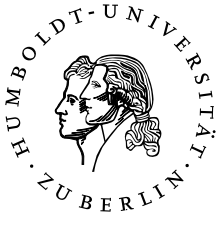
Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik



Host-based Card Emulation einer PKCS15-kompatiblen Smartcard

Bachelorarbeit

zur Erlangung des akademischen Grades
B. Sc.

eingereicht von: Erik Nellessen
geboren am: 02.08.1989
geboren in: Meerbusch

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

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

Inhaltsverzeichnis

1 Einleitung

In dieser Bachelorarbeit wird gezeigt, wie man eine (kontaktlose) Smartcard, die zum Entschlüsseln und Signieren von E-Mails eingesetzt wird, durch ein Smartphone ersetzen kann. Auf diese Art und Weise wird dem Nutzer das vertrauliche und authentische Verschicken und Empfangen von E-Mails in einfacherer Weise zugänglich gemacht. Da Smartphones weit verbreitet sind, muss der Nutzer sich in der Regel nur noch einen Kartenleser beschaffen und kann auf den Kauf einer Smartcard verzichten. Der Aufwand zum Nutzen von authentischen und vertraulichen E-Mails wird so herabgesetzt. Es wird in dieser Bachelorarbeit aber auch darauf eingegangen, welche Schwachstellen sich auf einem Smartphone gegenüber einer Smartcard auftun.

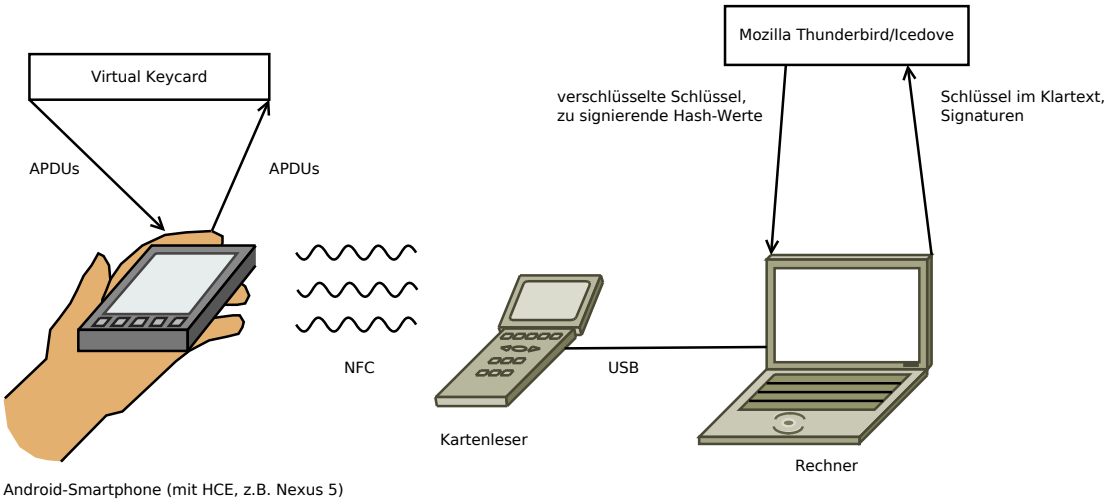


Abbildung 1: Vereinfachte Darstellung des behandelten Anwendungsfalls.

Es wird ein Software-System vorgestellt, das das Signieren und Entschlüsseln von E-Mails mit Hilfe von einem Smartphone über die NFC-Schnittstelle ermöglicht. Dazu wurde eine App namens „Virtual Keycard“ entwickelt. Mit Hilfe von Virtual Keycard lassen sich RSA-Schlüssel erzeugen und Daten mit diesen Schlüsseln entschlüsseln und signieren. Außerdem bietet die App die Möglichkeit, die NFC zum Rechner vertraulich und authentisch aufzubauen.

Mit Android 4.4 (Android Kitkat) erhalten Smartphones das Host-based Card Emulation-Feature (HCE). Dieses ermöglicht die Nutzung der NFC-Hardware der Smartphones durch App-Programmierer in einfacher Weise. Das bedeutet, dass Apps mit einem NFC-Kartenleser und somit mit der dahinter liegenden Software kommunizieren können. Dies ist auch bei Virtual Keycard der Fall.

Auf der Rechner-Seite wurde die bestehende Software erweitert, sodass von hier aus die NFC mit der App Virtual Keycard vertraulich und authentisch aufgebaut werden kann. Alternativ dazu wäre es auch möglich gewesen, einen Komfort-Leser zu verwenden, der die NFC bereits von sich aus vertraulich und authentisch aufbauen kann. Will man einen solchen Leser verwenden, sind noch einige Anpassungen in der Software des in dieser Bachelorarbeit vorgestellten Systems notwendig.

Man erhält ein System, mit dem man einige Vorteile des ursprünglichen Smartcard-Systems nutzen kann, so wird z. B. der private Schlüssel getrennt aufbewahrt und kann nicht ohne Weiteres vom Rechner aus gelesen werden.

2 Entschlüsseln und Signieren von E-Mails mit dem RSA-Kryptosystem

Im RSA-Kryptosystem besteht der Schlüsselraum aus Schlüsselpaaren, die aus einem öffentlichen und einem privaten Schlüssel bestehen. Verschlüsseln mit dem öffentlichen Schlüssel und Entschlüsseln mit dem privaten Schlüssel sind dabei zueinander inverse Operationen (siehe [Sti06, S. 173]), d. h. es gehören nur solche Schlüsselpaare zum Schlüsselraum, für die diese Eigenschaft gegeben ist. Das RSA-Kryptosystem kann dazu benutzt werden, E-Mails vertraulich und authentisch zu versenden. Um Vertraulichkeit zu gewährleisten werden E-Mails verschlüsselt und entschlüsselt. Um Authentizität zu gewährleisten werden Signaturen von E-Mails erstellt und diese verifiziert.

Wie auch in anderen asymmetrischen Kryptosystemen besitzt jeder Teilnehmer einen öffentlichen und einen privaten Schlüssel. Den öffentlichen Schlüssel benutzen Teilnehmer, um E-Mails zu verschlüsseln, die dann nur der Besitzer des dazugehörigen privaten Schlüssels (mit Hilfe ebendieses Schlüssels) entschlüsseln kann.

Dabei wird aber nicht die gesamte E-Mail per RSA-Verfahren ver- und entschlüsselt. Stattdessen wird die E-Mail mit einem symmetrischen Kryptoverfahren verschlüsselt und der benutzte Schlüssel, den man noch per RSA verschlüsselt, an die Mail angehangen [Hou09, S. 25-26].

Der private RSA-Schlüssel kann auch zum Signieren von E-Mails verwendet werden, die Signatur wird dann von den anderen Teilnehmern mit dem öffentlichen Schlüssel verifiziert. Dabei wird nicht der Text der Mail selbst, sondern ein Hash-Wert (mit Padding) davon signiert [Hou09, S. 14-15]. Die Krypto-Operationen sowie das Padding des zu signierenden Hash-Werts folgen dabei dem PKCS1-Standard [Lab12]. Man signiert eine E-Mail, indem man ihren Hash-Wert (inklusive Padding) mit dem privaten Schlüssel entschlüsselt. Andere Teilnehmer können die Signatur überprüfen, indem sie den Hash-Wert der E-Mail berechnen und den entschlüsselten Hash-Wert mit dem öffentlichen Schlüssel verschlüsseln. Erhalten sie in beiden Fällen das gleiche Ergebnis, sehen die Teilnehmer die Signatur als korrekt an (da Ver- und Entschlüsseln in dieser Form wie oben bereits erwähnt inverse Operationen sind).

Es kann nachteilig sein, zum Entschlüsseln und Signieren von E-Mails den gleichen privaten Schlüssel zu benutzen. Man sollte daher abwägen, ob man nicht besser zwei verschiedene private Schlüssel verwendet. Für eine Diskussion dieser Frage siehe beispielsweise [Sta14].

In beiden Fällen ist wichtig, dass der öffentliche Schlüssel auch zu der Person gehört, zu der er angeblich gehören soll, d. h. dass nur diese Person den dazugehörigen

privaten Schlüssel besitzt. Um dies sicherzustellen, wird es im Anwendungsfall, der hier behandelt werden soll, d. h. S/MIME (siehe [RT10]), eine sogenannte Public Key Infrastruktur (PKI) eingesetzt [Eck08, S. 385 ff.]. Dazu gehören Certificate Authorities (CAs), die die Zugehörigkeit von Personen zu Schlüsseln mit bestimmten Policies überprüfen [Eck08, S.381 ff.]. Die Zertifikate von sogenannten Root-CAs werden an die Teilnehmer verteilt und lokal gespeichert. Die Teilnehmer vertrauen all den öffentlichen Schlüsseln, die von einer der Root-CAs unterschrieben wurden.

Gelingt es einem Angreifer, an den privaten Schlüssel eines Teilnehmers zu gelangen, ist er offensichtlich in der Lage, selbst E-Mails zu entschlüsseln und zu signieren. Die Vertraulichkeit und Authentizität der E-Mails des Teilnehmers, dessen privater Schlüssel gestohlen wurde, ist also nicht mehr gewährleistet. Daher ist es nötig, den privaten Schlüssel besonders zu schützen [Eck08, S. 393].

3 Einsatz von Smartcards zum Signieren und Entschlüsseln von E-Mails

Bei Smartcards handelt es sich um Rechensysteme in einem handlichen Format, die spezifische Aufgaben erfüllen. In der Regel können auf ihnen Schlüssel erzeugt werden. Die Schlüssel können anschließend zum Signieren und Entschlüsseln von E-Mails eingesetzt werden, wobei der private Schlüssel die Karte (im Idealfall) niemals wieder verlässt (oft wird der Schlüssel direkt auf der Karte erzeugt und steht daher niemals auf einer anderen Hard- oder Software zur Verfügung). Außerdem kann der private Schlüssel auf der Karte durch eine PIN geschützt werden.

Zum Signieren und Entschlüsseln von E-Mails per Smartcard geht man wie folgt vor:

1. Erzeugen des Schlüsselpaars (öffentlicher und privater Schlüssel) auf der Karte.

2. Erzeugen eines Certificate Signing Request (CSR) aus dem öffentlichen Schlüssel auf der Karte.

3. Der CSR wird an eine CA geschickt, diese unterschreibt ihn und schickt das unterschriebene Zertifikat zurück. Mit dem Zertifikat bezeugt die CA die Bindung des öffentlichen Schlüssels an einen Namen. Das heißt, dass sie überprüft hat, ob der öffentliche Schlüssel auch tatsächlich zu der im CSR angegebenen Person gehört und zu einem positiven Ergebnis gekommen ist.

4. Verteilen des Zertifikats an die anderen Teilnehmer (öffentlich).

Zum Signieren schickt man die zu signierenden Daten an die Karte und gibt die PIN des privaten Schlüssels ein, die Karte signiert die Daten und schickt die Signatur zurück.

Zum Entschlüsseln schickt man die zu entschlüsselnden Daten an die Karte und gibt die PIN des privaten Schlüssels ein, die Karte entschlüsselt die Daten und schickt die entschlüsselten Daten zurück.

Durch dieses Vorgehen wird garantiert, dass der private Schlüssel die Karte niemals verlässt, er also nie auf dem Rechner, auf dem die E-Mails empfangen und versendet werden, eingelesen wird. Trotzdem können E-Mails authentisch und vertraulich verschickt und empfangen werden.

Das Konzept der Smartcard soll garantieren, dass ein Angreifer, selbst wenn er Zugriff auf den Rechner erhält, nicht ohne Weiteres E-Mails signieren oder entschlüsseln kann. Zum Signieren oder Entschlüsseln von E-Mails müsste ein Kartenleser angeschlossen sein, an dem die Smartcard angeschlossen ist, und die PIN müsste eingegeben werden. Wenn der Angreifer die PIN nicht kennt, kann er selbst mit Zugriff auf den Rechner und die angeschlossene Smartcard die Vertraulichkeit und Authentizität von E-Mails nicht gefährden. Das liegt daran,

dass auf der Smartcard ein PIN-Management enthalten ist, dass die Smartcard sperrt, wenn die PIN zu oft falsch eingegeben wird.

Da die E-Mails auf dem Rechner aber im Klartext vorliegen, sobald sie entschlüsselt wurden, kann er an dieser Stelle an geheime Informationen gelangen, wenn er Kontrolle über den Rechner erlangt. Auch könnte der Angreifer versuchen, E-Mails zu signieren oder entschlüsseln, sobald der Nutzer sich mit der PIN auf der Karte authentisiert hat. Diese Angriffe können auch von einer Smartcard nicht verhindert werden.

Weitere Informationen zu Smartcards finden sich im Standard ISO/IEC 7816. Zur Kommunikation mit Smartcards stehen weitere Informationen im Standard ISO/IEC 7816-4[fS13], in diesem werden auch die Strukturen von Application Protocol Data Units (APDUs) beschrieben.

4 Absicherung von NFC mit Hilfe des PACE-Protokolls

Das Entschlüsseln von E-Mails soll mit Hilfe eines Smartphones geschehen, das per NFC (Near Field Communication) mit einem NFC-Leser kommuniziert.

Der Zugriff auf NFC steht prinzipiell jedem (mit der nötigen Hardware) offen, der sich in der Nähe (d. h. in bis zu 1m Entfernung) der kommunizierenden Hardware befindet. Daher ist es notwendig, für die NFC Vertraulichkeit und Authentizität zu sichern. Im Fall des Entschlüsselns und Signierens von E-Mails werden als vertrauliche Daten die PIN und die entschlüsselten symmetrischen Schlüssel, die zur Entschlüsselung der E-Mails gebraucht werden, per NFC übertragen. Authentisch übertragen werden sollten die Hash-Werte der zu signierenden E-Mails und die zu entschlüsselnden Daten, sodass nur solche Krypto-Operationen durchgeführt werden, die mit einer PIN authorisiert wurden.

In dem im Zuge dieser Bachelorarbeit entworfenen System wird zur Sicherung der Vertraulichkeit und Authentizität von PIN, symmetrischen Schlüsseln und zu entschlüsselnden oder zu signierenden Daten das PACE-Protokoll (Password Authenticated Connection Establishment, siehe [WG310]) benutzt. Dieses sichert die Authentizität der Teilnehmer durch die Kenntnis eines gemeinsamen Geheimnisses (der PIN) und baut einen vertraulichen Kanal auf, der mit einem deutlich komplexeren Schlüssel als der PIN gesichert ist.

Auf der Rechner-Seite wird die PIN vom Benutzer eingegeben. Auf der App-Seite ist die PIN persistent gespeichert (kann aber im Zuge des PIN-Managements auch geändert werden), hier wird sie ausgelesen. Aus diesem gemeinsamen Geheimnis wird der vertrauliche und authentische Kanal aufgebaut.

4.1 Das PACE-Protokoll

Beim PACE-Protokoll spielen Karte und Leser verschiedene Rollen. Zunächst liest der Leser die Datei EF.CardAccess von der Karte aus. In dieser sind die von der Karte unterstützten Parameter für das Protokoll gespeichert, die sogenannten „SecurityInfos“ [WG310, S.16]. Der Leser sucht dann die Parameter aus EF.CardAccess heraus, die für das Protokoll genutzt werden sollen und überträgt diese in einer APDU an die Karte, diese APDU wird „MSE:Set AT“ (Manage Security Environment: Set Authentication Template for mutual authentication [WG310, S.17]) genannt [WG310, S. 11].

Auf diese Art werden also die Parameter für das PACE-Protokoll festgelegt. Als nächstes werden die Schlüssel vereinbart und mit dem gemeinsamen Geheimnis authentisiert. Dazu werden die folgenden Schritte ausgeführt (siehe [WG310, S. 11-12]):

1. Die Karte generiert eine Zufallszahl, die sie mit einem von dem beiderseits bekannten Passwort abgeleiteten Schlüssel verschlüsselt und schickt den Kryptotext an den Leser.

2. Der Leser leitet ebenso den Schlüssel aus dem beiderseits bekannten Passwort ab und entschlüsselt den Kryptotext, erhält damit also auch die Zufallszahl (die von der Karte in Schritt 1 generiert wurde).

3. Mit Hilfe der Zufallszahl und weiteren Daten, die aus EF.CardAccess stammen oder noch zusätzlich übertragen werden, werden Parameter für einen Diffie-Hellman-Schlüsselaustausch berechnet (mit Hilfe der sogenannten „Mapping"-Funktion).

4. Es wird ein Diffie-Hellman-Schlüsselaustausch durchgeführt. Dieser zeichnet sich dadurch aus, dass nur öffentliche Daten ausgetauscht werden, am Ende aber die beiden Teilnehmer ein gemeinsames Geheimnis besitzen. Er kann aber keine Authentizität der Teilnehmer garantieren. In diesem Fall wurde die Authentizität bereits durch die Kenntnis des Passworts sicher gestellt.

5. Aus dem gemeinsamen Geheimnis leiten beide Seiten einen Schlüssel zur (symmetrischen) Ver- und Entschlüsselung sowie einen Schlüssel zum Sicherstellen der Integrität und Authentizität per Message Authentication Code (MAC) ab.

6. Die Seiten tauschen MAC-Werte über MAC-Schlüssel und öffentliche Schlüssel aus.

MRTD Chip (PICC)		***Inspection System (PCD)***
static domain parameters D_{PICC}		
choose random nonce $s \in_R Dom(E)$		
$z = \mathbf{E}(K_\pi, s)$	$\underline{z}\rangle$	$s = \mathbf{D}(K_\pi, z)$
additional data required for **Map**( )	$\langle$ - $\rangle$	additional data required for **Map**( )
$\widetilde{D} = \mathbf{Map}(D_{PICC}, s)$		$\widetilde{D} = \mathbf{Map}(D_{PICC}, s)$
choose random ephemeral key pair $(\widetilde{SK_{PICC}}, \widetilde{PK_{PICC}}, \widetilde{D})$		choose random ephemeral key pair $(\widetilde{SK_{PCD}}, \widetilde{PK_{PCD}}, \widetilde{D})$
check that $\widetilde{PK_{PCD}} \neq \widetilde{PK_{PICC}}$	$\langle \frac{\widetilde{PK_{PCD}}}{\widetilde{PK_{PICC}}} \rangle$	check that $\widetilde{PK_{PICC}} \neq \widetilde{PK_{PCD}}$
$K = \mathbf{KA}(\widetilde{SK_{PICC}}, \widetilde{PK_{PCD}}, \widetilde{D})$		$K = \mathbf{KA}(\widetilde{SK_{PCD}}, \widetilde{PK_{PICC}}, \widetilde{D})$
$T_{PICC} = \mathbf{MAC}(KS_{MAC}, \widetilde{PK_{PCD}})$	$\langle \frac{T_{PCD}}{T_{PICC}} \rangle$	$T_{PCD} = \mathbf{MAC}(KS_{MAC}, \widetilde{PK_{PICC}})$
verify T_{PCD}		verify T_{PICC}

Abbildung 2: Grafische Darstellung des PACE-Protokolls [WG310, S. 11].

Informationen darüber, wie PACE auf deutschen Ausweisdokumenten eingesetzt wird, finden sich unter [fISB14], insbesondere in den Dokumenten [fISB12a] und [fISB12b].

5 Die Standards PKCS11 und PKCS15

5.1 PKCS11

Der Standard PKCS11 spezifiziert ein Application Programming Interface (API) namens „Cryptoki“ , in dem geregelt ist, wie Software mit Geräten kommuniziert, um kryptographische Operationen durchzuführen. Geräte aller Art, die sicherheitsrelevante Informationen wie geheime Schlüssel, Passwörter etc. beinhalten und kryptographische Operationen durchführen, werden auch „PKCS11-Tokens“ (bzw. im PKCS11-Standard „cryptographic tokens“ oder „tokens“) genannt [Lab04, S. 13].

Das Ziel des PKCS11-Standards ist es, sicherzustellen, dass jede Art von Gerät (mit bestimmten Voraussetzungen) verwendet werden kann (Unabhängigkeit von der Technologie) und dass mehrere Anwendungen auf mehrere Geräte zugreifen können (Teilen von Ressourcen), indem den Anwendungen eine gemeinsame, logische Sicht auf das Gerät präsentiert wird [Lab04, S.2].

5.2 PKCS15

Der PKCS15-Standard beschreibt, wie man einen PKCS11-Token auf einer Smartcard implementiert, d. h. in welcher Form Schlüssel, Zertifikate etc. auf einer Smartcard gespeichert und interpretiert werden sollten. Dies hat zum Ziel, dass verschiedene Anwendungen alle auf die gleiche Weise auf z. B. ein Zertifikat auf einer Smartcard zugreifen können [Lab06, S. 3].

5.2.1 Objektklassen

Der PKCS15-Standard enthält die folgenden vier Objektklassen: Schlüsselobjekte, Zertifikatsobjekte, Datenobjekte und Authentikationsobjekte. In welche Unterklassen diese unterteilt sind, sieht man in der folgenden Abbildung.

