

Entwurf und Implementierung von Applikationen für den FINREAD Kartenleser

Adrian Roth

Fachgebiet Theoretische Informatik - Kryptographie und
Computeralgebra
Fachbereich Informatik
Technische Universität Darmstadt
Prof. Dr. Johannes Buchmann

Betreuer: Dr. Evangelos Karatsiolis

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 18. Juni 2008

Zusammenfassung

Die sichere Kommunikation im Internet ist heutzutage ein sehr wichtiges Anliegen. Vor allem im Umfeld von Banken und Kreditinstituten ist es wichtig die höchst sensiblen Daten zu schützen. Im Allgemeinen bieten Public Key Infrastrukturen eine gute Basis dafür, die sichere Kommunikation im Internet zu ermöglichen. Im Vergleich zu der Authentifizierung mit statischen Passworten ist die Verifizierung von Zertifikaten eine weitaus sicherere Alternative. Smartcards haben sich als Aufbewahrungsort für die wichtigen kryptographischen Schlüssel und Zertifikate bewährt. Um den Einsatz von Smartcards sicherer zu machen wurde ein internationales Konsortium gebildet, das für die Entwicklung einer allgemeinen Spezifikation für programmierbare Kartenleser zuständig war: FINREAD Working Group (Financial Transactional IC Card Reader).

Die vorliegende Bachelorarbeit beschäftigt sich mit der FINREAD Spezifikation und dessen Möglichkeiten und Grenzen. Zu Beginn dieses Dokumentes wird ein Überblick darüber gegeben, wie mit dem Einsatz von Chipkarten die Sicherheit der digitalen Daten verbessert werden kann. Speziell wird auf die Sicherheit bei Online Geldtransaktionen eingegangen. In den weiteren Kapiteln wird die FINREAD Spezifikation vorgestellt. Es werden die Sicherheitskonzepte gezeigt, die der Spezifikation zu Grunde liegen sowie einige Grundlagen zur Programmierung von Applikationen für den FINREAD konformen Kartenleser. Ein weiteres Kapitel beschreibt in Detail wie man nach der Erstellung der Kartenleser Applikation (Finlet) diese auf den Kartenleser installiert und ausführt. Zum besseren Verständnis der Idee hinter der FINREAD Spezifikation ist es wichtig die Ausführung eines Finlets in Detail zu betrachten. Neben der Kartenleserapplikation wird noch eine so genannte Host Applikation benötigt. Dies ist die Applikation, die von einem PC aus den Kartenleser anspricht und z. B. die Ausführung eines Finlets startet. Zu dessen Programmierung beschreibt die FINREAD Spezifikation eine API. Der Hersteller des Kartenlesers implementiert diese API in einer Windows DLL. Im Rahmen dieser Bachelorarbeit wurde für diese DLL ein Java Wrapper programmiert, der in den weiteren Kapiteln dieses Dokumentes beschrieben wird. Zuletzt werden die Applikationen für den Kartenleser vorgestellt, die als Proof of Concept entwickelt wurden.

Kapitelübersicht

Kapitel 1: Einführung Hintergründe, Thema und Motivation dieser Arbeit.

Kapitel 2: Chipkarten und Sicherheit Sicherheitsprobleme im Online Zahlungsverkehr und wie der Einsatz von Chipkarten diese Probleme zum Teil beheben kann.

Kapitel 3: FINREAD - Eine Einführung Gibt eine Übersicht darüber was FINREAD bedeutet, welche Ziele man mit der Arbeit des FINREAD Konsortiums verfolgt und wie diese Ziele realisiert werden.

Kapitel 4: Entwicklung von Applikationen für den FINREAD Kartenleser Eine allgemeine Anleitung zur Entwicklung der Reader Applikationen: der Finlets (oder auch FCRA's).

Kapitel 5: Finlets in Betrieb In diesem Kapitel wird beschrieben wie die Finlets mit den vom Hersteller zur Verfügung gestellten Tools auf den Kartenleser installiert und ausgeführt werden. Des Weiteren wird die Interaktion der Komponenten bei der Ausführung eines Finlets gezeigt.

Kapitel 6: Java Wrapper Motivation, Entwurf, Implementierung und Benutzung des Java Wrappers für die `Finread.dll`.

Kapitel 7: Beispiel Applikationen Eine Übersicht über die Finlets, die als Proof-of-Concept entwickelt wurden.

Kapitel 8: Zusammenfassung und Ausblick Zusammenfassung der wichtigsten Punkte dieser Bachelorarbeit und Überlegungen zu möglichen Weiterentwicklungen auf diesem Gebiet

Inhaltsverzeichnis

1	Einführung	1
2	Chipkarten und Sicherheit	3
2.1	Authentifikation	3
2.2	Digitale Signaturen und MAC	4
2.3	Verschlüsselung	5
2.4	Zusammenfassung und Ausblick	5
3	Finread - Eine Einführung	9
3.1	Was ist FINREAD?	9
3.2	Allgemeine Anforderungen	9
3.2.1	Allgemeine Hardware Anforderungen	9
3.2.2	Interoperabilität	10
3.3	Programmierschnittstellen	11
3.3.1	Host API	11
3.3.2	Core Software	11
3.3.3	Reader API - Das STIP Framework	11
3.4	Sicherheitskonzepte	12
3.4.1	Signierte Software	12
3.4.2	Kryptographische Schlüssel	13
3.4.3	Operating Modes	14
3.5	Zusammenfassung	15
4	Entwicklung von Applikationen für den FINREAD Kartenleser	17
4.1	Grundlagen zu Finlets	17
4.2	STIP-Small Terminal Interoperability Platform	17
4.3	Finlets entwickeln	18
4.4	Konsequenzen aus dem Single Thread Paradigm	22
5	Finlets in Betrieb	23
5.1	Kompilieren des Codes	24
5.2	Finlet erzeugen	24
5.3	Administration Tool	25
5.4	Die Host Applikation	27
5.4.1	Finlet aktivieren	27
5.4.2	Ergebnisse vom Finlet entgegennehmen	28
5.5	Die Core Applikation	28

5.6	Kontrollfluss während der Ausführung	29
5.7	Debugging	30
6	Java Wrapper	31
6.1	Motivation	31
6.2	Plattform und Werkzeuge	31
6.3	JNI Grundlagen	32
6.3.1	Vorgehen bei der Benutzung von JNI	32
6.4	Die Funktionen der Host API	35
6.5	Design und Implementierung	38
6.5.1	Annahmen und Abhängigkeiten	38
6.5.2	Besonderheiten der Implementierung	40
6.5.3	Die Komponenten des Wrappers	41
6.6	Benutzung des Wrappers	43
6.6.1	FINREAD konforme API	43
6.6.2	Façade API	44
7	Beispiel Applikationen	45
7.1	Web Applikation - ATR im Browser anzeigen	46
7.2	GetCertFinlet	47
7.3	SignFinlet	51
8	Zusammenfassung und Ausblick	53
	Literaturverzeichnis	55
	Abbildungsverzeichnis	59

Kapitel 1

Einführung

Online Banking, Online Aktienhandel, Elektronische Steuererklärung oder E-Commerce: In den letzten Jahren hat der Austausch höchst sensibler Daten im Internet extrem zugenommen. Sowohl Privatpersonen als auch große Institutionen oder staatliche Instanzen nutzen die Bequemlichkeit und Schnelligkeit des Internets. Doch damit die transportierten Daten geschützt sind müssen einige Vorkehrungen getroffen werden.

Zu den Sicherheitsaspekten gehören die Verbindlichkeit von Aufträgen, die Authentizität der übertragenen Daten sowie die Identifikation der Kommunikationspartner und der vertrauliche Austausch der Daten. Die Public Key Kryptographie hat sich als ein Werkzeug entwickelt, das dazu eingesetzt wird die oben genannten Sicherheitsziele weitestgehend zu erreichen. Der geheime (private) Schlüssel dieses Verfahrens bekommt eine besondere Bedeutung: er wird zur Identifikation, zum Beweis der Authentizität und zur Wahrung der Integrität der Daten genutzt. Ein gestohlener privater Schlüssel kann demnach katastrophale Folgen haben. Aus diesem Grund genießt der Schutz des privaten kryptographischen Schlüssels eine besondere Bedeutung.

Zur Aufbewahrung des Schlüssels auf dem Computer (in Software) wurde der PKCS#12 Standard definiert. Es gibt außerdem noch weitere softwarespezifische Lösungen. Eine sicherere alternative zur Aufbewahrung des Schlüssels in Software ist die Aufbewahrung in Hardware. Dazu bieten sich an HSM (Hardware Security Module), USB-Token oder Chipkarten. Die Chipkarten sind heutzutage sehr verbreitet wie zum Beispiel Bankkarten, Krankenversicherungskarten oder Essenkarte in Kantinen. Aber nur die wenigsten von ihnen werden auch im Zusammenhang mit Public Key Kryptographie genutzt. Dies liegt nicht zuletzt an den Kosten: unter Umständen hohe Kosten für Zertifikate sowie zusätzliche Kosten für die Kartenleser. Da aber viele Unternehmen auf die Vorteile der Chipkarten im Zusammenhang mit Public Key Infrastrukturen nicht mehr verzichten wollen, wird immer wieder versucht die Kosten dafür zu senken. In letzter Zeit wurden immer wieder Projekte gestartet, die einen effizienten Einsatz der Chipkarten als Ziel hatten.

Mit diesem Ziel wurde bereits 2001 auch das FINREAD Konsortium initiiert. Es sollte einen EU Standard definieren, der Kartenleser die im Finanzwesen eingesetzt werden sicher, einheitlich in der Funktionalität und günstig in der Produktion machen sollte. Das FINREAD Konsortium hat bis 2004 in einer Reihe von so genannten s (CEN Workshop Agreement) die Anforderungen an einem solchen Kartenleser definiert. Verschiedene Hersteller von Kartenlesern haben diese Anforderungen umgesetzt und FINREAD konforme Kartenleser hergestellt. Leider konnten sich die herausgearbeiteten Anforderungen nicht als ISO Standard durchsetzen. Trotzdem bieten die nach der FINREAD Spezifikation hergestellten Kartenleser ein höheres Maß an Sicherheit im Vergleich zu herkömmlichen Kartenleser. Außerdem sind sie universell einsetzbar, d.h. sie unterstützen verschiedene Chipkarten und verschiedene Protokolle.

Die vorliegende Arbeit beschäftigt sich mit den Möglichkeiten und Grenzen des Einsatzes von FINREAD Kartenlesern, insbesondere mit den Aspekten der Programmierung dieser. Es wird anhand einiger Beispiele gezeigt wie so genannte Finlets (das sind die auf dem Kartenleser ausgeführten Applets) programmiert, erstellt und auf den Kartenleser importiert werden. Im Rahmen der Bachelorarbeit wurde auch ein Java Wrapper für die FINREAD Host API entwickelt, die als DLL vorlag. Das ermöglicht, dass auch die Host Applikation in Java programmiert werden kann. Weil die Finlets ohnehin in Java programmiert werden ermöglicht es der Wrapper, dass FINREAD Applikationen einheitlich in Java programmiert werden. Das Ziel dieser Arbeit ist somit die Konzentration einiger relevanter Informationen, die für das Programmieren von FINREAD Applikationen nötig sind. Es soll späteren Programmierern von Finlets ein Werkzeug zur Herstellung von Hostapplikationen bieten (durch den Java Wrapper) sowie als Schnelleinstieg und Hilfe dienen.

Kapitel 2

Chipkarten und Sicherheit

Chipkarten haben seit vielen Jahren Einzug in das tägliche Leben gefunden. Sie sind bekannt als Essenskarten in Kantinen oder Krankenversicherungskarten. Diese sind 2 einfache Arten von Chipkarten, die vor allem als Datenspeicher persönlicher sensibler Daten dienen und nur wenige Berechnungen ausführen. Chipkarten können aber mehr als nur Daten zu speichern und so sind ihre Einsatzmöglichkeiten sehr vielfältiger. Mit PKI Schlüsseln ausgestattete Chipkarten können zum Beispiel zur Authentifikation herangezogen werden, digitale Signaturen erstellen oder kryptographische Operationen auf Daten anwenden wie MAC Berechnung oder Verschlüsselung von Daten.

2.1 Authentifikation

Der Prozess der Authentifikation ist der Prozess des “sich Ausweisens”. Im Bereich des Online Bankings “will” demnach das System sich davon überzeugen, dass sein Kommunikationspartner auch tatsächlich derjenige ist, der er behauptet. Bei der Authentifikation unterscheidet man zwischen wissensbasierter und besitzbasierter Authentifikation: Bei der ersten Form wird ein gemeinsames Geheimnis überprüft bei der zweiten ist der Nachweis des Besitzes eines Gegenstandes (Mensakarte, Krankenversicherungskarte usw.) der Nachweis der Identität. Sehr häufig trifft aber auch eine Mischform auf, wie z. B. bei der SIM Karte in Mobiltelefonen, wo man sowohl die Karte besitzen muss als auch die zugehörige PIN kennen muss um die Karte benutzen zu können.

Im Bereich von Online Banking wird noch sehr oft die Authentifikation durch Wissen praktiziert. Dazu muss der Benutzer ein vorher bestimmtes Passwort zur Verifikation der Identität eingeben. Diese Art von Authentifikation hat sich als sehr anfällig gegen Angriffe gezeigt, denn das statische Passwort kann sehr leicht entwendet werden. Mögliche Angriffe auf statische Passwörter oder andere mögliche Kompromittierungen sind:

Viren bzw. Trojaner: Angreifer können die Zugangsdaten der Anwender durch installierte Programme ausspähen.

Phishing: Anwender werden getäuscht ihre Zugangsdaten preis zu geben in-

dem Ihnen eine vom Aussehen her identische Webseite des vermeintlichen Kreditinstitutes angezeigt wird.

Online channel-breaking: Der Angreifer agiert als “Man in the Middle” zwischen Server und Benutzer und kann so die ausgetauschten Daten manipulieren.

Um die Sicherheitslücken der Authentifikation mit statischen Passwörtern teilweise zu schließen wurden von vielen Banken die so genannten iterierten TANs eingeführt, eine Liste von zusätzlichen statischen Einweg Passwörtern, die komplementär zu den Zugangsdaten benutzt werden. Diese erschweren zwar die Angriffe durch Phishing oder Trojaner sind aber gegen Online Channel Breaking Angriffe nutzlos.

Um ein höheres Maß an Sicherheit bei der Authentifikation zu erreichen, könnten Chipkarten eingesetzt werden. Die Chipkarten sind sehr flexibel, weil sie einen sicheren Datenspeicher anbieten mit zusätzlicher Möglichkeit der Berechnung kryptografischer Funktionen. Sie können personalisiert werden, d. h. für jeden Benutzer werden in einer sicheren Umgebung seine persönlichen Daten eventuell inklusive kryptographischer Schlüssel auf die Chipkarte geschrieben. Diese Daten sind gegen Manipulation gesichert und nahezu unmöglich zu “stehlen”. Die Benutzung der Chipkarte ist in der Regel erst nach der Verifikation einer PIN möglich. Die Authentifikation gegenüber eines Webservers (beim Beispiel des Online Bankings) erfolgt dann in einem Challenge-Response Verfahren bei dem die Challenge entweder verschlüsselt oder digital signiert wird.

2.2 Digitale Signaturen und MAC

Bei Online Banking ist neben der Authentifikation auch die Gewährleistung der Authentizität und Integrität der getauschten Daten wichtig. Um die Authentizität zu gewährleisten benutzt man digitale Signaturen. Indem man Prüfsummen über die Daten berechnet (Message Authentication Code), kann man deren Integrität überprüfen. Zum Beispiel kann von den Daten einer Geldtransaktion (Kontonummern, Geldsumme, Empfänger, usw.) der MAC Wert berechnet werden und die Daten können signiert werden. Diese Maßnahme würde auch einen Online Channel Breaking Angriff erschweren. Weil man auf Chipkarten auch kryptographische Schlüssel, die zur Erzeugung von digitalen Signaturen nötig sind, speichern kann, können Chipkarten dazu eingesetzt werden das Sicherheitsniveau zu erhöhen. Die Berechnung der digitalen Signatur erfolgt direkt auf der Karte, so dass der kryptographische Schlüssel die Karte nie verlassen muss. Das macht die Chipkarte zu einer sicheren Umgebung für die Speicherung und Berechnung von digitalen Signaturen.

2.3 Verschlüsselung

Chipkarten mit PKI konformen Schlüsseln können auch Verschlüsselung von Daten durchführen. Die meisten kryptographiefähigen Chipkarten beherrschen symmetrische Verschlüsselungsverfahren (wie AES oder 3-DES) sowie asymmetrische Verfahren (wie RSA). Analog zur Erstellung einer digitalen Signatur erfolgt auch die Verschlüsselung direkt auf der Chipkarte, so dass der geheime Schlüssel niemals die Karte verlässt. Im Szenario des Online Bankings kann man die Verschlüsselung dazu einsetzen, um die Vertraulichkeit der Daten zu sichern zum Beispiel dadurch, dass die Transaktionsdaten verschlüsselt werden. Zusätzlich kann man bei der Challenge-Response Authentifikation die Verschlüsselung zur Authentifikation benutzen.

