithmen gewählt werden. Allerdings schreibt der ISO Standard nicht vor, welche Algorithmen zur Auswahl stehen, und wie ein bestimmter Algorithmus referenziert wird. Wir haben daher eine eigene Festlegung getroffen. Zur Zuordnung von Algorithmen zu Referenznummern dient das dictionary `ciphertable` im Modul `SAM`. Unser mapping sieht dabei wie folgt aus:

- `0x00`: DES3-ECB
- `0x01`: DES3-CBC
- `0x02`: DES-ECB

- `0x03`: DES-CBC
- `0x04`: AES-ECB
- `0x05`: AES-CBC
- `0x06`: RSA
- `0x07`: DSA

Dieses Dictionary muss zur Verwendung mit einer konkreten Anwendung, die andere Festlegungen trifft, eventuell überschrieben werden. Dies ist durch die Objektorientierung unserer Implementierung aber leicht möglich.

3.4.1 SAM und CardContainer

Die Klasse SAM

Mit dem Begriff *Secure Access Module* wird normalerweise ein Hardwaremodul bezeichnet, welches durch spezielle Maßnahmen gegen externen Zugriff gesichert ist. Man spricht auch von temper proof hardware. Durch den gesicherten Speicher einer Smartcard, welcher nicht oder nur mit hohem Aufwand von außen ausgelesen werden kann, ist eine solche Abschirmung gewährleistet. Dieser Hardwareschutz kann durch eine Softwarelösung natürlich nur bedingt abgebildet werden. Um ein möglichst hohes Schutzniveau zu erreichen, halten wir das Dateisystem der Smartcard verschlüsselt im Speicher vor und entschlüsseln die benötigten Daten erst beim Zugriff. Die Ver-, bzw. Entschlüsselung wird durch das `SAM` durchgeführt. Darüber hinaus implementiert es die Authentisierungsfunktionen von ISO 7816-4 (siehe Abschnitt 2.5) und bietet ein Interface für den `CardContainer`.

Darüber hinaus bietet das `SAM` folgende Funktionen zum Zugriff auf den `CardContainer` an:

- `saveConfiguration(self,path,password)`
- `loadConfiguration(self,path,password)`
- `loadFromDisk(self,path,password=None)`
- `saveToDisk(self,data,path,password=None)`

Die Klasse CardContainer

Der `CardContainer` dient zur sicheren Speicherung der Daten der Karte. Hierzu zählen insbesondere die verwendeten Security Environments, Schlüssel, PINs, etc. Da in dieser Klasse die Kartenschlüssel verwaltet werden, müssen alle Klassen, die Zugriff auf einen oder mehrere Schlüssel benötigen, auf ein Objekt vom Typ `CardContainer` zugreifen. Um die Daten der Karte über mehrere Programmstarts hinweg zu erhalten,

SAM	
`CardContainer`	Enthält die wesentlichen Daten und Schlüssel einer konkreten Smartcard (siehe Abbildung 3.4.1)
`internal_authenticate(self, p1, p2, data)`	Authentisierung der Karte gegenüber dem Terminal
`external_authenticate(self, p1, p2, data)`	Authentisierung des Terminals gegüber der Karte
`mutual_authenticate(self, p1, p2, data)`	Gegenseitige Authentisierung von Karte und Terminal
`verify(self, p1, p2, data)`	Überprüfung der PIN des Nutzers
`get_challenge{self,p1,p2,data)`	Anforderung einer zufälligen Nonce

Abbildung 3.13: Die Klasse `SAM` mit ihren wichtigsten Attributen und Funktionen.

kann der Container in verschlüsselter Form gespeichert werden. Darüber hinaus kann er dazu verwendet werden beliebige Daten persistent zu speichern. Diese Funktion wird zum Abspeichern des Dateisystems verwendet.

Schlüsselmanagment Der CardContainer speichert mehrere sicherheitskritische Daten, wie zum Beispiel die PIN, die Security Environments, welche wiederum den gerade geladenen Schlüssel enthalten und die Schlüssel für das Dateisystem. Um diese Daten vor Manipulation zu schützen und sie geheim zu halten, dürfen sie ausschließlich in verschlüsselter Form gespeichert werden. Dazu ist wiederum ein Schlüssel notwendig. Es zeigt sich hier das Henne-Ei-Problem: Die PIN kann nicht als Schlüssel dienen, da sie im CardContainer gespeichert ist. Außerdem ist sie zu kurz und könnte dadurch recht schnell gebrochen werden (die Sicherheit der PIN basiert darauf, dass ein Angreifer nur sehr wenige Versuche hat die korrekte PIN zu erraten). Es ist also ein zusätzliches Nutzerpasswort nötig, welches beim Laden der virtuellen Smartcard eingegeben werden muss. Aus diesem Passwort wird dann das notwendige Schlüsselmaterial abgeleitet. Hierzu wird die so genannte password based key derivation function 2 verwendet. Details hierzu finden sich im RFC 2898 (siehe [Kal00]). Das Nutzerpasswort dient sowohl zur Ableitung der Schlüssel für die persistente Speicherung von SAM und Dateisystem als auch zur Generierung der Schlüssel, um die Daten des Dateisystems im Speicher zu verschlüsseln.

Speicherung von Daten Die Daten der Smartcard werden in zwei Dateien (`*.sam` für den `CardContainer` und `*.mf` für das Dateisystem). Beim Starten der virtuellen Smartcard können die entsrechenden Dateien mit dem Parameter `-f` geladen wer-

CardContainer

`cardNumber` Die Seriennummer der Karte

`PIN` Persönliche Identifikations Nummer zur Freischaltung der Karte

`FSkeys` Dictionary, das Schlüssel für einzelne Dateien enthält. Als Zugriffsschlüssel dient der Pfad der Datei.

`cipher` Spezifiziert den verwendeten Verschlüsselungsalgorithmus.

`block_size` Blockgröße von `cipher`

`master_password` Nutzerpasswort, das zur Gewinnung der Schlüssel verwendet wird

`master_key` Aus dem Nutzerpasswort abgeleiteter Schlüssel. Wird zur Verschlüsselung von SAM und MF bei persistenter Speicherung verwendet.