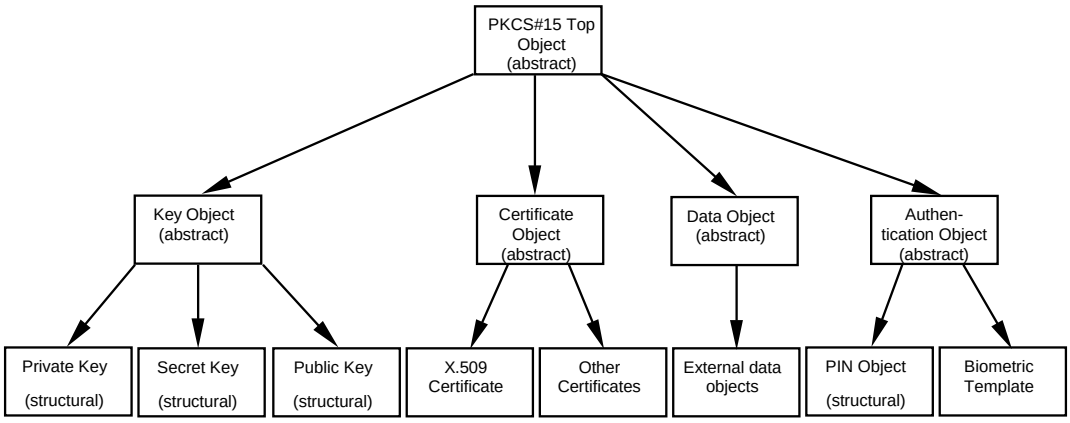


Abbildung 3: Objektklassen und Unterklassen [Lab06, S. 8].

5.2.2 Zugriffsmethoden

Objekte können öffentlich oder privat sein. Der Zugriff auf private Objekte ist durch Authentikationsobjekte geregelt [Lab06, S. 9]. Im vorliegenden Anwendungsfall soll der Zugriff über das Wissen einer PIN geregelt sein.

5.2.3 Dateistruktur auf der Karte

Die hier dargestellten Dateistrukturen sind übliche Beispiele (die auch im gewählten Anwendungsfall zutreffend sind), prinzipiell sind auch andere PKCS15-kompatible Strukturen möglich. Mit EF ist im folgenden Elementary File gemeint, eine Art von Datei, die keine weiteren Dateien enthält, mit DF ist Directory File gemeint, also ein Verzeichnis, das weitere Dateien enthalten kann [WR95, S. 102-103].

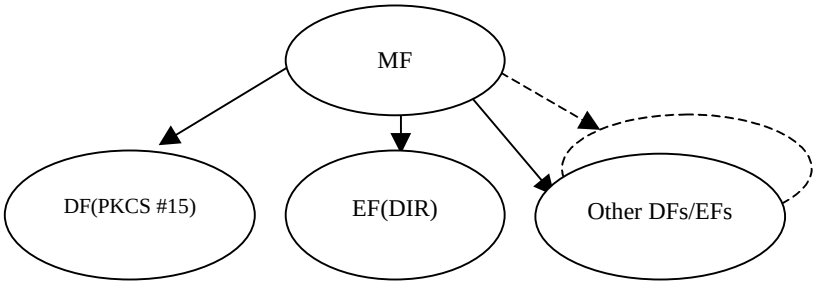


Abbildung 4: Darstellung einer typischen PKCS15-kompatiblen Dateistruktur [Lab06, S. 10]

.

Es wird ein direkt vom Master File (der Datei, in der sich alle anderen Dateien befinden, siehe [WR95, S. 102]) ausgehendes Verzeichnis erstellt, in dem die für den PKCS15-Standard relevanten Daten gespeichert werden, dieses wird hier DF(PKCS #15) genannt. Die Karte kann natürlich weiterhin auch für andere Zwecke benutzt werden, es können auch noch andere Daten auf der Karte liegen (hier mit Other DFs/EFs angedeutet).

Die Datei EF(DIR) ist optional und enthält ausgefüllte Masken für Applikationen. Die Maske muss mindestens den Application Identifier und den Pfad der Applikation enthalten. Die Datei ist nur relevant, wenn die Karte kein direktes Auswählen von Applikationen unterstützt oder mehrere PKCS15-Applikationen auf einer Karte laufen sollen.

In Virtual Keycard enthält das Master File folgende Dateien (die Größe ist in Byte angegeben):

```
OpenSC [3F00]> ls
FileID  Type  Size
 2F00    wEF   128
[5015]    DF     1
```

(Um abschließende Nullen gekürzter) Inhalt eines EF(DIR) in Virtual Keycard:

```
OpenSC [3F00]> cat 2F00
00000000: 61 1C 4F 0C A0 00 00 00 63 50 4B 43 53 2D 31 35 a.O.....cPKCS-15
00000010: 50 06 4D 55 53 43 4C 45 51 04 3F 00 50 15 00 00 P.MUSCLEQ.?.P...
```

Der Tag (eine Auszeichnung, die dazu dient, Daten zu strukturieren und zu klassifizieren) 61 steht für den Anfang eines „Application Template“. Der Tag 4F steht dabei für den „Application Identifier“ der PKCS15-Applikation, dieser besteht aus 12 Bytes und ist im Standard festgelegt [Lab06, S. 17]. Der Tag 50 ist das „Application label“ (in diesem Fall „MUSCLE“), dieses kann benutzt werden, um mehrere PKCS15-Applikationen auseinander halten zu können. Der Tag 51 steht für den Pfad, in diesem Fall 3F005015. [Lab06, S. 17]

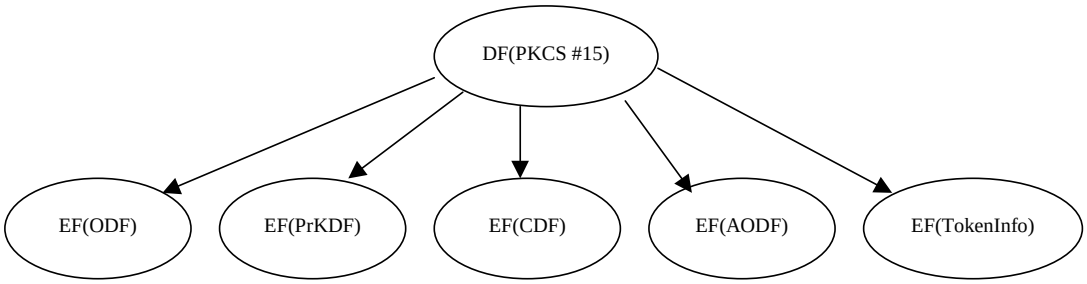


Abbildung 5: Inhalt eines PKCS15-Verzeichnisses [Lab06, S. 10].

An dieser Stelle sei auf den Anhang (siehe Kapitel 9.4) verwiesen, der ein OpenSC-Profil von Virtual Keycard enthält, in dem festgelegt ist, welche Datei welchen Object Identifier, welche Größe etc. erhält.

Das PKCS15-Verzeichnis enthält Dateien, die Informationen zu bestimmten Bestandteilen des PKCS11-Tokens beinhalten:

- EF(ODF): Beim Object Directory File (ODF) handelt es sich um eine Datei, die verpflichtend angelegt werden muss. Sie enthält Zeiger auf die anderen EFs (PrKDFs, PuKDFs, SKDFs, CDFs, DODFs und AODFs) [Lab06, S. 11].

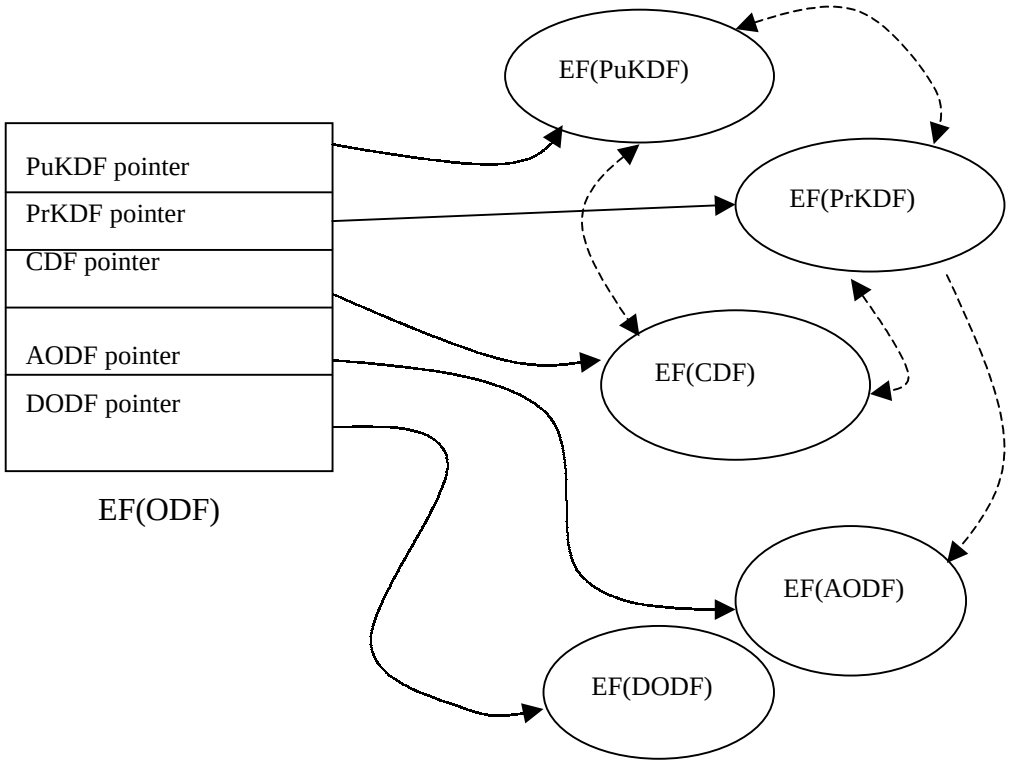


Abbildung 6: Beziehungen der Dateien untereinander [Lab06, S. 12].

(Um abschließende Nullen gekürzter) Inhalt eines ODF in Virtual Keycard:

```
OpenSC [3F00/5015]> cat 5031
00000000: A8 0A 30 08 04 06 3F 00 50 15 44 01 A0 0A 30 08 ..0...?.P.D...0.
00000010: 04 06 3F 00 50 15 44 02 A1 0A 30 08 04 06 3F 00 ..?.P.D...0...?.
00000020: 50 15 44 03 A4 0A 30 08 04 06 3F 00 50 15 44 04 P.D...0...?.P.D.
```

Leicht herauszulesen sind die Zeiger auf das AODF (3F0050154401), PrKDF (3F0050154402), PuKDF (3F0050154403) und das CDF (3F0050154404).

- PrKDFs: Private Key Directory Files (PrKDFs) haben die Funktion von Verzeichnissen, in denen sich die privaten Schlüssel befinden, die der PKCS15-Applikation bekannt sind. Es handelt sich aber weiterhin um EFs. PrKDFs sind optional, befindet sich aber ein privater Schlüssel auf einer Karte, muss mindestens ein PrKDF vorhanden sein. Die PrKDFs enthalten allgemeine Informationen zu Schlüsseln, also z. B. die ID, beabsichtigte Nutzungsarten etc. Sie können auch Querverweise auf Authentikationsobjekte beinhalten, die dazu da sind, den Schlüssel zu schützen. [Lab06, S. 12]

  (Um abschließende Nullen gekürzter) Inhalt eines PrKDF in Virtual Keycard:

```
OpenSC [3F00/5015]> cat 4402
00000000: 30 81 C9 30 14 0C 0B 43 65 72 74 69 66 69 63 61 0..0...Certifica
00000010: 74 65 03 02 06 C0 04 01 FF 30 21 04 14 72 B7 1C te.......0!..r..
00000020: DD 2D 42 BA FA CC 1A 90 87 25 4F 46 21 7F 39 34 .-B......%OF!.94
```

```
00000030: 59 03 02 02 74 03 02 03 B8 02 01 00 A0 7E 30 7C Y...t........~0|
00000040: 30 7A 31 0B 30 09 06 03 55 04 06 13 02 44 45 31 0z1.0...U....DE1
00000050: 0F 30 0D 06 03 55 04 08 0C 06 42 65 72 6C 69 6E .0...U....Berlin
00000060: 31 21 30 1F 06 03 55 04 0A 0C 18 49 6E 74 65 72 1!0...U....Inter
00000070: 6E 65 74 20 57 69 64 67 69 74 73 20 50 74 79 20 net Widgits Pty
00000080: 4C 74 64 31 17 30 15 06 03 55 04 03 0C 0E 48 61 Ltd1.0...U....Ha
00000090: 6E 6E 65 73 20 53 63 68 75 6C 74 7A 31 1E 30 1C nnes Schultz1.0.
000000A0: 06 09 2A 86 48 86 F7 0D 01 09 01 16 0F 68 61 6E ..*.H........han
000000B0: 6E 65 73 2D 37 36 40 6F 6B 2E 64 65 A1 0E 30 0C nes-76@ok.de..0.
000000C0: 30 06 04 04 3F 00 50 15 02 02 08 00 00 00 00 00 0...?.P.........
```

In diesem sind offensichtlich die Zertifikatsinformationen des zum privaten Schlüssel zugehörigen Zertifikats abgelegt. Leicht herauszulesen ist auch der Identifier (72B71CDD2D42BAFACC1A9087254F46217F393459). Der Schlüssel selbst wurde nicht als Datei auf der Karte abgelegt, es ist als Pfad nur 3f005015 angegeben. Im (im Software-System verwendeten) MuscleCard-Applet werden private Schlüssel nur intern im Applet gespeichert und nicht auf der Karte selbst als Datei abgelegt. Es gibt im MuscleCard-Applet auch noch ein extra Access-Control-List-System, mit dem festgelegt wird, welche PIN zum Benutzen welches Schlüssels benötigt wird [DC01, S. 8-10]. Hier nutzt das MuscleCard-Applet also nicht die im PKCS15-Standard festgelegten Strukturen. Das ist aber (insbesondere für den Anwendungsfall) auch nicht notwendig, da ohnehin nie direkt auf den privaten Schlüssel zugegriffen wird (dies sollte ja auch nicht möglich sein), sondern der private Schlüssel für Entschlüsselungs- und Signaturvorgänge mit seiner Nummer (Key Ref) identifiziert wird. Diese findet sich unter dem Tag 0x02 und ist in diesem Fall 0x00.

- PuKDFs: Public Key Directory Files verhalten sich wie PrKDFs, nur für öffentliche Schlüssel. Gehören ein privater und ein öffentlicher Schlüssel zusammen, müssen sie den gleichen Identifier haben [Lab06, S. 13].

  (Um abschließende Nullen gekürzter) Inhalt eines PuKDF in Virtual Keycard:

  ```
  OpenSC [3F00/5015]> cat 4403
  00000000: 30 47 30 11 0C 0B 50 72 69 76 61 74 65 20 4B 65 0G0...Private Ke
  00000010: 79 03 02 06 40 30 20 04 14 72 B7 1C DD 2D 42 BA y...@0 ..r...-B.
  00000020: FA CC 1A 90 87 25 4F 46 21 7F 39 34 59 03 02 00 .....%OF!.94Y...
  00000030: 8B 01 01 00 02 01 00 A1 10 30 0E 30 08 04 06 3F .........0.0...?
  00000040: 00 50 15 30 00 02 02 08 00 00 00 00 00 00 00 00 .P.0............
  ```

  „Private Key“ ist dabei das Standard-Label, das bei der Erzeugung von Schlüsseln von OpenSC vergeben wird, daher steht es in einer Datei, die einen öffentlichen Schlüssel beschreibt. Leicht herauszulesen sind der Zeiger auf den öffentlichen Schlüssel (3F0050153000) und der Identifier (72B71CDD2D42BAFACC1A9087254F46217F393459).

- SKDFs: Haben die gleichen Eigenschaften wie die beiden anderen DFs für Schlüssel, es handelt sich dabei um „secret keys“ [Lab06, S. 13]. Diese optionale Datei wird im vorgestellten Software-System nicht angelegt.

- CDFs: Certificate Directory Files verhalten sich wie PrKDFs, nur für Zertifikate. Enthält ein Zertifikat einen öffentlichen Schlüssel, dessen privater Schlüssel sich auf der Karte befindet, müssen Zertifikat und privater Schlüssel den gleichen Identifier haben [Lab06, S. 14].

  (Um abschließende Nullen gekürzter) Inhalt eines CDF in Virtual Keycard:

```
OpenSC [3F00/5015]> cat 4404
00000000: 30 39 30 11 0C 0B 43 65 72 74 69 66 69 63 61 74 090...Certificat
00000010: 65 03 02 06 40 30 16 04 14 72 B7 1C DD 2D 42 BA e...@0...r...-B.
00000020: FA CC 1A 90 87 25 4F 46 21 7F 39 34 59 A1 0C 30 .....%OF!.94Y..0
00000030: 0A 30 08 04 06 3F 00 50 15 31 00 00 00 00 00 00 .0...?.P.1......
```

  Leicht herauszulesen sind der Zeiger auf das Zertifikat (3F0050153100) und der Identifier (72B71CDD2D42BAFACC1A9087254F46217F393459).

- DODFs: Data Object Directory Files enthalten Informationen über Datenobjekte, die keine Schlüssel oder Zertifikate sind. Sie sind optional, aber wenn es Datenobjekte auf einer Karte gibt, muss es auch ein DODF geben [Lab06, S. 14 - 15].

  Im vorzustellenden Software-System wurde kein DODF benötigt.

- AODFs: Authentication Object Directory Files stellen eine Art Verzeichnis über Authentikations-Objekte (z. B. PINs, Passwörter oder biometrische Daten wie ein Fingerabdruck) dar. Es handelt sich um optionale Dateien, wenn es jedoch ein Authentikations-Objekt auf der Karte gibt, muss es auch ein AODF geben. Neben einigen allgemeinen Informationen enthalten sie auch einen Zeiger auf das Authentikationsobjekt [Lab06, S. 15].

  (Um abschließende Nullen gekürzter) Inhalt eines AODF in Virtual Keycard:

```
OpenSC [3F00/5015]> cat 4401
00000000: 30 37 30 0E 0C 08 55 73 65 72 20 50 49 4E 03 02 070...User PIN..
00000010: 06 C0 30 03 04 01 FF A1 20 30 1E 03 02 03 18 0A ..0..... 0......
00000020: 01 01 02 01 06 02 01 08 02 01 08 80 01 01 04 01 ................
00000030: 00 30 06 04 04 3F 00 50 15 00 00 00 00 00 00 00 .0...?.P........
```

  Die PIN selbst ist nicht auf der Karte abgelegt worden, es ist als Pfad nur 3f005015 angegeben. Im MuscleCard-Applet werden PINs nur intern im Applet gespeichert und nicht auf der Karte als Datei abgelegt.

- EF(TokenInfo): Diese Datei ist verpflichtend und enthält allgemeine Informationen über die Karte. Dazu gehören die Seriennummer der Karte, die unterstützten Dateitypen, die unterstützten Algorithmen usw. [Lab06, S. 16].

  (Um abschließende Nullen gekürzter) Inhalt eines EF(TokenInfo) in Virtual Keycard:

```
OpenSC [3F00/5015]> cat 5032
00000000: 30 39 02 01 00 04 02 00 00 0C 11 49 64 65 6E 74 09.........Ident
00000010: 69 74 79 20 41 6C 6C 69 61 6E 63 65 80 06 4D 55 ity Alliance..MU
00000020: 53 43 4C 45 03 02 04 10 A5 11 18 0F 32 30 31 34 SCLE........2014
00000030: 30 39 32 30 31 31 30 36 32 35 5A 00 00 00 00 00 0920110625Z.....
```

6 Komponenten der praktischen Umsetzung

6.1 Veranschaulichung des Anwendungsfalls

Im Zuge dieser Bachelorarbeit wurde die App Virtual Keycard entwickelt und die Bibliothek OpenSC erweitert. Virtual Keycard wird auf einem Android-Smartphone installiert und OpenSC auf einem Rechner. Mit Hilfe des entwickelten Systems können E-Mails vertraulich und authentisch verschickt werden. Wie ein Nutzer dies tun kann, ist in diesem Unterkapitel dargestellt.

Wurde die Smartcard auf dem Smartphone eingerichtet wie in Kapitel 6.4.1, Abschnitt „Personalisieren der Smartcard mit OpenSC“ beschrieben (d. h. die PIN ist gesetzt und Schlüssel und Zertifikat befinden sich auf der Smartcard), und Icedove wie in Kapitel 6.4.4 beschrieben konfiguriert, kann das System wie folgt genutzt werden:

- Man startet die App auf dem Smartphone.
- Man schließt einen Kartenleser für kontaktlose Smartcards an den Rechner an und legt das Smartphone darauf.

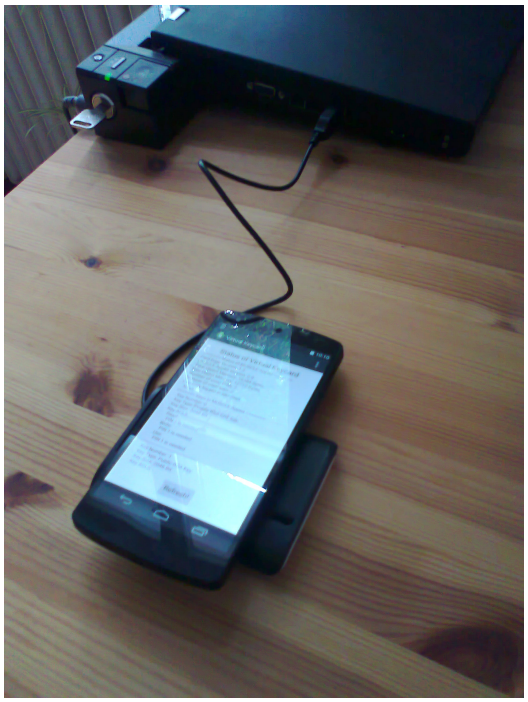


Abbildung 7: Smartphone auf Kartenleser.