2.4 Zusammenfassung und Ausblick

Durch den Einsatz von Chipkarten kann ein höheres Maß an Sicherheit erreicht werden. Die Sicherheitsaspekte der Chipkarten sind vor allem im Umfeld des Bankwesens interessant. Banken und Kreditinstitute werden im besonderen Masse von Angreifern bedroht. Vor allem im Bereich der Online Zahlung und der Online Kontoverwaltung haben sich die Sicherheitsmaßnahmen, die auf statische Passwörter und Einweg Passwörter aufbauen als sehr verwundbar erwiesen. Der Schaden, der jährlich durch gestohlene Zugangsdaten und Daten von Kreditkarten verursacht wird, ist enorm. Durch den Einsatz von Chipkarten mit kryptographischen Schlüsseln und Zertifikaten könnte das Bedrohungspotential drastisch reduziert werden. Eine solche Chipkarte kann zur Authentisierung des Benutzers (statt des Zugangspassworts) und zur Sicherung der Integrität der übertragenen Daten (digitale Signatur) eingesetzt werden.

Die Daten auf der Chipkarte sind wesentlich schwieriger zu entwenden als die statischen Passwörter, denn:

- Die Benutzung der sicherheitsrelevanten Funktionen der Chipkarte ist durch ein Passwort geschützt. Bei Verlust oder falls diese gestohlen wird, ist sie wertlos (ohne das zugehörige Passwort).
- Der geheime Schlüssel auf der Chipkarte, der zu Authentisierungs- und Signaturzwecken wichtig ist verlässt nie die Chipkarte. Operationen, die den Einsatz dieses Schlüssels erfordern werden auf der Karte selbst ausgeführt.
- Chipkarten können/sollten Schutzmechanismen gegen physische Angriffe anbieten, d.h. der geheime Schlüssel auf der Karte sollte zerstört werden, falls versucht wird den Chip der Karte zu manipulieren.

Chipkarten können noch zusätzliche Sicherheitsfunktionen besitzen. Zum Beispiel könnte eine gegenseitige Authentifikation zwischen Karte und Terminal stattfinden, bei der sowohl das Terminal als auch die Chipkarte sich ausweisen. Man könnte somit den Einsatz bestimmter Karten auf bestimmte Terminals

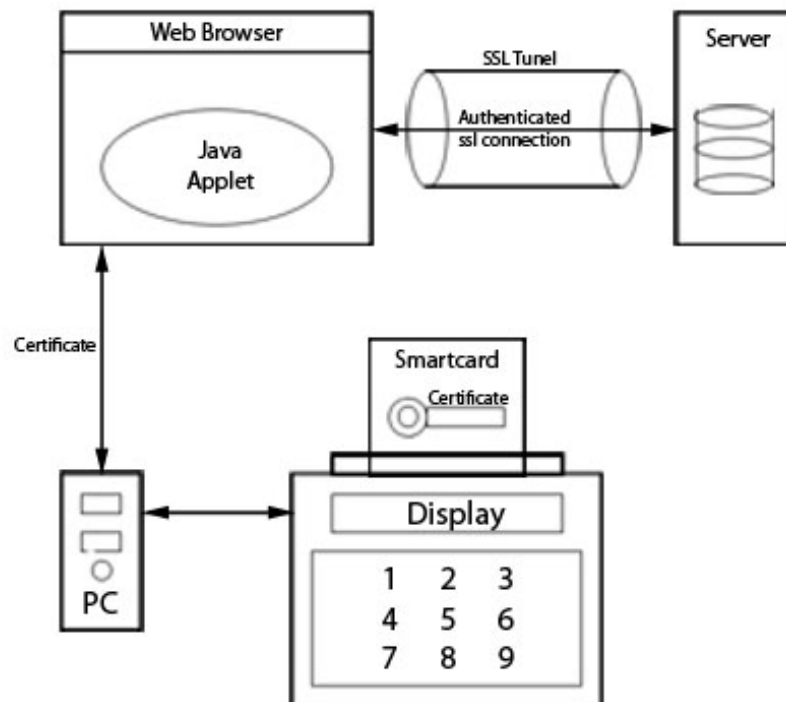


Abbildung 2.1: Beispielszenario für den Einsatz von Chipkarten im Online Zahlungsverkehr

eingrenzen.

Trotz dieser Vorteile hat sich der Einsatz von Chipkarten im Bankwesen für die Authentisierung der Kunden im Online Zahlungsverkehr noch nicht durchgesetzt. Dies hat verschiedene Gründe. Zunächst ist die Ausstattung aller Endbenutzer mit einem Kartenleser sowie mit Chipkarten und PKI konformen Schlüsseln und Zertifikaten eine kostenintensive Angelegenheit. Ein weiterer Grund ist der Mangel an standardisierten Chipkarten und Kartenleser. Speziell im Umfeld des Finanzwesens wäre es wünschenswert eine sichere Umgebung für den Einsatz der Chipkarten zu haben, denn die meisten herkömmlichen Kartenleser besitzen keinerlei Sicherheitsfunktionen.

Im Chipkartenbereich haben sich Java Cards als sehr flexible und zuverlässige Karten erwiesen. Im Bereich der Kartenleser wird aber immer noch um eine einheitliche Definition der Fähigkeiten "gekämpft". Es gibt immer wieder neue Projekte, deren Ziel die Etablierung eines Standards für Kartenleser ist. Auch die FINREAD Working Group wurde mit diesem Ziel gegründet. Auch wenn ihre Vorschläge nicht den Status des Standards erreicht haben, so bilden sie die Basis zu einem im Vergleich mit herkömmlichen Kartenlesern sichereren Kartenleser mit einheitlich definierten Funktionen, der zwar für den Einsatz im

2.4. ZUSAMMENFASSUNG UND AUSBLICK

Finanzwesen konzipiert wurde aber trotzdem universell einsetzbar ist.

Kapitel 3

Finread - Eine Einführung

3.1 Was ist FINREAD?

FINREAD steht für **F**inancial Transactional Integrated Circuit Card **R**eaders. Es ist eine technische Spezifikation für die Anforderungen an einen programmierbaren Kartenleser, der speziell die Bedürfnisse im bargeldlosen Zahlungsverkehr im Internet abdecken sollte. Das FINREAD Konsortium wurde 2001 gegründet und wurde durch Mittel der Europäischen Union unterstützt. Dem Konsortium schlossen sich mehrere große europäische Kreditkarten Institute an sowie Hersteller von Kartenlesern. Das Ziel war einen Standard für die Hardware Architektur und die Software API eines programmierbaren Kartenlesers zu definieren. Die Motivation dazu kam dadurch, dass immer mehr Banken und Kreditkarten Institute von Magnetkarten auf Chipkarten umstiegen, während die Kartenleser, die es auf dem Markt gab weder Sicherheitsvorkehrungen hatten, noch eine einheitliche Architektur der Hardware. Zu dieser Zeit (um das Jahr 2000) haben sich die Kartenleser darauf beschränkt Daten zwischen PC und Chipkarte weiterzuleiten, waren also nicht programmierbar.

Die FINREAD Spezifikation sollte also diese Missstände beheben. Sie definiert demnach die Hardware Architektur eines universellen, programmierbaren Kartenlesers sowie die APIs, die ein solcher Kartenleser anbieten soll. Insbesondere muss die Spezifikation auch Sicherheitsanforderungen berücksichtigen. Um diese Sicherheitsanforderungen zu erfüllen, wurden für den FINREAD Kartenleser so genannte Betriebsmodi definiert. Im Folgenden werden einige Aspekte der Spezifikation und wichtige Konzepte zum Erreichen eines höheren Sicherheitsmaßes vorgestellt.

3.2 Allgemeine Anforderungen

3.2.1 Allgemeine Hardware Anforderungen

Der FINREAD Kartenleser muss ein Display sowie Eingabetasten für Ziffer besitzen. Diese sollen eine Interaktion der FINREAD Applikation (Finlet) mit dem Benutzer ermöglichen um dadurch das Sicherheitsniveau zu erhöhen. Z. B. wird das Display benutzt um den Benutzer über die Änderung des Betriebsmodus

zu informieren. Über die Tastatur wird seine Zustimmung für bestimmte Aktionen eingeholt oder die PIN abgelesen um bestimmte Operationen der Chipkarte freizuschalten. Natürlich muss der Kartenleser über ein ICC Interface verfügen, das sowohl den EMV Anforderungen als auch dem ISO Standard genügen. Des Weiteren muss der Kartenleser über ein Interface zum Host Computer verfügen, das eine möglichst breite Unterstützung hat. In letzter Zeit hat sich USB als quasi Standard für ein Interface zum Benutzer Host entwickelt. Das genaue Design der einzelnen Hardwarekomponente wird den Herstellern überlassen.

Eine besondere Beachtung findet das so genannte Security Module, in dem der private Schlüssel aufbewahrt wird, der für die Identifikation des Kartenlesers benutzt wird. Sowohl der Speicherplatz als auch die Subsysteme, die mit diesem Schlüssel arbeiten, müssen Manipulationssicherheit garantieren.

3.2.2 Interoperabilität

Herkömmliche Kartenleser wurden zum Teil für spezielle Einsatzbereiche entwickelt. Darüber hinaus verwendeten viele Hersteller proprietäre Programmierschnittstellen, so dass die Entwicklung von Applikationen für Kartenleser erheblich erschwert wurde. Vor allem waren die Applikationen an den Hersteller des Kartenlesers gebunden bzw. an dessen Programmierschnittstellen und konnten nicht wieder verwendet werden. Als Entwickler musste man für die gleiche Funktionalität für n verschiedene Kartenleser n verschiedene Applikationen schreiben. Durch die Einführung des PC/SC Standards wurde ein gewisses Maß an Interoperabilität erreicht, denn die Applikationen für Kartenleser konnten unabhängig vom proprietären Treiber entwickelt werden. Die FINREAD Spezifikation soll einen weiteren Grad an Unabhängigkeit einführen: sie beschreibt einen Kartenleser, der in verschiedenen Anwendungsbereichen einsetzbar ist, und für den einheitliche Applikationen programmiert werden können. Es soll ermöglichen, dass eine Applikation auf Kartenleser verschiedener Hersteller lauffähig ist, also eine Applikation für einen Kartenleser-Typ. Dies wird sichergestellt durch:

- Beschreibung einer gemeinsamen Hardwarearchitektur
- Beschreibung der API, die der Rechner benutzen kann um Kartenleser und Chipkarte anzusprechen (Host API)
- Beschreibung einer Kartenleser Software, die eine JVM für die Ausführung von Java Applikationen bereithält
- Beschreibung der API für Applikationen, die auf dem Kartenleser ausgeführt werden (Reader API)

Weil der FINREAD Kartenleser in verschiedenen Bereichen einsetzbar sein soll, muss er auch verschiedene Chipkarten unterstützen mit verschiedenen Protokollen. Speziell muss er für den Einsatz im Finanzbereich bestimmte Sicherheitsanforderungen erfüllen, die später in diesem Kapitel angesprochen werden.

3.3 Programmierschnittstellen

3.3.1 Host API

Über die Host API können Host Applikationen auf den Kartenleser zugreifen und im Betriebsmodus “Transparent Mode” auch auf die Chipkarte. Die in der FINREAD Spezifikation beschriebene Host API bietet vor allem allgemeine Funktionen an, um Informationen über den Kartenleser und die darauf installierte Software herauszufinden. Außerdem besteht die Möglichkeit Befehle (APDUs) an die Chipkarte zu senden. Eine FINREAD typische Funktionalität ist die Möglichkeit, die auf dem Kartenleser befindlichen Applikationen (die Finlets) zu starten, ihnen Parameter zu übergeben und die Ergebnisse ihrer Ausführung zu übernehmen. Die Host API wird in der FINREAD Spezifikation genau definiert aber ihre letztendliche Implementierung ist herstellerspezifisch. Im Rahmen dieser Bachelorarbeit wurde für die Implementierung der Host API von Omnikey, die als Windows DLL vorlag ein Java Wrapper programmiert, der in einem späteren Kapitel beschrieben wird. Im Kapitel 6 werden dann auch die Funktionen der Host API detaillierter behandelt. Eine genaue Beschreibung der Host API befindet sich im CWA 8 ([11]).

3.3.2 Core Software

Das Betriebssystem des Kartenlesers ist Herstellerspezifisch. Es kontrolliert über herstellerspezifische Treiber die einzelnen Komponenten des Kartenlesers, wie Display, Tastatur, Chipkarten-Interface. Die Core Software muss eine Java Virtual Machine bereithalten, die es ermöglicht Java Applikationen auszuführen. Diese Applikationen für den FINREAD Kartenleser werden im Folgendem Finlets (Finread Card Reader Application) oder auch FCRAAs genannt. Die Core Software ermöglicht den Finlets den Zugriff auf die Ressourcen des Kartenlesers durch die so genannte Reader API. Die CWAs 2, 4 und 6 ([5], [7], [9]) beinhalten detaillierte Informationen über die Core Software des FINREAD Kartenlesers.

3.3.3 Reader API - Das STIP Framework

Nach der FINREAD Spezifikation muss die Core Software eine JVM bereithalten, die die Ausführung von Java Applikationen auf dem Kartenleser ermöglichen. Diese Applikationen können durch ein wohl definiertes Interface mit den einzelnen Komponenten des Kartenlesers kommunizieren. Dieses Interface wird in der FINREAD Spezifikation genau beschrieben. Im Folgenden wird dieses Interface auch als Reader API bezeichnet.

Die FINREAD Spezifikation der Reader API basiert hauptsächlich auf das STIP Framework, das um einige Funktionen erweitert wurde.

STIP

Steht für “Small Terminal Interoperable Platform” und beschreibt eine technische Spezifikation, die vom STIP Konsortium herausgearbeitet wurde. Das

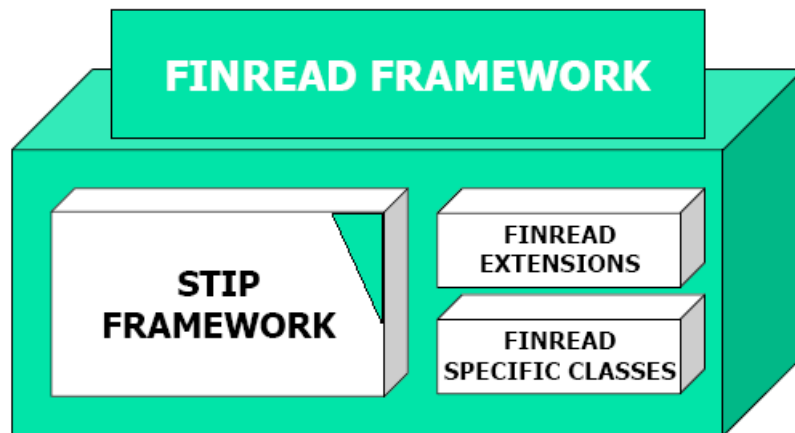


Abbildung 3.1: Das FINREAD Framework

STIP Konsortium kam im Jahr 2000 zusammen. Ihm schlossen sich einige große Finanzinstitute sowie Hersteller von Kartenlesern an. Ähnlich der FINREAD Initiative verfolgte man das Ziel mit der STIP Spezifikation eine offene Plattform für programmierbare Kartenleser zu erschaffen, die die verschiedensten Kartenlesegeräte unterstützen sollte, und die eine sichere Umgebung für die Ausführung sicherheitskritischer Applikationen auf den Kartenlesern anbieten sollte. Die Spezifikationen des STIP Konsortiums wurden als Basis für die FINREAD Spezifikation der Reader API genommen. Im Kapitel 4: "Entwicklung von Applikationen für den FINREAD Kartenleser" wird ein Überblick über die Architektur des STIP Frameworks gegeben sowie über Grundlagen der Entwicklung von STIP konformen Applikationen. Die Arbeit des STIP Konsortiums selbst wurde eingestellt aber das Projekt wurde in ein größeres Projekt eingegliedert: das so genannte **Embedded FINREAD**.

3.4 Sicherheitskonzepte

Das Sicherheitskonzept des FINREAD Kartenlesers unterscheidet ihn von herkömmlichen Kartenlesern. Als Anforderung durch den Einsatz im Finanzbereich muss er eine sichere Umgebung für die Ausführung kritischer Operationen anbieten. Die FINREAD Spezifikation erreicht dieses höhere Sicherheitsmaß durch die so genannten Betriebsmodi (Operating Modes) und dadurch, dass nur signierte Finlets (FCRAs) im Kartenleser ausgeführt werden.

3.4.1 Signierte Software

Die FINREAD Spezifikation verlangt, dass auf dem Kartenleser nur signierte Software zur Ausführung kommt. Sowohl die Core Software als auch die Finlets müssen eine für den Kartenleser prüfbare Signatur besitzen. Die Core Software des FINREAD Kartenlesers ist herstellereigentlich. Der Hersteller muss demnach dafür sorgen, dass die Core Software in einer signierbaren Form gepackt auf den

Kartenleser heruntergeladen wird, und dass ihre Signatur überprüft wird. Die Finlets ihrerseits basieren auf dem Datentyp Jeff, der wie Jar-Archive signiert werden kann. Wenn ein Finlet oder neue Core Software auf den Kartenleser installiert wird, wird die Signatur überprüft. Es wird nur Software installiert, die eine gültige Signatur besitzt. Diese Vorkehrung im Zusammenhang mit dem Secure Mode machen den FINREAD Kartenleser zu einer sicheren Ausführungsumgebung für Applikationen.

