Java-basierte Kryptographie wird interoperabel

Vlad Coroama[2], Markus Ruppert[1],
Michael Seipel[1], Markus Tak[1]

[1]Technische Universität Darmstadt
Alexanderstr. 10, D-64283 Darmstadt
{mruppert, seipel, tak}@cdc.informatik.tu-darmstadt.de

[2]ETH Zürich
ETH-Zentrum, IFW D47.2, CH-8092 Zürich
coroama@inf.ethz.ch

Zusammenfassung

Im März 2000 wurde auf der Konferenz Systemsicherheit gezeigt, daß die Notwendigkeit für einen Paradigmenwechsel bei der Entwicklung von Sicherheitsmechanismen für Public-Key-Infrastrukturen notwendig ist (vgl. [BuMa00] und [BuRT00]). Kryptographie muß auf einfache, transparente Weise modular und austauschbar werden, um Rückfallmechanismen bereitzustellen. Sowohl das Signaturgesetz [BaBN99], als auch moderne Sicherheitsprotokolle wie WTLS [WTLS00] empfehlen bzw. spezifizieren die Verwendung alternativer Verfahren wie ECDSA oder ECDH. Bei unserem in der Forschungsgruppe von Prof. J. Buchmann an der TU Darmstadt gewählten Ansatz, ist Java und die JCA (Java Cryptographic Architecture) der zentrale Mechanismus für die Flexibilisierung der Basismechanismen. Die in diesem Artikel dargelegten Entwicklungsergebnisse, *JCA-Provider für PKCS#11* und *GSS-API und JAVA*, zeigen exemplarisch, daß unter Verwendung der JCA als Schnittstelle für kryptographische Funktionen objektorientierte Ansätze auf Standardschnittstellen übertragbar werden, die deren Verwendung zum einen erheblich vereinfachen und zugleich ihre Fähigkeiten erweitern. Die Integrationsfähigkeit von Java und der JCA bietet eine gute Basis für die Realisierung flexibler Public-Key-Infrastrukturen, die sich konform zu den bestehenden Strukturen und Standards verhält.

1 Kryptographie in Standard-Software

Viele verbreitete Anwendungen unterstützen bereits Kryptographie über diverse Schnittstellen. Einen Überblick über die gängigsten Schnittstellen im kryptographischen Umfeld bietet [BaJö00]. Standard-Software ist in der Regel modular aufgebaut und nutzt für kryptographische Funktionen Schnittstellen, an die beliebige Produkte von Drittherstellern angebunden werden können. Über diese Schnittstellen werden sowohl Funktionen wie *Verschlüsselung* und *Digitale Signatur* angeboten als auch einfache Funktionen zur Verwaltung der kryptographischen Schlüssel und Zertifikate.

Die am meisten verbreiteten Schnittstellen dieser Art sind PKCS#11, die Microsoft Crypto API und die GSS-API. Sie kommen u.a. in folgenden Standardprodukten zum Einsatz:

- *Netscape Communicator:* Der verbreitete Internet-Browser nutzt die PKCS#11-Schnittstelle zur Signatur und Verschlüsselung von E-Mails, zur Authentifikation über SSL sowie für das Zertifikats- und Schlüsselmanagement.
- *SAP R/3:* Das in der Wirtschaft weit verbreitete R/3-System kann kryptographische Funktionen über die GSS-API ansprechen. Diese wird zur gegenseitigen Authentifikation und zur Verschlüsselung von Netzwerkverbindungen genutzt.
- *Microsoft Internet Explorer / Outlook:* Der Internet-Browser greift über die von Microsoft definierte Crypto-API auf Kryptofunktionen zu. Es werden die gleichen Funktionen wie beim Netscape Communicator bereitgestellt.

Ein wesentlicher Nachteil dieser Schnittstellen liegt darin, daß Zertifikats- und Schlüsselmanagement, wenn überhaupt, nur auf Basis der vorhandenen Kryptomodule gewährleistet werden. Ein abgesicherter Austausch von Zertifikaten, Schlüsseln oder gar Algorithmen im Sinne von [BuRT99] ist nicht vorgesehen. Um dennoch vorhandene Standard-Software mit wirklich flexibler Kryptographie ausstatten zu können, müssen Brücken zu den jeweiligen Standards geschlagen werden.