Das GUI in der App sieht dann wie folgt aus:

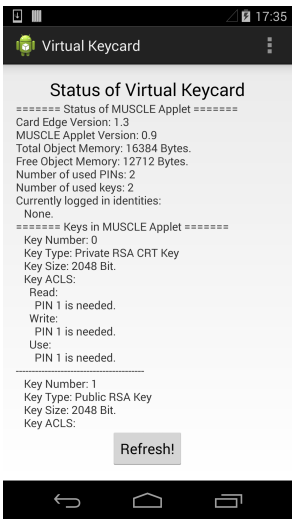


Abbildung 8: GUI der App, das den Status der Smartcard anzeigt.

- Man startet Icedove.
- Zum Entschlüsseln von E-Mails klickt man die E-Mail an, die entschlüsselt werden soll. Es öffnet sich ein Fenster zur PIN-Eingabe:

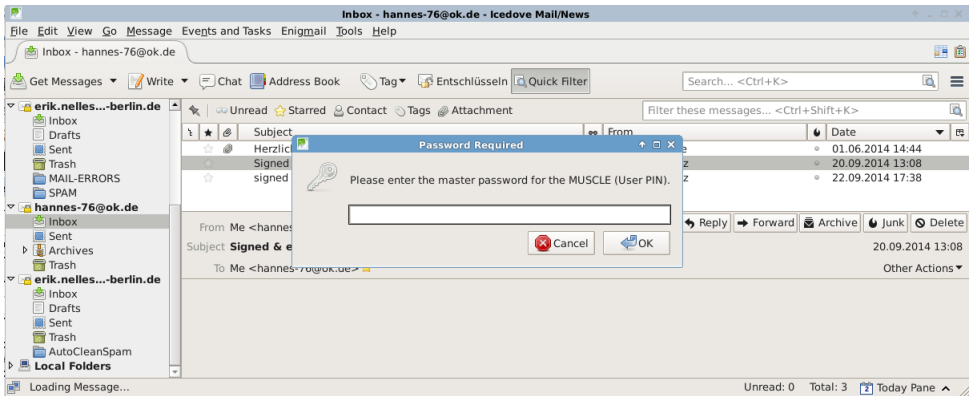


Abbildung 9: Fenster zur PIN-Eingabe beim Entschlüsseln einer E-Mail.

Wird die PIN korrekt eingegeben, kann die Mail entschlüsselt werden. Es wird allerdings noch das Einverständnis des Nutzers auf dem Smartphone abgefragt.

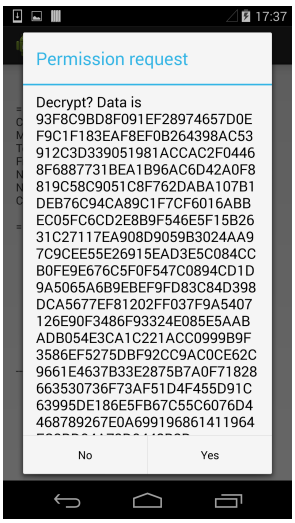


Abbildung 10: Abfrage des Einverständnisses des Nutzers zum Entschlüsselungsvorgang.

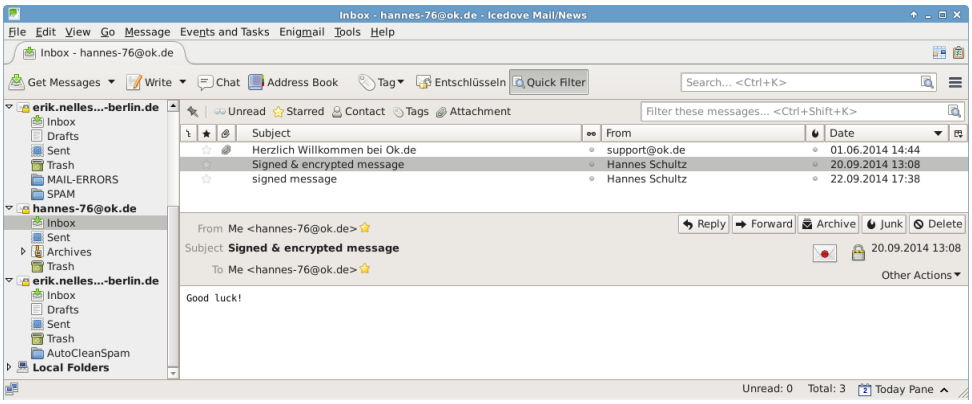


Abbildung 11: Entschlüsselte E-Mail.

Wurde die PIN fehlerhaft eingegeben oder gibt der Nutzer nicht sein Einverständnis auf dem Smartphone, gibt Icedove eine Fehlermeldung aus.

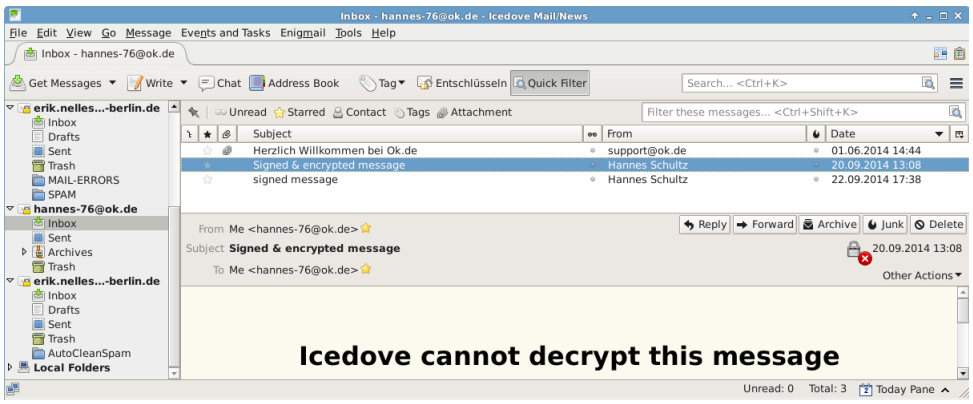


Abbildung 12: Fehler beim Entschlüsseln einer E-Mail.

- Zum Signieren von E-Mails wählt man das Signieren-Feld unter S/MIME an.

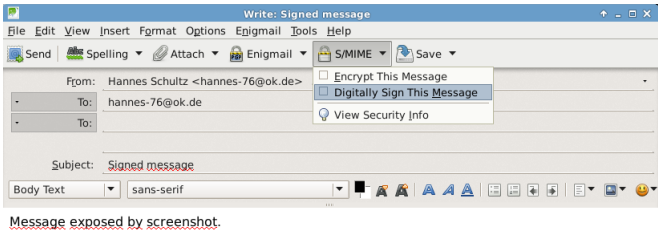


Abbildung 13: Checkbox zum Signieren von E-Mails.

Es folgt wieder eine PIN-Eingabe.

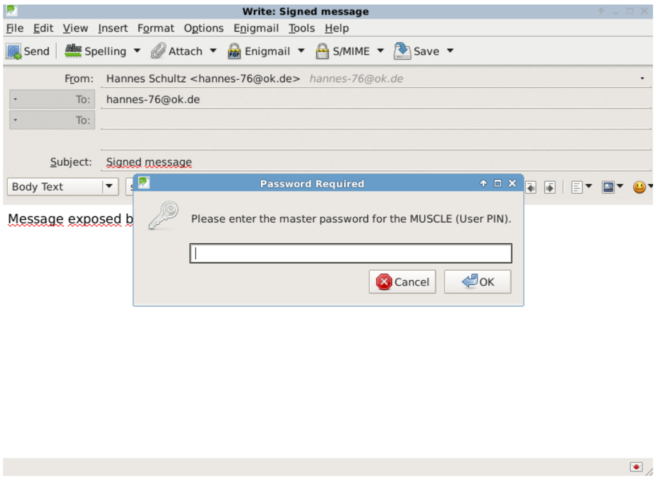


Abbildung 14: PIN-Eingabe beim Signieren von E-Mails.

Der Nutzer muss wieder sein Einverständnis geben.

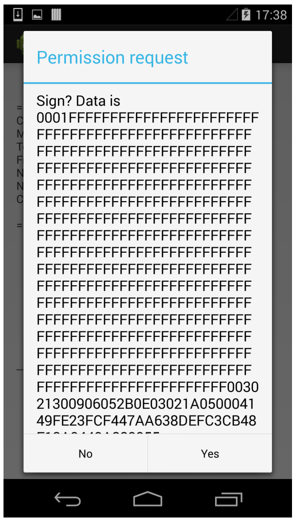


Abbildung 15: Abfrage des Einverständnisses des Nutzers zum Signaturvorgang.

Wird die PIN korrekt eingegeben und gibt der Nutzer sein Einverständnis auf dem Smartphone, wird die E-Mail signiert und verschickt, ansonsten gibt Icedove einen Fehler aus. Der Vorteil der Abfrage ist in erster Linie, dass der Nutzer es an einem vom Rechner unabhängigen Gerät bemerkt, wenn ein Angreifer viele Signaturen erstellen will.

In den folgenden Unterkapiteln wird die praktische Umsetzung im Detail erklärt.

6.2 Eingesetzte Hardware und Software

Die folgende Grafik soll einen Überblick geben, welche Softwarekomponenten auf welchen Hardwarekomponenten zum Einsatz kommen und welche Softwarekomponenten in welcher Weise miteinander kommunizieren:

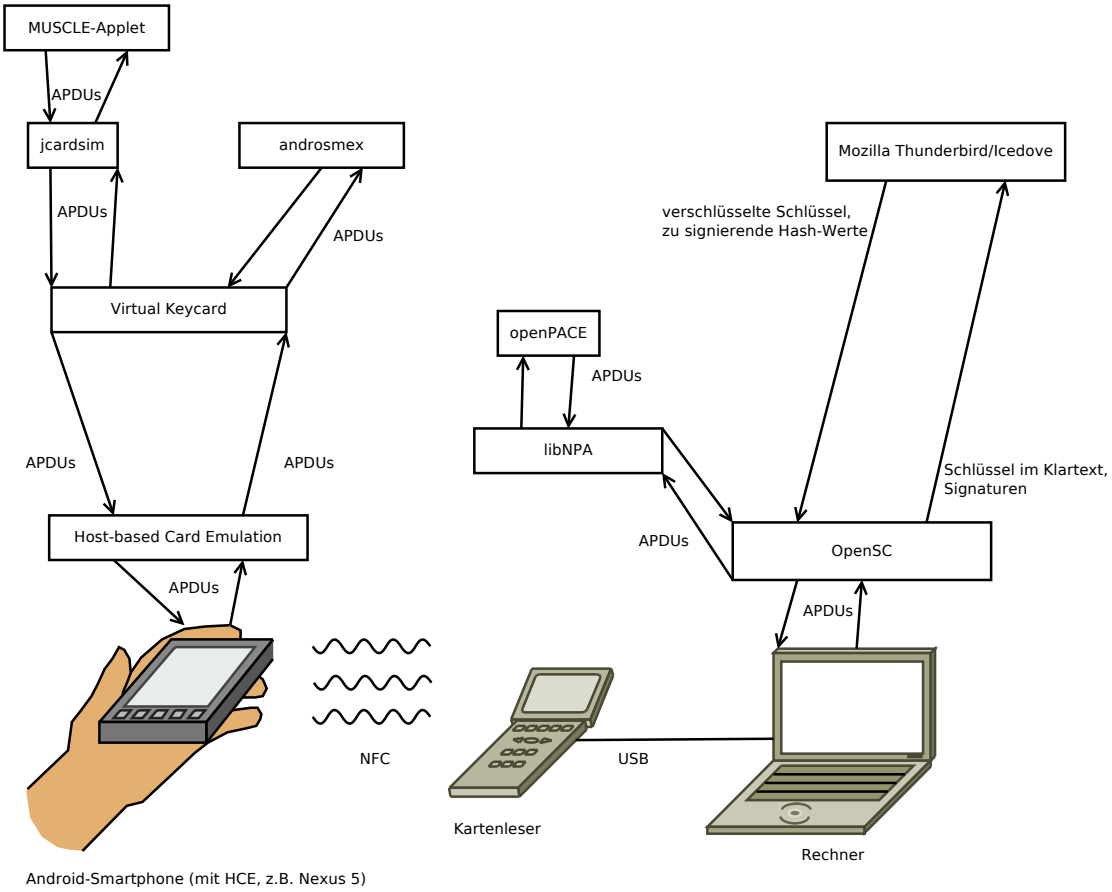


Abbildung 16: Übersicht über die eingesetzte Hard- und Software.

Als Hardware werden eingesetzt:

- Kartenleser: cyberJackRFID basis Contactless Smartcard Reader von REINERSCT
- Smartphone: Nexus 5 (LG-D821), Betriebssystem Android 4.4 (Kitkat)
- Rechner: Thinkpad T61, Betriebssystem Debian Testing, Stand September 2014

6.3 Software auf der Smartphone-Seite

Als Betriebssystem auf dem Smartphone wurde Android Kitkat (4.4.2) genutzt.

6.3.1 Host-based Card Emulation

Auf Android gibt es zwei Konzepte zum Abwickeln der NFC [Pro14b, Abschnitte „Card emulation with a secure element“ und „Host-based card emulation“]:

1. Secure Element: Der NFC-Controller des Smartphones leitet die per NFC einkommenden Daten direkt an das Secure Element des Smartphones weiter und gibt die vom Secure Element ausgehenden Daten direkt an den Leser weiter. Auf diese Art und Weise kann keine Android-Applikation auf die Daten zugreifen.

2. Host-based card emulation (HCE): Der NFC-Controller des Smartphones übergibt die einkommenden Daten an die CPU des Smartphones. Android-Applikationen können die Daten dann verarbeiten und Antwortdaten bereitstellen, die vom NFC-Controller an den Leser übertragen werden.

Um einen HCE-Service auf einem Android-Smartphone zu implementieren, sind die folgenden Schritte notwendig [Pro14b, Abschnitt „Implementing a HCE service“]:

- Sicherstellen der HCE-Unterstützung: Dies geschieht in der Datei „AndroidManifest.xml“ im Projekt Virtual Keycard. Mit Hilfe der folgenden Definitionen von XML-Elementen wird sichergestellt, dass das Smartphone, auf dem die App installiert wird, HCE unterstützt (das wäre z. B. für alle Smartphones mit einem Android < 4.4 nicht der Fall):

  ```
  <uses-feature
          android:name="android.hardware.nfc.hce"
          android:required="true" />

  <uses-permission android:name="android.permission.NFC" />
  ```

- Implementation des Service: Android 4.4 bietet eine Klasse an, die als Basis für HCE-Klassen genutzt werden kann, diese heißt *HostApduService*. Diese Klasse erweitert man mit der Klasse, die HCE nutzen soll. Man muss dann die abstrakten Methoden

  ```
  public byte[] processCommandApdu(byte[] apdu, Bundle extras)
  ```

  und

  ```
  public void onDeactivated(int reason)
  ```

implementieren. Wenn eine APDU an den Service gesendet wird, ruft Android die Methode *processCommandApdu* auf und übergibt ihr die eintreffende APDU. Den Rückgabewert der Methode übergibt Android dann an den Leser.

Welche APDUs an welchen Service weitergeleitet werden müssen, unterscheidet Android mit Hilfe der sogenannten Application IDs (AIDs). Die erste APDU, die der Leser an das Smartphone schickt, enthält in der Regel den AID der Applikation, mit der die Kommunikation aufgebaut werden soll. Entspricht diese dem AID unseres Service, werden alle folgenden APDUs an unseren Service weitergeleitet, bis eine APDU mit einem anderen AID eintrifft, der nicht zu unserem Service gehört, oder die Verbindung zwischen Leser und Smartphone abbricht. In beiden Fällen wird die Methode *onDeactivated* aufgerufen, wobei der Parameter der Methode anzeigt, welcher der beiden Fälle der Grund für das Deaktivieren des Service war.

- Bekanntmachung des Services und Registrierung des AID: Um Android mitzuteilen, dass die App einen HCE-Service unterstützt und den AID für die App zu registrieren, sind weitere Einträge in der Datei „AndroidManifest.xml“ nötig:

```
<service
            android:name="de.nellessen.muscle_card_on_android
               .processCommandApduWrapper"
            android:exported="true"
            android:permission="android.permission.
               BIND_NFC_SERVICE" >
            <intent-filter>
                <action android:name="android.nfc.
                   cardemulation.action.HOST_APDU_SERVICE" />
            </intent-filter>

            <meta-data
                android:name="android.nfc.cardemulation.
                   host_apdu_service"
                android:resource="@xml/apduservice" />
</service>
```

In der Service-Deklaration wird festgelegt, welche Klasse den Service implementiert („name“), ob der Service auch von anderen Apps aus genutzt werden kann („exported“ , hier hätte man auch *false* wählen können, da der Service bisher nur von unserer App genutzt wird) und welche Befugnisse man haben muss, um diesen Service aufrufen zu können („permission“) [Pro14f].

Anschließend wird der intent-filter festgelegt. Dieser zeigt an, auf welche Art von Intents die App antworten kann [Pro14d].

Als letztes wird als *meta-data* angegeben, welche AIDs zu dem Service gehören. Dazu wird auf eine zweite Datei namens „apduservice“ im XML-Ordner verwiesen. In dieser ist der AID wie folgt festgelegt:

```
<aid-group android:description="@string/aiddescription"
                android:category="other">
        <aid-filter android:name="A00000000101"/>
</aid-group>
```

Die Kategorie kann entweder als „payment“ oder „other“ gewählt werden, da keine App zum Bezahlen programmiert wurde, wurde hier „other“ gewählt. [Pro14b, Abschnitt „Service manifest declaration and AID registration“]

Wenn eine APDU mit dem passenden AID eintrifft, erstellt Android ein Objekt der Klasse, die in der Service-Deklaration angegeben wurde. Android erwartet dabei, dass die Klasse einen leeren, öffentlichen Konstruktor (d. h. einen öffentlichen Konstruktor ohne zu übergebende Parameter) besitzt. Anschließend ruft Android die Methode *processCommandApdu* des erstellten Objekts auf und übergibt ihr die eintreffenden APDUs.

6.3.2 jCardSim

jCardSim ist ein Open Source Simulator, der Java Card implementiert. Darunter fallen die Pakete (siehe [Lic14]):

- javacard.framework.*
- javacard.framework.security.*
- javacardx.crypto.*

jCardSim kam in der Version 2.2.2 zum Einsatz und wurde als jar-Datei eingebunden.

Auf einem jCardSim-Objekt können Applets wie auf einer Java Card installiert und ausgewählt werden. Dazu sind nur die folgenden Schritte nötig (siehe [Lic14]):

```
//1. create simulator
JavaxSmartCardInterface simulator = new JavaxSmartCardInterface();
//2. install applet
simulator.installApplet(appletAID, HelloWorldApplet.class);
//3. select applet
simulator.selectApplet(appletAID);
```

jCardSim ist in Java geschrieben und ist unter anderem von den Klassen *CommandAPDU* und *ResponseAPDU* aus dem Paket javax.smartcardio abhängig. Da es dieses Paket nicht auf Android gibt, war es notwendig, die Klassen *CommandAPDU* und *ResponseAPDU* unter dem Paketnamen javax.smartcardio mit in das Projekt Muscle Card on Android zu übernehmen, in dem jCardSim verwendet wurde. Diese Klassen wurden aus der Open Mobile API [Gmb14] übernommen. Ansonsten waren keine weiteren Schritte nötig, um jCardSim für den Funktionsumfang von Virtual Keycard auf Android zu portieren.

6.3.3 MuscleCard-Applet

M.U.S.C.L.E. steht für „Movement for the use of smart cards in a linux environment“ [MP01a]. Im Rahmen dieses Projekts wurde auch das MuscleCard-Applet entwickelt [MP01b]. Das MuscleCard-Applet stellt unter anderem ein PIN-System, einen Objektmanager und eine Schlüsselverwaltung inklusive Nutzung der Schlüssel bereit [DC01]. Daher können im MuscleCard-Applet mit PINs geschützte Schlüssel erzeugt werden, mit denen anschließend Signatur- und Entschlüsselungsvorgänge durchgeführt werden können.

In der App Virtual Keycard wird das MuscleCard-Applet wie im Kapitel jCardSim (6.3.2) beschrieben auf einem *JavaxSmartCardInterface*-Objekt installiert und anschließend verwendet. Bevor das MuscleCard-Applet die genannte Funktionalität bereitstellt, muss es allerdings noch initialisiert werden. Für diesen Schritt gibt es keine offizielle Dokumentation. Eine Skriptsammlung für das MuscleCard-Applet befindet sich auf der unter [Con13] im Literaturverzeichnis angegebenen Website. Im Abschnitt „Applet Initialisation“ findet man ein Skript, das beschreibt, wie genau die setup-APDU zusammengesetzt wird.

Solange die Initialisierung noch nicht ausgeführt wurde, gibt das MuscleCard-Applet für alle APDUs außer einer setup-APDU nur eine Fehlermeldung in Form eines Statusworts aus, dieses lautet 0x9C05 und bedeutet „Unsupported Feature“ (siehe [DC01, S. 19]). In der setup-APDU, die an das MuscleCard-Applet geschickt wird, um es zu initialisieren, muss das Standardpasswort „Muscle00“ eingetragen werden. Folgende Dinge werden in der setup-APDU festgelegt:

- PIN 0 inklusive Entsperrungscode und jeweils Anzahl der zulässigen Fehlversuche
- PIN 1 inklusive Entsperrungscode und jeweils Anzahl der zulässigen Fehlversuche
- Größe des Speichers, der für Schlüssel und Objekte zur Verfügung steht
- Zugriffsregeln zum Erstellen von Objekten, Schlüsseln und PINs

Anschließend kann das MuscleCard-Applet mit der genannten Funktionalität genutzt werden.

Für das MuscleCard-Applet gibt es einen Treiber im Projekt OpenSC, mit Hilfe dessen von Rechner-Seite aus mit dem MuscleCard-Applet kommuniziert werden kann. Dieser konnte mit Anpassungen verwendet werden.

