Eine offene Software-Architektur zur Ansteuerung moderner Smartcards

Philip Wendland · Michael Roßberg · Günter Schäfer

Technische Universität Ilmenau
Fachgebiet Telematik/Rechnernetze
{philip.wendland | michael.rossberg | guenter.schaefer}@tu-ilmenau.de

Zusammenfassung

Schwachstellen in kryptografischer Software haben oft gravierende Auswirkungen auf die Sicherheit von IT-Systemen. Die Verwendung vertrauenswürdiger Smartcards wäre eine Alternative. Allerdings ist man beim Einsatz moderner Smartcards häufig auf proprietäre Client-Software angewiesen, die operative Nachteile mit sich bringt: etwa eine fehlende Unterstützung von Systemarchitekturen und Betriebssystemen, eine komplizierte Isolation des Codes im Hostsystem und schlechte Adaptionsmöglichkeiten. Offene Lösungen wie das OpenSC Framework besitzen diese Nachteile nicht, unterstützen jedoch moderne Smartcards nur in Einzelfällen, weil Kartentreiber häufig erst durch Reverse Engineering entstehen müssen. Da die Funktionalität der Smartcards ohnehin oft durch sogenannte *Applets* der Java-Card-Technologie erbracht wird, stellt dieser Artikel eine alternative Architektur vor, bei dem ein offenes Java-Card-Applet mit korrespondierenden Komponenten zur Integration in bestehende Open Source Umgebungen konzipiert wurde. Somit entsteht ein leichter zu sicherndes, flexibles System zur Verwendung von Java-Cards.

1 Einsatz von Smartcards

Im Laufe der letzten Jahre wurde eine ganze Reihe von Sicherheitslücken aufgedeckt, welche die kryptografischen Funktionen in IT-Systemen betroffen haben und somit zur vollständigen Umgehung von Sicherheitsfunktionen genutzt werden konnten. Ein besonders eklatantes Beispiel ist etwa der *Heartbleed* getaufte Programmfehler von OpenSSL, der dazu genutzt werden konnte, Speicherbereiche und damit auch private Schlüssel auszulesen [Code14]. Mit der wachsenden Komplexität heutiger Software wächst auch die Wahrscheinlichkeit, dass diese folgenschwere Fehler beinhaltet. Ein Lösungsansatz für dieses Problem ist, sicherheitskritische Operationen auf spezialisierte Hardware-Sicherheitsmodule, wie zum Beispiel Smartcards, auszulagern. Diese erschweren konstruktiv das Auslesen privater Schlüssel [RaEf08], ihre Handhabung ist zurzeit jedoch relativ aufwändig.

Dies liegt vor allem daran, dass viele moderne Smartcards es zwar ermöglichen, die eigentliche Funktionalität durch sogenannte *Applets* der Java Card Technologie zu erbringen. Jedoch werden typischerweise von den Kartenherstellern proprietäre Applets verwendet, was mit gewissen Nachteilen verbunden ist: Man ist etwa ebenfalls an proprietäre Client-Software gebunden, die nur für einige Plattformen und Betriebssysteme verfügbar ist. Außerdem ist sie oft mit nicht unerheblichen Lizenzkosten verbunden und kann nur schwer durch unabhängige Dritte verifiziert werden. Applets und Middleware sind darüber hinaus nicht einfach an spezielle Anforderungen anzupassen. Weiterhin ist die Unterstützung neuer Funktionen oftmals an die Veröffentlichung

P. Schartner · K. Lemke-Rust · M. Ullmann (Hrsg.) · D•A•CH Security 2015 · syssec (2015) 267-278.

einer neuen Hardware-Generation gebunden, sodass sich für den Nutzer Kosten oder längere Wartezeiten ergeben.

Im Gegensatz dazu hat sich OpenSC [Ope15b] als offene Middleware für die Verwendung von Smartcards etabliert. OpenSC stellt Bibliotheken und Werkzeuge für verschiedene Smartcards zur Verfügung. Um die Unterstützung durch OpenSC gewährleisten zu können, müssen für unterschiedliche Smartcards beziehungsweise proprietäre Java Card Applets jedoch spezielle Kartentreiber existieren, die für viele moderne Karten erst noch (etwa durch Reverse Engineering) entstehen müssten. Der in diesem Artikel vorgestellte Ansatz unterscheidet sich grundlegend von dieser Methode. Durch die Neuentwicklung eines offenen Java Card Applets mit einem korrespondierenden OpenSC-Treiber kann ganz auf proprietäre Komponenten verzichtet werden. Die Vielzahl unterstützter Smartcards, auf denen das Applet installiert werden kann, kann so in unterschiedlichsten Szenarien, wie SSH, OpenSSL oder VPN mit OpenSC als Provider für asymmetrische Kryptografie eingesetzt werden.

Im Folgenden werden zunächst relevante Technologien und Standards vorgestellt, Anforderungen an ein Smartcard-Interface abgeleitet, sowie bestehende Open-Source-Lösungen präsentiert und bewertet. Anschließend wird die Architektur des neu konzipierten und implementierten Systems dargestellt und die geschaffene Lösung evaluiert.

2 Relevante Technologien und Standards

Im Rahmen dieses Artikels sind einige Standards, Spezifikationen und Technologien relevant. Abbildung 1 veranschaulicht, wie diese zusammenspielen können.

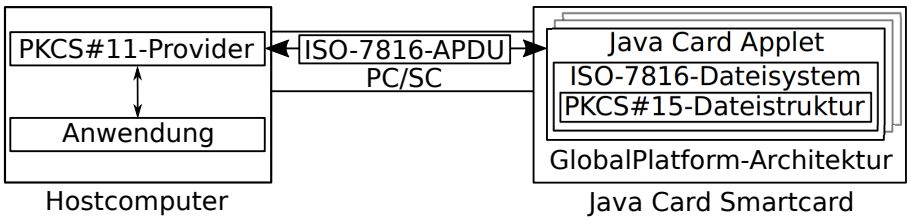


Abb. 1: Mögliches Zusammenspiel der relevanten Standards und Technologien

Die **GlobalPlatform Card Spezifikation** beschreibt unterschiedliche Merkmale konformer Smartcards, etwa eine Hardware-unabhängige Laufzeitumgebung (z.B. Java Card) oder die *CardManager*-Komponente, die für die Installation beziehungsweise die Verwaltung der Applets verantwortlich ist. Bei der Installation eines Applets wird die Kommunikation über das *Secure Channel Protocol* abgesichert [Glob11]. Des Weiteren werden die Zustände der Smartcard und Applets spezifiziert. So kann sich die Smartcard unwiderruflich sperren, falls die Authentifizierung auffällig oft fehlschlägt, beispielsweise weil versucht wird, ein schädliches Applet nachzuladen.

