

# Entwurf und Implementierung einer Revokationskomponente für ein Java-basiertes Trustcenter

Anna Pitaev \*

April 8, 2004

Diplomarbeit  
am Lehrstuhl für Theoretische Informatik  
der Technischen Universität Darmstadt

Prof. Dr. J. Buchmann

Betreuer: Evangelos Karatsiolis

---

\*TU Darmstadt, Fachbereich Informatik, Matrikelnr.1048265

# 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 werden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 8. April 2004

Anna Pitaev

# Vorwort

Der Gegenstand dieser Diplomarbeit war Entwicklung und Implementierung einer Java-basierten Revokation-Komponente (RI), die verschiedene Revokationsmethoden zur Verfügung stellt.

Zur sicheren Kommunikation mit anderen FlexiTrust-Komponenten (z.B. FlexiTrust IS) benutzt RI das auf dem Fachgebiet entwickelte XML-basierte Transferprotokoll. Das Protokoll befindet sich noch in der Bearbeitung, seine erste Version wurde jedoch im Rahmen dieser Diplomarbeit implementiert und kann zukünftig erweitert werden. Der aktuelle Stand der Entwicklung kann aus der Dokumentation und dem Quelltext im CVS-Repository entnommen werden.

Besonderer Dank gilt Evangelos G. Karatsiolis für ständige Unterstützung und fachliche Betreuung meiner Diplomarbeit. Herzlichen Dank an Thilo Planz, dessen Diplomarbeit ([3]) für mich eine grosse Orientierungshilfe war. Vielen Dank an Professor Buchmann, der für mich die Welt der Kryptographie eröffnete.

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Einleitung</b>                           | <b>5</b>  |
| 1.1      | Public-Key Verschlüsselung. . . . .         | 5         |
| 1.2      | Kryptographische Hashfunktionen. . . . .    | 6         |
| 1.3      | Digitale Signaturen. . . . .                | 7         |
| 1.4      | Non-Repudiation. . . . .                    | 9         |
| 1.5      | Komponenten und Benutzer einer PKI. . . . . | 9         |
| 1.5.1    | Persönliche Sicherheitsumgebung. . . . .    | 10        |
| 1.5.2    | Zertifizierungsstellen . . . . .            | 10        |
| 1.6      | PKI-Architekturen. . . . .                  | 14        |
| 1.6.1    | Simple PKI Architectures. . . . .           | 14        |
| 1.6.2    | Enterprise PKI Architectures. . . . .       | 15        |
| 1.6.3    | Hybrid PKI Architectures. . . . .           | 16        |
| 1.7      | Mehrmandantenfähigkeit. . . . .             | 20        |
| 1.8      | PKI-Benutzer. . . . .                       | 20        |
| 1.8.1    | Certificate Holders. . . . .                | 20        |
| 1.8.2    | Relying Party. . . . .                      | 21        |
| <b>2</b> | <b>Digitale Zertifikaten und CRLs.</b>      | <b>22</b> |
| 2.1      | ITU-T Recommendation X.509. . . . .         | 22        |
| 2.2      | X.509 Zertifikaten. . . . .                 | 22        |
| 2.2.1    | Basic Certificate Content. . . . .          | 23        |
| 2.2.2    | Extensions . . . . .                        | 24        |
| 2.3      | X.509 Certificate Revocation Lists. . . . . | 26        |
| 2.3.1    | Basic CRL Contents . . . . .                | 27        |
| 2.3.2    | Extensions. . . . .                         | 28        |
| <b>3</b> | <b>Revokationsmethoden.</b>                 | <b>31</b> |
| 3.1      | Full CRLs. . . . .                          | 31        |
| 3.2      | Indirekte CRLs. . . . .                     | 32        |
| 3.3      | Over-Issued CRLs. . . . .                   | 33        |
| 3.4      | Delta CRLs. . . . .                         | 36        |
| 3.5      | Sliding window delta-CRLs. . . . .          | 39        |
| <b>4</b> | <b>Transferprotokol</b>                     | <b>41</b> |
| 4.1      | Anforderungen an die Software . . . . .     | 42        |
| 4.1.1    | Ist-Zustand . . . . .                       | 42        |
| 4.1.2    | Soll-Zustand . . . . .                      | 43        |
| 4.2      | Implementierung . . . . .                   | 46        |
| 4.2.1    | Übersicht . . . . .                         | 47        |

|          |  |           |
|----------|--|-----------|
| 4.2.2    | P7Creator . . . . .                              | 47        |
| 4.2.3    | P7Importer . . . . .                             | 49        |
| 4.2.4    | Code-Beispiel . . . . .                          | 49        |
| <b>5</b> | <b>Revokation-Komponente</b>                     | <b>51</b> |
| 5.1      | Entwurf . . . . .                                | 51        |
| 5.1.1    | Anforderungen an die Software . . . . .          | 51        |
| 5.1.2    | Revokation-Servlet . . . . .                     | 53        |
| 5.1.3    | RI-Dämon . . . . .                               | 54        |
| 5.1.4    | Operatoren . . . . .                             | 55        |
| 5.2      | Implementierung . . . . .                        | 56        |
| 5.2.1    | Revokation-Servlet . . . . .                     | 56        |
| 5.2.2    | Revokation-Komponente . . . . .                  | 59        |
| 5.2.3    | Benutzte Bibliotheken . . . . .                  | 62        |
| 5.3      | Betriebsanleitung . . . . .                      | 62        |
| 5.3.1    | Installation der Web-Applikation . . . . .       | 63        |
| 5.3.2    | Installation der Revokation-Komponente . . . . . | 64        |
| <b>6</b> | <b>Ausblick</b>                                  | <b>66</b> |
| <b>7</b> | <b>Anhang</b>                                    | <b>67</b> |

# 1 Einleitung

1976 stellten Whitfield Diffie und Martin Hellman die *Public Key Cryptography* vor: die Idee vom öffentlichen und privaten Schlüssel ([13]).

1978 kam Loren Kohnfelder mit der Idee vom *digital certificate*, einem digital signierten Dokument, das den Namen des Public-Key-Inhabers an seinen Öffentlichen Schlüssel bindet ([14]).

Dies war der Ausgangspunkt für die Entwicklung der Public-Key Kryptosysteme, die heutzutage bei immer wachsender Bedeutung der elektronischen Kommunikation im privaten und öffentlichen Bereich der Aufgabe dienen, Daten geheim, authentisch und vertraulich zu speichern und zu übertragen.

Im folgenden Kapitel werden die kryptographischen Techniken beschrieben, die der Entwicklung und Implementierung der im Rahmen dieser Diplomarbeit entstehenden Revokation-Komponente zugrunde liegen.

## 1.1 Public-Key Verschlüsselung.

In diesem Verfahren besitzt jeder Teilnehmer einen *privaten* und einen *öffentlichen* Schlüssel. Der private Schlüssel ist nur dem Inhaber allein bekannt und muss am sicheren Ort aufbewahrt werden, der öffentliche darf dagegen jedem Kommunikationsteilnehmer mitgeteilt werden.

Will Alice eine Nachricht an Bob schicken, so soll sie zuerst sicherstellen, dass sie im Besitz vom Bobs öffentlichen Schlüssel ist. Sie verschlüsselt damit ihre Nachricht und schickt sie an Bob. Bob wird diese mit seinem privatem Schlüssel entschlüsseln.

Die Public-Key Verschlüsselung hat zwei Probleme der symmetrischen Verfahren gelöst:

1. Alice braucht nur einen Schlüssel, um mit allen anderen Teilnehmern kommunizieren zu können;
2. Es müssen keine geheime Schlüssel (z.B. zwischen Alice und Bob) ausgetauscht werden.

Es hat aber ein anderes Problem: Alice muss sicher sein, dass sie den öffentlichen Schlüssel von Bob hat. Sollte es einem Angreifer gelingen, Bobs öffentlichen Schlüssel durch sein eigenen zu ersetzen, so kann er dann Alices Nachrichten an Bob lesen.

Um dies zu verhindern, werden alle öffentlichen Schlüssel in einem Verzeichnis abgelegt, das gegen solchen *Man-In-The-Middle* Angriffe gesichert werden muss.

Einer der bekanntesten Algorithmen, das Public-Key Verfahren implementiert, ist nach seinen Erfindern Ronald Rivest, Fiat Shamir und Leonard Adleman benannte *RSA-Verfahren* ([15]). Dieser sowie der *Rabin*-Verschlüsselungsverfahren beruht auf dem Faktorisierungsproblem und kommt in der asymmetrischer Kryptographie überall zum Einsatz (auch bei digitalen Signaturen).

Da die bekannten Public-Key Verfahren nicht so effizient wie viele symmetrische sind, benutzt man in der Praxis auch die Kombination dieser zwei Klassen von Algorithmen das sogenannte *Hybrid-Verfahren*[1, S. 112]: Soll ein Dokument  $\mathbf{d}$  verschlüsselt werden, so erzeugt man einen *Sitzungsschlüssel*, der nur zur Verschlüsselung von  $\mathbf{d}$  verwendet wird und danach nicht mehr. Nachdem  $\mathbf{d}$  mit diesem Sitzungsschlüssel verschlüsselt ist, wird der Schlüssel selbst mit einem Public-Key Verfahren verschlüsselt, an das verschlüsselte Dokument angehängt und an Empfänger geschickt. Nachdem der Empfänger den einmaligen Sitzungsschlüssel entschlüsselt hat, ist er in der Lage das Dokument  $\mathbf{d}$  zu entschlüsseln.

## 1.2 Kryptographische Hashfunktionen.

Die Kryptographische Hashfunktionen finden ihre Anwendung zum einem bei Digitalen Signaturen, zum anderen zur Überprüfung von Authentizität einer Nachricht. Da asymmetrische Verfahren rechenaufwendig sind, berechnet man die digitale Signatur eines Dokumentes  $\mathbf{d}$ , indem  $\mathbf{d}$  auf seine Prüfsumme reduziert wird. Diese Prüfsumme wird mit Hilfe einer Hashfunktion (auch *message digest* genannt) berechnet, wobei  $\mathbf{d}$  auf einen Funktionswert fester und kürzer Länge abgebildet wird. Zu effizienten Hashfunktionen, die oft in der Praxis zum Einsatz kommen, zählt man **MD4**, **MD5**, **RIPEMD-128**, **SHA-1**, **RIPEMD-160**. Über die wichtigsten Eigenschaften der Hashfunktionen ist bei [3, S. 9] nachzulesen.

Wenn Alice ein Dokument an Bob schicken will, berechnet sie einen *message digest* von diesem Dokument und schickt es auch an Bob mit. Wenn Bob sicherstellen will, dass  $d$  von einer dritten Person nicht verändert wurde, berechnet er auch einen *message digest* vom erhaltenen Dokument und vergleicht es mit dem von Alice. Sollten diese zwei Werte nicht übereinstimmen, so ist ein Angriff zu vermuten. Die einzige Voraussetzung für dieses Szenario ist, Bob muss wissen, welche Hashfunktion Alice verwendet hat und in der Lage sein auch diese zu benutzen. Wenn aber diese Hashfunktion auch dem Angreifer bekannt ist, berechnet dieser den neuen Hashwert des neuen Dokumenten und schickt an Bob mit. Der Bob wird in diesem Fall nicht merken, dass das Dokument geändert wurde, da die Hashwerte übereinstimmen.

Ein sicherer Weg ist es, zu überprüfen, dass die Nachricht tatsächlich

von Alice kommt. [1, S. 173]: Will man die Authentizität einer Nachricht überprüfbar machen, so kann man parametrisierte Hashfunktionen verwenden. Eine parametrisierte Hashfunktion heisst auch *Message Authentication Code*, kurz **MAC**. Alice und Bob tauschen einen geheimen Schlüssel **k**. Mit Hilfe dieses Schlüssels berechnet Alice den Hashwert von **d** und schickt es mit an Bob. Bob nimmt **k** und berechnet wieder den Hashwert von **d**. Sollten zwei Hashwerte übereinstimmen, so kann Bob sich sicher sein, dass **d** tatsächlich von Alice erstellt wurde. Damit das oben beschriebene Verfahren sicher ist, muss der **MAC** fälschungsresistent sein, d.h. mit Hilfe eines Schlüssels **k** können zwei gleiche Hashwerte entstehen, nur wenn zwei Dokumenten, von denen diese MACs berechnet werden, übereinstimmen.

### 1.3 Digitale Signaturen.

Das oben beschriebene **MAC**-verfahren hat ein spezifisches für die symmetrische Verschlüsselung Problem: Es setzt einen Schlüsselaustausch voraus.

