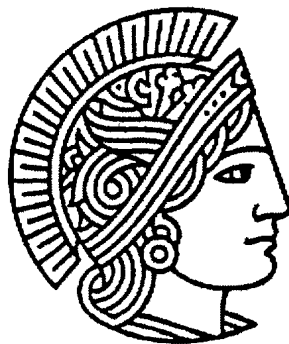


Diplomarbeit

Design und Implementierung eines
Crypto Service Providers
zur Anbindung der
Java Cryptographic Architecture
an das Windows CryptoAPI
am Beispiel der CDC-Provider

Torsten Vest

April 2001



Technische Universität Darmstadt

Betreuer: Dipl.-Inf. Markus Tak

Lehrstuhl: Prof. Dr. Johannes Buchmann

Für Tine.

Danke für alles. Für immer.

Danksagung

Für die Unterstützung und Hilfe bei der Anfertigung dieser Arbeit geht mein besonderer Dank an Prof. Dr. Johannes Buchmann und seine Mitarbeiter. Insbesondere sind dies Markus Tak, der diese Arbeit betreut hat, und Markus Ruppert, der zusätzliche Unterstützung bereitstellte. Weiterhin mein Kommilitone Dirk Schramm, der immer wieder neue Anregungen gab.

Die Verantwortung für das Korrekturlesen und damit für alle Rechtschreibfehler liegt bei Frank, Susi und Christoph; bei Sandra und Claudia die für die erfolgreiche Motivation. Euch und allen meinen anderen Freunden vielen Dank hierfür.

Ganz besonderer Dank gilt meinen Eltern, die mich immer voll unterstützt haben.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit 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. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mainz, den 30. April 2001.

Torsten Vest

Nackstrasse 9

55118 Mainz

Torsten.Vest@gmx.net

Der Autor ist alternativ unter folgender Adresse zu erreichen:

Torsten Vest, Schwarzwaldweg 5, 65462 Ginsheim

Inhalt

Motivation und Überblick	4
1 Schnittstellen unter Windows	7
1.1 Das CryptoAPI in Windows	7
1.1.1 Signieren (Sign)	10
1.1.2 Verifizieren (Verify)	12
1.1.3 Verschlüsseln (Encrypt)	14
1.1.4 Entschlüsseln (Decrypt)	16
1.1.5 Funktionalität des CryptoAPIs	17
1.1.6 Funktionsprototypen auf der Applikationsseite	18
1.2 Crypto Service Provider (CSP)	22
1.2.1 Architektur eines CSP	22
1.2.2 Das CSP Development Kit (CSPDK)	25
1.2.3 Die Schnittstelle des CryptoAPI zum CSP (SPI)	27
1.2.4 Funktionsprototypen auf der CSP-Seite	31
1.3 Object und Algorithm Identifier	34
1.3.1 Object Identifier	34
1.3.2 Algorithm Identifier	35
1.3.3 Umsetzung zwischen verschiedenartigen Identifiern	36
1.4 Zertifikate	38
1.4.1 Aufbau eines Zertifikates	38
1.4.2 Aufbewahrung und Transport	39
1.5 Microsoft Outlook 2000 als Testapplikation	42
1.5.1 Auswählen eines Zertifikates für ein Konto	42
1.5.2 Workflow ‚Signieren einer Email‘	43
1.5.3 CSP-Auswahl durch Outlook	44
2 Schnittstellen zur Java-Funktionalität	45
2.1 Architektur der Java-Funktionalität	45
2.2 Benutzung der CDC-Provider	47
2.3 Zertifikate, Keys und Object Identifier	49
2.4 Das Java Native Interface	50
2.5 Konzept der Abbildung von CryptoAPI auf JCA	55

3	Implementierung	59
3.1	Das Gesamtsystem	59
3.2	Der Crypto Service Provider	61
3.2.1	Architektur und Implementierung	61
3.2.2	Implementierung der Entry Point-Funktionen	65
3.2.3	Eigene Datenstrukturen	71
3.2.4	Die Kernfunktionalitäten	74
3.2.5	Die Schnittstelle zur JCA	76
3.3	Registrieren von Object Identifiern	81
3.3.1	Implementierung des Registrierungsprozesses	81
3.3.2	Nutzung von Object Identifiern im System	83
3.3.3	Neu vergebene Object Identifier	85
3.4	Publizieren von Zertifikaten	86
3.4.1	Implementierung des Publizierungsprozesses	86
3.4.2	Nutzung von Zertifikaten im System	89
3.5	Installation der Komponenten	91
3.6	Microsoft Outlook als Testapplikation	92
3.6.1	Workflow 'Signieren einer Email'	92
3.6.2	Aufrufe von CSP-Funktionen durch Outlook	95
4	Ergebnisse	104
4.1	Fragestellungen und Lösungen	104
4.2	Ausblick	107
	Anhang	108
A.1	Installation eines CSP in Windows	108
A.2	Das Cookbook für die CDC-CSP-Installation	111
A.3	Aufbau der Konfigurationsdatei für die OID-Informationen	115
A.4	Verwendete Softwareinstallation	117
	Begriffe	118
	Abkürzungen	120
	Abbildungen, Tabellen, Code-Beispiele	121
	Bibliographie	123

Motivation und Überblick

Sicherheit von Daten als eines der zentralen Themen unserer heutigen Informationsgesellschaft ist auch im praktischen Einsatz von Software zu einer unabdingbaren Größe geworden. Im Alltag finden sich hier oft Betriebssysteme und Applikationen von Microsoft. Weitere genutzte De-Facto-Standards wie Java lassen sich hiermit nicht ohne Weiteres verbinden. In anderen Sprachen und Umgebungen verfügbare Dienstapplikationen wie die am Lehrstuhl von Professor Buchmann in Java entwickelten können also nicht von Standardapplikationen unter Windows z.B. zum Signieren elektronischer Nachrichten genutzt werden.

Das Ziel dieser Arbeit ist es, die Verbindung zwischen den Schnittstellen der vorhandenen kryptographischen Dienstfunktionalitäten so herzustellen, dass Standardapplikationen diese Abbildung von Windows- auf Java-Funktionalität nutzen können, ohne dass hierzu eine Veränderung des von Windows vorgegebene Modelles notwendig wird.

Das erste Kapitel gibt eine Einführung in die von Microsoft in den Windows-Betriebssystemen angebotene Architektur und ihre Fähigkeiten. Dies erfolgt unter Berücksichtigung von vier Kernaspekten und den jeweils benötigten Schnittstellen: den kryptographischen Dienstleistungsprogrammen (Crypto Service Provider), der eindeutigen Objekt- und Algorithmusidentifizierung, der Identifizierung von Benutzern und der Verwendung durch Standardapplikationen an einem ausgewählten Beispiel.

Im zweiten Kapitel wird die der Windows Kryptographie-Architektur gegenüberstehende Architektur analoger Dienste für Java-Applikationen in Hinsicht auf die gleichen Kernaspekte analysiert. Hinzu kommt die Beschreibung der Schnittstelle, der Art und Weise der Benutzung dieser Dienste aus einer Windows-Applikation heraus und die der Abbildung der grundsätzlichen Funktionalität beider Seiten aufeinander.

Die Beantwortung der aufgeworfenen Fragen anhand der praktischen Implementierung zeigt Kapitel 3. Für jeden der Kernaspekte werden anhand von Auszügen der Implementierung die gewonnenen Erfahrungen und Lösungen geschildert. Im einzelnen sind dies das Dienstprogramm und zwei Hilfsapplikationen,

die ein Gesamtsystem bilden und deren praktischer Einsatz anhand der Beispielapplikation unter Berücksichtigung der Funktionalitätsabbildung demonstriert wird.

Die aus den ersten beiden Kapiteln extrahierten Hauptfragestellungen und der im dritten Kapitel aufgezeigte Lösungsweg sind abschliessend als Ergebnisse zusammengefasst. Der Ausblick bietet weitere Ansätze für zukünftige Betrachtungen.

Zusammengefasst sind die zu beantwortenden Problematiken:

1. Wie nutzen Applikationen die von Windows angebotenen Dienste, und wie werden diese von Windows angesprochen ?
2. Wie werden die vorhandenen Akteure - Dienste, Algorithmen, Identifizierungsmechanismen und Benutzerinformationen - eingesetzt?
3. Wie lassen sich Workflows der Standardapplikationen hiermit auf die Java-Dienste abbilden?

Die Lösung dieser Problematiken führt zur Beantwortung der Frage, ob und wie diese Abbildung im praktischen Einsatz möglich ist.

1 Schnittstellen unter Windows

Im ersten Schritt ist es nötig, die in Windows zur Verfügung stehenden Schnittstellen und Objekte auf ihre Fähigkeiten hin zu untersuchen. Hierzu werden die Architektur des Microsoft CryptoAPIs in Windows, allgemeine kryptographische Operationen und die Bedeutung und Schnittstellen von Crypto Service Providern untersucht. Desweiteren gilt es, die Bedeutung sogenannte Object Identifier und Zertifikate und die Art und Weise der Nutzung des CryptoAPIs durch Applikationen zu überprüfen.

1.1 Das CryptoAPI in Windows

Das CryptoAPI in Windows¹ ist eine universelle Schnittstelle zur Bereitstellung kryptographischer Funktionalität. Alle Windows-Applikationen können damit auf Algorithmen unterschiedlichster Anbieter mit einer einheitlichen Syntax und Semantik zurückgreifen, ohne diese selbst implementieren zu müssen. Damit stehen standardisierte Verfahren zur Verschlüsselung und Authentifizierung von Daten zur Verfügung.

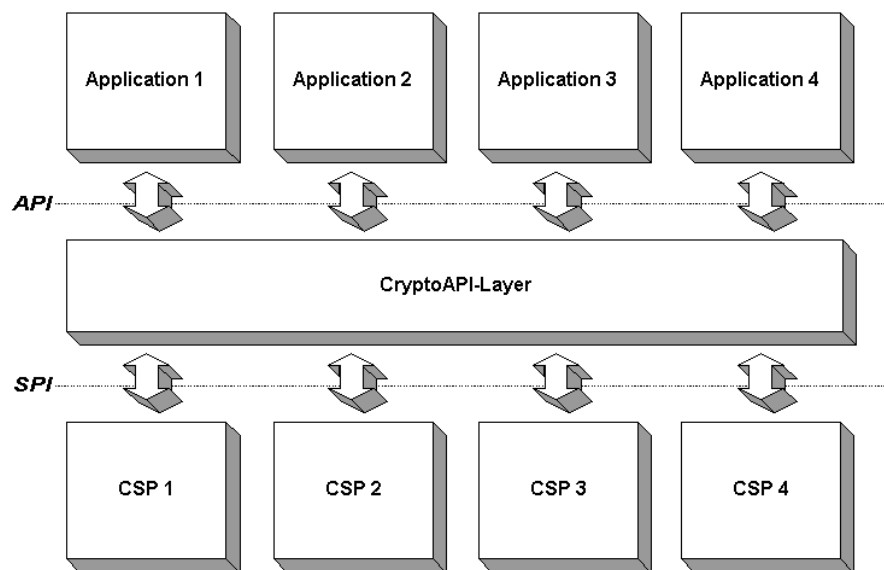


Abbildung 1: Überblick über die Architektur des CryptoAPI

¹ API: Application Programming Interface; das CryptoAPI als Schnittstelle für alle kryptographischen Funktionen ist in allen Windowsversionen ab Windows 98 (erste Version) verfügbar.

In der CryptoAPI-Schicht² sind die jeweiligen abstrakten Funktionen zum Signieren und Verschlüsseln sowie deren Umkehroperationen enthalten. Wie Abbildung 1 zeigt werden diese Funktionen anhand von zusätzlichen Informationen über das Service Provider Interface (SPI) auf die darunterliegende Schicht der sogenannten *Crypto Service Provider* (CSPs) abgebildet. Diese stellen wiederum dem API die eigentlichen Funktionalitäten zur Verfügung und sind zumeist auf bestimmte *kryptographische Verfahren* spezialisiert. Es können beliebig viele CSPs im Betriebssystem koexistieren, unabhängig von der Art ihrer Implementierung (hard- oder softwarebasiert). Ihre Auswahl erfolgt über die Infrastruktur-Informationen.³ Hierzu stellt das CryptoAPI die nötige umliegende Infrastruktur zur Verfügung. Diese betrifft sowohl die Verbreitung und Aufbewahrung von *Zertifikaten* und den zugehörigen *Private* und *Public Keys* als auch die Registrierung von *Object Identifiern* für *Algorithmen*, *Key-Objekte* und *Benutzeridentitäten*.

Zur Aufbewahrung der Zertifikate werden sogenannte *Stores* benutzt. Abhängig vom aktuellen Benutzer in Windows stehen persönliche als auch systemweite Stores zur Verfügung. Die Verfügbarkeit bestimmter Zertifikate ist von der Benutzeridentität abhängig. Jeder Benutzer kann nur auf die ihm eigenen Zertifikate zurückgreifen. Auf der Applikationsseite existieren sowohl einfache Basis-Funktionen als auch komplexere (Simplified) Funktionen zum Bearbeiten von Daten (Message-Funktionen, siehe Abbildung 2). Diese werden allerdings nicht auf unterschiedliche Weise in die Crypto Service Provider abgebildet, sondern benutzen alle die identische Basis-Funktionalität⁴, die jeder CSP zur Verfügung stellen muß.

Desweiteren werden Funktionen zur Verwaltung und Kodierung der Zertifikate benötigt. Auch diese werden vom CryptoAPI zur Verfügung gestellt.

Private Keys werden immer vom CSP aufbewahrt, Public Keys hingegen im Zertifikat und diese im Zertifikats-Store. Dem Entwickler des CSPs ist es freigestellt, wo er die zum CSP gehörigen Private Keys (mit Zuordnung zu einer Identität) speichert. Möglich ist beispielsweise verschlüsselte Speicherung in der Windows-

² Engl. „Layer“.

³ Ein Beispiel hierfür ist das Zertifikat.

⁴ vgl. Kapitel 1.2, Crypto Service Provider.

Registry, einer Datei oder auch auf einem physikalischem Medium wie einer Smartcard. Eine Applikation kann niemals Zugriff auf die Private Keys erhalten.

Die für die Problemstellung dieser Arbeit notwendigerweise zu betrachtenden Objekte dieser Architektur sind:

- ☞ Schnittstellen und Datenobjekte des CSP,
- ☞ Identifizierungsmechanismen für Objekte wie Algorithmen und Keys,
- ☞ Aufbewahrung und Zuordnung von Zertifikaten mittels Stores, und
- ☞ Interaktion mit Applikationen.

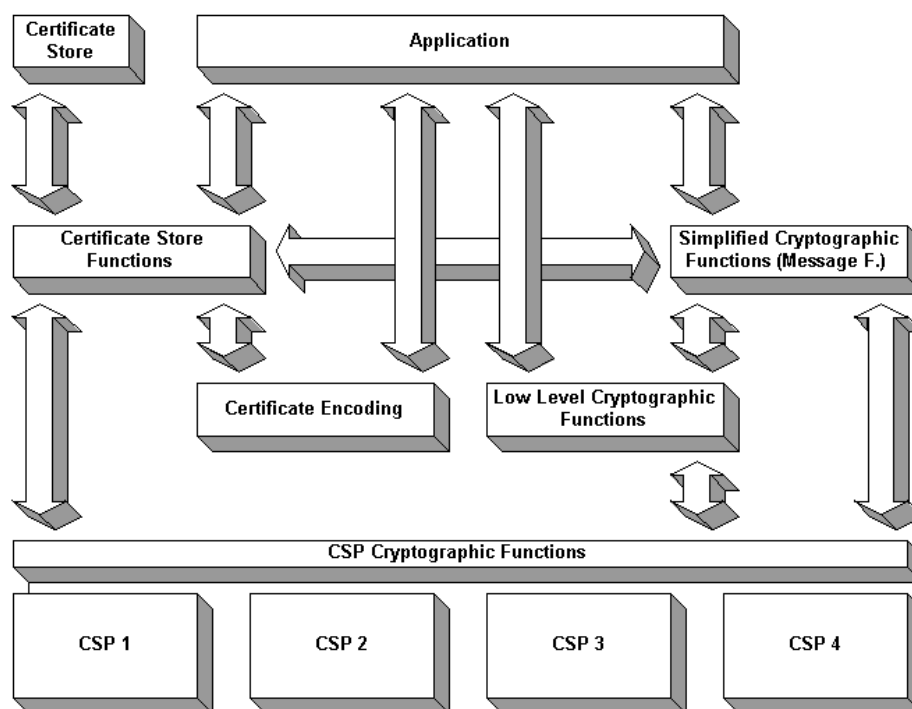


Abbildung 2: Beziehungen der Architekturelemente

Außerdem ist es nötig, die Einbindung grundlegender kryptographischer Operationen in diese Architektur zu erklären. Dies sind Signieren und Verifizieren, Ver- und Entschlüsseln von Information. Interessant ist hierbei, welche zusätzlichen Informationen zu welchem Zeitpunkt benötigt werden und wo diese im CryptoAPI bereitgestellt werden.

Erklärt wird dies beispielhaft am Versenden von vertrauenswürdigen Nachrichten.

1.1.1 Signieren (Sign)

Für alle Basisfunktionalitäten mittels des CryptoAPIs ist stets eine spezielle Vorgehensweise nötig.

Im Falle des Signierens von Information (Nachricht) ist die prinzipielle Herangehensweise zuerst das *Hashen* der Information unter Verwendung eines gewählten Hashalgorithmus. Anschließend wird die Signatur mittels des ausgewählten Signaturalgorithmus erzeugt. Es werden Identität und (im CSP) der Private Key des Besitzers benötigt.

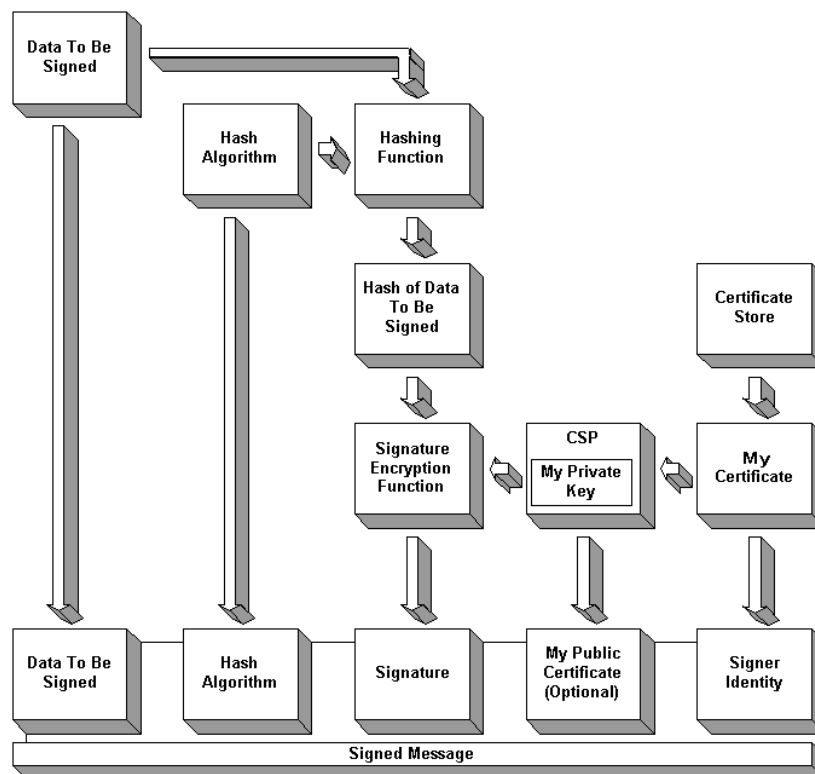


Abbildung 3: Informationsfluß beim Signieren einer Nachricht

Wie die graphische Darstellung (Abbildung 3) verdeutlicht, besteht die signierte Nachricht aus mehreren Teilen: Den Originaldaten, der eigentlichen Signatur, Informationen zu verwendeten Algorithmen (der Hashalgorithmus und optional das Zertifikat mit dem Public Key) und der Identität.

Die wesentlichen Verarbeitungsschritte sind:

1. Eingabe der zu signierenden Daten von der Applikation.
2. Lesen des zur aktuellen Benutzeridentität gehörenden Zertifikates aus einem Store des Betriebssystems.
3. Erhalten des Private Keys des Zertifikates:
Hierzu wird ausgehend von der Zertifikatsstruktur auf einen bestimmten (oder den default) CSP und auf einen sogenannten *Key Container*, der den Private Key im CSP beinhaltet, verwiesen. Zudem ist in dieser Struktur eine Information über den zu verwendenden Hashalgorithmus enthalten. Diese Informationen werden generiert, sobald ein Zertifikat im System installiert wird⁵. Hierbei verlässt der Private Key den CSP nicht.
4. Mit dem gewählten Hashalgorithmus wird ein Hash generiert.
5. Anschließend wird vom CSP mittels des Signaturalgorithmus der Hashwert verschlüsselt und damit die Signatur generiert.
6. Beim Transport der signierten Nachricht kann das Zertifikat inklusive dem Public Key hinzugefügt werden.

Eine detailliertere Betrachtung des genauen Ablaufes und der korrespondierenden Funktionsaufrufe wird im dritten Kapitel, der Beschreibung der Implementierung, gegeben.

⁵ vgl. Kapitel 1.4 Zertifikate.

1.1.2 Verifizieren (Verify)

Die dem Signieren von Nachrichten entgegengesetzte Funktionalität ist das Verifizieren der Signatur. Hierzu werden der in Klartext vorliegende Nachrichteninhalt, die Signatur und das Zertifikat mit dem Public Key benötigt.

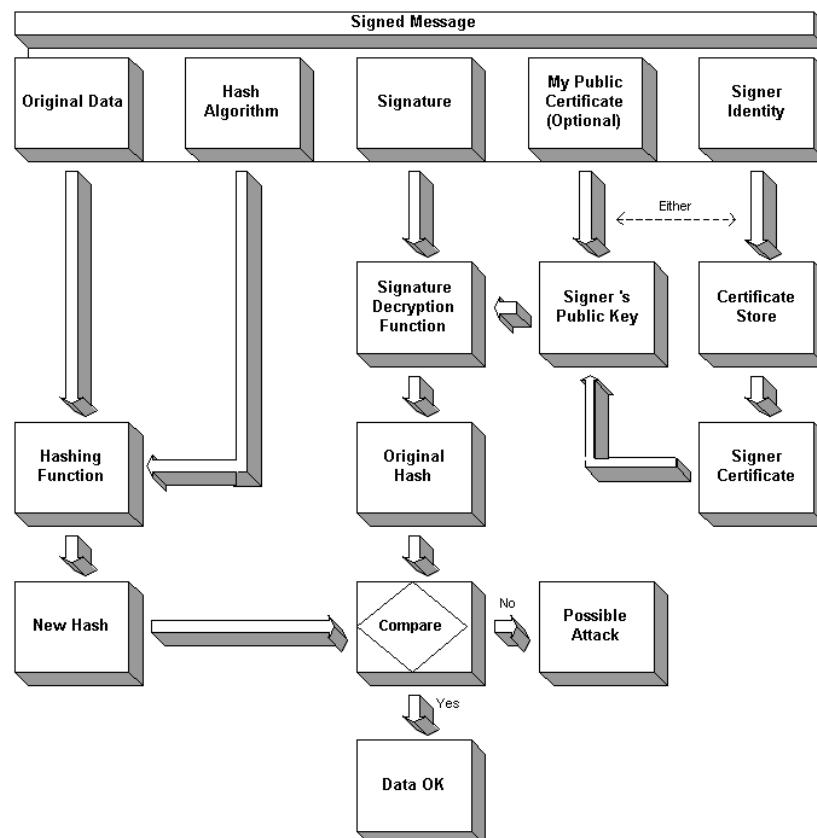


Abbildung 4: Informationsfluß beim Verifizieren einer signierten Nachricht

Die signierte Nachricht muß in der bereits beschriebenen Form vorliegen.

Die wesentlichen Verarbeitungsschritte sind:

1. Eingabe der Daten und der Signatur von der Applikation.
2. Erhalten des (Public) Zertifikates aus einem Store des Betriebssystems oder aus der Nachricht. Hier wird nicht das Zertifikat des aktuellen Benutzers, sondern das des Absenders der Nachricht mittels der Identitätsinformationen gewählt.

3. Unter Benutzung des Public Keys des Zertifikates wird die Signaturinformation entschlüsselt und damit der ursprüngliche Hash der Information wiederhergestellt. Dies bedingt, daß auch beim Empfänger ein CSP vorliegt, der den zu benutzenden Signaturalgorithmus⁶ zur Verfügung stellt.
4. Mittels des in der Nachricht angegebenen Hashalgorithmus wird ein neuer Hash der ursprünglichen Nachricht erzeugt.
5. Der entschlüsselte ursprüngliche und der neu erzeugte Hash werden verglichen.
6. Falls dieser Vergleich ein positives Ergebnis liefert, bedeutet das:
 - ☞ Die Nachricht wurde nach dem ursprünglichen Signieren nicht verändert.
 - ☞ Die beiden Schlüssel (Public und Private Key) gehören als Paar zu einer Identität

Hieraus folgt, daß die Nachricht als vertrauenswürdig erachtet wird.

Falls eine oder beide dieser Bedingungen nicht erfüllt sind, ist die Nachricht nicht vertrauenswürdig.

⁶ Oder eine Alternativlösung, die diesen Algorithmus beinhaltet.

1.1.3 Verschlüsseln (Encrypt)

Um nicht lediglich die Integrität einer Nachricht zu gewährleisten, sondern ihren Inhalt niemandem außer dem Adressaten zugänglich zu machen, ist es nötig, ihren Inhalt zu verschlüsseln. Damit wird gewährleistet, daß kein Unbefugter Zugang zu diesen Informationen erhält.

Hierzu benötigt der Absender als Information über den Adressaten dessen Zertifikat mit dem Public Key sowie eine Implementierung des zugehörigen Algorithmus.

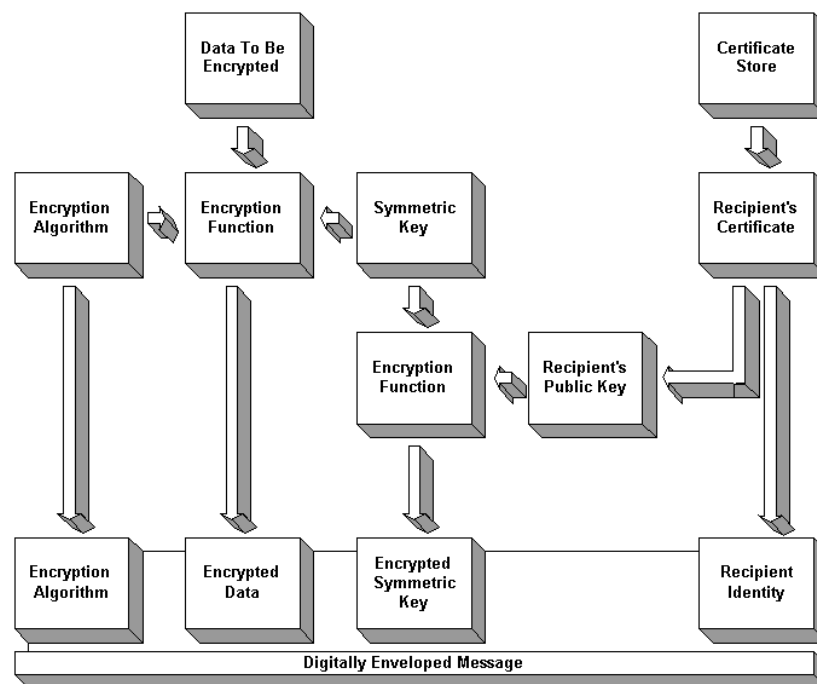


Abbildung 5: Informationsfluß beim Verschlüsseln einer Nachricht

Die wesentlichen Verarbeitungsschritte sind:

1. Eingabe der zu verschlüsselnden Daten von der Applikation.
2. Auswählen eines symmetrischen Verschlüsselungsalgorithmus und Erzeugen des zugehörigen Session Keys. Dabei gibt es zwei Möglichkeiten:
 - entweder wird ein zufälliger neuer vom CSP erzeugt,
 - oder es wird aus einem eingegebenen Passwort ein Key erzeugt.
3. Zusätzlich besteht an diesem Punkt die Möglichkeit, dem ausgewählten Algorithmus Parameter zu übergeben (z.B. den Initialisierungsvektor).

4. Verschlüsselung der Daten mittels des ausgewählten symmetrischen Verschlüsselungsalgorithmus und des zugehörigen Keys. Hierbei kann wegen eventuell vorgenommenen *Paddens*⁷ die Nachricht verlängert werden.
5. Erhalten des Zertifikates des Adressaten aus einem Store des Betriebssystems.
6. Verschlüsseln des Session Keys mittels des vom Zertifikat bestimmten asymmetrischen Algorithmus und des zugehörigen Public Keys des Adressaten. Der Session Key muss dazu vorher aus dem CSP heraus exportiert werden.

Weiterhin kann dem Absender ermöglicht werden, die von ihm verschlüsselte Nachricht später selbst wieder zu entschlüsseln. Dazu muß allerdings der unverschlüsselte Session Key aus dem CSP heraus exportiert werden. Mit der aufgerufenen CSP-Funktion wird dieser nur in verschlüsselter Form herausgegeben, und zwar in diesem Falle mit dem Public Key des Absenders anstatt des Adressaten. Die Applikation kann diesen sogenannten *Key Blob* selbständig speichern. Zur Benutzung muß dann das Zertifikat des Absenders geladen werden, das auf seinen Private Key im zugehörigen CSP verweist.

Die fertiggestellte Nachricht beinhaltet dann die folgenden Bestandteile:

- ~~☞~~ die verschlüsselten Daten,
- ~~☞~~ die Informationen über die verwendeten Verschlüsselungsalgorithmen,
- ~~☞~~ den verschlüsselten Session Key und
- ~~☞~~ die Identität des Adressaten

Die Entscheidung, ausschließlich symmetrische Algorithmen zur Verschlüsselung der eigentlichen Daten zu benutzen, wird üblicherweise aus Performancegründen getroffen⁸. Anschließend wird dieser Session Key⁹ dann mit dem langsameren Public Key Algorithmus verschlüsselt. Wie später gezeigt wird, kann dies jedoch auch umgangen werden (vgl. Kapitel 3.2).

⁷ Unter Padden versteht man das Einfügen von Füllzeichen, um als Gesamtlänge das Vielfache einer Standardgröße zu erhalten.

⁸ Die Daten können durchaus in größerer Menge vorliegen, und die Geschwindigkeit ist im allgemeinen um einen Faktor der Größenordnung 10^3 besser.

⁹ Der Session Key hat eine feste, geringe Länge.

1.1.4 Entschlüsseln (Decrypt)

Die Entschlüsselung arbeitet invers zur Verschlüsselungsfunktion. Es muss auch hier entsprechend mit symmetrischem Session Key und asymmetrischem Private Key gearbeitet werden.

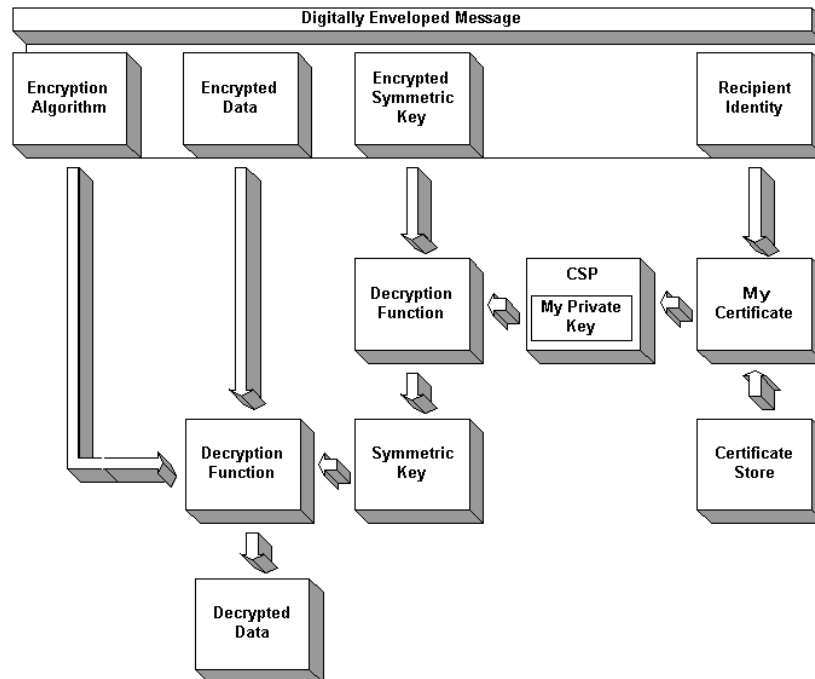


Abbildung 6: Informationsfluß beim Entschlüsseln einer verschlüsselten Nachricht

Die wesentlichen Verarbeitungsschritte sind:

1. Eingabe der zu entschlüsselnden Daten von der Applikation.
2. Erhalten des Zertifikates aus einem Store des Betriebssystems anhand der Identitätsinformation in der Nachricht.
3. Mittels des Private Keys und der Information in der Nachricht zum asymmetrischen Algorithmus kann der zugehörige CSP den symmetrischen Session Key entschlüsseln.
4. Erhalten des zu verwendenden symmetrischen Algorithmus anhand der Algorithmusinformationen in der Nachricht.
5. Entschlüsseln der Daten unter Benutzung des Session Keys.

1.1.5 Funktionalität des CryptoAPIs

Die vorgestellten grundlegenden kryptographischen Operationen werden vom CryptoAPI in Form von einzelnen Basis-Funktionen zur Verfügung gestellt. Diese lassen sich in mehrere Gruppen einteilen:

- ✂ **Auswahl und Beeinflussen von CSPs:** Erhalten von Handles auf CSPs und Lösen dieser; Lesen und Setzen von allgemeinen CSP-Parametern wie Namen, Typ, Algorithmenliste.¹⁰
- ✂ **Erzeugen und Verwalten von Keys:** Erzeugen von Sessions Keys, Handles auf Keys in Zertifikaten, Lesen und Setzen von Key-Parametern, Im- und Export von Keys (zwischen Funktion des CryptoAPI und CSP, *nicht* in die Applikation).
- ✂ **Hashen von Daten:** Erzeugen und Initialisieren von Hash Informationen; Setzen und Lesen von Parametern; Hashen von Sessions Keys und Daten.
- ✂ **Signieren und Verifizieren:** Signieren und Verifizieren bereits gehashter Informationen.
- ✂ **Ver- und Entschlüsseln:** Ver- und Entschlüsseln von Daten unter Eingabe von Keys¹¹.

Das CryptoAPI bildet diese Funktionen¹² unter Veränderung der Parameter auf den CSP ab.

Es stehen innerhalb des CryptoAPIs für die Applikation weitere, zum Teil komplexe Funktionen zur Verfügung. Diese stehen nicht in direktem Zusammenhang mit der Benutzung von CSPs, und werden im weiteren Kontext nicht benötigt.

Als Ausnahmen können hier die Simplified-Message-Funktionen gesehen werden. Diese bieten alle für Nachrichten benötigte komplexe Funktionalität, und nutzen die oben angeführten Basis-Funktionen. Informationen hierzu finden sich in [MSDN].

¹⁰ Unter Windows NT4 und 2000 existieren zusätzliche Funktionen zum Aufzählen aller im System vorhandenen CSPs.