2 Java und der Rest der Welt

Die Vorteile der Java Cryptographic Architecture (JCA) gegenüber anderen Krypto-Schnittstellen sind in [BuRT99] geschildert. Über die JCA haben zunächst alle Java-basierten Anwendungen Zugriff auf eine einheitliche Schnittstelle für kryptographische Funktionalitäten wie digitale Signaturen, Verschlüsselung (symmetrisch und asymmetrisch), Message Digests und Schlüsselaustausch. Um auch Standard-Software wie die in Abschnitt 1 genannten Produkte mit der JCA zusammenzubringen, müssen zwei Wege betrachtet werden:

1. *Java greift auf vorhandene Kryptoschnittstellen zu:* Hier wird von Java aus parallel zu Standard-Software auf vorhandene Kryptoschnittstellen zugegriffen. Über diesen Weg kann Standard-Software in Verbindung mit vorhandenen Kryptomodulen von der zusätzlichen Funktionalität von FlexiPKI profitieren. Außerdem können Java-Anwendungen, die bereits die JCA nutzen, nun zusätzlich eine Vielzahl von vorhandenen Produkten im Kryptobereich nutzen.

2. *Standard-Schnittstellen werden direkt von Java bedient:* In diesem Fall sieht die Standard-Software weiterhin ihre gewohnte Schnittstelle. Hinter dieser Schnittstelle verbirgt sich jedoch eine Brücke in die Java-Welt. Kryptographische Mechanismen, die von der Schnittstelle implementiert werden müssen, werden auf die JCA-Konzepte abgebildet. Dadurch sind die Funktionen von FlexiPKI inhärent für Standard-Software zugreifbar, ohne auf die gewohnten Schittstellen verzichten zu müssen.

Beide Wege sollen anhand von Projekten skizziert werden, die am Lehrstuhl Buchmann realisiert wurden. Im ersten Fall wird ein JCA-Provider für die PKCS#11 Schnittstelle beschrieben [Coro00]. Dadurch kann Java auf jedes verfügbare PKCS#11-Modul zugreifen. Als Beispiel für den zweiten Weg soll eine Implementierung der GSS-API durch die JCA beschrieben werden [Seip00]. Dadurch kann Standard-Software, die auf die GSS-API aufsetzt, transparent auf jeden beliebigen JCA-Provider zugreifen.

2.1 Ein JCA-Provider für PKCS#11

Für viele JCA-Anwendungen - z.B. für eine *Public-Key-Infrastruktur* - ist die höhere Sicherheit von Hardware-basierter Kryptographie wünschenswert. Hierfür sind die vergleichsweise unsicheren Software-Implementierungen unzureichend. Alle uns bislang bekannten JCA-Provider liefern jedoch reine Software-Implementierungen der in der JCA spezifizierten Dienste und sind somit für solche Anwendungen ungeeignet.

Um diese Lücke zu füllen, entstand der PKCS#11-Provider [Coro00]. PKCS#11 ist eine C-Schnittstelle, über die eine Anwendung Hardware-Module mit kryptographischer Funktionalität ansprechen kann ([PKCS11]). Meist kommen Smartcards in Verbindung mit einem Chipkartenterminal zum Einsatz, es kann sich aber auch um SmartDisks, PCMCIA-Karten oder Hardware-Sicherheitsmodule (HSM) handeln.

Der PKCS#11-Provider implementiert selbst keine Algorithmen, bringt aber zwei Welten zusammen - die Welt der Hardware-basierten Kryptographie (d.h. vor allem Smartcards) und die JCA-Welt. Damit können JCA-Anwendungen von dem vergleichsweise höheren Sicherheitsniveau profitieren, das Hardware inhärent bietet: Nichtpreisgabe des privaten Schlüssels, Blockieren nach mehrmaliger falscher PIN-Eingabe, usw.

Abbildung 1 zeigt die Architektur des PKCS#11-Providers und die Einordnung in die JCA. Als Nebenprodukt der Entwicklung entstand ein Java-Layer für PKCS#11, der die Funktionalität von PKCS#11-Bibliotheken auf Java-Ebene bereitstellt (JavaPKCS#11-Layer). Da dieser Layer unabhängig von dem darüberliegenden Provider ist, wird hierüber beliebigen Java-Anwendungen die PKCS#11-Funktionalität zur Verfügung gestellt, nicht nur der JCA.