Die asymmetrische Kryptographie stellt auch eine Lösung des Authentizitätsproblems vor, die in der Literatur als *Digitale Signaturen* bezeichnet wird.

Will Alice ein elektronisches Dokument signieren, so nimmt sie ihren privaten Schlüssel **d** und den Klartext des Dokumentes **m** und berechnet daraus eine Funktion **s(m,n)**, die Bob als Alice's Digitale Unterschrift betrachten darf.

Um diese Signatur zu verifizieren, besorgt sich Bob den Alice's öffentlichen Schlüssel **n** und berechnet die Funktion **m'(n,s)**. Dann vergleicht er zwei Werte **m**, das elektronische Dokument von Alice, und von ihm berechneten **m'**. Sollten sie übereinstimmen, so kann Bob sicher sein, dass die Unterschrift von Alice ist.

Als digitales Dokument kann z.B einen Kaufvertrag im Internet, eine Banktransaktion oder eine verbindliche E-mail in Frage kommen. Nachdem Alice dieses digital signiert hat, kann jeder, der im Besitz vom Alice's öffentlichen Schlüssel ist, ihre Signatur zu verifizieren. Nach der erfolgreicher Verifikation kann Alice nicht mehr bestreiten, dieses Dokument signiert zu haben, so kann die Digitale Signatur [1, S. 175]: als Beweis in Gerichtsverfahren herangezogen werden.

Dieses Verfahren hat aber einen schwachen Punkt: Sollte es einem Angreifer gelingen, Bob seinen eigenen Schlüssel als den von Alice zu unter-schieben, so wird Bob alle seine Signaturen als die von Alice verifizieren.

Um diesen Angriff zu vermeiden, werden in der Praxis alle öffentlichen Schlüssel in den sogenannten *Trustcenters* gehalten, die nur einen kontrollierten

Zugriff von ausgewählten Personen erlauben. Die Komponenten und Funktionsweise eines Trustcenters werden in dieser Diplomarbeit später beschrieben.

Man kann sich das Szenario vorstellen, dass ein digitales Dokument sowohl von Alice als auch von Bob unterzeichnet werden muss. In diesem Fall wird es von *mehrfachen Signaturen* gesprochen, bei deren Realisierung folgendes Problem auftritt.

Wie es aus dem Verifikationsprozess zu folgern ist, hängt der Wert der digitalen Signatur von dem zu signierendem Dokument ab. Sollte dieses nach dem Signieren geändert werden, so wird die Verifikation fehlschlagen. Im Falle der XML-Signaturen, die im Rahmen dieser Diplomarbeit oft zum Einsatz kommen, wird dieses Problem auf folgende Weise gelöst:

Beim Signieren eines XML-Dokumentes, kann die Signatur über ein externes Kontext (ein XML-Element) *detached signature*, über ein internes Kontext (das Signature-Element selbst) *enveloping signature* oder über ein Kontext, das das Signatur-Element enthält *enveloped signature*, berechnet werden. Sollten ein paar Elemente des XML-Dokumentes geändert werden, nachdem das Signieren vorgenommen wurde, so werden diese mit Hilfe einer entsprechenden **Transformation** aus dem Signaturkontext rausgenommen.

Sollte ein XML-Dokument (s. Figure 1) zwei *enveloped signatures* enthalten, so wird die erste Signatur aus ihrem Kontext automatisch rausgenommen. Das zweite Signaturelement muss aber von der Berechnung der ersten Signatur explizit rausgenommen werden. Die Struktur und Implementierung-

```
<Request>
  <profile1>
    <!-- Kontext1,Kontext2 -->
    <element1>value1</element1>
    <element2>value2</element2>
    <!-- Kontext1 -->
    <!-- enveloped signature1 -->
    <ds:Signature>..</ds:Signature>
    <!-- Kontext2 -->
    <!-- enveloped signature2, -->
    <!-- muss aus Kontext1 explizit rausgenommen werden -->
    <ds:Signature>..</ds:Signature>
  </profile1>
</Request>
```

Figure 1: XML-Signaturen

stechniken der XML-Signaturen werden im Kapitel 4 dieser Diplomarbeit näher eingegangen.

## 1.4 Non-Repudiation.

Auch nach der erfolgreichen Verifikation digitaler Signatur, kann die signierende Person ihre Unterschrift bestreiten.

Man denke dabei an folgendes Szenario: Alice hat einen Kaufvertrag im Internet digital unterzeichnet und an Bob verschickt. Danach vernichtet sie ihren privaten Schlüssel und behauptet, dass der Kaufvertrag von einer anderen Person signiert wurde, die den privaten Schlüssel von Alice gestohlen hat.

Um ein solches Verhalten zu vermeiden, erweitert man eine digitale Signatur zu *Non-Repudiation*, indem die Zeit des Signierens in das Dokument eingefügt wird, bevor man den Hashwert dieses Dokumentes ermittelt. Sollte in Zukunft eine Änderung dieses Zeitpunktes stattfinden, so wird sie unvermeidlich zu Verifikationsfehlschlag führen.

Alice verpflichtet sich in Non-Repudiation, dass sie in Zukunft (z.B. nach einem Monat seit dem Signieren des Dokumentes) ihre Unterschrift (aus irgendeinem Grund) nicht bestreiten kann.

Die Sicherheit von Non-Repudiation hängt stark von Sicherheit des Zeitmechanismus ab. Sollte es möglich sein, dem zu signierenden Dokument einen späteren Zeitpunkt zuzuweisen, so kann die Non-Repudiation in dieser Zeitlücke als einen Gerichtsnachweis nicht verwendet werden.

Einen sicheren Zeitmechanismus zu implementieren, ist eine der schwierigsten Aufgaben *der Public-Key-Infrastrukturen*, die den Einsatz der Methoden asymmetrischer Kryptographie in der Praxis ermöglichen.

## 1.5 Komponenten und Benutzer einer PKI.

Die Sicherheit der asymmetrischen Verschlüsselung und Signierens hängt stark von der Sicherheit des Key-paares ab. Während der private Schlüssel unzugänglich für jede andere Person (abgesehen vom Schlüsselinhaber) zu speichern ist, muss der öffentliche Schlüssel vor Fälschung und Missbrauch geschützt werden .

[1, S. 205] Die Verteilung und Speicherung der öffentlichen und privaten Schlüssel geschieht in einer sogenannten *Public-Key-Infrastruktur (PKI)*.

### 1.5.1 Persönliche Sicherheitsumgebung.

Da es zum Signieren und Entschlüsselung eines Dokumentes der private Schlüssel verwendet wird, sollten diese Prozesse in derselben Umgebung stattfinden, wo der private Schlüssel gespeichert ist.

Manchmal erfüllt die Persönliche Sicherheitsumgebung (auch *Personal Security Environment PSE* genannt) die Aufgabe der Schlüsselerzeugung. Dies hat den Vorteil, dass der private Schlüssel in dem Fall selbst dem Inhaber nicht bekannt ist. Von anderer Seite, sollte die PSE einen schwachen Mechanismus für die Schlüsselerzeugung implementieren (z.B. einen schwachen Zufallszahlgenerator), so ist es besser, diese Aufgabe an eine entsprechende Instanz zu delegieren, da ein sicheres Schlüsselpaar in der asymmetrischen Kryptographie eine zentrale Rolle spielt.

Als eine Persönliche Sicherheitsumgebung kann entweder ein durch Passwort geschützter Speicherbereich auf der Festplatte oder auch eine Chipkarte in Frage kommen. In dem ersten Fall ist man sehr auf die Sicherheit des Betriebssystems angewiesen und, obwohl es auf diesem Gebiet immer neue Ideen eingesetzt werden, lässt sich das Sicherheitsgrad dieser Lösung noch weiter erhöhen.

Da Berechnungen auf Chipkarten sehr langsam sind, lassen sie sich viel effizienter bei den *Hybrid-Verfahren* (s. Kapitel 1.1) einsetzen. Dabei wird nur der *Sitzungsschlüssel* mit der Karte entschlüsselt, während die Entschlüsselung des gesamten Dokumentes im Rechner erfolgt.

Sollte Alice ihre PSE auf einer Chipkarte platzieren, so kann sie sich nicht sicher sein, dass sie das gewünschte Dokument unterzeichnet, da sie dieses nicht sehen kann. Es besteht dann die Möglichkeit, dieses zu vertauschen und von Alice signieren zu lassen. Dieses Problem ist in der Literatur als *Darstellungsproblem* bekannt und von seiner Lösung [1, S. 207]: hängt es ab, ob digitale Signaturen in wirklich sicherheitskritischen Situationen eingesetzt werden können.

### 1.5.2 Zertifizierungsstellen

Die zentrale Aufgabe einer Zertifizierungsstelle *Certification Authority CA* ist das Erstellen eines digitalen Kontextes, das die Identität einer Person an ihren öffentlichen Schlüssel bindet. Dieser Kontext wird im Anschluss mit dem privaten Schlüssel der Zertifizierungsinstanz signiert, hiermit wird ein *digitales Zertifikat* erzeugt.

Nachdem Alice ein Zertifikat erhalten hat, wird es veröffentlicht und kann vom jeden anderen Teilnehmer erhalten werden. [1, S. 207]: Die CA bürgt mit ihrer Signatur für die Korrektheit und Gültigkeit der öffentlichen Schlüssel

der Teilnehmer.

Gehört Bob zu diesem PKI-System, so ist er im Besitz des öffentlichen Schlüssels dieser CA und kann ihre Signatur verifizieren.

Das Erstellen (*Issuing*), Speicherung und Verwaltung der Zertifikaten sind sehr umfangreiche und komplizierte Prozesse, die aber ein hohes Grad an Sicherheit verlangen. Um dies zu ermöglichen, wird oft in der Praxis eine Zertifizierungsstelle in mehrere Komponenten zerlegt, deren Aufgaben und Funktionsweise folgend im einzelnen beschrieben werden.

**Registrierung:** Will Alice ein Zertifikat beantragen, so muss sie zu der Registrierungsdienst (*Registration Authority RA*) dieser PKI gehen, die gegen Vorlage ihres Ausweises ihre Identität überprüft.

Nach der erfolgreichen Verifikation Alices persönlicher Daten wird sie in dieser PKI registriert, indem ihr ein Benutzername (oder auch ein Pseudonym) zugeteilt wird (über dem Alices realen Namen verfügt nur die RA, keine der Teilnehmer).

Anschließend reicht die RA den Antrag auf das Zertifikat an die CA weiter. Die Kommunikationswege zwischen einzelnen Komponenten einer PKI werden im weiteren beschrieben.

Die Generierung des Schlüsselpaares für den Antragstellenden kann auch als Aufgabe einer RA in Betracht kommen. Der private Schlüssel soll danach in seine PSE installiert werden, während der öffentliche an die CA mit dem Zertifikatsantrag mitgeschickt wird.

Zu sicherer Übermittlung des privaten Schlüssels zu seinem Besitzer kommen die Chipkarten und kennwortgeschützte Dateien zum Einsatz. [3, S. 12]: Es ist empfehlenswert, den Empfang des Schlüssels durch seinen Besitzer zu überprüfen, bevor man das Zertifikat veröffentlicht (ohne Zertifikat ist der Schlüssel wertlos).

Sollte ein Zertifikat aus irgendeinem Grund (z.B. Verlust des privaten Schlüssels) als ungültig erklärt (*revoziert*) werden, so überprüft die RA die Revokationsdaten (z.B. Name des Zertifikatsinhabers, die Zertifikatsnummer usw.), erstellt den Antrag auf den Rückruf dieses Zertifikaten und übergibt ihn an die CA, die den entsprechenden Revokationsprozess ausführt.

Im Rahmen dieser Diplomarbeit wird eine neue PKI-Komponente entwickelt, die sowohl RA als auch CA um die mit der Revokation verbundenen Aufgaben entlasten soll.

**Zertifizierung:** Die Zertifizierungsdienst (*Certification Authority CA*) ist die zentrale Komponente einer PKI, die für *Certificate Issuing und Revocation* zuständig ist.

Im Gegensatz zu den anderen PKI-Instanzen verfügt sie über ein Schlüssel-paar und kann damit von ihr erstellte Produkte (Zertifikate, CRLs) signieren. Sowohl alle Systemteilnehmer als auch andere Instanzen kennen ihren öffentlichen Schlüssel und können deswegen ihre Signatur verifizieren.

Der Lebenszyklus eines Zertifikaten kann sowohl rechtzeitig (nach dem Gültigkeitsablauf) als auch frühzeitig (durch eine Revokation) enden.

In dem ersten Fall, gehört es zu den Aufgaben einer CA, die Rezertifizierung vorzunehmen, indem ein neues Zertifikat erzeugt und veröffentlicht wird.