3.4.2 Kryptographische Schlüssel

Die Signaturen der installierten Software wird auf Basis von PKI verifiziert. D. h., dass zur Überprüfung der Authentizität eines Public Keys, der eine Signatur verifiziert ein entsprechendes Zertifikat vorhanden sein muss. Die FINREAD Spezifikation verlangt, dass Zertifikatsketten mit maximal 3 Stufen verifiziert werden können. Um diese Verifikation durchzuführen, befinden sich auf dem Kartenleser 2 Public Keys (Root Public Keys). Des Weiteren muss der FINREAD konforme Kartenleser die Möglichkeit haben sich durch ein Zertifikat zu identifizieren. Für die Verwaltung der Schlüssel ist ein weiterer Public Key nötig, der die Signatur eines neuen Schlüssel Paketes überprüfen kann.

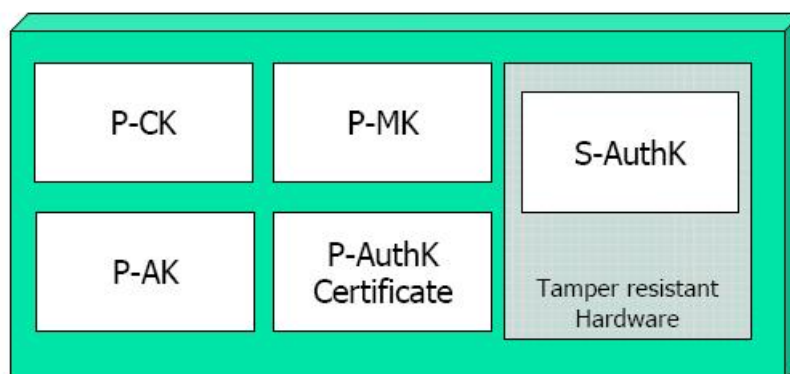


Abbildung 3.2: Die kryptographischen Schlüssel auf dem FINREAD Kartenleser

- P-CK(Public Core Software Key): Public Root Key zur Überprüfung der Signatur neuer Core Software
- P-AK(Public Application Key): Public Root Key zur Überprüfung der Signatur der Applikationen, die auf dem Kartenleser installiert werden
- P-MK(Public Master Key): Public Root Key zur Überprüfung der Signatur neu geladener Schlüsselpakete
- P-AuthK Certificate: Zertifikat, mit dem der Authentifizierungsschlüssel des Kartenlesers verifiziert werden kann
- S-AuthK: Privater Schlüssel, der vom Kartenleser zur Authentifizierung herangezogen wird

S-Authk und P-AuthK bilden ein Schlüsselpaar, das zur Authentifikation des Kartenlesers benutzt wird. Der Kartenleser wird mit diesem Schlüsselpaar ausgeliefert und das Schlüsselpaar kann auch nicht mehr erneuert werden. Das Zertifikat des Public Keys kann mit einem vom Hersteller zur Verfügung gestellten Public Key verifiziert werden. Um eine Applikation zu programmieren, die den Kartenleser authentifiziert, muss man also vorher sich den zugehörigen Public Key des Herstellers besorgen und dessen Zertifikatspfad verifizieren.

Die Schlüssel P-CK und P-AK sind Public Root Keys. Sie werden benutzt um Core Software bzw. Finlets zu verifizieren. Diese Schlüssel können erneuert werden, wenn es nötig sein sollte. Man kann signierte Schlüsselpakete herunterladen, die mit dem Public Master Key (P-MK) verifizierbar sein müssen. Der Public Master Key wird in einem Personalisierungsprozess auf den Kartenleser übertragen und der zugehörige private Schlüssel dem Benutzer in einem PKI konformen Prozess übergeben. Der Master Key kann nicht erneuert werden.

Um eine Applikation auf dem Kartenleser installieren und ausführen zu können, muss man folgende Voraussetzungen erfüllen:

- Besitz des Secret Master Keys
- Public Root Keys (P-AK, P-CK) in einem signierten Paket auf den Kartenleser laden (Signatur mit dem zum P-MK zugehörigen privaten Schlüssel)
- Applikation mit dem zum P-AK zugehörigen privaten Schlüssel signieren

3.4.3 Operating Modes

Man kann den Kartenleser auf verschiedenen Sicherheitsstufen betreiben. Diese Sicherheitsstufen werden Betriebsmodi (Operating Modes) genannt.

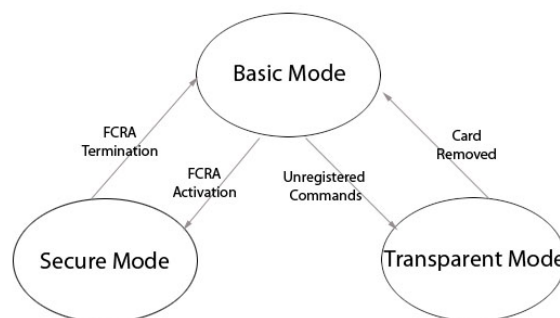


Abbildung 3.3: Operating Modes

Basic Mode: Ist der initiale Betriebsmodus des Kartenlesers. Befehle, die von außerhalb des Kartenlesers an die Chipkarte gesendet werden, werden untersucht und sicherheitskritische Befehle werden blockiert. Der Kartenleser wechselt vom Basic in den Secure Mode sobald ein Finlet auf dem Kartenleser durch die Host

Applikation aktiviert wird.

Secure Mode: Wird immer aktiviert, wenn ein FCRA (Finlet) aktiviert wurde. Alle Befehle, die von außerhalb des Kartenlesers an die Chipkarte gesendet werden, werden blockiert. Nur die aktivierte FCRA hat Zugriff auf die Chipkarte. Der Secure Mode kann auch dann aktiviert werden, wenn kein Finlet aktiv ist und eine Chipkarte eingeführt wurde. Dazu muss ein spezielles Finlet programmiert werden (eine so genannte FCRIA)

Transparent Mode: Kann vom Benutzer aktiviert werden. Der Kartenleser funktioniert wie ein gewöhnlicher, herkömmlicher Leser, d. h. alle Befehle werden bedingungslos an die Chipkarte weitergegeben. Der FINREAD konforme Kartenleser sollte den Benutzer über diesen Zustand deutlich informieren (z. B. mit einem Hinweis auf dem Display des Kartenlesers), denn der Transparent Mode schaltet alle FINREAD Sicherheitsmassnahmen aus. Es ist auch möglich spezielle Applikationen zu programmieren, die beim Einführen einer Chipkarte automatisch gestartet werden, die Chipkarte überprüfen und gegebenenfalls den Wechsel in den Transparent Mode vollständig unterbinden. Diese Applikationen sind spezielle Finlets und werden FCRIA genannt. So können z. B. Kreditinstitute sicherstellen, dass ihre Karten nur in einer sicheren Umgebung benutzt werden.

3.5 Zusammenfassung

Die FINREAD Spezifikation soll eine sichere Plattform für Kartenlesegeräte ermöglichen sowie die Interoperabilität zwischen den verschiedenen Kartenlesern und Chipkarten sicherstellen. Um die Interoperabilität sicherzustellen, definiert die FINREAD Spezifikation zwei APIs: eine Host API, die die Kommunikation mit dem PC beschreibt und eine API für die Applikationen, die auf dem Kartenleser ausgeführt werden (Reader API). Die Reader API basiert auf dem STIP Framework, das um einige Funktionen erweitert wurde. Weiterhin beschreibt die FINREAD Spezifikation auch die Anforderungen an die Hardware eines FINREAD konformen Kartenlesers. Um eine sichere Umgebung zu gewährleisten, kann der Kartenleser die Ausführung signierter Applikationen erzwingen. Des Weiteren bietet der FINREAD konforme Kartenleser einen sicheren Speicher für kryptographische Schlüssel an, die dazu benutzt werden um die Zertifikate der Software zu verifizieren, die auf dem Kartenleser zur Ausführung kommen soll. Der Kartenleser kann sich gegenüber einer Host Applikation oder Chipkarte identifizieren.

Kapitel 4

Entwicklung von Applikationen für den FINREAD Kartenleser

4.1 Grundlagen zu Finlets

Finlets oder FCRA's werden die signierten Applikationen genannt, die auf dem FINREAD Kartenleser ausgeführt werden. Die Finlets sind gewöhnliche Java Programme, die mit den entsprechenden Editoren bzw. Entwicklungsumgebungen geschrieben werden. Das Besondere ist, dass die kompilierten Klassen mit Hilfe von Programmen, die gewöhnlich von den Herstellern von Kartenleser zur Verfügung gestellt werden, in ein besonderes Format umgewandelt werden. Dieses Format basiert auf JEFF (J Executable File Format). Die Klassen werden zunächst in JEFF konvertiert und signiert. Mit einem Hersteller Tool wird aus der JEFF-Datei eine FIN-Datei erzeugt. Diese Datei ist das so genannte Finlet, das auf dem Kartenleser heruntergeladen wird.

Wie bereits angesprochen, haben die Finlets Zugriff auf die Ressourcen des Kartenlesers. Als API für diese Funktionalität steht dem Finlet das STIP-Framework zur Verfügung. D. h. ein Finlet muss das STIP-Interface implementieren.

Bei der Entwicklung von Finlets ist zu beachten, dass es sich bei der Java Virtual Machine auf dem Kartenleser um eine KVM handelt, eine Virtual Machine für mobile Geräte mit wenig Speicher und geringer Prozessorleistung. Diese Virtual Machine ist im Umfang begrenzter als die Standard JVM.

4.2 STIP-Small Terminal Interoperability Platform

Wie bereits im Kapitel 3.3.3 erwähnt, basiert das FINREAD Framework hauptsächlich auf das STIP Framework. Ähnlich dem FINREAD Projekt versuchte auch die STIP Gruppe eine sichere Plattform für programmierbare Kartenleser zu erschaffen. Zu den Sicherheitsaspekten gehört auch die Architektur des STIP Frameworks. Es hat eine Service orientierte Architektur. D. h., dass der Zugriff

auf die Ressourcen des Kartenlesegerätes nur durch so genannte Service Control Klassen möglich ist. Eine Applikation muss sich die gewünschten Services über einen zentralen Access Manager beschaffen und kann danach Anfragen an die Services stellen (wie z. B. die Anforderung an den UIControl Service das Display zu löschen).

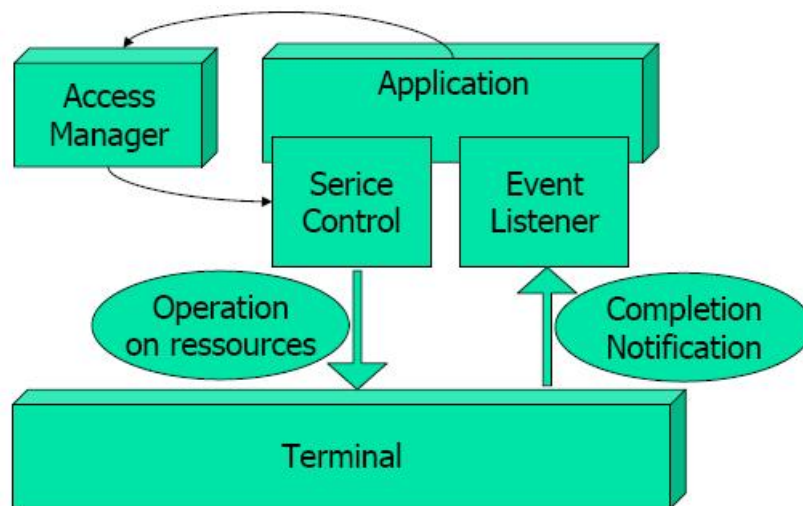


Abbildung 4.1: Service orientierte Architektur des STIP Frameworks

Die Operationen, die ein Service Control ausführt, sind asynchron. D. h. der Service Control initiiert die Operation wartet aber nicht auf die Ergebnisse der Ausführung. Z. B. initialisiert der UIControl das Löschen des Displays und die Applikation wird weiter ausgeführt ohne zu wissen ob das Löschen des Displays erfolgreich war. Um auf die Ergebnisse der Ausführung des Service Controls zugreifen zu können, müssen Operation Listener mit dem Service Control registriert werden (z. B. nach der Operation des Smartcard Services `open` folgt ein Event `Open Completed`). Zusätzlich zu den Operation Listeners können so genannte Status Listeners mit einem Service Control registriert werden. Diese initiieren ein Event jedes Mal wenn sich der Status des Services ändert (z. B. `Card Inserted Event` wenn der Smartcard Service registriert, dass eine Chipkarte eingeführt wurde). Eine Applikation bekommt die Service Control Objekte von einem Access Manager. Dieser fungiert als eine Factory für Service Control Instanzen. Das Konzept des Access Managers ermöglicht auch das Durchsetzen von gewissen Sicherheitsregeln gegenüber der Applikation. Das STIP Framework soll zusätzlich noch die Interoperabilität zwischen mehreren Applikationen ermöglichen.

4.3 Finlets entwickeln

Weil die Applikation für den Kartenleser auf das STIP Framework basiert, muss ein Finlet das Interface `stip.stiplet.Stiplet` implementieren. Dieses Inter-

face wird vom Framework wie ein Applet benutzt. D. h. das Interface besitzt die abstrakten Methoden `init()`, `start()`, `terminate()`, `install()` und noch einige andere. Wie bei einem Applet üblich ruft das Framework die entsprechenden Methoden zum geeigneten Zeitpunkt auf. Die Methode `init()` bekommt als Parameter den Access Manager von dem die Applikation sich die benötigten Services holen kann. Um diese Services auch benutzen zu können, muss das Finlet zusätzlich noch die entsprechenden Listeners implementieren und sie mit dem zugehörigen Service registrieren. Ein typisches Finlet würde in etwa so aussehen:

```
import stip.stiplet.Stiplet;
import stip.control.*;
import stip.devicecontrol.smartcardslot.SCSControl;
import stip.devicecontrol.smartcardslot.SCSOperationListener;
import stip.devicecontrol.smartcardslot.SCSStatusListener;

public class Finlet implements Stiplet,
                               SCSOperationListener,
                               SCSStatusListener {

    private AccessManager am;
    private SCSControl scs;

    private final static String SCS = "finread.devicecontrol.
        smartcardslot.SCSControl";

    public void init(AccessManager accessManager, ClientProxy
        clientProxy, byte[] obj) {
        this.am = accessManager;
        this.scs = (SCSControl) am.getServiceControl(SCS);
    }

    public void start(int i, byte[] obj) {
        // Method body
    }
    // Class body
}
```

Listing 4.1: Ein Finlet implementieren

Um auf die Ressourcen des Kartenlesers zuzugreifen, muss man die Services benutzen. Zum Beispiel um das Smartcard Interface zu aktivieren nutzt man den `SCSControl`, den man zuvor vom `AccessManager` bekommen hat:

```
scs.open();
```

Der Service initiiert den Vorgang des Öffnens des Smartcard Interfaces, wartet aber nicht auf das Ergebnis der Ausführung. Um zu erfahren ob das Interface erfolgreich geöffnet wurde, muss die Methode des `SCSOperationListeners` vom `Finlet` implementiert werden. Die Implementierung könnte so aussehen:

```
public void openCompleted(OperationEvent event) {
    if (event.getStatus() != OperationEvent.EVT_SUCCESS) {
        // Öffnen hat nicht geklappt
        // Ausführung beenden
    }else{
```

KAPITEL 4. ENTWICKLUNG VON APPLIKATIONEN FÜR DEN FINREAD KARTENLESER

```
        // Öffnen des Smartcard Interfaces war erfolgreich
        // Man kann nun mit der Smartcard kommunizieren
    }
}
```

Listing 4.2: Operation Event Methode openCompleted

Der Aufbau des Finlets ist also streng Event orientiert. Nach jedem Aufruf einer Servicefunktion wird die Ausführung des Finlets (falls das Ergebnis der Funktion wichtig ist) in einer anderen Methode fortgesetzt. Im obigen Beispiel sollte also nach dem

```
scs.open();
```

keine weitere Service Methode mehr aufgerufen werden, die auf ein Event wartet. Ein negatives Beispiel:

```
public class Finlet implements ... {
    public void start(int i, byte[] obj) {
        getATR();
    }

    private void getATR(){
        scs.open();
        scs.powerOn();
        scs.powerOff
    }
    ...
    public void openCompleted(OperationEvent event) {
        if (event.getStatus() == OperationEvent.EVT_SUCCESS) {
            System.out.println("Open succeeded");
        }
    }
    public void powerOnCompleted(SCSOperationEvent event) {
        // Wenn open() erfolgreich war können wir das ATR bekommen
        if (event.getStatus() == OperationEvent.EVT_SUCCESS) {
            this.atr = new byte[event.getDataLength()];
            event.getData(this.atr, 0);
        }
    }
    public void powerOffCompleted(){
        if (event.getStatus() == OperationEvent.EVT_SUCCESS) {
            this.clientProxy.notifyCompletion(...);
        }
    }
} // End Class
```

Listing 4.3: Negatives Beispiel der Programmierung eines Finlets

In diesem Beispiel soll die ATR der Chipkarte ausgelesen werden. Weil das Design des STIP Frameworks nur die Ausführung eines einzigen Threads (**Single Thread**) kennt, werden zunächst die Methoden

```
scs.open();
scs.powerOn();
scs.powerOff();
```

hintereinander ausgeführt. Die Eventmethoden werden erst anschließend bearbeitet (im gleichen Thread). In der Eventmethode `powerOnCompleted()` wird die Antwort der Chipkarte mit `event.getData()` ausgelesen und in einer lokalen Variable gespeichert. Doch in diesem Fall ist die Instanz `event` auf die zugegriffen wird nicht diejenige, die erwartet ist. Nach dem Aufruf von `powerOn()` liegt die "richtige" Instanz der Variablen `event` vor. Doch der Aufruf von `powerOff()` überschreibt diese. Weil die Eventmethode später aufgerufen, wird erhält sie die falsche Instanz von `event`. Die Ausführung des obigen Codes würde wahrscheinlich zu einem fehlerhaften Ergebnis führen, wenn nicht sogar zum Absturz des Programms. Um die ATR zu bekommen müsste der obige Code folgendermaßen geändert werden:

```
public class Finlet implements ... {

    public void start(int i, byte[] obj) {
        scs.open();
    }
    public void openCompleted(OperationEvent event) {
        if (event.getStatus() == OperationEvent.EVT_SUCCESS) {
            scs.powerOn();
        }
    }
    public void powerOnCompleted(SCSOperationEvent event) {
        if (event.getStatus() == OperationEvent.EVT_SUCCESS) {
            this.atr = new byte[event.getDataLength()];
            event.getData(this.atr, 0);
            scs.powerOff();
        }
    }
    public void powerOffCompleted(){
        if (event.getStatus() == OperationEvent.EVT_SUCCESS) {
            this.clientProxy.notifyCompletion(...);
        }
    }
} // End Class
```

Listing 4.4: Richtiger eventorientierter Ansatz

D. h. nach jedem Aufruf einer Service Klasse wird in der entsprechenden Methode des Listeners die erfolgreiche Ausführung abgewartet bevor die nächste Service Methode aufgerufen wird.