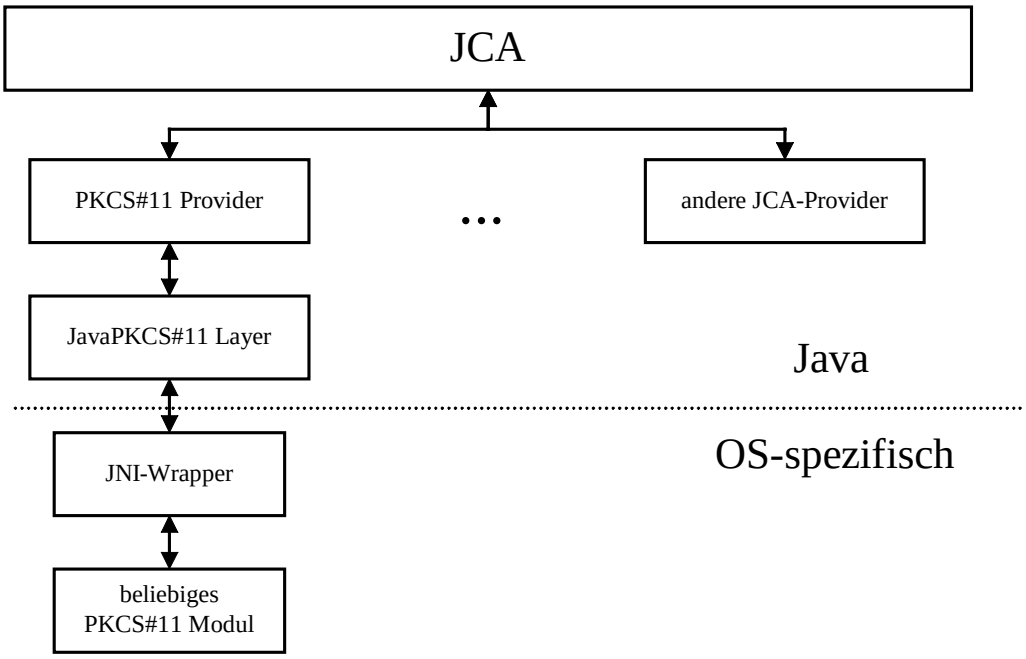

Abbildung 1: Die PKCS#11-Anbindung

2.1.1 Der JavaPKCS#11-Layer

JavaPKCS#11 wurde so konzipiert, daß sich sowohl erfahrene PKCS#11-Entwickler als auch PKCS#11-Einsteiger aus der Java-Welt mit dem Modul schnell anfreunden können. Die Semantik von PKCS#11 wurde beibehalten, damit erfahrene PKCS#11-Programmierer die vertraute Umgebung wiederfinden.

Es werden jedoch alle Vorteile von Java ausgenutzt. Alle C-spezifischen Besonderheiten werden im JNI-Wrapper gekapselt, der durch eine betriebssystemabhängige, dynamisch ladbare Bibliothek implementiert ist und die Aufrufkonventionen von Java und der in C formulierten PKCS#11-Schnittstelle aneinander anpaßt. Dadurch bietet der JavaPKCS#11-Layer den darüberliegenden Applikationen - auch dem PKCS#11-Provider - eine saubere objektorientierte Schnittstelle an. Durch die konsequente Benutzung von Design Patterns ist das System einfach zu verstehen, zu pflegen und zu erweitern.

Wenn das JavaPKCS#11-Modul um neue kryptographische Mechanismen erweitern werden soll - beispielsweise weil auf eine neue PKCS#11-Version umgestiegen wird -, müssen dabei keinerlei Anpassungen am JNI-Wrapper vorgenommen werden.

Zu erwähnen ist noch die Möglichkeit, über PKCS#11 - und damit auch über JavaPKCS#11 - Smartcards zu personalisieren. Mit dieser für eine PKI wichtige Funktionalität können Java-Anwendungen nicht nur aus Client-Sicht, sondern auch aus Administrator-Sicht die sichere Umgebung von Smartcards benutzen. Diese Eigenschaft wird beispielsweise von FlexiTRUST, dem Java-basierten Trustcenter der FlexiPKI, genutzt, um beliebige Kryptomodule mit PKCS#11 Schnittstelle zu personalisieren.

2.1.2 Der PKCS#11-Provider

Der PKCS#11-Provider baut auf JavaPKCS#11 auf. Er kapselt die PKCS#11-Schnittstelle und die darunter befindlichen Schlüssel, Zertifikate und kryptographischen Mechanismen und präsentiert sie in der JCA.