¹¹ Zusätzliches Hashen ist auch bei der Verschlüsselung möglich.

¹² Die CSP-Auswahl wird als einzige Ausnahme nicht auf einen CSP abgebildet.

1.1.6 Funktionsprototypen auf der Applikationsseite

Um die Beschreibung der Basis-Funktionen zu demonstrieren, sei hier beispielhaft die Funktion *CryptAcquireContext* (aus der ersten Gruppe) angeführt.

A. CryptAcquireContext		
Prototyp	BOOL WINAPI CryptAcquireContext(HCRYPTPROV *phProv, (Out) LPCTSTR pszContainer, (In) LPCTSTR pszProvider, (In) DWORD dwProvType, (In) DWORD dwFlags); (In)	
Mögliche Flags	CRYPT_VERIFYCONTEXT CRYPT_NEWKEYSET CRYPT_DELETEKEYSET CRYPT_MACHINE_KEYSET CRYPT_SILENT	Erhalten des Handles ohne Benutzung der Private Keys Erzeugen eines neuen Key Containers Löschen eines neuen Key Containers Auswahl aus systemweiten, nicht benutzer-abhängigen Key Containern Benutzung des CSP ohne User- Interface (z.B. Passwortabfragen für Private Keys sind nicht möglich)
Ausge- wählte Error- codes	NTE_BAD_FLAGS NTE_BAD_KEYSET NTE_BAD_KEYSET_PARAM NTE_EXISTS NTE_KEYSET_ENTRY_BAD NTE_KEYSET_NOT_DEF NTE_NO_MEMORY	Unbekanntes Flag Der Key Container kann nicht geöffnet werden Der Key Containername ist unbekannt Der Key Container existiert schon Der Key Container ist defekt Der Key Container existiert nicht Es steht kein Speicher zur Verfügung

Tabelle 1: Prototyp der Funktion *CryptAcquireContext*

Diese Funktion dient der Applikation dazu, ein Handle¹³ auf einen CSP zu erhalten, was zur Ausführung aller weiteren Funktionen benötigt wird.

¹³ Unter einem Handle versteht man eine Art Zeiger.

Hierbei können Informationen zu einem gewünschten Containernamen, der eine Benutzeridentität spezifiziert, zu einem gewünschten CSP über seinen im System registrierten Namen, dessen Typ und spezielle Flags übergeben werden. Diese Informationen können z.B. aus einem Zertifikat extrahiert werden. Ohne diese werden jeweils im System als Defaultwert eingetragene Informationen ausgewählt.

Der Rückgabewert vom CryptoAPI an die Applikation ist ausschließlich das Handle auf den CSP und die Auskunft über das Auftreten von Fehlern als Funktions-Returncode.

Detailliertere Information zu den einzugebenden Datenstrukturen sind in den nächsten Abschnitten dieses Kapitels zu finden.

Hierbei ist zu beachten, dass die Basis-Funktionen zwar auf die vom CSP zur Verfügung gestellten abgebildet werden, allerdings nicht mit identischen Parametern. Die benutzte Beispielfunktion wird z.B. auf die CSP-Funktion *CPAcquireContext* abgebildet (vgl. Kapitel 1.2 Crypto Service Provider und Kapitel 3.2 zur Implementierung des CSP).

Weitere zur Verfügung gestellte Funktionalität betrifft folgende Punkte:

- ✍ Kodieren und Dekodieren von Daten¹⁴,
- ✍ Verwalten, Transportieren und Bearbeiten von Zertifikaten,
- ✍ Verwalten der Stores und
- ✍ Registrierung von Object Identifiern.

Weitere Details zu allen Einzelfunktionen und den zugehörigen Parametern können der MSDN-Dokumentation entnommen werden [MSDN].

Um die Benutzung der Basis-Funktionen auf Applikationsseite zu demonstrieren, wird im folgenden ein einfaches Beispiel angegeben.

¹⁴ Zum Transport müssen Informationen sicherheitshalber so kodiert werden, um auf allen Plattformen die Integrität der Daten in Bezug auf die verwendeten Kodierungsschemata sicherzustellen. Die z.B. im Internetdatenverkehr benutzte Schema ist 7-Bit-ASCII, und wird mittels DER- oder Base64-Kodierung sichergestellt, wobei die eigentlichen Daten üblicherweise als 8-Bit-ASCII oder UNICODE vorliegen.

```

#define CSP_TYPE 1
#define CSP_NAME "CDC Crypto Service Provider"
#define CALG_CDC_TEST(ALG_CLASS_DATA_ENCRYPT|ALG_TYPE_BLOCK|ALG_SID_CDC_ANY)

HCRYPTPROV hProv = 0;           // csp handle
HCRYPTKEY hKey=0;             // key handle
char input[...];              // input buffer
char output[...];            // output buffer
DWORD inputlen=0;             // input length
DWORD outputlen=0;           // output length

// set data
input = ...
inputlen = ...
strcpy(output, input);
outputlen=inputlen;

// get context & csp handle
if(!CryptAcquireContext(&hProv, NULL, CSP_NAME, CSP_TYPE, CRYPT_NEWKEYSET))
{
    printf("Error %x during CryptAcquireContext!\n", GetLastError());
    return FALSE;
}

// generate key
if (!CryptGenKey(hProv, CALG_CDC_TEST, 0, &hKey))
{
    printf("Error %x during CryptGenKey\n", GetLastError());
    return;
}

// call encrypt
if (!CryptEncrypt(hKey,
                 (HCRYPTHASH) 0, // no hash
                 TRUE,          // final block (or only)
                 0,             // no flags
                 output,        // *output buffer
                 &outputlen,   // its *size
                 outputlen);   // number of bytes to encrypt
{
    printf("Error calling encrypt: '%x'\n", GetLastError());
    return;
}

// release CSP handle.
if(hProv) CryptReleaseContext(hProv,0);

```

Code-Beispiel 1: Einfache Verschlüsselung mit dem CryptoAPI

[Das angeführte Beispiel erzeugt einen neuen Key Container und neue Keys. Es werden keinerlei im System vorhandene genutzt. Es demonstriert nicht die bereits angeführte Verwendung symmetrischer Session Keys.]

Erklärung der Funktionsaufrufe:

☞ CryptAcquireContext:

Zuerst wird das Handle auf den Provider und den dort gespeicherten Key Container geholt. Dieser Key Container wird vom CSP für den Default-Benutzer neu angelegt. Hierin kann jetzt ein Key erzeugt werden. Ausgewählt wurden der Name des Providers, sein Typ und die Aktion.

☞ CryptGenKey:

Ein Schlüsselpaar wird innerhalb des CSPs im Key Container erzeugt und initialisiert.

Als Eingabe dient das Provider-Handle und der Identifier des Algorithmus¹⁵.

☞ CryptEncrypt:

Verschlüsselung der eingehenden Daten und Rückgabe des Ergebnisses. Eingaben sind das Key-Handle und damit der Provider, mehrere Flags, der Zeiger auf die Daten und ihre Länge als Eingabe.

Ausgabe ist die Länge des Ergebnisses in der letzten Variablen und das Ergebnis im Eingabearray.

Die Flags dienen dem Setzen von Optionen, z.B. falls Daten in mehrere Blöcke aufgeteilt übergeben werden.

☞ CryptReleaseContext:

Das Handle auf den Provider wird zerstört und dem CSP damit das Ende der Aktion signalisiert.

Fragestellung 1:

Die sich hieraus ergebende Fragestellung zur weiteren Betrachtung ist, wie Applikationen im konkreten Fall das CryptoAPI nutzen und welche Voraussetzungen dafür erfüllt sein müssen.

¹⁵ Die AlgorithmID; vgl. Kapitel 1.3 und 3.3.

1.2 Crypto Service Provider (CSP)

Die Konstruktion und Verwendung der Crypto Service Provider (CSP) als Hauptpfeiler der CryptoAPI-Architektur muss detailliert beleuchtet werden. Interessant sind hier interne Architektur, Erzeugung und Einsatz eines CSPs unter Benutzung des Microsoft CSP Development Kits und die Schnittstellen zum CryptoAPI.

1.2.1 Architektur eines CSP

Die vom CryptoAPI der Applikation angebotenen Funktionen benutzen die von unterschiedlichen Crypto Service Providern (CSPs) angebotenen Dienste. Dies beinhaltet Funktionalität zu Ver- und Entschlüsselung, Hashen, Signieren und Verifizieren sowie der Speicherung von Keys. Normalerweise sind diese CSPs nicht auf eine bestimmte Applikation zugeschnitten sondern allgemein nutzbar¹⁶.

Zudem wird der Zugriff von Applikationen auf die sicherheitsrelevanten Informationen streng reguliert:

☞ **Kein Zugriff auf Private Keys:** Diese werden immer innerhalb eines CSPs hergestellt und aufbewahrt, und können nur über sogenannte *Opaque Handles*¹⁷ angesprochen werden. Sie stellen nicht wie sonst üblich einen Zeiger auf eine Speicheradresse dar, sondern einen Index, der vom CSP selbst zum Indexieren der Datenstruktur genutzt werden kann, jedoch nicht von der Applikation.

☞ **Keine Beeinflussung der algorithmischen Abläufe im CSP:** Applikationen können zwar erlaubte Parameter eines Algorithmus setzen, allerdings nur über vom CSP angebotene Funktionen. Hiermit wird sichergestellt, dass eine Applikation einen Algorithmus nicht modifizieren kann.

☞ **Authentifizierung der Benutzer nur durch den CSP:** Die zur Ausführung kryptographischer Operationen benötigte Benutzerauthentifizierung wird immer innerhalb des CSPs ausgeführt, wodurch unerlaubte Zugriffe vermieden werden. Zudem ist hiermit eine Erweiterung um neuere Authentifizierungsmechanismen¹⁸ leicht möglich.

¹⁶ Es existieren jedoch auch auf eine Applikation spezialisierte CSPs.

¹⁷ Wörtlich: opaque = undurchlässig

¹⁸ Als Beispiel lassen sich Biometrische Verfahren (Fingerabdrücke, Iriserkennung etc.) anführen.

Typischerweise wird die Implementierung eines CSPs folgende Bestandteile umfassen: Als Schnittstellen die Funktionen zur Seite des CryptoAPIs hin (DLL Entry Point Layer, Abbildung 7), die die CSP-internen Funktionen aufrufen. Diese wiederum haben die Möglichkeit, über weitere Schnittstellen andere APIs zur Unterstützung aufzurufen. Hierbei spielt es keine Rolle, ob diese in Hard- oder Software implementiert sind, da beide Möglichkeiten vorgesehen sind¹⁹. Die CSP-internen Funktionen müssen die folgenden von einem CSP erwarteten Aktionen anbieten:

- ☞ Implementierungen der unterstützten Algorithmen,
- ☞ Verwaltung von Benutzeridentitäten, Keys und zugehörigen Key Containern,
- ☞ Zwischenspeicherung aller Datenobjekte wie Originaldaten, verschlüsselten Daten, Hashinformationen und Signaturen sowie
- ☞ Verwaltung und Behandlung von CSP-Parametern (z.B. die Liste unterstützter Algorithmen).

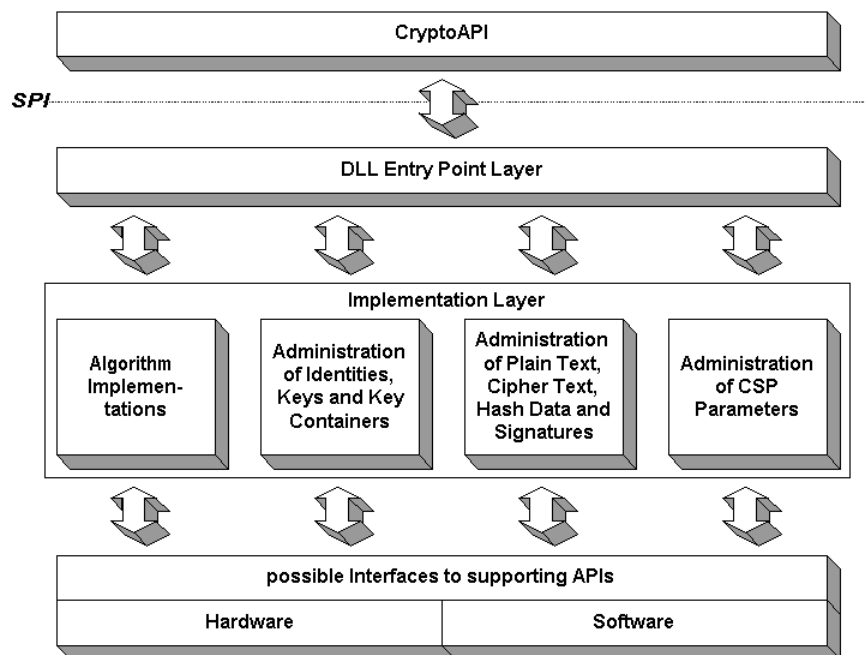


Abbildung 7: Prinzipielle Architektur eines Crypto Service Providers

Typisierung des CSP

¹⁹ Z.B: Smartcards (hardwarebasiert)

Weiterhin muss jeder CSP von einem bestimmten Typ sein. Dieser sollte entweder einem bereits existierenden entsprechen oder es kann ein neuer, eigener Typ definiert werden. Oftmals wird der von Microsoft und RSA definierte Standardtyp eines RSA-Providers genutzt (RSA_FULL_PROV), der die Erfüllung einiger Voraussetzungen vom CSP verlangt. Für jeden der möglichen Typen existiert ein *Default-Provider* in Windows, der bei fehlender Namensangabe in Funktionen als Standard benutzt wird. Welcher CSP jeweils diese Rolle einnimmt, kann über vom CryptoAPI angebotene Funktionen²⁰ bestimmt werden.

Installation des CSP

Der CSP wird als DLL-File (Dynamic Link Library) mit beliebigen Namen (*.dll) für Windows erstellt. Er muss alle der weiter unten angeführten Einsprungspunkte anbieten²¹. Auch allgemeine DLLs unterliegen einer gewissen Überwachung durch das Betriebssystem - qie können nur ausgetauscht werden, solange sie nicht in Benutzung sind²². Hierzu kommt im Falle der CSP-DLL, daß Windows eine Überprüfung der Signatur einer kryptographischen DLL mittels einer System-DLL vornimmt (Advapi32.dll). Um daher einen CSP dauerhaft auf kommerziell erhältlichen Windowsversionen installieren zu können, muss dieser explizit von Microsoft signiert worden sein.

Der CSP wird im System installiert und seine Integrität in Windows durch die eigene Signaturdatei sichergestellt. Diese wird regelmäßig von Windows überprüft und der CSP nur im Erfolgsfalle geladen.

Zu Testzwecken kann dieser Sicherheitsmechanismus mittels des CSP Development Kits (CSPDK) umgangen werden.

²⁰ Es handelt sich hierbei um die Funktionen namens CryptSetProvider & CryptGetDefaultProvider.

²¹ Vgl. Entry Point Funktionen.

²² Vgl. hierzu Betriebssystem-DLLs. Diese können nur ausgetauscht werden, solange Windows nicht aktiv ist und seit Windows 2000 unterliegen sie sogar einer dauerhaften Kontrolle auf Authentizität.

1.2.2 Das CSP Development Kit (CSPDK)

Das von Microsoft erhältliche²³ CSPDK beinhaltet:

- ☞ eine DLL-Hülle, die alle erforderliche Prototypen als C-Sourcecode enthält,
- ☞ Austausch-DLLs für alle unterstützten Windowsversionen (Windows 98/ME, Windows NT4 SP3/4/6, Windows 2000, Windows CE²⁴),
- ☞ zwei Programme zum Signieren der CSP-DLL (nur ausführbare Dateien),
- ☞ ein Programm zur Installation und Registrierung der CSP-DLL in Windows (ausführbar und als C-Sourcecode),
- ☞ ein einfaches Testprogramm (ausführbar und als C-Sourcecode) und
- ☞ eine Kurzanleitung des Signier- und Installationsvorgangs.

Die Benutzung des CSPDK ist beschrieben im Anhang A.1.

Verfügbar ist das gesamte CryptoAPI und die Einbindung von CSPs in der Windows-9X-Familie seit Einführung von Windows 98 (Erste Version). In Windows 95 sind Teilfunktionalitäten nach der Installation aktueller Internet-Explorer-Versionen vorhanden. Es ist anzuraten, als Betriebssystem Windows 98/ME oder Windows NT4 mit SP6a/7 bzw. Windows 2000 zu verwenden, da einzelne Flagmodes bestimmter Funktionen und mehrere einzelne Funktionen vorher nicht unterstützt werden. Auch sind Fehler und unsicheres Verhalten alter Microsoft-CSPs mittlerweile behoben worden²⁵.

Mit Windows mitgelieferte²⁶ CSPs:

- ☞ Microsoft Base Cryptographic Provider,
- ☞ Microsoft Strong Cryptographic Provider,
- ☞ Microsoft Enhanced Cryptographic Provider,
- ☞ Microsoft DSS Cryptographic Provider,
- ☞ Microsoft Base DSS and Diffie-Hellman Cryptographic Provider,
- ☞ Microsoft DSS and Diffie-Hellman / Schannel Cryptographic Provider und
- ☞ Microsoft RSA / Schannel Cryptographic Provider.

Außer den Microsoft-eigenen CSPs sind zusätzlich solche von Drittanbietern hinzugefügt. Als Beispiel lassen sich anführen:

²³ Erhältlich unter [CSPDK].

²⁴ Windows CE wird nur z.T. unterstützt und wird hier nicht weiter betrachtet.

²⁵ In der benutzten CSPDK-Version waren die Prototypen nicht exakt identisch zur Dokumentation. Dies war allerdings leicht zu korrigieren und hatte keinen Einfluss auf die Programmierung.

- ☞ Gemplus GemSAFE Card CSP und
- ☞ Schlumberger Cryptographic Service Provider.

Welche dieser CSPs allerdings tatsächlich in einer Windowsversion enthalten sind, ist abhängig von der Exportpolitik der USA und der Importpolitik des Landes der verwendeten Windowsversion.

Da starke Kryptographie in den USA unter das Kriegswaffenkontrollgesetz fällt, und somit staatlicher Überwachung unterliegt, darf diese nur in bestimmten Fällen und in ausgewählte Länder exportiert werden. Dies betrifft zwar nicht die Basis-CSPs, aber unter Umständen die, die längere Schlüssellängen verwenden (Enhanced-CSPs). Gleichermaßen ist hiervon das CSPDK zur Erstellung eines CSPs betroffen. Dieses konnte bis vor März 2001 nur auf Anfrage von Microsoft bezogen werden.

Seit Sommer des Jahres 2000 sind die Restriktionen der USA zum Export von starker Kryptographie allerdings erleichtert worden, so daß sich der Signierungsprozeß vereinfacht hat. Mittlerweile ist sogar das CSPDK per Download von einer Webseite erhältlich unter [CSPDK].

Als Beispiel für restriktive Importpolitik eines Landes mag Frankreich dienen. In der dortigen lokalen Windowsversion ist das CryptoAPI nicht enthalten (bzw. nicht funktionsfähig), da hier der *Import* starker kryptographischer Funktionalität an sich verboten ist.

Zu Microsofts Signierungspolitik ist zu erwähnen, dass es prinzipiell für jeden möglich ist, einen CSP von Microsoft signiert zu bekommen, so dass er auf jedem handelsüblichen Windowssystem verwendet werden kann. Dies gilt auch für konkurrierende Unternehmen²⁷. Allerdings ist auch dieser Prozeß an sich den Exportkontrollen der USA unterworfen.

Details zu den Ausführungsbestimmungen, Formularen zum Antrag etc. finden sich in [MSDN].

²⁶ Bei Windows 2000.

²⁷ Es finden sich sogar Netscape-spezifische Erweiterungen in den Headerfiles.

1.2.3 Die Schnittstelle des CryptoAPI zum CSP (SPI)

Die Schnittstelle zwischen CryptoAPI und CSP, das Service Provider Interface (SPI), wird von den vom CSP zur Verfügung gestellten Funktionsaufrufen und den übergebenen Datenstrukturen bestimmt. Dies sind die sogenannten *Entry Point Funktionen*, über deren Funktionsnamen die Aufrufe der Applikation durch das CryptoAPI an den CSP weitergeleitet werden. Weiterhin werden die zugehörigen, externen Datenstrukturen behandelt.

Die Entry Point Funktionen

Analog der vom CryptoAPI an der Applikationsschnittstelle zur Verfügung gestellten Funktionen kann man die vom Crypto Service Provider geforderten einteilen in:

Funktionen zu

- ☞ Anbinden an den CSP,
- ☞ Erzeugen und Verwalten von Keys,
- ☞ Hashen von Daten,
- ☞ Signieren und Verifizieren sowie
- ☞ Ver- und Entschlüsseln.

Die *Entry Points*, also die nach außen hin exportierten Funktionen des CSPs, müssen unbedingt alle zur korrekten Funktionsweise des CSPs implementiert werden. Allerdings ist es möglich, für nicht unterstützte Funktionalität in denjenigen Funktionen Nullwerte zurückzugeben, falls der implementierte CSP keinen der Standardtypen darstellt. Da die Applikationen allerdings keinerlei Information hierüber erhalten, sollten in diesem Falle vernünftige Fehlermeldungen und Return Codes gesetzt werden.

Tabelle der Entry Points:

Anbindung an den CSP

<i>Funktion</i>	<i>Beschreibung</i>
CPAcquireContext	Erzeugen und Rückgabe eines Handles auf einen Key Container und damit auf den CSP selbst.
CPReleaseContext	Lösen des Handles auf den Key Container bzw. CSP.
CPGetProvParam	Lesen der Eigenschaften des CSPs wie Name, Version, Typ, Algorithmenliste, Key Containerliste.
CPSetProvParam	Setzen von Eigenschaften des CSPs. Zur Zeit wird nur eine Eigenschaft angeboten, die für diese Arbeit irrelevant ist.

Erzeugen und Verwalten von Keys

<i>Funktion</i>	<i>Beschreibung</i>
CPDeriveKey	Erzeugen eines neuen symmetrischen Keys aus einem Passwort. Dies erfolgt im CSP.
CPGenKey	Erzeugen eines zufälligen neuen symmetrischen Keys (bzw. ein asymmetrisches neues Public und Private Key-Paar).
CPGenRandom	Erzeugen eines mit zufälligen Daten gefüllten Puffers. Diese Funktion wird von CPGenKey genutzt.
CPDuplicateKey	Erzeugen eines Duplikates eines Keys inklusive seines aktuellen Zustandes.
CPDestroyKey	Zerstören eines Keys. Dies erfolgt im CSP.
CPGetUserKey	Erzeugen und Herausgeben eines Handles auf einen im CSP vorhandenen asymmetrischen Key.
CPExportKey	Exportieren eines Keys aus dem CSP heraus in die Applikation. Public Keys werden unverschlüsselt exportiert, Private Keys hingegen können nur verschlüsselt erhalten werden.
CPImportKey	Importieren eines Public oder Private Keys in den CSP.
CPGetKeyParam	Lesen von Key Parametern wie Algorithmus-ID, Initialisierungsvektoren, Paddings, Modi, ...
CPSetKeyParam	Setzen der identischen Parameter, falls dies sinnvoll möglich ist.

Tabelle 2: Entry Point Funktionen des CryptoAPI (Teil a,b)

Hashen von Daten

<i>Funktion</i>	<i>Beschreibung</i>
CPCreateHash	Erzeugen und Initialisieren eines Hash-Objektes im CSP und Rückgabe eines Handles auf dieses.
CPDestroyHash	Zerstören eines solches Hash-Objektes im CSP.
CPDuplicateHash	Erzeugen eines Duplikates eines Keys inklusive seines aktuellen Zustandes.
CPGetHashParam	Lesen der Parameter eines Hash-Objektes wie der ID des Algorithmus und des eigentlichen Hash-Ergebnisses als auch dessen Größe.
CPHashData	Einfügen von Daten in ein Hash-Objekt.
CPHashSessionKey	Einfügen eines Keys in ein Hash-Objekt. Hiermit wird sichergestellt, daß ein Key ohne Zugriff der Applikation gehasht werden kann.
CPSetHashParam	Setzen von Parametern eines Hash-Objektes. Zur Zeit wird nur die Möglichkeit unterstützt, einem solchen Objekt eine eigene Signatur hinzuzufügen.

Signieren und Verifizieren

<i>Funktion</i>	<i>Beschreibung</i>
CPSignHash	Signieren der Daten in einem Hash-Objekt. Dies muss zuvor mit den angegebenen Hash-Funktionen korrekt erstellt worden sein. Eingabewerte außer dem Hash-Objekt sind die Identität (der Key Container) und der Typ des Keys (Keyexchange oder Signature).
CPVerifySignature	Verifizieren der Signatur eines Hash-Objektes. Analog CPSignHash. Zusätzlicher Eingabewert ist die zu überprüfende Signatur.

Ver- und Entschlüsseln

<i>Funktion</i>	<i>Beschreibung</i>
CPEncrypt	Verschlüsseln eines Blocks von Daten. Eingabewerte außer den Daten sind die Identität (der Key Container), der Key mit der Algorithmus ID, die Datenpuffer und ihre Größen und ein Flag. Hiermit kann bestimmt werden, ob nur ein Datenblock (quasi Streammode oder nur ein einzelner Block) existiert oder mehrere (Blockmode). Für mehrere Datenblöcke wird diese Funktion wiederholt aufgerufen. Zusätzlich besteht die Möglichkeit, vor der Verschlüsselung einen Hash des Datenblocks generieren zu lassen.
CPDecrypt	Entschlüsseln eines Blocks von Daten analog CPEncrypt.

Tabelle 2: Entry Point Funktionen des CryptoAPI (Teil c-e)

Die exakten Prototypen aller Funktionen und ihre Verwendung sind beschrieben in [MSDN].

Bei der Implementierung ist zu beachten, dass meist auf diverse unterschiedliche Flags innerhalb einer Funktion getestet werden muss und anhand dieser die eigentliche Aktion aufgerufen wird. Hier können neue Werte definiert werden, die allerdings auch von der Applikation unterstützt werden müssen. Die Entwicklung des CryptoAPIs ist sicherlich noch nicht abgeschlossen, so dass hier weitere Möglichkeiten hinzukommen²⁸ können.

Zudem müssen die vom CSP über das CryptoAPI an die Applikation gegebenen Datenstrukturen definiert werden. Da von außen nicht auf die Datenstrukturen zugegriffen werden kann, werden folgende Handles benutzt:

- ☞ für CSPs bzw. die in ihnen enthaltenen Key Container,
- ☞ für Keys und
- ☞ für Hashinformationen.

Diese sind definiert (in [WINCRYPT.H])als:

```
typedef unsigned long HCRYPTHASH
typedef unsigned long HCRYPTKEY
typedef unsigned long HCRYPTPROV
```

Zusätzlich erfolgen Referenzierungen von Namen stets über Strings²⁹. Zur Aufzählung des Algorithmensangebotes eines CSPs existiert eine komplexe Datenstruktur, die nach außen weitergegeben wird, da sie nicht sicherheitskritisch ist. Ihre Definition findet sich in [MSDN].

Für alle im CSP enthaltenen, nicht nach außen exportierten Objekte müssen eigene Datenstrukturen entwickelt werden. Hierzu können bereitgestellte Hilfsstrukturen z.B. für bestimmte Key-Typen genutzt werden.

Weiterhin existieren fest definierte Strukturen für *Object Identifier*, *Zertifikate* usw., die in den folgenden Unterkapiteln besprochen werden.

²⁸ Insbesondere, wenn man die Dokumentation [MSDN] in verschiedenen Entwicklungsstadien und zudem noch die aktuellsten Header-Files vergleicht.

²⁹ Dies erfolgt je nach Implementierung in einfachen C-Strings oder UNICODE.

1.2.4 Funktionsprototypen auf der CSP-Seite

Wie bereits im ersten Abschnitt wird hier die entsprechende Funktion *CPAcquireContext* als Beispiel angeführt. Die genaueren Implementierungsdetails finden sich im dritten Kapitel.

B. CPAcquireContext		
Proto- typ	BOOL WINAPI CPAcquireContext(HCRYPTPROV *phContainer, LPCTSTR *pszContainer, DWORD dwFlags, PVTableProvStruc pVTable);	(Out) (In / Out) (In) (In)
Mög- liche Flags	CRYPT_VERIFYCONTEXT CRYPT_NEWKEYSET CRYPT_DELETEKEYSET CRYPT_MACHINE_KEYSET CRYPT_SILENT	Erhalten des Handles ohne Benutzung der Private Keys Erzeugen eines neuen Key Containers Löschen eines neuen Key Containers Auswahl aus systemweiten, nicht benutzer-abhängigen Key Containern Benutzung des CSP ohne User- Interface (z.B. Passwortabfragen für Private Keys sind nicht möglich)
Ausge- wählte Error- codes	NTE_BAD_FLAGS NTE_BAD_KEYSET NTE_BAD_KEYSET_PARAM NTE_EXISTS NTE_KEYSET_ENTRY_BAD NTE_KEYSET_NOT_DEF NTE_NO_MEMORY	Unbekanntes Flag Key Container kann nicht geöffnet werden PszContainer ist unbekannter Name Key Container existiert schon Key Container ist defekt Key Container existiert nicht Es steht kein Speicher zur Verfügung

Tabelle 3: Prototyp der Funktion *CPAcquireContext*

Die Prototypen der API-Funktion *CryptAcquireContext* und der SPI-Funktion *CPAcquireContext* sind sich sehr ähnlich. Hierbei werden Informationen über den CSP (Name und Typ) vom CryptoAPI ausgefiltert, da anhand derer schon der CSP ausgewählt wurde. Zusätzlich wird dem CSP eine Hilfsstruktur übergeben. Der Rückgabewert, das *Opaque Handle* des Key Containers wird an die Applikation in Form des CSP-Handles weitergegeben.

Interessant ist der Mechanismus der Abbildung im CryptoAPI:

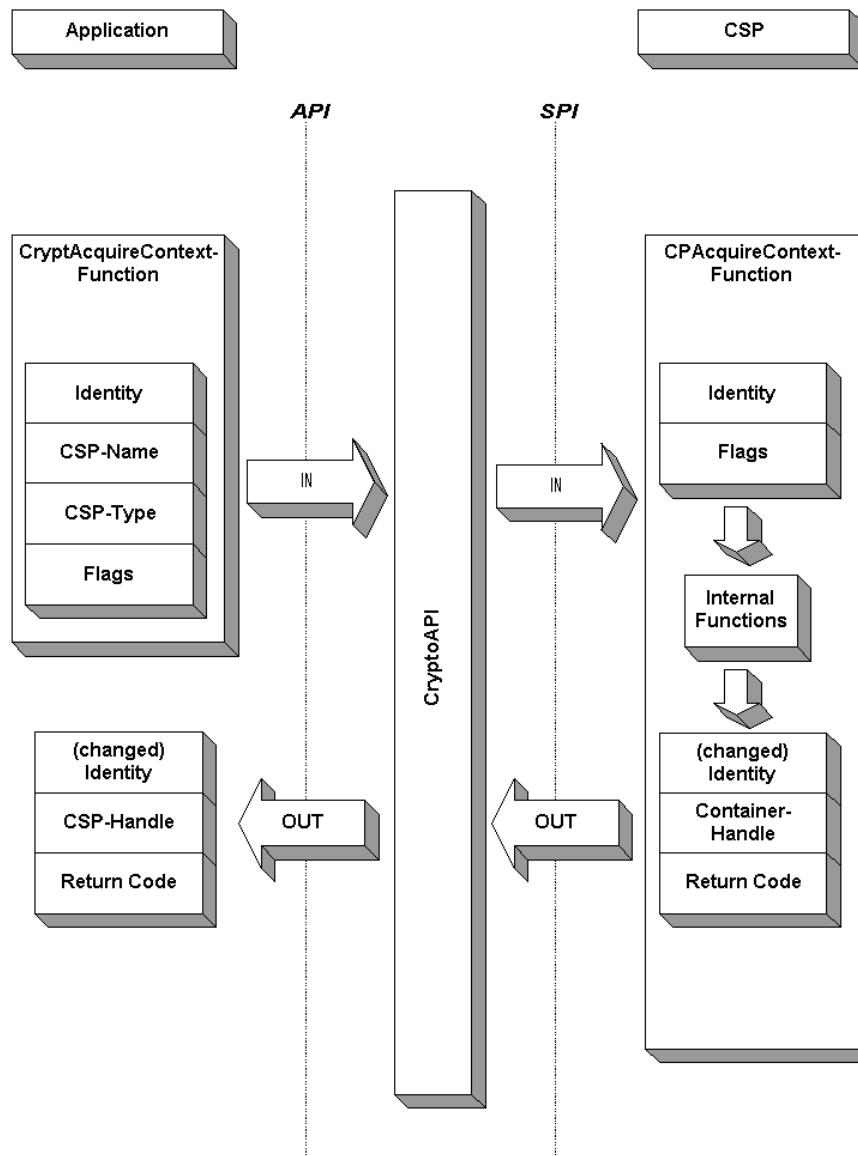


Abbildung 8: Abbildungsmechanismus im CryptoAPI

Das von der Applikation erwartete Handle wird insofern vom CSP erzeugt, da dieser ein Handle auf den in ihm enthaltene Key Container zurückgibt³⁰. Dies sollte aus Sicherheitsgründen nicht die Speicheradresse der Containerstruktur im Speicher sein, sondern nur eine CSP-weite eindeutige Kennung. Der vom CryptoAPI erhaltene

³⁰ Es ist nicht angegeben, ob und wie dieses Handle noch verändert wird. Für den Fall, dass der CSP keines zurückgibt, müsste dies z.B. dann vom CryptoAPI erzeugt werden.

Containername wird an den CSP weitergereicht. In Abhängigkeit von der auszuführenden Aktion, die durch das Flag symbolisiert wird, kann er im CSP gelesen oder sogar gesetzt werden. Das eingegebene Flag veranlasst den CSP zur Auswahl der nötigen internen Aktionen. Mit der letzten Variablen werden von Windows Zeiger auf Callbackfunktionen übergeben, was jedoch zur Zeit nicht genutzt wird.

Weitere Details zu allen Einzelfunktionen und den zugehörigen Werten können der MSDN-Dokumentation entnommen werden [MSDN]. Die Definitionen aller Prototypen und statischer Werte wie Flags zur Programmierung finden sich auch in der C-Header-Datei des CryptoAPIs [WINCRYPT.H]³¹.

Fragestellung 2:

Die sich hieraus ergebende Fragestellung zur weiteren Betrachtung ist, wie ein CSP vom CryptoAPI im konkreten Fall aufgerufen wird und welche Voraussetzungen dafür erfüllt sein müssen.

³¹ Achtung: es ist unbedingt erforderlich, hier die aktuellste Version einzusetzen. Unter Umständen findet sich diese nur im Platform-SDK von Windows [PLATFORM-SDK].

1.3 Object und Algorithm Identifier

Um Interoperabilität zwischen verschiedenen Applikationen und Betriebssystem-Plattformen sicherzustellen, ist eine eindeutige Klassifikation der benutzten Objekte und Verfahren nötig. Wichtig ist, dass es zwei verschiedene Ansätze zur Lösung gibt, die in Einklang gebracht werden müssen – der weltweit gültige Object Identifier-Standard und die Microsoft AlgorithmIDs.