Java Card ist eine Variante der Java-Programmiersprache, die dafür eingesetzt werden kann, Applets für kompatible Smartcards zu erstellen. Bis 2007 wurden circa 3,5 Milliarden Java-Card-fähige Karten verkauft [RaEf08, S. 544]. Der große Vorteil von Lösungen wie Java Card ist, dass die Funktion der Smartcard von der hardwaretechnischen Realisierung weitgehend entkoppelt ist. Java Card setzt auf der GlobalPlatform-Spezifikation auf und ermöglicht den Zugriff auf bestimmte Funktionen der Smartcard über die Java Card API.

ISO-7816 [iso06] standardisiert in mehreren Teilen wesentliche Merkmale und Schnittstellen von Chipkarten. Die Kommunikation mit dem Hostrechner findet über sogenannte *Application Protocol Data Units* (APDUs) statt. Interessant für die Implementierung eines PKI-Applets ist hierbei die standardisierte APDU-Schnittstelle und das ISO-7816-Dateisystem. Das Dateisystem besteht aus unterschiedlichen Dateitypen, die sich grundlegend in *Elementary Files* (Daten) und *Dedicated Files* (Ordner) unterteilen lassen. Elementary Files werden weiterhin in verschiedene Untertypen unterteilt, u.a. mit transparentem Inhalt, einer weiteren Strukturierung der in der Datei enthaltenen Daten in *Records* fester oder variabler Länge oder sogar als Ringpuffer. Des Weiteren wird das *Master File* als Vaterknoten des Dateibaums standardisiert.

PKCS#15 (Public Key Cryptography Standards) spezifiziert eine Datei- und Ordnerstruktur für das Speichern von sicherheitsrelevanten Daten in einem ISO-7816-Dateisystem auf kryptografischen Token. Das Ziel dieses Systems ist eine größere Interoperabilität von Hard- und Software [PKC00]. Die Dateistruktur wird dynamisch in einem Dedicated File namens *DF(PKCS#15)* abgelegt.

PKCS#11 [PKC09], die auch *Cryptoki* genannte Anwendungsschnittstelle, abstrahiert von technischen Einzelheiten unterschiedlicher kryptographischer Token. Diese Schnittstelle wird den Hostanwendungen durch beispielsweise OpenSC [Ope15b] präsentiert, um kryptografische Operationen auf verschiedenen Smartcards durchführen zu können.

3 Anforderungen an das Smartcard-Interface

An ein offenes und flexibles Smartcard-Interface werden bestimmte Anforderungen gestellt, die im Folgenden dargestellt werden.

Sicherheit: Bereits die Smartcard-Initialisierung ist sicherheitskritisch und muss ohne proprietäre Werkzeuge zu erreichen sein. Dies umfasst die Applet-Installation und Individualisierung durch eine Benutzer-PIN, Zugriffskontrollparameter etc. Durch das *Generieren privater Schlüssel* direkt auf der Smartcard kann verhindert werden, dass dieser jemals ausgelesen wird. In manchen Anwendungsfällen muss man sich allerdings gegen eine Zerstörung der Smartcard absichern. Da jedoch das Auslesen privater Schlüssel aus der Smartcard nie möglich sein soll, kann dies nur durch *Importieren privater Schlüssel* gewährleistet werden, das als Option ebenfalls mit unterstützt werden sollte. Das Applet sollte außerdem verhindern, dass angreifbare kryptografische Algorithmen eingesetzt werden können, die unter Umständen zum Auslesen des privaten Schlüssels führen können.

Erweiterbare Softwarearchitektur: Ein wichtiger Vorteil von offener Software ist die einfache Erweiterung und Anpassung an neue Anforderungen. Als Grundvoraussetzung muss die Softwarearchitektur eine solche Flexibilität erlauben, beispielsweise indem Methoden aus der Objektorientierung zum Einsatz kommen. Weiterhin wird dies durch eine Smartcard-seitige standardkonforme Implementierung erreicht, sodass eine aufwändige Emulation von Dateisystemen oder PKCS#15-Dateistrukturen auf Hostseite, wie bei einigen Applets und Smartcards gängig, entfallen kann.

Bearbeitungszeiten: Die Leistungsfähigkeit von Smartcards ist aufgrund von Platz- und Kostenbeschränkungen sehr begrenzt. Gleichzeitig sind die bei asymmetrischer Kryptografie benötigten Operationen sehr aufwendig. Für die Akzeptanz der Benutzer sind akzeptable Bearbeitungszeiten für Dateioperationen und Signaturoperationen wichtig.

Speichermanagement: Im Gegensatz zur herkömmlichen Java Virtual Machines (VM) besitzt

die Java Card VM nur in Einzelfällen einen *Garbage Collector*, der nicht mehr referenzierte Objekte im Heap wieder freigibt. Gleichzeitig verbleiben in Java-Card-Applets „verlorene“ Objekte für immer im persistenten Speicher [Chen00, S. 38]. Viele der modernen Smartcards unterstützen eine Methode, um nicht mehr referenzierten Speicher wieder freizugeben (`requestObjectDeletion()`). Allerdings zeigten Messungen eine sehr hohe Laufzeit von bis zu drei Sekunden. Dies und der sehr begrenzte Speicherplatz der Smartcards setzen eine Implementierung voraus, die Speicher sowohl bedarfsgerecht alloziert, als auch nur in besonderen Anwendungsfällen wieder freigeben muss. Eine bedarfsgerechte Allokation erleichtert die Verwendung verschiedener Applets auf einer Smartcard, da so vermieden wird, dass unnötig viel Speicher reserviert wird, der unter Umständen nie benutzt wird. Ein Smartcard-Applet sollte unter Berücksichtigung dieser Anforderungen Speichermöglichkeiten, zum Beispiel für Zertifikate, bereitstellen.