Im Falle einer Revokation, erstellt die CA eine Liste mit den zu revozierenden Zertifikaten (*Certificate Revocation List CRL* genannt), unterzeichnet diese und gibt sie an eine weitere Komponente, die sich um die Veröffentlichung von Zertifikaten und CRLs kümmert.

**Verzeichnisdienst:** Die von CA erstellten Produkte werden in einem Verzeichnis (*Directory*) veröffentlicht. Dieser Vorgang wird von einer neuen Komponente (*Certificate Management Authority CMA* genannt) verwaltet. Nach erfolgreicher Verifikation der Signaturen, speichert die CMA Zertifikate und CRLs in Directory. Gleich im Anschluss können alle Teilnehmer auf die Directory zugreifen, um aktuelle Statusinformation über sie interessierten Zertifikaten zu holen.

Sollte Alice öfter das Zertifikat von Bob (z.B. zu Verschlüsselung ihrer Nachrichten an Bob) verwenden, so kann sie es von der Directory runterladen. Dabei soll sie sich in bestimmte reguläre Zeitabschnitte aktuelle Statusinformationen vom Verzeichnis holen.

Das Verzeichnis hat einen Engpass bei der Veröffentlichung einer neuen Revokationsliste CRL. Gleich im Anschluss wird die Directory von vielen Clients gleichzeitig angesprochen und dann wieder über den längeren Zeitraum nicht mehr benutzt. Um dieses Problem zu lösen, erstellt man eine neue CRL in gleichmässigen Zeitabschnitten, wobei die alte CRL noch nicht abgelaufen ist. Diese Massnahme führt zu Verteilung der Clientszugriffe innerhalb der Zeitachse.

Eine andere Lösung besteht darin, mehrere Kopien von dem Verzeichnis zu erstellen und allen Teilnehmer zwischen diesen Kopien zu verteilen.

Um Sicherheit vom Verzeichnis zu gewährleisten, werden die Schreibrechte nur der CMA zugeteilt, für alle Systemteilnehmer sind ausschliesslich Lesezugriffe auf Directory erlaubt.

**Archivierung:** Sollte ein digitales Dokument über den längeren Zeitraum verifizierbar bleiben, so muss der entsprechende öffentliche Signaturschlüssel

für diese Zeit archiviert werden.

Das Gleiche gilt auch für den privaten Schlüssel im Falle der Archivierung eines verschlüsselten Dokumentes.

Mit der Archivierung werden gleichzeitig mehrere PKI-Komponenten beauftragt: Während die privaten Schlüssel in PSE archiviert werden, übernimmt Repository die Archivierung von öffentlichen Schlüsseln.

Man beachte dabei, dass nicht jeder Schlüssel der Archivierung abliegt: Sowohl privaten Signaturschlüssel als auch öffentliche Verschlüsselungsschlüssel werden für eine bestimmte Zeit (zur Signaturberechnung bzw. Verschlüsselung) verwendet und können sofort danach gelöscht werden.

**Aufbau eines Trustcenters:** Die Architektur und Kommunikation der einzelnen Komponenten eines Trustcenters wird an dem Beispiel der an dem Lehrstuhl von Professor Johannes Buchmann an der TU-Darmstadt entwickelten Trustcenter-Software *FlexiTrust* erläutert.

Dieses Software-Produkt unterscheidet sich von den meisten kommerziellen Lösungen durch seine flexible [3, S. 16] Sicherheitsarchitektur, in der sich einzelne Komponenten schnell und mit minimaler Beeinträchtigung des laufenden Betriebs austauschen lassen.

Die Sicherheit einer PKI hängt stark von der Sicherheit der kryptographischen Basistechnik, die sie benutzt, ab. Sollte diese Technik unsicher werden, so muss sie durch eine andere ersetzt werden.

Unter der Flexibilität einer PKI wird in der ersten Linie [4, S. 2] die Austauschbarkeit der kryptographischen Basistechniken verstanden. Das FlexiTrust-Paket besitzt diese Eigenschaft und kann neben dem RSA-Verfahren auch mit anderen kryptographischen Algorithmen (z.B. mit dem Elliptic Curve Digital Signature Algorithmus ECDSA) arbeiten.

Die Software besteht aus folgenden Komponenten:

- Registrierung (**FlexiTrust RA**),
- Zertifizierung (**FlexiTrust CA**),
- Verzeichnisdienst (**FlexiTrust IS**).

Die Kommunikation zwischen diesen drei Komponenten erfolgt mit Hilfe einer möglicherweise digitalsignierten *p7-Datei*, die das *PKCS7-Format* unterstützt und für den Transport aller Daten von RA zu CA und von CA zu IS zuständig ist.

Da die CA meistens als eine Offline-Komponente implementiert ist, findet dieses Datentransfer einmal pro einen bestimmten Period (z.B einmal am Tag) statt, dabei wird die *p7-Datei* auf einer Diskette zwischengespeichert.

Der Offline-Status der CA verzögert alle mit der Zertifizierung verbundenen Prozesse, u.a. Revokation.

Zur Lösung dieses Problems wird im Rahmen dieser Diplomarbeit eine neue Online-Instanz (**FlexiTrust RI**) implementiert, die die **FlexiTrust CA** um die Revokationsaufgaben entlasten soll.

## 1.6 PKI-Architekturen.

Eine PKI-Architektur beschreibt die Art der Beziehungen der CAs innerhalb dieser PKI. Da sich die Unternehmen in ihrer Grösse und Struktur unterscheiden, hat man verschiedene PKI-Modelle entwickelt, wobei jedes seine eigenen Stärken und Schwächen besitzt. Bei der Wahl einer Architektur sind folgende Kriterien zu beachten:

1. Anzahl der CAs in der PKI,
2. Art der Beziehungen zwischen einzelnen CAs,
3. Erweiterbarkeit der PKI (wie einfach kann man diese PKI um eine neue CA erweitern),
4. Unternehmensgrösse sowie
5. Anzahl der Unternehmen in einer PKI.

Es werden im weiteren die einzelnen PKI-Modelle behandelt.

### 1.6.1 Simple PKI Architectures.

Diese Lösung ist besonders attraktiv für die kleineren PKIs, die eine oder wenige CAs beinhalten.

**Single CA:** Alle Teilnehmer vertrauen nur einer CA.

Diese Architektur ist sehr einfach, hat aber das folgende Problem: Sollte es zu CA Ausfall (z.B durch CA Key Compromise) kommen, gibt es keine Möglichkeit eine andere CA zu benutzen.

**Basic Trust Lists:** Jeder Teilnehmer kann mehreren CAs vertrauen, es bestehen aber keine Beziehungen zwischen einzelnen CAs.

In diesem Fall führt jeder Teilnehmer eine Liste mit Zertifikaten von CAs, denen er vertrauen will. Gehört sein Kommunikationspartner zu einer fremden CA, so versucht er mit Hilfe dieser Liste die PartnerCA zu verifizieren. Erst im Anschluss kann das Zertifikat vom Partner verifiziert werden.

Diese Lösung ist auch sehr einfach und benutzt keine Zertifikatsketten. Da Basic Trust List zu jeder Zeit nur aktuelle Informationen beinhalten darf, kann ihre Verwaltung bei den grossen Anzahl von CAs ziemlich kompliziert werden.

### 1.6.2 Enterprise PKI Architectures.

Jeder Teilnehmer kann mehreren CAs vertrauen. Es bestehen Beziehungen zwischen einzelnen CAs innerhalb des Unternehmens.

**Hierarchical PKI:** Alle Teilnehmer vertrauen einer *root-CA*. Jede CA (ausser *root-CA*) hat einen Vorgänger. Jede Beziehung zwischen zwei CAs ist durch ein Zertifikat representiert, wobei der CA-Vorgänger als *issuer* und CA-Nachfolger als *subject* dieses Zertifikaten dastehen.

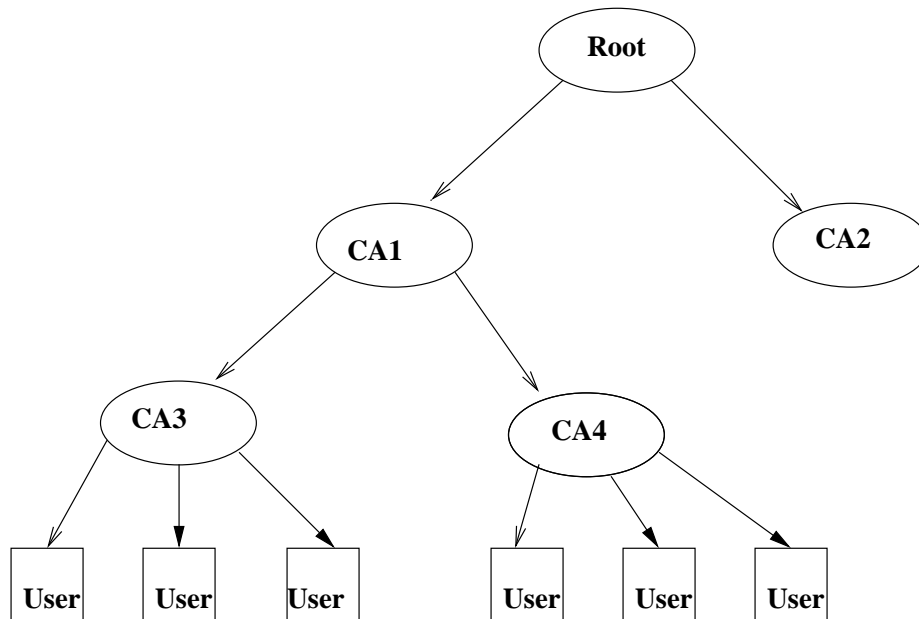


Figure 2: Hierarchical PKI

Da jede CA den einzigen Vorgänger besitzt (s. Figure 2), kann die Zertifikatskette sehr einfach berechnet werden. Die längste Kette ist gleich der Tiefe des Baums: das CA Zertifikat für jeden Nachfolger + Zertifikat des Teilnehmers.

Der Ausfall einzelner CA kann einfach behandelt werden, solange es keine *root-CA* ist.

Der Ausfall einer *root-CA* equivalent dem CA-Ausfall bei der Single CA Architektur.

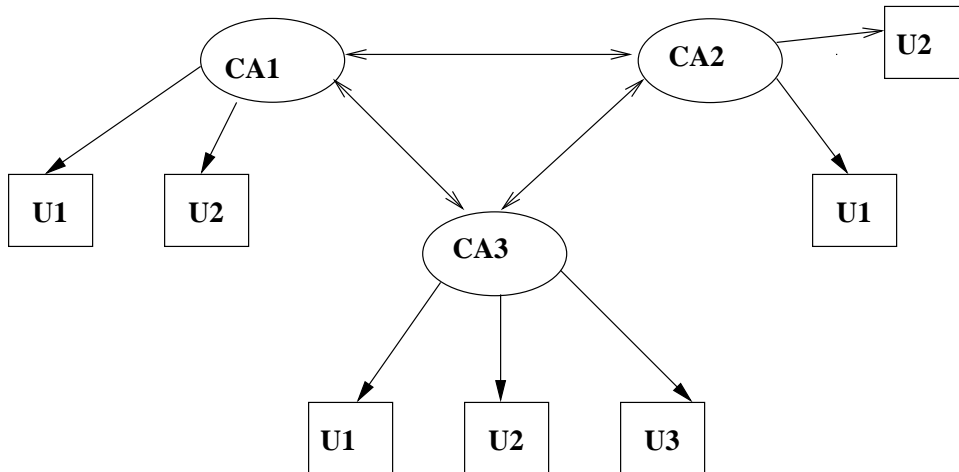


Figure 3: Mesh PKI

**Mesh PKI:** Jeder Teilnehmer vertraut nur einer CA, die sein Zertifikat erstellt hat. Verschiedene CAs können gegenseitig Zertifikate erstellen, wodurch bidirektionale Beziehungen entstehen (s. Figure 3).

Das Einfügen einer neuen CA ist einfach. Die Zertifikatsketten können aber aufgrund der bidirektionalen Beziehungen sehr lang und schwer berechenbar werden.

Der Vorteil dieser Lösung besteht darin, dass der Ausfall jedes CA-Knoten einfach behandelt werden kann.

### 1.6.3 Hybrid PKI Architectures.

Hybrid PKI Architecturen benutzen die oben beschriebenen CA trust list, hierarchical PKI und Mesh PKI, um die Kommunikation zwischen Teilnehmern verschiedener Unternehmen zu ermöglichen.