Die ATR der Karte wurde mit

```
event.getData(this.atr, 0);
```

in die Klassenvariable `atr` kopiert und kann eventuell an die Host Applikation zurückgegeben werden. Um Daten an die Hostapplikation zu senden, wird der so genannte `ClientProxy` benutzt, der von dem STIP-Framework zusammen mit dem `AccessManager` in der Initialisierungsphase dem `Finlet` übergeben wird.

```
clientProxy.notifyCompletion(ClientProxy.COMPLETION_SUCCESS,
    returnData);
```

Listing 4.5: Return Data an den Client weiterleiten

Die Variable `returnData`, die an die Host Applikation übermittelt wird, ist ein Byte Array.

4.4 Konsequenzen aus dem Single Thread Paradigm

Dass die Ausführung einer STIP Applikation in einem einzigen Thread erfolgt, hat zunächst Vorteile für die Portabilität der Applikation. Es entfällt auch die fehleranfällige Koordination mehrerer Threads. Falls die Events in parallelen, konkurrierenden Threads bearbeitet würden, so würde man synchronisierte Methoden brauchen um die potentiellen Konflikte zu vermeiden. Des Weiteren hat jede Applikation eine einfache Struktur.

Ein großer Nachteil dieses Design Prinzips ist, dass die Applikation sehr schwer lesbar ist. Um dem Programmfluss zu folgen, muss man im Code stets von einem Event zum nächsten springen. Je mehr Funktionalität ein Finlet hat desto mehr Listeners muss es implementieren und somit umso schwerer wird es den Überblick zu behalten. Beim "Springen" zwischen den Events ist es manchmal nötig den Zustand der Ausführung in Klassenvariablen zu führen.

Die JVM auf dem Kartenleser unterstützt aber mehrere Threads. Wenn es unbedingt nötig sein sollte, könnte man also die Ausführung einzelner Funktionen in separaten Threads laufen lassen. Der Programmierer ist dann aber für die Verwaltung der Threads verantwortlich. STIP bietet mit der Methode `waitEvent()` des "AccessManager" eine Möglichkeit den laufenden Thread zu stoppen und auf das Eintreten eines Events zu warten (es wird auf das Eintreten irgendeines Events gewartet und nicht eines bestimmten Events). Diese Methode ist dazu gedacht synchrone Methoden mit der asynchronen Architektur von STIP nutzbar zu machen. Sie sollte in einfachen Finlets nicht benutzt werden.

Kapitel 5

Finlets in Betrieb

Die bisherige Beschreibung der Eigenschaften von Finlets und deren Programmierung war allgemeiner Art. Sie bezog sich hauptsächlich auf die FINREAD Spezifikation (CWAs). Um Finlets zu erstellen, ist man aber auf spezielle Programme der Hersteller der Kartenleser angewiesen. Z. B. um aus der Jeff Datei eine Fin Datei zu erzeugen, wird ein Hersteller Programm benutzt. Der Einsatz von Programmen, die durch den Hersteller des Kartenlesers zur Verfügung gestellt werden, ist auch in der FINREAD Spezifikation so gefordert. Es schränkt die Portabilität der Finlets allerdings nicht ein, denn jedes Hersteller Tool stellt aus einer Jeff Datei die gleiche Fin Datei her. Ab Diesem Kapitel werden sich die Angaben auf einen speziellen Kartenleser beziehen, der im Rahmen dieser Bachelorarbeit benutzt wurde. Es handelt sich um den **3821 USB Pinpad FINREAD** Kartenleser der Firma **Omnikey**.



Abbildung 5.1: Kartenleser Omnikey 3821 USB Pinpad FINREAD

5.1 Kompilieren des Codes

Die JVM auf dem Kartenleser ist eine Version 1.1 JVM. Aus diesem Grund muss der Code aus dem das Finlet erzeugt werden soll, mit dem Parameter `-target 1.1` kompiliert werden. Also:

```
javac -source 1.3 -target 1.1 -classpath C:/path/to/  
finread_interface.jar -d ./classes ./dirs/*.java
```

Listing 5.1: Parameter für den Java Compiler

Damit mit dem Parameter `-target 1.1` kompiliert werden kann, müssen auch die Source Dateien als Version 1.3 oder kleiner deklariert werden. D. h. man muss auch das Parameter `-source 1.3` verwenden.

5.2 Finlet erzeugen

Das Erstellen eines Finlets erfolgt in mehreren Schritten. Zuerst muss aus allen Klassen, die zu dem Finlet gehören, eine Jeff-Datei erstellt werden. Dazu benutzt man den von Omnikey zur Verfügung gestellten Jeff-Converter, ein Tool für die Kommando Zeile.

```
jeffc -o /path/for/output/Finletname.jeff /path/to/finlet/classes  
/*.class
```

Listing 5.2: Erzeugen der Jeff Datei

Dies erzeugt aus allen Klassen eine Jeff-Datei. Damit das entstehende Finlet überhaupt auf den Kartenleser installiert werden kann, muss die Jeff-Datei zu diesem Zeitpunkt signiert werden. Dazu benötigt man allerdings den geheimen Schlüssel, den man vom Hersteller in einem Personalisierungsprozess überreicht bekommt. Leider konnte der Schlüssel des Kartenlesers, der für diese Arbeit verwendet wurde von Omnikey nicht erhalten werden. Damit man die Finlets trotzdem installieren und testen konnte, stellte Omnikey eine spezielle Core Software bereit, die einen zusätzlichen Development Mode besaß (zusätzlich zu dem Secure, Basic und Transparent Mode). In diesem Modus konnten auch nicht signierte Finlets installiert und ausgeführt werden. Aus diesem Grund wurde der Schritt der Signierung der Jeff-Datei auch nicht durchgeführt.

Die Jeff-Datei wird mit Hilfe eines weiteren Tools von Omnikey dem so genannten BuildPackage in eine .fin Datei überführt.

- Als Input File wird die vorher erstellte Jeff-Datei gewählt.
- Der Dateiname des Outputs endet auf .fin: Finletname.fin.
- FCRA Name benennt das Finlet und kann beliebig gewählt werden. Er kann dazu benutzt werden um ein Finlet zu aktivieren.
- FCRA Fully Qualified Name muss den voll qualifizierten Namen der Klasse angeben, die das Stiplet Interface implementiert: `pac.name.FinletName`.



Abbildung 5.2: BuildPackage

Die restlichen drei Felder haben keine direkte Auswirkung auf die Lauffähigkeit des Finlets. Sie sind nur zusätzliche Informationen zum dem Finlet und können von einer Host Applikation abgefragt werden.

Wenn man auf dem Button Build Package klickt, wird das Finlet erstellt. Das Programm BuildPackage gibt kaum Fehlermeldungen aus. Das Finlet wird auch mit einem falschen Fully Qualified Name erstellt, der Aufruf des Finlets wird aber mit einem Fehler enden.

5.3 Administration Tool

Das erstellte Finlet muss nun auf den Kartenleser heruntergeladen werden. Dazu bieten in der Regel die Hersteller Tools an, mit welchen das gemacht werden kann. Von Omnikey wird das Administration Tool dazu benutzt die Verwaltung des Kartenlesers durchzuführen sowie Finlets zu löschen und herunterzuladen.

Die wichtigsten Felder des Verwaltungsprogramms sind:

- Installed Applications: zeigt eine Liste aller auf dem Kartenleser befindlichen Finlets an, sowie Informationen zu den einzelnen Finlets. Mit dem

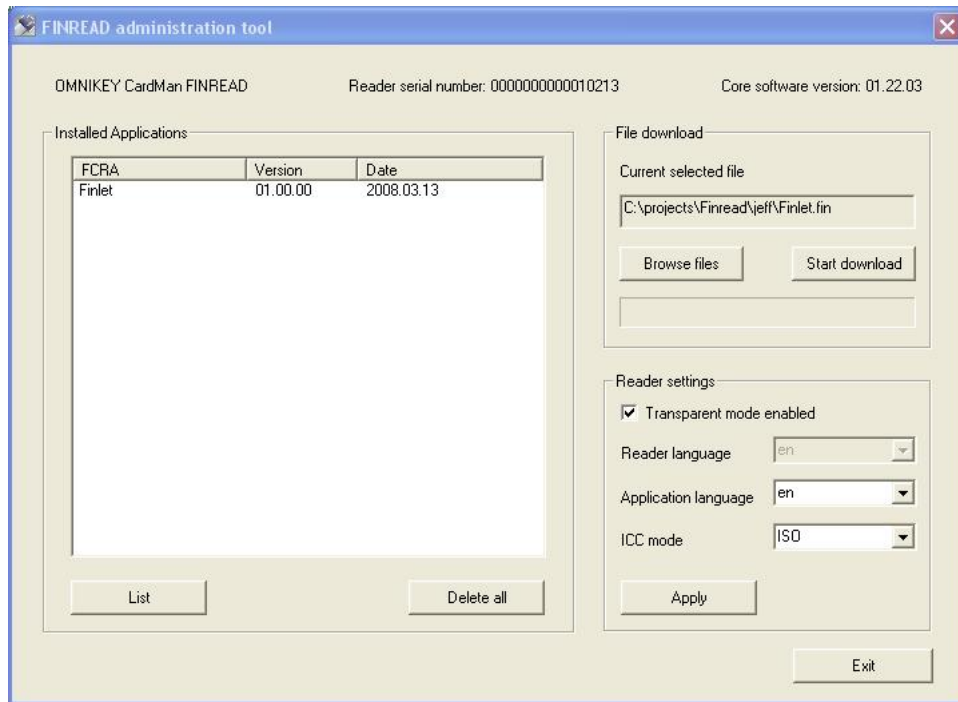


Abbildung 5.3: Administration Tool

Button *List* wird die Liste aktualisiert. Mit dem Button *Delete All* werden alle Finlets vom Kartenleser gelöscht. Sie können nicht einzeln gelöscht werden.

- **File Download:** um ein Finlet zu installieren, muss die *.fin* Datei einfach auf den Kartenleser heruntergeladen werden. Man benutzt den *Browse File* Button um die Datei auszuwählen. Mit *Start Download* wird die Datei heruntergeladen. Dieser Mechanismus wird nicht nur für Finlets benutzt. Man kann damit auch neue Schlüssel sowie eine neue Core Software auf den Kartenleser herunterladen. Die Signatur des heruntergeladenen Pakets wird auf dem Kartenleser überprüft. Es wird ebenfalls überprüft ob es sich um Schlüssel, Core Software oder eine Applikation handelt und der Inhalt des Paketes an entsprechender Stelle gespeichert.
- **Reader Settings:** Hier kann man verschiedene Einstellungen für den Kartenleser vornehmen. Eine kritische Einstellung ist das Zulassen des *Transparent Modes*. In diesem Betriebsmodus leitet der Kartenleser alle APDUs an die Chipkarte weiter. Das kann unter Umständen ein Sicherheitsrisiko darstellen.

Damit hätte man auf dem Kartenleser einen betriebsbereiten Finlet. Die Ausführung des Finlets wird von der Host Applikation aus gestartet.

5.4 Die Host Applikation

Für die Programmierung der Host Applikation beschreibt die FINREAD Spezifikation die so genannte Host API. Diese bietet die Möglichkeit an Befehle an den Kartenleser oder direkt an die Chipkarte zu senden. Von Omnikey wurde eine Implementierung dieser API bereitgestellt als DLL-Datei: die `Finread.dll`. Des Weiteren bietet Omnikey die Möglichkeit die Host Applikation gegen die Windows COM-Schnittstelle¹ zu programmieren. In diesem Kapitel wird nur auf die Erstellung von Host Applikationen, die die `Finread.dll` benutzen eingegangen. Es werden auch nur einige wichtige Funktionen der API vorgestellt. Die gesamte API wird in einem späteren Kapitel detailliert behandelt. Weil man mit einer DLL arbeitet, liegt es nahe die Host Applikation in C oder C++ zu implementieren.

In der Header Datei `Finread.h` sind die spezifischen Funktionen und Datentypen definiert. Finread **Funktionen** haben immer die Form `finread_funcname`, die **Datentypen** haben die Form `finred_TYPENAME`. Im Folgenden werden die Hauptmerkmale einer Host Applikation vorgeführt.

5.4.1 Finlet aktivieren

Um ein Finlet zu aktivieren, muss eine Reihe von Befehlen an den Kartenleser gesendet werden. Jede der Funktionen hat einen Rückgabewert vom Typ `finread_RESULT` (der eigentlich einem Integer entspricht). Ein Rückgabewert kleiner 0 signalisiert das Fehlschlagen der Funktion. Mit der Funktion der Host API `finread_getLastError()` kann der Fehlercode des zuletzt aufgetretenen Fehlers abgefragt werden.

```
finread_RESULT result;
finread_FCRA_HANDLE handle;
finread_ACTIVE_FCRA_HANDLE activeHandle;
int timeout = 60000; // Millisekunden

result = finread_open("OMNIKEY CardMan FINREAD 0");
handle = finread_getFCRAByName("Finletname");
activeHandle = finread_activateFCRA(handle, timeout);
```

Listing 5.3: Aktivieren eines Finlets durch die Host Applikation

Im obigen Code wird zunächst eine Verbindung zum Kartenleser hergestellt. Danach wird ein Handle für das gewünschte Finlet geholt. Dabei ist es wichtig, dass der Parameter der Funktion `finread_getFCRAByName()` genau dem Namen des Finlets entspricht, der bei der Erstellung gegeben wurde (siehe Kapitel Finlet erzeugen). Der Handle wird benutzt um das Finlet zu aktivieren.

¹COM bedeutet so viel wie Component Object Model. Es ist eine serviceorientierte Plattform für Windows, die die Kommunikation zwischen den Prozessen vereinfachen soll. Siehe auch: <http://www.microsoft.com/com/default.msp>

5.4.2 Ergebnisse vom Finlet entgegennehmen

Ein aktiviertes Finlet wird noch nicht ausgeführt. Um seine Ausführung anzustoßen, wird die folgende Funktion aufgerufen:

```
result = finread_sendDataToFCRA(activeHandle ,
                                commandID ,
                                sendData ,
                                sendDataLength ,
                                receiveData ,
                                & receiveDataLength ,
                                timeout);
```

Listing 5.4: Ausführung eines Finlets starten und Ergebnis entgegennehmen

Bedeutung der Parameter:

- `activeHandle`: Der zuvor erhaltene Handle.
- `commandID`: Wird dem Finlet zur Ausführungszeit zur Verfügung gestellt. Falls das Finlet mehrere Aufgaben erfüllen kann, kann die Command ID benutzt werden um eine Aufgabe auszuwählen.
- `sendData`: Ein Byte Array mit Daten, die dem Finlet übergeben werden. Dies können APDUs sein oder andere Daten, die für die Ausführung des Finlets wichtig sind.
- `sendDataLength`: Die Länge des Byte Arrays `sendData`
- `receiveData`: Das ist das Byte Array, in das das Ergebnis des Finlets geschrieben wird. Das Finlet sendet sein Ergebnis mit `clientProxy.notifyCompletion(COMPLETION_CODE, RETURN_DATA);` als Byte Array an die Host Applikation zurück.
- `&receiveDataLength`: Zeiger auf den Speicherbereich, in den die Länge des Byte Arrays `receiveData` geschrieben wird.
- `timeout`: Zeitangabe in Sekunden nach der die Ausführung des Finlets abgebrochen wird.

Die Funktion `finread_sendDataToFCRA(...)` kann mehrmals hintereinander aufgerufen werden z. B. mit verschiedenem Parameter `commandID`. Die Kommunikation zwischen Host API und Finlet wird durch die Core Software vermittelt.

5.5 Die Core Applikation

Die Core Software bildet unter anderem die Schnittstelle zwischen der Host Applikation und Finlet. Sie besteht aus den Treibern für die Komponente des Kartenlesers sowie herstellerepezifischen Routinen für das Zugreifen auf die Ressourcen des Kartenlesers. Die Core Software integriert auch das STIP Framework als API für Applikationen für den Kartenleser. Sie ist auch zuständig für das Aufrufen der verschiedenen Methoden des Finlets, sowie für die Weiterleitung des Ergebnisses der Finlet Ausführung an die Host Applikation.