Portabilität: Ziel der Lösung ist es, das Applet mit möglichst vielen verschiedenen Smartcards einsetzen zu können. Durch das Voraussetzen bestimmter Funktionen der Smartcard, z.B. das Vorhandensein eines Garbage Collectors, oder einer zu neuen und damit wenig verbreiteten Java-Card-Version könnte die Menge der unterstützten Smartcards eingeschränkt werden.

4 Bestehende Java Card Applets

Es konnten drei verschiedene offene Java Card Applets für den Einsatz in PKI-Systemen identifiziert werden.

Das **M.U.S.C.L.E. Card Applet** aus dem Jahr 2001 war ein erstes von OpenSC unterstütztes Java-Card-Applet, welches aber nicht mehr gepflegt wird. Anstelle von akzeptierten Standards implementiert es zum Host eine eigene Schnittstelle namens *CardEdge* [CoCu01], weshalb eine umständliche Abbildung der ISO-7816-Befehle auf die entsprechenden CardEdge-Befehle in OpenSC beziehungsweise anderer Software nötig ist. Außerdem wird die Interpretierbarkeit der Kommunikation erschwert.
Das Dateisystem wird in einem fest allozierten Bytearray von maximal 32KB Größe gespeichert, welches zudem anfällig für Fragmentierung ist. Hierdurch wird ein großer Teil des begrenzten Speichers der Smartcard verschwendet. Durch die statische Konfiguration der Arraygröße kann eine Verringerung des Wertes wiederum zu Problemen führen: so kann das Dateisystem des Applets voll sein, obwohl die Smartcard selbst noch freien Speicher zu Verfügung hat. Auch die Dateistruktur ist nicht standardkonform, weshalb hostseitig eine schwer zu durchschauende Emulation einer PKCS#15-Dateistruktur erfolgen muss.
Werden bei RSA-Operationen keine Padding-Algorithmen vorausgesetzt, die eine bestimmte Struktur der Nachrichten voraussetzen, können zum Beispiel bei kleinen Exponenten, kleinen Nachrichtenräumen oder durch multiplikative Eigenschaften Angriffsvektoren entstehen [RaEf08, S. 288 f.]. Das M.U.S.C.L.E. Applet erzwingt den Einsatz von Padding-Algorithmen allerdings nicht. Stattdessen wird im OpenSC-Treiber für das Applet beispielsweise für Signaturoperationen das Padding in Software hinzugefügt und die Daten anschließend in einem *decrypt raw* Befehl an das M.U.S.C.L.E.-Applet gesendet, vermutlich um die unvollständige Implementierung der Signaturoperation des Applets zu umgehen. Das Applet unterstützt keine Elliptische-Kurven-Kryptographie.

Das **OpenPGP Applet** [Ope15a] besitzt keine Dateisystemabstraktion und speichert drei private RSA-Schlüssel für Signaturen, Entschlüsselung und Authentifizierung. Im OpenSC-Treiber wird teilweise undurchsichtig ein Dateisystem und eine PKCS#15-Dateistruktur „emuliert“,

wobei natürlich Flexibilität verloren geht. Auch die feste Beschränkung auf drei Schlüsselpaare ist eine unerwünschte Einschränkung. Der Treiber ist dafür ausgelegt, nur die *OpenPGP-Karte* und einen *CryptoStick v1.2 (OpenPGP v2.0)* zu erkennen. Daher können in Verbindung mit OpenSC keine beliebigen Java-Card-fähigen Smartcards verwendet werden.

Das **JavaCard PKI Applet** [Jav13] wird zwar nicht von OpenSC unterstützt, hat aber ebenfalls ISO-7816-Konformität als Ziel. Das implementierte Dateisystem speichert Dateien in drei verschiedenen Arrays: Ein Array, das die Dateiinhalte speichert, eines für Zugriffskontrollinformationen und eines für die Abbildung der Dateistruktur. Die Implementierung hat zwar keine so strikten Größenbeschränkungen wie das M.U.S.C.L.E.-Applet, sieht aber kein Löschen von Dateien vor. Außerdem wird die Dateistruktur zum Initialisierungszeitpunkt festgelegt, weshalb ein nachträgliches Hinzufügen von Schlüsseln oder Zertifikaten problematisch wäre. Außerdem werden objektorientierte Konzepte nicht ausreichend eingesetzt, sodass eine nachträgliche Implementierung der fehlenden Funktionalitäten zu Problemen führen kann, da beispielsweise die Integrität der verschiedenen Arrays beim Auflösen von Fragmentierung nur schwer gewahrt werden kann.

Aufgrund der dargestellten Defizite bestehender Lösungen wurde die Notwendigkeit abgeleitet, ein Applet neu zu konzipieren und implementieren.

5 Architektur & Implementierung

Das konzipierte und implementierte System gliedert sich grundlegend in ein offenes Applet und verschiedene OpenSC-Komponenten.

5.1 OpenSC-Komponenten

Die in OpenSC integrierten Komponenten lassen sich in einen *Kartentreiber*, einen Initialisierungstreiber und ein konfigurierbares PKCS#11-Profil unterteilen. Der Kartentreiber ist für Operationen wie das Signieren oder Entschlüsseln, die PIN-Verifikation oder verschiedene Dateisystemoperationen zuständig. Hierfür konnten aufgrund der Standardkompatibilität des Applets meist Funktionen des ISO-7816-Referenztreibers von OpenSC verwendet werden. Die Initialisierung des Applets wird in Verbindung mit dem Initialisierungstreiber ebenfalls durch Komponenten innerhalb des OpenSC-Frameworks (z.B. `pkcs15-init`) realisiert und kann durch ein PKCS#15-Profil konfiguriert werden. Abbildung 2 zeigt einen Überblick über das Zusammenspiel der Komponenten. OpenSC ordnet den Treiber jeder beliebigen Smartcard zu, auf der das Applet installiert ist und kann als PKCS#11-Provider für unterschiedliche Anwendungen verwendet werden.