Im konkreten Fall sind dies Key Objekte und die dazugehörigen Algorithmen. Beispielsweise muss ein Nutzer einer Unix-Plattform mit einem beliebigen Email-Programm³² für ihn bestimmte Nachrichten verifizieren oder entschlüsseln können, auch wenn diese von einem Windows-System mit z.B. Microsoft Outlook 2000³³ verschickt wurden. Diese Identifier (OIDs) werden üblicherweise als Strings bzw. Nummernfolgen definiert und sind hierarchisch organisiert.

1.3.1 Object Identifier

Um die Eindeutigkeit weltweit zu gewährleisten, wurden zwei von internationalen Organisationen geschaffene Standards genutzt: Zum einen ist der ASN.1³⁴-Standard anzuführen. Er dient zur textuellen Beschreibung von Objekten nach [X.208, RFC³⁵ 1778, RFC2252], herausgegeben von ITU³⁶ und IETF³⁷ im Stile einer BNF³⁸-Sprachbeschreibung. Zum anderen ist die sogenannte „dot-notation“ zu nennen, die Beschreibung im Stile von IP³⁹-Adressen im Internet anhand von durch Punkte getrennter Nummern.

Verschiedene Namensräume innerhalb dieser Nummerhierarchie werden von weiteren Organisationen zur Standardisierung vergeben, z.B. dem ANSI⁴⁰ für Namen aus dem US-amerikanischen Raum oder der IANA⁴¹ für Internet-Adressen.

³² Das Email-Programm muss die entsprechenden Algorithmen und Keys unterstützen.

³³ Im weiteren Outlook genannt.

³⁴ Abstract Syntax Notation One

³⁵ Request For Comments (definierter Standardvorschlag)

³⁶ International Telecommunications Union

³⁷ Internet Engineering Task Force

³⁸ Backus-Naur-Form zur Beschreibung formaler Sprachen.

³⁹ Internet Protocol

⁴⁰ American National Standards Institute

⁴¹ Internet Assigned Names (and Numbers) Authority

Die für kryptographische Objekte und Algorithmen zuständige Organisation ist die PKIX⁴²-Working-Group des IETF, deren Festlegungen im Internet-Draft [IPKI] vorgeschlagen wurden. Diese haben zur Zeit (bis September 2001) noch nicht den Status eines RFC und sind damit noch eventuellen Änderungen unterworfen.

Als Beispiel hierfür die Beschreibung des *Elliptic-Curve Digital Signature Algorithm* (ECDSA, definiert nach [X9.62]) in beiden Formen:

(ASN.1)

```
ansi-X9-62 OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840) 10045 4 1 }
```

(dot-notation)

```
1.2.840.10045.4.1
```

Verwendungsbeispiel:

```
#define CDC_OID_SHA1withECDSA "1.2.840.10045.4.1"
```

Die erste Stelle gibt die Dachorganisation (hier ISO) an, darauf folgt die Unterorganisationseinheit (ANSI), das Land (USA) und die ID des Objektes inklusive weiterer Abstufungen.

Genauere Informationen hierzu finden sich bei den genannten Organisationen in [X.208, RFC 1778, RFC2252, IPKI] und auch z.B. in [ALVESTRAND]. Hier ist auch eine kostenfreie Übersicht fast aller vergebenen OIDs zu finden.⁴³

1.3.2 Algorithm Identifier

In Windows-Systemen wird zudem eine proprietäre Bezeichnung für die zur Verfügung stehenden Algorithmen benutzt⁴⁴. Diese sind in Abhängigkeit vom aktuell verwendeten CSP definiert. Identifier, die zu den im System mitgelieferten CSPs gehören, sind bereits statisch in den CSPs und den Definitionsdateien zur Programmierung definiert [WINCRYPT.H]. Hier finden sich nicht nur von Microsoft

⁴² Public Key Infrastructure

⁴³ Anzumerken ist, dass die Standardisierungsorganisationen üblicherweise Geld für ihre Dienste verlangen.

⁴⁴ Dasselbe gilt für die Key Objekte. Diese sind nicht getrennt, sondern benutzen eine zusätzliche Typgruppe.

selbst definierte Werte, sondern auch die von Mitbewerbern bei Microsoft eingereichten. Der Entwickler des CSPs ist frei in der Definition eigener Werte, unter Beachtung des CSP-Types. Diese sind nur im Kontext des eigenen CSPs gültig.

Üblicherweise werden die AlgorithmIDs in festgelegter Form bestimmt: sie setzen sich aus einem Wert für die Algorithmenklasse⁴⁵, dem Blockformat⁴⁶ und einer Algorithmennummer zusammen. Dies sind jeweils 1-Byte-Werte, die zu einer 4-Byte-Zahl (Typ: DWORD, entsprechend 4-Byte Integer) zusammengesetzt werden.

Beispielsweise:

```
#define CDC_ALG_SID_SHA1withECDSA 1
#define CDC_CALG_SHA1withECDSA
        (ALG_CLASS_SIGNATURE|ALG_TYPE_ANY|CDC_ALG_SID_SHA1withECDSA)
```

1.3.3 Umsetzung zwischen verschiedenartigen Identifiern

Da in der Interaktion unterschiedlicher Plattformen und Applikationen die weltweit gültigen OIDs eingesetzt werden, das CryptoAPI an die CSPs allerdings die windows-spezifische AlgorithmID weitergibt, muss es in Windows eine Möglichkeit zur Umsetzung dieser Identifier geben. Hierzu werden die Abbildungsinformationen mit speziellen Funktionen in der Windows-Registry eingetragen und können von jeder Applikation ausgelesen werden.

Die hier eingetragenen Informationen sind:

- ☞ die OID im dot-notation-Format (als String),
- ☞ die zusammengesetzte AlgorithmID (als DWORD),
- ☞ der Name des Objekts (als String) und
- ☞ die Objektgruppe⁴⁷ (als Byte).

Ein Beispiel für eine solche Definition ist:

```
#define CDC_ALG_SID_SHA1withECDSA 1
#define CDC_CALG_SHA1withECDSA      (ALG_CLASS_SIGNATURE |
        ALG_TYPE_ANY |
        CDC_ALG_SID_SHA1withECDSA)
#define CDC_OID_SHA1withECDSA      "1.2.840.10045.4.1"
#define CDC_NAME_SHA1withECDSA     "ECDSA-SHA1"
#define CDC_GROUP_SHA1withECDSA    CRYPT_SIGN_ALG_OID_GROUP_ID
```

⁴⁵ z.B. Signatur-, Hash- oder Verschlüsselungsalgorithmen.

⁴⁶ Block- oder Stream-Format, abhängig vom verwendeten Algorithmus.

⁴⁷ Analog der Algorithmenklasse; zusätzlich sind auch andere Objekttypen möglich.

In der folgenden Abbildung 9 wird ein Überblick gegeben, wie bei Einführung eines neuen CSPs mit neuen Identifiern diese im System bekannt gemacht werden müssen.⁴⁸

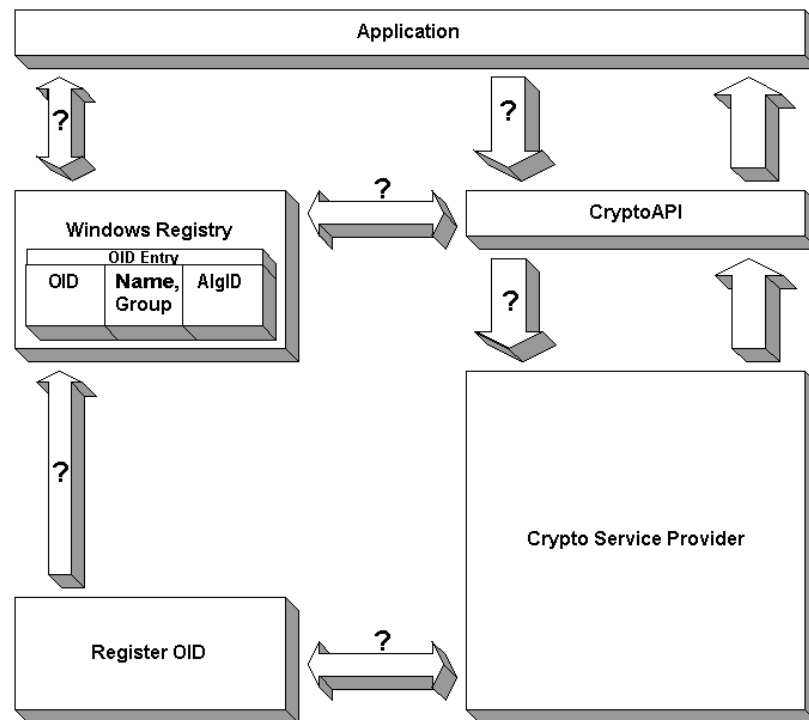


Abbildung 9: Prinzipielle Registrierung von Object Identifiern

Die komplette OID-Definition muss sowohl zur Laufzeit im CSP bekannt sein, als auch bei Installation oder Erweiterung dem CryptoAPI und der Applikation bekannt gemacht werden. Hierzu sollte eine zusätzliche Software diese Informationen zusammenstellen und registrieren. Welcher Systemteil zu welchem Zeitpunkt OID und bzw. oder AlgorithmID benötigt, muss allerdings noch geklärt werden.

Fragestellung 3:

Die sich hieraus ergebende Fragestellung zur weiteren Betrachtung ist, wie Object Identifier von Applikation, CryptoAPI und CSP im konkreten Fall genutzt werden und welche Voraussetzungen dafür erfüllt sein müssen.

⁴⁸ Die Fragezeichen bezeichnen noch zu klärende Abläufe und Datenflüsse.

1.4 Zertifikate

Zur Speicherung aller zu einer Benutzeridentität gehörenden Informationen, vor allem des Private und des Public Keys, wird ein sogenanntes Zertifikat benutzt. Es ist zur Wahrung der Kompatibilität über Systemgrenzen nach einem allgemein gültigen Standard aufgebaut, und wird selbst digital signiert, um eine Verifizierung des Inhaltes zu ermöglichen. Hiermit lässt sich die Identität des Benutzers authentifizieren, sofern das Zertifikat überprüfbar von einer anerkannten Zertifizierungsstelle ausgestellt wurde⁴⁹. Interessante Eigenschaften sind Aufbau, Aufbewahrung und Transport des Zertifikates.

1.4.1 Aufbau eines Zertifikates

Der Aufbau des Zertifikats nach dem zur Zeit üblichen Standard X.509, beschrieben in [X.509, RFC 2459, CERT], beinhaltet die folgenden Eigenschaften:

- ☞ die **Identität des Benutzers** (aufgebaut nach [X.500]). Hierin sind Namen, zugehörige Organisation, Email-Adresse und weitere Informationen enthalten. Siehe Abbildung 10,
- ☞ der zugehörige **Public Key**,
- ☞ die **Gültigkeitsdauer**,
- ☞ die **Identität der ausstellenden Zertifizierungsstelle**,
- ☞ die **Version** des Zertifikattypes,
- ☞ die **Seriennummer**.

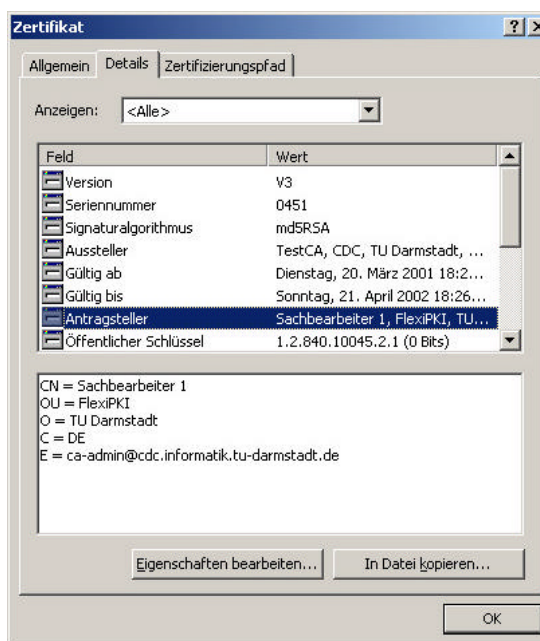


Abbildung 10: Ansicht eines Zertifikates im Zertifikatsmanager von Windows

⁴⁹ Dies ist dann überprüfbar anhand des Zertifikates der Zertifizierungsstelle.

Zusätzlich können noch sogenannte erweiterte Eigenschaften angegeben werden. Diese ermöglichen das Weitergeben von applikations- oder systemspezifischen Informationen.

Microsoft benutzt diese Eigenschaften, um eigene Information für das CryptoAPI zu transportieren. Eingetragen wird der Name des zu benutzenden CSPs und der Name des zur Identität gehörigen Key Containers in diesem CSP. Wichtig ist hierbei, dass diese Informationen nur von Windows-Systemen ausgewertet werden und nicht plattformübergreifend sind. Zudem ist es abhängig von der Applikation, die das Zertifikat nutzt, ob und wie diese Informationen ausgewertet werden (vgl. Abschnitt 1.5).

1.4.2 Aufbewahrung und Transport

Windows sieht zur Aufbewahrung von Zertifikaten Stores vor. In diese wird der Inhalt eines Zertifikates aus einer Datei übertragen. Zuvor wurde diese Zertifikatsdatei von einer Zertifizierungsstelle in einem standardisierten Format erzeugt. Hierbei wird für zur Veröffentlichung bestimmte Zertifikate⁵⁰ das PKCS12-Format verwendet, das PKCS7-Format analog für private Zertifikate⁵¹ für den persönlichen Gebrauch (Formatdetails unter [PKCS7, PKCS12]).

Es existieren mehrere Stores zur Aufbewahrung in Windows:

<i>Store</i>	<i>Beschreibung</i>
My	Persönliche Zertifikate
Trust	Zertifikate vertrauenswürdiger Personen, nur deren Public Keys
CA	Zertifikate von Zwischenzertifizierungsstellen (als Bezugszertifikat)
Root	Zertifikate von Stammzertifizierungsstellen (als Bezugszertifikat)
Addressbook	Eigener Store von MS Outlook (zusätzlich zu den anderen genutzt)

Tabelle 4: Zertifikat-Stores in Windows

Diese sind physikalisch nicht getrennt, sondern die Aufteilung in mehrere wird vom für die Zertifikatsverwaltung genutzten API vorgenommen. Gespeichert werden die

⁵⁰ Diese enthalten nur den Public Key zur Identität.

⁵¹ Hier ist zusätzlich der nicht zur Veröffentlichung bestimmte Private Key enthalten.

Informationen in der Windows-Registry⁵². Zudem ist es möglich, hier eigene Erweiterungen um physikalische als auch solche System-Stores unter Implementierung der vollständigen Schnittstellenfunktionalität vorzunehmen. Dies wird hier nicht weiter betrachtet und ist generell zu finden in [MSDN].

Aus diesen Feststellungen zum speziellen Aufbau der Zertifikate unter Windows ergibt sich, dass ein separates Programm zum Publizieren der Zertifikate benötigt wird.

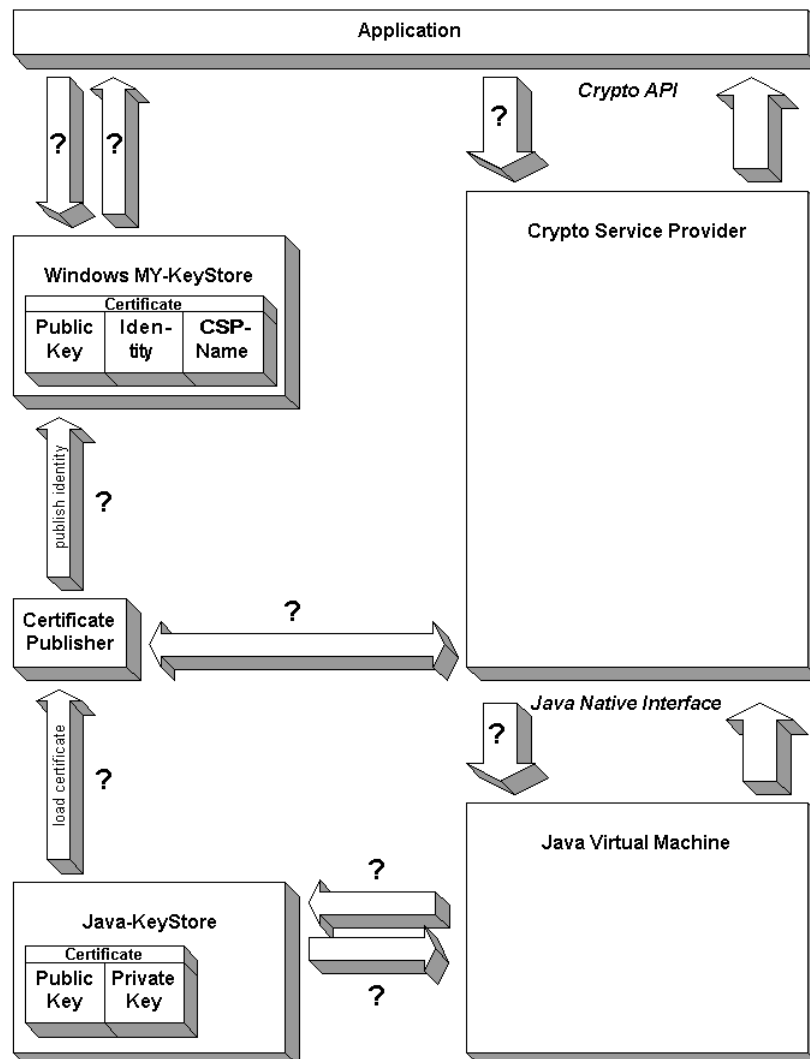


Abbildung 11: Prinzipielles Publizieren von Zertifikaten⁵³

⁵² Mit Einführung von Windows 2000 und Active Directory auch serverbasiert im Directory.

⁵³ Die Fragezeichen bezeichnen noch zu klärende Abläufe und Datenflüsse.

Das von einer Zertifizierungsstelle herausgegebene Zertifikat wird aus einer Zertifikats- oder KeyStore-Datei⁵⁴ geladen. Anschließend muss das Programm auf die Existenz des zugehörigen CSPs testen, und von diesem den zur Identität gehörenden Key Container-Namen abfragen. Diese Informationen werden im Zertifikat eingetragen, bevor es in einem der Windows-Stores gespeichert wird.

Dieser Vorgang muss allerdings nicht bei jeder Installation eines Zertifikates vorgenommen werden. Da die Informationen auch beim Transport des Zertifikates als Erweiterungen des X.509-Form übertragen werden, sind diese damit auf dem Zielsystem zugänglich bei Vorhandensein des korrekten CSPs inklusive des Key Containers. Falls diese Eintragungen bereits vorliegen, bietet üblicherweise die Applikation über Aufruf des Windows Certificate Manager die Möglichkeit, Zertifikate in den gewünschten Zertifikatsstore zu importieren.

Fragestellung 4:

Die sich hieraus ergebende Fragestellung zur weiteren Betrachtung ist, wie Zertifikate von Applikation, CryptoAPI und CSP im konkreten Fall genutzt werden und welche Voraussetzungen dafür erfüllt sein müssen.

⁵⁴ Im konkreten Fall wurde das Zertifikat mit einer Test-Certificate Authority in Java erstellt und als Zertifikatsfile in PKCS12 bzw. als Java-Key Store gespeichert.

1.5 Microsoft Outlook 2000 als Testapplikation

Um die Verwendung des CryptoAPI an einer Standardapplikation zu untersuchen, wird das Email-Programm Microsoft Outlook 2000 benutzt. Dieses dient dazu, Emails zu erstellen und zu versenden, die kryptographisch signiert⁵⁵, verifiziert, ver- und entschlüsselt werden können. Hierzu können mittels Outlook verschiedene Benutzerkonten erstellt werden, denen Identitätsinformationen und eigene Zertifikate zugeordnet werden können. Zudem werden Kontaktinformationen für andere Benutzer gespeichert, inklusive deren Public-Key-Zertifikaten. Interessant ist die Fragestellung, inwiefern Outlook mittels des CryptoAPIs vorhandene CSPs nutzt.

1.5.1 Auswählen eines Zertifikates für ein Konto

Um bei einem Konto die notwendigen Voraussetzungen für das Benutzen eines Zertifikates zu schaffen, ist es notwendig, dass die im Konto von Outlook eingetragene Email-Adresse mit der des Zertifikates übereinstimmt. Diese Einstellung wird vorgenommen im Menüpunkt „Extras/Konten“ (Aufruf der „Eigenschaften“ des aktuellen Kontos). Hier erhält man folgendes Bild:

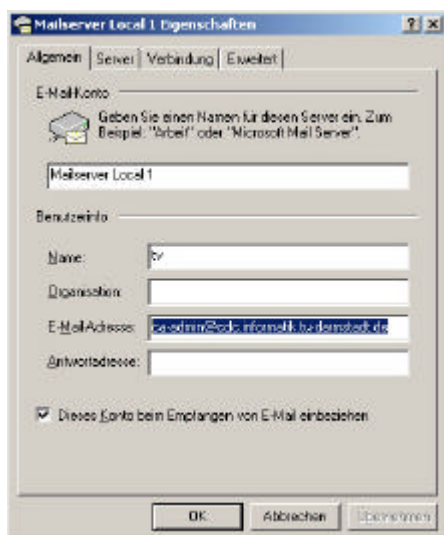


Abbildung 12:
Eigenschaften eines
Kontos bei Outlook

⁵⁵ In diesem Zusammenhang nicht zu verwechseln mit der sog. „Signatur“ in Outlook, die das automatische Anhängen eines Standardtextes an Emails bezeichnet.

Desweiteren ist es notwendig, die Sicherheitseinstellungen des Kontos zu bearbeiten. Dies erfolgt im Menüpunkt „Extras/Optionen/Sicherheit“ (Aufruf von „Einstellungen ändern“). Hier erhält man folgendes Bild:

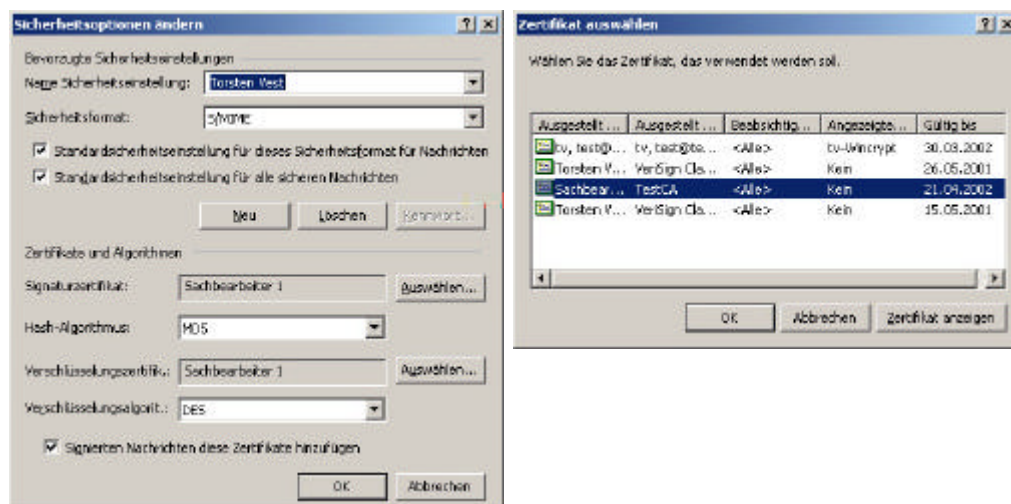


Abbildung 13a, 13b: Ändern der Sicherheitseinstellungen in Outlook

Um das gewünschte Zertifikat zum Signieren und bzw. oder Verschlüsseln zuzuweisen⁵⁶, muss jeweils der Button „Auswählen“ aus Abbildung 13a gewählt werden. In der Auswahlmaske (Abbildung 13b) wird das zur Email-Adresse und Identität passende Zertifikat gewählt.

1.5.2 Workflow ‚Signieren einer Email‘

Mit Outlook ist es möglich, die im ersten Abschnitt vorgestellten, vier typischen Workflows Signieren, Verifizieren, Verschlüsseln und Entschlüsseln durchzuführen. Um beispielsweise eine geschriebene Nachricht zu signieren, muss wie bereits beschrieben ein gültiges Zertifikat zum passenden Konto zugewiesen sein. Zudem muss für die Nachricht die Option „Digitale Signatur ausgehenden Nachrichten hinzufügen“⁵⁷ ausgewählt werden.

⁵⁶ Dieses wird hier als bereits im System installiert vorausgesetzt. Zur Publizierung eines Zertifikates vgl. Kapitel 1.4 und Kapitel 3.4.

⁵⁷ Einstellbar ist dies entweder in der neuen Nachricht im Feld „Optionen“ oder als Standardeinstellung für alle Nachrichten in Outlook unter „Extras/Optionen/Sicherheit“.

Hierdurch wird beim „Senden“ der Nachricht von Outlook die Nachricht entsprechend bearbeitet und die Signatur hinzugefügt. Um diese Signatur zu bestimmen, werden die entsprechenden Funktionen des CryptoAPIs genutzt. Die Bestimmung des exakten Ablaufes erfolgt in Kapitel 3.5.

Ein wichtiger Faktor ist die Auswahl des CSPs durch Outlook.

1.5.3 CSP-Auswahl durch Outlook

Da keinerlei Dokumentation der internen Abläufe in Outlook erhältlich ist, müssen sowohl Auswahlkriterien für den CSP als auch die exakten Funktionsaufrufe durch Tests bestimmt werden (vgl. Kapitel 3.6).

Allerdings sind hierfür die im Zertifikat eingetragenen Informationen Voraussetzung:

- ☞ Informationen über den zugehörigen CSP (Name des CSP),
- ☞ Informationen über die zugehörige Identität und
- ☞ Informationen zum Signaturalgorithmus.

Es ist zu erwarten, dass diese Eintragungen für die Auswahl eines CSPs durch Outlook genutzt werden.

Anmerkung:

Der symmetrische Verschlüsselungsalgorithmus und der benötigte Hashalgorithmus werden in Outlook unter „Extras/Optionen/Sicherheit“ eingestellt.

Außerdem sind diese Angaben nicht zu verwechseln mit folgenden: Im Zertifikat wird auch ein Hash- und Signaturalgorithmus eingetragen, mit denen der Public Key selbst signiert wird. Diese stimmen nicht notwendig mit denen oben genannten überein, sondern sind üblicherweise von anderer kryptographischer Härte.

Fragestellung 5:

Die sich hieraus ergebende Fragestellung zur weiteren Betrachtung ist, wie die Workflows der Applikation auf die vom CSP angebotenen Funktionen im konkreten Fall abgebildet werden und welche Voraussetzungen dafür erfüllt sein müssen.

2 Schnittstellen zur Java-Funktionalität

Als zweiter Schritt nach der Betrachtung der von Windows zur Verfügung gestellten Schnittstellen ist die Schnittstelle zur Java-Funktionalität von Interesse. Es ist das Ziel, von Applikationen an einen CSP gestellte Anforderungen auszuführen, sobald dieser die nötigen Informationen übermittelt. Sowohl die hierzu benötigte Architektur, die ihrerseits mittels der Java Cryptographic Architecture (JCA) auf die entsprechenden, in Java implementierten kryptographischen Provider zurückgreift, als auch die benutzten Schnittstellen müssen analysiert werden. Weiterhin sind die genutzten Objekte (Object Identifier und Zertifikate) und die Abbildung der Aufruf-Semantik des CryptoAPI auf die Java-Funktionen inklusive der hierzu benötigten Schnittstelle, dem Java Native Interface (JNI), im Blickpunkt.

2.1 Architektur der Java-Funktionalität

Da Java eine interpretierte⁵⁸ Programmiersprache ist, wird zur Ausführung der Funktionalität ein Interpreter benötigt, also ein Programm, das den Java-Funktionen eine Ablauf-Umgebung bietet. Dies ist die sogenannte Java-Virtual Machine (Java-VM). Innerhalb dieses Programms wird das Java-Programm gestartet und führt die entsprechenden Aktionen aus. Um von einem anderen Programm - dem CSP - aus auf in Java implementierte Funktionen zurückzugreifen, muss zuerst die Java-VM gestartet werden. Diese bietet die Möglichkeit, auf in ihr enthaltene Daten und Programmteile zuzugreifen. Die in Java benötigten Klassen enthalten die den abstrakten kryptographischen Funktionen analogen Implementierungen für Signieren, Verifizieren, Ver- und Entschlüsseln. Desweiteren sind dies die Initialisierung von Daten, der Zugriff auf die in Java genutzte JCA und damit die hier implementierten Java-Provider des Fachgebietes (CDC) sowie die Zertifikats-Dateien. Diese Dateien sind als Java-Key-Stores implementiert, die jeweils ein dem Benutzer zugeordnetes Zertifikat enthalten. Das genutzte Teilsystem ist in Abbildung 14 gezeigt.

⁵⁸ Dies steht im Gegensatz zu C. Es ist insofern korrekt, da der in Java implementierte Programmcode in einen Zwischencode übersetzt wird, der dann interpretiert wird. Es wird nicht wie in C eine direkt vom Betriebssystem ausführbare Datei (Executable) erzeugt.

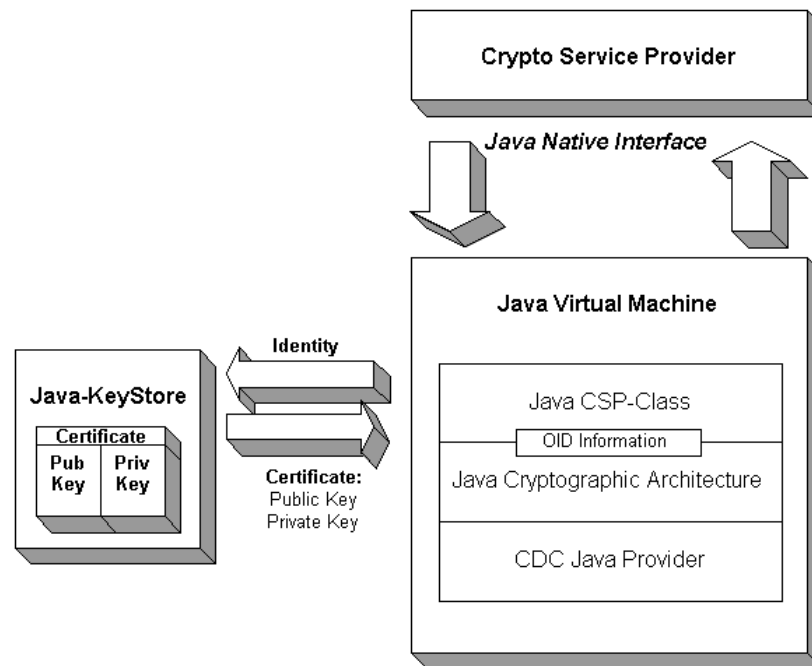


Abbildung 14: Architektur der Java-Funktionalität

Die Interaktion mit dem Crypto Service Provider erfolgt mittels der von der Java-VM angebotenen Schnittstelle, dem JNI. Dieses bietet die Möglichkeit, auf einzelne Funktionen in der Java-VM zuzugreifen, diese unter Angabe von Parametern zu starten und ihre Ausführung zu überwachen⁵⁹. Zudem können sowohl einzelne Daten als auch Arrays von Daten von beiden Seiten aus bearbeitet werden, womit die Übergabe größerer Datenblöcke realisiert werden kann. Informationen hierzu finden sich in [JNI-SPEC, SCHRAMM, SEIPEL].

Interessant ist zudem die Java Cryptographic Architecture. Sie definiert einen Java-Standard zur Implementierung kryptographischer Dienste ähnlich des Windows CryptoAPI. Auch hier werden sogenannte Provider genutzt. Die hier betrachteten sind die vom Fachgebiet bereits implementierten Provider, der CDC-Standard-Provider und der CDC-EllipticCurve-Provider.

Detaillierte Angaben zur JCA können [OAKS] entnommen werden.

⁵⁹ Die Abfrage von Fehlermeldungen – Exceptions in Java – ist hierzu möglich.

2.2 Benutzung der CDC-Provider

Die angeführten Provider der JCA, die CDC-Provider, bieten sowohl Implementierungen der üblichen kryptographischen Verfahren und Objekte (CDC-Standard-Provider) als auch Implementierungen der Verfahren unter Benutzung elliptischer Kurven (CDC-EllipticCurve-Provider). Im Rahmen dieser Arbeit steht die Benutzung dieser Algorithmen im Vordergrund, nicht die Implementierung derselben. Daher wird auf die Erklärung der Funktionsweise verzichtet. Genauere Informationen können der Standard-Literatur (z.B. [BUCHMANN, SCHNEIER, SIEGEN]) entnommen werden; die Benutzung der Provider ist zudem ausführlich in [CDC-STD, CDC-EC] dokumentiert.

Zur Benutzung der Provider kann ein einfaches Beispiel zum Signieren in Java angeführt werden. Die benötigten Schritte sind:

- ☞ Aufrufen der JCA und Hinzufügen des gewählten Providers,
- ☞ Laden eines Keystores,
- ☞ Laden des enthaltenen Zertifikates und
- ☞ Signieren von zuvor eingegebenen Daten⁶⁰.

```
byte meineDaten[] = new byte[100];
String eingabe, ausgabe="          ";
KeyStore ks;
String password="12345678";
char pw[];
String ks_filename="Sachbearbeiter_1.ks";
FileInputStream file;
X509Certificate cert;
PrivateKey pk;
Signature sig;
byte[] signatur;
String sig_string;

// set password
pw = password.toCharArray();

// set data
meineDaten = ...
eingabe = new String(meineDaten);

// add provider to jca
java.security.Security.addProvider(new
    cdc.standard.CDCStandardProvider());
```

⁶⁰ Das Hashen und Signieren ist in der JCA nur ein Schritt, im Gegensatz zum CryptoAPI. Hierauf wird im folgenden noch detailliert eingegangen.

```
try{
    // load keystore with certificate
    ks = KeyStore.getInstance("JKS");

    // open file
    file = new FileInputStream("Sachbearbeiter_1.ks"); // filename
    ks.load(file, pw); // password

    // get certificate
    cert=(X509Certificate) ks.getCertificate("default");
    pk=(PrivateKey) ks.getKey("default", pw);

    // Signatur erzeugen
    sig = Signature.getInstance("MD5withRSA", "CDCStandard");
    sig.initSign(pk);
    sig.update(meineDaten);
    signatur = sig.sign();
    sig_string = new String(signatur)
}
catch (java.security.KeyStoreException myException) // { ... error handling ...}
// catch ( all other errors ...)
```

Code-Beispiel 2: Einfaches Signieren mit der JCA

Dieses Vorgehen lässt sich analog auf die übrigen kryptographischen Grundverfahren anwenden.

Fragestellung 6:

Die sich hieraus ergebende Fragestellung zur weiteren Betrachtung ist, welche Java-Funktionalität im konkreten Fall zum Aufruf durch den CSP benötigt wird und welche Voraussetzungen dafür erfüllt sein müssen.