**Extended Trust Lists Architecture:** Bei dieser Lösung wird eine Liste von *trust points* geführt. Unter einem trust point kann eine single CA, eine Mesh oder eine hierarchical PKI verstanden werden. Wollen zwei Teilnehmer verschiedener Unternehmen mit einander kommunizieren, so werden in ihren Extended Trust Lists die Zertifikate der Partners-PKI eingetragen.

Diese Architektur ist einfach und erlaubt schnell einen neuen trust point hinzuzufügen, hat aber folgende Schwächen:

1. Die trust points Listen dürfen nur aktuelle Informationen(Zertifikaten) beinhalten, dies macht die Verwaltung von grossen Listen ziemlich kompliziert,

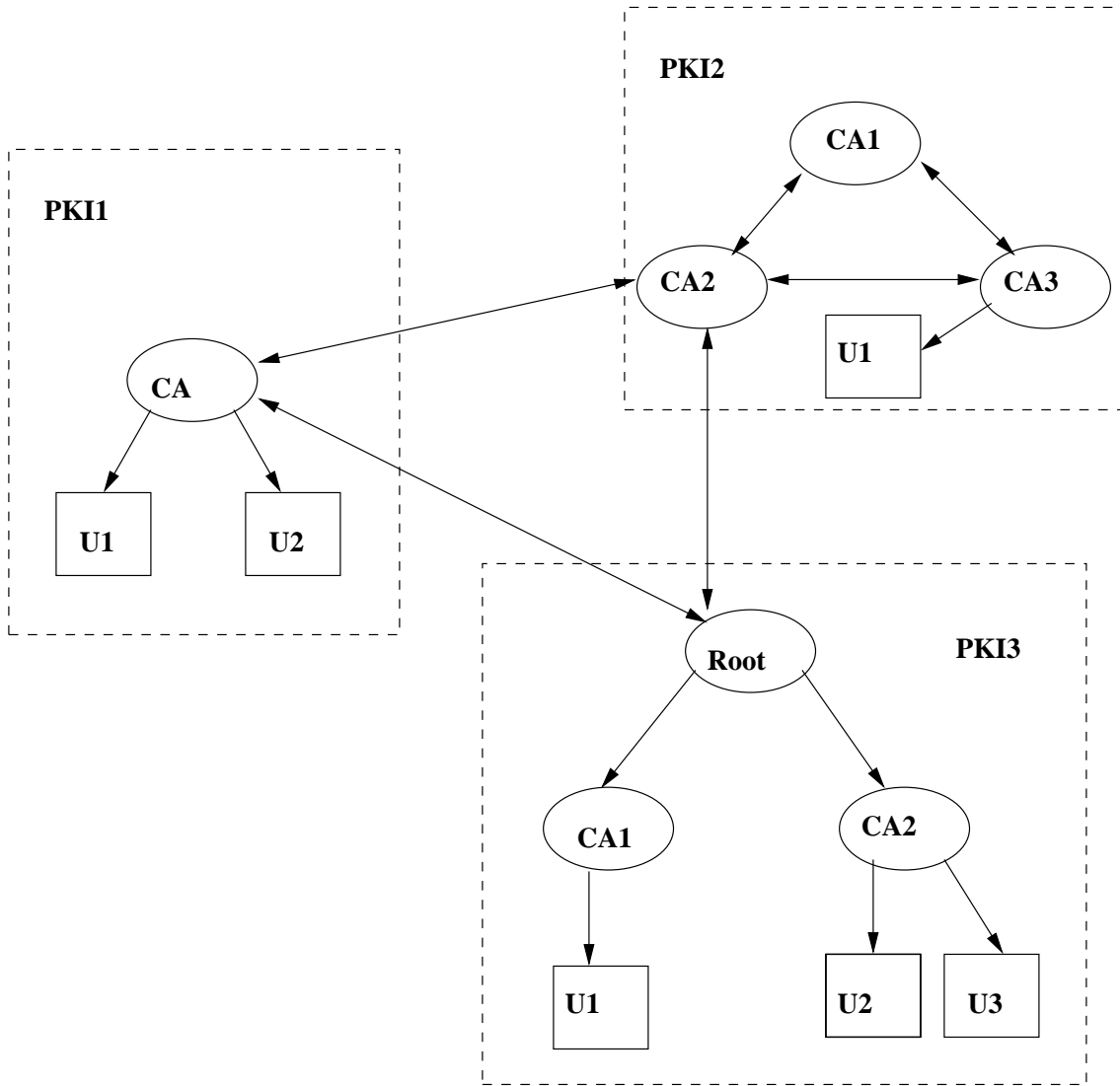


Figure 4: Cross-certified Enterprise PKI

2. Es sind keine Methoden zur Ausfallbehandlung eines trust points vorhanden,
3. Die Zertifikatskette kann nur schwierig berechnet werden : Da der trust point nur das Unternehmen des Kommunikationspartners identifiziert, ist der Weg vom trust point bis zu Teilnehmerszertifikaten innerhalb der Unternehmen-PKI unbekannt und muss rückwärts vom Benutzerzertifikaten bis zu seiner CA berechnet werden.

**Cross-Certified Enterprise PKIs:** Bei dieser Lösung bestehen die direkten Beziehungen zwischen den einzelnen CA verschiedener Unternehmen (s. Figure 4). Die Kommunikationspartner brauchen keine Listen zu verwalten, was die Kommunikation erheblich vereinfacht.

Bevor eine Beziehung zu der Partner-CA entstehen kann, müssen ihre policy überprüft werden. Diese Aufgabe übernimmt der CA-Administrator, der im Falle einer schwächeren Policy diese Beziehung abstreitet.

Die Berechnung der Zertifikatsketten hängt stark von den PKI-Architekturen innerhalb der Unternehmen und kann deswegen sehr kompliziert werden.

Diese Lösung ist sehr attraktiv, solange die Anzahl der Unternehmen-PKIs klein bleibt. Bei wachsender Teilnehmeranzahl wird die Architektur ziemlich kompliziert , so sind bei acht Unternehmen 28 Beziehungen, d.h. 56 Zertifikaten zu erstellen.

**Bridge CA Architecture:** Bei dieser Architektur wird eine neue CA namens *Bridge CA* eingefügt, die Beziehungen jeweils nur zu einem Knoten jeder Unternehmen-PKI erstellt. Dieser Knoten (als *principal CA* bezeichnet) ist eine root CA bei der hierarchical oder eine beliebige CA bei der mesh PKI.(s. Figure 5)

Obwohl die Bridge CA Architektur die Berechnung der Zertifikatsketten nicht vereinfacht, ist sie aufgrund folgender Verbesserungen bei der grossen Unternehmenszahl zu bevorzugen:

1. Bei  $n$  Unternehmen sind  $n$  Beziehungen, d.h.  $2n$  Zertifikaten erforderlich (d.h. 16 anstatt 28),
2. Der Ausfall einer principal CA wird sofort von der Bridge CA notiert und führt nicht zum Ausfall des ganzen Systems,
3. Der Bridge CA Ausfall wird jedenfalls von allen principal CAs leicht festgestellt.

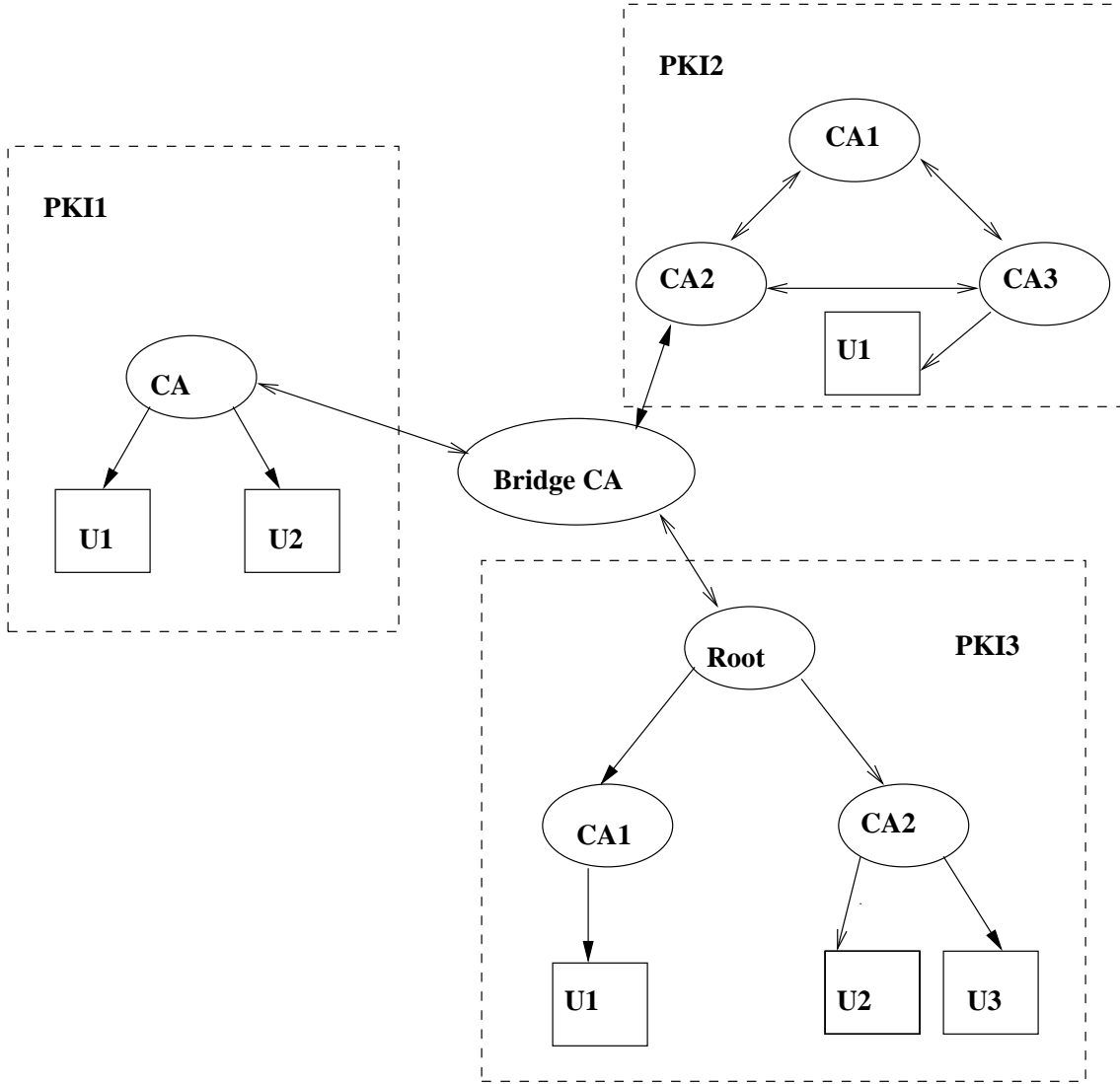


Figure 5: Bridge CA PKI

## 1.7 Mehrmandantenfähigkeit.

Die Mehrmandantenfähigen Systeme kommen zum Einsatz, wenn eine PKI von mehreren Unternehmen (*Mandanten*) geteilt wird.

Es entsteht infolgedessen nur ein sicheres TrustCenter mit starken Sicherheitsmechanismen, wobei die Kommunikation zwischen einzelnen Mandanten innerhalb dieser PKI an Sicherheit gewinnt.

Diese Lösung wird oft auch aus wirtschaftlichen Gründen bevorzugt, da die Kosten der PKI-Planung und Installation zwischen einzelnen Mandanten geteilt werden können.

## 1.8 PKI-Benutzer.

Es werden zwei Klassen der PKI-Benutzer unterschieden:

1. *Certificate Holders*, die im Subject-Feld des Zertifikaten eingetragen sind und die entsprechenden private Schlüssel besitzen,
2. *Relying Parties*, die den öffentlichen Schlüssel ihrer Kommunikationspartner (z.B. zur Signaturverifikation oder Datenverschlüsselung) benutzen.

Jeder PKI-Teilnehmer kann gleichzeitig bei den verschiedenen kryptographischen Prozessen sowohl als Certificate Holder als auch als Relying Party auftreten. Da jede CA ihre eigenen Zertifikaten besitzt, kann sie auch als ein Certificate Holder betrachtet werden. Dabei stehen andere PKI-Komponenten, die ihre Zertifikaten verifizieren (z.B. RA), als Relying Parties da.

### 1.8.1 Certificate Holders.

Zum Erstellen eines Zertifikaten sind von einem Certificate Holder folgende Schritte vorzunehmen:

1. Es muss eine CA ausgewählt werden,
2. Bei dieser CA wird direkt oder mittels RA ein Zertifikat beantragt,
3. Sowohl das neu erstellte Zertifikat als auch der entsprechende private Schlüssel werden streng geschützt abgespeichert.