6.3.4 Virtual Keycard

Virtual Keycard ist die App, die auf dem Android-Smartphone installiert wird, um die Funktionalität der Smartcard zur Entschlüsselung und Signierung von E-Mails zu gewährleisten. Das Projekt Virtual Keycard wurde im Zuge dieser Bachelorarbeit in Java implementiert. Es hängt vom Projekt Muscle Card on Android ab. Auch

dieses Projekt wurde im Zuge dieser Bachelorarbeit implementiert und hängt von jCardSim, dem (von einigen Bugs bereinigten) MuscleCard-Applet und einem von Ole Richter (siehe [Ric14]) modifizierten androsmex ab.

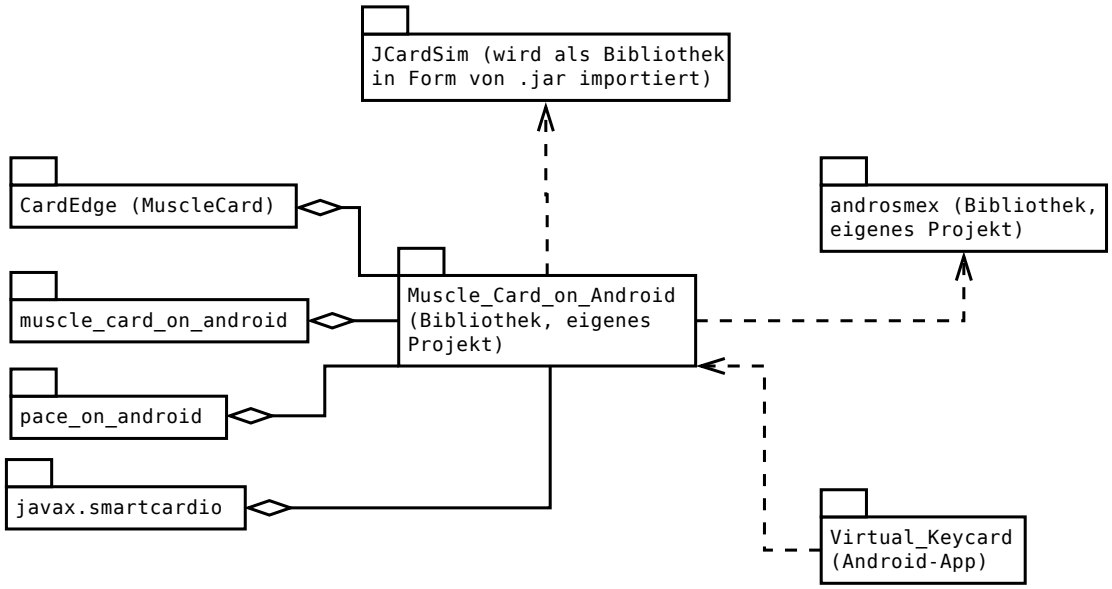


Abbildung 17: Veranschaulichung der Beziehungen der Projekte und Pakete zueinander.

GUI und Activities

Virtual Keycard selbst stellt das Graphical User Interface (GUI) zur Verfügung, über das der Nutzer am Anfang die PIN wählen kann und später Informationen zum Zustand der Smartcard erhält. In der GUI werden auch die Nutzerentscheidungen zu Entschlüsselungs- und Signaturvorgängen abgefragt und der letzte Login-Versuch sowie das Entsperren von PINs vorgenommen. Um dies zu realisieren, wurden drei Activities implementiert, *MainActivity*, *CreatePinPersistent* und *LastLoginTryWithGUIActivity*.

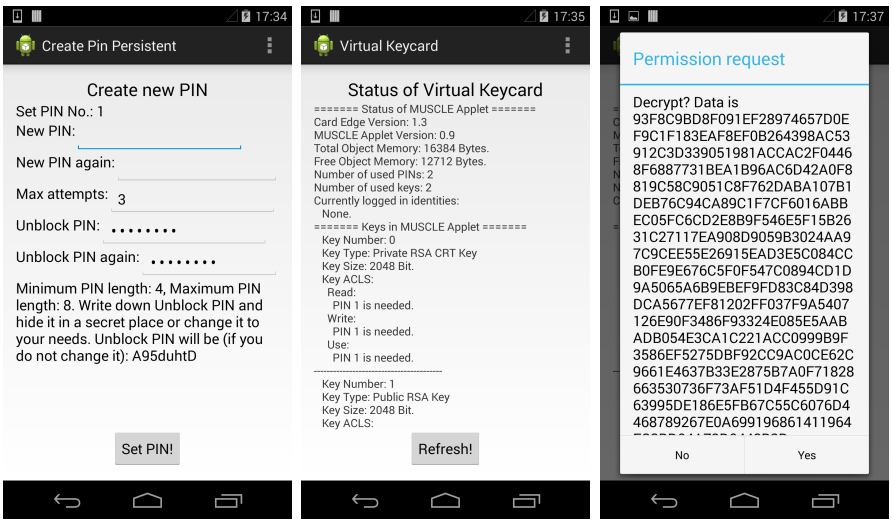


Abbildung 18: Teile des Graphical User Interface der App Virtual Keycard.

Ansonsten initialisiert Virtual Keycard eine Instanz der Klasse *MSCHostApduService*, die die gewünschte Smartcard-Funktionalität zur Verfügung stellt.

MSCHostApduService

MSCHostApduService nimmt die vom Leser kommenden APDUs entgegen, bereitet sie so auf, dass sie der Form entsprechen, die das MuscleCard-Applet erwartet und gibt APDUs an den Leser zurück. Außerdem werden von *MSCHostApduService* PACE-Kanäle aufgebaut (immer nur einer zur Zeit), APDUs ent- und verschlüsselt und Objekte dauerhaft auf dem Speicher des Smartphones gespeichert und von diesem wieder geladen.

Im Konstruktor der Klasse *MSCHostApduService* wird zunächst das MuscleCard-Applet wie im Unterkapitel jCardSim beschrieben (siehe 6.3.2) installiert. Der Konstruktor von *MSCHostApduService* ist damit beendet. Anschließend wird das MuscleCard-Applet von Virtual Keycard aus initialisiert, wie im Unterkapitel MuscleCard-Applet (siehe 6.3.3) beschrieben wurde. In Virtual Keycard wurde die setup-APDU so gewählt, dass nur PIN 1 authentisiert worden sein muss, um Objekte, Schlüssel und PINs zu erstellen. Das ist auch die PIN, die von OpenSC verwendet wird, um diese Aktionen durchzuführen. Wir verwenden also nur eine PIN, obwohl das MuscleCard-Applet auch die Möglichkeit bietet, mehrere PINs zu verwenden.

PIN 0 wird zufällig erstellt, da sie für den Setup-Schritt zwar nötig ist, aber danach nicht mehr benötigt wird. PIN 1 (inklusive der Anzahl der möglichen Fehlversuche und der Unblock-PIN) wird beim Start der App vom Nutzer abgefragt, falls sich nicht bereits eine Datei namens „pin1“ im Ordner „pins“ im persistenten

Speicher befindet. Sonst wird ebendiese Datei eingelesen und die Daten werden ins MuscleCard-Applet übernommen. Um zu verstehen, warum und wie dies durchgeführt wird, siehe auch den Abschnitt „Persistentes Speichern von PINs“ in diesem Unterkapitel.

Um die 4 bis 8-stellige PIN vom Nutzer abzufragen, wird von *MainActivity* aus die Activity *CreatePinPersistent* aufgerufen. Diese bietet dem Nutzer ein grafisches Interface, in das er seine gewünschten PIN-Daten eingibt. Beim Drücken eines Buttons wird die PIN dann persistent abgespeichert.

Um die Funktionen der Klasse *MSCHostApduService* in beiden Activities nutzen zu können, kann man das Objekt der Klasse zwischen den Activities übertragen. Android bietet dazu *Intents* an, denen man *Extras* hinzufügen kann. Die Funktionen mit dem Namen *putExtra*, die man dazu benutzen kann, erwarten aber, dass es sich um einen in Java bekannten Datentyp oder einen Datentyp, der Parcelable oder Serializable ist, handelt (siehe [Pro14c]). Dies ist für Objekte der Klasse *MSCHostApduService* nicht der Fall, da diese ein Objekt der Klasse *JavaxSmartCardInterface* beinhalten, das weder Parcelable noch Serializable ist.

Diese Problem könnte man lösen, indem man aus *MainActivity* heraus die Activity *CreatePinPersistent* aufruft und sich die eingegebenen PIN-Daten z. B. als Byte-Arrays übergeben lässt. Dafür bietet Android die Methoden *startActivityForResult* und *onActivityResult* an [Pro14a]. Diese Lösung hat den Nachteil, dass die gerufenen Activities nicht direkt mit dem *MSCHostApduService*-Objekt kommunizieren können.

Daher wurde *MSCHostApduService* als Singleton-Klasse implementiert, d. h. es kann nur eine Instanz der Klasse geben. Dafür wird der Konstruktor der Klasse versteckt (als „private“ markiert), eine private, globale Variable angelegt, die auf das Objekt zeigt und der Zugriff über eine öffentliche, statische Methode gesichert. Diese gibt in dem Fall, dass es eine Instanz der Klasse gibt, die Instanz zurück. Falls es keine Instanz gibt, erstellt sie eine solche und speichert sie in der privaten, globalen Variable. Activities können nun einfach die öffentliche, statische Methode aufrufen, um eine Referenz auf das (einzige) Objekt der Klasse *MSCHostApduService* zu erhalten - so haben alle Activities Zugriff darauf.

Da für die Host-based Card Emulation aber ein öffentlicher, leerer Konstruktor in der Klasse erwartet wird, kann *MSCHostApduService* nun nicht mehr für HCE genutzt werden. Daher wurde eine Klasse namens *processCommandApduWrapper* implementiert, die die nötigen Methoden von *HostApduService* implementiert und einen leeren Konstruktor hat. In der Klasse wird eine Referenz des *MSCHostApduService*-Objekts gespeichert und die APDUs werden an dieses Objekt weitergeleitet. Der Antwort-Wert aus dem *MSCHostApduService*-Objekt wird dann einfach (an den Leser) zurückgegeben.

Ist das MuscleCard-Applet initialisiert, wird PIN 1 authentisiert und das Wurzelobjekt mit der ID 3F000000 erzeugt, da von der Rechner-Seite aus die Existenz dieses Wurzelobjekts erwartet wird. Anschließend werden die Objekte, PINs

und Schlüssel vom persistenten Speicher des Smartphones eingelesen und in das MuscleCard-Applet übertragen.

Von diesem Punkt an werden dem *MSCHostApduService*-Objekt APDUs an die Methode

```
public byte[] processCommandApdu(byte[] apdu, Bundle extras)
```

übergeben. Die Methode ruft dann abhängig von der einkommenden APDU weitere Methoden auf und gibt am Ende eine APDU zurück. Die APDUs kommen vom Leser und die Antworten gehen an den Leser zurück. Es wird auch noch von Virtual Keycard aus auf die Methode

```
public String getKeyCardStatus()
```

zugegriffen, um dem Nutzer Informationen zum Zustand des MuscleCard-Applets anzuzeigen.

Persistentes Speichern von Objekten, PINs und Schlüsseln

Wie bereits beschrieben, läuft das MuscleCard-Applet auf einem Objekt aus jCardSim. Werden Daten als Objekte aus Sicht des MuscleCard-Applets persistent gespeichert, so schreibt es sie einfach auf die Karte. In diesem Fall ist die Karte allerdings nur die Instanz einer Java-Klasse, existiert also lediglich im Hauptspeicher des Smartphones und wird beim Beenden der App wieder gelöscht. Um Objekte persistent speichern zu können, bietet Java die Möglichkeit, Klassen als *Serializable* zu implementieren. Man kann dann Objekte dieser Klassen leicht in ein Format bringen lassen, in dem man sie auf einem persistenten Speicher ablegen und bei Bedarf (im Fall von Virtual Keycard beim Start der App) wieder einlesen kann.

Dafür müssen aber alle enthaltenen Objekte anderer Klassen *Serializable* sein (oder man benutzt „custom serialization“). Da jCardSim nicht als *Serializable* implementiert wurde und auch für PACE auf die PINs aus dem MuscleCard-Applet zugegriffen werden muss, wurde in Virtual Keycard eine andere Möglichkeit des persistenten Speicherns gewählt.

Soll eine Veränderung an einem Objekt, einer PIN oder einem Schlüssel vorgenommen werden, erkennt man dies am Instruktionsbyte der eintreffenden APDU. Die APDU wird eventuell vorbereitet und dann an das MuscleCard-Applet übergeben. Am zurückkommenden Statuswort kann man erkennen, ob die Aktion erfolgreich durchgeführt wurde (es ist dann 0x9000). In diesem Fall wird die Änderung persistent gespeichert. Sollte die App also beendet werden, bevor das persistente Speichern beendet wurde, kann es sein, dass die Änderung bereits im MuscleCard-Applet enthalten war, aber nicht auf dem persistenten Speicher abgelegt wurde; der Zustand ist also nicht konsistent und die Änderung geht verloren, obwohl sie im MuscleCard-Applet erfolgreich durchgeführt wurde. Beim nächsten Start der App befindet man sich aber wieder in einem konsistenten Zustand. Außerdem wird die APDU mit dem Statuswort 0x9000 erst zurückgegeben, wenn auch das persistente Speichern abgeschlossen wurde.

Das Ziel beim persistenten Speichern der Objekte, PINs und Schlüssel ist es, die relevanten Teile des aktuellen Zustands des MuscleCard-Applets festzuhalten, sodass sie nach dem Beenden der App wieder hergestellt werden können.

Persistentes Speichern von Objekten

Für Objekte wird dazu im „files“-Ordner der App (dies ist ein Ordner in dem der App zugewiesenen Ordner) ein Ordner „objects“ erstellt. Um den „files“-Ordner der App zu bestimmen, wird im Konstruktor von *MSCHostApduService* eine Instanz der Klasse

```
Context
```

übergeben und in der globalen Variable

```
context
```

gespeichert. In *MainActivity* aus Virtual Keycard wird dazu eine Referenz auf das aufrufende Objekt übergeben:

```
MyHostApduService = MSCHostApduService.getInstance(this);
```

Aus diesem Context kann in *MSCHostApduService* der Verzeichnispfad ausgelesen werden:

```
this.context.getFilesDir()
```

Wurde ein Objekt erfolgreich erstellt (erkennbar am Instruktionsbyte 0x5A, siehe [DC01, S. 46] und dem Statuswort 0x9000), wird die Methode

```
private void createObjectPersistent(byte[] createObjectApdu)
```

aufgerufen. Sie erstellt im Ordner „objects“ eine Datei, die den Namen des Object Identifiers verbunden mit der Endung _c hat. In ihr wird einfach die APDU zur Erstellung des Objekts gespeichert. Den Object Identifier kann man aus den Bytes 6 bis 9 der APDU auslesen [DC01, S. 46].

Wurde in ein Objekt erfolgreich geschrieben (erkennbar am Instruktionsbyte 0x54, siehe [DC01, S. 50] und dem Statuswort 0x9000), wird die Methode

```
private void writeObjectPersistent(byte[] writeObjectApdu)
```

aufgerufen. Sie erstellt im Ordner „objects“ eine Datei, die wie der Object Identifier heißt, falls sie noch nicht existiert. Den Object Identifier kann man aus den Bytes 6 bis 9 der APDU auslesen, den Offset aus den Bytes 10 bis 13 [DC01, S. 50]. Die Datei wird ausgelesen. War die Datei leer, erstellt *writeObjectPersistent* ein Byte-Array der Größe Offset addiert mit der Anzahl der Datenbytes, die geschrieben werden sollen, schreibt in dieses ab dem Offset die Daten hinein und schreibt dieses Byte-Array dann in die Datei. War die Datei nicht leer, erstellt *writeObjectPersistent* ein Byte-Array, das mindestens so groß ist wie die ausgelesenen Daten und mindestens so groß wie der Offset addiert mit der Anzahl der Datenbytes, die geschrieben werden sollen. In dieses schreibt *writeObjectPersistent* dann vom Anfang an die ausgelesenen Daten hinein und anschließend die Datenbytes, die geschrieben

werden sollen, ab dem Offset. Das Byte-Array schreibt *writeObjectPersistent* dann in die Datei.

Persistentes Speichern von PINs

Für PINs wird im „files“-Ordner der App ein Ordner „pins“ erstellt.

Wurde eine PIN erfolgreich erstellt (erkennbar am Instruktionsbyte 0x40 der einkommenden APDU, siehe [DC01, S. 37] und dem Statuswort 0x9000 der ausgehenden APDU), wird die Methode

```
private void writePinPersistentOnCreate(byte[] createPinApdu)
```

aufgerufen. Sie schreibt zwei Nullen und daran anschließend die einkommende APDU so wie sie ist unter dem Namen „pin$“ in den persistenten Speicher, wobei „$“ der Nummer der erstellten PIN entspricht, auslesbar im 3. Byte der APDU. Die zwei Nullen stehen für die Anzahl der Fehlversuche, die bereits gemacht wurden, die erste für die Fehlversuche an der PIN, die zweite für die Fehlversuche an der Unblock-PIN.

Wie bereits erwähnt wurde, wird das MuscleCard-Applet beim Start der App mit Hilfe der Funktion *setup_mscapplet* aus der Klasse *MSCHostApduService* initialisiert und dabei werden auch die Werte für PIN 0 und PIN 1 inklusive der jeweiligen Unblock-PINs gesetzt. Eigentlich ist dies eine Aufgabe, die der Benutzer vom Rechner aus übernehmen sollte. Daher wird in OpenSC so vorgegangen, dass beim Erstellen der PKCS15-Struktur eine PIN vom Benutzer festgelegt wird. Der PKCS15-„muscle“-Treiber enthält aber nicht die Initialisierungsfunktion, sodass die PIN niemals auf der Smartcard eintrifft.

Es wäre möglich, aus der vom Nutzer eingegebenen PIN und Unblock-PIN eine setup-APDU aus OpenSC an die Karte zu senden. Dies kann in der Funktion

```
static int muscle_create_pin(sc_profile_t *profile,
    sc_pkcs15_card_t *p15card, sc_file_t *df, sc_pkcs15_object_t *
    pin_obj, const unsigned char *pin, size_t pin_len, const
    unsigned char *puk, size_t puk_len)
```

geschehen, in der die vom Nutzer eingegebene PIN samt Unblock-PIN vorliegt. Man könnte also aus den vorliegenden Informationen eine setup-APDU erstellen und diese an das Smartphone senden. Der Nachteil bei diesem Vorgehen ist aber, dass diese APDU unverschlüsselt übertragen wird und die PIN für einen Angreifer, der die NFC belauscht, ersichtlich ist. Man kann zu diesem Zeitpunkt auch keinen PACE-Kanal einrichten, da die PIN sich noch nicht auf der Karte befindet.

In Virtual Keycard wurde aus diesem Grund eine andere Strategie gewählt. PIN 0 (und die dazugehörige Unblock-PIN) wird wie oben bereits erwähnt zufällig gewählt und die Daten zu PIN 1 werden vom Nutzer auf dem Smartphone abgefragt. Im OpenSC-PKCS15-Treiber von Virtual Keycard wurde dann noch ein Hinweis eingefügt, der dem Nutzer mitteilt, dass die vermeintlich neue PIN, die er eingibt,

nicht auf dem Smartphone ankommen wird und er sie deshalb beliebig wählen kann.

Bei Login-Vorgängen müssen die Fehlversuche mitgeschrieben werden, damit sie auch nach dem Neustart der App noch vorhanden sind. Ansonsten könnte man die App neu starten und so wieder die volle Anzahl an möglichen Fehlversuchen erhalten. Um die Anzahl der Fehlversuche zu speichern, wird die Methode

```
private void changeUnsuccessfulTriesPersistent(byte number,
    boolean unblockPin, boolean increase)
```

aufgerufen, wenn eine APDU mit Instruktionsbyte 0x42 (Verify PIN, [DC01, S. 39]) oder 0x44 (Change PIN, [DC01, S. 41]) eintrifft. Ist das Statuswort der Response-Apdu 0x9000, wird die Methode mit *increase=false* aufgerufen und setzt den Fehlbedienungszähler wieder auf 0. Ist das Statuswort der Response-Apdu 0x9C02, wird die Methode mit *increase=true* aufgerufen und erhöht den Fehlbedienungszähler in der Datei um 1.

Wird ein PACE-Kanal aufgebaut, so lässt sich auch dabei auf der Rechnerseite überprüfen, ob die eingegebene PIN korrekt war. Wird der PACE-Kanal erfolgreich aufgebaut, war die eingegebene PIN vermutlich korrekt, wird er nicht erfolgreich aufgebaut, war die PIN vermutlich nicht korrekt (insofern keine anderen Fehler aufgetreten sind). Um einen Brute-Force-Angriff auf die PIN per Aufbauen von PACE-Kanälen zu unterbinden, muss der Fehlbedienungszähler also auch erhöht werden, wenn damit begonnen wird, einen PACE-Kanal aufzubauen.