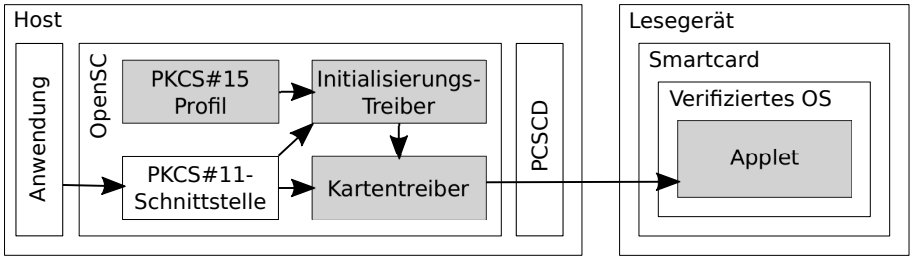


Abb. 2: Überblick über Zusammenspiel existierender und neuer Komponenten

5.2 Architektur des Java Card Applets

Das konzipierte Applet gliedert sich grundlegend in ein flexibles, objektorientiertes Dateisystem und eine Komponente zur Durchführung kryptografischer und operationeller Funktionen.

Dateisystem: Im Gegensatz zu herkömmlichen Dateisystemen, wie etwa im M.U.S.C.L.E. Applet, die Daten in einem zusammenhängenden Speicherblock ablegen, wird im entworfenen Java-Applet ein flexiblerer Weg gewählt: Zur Implementierung eines an den ISO-7816-Standard angelehnten Dateisystems wird eine Objekthierachie von persistenten Java-Objekten angelegt. Unterschiedliche Dateitypen werden hierbei durch unterschiedliche Ableitungen einer Oberklasse dargestellt. Die Klassenhierarchie ist in Abbildung 3 dargestellt.

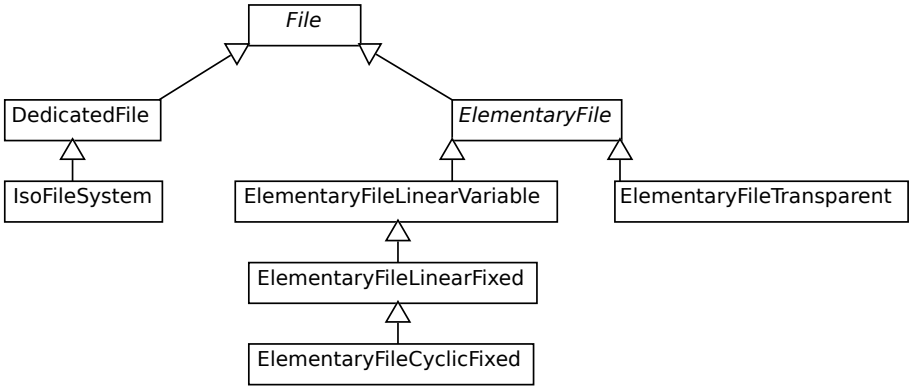


Abb. 3: Klassendiagramm des Dateisystems

Der gewählte Ansatz besitzt eine ganze Reihe von Vorteilen: Durch die Flexibilität der Objekthierarchie können zur Speicherung von Metainformationen nahezu beliebige (PKCS#15-) Dateistrukturen auf die Smartcard geschrieben werden. Dadurch bleibt der Java-Code kompakt und leicht verständlich. Da Dateien durch Objekte dargestellt werden, wird der Speicher bedarfsgerecht alloziert. Dies ist vor allem wichtig, wenn mehrere unterschiedliche Applets auf einer Smartcard parallel verwendet werden sollen. Eine mögliche Fragmentierung des Speichers wird durch die ohnehin vorhandene Speicherverwaltung der Smartcard aufgelöst.

APDU-Schnittstelle: Auch die APDU-Schnittstelle hält sich mit begründeten Ausnahmen, die sich aus dem beträchtlichem Alter und der Größe des Standards ergeben, an den ISO-7816-Standard. Ein besonderer Vorteil ergibt sich daraus, dass auch Smartcard-Lesegeräte mit Pinpads verwendet werden können. Diese idealerweise zertifizierten Lesegeräte senden die PIN in einem ISO-7816-standardisierten Befehl direkt an die Smartcard, ohne dass der Hostcomputer Zugriff auf die PIN zur Entsperrung des Applets erhält. Durch die Verwendung standardisierter Schnittstellen lassen sich die APDU-Befehle durch Dritte außerdem wesentlich leichter interpretieren.

Unterstützte kryptografische Algorithmen: Das Applet unterstützt sowohl RSA mit einer Schlüssellänge von 2048 Bit als auch ECDSA mit verschiedenen Schlüssellängen. Bei der Generierung von ECDSA-Schlüsselpaaren kann zwischen verschiedenen Kurven gewählt werden, da die Parameter vor dem Generieren an die Smartcard gesendet werden können. Aufgrund einer Limitierung der Java-Card-API in der Version 2.2.2 kann ECDSA nur mit SHA-1 eingesetzt werden [Jc206]. Sobald Smartcards mit der Version 3.0.4 erscheinen, können extern vorberechnete Hashwerte signiert werden [Jc311].

Zugriffskontrolle: Um unautorisiertes Verwenden der Smartcard und darin gespeicherter privater Schlüssel zu verhindern, wird die Authentifizierung mittels einer PIN vorausgesetzt. Durch eine begrenzte Anzahl an Fehlversuchen werden Brute-Force-Angriffe verhindert. Wird die PIN versehentlich gesperrt, kann diese durch einen deutlich längeren *Personal Unblock Key* (PUK) wieder entsperrt oder geändert werden. Dieses Verhalten ist etwa von SIM-Karten oder herkömmlichen Smartcards bekannt. Für verschiedene Dateien im Dateisystem des Applets können vor der Initialisierung unterschiedliche Zugriffsrechte mittels einer Profildatei in OpenSC konfiguriert werden. Private Schlüssel sind davon allerdings ausgenommen, da diese nie auslesbar und die Verwendung immer mit der PIN abgesichert wird.