Die Zertifikatsinhaber können Repository einmalig zum Abruf ihrer Zertifikaten zugreifen, brauchen aber keine regelmäßigen Verbindungen in den festen Zeitabschnitten zu Repository zu erstellen.

### 1.8.2 Relying Party.

Zur Verifikation eines Zertifikaten hat der Relying Party folgende Massnahmen zu ergreifen:

1. Den Zertifikatsersteller (eine CA) identifizieren,
2. Das Zertifikat und eine aktuelle CRL vom Repository abrufen,
3. Die Signaturen vom Zertifikaten und der CRL verifizieren,
4. Zertifikatskette berechnen und verifizieren.

Im Gegensatz zu Certificate Holders, haben Relying Parties das Repository regelmässig für neue Zertifikate sowie aktuelle CRLs abzufragen. Im Rahmen dieser Diplomarbeit werden Algorithmen zur Verteilung der dadurch entstehenden Repository-Belastung beschrieben und implementiert.

## 2 Digitale Zertifikaten und CRLs.

Ein Zertifikat(s. Kapitel 1), ist ein digitales vom seinem Ersteller (*issuer*) signiertes Dokument, das den Namen des Zertifikatsinhabers mit seinem öffentlichen Schlüssel verbindet. Sollte ein Zertifikat aus irgendeinem Grund verworfen werden, so wird es in eine Revokationsliste (*Certificate Revocation List CRL*) eingetragen und somit als ungültig erklärt. Eine Revokationsliste stellt ebenfalls ein digitalsigniertes Dokument dar.

Zur schnellen automatischen Bearbeitung benötigen diese Dokumente ein einheitliches digitales Format. Von ITU-T entwickelt, ist *X.509 public key certificate* zu einem weit verbreiteten Zertifikatsformat geworden ([16], das auch im Rahmen des FlexiTrust-Projektes zum Einsatz kommt.

### 2.1 ITU-T Recommendation X.509.

Dieses Dokument wurde am 9. August 1997 verabschiedet und beschreibt die Kommunikation zwischen End-user und Repository. Dabei werden zwei Formen der Authentifikation unterschieden:

1. einfache durch Passwordeingabe *weak authentication* und
2. *strong authentication* benutzt die kryptographischen Techniken, die im weiteren beschrieben werden.

### 2.2 X.509 Zertifikaten.

Ein X.509 Zertifikat besteht aus (s. Figure 6):

**tbsCertificate**, das wichtige Zertifikatsinformationen beinhaltet und als Basis zur Signaturberechnung dient,

**signatureAlgorithm**, das *ID algorithm identifier* der vom Zertifikatsersteller benutzten kryptographischen Technik beinhaltet,

**signatureValue**, der mit signatureAlgorithm über tbsCertificate berechneten Signatur.

Dabei sind alle obligatorischen und optionalen Informationen eines Zertifikaten im tbsCertificate-Teil zu finden.

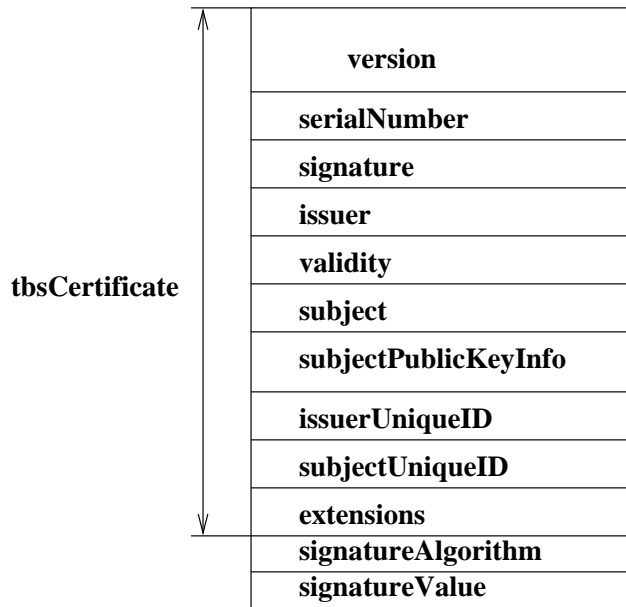


Figure 6: X.509 Zertifikat

### 2.2.1 Basic Certificate Content.

Zu dem Basic Certificate Content eines X.509 Zertifikaten gehören:

**version.** Der Wert 1 dieses optionalen Feldes entspricht einem Basic Certificate Content ohne extensions und UniqueIDs. Bei den Version2-Zertifikaten sind keine extensions erlaubt. Die modernsten Zertifikaten enthalten diese beiden Felder und somit die Version3 unterstützen ([16]).

**serialNumber.** Die Nummer des Zertifikaten wird von seinem Ersteller vergeben und zusammen mit dem Namen des Erstellers ein Zertifikat eindeutig identifiziert. Jeder Zertifizierungstelle muss daher darauf achten, dass die Nummer der von ihr erstellten Zertifikaten sequentiell wachsend (d.h. einmalig) sind.

**signature.** Das signature-Feld ist eine obligatorische Kopie des signature-Algorithm-Feldes bei X.509 Public Key Certificate.

**issuer.** Der Name der Zertifizierungsinstanz ist jedenfalls ein obligatorisches Attribut, das X.500 distinguished name Format unterstützen muss (s. [2, S. 72]).

**validity.** Das validy-Attribut beschreibt die Lebensdauer (=Gültigkeitsperiode) dieses Zertifikaten. Die Felder notBefor und notAfter definieren

dabei den Zeitraum, während dessen ein Zertifikat von Relying Party benutzt werden darf. Die Daten und Zeit Angabe soll das UTCTime bzw. das GeneralizedTime Zeitformate unterstützen.

**subject.** Das subject-Feld ist ein X.500 distinguished name eines Zertifikatsinhabers. Als CertificateHolder kann eine CA, eine Hardware, ein menschlicher Benutzer oder jede andere Instanz, die das Schlüsselpaar besitzt, bezeichnet werden.

**subjectPublicKeyInfo.** Dieses Attribut enthält den öffentlichen Schlüssel von subject und AlgorithmID, mit dem das Schlüsselpaar erzeugt wurde.

**issuerUniqueID** und **subjectUniqueID.** Diese beiden Attributen für die Wiederverwendbarkeit von subject-und issuerNamen werden in der Praxis nur selten benutzt.

**extensions.** Extensions sind nur in der Version3 X.509 Public Key Certificate vorhanden und kommen in der Praxis oft zum Einsatz.

### 2.2.2 Extensions

Jedes Extension-Attribut eines X.509 Zertifikaten besteht aus einem *identifier*-Feld, einem *criticality flag*-Feld und einem *value*-Feld. Man unterscheidet fünf Typen zusätzlicher Information, die einem Zertifikaten mittels Extension-Attributen zugefügt werden kann:

**Subject type.** Legt fest, ob Certificate Holder eine *CA* oder ein *end-entity* ist.

**Names und identity information.** Ermöglicht verschiedene Namesformate für Certificate Holder.

**Key attributes.** Gibt an, für welche kryptographische Operationen (Verschlüsselung, Signieren usw.) das Schlüsselpaar erstellt wurde.

**Policy information.** Unterstützt das Erstellen der Zertifikaten unter verschiedenen Policy-Klassen.

**Additional information.** Beschreibt, wie und wo ein gültiges Zertifikat bzw. eine neue erstellte CRL erhalten werden kann.

Im Rahmen dieser Diplomarbeit werden verschiedene Verfahren der *Indirekten Revokation* implementiert. Bei der Indirekten Revokation handelt es sich um eine Revokation, die nicht von *certificate issuer*, sondern von einer anderen Komponente (fremde CA, RevokationsInstanz) durchgeführt wird.

Zertifikaten, die zwei verschiedenen Stellen als *issuer* und *revocator* benutzen, müssen ein CRL Distribution Points-Attribut besitzen. *Key Usage* ist ein anderes Attribut, das in der Praxis oft zum Einsatz kommt.

Diese zwei Erweiterungen sind von besonderer Bedeutung und werden folgend im einzelnen betrachtet.

**Key Usage:** Dieses Attribut gehört zu der Klasse *Key Attributes*-Erweiterungen und ermöglicht der CA, einem Zertifikat die Kombination von neuen folgenden kryptographischen Diensten hinzuzufügen:

**keyCertSign:** Verifikation der Zertifikaten.

**cRLSign:** Verifikation von CRLs.

**non-Repudiation:** Verifikation der digitalen Signaturen bei non-Repudiation (s. Kapitel 1).

**digitalSignature:** Verifikation der digitalen Signaturen. Sollten drei ersten Attribute fehlen, so muss dieses gesetzt werden, wenn das Zertifikat u. a. zur Signatureverifikation benutzt wird.

**keyEnchipherment:** Verschlüsselung eines privaten Schlüssels (z. B. im Falle Verteilung via Internet).

**dataEnchipherment:** Datenverschlüsselung. Unter Daten werden hier alle digitalen Dokumenten, die keine Schlüssel sind, gemeint.

**keyAgreement:** Transfer eines privaten Schlüssels mittels Diffie-Hellman-Schlüsselaustausches.

**enchiperOnly:** Wird nur zusammen mit keyAgreement gesetzt und weist darauf hin, dass der zu übergebende symmetrische Schlüssel nur zur Datenverschlüsselung erstellt ist.

**dechiperOnly:** Wird nur zusammen mit keyAgreement gesetzt und weist darauf hin, dass der zu übergebende symmetrische Schlüssel nur zur Datenentschlüsselung erstellt ist.

**CRL Distribution Points:** Dieses Attribut enthält Informationen darüber, wie und wo (von welchem CRL Ersteller) eine CRL zu erhalten ist. Dieses Attribut besteht aus einer homogenen Menge der Unterattributen (*CRL Distribution Point* genannt), die drei folgenden Felder beinhalten:

1. Name,

2. Revokationsgrund (*reasons*),
3. CRLHersteller.

Dabei kann ein der folgenden Ereignisse den frühzeitigen Rückruf eines Zertifikaten (eine Revokation) verursachen:

**keyCompromise:** Es ist bekannt oder vermutet, dass der entsprechende private Schlüssel eines PKI-Teilnehmers gefährdet ist.

**cACompromise:** Es ist bekannt oder vermutet, dass der entsprechende private Schlüssel einer CA gefährdet ist.

**affiliationChanged:** Eine der im Zertifikat erhaltenen Attribute (z.B. subject name) wurde geändert. Eine Verletzung des privaten Schlüssels ist hier nicht zu vermuten.

**superseded:** Das Zertifikat wird nicht mehr benutzt. Es ist wiederhin eine Gefährdung des privaten Schlüssels nicht zu vermuten.

**cessationOfOperation:** Das Zertifikat wird nicht mehr für die im *Key Usage*-Feld definierte kryptographische Operation benutzt. Eine Verletzung des privaten Schlüssels ist nicht zu vermuten.

**certificateHold:** Dabei wird das Zertifikat in eine sogenannte *Warteschleife* für eine bestimmte Zeit gelegt. Während dieser Zeit gilt das Zertifikat als ungültig. Nach ihrer Ablauf kann es entgültig revoziert oder auch aus der CRL entfernt und somit wieder als *valid* deklariert werden.

**removeFromCRL:** Dieses Flag wird bei einer *Delta-CRL* (s. Kapitel 3.4) gesetzt, sollte das entsprechende *Basis-CRL* (s. Kapitel 3.1) für das gleiche Zertifikat certificateHold-Flag gesetzt haben. Wird das Zertifikat aus der Warteschleife entfernt, so ist auch der entsprechende Eintrag in der DeltaCRL zu löschen. Nach dem Ablauf seiner Gültigkeitsdauer kann das Zertifikat genauso aus der DeltaCRL entfernt werden.

## 2.3 X.509 Certificate Revocation Lists.

Laut X.509 Standard besteht eine Revokationsliste aus folgenden digitalen Fragmenten :

**tbsCertList:** Beinhaltet alle Informationen über die zu revozierenden Zertifikaten, sowie der Revokationsliste selbst. Über dieses Fragment wird die Signatur einer CRL berechnet, sein Inhalt soll deswegen nach dem Signieren unverändert bleiben.

**signatureAlgorithm:** Entspricht dem signatureAlgorithm-Feld eines X.509 Zertifikaten (s. Kapitel 2.2).

**signatureValue:** Entspricht dem signatureValue-Feld eines X.509 Zertifikaten (s. Kapitel 2.2).

### 2.3.1 Basic CRL Contents

Ähnlich dem X.509 Standard für digitale Zertifikate enthält eine CRL eine obligatorische (`tbsCertList`) und mehrere optionale Komponenten (`crlExtensions`). Dabei werden alle Zertifikatsdaten in einer Struktur namens *revokedCertificate* enthalten, die eine homogene Menge der Zertifikatsobjekten darstellt.