Dies erkennt man an der „get Nonce“-APDU. Beim Eintreffen einer solchen APDU wird der Fehlbedienungszähler im persistenten Speicher erhöht. In Schritt 4 des PACE-Protokolls werden die MAC-Werte über die MAC-Schlüssel und die öffentlichen Schlüssel ausgetauscht und daran überprüft, ob der Aufbau des PACE-Kanals erfolgreich war. War er erfolgreich, wird ein *SecureMessaging*-Objekt angelegt. In *MSCHostApduService* zählt man also mit, in welchem Schritt von PACE man sich befindet (d. h. wie viele General Authenticate-APDUs (siehe [WG310, S. 17]) man bereits bekommen hat). Befindet man sich in Schritt 4 und es wurde ein *SecureMessaging*-Objekt erstellt, setzt man den Fehlbedienungszähler im persistenten Speicher wieder auf 0. Befindet man sich in Schritt 4 und es wurde kein *SecureMessaging*-Objekt erstellt, erhöht man auch im MuscleCard-Applet den Fehlbedienungszähler, indem man sich mit einer falschen PIN zu authentisieren versucht (für die falsche PIN siehe auch den Abschnitt über das Einlesen der PINs in diesem Unterkapitel).

Ein klassischer Angriff auf Smartcards ist es, Login-Versuche auf der Smartcard zu unternehmen und über einen Seitenkanal herauszubekommen, ob der Versuch erfolgreich war, bevor der Fehlbedienungszähler erhöht werden kann [Eck08, S. 465]. Wenn der Login-Versuch nicht erfolgreich war, entzieht man der Karte den Strom und sie kann den Fehlbedienungszähler nicht mehr erhöhen. Der Angriff kann verhindert werden, indem man erst den Fehlbedienungszähler erhöht, dann den

Login-Vorgang durchführt und im Fall des Erfolgs den Fehlbedienungszähler wieder auf 0 setzt. Dieses Vorgehen wurde auch in der Methode *processCommandApdu* in der Klasse *MSCHostApduService* umgesetzt. Wird die App beendet, bevor der Fehlbedienungszähler wieder dekrementiert wird, wird der Fehlversuch (eventuell zu Unrecht) dauerhaft gespeichert, dies kommt in der Praxis aber bei normalem Betrieb in der Regel nicht vor.

Ist nur noch ein Login-Versuch übrig, wird die Karte für Zugriffe von außen gesperrt, es kommt dann unabhängig von der eintreffenden APDU immer eine APDU mit dem Statuswort 0x6900 („not allowed“) zurück. Außerdem wird in der App die Activity *LastLoginTryWithGUIActivity* geöffnet, in dem der Nutzer den letzten Login-Versuch unternehmen kann. Schlägt auch dieser fehl, hat er in dem Fenster die Möglichkeit, die Unblock-PIN einzugeben (so oft, wie gültige Fehlversuche beim Erstellen der PIN angegeben wurden). Die Activity kommuniziert dabei mit dem *MSCHostApduService*-Objekt, indem sie die Singleton-Eigenschaft nutzt und sich eine Referenz auf das Objekt verschafft. Ein Entsperren der PIN von außen ist über den im MuscleCard-Applet implementierten Weg nicht möglich (die Command-APDU muss dafür das Instruktionsbyte 0x46 enthalten, siehe [DC01, S.43]), es wird im Fall einer eintreffenden APDU mit Instruktionsbyte 0x46 immer das Statuswort 0x6900 zurückgegeben.

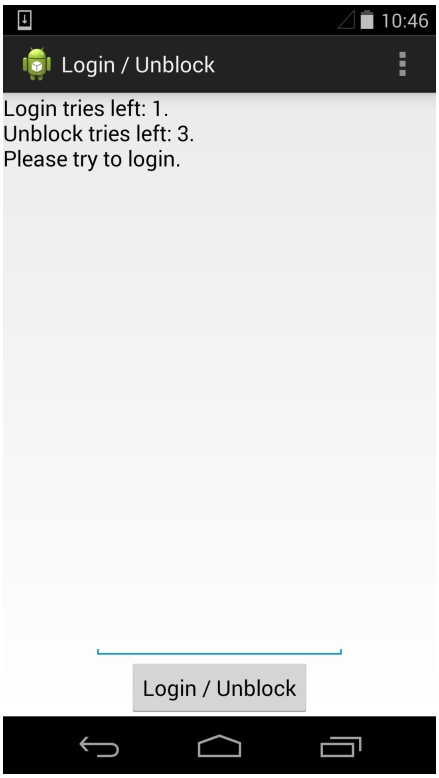


Abbildung 19: GUI für den letzten Login-Versuch.

Durch dieses Vorgehen verhindert man, dass die Karte gesperrt werden kann,

indem sie Kontakt zu einem Leser aufnimmt, der Login-Versuche auf ihr unternimmt. Der letzte Login-Versuch wird direkt auf dem Smartphone vorgenommen, da man die Unblock-PIN nicht immer dabei hat.

Schafft der Nutzer es nicht, innerhalb der Fehlversuche die PIN zu entsperren, ist die Karte gesperrt. Die App muss dann neu installiert werden und das Schlüsselmaterial ist verloren. An dieser Stelle ist es wichtig, dass die Schlüssel sicher gelöscht werden, sodass sie nicht wieder hergestellt werden können, siehe dazu auch das Kapitel 7.3.

Wurde eine PIN erfolgreich geändert (erkennbar am Instruktionsbyte 0x44 [DC01, S. 41] und am Statuswort 0x9000), muss auch das mitgeschrieben werden. Dafür wurde die Methode

```
private void writePinPersistentOnChange(byte[] changePinApdu)
```

implementiert. Ihr wird die eintreffende APDU übergeben, sie liest dann den bisherigen Wert der APDU vom persistenten Speicher und überschreibt ihn mit dem neuen Wert. Der Wert der Unblock-PIN kann nicht geändert werden, er wird aus der alten APDU vom persistenten Speicher übernommen.

Persistentes Speichern von Schlüsseln

Für Schlüssel wird im „files“-Ordner der App ein Ordner „keys“ erstellt.

Wurde ein Schlüsselpaar erfolgreich generiert (erkennbar am Instruktionsbyte 0x30, siehe [DC01, S. 21] und am Statuswort 0x9000), sollen die Schlüssel persistent gespeichert werden. Dazu wurde die Methode

```
private void writeKeysPersistent()
```

implementiert.

Sie benutzt die Funktion „MSCListKeys“ des MuscleCard-Applets (siehe [DC01, S. 35]), um sich nacheinander Informationen zu den einzelnen Schlüsseln zu holen. Sie hält auch ein *boolean*-Array, in dem steht, welche Schlüssel existieren. Für existente Schlüssel wird der Wert auf *true* gesetzt. Aus den Informationen aus der Antwort holt sie bereits die Access Control Lists (ACLs) heraus und schreibt diese in einen *ByteArrayOutputStream*.

Dann liest sie den Schlüssel aus. Dazu benutzt sie die Funktion „MSCExportKey“ des MuscleCard-Applets (siehe [DC01, S. 26]). Diese Funktion legt den Schlüssel im Export-Objekt (mit der ID 0xFFFFFFFF) ab. Jetzt ruft *writeKeysPersistent* die *MSCHostApduService*-Methode

```
private byte[] readOutputObject()
```

auf. Diese liest das Export-Objekt je nach Größe in mehreren APDUs aus dem MuscleCard-Applet aus und gibt es zurück. Anschließend löscht *writeKeysPersistent* das Export-Objekt und schreibt den aus dem MuscleCard-Applet exportierten Schlüssel in den *ByteArrayOutputStream*.

Dieser wird dann in ein Byte-Array umgewandelt und auf den persistenten Speicher unter dem Namen „key$“ , wobei $ für die Nummer des Schlüssels steht (die aus den Informationen aus „MSCListKeys“ gewonnen wurde) geschrieben. Schlüssel, die in dem MuscleCard-Applet nicht existieren, wohl aber auf dem persistenten Speicher, werden anschließend gelöscht. Welche das sind, erschließt sich aus dem erwähnten *boolean*-Array.

Für die persistente Speicherung von Schlüsseln müssen also auch private Schlüssel aus dem MuscleCard-Applet extrahierbar sein. Dies ist aber ein klarer Nachteil gegenüber einer klassischen Smartcard - ist der Schlüssel extrahierbar, kann er leicht ausgelesen und z. B. auf einem Rechner gespeichert werden. Daher wird in Virtual Keycard ein weiterer Schutzmechanismus verwendet: In der Methode *processCommandApdu* wird bei einkommenden APDUs, die auf das Extrahieren eines Schlüssels abzielen (erkennbar am Instruktionsbyte 0x34, siehe [DC01, S. 26]) überprüft, ob es sich um einen privaten Schlüssel handelt. Das ist mit Hilfe der Funktion „MSCListKeys“ aus dem MuscleCard-Applet möglich (siehe [DC01, S. 35]). Handelt es sich um einen privaten Schlüssel, wird ohne weiteres Mitwirken des MuscleCard-Applets das Statuswort 0x9C06 („Unauthorized“, siehe [DC01, S. 19]) zurückgegeben.

Lesen der persistent gespeicherten Objekte, PINs und Schlüssel

Das Ziel beim Einlesen der persistent gespeicherten Objekte, PINs und Schlüssel ist es, das MuscleCard-Applet in den Zustand zu versetzen, der durch das Abspeichern in die Dateien festgehalten wurde. Funktioniert das Abspeichern und Einlesen wie erhofft, speichert die virtuelle Smartcard also alle relevanten Änderungen, die an ihr durchgeführt werden, dauerhaft.

Lesen der persistent gespeicherten Objekte

Zum Einlesen der persistent gespeicherten Objekte wurde die Methode

```
private void readObjectsPersistent()
```

implementiert. Sie erstellt zunächst die Objekte im MuscleCard-Applet. Dazu liest sie die Dateien aus dem Ordner „objects“ ein, deren Name auf „_c“ endet und schickt die enthaltenen APDUs so wie sie sind an das MuscleCard-Applet.

Anschließend schreibt sie die Daten in die erstellten Dateien. Dazu liest sie die Dateien aus dem Ordner „objects“ ein, die nicht auf „_c“ enden. Hat sie die Daten aus einer Datei ausgelesen, schreibt sie die Daten in das Objekt auf dem MuscleCard-Applet, das als Object ID den Namen der Datei hat.

Das muss in Portionen von maximal 246 Byte geschehen, da in einer „normalen“ , d. h. non-extended APDU der Datenteil maximal 255 Byte groß sein kann und 9 Byte auf die Informationen Object ID, Offset und Größe des Datenteils entfallen. Die Funktion liest die Datei im Ganzen ein und setzt eine Variable Offset auf 0. Sie bestimmt dann die Länge der nächsten APDU, die sie versenden wird. Diese

ist 259, wenn die Länge der Daten größer oder gleich 246 ist und sonst 13 addiert mit der Länge der Daten.

Dass ein Byte fehlt (eigentlich sind ja 5 + 255 = 260 Bytes möglich) liegt daran, dass die APDU dem MuscleCard-Applet über die Funktion

```
byte[] sendApduToMscApplet(byte[] apdu)
```

übergeben wird. Diese Funktion erwartet eine APDU ohne LC-Byte als Eingabe und baut daraus eine *CommandAPDU*. Der in diesem Fall benutzte Konstruktor von *CommandAPDU* erwartet die Header-Felder der APDU einzeln und den Datenteil extra und fügt das LC-Byte selbstständig ein.

Anschließend baut die Methode *readObjectsPersistent* aus dem Header (siehe [DC01, S. 50]), der Object ID, die sie dem Dateinamen entnimmt, dem Offset, der Länge der in dieser APDU zu verschickenden Daten (diese ist Länge der APDU minus 13) und den Daten selbst eine APDU zusammen und verschickt sie an das MuscleCard-Applet. Sie erhöht den Offset um die Länge der verschickten Daten. Dieser ganze Ablauf wiederholt sich, bis der Offset gleich der Länge der eingelesen Daten aus der Datei ist.

Einlesen der persistent gespeicherten PINs

Zum Einlesen der persistent gespeicherten PINs wurde die Methode

```
private void readPinsPersistent()
```

implementiert.

Sie liest der Reihe nach die Dateien „pin0“ bis „pin15“ aus dem Ordner „pins“ aus. Prinzipiell sind im MuscleCard-Applet bis zu 16 PINs möglich. Im vorgestellten Anwendungsfall mit OpenSC wird aber nur PIN 1 genutzt.

Die APDU zum Erstellen der PIN (d. h. die eingelesenen Daten ab dem dritten Byte) wird so wie sie ist an das MuscleCard-Applet geschickt. Anschließend müssen noch die Fehlbedienungszähler korrekt eingestellt werden. Dazu wird so oft wie im ersten bzw. zweiten Byte angegeben versucht, die PIN mit einer falschen PIN zu verifizieren bzw. zu entsperren. Die falsche PIN wird erhalten, indem die richtige PIN aus der APDU zur Erstellung der PIN extrahiert und das erste Byte mit 0xFF mit der bitweisen XOR-Funktion verrechnet wird. Dabei ist anzumerken, dass bei diesem Vorgang die Authentisierung der PIN aufgehoben wird. Das Einlesen der persistent gespeicherten PINs sollte daher nach dem Einlesen der persistent gespeicherten Objekte und Schlüssel geschehen, da dafür im Fall der Virtual Keycard PIN 1 authentisiert worden sein muss.

In Virtual Keycard wird das *MSCHostApduService*-Objekt in *MainActivity* initialisiert. Dabei wird zunächst der Konstruktor aufgerufen und anschließend die Methode *setup_mscapplet*, in der die PINs mit den Fehlversuchen eingestellt werden. Wurde der Konstruktor von *MSCHostApduService* bereits ausgeführt, und wurde die setup-APDU durch *setup_mscapplet* an das MuscleCard-Applet geschickt, ist die Kommunikation per NFC bereits möglich. Die PINs mit den Fehlversuchen

wurden aber noch nicht eingelesen. Dieser Zustand existiert nur sehr kurz, da als nächster Schritt von *setup_mscapplet* die Schlüssel, Objekte und PINs mit den Fehlversuchen auf das MuscleCard-Applet übertragen werden. Es wäre aber dennoch in dieser Zeit möglich, Login-Versuche auf dem MuscleCard-Applet zu unternehmen, auch wenn die PIN eigentlich schon gesperrt ist (also die möglichen Fehlversuche auf dem persistenten Speicher bereits „0“ betragen). Um diesen Angriff zu verhindern, könnte man noch implementieren, dass erst APDUs von außen angenommen werden, nachdem die Fehlversuche der PINs auf das MuscleCard-Applet übertragen wurden.

Einlesen der persistent gespeicherten Schlüssel

Zum Einlesen der persistent gespeicherten Schlüssel wurde die Methode

```
private void readKeysPersistent()
```

implementiert.

Im MuscleCard-Applet können bis zu 16 Schlüssel gespeichert werden (siehe [DC01, S. 21]). Die Methode geht die Dateien keys/key0 bis keys/key15 durch und liest die Daten ein. An Hand der ersten 6 Bytes liest sie die ACLs ein. Sie übergibt dann die Nummer des Schlüssels (erkennbar am Dateinamen), den Schlüssel selbst und die ACLs an die Methode

```
private void importKeyFromData(byte number, byte[] data, byte[]
    acl_r, byte[] acl_w, byte[] acl_u).
```

Diese Methode löscht das Import-Objekt auf dem MuscleCard-Applet und legt es neu mit den gegebenen ACLs in der Größe des gegebenen Schlüssels an. Anschließend ruft sie die Methode

```
private void writeImportObject(byte[] data)
```

auf. Diese Methode schreibt, ähnlich wie beim Einlesen von Objekten, in maximal 246-Byte-Portionen die gegebenen Daten in das Import-Objekt auf dem MuscleCard-Applet. Danach ruft *importKeyFromData* die Methode

```
private void importKey(byte number, byte[] acl_r, byte[] acl_w,
    byte[] acl_u)
```

auf, die veranlasst, dass die Daten, die im Import-Objekt stehen, als Schlüssel unter der gegebenen Nummer mit den gegebenen ACLs in das MuscleCard-Applet importiert werden. (Siehe auch [DC01, S. 24])

Dieses Vorgehen wird für alle im Ordner „keys“ befindlichen Dateien wiederholt.

Abfragen der Nutzerentscheidung zu Entschlüsselungs- und Signaturvorgängen

Als zusätzliche Sicherheitsmaßnahme wurde implementiert, dass vor jedem Entschlüsseln und vor jedem Signieren der Nutzer direkt auf dem Smartphone gefragt wird, ob die Aktion wirklich durchgeführt werden soll.

Treffen APDUs ein, die anzeigen, dass Daten entschlüsselt oder signiert werden sollen, geschieht das in einem *MSCHostApduService*-Objekt (bzw. in einem *processCommandApduWrapper*-Objekt). Dieses Objekt soll dann eine Funktion aufrufen, die ein Fenster erzeugt, in dem der Nutzer sieht, welche kryptographische Operation durchgeführt werden soll und welche Daten verarbeitet werden sollen. Der Nutzer soll entscheiden, ob die Operation durchgeführt werden soll oder nicht. Da die App Virtual Keycard für das GUI zuständig ist, lag es nahe, diese Funktion in der *MainActivity* anzusiedeln, da diese zu dem Zeitpunkt ohnehin angezeigt wird.

Zu diesem Zweck könnte man die Klasse *MainActivity* aus Virtual Keycard in *MSCHostApduService* importieren und eine Referenz auf das *MainActivity*-Objekt im *MSCHostApduService*-Objekt speichern. Dann würde aber eine zirkuläre Abhängigkeit zwischen den beiden Projekten bestehen. Um das zu vermeiden, wurde im Projekt Muscle Card On Android das Interface

```
public interface AskForOk
```

hinzugefügt. Es besteht nur aus der Funktion

```
public boolean askForOk(String data).
```

Dieses Interface wurde in *MainActivity* aus Virtual Keycard importiert und es wurde angegeben, dass *MainActivity* dieses Interface implementiert. Die Funktion *askForOk* in *MainActivity* erstellt ein Dialogfeld, in dem der Nutzer erfährt, welche kryptographische Operation auf welchen Daten durchgeführt werden soll, und in dem er entscheiden soll, ob die Operation durchgeführt werden soll. Solange sich der Nutzer nicht entschieden hat, wird der restliche Programmablauf unterbrochen.

Um diese Funktion nun aus einem Objekt der Klasse *MSCHostApduService* aufrufen zu können, wird der enthaltene *Context* (der bereits zur Bestimmung des Verzeichnisses, in dem Daten persistent abgespeichert werden, benutzt wurde) in ein *AskForOk*-Objekt gecastet und die Methode *askForOk* wird aufgerufen:

```
boolean result = ((AskForOk) context).askForOk(data);
```

Der übergebene String enthält die kryptographische Operation und die Daten, die verarbeitet werden sollen. Welche kryptographische Operation durchgeführt werden soll, wird aus der APDU ausgelesen, die die kryptographische Operation einleitet. Um die Daten zu erhalten, die verarbeitet werden sollen, wird das Import-Objekt des MuscleCard-Applets ausgelesen.

6.3.5 androsmex

Um PACE auf einem Smartphone durchführen zu können, wurde androsmex (siehe [ts14]) als Bibliothek benutzt. androsmex stellt PACE für die Leserseite bereit, sodass man das Smartphone benutzen kann, um z. B. mit dem neuen Personalausweis (nPA) zu kommunizieren. Ole Richter hat androsmex im Zuge seiner Bachelorarbeit um die Karten-Funktionalität von PACE erweitert (siehe [Ric14]), sodass dieses erweiterte androsmex genutzt werden konnte, um PACE für die Karten-Seite auf dem Smartphone durchführen zu können.

Die folgende Grafik soll veranschaulichen, welche Bereiche der Kommunikation durch den PACE-Kanal geschützt sind. Im grünen Bereich werden verschlüsselte APDUs ausgetauscht. In Virtual Keycard und OpenSC werden die APDUs dann wieder entschlüsselt und liegen im Klartext vor.

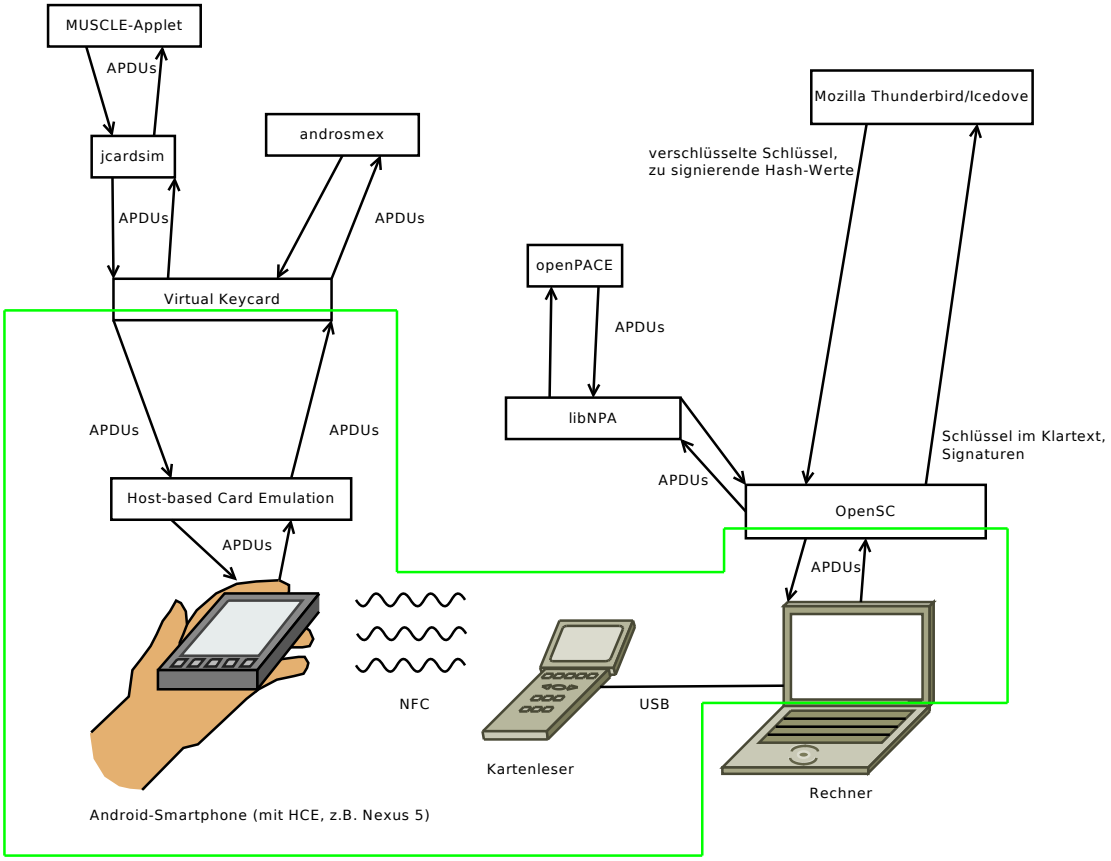


Abbildung 20: Veranschaulichung des PACE-Kanals.

androsmex wurde wie folgt in Virtual Keycard integriert:

Im Projekt Muscle Card on Android ist eine Klasse *Pace* enthalten, die auf die Klasse *PaceOperator* aus androsmex zugreift. Die Klasse *Pace* wird mit einer PIN initialisiert und initialisiert selbst ein *PaceOperator*-Objekt mit der PIN. Außerdem liest sie die Datei EF.Cardaccess (wobei diese als Byte-Array fest im Code gespeichert ist, man könnte noch implementieren, dass sie tatsächlich erst zur Laufzeit ausgelesen wird) und reicht sie an das *PaceOperator*-Objekt weiter, dieses legt damit das Authentication Template fest.