2.3 Zertifikate, Keys und Object Identifier

Wichtig ist die Rolle der genutzten Objekte, also von Zertifikaten, Keys und Object Identifiern. Wie im vorherigen Beispiel erkennbar, sind Zertifikate in der JCA in sogenannten Key-Store-Dateien gespeichert. Ein solche Datei kann eines oder mehrere Zertifikate beinhalten. Im weiteren werden nur solche Dateien angeführt, die jeweils genau ein Zertifikat enthalten und deren Dateiname dem Namen des Benutzers, und damit der aus dem CryptoAPI vorhandenen Identitätsinformation entspricht⁶¹. Um Zugriff auf das enthaltene Zertifikat zu erhalten, das sowohl Public als auch Private Key beinhaltet, ist die Angabe eines Passwortes⁶² nötig. Dies muss im Gegensatz zum angeführten Beispiel im späteren System vom Benutzer angegeben werden.

```
file = new FileInputStream("Sachbearbeiter_1.ks"); // filename
cert=(X509Certificate) ks.getCertificate("default"); // default user name
pk=(PrivateKey) ks.getKey("default", pw); // password
```

Im folgenden ist zum Zugriff auf einen ausgewählten Algorithmus sowohl ein Namensstring als auch der den Algorithmus zur Verfügung stellende Provider bekannt. Der im Beispiel genutzte Algorithmus MD5withRSA⁶³ ist im CDC-Standard-Provider implementiert.

```
sig = Signature.getInstance("MD5withRSA", "CDCStandard");
```

Ziel ist es, diese Namens-Informationen, falls möglich, dynamisch zu erzeugen und zuzuweisen.

Fragestellung 7:

Die sich hieraus ergebende Fragestellung zur weiteren Betrachtung ist, wie auf Seiten der Java-VM Object Identifier, Zertifikate und Keys im konkreten Fall genutzt werden und welche Voraussetzungen dafür erfüllt sein müssen.

⁶¹ Die Beschränkung auf solche Key-Store-Dateien ist nicht systemkritisch. Es wäre ohne weiteres möglich, innerhalb einer Datei mit vielen Zertifikaten anhand des Namens das Gesuchte auszuwählen.

⁶² Zusätzlich wird der Name des Zertifikates benötigt. Da nur Key-Stores mit genau einem, dem gewünschten Zertifikat, genutzt werden, ist dies der Default-Name und damit hinfällig.

⁶³ RSA-Signaturalgorithmus unter Benutzung des MD5-Hashalgorithmus.

2.4 Das Java Native Interface

Das Java Native Interface ist die von Java zur Verfügung gestellte Schnittstelle, mittels derer in anderen Programmiersprachen⁶⁴ erstellte Programme auf eine Java-VM und die in ihr enthaltenen Funktionen und Datenobjekte zugreifen können. Die benötigte, hier zur Verfügung gestellte Funktionalität lässt sich in drei Gruppen gliedern:

- ☞ Starten einer Java-VM und Initialisieren des zugehörigen Environments⁶⁵,
- ☞ Ermitteln von Zugriffshandles auf Funktionen und Datenobjekte und
- ☞ Aufrufen von Funktionen und Bearbeiten von Datenobjekten.

Diese Funktionalität wird benötigt, um dem in der Sprache C programmierten CSP den Aufruf der in Java implementierten CDC-Provider zu ermöglichen.

Die wesentlichen Elemente des JNI sind Handles und die mit ihnen verknüpften Schnittstellenfunktionen⁶⁶. Als erster Schritt muß stets eine Java-VM gestartet und initialisiert werden. Die hierzu benötigte JNI-Funktion `JNI_CreateJavaVM` benutzt als Eingangsparameter die Pfadinformation zum sogenannten Java Runtime Environment. Dies bezeichnet die für Windows direkt ausführbaren Java-Interpreter-Dateien und die zugehörigen Bibliotheken⁶⁷. Zudem ist der Zugriffspfad auf die genutzten Java-Klassen anzugeben. Als Rückgabewert erhält das aufrufende Programm ein Handle auf die Java-VM und ein weiteres auf das zugehörige Environment, die für alle weiteren Schritte benötigt werden.

Dies wird verdeutlicht im folgenden Code-Beispiel 3, das von der in Kapitel 3 beschriebenen CSP-Implementierung genutzt wird (die Funktion `CDC_Java_StartVM`). Diese Handles sind dauerhaft, d.h. sie enthalten gültige Werte, solange das Programm die Java-VM nicht beendet und diese nicht aufgrund externer Fehler beendet wird. In der Java-VM auftretende Fehler beeinflussen die Handles nicht.

⁶⁴ Hier wurde die Programmiersprache C verwendet.

⁶⁵ Die Umgebungsinformation der Java-VM.

⁶⁶ Die Funktionen des JNI, nicht die durch sie referenzierten Java-Funktionen.

⁶⁷ Die Installation dieser Komponenten ist beschrieben in Kapitel 3.5 und im Anhang („Cookbook“).

```

#define USERCLASSPATH "C:\\java\\;cdcstd.jar;codec.jar;jce.jar"

BOOL CDC_Java_StartVM(void)
{
    BOOL rc=CRYPT_FAILED;           // return code
    char message[MAX_MESSAGE];     // message text

    JNIEnv *env=NULL;             // pointer to the JNI environment
    JavaVM *jvm=NULL;            // pointer to the actual vm
    JavaVMInitArgs vm_args;       // arguments for vm's init
    JavaVMOption options[3];      // options for starting vm
    char classPath[1024];         // path for our classes
    jint res = (jint) 0;          // error code (jint = long)

    // set the flags for starting the vm

    // build option values for java vm
    sprintf(classPath, "-Djava.class.path=%s", USERCLASSPATH);
    options[0].optionString = "-Djava.compiler=NONE";// disable JIT
    options[1].optionString = classPath;
    options[2].optionString = "-verbose:";

    // build init args for java vm
    vm_args.version=JNI_VERSION_1_2;
    vm_args.options=options;
    vm_args.nOptions=3;
    vm_args.ignoreUnrecognized=TRUE;

    // create the java vm
    res = JNI_CreateJavaVM(&jvm, (void **) &CDC_JNIEnv, &vm_args);
    if (res < (jint) 0)
    {
        sprintf(message, "Can't create Java VM, sorry...
            Error-code='%d'", res);
        CDC_JavaMessageBox(NULL,message,"Error",0);
        return CRYPT_FAILED;
    }

    return rc;
}

```

Code-Beispiel 3: Starten einer Java-VM mit dem JNI (gekürzt)

Um die so erzeugte Java-VM zu nutzen, wird mittels der JNI-Funktion `FindClass` in Abhängigkeit vom Environment auf die Klasse mit der Java-Implementierung von Funktionen und Objekten zugegriffen. Im Gegensatz zu den beiden vorherigen ist das hierfür genutzte Handle zeitlich nur begrenzt gültig⁶⁸ und wird zur Sicherheit bei jedem Zugriff neu erzeugt.

Sowohl Funktionen als auch Datenobjekte der Klasse können mittels des Klassen-Handles über ihren Namen referenziert werden. Als Rückgabewert der hierzu

⁶⁸ Die Java-VM kann die Klasse selbständig wieder entladen.

genutzten Funktionsgruppe wird eine ID⁶⁹ mit zeitlich begrenzter Gültigkeit erzeugt. Verdeutlicht wird dies anhand Code-Beispiel 4.

```

BOOL CDC_Java_GetMethodID(JNIEnv *Env,
                          char *MethodName,
                          char *TypeSig,
                          jclass *Class,
                          jmethodID *MethodID)
{
    char message[MAX_MESSAGE];           // message text
    BOOL rc=FALSE;                       // return code
    jthrowable exception;

    // find class csp.java
    *Class = (*Env)->FindClass(Env, CSP_CLASSNAME);
    if (*Class==0)
    {
        sprintf(message, "Java VM: Can't find class '%s.class'.",
                CSP_CLASSNAME);
        CDC_JavaMessageBox(NULL,message,"Error",0);

        exception = (*Env)->ExceptionOccurred(Env);
        if (exception != NULL)
        {
            (*env)->ExceptionDescribe(env);
        }
        return(CRYPT_FAILED);
    }
    else
    {
        sprintf(message, "Java VM: class found.");
        CDC_JavaMessageBox(NULL,message,"Ok",0);
    }

    // get method id
    *MethodID = (*Env)->GetStaticMethodID(Env, *Class, MethodName, TypeSig);
    if (*MethodID==0)
    {
        sprintf(message, "Java VM: Can't find method.");
        CDC_JavaMessageBox(NULL,message,"Error",0);

        exception = (*Env)->ExceptionOccurred(Env);
        if (exception != NULL)
        {
            (*env)->ExceptionDescribe(env);
        }
        return(CRYPT_FAILED);
    }
    else
    {
        sprintf(message, "Java VM: method found.");
        CDC_JavaMessageBox(NULL,message,"Ok",0);
    }

    return CRYPT_SUCCEED;
}

```

Code-Beispiel 4: ID-Bestimmung zum Funktionsaufruf mit dem JNI

⁶⁹ Eindeutiger Identifier-Wert.

Das Vorgehen ist analog für Datenobjekte. Allerdings wird abhängig vom Typ der Funktion bzw. des Datenobjektes eine entsprechende JNI-Funktion aus der Funktionsgruppe gewählt. Desweiteren wird eine passende sogenannte Typ-Signatur⁷⁰ benutzt. Informationen zu verfügbaren Funktionen und die Erstellung der Typ-Signatur sind beschrieben in [JNI-SPEC].

Sobald diese ID zur Verfügung steht, kann die gewünschte Funktion folgendermaßen aufgerufen werden:

```
// call java signing method
jrc = (*Env)->CallStaticIntMethod(Env, Class, MethodID);
```

Code-Beispiel 5a: Funktionsaufruf mit dem JNI

Analog erfolgt das Benutzen von in Java abgelegten Datenobjekten.

Zur Demonstration des Vorgehens wird der Zugriff auf einen Datenpuffer in Code-Beispiel 5b beschrieben. Wie im obigen Beispiel 4 wird eine ID zum Zugriff auf das Datenobjekt erzeugt. Diese wird genutzt, um das Datenobjekt direkt anzusprechen und seinen Inhalt zu lesen. Hierzu wird dieser vom JNI als Kopie in einem Block⁷¹ zur Verfügung gestellt und kann über einen gewöhnlichen Zeiger in C manipuliert werden. Nach Bekanntmachen der Beendigung des Zugriffs an das JNI werden die Daten von diesem in die Java-VM übertragen und können von dort wieder genutzt werden.

Da das JNI in dieser Arbeit als Hilfs-API zur Abbildung von C-Funktionalität auf Java-Funktionalität benutzt wird und die Implementierung analog der beschriebenen Beispiele erfolgt, wird dies im weiteren nicht als eine der Hauptfragestellungen betrachtet, sondern es werden lediglich an den entsprechenden Stellen die nötigen Informationen bereitgestellt.

⁷⁰ Nicht zu verwechseln mit der Signatur von Daten mittels kryptographischer Funktionen. Die Typ-Signatur bestimmt die Datentypen aller Ein- und Ausgangsparameter einer Java-Funktion oder den Datentyp eines Objektes in Java.

⁷¹ Serialisiert. Dies beeinflusst die Objektverwaltung der Java-VM und muß von dieser unterstützt werden.

```

BOOL CDC_Java_SetInBuffer(JNIEnv *Env,
                          jclass Class,
                          BYTE *pbData,
                          DWORD iDataSize)
{
    char message[MAX_MESSAGE];           // message text
    BOOL rc=CRYPT_SUCCEED;
    jfieldID FieldID=0;
    jthrowable exception;
    jbyte *InBuf;
    jobject ArrayObject;
    jbyte elem='Z';
    jboolean isCopy;
    DWORD i;                             // loops

    // get the field id
    FieldID = (*Env)->GetStaticFieldID(Env, Class, "InBuffer", "[B");
    if(FieldID==0)
    {
        sprintf(message, "Java VM: Can't find field 'InBuffer'.");
        CDC_JavaMessageBox(NULL,message,"Error",0);

        exception = (*Env)->ExceptionOccurred(Env);
        if (exception != NULL)
        {
            (*env)->ExceptionDescribe(env);
        }
        return(CRYPT_FAILED);
    }
    else
    {
        sprintf(message, "Java VM: field 'InBuffer' found.");
        CDC_JavaMessageBox(NULL,message,"Ok",0);
    }

    // get the field's object
    ArrayObject = (*Env)->GetStaticObjectField(Env, Class, FieldID);

    // get the array content:
    isCopy = JNI_TRUE;
    InBuf = (*Env)->GetByteArrayElements(Env, ArrayObject, &isCopy);
    // new was made with declaration in java with fixed size !
    // so we get the clean array from java, change it and write it back

    // change the array content
    for(i=0; i<iDataSize; i++)
        InBuf[i]=pbData[i];

    // set array content
    (*Env)->SetByteArrayRegion(Env, ArrayObject, 0, 10, InBuf);

    // set the field's object
    (*Env)->SetStaticObjectField(Env, Class, FieldID, ArrayObject);

    // Release the handle, and destroy the local buffer
    (*Env)->ReleaseByteArrayElements(Env, ArrayObject, InBuf, 0);

    // set the size of the InBuffer
    rc = CDC_Java_SetInBufferSize(Env, Class, iDataSize);

    sprintf(message, "Java VM: after set InBuffer %d", iDataSize);
    CDC_JavaMessageBox(NULL,message,"Ok",0);

    return rc;
}

```

Code-Beispiel 5b: Datenmanipulation mit dem JNI

2.5 Konzept der Abbildung von CryptoAPI auf JCA

Um die beiden Ansätze des Bereitstellens kryptographischer Funktionalität, CryptoAPI und JCA, zu verbinden, muss untersucht werden, ob und wie diese aufeinander abgebildet werden können. Ansatzpunkt hierzu ist eine ideale 1:1-Abbildung der jeweiligen Verfahren und Funktionen aufeinander.

Die 1:1-Abbildung sieht vor, dass die vier betrachteten kryptographischen Verfahren Signieren, Verifizieren, Ver- und Entschlüsseln in beiden Schnittstellen identisch behandelt werden. Im allgemeinen muß dies möglich sein, da diese Verfahren an sich durch die benutzten Algorithmen und deren Funktionsweise festgelegt sind. Es ist allerdings zu beachten, dass für das zum Signieren und Verifizieren benötigte Hashen ein grundsätzlicher Unterschied besteht. Im CryptoAPI wird Hashen als eigenständiges Verfahren betrachtet. Im Gegensatz hierzu ist es in der JCA als Bestandteil des Signierungsprozesses vorgesehen. Da aber auch im CryptoAPI Hashen unabdingbare Voraussetzung des Signieren ist und damit immer vorher ausgeführt wird, können hier die zwei Verfahren auf das beide umfassende der JCA abgebildet werden⁷².

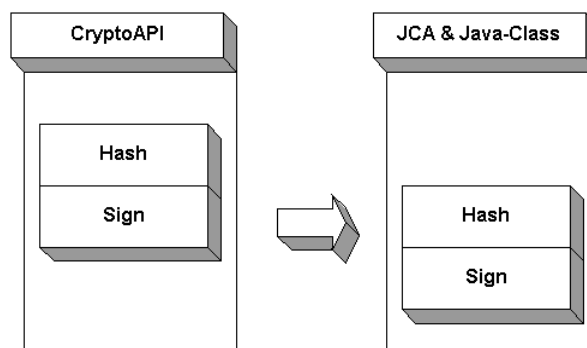


Abbildung 15a: Abbildung des Signierens

Voraussetzung hierfür ist allerdings, dass auf der CSP-Seite die Anfrage durch die Hash-Funktion und die eingegebenen Parameter zwischengespeichert werden. Entsprechend wird diese Funktion, obwohl noch nicht in Java ausgeführt, der Applikation gegenüber als „erfolgreich ausgeführt“ gekennzeichnet. Beim

⁷² Es ist möglich, mittels der Signierungsfunktion der JCA auf die Werte des Hashes zuzugreifen. Damit kann auch eine CryptoAPI Hashfunktion ohne anschließendes Signieren realisiert werden.

folgenden Aufruf des Signierens werden alle erforderlichen Daten zusammengestellt und der Signierungsfunktion übergeben.

Verifizieren ist analog zu implementieren. Die prinzipiell gleiche Vorgehensweise wie beim Signieren wird ausgeführt⁷³. Als weiterer Schritt wird der Vergleich von bisherigem und neuem Hash hinzugefügt⁷⁴.

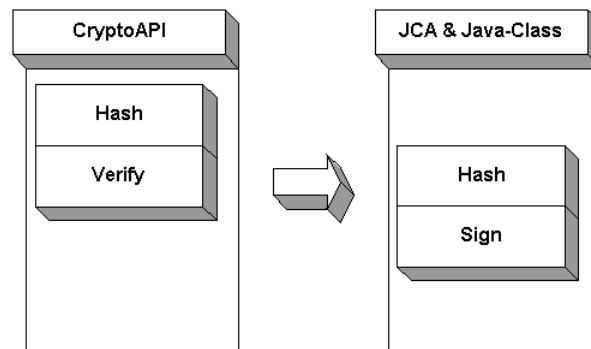


Abbildung 15b: Abbildung des Verifizierens

Die beiden übrigen Verfahren können hingegen direkt abgebildet werden:

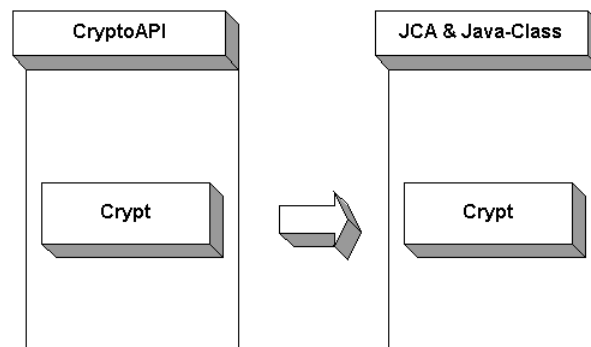


Abbildung 15c: Abbildung des Verschlüsseln

⁷³ Es wird auf der Java-Seite auch die Signieren-Funktion benutzt. Alternativ kann auch die Verifizieren-Funktion des Java-Providers genutzt werden.

⁷⁴ Dieser Vergleich ist nicht in der Grafik gezeigt, da er nur die letzte Teilaktion des Verify-Verfahrens im CryptoAPI darstellt und nicht durch einen zusätzlichen Funktionsaufruf durch die Applikation ausgelöst wird.

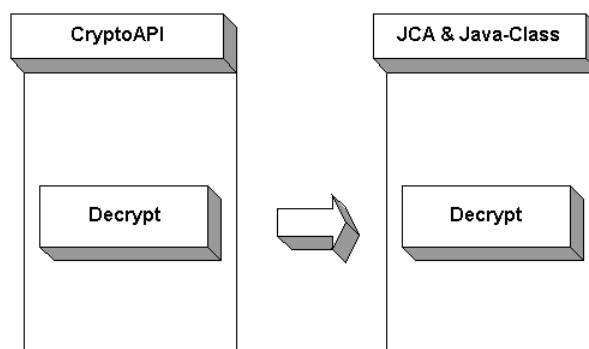


Abbildung 15d: Abbildung des Entschlüsselns

Zu beachten ist allerdings, dass nur Verschlüsselungsaufrufe mit asymmetrischen Algorithmen wie in Kapitel 1.5 beschrieben diesen Punkt erreichen. Zudem wurde darauf verzichtet, mögliche Mehrfachaufrufe der Funktionen zu kennzeichnen. Die Verfahren Hashen, Ver- und Entschlüsseln können auf beiden Seiten in Teilschritten abgearbeitet werden, d.h. die Ursprungsdaten werden in Blöcke zerlegt, die jeder für sich das jeweilige Verfahren durchlaufen.

Auf die Auswahl der entsprechenden Algorithmen anhand der Object Identifier und der Zertifikate und Keys anhand der Identitätsinformationen wurde bereits im zweiten Abschnitt des Kapitels eingegangen. Die konkrete Implementierung hierzu ist im folgenden Kapitel beschrieben.

Weiterhin sind, wie in den vorherigen Abschnitten beschrieben, Schritte zum Starten (Funktion `startVM`) und Initialisieren der Java-VM (`csp::CSP_Init`) und Hinzufügen der CDC-Provider (`csp::CSP_AddProv`) in die JCA nötig. Abhängig von den Identitätsinformationen wird die Key-Store-Datei mit dem entsprechenden Zertifikat geladen (`csp::CSP_GetCert`) und im Anschluss das eigentliche Verfahren, beispielsweise das Signieren (`csp::CSP_Sign`), gestartet. Nach Abschluss des Verfahrens ist es unter Umständen nötig, vorhandene Daten ordnungsgemäß freizugeben (`csp::CSP_Exit`). Die detaillierte Reihenfolge der Aufrufe von CSP-Funktionen durch die Applikation muss hierzu noch bestimmt werden⁷⁵.

Ein Überblick über das zu erwartende Funktionsdiagramm gibt Abbildung 16.

⁷⁵ Vgl. Kapitel 3.6.

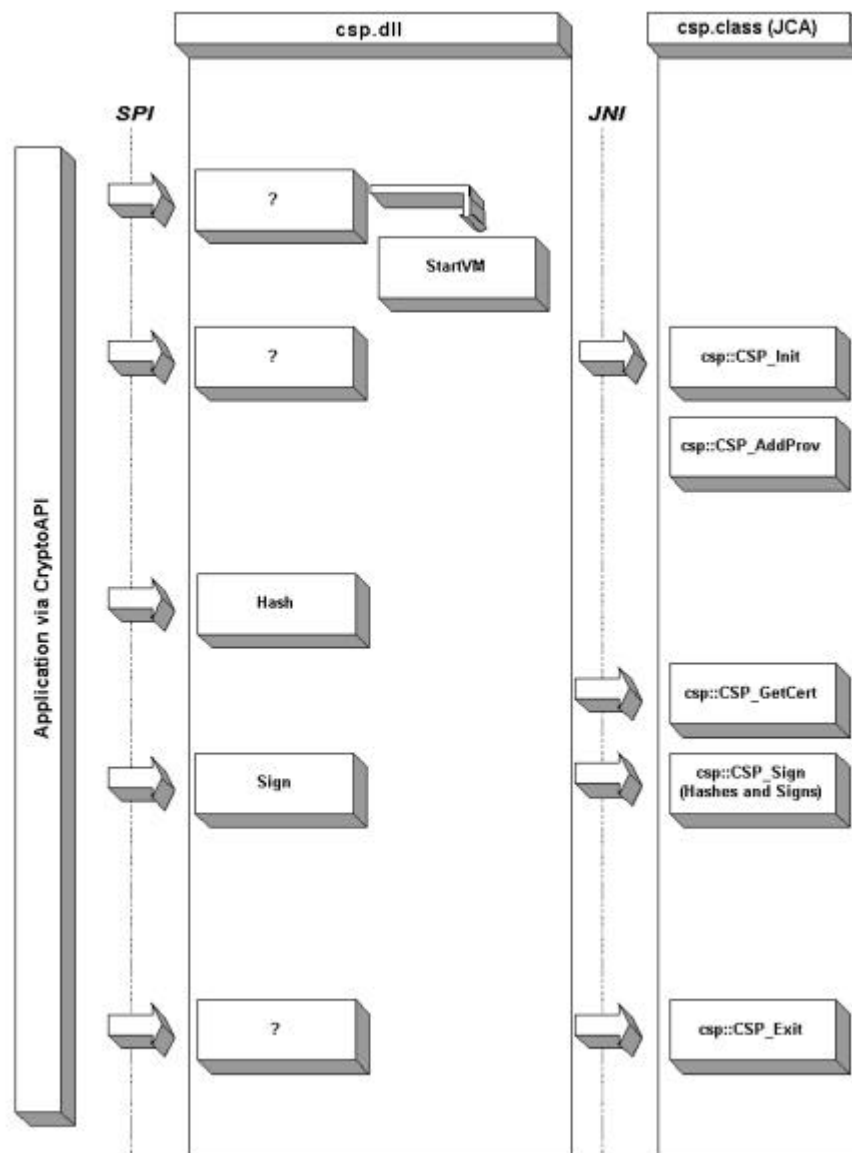


Abbildung 16: Abbilden der CryptoAPI-Semantik auf die JCA-Semantik⁷⁶

Fragestellung 8:

Die sich hieraus ergebende Fragestellung zur weiteren Betrachtung ist, wie die Workflows der Applikation auf die von der JCA angebotenen Funktionen im konkreten Fall abgebildet werden und welche Voraussetzungen dafür erfüllt sein müssen.

⁷⁶ Die Fragezeichen bezeichnen noch zu klärende Abläufe und Datenflüsse.

3 Implementierung

Die Implementierung der Einzelkomponenten beschreibt, wie diese unter Berücksichtigung der aufgeworfenen Fragestellungen umzusetzen sind und wie sie im Gesamtsystem interagieren.

3.1 Das Gesamtsystem

Die sich bisher ergebenden Anforderungen an das Gesamtsystem können folgendermaßen zusammengefasst werden:

☞ Implementierung eines CSPs unter Berücksichtigung von:

- Schnittstellen zum CryptoAPI (SPI, Entry Point-Funktionen)
- Interner Implementierung der benötigten Objekte und Workflows
- Abbildung der CryptoAPI-Semantik auf die JCA-Semantik
- Schnittstelle zwischen der CSP-DLL (in C) und der JCA (mittels des JNI),

☞ Implementierung des Registrierungsprogramms für Object Identifier und

☞ Implementierung des Publizierungsprogrammes für Zertifikate.

Dies erfolgt unter Berücksichtigung der

☞ Installation aller Komponenten im System und von

☞ Testen und Implementieren der Aufrufsemantik von Outlook.

Ein Überblick über das zu erwartende Gesamtsystem wird in Abbildung 17 gegeben.

Zusammengefasst sind dies der CSP in Interaktion mit Applikation mittels des CryptoAPI, der Java-Schnittstelle mittels des JNI, das Zusammenspiel mit der Object Identifiern und Zertifikaten. Es muss jeweils Funktionsweise und Datenfluß bestimmt werden. Hierzu werden im folgenden die einzelnen Bestandteile dieses Gesamtsystems betrachtet.

Die Anforderungen und ihre Implementierung werden in den folgenden Abschnitten detailliert ausgeführt.

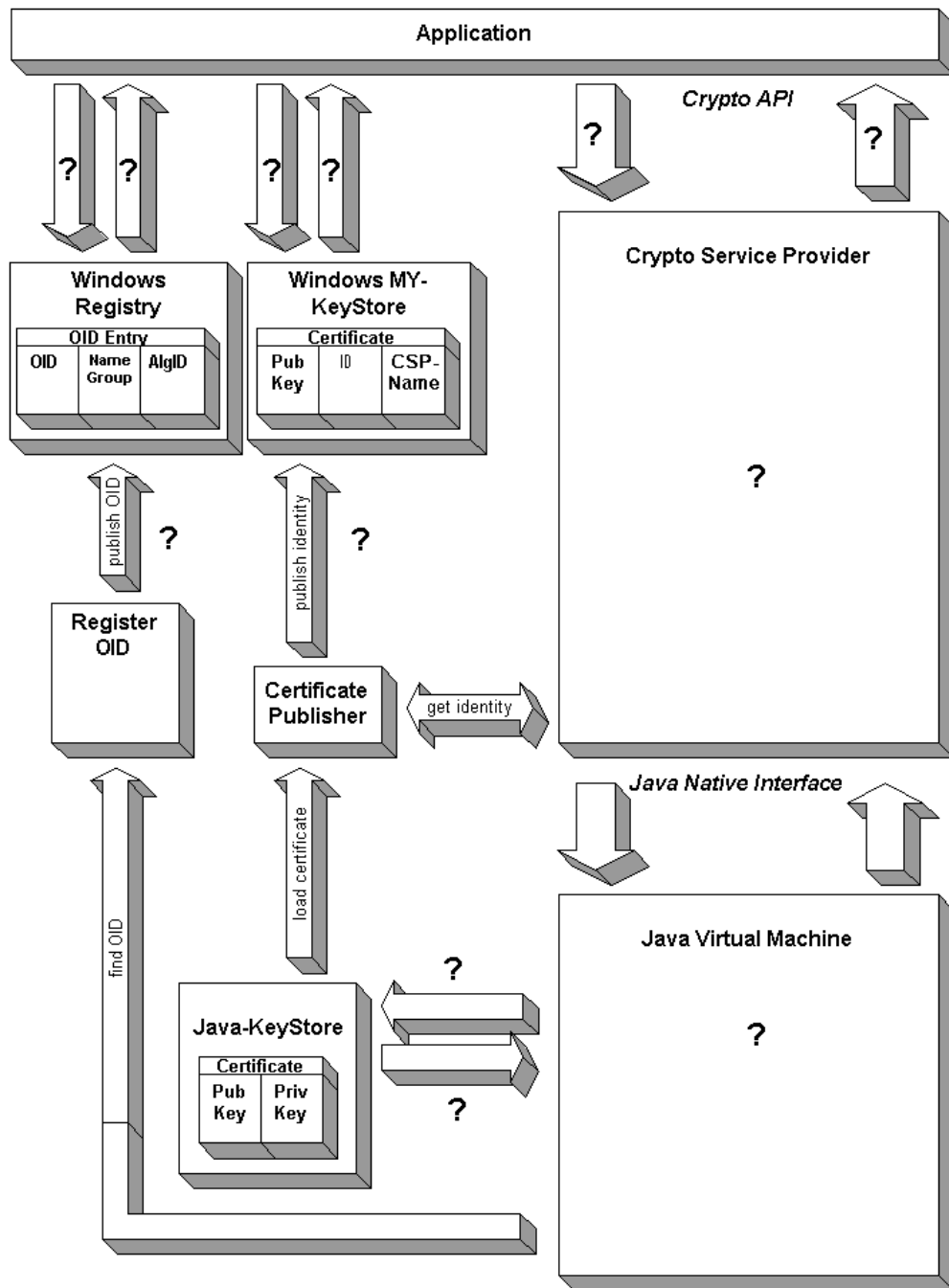


Abbildung 17: Geplantes Gesamtsystem⁷⁷

⁷⁷ Die Fragezeichen bezeichnen noch zu klärende Abläufe und Datenflüsse.

3.2 Der Crypto Service Provider

Um einen vollständigen CSP zu implementieren, müssen die in Kapitel 1 genannten Eigenschaften vollständig unterstützt werden. Dies sind die Erfüllung der prinzipiellen Architektur, Implementierung der bezeichneten Entry-Point Funktionen inklusive der Bearbeitung aller unterschiedenen Fälle (Flags), der weiteren CSP-internen Schichten und der intern genutzten Datenstrukturen. Beachtet werden müssen hierbei die drei Grundsätze der Architektur⁷⁸.

3.2.1 Architektur und Implementierung

Die Architektur kann anhand Abbildung 18 beschrieben werden:

Es erfolgt eine Aufteilung des CSPs in Schichten zur Bearbeitung der einzelnen Aufgaben. Dies sind:

✍ **DLL Entry Point Layer:**

Hier werden alle für einen CSP vorgeschriebenen Entry-Point Funktionen implementiert. Anhand der Unterscheidung der Einzelaufgaben durch die Flags wird jeweils eine entsprechende Funktion des Core Functionality Layers aufgerufen.

✍ **Core Functionality Layer:**

Hier werden die im CSP implementierten Aufgaben ausgeführt. Abhängig von der Art der Aufgabe werden eine oder mehrere der Funktionen des Implementation Layers aufgerufen, bis alle Teilaufgaben abgearbeitet sind und alle Zwischenergebnisse vorliegen. Danach werden die Ergebnisse an den höherliegenden Layer weitergegeben.

✍ **Implementation Layer:**

Dieser Layer ist in vier Bereiche aufgeteilt. Im Falle von Java-Algorithmen-Aufrufen werden die Daten zum Aufruf der Java-Funktionen zusammengestellt. Anschliessend wird jeweils eine Funktion des Java Functionality Layer aufgerufen und die Parameter übergeben. In den anderen drei Fällen zur CSP-internen Datenbearbeitung, werden die anstehenden Aufgaben direkt ausgeführt. Danach werden die Ergebnisse an den höherliegenden Layer weitergegeben.

⁷⁸ Wie in Kapitel 1.2.1 beschrieben

Java Functionality Layer

An dieser Stelle erfolgt die Vorbereitung der Daten zur Übergabe an das JNI. Entsprechend der benötigten Java-Funktionen werden die zugehörigen Funktionen des Java Calling Layer genutzt. Danach werden die Ergebnisse an den höherliegenden Layer weitergegeben.

Java Calling Layer

Diese Schicht führt die notwendigen Aufrufe der Java-Funktionen mittels des JNI direkt aus. Die zuvor erhaltenen Parameter werden übergeben und die Ergebnisse an den höherliegenden Layer zurückgegeben.

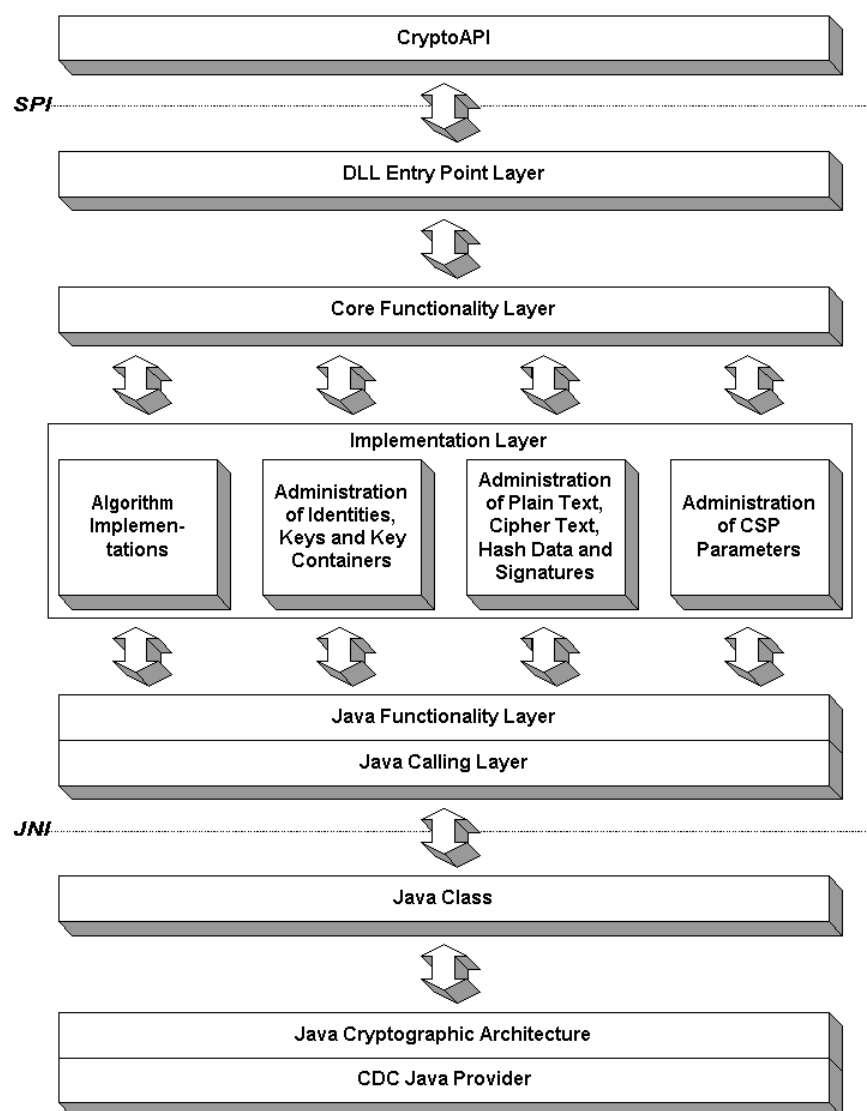


Abbildung 18: Architektur des CDC-Crypto Service Providers

Es ist zu betonen, dass *alle* Entry-Point Funktionen implementiert werden müssen. Im Gegensatz dazu müssen nicht alle der möglichen Werte von Flags ausgewertet werden⁷⁹. Es genügt durchaus, nicht gewünschte Aktionen mit Setzen von Fehlercode und Rückgabe eines negativen Return Codes abzuweisen. Dies ist unter Umständen vom gewählten Providertyp abhängig⁸⁰.