**tbsCertList:** Zu den Elementen dieser Liste gehören:

**version.** Der Wert 2 dieses optionalen Feldes weist auf die vorhandenen Extensions hin. Man verwendet momentan nur selten Version1-CRLs und Version2 kann auch im Falle fehlender Extensions gesetzt werden.

**signature.** Dieses Feld entspricht dem signature-Feld eines digitalen Zertifikaten (s. Kapitel 2.2).

**issuer.** Enthält X.500 distinguished name [2, S. 72] des CRL-Erstellers. Im Falle indirekter Revokation(s. Kapitel 3.2) delegiert die CA diese Aufgabe an eine weitere Instanz, deren Name dann in diesem Feld eingetragen wird.

**thisUpdate.** Speichert den Zeitpunkt der Revokation (Datum und Zeit, wann diese CRL erstellt wird), muss sowohl *UTC time* als auch *generalized time* Formate unterstützen.

**nextUpdate.** Legt den spätesten Zeitpunkt für das Erstellen einer neuen CRL fest, der wiederum in *UTC time* und *generalized time* angegeben werden kann.

**revokedCertificates.** Eine homogene Menge von Objekten, die alle Informationen über zu revozierende Zertifikaten beinhalten. Sollte diese Menge leer sein (z. B. bei Initialization einer Trust-Software), so wird die entsprechende CRL als *emptyCRL* bezeichnet.

**crlExtensions.** Alle zusätzliche Informationen über die entstehende CRL werden in diesen optionalen Komponenten gespeichert.

**revokedCertificates:** Jedes Element dieser Menge kann wiederhin in folgende feste und optionale Komponente aufgeteilt werden:

**userCertificate.** Enthält die sequentielle Nummer (*serialNumber*) des Zertifikaten (s. Kapitel 2.2). Im Falle der Indirekten Revokation muss zur eindeutigen Identifizierung eines Zertifikaten (s. Kapitel 3.2) der Zertifikatsersteller (*certificate issuer*) mittels entsprechender *CRL Entry* Extension angegeben werden.

**revocationDate.** Damit wird den Zeitpunkt der Revokation eines Zertifikaten festgelegt. Die Datumeingabe erfolgt wiederum in den *UTC time* und *generalized time* Zeitformaten.

**crlEntryExtensions.** Dieses optionle Feld ist nur in Version2-CRLs enthalten und über zusätzlichen Informationen für dieses Zertifikat verfügt.

### 2.3.2 Extensions.

Alle CRL- und CRL Entry-Extensions müssen entweder als *critical* oder als *non-critical* deklariert werden. Sollte die zu verifizierende CRL eine unbekannte *critical*-Extension haben, so wird die Verifikation dieser CRL fehlschlagen. Eine *non-critical*-Extension dagegen wird bei der CRL-Verifikation ignoriert. Im weiteren werden die wichtigsten CRL- und CRL Entry-Extensions betrachtet, die auch im Rahmen dieser Diplomarbeit zum Einsatz kommen.

**CRL Extensions:** Dazu zählt man u.a. :

**CRL Number.** Dabei wird jeder CRL eine Nummer vergeben. Da die CRL-Nummer streng monoton-wachsend sind, kann der Relying Party mit Hilfe dieser Erweiterung entscheiden, welche der zwei CRLs zum späteren Zeitpunkt erstellt wurde und somit als aktuelle gilt. Die Erweiterung gilt immer als *non-critical*.

**Delta CRL Indicator.** Damit wird eine CRL zu einer *Delta CRL* erklärt . Dabei enthält das Delta CRL Indicator-Attribut die CRL-Nummer(*CRL Number*) der dazu gehörenden *Basis-CRL* (s. Kapitel 3.1). Diese Extension ist immer *critical*.

**Issuing Distribution Point.** Diese Erweiterung ist immer *critical* und kann festlegen (s. Figure 7):

1. wo und wie diese CRL zu erhalten ist,
2. dass sie ausschlieslich End-Entity-Zertifikate revoziert,

```

id-ce-issuingDistributionPoint OBJECT IDENTIFIER ::= {id-ce 28}

issuingDistributionPoint ::= SEQUENCE {
    distributionPoint      [0] DistributionPointName OPTIONAL,
    onlyContainsUserCerts [1] BOOLEAN DEFAULT FALSE,
    onlyContainsCACerts   [2] BOOLEAN DEFAULT FALSE,
    onlySomeReasons       [3] ReasonFlags OPTIONAL,
    indirectCRL           [4] BOOLEAN DEFAULT FALSE }

```

Figure 7: Issuing Distribution Point-Extension

3. dass sie ausschließlich CA-Zertifikate revoziert,
4. dass sie ausschließlich Zertifikate mit der bestimmten Reason-Menge revoziert. Unter einer *Reason*-Menge kann jede beliebige Kombination im Kapitel 2.2 eingeführten Revokationsgründen verstanden werden,
5. dass es sich um eine Indirekte CRL handelt.

```

id-ce-cRLReason OBJECT IDENTIFIER ::= {id-ce 21}

CRLReason ::= ENUMERATED {
    unspecified          (0),
    keyCompromise       (1),
    caCompromise        (2),
    affiliationChanged   (3),
    superseded          (4),
    cessationOfOperation (5),
    certificateHold      (6),
    removeFromCRL       (8) }

```

Figure 8: CRL Reason Extension

**CRL Entry Extensions:** Bei der Implementierung der Indirekten CRLs werden insbesondere zwei folgenden CRL Entry Extensions benutzt:

**Reason Code.** Mit diesem immer *non-critical* Attribute kann der Revokationsgrund eines Zertifikaten festgelegt werden(s. Kapitel 2.2). Die

Syntaxstruktur dieser Erweiterung kann aus Figure 8 dem entnommen werden.

**Certificate Issuer.** Diese Erweiterung ist immer *critical* und erlaubt einer CRL Zertifikate verschiedener Zertifizierungsstellen zu revozieren. Dabei handelt es sich um eine Indirekte Revokation und das entsprechende Flag der *Issuing Distribution Point*-CRLerweiterung gesetzt werden muss.

Fehlt dieses Attribut bei dem ersten Eintrag (*CRL-Entry*) einer indirekten CRL, so ist der *Certificate Issuer* dem *CRL Issuer* gleichzusetzen.

Im allgemeinen, fehlt der *Certificate Issuer* bei einem Nachfolger-Eintrag einer indirekten CRL, so ist es von dem entsprechenden Attribut des Vorgänger-Eintrages zu entnehmen.

### 3 Revokationsmethoden.

Zu dem Aufgabenbereich dieser Diplomarbeit gehört u.a. die Implementierung verschiedener Revokationsmethoden. Die Wahl einer bestimmten Methode wird bei der Implementierung der Revokationsinstanz konfigurierbar gehalten (s. Kapitel 5) und ist von den folgenden Faktoren abhängig (s. [7, Kapitel 6]):

1. Verifikationsrate: Wie oft wollen die bestehenden Relying parties Repository für neue CRLs abfragen,
2. Lebensdauer einer CRL: Wie lange eine CRL von Relying parties benutzt wird und somit im Repository unersetzt bleibt,
3. Revokationsvolumen: Die Prognose für die Anzahl der zu revozierenden Zertifikaten,
4. PKI-Grösse und Architektur.

Die im weiteren beschriebenen Revokationsmethoden können in zwei Klassen aufgeteilt werden:

1. Traditionell Methoden. Alle Methoden dieser Klasse sind in *RFC 3280: Internet X.509 Public Key Infrastructure Certificate and CRL Profile* standardisiert und müssen von allen Clients unterstützt werden. Zu dieser Klasse gehören *Full CRLs*, *Indirekte CRLs*, *Delta CRLs*.
2. Neue Methoden. Nachdem es beim praktischen Einsatz der oben beschriebenen traditionellen Methoden eine Reihe von Problemen aufgetaucht ist, hat man ein paar neue Methoden entwickelt, die basierend auf den alten Algorithmen zu deren Effizienz beitragen sollten. Die dazu gehörenden *Over-Issued CRLs* kommen bereits in der Praxis zum Einsatz und werden auch im Rahmen des FlexiTrust-Projektes benutzt. Die *Sliding window delta-CRLs* ist auch eine neue Methode und kann in der Praxis nur im Zusammenhang mit *Delta CRLs* benutzt werden.

#### 3.1 Full CRLs.

Bei dieser oft verwendeten in der Praxis Methode revoziert jede CA *ausschliesslich* ihre eigene Zertifikate. Dabei enthält jede neue CRL alle Zertifikaten, die von dieser CA erstellt und revoziert wurden, und wird somit als *Full* oder *Complete* CRL bezeichnet.

Bei der Implementierung von Full CRLs ist folgendes zu beachten:

1. Alle Full CRLs enthalten Extensions und sind somit Version2-CRLs.
2. Alle Full CRLs haben CRL number extensions, damit Relying parties in der Lage sind, eine ältere von aktueller CRL zu unterscheiden.
3. Alle Full CRLs haben CRL reason code entry extension, falls die entsprechende CA über die Revokationsursache verfügt.
4. Alle Full CRLs enthalten das Feld nextUpdate, aus dem die Gültigkeit dieser CRL zu enthemmen ist.

Diese einfache und deswegen auch attraktive Methode ist aber mit folgenden Problemen verbunden:

- Da eine Full CRL alle von ihrer CA revozierten Zertifikate enthält, wird sie mit der Zeit sehr gross, was Signieren und Transfer dieser CRL sehr uneffizient macht.
- Zur Verifikation einer Zertifikatskette sind mehrere CRLs von Repository zu erhalten, da jede CA nur ihre eigenen Zertifikate revoziert.
- Jede neue Full CRL wird einmal pro CRL-Period erstellt und im Repository veröffentlicht. Dabei wird die Verzeichisdienst einmal pro Period am Zeitpunkt der CRL-Veröffentlichung mit Clients-Anfragen überlastet und danach nicht mehr benutzt.
- Es ist sehr schwierig, das CRL-Period richtig abzuschätzen. Die zu kurze Länge ist nur aus den Effizienzgründen unerwünscht (unnötige CRLs sind zu verwalten), während die grosse Periodlänge zur erfolgreichen Verifikation der bereits revozierten Zertifikaten führen kann.

Im weiteren wird die *Indirekte Revokation* betrachtet, die einen effizienteren Mechanismus zur Verifikation der Zertifikatsketten zur Verfügung stellt.

### 3.2 Indirekte CRLs.

Die Zertifizierungstellen können die Revokationsaufgabe an eine neue PKI-Komponente namens *Indirect CRL Authority* (ICRLA) delegieren, die sich ausschliesslich mit der Zertifikatenrevokation beschäftigt.

Jede *indirekte CRL* enthält im *revokedCertificate*-Element die Zertifikate der verschiedenen CAs, die aber alle in ihrem *CRL distribution point extension*-Attribut (s. Kapitel 2.2) die entsprechende ICRLA deklarieren müssen (s. Figure 9).

Ausserdem sind die CRL selber in ihrem *issuer distribution point extension*-Feld als indirekte CRLs zu definieren.

Das Beispiel aus Figure 9 stellt eine ICRLA vor, die Revokation der Zertifikate nur im Falle `keyCompromise` bzw. `caCompromise` durchführt. In anderen Fällen werden die Zertifikate von der entsprechenden CA revoziert. Alle PKI-Komponenten (CAs, Verzeichnisdienst CMA usw.), die mit ICRLA

```
CRLDistributionPoints {
  DistributionPoint {
    distributionPoint = ldap://dir.hawk.com/CN=ICRLA,
                      O=HAWK, C=US$certificateRevokationList
    reasons           = keyCompromise, caCompromise
    cRLIssuer         = CN=ICRLA, O=Hawk, C=US
  }
  DistributionPoint {
    distributionPoint = ldap://dir.hawk.com/CN=CRLDP48,
                      O=HAWK, C=US$certificateRevokationList
    reasons           = unused, affiliationChanged, superseded,
                      cessationOfOperation, certificateHold
  }
}
```

Figure 9: CRLDistributionPoint Extension

kommunizieren, müssen in der Lage sein, diese eindeutig zu identifizieren sowie die Daten an sie auf einem sicheren Wege zu übergeben. Das im Rahmen des FlexiTrust-Projektes entwickelte Transferprotokoll ermöglicht sichere Kommunikation zwischen einzelnen PKI-Komponenten (einschliesslich der Revokationsinstanz) und wird im Kapitel 4 dieser Diplomarbeit ausführlich beschrieben.