Pace enthält die Methode

```
public byte[] performPace(byte[] apdu),
```

die die nötigen Schritte zum Aufbau eines PACE-Kanals durchführt. Die App (d. h. genauer, das *MSCHostApduService*-Objekt) muss nur alle APDUs mit Instruktionsbyte 0x86 an diese Methode weiterleiten und die Rückgabewerte an den Leser zurückgeben, um den PACE-Kanal aufzubauen (und alle APDUs mit Instruktionsbyte 0x22 mit dem Statuswort 0x9000 beantworten).

Ist der PACE-Kanal aufgebaut, enthält das *PaceOperator*-Objekt ein *SecureMessaging*-Objekt. Die in der Klasse *Pace* enthaltenen Methoden

```
public byte[] decryptCommandApdu(byte[] apdu)
```

und

```
public byte[] encryptResponseApdu(byte[] apdu)
```

nutzen dieses Objekt, um APDUs zu entschlüsseln und zu verschlüsseln. Sie bekommen Byte-Arrays als Eingabe und geben Byte-Arrays zurück, auf diese Weise bleibt die Datenstruktur, die in der Methode processCommandApdu aus *MSCHostApduService* üblich ist, gewahrt. In den beiden Methoden müssen die Byte-Arrays in CommandApdus bzw. in ResponseApdus und wieder zurück umgewandelt werden, um den gewünschten Datenstrukturen der *SecureMessaging*-Klasse aus androsmex zu entsprechen.

Kommt eine „get Nonce“-APDU (d. h. 10 86 00 00 02 7C 00 00) in der Methode *processCommandApdu* an, wird ein *Pace*-Objekt initialisiert, dem PIN 1 aus dem MuscleCard-Applet übergeben wird. Der Wert der PIN wird von der Methode

```
private byte[] getPinValue(byte number)
```

ausgelesen. An dieser Stelle zahlt es sich aus, dass die PINs ohnehin auf dem persistenten Speicher in für die App auslesbarer Form gespeichert wurden. Wäre das MuscleCard-Applet als serialisiertes Objekt gespeichert worden, wäre es nicht ohne weiteres möglich, die PIN aus dem MuscleCard-Applet auszulesen (da das MuscleCard-Applet dafür keine Methoden bietet). Hat eine eintreffende APDU das Instruktionsbyte 0x86, übergibt *processCommandApdu* aus *MSCHostApduService* die APDU an *performPace*. Hat sie das Instruktionsbyte 0x22, gibt *processCommandApdu* einfach das Statuswort 0x9000 zurück.

Ist die APDU verschlüsselt (erkennbar am Class-Byte 0xBC), übergibt sie die APDU so wie sie ist (also als Byte-Array) an die Methode *decryptCommandApdu* aus dem *Pace*-Objekt. Sie bekommt dann die APDU entschlüsselt als Byte-Array zurück und kann mit ihr wie gewohnt verfahren. Es wird in dieser Methode noch durch eine *boolean*-Variable markiert, dass die APDU verschlüsselt war. Wurde die APDU verarbeitet und steht die Antwort fest, so wird sie einfach als Byte-Array an die *encryptResponseApdu*-Methode aus dem *Pace*-Objekt übergeben. Zurück kommt die APDU als verschlüsseltes Byte-Array, dieses kann so an den Leser zurückgegeben werden.

6.4 Software auf der Rechner-Seite

Auf dem Rechner war als Betriebssystem Debian Jessie installiert (Stand ca. September 2014). Die einzelnen Programmversionen werden in den Unterkapiteln zu den Programmen angegeben.

6.4.1 OpenSC

OpenSC ist eine Smartcard-Bibliothek, d. h. OpenSC implementiert Funktionen, mit deren Hilfe man Aktionen auf Smartcards durchführen kann. Um mit Smartcards zu kommunizieren, spricht OpenSC den Treiber eines Lesers an, das kann z. B. ein PC/SC-Treiber sein. Der Treiber ist dafür verantwortlich, APDUs zum Leser zu schicken und APDUs von ihm zu empfangen [vik12]. OpenSC ist in C geschrieben (siehe [Ope14]) und kam in der Version 0.13.1 und 0.14 zum Einsatz. Der Proof of Concept benutzt OpenSC 0.14.

OpenSC beinhaltet Treiber für verschiedene Arten von Smartcards. Um die im Zuge dieser Bachelorarbeit entwickelte virtuelle Smartcard über OpenSC anzusprechen, konnte der in OpenSC enthaltene „muscle“-Treiber mit Anpassungen genutzt werden. Dieser beinhaltet Funktionen, die mit dem MuscleCard-Applet kommunizieren.

Der „virtualkeycard“-Treiber

Um PACE zu unterstützen, wurde (im Rahmen dieser Bachelorarbeit) ein neuer Treiber namens „virtualkeycard“ implementiert, der den „muscle“-Treiber ergänzt. So wird in der Funktion, die für das authentisieren per PIN verantwortlich ist, ein PACE-Durchlauf durchgeführt. Dazu wurde auf libnpa (siehe 6.4.2) zurückgegriffen und dem „muscle“-Treiber einige Funktionalität hinzugefügt, um die Funktion in libnpa korrekt aufrufen zu können. libnpa selbst greift auf libeac, d. h. auf OpenPACE zu, um PACE durchzuführen. Auf diese Weise wird das PACE-Protokoll für die Leserseite auf dem Rechner durchgeführt. Die nachfolgende Kommunikation ist durch PACE abgesichert, d. h. vertraulich und authentisch.

Es wäre auch möglich gewesen, einen Komfortleser zu verwenden, der bereits PACE beherrscht. In OpenSC lässt sich auslesen, ob ein Leser PACE beherrscht oder nicht. Mit einigen Anpassungen im „virtualkeycard“-Treiber wäre es dann möglich, den Leser selbst mit dem Smartphone PACE durchführen zu lassen.

Wenn man nicht speichert, ob PACE für die Karte bereits ausgeführt wurde, wird beim nächsten Aufruf des PIN-Kommandos erneut versucht, einen PACE-Kanal (in dem bereits existierenden PACE-Kanal) aufzubauen. Daher muss im Treiber vermerkt werden, ob bereits ein PACE-Kanal existiert. Falls ja, sollte kein neuer PACE-Kanal aufgebaut werden. Die Parameter (z. B. die vereinbarten Schlüssel) von PACE werden in einer Karten-Struktur namens

```
sc_card_t
```

gespeichert. Wird das Smartphone vom Leser entfernt und wieder darauf gelegt, wird eine neue Karten-Struktur erstellt. Würde man die Information, ob es bereits einen PACE-Kanal gibt, in einer globalen Variable speichern, würde also in dem Fall kein PACE-Kanal aufgebaut werden, obwohl für die Karten-Struktur noch gar kein PACE-Kanal existiert. Die nachfolgende Kommunikation würde also unverschlüsselt stattfinden. Daher muss die Information, ob ein PACE-Kanal aufgebaut wurde, in der Karten-Struktur gespeichert werden. Im Treiber „virtualkeycard“ geschieht das auch so, die Information steckt in den Daten des Treibers, die in den einzelnen Karten-Strukturen abgelegt werden.

Der Treiber enthält neben den üblichen „muscle“-Funktionen auch noch eine Datei EF.Cardaccess (die im Code fest als Byte-Array gespeichert ist, man könnte noch implementieren, dass diese tatsächlich zur Laufzeit von der Karte ausgelesen wird) und eine Funktion

```
void muscle_get_cache(struct sc_card *card, unsigned char pin_id,
    const unsigned char **pin, size_t *pin_length, unsigned char
    **ef_cardaccess, size_t *ef_cardaccess_length),
```

die diese als Information für den PACE-Kanal-Aufbau hinzufügt.

Außerdem wird in der Funktion

```
int muscle_pin_cmd(sc_card_t *card, struct sc_pin_cmd_data *cmd,
    int *tries_left)
```

der PACE-Kanal mit der gegebenen PIN aufgebaut (indem eine Funktion aus libnpa aufgerufen wird).

Die Funktion

```
static int muscle_compute_signature(sc_card_t *card, const u8 *
    data, size_t data_len, u8 * out, size_t outlen)
```

wurde verändert. Der Grund dafür ist, dass Icedove beim Entschlüsseln von Daten ein Byte weniger als Antwort erwartet als beim Signieren. Sendet man beim Entschlüsseln ein Byte zuviel, kann Icedove keine verschlüsselten Mails entschlüsseln. Im „muscle“-Treiber war aber für beide Fälle die gleiche „Cipher direction“ angegeben, in beiden Fällen wurde eine Entschlüsselung verlangt, da das MuscleCard-Applet noch keine Signaturen unterstützt. Es war also in Virtual Keycard nicht zu unterscheiden, welcher der beiden Fälle gemeint war. Daher wurde *muscle_compute_signature* so verändert, dass die Cipher direction „sign“ ist. Daran kann Virtual Keycard unterscheiden, welcher Fall gefragt ist, und je nachdem aus der Antwort des MuscleCard-Applets (die in beiden Fällen immer noch mit der Cipher direction „decrypt“ erstellt wird) das letzte Byte entfernen oder nicht.

Für das Entschlüsseln und Signieren mit 2048 Bit RSA-Schlüsseln war es nötig, dass die zu verarbeitenden Daten in einem Objekt auf der Karte gespeichert werden, da sie wegen der Größe von 256 Byte nicht in einer einfachen APDU geschickt werden können und die Übertragung von extended APDUs auf das verwendete Smartphone auf Grund des verwendeten Android-Betriebssystems nicht funktionierte (dieses kann man mit einem Software-Patch dazu bringen, größere

APDUs zu versenden und zu empfangen, siehe [Pro14e]). Sollen Daten entschlüsselt oder signiert werden, werden die Funktionen *muscle_compute_signature* oder

```
static int muscle_decipher(sc_card_t * card, const u8 * crgram,
    size_t crgram_len, u8 * out, size_t out_len)
```

aufgerufen. Diese rufen dann beide die Funktion

```
int msc_compute_crypt(sc_card_t *card, int keyLocation, int
    cipherMode, int cipherDirection, const u8* data, u8*
    outputData, size_t dataLength, size_t outputDataLength)
```

auf. In dieser wird unterschieden, ob die zu verarbeitenden Daten in ein Objekt geschrieben werden müssen oder in einer APDU mitgeschickt werden können, indem überprüft wird, ob die Datenlänge kleiner ist als die, die man maximal in einer APDU schicken kann oder ob die Karte extended APDUs unterstützt. Stellt man also im Virtual Keycard-Treiber ein, dass die Karte keine extended APDUs unterstützt, so werden die Daten in ein Import-Objekt geschrieben. Das Problem dabei ist, dass die Funktion

```
int sc_check_apdu(sc_card_t *card, const sc_apdu_t *apdu)
```

(aus der Datei libopensc/apdu.c), die beim Versenden der Dateien anschließend aufgerufen wird, überprüft, ob die APDU vom Typ extended ist und ob die Karte extended APDUs unterstützt, zu dem Ergebnis kommt, dass die APDU extended ist und die Karte keine extended APDUs unterstützt und mit einem Fehler abbricht.

Um ins Import-Objekt zu schreiben, müssen wir also extended APDUs abschalten. Tun wir das, kommt es aber zu einem Fehler. Die Funktion *msc_compute_crypt* befindet sich in der Datei

```
muscle.c,
```

die Teil von OpenSC ist. Würden wir sie verändern, könnte man also nicht mehr das originale OpenSC verwenden. Daher wurde die Funktion *msc_compute_crypt* mit in den Virtual Keycard-Treiber gezogen, verändert, sodass sie immer die Daten per Import-Objekt überträgt und nie per einzelner APDU (was nicht schlimm ist, da so immer noch 1024 Bit Schlüssel und auch größere verwendet werden können) und in

```
int virtual_keycard_compute_crypt(sc_card_t *card, int keyLocation
    , int cipherMode, int cipherDirection, const u8* data, u8*
    outputData, size_t dataLength, size_t outputDataLength)
```

umbenannt. Beim Entschlüsseln von Daten ist die Ausgabe des MuscleCard-Applets wie bereits beschrieben um ein Byte kürzer. Es muss daher von OpenSC auch ein Byte weniger gelesen werden; wird dies nicht gemacht, versucht OpenSC über das Ende des Export-Objekts hinaus zu lesen und erhält eine Fehlermeldung, daraufhin bricht OpenSC den Entschlüsselungsvorgang mit einer Fehlermeldung ab. Wie viele Bytes gelesen werden sollen, ist in der Funktion

```
static int msc_compute_crypt_final_object(sc_card_t *card, int
    keyLocation, const u8* inputData, u8* outputData, size_t
    dataLength, size_t* outputDataLength)
```

festgelegt, die von *msc_compute_crypt* (bzw. von *virtual_keycard_compute_crypt*) aufgerufen wird. Diese liest immer so viele Daten ein, wie sie auch zum Bearbeiten erhalten hat. Um dies zu ändern, wurde die Funktion in veränderter Form in den Virtual Keycard-Treiber übernommen. Da aus den Parametern nicht erkennbar ist, ob gerade entschlüsselt oder signiert werden soll, wurde der Parameter

```
int decrypt
```

hinzugefügt. Dieser kann in *virtual_keycard_compute_crypt* leicht gesetzt werden, da hier die Cipher Direction als Parameter vorliegt. So erhält man die Funktion

```
static int virtual_keycard_compute_crypt_final_object(sc_card_t *
    card, int keyLocation, const u8* inputData, u8* outputData,
    size_t dataLength, size_t* outputDataLength, int decrypt).
```

Installation des „virtualkeycard“-Treibers

Der neue Treiber kann als interner oder als externer Treiber in OpenSC integriert werden. Wenn man den Treiber als internen Treiber integriert, muss man einige Änderungen in OpenSC selbst vornehmen, bevor man kompiliert (siehe [Jan14]).

Wenn man den Treiber als externen Treiber integriert, kann man OpenSC ohne Veränderungen kompilieren und den Treiber als Shared Object dynamisch laden. Dazu muss man in der Konfigurationsdatei von OpenSC (opensc.conf) die richtigen Einstellungen vornehmen. So muss der Pfad zum externen Treiber angegeben werden und ein Pfad zum dazugehörigen PKCS15init-Treiber. Man kann dann noch festlegen, dass beim ATR des Smartphones der Treiber direkt ausgewählt werden soll. Wie die Stellen der Konfigurationsdatei genau auszusehen haben, findet man im Anhang (siehe 9.2). Es waren noch einige zusätzliche Änderungen an der „muscle“-Treiberdatei notwendig, um sie als externen Treiber zu integrieren, so erwartet OpenSC z. B. bestimmte Funktionen in den Shared Object-Dateien. Die eine ist

```
void *sc_module_init(const char *name)
```

und gibt eine Funktion zurück, die die Funktionen des Treibers in einer bestimmten Datenstruktur zurückgibt. Die andere ist

```
const char *sc_driver_version(void)
```

und gibt die OpenSC-Version zurück, mit der der Treiber gebaut wurde. OpenSC erwartet auch, dass ein Profil für den Treiber angelegt wird; es reicht, das „muscle“-Profil zu kopieren (nach $PREFIX/share/opensc/). Befehle zum Kompilieren und Linken der Treiberdateien finden sich im Anhang (siehe 9.1).

Zu beachten ist, dass libopensc die Symbole aus muscle.c und muscle-filesystem.c nicht als globale, sondern als lokale Symbole beinhaltet, wie

```
nm -g /usr/lib/libopensc.so
```

bestätigt. Aus diesem Grund kann z. B. das Symbol *msc_select_applet* nicht gefunden werden, auch wenn gegen libopensc dynamisch gelinkt wurde. Es wurde

demnach statisch gegen die muscle-Dateien gelinkt. Auch card.o und sc.o wurden statisch gelinkt.

Außerdem ist zu beachten, dass libopensc dynamisch gegen ein gepatchtes OpenSSL gelinkt werden muss [Mor14]. Zur Installation des Treibers wird also erst einmal libnpa wie auf der Seite [Mor14] installiert. Es empfiehlt sich, ein PREFIX ungleich „/usr“ zu wählen, um nicht die auf dem System ansonsten genutzte OpenSSL-Bibliothek (d. h. libcrypto und libssl) zu überschreiben. Anschließend kopiert man den Treiber für Virtual Keycard (d. h. „card-virtualkeycard.c“) nach

```
$PREFIX/vsmartcard/npa/src/opensc/src/libopensc/
```

und den PKCS15-Treiber für Virtual Keycard (d. h. „pkcs15-virtualkeycard.c“) nach

```
$PREFIX/vsmartcard/npa/src/opensc/src/pkcs15init/.
```

Dann kompiliert und linkt man die Dateien wie im Anhang angegeben (siehe Kapitel 9.1) und fügt der Konfigurationsdatei die Änderungen aus dem Anhang hinzu (siehe Kapitel 9.2). Jetzt kann man die OpenSC-Programme aus

```
$PREFIX/bin
```

und die Bibliotheken aus

```
$PREFIX/lib
```

benutzen, um mit Virtual Keycard zu kommunizieren. Durch das Einbinden des Treibers als externen Treiber konnte OpenSC also im Originalzustand gebaut werden und das Vertrauen in die OpenSC-Installation bleibt erhalten. Außerdem wurde dadurch eine zirkuläre Abhängigkeit vermieden. libnpa hängt von OpenSC ab, wenn wir den Treiber (der ja von libnpa abhängt) direkt in OpenSC gebaut hätten, würde OpenSC auch von libnpa abhängen. So hängt OpenSC aber nicht von libnpa ab und der Treiber wird separat gebaut, er hängt sowohl von OpenSC als auch von libnpa ab.

Personalisieren der Smartcard mit OpenSC

Mit Hilfe von OpenSC wird die PKCS15-Struktur auf der simulierten Smartcard erstellt und verwaltet. Je nach verwendetem System kann es helfen, vor dem Aufruf von Programmen, die OpenSC nutzen, die Umgebungsvariable LD_LIBRARY_PATH auf $PREFIX/lib zu setzen und sie zu exportieren. Auf 64-bit-Systemen müssen Pfadangaben mit „lib“ eventuell um Pfadangaben mit „lib64“ ergänzt werden.

Wichtig ist, dass zu dem Zeitpunkt, zu dem die PKCS15-Struktur erstellt werden soll, alle Identitäten auf dem MuscleCard-Applet authentisiert worden sind, die zum Erstellen und Schreiben von Objekten benötigt werden. Im Fall der Virtual Keycard handelt es sich dabei um PIN1. Diese ist nach dem Start der App automatisch eingeloggt, wenn das Objekt mit der ID 50154401 nicht existiert (was bei einer Karte, auf der noch keine PKCS15-Struktur erzeugt wurde, der Fall ist). Das zu OpenSC gehörige Programm pkcs15-tool kann zwar wie folgt PINs einloggen:

```
$PREFIX/bin/pkcs15-tool --verify-pin 1
```

Allerdings funktioniert das nur, wenn die PKCS15-Struktur auf der Karte bereits erzeugt wurde. Im Zuge der Erstellung der PKCS15-Struktur wird auch das Objekt 50154401 erzeugt, somit ist PIN 1 ab dann nach dem Start der App immer automatisch nicht authentisiert. Man kann PIN 1 dann allerdings mit OpenSC authentisieren.