6 Evaluierung

In diesem Abschnitt wird die geschaffene Lösung unter praktischen Gesichtspunkten evaluiert.

6.1 Erfüllung der qualitativen Anforderungen

Smartcard-Initialisierung: Für die Installation von Applets auf GlobalPlatform-konforme Smartcards können bestehende, quelloffene Werkzeuge verwendet werden, etwa GlobalPlatformPro [Glo15] oder GPShell [GPS15]. Diese Werkzeuge unterstützen auch die neue Version 3 des *Secure Channel Protokolls* der GlobalPlatform Card Spezifikation [Glob11].
Mittels des OpenSC-Werkzeugs `pkcs15-init` kann zunächst eine PKCS#15-Dateistruktur in das Dateisystem des Applets geschrieben werden. Hierbei wird auch die PIN und PUK spezifiziert. Anschließend können private Schlüssel direkt auf der Smartcard generiert werden, wobei der zugehörige öffentliche Schlüssel in das Dateisystem des Applets geschrieben wird. Das Importieren privater Schlüssel oder Zertifikate ist ebenfalls möglich.

Speichermanagement: Persistente Objekte, die im EEPROM gespeichert werden, werden ausschließlich zum Installationszeitpunkt des Applets oder beim Erstellen von Dateien oder Schlüsseln erstellt. Nur wenn Dateien explizit wieder gelöscht werden, wird die relativ aufwendige `requestObjectDeletion()`-Methode der Java Card API aufgerufen, um das im Speicher verbleibende Objekt freizugeben. Da das Löschen von Elementen durch den Nutzer nur in seltenen Fällen geschehen sollte, beispielsweise weil ein Zertifikat abgelaufen ist und entfernt werden soll, ist der Zeitverbrauch dieser Methode vertretbar. Durch die objektorientierte Architektur wird der Speicher bedarfsgerecht alloziert.

Portabilität: Als Java Card Version wurde 2.2.2 [Jc206] gewählt, da diese Version von den meisten frei erhältlichen Smartcards unterstützt wird. Relevante funktionale Erweiterungen wurden in der Version 3.0.4 [Jc311] hinzugefügt. Allerdings sind noch keine Smartcards verfügbar, die diese Version unterstützten, weshalb eine Erweiterung um diese Funktionen erst in Zukunft geschehen kann. Die Unterstützung durch bestimmte Smartcards könnte weiterhin dadurch eingeschränkt werden, optionale Klassen der Java Card API vorauszusetzen. Dies wurde, sofern möglich, vermieden: Einzig die Klasse `javacardx.crypto.cipher` für RSA-Operationen und das Interface `javacardx.apdu.extendedLength` werden vorausgesetzt, falls die Verwendung APDUs erweiterter Länge konfiguriert wurde.

6.2 Ausführungsgeschwindigkeiten

Für die praktische Beurteilung des Ansatzes wurden folgende aktuelle, frei verfügbare Smartcards verschiedener Leistungsklassen und unterschiedlichen Funktionsumfangs eingesetzt:

- J3A080 NXP JCOP 2.4.1R3
- J3A081 NXP JCOP 2.4.1R3
- MARX Cryptoken
- Giesecke & Devrient SmartCafe Expert 6.0 80K Dual (GD)

Für die praxisnahe Evaluierung des implementierten Dateisystems wurden Operationen auf einer PKCS#15-Dateistruktur mit einem auf der Smartcard generierten Schlüsselpaar vermessen. Die Dateistruktur ist in Abbildung 4 abgebildet und wurde durch ein OpenSC-Werkzeug mit folgendem Befehl auf der Smartcard erstellt:

```
pkcs15-init --create-pkcs15 --generate-key "rsa/2048" --auth-id FF
```

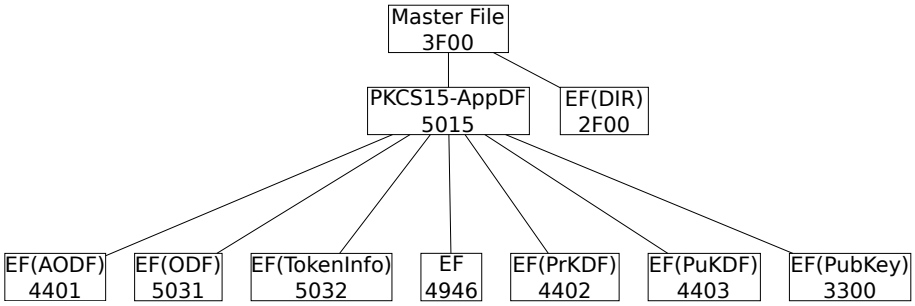
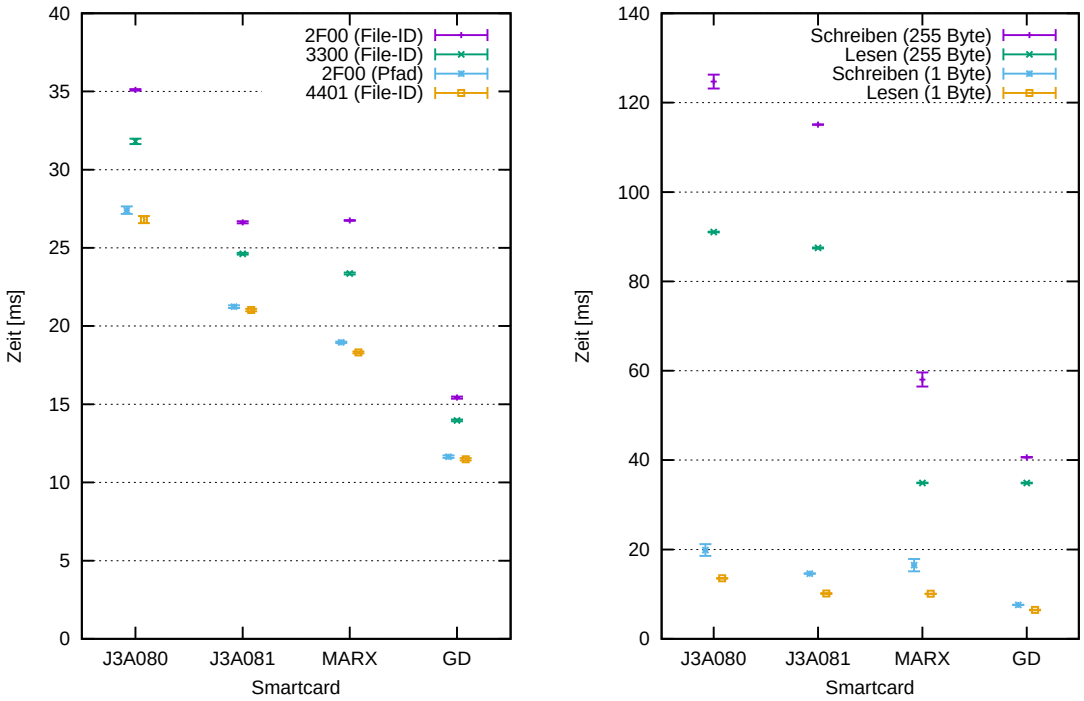