5.6 Kontrollfluss während der Ausführung

Die folgende Graphik zeigt am Beispiel des von Omnikey bereitgestellten Demo-Finlets welche Komponente während der Ausführung eines Finlets beteiligt sind. Das DemoFinlet liest die ATR der Chipkarte und zeigt sie an.

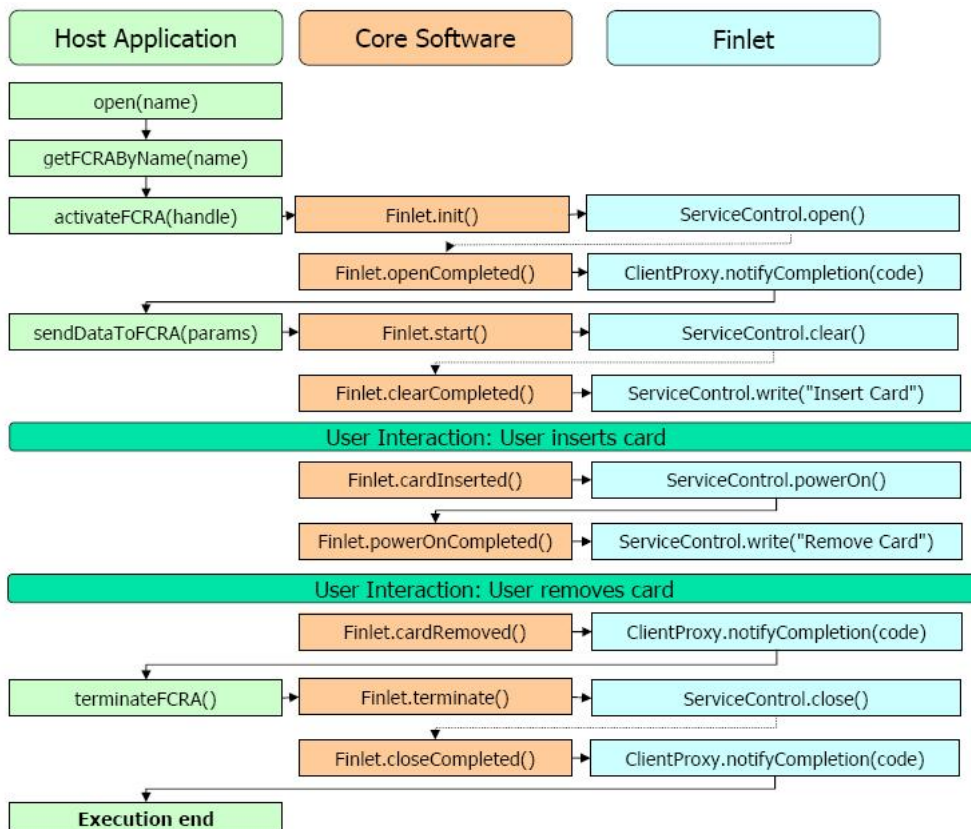


Abbildung 5.4: Kontrollfluss während der Ausführung eines Finlets

Anhand dieser Graphik ist zu erkennen wie die Hostapplikation und das Finlet aufgebaut sind und wie die Interaktion mit der Core Software stattfindet. Die Hostapplikation besteht aus einer Reihe von sequentiellen Befehlen. Nach bestimmten Befehlen ist es nötig auf Ergebnisse zu warten, die erst von dem Finlet auf dem Kartenleser berechnet werden. Die Befehle der Host Applikation werden von der Core Software empfangen. Wie in einem vorherigen Kapitel erwähnt, ist das STIP Framework ein Teil der Core Software und jedes Finlet muss bestimmte Interfaces implementieren. Auf Grund des empfangenen Befehls ruft die Core Software die entsprechenden Methoden des Finlets auf. Z. B. wenn die Host Applikation das Finlet aktiviert (`activateFCRA()`) ruft die Core Software die Methode `init()` des Finlets auf. Die Implementierung dieser Methode ruft eine Service Methode auf um das Smartcard Interface zu öffnen (`ServiceControl.open()` wobei es sich beim ServiceControl um den SCSCControl handelt). Nach dem Aufruf dieser Methode macht das Finlet erst ein Mal nichts mehr. Die Core Software versucht das Smartcard Interface zu

öffnen und wenn diese Aktion beendet ist, ruft die Core Software die Interface Methode `openCompleted()` auf. Das Finlet sollte dieses Interface implementieren damit es auf diese Ereignisse (Operation Events) reagieren kann. Die Implementierung der Methode `openCompleted()` prüft ob das Smartcard Interface erfolgreich geöffnet werden konnte und benachrichtigt die Host Applikation entsprechend (`ClientProxy.notifyCompletion()`). Für die Host Applikation bedeutet das, dass der letzte Befehl (`activateFCRA()`) ausgeführt wurde. Ob er auch erfolgreich war, d. h. ob das Finlet auch tatsächlich aktiviert wurde, kann anhand des Rückgabewertes ermittelt werden (größer 0 bedeutet erfolgreich). Der nächste Befehl der Host Applikation kann ausgeführt werden. Hier ist es `sendDataToFCRA()`. Die Auswirkungen dieses Befehls und die Interaktion mit der Core Software finden analog zum vorher beschriebenen Befehl statt mit dem Unterschied, dass nach dem der Benutzer aufgefordert wurde die Karte einzuführen die Ausführung in der Methode eines Status Events weitergeht. Das Finlet sollte auch das `StatusEvent` Interface implementieren. Der Befehl `activateFCRA()` der Host Applikation übergibt unter anderem auch ein Byte Array als Parameter in das das Finlet ein mögliches Ergebnis der Ausführung hinschreiben kann. Um die Ausführung des Finlets zu beenden, ruft die Host Applikation `terminate()` auf und kann nun mit dem Ergebnis des Finlets weiterarbeiten (in diesem Beispiel wird die ATR der Chipkarte angezeigt).

5.7 Debugging

Das Debuggen der Finlets ist im Allgemeinen eine aufwendige Angelegenheit. Zunächst sind die Fehlermeldungen der Core Software sehr beschränkt. Diese gibt bei fehlerhafter Ausführung Error Codes in hexadezimal Form an. Die Bedeutung dieser Codes muss dann in der Dokumentation bzw. im Javadoc der STIP API gesucht werden. Der Kartenleser von Omnikey bietet die Möglichkeit an zur Ausführung des Finlets mit `System.out.println()` Ausgaben in eine Datei auf dem Host PC zu erzeugen. Um das nutzen zu können, müssen unter Windows folgende Einstellungen an der Registry vorgenommen werden:

- Falls nicht vorhanden, muss unter `HKEY \LOCAL_MACHINE \System \CurrentControlSet \Control` ein neuer Schlüssel erstellt werden. Dieser Schlüssel muss `CardMan` heißen.
- Unter diesem Schlüssel wird ein neues DWORD mit dem Namen `LogToFile` und dem Wert `1` erzeugt.

Diese Möglichkeit die Ausgaben des Finlets auf dem PC zu lenken, ist eine der effektivsten Methoden um Fehler zu entdecken. Nach jeder Änderung des Codes des Finlets müssen alle in diesem Kapitel beschriebenen Schritte zur Finleterzeugung durchgeführt werden, d. h. Code kompilieren, Jeff erstellen, .fin Datei erstellen, fehlerhaftes Finlet vom Kartenleser löschen, neu erstelltes Finlet auf dem Kartenleser laden, Finlet starten und anschließend erneut debuggen.

Kapitel 6

Java Wrapper

Teil dieser Bachelorarbeit ist es als Proof-of-Concept Beispielapplikationen zu entwickeln. Um die Entwicklung der Host Applikationen und der Finlets einheitlich in Java schreiben zu können, wurde für die `Finread.dll` ein Java Wrapper geschrieben. Unter einem Wrapper versteht man das “Umhüllen” der DLL der Art, dass ihre Funktionen aus einem Java Programm heraus benutzt werden können. Das ist möglich durch die Java Native Interface (JNI), die es ermöglicht nativen Code (in diesem Fall C Code) von Java aus auszuführen.

6.1 Motivation

Bei der Entwicklung von Finlets mit der von Omnikey zur Verfügung gestellten SDK tritt das Problem auf, dass man zwei Programmiersprachen benutzen muss: Java für die Finlets und C bzw. C++ für die Host Applikation. Dieser Umstand macht das Entwickeln von Applikationen schwieriger für Entwickler denn sie müssen beide Programmiersprachen beherrschen. Um dieses Problem zu beheben wurde im Rahmen der Bachelorarbeit ein Wrapper für die `Finread.dll` programmiert, der es ermöglicht auch die Host Applikation in Java zu schreiben. Auch wenn dieser Wrapper für die `Finread.dll` von Omnikey geschrieben wurde, so ist er mit jeder anderen Implementierung der Host API (die als DLL vorliegt) nutzbar, denn jede Implementierung muss sich an die Vorgaben der FINREAD Spezifikation halten.

6.2 Plattform und Werkzeuge

Für die Programmierung des Java Wrappers für die `Finread.dll` wurden eingesetzt: **Kartenleser:**

- OMNIKEY 3821 USB Pinpad FINREAD Kartenleser

Software:

- Windows als Betriebssystem des Host-Computers
- Zugehörige Windows Treiber für den Kartenleser (PC/SC-Treiber)

- **Build Package:** Von OMNIKEY bereitgestelltes Programm zur Konvertierung der Finlets
- **Administration Tool:** Von OMNIKEY bereitgestelltes Programm zur Verwaltung des Kartenlesers
- **Finread.dll:** die Implementierung der FINREAD Host API von OMNIKEY

Chipkarte:

- Betriebssystem: TCOS 2.03
- 1024 Bits RSA Schlüssel
- Fähigkeit der digitalen Signaturerzeugung
- Fähigkeit der Verschlüsselung
- X.509 Zertifikat der TU-Darmstadt

6.3 JNI Grundlagen

Um den Java Wrapper für die `Finread.dll` zu realisieren, wurde JNI benutzt. JNI ist seit Java 1.1 das Java Interface zu nativem Code. Mit Hilfe von JNI kann man aus einem Java Programm aus Funktionen aufrufen, die in einer anderen Programmiersprache erstellt wurden und es ist möglich auf ihre Ergebnisse zuzugreifen. Das wird häufig benutzt um zeitkritischen Code in einer maschinennahen Sprache zu programmieren während die Hauptapplikation in Java programmiert ist. Es ist auch nützlich, wenn wie im vorliegenden Fall bestimmte Funktionalitäten nur als Bibliothek einer anderen Programmiersprache vorliegen diese aber in einer Java Applikation benutzt werden müssen. Das Java Native Interface bietet umfangreiche Funktionen an, um auf die Ergebnisse von nativem Code zuzugreifen oder im nativen Code auf Java Objekte zuzugreifen und Java Variablen zu erzeugen oder zu ändern. Für den Java Wrapper kommt nur ein Teil dieser Funktionalität zum Einsatz. Dieses Kapitel behandelt kurz nur die JNI Funktionen, die bei der Erstellung des Wrappers eine Bedeutung spielten. Für eine detaillierte Beschreibung dessen, was mit JNI möglich ist empfiehlt es sich die JNI Spezifikation zu lesen unter <http://java.sun.com/j2se/1.4.2/docs/guide/jni/>.

6.3.1 Vorgehen bei der Benutzung von JNI

Um mit JNI auf nativen Code zuzugreifen, müssen folgende Schritte durchgeführt werden:

1. Native Methoden in einer Java Klasse deklarieren
2. Header Datei für diese Methoden erstellen
3. Funktionen in C bzw. C++ implementieren und zu einer Bibliothek kompilieren

4. Erstellte Bibliothek in der Java Klasse mit den nativen Methoden einbinden

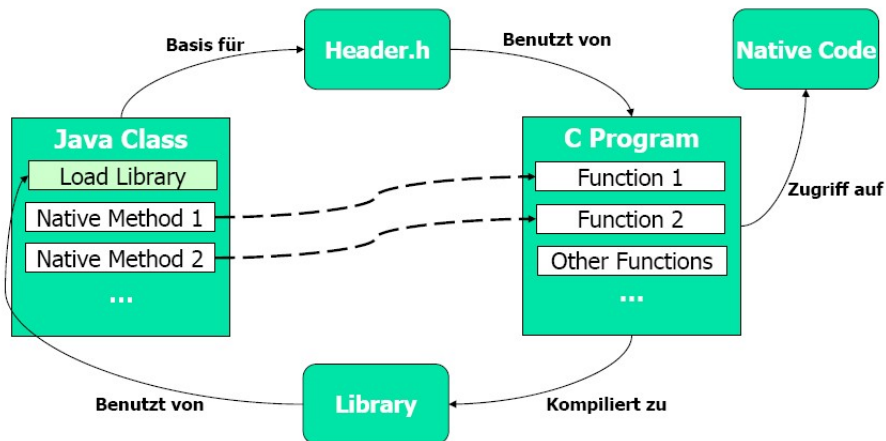


Abbildung 6.1: Java Native Interface - Vorgehen bei der Benutzung

Deklaration der nativen Methoden

Grundsätzlich definiert man die nativen Methoden in Java mit der besonderen Signatur:

```
public native returnType methodName (parameters);
```

Zu jeder so definierten Methode muss es eine native Implementierung geben.

Header Datei erstellen

Sofern man in einer Klasse native Methoden deklariert hat, kann man sich mit Hilfe von `javah` eine C bzw. C++ Header Datei erstellen.

```
cd c:\project\home
javah -classpath ./path/to/classes/*.class
      -o headerName.h package.name.class
```

Dieser Befehl erzeugt eine Header Datei mit dem Namen `headerName.h`. Diese Datei wird dann als Grundlage für die Implementierung der nativen Funktionen verwendet.

Native Implementierung der Java Methoden

Die gewünschte Funktionalität kann als native Funktion (z. B. in C oder C++) implementiert werden. Die implementierende Datei muss die zuvor erstellte Header Datei einbinden und die nativen Funktionen implementieren. Die implementierenden C bzw. C++ Funktionen haben eine besondere Signatur.

```
JNIEXPORT JNICALL Java_package_Hello_hello(JNIEnv *env, jobject o)
```

Dabei wird der Funktionsname `Java_package_Hello_hello` zusammengesetzt aus dem Namen der zugehörigen Javamethode (inklusive Paketname und Klassenname) mit dem vorangestellten `Java_`. Der erste Parameter der Funktion ist immer der Pointer auf das JNI Environment, das ein Handle des aktuellen Threads in der Java Virtual Machine darstellt. Über dieses Environment erfolgt die Kommunikation zwischen JVM und nativem Code. Die Implementierung der nativen Methoden kann zu einer Bibliothek kompiliert werden, die von der Java Klasse, die die nativen Methoden deklariert eingebunden werden muss.

Benutzung der nativen Methoden

Bevor man die nativen Methoden benutzen kann, muss die Klasse, die die nativen Methoden deklariert die zuvor erstellte DLL einbinden. Jetzt können die nativen Methoden wie gewöhnliche Java Methoden benutzt werden. Bei der weiteren Programmierung wird man den Unterschied zwischen lokalem und nativem Methodenaufruf nicht mehr merken.

Ein Beispiel

Anhand eines kurzen HelloWorld Beispiels soll die grundsätzliche Benutzung von JNI verdeutlicht werden.

- Zunächst wird in einer Java Klasse die native Methode deklariert. Mit `System.loadLibrary("libName");` wird die Bibliothek eingebunden, die die Implementierung der nativen Methode beinhaltet.

```
public class Hello{
    System.loadLibrary("libName");
    public native void hello();
}
```

- Als nächstes wird die Header Datei mit `javah` erstellt

```
javah -classpath ./path/to/classes/hello.class
      -o hello.h package.name.Hello
```

- Native Implementierung der nativen Methoden bindet die zuvor erstellte Header Datei ein

```
#include <jni.h>
#include <stdlib.h>
#include "hello.h"

JNIEXPORT JNICALL Java_package_Hello_hello(JNIEnv *env,
                                             jobject o){
    printf("C is saying HELLO\n");
}
```

- Zu DLL (libName.dll) kompilieren und ins Java System.library.path kopieren (damit es mit `System.loadLibrary("libName");` von der Java Klasse eingebunden werden kann).
- Aufruf: `(new Hello()).hello();`

6.4 Die Funktionen der Host API

Alle Funktionen der `Finread.dll`, die in der FINREAD Spezifikation beschrieben werden, werden auch gewrappt. Im Folgenden wird eine Liste aller dieser Funktionen präsentiert mit ihren Parametern und Rückgabewerten. Diese Liste wurde auf Basis der Angaben in der CWA 8 ([11]) erstellt, die sich auch als ergänzende Literatur empfiehlt.