Ein weiterer Teil von OpenSC ist das Programm pkcs15-init. Mit diesem kann die PKCS15-Struktur auf der Karte erstellt werden:

```
$PREFIX/bin/pkcs15-init --create-pkcs15 --card-profile
    virtualkeycard
```

Als nächstes generiert man ein RSA-Schlüsselpaar auf der simulierten Smartcard:

```
$PREFIX/bin/pkcs15-init --generate-key rsa/2048 --auth-id ff --key
    -usage sign,decrypt
```

Um E-Mails mit Icedove signieren und entschlüsseln zu können, benötigt man ein Zertifikat. Dabei handelt es sich um den öffentlichen Schlüssel, die Information, zu wem genau der öffentliche Schlüssel gehört und eine Signatur einer vertrauenswürdigen Instanz (Certificate Authority, CA), die überprüft hat, ob der öffentliche Schlüssel auch tatsächlich zu der angegebenen Person gehört. Um ein Zertifikat zu bekommen, müssen wir einen „Certificate Signing Request“ (CSR) erstellen, also die Anfrage an eine CA, unseren öffentlichen Schlüssel mit unseren Angaben zur Person zu unterschreiben. Dieser soll dann von der CA bearbeitet werden. Der CSR besteht aus dem öffentlichen Schlüssel, den Angaben zur Person und der Signatur eines Hashwerts dieser Daten mit dem dazugehörigen privaten Schlüssel. Wir erhalten den CSR wie folgt:

Um zu sehen, welche Schlüssel verfügbar sind und welche ID und Slot-Nummer sie haben, kann man den folgenden Befehl benutzen:

```
$PREFIX/bin/pkcs11-tool --module $PREFIX/lib/opensc-pkcs11.so -L -
    O
```

Wir setzen LD_LIBRARY_PATH so, dass das von uns kompilierte OpenSC mit dem von uns kompilierten OpenSSL benutzt wird und starten OpenSSL:

```
export LD_LIBRARY_PATH=$PREFIX/lib
$PREFIX/bin/openssl
```

Wir geben in die openssl-Konsole ein:

```
engine -t dynamic -pre SO_PATH:/usr/lib/ssl/engines/engine_pkcs11.
    so -pre ID:pkcs11 -pre LIST_ADD:1 -pre LOAD -pre MODULE_PATH:
    $PREFIX/lib/opensc-pkcs11.so
req -engine pkcs11 -new -key slot_01-
    id_298d1ccb990f7a44a0523e1e395d07e056e48fa1 -keyform engine -
    out csroutofmsc.csr -sha256
```

Hierbei ist zu beachten, dass die Zahlen hinter slot und id angepasst werden müssen. Außerdem ist wichtig, dass in der ersten Zeile die Datei opensc-pkcs11.so aus

dem kompilierten OpenSC stammt, mit dem auch libnpa und der „virtualkeycard“-Treiber gebaut wurden. $PREFIX muss durch den tatsächlichen Pfad ersetzt werden, da die OpenSSL-Konsole die Umgebungsvariable nicht auflöst. Auch ist wichtig, dass das Paket „engine_pkcs11“ installiert ist, der Pfad zur Datei „engine_pkcs11.so“ kann je nach verwendetem System eventuell abweichen.

Welche Daten in einem CSR stehen, kann man sich mit

```
openssl req -text -in csroutofmsc.csr
```

anzeigen lassen. Eine beispielhafte Ausgabe dieses Befehls befindet sich im Anhang (siehe 9.3).

Der CSR kann jetzt an die CA geschickt und dort bearbeitet werden, man erhält dann das Zertifikat zurück. Zu Testzwecken kann man den CSR auch mit einer eigenen CA bearbeiten:

```
openssl ca -in csroutofmsc.csr -out crtoutofmsc.crt -keyfile ca.
   key -cert ~/.certificates/public/ca.pem
```

Anschließend muss das Zertifikat noch auf die Virtual Keycard übertragen werden:

```
$PREFIX/bin/pkcs15-init --store-certificate crtoutofmsc.crt --auth
   -id ff --id 298d1ccb990f7a44a0523e1e395d07e056e48fa1
```

Hierbei ist zu beachten, dass die Zahl hinter id der ID des zugehörigen Schlüsselpaars gleichen muss.

6.4.2 libnpa

Die nPA Smart Card Library (hier auch libnpa genannt), soll ein Application Programming Interface (API) für den neuen Personalausweis bieten. Die Bibliothek bietet aber auch Secure-Messaging an, wobei sie auf OpenSC angepasst ist, sodass ein Secure-Messaging-Kanal für Karten in OpenSC von libnpa eingerichtet werden kann [Mor14].

Um diese Funktionalität zu realisieren, greift libnpa auf OpenPACE und OpenSC zurück. Diese beiden Programme bzw. Bibliotheken müssen daher vor der Installation von libnpa installiert werden. Um das tun zu können, muss auch eine gepatchte Version von OpenSSL installiert werden.

Schlussendlich wird vom Treiber der Virtual Keycard in OpenSC nur die Funktion

```
int perform_pace(sc_card_t *card, struct
   establish_pace_channel_input pace_input, struct
   establish_pace_channel_output *pace_output, enum
   eac_tr_version tr_version)
```

aus libnpa aufgerufen. Diese übernimmt eine Karten-Struktur (der vorher ein EF.Cardaccess hinzugefügt wurde) und führt den Aufbau eines PACE-Kanals mit dieser Karte durch. Es mussten keine Veränderungen an libnpa vorgenommen werden.

libnpa kam in der Form des git-Commits
04d26c376c7e8e3ff7f314942fd245d5fff150a0 zum Einsatz.

6.4.3 OpenPACE

OpenPACE (hier auch libeac genannt) implementiert Extended Access Control (EAC) Version 2, wie es in der technischen Richtlinie BSI TR-03110 spezifiziert wird [uDO14]. Die Berechnungen für die einzelnen Schritte geschehen also in libeac, während sich libnpa mit Hilfe von OpenSC um die Beschaffung und Übertragung der nötigen Daten sowie das Einrichten des Secure-Messaging-Kanals in den Datenstrukturen von OpenSC kümmert. libeac musste im Laufe der Programmierung des Software-Systems nie direkt angesprochen werden, sondern war nur als Abhängigkeit von libnpa im Projekt involviert.

OpenPACE kam in der Form des git-Commits 5d4c25848b0ec45329213475152d4f3cf19d342c zum Einsatz.

6.4.4 Mozilla Thunderbird/Icedove

Icedove wurde als Mail-Client auf dem Rechner in der Version 24.5.0 verwendet.

Icedove kann OpenSC benutzen, um mit Virtual Keycard zu kommunizeren. Dazu wählt man

Edit->Preferences->Bild „Advanced“->Reiter „Certificates“->Security Devices.

In dem sich öffnenden Geräte-Manager wählt man „Load“ und gibt den Pfad zur Datei opensc-pkcs11.so an. Das kann zum Beispiel

```
$PREFIX/lib/opensc-pkcs11.so
```

sein. Falls das nicht funktioniert, kann es helfen, alle Smartcards/Smartphones vom Leser zu entfernen und es dann noch einmal zu probieren.

Von nun an kann Icedove per OpenSC auf Virtual Keycard zugreifen. Man sollte noch das Zertifikat für den passenden Mail-Account einstellen. Dazu macht man einen Rechtsklick auf den Mail-Account und klickt „Settings“ an. Man wählt dann den Unterpunkt „Security“ des Accounts und geht auf „Select“ . Man wird aufgefordert, die PIN für die Smartcard einzugeben. Anschließend kann man das Zertifikat auswählen.

Von nun an werden einkommende Mails, die mit dem öffentlichen Schlüssel verschlüsselt wurden, entschlüsselt, falls das Smartphone auf dem Leser liegt und die PIN eingegeben wird. Möchte man eine Mail signieren, so klickt man während des Verfassens der Mail S/MIME->Digitally Sign This Message an. Liegt das Smartphone auf dem Leser und gibt man die PIN ein, wird die Mail signiert (siehe auch Kapitel 6.1).

7 Sicherheitsimplikationen des vorgestellten Systems und Ausblick

Das vorgestellte System kann zur Verbreitung von E-Mail-Verschlüsselung beitragen, ohne dass der Schlüssel direkt auf dem Rechner liegen muss, da der Nutzer um den Kauf der Smartcard herum kommt. Eine Eigenschaft, die aus dem Anwendungsfall mit einer richtigen Smartcard übernommen werden konnte, ist, dass der Schlüssel auf dem Smartphone erzeugt wird und dieses (im besten Fall) auch nicht verlässt. Dieses Vorgehen wird nicht bei allen Mail-Anbietern durchgeführt.

Die Humboldt-Universität zu Berlin beispielsweise bietet zwar das Ausstellen von Zertifikaten an, dabei erhalten die Nutzer allerdings die privaten Schlüssel von der Humboldt-Universität zu Berlin, müssen dieser also vertrauen, dass sie den privaten Schlüssel sicher verwahrt bzw. gelöscht hat und ihn nicht benutzt, um Mails mitzulesen oder Mails fälschlicherweise zu signieren (siehe [Pla14c], [Pla14a] und [Pla14b]).

Das vorgestellte System kann (mit einigen zusätzlichen Ideen) Nutzern das Erstellen von Schlüsseln und CSRs so weit vereinfachen, dass sie problemlos ihre eigenen CSRs stellen können, ohne viel Fachwissen zu benötigen. Siehe dazu die Kapitel 7.1 und 7.2.

Zu beachten ist allerdings, dass zwischen Rechner und Smartphone keine so klare Trennung wie bei einer Smartcard besteht, da das Smartphone auch per USB an den Rechner angeschlossen wird und auch über viele andere Kanäle mit anderen Rechnern kommunizieren kann (WLAN, Internet, Bluetooth, etc.). Auch ist der Speicher auf dem Smartphone leichter auslesbar. Wie wir in Kapitel 7.3 sehen werden, ist es daher notwendig, die gespeicherten Daten noch einmal besonders gegen unautorisierten Zugriff zu sichern.

Wie das vorgestellte System zeigt, können sich Systeme, die mit Smartcards arbeiten, nicht darauf verlassen, auch tatsächlich nur mit Smartcards benutzt zu werden, die bestimmte Sicherheitseigenschaften haben. Sie könnten ebenso gut mit einem Smartphone kommunizieren, ohne dass dies bemerkt wird.

7.1 Automatisches Erzeugen von PKCS15-Strukturen und Schlüsseln

Das Erzeugen der PKCS15-Struktur und das Anstoßen der Erzeugung der Schlüssel in Virtual Keycard muss nicht zwingend von der Rechner-Seite aus via OpenSC erfolgen, sondern könnte beim ersten Start der App automatisch durchgeführt werden. Dazu könnte man z. B. die APDUs abfangen, die OpenSC versendet, um die PKCS15-Struktur und die Schlüssel zu erzeugen und diese beim Start der App an das MuscleCard-Applet schicken, falls PKCS15-Struktur und Schlüssel noch nicht existieren. Auf diese Weise wäre die App einfacher zu benutzen; es wäre nicht mehr nötig, die Werkzeuge aus OpenSC für diesen Schritt zu nutzen.

7.2 Erstellen und Ausfüllen des CSRs auf dem Smartphone

Der Certificate Signing Request muss nicht zwingend von der Rechner-Seite aus unter Benutzung von OpenSSL und OpenSC geschehen. Es wäre auch möglich, den CSR direkt auf dem Smartphone zu erstellen.

Für den CSR benötigt man den öffentlichen Schlüssel, die vom Nutzer einzugebenden Daten (Standort, Name usw.) und die Signatur des Hashwertes über öffentlichen Schlüssel und Daten. All dies lässt sich auf dem Smartphone beschaffen. Den öffentlichen Schlüssel kann man aus dem MuscleCard-Applet auslesen, die Daten könnte man in einem GUI vom Nutzer abfragen und die Signatur könnte man vom MuscleCard-Applet erstellen lassen (nach Eingabe der PIN durch den Nutzer).

Aus diesen Daten könnte man den CSR zusammenstellen und ihn beispielsweise per Mail (vom Smartphone aus) an die CA schicken. Diese müsste dann überprüfen, ob der CSR, den sie per Mail erhalten hat, auch der vom Nutzer versendete ist und ob die Daten, die der Nutzer eingegeben hat, stimmen, und könnte im positiven Fall das Zertifikat (authentisch) per Mail an den Nutzer zurückschicken. Der Nutzer könnte dann die Mail direkt auf seinem Smartphone empfangen. Ein weiterer Teil der App könnte sein, ein GUI zur Verfügung zu stellen, mit dem der Nutzer die Zertifikatsdatei aus der Mail in das MuscleCard-Applet importieren kann.

So wäre es möglich, das Zertifikat bereits in Virtual Keycard zu haben, ohne mit dem Rechner per NFC kommuniziert haben zu müssen. Die App könnte der Nutzer aus dem Google Play Store herunterladen. Anschließend müsste er libnpa (inklusive der Abhängigkeiten) und den Treiber für Virtual Keycard bauen und OpenSC und Icedove korrekt konfigurieren. Das Bauen der nötigen Programme und Konfigurieren von OpenSC könnte man in einem Makefile automatisieren, Icedove kann per GUI konfiguriert werden. Auf diese Weise könnte man weitere Hürden für den Benutzer entfernen.

7.3 Sicheres Speichern der Schlüssel

Die vorgestellte Software speichert Schlüssel bisher im Ordner der App im internen Speicher des Smartphones. Das Android-Betriebssystem sorgt dafür, dass niemand außer der App auf den Ordner zugreifen kann, auch nicht der Benutzer des Smartphones [Pro14g]. Prinzipiell wäre es aber denkbar, z. B. über folgende Wege an die Daten zu kommen:

1. Vorausgesetzt, das Smartphone ist entsperrt: Man schließt das Smartphone per USB an den Rechner an, wählt am Smartphone aus, dass man USB-Debugging ermöglichen möchte, führt

   ```
   adb shell
   ```

   aus den Android Developer Tools auf dem Rechner aus und gibt

   ```
   run-as de.nellessen.virtual_keycard
   ```

in die sich öffnende Kommandozeile ein. Daraufhin hat man Zugriff auf die Dateien der App und somit auch auf die privaten Schlüssel. Aus „Haben und Wissen“ wird so nur „Haben“ : Um den Schlüssel zu benutzen, braucht man die PIN nicht zu wissen, man liest ihn einfach wie dargestellt aus und benutzt ihn dann. Auch dass der Schlüssel überhaupt ausgelesen werden kann, ist ein Nachteil gegenüber einer klassischen Smartcard.

2. Man „rootet“ das Smartphone, verschafft sich also Zugriff auf den root-Account des Betriebssystems und hat somit auch Zugriff auf die Dateien der App Virtual Keycard.

3. Man findet einen Weg, den internen Speicher des Smartphones auszulesen, ohne das Android-Betriebssystem auszuführen, das den Zugriff auf die Dateien untersagt.

Um Angriffe dieser Art verhindern zu können, dürfen die Daten nicht im Klartext auf dem Smartphone vorliegen, sie müssen verschlüsselt werden.

Es wäre möglich, die Daten mit einem nutzergewählten Passwort zu verschlüsseln und beim Start der App nach der Eingabe des Passworts durch den Nutzer wieder zu entschlüsseln. Dies würde die drei vorgestellten Angriffe deutlich erschweren bzw. nicht mehr profitabel machen.

Es wäre noch möglich, das Passwort durch eine Brute-Force-Verfahren zu brechen. Ob das lohnenswert ist, hängt von der Länge und der Komplexität des Passworts ab. Eine PIN (4-8 Stellen) würde einem Brute-Force-Angriff nicht standhalten, es bietet sich also nicht an, Passwort und PIN gleichzusetzen. Dass die PIN ausreichend ist, um die Funkverbindung per PACE zu sichern, liegt daran, dass ein komplexerer Schlüssel in dem Verfahren abgeleitet wird und man auch nur wenige (in der Regel drei) Versuche hat, den PACE-Kanal aufzubauen (bzw. sich auf der Smartcard einzuloggen).

Der Nutzer müsste sich also noch ein Passwort merken (und dann auch noch ein längeres, komplexereres).

Android bietet auch eine Verschlüsselung der Daten an [Goo14]. Dabei wird die gleiche PIN verwendet wie diejenige, die für das Entsperren des Bildschirms benötigt wird. Bedenken sollte man, dass diese auch über Seitenkanäle einsehbar sein kann (zum Beispiel Fettspuren auf dem Bildschirm) oder, da sie für jede Benutzung des Smartphones und nicht nur für das Benutzen des privaten Schlüssels eingegeben werden muss, auch einfach per Blick über die Schulter einsehbar ist. Die Verschlüsselung wird ab Android L Standard[Eik14]. Fraglich ist, wie der tatsächliche Schlüssel, der für die Verschlüsselung genutzt wird, aus der PIN abgeleitet wird. Geht kein zusätzliches Schlüsselmaterial (von der Hardware des Smartphones) ein, so würde dieses System keinem Brute-Force-Angriff stand halten können.

7.4 Sicheres Löschen der Schlüssel

Wenn die PIN und auch die Unblock-PIN gesperrt wurden, bietet die App keine Möglichkeit mehr, das gespeicherte Schlüsselmaterial zu nutzen. Man soll die App dann neu installieren. An dieser Stelle ist nicht ganz klar, wie sicher Android die Daten einer App löscht, die deinstalliert wird. Eventuell ist es möglich, mit Datenwiederherstellungswerkzeugen das Schlüsselmaterial zu rekonstruieren.

Einen vollständigen Schutz zu erreichen ist aber vermutlich ohnehin unmöglich. In Flash-Speichern werden defekte Bereiche einfach aussortiert, der Controller kann sie nicht mehr ansprechen. Befindet sich in einem solchen Bereich Schlüsselmaterial, so könnte ein Angreifer den Bereich herauslösen und auslesen.

Als Schutzmaßnahme könnte man den Speicherbereich der Schlüssel zu überschreiben. Damit verhindert man das Auslesen der Schlüssel über das Android-Betriebssystem.

7.5 Doppeltes Entschlüsseln von E-Mails

Auffällig ist, dass beim Entschlüsseln von E-Mails zwei Anfragen zur Entschlüsselung an die Karte gestellt werden. Man erkennt dies daran, dass man zweimal auf dem Smartphone gefragt wird, ob man Daten entschlüsseln möchte. Beim Signieren von E-Mails geschieht dies nicht, hier wird nur eine Anfrage gestellt.

Die Daten sehen dabei gleich aus; ob sie tatsächlich genau gleich sind, müsste man noch überprüfen. Lehnt man die erste Abfrage ab, so meldet Icedove, dass die E-Mail nicht entschlüsselt werden konnte und stellt keine zweite Anfrage. Bestätigt man die erste Anfrage und lehnt die zweite Anfrage ab, so kann Icedove die E-Mail entschlüsseln und gibt keine Fehlermeldung aus. Bestätigt man beide Anfragen, verhält sich Icedove genauso.

Mit Hilfe der Debugging-Funktionen von OpenSC wurde versucht, den zweiten Aufruf des Entschlüsselns zurückzuverfolgen. Es konnte dabei nicht festgestellt werden, dass der Entschlüsselungsvorgang zweimal vom Virtual Keycard-Treiber gestartet wird. Es konnte auch nicht festgestellt werden, dass eine Funktion aus OpenSC die Entschlüsselung ein zweites Mal startet. Es wird vielmehr vermutet, dass Icedove die Entschlüsselung zweimal startet. Diese Vermutung stützt sich darauf, dass die Funktion

```
CK_RV C_OpenSession(CK_SLOT_ID slotID, CK_FLAGS flags,
    CK_VOID_PTR pApplication, CK_NOTIFY Notify,
    CK_SESSION_HANDLE_PTR phSession)
```

aus der Datei „pkcs11-session.c“ im Ordner „pkcs11“ zweimal aufgerufen wird und das nicht innerhalb von OpenSC geschieht. Dies wurde überprüft, indem die Stellen in OpenSC markiert wurden, an denen die Funktion aufgerufen wird, und diese beim Durchlauf nicht angesteuert wurden.

8 Literaturverzeichnis

Literatur

[Con13] CardContact Software & System Consulting, *Script Collection*, Website, 2013, Online verfügbar auf: http://www.openscdp.org/scripts/musclecard/, abgerufen am 12.10.2014.

[DC01] Tommaso Cucinotta David Corcoran, *MUSCLE Cryptographic Card Edge Definition for Java Enabled Smartcards*, Website, 2001, Online verfügbar auf: http://pcsclite.alioth.debian.org/musclecard.com/musclecard/files /mcardprot-1.2.1.pdf, abgerufen am 12.10.2014.

[Eck08] Claudia Eckert, *IT-Sicherheit*, De Gruyter, 2008.

[Eik14] Ronald Eikenberg, *Google: Verschlüsselung wird Standard bei Android*, 2014, Online verfügbar auf: http://heise.de/-2400147, abgerufen am 12.10.2014.

[fISB12a] Federal Office for Information Security (BSI), *Advanced Security Mechanisms for Machine Readable Travel Documents - Part 1 - eMRTDs with BAC/PACEv2 and EACv1*, Website, 2012, Online verfügbar auf: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications /TechGuidelines/TR03110/TR-03110_v2.1_P1pdf.pdf, abgerufen am 12.10.2014.

[fISB12b] Federal Office for Information Security (BSI), *Advanced Security Mechanisms for Machine Readable Travel Documents - Part 3 - Common Specifications*, Website, 2012, Online verfügbar auf: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications /TechGuidelines/TR03110/TR-03110_v2.1_P3pdf.pdf, abgerufen am 12.10.2014.

[fISB14] Federal Office for Information Security (BSI), *BSI: BSI TR-03110 Technical Guideline Advanced Security Mechanisms for Machine Readable Travel Documents*, Website, 2014, Online verfügbar auf: https://www.bsi.bund.de/EN/Publications/TechnicalGuidelines /TR03110/BSITR03110.html, abgerufen am 12.10.2014.

[fS13] International Organization for Standardization, *Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange*, 2013, Online (kostenpflichtig) verfügbar auf: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm ?csnumber=54550, abgerufen am 12.10.2014.

[Gmb14] Giesecke & Devrient GmbH, *Open Mobile API*, Website, 2014, Online verfügbar auf: https://code.google.com/p/seek-for-android/source/browse/trunk/src/smartcard-api/src/org/simalliance/openmobileapi/?r=649#openmobileapi%2Fservice%2Fsecurity, abgerufen am 12.10.2014.