Die Schlüssel und Zertifikate werden über einen JCA-KeyStore gekapselt, wobei ein PKCS#11-Modul in der Regel natürlich keine geheimen Schlüssel preisgibt. Die JCA erwartet aber ein PrivateKey-Objekt, mit dem Signatur- und Entschlüsselungsoperationen initialisiert werden müssen. Daher mußte ein JCA-konformes PrivateKey-Objekt entworfen werden, in dem keine Schlüsselinformationen abgespeichert sind. Stattdessen sind darin Informationen über den aktuellen Zustand des PKCS#11-Moduls abgelegt.

Die Mechanismen des PKCS#11-Moduls werden durch ihre Pendants in der JCA abgebildet. Der PKCS#11-Provider bietet zunächst folgende kryptographische Algorithmen: Message Digests (MD2, MD5 und SHA) und RSA-Signaturen mit jeder dieser drei Digest-Algorithmen kombiniert. Die Menge der angebotenen Algorithmen kann jedoch mit geringem Aufwand jederzeit erweitert werden. Als Herausforderung in der Entwicklung des PKCS#11-Providers haben sich die zum Teil sehr unterschiedlichen Paradigmen von JCA und PKCS#11 herausgestellt. Die starke Ausrichtung der JCA auf Software-Kryptographie wird besonders deutlich, wenn man die Registrierung der kryptographischen Funktionalitäten betrachtet. In der JCA erfolgt die Anmeldung statisch bei der Initialisierung des Providers, während in PKCS#11 die Dienste erst zur Laufzeit bekannt sind. Dies liegt vor allem daran, daß ein Token jederzeit gegen ein anderes ausgetauscht werden kann, welches dann u.U. andere Mechanismen unterstützt. Hier mußte ein Kompromiß gefunden werden, bei dem der PKCS#11-Provider zunächst alle unterstützten Mechanismen anmeldet und zur Laufzeit ggf. eine Exception wirft, falls der gewünschte Mechanismus nicht verfügbar ist.

Eine andere Herausforderung war die Interoperabilität des PKCS#11-Providers mit anderen (Software-)Providern. In der JCA-Philosophie ist es von großer Bedeutung, daß die Benutzung der verschiedenen Provider transparent für die Anwendung verläuft und diese untereinander kombinierbar sind. Der PKCS#11-Provider darf dabei keine Ausnahme bilden. Dabei mußte geklärt werden, wie sich der PKCS#11-Provider zu verhalten hat, wenn eine Anwendung Schlüssel eines anderen Providers mit Algorithmen des PKCS#11-Providers benutzen will oder umgekehrt. Immerhin bietet die JCA mit dem Konzept der *KeySpecs* implementierungsunabhängige Darstellungen für gängige Schlüsseltypen an, z.B. für RSA- und DSA-Schlüssel, jedoch nicht für ECDSA-Schlüssel (Elliptische Kurven Kryptographie). Das Problem ist nicht trivial gewesen, da zumindest die geheimen Schlüssel des PKCS#11-Providers leere Hüllen sind, die lediglich einen Verweis auf ein im PKCS#11-Modul gespeichertes Objekt enthalten.

2.2 GSS-API und Java

Die *Generic Security Services API* (GSS-API) stellt wie PKCS#11 eine Programmierschnittstelle bereit, über die eine Anwendung auf Krypto-Module zugreifen kann ([GSS]). Im Gegensatz zu PKCS#11 arbeitet die GSS-API sitzungsorientiert. Bevor kryptographische Dienste in Anspruch genommen werden können, müssen sich die Kommunikationspartner gegenseitig "kennenlernen".

Ein weiterer Unterschied zu PKCS#11 liegt darin, daß von den zugrundeliegenden kryptographischen Algorithmen und Primitiven abstrahiert wird. Es handelt sich um allgemeine Dienste zur gegenseitigen Authentifikation, zur Absicherung der Nachrichtenintegrität und zur Vertraulichkeit. Die Ausprägung dieser Dienste ist der Implementierung der GSS-API überlassen. Beide Kommunikationspartner müssen jedoch den gleichen Mechanismus unterstützen, in der Regel sind zwei verschiedene Implementierungen der GSS-API zueinander inkompatibel.

2.2.1 Architektur von GSS2Java

Auch bei der GSS-API handelt es sich um eine in C formulierte Schittstelle. Genau wie beim PKCS#11-Provider mußte eine Brücke zwischen der Java-Welt und der nativen Welt des jeweiligen Betriebssystems geschlagen werden. Dazu wurde in beiden Fällen das *Java Native Interface* (JNI) genutzt, jedoch in unterschiedlichenn Richtungen. Während beim PKCS#11-Provider aus Java heraus Aufrufe in die native Betriebssystem-Welt getätigt wurden, muß GSS2Java aus der Betriebssystem-Welt heraus Java aufrufen.