Jede indirekte CRL kann Zertifikate verschiedener CAs revozieren. Die indirekten CRLs lösen demzufolge das Problem der Zertifikatskettenberechnung, indem nur eine CRL mit Zertifikatseinträgen verschiedener CAs von Repository abzurufen ist.

Die einzelnen Zugriffe auf Repository erfolgen aber immer noch unverteilt zum gleichen Zeitpunkt. Die Lastverteilung für die Verzeichnisdienst wird mit dem *Over-Issued CRLs*-Verfahren wesentlich verbessert.

### 3.3 Over-Issued CRLs.

Bei der traditionellen Methode wird keine neue CRL erstellt solange die Lebenszeit (mit dem Attribute *nextUpdate* vorgegeben) der aktuellen CRL

## requests per second

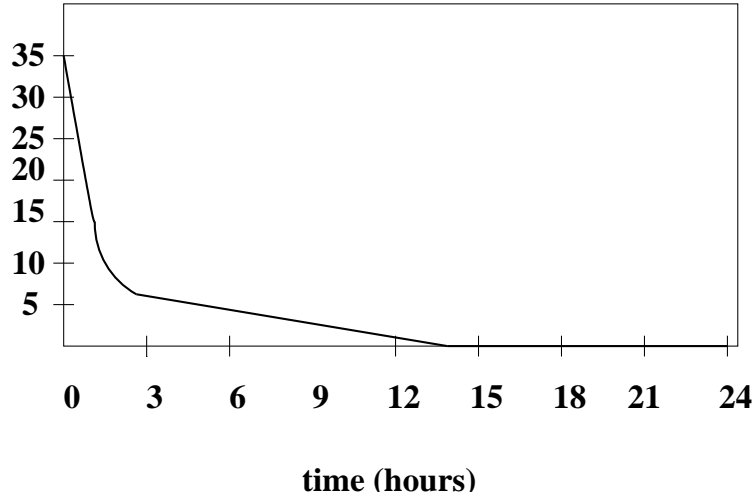


Figure 10: Traditionelle CRL

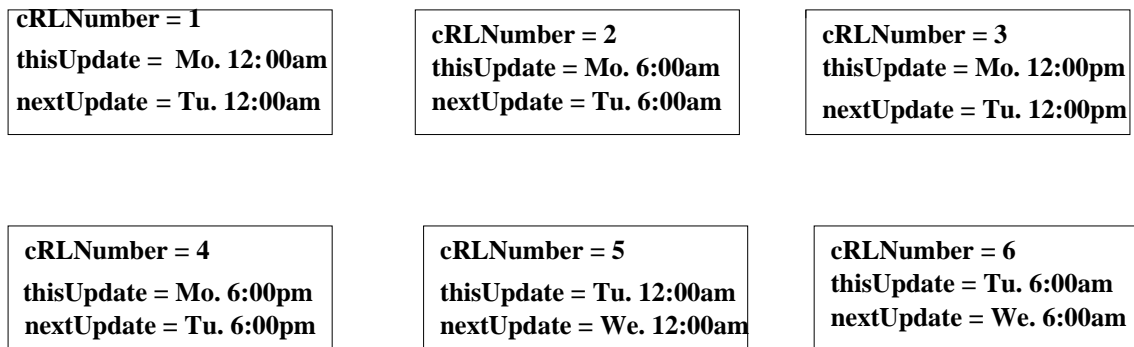


Figure 11: Over-issued CRLs

nicht abgelaufen ist. Dies führt aber dazu, dass alle Clients zu dem selben Zeitpunkt eine neue CRL von Repository abrufen, weil die alten nicht mehr gültig sind.

Sollte zum Beispiel (s. Figure 10) eine neue CRL um 0:00 Uhr erstellt werden, so ist die höchste Zugriffsrate zwischen 0:00 und 1:00 Uhr zu beobachten, während sie zwischen 3:00 und 12:00 rasch absinkt. Bis um 12:00 haben schon alle Clients die neue CRL lokal abgespeichert und seit diesem Zeitpunkt (d.h. die Hälfte des CRLPeriods) wird Repository nicht mehr benutzt.

Bei der *Over-Issued CRLs*-Methode wird eine neue CRL erstellt, obwohl die alte CRL noch gültig ist. Alle CRLs besitzen dabei die gleiche

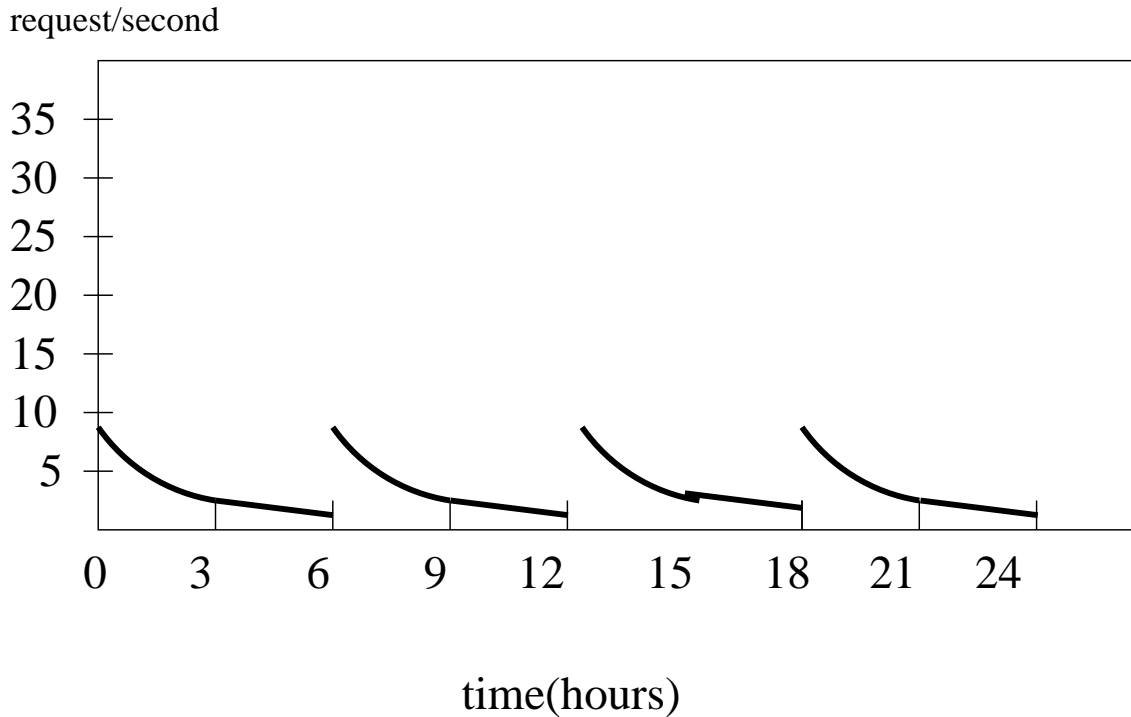


Figure 12: Over-issued CRLs: Requestrate

Gültigkeitsdauer, was dazu führt, dass zu der Zeit des CRL-Erstellens immer die gleiche Anzahl von CRLs zu erneuern ist. Diese entspricht der Anzahl der Clientszugriffen auf Repository, und ist wesentlich geringer als die gesamte Anzahl von Clients, die Repository benutzen.

Man kann zum Beispiel (s. Figure 11) eine neue CRL jede sechs Stunden veröffentlichen, wobei ihre Lebensdauer auf 24 Stunden begrenzt wird. Alle Clients, die im Besitz von CRL1 (erstellt am Montag um 12:00 Uhr morgens) sind, werden das Repository demnächst am Dienstag um 12:00 Uhr morgens abfragen. Zu diesem Zeitpunkt wird die CRL5 mit *nextUpdate* am Mittwoch um 12:00 Uhr morgens erstellt.

Die anderen Clients, die Repository am Montag zu einem späteren Zeitpunkt ab 6:00 Uhr morgens abgerufen haben, haben CRL2 bekommen, die erst am Dienstag um 6:00 Uhr morgens zu ersetzen ist.

Aus Figure 12 ist die Lastverteilung für Repository im diesen Fall zu entnehmen. Die höchste Zugriffsrates wird einmal pro sechs Stunden zu dem Zeitpunkt des Erstellens einer neuen CRL beobachtet. Ihr Wert ist aber viel niedriger als im Falle der traditionellen CRL.

Diesem Beispiel zufolge sind die *Over-Issued CRLs* eine sehr effektive Methode, solange man eine gleichmässige Lastverteilung für Repository anstrebt.

### 3.4 Delta CRLs.

Das *Over-Issued CRLs*-Verfahren löst nicht das Problem der CRL-Grösse. Um diese zu reduzieren, hat man die sogenannte *Delta-CRLs* eingeführt, die nur zusätzliche (update) Informationen für eine *Full-CRL* beinhalten. Diese Full-CRL ist damit als *Basis-CRL* bezeichnet, ihre Nummer im *Delta-CRL Indikator-Attribute* der entsprechenden Delta-CRL enthalten. Jede weitere Delta CRL übernimmt alle Revokationen von der alten Delta CRL, sowie die neuen Informationen. Da nicht alle Clients Delta-CRLs unterstützen, müssen immer zwei CRLs : eine *Full*- und eine *Delta-CRL* ausgestellt werden. Die CRL Number-Extensions dieser beiden CRLs müssen übereinstimmen.

Es seien zum Beispiel (s. Figure 14) zu den Zeitpunkten  $t_0$  und  $t_5$  zwei grosse Full CRLs erstellt (als  $F_1$  und  $F_2$  gekennzeichnet). Da die Anzahl der Revokationen, die zwischen diesen Zeitpunkten stattgefunden haben, nur sehr gering ist, hat man sich im diesen Fall für die Delta-CRLs  $D_1^1$ ,  $D_1^2$ ,  $D_1^3$  und  $D_1^4$  entschieden, wobei z.B. die  $D_1^4$  abgesehen von ihren eigenen Revokationen noch alle Einträge von  $D_1^3$  (d.h. auch von  $D_1^2$  und  $D_1^1$ ) enthält. Alle Delta CRLs beziehen sich auf  $F_1$  als *Basis-CRL*, indem das *Delta CRL Indikator-Attribut* dieser CRLs (s. Figure 13) auf das *CRL-Number-Attribut* von  $F_1 = 1$  gesetzt ist.

Gleichzeitig werden auch Full-CRLs  $F_1^1$ ,  $F_1^2$ ,  $F_1^3$ ,  $F_1^4$  erstellt. Diese sind keine Deltas und haben daher kein Delta CRL Indikator-Feld, dafür aber *CRL Number*-Erweiterung, die mit der *CRL Number*-Erweiterung der entsprechenden Delta-CRL übereinstimmt.

Die CRL-Nummerzuweisung ist aus Figure 13 zu entnehmen: Die CRL  $F_1$  wird zu dem Zeitpunkt  $t_0$  erstellt und erhält beispielweise die Nummer 1. Zu dem Zeitpunkt  $t_1$  wird eine neue Full CRL  $F_1^1$  erstellt, der demzufolge die Nummer 2 zugewiesen wird. Weil ihre Nummer mit der entsprechenden DeltaCRL-Nummer übereinstimmen muss, wird CRL Number von  $D_1^1$  auch auf 2 gesetzt usw.