Abb. 4: PKCS#15-Dateistruktur für die Evaluierung



(a) Dateisuche (`SELECT FILE`) **(b)** Lesen und Schreiben

Abb. 5: Durchschnittliche Dauer von Dateisystemoperationen (mit 99% Konfidenz)

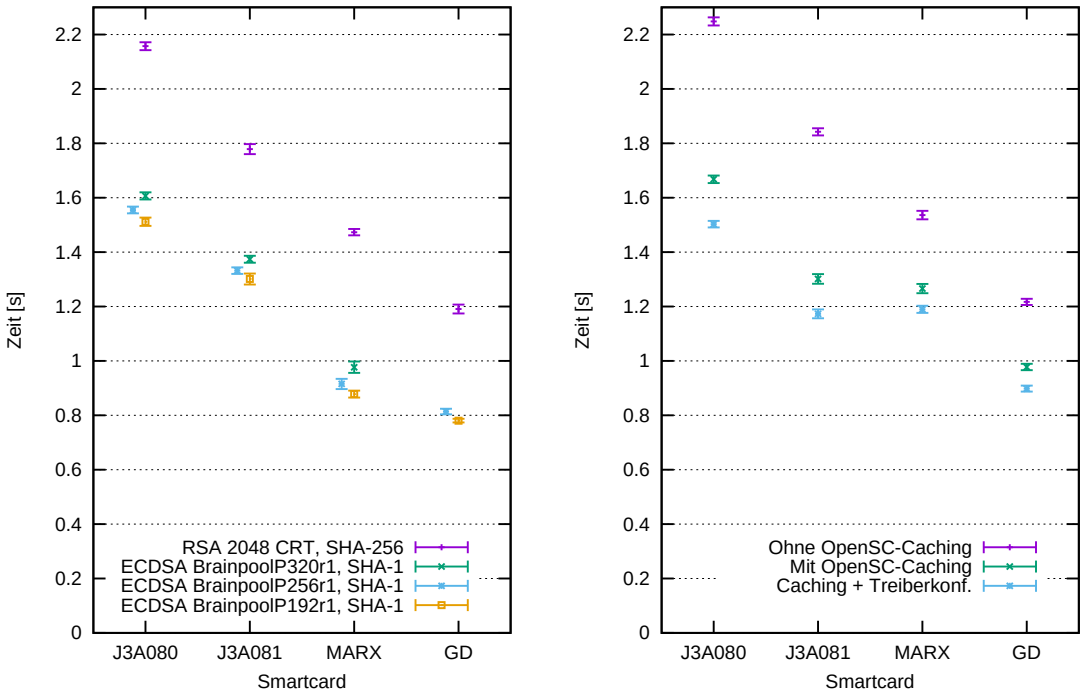
Abbildung 5a zeigt die für das Auswählen bestimmter Dateien benötigten Zeiten. Dateien werden im ISO-7816-Dateisystem mittels einer zwei Byte großen Identifikationsnummer (*File-ID*)

adressiert. Das *Auswählen* meint immer auch das *Suchen* der Datei, was als Tiefensuche implementiert wurde. Der Hostcomputer kann optional einen Dateipfad zur Dateisuche angeben. Das Applet nutzt diese Zusatzinformationen zur schnelleren Ausführung der Dateisuche. Dieser Geschwindigkeitsvorteil ist im Vergleich der Daten *2F00 (File-ID)* und *2F00 (Pfad)* ersichtlich. Durch die Zusatzinformation kann in diesem Fall bei der Suche der komplette Unterbaum unter der Datei mit der ID 5015 ignoriert werden.

Abbildung 5b zeigt die für das Lesen und Schreiben von Dateiinhalten benötigten Zeiten. Vor dem eigentlichen Lesen oder Schreiben müssen einige Befehle ausgeführt werden, etwa um die Kompatibilität mit der Dateistruktur oder die Zugriffsvoraussetzungen zu überprüfen. Um einen Eindruck für diesen statischen Aufwand zu erlangen, wurden zunächst Messungen mit nur einem Byte ausgeführt. Wie im Vergleich mit 255 Byte zu sehen ist, überwiegt die Zeit der eigentlichen Lese- oder Schreiboperation stark. Dies deutet darauf hin, dass die Geschwindigkeit des EEPROM-Speichers der Smartcards und/oder die Übertragungsbandbreite zum Host die Zeiten dominieren, nicht aber die Implementierung im Applet.

Das Erstellen oder Löschen einer Datei geschieht nicht im normalen Betrieb und daher werden an diese Operationen weniger strikte Zeitanforderungen gestellt. Das Erstellen einer Datei geht in etwa 81 (GD) bis 164 (J3A080) Millisekunden vonstatten. Das Löschen ist aufgrund der bereits beschriebenen Komplexität der `requestObjectDeletion()`-Methode mit circa 1.2 (GD) bis 2.6 (J3A080) Sekunden recht aufwändig. Auf die Geschwindigkeit dieser Methode kann kein Einfluss genommen werden. Dies gilt auch für das Generieren von Schlüsselpaaren. Hierbei variiert die Zeit sehr stark, wobei für 2048 Bit RSA Schlüssel Zeiten im Bereich von 3,7 Sekunden bis über 2 Minuten gemessen wurden.