Eine Anwendung wie SAP R/3, welche auf die GSS-API aufsetzt, erwartet eine betriebssystem-spezifische dynamisch ladbare Bibliothek[1]. Für GSS2Java wurde solch eine Bibliothek geschrieben, die im wesentlichen Java aufruft und die in der GSS-API definierten Parameter zwischen C und den objektorientierten Java-Datentypen wandelt. Daher wurde auch in diesem Fall die anzusprechende Schnittstelle zunächst in der Java-Welt abgebildet. Die Architektur von GSS2JCA ist in Abbildung 2 zu sehen.

2.2.2 Abbildung der GSS-API auf die JCA

Eine direkte Abbildung der GSS-API auf die JCA ist nicht durchgängig möglich, da die JCA z.B. keinen Dienst zur Authentifikation anbietet. Solch ein Dienst muß durch Kombination von kryptographischen Primitiven in der JCA wie z.B. Signaturen, Verschlüsselung und Message Authentication Codes (MAC) abgebildet werden. Es wurde eine Authentifikationsschnittstelle entworfen, unter die beliebige Authentifkationsmechanismen

[1] im Falle von Windows sind dies bspw. DLL´s

eingeklinkt werden können, die ihrerseits die JCA benutzen. Als eine mögliche Implementierung wurde eine Drei-Wege-Authentifikation erstellt.

Die Verschlüsselung auf der GSS-API Ebene kann hingegen direkt auf das JCA-Konzept abgebildet werden, da dort eine Verschlüsselungsprimitive bereitgestellt wird.

Schwieriger wird es wieder bei der Abbildung der Nachrichtenintegrität. Zwar könnte man auf den ersten Blick jede zu übertragende Nachricht mit einer digitalen Signatur gegen unbefugte Veränderungen schützen. Man darf dabei jedoch nicht vergessen, daß der Anwendung die Blockbildung überlassen bleibt, so daß im worst case digitale Signaturen über einzelne Bytes geleistet werden, die leicht reproduzierbar und damit wertlos sind. Also wurde auch hier wie bei der Authentifikation eine Schnittstelle geschaffen, über die z.B. agreggierende Message Authentication Codes eingebunden werden können.