1. `finread_open(const char* readername)`

- Stellt die Verbindung zu dem genannten Kartenleser her.
- Rückgabewert: `finread_RESULT`

2. `finread_close()`

- Beendet eine bestehende Verbindung zu einem Kartenleser
- Rückgabewert: `finread_RESULT`

3. `finread_setICCMode(finread_INT mode)`

- Stellt den Modus ein, der bestimmt wie die Daten mit dem Befehl `finread_sendDataToICC()` an die Chipkarte gesendet werden. Mögliche Werte sind *EMV* oder *ISO*. Diese werden als `finread_INT` kodiert.
- Rückgabewert: `finread_RESULT`

4. `finread_downloadFile(const char* path)`

- Lädt eine Datei (Finlet, Schlüssel oder neue Core Software) herunter auf den Kartenleser.
- Rückgabewert: `finread_RESULT`

5. `finread_locateFirstFCRA()`

- Lokalisiert die Position des ersten Finlets in der Liste der Finlets des Kartenlesers.
- Rückgabewert: `finread_POSITION`

6. `finread_getFCRAByPosition(finread_POSITION position)`

- Stellt die Verbindung zu dem Finlet an der gegebenen Position her und gibt einen Handle zurück.
- Rückgabewert: `finread_HANDLE`

7. `finread_getFCRAByName(const char* name)`

- Stellt die Verbindung zu dem Finlet mit dem gegebenen Namen her und gibt einen Handle zurück. Es ist zu beachten, dass es möglich ist, dass sich mehrere Finlets mit dem gleichen Namen auf dem Kartenleser befinden können. Dann wird ein Handle auf das Finlet mit der kleinsten Position zurückgegeben.
- Rückgabewert: `finread_FCRA_HANDLE`

8. `finread_getFCRAByID(finread_ID id)`

- Stellt die Verbindung zu dem Finlet mit der gegebenen ID her und gibt einen Handle zurück. Diese ID ist eine Unique ID, die ein Finlet eindeutig identifiziert.
- Rückgabewert: `finread_FCRA_HANDLE`

9. `finread_getFCRAProperties(finread_FCRA_HANDLE handle, finread_FCRAProperties* properties)`

- Schreibt die Eigenschaften des durch den Handle bestimmten Finlets in die Datenstruktur `properties`.
- Rückgabewert: `finread_RESULT`

10. `finread_getReaderProperties(finread_FCRProperties* properties)`

- Schreibt die Eigenschaften des Kartenlesers in die Datenstruktur `properties`.
- Rückgabewert: `finread_RESULT`

11. `finread_getReaderStatus(finread_FCRStatus* status)`

- Schreibt den Status des Kartenlesers in die Datenstruktur `status`.
- Rückgabewert: `finread_RESULT`

12. `finread_activateFCRA(finread_FCRA_HANDLE handle, finread_INT timeout)`

- Aktiviert das Finlet, das von dem Handle identifiziert wird.
- Rückgabewert: `finread_ACTIVE_FCRA_HANDLE`

13. `finread_sendDataToFCRA(finread_ACTIVE_FCRA_HANDLE handle, finread_CoMMAND_ID commandID, const finread_BYTE* sendData, finread_INT sendDataLength, finread_BYTE* responseData, finread_INT* responseDataLength, finread_INT timeout)`

- Sendet die Daten als Byte Array an das Finlet und übergibt auch die Adresse, an welche das Ergebnis als Byte Array geschrieben werden soll. Der Wert für Timeout gibt an nach welcher Zeit (in Millisekunden) die Aktion abgebrochen wird.
- Rückgabewert: `finread_RESULT`

14. `finread_sendDataToICC(finread_COMMAND_ID commandID,
const finread_BYTE* sendData,
finread_INT sendDataLength,
finread_BYTE* responseData,
finread_INT* responseDataLength,
finread_INT timeout)`

- Sendet die Daten als Byte Array an die Chipkarte und übergibt auch die Adresse, an welche das Ergebnis als Byte Array geschrieben werden soll. Der Wert für Timeout gibt an nach welcher Zeit (in Millisekunden) die Aktion abgebrochen wird.
- Rückgabewert: `finread_RESULT`

15. `finread_cancelFCRA(finread_FCRA_HANDLE handle,
finread_INT timeout)`

- Veranlasst, dass die `cancel()` Methode des Finlets aufgerufen wird, das durch den Handle identifiziert wird.
- Rückgabewert: `finread_RESULT`

16. `finread_terminateFCRA(finread_FCRA_HANDLE handle,
finread_INT timeout)`

- Veranlasst, dass die `terminate()` Methode des Finlets aufgerufen wird, das durch den Handle identifiziert wird.
- Rückgabewert: `finread_RESULT`

17. `finread_abortFCRA(finread_FCRA_HANDLE handle,
finread_INT timeout)`

- Veranlasst, dass die `abort()` Methode des Finlets aufgerufen wird, das durch den Handle identifiziert wird.
- Rückgabewert: `finread_RESULT`

18. `finread_getLastError()`

- Gibt den Code des zuletzt aufgetretenen Fehlers zurück.
- Rückgabewert: `finread_RESULT_CODE`

6.5 Design und Implementierung

6.5.1 Annahmen und Abhängigkeiten

Der Wrapper soll es ermöglichen die `Finread.dll` von Java aus zu benutzen. Dabei sollten die Namen der Funktionen so erhalten bleiben wie sie in der FINREAD Spezifikation beschrieben sind. D. h., dass jede Funktion der `Finread.dll` mit dem Namen `finread_funcName` eine entsprechende Methode im Java Wrapper hat mit dem gleichen Namen. Die Finread Bibliothek besitzt auch eine Reihe von selbst definierten Datentypen und Datenstrukturen. Diese sollten in der Java Version der `Finread.dll` ebenfalls möglichst originalgetreu erhalten bleiben. Bei den Datentypen handelt es sich zum größten Teil um umbenannte C-Datentypen. So ist z. B. ein `finread_INT` nichts anderes als ein gewöhnliches C-Integer. Für diese einfachen Datentypen wird in Java kein besonderes Mapping stattfinden sondern es wird der nahe liegende primitive Java Datentyp benutzt. Z. B. wird für `finread_INT` ein Java `int` benutzt. Die Finread Datenstrukturen stellen aber komplexere Ansammlungen von Daten dar. Es gibt z. B. eine Datenstruktur für die Eigenschaften des Kartenlesers wo die ID, Angaben zum Hersteller, usw. gespeichert werden können. Für diese Datenstrukturen findet ein Mapping in Java statt. Die komplexen Datenstrukturen werden in Java als `HashMap<String, Object>` dargestellt wobei der Schlüsselstring den Vorgaben der FINREAD Spezifikation für die C Datenstrukturen entspricht. Folgende Tabelle stellt die C Datenstrukturen der `Finread.dll` ihren entsprechenden Java Mappings gegenüber.

C Data Structure: <code>finread_FCRProperties</code>	
<pre>struct finread_FCRProperties{ finread_SUPPLIER_ID vendorId; finread_READER_ID serialNo; finread_VERSION FinreadSpecificationVersion; finread_VERSION coreSWVersion; finread_ISO639 preferredApplicationLanguage; finread_ISO639 coreLanguage; finread_SupportedLanguages* supportedLanguages; finread_KeyInfo* keyInfo; finread_FCRStatus* status }</pre>	
Java Mapping	
<pre>HashMap<String, Object> FCRProperties "vendorId" => Integer "serialNo" => Integer "FinreadSpecificationVersion" => Integer "coreSWVersion" => Integer "preferredApplicationLanguage" => String "coreLanguage" => String "supportedLanguages" => HashMap<String, Object> "keyInfo" => HashMap<String, Object></pre>	

"status"	=> HashMap<String, Object>
C Data Structure: finread_KeyInfo	
<pre>struct finread_KeyInfo{ finread_KeyData masterKey; finread_KeyData applicationKey; finread_KeyData coreKey; finread_KeyData authenticationKey }</pre>	
Java Mapping	
<pre>HashMap<String, Object> KeyInfo "masterKey" => HashMap<String, Object> "applicationKey" => HashMap<String, Object> "coreKey" => HashMap<String, Object> "authenticationKey" => HashMap<String, Object></pre>	
C Data Structure: finread_KeyData	
<pre>struct finread_KeyData{ finread_STR_SIZE keyIdLen; char* keyId; finread_BYTE keySerialNumber[4]; finread_DATE expirationDate }</pre>	
Java Mapping	
<pre>HashMap<String, Object> KeyData "keyIdLen" => Integer "keyId" => String "keySerialNumber" => byte[4] "expirationDate" => byte[]</pre>	
C Data Structure: finread_FCRStatus	
<pre>struct finread_FCRStatus{ finread_BYTE transparentEnabled; finread_BYTE readerMode; finread_ID activeFCRAId }</pre>	
Java Mapping	
<pre>HashMap<String, Object> FCRStatus "transparentEnabled" => Boolean "readerMode" => String "activeFCRAId" => byte[]</pre>	
C Data Structure: finread_FCRAProperties	
<pre>struct finread_FCRAProperties{ finread_SUPPLIER_ID supplierId; finread_DATE FCRAreleaseDate; finread_BYTE status; finread_PERSISTENT_STORAGE_SIZE persistentStorageSize; finread_STR_SIZE FCRAnameSize; char* FCRAname; }</pre>	

<pre> finread_ID FCRAid; finread_VERSION FCRAversion; finread_ID FCRIAid; finread_VERSION FCRIAversion } </pre>
Java Mapping
<pre> HashMap<String, Object> FCRAProperties "supplierId" => byte [] "FCRAreleaseDate" => byte [] "status" => byte "persistentStorageSize" => Integer "FCRAnameSize" => Integer "FCRAname" => String "FCRAid" => byte [] "FCRAversion" => byte [] "FCRIAid" => byte [] "FCRIAversion" => byte [] </pre>
C Data Structure: finread_SupportedLanguages
<pre> struct finread_SupportedLanguages{ finread_BYTE numLanguages finread_IS0639* languages } </pre>
Java Mapping
<pre> HashMap<String, Object> SupportedLanguages "numLanguages" => Integer "readerMode" => String[] </pre>

Um die Spezifikation möglichst getreu in Java umzusetzen, werden für die Methoden auch die entsprechenden Parameter gewählt. Wenn es sich bei den Parametern um C Structures handelt, so wird in Java die entsprechende Datenstruktur als Parameter der Methode übergeben. Z. B. für die C Funktion

```
finread_getReaderProperties(finread_FCRProperties properties)
```

wird die entsprechende Java Methode lauten:

```
finread_getReaderProperties(HashMap<String, Object> properties)
```

Auch die Rückgabewerte der Methoden entsprechen der FINREAD Spezifikation.

6.5.2 Besonderheiten der Implementierung

Zu jeder nativen Methode in Java wird eine entsprechende Funktion in C erstellt. Es ist eine JNI Naming Convention, dass der Name der C-Funktion sich aus dem voll qualifizierten Namen der Java-Methode zusammensetzt. D. h. für die Methode *method* aus der Klasse *Class* aus dem Paket *package.name* wird eine C-Funktion erstellt mit dem Namen:

JAVA_package_name_class_method

Das ergibt einen Konflikt, wenn man für die Java Methoden die Namen aus der FINREAD Spezifikation beibehalten möchte, denn diese beinhalten bereits einen Underscore (*finread_functionName*). Dieses Problem wurde durch Indirektion gelöst, d. h. gewöhnliche Java-Methoden, die die erforderliche Nomenklatur erfüllen, rufen die nativen Methoden auf. Diese können dann umbenannt werden, so dass ihre Namen den JNI Anforderungen genügen.

```
public class FinreadDLLWrapper{
    // Spezifikationskonforme Deklaration
    public int finread_open(String readername){
        finreadOpen(readername);
    }
    // JNI konforme Deklaration der nativen Methode
    private native int finreadOpen(String rn);
}
```

6.5.3 Die Komponenten des Wrappers

Der Wrapper bietet die gleiche API wie die FINREAD Spezifikation sie beschreibt um Host Applikationen zu schreiben. Insgesamt besteht der Wrapper aus den Java Klassen `NativeFunctions` und `FinreadDLLWrapper` sowie aus der C Implementierung der nativen Methoden `NativeFinreadWrapper`.

NativeFunctions

Diese Klasse enthält die Deklarationen der nativen Methoden. Die Bibliothek mit den Implementierungen der nativen Methoden `NativeFinreadWrapper.dll` muss statisch eingebunden werden. Die Nomenklatur der nativen Methoden wurde den JNI Naming Conventions angepasst. Damit die FINREAD konforme API auch in dem Java Wrapper erhalten werden kann wurden in der Klasse `FinreadDLLWrapper` FINREAD Host API konforme Methoden deklariert.

FinreadDLLWrapper

Die Methoden dieser Klasse bilden die FINREAD API möglichst genau nach (siehe Abschnitt 6.5.1). Um die nativen Funktionen zu benutzen, wird eine Instanz von `NativeFunctions` erstellt. Die Methoden, die Daten an den Kartenleser senden (z. B. `finread_sendDataToFCRA()`) oder Daten von dem Kartenleser erwarten (z. B. `finread_getReaderProperties()`), müssen die Datenstrukturen dafür vorbereiten. Diese Datenstrukturen sind in der Regel Arrays von Bytes. Diese Datenstrukturen werden den nativen Methoden als Parameter übergeben.

NativeFinreadWrapper

Der `NativeFinreadWrapper` ist die C Implementierung der nativen Java Methoden. Diese Datei bindet die mit `javah` erstellte `NativeFinreadWrapper.h` Hea-

der Datei ein. Um mit dem Kartenleser zu kommunizieren, wird die `Finread.dll` benutzt, die von Omnikey zur Verfügung gestellt wurde. Das folgende Beispiel soll verdeutlichen wie eine Funktion die Parameter aus der Java Umgebung übernimmt, die entsprechende Funktion der `Finread.dll` aufruft und das Ergebnis in die Java Umgebung zurücksendet.

- Die JNI konforme Deklaration der C Funktion.

```
JNIEXPORT jint JNICALL
Java_wrapper_NativeFunctions_finreadSendDataToFCRA
(JNIEnv *env, jobject o, jint handle, jint commandID, jbyteArray
 data,
 jint dataLength, jbyteArray response, jint responseLength,
 jint timeout){
```

Listing 6.1: Deklaration der nativen Funktion

- Hier wird ein Byte Array in C definiert. Über das Environment `*env` werden die Daten aus den Parametern `data` und `response` aus dem Java Runtime Environment in die lokalen Variablen `_data` und `_response` kopiert.

```
BYTE* _data = (*env)->GetByteArrayElements(env, data, NULL);
BYTE* _response = (*env)->GetByteArrayElements(env, response,
 NULL);
```

Listing 6.2: Übernahme der Parameter aus dem Java Environment in lokale Variablen

- Die entsprechende Funktion der `Finread.dll` wird aufgerufen.

```
finread_RESULT result;
int _responseLength = (int)responseLength;
result = finread_sendDataToFCRA(handle, commandID, _data,
 dataLength, _response,
 &_responseLength, timeout);
```

Listing 6.3: Funktion der `Finread.dll` aufrufen

- Es wird überprüft ob die Ausführung erfolgreich war.

```
if(!result){
    return result;
}
```

Listing 6.4: Ergebnis Prüfen

- Die aufgerufene Funktion hat ihr Ergebnis (Byte Array) in `_response` geschrieben. Der Inhalt dieses Arrays wird mit `ReleaseByteArrayElements` in das Java Runtime Environment zurück kopiert.

```
(*env)->ReleaseByteArrayElements(env, data, _data, 0);
(*env)->ReleaseByteArrayElements(env, response, _response, 0);
```

Listing 6.5: Rückgabe des Ergebnisses in das Java Environment

- Falls die Ausführung erfolgreich war, gibt die Funktion die Länge der Response Byte Array zurück.

```
return _responseLength;
}
```

Listing 6.6: Rückgabe des FINREAD konformen Wertes

Dieses Beispiel soll exemplarisch die Implementierung der nativen Funktionen zeigen. Alle anderen Funktionen wurden nach dem gleichen Muster wie dieses Beispiel implementiert.

6.6 Benutzung des Wrappers

Um den Java Wrapper zu benutzen, müssen die dynamischen Bibliotheken `Finread.dll` und `NativeFinreadWrapper.dll` vorhanden sein. Außerdem muss sich die `NativeFinreadWrapper.dll` im `System.java.path` befinden damit sie von der zugehörigen `FinreadDLLWrapper` geladen werden kann. Die `Finread.dll` wird von der Implementierung der nativen Funktionen benutzt und muss sich im Suchpfad des Systems befinden (also z. B. unter `C:\Windows\System`).

Um die Erstellung der Host Applikationen für den FINREAD Kartenleser zu erleichtern, wurde das Façade Pattern auf den zuvor beschriebenen Wrapper angewendet. Der Façade Pattern versteckt die Komplexität eines Subsystems und bietet eine einfachere Schnittstelle für die Programmierung. In diesem Fall bietet der `EasyWrapper` einige Methoden an, die intuitiv zu benutzen sind und die von der überflüssigen Anreihung von einfachen Finread Befehlen abstrahiert.

6.6.1 FINREAD konforme API

Um eine Host Applikation zu schreiben, kann auf die FINREAD konforme API zugegriffen werden, die die Java Implementierung der `Finread.dll` anbietet. Man kann dann die Java Applikation analog zu der in C programmierten Host Applikation gestalten, die im Kapitel “Entwicklung von Applikationen für den FINREAD Kartenleser” vorgestellt wurde.

```
public static void main(String[] args){
    FinreadDLLWrapper wrapper = new FinreadDLLWrapper();
    int result = wrapper.finread_open("readerName");
    if(result <= 0) {
        int errorCode = wrapper.finread_getLastError();
        if(errorCode == ...){
            // do something
        }else if(errorCode == ...){
            // do something
        }
    }
}
```

```
        return; // Reader could not be opened
    }
    // else reader has been opened
    result = wrapper.finread_getReaderProperties();
    // Further Method Body
```

Listing 6.7: Host Applikation mit dem Standard Java Wrapper

Eine solche Applikation nutzt alle Besonderheiten der FINREAD Spezifikation aus, muss aber auf der anderen Seite auch mit allen Unzulänglichkeiten einer primitiven API ‘kämpfen’. Z. B. muss nach jedem Aufruf einer `finread_function` das Ergebnis überprüft werden: ein negativer Wert bedeutet, dass die Ausführung nicht erfolgreich war. In diesem Fall könnte man den letzten Error Code abfragen und durch Vergleich mit erwarteten Error Codes entscheiden wie die Ausführung weitergeht.