Die Implementierung eines CSPs erfordert die Erstellung eigener Datenstrukturen, z.B. zur Speicherung von Keys und Key Containern, Identitätsinformationen, Eingabedaten, Hashwerten, Signaturen usw. In der hier beschriebenen Implementierung werden keine Private Key-Daten gespeichert, da diese im CSP nicht zur Verfügung stehen. Durch Übergabe der Identitätsinformation an die Java-Schnittstelle wird in der Java-Funktionalität der Private Key zur Identität ermittelt. Er verlässt die Java-Virtual Machine nicht.

Überlegung zum Typ des CSPs in der Implementierung:

Um zu Applikationen kompatibel zu sein, kann es von Vorteil sein, den Microsoft-Standard-Typ zu unterstützen. Hieran werden allerdings besondere Anforderungen gestellt:

Für Provider des Types RSA_PROV_FULL oder PROV_RSA_SIG ist bei den symmetrischen Schlüsselalgorithmen die Unterstützung von MD5- und SHA1-Hash-Algorithmen und von RC2 (Blockmode) und RC4 (Streammode) Voraussetzung, da diese für die symmetrischen Session Keys benötigt werden.

Bei den asymmetrischen sind dies der RSA Public Key-Algorithmus sowohl für Verschlüsselung als auch für Signatur und Schlüsselaustausch (Key Exchange, was hier nicht weiter betrachtet wird).

Die Schlüssellängen sind dabei allerdings *nicht* vorgeschrieben, so daß auch längere als die von Microsoft selbst benutzten 512-Bit Key Paare benutzt werden können⁸¹.

Es hat sich durch entsprechende Tests⁸² herausgestellt, dass eine Erweiterung um weitere symmetrische Algorithmen nicht ohne weiteres vorgenommen werden kann.

⁷⁹ Vgl. Abschnitt ‚Implementierung der Funktionsprototypen‘.

⁸⁰ Da manche der Typen nach Definition die Erfüllung festgelegter Standards erfordern.

⁸¹ Eine Einschränkung ist allerdings, dass die CSP-Funktion zur Verifizierung der Signatur Schlüssellängen bis 2048-Bit unterstützen muss. Dies kann unter Umständen dazu führen, dass

Die genutzte Testapplikation Microsoft Outlook nutzt nur eine festgelegte Auswahl von symmetrischen Algorithmen, auch wenn der CSP weitere zur Verfügung stellt und diese in der von ihm abfragbaren Liste verfügbarer Algorithmen angibt. Diese Information ist in Outlook unveränderbar festgelegt. Zudem nutzt Outlook nur für diese Algorithmen nicht einen mittels des Zertifikates ausgewählten Provider, sondern den für den Typ RSA_FULL_PROV (Typ 1) festgelegten Default-Provider. Es ist möglich, mittels der zugehörigen CryptoAPI-Funktion⁸³ einen anderen als den standardmäßig eingetragenen „Microsoft Base Crypto Service Provider v1.0“ als Default-CSP einzutragen. Dies bedeutet für einen eigenen CSP, dass er diese Algorithmen exakt analog dem eigentlichen Default-CSP implementieren muss. Hiervon wird abgeraten, da dies einen tiefen Eingriff in Windows darstellt⁸⁴.

Installation des CSPs:

Um einen CSP unter Windows bekannt zu machen, muss er mit seinem Namen und Typ innerhalb der Windows-Registry eingetragen werden. Hierzu wird das im CSPDK angegebene Programm benutzt. Entscheidend ist hierbei, daß benutzter Namensstring⁸⁵ und Typ identisch benutzt werden, und zwar auch von der Applikationsseite. Der genutzte Programmcode ist wiedergegeben in Anhang A.1.

hardwarebasierte CSPs (Smartcard-CSPs) zusätzliche Algorithmen in Software ausführen müssen, falls die Karten die notwendige Länge nicht unterstützen.

⁸² Vgl. Kapitel 3.6.

⁸³ Die Funktion ist ‚CryptSetProvider‘ mit entsprechendem Namen und Typ als Parameter.

⁸⁴ Einfache Tests hierfür ergaben, dass Windows 2000 diesen z.B. zur Nutzung des Dateisystems mittels Dateimanager und zur Installation von Treibern und Software nutzt. Falls der CSP nicht exakt implementiert ist, können beispielsweise keine weiteren Treiber mehr installiert werden.

⁸⁵ Dieser ist auf maximal 64 Zeichen festgelegt.

3.2.2 Implementierung der Entry Point-Funktionen

Hauptbestandteil der Implementierung sind die Entry Point-Funktionen der Schicht „DLL-Entry Point Layer“.

Die Vorgehensweise ist für alle identisch. Es wird anhand eingehender Flagwerte unterschieden, welche Aktion ausgeführt werden soll. Dazu müssen alle anderen Eingangsparameter dahingehend überprüft werden, ob die zur Aktion nötigen Informationen korrekt bereitgestellt werden. Falls dies nicht der Fall ist, muss die entsprechende Fehlermeldung gesetzt werden und der Fehler an die Applikation gemeldet werden. Falls alle Informationen korrekt vorliegen, wird die der Aktion entsprechende Funktion der Funktionalitätsschicht ausgeführt.

Als Beispiel der Implementierung der Entry Point-Funktionen wird hier die Funktion CPAcquireContext angeführt.

```

BOOL WINAPI CPAcquireContext(OUT HCRYPTPROV *phContainer,
                             IN OUT CHAR *pszContainer,
                             IN DWORD dwFlags,
                             IN PVTTableProvStruc pVTable)
{
    char message[MAX_MESSAGE];           // message text
    BOOL rc;                             // return code
    char string[MAX_MESSAGE];

    rc = CRYPT_SUCCEED;

    sprintf( message, "CPAcquireContext: IN");
    CDC_MessageBox(NULL,message,"In",0);

    // switch tasks depending on dwFlags given by application
    switch (dwFlags)
    {
        case CRYPT_VERIFYCONTEXT:
            if(pszContainer!=NULL)
            {
                strcpy(string, pszContainer);

                rc = CDC_AcquireContext_WakeContainer(string);

                // do the java
                rc = CDC_Java_StartVM();
                rc = CDC_Java_Init();
            }
            else
            {
                // do nothing
                sprintf( message, "CPAcquireContext:
                    CRYPT_VERIFYCONTEXT : No Container Name");
                CDC_MessageBox(NULL,message,"In",0);
            }
            break;
    }
}

```

```

        case CRYPT_NEWKEYSET:
            rc = CDC_AcquireContext_NewContainer(phContainer,
                                                pszContainer, dwFlags);
            break;

        case CRYPT_MACHINE_KEYSET:
            sprintf( message, "CPAcquireContext: CRYPT_MACHINE_KEYSET
                is not supported");
            CDC_MessageBox(NULL,message,"In",0);
            SetLastError(NTE_BAD_FLAGS);
            rc = CRYPT_FAILED;
            break;

        case CRYPT_DELETEKEYSET:
            sprintf( message, "CPAcquireContext:CRYPT_DELETEKEYSET");
            CDC_MessageBox(NULL,message,"In",0);

            rc = CDC_AcquireContext_DelContainer(phContainer,
                                                pszContainer, dwFlags);
            break;

        case CRYPT_SILENT:
            CDC_CSPCryptSilent=TRUE;
            sprintf( message, "CPAcquireContext: CRYPT_SILENT");
            CDC_MessageBox(NULL,message,"In",0);
            break;

        case 0L:
        default: // sollte eigentlich NULL sein
            sprintf( message, "CPAcquireContext: dwFlags='%x'",
                    dwFlags);
            CDC_MessageBox(NULL,message,"In",0);
            SetLastError(NTE_BAD_FLAGS);
            rc = CRYPT_FAILED;
            break;
    }

    // set statics
    sprintf(CDC_CSPName, CSP_NAME);
    CDC_CSPNameLen= (int) strlen(CDC_CSPName);
    CDC_CSPVersion = CSP_VERSION;
    CDC_CSPProvType = CSP_TYPE;
    CDC_CSPImpType = CSP_IMPTYPE;

    return(rc);
}

```

Code-Beispiel 6 : Implementierung der CSP-Funktion CPAcquireContext

Anmerkung: Zusätzliche Dokumentation des Codes und die Aufrufe von Message-Boxen zur Ausgabe bei Tests sind im Beispiel aus Platzgründen gekürzt.

Die folgende Tabelle führt die im Beispiel ausgeführten Aktionen auf, unterschieden anhand der Werte des Flagparameters ‚dwFlags‘:

C.1 CPACquireContext mit dwFlags=CRYPT_VERIFYCONTEXT		
In	pszContainer=NULL oder pszContainer=<Name>	Key Container-Name
Out	pszContainer=NULL phContainer=NULL oder pszContainer=<Name> phContainer=<Handle>	Kein Rückgabewert Oder Name und Handle des Key Containers
Call	CDC_AcquireContext_WakeContainer CDC_Java_StartVM	Zugriff auf den Key Container Starten der Java-VM
Error	NTE_BAD_KEYSET NTE_BAD_KEYSET_PARAM NTE_KEYSET_ENTRY_BAD NTE_NO_MEMORY	Key Container kann nicht geöffnet werden PszContainer ist unbekannter Name Key Container ist defekt Es steht kein Speicher zur Verfügung
Erzeugen eines Handles auf CSP bzw. Key Container.		

Tabelle 5a: Beschreibung der Implementierung der Funktion CPACquireContext

Die Applikation erhält ein Handle auf den CSP.

- a) Falls der Key Container-Name keinen Wert enthält (NULL), wird mit dieser Funktion nur das Vorhandensein des CSP getestet. Keine weitere Aktion ist nötig.
- b) Falls ein Key Container-Name angegeben wird, wird der zugehörige Container mittels der Funktion CDC_AcquireContext_WakeContainer gesucht und initialisiert. Diese überprüft den Container und setzt bei Problemen entsprechende Fehlermeldungen. Anschliessend wird die Java-VM aufgerufen und initialisiert (CDC_Java_StartVM). Hier wird überprüft, inwiefern schon eine Java-VM existiert oder neu gestartet werden muss und die entsprechenden Statusinformation wird gesetzt.

Anschließend hat die Applikation im Falle b die Möglichkeit, weitere Aufrufe von CSP-Funktionen auszuführen. Hierzu wird in diesem Falle das Handle auf den Key Container als Rückgabewert übergeben.

C.2 CPACquireContext mit dwFlags=CRYPT_SILENT		
In	pszContainer=	Wird nicht ausgewertet
Out	pszContainer= phContainer=	Wird nicht ausgewertet
Call	Keiner	-
Error	Keiner	-
Unterdrückung eines eventuell benutzten User Interfaces im CSP.		

Tabelle 5b: Beschreibung der Implementierung der Funktion CPACquireContext

Hier kann die Applikation eine benötigte graphische Interaktion des CSPs mit dem Benutzer unterdrücken. Dies betrifft nur die Abfrage eines Passwortes zum Zugriff auf den Private Key des Benutzers. Im CSP wird ein entsprechender Wert gesetzt. Als Folge ist kein Zugriff auf den Private Key mehr möglich.

C.3 CPACquireContext mit dwFlags=CRYPT_NEWKEYSET		
In	pszContainer=NULL oder pszContainer=<Name>	Key Container-Name
Out	pszContainer=<Name> phContainer=<Handle>	Rückgabe des Namens des neuen Key Containers und des Handles
Call	CDC_AcquireContext_NewContainer	Erzeugen eines neuen Key-Containers.
Error	NTE_BAD_KEYSET NTE_BAD_KEYSET_PARAM NTE_EXISTS NTE_KEYSET_ENTRY_BAD NTE_NO_MEMORY	Key Container kann nicht geöffnet werden PszContainer ist unbekannter Name Key Container existiert schon Key Container ist defekt Es steht kein Speicher zur Verfügung
Erzeugen eines neuen Key Containers im CSP (benutzerabhängig)		

Tabelle 5c: Beschreibung der Implementierung der Funktion CPACquireContext

Erzeugen eines neuen Key Container im CSP. Dieser muss noch kein Key-Paar enthalten, da dieses später mit CPGenKey erzeugt werden sollte. Allerdings darf dies vom CSP bereits ausgeführt und der spätere Aufruf ignorieren werden.

Es wird im CSP der Name des Key Containers in einer neuen Container-Struktur im zugehörigen Array gespeichert, um hiermit Zugriff auf eine Identität in Java zu

erhalten. Falls kein Name angegeben ist, wird der Default-Name benutzt. Hier wird unter Windows üblicherweise der Login-Name von Windows gewählt.

C.4 CPACquireContext mit dwFlags=CRYPT_MACHINEKEYSET		
In	pszContainer=NULL oder pszContainer=<Name>	Analog CRYPT_NEWKEYSET
Out	pszContainer=<Name> phContainer=<Handle>	Analog CRYPT_NEWKEYSET
Call	Keiner	-
Error	NTE_BAD_FLAGS	Wird nicht unterstützt.
Erzeugen eines neuen Key Containers im CSP (maschinenabhängig)		

Tabelle 5d: Beschreibung der Implementierung der Funktion CPACquireContext

Dieser Aufruf arbeitet analog dem vorherigen. Allerdings soll ein benutzerunabhängiges Key-Paar, das auf die Betriebssysteminstallation des Rechners bezogen ist, erstellt werden. Dies wird nicht unterstützt. Die explizite Reaktion unter Setzen des Fehlerwertes ist notwendig.

C.5 CPACquireContext mit dwFlags=CRYPT_DELETEKEYSET		
In	pszContainer=<Name>	Key Container-Name
Out	pszContainer=NULL phContainer=NULL	Keine Rückgabe
Call	CDC_AcquireContext_DelContainer	Unsere Funktion zur Erzeugung eines neuen Key-Containers.
Error	NTE_BAD_KEYSET	Key Container kann nicht geöffnet werden
	NTE_BAD_KEYSET_PARAM	PszContainer ist unbekannter Name
	NTE_KEYSET_ENTRY_BAD	Key Container ist defekt
	NTE_KEYSET_NOT_DEF	Key Container existiert nicht
	NTE_NO_MEMORY	Es steht kein Speicher zur Verfügung
Löschen eines vorhandenen Key Containers		

Tabelle 5e: Beschreibung der Implementierung der Funktion CPACquireContext

Löschen des Key Containers im CSP und des enthaltenen Key-Paares. Dies bedeutet nicht, dass Informationen auf der Java-Seite gelöscht werden sollen; es wird nur die hergestellte Beziehung zwischen Benutzername und Key-Paar in Java gelöst durch das Löschen dieser Informationen im CSP.

Anschliessend werden unabhängig von der Flageinstellung interne Werte des CSPs initialisiert.

Die notwendigen Informationen zur Implementierung aller übrigen Entry Point-Funktionen werden hier nicht angegeben. Sie können dem Sourcecode und der MSDN-Dokumentation entnommen werden [MSDN].

Zu beachten bei der Implementierung:

Die meisten anderen Funktionen müssen abgesehen von dieser typischen Implementierung nach folgendem Prinzip zur Behandlung dynamischer Datengrößen arbeiten:

Zuerst ruft die Applikation eine Funktion mit dem entsprechenden Flag, einer gewünschten, oder bei Fixgrößen einer leeren, Größenangabe und einem Null-Zeiger auf die entsprechende Datenstruktur auf. Dies dient dazu, zuerst auf Applikationsseite vom CSP die Größe der zu erwartenden Datenmenge zu erfahren, und diese entsprechend im Speicher allokkieren zu können. Als nächstes wird der identische Funktionsaufruf, diesmal allerdings mit dem korrekten Zeiger auf den neu reservierten Speicherbereich, ausgeführt. Dieser gibt dann die eigentlichen Daten und deren jetzt aktuelle Länge zurück.

Dies gilt nicht für Funktionen zur Rückgabe von Handles, da deren Länge eine statische Größenordnung hat, und zwar üblicherweise DWORD.

3.2.3 Eigene Datenstrukturen

Innerhalb des CSP werden die folgenden eigenen Datenstrukturen genutzt:

- ☞ Array von Identitäts und Key Informationen (Key Container),
- ☞ Array von Hash-Informationen und
- ☞ Puffer zur Zwischenspeicherung von Ein- und Ausgangsdaten.

Desweiteren existieren statische Informationen wie Name und Typ des CSPs und interne Flags. Um diese zu beschreiben, werden hier die wichtigsten Datenstrukturen und ihre Funktion angegeben.

Key-Paar

```
typedef struct _CDC_KEYSET {
    BYTE    bType;                // key set type
    BYTE    bVersion;            // struct version
    ALG_ID  aiKeyAlg;            // algorithm identifier
    DWORD   dwPrivateKey;        // private keyindex
    DWORD   dwPublicKey;        // public key index
} CDC_KEYSET;
```

Struktur zum Speichern von Key Informationen. Es wird nicht der Inhalt des Keys gespeichert, sondern nur Informationen zum verwendeten Algorithmus und der jeweilige Index⁸⁶.

Key Container

```
typedef struct _CDC_CONTAINER {
    BYTE    bVersion;            // struct version
    CHAR    *pcConName;          // container name
    DWORD   dwConNameLen;        // its length
    CDC_KEYSET *pKeySet;        // key set
} CDC_CONTAINER;
```

Struktur zum Speichern von Identitätsinformationen. Es werden der Name des Key Containers (und damit der Identität), dessen Länge und ein Verweis auf das Key-Paar gespeichert.

Hashinformationen

```

typedef struct _CDC_HASHINFO {
    BYTE          bVersion;           // struct version
    ALG_ID        Algid;              // alg id
    HCRYPTKEY      hKey;               // key handle
    char          sContainerName[MAX_CONNAME]; // container name
    BYTE          *pbData;            // pointer to the message
                                           // contents for hashing
    DWORD         *dwDataLen;         // len of data
    CDC_CONTAINER *pContainer;        // pointer to container
} CDC_HASHINFO;

```

Struktur zum Speichern von Hash-Informationen. Es werden die von der Applikation übergebenen Information zu Algorithmus, Key Handle, Key Container-Name als auch der Container an sich und Zeiger auf die Hash-Daten gespeichert. Dies ist nötig, da es zur gleichen Zeit mehrere dieser Datenstrukturen geben kann.

Ein- und Ausgangspuffer

```

BYTE          *pbData;           // pointer to the data
                                           // contents for i/o
DWORD         *dwDataLen;        // length of data

```

Datenpuffer zur Zwischenspeicherung von ein- oder ausgehenden Daten. Dies sind z.B. Nachrichteninhalte in Klartext oder verschlüsselt und Signaturen.

Alle Daten, die diese Datenstrukturen nutzen, werden vom CSP dynamisch angelegt⁸⁷ und wieder zerstört, sobald sie nicht mehr benötigt werden.

Da diese Daten zur gleichen Zeit mehrfach vorkommen können, werden in diesem Falle entsprechende Arrays aufgebaut. Anhand statischer globaler Variablen kann der jeweils aktuelle Datensatz ausgewählt werden.

Die hierzu nötigen Zeiger sind folgendermaßen gespeichert:

Ausgewählte Beispiele der statischen Informationen

```

static char CDC_CSPName[MAX_CSPNAME]=CSP_NAME; // CSP name
static int  CDC_CSPNameLen=0;                  // its length
static DWORD CDC_CSPVersion=CSP_VERSION; // its version number
static DWORD CDC_CSPImpType=CSP_IMPTYPE; // its implementation type
static DWORD CDC_CSPProvType=CSP_TYPE; // its type

CDC_KEYSET *CDC_KeySet;           // current key set
CDC_CONTAINER *CDC_Container;     // current container
CDC_HASHINFO CDC_HashInfo;        // current hash info

JNIEnv *CDC_JNIEnv;              // pointer to the JNI environment

```

⁸⁶ Dies ist zur Zeit ein festgelegter Wert, der wie auch die Strukturversionierung für spätere Erweiterungen genutzt werden kann.

⁸⁷ Das bedeutet, der benötigte Speicher wird allokiert bzw. freigegeben.

Die angeführten Beispiele betreffen Namen, Versionsnummer, Typ und Implementierungstyp des CSPs und die Zeiger auf die jeweils aktuellen Datenstrukturen aller Typen.

Dieser Zeiger auf die Java-Virtual Machine ist global abgelegt.

Ausgewählte Beispiele für Flags

```

BOOL CDC_CSPCryptSilent;           // Crypt Silent Flag
BOOL CDC_JVMStarted;              // does the java vm run?
BOOL CDC_JNIEnvHold;              // do we have the JNI environment

```

Flagwerte zur Überprüfung von Einstellungen des CSP. Als Beispiele dienen die User-Interface-Abschaltung (benutzt in der angeführten Funktion CPAAcquireContext) und der Status der Java-Virtual Machine (die unteren beiden Flags). Diese werden benötigt, um in Funktionen bei vorherigen CSP-Aufrufen gesetzte Informationen zu überprüfen.

Da die Möglichkeit besteht, eine Liste der unterstützten Algorithmen vom CSP abzufragen, wird hier die dazu benötigte Datenstruktur angegeben. Diese existiert außer in der angeführten noch in einer komplexeren Form. Beide sind durch die CryptoAPI-Definition festgelegt und ihre Verwendung ist dort entsprechend dokumentiert.

```

typedef struct _PROV_ENUMALGS {
    ALG_ID      aiAlgid;
    DWORD      dwBitLen;
    DWORD      dwNameLen;
    CHAR       szName[20];
} PROV_ENUMALGS;

```

Die Struktur besteht aus den drei Informationen AlgorithmID, Bitlänge des Algorithmus und Name inklusive Länge.

Dies ist beispielsweise für den SHA1-Hash-Algorithmus:

```

{
    CALG_SHA1;           // 8004: AlgorithmID for SHA1
    160;                 // bitsize: 160 bits
    5;                   // name length incl. terminal null
    „SHA1“;             // namestring
}

```

Alle weiteren Datenstrukturen können dem Sourcecode entnommen werden. Ihre Bedeutung ist dort entsprechend dokumentiert.

3.2.4 Die Kernfunktionalitäten

Die Kernfunktionalitäten des CSP umfassen die beiden bereits definierten Schichten „Core Functionality Layer“ und „Implementation Layer“. Hier sind die Funktionen definiert, die die eigentlichen CSP-internen Aktionen ausführen.

Dies betrifft die Bereiche:

- a. Implementierung der Algorithmen.
Üblicherweise werden hier die kryptographischen Algorithmen implementiert. In unserem Falle erfolgt hier keine Bearbeitung der Daten, da diese nur zwischengespeichert und an die Java-Funktionalität zur Bearbeitung weitergegeben werden.
- b. Verwaltung von Identitäten, Keys und Key Containern.
- c. Verwaltung von Nachrichtentext, verschlüsseltem Text, Hashinformationen und Signaturen.
- d. Verwaltung der CSP-Parameter wie Name, Version, Typ, Algorithmenliste.

Beispiele für solche Funktionen sind:

- a. `CDC_Sign`: Erzeugen einer Signatur. Zuvor müssen die Daten korrekt zu einer Hashinfo-Struktur zugeordnet worden sein. Hier wird nur die entsprechende Funktion der Java-Schnittstelle `CDC_Java_Sign` aufgerufen.
- b. `CDC_AcquireContext_NewContainer`: Erzeugen eines neuen Key Containers und Eintragen der gewünschten Parameter.
- c. `CDC_SetHashInfo`: Belegen einer HashInfo-Struktur mit Werten (Code-Beispiel im folgenden).
- d. `CDC_GetProvParam_GetVersion`: Erzeugen der Versionsinformation des CSP und Rückgabe an die Applikation.

Als Programmbeispiel sei hier die Funktion CDC_SetHashInfo angeführt, die nach Prüfen von AlgorithmID und Key Handle eine neue Hashinfo-Struktur mit Werten belegt und zurückgibt.

```

BOOL WINAPI CDC_SetHashInfo(
    IN ALG_ID Algid,
    IN HCRYPTKEY hKey,
    OUT CDC_HASHINFO *phHash)
{
    char message[MAX_MESSAGE];           // message text
    char ConName[MAX_CONNAME];          // container name

    sprintf( message, "CPCreateHash");
    CDC_MessageBox(NULL,message,"In",0);

    // test if we know the AlgId
    if(CDC_TestAlgId(Algid))
    {
        sprintf( message, "CPCreateHash: Bad Alg ID %d", Algid);
        CDC_MessageBox(NULL,message,"In",0);
        SetLastError(NTE_BAD_ALGID);
        return(CRYPT_FAILED);
    }

    //hKey
    // if we have a keyed hash alg, must not be NULL (ie. CALG_MAC)
    // else we do not care
    if(hKey!=0L)
    {
        sprintf( message, "CPCreateHash: got a key handle, but want an
            unkeyed hash");
        CDC_MessageBox(NULL,message,"In",0);
        SetLastError(NTE_BAD_KEY);
        return(CRYPT_FAILED);
    }

    // get the current container name set by CPAcquireContext before
    sprintf(ConName, "%s", CDC_ConName);

    // set the current hash info struct
    strcpy(CDC_HashInfo.sContainerName, ConName);
    CDC_HashInfo.Algid=Algid;
    CDC_HashInfo.hKey=hKey;

    // give back the hashinfo
    *phHash = CDC_HashInfo;

    sprintf( message, "CPCreateHash - AlgId=%d ConName='%s'", Algid,
        ConName);
    CDC_MessageBox(NULL,message,"Out",0);

    return(CRYPT_SUCCEED);
}

```

Code-Beispiel 7 : Implementierung der CSP-Funktion CDC_SetHashInfo

3.2.5 Die Schnittstelle zur JCA

Der Kontakt zwischen dem CSP und der Java-Virtual Machine wird durch die Schichten „Java Functionality Layer“ und „Java Calling Layer“ realisiert. In der ersten werden alle benötigten Java Funktionen abgebildet. Hierzu müssen jeweils der Kontakt zur Java-VM hergestellt, Funktionsparameter gesetzt und Funktionshandles erzeugt werden. Jede Funktion dieser Schicht kann von der Implementierungsschicht des CSP aufgerufen werden und ruft ihrerseits die benötigten Unterfunktionen auf. Diese sind in der darunterliegenden Aufruf-Schicht implementiert. Zudem werden hier die benötigten Datenpuffer gefüllt bzw. ausgelesen.

Der „Java Functionality Layer“ umfasst die Funktionen zum

- ☞ Signieren, Verifizieren, Ver- und Entschlüsseln der Daten,
- ☞ Starten und Initialisieren der Java-VM

Hingegen sind in der zweiten Schicht „Java Calling Layer“ alle Funktionen zusammengefasst, die

- ☞ Handles auf Java-Funktionen und –Datenpuffer erzeugen,
- ☞ Parameter von Java-Funktionen setzen und lesen sowie
- ☞ Java-Funktionen aufrufen.

Als Beispiel des Funktionalitätslayers sei die Funktion `CDC_Java_Sign` angeführt. Diese übergibt die zu hashenden und zu signierenden Daten an Java, löst die entsprechende Aktion aus und liest die erzeugte Signatur der Daten ein.

```

BOOL CDC_Java_Sign(
    BYTE *pbData,
    DWORD iDataSize,
    BYTE **pbSignature,
    DWORD *iSignSize)
{
    BOOL rc=CRYPT_FAILED;           // return code
    char message[MAX_MESSAGE];     // message text

    JNIEnv *env=NULL;             // pointer to the JNI environment

    jclass jcl;                   // current java class used
    jmethodID mid;                // current java method used
    jint res = (jint) 0;          // error code (jint = long)
    DWORD i;                       // loops

    // look for vm and env
    if( !CDC_Java_StartedAndHold) return CRYPT_FAILED;

```

```
        // get environment
        rc=CRYPT_SUCCEED;
        env=CDC_JNIEnv;

        // get method id
        if( !CDC_Java_GetMethodID(env, "CSP_Sign", "()I", &jcl, &mid))
            return CRYPT_FAILED;

        // call method
        if( !CDC_Java_CallMethod_Sign(env, jcl, mid,
            pbData, iDataSize, &pbSignature, &iSignSize))
            return CRYPT_FAILED;

        // get the data back
        // it is now in pbSignature

        sprintf(message, "exit from java-sign");
        CDC_JavaMessageBox(NULL,message,"Ok",0);

    return rc;
}
```

Code-Beispiel 8 : Implementierung der CSP-Funktion CDC_Java_Sign

Im ersten Schritt wird die Existenz und Korrektheit der Java-VM überprüft. Als nächstes wird das Funktionshandle der Java-Funktion `csp::CSP_Sign` erzeugt und die Unterfunktion zum Aufrufen dieser Java-Funktion gestartet.

Die korrespondierende Funktion der „Java Calling Schicht“ ist `CDC_Java_CallMethod_Sign`. Hier werden die Datenpuffer gefüllt und die genannte Java-Funktion aufgerufen.

Zunächst werden die zu signierenden Daten in den auf der Java-Seite bereitsstehenden Eingangs-Puffer kopiert. Danach wird die Funktion `csp::CSP_Sign` innerhalb der Java-VM aufgerufen, die dann die eigentliche Aktion des Signierens ausführt und die Signaturdaten im Ausgangspuffer⁸⁸ bereitstellt. Im Fehlerfall wird eine Fehlermeldung an die aufrufende Funktion zurückgegeben und die Funktion beendet. Bei korrektem Ablauf wird der gefüllte Ausgangspuffer aus der Java-VM ausgelesen und im CSP bereitgestellt. Anschliessend werden diese Daten an die aufrufende Funktion zurückgegeben.

⁸⁸ Sowohl Ein- als auch Ausgangspuffergröße können dynamisch verändert werden.

```

BOOL CDC_Java_CallMethod_Sign(JNIEnv *Env,
                              jclass Class,
                              jmethodID MethodID,
                              BYTE *pbData,
                              DWORD iDataSize,
                              BYTE **pbSignature,
                              DWORD *iSignSize)
{
    char message[MAX_MESSAGE];           // message text
    BOOL rc=FALSE;                       // return code
    jthrowable exception;
    jint jrc;                            // jata return code

    // set the message data in the InBuffer
    rc = CDC_Java_SetInBuffer(Env, Class, pbData, iDataSize);

    // call java signing method
    jrc = (*Env)->CallStaticIntMethod(Env, Class, MethodID);

    exception = (*Env)->ExceptionOccurred(Env);
    if (exception != NULL)
    {
        // just print an error message
        sprintf(message, "Java VM: The method CSP_Sign threw an
            exception.");
        CDC_JavaMessageBox(NULL,message,"Error",0);
        return(CRYPT_FAILED);
    }
    else
    {
        sprintf(message, "Java VM: The method CSP_Sign worked fine.");
        CDC_JavaMessageBox(NULL,message,"Ok",0);
        rc=CRYPT_SUCCEED;
    }

    // get the sign data from the OutBuffer
    rc = CDC_Java_GetOutBuffer(Env, Class, pbSignature, iSignSize);

    sprintf(message, "Java VM: exit-sign %d", *iSignSize);
    CDC_JavaMessageBox(NULL,message,"Ok",0);

    return rc;
}

```

Code-Beispiel 9 : Implementierung der CSP-Funktion CDC_Java_CallMethod_Sign

Der Gesamt Ablauf des Signierens einer Nachricht mittels dieser Schnittstelle wird im folgenden Abschnitt in der Benutzung mit Outlook demonstriert.

Weitere Implementierungsdetails

Die von einer CSP-Implementierung geforderten drei Leitsätzen werden in der Implementierung eingehalten. Applikationen erhalten keinen Zugriff auf Keys im CSP, da die genutzten Handles nie die Adresse einer Struktur im Speicher angeben, sondern einen Indexwert darstellen, über den nur der CSP auf die Datenstruktur zugreifen kann⁸⁹. Zudem sind die im Array gelisteten Informationen nicht die Private Keys, sondern es sind die Informationen zum Zugriff auf die in Java abgelegten Keys gespeichert. Analog ist das Vorgehen bezüglich des Zugriffs der Applikation auf die algorithmischen Abläufe. Auch diese finden nicht im CSP statt, sondern werden in die Java-Schnittstelle abgebildet. Zur Authentifizierung des Benutzers wurde im CSP die Möglichkeit geschaffen, mittels einer Message-Box das zum Zugriff auf den Private Key nötige Passwort einzugeben. Dieses wird nicht zwischengespeichert, sondern über die Java-Schnittstelle übergeben und auf CSP-Seite sofort zerstört.

Wie im Abschnitt 2.5 zur Abbildung des CryptoAPI auf die JCA beschrieben, muss die Frage der Lebenszeit einer Java-Virtual Machine geklärt werden. Diese ist abhängig davon, zu welchem Zeitpunkt die CSP-DLL vom Betriebssystem ge- bzw. entladen wird. Tests mit der Applikation ergeben, dass der CSP auch während der Laufzeit der Applikation beendet und später wieder neu gestartet werden kann. Deshalb ist es unbedingt erforderlich, bei jedem Start des CSP auch die Java-Virtual Machine erneut zu starten. Da der CSP-Start immer mittels der Funktion `CryptAcquireContext` (bzw. dem Pendant `CPAquireContext` auf CSP-Seite) erfolgt, wird hier die Verbindung zur Java-Welt geschaffen. Der Aufruf der abschliessenden Funktion `CryptReleaseContext` bedeutet nicht unbedingt, dass die CSP-DLL entladen wird, somit muss die Java-VM nicht zerstört werden. Das nötige Zugriffshandle wird im CSP gespeichert⁹⁰ und bei weiteren Aufrufen genutzt. Falls allerdings von Windows der CSP aus dem Speicher entfernt wird, werden hiermit automatisch alle abhängigen Prozesse und damit auch die Java-VM beendet. Dies bedeutet auch das Löschen des Zugriffshandles im CSP⁹¹.

⁸⁹ Die Applikation kennt die Basisadresse des Arrays nicht.

⁹⁰ Und ein entsprechendes Flag gesetzt, das bei nachfolgenden Aufrufen von `CPAquireContext` überprüft wird.

⁹¹ Das Handle wird dann beim nächsten Aufruf von `CPAquireContext` neu erzeugt.

Weiterhin wurde zu Testzwecken die Ausgabe von Nachrichten des CSPs implementiert. Wenn es zur Übersetzungszeit gewünscht ist, kann über das Setzen eines Wertes im Sourcecode bestimmt werden, dass zur Laufzeit bei Aufruf jeder Einzelfunktion des CSPs eine sogenannten Message-Box auf dem Bildschirm ausgegeben wird. Dies ist besonders nützlich, um die Aufrufsemantik von Applikationen festzustellen⁹².

Als Besonderheit zum Übersetzen des Sourcecodes wird angemerkt, dass eine Erweiterung der benötigten Bibliotheken des C-Compilers um die `crypt32.dll`⁹³ erfolgen muss, und das Microsoft Platform SDK mit den aktuellsten Bibliotheks- und Include-Informationen der `wincrypt.h` benötigt wird. Analog müssen die Bibliotheks- und Include-Dateien des JNI⁹⁴ dem Projekt hinzugefügt werden.

Die aktuelle Implementierung ist nicht auf ihr Verhalten unter Multitasking- bzw. Multithreadinganforderung hin getestet. Dies betrifft hauptsächlich die Verwendung von internen Indizes und Zeigern zu Datenarrays, globalen Variablen und dem Allokieren und Freigeben von Hauptspeicher. Ein interessanter Versuch wäre es zudem, genau eine Java-Virtual Machine für alle Prozesse bzw. Threads zu verwenden, was aber auch entsprechende Datenstrukturen an der Schnittstelle zwischen C und Java verlangt⁹⁵.