`salt` Zufälliges Salz, erschwert Wörterbuch Attacken

`asym_key` Objekt, das den öffentlichen und privaten Schlüssel enthält, falls für die Authentifizierungsfunktionen des SAM ein assymetrisches Verfahren verwendet wird.

`loadConfiguration(self, path, password)` lädt die Konfiguration der Karte aus einer `*.sam`-Datei, entschlüsselt sie und wendet sie auf den aktuellen CardContainer an.

`saveConfiguration(self, path, password)` speichert die aktuelle Konfiguration des CardContainer verschlüsselt in eine `*.sam`-Datei.

`saveToDisk(self,data,path,password)` speichert beliebige Daten, welche als String in `data` übergeben werden in verschlüsselter Form mit einem HMAC versehen in einer Datei ab.

`loadFromDisk(self,path,password=None,loading_sam=False)` lädt von `saveToDisk` gespeicherte Daten, entschlüsselt sie, prüft ihren HMAC und gibt sie im Erfolgsfall als String zurück.

Abbildung 3.14: Die Klasse `CardContainer` mit ihren wichtigsten Attributen und Funktionen.

den. Sind keine Dateien mit dem angegeben Namen vorhanden, so wird eine neue virtuelle Smartcard mit default Werten erzeugt und beim Programmende unter dem angegeben Namen abgespeichert. Wurde kein Dateiname angegeben so werden die Kartendaten verworfen.

Beide Dateien werden verschlüsselt gespeichert. Der zu verwendende Algorithmus wird in der Variable `cipher` der CardContainer Klasse festgelegt. Als Schlüssel wird der in Abbildung 3.4.1 beschriebene Master Key, welcher aus dem Nutzerpasswort abgeleitet wird, verwendet. Die zu speichernde Datei beginnt mit dem Klartextstring `$p5k2$`, welcher andeuten soll, dass der Schlüssel mittels password based key derivation function 2 gewonnen wurde. Nach diesem String wird das verwendete salt in Dollarzeichen eingeschlossen gespeichert. Erst danach folgen die eigentlichen verschlüsselten Daten. Nach den Daten folgt ein HMAC zur Integritätssicherung.

Wenn verschlüsselte Daten aus einer Datei geladen werden, wird zunächst mit Hilfe des Salzes und des Nutzerpasswortes ein entsprechender Schlüssel abgeleitet. Anhand des HMACs kann anschließend die Integrität der Daten überprüft werden. Stimmt der errechnete HMAC nicht mit dem in der Datei gespeicherten überein, so wurde die Datei entweder manipuliert, oder der Nutzer hat ein falsches Passwort eingegeben.

3.4.2 Secure Messaging Handler und Security Environments

SM handler Die Klasse des Secure Messaging Handlers stellt die Funktionen, welche für das Secure Messaging benötigt werden, zur Verfügung. Außerdem sind auch die Kommandos nach ISO 7816-8 in dieser Klasse realisiert, da sie, genau wie die Secure Messaging Funktionen, auf die Security Environments zugreifen und sehr ähnlich wie die Funktionen zum erzeugen der SM data objects arbeiten.

Die Hauptaufgabe dieser Klasse ist es, mit Secure Messaging geschützte Kommando APDUs zu verifizieren und zu entschlüsseln, beziehungsweise ungesicherte Antwort-APDUs zu schützen. Deutet das Class Byte einer APDU an, dass Secure Messaging verwendet wird, so wird die komplette APDU von `SmartcardOS` mit der Methode `parse_SM_CAPDU` an den Secure Messaging handler übergeben. Dort werden sukzessive alle im Datenfeld enthaltenen SM data objects ausgewertet. Der Parameter `header_authentication` steuert dabei, ob der Header der APDU in die Berechnung von Prüfsummen einfließt oder nicht. Kommt es bei der Auswertung zu einem Fehler, etwa weil die erforderlichen Parameter in der Security Environment nicht gesetzt sind oder eine Prüfsumme nicht stimmt, so wird eine Exception geworfen, die im `SmartcardOS` die Ausgabe einer entsprechenden RAPDU auslöst. Können alle SM data objects erfolgreich bearbeitet werden, so wird eine ungeschützte CAPDU zurückgegeben, welche dann ganz normal vom `SmartcardOS` verarbeitet wird.

Verwendet eine CAPDU Secure Messaging, so muss auch die dazugehörige RAPDU mit den in der Security Environment spezifizierten Mechanismen geschützt werden. Dazu wird die ungeschützte RAPDU vom `SmartcardOS` durch die Methode `protect_response` an den Secure Messaging Handler übergeben. Dort werden die Daten in Secure Messaging Objekte gekapselt und diese werden zu einer neuen, geschützen RAPDU zusammengefasst. Diese wird an das Betriebssystem zurückgege-

ben und von dort an das Terminal gesendet.

Neben diesen Hauptaufgaben übernimmt die SM handler auch die Verwaltung der Security Environments. Über das manage security environment Kommando können die Befehle zum Speichern, Löschen und Laden eine SE aufgerufen werden. Abgelegt werden sie in dem Dictionary `SEcontainer` innerhalb des `CardContainer`s. Um auf den `CardContainer` zugreifen zu können, muss im `SMhandler` eine Referenz auf das SAM vorhanden sein. Als Schlüssel für `SEcontainer` dient dabei das SEID Byte der jeweiligen SE. Durch die Speicherung im `CardContainer` wird sichergestellt, dass die Security Environments auch über mehrere Programmaufrufe hinweg erhalten bleiben. Das Setzen der Parameter selber erfolgt über einen Aufruf der `mse` Funktion des entsprechenden `SecurityEnvironment` Objekts (siehe unten).