6.6.2 Façade API

Um die Vorzüge von Java bei der Erstellung von Host Applikationen zu nutzen, wurde für die FINREAD API eine weitere Abstraktionsebene geschaffen. Der `EasyWrapper` bietet die Möglichkeit eine Host Applikation mit wenigen Methodenaufrufe und etwas Konfiguration zu erstellen. Wenn der Name des Kartenlesers in einer Konfigurationsdatei gespeichert wurde kann das Herstellen der Verbindung zum Kartenleser mit nur einem Methodenaufruf erfolgen. Durch Überladen von Methoden kann auch die Ausführung eines `Finlets` mit nur einem Methodenaufruf erfolgen. Man muss dementsprechend die Parameter dieser Methode setzen wie Name des Kartenlesers, Name des `Finlets`, usw. Die Methoden des `EasyWrappers` werfen im Falle einer fehlerhaften Ausführung *Finread Exceptions*.

Der `EasyWrapper` erleichtert die Erstellung der FINREAD Host Applikation und gibt verständlichen Feedback im Falle dass die Ausführung misslungen ist. Eine detaillierte Übersicht über die Methoden des `EasyWrappers` gibt auch die Javadoc des Java Wrappers.

Kapitel 7

Beispiel Applikationen

Als Proof-of-Concept wurden einige Applikationen entwickelt. Jede Applikation besteht sowohl aus einer Host Applikation als auch aus einer FCRA (einem Finlet). Für die Host Applikation wird jedes Mal der Java Wrapper benutzt. Sie besteht in der Regel aus immer der gleichen einfachen Folge von Methodenaufrufen:

```
finread_open(readerName);
finread_getFCRAByName(fcraName);
finread_activateFCRA(handle);
finread_sendDataToFCRA(...);
finread_terminateFCRA();
finread_close();
```

und der Überprüfung der korrekten Ausführung der Methoden. Zusätzlich werden die Ergebnisse der Aufrufe ausgegeben. In diesem Kapitel werden die Host Applikationen nicht näher betrachtet. Es wird nur auf die Finlets eingegangen. Es wurden folgende vier Finlets entwickelt:

- **GetATRFinlet:** Analog zu dem Demofinlet von Omnikey wurde das GetATRFinlet entwickelt. Nachdem die Karte eingeführt wurde, wird ein Reset der Karte durchgeführt. Die abgelesene ATR wird an die Host Applikation zurück gesendet.
- **VerifyPINFinlet:** Ein Finlet das den Benutzer auffordert die PIN für die Chipkarte einzugeben. Die eingegebene PIN wird aufbereitet und verifiziert. Entsprechend dem Ergebnis der Verifikation wird eine Ausgabe erfolgen. Es ist wichtig, dass die PIN nicht dreimal hintereinander falsch eingegeben wird, da sonst die Karte gesperrt wird.
- **GetCertFinlet:** Dieses Finlet fordert das auf der Chipkarte befindliche Zertifikat an. Das X.509 Zertifikat wird als Byte Array an die Host Applikation übertragen.
- **SignFinlet:** Dieses Finlet sendet an die Chipkarte eine APDU, die die Aufforderung enthält ein Byte Array von Daten zu signieren. Nach Verifikation der PIN werden die Daten signiert und die Signatur wird an die Host Applikation zurück gesendet.

Um den Einsatz des Kartenlesers im Internetbereich exemplarisch zu zeigen, wurde eine Hostapplikation als ein Java Applet entwickelt der aus einem Webbrowser aus agiert und die Befehle an den Kartenleser schickt. Diese Web Applikation wurde im Zusammenhang mit dem “GetATRFInlet” benutzt.

Die Finlets benutzen einige Komponente, die Chipkarten spezifisch sind, wie z. B. die APDUs, die an die Karte geschickt werden, das Protokoll das benutzt wird um mit der Karte zu kommunizieren (synchron, asynchron) oder der Betriebsmodus der Karte (ISO, EMV). Deshalb sind die im Rahmen dieser Bachelorarbeit entwickelten Finlets für die verwendete Chipkarte konzipiert und nicht unbedingt mit anderen Karten kompatibel. Sie können jedoch durch geringe Modifikationen auch mit anderen Chipkarten benutzbar gemacht werden.

7.1 Web Applikation - ATR im Browser anzeigen

Bei dieser Beispielapplikation wird die Host Applikation in ein Java Applet integriert. Sie aktiviert das Finlet auf dem Kartenleser aus einem Webbrowser heraus. Das Finlet liest die ATR der Chipkarte aus und sendet sie an die Host Applikation zurück. Die ATR wird im Browser angezeigt.

Um diese Applikation auszuführen, ist ein Webserver nötig. Für diese Bachelorarbeit wurde zu Testzwecken XAMPP installiert. XAMPP bietet unter Windows unter anderem einen leicht konfigurierbaren Webserver mit SSL Modul an. Für die Webapplikation wurde eine Webseite erstellt, in der ein signiertes Java Applet eingebettet ist. Das Java Applet muss deshalb signiert werden, weil es Zugriff auf das Datei-System braucht, nämlich um die `Finread.dll` benutzen zu können. Die `Finread.dll` sowie die DLL Datei mit der Implementierung des Java Wrappers müssen sich im lokalen Suchpfad befinden damit sie von dem Applet benutzt werden können. Der Webbrowser stellt eine SSL Verbindung zum Webserver her und lädt die Webseite mit dem Applet. (Bei der Herstellung der SSL Verbindung könnte mit Hilfe eines Zertifikats auf der Chipkarte eine signierte SSL Verbindung aufgebaut werden.) Beim Laden der Seite wird die Hostapplikation aktiviert. Diese aktiviert ihrerseits das “GetATRFInlet”, das sich auf dem Kartenleser befinden muss.

Das “GetATRFInlet” ist so aufgebaut wie im Kapitel 4 beschrieben. Beim aktivieren des Finlets wird ein Reset der Karte durchgeführt, die daraufhin mit der ATR antwortet. Diese ATR wird an die Hostapplikation übertragen und im Webbrowser angezeigt.

Die Webapplikation soll zeigen wie der FINREAD Kartenleser in Internetanwendungen eingesetzt werden kann. Dieser Einsatz kann ein höheres Maß an Sicherheit darstellen z. B. bei Online Geldtransaktionen, denn der Kartenleser bietet eine sichere Plattform zur Ausführung sicherheitskritischen Codes. Um einem solchen Szenario zu genügen, müsste diese Beispielapplikation noch um eine signierte SSL Verbindung erweitert werden und natürlich für den Kartenleser

um die entsprechenden Finlets. In dem Artikel [12] wird die Motivation eines solchen Szenarios und das Design detailliert beschrieben.

7.2 GetCertFinlet

Weil die Finlets alle nach dem gleichen Muster implementiert sind, wird in diesem Abschnitt exemplarisch ein Finlet vorgestellt. Der Unterschied zwischen den verschiedenen Finlets besteht hauptsächlich darin, dass sie unterschiedliche APDUs an die Chipkarte schicken. Die Finlets erfüllen die asynchrone Architektur des STIP Frameworks, d. h. jedes Finlet implementiert das STIP Interface sowie alle nötigen Event und Status Interfaces. Der allgemeine Aufbau einer Applikation für den Kartenleser wurde bereits im Kapitel “Entwicklung von Applikationen für den Finread Kartenleser” besprochen.

```
public class GetCertFinlet implements Stiplet,
    SCSOperationListener, SCSStatusListener, UIOperationListener
{
}
```

Listing 7.1: Signatur der Finlet Klasse

- Stiplet: Das Interface, das von jeder STIP Applikation implementiert werden muss. Bietet Methoden an wie `init()`, `start()` oder `terminate()` um die Ausführung der Finlets zu steuern.
- SCSOperationListener: Listener, der nach der Beendigung von Operationen des Smartcard Interfaces aufgerufen wird. Z. B. `openCompleted()` nachdem `open()` vom SCSControl aufgerufen wurde.
- SCSStatusListener: Listener, der bei Änderungen des Status des Smartcard Interfaces aufgerufen wird. Eine solche Statusänderung ist zum Beispiel die Einführung der Smartcard in das Slot: `cardInserted()`-Event
- UIOperationListener: Listener, der nach der Beendigung der Operationen aufgerufen wird, die vom UIControl Services ausgeführt werden. So wird z. B. nach dem Beendigung von `write()`, die etwas auf das Display des Kartenlesers schreibt die Listener Methode `writeCompleted()` aufgerufen.

Das “GetCertFinlet” liest das X.509 Zertifikat von der Chipkarte und überträgt es an die Hostapplikation als Byte Array. Wie bereits beschrieben, holt sich das Finlet in der Initialisierungsphase den AccessManager und den ClientProxy, die es dazu braucht um Services nutzen zu können bzw. um mit der Hostapplikation zu kommunizieren. Die `init()` Methode wird aufgerufen wenn die Hostapplikation das Finlet aktiviert, also wenn diese die `finread_activateFCRA()` Methode aufruft.

```
public void init(AccessManager accessManager, ClientProxy
    clientProxy, byte[] obj) {
    this.accessManager = accessManager;
    this.clientProxy = clientProxy;
```

```
try {
    ui = (UIControl) accessManager.getServiceControl(UI);
    scs = (SCSControl) accessManager.getServiceControl(SCS
    );

    ui.addOperationListener(this);
    scs.addOperationListener(this);

    openState = OPEN_UI;
    ui.open(UI_SLOT);
} catch (Exception e) {
    debug("Exception during init: " + e);
    clientProxy.notifyCompletion(ClientProxy.
    COMPLETION_ERROR, null);
}
}
```

Listing 7.2: Implementierung der `init()` Methode des STIP Interfaces

Bei der Initialisierungsphase werden auch die `OperationListener` registriert für die `Smartcard-` sowie die `UserInterface-Services`. Anschließend wird die Methode `open()` des `User Interfaces` aufgerufen. Sowohl `UserInterface` als auch `Smartcardslot` müssen vor ihrer Benutzung "geöffnet" werden. Die Methode `init()` endet mit diesem Aufruf der `open()` Methode. Für die weitere Ausführung wartet das `Finlet` auf "Benachrichtigung" über die Beendigung der zuletzt angestoßenen `Service Methode`, also auf den Aufruf von `openCompleted()` durch die `Core Applikation`. Beim Aufruf von `openCompleted()` wird auch das Öffnen des `Smartcardslots` initiiert. Nach Beendigung wird die `Hostapplikation` benachrichtigt. Die Initialisierungsphase ist damit beendet. Bei erfolgreicher Initialisierung wird die `Hostapplikation` `finread_sendDataToFCRA()` aufrufen. Dadurch ruft das `Framework` die `start()` Methode des `Finlets` auf.

```
public void start(int i, byte[] obj) {
    try {
        scs.addStatusListener(this);
        if (scs.getPowerState() == SCSControl.OFF) {
            scs.powerOn();
        } else if (scs.getPowerState() == SCSControl.ON) {
            scs.powerOff();
            scs.powerOn();
        } else {
            displayText = " GetCertFinlet\n\n Insert card
            please";
            ui.clear();
        }
    } catch (Exception e) {
        clientProxy.notifyCompletion(ClientProxy.
        COMPLETION_ERROR, null);
    }
}
```

Listing 7.3: Implementierung der `start()` Methode des STIP Interfaces

Zunächst wird überprüft ob die Karte sich bereits im Kartenleser befindet und eingeschaltet ist. Wenn sich die Karte nicht im Kartenleser befindet wird der Benutzer aufgefordert diese einzuführen. Die Ausführung der Applikation

würde also mit `clear()->clearCompleted()->write()->writeCompleted()->powerOn()` weitergehen. Wenn die Karte bereits eingeführt ist wird direkt mit `powerOn()` weitergemacht.

```
public void powerOnCompleted(SCSOperationEvent event) {
    if (event.getStatus() != OperationEvent.EVT_SUCCESS) {
        clientProxy.notifyCompletion(ClientProxy.
            COMPLETION_ERROR, null);
    } else {
        // Power on succeeded
        try {
            scs.sendData(APDUS.SELECT_MASTER_FILE, 0, APDUS.
                SELECT_MASTER_FILE.length);
        } catch (Exception e) {
            debug("Exception during powerOnCompleted: " + e);
            clientProxy.notifyCompletion(ClientProxy.
                COMPLETION_ERROR, null);
        }
    }
}
```

Listing 7.4: Implementierung der `powerOn()` Methode des `OperationEvent` Listeners

Um an das Zertifikat auf der Karte zu kommen, muss auf dem Dateisystem der Chipkarte in das richtige Verzeichnis gewechselt werden. Dazu werden mehrere APDUs hintereinander an die Chipkarte gesendet. Die APDUs werden mit Hilfe der Service Methode `sendData()` gesendet. Die Antwort und eventuelle Daten von der Chipkarte werden in der Listener Methode `sendDataCompleted()` entgegengenommen.

```
public void sendDataCompleted(SCSOperationEvent event) {
    if (event.getStatus() != SCSOperationEvent.EVT_SUCCESS) {
        clientProxy.notifyCompletion(ClientProxy.
            COMPLETION_ERROR, null);
    } else {
        try {
            SEND_DATA_STATE++;
            switch (SEND_DATA_STATE) {
                case MASTER_FILE_SELECTED:
                    // check response data
                    if (!checkResponseData(event)) {
                        displayText = "\nRemove card to end";
                        ui.clear();
                        break;
                    } else {
                        // 2. Select Application
                        scs.sendData(APDUS.SELECT_APPLICATION,
                            0, APDUS.SELECT_APPLICATION.
                                length);
                    }
                    break;
                case APPLICATION_SELECTED:
                    // check response data
                    if (!checkResponseData(event)) {
                        displayText = "\nRemove card to end";
                        ui.clear();
                    }
            }
        }
    }
}
```

```
        break;
    } else {
        // 3. Select Certificate Folder
        scs.sendData(APDUS.SELECT_CERTFOLDER,
                    0, APDUS.SELECT_CERTFOLDER.length
                    );
    }
    break;
case CERTFOLDER_SELECTED:
    // check response data
    if (!checkResponseData(event)) {
        displayText = "\nRemove card to end";
        ui.clear();
        break;
    } else {
        // 4. Get the first part of the
        // certificate
        getCertCount++;
        scs.sendData(APDUS.getCompleteGetCert
                    ((byte) (getCertCount - 1)), 0,
                    APDUS._GET_CERT.length);

        break;
    }

default: // This will repeat until the
        // response APDU is 6B 20
        // Append the received part to the cert
        if (getResponseData(event)) {
            displayText = "\nRemove card to end";
            ui.clear();
            break; // We're done
        }
        if (getCertCount > 20) { // Stop if
            // certificate is going to grow to much
            displayText = "\nRemove card to end";
            ui.clear();
            break;
        }
        getCertCount++;
        scs.sendData(APDUS.getCompleteGetCert((
            byte) (getCertCount - 1)), 0,
                    APDUS._GET_CERT.length);
        break;
    }
} catch (AccessException ex) {
    ex.printStackTrace();
}
}
```

Listing 7.5: Implementierung der Methode `sendDataCompleted()`

In der Methode `sendDataCompleted()` befindet sich der Hauptteil der Logik dieser Applikation. Jedes Mal wenn eine APDU an die Chipkarte gesendet wurde, kehrt die Ausführung der Applikation in diese Methode zurück. Hier wird das Ergebnis überprüft und gegebenenfalls die nächste APDU gesendet. Um

zu wissen welche APDU gerade an der Reihe ist, muss der Status der Ausführung in einer Variable geführt werden (hier `getCertCount`). Mögliche Zustände sind: `MASTER_FILE_SELECTED`, `APPLICATION_SELECTED`, `CERTFOLDER_SELECTED`. Wenn das richtige Verzeichnis erreicht wurde, kann das Zertifikat ausgelesen werden. Weil die Antwort der Chipkarte nur eine begrenzte Größe hat (hier 256 Bytes) aber das Zertifikat mehrere Kilobytes groß sein kann, muss es in mehreren Schritten gelesen werden. Von der Klasse `APDUS` wird die passende APDU geliefert in Abhängigkeit der Anzahl der Abschnitte, die bis Dato geholt wurden. Die Methode `getResponseData()` ist zuständig für das Zusammenfügen der Abschnitte des Zertifikats. Das gesamte Zertifikat wird in einer globalen Variable zusammengefügt. Diese Methode prüft auch, ob bereits das gesamte Zertifikat erhalten wurde. Das ist der Fall, wenn die Chipkarte mit den Bytes antwortet: `0x6B 0x00`. Wenn das Zertifikat vollständig erhalten wurde oder wenn die Größe der bereits erhaltenen Daten größer ist als 5KB wird die Ausführung abgebrochen. Der Benutzer wird aufgefordert, die Karte zu entfernen und im Falle, dass die Ausführung erfolgreich war, werden die Daten mit

```
clientProxy.notifyCompletion(ClientProxy.COMPLETION_SUCCESS,
    certData);
```

an die Hostapplikation übertragen. Das Finlet kann beendet werden, d. h. die Hostapplikation kann die Methode `finread_terminateFCRA()` aufrufen, die das Framework veranlasst die `terminate()` Methode des Finlets aufzurufen.

```
public void terminate(int reason) {
    try {
        scs.close();
    } catch (Exception e) {
        debug("Exception during terminate: " + e);
        clientProxy.notifyCompletion(ClientProxy.
            COMPLETION_ABORT, null);
    }
}
```

Listing 7.6: Implementierung der Methode `terminate()`

Hier wird zunächst das Smartcardslot geschlossen. Die Ausführung der Applikation geht weiter mit der Methode `closeCompleted()`, wo auch das Userinterface geschlossen wird. Anschließend wird die Hostapplikation über den Status der Ausführung dieser Operationen benachrichtigt. Die Hostapplikation unterbricht die Verbindung zum Kartenleser und kann nun mit den erhaltenen Daten (X.509 Zertifikat als Array von Bytes) arbeiten.