Dieser Abschnitt beantwortet also die in Fragestellung 1 und 2 aufgeworfenen Fragen, wie Applikationen das CryptoAPI nutzen und wie der CSP vom CryptoAPI aufgerufen wird. Außerdem ist der Aufruf der JCA durch den CSP beschrieben worden, was Fragestellung 6 entspricht. Die weitere Benutzung der JCA durch den CSP beschreibt Kapitel 3.6.

⁹² Vgl. Kapitel 3.6.

⁹³ Bei VisualStudio 6 in Projekt/Einstellungen/Linker/Bibliotheken

⁹⁴ `jvm.lib`, `jni.h` und `jni_md.h`.

⁹⁵ Die Erweiterung um Multitasking bzw. -threading ist nicht Ziel dieser Arbeit.

3.3 Registrieren von Object Identifiern

Um mit den eingeführten Object Identifiern, also den Standard-OIDs und den Microsoft-spezifischen AlgorithmIDs, arbeiten zu können, müssen diese allen Teilen des gesamten Systemaufbaus zur Verfügung stehen. Als erster Schritt ist hierbei die Implementierung des Registrierungsprozesses zu betrachten. Des Weiteren ist interessant, wie diese Informationen in den einzelnen Systemteilen genutzt werden.

3.3.1 Implementierung des Registrierungsprozesses

Der komplette Ablauf des implementierten Registrierungstools regoid.exe ist wie folgt:

1. Bestimmen des Namens der Konfigurationsdatei (Aufrufparameter).
2. Lesen und Parsen⁹⁶ der Informationen der Konfigurationsdatei.
3. Bildschirm-Ausgabe der gültigen Einträge.
4. Registrieren der gültigen Einträge.
5. Test des Eintrages auf Existenz mit Ausgabe der Werte.

Zuvor muss die benötigte Konfigurationsdatei erstellt worden sein. Dies geschieht zur Zeit noch manuell. Die Automatisierung des Prozesses ist unter Benutzung des JNI und Abfragen in der JCA registrierter Provider leicht möglich. Allerdings wird bewusst eine Konfigurationsdatei benutzt, um zur Laufzeit des CSPs unnötigen Overhead durch wiederholtes Zusammenstellen dieser statischen Informationen zu vermeiden.

Informationen zum Aufbau dieser Datei finden sich in Anhang A.3.

⁹⁶ Parsen bedeutet Umsetzen der textuellen Beschreibung in interne Datenstrukturen.

Das folgende Code-Beispiel gibt denjenigen Teil des Programmes wieder, der die eigentliche Zusammenstellung der Datenstruktur `CRYPT_OID_INFO` vornimmt und mittels der Funktion `CryptRegisterOIDInfo` des CryptoAPIs die Informationen in der Registry einträgt.

```
// these were set by data from config file before
LPCSTR pszOID;
ALG_ID aiAlgID;
DWORD dwGroupID;
char *szName;

// new data struct
CRYPT_OID_INFO CryptOIDInfo;
LPWSTR pwszName;

/*****
/* build the CRYPT_OID_INFO struct */
/*****
// convert the name into a wide char
pwszName=(LPWSTR) calloc(MAX_NAME,1);
Namelen=0;
if( (Namelen=MultiByteToWideChar(0,0, szName, -1, pwszName, MAX_NAME))
    ==0)
    HandleError("no multi byte to wide char.\n");

// clear memory
memset(&CryptOIDInfo, 0, sizeof(CRYPT_OID_INFO));

// set the struct
CryptOIDInfo.Algid=aiAlgID;
CryptOIDInfo.pszOID=pszOID;
CryptOIDInfo.pwszName=pwszName;
CryptOIDInfo.dwGroupId=dwGroupID;

CryptOIDInfo.cbSize=sizeof(CRYPT_OID_INFO);

/*****
/* do the registration */
/*****
c=my_wait2("Please press 'y' for registration..");
fflush(stdin);

if(c=='y' || c=='Y')
{
    printf("\n");
    // register it in the system
    rc=CryptRegisterOIDInfo(&CryptOIDInfo,
        CRYPT_INSTALL_OID_INFO_BEFORE_FLAG);

    if(rc)
        printf("Successfully registered OID info.\n");
    else
    {
        printf("Cannot register OID info.\n");
        return(FALSE);
    }
}
}
```

Code-Beispiel 10: Teile der Implementierung des Programms `RegOID.exe`

3.3.2 Nutzung von Object Identifiern im System

Wie in Abbildung 19 ersichtlich, befinden sich die mit der Registrierung eingetragenen Informationen in der Registry von Windows. Aus dieser sind sie sowohl für Applikationen, das CryptoAPI und den CSP zugänglich.

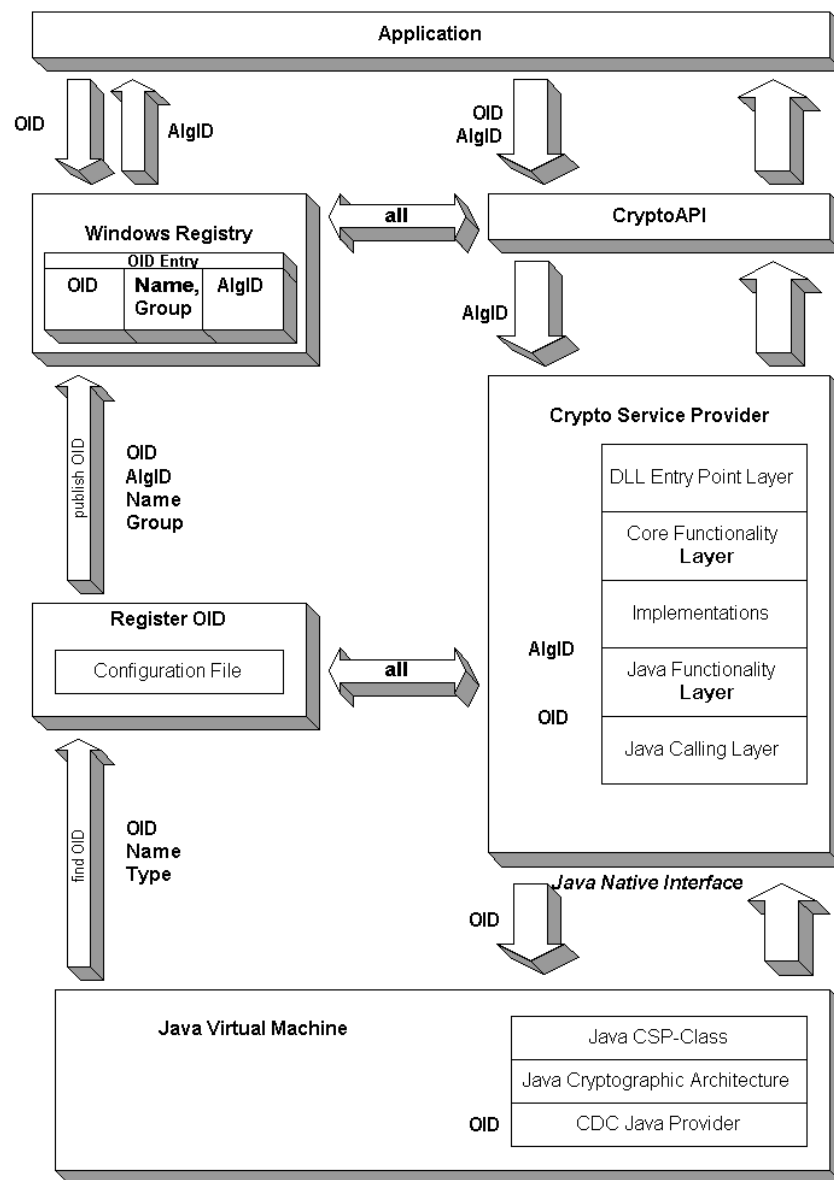


Abbildung 19: Registrierung von Object Identifiern für den CDC-Crypto Service Provider

Die Umwandlung von OID in die AlgorithmID erfolgt entweder von Applikationsseite, falls diese die benötigten Datenstrukturen für die einfacheren CSP-Funktionen selbst zusammensetzt oder im CryptoAPI für die komplexeren Simplified-Funktionen. Vom CryptoAPI an den CSP wird nur die für ihn eindeutige⁹⁷ AlgorithmID weitergegeben.

Zur Abbildung auf die JCA muss diese im CSP wieder anhand der Informationen der Konfigurationsdatei zurückgewandelt werden. Die OID wird an die Java-Funktionalität übergeben und der zu benutzende CDC-Provider und der zugehörige Algorithmus bestimmt.

Wie in Kapitel 2 beschrieben, wird in Java der Algorithmus anhand eines Namensstrings ausgewählt. Dabei werden von beiden zur Zeit genutzten CSPs die unterstützten Algorithmen abgefragt und die Namensstrings bestimmt. Die angeführten Java-Funktionen zur Umwandlung der Namen in entsprechende OID-Werte in dot-notation werden genutzt und anschließend die erzeugten Werte mit den Eingabewerten vom CSP verglichen. Bei Übereinstimmung sind zu nutzender Provider und Algorithmus gefunden, andernfalls wird bis zum letzten unterstützten Algorithmus weitergesucht. Falls kein entsprechender Algorithmus in den Providern gefunden werden kann, wird die zugehörige Fehlermeldung an den CSP zurückgegeben.

Um das System um weitere Provider auf Java-Seite zu erweitern, bestehen grundsätzlich zwei Möglichkeiten. Es kann der Name des Providers in der Java-CSP-Klasse statisch zu den anderen hinzugefügt werden. Vorteilhafter ist es, mit der von der JCA angebotenen Funktion die zur Verfügung stehenden Provider abzufragen⁹⁸. Da diese zuvor allerdings anhand der festgelegten Namen zur JCA-Schnittstelle während der Laufzeit der Java-VM hinzugefügt werden, müssen auch an dieser Stelle die Namen bekannt sein. Daher müssen diese statisch festgehalten werden. Folglich ist es nötig, zur Erweiterung des Systems um weitere Provider und deren Algorithmen die Java-Klassen-Datei auszutauschen.

⁹⁷ Diese ist nicht über CSP-Grenzen hinweg eindeutig.

⁹⁸ Weitere Informationen zur JCA finden sich in [OAKS].

3.3.3 Neu vergebene Object Identifier

Abbildung 20 gibt einen kurzen Überblick über noch nicht von anderen Institutionen standardisierte, sondern durch das Fachgebiet neu vergebene Object Identifier:

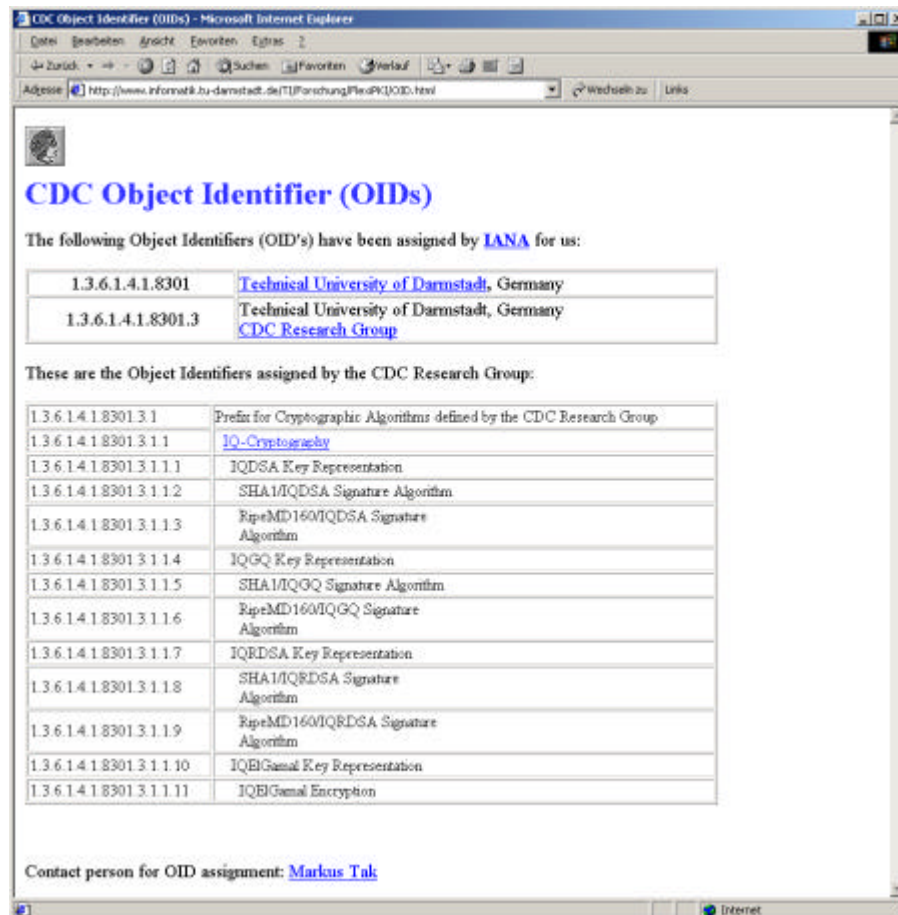


Abbildung 20: OID-Definitionen des CDC

Diese Informationen können im Internet unter [CDC-OID] eingesehen werden. An dieser Stelle sind die jeweils aktuell vergebenen Werte eingetragen, die auch anderen Institutionen gegenüber publiziert werden.

Eine vollständige Liste der verwendeten Object Identifier-Werte ist der Konfigurationsdatei zu entnehmen.

Dieser Abschnitt beantwortet die in Fragestellung 3 und 7 aufgeworfenen Fragen, wie Object Identifier in CryptoAPI und JCA und damit im Gesamtsystem eingesetzt werden.

3.4 Publizieren von Zertifikaten

Zertifikate müssen im System publiziert werden, um der Applikation zur Benutzung zur Verfügung zu stehen. Hierbei ist es notwendig, neben dem Eintragen des Zertifikates in einen Store die Verknüpfung zu CSP und Key Container herzustellen. Näher betrachtet werden die Implementierung des Publizierungsprozesses und die Integration in das Gesamtsystem.

3.4.1 Implementierung des Publizierungsprozesses

Das Zertifikat liegt als Zertifikatsdatei vor, die von einer Zertifizierungsstelle erzeugt wurde. Diese Datei ist standardkonform nach X.509 aufgebaut und kodiert. Um das Zertifikat als zugehörig zum CSP zu kennzeichnen, muss die Identität des Besitzers aus dem Zertifikat extrahiert werden und der zugehörige Key Container im CSP zur Verfügung stehen. Hierbei wird mit dem Identitätsnamen im CSP die Java-VM aufgerufen und dort die zugehörige Java-Keystore-Datei mit dem entsprechenden Zertifikat gesucht⁹⁹. Falls diese Datei korrekt angesprochen werden kann, wird die Verknüpfung durch Eintragen des CSP-Namens und des Key Containernamens im Zertifikat hergestellt.

Das Vorgehen zum Publizieren eines Zertifikates ist also:

- ☞ Auswerten der Eingabeparameter hinsichtlich des Zertifikats-Dateinamens,
- ☞ Testen auf Existenz der Datei,
- ☞ Laden und Dekodieren der Datei (im X.509-Format, in base64- oder DER-Kodierung),
- ☞ Feststellen der Besitzeridentität aus dem geladenen Zertifikat,
- ☞ Erzeugen eines Handles auf den CSP,
- ☞ Eintragen der CSP und Identitätsinformationen in die Key-Daten der Zertifikates und
- ☞ Speichern des Zertifikates im MY-Store von Windows.

⁹⁹ Die Funktion `Crypt/CPAcquireContext` mit dem Key Containernamen initialisiert die Java-VM und startet die entsprechende Initialisierungsfunktion der Java-Klasse. Diese sucht die zugehörige Java-Keystore-Datei.

```

// a. set a handle to the CDC-CSP
if(CertSetCertificateContextProperty(pCert, CERT_KEY_PROV_HANDLE_PROP_ID,
    0, (void *) hCSP))
    printf("Set the CSP handle within the cert.\n");
else
    HandleError("Setting the CSP handle within the cert failed.\n");

// create a new private key struct for the cert and :
// b. set the csp name to the name of the CDC-CSP
// c. set the key name to the name of the private key container
//    within the CDC-CSP
if(pKey = (CRYPT_KEY_PROV_INFO *) malloc(sizeof(CRYPT_KEY_PROV_INFO)))
    printf("Memory allocated for private key & CSP info struct.\n");
else
    HandleError("Cannot allocate memory for new private key & CSP info
        struct.\n");

// convert csp name to wide char
if(CSPName = (LPWSTR) malloc(MAX_CSPNAME))
    printf("Memory allocated for CSP name.\n");
else
    HandleError("Cannot allocate memory for CSP name.\n");

if(!MultiByteToWideChar(0,0, CSP_NAME, -1, CSPName, MAX_CSPNAME))
    HandleError("Cannot convert multi byte to wide char.\n");

// ConName=...; this was set before from the certificate
// convert container name to wide char
if(ConName = (LPWSTR) malloc(MAX_CSPNAME))
    printf("Memory allocated for container name.\n");
else
    HandleError("Cannot allocate memory for container name.\n");
if(!MultiByteToWideChar(0,0, CON_NAME, -1, ConName, MAX_CSPNAME))
    HandleError("Cannot convert multi byte to wide char.\n");

pKey->pwszContainerName=ConName;           // container name as wide string
pKey->pwszProvName=CSPName;               // CSP name as wide string
pKey->dwProvType=(DWORD) CSP_TYPE;        // CSP type
pKey->dwFlags=CRYPT_VERIFYCONTEXT;        // key action
pKey->cProvParam=(DWORD) 0L;              // no params
pKey->rgProvParam=NULL;                   // no params
// pKey->dwKeySpec=AT_SIGNATURE;           // does often not work with outlook
pKey->dwKeySpec=AT_KEYEXCHANGE;          // so use this even for
                                           // only signatures

// save the pointer to the struct back in the cert
if(CertSetCertificateContextProperty(pCert, CERT_KEY_PROV_INFO_PROP_ID,
    0, (void *) pKey))
    printf("Set the private key & CSP info struct within the cert.\n");
else
    HandleError("Setting the private key & CSP info struct within
        the cert failed.\n");

// save the certificate into the windows MY-store
if(CertAddCertificateContextToStore(
    hStore, pCert,
    CERT_STORE_ADD_REPLACE_EXISTING, // add always and replace
                                     // existing
    NULL))
    printf("Certificate added to the system store. \n");
else
    HandleError("Could not add the certificate to the system store.\n");

```

```
// you'll have to destroy the context before closing the store
// because that just decrements the count on the handles
if(pCert) CertFreeCertificateContext(pCert);

// close store
if(hStore)
{
    if (CertCloseStore( hStore, CERT_CLOSE_STORE_CHECK_FLAG))
        printf("Successfully closed the store.\n");
    else HandleError("Cannot close store.\n");
}
```

Code-Beispiel 11: Teile der Implementierung des Programms PubCert.exe

In Code-Beispiel 11 ist die Existenz des Key Containers bereits geprüft und das Handle auf den CSP vorhanden. Es wird eine Key Struktur erzeugt und die entsprechenden gefundenen Informationen eingetragen¹⁰⁰. Anschliessend wird diese in der Zertifikats-Struktur im Speicher eingetragen und das gesamte Zertifikat wird zu einem vorher geöffneten System-Store hinzugefügt. Danach wird der Store geschlossen und das Zertifikat steht dem Anwender zur Verfügung.

¹⁰⁰ Die Namen müssen zuvor von einfachen Strings in Widechar-Strings umgewandelt werden. Hier belegt jedes Zeichen statt einem Byte zwei (Unicode).

3.4.2 Nutzung von Zertifikaten im System

Die Einbindung der Zertifikate im gesamten System zeigt Abbildung 21. Das aus der Java-Keystore Datei geladenen Zertifikat wird mittels des beschriebenen Publizierungsprogramms in den Windows Key Store¹⁰¹ übertragen.

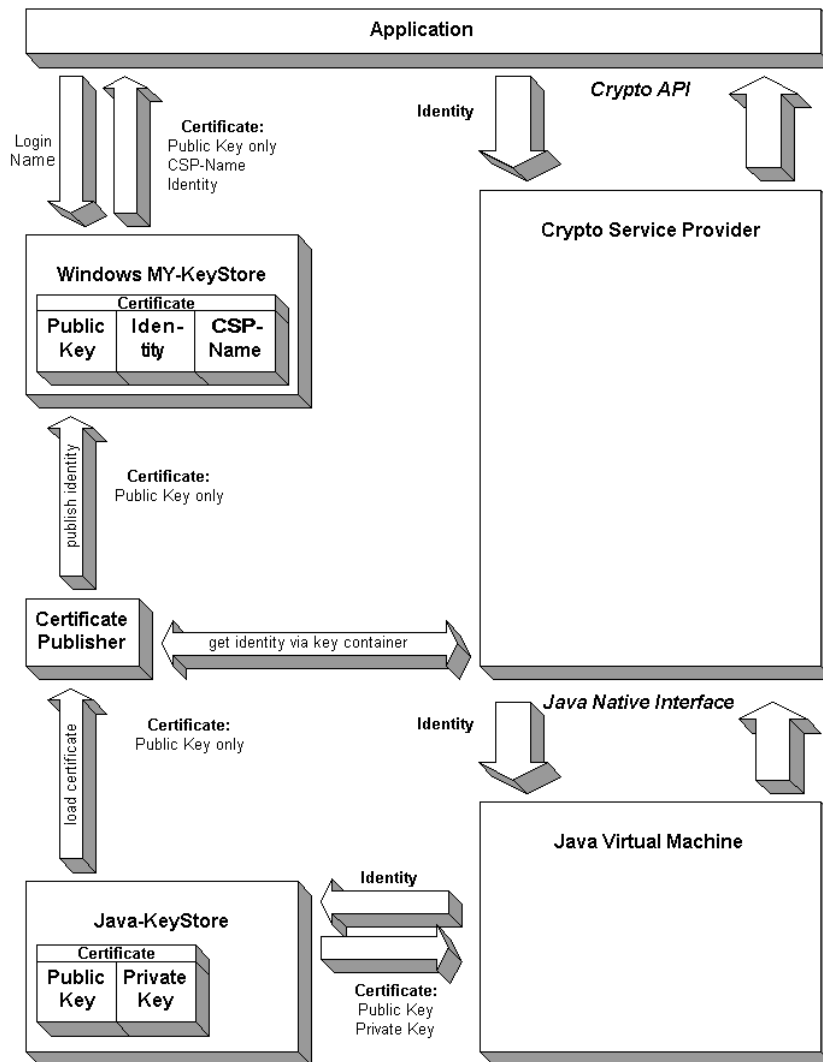


Abbildung 21: Publizieren von Zertifikaten für den CDC-Crypto Service Provider

¹⁰¹ Es wird üblicherweise im MY-Key-Store der eigenen Zertifikate abgelegt.

Die Applikation hat somit die Möglichkeit, das Zertifikat zu nutzen. Anhand der Eintragungen in der Zertifikatsstruktur wird der zugehörige CSP aufgerufen. Dieser gibt im folgenden die Identitätsinformation des Zertifikates an die Java-Funktionalität weiter. Hier kann diese Information letztendlich zur Ausführung der gewünschten Aktionen genutzt werden. Anhand des zugeordneten Namens wird die zugehörige Java-Key-Store Datei ausgewählt und Public als auch Private Key stehen zur Verfügung.

Wie in Kapitel 2 beschrieben, wird auf der Java-Seite eine Key-Store-Datei geladen. Die Implementierung legt fest, dass der Dateiname folgender Konvention entsprechen muss:

```
{Namensstring der Benutzeridentität}{„.ks“}
```

Falls der Namensstring nicht exakt dem im Windows Key Store eingetragenen Zertifikatsnamen entspricht, kann die zugehörige Datei nicht identifiziert werden und es wird mit einer Fehlermeldung abgebrochen. Es ist also darauf zu achten, dass die genutzten Identitätsnamen überall im System als Strings identisch sind.

Weiterhin ist zum Zugriff auf den Private Key des Zertifikates ein Passwort erforderlich. Dieses kann nur vom Benutzer eingegeben werden, da ansonsten die Sicherheit des Systems nicht gewährleistet werden kann. Es wird mit der CSP-Funktion `CDC_GetUserPassword` vom Benutzer abgefragt¹⁰² und an die entsprechende Java-Funktion weitergegeben. Anschliessend können mittels der erhaltenen Keys die entsprechenden Funktionen ausgeführt werden.

Dieser Abschnitt beantwortet die in Fragestellung 4 und 7 aufgeworfenen Fragen, wie Zertifikate in CryptoAPI und JCA und damit im Gesamtsystem eingesetzt werden .

¹⁰² Dies bedingt, dass dem CSP eine solche Abfrage erlaubt ist. Durch Setzen des bereits erwähnten Flags „CRYPT_SILENT“ wird diese Abfrage unterbunden. Dann ist kein Zugriff auf den Private Key möglich und es wird eine entsprechende Fehlermeldung generiert. In der Folge muss die Aktion abgebrochen werden.

3.5 Installation der Komponenten

Erst nach Installation aller bereits beschriebenen Komponenten in Windows ist es möglich, das Zusammenspiel mit existierenden Applikationen zu testen. Hierzu muss eine Reihe von Einzelschritten ausgeführt werden.

Voraussetzung ist die Übersetzung der drei Hauptprogramme CSP, OID-Registrierung und Zertifikats-Publizierung in die ausführbaren Dateien `csp.dll`, `regoid.exe` und `pubcert.exe`.

Das Vorgehen bei der Installation ist wie folgt:

- ~~☞~~ Verändern der Windowsinstallation bedingt durch den Einsatz des CSPDK,
- ~~☞~~ Installieren der Java-Runtime-Umgebung,
- ~~☞~~ Installation und Registrierung der CSP-DLL,
- ~~☞~~ Registrierung der OIDs und
- ~~☞~~ Publizieren eines Zertifikates.

Es werden weitere Dateien benötigt. Dies sind u.a. das CSP-Registrierungsprogramm, die OID-Konfigurationsdatei, die nötigen Zertifikatsdateien, das Java-Paket und die Windows-Dateien (Austausch-DLLs und Signierungswerkzeuge) des CSPDK.

Durch korrekte Installation ist es möglich, den neu erstellten CSP so in Windows zu integrieren, dass er von Applikationen wie z.B. Microsoft Outlook genutzt werden kann.

Die Vorgehensweise zur Installation ist detailliert in Anhang A.2 „Cookbook für die CDC-CSP-Installation“ beschrieben. Dies beinhaltet die Auflistung aller benötigten Dateien und eine Schritt-für-Schritt Anleitung bis hin zur Überprüfung der Installation.

Die erfolgreiche Installation wird für den folgenden Abschnitt vorausgesetzt.

3.6 Microsoft Outlook als Testapplikation

In diesem Abschnitt wird das Benutzen des in Windows installierten CSPs durch Microsoft Outlook anhand der typischen Workflows (als Beispiel das Signieren einer Email) demonstriert. Es werden die Aufrufe des CSPs durch die Applikation¹⁰³ untersucht und somit die notwendige Abbildung auf die Java-Funktionalitäten bestimmt.

3.6.1 Workflow 'Signieren einer Email'

Als Beispiel wird der Arbeitsablauf „Signieren einer Email“ in Outlook 2000 betrachtet. Hierzu wird ein Benutzerkonto vorausgesetzt, das anhand der bereits beschriebenen Daten entsprechend den Informationen im Zertifikat eingestellt ist¹⁰⁴. Desweiteren müssen die Optionseinstellungen so gewählt sein, dass das publizierte Zertifikat der Benutzeridentität zugeordnet und ausgewählt ist.

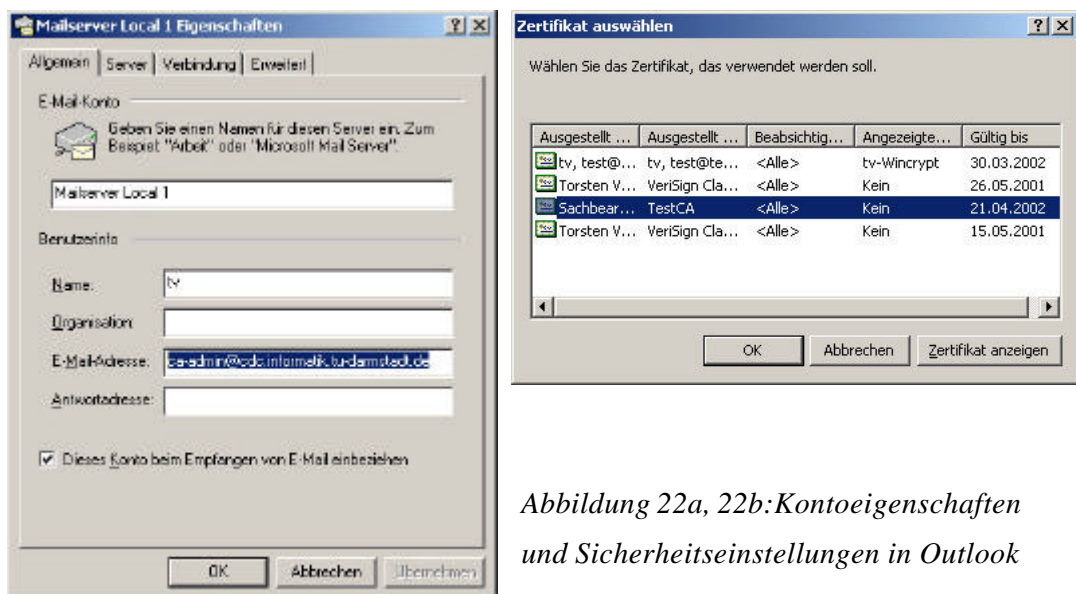


Abbildung 22a, 22b: Kontoeigenschaften und Sicherheitseinstellungen in Outlook

Zudem muß die Option „Nachrichten digitale Signatur hinzufügen“ ausgewählt sein (in Extras/Sicherheitsoptionen).

¹⁰³ Via des CryptoAPI

¹⁰⁴ Dies betrifft die Benutzeridentität (Name im Zertifikat sowohl im CryptoAPI als auch in der JCA; nicht der Benutzername im Konto) und die Email-Adresse, die übereinstimmen müssen.

Bei Aufruf des Arbeitsvorgangs „Neue Email erstellen“ kann der Inhalt der Nachricht, ihr Titel und der Adressat angegeben werden. Als nächster Schritt wird mittels des „Senden“-Buttons des Nachrichtenfensters der Signiervorgang initiiert, bevor die signierte Email von Outlook versendet wird. Dieser Vorgang wird in den folgenden Abschnitten genauer untersucht.

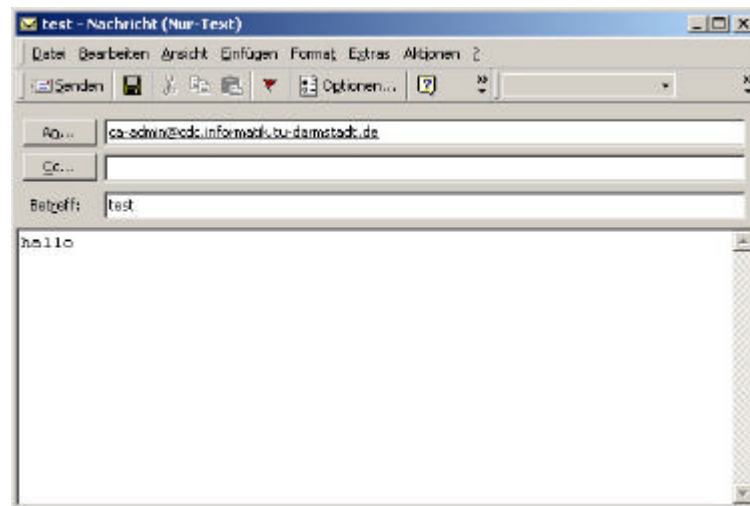


Abbildung 23: Erstellen einer signierten Nachricht in Outlook

Nach erfolgreichem Abschluß ist die Nachricht ohne weitere Interaktion des Benutzers¹⁰⁵ signiert worden und wird versendet.

¹⁰⁵ Es erfolgt keinerlei Bildschirmausgabe der Signatur der Nachricht. Outlook benutzt nur ein Symbol im Posteingang (auf Empfängerseite), das anzeigt, dass eine Nachricht signiert wurde.

Der Adressat erhält die Nachricht. Diese ist als signiert markiert worden.

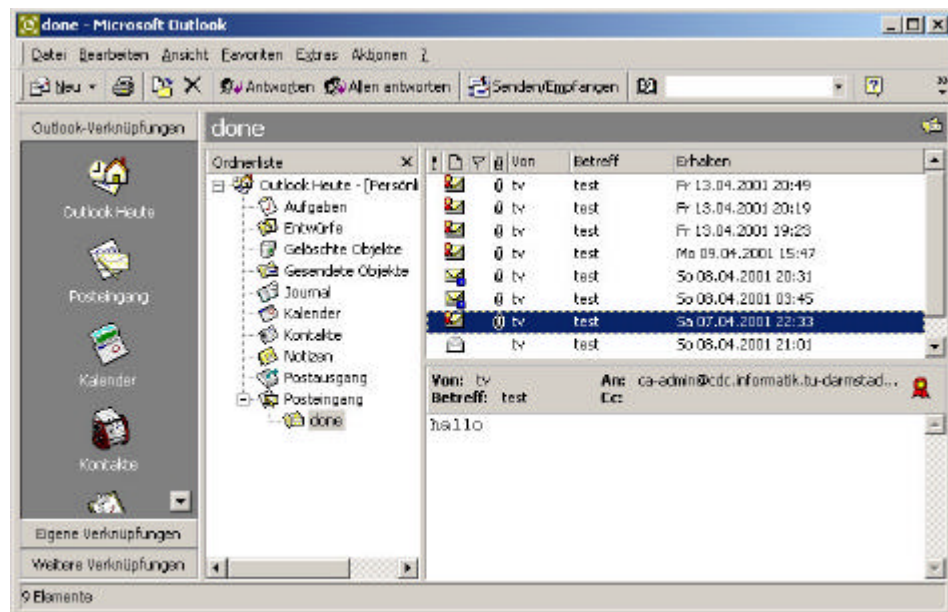


Abbildung 24: Posteingang mit signierter Nachricht in Outlook

Durch Klicken auf das zum Markieren signierter Nachrichten genutzte Symbol werden die Eigenschaften der Signatur angezeigt.

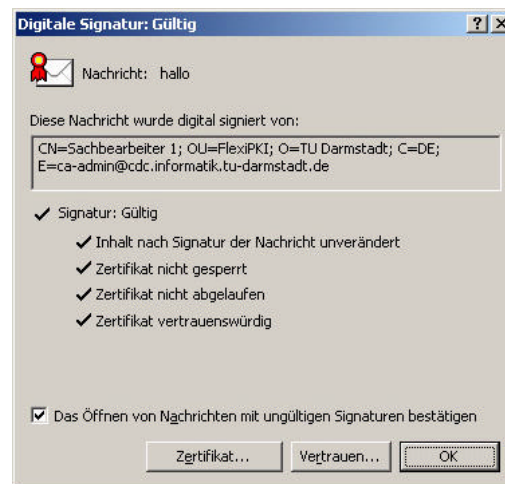


Abbildung 25: Informationen zur Signatur der Nachricht in Outlook

Nach erfolgreichem Verifizieren wird die Signatur als gültig gekennzeichnet.