Asymmetrische Kryptooperationen sind bekannt für ihre langen Laufzeiten auf Smartcards und damit ausschlaggebend für die Akzeptanz des Ansatzes. Daher wurde das erstmals verfügbare, einheitliche System verwendet, um vergleichende Messungen zwischen den Smartcards durchzuführen. Dazu wurden jeweils 128 Byte Daten durch OpenSC über die PKCS#11-Schnittstelle signiert. Zum einen wurde RSA mit 2048 Bit verwendet und zum anderen ECDSA unter Einsatz der vom BSI mitentwickelten Brainpool-Kurven über $\mathbb{Z}_p$ mit verschiedenen Schlüssellängen [LoMe10]. Die Länge der signierten Daten spielt hierbei nur indirekt eine Rolle: Bei RSA übermittelt der Hostcomputer nur den Hashwert an die Smartcard. Bei ECDSA muss die Smartcard aufgrund von Einschränkungen der Java-Card-API selbst den Hashwert berechnen. Allerdings existiert in OpenSC bezüglich des PKCS#11-Mechanismuses `CKM_ECDSA_SHA1` die Einschränkung, dass nur ein längenbeschränkter Puffer von 512 Bytes signiert werden kann. Die Messungen umfassen die Signaturerstellung sowie das Erfassen sämtlicher Metadaten aus dem Dateisystem des Applets und weitere Operationen, beispielsweise die Erkennung und Auswahl des Applets durch OpenSC oder die Verifikation der PIN. Wie aus Abbildung 6a ersichtlich ist, variiert die Geschwindigkeit um fast 100% je nach eingesetzter Smartcard. Ob diese Varianz von den Betriebssystemen oder der Hardware der Smartcards abhängt, ist nur schwer nachzuvollziehen, da die Geschwindigkeiten dieser Komponenten nicht unabhängig voneinander betrachtet werden können. Durch das offene Smartcard-Applet ist man zukünftig jedoch nicht auf eine bestimmte Hardware angewiesen und da insbesondere keine proprietären oder zusätzlichen Programmierschnittstellen verwendet werden, können sich ohne weiteren Portierungsaufwand durch neue Generationen von Smartcards weitere Geschwindigkeitsvorteile ergeben. Abbildung 6a zeigt ebenfalls, dass sich durch die Verwendung von ECDSA erhebliche Geschwindigkeitsvorteile ergeben, obwohl die Rohdaten an die Smartcard gesendet werden müssen. Auch die Schlüssellänge hat bei ECDSA insgesamt wenig Einfluss auf die Berech-



(a) Signaturerstellung **(b)** Entschlüsselung (RSA 2048 Bit)

Abb. 6: Durchschnittliche Dauer von Kryptooperationen (mit 99% Konfidenz)

nungszeiten. Mit dem Erscheinen modernerer Smartcards, die es ermöglichen, vorberechnete Hashwerte zu signieren, können in Zukunft mit ECDSA beliebige Hashfunktionen eingesetzt werden. Aufgrund einer Einschränkung der aktuellen Smartcard-Generation ist man derzeit auf den PKCS#11-Mechanismus `CKM_ECDSA_SHA1` [PKC09] beschränkt.

Für die Beurteilung der Dauer der Entschlüsselung auf den Smartcards wurden 128 Byte große Daten zunächst mit dem öffentlichen Schlüssel verschlüsselt und anschließend an die Smartcard zum Entschlüsseln gesendet. Die Größe der zu entschlüsselnden Daten ist auch hier praktisch nicht relevant, da die bei hybrider Verschlüsselung eingesetzten symmetrischen Schlüssel ohnehin auf die Blockgröße von RSA 2048 Bit aufgefüllt werden müssen. Die Entschlüsselungszeiten unterscheiden sich nicht signifikant von den für das Signieren benötigten Zeiten. In derselben Abbildung wurden exemplarisch ebenfalls die Geschwindigkeitsvorteile dargestellt, die sich durch den Einsatz von Caching-Mechanismen von OpenSC ergeben. Dieser Mechanismus speichert manche Dateien aus dem Dateisystem des Applets auf der Festplatte zwischen. Konkret wurden bei den durchgeführten Messungen drei ISO-7816 `SELECT` Befehle und fünf `READ BINARY` Befehle eingespart. Der größte Geschwindigkeitsvorteil ergibt sich bei dem Lesen der Dateien, das sich aus dem Lesen aus dem EEPROM-Speicher und dem Übertragen an den Hostcomputer zusammensetzt. Eine weitere Optimierungsmöglichkeit ist das statische Konfigurieren des von OpenSC zu verwendenden Kartentreibers. Dies spart fünf ISO-7816 `SELECT` Befehle ein, mit denen OpenSC vergeblich versucht, auf der Smartcard nicht vorhandene Applets auszuwählen. Die in diesem Abschnitt beschrieben Optimierungen lassen sich ebenfalls auf Signatur-Operationen übertragen.

6.3 Bedingungen für sicheren Einsatz

Koexistenz mit anderen Applets: Das Applet verwendet keine Schnittstellen, die eine Kommunikation mit anderen Applets auf der Smartcard ermöglichen. Allerdings sind Angriffe bekannt, die die Firewall zwischen den verschiedenen Java Card Applets auf der Smartcard durchbrechen können [HoMo09, MoPo08]. Um dieses Risiko zu minimieren, sollte darauf geachtet werden, die Standardschlüssel der Smartcard auszutauschen, sodass keine schädlichen Applets auf die Smartcard installiert werden können.