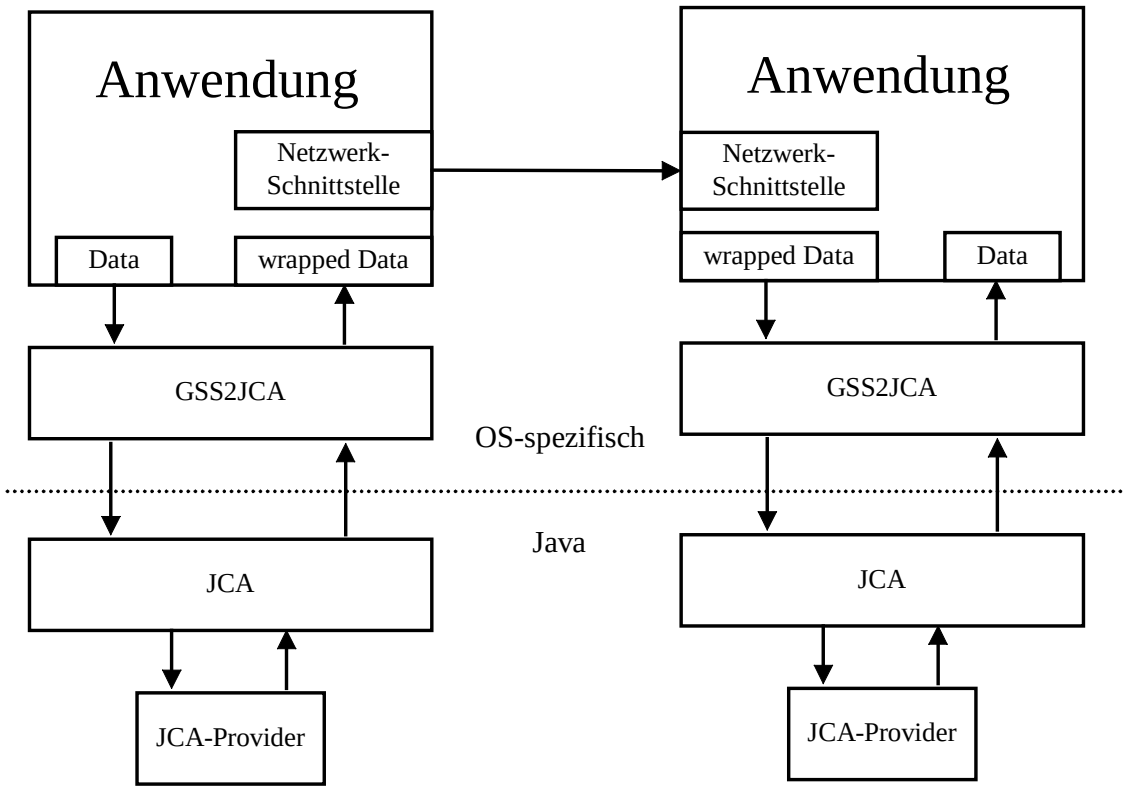

Abbildung 2: Die GSS-API Anbindung

3 Zusammenfassung und Ausblick

Die Java Cryptographic Architecture wurde in zwei Richtungen mit der "Außenwelt" von kryptographischen Standard-Schnittstellen verbunden. Zum einen kann die JCA nun auf beliebige externe Kryptomodule zugreifen und diese mit Zertifikats- und Schlüsselmanagement-Funktionen von FlexiPKI bereichern. Zum anderen kann die JCA selbst als Implementierung von Standard-Krypto-Schnittstellen genutzt werden. Damit wurden wichtige Brücken der Interoperabilität geschlagen. Die Beispiele können als *proof of concept* verstanden werden und als Ausgangspunkt für weitere Brücken zu anderen Standards dienen.

Die Vorgehensweise ist insofern generisch, als daß andere Schnittstellen zunächst mit den objektorientierten Möglichkeiten von Java abgebildet und danach an die JCA angebunden werden.

Für das FlexiPKI-Projekt eröffnet sich damit die Welt der kommerziellen Standard-Software. Damit können vorhandene Lösungen ohne großen Aufwand um die zusätzlichen Möglichkeiten und Sicherheitsmechanismen von FlexiPKI erweitert werden.

Literatur

[BaBN99] F. Bauspieß, J. Biester, N. Neundorf: Spezifikation zur Entwicklung interoperabler Verfahren und Komponenten nach SigG/SigV Signatur-Interoperabilitätsspezifikation SigI Abschnitt A2 Signatur, 1999

[BaJö00] M. Bartosch, J. Schneider: Nutzen und Grenzen von Kryptographie-Standards und ihrer APIs, Konferenz Systemsicherheit, DuD Fachbeiträge 2000

[BuMa00] J. Buchmann, M. Maurer: Wie sicher ist die Public-Key Kryptographie? Tagungsband der Konferenz Systemsicherheit: Grundlagen, Konzepte, Realisierungen, Anwendungen, Vieweg Verlag, ISBN 3-528-05745-9, 2000

[BuRT00] J. Buchmann, M. Ruppert, M. Tak: FlexiPKI - Realisierung einer flexiblen Public-Key-Infrastruktur, Tagungsband der Konferenz Systemsicherheit: Grundlagen, Konzepte, Realisierungen, Anwendungen, Vieweg Verlag, ISBN 3-528-05745-9, 2000

[Coro00] V. Coroama: Flexible Anbindung von Smartcards an eine Java-Sicherheits-infrastruktur (JCA), Diplomarbeit am Lehrstuhl Buchmann TUD, August 2000

[FlexiPKI2] Homepage des FlexiPKI-Projektes am Lehrstuhl Buchmann TUD, http://www.informatik.tu-darmstadt.de/TI/Forschung/FlexiPKI/Welcome.html

[GSS] RFC 2743 und RFC 2744, J. Linn und J. Wray, 2000

[JCA100] Java Online-Dokumentation: How to implement a JCA provider, java.sun.com/products/jdk/1.2/docs/guide/security/HowToImplAProvider.html

[JCA200] Java Online-Dokumentation: The JCA specification, java.sun.com/products/jdk/1.2/docs/guide/security/ CryptoSpec.html

[PKCS11] PKCS#11 Cryptographic Token Interface Standard, RSA-Laboratories, http://www.rsa.com/rsalabs/pkcs, 1997

[Seip00] M. Seipel: Eine Abbildung zwischen der GSS-API und der JCA, Diplomarbeit am Lehrstuhl Buchmann TUD, September 2000.

[WTLS00] WAP-199-WTLS Wireless Application Protocol Wireless Transport Layer Security Specification Version 18-Feb-2000, Wireless Application Forum, Ltd. 2000