Security Environments Die Repräsentation der SEs selbst erfolgt über eine eigene Klasse names `SecurityEnvironment`. Wie aus 2.9 ersichtlich ist, sieht der ISO Standard für jede SE vier Komponenten vor: Eine für Secure Messaging in der CAPDU , eine für Secure Messaging in der RAPDU , eine für Computation, decipherment, internal authentication and key agreement und die letzte für verification, encipherment, external authentication and key agreement. Die letzten beiden Komponenten dienen zur Speicherung für die Parameter für die Kommandos nach ISO 7816-8. Die Klasse `SecurityEnvironment` bildet jeweils eine dieser Komponenten ab. Sie verfügt lediglich über eine Methode `mse(self,config)`, welcher die im String `config` angegebene Konfiguration anwendet. Auch im Konstruktor kann bereits eine Konfiguration angegeben werden. Intern wird die Hilfsklasse `crt` verwendet, um die einzelnen control reference templates zu implementieren.

Secure Messaging
`current_SE` Die aktuell geladene Security Environment
`manage_security_environment(self, p1, p2, data)` Security Environment, laden, speichern, löschen und verändern (siehe Tabelle 2.6)
`parse_SM_CAPDU(self,CAPDU,header_authentication)` Umwandlung einer geschützen in eine ungeschütze Kommando-APDU. Fehler bei der Integritätsprüfung führen zur Erzeugung einer entsprechenden RAPDU, durch das Werfen einer Exception.
`protect_response(self,sw,result)` Umwandlung einer ungeschützten in eine geschütze RAPDU
`perform_security_operation(self,p1,p2,data)` kann eine Viezahl kryptografischer Funktionen durchführen (siehe 29)
`manage_security_environment(self,p1,p2,data)` verwaltet die in der Karte gespeicherten Security Environments
`generate_public_key_pair(self,p1,p2,data)` dient zum Erzeugen eines neuen asymmetrischen Schlüsselpaars, oder zum Zugriff auf ein zuvor erzeugtes
`__store_SE(self,SEID)` speichert die aktuelle SE mit dem Identifier `SEID` in `CardContainer`
`__restore_SE(self,SEID)` lädt die SE mit dem Identifier `SEID` aus `CardContainer` und macht sie zu aktuell aktiven SE
`__erase_SE(self,SEID)` löscht die SE mit dem Identifier `SEID` aus `CardContainer`

Abbildung 3.15: Die Klasse `Secure Messaging` mit ihren wichtigsten Attributen und Funktionen.

4 Weitere Komponenten

4.1 IFD-Handler: Der virtuelle Smartcard Reader

PC/SC ist eine systemunabhängige Schnittstelle, die es Anwendungen erlaubt eine Smartcard anzusprechen, ohne Kenntnis hardwarespezifischer Details der Smartcard oder des Readers. Das M.U.S.C.L.E Framework stellt eine freie Implementierung dieser API dar. Ein Daemon Programm koordiniert hier das (Ent-) Laden von Gerätetreibern für Smartcard Reader, sogenannten IFD-Handlern. Der Daemon übersetzt die Befehle der PC/SC-API?? in eine spezielle IFD-API??. Der IFD-Handler wiederum übersetzt von der IFD-API zu hardwarespezifischen Befehlen eines Smartcard Readers. Die Aufgabe besteht nun also darin, einen Treiber für einen virtuellen Smartcard Reader umzusetzen. Das virtuelle Lesegerät soll das "Einlegen" und "Entfernen" virtueller Smartcards unterstützen und eine Kommunikation zu ihnen ermöglichen.

Erfolgreich wurde ein IFD-Handler umgesetzt, der die notwendigen Funktionen der IFD-API unterstützt. Wird der Treiber geladen, erstellt dieser einen Socket und über den entsprechenden Port kann sich die virtuelle Smartcard mit dem virtuellen Reader verbinden. Die bidirektionale Kommunikation läuft in einem einfachen Datenformat ab: Die Länge einer Nachricht wird ohne Vorzeichen mit drei Bytes kodiert, dann folgen entsprechend viele Bytes der eigentlichen Nachricht (siehe Tabelle 4.1). Per Polling wird die Kommunikation vom IFD-Handler gesteuert.

Die Umsetzung ist im Detail relativ simpel gehalten. So reagiert der Reader bei einer gescheiterten Übertragung zur Smartcard beispielsweise einfach mit dem Trennen der Verbindung. Weiterhin, wenn das Längen- oder Nachrichtenfeld gesendet bzw. empfangen werden soll, wird dies solange probiert bis alle Bytes übertragen wurden oder ein Fehler bei `send` bzw. `recv` auftritt. Dies kann unter Umständen

Tabelle 4.1: Nachrichtenformat zwischen virtuellem Smartcard Reader und virtueller Smartcard.

IFD-Handler				Virtuelle Smartcard			
Längen-Bytes			Nachrichten-Bytes	Längen-Bytes			Nachrichten-Bytes
`0x00`	`0x00`	`0x00`	leer	`0xXX`	`0xXX`	`0xXX`	ATR
`0x00`	`0x00`	`0x01`	`0x00`	Smartcard wird ausgeschaltet, keine Antwort			
`0x00`	`0x00`	`0x01`	`0x01`	Smartcard wird eingeschaltet, keine Antwort			
`0xXX`	`0xXX`	`0xXX`	APDU	`0xXX`	`0xXX`	`0xXX`	RAPDU

zu einer Endlosschleife führen, wenn einer der Kommunikationspartner nicht das korrekte Nachrichtenformat einhält. Wir nehmen dieses Problem in Kauf, da zur Kommunikation nur die von uns bereitgestellten Programme vorgesehen sind.

4.2 USB Schnittstelle

In einem Anwendungsszenario (siehe Abschnitt 5.1.1) soll ein mobiles Gerät per USB an einen Desktop-PC geschlossen werden. Auf dem mobilen Gerät sind im M.U.S.C.L.E Framework verschiedene Smartcard Lesegeräte (insbesondere der virtuelle Smartcard Reader, siehe Abschnitt 4.1) registriert, die an den PC weitergeleitet werden sollen.

Das Programm `ccid` übernimmt genau diese Aufgabe. Per Programmparameter legt der Benutzer beim Starten fest, an welchen lokalen Smartcard Reader die USB-Anfragen geleitet werden. Nur die USB-Befehle zum Ein- und Ausschalten der Smartcard, sowie übermitteln von APDUs und zur Abfrage des Status des Readers werden von `ccid` bearbeitet. Eine Antwort wird dabei direkt ermittelt oder das Programm vermittelt zwischen dem lokalen Lesegerät und dem USB-Kommunikationspartner.