3.6.2 Aufrufe von CSP-Funktionen durch Outlook

Im folgenden werden die von Outlook an den CSP gestellten Funktionsaufrufe zum Signieren einer Nachricht analysiert. Diese sind durch Testen anhand der Ausgabe von Message-Boxen¹⁰⁶ ermittelt worden.

Die festgestellte Abfolge der Aufrufe ist:

<i>Nr.</i>	<i>Funktion</i>	<i>Bedeutung</i>
1.	CPAcquireContext	Erzeugen des Kontexthandles auf CSP und Key Container.
2.	CPCreateHash	Erzeugen und Initialisieren der Hashinfo-Struktur im CSP.
3.	CPHashData	Einfügen der Daten in die Hashinfo-Struktur im CSP.
4.	CPGetHashParam	Lesen der Größenangabe des zu erwartenden Hashes.
5.	CPGetHashParam	Lesen des Hashes.
6.	CPSignHash	Signieren der gehashten Nachrichtendaten.
7.	CPDestroyHash	Zerstören der Hashinfo-Struktur im CSP.
8.	CPReleaseContext	Lösen des Handles auf Key Container und CSP.

Tabelle 6: Die CSP-Funktionsaufrufe beim Outlook-Workflow ‚Signieren‘

Um die Beantwortung dieser Aufrufe korrekt implementieren zu können, ist es notwendig, alle relevanten Parameter zu berücksichtigen und die von der Applikation beabsichtigte Aktion zu bestimmen.

¹⁰⁶ Fenster mit Hinweisen, die den Funktionsnamen und die Parameter ausgeben.

Die Aufrufe sind im einzelnen:

1. Erzeugen des Kontexthandles auf CSP und Key Container:

C.1 CPAcquireContext mit CRYPT_VERIFYCONTEXT		
In	pszContainer	<Identity Name>
Out	pszContainer phContainer	<Identity Name> <Identity Handle>
Call	(CSP) CDC_Java_StartVM	Starten einer Java-Virtual Machine, falls diese nicht bereits gestartet ist. Es wird ein Zeiger auf das zugehörige Environment zurückgegeben und gespeichert.
Error	Diverse	Alle Fehler zu Identität und Key Container

Tabelle 7a: Aufrufparameter der Funktion CPAcquireContext

Die Identity ist der Name des Besitzers des Private Keys, mit dem er im JCA-Provider referenziert wird. Dieser muss beim Publizieren des Zertifikates in der Zertifikatsstruktur korrekt eingetragen worden sein. Hiermit wird dann später der Private Key zum Signieren angesprochen. Im CSP wird dieser Name zum Referenzieren des gesuchten Key Containers genutzt. Zusätzlich wird überprüft, ob die Java-VM bereits existiert. Ist dies nicht der Fall, wird diese gestartet.

2. Erzeugen und Initialisieren der Hashinfo-Struktur im CSP:

C.2 CPCreateHash		
In	hContainer AlgID hKey	<Identity Handle> <Algorithm ID> <null> (<i>nicht benötigt</i>)
Out	phHash	<Hashinfo Handle>
Call	Keiner	-
Error	Diverse	Alle Fehler zu Identität und Algorithmus ID.

Tabelle 7b: Aufrufparameter der Funktion CPCreateHash

Dem CSP wird die Möglichkeit zum Erstellen seiner internen Struktur für einen Hash gegeben. Die übergebenen Daten sind die AlgorithmID zum Hashen und das Handle auf den Key Container. Zurückgegeben wird vom CSP ein Handle auf die ausgefüllte HashInfo-Struktur. Es erfolgt kein Aufruf zur Java-Schnittstelle.

3. Einfügen der Daten in die Hashinfo-Struktur im CSP:

C.3 CPHashData		
In	hContainer hHash pbData dwDataLen	<Identity Handle> <Hashinfo Handle> <Zeiger auf die Daten> <Länge der Daten>
Out	-	-
Call	(CSP) CDC_Java_Init (Java) csp::CSP_Init (Java) csp::CSP_AddProv	Initialisieren der Identitätsinformationen auf Java-Seite. Hierzu werden Provider und Keystore geladen.
Error	Kein direkter.	-

Tabelle 7c: Aufrufparameter der Funktion CPHashData

Da dieser Aufruf nicht direkt zur Java-Schnittstelle weitergegeben wird, werden die Daten zwischengespeichert. Der spätere Rückgabewert ist also *kein* korrekter Hash der Daten sondern ein Ersatzwert. Intern werden später zum Signieren die zwischengespeicherten Daten weiterverwendet. Der Aufruf der Java-Funktionalität sorgt für die korrekte Initialisierung der CSP-Klasse in Java und der enthaltenen Datenobjekte. Anschließend wird der benötigte Java-Provider hinzugefügt.

4. Lesen der Größenangabe des zu erwartenden Hashes:

C.4 CPGetHashParam		
In	pbData pdwDataLen	<null> <0 als erwartete Länge>
Out	pdwDataLen	< Tatsächliche Länge>
Call	Keiner	-
Error	Kein direkter	-

Tabelle 7d: Aufrufparameter der Funktion CPGetHashParam

Die von der Applikation zu erwartende Länge des Hashes wird in Abhängigkeit vom gewählten Algorithmus bestimmt. Da diese jeweils fest definiert ist, kann der entsprechende Wert zurückgegeben werden¹⁰⁷. Es erfolgt kein Aufruf zur Java-Schnittstelle.

¹⁰⁷ Aus Flexibilitätsgründen wird eine Größe zurückgegeben, die für jeden betrachteten Hashalgorithmus ausreichend ist. Dies ist möglich, da kein korrekter Hash berechnet wird und vermeidet einen Aufruf des Java-Providers, um den Wert zu erfragen.

5. Lesen des Hashes:

C.5 CPGetHashParam		
In	pbData pdwDataLen	<Zeiger auf allokierte Speicherbereich der benötigten Länge> < Tatsächliche Länge>
Out	pbData	<Ersatz Hash>
Call	Keiner	-
Error	Diverse	Fehler zur Korrektheit des Zeigers

Tabelle 7e: Aufrufparameter der Funktion CPGetHashParam

Anhand der nun bekannten Größe des zu erwartenden Hashes hat die Applikation den benötigten Speicherplatz reserviert und übergibt einen Zeiger auf diesen. Sie erwartet nun vom CSP die Rückgabe des Hashes der Daten. In unserem Falle wird statt dessen vom CSP nur ein Ersatz-Hash zurückgegeben, da der korrekte zu diesem Zeitpunkt noch nicht berechnet wurde. Da die Daten intern zwischengespeichert wurden, können beim späteren Signierungs-Aufruf von der Java-Seite sowohl Hash als auch die Signatur erzeugt werden. Es erfolgt kein Aufruf zur Java-Schnittstelle.

6. Signieren der gehashten Nachrichtendaten:

C.6 CPSignHash		
In	hContainer hHash pbData dwDataLen	<Identity Handle> <Hashinfo Handle> <Zeiger auf den Signaturspeicher> <Erwartete Länge der Signatur>
Out	pbData dwDataLen	<Signatur> <Tatsächliche Länge der Signatur>
Call	(CSP) CDC_Java_Sign (Java) csp::CSP_GetCert (Java) csp::CSP_Sign	Hashen und Signieren der Nachricht in einem Schritt. Zunächst wird der Keystore und daraus das Zertifikat geladen, danach gehasht und signiert.
Error	Keiner	-

Tabelle 7f: Aufrufparameter der Funktion CPSignHash

Unter Angabe der bisher erzeugten Informationen zum Key Container und der Hashinfo-Struktur wird das Signieren der im Hash befindlichen Daten ausgelöst. Der benötigte Zeiger auf den von der Applikation zur Verfügung gestellten Speicherbereich und dessen erwartete Länge werden zusätzlich angegeben. Beim Aufrufen der Funktion CDC_Java_Sign werden die daran angeschlossenen Java-

Funktionen zum Laden des Zertifikates und zum Erzeugen der Signatur ausgeführt. Da zum Laden des Zertifikates ein Passwort nötig ist, wird dieses durch den CSP vom Benutzer abgefragt und an die Java-Schnittstelle übergeben. Rückgabewerte an die Applikation sind die Signatur und ihre tatsächliche Länge.

7. Zerstören der Hashinfo-Struktur im CSP:

C.7 CPDestroyHash			
In	hContainer	<Identity Handle>	
	hHash	<Hashinfo Handle>	
Out	-	-	
Call	Keiner	-	
Error	Keiner	-	

Tabelle 7g: Aufrufparameter der Funktion CPDestroyHash

Die Applikation informiert den CSP, die nicht mehr benötigte Hashinfo-Struktur freizugeben. Diese wird aus dem Speicher entfernt. Es erfolgt kein Aufruf zur Java-Schnittstelle.

8. Lösen des Handles auf Key Container und CSP:

C.8 CPReleaseContext			
In	hContainer	<Identity Handle>	
	Out	-	
Call	(CSP) CDC_Java_Exit (Java) csp::CSP_Exit	Lösen etwaiger Handles und Freigeben von Speicher. Die Java-VM wird aus Performancegründen nicht beendet. Dies erfolgt erst mit dem Entladen der CSP-DLL aus Windows (Spätestens mit Beendigung von Outlook).	
Error	Keiner	-	

Tabelle 7h: Aufrufparameter der Funktion CPReleaseContext

Zum Beenden des gesamten Workflows werden im CSP alle genutzten Datenobjekte entfernt. Durch Aufrufen der zugehörigen Java-Funktionalität werden auch innerhalb der Java-VM alle Datenobjekte freigegeben. Diese wird aus Performancegründen nicht beendet, falls weitere Anforderungen von Seiten der Applikation kommen. Im Falle der Beendigung der Applikation erfolgt ein automatisches Entladen der CSP-

DLL aus dem Speicher durch Windows. Dies erzwingt gleichzeitig das Beenden aller von der CSP-DLL gestarteten Programme und beendet damit die Java-VM.

Im gesamten Ablauf ist zu beachten, dass die Gruppe Erzeugen-Hashen-Signieren-Zerstören (Aufrufe 2-7) mehrmals nacheinander benutzt werden kann - z.B. getrennt nur für den Inhalt der Nachricht und zusätzliche Informationen. Im Falle von Outlook wird eine Wiederholung benötigt, entsprechend jeweils zwei Aufrufen. Dies verlangt vom CSP damit die gleichzeitige Zwischenspeicherung und korrekte Indexierung *mehrerer* Hashinformationen.

Anmerkung:

Die angegebenen Funktionsaufrufe stellen die tatsächlich von Outlook erfolgten Aufrufe dar. Es wurden nur die benötigten Parameter angegeben. Informationen zu allen verfügbaren Parametern finden sich in [MSDN]. Alle angeführten Funktionen können den Gesamtprozess durch Setzen von Fehlerwerten und Rückgabe des Fehlerstatus abbrechen.

In der folgenden Tabelle wird der Zusammenhang von CSP-Aufrufen durch die Applikation, CSP-internen Funktionsaufrufen und Zugriffen auf die Java-Funktionalität zusammengefasst:

<i>Nr.</i>	<i>CSP-Funktion</i>	<i>Interne Funktion</i>	<i>Java-Funktion</i>
1.	CPAcquireContext	CDC_Java_StartVM	-
2.	CPCreateHash	-	-
3.	CPHashData	CDC_Java_Init	csp::CSP_Init csp::CSP_AddProv
4.	CPGetHashParam	-	-
5.	CPGetHashParam	-	-
6.	CPSignHash	CDC_Java_Sign	csp::CSP_GetCert csp::CSP_Sign
7.	CPDestroyHash	-	-
8.	CPReleaseContext	CDC_Java_Exit	csp::CSP_Exit

Tabelle 8: Funktionsaufrufe für CSP und Java

Abschließend ist der gesamte Ablauf in der folgenden Abbildung 26 dargestellt.

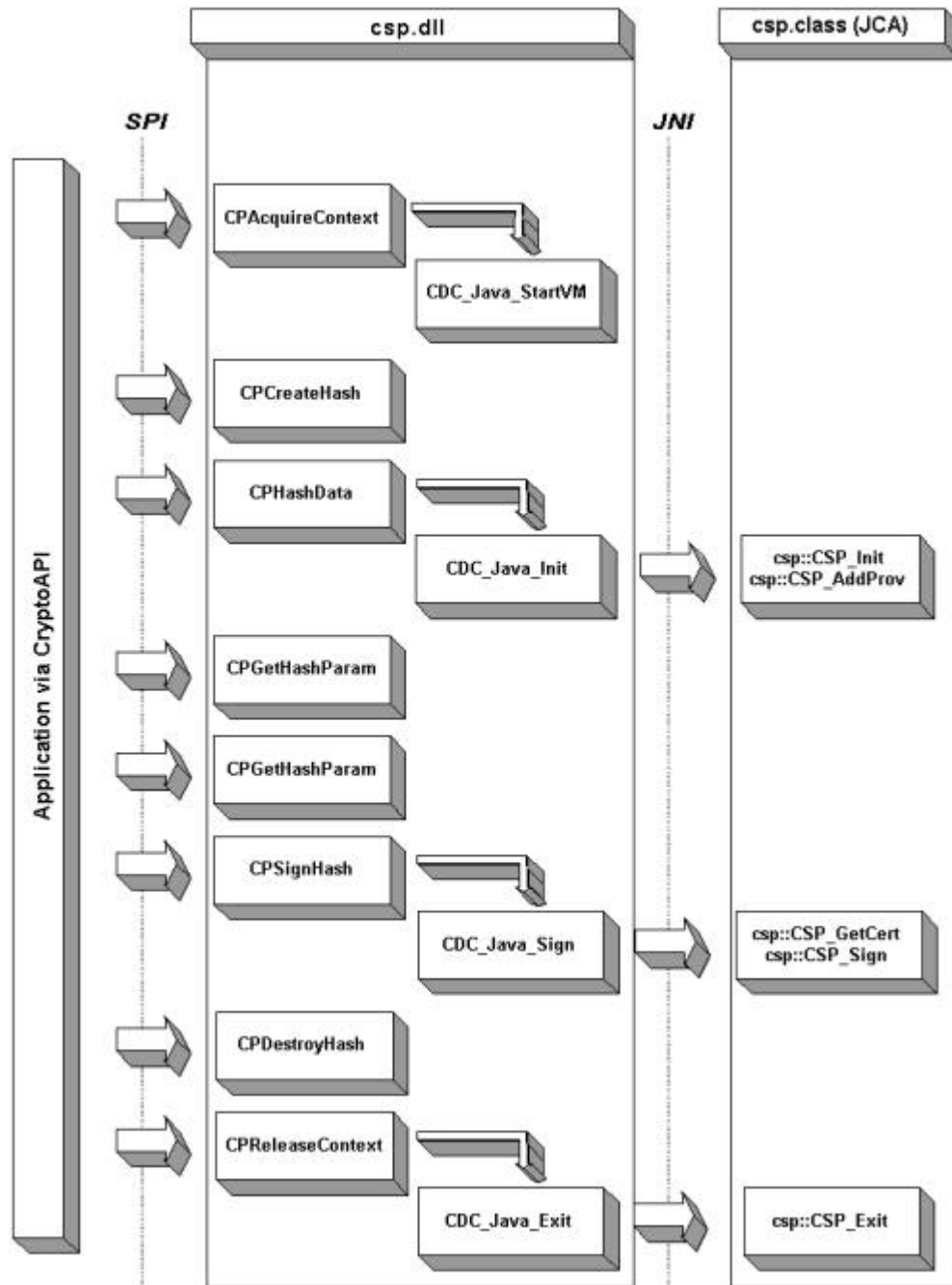


Abbildung 26: Abbilden der CryptoAPI-Semantik auf die JCA-Semantik zum Workflow ‚Signieren‘ in Outlook

Der Vollständigkeit halber erfolgt hier noch die Angabe der Funktionsaufrufe für die übrigen drei kryptographischen Verfahren Verifizieren, Ver- und Entschlüsseln.

Verifizieren einer Signatur wird analog zum Signieren gehandhabt, verwendet also die gleichen Funktionsaufrufe. Anstelle der Funktion CPSignHash wird CPVerifySignature aufgerufen. Hinzugekommen ist damit der Vergleich der beiden Signaturen.

<i>Nr.</i>	<i>Funktion</i>
1.	CPAcquireContext
2.	CPCreateHash
3.	CPHashData
4.	CPGetHashParam
5.	CPGetHashParam
6.	CPVerifySignature
7.	CPDestroyHash
8.	CPReleaseContext

*Tabelle 9a:
Die Funktionsaufrufe
beim Outlook-Workflow
,Verifizieren'*

Ver- und Entschlüsseln einer Nachricht können direkt abgebildet werden. Die genutzten Funktionsaufrufe sind hierbei für das Verschlüsseln:

<i>Nr.</i>	<i>Funktion</i>
1.	CPAcquireContext
2.	CPGetUserKey
4.	CPEncrypt
3.	CPExportKey
5.	CPDestroyKey
6.	CPReleaseContext

*Tabelle 9b:
Die Funktionsaufrufe
beim Outlook-Workflow
,Verschlüsseln'*

Und entsprechend für das Entschlüsseln:

<i>Nr.</i>	<i>Funktion</i>
1.	CPAcquireContext
2.	CPGetUserKey
3.	CPImportKey
4.	CPDecrypt
5.	CPDestroyKey
6.	CPReleaseContext

*Tabelle 9c:
Die Funktionsaufrufe
beim Outlook-Workflow
,Entschlüsseln'*

Im Unterschied zum Signieren wird hier kein Hash erzeugt; anstelle dessen muss der benötigte Private bzw. Public Key im CSP aktiviert werden. Auch hier gilt, dass Ver- bzw. Entschlüsselungs-Basis-Funktion jeweils mehrmals aufgerufen werden können.

Bemerkung und Kritik zur Benutzung symmetrischer Algorithmen:

Problematisch ist, dass keine Erweiterung um symmetrische Algorithmen in Outlook vorgesehen ist. Es hat sich durch Untersuchungen anhand der vom CSP zurückgelieferten Algorithmenliste herausgestellt, dass hier *nur* die von Outlook erwarteten Standardalgorithmen erkannt und alle anderen symmetrischen ignoriert werden.

Es besteht allerdings eine Möglichkeit, diese Vorgaben zu umgehen. Hierzu müsste man einen CSP als zu benutzenden Default-Provider eintragen und zur Applikationsseite hin vortäuschen, welche Algorithmen benutzt werden. Diese Maskierung muss auf beiden Seiten (Ver- und Entschlüsselung) eines solchen Prozesses analog erfolgen.

Problematisch ist die Unterscheidung, in welchem Falle die Maskierung auszuführen ist und in welchem nicht. Dies liegt daran, dass auch andere Teile des Betriebssystems auf die Funktionalität des Default-Providers zurückgreifen (z.B. die Dateiverschlüsselung im Filesystem unter Windows 2000). Hier muss der CSP also

- ☞ sowohl bei Bedarf (gekennzeichnet in den zugehörigen Zertifikaten) seine eigene symmetrische Verschlüsselung anbieten,
- ☞ als auch im anderen Falle die identische Mimik des üblichen Default-Providers des Betriebssystems nachahmen. Dies wäre z.B. über einfache Weiterreichung der Funktionsaufrufe an den vorher festgestellten ‚alten‘ Default-Provider möglich.

Ausserdem kann eine Erweiterung der symmetrischen Algorithmen von eigenen Applikationen genutzt werden. Dies ist allerdings für Standardapplikationen nicht von Nutzen.

Dieser Abschnitt beantwortet die in Fragestellung 5 und 8 aufgeworfenen Fragen, wie Applikationsworkflows auf den CSP und weiterhin auf die JCA abgebildet werden. Zudem ist die Benutzung der JCA durch den CSP von Applikationsseiten aus beschrieben worden, was Fragestellung 6 beantwortet. Insgesamt beschreibt dies die Benutzung der JCA durch Applikationen unter Windows mittels des CryptoAPIs und des CSPs.

4 Ergebnisse

4.1 Fragestellungen und Lösungen

Die in bisherigen Kapiteln aufgeworfenen Fragestellungen und die dazu präsentierten Lösungen lassen sich folgendermaßen zusammenfassen:

Fragestellungen:

4. *Wie nutzen Applikationen das CryptoAPI ?*
5. *Wie wird ein CSP vom CryptoAPI aufgerufen ?*
6. *Wie werden Object Identifier im System eingesetzt ?*
7. *Wie werden Zertifikate im System eingesetzt ?*
8. *Wie werden Applikationsworkflows auf den CSP abgebildet ?*
9. *Wie nutzt ein CSP die JCA?*
10. *Wie werden Zertifikate und OIDs in der JCA eingesetzt?*
11. *Wie wird ein Applikationsworkflow auf die JCA abgebildet?*

Durch Lösung dieser Fragestellungen lässt sich die abschliessende Gesamtfrage beantworten:

Können typische kryptographische Applikationsworkflows mittels des CryptoAPIs auf die JCA abgebildet werden ? Wenn ja, wie ?

Anhand des im dritten Kapitel beschriebenen Vorgehens und der vorher zusammengefassten Grundlagen auf Seiten von CryptoAPI und JCA ist die grundsätzliche Aussage zu treffen, dass dies prinzipiell möglich ist. Als unbedingt benötigte Teilkomponenten lassen sich analog der Beschreibung des Gesamtsystems bestimmen:

- A. Design und Implementierung des CSP unter Berücksichtigung aller besonderen Eigenschaften zur Abbildung auf die JCA,
- B. Design und Implementierung der Object Identifier-Registrierung und
- C. Design und Implementierung der Zertifikat-Publizierung.

Diese wurde unter Beachtung der geforderten Voraussetzungen des CryptoAPIs implementiert:

- I. Kein Zugriff auf Private Keys.
- II. Keine Beeinflussung der algorithmischen Abläufe im CSP.
- III. Authentifizierung der Benutzer nur durch den CSP.

Es läßt sich als positives Ergebnis zusammenfassen, dass die gewünschte Abbildung kryptographischer Workflows in Applikationen auf die JCA für die nahezu automatisierbare Erweiterung um asymmetrische Algorithmen, Object Identifier und die hierzu benötigten Zertifikate möglich ist. Eine Erweiterung um symmetrische Algorithmen ist vom Design der Windows-Schnittstellen her nicht gewünscht und nur unter Zuhilfenahme unsicherer Eingriffe möglich. Die Verbindung der kryptographischen Schnittstellen unter Abbildung der Windows- auf die Java-Schnittstellen als Ziel dieser Arbeit wird wie beschrieben hergestellt.

An dieser Stelle sei der gesamte Daten- und Workflow in Abbildung 27 zusammengefasst dargestellt. Hierin sind alle benutzten Applikationen, Datenstrukturen und Objekte entsprechend der zuvor festgestellten Lösungen eingetragen.

Ein weiteres Ergebnis ist der im Rahmen dieser Arbeit entstandene funktionsfähige Prototyp¹⁰⁸, dem die angeführten Code-Beispiele entnommen sind. Zusätzlich stehen alle notwendigen Informationen zur Installation der Komponenten in Form einer Anleitung, dem „CookBook“, zur Konfigurationsdatei und der verwendeten Softwareinstallation zur Verfügung.

¹⁰⁸ Der Quellcode des Prototypen ist auf Anfrage vom Autor über das Fachgebiet an der Technischen Universität Darmstadt erhältlich.

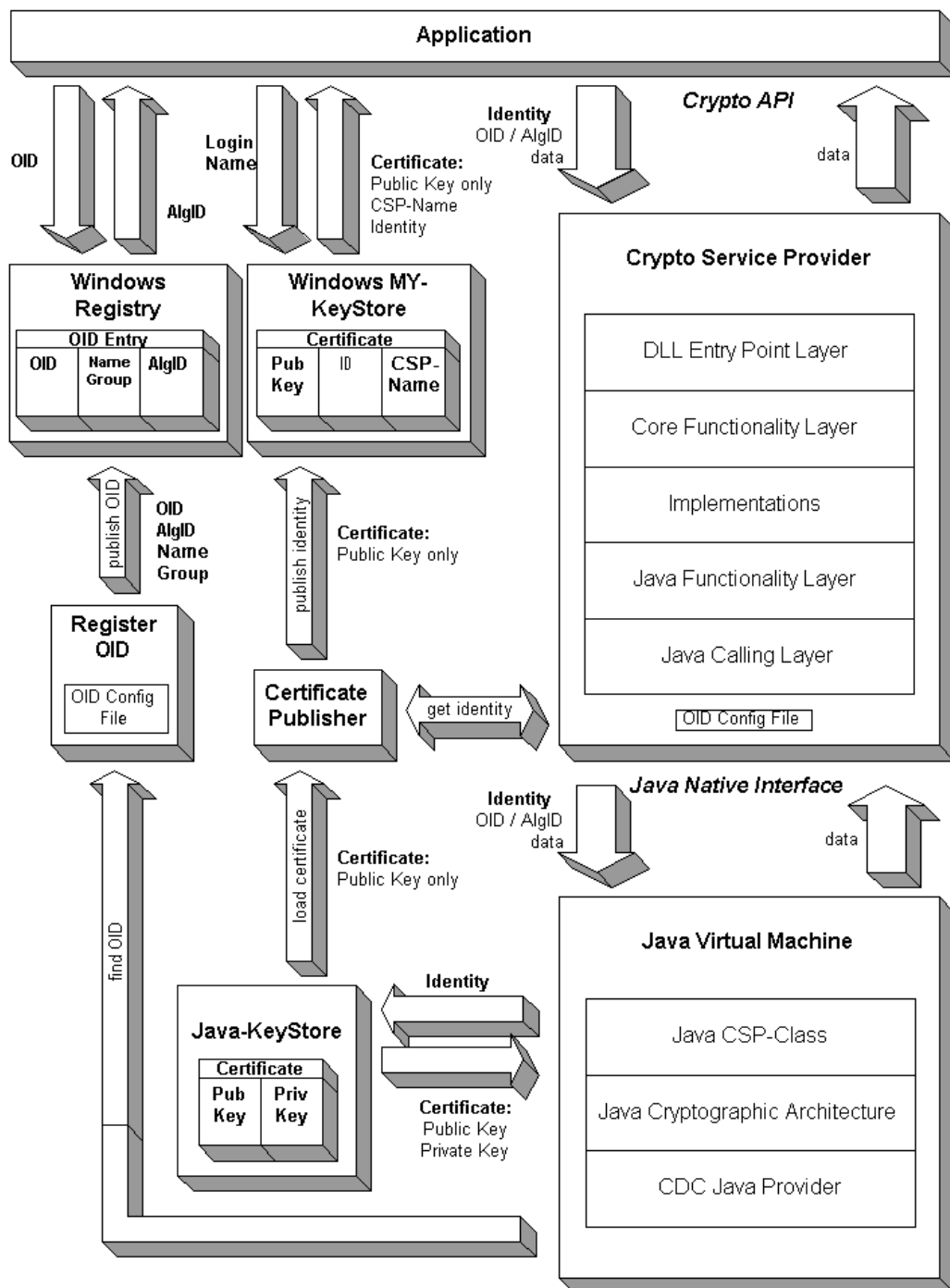


Abbildung 27: Das Gesamtsystem

4.2 Ausblick

Abgesehen von den bereits erreichten Ergebnissen ist es möglich, weitere Schritte der bisherigen Strategie zu verfolgen, um die Integration der JCA in die aktuelle Version des CryptoAPIs zu verbessern und an zukünftige Versionen anzupassen.

Vorteilhaft sind vor allem Tests des implementierten CSPs unter Benutzung einer breiteren Applikationsbasis, um die von anderen Applikationen gestellten Anforderungen in der tatsächlichen Anwendung besser beurteilen zu können.

Auch sind Verbesserungen der Implementierung hinsichtlich automatischer Erweiterung um neue Provider und um Multitasking- und Multithreading-Fähigkeiten von Interesse.

Weiterhin sollte der implementierte CSP als Releaseversion zur Signierung durch Microsoft eingereicht werden, um seine Anwendung auf Endbenutzer-Plattformen testen zu können.

Vorschläge für eine breitere Applikationsbasis:

1. Testen weiterer Microsoft-Applikationen. Dies sind z.B. Internet Explorer und Dateimanager inklusive der Dateisystemverschlüsselung in Windows 2000.
2. Testen von Windows-internen Prozessen, wie z.B. Treiberinstallation und Kontextmenüs in Windows, da diese den Default CSP aufrufen.
3. Feststellen von Unterschieden zwischen Outlook 2000 und Outlook Express Version 5.
4. Testen der zur Zeit erscheinenden neuen Software-Versionen (Outlook 2002/XP, Outlook Express 6, Internet Explorer 6, alle im Beta-Stadium) . Eventuell sind die Möglichkeiten hier gegenüber den aktuellen Versionen erweitert, da das CryptoAPI und die angebotenen Möglichkeiten für Applikationen auch in der Vergangenheit bereits stark erweitert wurden.
5. Testen von Standard-Applikationen anderer Hersteller, wie z.B. Netscape Messenger, Eudora Mail, Lotus Notes als Emailapplikationen oder von Programmen zur Verschlüsselung des Dateisystems.

Anhang

A.1 Installation eines CSP in Windows

Das hier angeführte Programm wird benötigt, um die CSP-DLL in Windows zu registrieren. Die Originalversion stammt aus dem CSP Development Kit und wurde an den speziellen CSP angepasst [CSPDK].

```
#define CSP_NAME "CDC Crypto Service Provider" // the CSPs name
#define CSP_VERSION 0x00000100 // and its version (1.0)
#define CSP_TYPE 1 // PROV_RSA_FULL

// Registry Keys
CHAR szprovider[] = "SOFTWARE\\Microsoft\\Cryptography\\Defaults\\Provider\\CDC
Crypto Service Provider";
CHAR szdef[] = "SOFTWARE\\Microsoft\\Cryptography\\Defaults\\Provider\\list ein
Testprovider";
CHAR szcsp[] = "SOFTWARE\\Microsoft\\Cryptography\\Defaults\\Provider\\CDC
Crypto Service Provider";
CHAR szImagePath[] = "c:\\WINNT\\csp.dll";

DWORD dwIgn;
HKEY hKey;
DWORD err;
DWORD dwValue;
HANDLE hFileSig;
DWORD NumBytesRead;
DWORD lpdwFileSizeHigh;
LPVOID lpvAddress;
DWORD NumBytes;

int __cdecl main(int cArg, char *rgszArg[])
{
    // Just to open csp.dll signature file. This file was created by
    // sign.exe.
    if ((hFileSig = CreateFile("c:\\WINNT\\cspsign",
        GENERIC_READ, 0, NULL,
        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
        0)) != INVALID_HANDLE_VALUE)
    {
        if ((NumBytes = GetFileSize((HANDLE) hFileSig, &lpdwFileSizeHigh)) ==
            0xffffffff)
        {
            printf("Install failed: Getting size of file cspsign: %x\n",
                GetLastError());
            CloseHandle(hFileSig);
            return(FALSE);
        }

        if ((lpvAddress = VirtualAlloc(NULL, NumBytes, MEM_RESERVE |
            MEM_COMMIT,
            PAGE_READWRITE)) == NULL)
        {
            CloseHandle(hFileSig);
            printf("Install failed: Alloc to read uisign: %x\n",
                GetLastError());
            return(FALSE);
        }
    }
}
```

```
if (!ReadFile((HANDLE) hFileSig, lpvAddress, NumBytes,
             &NumBytesRead, 0))
{
    CloseHandle(hFileSig);
    printf("Install failed: Reading uisign: %x\n",
           GetLastError());
    VirtualFree(lpvAddress, 0, MEM_RELEASE);
    return(FALSE);
}

CloseHandle(hFileSig);

if (NumBytesRead != NumBytes)
{
    printf("Install failed: Bytes read doesn't match file size\n");
    return(FALSE);
}

// Create or open in local machine for provider:
//
if ((err = RegCreateKeyEx(HKEY_LOCAL_MACHINE,
                        (const char *) szprovider,
                        0L, "", REG_OPTION_NON_VOLATILE,
                        KEY_ALL_ACCESS, NULL, &hKey,
                        &dwIgn)) != ERROR_SUCCESS)
{
    printf("Install failed: RegCreateKeyEx\n");
}

// Set Image path to: csp.dll
//
if ((err = RegSetValueEx(hKey, "Image Path", 0L, REG_SZ, szImagePath,
                        strlen(szImagePath)+1)) != ERROR_SUCCESS)
{
    printf("Install failed: Setting Image Path value\n");
    return(FALSE);
}

// Set Type to: CSP_TYPE
//
dwValue = CSP_TYPE;
if ((err = RegSetValueEx(hKey, "Type", 0L, REG_DWORD,
                        (LPTSTR) &dwValue,
                        sizeof(DWORD))) != ERROR_SUCCESS)
{
    printf("Install failed: Setting Type value: %x\n", err);
    return(FALSE);
}

// Place signature
//
if ((err = RegSetValueEx(hKey, "Signature", 0L, REG_BINARY,
                        (LPTSTR) lpvAddress,
                        NumBytes)) != ERROR_SUCCESS)
{
    printf("Install failed: Setting Signature value for cspsign: %x\n",
           err);
    return(FALSE);
}

RegCloseKey(hKey);
VirtualFree(lpvAddress, 0, MEM_RELEASE);
```

```
// Create or open in local machine for provider type:
//
if ((err = RegCreateKeyEx(HKEY_LOCAL_MACHINE,
                        (const char *) sztype,
                        0L, "", REG_OPTION_NON_VOLATILE,
                        KEY_ALL_ACCESS, NULL, &hKey,
                        &dwIgn)) != ERROR_SUCCESS)
{
    printf("Install failed: Registry entry existed: %x\n", err);
}

if ((err = RegSetValueEx(hKey, "Name", 0L, REG_SZ, CSP_PROV,
                        strlen(CSP_PROV)+1)) != ERROR_SUCCESS)
{
    printf("Install failed: Setting Default type: %x\n", err);
    return(FALSE);
}

printf("Installed: %s\n", szImagePath);
}
else printf("Error: wrong pathnames.\n Can't find file cspsign.\n");
return(FALSE);
}
```

Vorgehensweise der CSP-Installation im Programm:

1. Existenzüberprüfung der Signaturdatei
2. Einlesen und Prüfen der Daten
3. Öffnen/Erzeugen der Registry-Eintragung
4. Setzen von
 - a. Pfadname der CSP-DLL
 - b. Typ des Providers
 - c. Signatur der CSP-DLL

A.2 Das Cookbook für die CDC-CSP-Installation

(Version. 1.3, 28.04.01)

Folgende Aktionen müssen in gegebener Reihenfolge ausgeführt werden:

1. Verändern der Windowsinstallation
2. Installieren der Java-Runtime-Umgebung
3. Installation des CSP (nur ausführbar nach Erfolg von 1. und 2.)
4. Registrierung der OIDs
5. Publizieren des Zertifikates (nur ausführbar nach Erfolg von 3. und 4.)

Benötigte Files:

- A. Die der Betriebssystemversion entsprechende Datei advapi32.dll aus dem CSPDK
- B. Das sign-Tool zum Betriebssystem aus dem CSPDK: sign.exe oder signw2k.exe (Windows 2000)
- C. Der CSP: csp.dll
- D. Das CSP-Registrierungstool: regcsp.exe
- E. Das OID-Registrierungstool: regoid.exe
- F. Das zugehörige Konfigurationsfile: oid.cfg
- G. Das Zertifikatstool: pubcert.exe
- H. Das Zertifikatsfile des Benutzers als X.509 Zertifikat, base64- oder DER-codiert
- I. Das Zertifikatsfile der TestCA als X.509 Zertifikat, base64- oder DER-codiert
- J. Setup-Files des aktuellen Java2-SDKs (ist inklusive JRE & JNI), alternativ jre.zip
- K. Java-Klassenpaket java.zip

Die Dateien A-H sind im cookbook.zip enthalten.