7.3 SignFinlet

Das "SignFinlet" sendet an die Chipkarte eine APDU, die die Chipkarte veranlasst die übergebenen Daten zu signieren. Dazu muss zunächst die Chipkarte eingeführt und das Smartcardslot geöffnet werden, danach muss die PIN der Chipkarte abgefragt und überprüft werden und anschließend wird die APDU zum Signieren der Daten an die Chipkarte gesendet. Der Aufbau der Applikation ist analog zu dem des "GetCertFinlet" mit dem Unterschied, dass das Abfragen und Überprüfen der PIN eine besondere Behandlung bekommen muss.

Die PIN wird abgefragt mit Hilfe des User Interface Services, der die Methode `readPIN()` anbietet. In dieser Applikation wird

```
ui.readPIN("\n Insert PIN please\n\n", 2, 10);
```

aufgerufen. Dieser Methodenaufruf hat zur Folge, dass auf dem Display des Kartenlesers der erste String-Parameter angezeigt wird und anschließend auf die Eingabe der PIN gewartet wird. Der zweite und dritte Parameter der Methode gibt die minimale und maximale Länge der PIN an. Die PIN Eingabe ist beendet, wenn der Benutzer die PIN bestätigt hat oder die maximale Länge erreicht wurde. Die Ausführung der Applikation wird in der Methode `readPINCompleted()` fortgesetzt, die von dem Framework aufgerufen wird sobald die PIN eingelesen wurde. In dieser Methode kann mit `event.getPINBlock()` die PIN als `PINBlock` erhalten werden. Die Klasse `PINBlock` ist eine `FINREAD` typische Klasse, die nicht zum `STIP` Interface gehört. Sie bietet einige zusätzliche Operationen an, die auf die abgelesene PIN angewendet werden können. Zur Überprüfung der PIN müssen die einzelnen Ziffer in ihre ASCII Repräsentation umgewandelt werden, was hier in der Methode `preparePIN(PINBlock pin)` gemacht wird. Die Methode `getVerifyPinAPDU(byte[] pin)` stellt die APDU zusammen, die die Chipkarte auffordert die Gültigkeit der PIN zu überprüfen. Diese APDU wird anschließend an die Chipkarte gesendet. In der Event Interface Methode `sendDataCompleted()` kann die Antwort der Chipkarte überprüft werden: `0x90 0x00` bedeutet die Verifikation war erfolgreich. Nun kann die APDU mit der Anforderung zur Signatur der Daten (die Daten selbst sind in der APDU integriert) an die Chipkarte geschickt werden. Die Chipkarte sollte eine digitale Signatur der gesendeten Daten zurückgeben (als byte Array), die an die Hostapplikation weitergeleitet wird.

Kapitel 8

Zusammenfassung und Ausblick

Im Jahr 2001 wurde die Herausgabe der FINREAD Spezifikation in Fachkreisen hoch gefeiert. Es war die Zeit, als die programmierbaren Kartenleser immer gefragter wurden, aber ihr Einsatz durch Mangel an einer gemeinsamen Basis erschwert wurde. Die FINREAD Spezifikation wurde entworfen um genau diesem Mangel zu begegnen. Darüber hinaus wurden bei der Entwicklung der FINREAD Spezifikation besonders die Bedürfnisse von Kreditinstituten nach einer sicheren Plattform für den Online Zahlungsverkehr berücksichtigt. Nachdem die Spezifikation publiziert wurde, haben viele Hersteller von Kartenlesegeräten FINREAD konforme Kartenleser hergestellt. Die Verbreitung der FINREAD Kartenleser hielt sich jedoch in Grenzen. Ein Grund dafür könnte sein, dass parallel zu der FINREAD Entwicklung auch das vom Open Card Konsortium unterstützte OCF (Open Card Framework) entstand, das ebenfalls eine interoperable Plattform für die Programmierung von Applikationen für Chipkarten etablierte, die sich bis heute großer Beliebtheit erfreut. Auch wenn FINREAD Kartenleser vor allem in Punkto Sicherheit deutlich besser sind als andere Lösungen wurden sie nicht immer eingesetzt. Um die Online Geldtransaktionen sicherer zu machen, haben viele Kreditinstitute auf andere Methoden gesetzt, wie z. B. Hardware-Generatoren von One Time Passwörtern, die vor allem eine zu FINREAD Kartenleser wesentlich günstigere Alternative darstellen. Auch wenn die Vorschläge des FINREAD Konsortiums nicht den offiziellen Status eines Standards erreicht haben, so bilden sie doch die Grundlage für eine stabile und sichere Plattform für Applikationen für Kartenleser.

Thema dieser Bachelorarbeit ist die FINREAD Spezifikation mit speziellem Fokus auf den Entwurf und die Entwicklung von Applikationen. Eine FINREAD Applikation besteht aus 2 Teilen: der Hostapplikation und der Kartenleser Applikation. Für beide Bestandteile definiert die Spezifikation APIs. Der FINREAD Kartenleser besitzt eine Core Software, die neben der Verwaltung der Hardwareressourcen auch eine JVM bereithält, für die Java Applikationen geschrieben werden können. Diese Java Applikationen werden zu so genannten Finlets kompiliert, signiert und auf den Kartenleser heruntergeladen. Als API für diese Applikationen dient ein erweitertes STIP Interface. Die Programmierung der Kartenleser Applikationen wurde im Rahmen dieser Bachelorarbeit detailliert

beschrieben: im Kapitel 4 wurde der allgemeine Aufbau eines Finlets besprochen und im Kapitel 5 wurde gezeigt wie das Finlet zum Teil mit Hersteller Tools erstellt, heruntergeladen und ausgeführt wird. Als praktischer Teil dieser Arbeit wurden einige Applikationen für den Kartenleser programmiert, die im Kapitel 7 beschrieben werden. Die Host API, die in der FINREAD Spezifikation beschrieben wird, besteht insgesamt aus 18 Funktionen. Die meisten dieser Funktionen dienen dazu, Informationen über den Kartenleser oder die darauf installierten Applikationen zu erhalten sowie um die Ausführung der Finlets zu steuern. Falls der Betriebsmodus des Kartenlesers dies zulässt (nur wenn der “Transparent Mode” nicht ausdrücklich ausgeschaltet wurde) können auch APDUs direkt an die Chipkarte gesendet werden. Die Funktionen der Host API für den im Rahmen dieser Bachelorarbeit benutzten Kartenleser lagen als Windows Library (DLL) vor. Um diese zu benutzen lag es nahe, die Hostapplikation in C bzw. C++ zu programmieren. Somit wäre es nötig gewesen eine FINREAD Applikation in 2 verschiedenen Programmiersprachen zu schreiben. Um diesen Umstand zu beseitigen wurde im Rahmen dieser Bachelorarbeit ein Java Wrapper für die `Finread.dll` erstellt. Dieser ermöglicht es auch die Hostapplikation in Java zu schreiben, was die zukünftige Entwicklung von Applikationen wesentlich erleichtern wird. Der Wrapper kann als eine eins zu eins Kopie der FINREAD Host API benutzt werden: die Namen der Methoden wurden den in der FINREAD API beschriebenen Funktionsnamen angepasst und die C Structures wurden auf Java Hashmaps abgebildet. Der so genannte “EasyWrapper” ergänzt den Java Wrapper. Dieser bietet die Möglichkeit an Host Applikationen auf einem höheren Abstraktionslevel zu programmieren und so von den Vorzügen der objektorientierten Sprache Gebrauch zu machen.

Die erarbeiteten Themen sollen in erster Linie den Entwicklern von FINREAD Applikationen als Einstieg und erste Orientierung dienen. Der Java Wrapper sollte helfen die Entwicklung der Host Applikation zu beschleunigen. Bei der Erarbeitung der Themen wurde festgestellt, dass in Bezug auf die Entwicklung von FINREAD Applikationen sehr wenig Informationsmaterial existiert und kaum praktische Beispiele. Diese Arbeit sollte auch diesem Missstand entgegenwirken.

Das grösste Problem bei der Entwicklung war das Testen der Finlets und das Beheben der Fehler. Wenn bei der Ausführung eines Finlets ein Fehler auftrat musste dieser zunächst mit `System.out.println()` mühsam gesucht werden. Um die Ausgabe in das Finlet einzubauen, war es notwendig das Finlet neu zu kompilieren, daraus eine Jeff Datei zu erstellen anschließend die Fin Datei zu erstellen diese dann auf den Kartenleser herunterzuladen und neu auszuführen. Dieser Prozess nimmt sehr viel Zeit in Anspruch. Außerdem kann man aus den Fehlermeldungen des STIP Frameworks (Fehlercodes in hexadezimal Form) nur mühsam auf das Problem Schließen. Um die praktische Arbeit in Zusammenhang mit der Entwicklung von Finlets abzurunden, wären noch einige Erweiterungen denkbar und begrüssenswert. So wäre zum Beispiel ein einfaches System sehr nützlich, das die Fehlercodes des STIP Frameworks auf einfache Meldungen abbildet, die das vorliegende Problem beschreiben. Dies kann wahrscheinlich nicht mit einem einfachen Mapping gemacht werden, weil die gleichen Codes in ver-

schiedenen Kontexte verschiedene Fehler beschreiben. Für die Realisierung eines solchen Systems muss vor allem die Dokumentation des STIP Frameworks studiert werden. Ein weiterer Punkt, der zur Vereinfachung der Entwicklung von FINREAD Applikationen beitragen könnte, wäre die Integration eines Plugins z. B. für Eclipse. Ein FINREAD Plugin in Eclipse könnte beispielsweise dafür sorgen, dass der Code immer mit den richtigen `-target` und `-source` Parameter kompiliert wird oder dass alle zu implementierenden Methoden der gewünschten STIP Interfaces automatisch erzeugt werden. Die Fähigkeiten eines solchen Plugins könnten nach Belieben und nach Bedarf erweitert werden. Meiner Erkenntnis nach existiert ein solches kommerzielles Eclipse Plugin für die Entwicklung von STIP Applikationen von der Firma "Blue Bamboo", die unter anderem POS Terminale entwickelt. Dieses Plugin kommt mit einem Simulator für ein Gerät, das diese Firma produziert, so dass Applikationen bereits in Eclipse getestet werden können bevor sie auf das Gerät heruntergeladen werden. Die Entwicklung einer solchen Testumgebung für FINREAD Applikationen wäre ein größeres Ziel, das Interessierte verfolgen könnten. Bei der Entwicklung von FINREAD Applikationen verbringt man sehr viel Zeit damit, die Fehler zu finden und das korrigierte Programm neu zu kompilieren und herunterzuladen. Aus diesem Grund würde eine solche Testumgebung die Entwicklung von FINREAD Applikationen geradezu revolutionieren. Allerdings bedarf ein solches Projekt sorgsamer Planung und viel Zeit bzw. die Entwicklung in einem Team.

Warum sollte sich aber jemand all diese Arbeit machen, wenn die FINREAD Spezifikation nicht zum Standard wurde und das Projekt abgeschlossen wurde? Nun das FINREAD Projekt ist abgeschlossen aber nur damit es einem anderen Projekt Platz macht, das noch breiter angelegt ist: das **Embedded FINREAD Projekt**. Während sich das klassische FINREAD Projekt nur auf Kartenleser konzentrierte, die als Peripherie Geräte von Computern betrieben werden, beschäftigt sich das Embedded FINREAD mit allen Typen von Geräten, die alle möglichen Typen von Chipkarten lesen können. Dazu gehören neben den klassischen Kartenleser auch Settopboxen, Handys oder PDAs. Die Verbreitung der FINREAD Kartenleser war nicht so stark, auch weil für die Probleme der Finanzinstitute sich doch kostengünstigere Alternativen fanden, die auch wenn sie nicht die gleiche Sicherheit anbieten wie der Einsatz einer Chipkarte in einer sicheren Umgebung dann doch den Bedürfnissen genügten. Dieser Umstand kann sich zumindest in Deutschland aber bald ändern: die Einführung der elektronischen Gesundheitskarte verlangt geradezu nach einer solchen Plattform, wie sie der FINREAD Kartenleser bietet ([3]). So könnte es für den FINancial transactional IC Card READER ein Revival geben im medizinischen Bereich. Das beweist wiederum, dass der Name dieser Spezifikation, der den Einsatz des Kartenlesers im Finanzbereich suggeriert über dessen Flexibilität und Interoperabilität hinwegtäuscht. Die FINREAD Kartenleser werden mit Sicherheit auch in Zukunft Verwendung finden und man wird immer wieder neue Applikationen benötigen, so dass jede Arbeit an die Weiterentwicklung der Möglichkeiten Applikationen zu schreiben berechtigt und sinnvoll ist.

Literaturverzeichnis

- [1] Omnikey AG. Finread whitepaper. Technical report, Omnikey AG, 2003.
- [2] Omnikey AG. Ok finread sdk. Technical report, Omnikey AG, 2005.
- [3] Verschiedene Autoren. Erarbeitung einer strategie zur einfuehrung der gesundheitskarte - sicherheitsarchitektur. Technical report, IBM Deutschland GmbH, Fraunhofer IAO, SAP Deutschland AG, InterComponentWare AG, Orga Kartensysteme GmbH, 2004.
- [4] FINREAD Working Group. Cen working agreement 14174-1: Business requirements. Technical report, European Committee for Standardization, 2004.
- [5] FINREAD Working Group. Cen working agreement 14174-2: Functional requirements. Technical report, European Committee for Standardization, 2004.
- [6] FINREAD Working Group. Cen working agreement 14174-3: Security requirements. Technical report, European Committee for Standardization, 2004.
- [7] FINREAD Working Group. Cen working agreement 14174-4: Architectural overview. Technical report, European Committee for Standardization, 2004.
- [8] FINREAD Working Group. Cen working agreement 14174-5: Download file format. Technical report, European Committee for Standardization, 2004.
- [9] FINREAD Working Group. Cen working agreement 14174-6: Definition of the virtual machine. Technical report, European Committee for Standardization, 2004.
- [10] FINREAD Working Group. Cen working agreement 14174-7: Finread card reader application programming interfaces (apis). Technical report, European Committee for Standardization, 2004.
- [11] FINREAD Working Group. Cen working agreement 14174-8: Finread client application programming interfaces (apis). Technical report, European Committee for Standardization, 2004.

LITERATURVERZEICHNIS

- [12] Alain Hiltgen, Thorsten Kramp, and Thomas Weigold. Secure internet banking authentication. *IEEE Security and Privacy*, 4(2):21–29, 2006.

Abbildungsverzeichnis

2.1	Beispielszenario für den Einsatz von Chipkarten im Online Zahlungsverkehr	6
3.1	Das FINREAD Framework	12
3.2	Die kryptographischen Schlüssel auf dem FINREAD Kartenleser	13
3.3	Operating Modes	14
4.1	Service orientierte Architektur des STIP Frameworks	18
5.1	Kartenleser Omnikey 3821 USB Pinpad FINREAD	23
5.2	BuildPackage	25
5.3	Administration Tool	26
5.4	Kontrollfluss während der Ausführung eines Finlets	29
6.1	Java Native Interface - Vorgehen bei der Benutzung	33

ABBILDUNGSVERZEICHNIS

List of Listings

4.1	Ein Finlet implementieren	19
4.2	Operation Event Methode openCompleted	19
4.3	Negatives Beispiel der Programmierung eines Finlets	20
4.4	Richtiger eventorientierter Ansatz	21
4.5	Return Data an den Client weiterleiten	21
5.1	Parameter für den Java Compiler	24
5.2	Erzeugen der Jeff Datei	24
5.3	Aktivieren eines Finlets durch die Host Applikation	27
5.4	Ausführung eines Finlets starten und Ergebnis entgegennehmen .	28
6.1	Deklaration der nativen Funktion	42
6.2	Übernahme der Parameter aus dem Java Environment in lokale Variablen	42
6.3	Funktion der Finread.dll aufrufen	42
6.4	Ergebnis Prüfen	42
6.5	Rückgabe des Ergebnisses in das Java Environment	43
6.6	Rückgabe des FINREAD konformen Wertes	43
6.7	Host Applikation mit dem Standard Java Wrapper	43
7.1	Signatur der Finlet Klasse	47
7.2	Implementierung der init() Methode des STIP Interfaces	47
7.3	Implementierung der start() Methode des STIP Interfaces	48
7.4	Implementierung der powerOn() Methode des OperationEvent Listeners	49
7.5	Implementierung der Methode sendDataCompleted()	49
7.6	Implementierung der Methode terminate()	51