Die Seriennummer kann vom Benutzer von `ccid` explizit gesetzt werden; standardmäßig wird sie dem USB-Kommunikationspartner randomisiert übermittelt. Obwohl es an der Abarbeitung des Programms nichts ändert, kann der Benutzer explizit eine vendor und product ID setzen. Diese Identifikatoren werden bei der Initialisierung der USB-Verbindung übertragen und legen fest, welcher Treiber auf der Gegenseite geladen wird (standardmäßig Smart Enterprise Guardian). Man kann also, sollte ein Treiber auf dem PC nicht vorhanden sein, auf dem PC das Laden eines anderen erzwingen. Bei der Auswahl der Treiber sollte man jedoch beachten, dass `ccid` nur basale USB-Befehle beherrscht und diese streng zur Spezifikation des USB Implementers Forum[USB05] interpretiert und umsetzt.

Technisch wurde das Programm `ccid` mit Hilfe des Kernelmoduls GadgetFS implementiert. Aus dem User-Space verbindet sich `ccid` mit dem Kernelmodul und initialisiert über Dateioperationen das (emulierte) USB-Gerät. Über die als Dateien registrierten Endpunkte kann nun `ccid` mit einem Programm kommunizieren, das auf das USB-Gerät zugreift, wobei GadgetFS als Mittler zwischen ihnen fungiert. Welche Daten über die Endpunkte transportiert werden, legt GadgetFS nicht fest. Insbesondere die Device und Interface Deskriptoren sind also frei wählbar.

Direkt im Anschluss an den Deskriptor, der den Typ des Geräts (in diesem Falle “Integrated Circuit(s) Cards Interface Devices”) deklariert, wird ein spezieller Smart Card Device Class Descriptor übermittelt. Für dieses Projekt wurde dafür der Datentyp `struct ccid_class_descriptor` erstellt. Dieser spezifiziert konform zu [USB05] die Eigenheiten des Smartcard Readers. Das Protokoll zur USB-Kommunikation ist paketbasiert. Für jeden Pakettyp wurde ein eigener `C`-Typ definiert (siehe Abbildung 4.1). Die Funktion `parse_ccid` liest einen Puffer, der einen zu [USB05] konformen Befehl enthält, und gibt einen Puffer mit der entsprechenden Antwort zurück. Die Behandlung der Anfragen geschieht in Unterfunktionen, die beispielsweise die

nötigen PC/SC-Anfragen durchführen (siehe Abbildung 4.1).

ccid.h

`struct ccid_class_descriptor` identifiziert einen Smartcard Reader mit seinen Eigenschaften, wie z. B. den Protokollfähigkeiten, Stromeigenschaften oder der maximalen Anzahl der Slots zu einer Karte.

`PC_to_RDR_XfrBlock_t` übermittelt eine APDU zur Smartcard und erwartet eine response APDU als Antwort.

`PC_to_RDR_IccPowerOff_t` schaltet die eingelegte Smartcard aus.

`PC_to_RDR_GetSlotStatus_t` ermittelt den Status des Slots. (Behandlungsroutine nicht implementiert)

`PC_to_RDR_GetParameters_t` ermittelt die aktuellen Slot Parameter. (Behandlungsroutine nicht implementiert)

`PC_to_RDR_ResetParameters_t` setzt die Slot Parameter zurück auf ihre default-Werte. (Behandlungsroutine nicht implementiert)

`PC_to_RDR_SetParameters_t` legt die Slot Parameter fest. Abhängig, ob T=1 oder T=0 gewählt wird, ändert sich die Interpretation des Parameters `abProtocolDataStructure`. Dafür wurden zwei weitere Typen definiert, `abProtocolDataStructure_[T0|T1]_t`. (Behandlungsroutinen nicht implementiert)

`PC_to_RDR_IccPowerOn_t` ist die Anweisung zum Einschalten der eingelegten Smartcard oder zum warmen Reset, falls sie bereits aktiviert ist. Die Antwort enthält das ATR der Karte.

`RDR_to_PC_SlotStatus_t` wird gesendet als Antwort auf `PC_to_RDR_IccPowerOff_t` und `PC_to_RDR_GetSlotStatus_t` und gibt Aufschluss darüber ob eine Smartcard eingelegt und aktiviert.

`RDR_to_PC_DataBlock_t` wird gesendet als Antwort auf `PC_to_RDR_IccPowerOn_t` und `PC_to_RDR_XfrBlock_t` und übermittelt Daten an die USB-Gegenstelle (hier ATR oder response APDU).

`perform_PC_to_RDR_`[Typ des Pakets] führt PC/SC Aufrufe aus, um die zu [USB05] konformen Anfragen durchzuführen und gibt Antwortpakete konform zu [USB05] zurück.

`parse_ccid` liest einen Puffer von Bytes und ruft entsprechend des Nachrichtentyps (`bMessageType`) eine Behandlungsroutine auf. Das dort ermittelte Ergebnis wird als Puffer von `parse_ccid` zurück gegeben.

Abbildung 4.1: Datentypen und Funktionen für die Kommunikation per USB

5 Diskussion

5.1 Anwendungsszenarien/Demos

Um die Funktionsfähigkeit unseres Systems zu erproben, haben wir zwei Demoanwendungen damit realisiert. Zum Einen einen Smartcard-basierten Login mittels des Programms `smartcard−login`, zum anderen die Emulationen eines elektronischen Reisepasses.

5.1.1 Smartcard Login

`smartcard−login` ist ein PAM für den Smartcard-basierten Login von Nutzern unter Linux. Es wurde von Martin Sägesser und Mario Strasser im Rahmen einer Projektarbeit an der Züricher Universität Winterthur entwickelt (siehe [SS01]). Das Programm ist auf die Verwendung von Smartcards vom Typ Cryptoflex des Herstellers Schlumberg ausgelegt.