Das Java Runtime Environment Paket I ist erhältlich von Sun, alternativ kann die gekürzte Fassung aus jre.zip benutzt werden.

Die eigentlichen Java-Klassenfiles und JCA-Provider sind im Paket java.zip enthalten.

Im allgemeinen sollten die Programme in der DOS-Box gestartet werden; falls sie von Windows aus gestartet werden, können die Ausgaben auf der Konsole nicht verfolgt werden. Ausserdem sollte diese DOS-Box wegen der Zeilenumbrüche der Ausgaben exakt 80 Zeichen breit sein (Standardgröße).

1. Verändern der Windowsinstallation

<i>Prinzip:</i>	Die Datei <code>advapi32.dll</code> von Windows muß mit der korrekten Version dieser Datei aus dem CSPDK ersetzt werden.
<i>Zweck:</i>	Die Laufzeit-Überprüfung der Signaturen von Betriebssystem-DLLs (die <code>msp.dll</code>) abschalten.
<i>Benötigte Files:</i>	(A) Die der Betriebssystemversion entsprechende Datei <code>advapi32.dll</code> aus dem CSPDK.
<i>ToDo:</i>	<p>(so nur ausführbar, falls das Dateisystem der Windows-Partition FAT32 ist).</p> <ol style="list-style-type: none"> 1. Kopieren der DLL nach <code>C:\</code> oder <code>C:\{WINDIR}</code> (Sonst u.U. nicht erreichbar). Ab jetzt darf Windows nicht mehr aktiv sein (keine DOS-Box!). 2. Also Booten von einer Notfalldisk (98/ME) oder Startdisk (NT4) oder der Installations-CD (2000). 3.a Bei 98/ME/NT4: Austausch der Datei von MS-DOS 7 aus. 3.b Bei 2000: Reparaturkonsole zum Austausch benutzen (Administrator-Passwort nötig). 4. Reboot mit der neuen DLL.

2. Installation der Java-Runtime-Umgebung

<i>Prinzip:</i>	Installation des Java-SDK mit dem Setup-Dateien und Setzen der Pfadinformation.
<i>Zweck:</i>	Installation der Java-Runtime-Umgebung.
<i>Benötigte Files:</i>	(J) Setup-Files des aktuellen Java2-SDKs oder <code>jre.zip</code> ; (K) <code>java.zip</code>
<i>ToDo:</i>	<ol style="list-style-type: none"> 1. Entpacken von <code>java.zip</code> nach <code>C:\java</code>. 2. Installieren des Java2SDKs nach <code>C:\java</code> (nicht der Standardpfad). Alternativ von <code>jre.zip</code> nach <code>C:\java</code>. 3. Erweitern des Systempfades unter Startmenü\Einstellungen\System\Erweitert Unter Umgebungsvariablen\Systemvariablen\ die Variable ‚path‘ Um den Wert: ‚<code>C:\java;C:\java\jre\bin\classic</code>‘.

3. Installation der CSP.dll

<i>Prinzip:</i>	Der CSP muss ins Systemverzeichnis kopiert, signiert und registriert werden.
<i>Zweck:</i>	Installation des CSP in Windows.
<i>Benötigte Files:</i>	(B,C,D) sign.exe/signw2k.exe, csp.dll, regcsp.exe (und oid.cfg)
<i>ToDo:</i>	<ol style="list-style-type: none"> 1. Kopieren aller drei Dateien nach C:\{WINDIR}. Standard 98/ME: C:\WINDOWS Standard NT4/2000: C:\WINNT\ 2. Signieren der csp.dll. 98/ME/NT4: "sign s csp.dll cpsign" 2000: "signw2k s csp.dll cpsign" 3. Ausführen von regcsp.exe (benutzt das File cpsign). 4. Kein Reboot nötig.

4. Registrierung der OIDs

<i>Prinzip:</i>	Registrierung der genutzten OIDs in der Registry.
<i>Zweck:</i>	Windows das Erkennen neuer Objekte ermöglichen.
<i>Benötigte Files:</i>	(E,F) regoid.exe und oid.cfg
<i>ToDo:</i>	<ol style="list-style-type: none"> 1. Ausführen von regoid.exe mit dem Konfigurationsfile oid.cfg im gleichen Verzeichnis.

5. Publizieren des Zertifikates

<i>Prinzip:</i>	CSP-bezogenes Publizieren des Zertifikates (X.509, base64 oder DER-codiert).
<i>Zweck:</i>	Für den User ein Zertifikat mit CSP-Info und Public Key installieren in Windows.
<i>Benötigte Files:</i>	(G) pubcert.exe (H) Benutzer, (I) TestCA
<i>ToDo:</i>	<ol style="list-style-type: none"> 1. „pubcert.exe <cert.cer>“ mit dem Zertifikatsfile (H) im gleichen Verzeichnis. 2. Mit dem CertMgr das Bezugswurzelzertifikat (I) der TestCA importieren; dies sollte per automatischer oder manueller Auswahl in einen der beiden Stores „CA“ oder „Root“ importiert werden (Zwischen- oder Stammzertifizierungsstellen). [Der CertMgr kann vom Platform SDK oder InternetExplorer aus aufgerufen werden.]

Überprüfen der Installation:

- A. Installation eines lokalen MailServers zum Mail-Senden ohne Netzwerk oder eines Exchange-Servers im Netzwerk.
- B. Aufruf von Outlook oder Outlook Express.
(oder direkt des CertManagers aus dem Platform SDK).
- C. „Enumerate Algorithms“: Hier wird unter den Kontoeinstellungen bzw. Optionen/Sicherheit zur Zertifikatsauswahl der Cert-Manager aufgerufen. Jetzt kann hiermit das publizierte Zertifikat (default: „Sachbearbeiter 1“) ausgewählt werden. Dabei sollten Message-Boxes vom aufgerufenen CSP erscheinen, mit „OK“ geht es weiter.
- D. Vorsicht: Die Email-Adresse im Konto **muss** mit derjenigen aus dem Zertifikat übereinstimmen, andernfalls ist es nicht zu sehen und kann nicht benutzt werden.
- E. Signieren von Emails:
 - 1. Einstellen von Signieren unter Optionen/Sicherheit.
 - 2. Schreiben einer neuen Mail.
 - 3. Der Senden-Button löst den Aufruf des CSP zum Signieren aus.
 - 4. Erkennbar ist dies an den Meldungen, die von den aufgerufene CSP-Funktionen gezeigt werden.
(Diese dienen dem Debugging und sollten in einer Release-Version abgeschaltet werden).

A.3 Aufbau der Konfigurationsdatei für die OID-Informationen

Wie bereits beschrieben wird eine Konfigurationsdatei für die OID-Informationen verwendet. Für diese wurde ein eigenes, textbasiertes Dateiformat entwickelt. Es existieren definierte Identifier-Strings, denen jeweils der gewünschte Wert folgt. Ein Eintrag wird erkannt, sobald alle zugehörigen Strings korrekt gelesen wurden.

Identifier:

```
# COUNT=1          # this is the number of entries placed in the file
```

Anzahl der Einträge.

Dieser Wert wird gelesen, falls vorhanden.

Dient in der aktuellen Version nur noch der Überprüfung, es werden alle Einträge gelesen und zur Registrierung zur Verfügung gestellt.

```
# ALGID=ff01      # the algorithm id, a hex digit (4 counting chars)
```

AlgorithmID

Definition nach Microsoft, vgl. [MSDN].

Integer in Hexadezimal mit 4 zählenden ziffern.

Da sie CSP-intern ist, frei wählbar. Sollte allerdings den Microsoft-Konventionen entsprechen.

Liste bereits benutzter Werte in [MSDN, WINCRYPT.H].

```
# OID='1.2'       # the OID string, ints with dots in between
```

OID

Definition nach [PKIX,RFC1778,CDC-OID].

String aus Integern in dot-notation, eingefasst von einfachen Anführungszeichen (').

Welweit eindeutig, muss beantragt werden.

Liste unter [ALVESTRAND].

```
# NAME='a b'      # the friendly name string, placed between ''
```

Name des Algorithmus oder Objektes in Klartext.

String mit Leerzeichen eingefasst von einfachen Anführungszeichen (')..

```
# GROUP=3        # the group ID, int from 1..7
```

Gruppeninformation des Algorithmus oder Objektes

Integer zwischen 1..7;

Liste der Werte analog [MSDN] CRYPT_OID_INFO.dwGroupID;

Zu Beachten beim Erzeugen der Datei:

- ~~/~~ Falls eingetragene Identifier nicht korrekt geschrieben sind, werden sie ignoriert.
- ~~/~~ Falls eingetragene Werte nicht dem korrekten Typ entsprechen, werden sie ignoriert.
- ~~/~~ Die Reihenfolge der Identifier pro Eintrag muss eingehalten werden.
- ~~/~~ Zeilen werden nur bis zu CR/LF-Zeichen gelesen
- ~~/~~ # ist das Kennzeichen für Kommentare, der Rest der Zeile wird ignoriert.
- ~~/~~ Sowohl Leerzeichen als auch leere Zeilen werden ignoriert.
- ~~/~~ Zeilen haben maximal 255 Zeichen.

Beispiel (gekürzt)

```

# this is an automatic generated file,
# do only edit if you know what you're doing !
#
#
# oid.cfg
#
# this is the config file for the OID information structures
# - used by regoid.cfg
# - will probably be used by csp.dll
# - will be generated out of the data from the JCA
#
#
# Copyright (C) 2001 Torsten Vest   All Rights Reserved
#
#
#
# file format:
#
# Identifiers:
# COUNT=1           # this is the number of entries placed in the file
#                   # can be bigger than the actual number, but never smaller
# ALGID=ff01        # the algorithm id, a hex digit (4 counting chars)
# OID='1.2'         # the OID string, ints with dots in between, placed
#                   # between ''
# NAME='a b'        # the friendly name string, placed between ''
# GROUP=3           # the group ID, int from 1..7
#
# - All identifiers must been found exactly in this order
# - The value is build from all chars after the identifier until CR/LF
# - Remarks are marked with a '#'; rest of the line is skipped
# - All lines with none or an uncorrect identifier are skipped
# - Whitespace before an identifier is overread, blank lines are skipped
# - Max length of one line in file is 255 chars
#
# - See documentation for setting of ALGID, OID and GROUP
#

COUNT=3           # this is the number of entries placed in the file
                   # can be bigger than the actual number, but never smaller

# ECDSA Public Key Object
ALGID=0000          # the algorithm id, a hex digit (4 chars)
OID='1.2.840.10045.2.1' # the OID string, ints in dot-notation
NAME='ECDSA Public Key Object' # the friendly name string
GROUP=7            # the group ID, int from 1..7

# ECDSA with SHA1 signature
ALGID=2001          # the algorithm id, a hex digit (4 chars)
OID='1.2.840.10045.4.1' # the OID string, ints in dot-notation
NAME='ECDSA-SHA1'   # the friendly name string
GROUP=4            # the group ID, int from 1..7

# ECElGamal
ALGID=6002          # the algorithm id, a hex digit (4 chars)
OID='1.3.6.1.4.1.8301.3.1.1' # the OID string, ints in dot-notation
NAME='EC-ElGamal'   # the friendly name string
GROUP=3            # the group ID, int from 1..7

```

A.4 Verwendete Softwareinstallation

Betriebssysteme:

- ☞ Microsoft Windows 2000 Deutsch, kein Servicepack
- ☞ Microsoft Windows NT 4 Deutsch, Servicepack 6a
- ☞ Microsoft Windows 98 Deutsch
- ☞ Microsoft Windows 98 Second Edition Deutsch
- ☞ Microsoft Windows ME (Millenium Edition) Deutsch, kein Servicepack

Entwicklungsinstallation:

Betriebssystem	Microsoft Windows 2000 Deutsch, kein Servicepack (5.00.2195)
Entwicklungs- umgebung „C“	Microsoft Visual Studio Enterprise 6.0 Deutsch (Visual C/C++), Servicepack 4 benötigt MDAC 2.5 Englisch
Zusätzliche Entwicklungs- werkzeuge	Microsoft Platform SDK Juli 2000 (V. 2201) (Beinhaltet den Certificate Manager)
Entwicklungs- umgebung „Java“	Java 2 SDK Standard Edition V.1.2.2 (1.2.2_006) Java 2 Runtime Environment V.1.2.2 (1.2.2_006) Java Native Interface V.1.1
CDC-Provider	Standard-Provider : Version 1.9 vom 12.02.2001 EC-Provider: Version 1.9 vom 12.02.2001
Office-Paket	Microsoft Office 2000 Premium Deutsch (9.0.2812)
Webbrowser	Microsoft Internet Explorer 5.5 Deutsch 56bit-Kryptographie (5.50.4134.0600) Microsoft Internet Explorer 6.0 Deutsch Beta1 128bit- Kryptographie ¹⁰⁹ (6.00.2462.0000)
Email-Client	Outlook 2000 Deutsch (9.0.0.2184) Outlook 2000 Deutsch SR1 (9.0.0.3821) Outlook Express 5.0 Deutsch (5.00.2919.6700) Outlook Express 5.5 Deutsch (5.50.4133.2400) Outlook Express 6.0 Deutsch Beta1 (6.00.2462.00)
Dokumentation	MSDN Library Visual Studio 6.0a Deutsch Oktober 2000 MSDN Library Deutsch Oktober 2000 MSDN Library Deutsch Januar 2001 CryptoAPI-Mailing Liste [CAPI-LIST]
Grafiken	Microsoft Visio 2000 SR1 Deutsch (6.0.2072)
Spezielle Erweiterungen	Das Windows Enhanced Encryption Pack wurde aus Kompatibilitätsgründen nicht genutzt.

¹⁰⁹ Dies ist nicht unter Windows 2000 möglich. Genutzt wurde hierfür Windows ME.

Begriffe

Begriff	Bedeutung
Absender	Der Benutzer, der eine Nachricht sendet. (Signieren und Verschlüsseln)
Adressat	Der Benutzer, für den eine Nachricht bestimmt ist. (Verifizieren und Entschlüsseln)
AlgID, AlgorithmID	Identitätsinformation für einen Algorithmus. Microsoftstandard vom Typ DWORD.
ASN.1, Abstract Syntax Notation One	Notation zur Beschreibung von Object Identifiern in einer formalen Sprache.
Asymmetrische Kryptographie (PublicKey Cryptography)	Kryptographie unter Nutzung zweier unterschiedlicher Schlüssel. Einer der beiden wird geheimgehalten, der zweite wird veröffentlicht.
Backus-Naur-Form (BNF)	Mathematische Definition zur Beschreibung formaler Sprachen. Findet Verwendung in ASN.1.
Base64-Kodierung	Ein Verfahren zur 7-Bit-ASCII Kodierung von Daten. Findet Verwendung beim Versand von Emails.
Benutzeridentität	Information zur Referenzierung des Benutzers. Meist wird hierfür sein vollständiger Name bzw. der Windows Login-Name genutzt.
Chiffre (Cipher Text)	Verschlüsselter Text bzw. Nachricht.
CryptoAPI	Die kryptographische Schnittstelle in Windows. Dies bezeichnet in der Literatur sowohl das Gesamtsystem als auch die Programmierschnittstelle an sich.
CSP	Crypto(graphic) Service Provider. Dienstprogramm des Windows CryptoAPI, das kryptographische Verfahren zur Verfügung stellt.
CSPDK	Crypto(graphic) Service Provider Development Kit. Werkzeug zur Entwicklung eines CSP.
Default Provider / CSP	Der als Standard in der Windows Registry eingetragene Provider. Wird verwendet, falls in einem Funktionsaufruf kein anderer Provider explizit angegeben ist.
DER-Kodierung	Weiteres Verfahren zur 7-Bit-ASCII Kodierung von Daten. Findet Verwendung beim Versand von Emails.
Elliptic-Curve Digital Signature Algorithm	Signatur-Algorithmus, der zur Berechnung elliptische Kurven nutzt.
Hardwarebasierter CSP	Ein CSP, der Smartcards oder ähnliche, in Hardware implementierte, Funktionalität nutzt.
Hash	(Hier) Die Abbildung einer beliebig langen Information auf eine endliche Länge mittels eines Hashalgorithmus. (Mathematisch) Die Abbildung beliebiger Zahlwerte auf eine endliche Menge mittels einer Hashfunktion.
Identität	Siehe Benutzeridentität.
Initialisierungsvektor	Menge von Startwerten zur Initialisierung eines Algorithmus.
Integrität einer Nachricht	Eine Nachricht wurde beim Transport nicht verändert.
Key Container	Datenstruktur, die alle Informationen zu einer Identität und der zugehörigen Keys speichert.

Key	Kryptographischer Schlüssel, mit einer Benutzeridentität assoziiert.
Klartext (Plain Text)	Nicht verschlüsselter Text bzw. Nachricht
Layer	Schicht einer Mehrschichten-Architektur eines Programmes.
Object Identifier	Identifizierungsmethode für Objekte wie Algorithmen und Keys. In Unterscheidung zu OID als Sammelbegriff für OID und AlgorithmID genutzt.
OID	Ausgeschrieben: Object Identifier Weltweite Standardbeschreibung zur Identifizierung.
Padding / Padden	Auffüllen eines Textes mit Füllzeichen (Leerzeichen), um ein Vielfaches einer bestimmten Größe zu erreichen.
Private Key	Bei asymmetrischer Kryptographie derjenige Key, der geheimgehalten wird. Bei symmetrischer Kryptographie der einzige, auch geheimgehaltene.
Public Key	Der veröffentlichte Key. Nur bei asymmetrischer Kryptographie erforderlich.
Provider	Dienstprogramm, das anderen Programmen Algorithmen zur Verfügung stellt. Hier synonym zu CSP genutzt.
Session Key	Einmalig genutzter symmetrischer Schlüssel zur Textverschlüsselung, der anschließend selbst asymmetrisch verschlüsselt und mit einer Nachricht verschickt wird.
Signatur	Digitale Unterschrift, die die Integrität einer Nachricht sicherstellt.
Smartcards	Chipkarten, die Keys speichern und intern Algorithmen ausführen können.
Softwarebasierter CSP	Ein CSP, der ausschließlich in Software implementiert ist und keine Hardwareerweiterungen nutzt.
Symmetrische Kryptographie (Private Key Cryptography)	Kryptographie unter Nutzung nur eines Schlüssels. Dieser muß geheimgehalten werden.
Typ des Keys	Keyexchange (Schlüsselaustausch) oder Signature
Zertifikat	Datenstruktur oder Datei, in der alle Informationen zu Benutzeridentität, Schlüsseln, verwendeten Algorithmen usw. gespeichert sind. Es kann ausschließlich einen Public Key beinhalten (fremde Zertifikate), aber auch zusätzlich einen Private Key (eigene Zertifikate).

Abkürzungen

<i>Abkürzung</i>	<i>Bedeutung</i>
ANSI	American National Standards Institute http://www.ansi.org/
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One, nach ITU-T Rec. X.208 bzw. ISO 8824/8825
DES	Digital Encryption Standard (Verschlüsselungsalgorithmus)
DLL	Dynamic Link Library
DSA	Digital Signature Algorithm
EC, ECDSA	Ellipic Curves (Digital Signature Algorithm)
IANA	Internet Assigned (Names and) Numbers Authority http://www.iana.org/
IEC	International Electrotechnical Commision http://www.iec.ch/
IETF	Internet Engineering Task Force http://www.ietf.org/
IP	Internet Protocol
ISO	International Organization for Standardization http://www.iso.ch/
ITU, ITU-T	International Telecommunication Union http://www.itu.int/
JCA	Java Cryptographic Architecture
JCE	Java Cryptographic Extension
JNI	Java Native Interface
MD2, MD5	Message Digest2 bzw. 5 (Hash-Algorithmen)
OID	Object Identifier
PKI	Public Key Infrastructure
PKIX	Public Key Infrastructure Working Group der IETF http://www.ietf.org/html.charters/pkix-charter.html
RC2, RC4	RC2-Verschlüsselungsalgorithmus nach RFC2268 bzw. Algorithmus RC4
RFC	Request for Comment, Dokumentation bereits akzeptierter Internet-Standards. Frei erhältlich unter http://www.cis.ohio-state.edu/Services/rfc/index.html http://www.faqs.org/rfcs/ http://www.uni-siegen.de/security/
RSA	Gruppe von Algorithmen benannt nach ihren Erfindern Rivest-Shamir-Adleman
SHA, SHA1	Secure Hash Algorithm (1)
SPI	Service Provider Interface
VM, Java-VM	(Java-) Virtual Machine

Abbildungen, Tabellen, Code-Beispiele

Abbildungen:

<i>Abbildung</i>	<i>Bezeichnung</i>	<i>Seite</i>
Abbildung 1 ¹¹⁰	Überblick über die Architektur des CryptoAPI	7
Abbildung 2 ¹¹⁰	Beziehungen der Architekturelemente	9
Abbildung 3 ¹¹⁰	Informationsfluß beim Signieren einer Nachricht	10
Abbildung 4 ¹¹⁰	Informationsfluß beim Verifizieren einer signierten Nachricht	12
Abbildung 5 ¹¹⁰	Informationsfluß beim Verschlüsseln einer Nachricht	14
Abbildung 6 ¹¹⁰	Informationsfluß beim Entschlüsseln einer verschlüsselten Nachricht	16
Abbildung 7	Prinzipielle Architektur eines Crypto Service Providers	23
Abbildung 8	Abbildungsmechanismus im CryptoAPI	32
Abbildung 9	Prinzipielle Registrierung von Object Identifiern	37
Abbildung 10	Ansicht eines Zertifikates im Zertifikatsmanager von Windows	38
Abbildung 11	Prinzipielles Publizieren von Zertifikaten	40
Abbildung 12	Eigenschaften eines Kontos bei Outlook	42
Abbildung 13a	Ändern der Sicherheitseinstellungen in Outlook	43
Abbildung 13b	Ändern der Sicherheitseinstellungen in Outlook	43
Abbildung 14	Architektur der Java-Funktionalität	46
Abbildung 15a	Abbildung des Signierens	55
Abbildung 15b	Abbildung des Verifizierens	56
Abbildung 15c	Abbildung des Verschlüsseln	56
Abbildung 15d	Abbildung des Entschlüsseln	57
Abbildung 16	Abilden der CryptoAPI-Semantik auf die JCA-Semantik	58
Abbildung 17	Geplantes Gesamtsystem	60
Abbildung 18	Architektur des CDC-Crypto Service Providers	62
Abbildung 19	Registrierung von Object Identifiern für den CDC-Crypto Service Provider	83
Abbildung 20	OID-Definitionen des CDC	85
Abbildung 21	Publizieren von Zertifikaten für den CDC-Crypto Service Provider	89
Abbildung 22ab	Kontoeigenschaften und Sicherheitseinstellungen in Outlook	92
Abbildung 23	Erstellen einer signierten Nachricht in Outlook	93
Abbildung 24	Posteingang mit signierter Nachricht in Outlook	94
Abbildung 25	Informationen zur Signatur der Nachricht in Outlook	94
Abbildung 26	Abilden der CryptoAPI-Semantik auf die JCA-Semantik zum Workflow ‚Signieren‘ in Outlook	101
Abbildung 27	Das Gesamtsystem	106

¹¹⁰ Anmerkung: Abbildungen 1-6 sind analog MSDN-Dokumentation mit Änderungen.

Tabellen:

<i>Tabelle</i>	<i>Bezeichnung</i>	<i>Seite</i>
Tabelle 1	Prototyp der Funktion CryptAcquireContext	18
Tabelle 2a	Entry Point Funktionen des CryptoAPI – „Anbindung an den CSP“	28
Tabelle 2b	Entry Point Funktionen des CryptoAPI – „Erzeugen und Verwalten von Keys“	28
Tabelle 2c	Entry Point Funktionen des CryptoAPI – „Hashen von Daten“	29
Tabelle 2d	Entry Point Funktionen des CryptoAPI – „Signieren und Verifizieren,,	29
Tabelle 2e	Entry Point Funktionen des CryptoAPI – „Ver- und Entschlüsseln“	29
Tabelle 3	Prototyp der Funktion CPAcquireContext	31
Tabelle 4	Zertifikat-Stores in Windows	39
Tabelle 5, Teil a-e	Beschreibung der Implementierung der Funktion CPAcquireContext	67-69
Tabelle 6	Die CSP-Funktionsaufrufe beim Outlook-Workflow ‚Signieren‘	95
Tabelle 7a	Aufrufparameter der Funktion CPAcquireContext	96
Tabelle 7b	Aufrufparameter der Funktion CPCreateHash	96
Tabelle 7c	Aufrufparameter der Funktion CPHashData	97
Tabelle 7d	Aufrufparameter der Funktion CPGetHashParam	97
Tabelle 7e	Aufrufparameter der Funktion CPGetHashParam	98
Tabelle 7f	Aufrufparameter der Funktion CPSignHash	98
Tabelle 7g	Aufrufparameter der Funktion CPDestroyHash	99
Tabelle 7h	Aufrufparameter der Funktion CPReleaseContext	99
Tabelle 8	Funktionsaufrufe für CSP und Java	100
Tabelle 9a	Die Funktionsaufrufe beim Outlook-Workflow ‚Verifizieren‘	102
Tabelle 9b	Die Funktionsaufrufe beim Outlook-Workflow ‚Verschlüsseln‘	102
Tabelle 9c	Die Funktionsaufrufe beim Outlook-Workflow ‚Entschlüsseln‘	102

Code-Beispiele:

<i>Beispiel</i>	<i>Bezeichnung</i>	<i>Seite</i>
Bsp. 1	Einfache Verschlüsselung mit dem CryptoAPI	20
Bsp. 2	Einfaches Signieren mit der JCA	48
Bsp. 3	Starten einer Java-VM mit dem JNI	51
Bsp. 4	ID-Bestimmung zum Funktionsaufruf mit dem JNI	52
Bsp. 5a	Funktionsaufruf mit dem JNI	53
Bsp. 5b	Datenmanipulation mit dem JNI	54
Bsp. 6	Implementierung der CSP-Funktion CPAcquireContext	65,66
Bsp. 7	Implementierung der CSP-Funktion CDC_SetHashinfo	75
Bsp. 8	Implementierung der CSP-Funktion CDC_Java_Sign	76,77
Bsp. 9	Implementierung der CSP-Funktion CDC_Java_CallMethod_Sign	78
Bsp. 10	Teile der Implementierung des Programms RegOID.exe	82
Bsp. 11	Teile der Implementierung des Programms PubCert.exe	87,88

Bibliographie

[ALVESTRAND]	Private OID-Sammlung von Harald Alvestrand, Norwegen, General Area Director der IETF http://www.alvestrand.no/objectid/top.html
[BONDI]	Richard Bondi "Cryptography for Visual Basic" Wiley, USA Oktober 2000 http://www.wiley.com/
[BUCHMANN]	Johannes Buchmann "Einführung in die Kryptographie", Erste Auflage Springer-Verlag BRD 1999 http://www.springer.de/
[CAPI-LIST]	Die CryptoAPI-Mailing-Liste von Microsoft Hier finden sich einige tausend Emails mit Fragen und Antworten zu den meisten Implementierungsproblemen bzgl. des CryptoAPIs, die nicht in der MSDN-Dokumentation enthalten sind. http://www.lsoft.com/scripts/wl.exe?SL1=CryptoAPI&H=DISCUSS.MICROSOFT.COM bzw. erreichbar von den Microsoft-Internetseiten
[CDC-EC]	Dokumentation des Elliptic-Curve-CDC-Providers vom Fachgebiet Theoretische Informatik (CDC) der TU Darmstadt, 2000 http://www.informatik.tu-darmstadt.de/TI/Forschung/cdcProvider/overview.html
[CDC-OID]	Dokumentation der genutzten OIDs vom Fachgebiet Theoretische Informatik (CDC) der TU Darmstadt, 2000 http://www.informatik.tu-darmstadt.de/TI/Forschung/FlexiPKI/OID.html
[CDC-STD]	Dokumentation des Standard-CDC-Providers vom Fachgebiet Theoretische Informatik (CDC) der TU Darmstadt, 2000 http://www.informatik.tu-darmstadt.de/TI/Forschung/cdcProvider/overview.html
[CERT]	"X.509 Certificates and Certificate Revocation Lists" Sun Microsystems Inc., May 1998 http://java.sun.com/products/jdk/1.2/docs/guide/security/cert3.html
[CSPDK]	Das CSP Development Kit von Microsoft Seit März 2001 per Download von Microsoft erhältlich unter http://msdn.microsoft.com/downloads/default.asp?URL=/code/sample.asp?url=/msdn-files/027/001/585/msdncompositedoc.xml
[JNI-SPEC]	„Java Native Interface Specification“, Release 1.1 (Revised May, 1997) Javasoft / Sun Microsystems Inc. USA 1997 http://javasoft.sun.com
[MSDN]	Microsoft Developer Network und die zugehörige MSDN Library Informationssammlung von Microsoft mit Beschreibung aller Windows-APIs Erhältlich mit den Dokumentations-CDs der Softwareentwicklungswerkzeuge von Microsoft (Visual Studio), in der MSDN-Universal-Subscription oder unter http://msdn.microsoft.com/default.asp Zu beachten ist die regelmäßige Aktualisierung der Informationen. Genutzt wurden die Ausgaben von Juli und Oktober 2000 und Januar 2001 und die Informationen der Ausgabe Visual Studio 6.0a Die Informationen um CryptoAPI finden sich unter: MSDN/PlatformSDK/Security/ in Cryptography und Certificate Services

[NUTSHELL]	David Flanagan „Java in a nutshell“ Deutsche Ausgabe zu Java 1.1, 2.Auflage O'Reilly & Associates, Inc. USA 1998 http://www.oreilly.com/
[OAKS]	Scott Oaks “Java Security”, First Edition O'Reilly & Associates, Inc. USA 1998 http://www.oreilly.com/
[OUTLOOK98]	„Microsoft Outlook 98 Security Whitepaper“ (Filename: 'securityo198.doc') Microsoft, Version 0.1 May 1998 Nicht offiziell erhältlich. URL des Dokumentes ist nicht mehr gültig. Das Dokument ist auf Anfrage vom Autor erhältlich.
[PKCS1]	„RSA Cryptography Standard“ RSA Security Systems / RSA Laboratories http://www.rsasecurity.com/rsalabs/pkcs/pkcs-1/index.html
[PKCS7]	„Cryptographic Message Syntax Standard“ RSA Security Systems / RSA Laboratories http://www.rsasecurity.com/rsalabs/pkcs/pkcs-7/index.html
[PKCS12]	„Personal Information Exchange Syntax Standard“ RSA Security Systems / RSA Laboratories http://www.rsasecurity.com/rsalabs/pkcs/pkcs-12/index.html
[PKCS13]	„Elliptic Curve Cryptography Standard“ RSA Security Systems / RSA Laboratories http://www.rsasecurity.com/rsalabs/pkcs/pkcs-13/index.html
[PKIX]	„Algorithms and Identifiers for the Public Key Infrastructure“ der Public Key Infrastructure Working Group der IETF, März 2001. Analog RFC2459 Dieses Dokument wird zur Zeit (April 2001) erweitert. http://www.imc.org/draft-ietf-pkix-ipki-pkalgs http://www.ietf.org/html.charters/pkix-charter.html
[PK-INTER]	White Paper “Public Key Interoperability” (Windows 2000 Server) http://www.microsoft.com/windows/server/Technical/security/default.asp
[PKI-INTRO]	White Paper “An Introduction to the Windows 2000 Public-Key Infrastructure” (Windows 2000 Server) http://www.microsoft.com/windows/server/Technical/security/default.asp
[PLATFORM- SDK]	Das Platform-Software-Development-Kit von Windows; hier sind zusätzliche Softwareentwicklungshilfen und erweiternde Definitionen für Windows untergebracht. Erhältlich von Microsoft mit der MSDN-Universal-Subscription oder als Einzel-CD-Rom. http://msdn.microsoft.com/developer/sdk
[RFC 1778]	“The String Representation of Standard Attribute Syntaxes” Request for Comment #1778, section 2.15, zum Thema OIDs nach ASN.1 erhältlich gegen Gebühren unter [IETF] alternativ (kostenlos) unter http://www.cis.ohio-state.edu/Services/rfc/index.html
[RFC 2252]	“Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions” Request for Comment #2252, section 4.1, Thema: OIDs nach ASN.1 erhältlich gegen Gebühren unter [IETF] alternativ (kostenlos) unter http://www.cis.ohio-state.edu/Services/rfc/index.html

[RFC 2459]	IETF RFC 2459 analog zu X.509 „Internet X.509 Public Key Infrastructure Certificate and CRL Profile“ erhältlich gegen Gebühren unter [IETF] alternativ (kostenlos) unter http://www.cis.ohio-state.edu/Services/rfc/index.html
[SCHNEIER]	Bruce Schneier “Angewandte Kryptographie” (Applied Cryptography, 1996) Addison-Wesley / Wiley USA 1996 http://www.awl.com
[SCHRAMM]	Dirk Schramm „Entwicklung einer flexiblen Java-basierten S/MIME- Erweiterung für Microsoft Outlook“ Diplomarbeit am Fachgebiet Theoretische Informatik (CDC) der TU Darmstadt, 2001 http://www.informatik.tu- darmstadt.de/TI/Projekte/README.crypto_theses.html
[SEIPEL]	Michael Seipel „Die JCA als Implementierung der Generic Security Services“ Diplomarbeit am Fachgebiet Theoretische Informatik (CDC) der TU Darmstadt, 2000 http://www.informatik.tu- darmstadt.de/TI/Projekte/README.crypto_theses.html
[SIEGEN]	Der Security-Server der Gesamthochschule Siegen; Linksammlung zu allem bzgl. Kryptographie http://www.uni-siegen.de/security/
[SMITH]	Richard E. Smith “Internet-Kryptographie” (Internet Cryptography, 1997) Addison-Wesley-Longman USA 1998 http://www.awl.com
[TAK]	Markus Tak „Public Key Infrastrukturen: ein Java-basiertes Trust- Center“ Diplomarbeit am Fachgebiet Theoretische Informatik (CDC) der TU Darmstadt, 1999 http://www.informatik.tu- darmstadt.de/TI/Projekte/README.crypto_theses.html
[WINCRYPT.H]	Die C-Header-Datei für Visual Studio/C++ für das CryptoAPI Erhältlich in unterschiedlichen Versionen mit der Installation von Visual Studio/C++ 6.0a und dem [PLATFORM-SDK]. Nicht im Internet erhältlich.
[X.200]	ITU-T Recommendation X.200 “Information technology – Open Systems Interconnection – Basic Reference Model: The basic model” erhältlich gegen Gebühren unter [ITU] http://www.itu.int
[X.208]	ITU-T Recommendation X.208 “Specification of Abstract Syntax Notation One(ASN.1)“ Formale Spezifikation der Abstract Syntax Notation One (ASN.1) erhältlich gegen Gebühren unter [ITU] http://www.itu.int
[X.509]	ITU-T Recommendation X.509 analog zu RFC 2459 „Internet X.509 Public Key Infrastructure Certificate and CRL Profile“ erhältlich gegen Gebühren unter [ITU] http://www.itu.int