Wirksamkeit der PIN-Authentifizierung: Sobald sich der Benutzer mittels der PIN gegenüber der Smartcard authentifiziert hat, sind sicherheitskritische Funktionalitäten im Applet freigeschaltet. Aus diesem Grund wird die Smartcard durch OpenSC nach dem Abschluss einer Operation wieder zurückgesetzt. Des Weiteren besteht die Möglichkeit, dass OpenSC in einem exklusiven Modus mit der Smartcard kommuniziert, sodass andere Anwendungen die Smartcard nicht gleichzeitig ansprechen können. Nichtsdestotrotz kann ein Angreifer durch Abfangen der auf der Computertastatur eingegeben PIN oder der APDU-Kommunikation mit der Smartcard eventuell Kenntnis über die PIN erlangen. Dieses Risiko wird allerdings eliminiert, wenn zertifizierte Lesegeräte mit Pinpad eingesetzt werden. Der Hostcomputer sendet dann nur ein Template des ISO-7816-Verify-Befehls an das Lesegerät. Dieses fügt nach Benutzereingabe auf dem sicheren Pinpad die PIN in den Befehl ein und leitet diesen an die Smartcard weiter. Die standardkonforme Implementierung des Verify-Befehls im Applet ermöglicht die Verwendung solcher Lesegeräte. Tests mit einem zertifizierten Cherry ST-2000U Lesegerät bestätigten dies.

Manipulation der übertragenen Daten: Da an die Smartcard übertragene Daten (z.B. Hashwerte) im Klartext an die Smartcard gesendet werden, können diese von einem Angreifer prinzipiell vor dem Senden an die Smartcard verändert oder mitgelesen werden. Dies könnte durch eine Kompromittierung von OpenSC oder das Mitlesen der Kommunikationsdaten mit Root-Rechten erfolgen. Das Mitlesen der Kommunikationsdaten kann verhindert werden, indem die Kommunikationsdaten durch Protokolle kryptografisch abgesichert werden. Dies ermöglicht ebenfalls die sichere Verwendung mit kontaktlosen (NFC-fähigen) Smartcards. Für Smartcards existieren verschiedene solche Protokolle, etwa das *GlobalPlatform Secure Channel* Protokoll, das *ISO-7816 Secure Messaging* Protokoll [iso06] oder das *PACE* Protokoll, welches auch von dem neuen deutschen Personalausweis eingesetzt wird. Die Unterstützung eines der genannten Protokolle ist in Zukunft vorgesehen.

7 Schlussbetrachtungen

In diesem Artikel wurde eine offene Lösung für den Einsatz verschiedener moderner Smartcards präsentiert. Durch ein offenes, portables und flexibles Java-Card-Applet, das zusammen mit OpenSC als PKCS#11-Provider mit einer großen Anzahl an Programmen eingesetzt werden kann, wird eine ganze Reihe an Anwendungsfällen abgedeckt. Neue Anforderungen können durch die Erweiterungsmöglichkeiten leicht berücksichtigt werden. So ist etwa davon auszugehen, dass neue Generationen von Smartcards neuere Java-Card-Versionen unterstützen, und so neue kryptografische Funktionen einfach genutzt werden können. Außerdem können sich ohne Portierungsaufwand Geschwindigkeitsvorteile durch neue Smartcard-Generationen ergeben. Durch die offene Lösung ist ein Vergleich verschiedener Smartcards einfach möglich, da keine Einschränkungen durch Lizenzbeschränkungen oder Geheimhaltungsverträge entstehen. Da sämtliche Host-Komponenten in das OpenSC-Projekt eingeflossen sind, genügt es zur Verwendung des Applets eine aktuelle Version der Middleware zu installieren.

Literatur

[Chen00] Z. Chen: Java Card Technology for Smart Cards: Architecture and Programmer's Guide (The Java Series). Addison-Wesley (2000).

[CoCu01] D. Corcoran, T. Cucinotta: MUSCLE Cryptographic Card Edge Definition for Java Enabled Smartcards. Movement for the Use of Smartcards in a Linux Environment, version 1.2.1 Aufl. (2001).

[Code14] Codenomicon: The Heartbleed Bug. http://heartbleed.com/ (2014), aufgerufen am 18. März 2015.

[Glo15] GlobalPlatformPro (Werkzeug). https://github.com/martinpaljak/GlobalPlatformPro (2015), aufgerufen am 9. Mai 2015.

[Glob11] GlobalPlatform: Card Specification. 2.2.1 Aufl. (2011).

[GPS15] GlobalPlatform/GPShell (Werkzeug). http://sourceforge.net/p/globalplatform/wiki/GPShell/ (2015), aufgerufen am 9. Mai 2015.

[HoMo09] J. Hogenboom, W. Mostowski: Full memory read attack on a java card. *In: 4th Benelux Workshop on Information and System Security Proceedings (WISSEC'09)* (2009).

[iso06] ISO/IEC 7816 - Identification card - Integrated circuit cards. 3. edition Aufl. (2006).

[Jav13] JavaCard PKI Applet. http://sourceforge.net/projects/javacardsign/ (2013), aufgerufen am 24. Mai 2015.

[Jc206] Application Programming Interface - Java Card Platform, Version 2.2.2 (2006).

[Jc311] Java Card Classic Platform Specification 3.0.4 (2011).

[LoMe10] M. Lochter, J. Merkle: Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation (2010), RFC 5639, IETF, Status: Standard, https://tools.ietf.org/html/rfc5639.

[MoPo08] W. Mostowski, E. Poll: Malicious Code on Java Card Smartcards: Attacks and Countermeasures. *In: G. Grimaud, F.-X. Standaert (Hrsg.), Smart Card Research and Advanced Applications*, Springer Berlin Heidelberg, *Lecture Notes in Computer Science*, Bd. 5189 (2008), 1–16, .

[Ope15a] Java Card OpenPGP Card. http://sourceforge.net/projects/javacardopenpgp/ (2015), aufgerufen am 23. Mai 2015.

[Ope15b] OpenSC - Open source smart card tools and middleware. https://github.com/OpenSC/OpenSC (2015), aufgerufen am 23. März 2015.

[PKC00] PKCS#15 v1.1: Cryptographic Token Information Syntax Standard (2000).

[PKC09] PKCS#11 Base Functionality v2.30: Cryptoki - Draft 4 (2009).

[RaEf08] W. Rankl, W. Effing: Handbuch der Chipkarten. Carl Hanser Verlag München Wien (2008), 5., überarbeitete und aktualisierte Auflage.