Wir wollen das Programm mit unserer virtuellen Smartcard nutzen, um uns mittels des OpenMoko an Rechnern anzumelden, welche für die Verwendung von `smartcard −login` eingerichtet sind. Dafür mussten wir einen neuen Kartentyp implementieren, welcher die Eigenheiten der Cryptoflex Karte berücksichtigt. Dieser neue Typ erbt von den bereits vorhandenen Modulen und erweitert bzw. überschreibt deren Funktionalität, wo dies notwendig ist. Es bleibt anzumerken, dass wir nicht die komplette Funktionalität einer Cryptoflex Karte implementiert haben, sondern lediglich die zur Verwendung von smartcard-login notwendigen Funktionen. Im Folgenden beschreiben wir kurz die Personalisierung einer Smartcard und den Ablauf des eigentlich logins, sowie einige Implementationsdetails des neuen Kartentyps.

Funktionsweise von `smartcard−login`

`makecard` ist das Kommando im `smartcard−login` Paket, das zur Personalisierung der Smartcard dient. Es erzeugt die notwendigen Datenstrukturen und Schlüssel auf der Smartcard und hinterlegt den öffentlichen Schlüssel des Benutzers in der Datei `/etc/pubkey`.

Zum Aufruf des Kommando `makecard` ist mindestens der Parameter `name` notwendig. `makecard −−name=root` personalisiert die Karte für den Nutzer `root`. Außerdem ist es meist noch notwendig, die PIN der Karte anzugeben. Dies geschieht mittels des Parameter `pin`. Ein typischer Aufruf von makecard sieht folgendermaßen aus: `makecard −−name=fm −−pin=1234`. Wie man sieht wird die PIN im Klartext auf der Konsole eingegeben und somit auch in der History der Shell gespeichert!

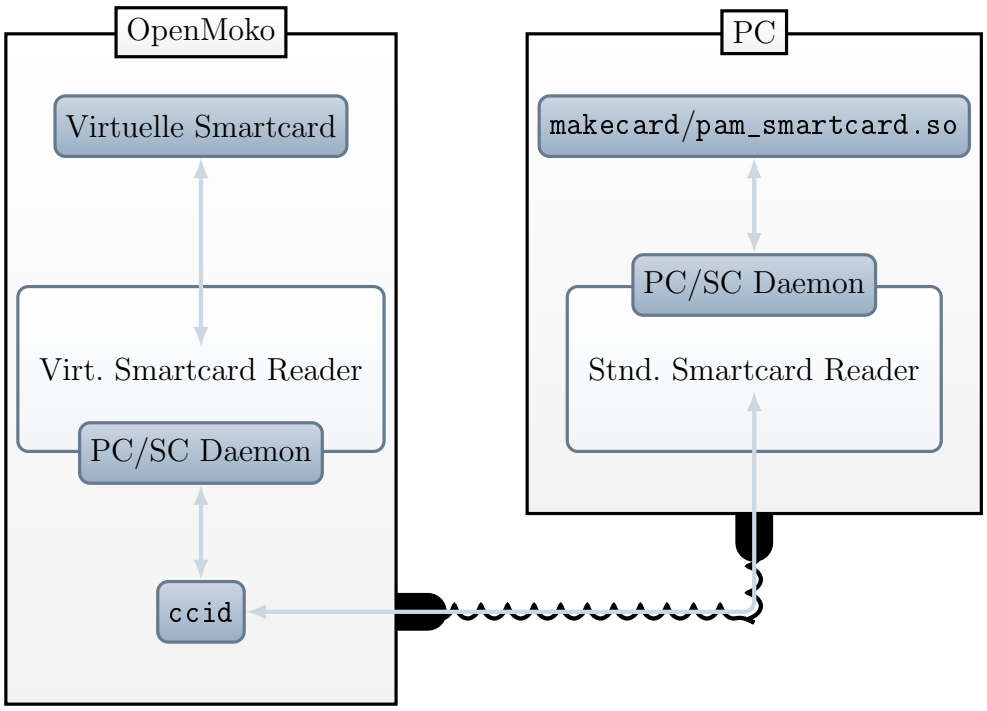

Abbildung 5.1: Komponenten beim Anmelden mit einer emulierten Smartcard an einem standard PC

Auf der Karte werden dabei RSA Schlüssel erzeugt. Dies geschieht mittels des generate public key Kommandos (siehe Punkt 2.8). Als öffentlicher Exponent wird hierbei immer 65537 gewählt. In der response APDU schickt die Karte den öffentlichen Schlüssel an den Computer.

Funktionsweise des PAM Liegt eine personalisierte Karte vor, so kann diese zum Login genutzt werden. Dafür muss sie zunächst mit der zur Karte gehörenden PIN freigeschaltet werden. Meldet sich der Nutzer an, so wird er also in diesem Szenario nicht nach seinem Passwort sondern nach der PIN seiner Smartcard gefragt. Hat er seine PIN eingegeben, so wird diese verwendet, um mittels des verify-Kommandos die Karte freizuschalten. Danach erfolgt die eigentliche Nutzerauthentisierung mit einem internal-authenticate-Kommando. Hierbei wird überprüft, ob die eingelegte Karte den passenden privaten Schlüssel zum in `/etc/pubkey` gespeicherten öffentlichen Schlüssel besitzt. Bei Verwendung von smartcard-login und Cryptoflex Karten ist hierbei implizit bekannt, dass es sich um RSA Schlüssel handelt. Die vom Terminal (in diesem Fall der Computer, auf dem der Login stattfinden soll) generierte Nonce wird mit dem auf der Karte gespeicherten privaten Schlüssel signiert. Da der Computer über den öffentlichen Schlüssel verfügt, kann er die Signatur überprüfen und somit Karte und Nutzer authentifizieren.

Implementation Geringfügige Änderungen mussten an generate public key und internal authenticate vorgenommen werden. Zum einen sendet die Cryptoflex Karte ihre Daten für diese beiden Kommandos in big-endian-Byteordnung (also mit dem höchstwertigen Bit zuerst). Die Daten müssen also vor der Weiterverwendung (und vor dem Verschicken der Antwort) umgedreht werden. Zum anderen muss der Schlüssel, der durch generate public key erzeugt wird, in einem festgelegten (zuvor durch `makecard` angelegten) transparenten EF gespeichert werden.