[Goo14] Google, *Encrypt your data*, Website, 2014, Online verfügbar auf: https://support.google.com/nexus/answer/2844831?hl=en, abgerufen am 12.10.2014.

[Hou09] R. Housley, *Cryptographic Message Syntax (CMS)*, RFC 5652 (INTERNET STANDARD), September 2009, Online verfügbar auf: http://www.ietf.org/rfc/rfc5652.txt, abgerufen am 12.10.2014.

[Jan14] Mark Janssen, *Adding a new card driver*, Website, 2014, Bezieht sich auf Revision 4, Online verfügbar auf: https://github.com/OpenSC/OpenSC/wiki/Adding-a-new-card-driver, abgerufen am 12.10.2014.

[Lab04] RSA Laboratories, *PKCS #11 v2.20: Cryptographic Token Interface Standard*, Website, 2004, RSA Security Inc. Public-Key Cryptography Standards (PKCS), Siehe http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm, Online verfügbar auf: ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20.pdf, abgerufen am 12.10.2014.

[Lab06] RSA Laboratories, *PKCS #15 v1.1: Cryptographic Token Information Syntax Standard*, Website, 2006, RSA Security Inc. Public-Key Cryptography Standards (PKCS), Siehe http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-15-cryptographic-token-information-format.htm, Online verfügbar auf: ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-15/pkcs-15v1_1.pdf, abgerufen am 12.10.2014.

[Lab12] RSA Laboratories, *PKCS #1 v2.2: RSA Cryptography Standard*, Website, 2012, EMC Corporation Public - Key Cryptography Standards (PKCS), Online verfügbar auf: http://www.emc.com/collateral/white-papers/h11300-pkcs-1v2-2-rsa-cryptography-standard-wp.pdf, abgerufen am 12.10.2014.

[Lic14] Licel, *jCardSim - About*, Website, 2014, Online verfügbar auf: http://jcardsim.org/about, abgerufen am 12.10.2014.

[Mor14] Frank Morgner, *nPA Smart Card Library*, Website, 2014, Online verfügbar auf: http://frankmorgner.github.io/vsmartcard/npa/README.html, abgerufen am 12.10.2014.

[MP01a] MUSCLE-Projekt, *MUSCLE*, Website, 2001, Online verfügbar auf: http://pcsclite.alioth.debian.org/musclecard.com/info.html, abgerufen am 12.10.2014.

[MP01b] MUSCLE-Projekt, *MuscleCard applet*, Website, 2001, Online verfügbar auf: http://pcsclite.alioth.debian.org/musclecard.com/musclecard/index.html, abgerufen am 12.10.2014.

[Ope14] OpenSC, *OpenSC/OpenSC (Online-Repository)*, 2014, Online verfügbar auf: https://github.com/OpenSC/OpenSC, abgerufen am 14.10.2014.

[Pla14a] Steffen Platzer, *HU-CA Smartcard mit persönlichem Zertifikat*, Website, 2014, Bezieht sich auf die Version vom 15.03.2013, Verantwortlicher Prof. Dr. P. Schirmbacher, Online verfügbar auf: https://www.cms.hu-berlin.de/dl/zertifizierung/HU-CA-Smartcard, angerufen am 12.10.2014.

[Pla14b] Steffen Platzer, *persönliches HU-CA Softzertifikat*, Website, 2014, Bezieht sich auf die Version vom 05.08.2014, Verantwortlicher Prof. Dr. P. Schirmbacher, Online verfügbar auf: https://www.cms.hu-berlin.de/dl/zertifizierung/softzertifikat/index/, abgerufen am 12.10.2014.

[Pla14c] Steffen Platzer, *PKI-Services / Public Key Infrastructure*, Website, 2014, Bezieht sich auf die Version vom 15.09.2014, Verantwortlicher Prof. Dr. P. Schirmbacher, Online verfügbar auf: https://www.cms.hu-berlin.de/dl/zertifizierung, abgerufen am 12.10.2014.

[Pro14a] Android Open Source Project, *Getting a Result from an Activity*, Website, 2014, Online verfügbar auf: http://developer.android.com/training/basics/intents/result.html, abgerufen am 12.10.2014.

[Pro14b] Android Open Source Project, *Host-based Card Emulation*, Website, 2014, Online verfügbar auf: http://developer.android.com/guide/topics/connectivity/nfc/hce.html, abgerufen am 12.10.2014.

[Pro14c] Android Open Source Project, *Intent*, Website, 2014, Online verfügbar auf: http://developer.android.com/reference/android/content/Intent.html, abgerufen am 12.10.2014.

[Pro14d] Android Open Source Project, *<intent-filter>*, Website, 2014, Online verfügbar auf: http://developer.android.com/guide/topics/manifest/intent-filter-element.html, abgerufen am 12.10.2014.

[Pro14e] Android Open Source Project, *Issue 76598 - android - NFC (IsoDep): maximum transceive length is hard-coded (261 Bytes), although some NFC Controllers are able to send more than 2000 Bytes - Android Open Source Project - Issue Tracker - Google Project Hosting*, 2014, Online verfügbar auf: https://code.google.com/p/android/issues/detail?id=76598, abgerufen am 14.10.2014.

[Pro14f] Android Open Source Project, *<service>*, Website, 2014, Online verfügbar auf: http://developer.android.com/guide/topics/manifest/service-element.html, abgerufen am 12.10.2014.

[Pro14g] Android Open Source Project, *Storage Options*, Website, 2014, Online verfügbar auf: http://developer.android.com/guide/topics/data/data-storage.html#filesInternal, abgerufen am 12.10.2014.

[Ric14] Ole Richter, *Prüfung von öffentlichen eID-Terminals mit einem Android-Smartphone*, Humboldt-Universität zu Berlin, 2014, Bachelorarbeit.

[RT10] B. Ramsdell and S. Turner, *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification*, RFC 5751 (Proposed Standard), January 2010, Online verfügbar auf: http://www.ietf.org/rfc/rfc5751.txt, abgerufen am 14.10.2014.

[Sta14] Stackexchange, *cryptography - Why should one not use the same asymmetric key for encryption as they do for signing? - Information Security Stack Exchange*, 2014, Online verfügbar auf: http://security.stackexchange.com/questions/1806/why-should-one-not-use-the-same-asymmetric-key-for-encryption-as-they-do-for-sig, abgerufen am 12.10.2014.

[Sti06] Douglas R. Stinson, *Cryptography: Theory and praxis*, vol. 3, Chapman & Hall/CRC, 2006.

[ts14] t senger, *androsmex - a PACE implementation for android smartphones with NFC capabilities*, 2014, Online verfügbar auf: https://code.google.com/p/androsmex/, abgerufen am 12.10.2014.

[uDO14] Frank Morgner und Dominik Oepen, *Welcome to OpenPACE's documentation!*, Website, 2014, Online verfügbar auf: https://frankmorgner.github.io/openpace/, abgerufen am 12.10.2014.

[vik12] viktorTarasov, *OpenSC/Overview*, Website, 2012, Bezieht sich auf Revision 3, Online verfügbar auf: https://github.com/OpenSC/OpenSC/wiki/Overview, abgerufen am 12.10.2014.

[WG310] ISO/IEC JTC1 SC17 WG3/TF5, *Supplemental Access Control for Machine Readable Travel Documents*, Website, 2010, Published by authority of the Secretary General, Version 1.01, Online verfügbar auf: http://www.icao.int/security/mrtd/downloads/technical%20reports /technical%20report.pdf, abgerufen am 12.10.2014.

[WR95] Wolfgang Effing Wolfgang Rankl, *Handbuch der Chipkarten*, vol. 1, Carl Hanser Verlag GmbH & Co. KG, 1995, Online verfügbar auf: http://files.hanser.de/hanser/docs/20040930_249315752-74_HdC1.pdf, abgerufen am 12.10.2014.

9 Anhang

9.1 Befehle zum Kompilieren und Linken der Treiberdateien

```
# Treiber "virtualkeycard" bauen:
cd $PREFIX/vsmartcard/npa/src/opensc/src/libopensc/
export C_INCLUDE_PATH=$PREFIX/include
gcc -DHAVE_CONFIG_H -I. -I../.. -DOPENSC_CONF_PATH=\"$PREFIX/etc/
    opensc//opensc.conf\" -I../../src -I$PREFIX/include -pthread -
    I/usr/include/PCSC -fno-strict-aliasing -g -O2 -Wall -Wextra -
    Wno-unused-parameter -Werror=declaration-after-statement -MT
    card-virtualkeycard.lo -MD -MP -MF .deps/card-virtualkeycard.
    Tpo -c card-virtualkeycard.c  -fPIC -DPIC -o .libs/card-
    virtualkeycard.o
gcc -shared -fPIC -DPIC -o .libs/card-virtualkeycard.so .libs/
    muscle.o .libs/muscle-filesystem.o .libs/card.o .libs/sc.o .
    libs/card-virtualkeycard.o -Wl,-rpath -Wl,$PREFIX/lib -L/home/
    erik/openpace-opensc-0.14/lib -lnpa -lcrypto -lopensc

# PKCS15-Treiber fuer virtualkeycard bauen:
cd $PREFIX/vsmartcard/npa/src/opensc/src/pkcs15init/
gcc -DHAVE_CONFIG_H -I. -I../.. -DSC_PKCS15_PROFILE_DIRECTORY=\"
    $PREFIX/share/opensc\" -I../../src -I$PREFIX/include -fno-
    strict-aliasing -g -O2 -Wall -Wextra -Wno-unused-parameter -
    Werror=declaration-after-statement -MT pkcs15-virtualkeycard.
    lo -MD -MP -MF .deps/pkcs15-virtualkeycard.Tpo -c pkcs15-
    virtualkeycard.c  -fPIC -DPIC -o .libs/pkcs15-virtualkeycard.o
gcc -shared -fPIC -DPIC -o .libs/pkcs15-virtualkeycard.so .libs/
    pkcs15-virtualkeycard.o .libs/profile.o ../common/
    compat_strlcpy.o
```

9.2 Änderungen an der OpenSC-Konfigurationsdatei

```
app default {
        reader_driver pcsc {
                max_send_size = 200;
        }

        card_drivers = virtualkeycard, internal;

        card_driver virtualkeycard {
                # The location of the driver library
                module = /home/erik/openpace/vsmartcard/npa/src/opensc/src/
                    libopensc/.libs/card-virtualkeycard.so;
        }

        card_atr 3b:80:80:01:01 {
                name = "VirtualKeycard";
                driver = "virtualkeycard";
        }

        framework pkcs15 {

                emulate virtualkeycard {
                        # The location of the driver library
                        module = /home/erik/openpace/vsmartcard/npa/src/opensc/
                            src/pkcs15init/.libs/pkcs15-virtualkeycard.so;
```

```
                }

                pkcs15init virtualkeycard {
                        # The location of the driver library
                        module = /home/erik/openpace/vsmartcard/npa/src/opensc/
                            src/pkcs15init/.libs/pkcs15-virtualkeycard.so;
                }
        }
}

app opensc-pkcs11 {
        pkcs11 {
                max_virtual_slots = 32;

                slots_per_card = 2;
        }
}
```

9.3 Ausgabe eines CSR durch OpenSSL

```
Certificate Request:
    Data:
        Version: 0 (0x0)
        Subject: C=DE, ST=Berlin, L=Berlin, O=Internet Widgits Pty Ltd, CN=Hannes
             Schultz/emailAddress=hannes-76@ok.de
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:ac:21:db:52:84:5f:dc:51:9d:8e:f0:ba:0b:d3:
                    43:17:6d:f9:cd:d6:b5:f5:bb:b9:39:28:21:fe:74:
                    6c:7a:20:92:90:cd:60:97:00:d0:7e:32:7e:a7:8b:
                    40:c6:c5:b7:f8:16:f4:a1:ab:52:e3:63:c0:1a:39:
                    28:5d:95:79:06:2a:62:e6:44:b9:0f:31:5d:8f:12:
                    b1:a9:3e:62:bd:91:b3:2a:30:94:03:97:26:e4:fb:
                    4c:86:ea:dc:d4:f6:03:d9:ef:26:3c:cb:85:97:97:
                    be:86:d2:2d:77:48:1c:f5:b3:06:be:39:c7:be:1b:
                    a0:3f:f6:c0:a5:4d:a9:92:e2:d8:3d:cf:db:5f:cd:
                    b2:e0:c7:8a:16:57:c3:ee:44:40:bd:76:7b:78:f5:
                    1b:55:8b:1f:d0:70:4d:8c:e9:6d:53:a2:c9:99:70:
                    aa:83:d3:06:cb:4a:5b:59:c6:2e:63:1a:8e:ad:47:
                    0b:c1:dd:7e:c7:b1:84:78:57:02:d9:5a:53:d1:d1:
                    84:c1:2f:c0:fe:f9:d9:6b:04:5c:cc:ae:09:8c:c8:
                    59:2a:f4:99:73:90:9b:98:02:b5:46:b6:74:00:5b:
                    ee:5d:c1:aa:77:a2:de:28:a3:a6:08:87:82:2c:78:
                    f4:b5:8e:00:f9:81:fc:f4:8b:52:12:0f:df:72:ff:
                    7d:c3
                Exponent: 65537 (0x10001)
        Attributes:
            a0:00
    Signature Algorithm: sha256WithRSAEncryption
         aa:4c:7f:03:24:3a:51:3c:0d:3e:63:1c:63:3f:8e:2c:4e:0c:
         ee:a1:8e:87:c1:41:1a:ae:7a:7f:54:b6:fa:67:86:6d:5c:a1:
         e5:25:4c:06:cd:ca:e2:bf:45:ba:75:1a:4b:c1:7d:ad:cf:c3:
         f4:0d:71:b2:ad:c8:d0:a1:11:fd:dd:82:fa:e6:be:66:8b:00:
         e8:01:fc:c7:37:23:81:2d:01:cd:24:08:3d:da:c8:7e:35:24:
         86:a0:07:4a:93:64:fe:97:77:db:ad:fb:b0:64:95:12:fd:94:
         7e:00:93:a8:f3:6c:0f:20:cf:c6:6d:6b:68:03:06:07:a3:44:
         0f:a2:cd:01:89:34:ec:0e:74:0e:7f:ac:10:90:28:74:26:21:
         6a:1b:2e:5e:f6:e5:60:89:3d:4a:9a:6b:c9:e9:93:95:f8:3d:
         44:84:47:c9:0c:21:1e:f2:21:e5:c0:fc:ef:c2:2e:ff:53:45:
         20:42:d1:d8:ce:f6:73:eb:b6:0d:45:b8:01:a5:37:21:0c:89:
         a8:5c:26:9b:c8:13:da:e6:be:13:dd:d1:ee:63:e5:56:58:81:
         b1:e1:c4:a8:a8:3b:a0:ac:c2:7d:53:5d:48:1c:e2:9b:c5:e9:
         86:b8:7d:ac:09:15:8d:c1:89:69:ff:5f:e5:c3:c1:78:79:5b:
         8f:79:cd:ed
```

```
-----BEGIN CERTIFICATE REQUEST-----
MIIC0TCCAbkCAQAwgYsxCzAJBgNVBAYTAkRFMQ8wDQYDVQQIDAZCZXJsaW4xDzAN
BgNVBAcMBkJlcmxpbjEhMB8GA1UECgwYSW50ZXJuZXQgV2lkZ2l0cyBQdHkgTHRk
MRcwFQYDVQQDDA5IYW5uZXMgU2NodWx0ejEeMBwGCSqGSIb3DQEJARYPaGFubmVz
LTc2QG9rLmRlMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEArCHbUoRf
3FGdjvC6C9NDF235zda19bu5OSgh/nRseiCSkM1glwDQfjJ+p4tAxsW3+Bb0oatS
42PAGjkoXZV5Bipi5kS5DzFdjxKxqT5ivZGzKjCUA5cm5PtMhurc1PYD2e8mPMuF
l5e+htItd0gc9bMGvjnHvhugP/bApU2pkuLYPc/bX82y4MeKFlfD7kRAvXZ7ePUb
VYsf0HBNjOltU6LJmXCqg9MGy0pbWcYuYxqOrUcLwd1+x7GEeFcC2VpT0dGEwS/A
/vnZawRczK4JjMhZKvSZc5CbmAK1RrZ0AFvuXcGqd6LeKKOmCIeCLHj0tY4A+YH8
9ItSEg/fcv99wwIDAQABoAAwDQYJKoZIhvcNAQELBQADggEBAKpMfwMkOlE8DT5j
HGM/jixODO6hjofBQRquen9Utvpnhm1coeUlTAbNyuK/Rbp1GkvBfa3Pw/QNcbKt
yNChEf3dgvrmvmaLAOgB/Mc3I4EtAc0kCD3ayH41JIagB0qTZP6Xd9ut+7BklRL9
lH4Ak6jzbA8gz8Zta2gDBgejRA+izQGJNOwOdA5/rBCQKHQmIWobLl725WCJPUqa
a8npk5X4PUSER8kMIR7yIeXA/O/CLv9TRSBC0djO9nPrtg1FuAGlNyEMiahcJpvI
E9rmvhPd0e5j5VZYgbHhxKioO6Cswn1TXUgc4pvF6Ya4fawJFY3BiWn/X+XDwXh5
W495ze0=
-----END CERTIFICATE REQUEST-----
```

9.4 OpenSC-Profil des MuscleCard-Applets / von Virtual Keycard

```
#
# PKCS15 r/w profile for MuscleCards
#

cardinfo {
    label       = "MUSCLE";
    manufacturer    = "Identity Alliance";

    max-pin-length      = 8;
    min-pin-length      = 4;
    pin-encoding        = ascii-numeric;

}
option default {
    macros {
        protected       = *=$PIN, READ=NONE;
        unprotected     = *=NONE;
        so-pin-flags    = local, initialized; #, soPin;
        so-min-pin-length = 4;
        so-pin-attempts = 2;
        so-auth-id      = 1;
        so-puk-attempts = 4;
        so-min-puk-length = 4;
        unusedspace-size = 128;
        odf-size        = 256;
        aodf-size       = 256;
        cdf-size        = 512;
        prkdf-size      = 256;
        pukdf-size      = 256;
        dodf-size       = 256;
    }
}

PIN so-pin {
    reference = 0;
        flags = local, initialized;
}
PIN so-puk {
    reference = 0;
}
PIN user-pin {
        reference = 1;
    attempts    = 3;
```

```
        flags   = local, initialized;
}
PIN user-puk {
        reference = 1;
    attempts    = 10;
}


filesystem {
    DF MF {
        path    = 3F00;
        type    = DF;
                acl = *=NONE, ERASE=$PIN;
        # This is the DIR file
        EF DIR {
            type        = EF;
            file-id     = 2F00;
            size        = 128;
            acl         = *=NONE;
        }

        # Here comes the application DF
        DF PKCS15-AppDF {
            type        = DF;
            file-id     = 5015;
            aid         = A0:00:00:00:63:50:4B:43:53:2D:31:35;
            acl         = *=$PIN;
            size        = 1; # NO DATA SHOULD BE STORED DIRECTLY HERE!

            EF PKCS15-ODF {
                file-id         = 5031;
                size            = $odf-size;
                ACL             = $unprotected;
            }

            EF PKCS15-TokenInfo {
                file-id         = 5032;
                ACL             = $unprotected;
                size = 128;
            }

            EF PKCS15-UnusedSpace {
                file-id         = 5033;
                size            = $unusedspace-size;
                ACL             = $unprotected;
            }

            EF PKCS15-AODF {
                file-id         = 4401;
                size            = $aodf-size;
                ACL             = $protected;
            }

            EF PKCS15-PrKDF {
                file-id         = 4402;
                size            = $prkdf-size;
                acl             = $protected;
            }

            EF PKCS15-PuKDF {
                file-id         = 4403;
                size            = $pukdf-size;
                acl             = $protected;
            }

            EF PKCS15-CDF {
```

```
            file-id         = 4404;
            size            = $cdf-size;
            acl             = $protected;
        }

        EF PKCS15-DODF {
            file-id         = 4405;
            size            = $dodf-size;
            ACL             = $protected;
        }
    template key-domain {
            BSO private-key {
                ACL = *=$PIN, READ=NEVER;
            }
            EF public-key {
                file-id     = 3000;
                structure   = transparent;
                ACL         = *=NEVER,
                                READ=NONE,
                                UPDATE=$PIN,
                                ERASE=$PIN;
            }

            # Certificate template
            EF certificate {
                file-id     = 3100;
                structure   = transparent;
                ACL         = *=NEVER,
                                READ=NONE,
                                UPDATE=$PIN,
                                ERASE=$PIN;
            }

            # Extractable private keys are stored in transparent EFs.
            # Encryption of the content is performed by libopensc.
            EF extractable-key {
                file-id     = 3200;
                structure   = transparent;
                ACL         = *=NEVER,
                                READ=$PIN,
                                UPDATE=$PIN,
                                ERASE=$PIN;
            }

            # data objects are stored in transparent EFs.
            EF data {
                file-id     = 3300;
                structure   = transparent;
                ACL         = *=NEVER,
                                READ=NONE,
                                UPDATE=$PIN,
                                ERASE=$PIN;
            }
            # private data objects are stored in transparent EFs.
            EF privdata {
                file-id     = 3400;
                structure   = transparent;
                ACL         = *=NEVER,
                                READ=$PIN,
                                UPDATE=$PIN,
                                ERASE=$PIN;
            }

        }
    }
}
```

```
}
```

Selbständigkeitserklärung

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

Berlin, den 16. Oktober 2014 .....