Emulation eines Dateisystems der Cryptoflex Smartcard

Wir werden am Beispiel des Dateisystems für die Simulation einer Cryptoflex Smartcard zeigen, dass die Objektorientierung des Smartcard Dateisystems es ermöglicht, durch leichte Modifikationen die Anforderungen eines anderen Smartcardtyps zu erfüllen. Das liegt zum einen daran, dass eine Cryptoflex Smartcard ähnlich der ISO-Spezifikation arbeitet. Zum anderen liegt der Grund in dem modularen Aufbau der Klassen innerhalb des Dateisystems und dem Kapseln wesentlicher Arbeitsschritte in unabhängige Funktionen.

Die Klasse `CryptoflexMF` wird abgeleitet von `MF` und erbt zunächst jede (auch private) Klassenfunktionen. Das Interface zu den Dateikommandos aus dem ISO-Standard bilden also die selben Funktionen wie auch bei `MF` (vergleiche Abschnitt 3.3). Im Folgenden wollen wir nur die von mir beispielhaft umgesetzten Dateioperationen zeigen. Eine vollständige Umsetzung eines Dateisystems für die Cryptoflex Smartcard liegt außerhalb des Fokus dieser Arbeit.

Kommandos an record-basierte Dateien Die Kommandos create, read record sind eine Untermenge der Kommandos von ISO-spezifizierten Kommandos.

Erstellen einer Datei Die Cryptoflex Smartcard definiert ein eigenes Interface zur Dateierstellung. Hier werden die verschiedenen Dateiattribute nicht in ein FCP template verpackt, sondern es wird ein etwas statischerer Ansatz umgesetzt. Übermittelt wird von Kartenlesegerät ein Datenstring bestimmter Länge, bei dem jedes Byte eine bestimmte Dateieigenschaft kodiert.

Dennoch ist fest zu halten, dass die Kodierung der Dateieigenschaften der einzige Unterschied zur ISO-Spezifikation ist. Weiterhin soll die erstellte Datei dem aktuellen DF hinzugefügt werden und zudem als aktuelle Datei markiert sein. Folglich musste in `CryptoflexMF` lediglich die Hilfsfuntkion `create` umgeschrieben werden, weil sie die Kommandoparameter interpretiert und ein Dateiobjekt daraus erstellt. Bei dem Modifizieren von `create` musste darauf geachtet werden, dass sowohl das Interface als auch die Rückgabewerte äquivalent zu denjenigen in `MF` sind. Andernfalls würde das Zusammenspiel mit den von `MF` abgeleiteten Funktionen nicht mehr funktionieren.

Elementary Files Hier zeigt sich, dass die Cryptoflex Smartcard sich zwar im Groben an ISO-7816 hält, jedoch mehrere kleine Besonderheiten hat.

Eine transparente Datei kann in der Cryptoflex mittels read und update binary angesprochen werden. Bei beiden Befehlen ist jeweils nur der gerade INS erlaubt. Die Cryptoflex interpretiert die Parameterbytes *immer* als offset, wobei die ISO-Normierung hier auch die Kodierung eines SID zuließe.

Ein weiterer Unterschied zeigt sich bei den Befehlen für record-basierte Dateien. Append record funktioniert in der Cryptoflex genau wie von ISO vorgeschrieben. Update und read record jedoch existieren in der Cryptoflex zum Einen nur mit geradem INS. Zum Anderen gibt es keine Selektion des records per record identifier. Die Selektion des nächsten/vorigen bzw. ersten/letzten record bezieht sich in diesem Fall auf die record number. Ist also der zweite record selektiert, so ist bei der Cryptoflex Smartcard der dritte record der nächste.

5.1.2 ePass Emulation

Genau wie die für den Smartcard Login verwendete Cryptoflex Karte wurde auch der ePass als eigener Kartentyp implementiert. Alternativ wäre es auch möglich gewesen, ihn als Applikation der ISO 7816 Karte zu realisieren. Der Pass ist als Demoanwendung besonders interessant, da er zur Zeit das einzige im Einsatz befindliche System ist, welches Secure Messaging verwendet. Das gibt uns die Möglichkeit unsere Implementierung des Secure Messaging zu testen.

Da wir nicht über die von der Bundesregierung geschützten Schlüssel verfügen, konnten wir die so genannte Extended Access Control nicht implementieren. Diese wird verwendet, um den Zugriff auf die im ePass gespeicherten Fingerabdrücke zu regeln. Unsere Implementierung enthält daher keine Fingerabdrucksdaten sondern lediglich die Datengruppen 1 und 2 (Passdaten des Inhabers und digitales Passfoto), sowie das Inhaltsverzeichnis EF.COM und das Sicherheitsdatenobjekt EF.SOD. Alle Daten sind in transparenten EFs gespeichert und dürfen nach der Basic Access Control gelesen werden.

Basic Access Control Die Basic Access Control (BAC) dient der Etablierung von Sitzungsschlüsseln, welche später im Secure Messaging verwendet werden. Durch die BAC wird der Zugriff auf Passdaten und Passbild freigeschaltet. Sie kann daher mit dem Vorzeigen des Passes an der Grenze verglichen werden. Notwendig für die Durchführung ist lediglich die Kenntniss der maschienenlesbaren Zone (MRZ), welche auf den Pass aufgedruckt wird. Diese MRZ ist in einem speziellen OCR Font gedruckt, so dass sie an einem Terminal automatisch eingelesen werden kann. Aus der eingelesenen MRZ werden zweimal zwei DES Schlüssel K_A und K_B gebildet. Diese dienen später zur Generierung eines Sitzungsschlüssels K_{Enc} für Ver- bzw. Entschlüsselung und zum Bilden eines Sitzungsschlüsses K_{Mac} für das Berechnen von Message Authentcation Codes (MAC). Zur Bildung dieser Schlüssel wird zunächst ein Seed K_{Seed} aus der MRZ gebildet. Dieser wird für K_{Enc} mit dem 32 Bit Wert `0x01` und für K_{Mac} mit `0x02` konkateniert. Über diese Werte wird jeweils ein SHA Hash gebildet. Die ersten 8 Byte der Ausgabe werden jeweils als K_A, die zweiten als K_B verwendet.

Die so gewonnen Schlüssel werden verwendet, um in einem leicht abgeänderten Mutual Authenticate Verfahren session keys zwischen Terminal und Pass zu vereinbaren. Dies läuft folgendermaßen ab:

Das Terminal fordert von der Karte via get challenge eine Nonce RND.ICC an und generiert selbst eine Nonce RND.IFD. Zusätzlich generiert es zufälliges Schlüsselmaterial K.IFD und konkateniert die drei Werte zur Botschaft S. S wird mittels des Schlüssel K_{ENC} verschlüsselt und über das Kryptogramm wird ein MAC mittels K_{Mac} gebildet. Kryptogramm und MAC werden aneinander gehängt und an die Karte übermittelt. Diese überprüft die Integrität der Nachricht anhand des MAC und entschlüsselt im Erfolgsfall das Kryptogramm. Sie überprüft, ob es den korrekten Wert für RND.ICC enthält. Ist dies der Fall, so erzeugt sie zufälliges Schlüsselmaterial K.ICC und bildet die Nachricht RND.ICC | RND.IFD | K.ICC. Wieder wird die Nachricht verschlüsselt und mit einem MAC versehen an das Terminal übermittelt. Anschließend werden K.ICC und K.IFD per XOR miteinander verknüpft. Auf das Ergebnis wird dasselbe Verfahren wie zur Gewinnung der ersten Schlüssel angewendet (Konkatenation mit `0x01`, bzw. `0x02` und Bilden eines Hash-Wertes). Das Ergebnis sind die Sitzungsschlüssel K_{Mac} und K_{ENC}, welche für das Secure Messaging, das ab sofort verwendet werden muss, eingesetzt werden.

Der Pass verwendet folgende Secure Messaging Objekte:

- Alle Daten, die in der Kommando oder Antwort-APDU übertragen werden, müssen in Kryptogramme mit padding-content indicator Byte (siehe Tabelle 2.7) mit dem Tag `0x87` gekapselt werden.
- Sowohl Kommando- als auch Antwort-APDU müssen alle Nachrichten mit kryptographischen Prüfsummen absichern (Tag `0x8E`).
- Falls das Terminal Daten von der Karte anfordert, so muss das Le Byte der Kommando-APDU in ein SM data object mit dem Tag `0x97` gekapselt werden.
- Die beiden Statuswörter der Antwort-APDU müssen in ein SM data object mit dem Tag `0x97` gekapselt werden.

Für die Verschlüsselung der Daten wird der Algorithmus 3DES verwendet. Die Daten werden vorher mit ISO Padding aufgefüllt (padding-content indicator Byte ist `0x01`, siehe Tabelle 2.7). Für die Berechnung der Prüfsummen wird ein session sequence counter verwendet. Der Anfangswert für diesen Zähler ergibt sich aus den 4 niederwertigsten Byte RND.ICC konkateniert mit den 4 niederwertigsten Byte von RND.IFD. Vor der Berechnung der Prüfsumme wird der aktuelle Zählerwert mit den Daten, für die eine Prüfsumme berechnet werden soll, konkateniert. Anschließend wird ein MAC über die daten berechnet und der Zähler um eins erhöht. Als Blockchiffre zur Berechnung des MAC wird DES mit einem Nullvektor als IV und ISO Padding verwendet.

Datenstrukturen Folgende Datenstrukturen sind auf dem deutschen ePass gespeichert:

- EF.COM enthält die auf dem Pass gespeicherten Datenobjekte. Enthalten sein können 14 durch die ICAO spezifizierte Datengruppen. Im deutschen ePass werden allerdings nur DG1 und DG2 verwendet.
- DG1 enthält die Daten der maschienenlesbaren Zone des Passes. Diese enthält dieselben Informationen, welche auf der Vorderseite des Passes abgedruckt sind.
- DG2 enthält das Passfoto im jpeg2000 Format. Das Foto ist eine eine CBEFF (common biometrics exchange format framework) Datenstruktur verpackt.
- EF.SOD enthält Hashwerte über die auf dem Pass gespeicherten Daten. EF.SOD ist signiert, um seine Integrität zu sichern. Die ganze Struktur ist in einem PKCS-7 Container gespeichert.

Der genaue Aufbau der einzelnen Datenstrukturen ist in [ica04b] und [ica04a] festgelegt und soll hier nicht weiter erörtert werden. Auf eine Implementation von EF.SOD haben wir verzichtet, da wir dafür keine gültige Signatur hätten erzeugen können. Es wäre allerdings möglich gewesen, ein selbstsigniertes EF.SOD zu erzeugen, welches von einem Terminal, welches Signaturen nicht vorschriftsgemäß überprüft, als gültig anerkannt werden würde. Die notwendigen Daten werden beim Start der virtuellen Smartcard im Konstruktor erzeugt und in die entsprechenden EF s geschrieben.

5.2 Probleme/Verbesserungsmöglichkeiten

Auf Grund des Umfangs des ISO Standard 7816 konnten nicht alle spezifizierten Funktionen implementiert werden. Vor allem fehlt eine Implementierung der (recht komplexen) Zugriffskontrollmechansimen. Diese müssten bei einem Ausbau des Projekts noch hinzugefügt werden. Allgemein könnte die Architektur für die Sicherheitsmechanismen überarbeitet werden, sie ist in ihrer jetzigen Form eher unelegant und weist unnötig viele Abhängigkeiten zwischen den einzelnen Komponenten auf.

Das Testen der virtuellen Smartcard erfolgte zum einen durch in die Module eingebettete Testfälle, zum anderen durch die Verwendung von externen Programmen wie z.B. `opensc-explorer`. Um eine höhere Codequalität zu erreichen, wären weitere Mechanismen wie automatisierte Regressionstest oder Unit Tests wünschenswert. Außerdem konnten wir leider kein Programm zum Testen des Secure Messagings finden, bis auf den ePass, welcher nur einen Bruchteil der Funktionalität verwendet.

Der Hauptkritikpunkt, welcher gegen den Einsatz einer virtuellen Smartcard spricht, ist sicherlich das Aufweichen der hohen Sicherheitsstandards einer realen Smartcard. Auf die verschlüsselten Daten der Karte kann ein Brute Force Angriff gestartet werden. Die Sicherheit der Daten hängt somit stark von der Qualität des Nutzerpasswortes ab. Zur Laufzeit liegen die Daten zwar die meiste Zeit verschlüsselt im Speicher,

jedoch muss auch der Schlüssel selbst vorgehalten werden, damit der Nutzer nicht bei jedem Kommando sein Passwort erneut eingeben muss. Das Schlüsselmanagement kann noch weiter ausgebaut werden, es wird zur Zeit noch keine Verwendung von der Möglichkeit unterschiedliche Schlüssel für unterschiedliche Dateien zu verwenden gemacht.

Für die bequeme Verwendung der virtuellen Smartcard wäre noch ein Zusatzprogramm zum Anlegen und Personalisieren von Kartenobjekten wünschenswert. Bis jetzt sind Kartendaten, wie etwa die PIN oder die Kartennummer, hart in den Quelltext des Programmes einkodiert. Für das Anlegen und Auslesen von Dateien existieren allerdings schon Lösungen, zum Beispiel der bereits erwähnte `opensc-explorer`

Mögliche Probleme der Implementierung des Dateisystems Die Signaturen einiger Funktionen zum Zugreifen auf die Daten erscheinen etwas ungewöhnlich. Sie sind stark auf die dazugehörigen APDU-Kommandos zugeschnitten (vergleiche hierzu Abbildung 3.6 und Abbildung 2.4.2 oder Abbildung 3.7 und Punkt 2.4.2). Dies könnte in manchen Fällen dazu führen, dass Instanzen dieser Klassen etwas schwer zu handhaben sind. Aber in dem Kontext eines Dateisystems einer Smartcard sollten die Funktionen genügen, weil sich das Gros der Smartcardhersteller im Kern nach ISO-7816 richtet. Ein Beispiel dafür gibt die Emulation der Cryptoflex (siehe Abbildung 5.1.1).

Falls es dennoch notwendig sein sollte, die Dateiklassen selbst zu modifizieren, so müsste man von `MF` die Funktion `create` derart modifizieren, dass hier angepasste Dateiobjekte erzeugt werden. Diese geänderten Klassen lassen sich z. B. durch Ableiten und Modifizieren der Dateiklassen erhalten.

5.3 Fazit/Weiterentwicklung

In seiner jetzigen Form bietet unser System die Möglichkeit, Smartcardmechanismen einfach und günstig zu nutzen. Notwendig ist lediglich ein Handy, das die benötigte Software unterstützt. Weitere Möglichkeiten würden sich durch die Nutzung der NFC Technik ergeben. Sie würde sowohl eine drahtlose Kommunikation mit kontaktlosen Terminals ermöglichen, als auch die Verwendung des Handys selbst als Terminal für real existierende, drahtlose Smartcards. Die von uns gewählte Architektur sollte es einfach machen, diese zusätzliche Technik zu unterstützen. Auch die Unterstützung anderer Smartcardtypen ist, wie unsere Anwendungsdemos gezeigt haben, leicht möglich, vorausgesetzt sie weichen nicht allzu stark vom ISO Standard ab. Dank der Virtualisierung braucht ein Anwender nicht mehr mehrere reale Smartcard für je eine Anwendung, sondern kann auf einem einzigen Endgerät eine Vielzahl an virtuellen Smartcards für unterschiedliche Verwendungszwecke speichern. Desweiteren kann die virtuelle Smartcard dazu dienen, Smartcard-basierte Anwendungen einfach zu testen, ohne im Besitz von physikalischen Smartcards zu sein.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[CR07] Corcoran, David und Ludovic Rousseau: *MUSCLE PC/SC Lite API. Toolkit API Reference Documentation 0.9.1.* Technischer Bericht, Januar 2007. verfügbar als http://pcsclite.alioth.debian.org/pcsc-lite/.

[CR08] Corcoran, David und Ludovic Rousseau: *MUSCLE PC/SC IFD Driver API 3.3.0.* Technischer Bericht, Januar 2008. verfügbar als http://pcsclite.alioth.debian.org/ifdhandler-3/.

[Deu08] Deutsche Bahn AG, 2008. http://www.touchandtravel.de.

[ica04a] *Development of a logical data structure - LDS.* Technischer Bericht, ICAO, Mai 2004.

[ica04b] *PKI for Machine Readable Travel Documents offering ICC Read-Only Access.* Technischer Bericht, ICAO, Oktober 2004.

[ISO04] ISO 7816-8: *Identification cards Integrated circuit(s) cards with contacts - Part 8: Interindustry commands for a cryptographic toolbox.* ISO/IEC, 2. Auflage, November 2004.

[ISO05a] ISO 7816-3: *Identification cards — Integrated circuit cards - Part 3: Electronic Signals and Transmission Protocols.* ISO/IEC, 2. Auflage, Januar 2005.

[ISO05b] ISO 7816-4: *Identification cards — Integrated circuit cards - Part 4: Organization, security and commands for interchange.* ISO/IEC, 2. Auflage, Januar 2005.

[Kal00] Kaliski, B.: *PKCS # 5: Password-Based Cryptography Specification Version 2.0.* RSA Laboratories, September 2000. http://tools.ietf.org/html/rfc2898.

[Ope08] Openmoko Inc., 2008. http://www.openmoko.com.

[RE02] Rankl, Wolfgang und Wolfgang Effing: *Handbuch der Chipkarten.* Carl Hanser Verlag, 4. Auflage, 2002.

[SS01] Sägesser, Martin und Mario Strasser: *Linux Login mit RSA SmartCard.* Projektarbeit, Züricher Hochschule Winterthur, Mai 2001.

[USB00] USB Implementers Forum: *Universal Serial Bus Specification Revision 2.0*, April 2000. verfügbar als http://www.usb.org/developers/docs/usb_20_122208.zip.

[USB05] USB Implementers Forum: *Universal Serial Bus. Device Class: Smart Card. CCID Revision 1.1*, April 2005. verfügbar als http://www.usb.org/developers/devclass_docs/DWG_Smart-Card_CCID_Rev110.pdf.

[Wel08] Welte, Harald, 2008. http://www.openpcd.org/openpicc.0.html.