Damit Relying Party das volle Revokationsbild (z.B. zu  $t_2$  rekonstruieren kann), sind folgende Schritte vorzunehmen:

1. Die  $F_1$  muss in dem Zeitintervall zwischen  $t_0$  und  $t_1$  vom Repository geholt werden. Da jede neue CRL die alte überschreibt, ist die  $F_1$  ab dem  $t_2$  nicht mehr im Repository.
2. Die  $D_1^2$  von Repository abrufen,
3. Aus der im Cache gespeicherten  $F_1$  und  $D_1^2$  eine neue Full CRL erstellen, deren Inhalt dem Inhalt von  $F_1^2$  entspricht.

| <b>CRL</b> | <b>CRL Number</b> | <b>Delta CRL Indicator</b> |
|------------|-------------------|----------------------------|
| $F_1$      | 1                 | –                          |
| $D_1^1$    | 2                 | 1                          |
| $F_1^1$    | 2                 | –                          |
| $D_1^2$    | 3                 | 1                          |
| $F_1^2$    | 3                 | –                          |
| $D_1^3$    | 4                 | 1                          |
| $F_1^3$    | 4                 | –                          |
| $D_1^4$    | 5                 | 1                          |
| $F_1^4$    | 5                 | –                          |
| $F_2$      | 6                 | –                          |

Figure 13: Delta-CRL: Extensions

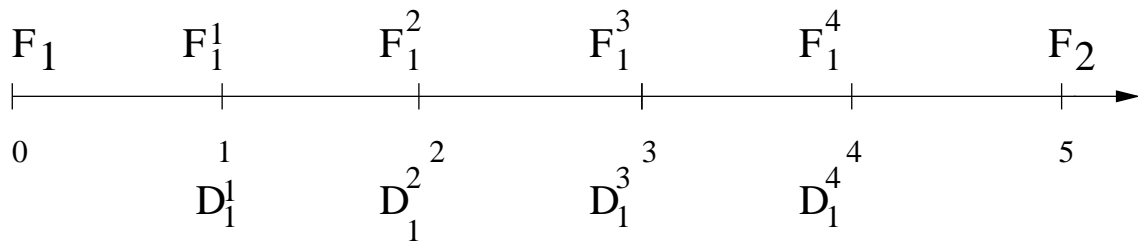


Figure 14: Delta-CRL

| <b>cRLNum</b> | <b>baseCRL</b>                                   | <b>delta-CRL</b>  |
|---------------|--|---|
| <b>1</b>      | <b>thisUpdate = 00:00<br/>nextUpdate = 04:00</b> | <b>thisUpdate = 00:00<br/>nextUpdate = 00:10<br/>BaseCRLNumber = 1</b>  |
| <b>2</b>      |  | <b>thisUpdate = 00:10<br/>nextUpdate = 00:20<br/>BaseCRLNumber = 1</b>  |
| <b>:</b>      | <b>:</b>   | <b>:</b>  |
| <b>24</b>     |  | <b>thisUpdate = 03:50<br/>nextUpdate = 04:00<br/>BaseCRLNumber = 1</b>  |
| <b>25</b>     | <b>thisUpdate = 04:00<br/>nextUpdate = 08:00</b> | <b>thisUpdate = 04:00<br/>nextUpdate = 04:00<br/>BaseCRLNumber = 1</b>  |
| <b>26</b>     |  | <b>thisUpdate = 04:10<br/>nextUpdate = 04:20<br/>BaseCRLNumber = 25</b> |
| <b>:</b>      | <b>:</b>   | <b>:</b>  |

Figure 15: Delta-CRL

Wenn Relying Party keine Delta CRLs unterstützen, so wird einfach um  $t_2$  die  $F_1^2$  von Repository geholt.

Sollte ein Zertifikat als *reason code-Attribute certificate-hold* haben, so wird das Delta CRL-Verfahren noch komplizierter.

Nachdem ein Zertifikat in einer CRL oder Delta CRL durch *certificate-hold* auf die Warteliste gesetzt wird, gibt es zwei folgenden Möglichkeiten:

1. Das Zertifikat wird aus der Warteliste entfernt, dabei muss der entsprechende CRL-Entry Eintrag aller weiteren Delta CRLs *remove-from-CRL* im reason code-Attribut enthalten bis dieses Zertifikat endgültig revoziert wird,
2. Das Zertifikat wird aus der Warteliste entfernt und endgültig revoziert. Dabei werden die üblichen Revokationsschritten vorgenommen.

Die Delta CRLs sind wegen ihrer Grösse sehr attraktiv und können auch eingesetzt werden, wenn die PKI eine andere Revokationsmethode als CRL benutzt (z.B. OCSP).

Ihr Einsatz in der Praxis ist aber mit folgenden Problemen verbunden:

| <b>cRLNum</b> | <b>baseCRL</b>                                   | <b>delta-CRL</b>  |
|---------------|--|---|
| ⋮             | ⋮  | ⋮   |
| <b>25</b>     | <b>thisUpdate = 04:00<br/>nextUpdate = 08:00</b> | <b>thisUpdate = 04:00<br/>nextUpdate = 04:10<br/>BaseCRLNumber = 1</b>  |
| <b>26</b>     | <b>thisUpdate = 04:10<br/>nextUpdate = 08:10</b> | <b>thisUpdate = 04:10<br/>nextUpdate = 04:20<br/>BaseCRLNumber = 2</b>  |
| ⋮             | ⋮  | ⋮   |
| <b>48</b>     | <b>thisUpdate = 07:50<br/>nextUpdate = 11:50</b> | <b>thisUpdate = 07:50<br/>nextUpdate = 08:00<br/>BaseCRLNumber = 24</b> |
| <b>49</b>     | <b>thisUpdate = 08:00<br/>nextUpdate = 12:00</b> | <b>thisUpdate = 08:00<br/>nextUpdate = 08:10<br/>BaseCRLNumber = 25</b> |
| ⋮             | ⋮  | ⋮   |

Figure 16: SlidingWindow Delta-CRL

1. Die Verifikationskosten eines Zertifikaten steigen,
2. Es müssen auch entsprechende Full CRLs, die sehr gross sein können, erstellt werden. Dabei ist das Signieren der grossen Dokumenten sehr zeitkritisch ist.
3. Nachdem eine Basis CRL in Repository überschrieben wird, ist sie nicht mehr erhaltbar. Selbst die Clients, die Delta CRL unterstützen, aber keine Basis CRL in ihren Cache speichern, können alle weiteren Delta CRLs nicht benutzen.

Insbesondere das dritte Problem macht den Delta CRL-Einsatz in der Praxis sehr uneffizient und wird mit Hilfe des nächsten Algorithmus behoben.

### 3.5 Sliding window delta-CRLs.

Jede Delta CRL enthält Zertifikate, deren Revokation in einem Zeitraum (*window*) zwischen thisUpdate von baseCRL und thisUpdate von Delta-CRL

stattfindet. Wie es aus Figure 15 zu sehen ist, hat dieses Zeitfenster bei den traditionellen Deltas unterschiedliche Grösse (von 10 Minuten bis zu 4 Stunden) Die Idee von Sliding window delta-CRLs besteht darin, eine feste Fenstergrösse für jede Delta CRL zu setzen, indem zu jeder Full CRL eine und nur eine Delta CRL erstellt wird.

Um dieses Fenster zu initialisieren, werden im Beispiel aus Figure 16 erste vier Stunden keine Delta CRLs erstellt. Die Revokation während dieser Zeit erfolgt durch Over-issued Delta CRLs mit einer Gültigkeitsdauer von 4 Stunden, die jede 10 Minuten erstellt werden.

Alle dieser Full CRLs sind gleichzeitig auch BasisCRLs. So werden um 4:00 Uhr eine Full CRL sowie eine Delta CRL mit BaseCRLNummer = 1 erstellt usw.

Der Vorteil dieser Methode besteht darin, dass alle Clients verteilt Delta CRLs benutzen können.

Hat z. B. ein Client die Full CRL 1 im Cache, so kann um 4:00 Uhr eine kurze Delta CRL anstatt Full CRL 25 vom Repository geholt werden.

Dasselbe gilt auch für die Clients die CRL 2, 3 usw. im Cache speichern und somit entsprechende Deltas benutzen.

Obwohl der Sliding window delta-CRL-Mechanismus die traditionelle Methode im wesentlichen verbessert, bleibt das Problem der Verifikation bei Delta CRLs immer noch ungelöst und trägt dazu bei, dass die Delta CRLs nur in seltenen Fällen in der Praxis zum Einsatz kommen.

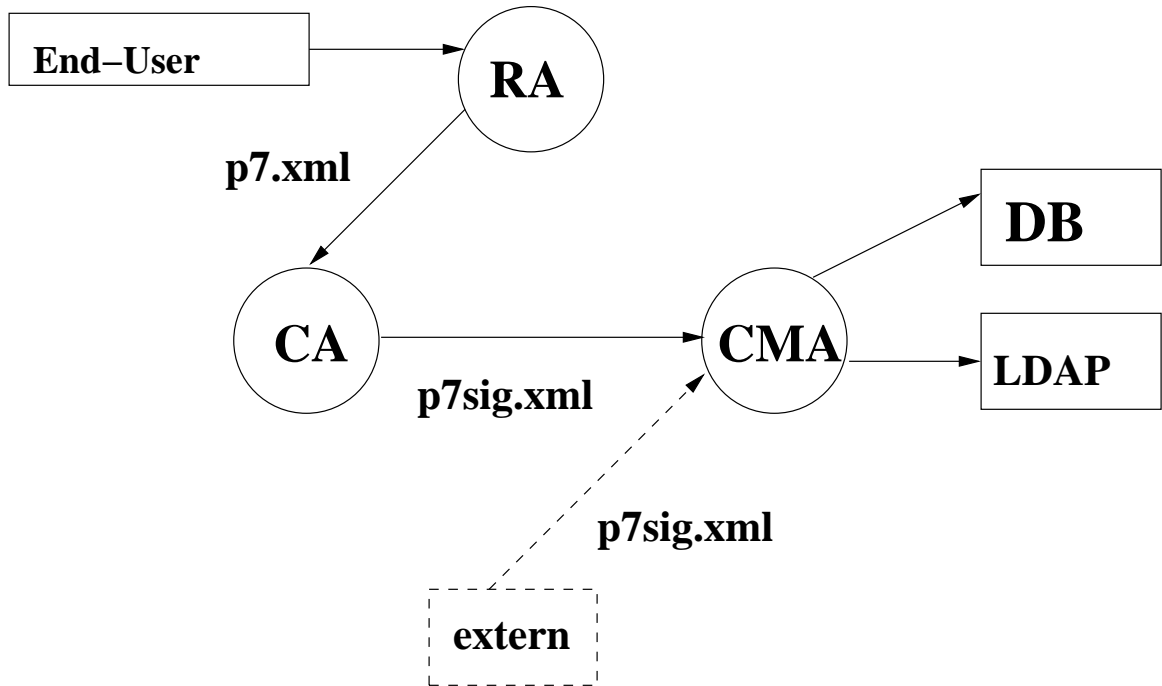


Figure 17: Kommunikationswege in einer PKI.

## 4 Transferprotokol

Die Kommunikation zwischen einzelnen Komponenten in einer PKI kann die Sicherheit des ganzen Systems beeinflussen und spielt deswegen eine zentrale Rolle bei der PKI-Planung und Entwicklung.

Das einfache Szenario für den Informationsaustausch innerhalb einer PKI ist aus dem Figure 17 zu entnehmen: Ein PKI-Teilnehmer geht zu der RA und beantragt das Schlüsselpaar (z.B. zum Datenverschlüsselung). Die RA stellt ihrerseits einen Antrag bei der CA, indem sie der CA eine Datei (auch *RA-Produkt* genannt) mit allen notwendigen Informationen z.B. *subject distinguished name* übergibt. Aus Sicherheitsgründen wird oft CA als eine off-Line Instanz implementiert und das RA-Produkt muss demzufolge auf einer Diskette bzw. einem USB-Token gespeichert mit Hilfe eines Administrators dieser zugeliefert werden. Das erzeugene und signierte Zertifikat wird dann in LDAP veröffentlicht. Zu diesem Zwecke wird ein CA-Produkt erzeugt, möglicherweise signiert und wieder auf einer Diskette der CMA zugeliefert. Nach einer erfolgreicher Verifikation kann die CMA alle Daten aus dem CA-Produkt extrahieren und sowie in LDAP als auch in der DB (für interne Zwecke) abspeichern.

Zur effizienten automatischen Bearbeitung digitaler Produkten wird ihre

Syntax mit Hilfe verschiedener Protokollen standardisiert (z.B. PKCS#7, PKCS#10, RFC 2510). Im Rahmen des FlexiTrust-Projektes wurde ein neues XML-basiertes Transferprotokoll entwickelt, das die maximale Flexibilität und Erweiterbarkeit der bestehenden Produkten anstrebt. Die Implementierung dieses Protokoll gehört zu dem Aufgabenbereich dieser Diplomarbeit.

## 4.1 Anforderungen an die Software

### 4.1.1 Ist-Zustand

Der Datentransfer zwischen den einzelnen Komponenten der FlexiTrust-Software erfolgt mit Hilfe von *p7*-Dateien, die das PKCS#7-Format ([9]) unterstützen. Dabei kann jede einzelne CA-Aufgabe *Processables* [3, Seite 18] signiert werden, bevor sie der *p7*-Datei zugefügt wird. Somit beinhaltet jede Transferdatei eine CA-Anfrage, die aus mehreren möglicherweise signierten *Processables* besteht. Im Anschluss wird diese mit der CA-Signatur versehen und serialisiert. Der Dateiempfänger (z.B. CMA) hat sowie die *p7*-Datei als auch die einzelnen *Processables* zu verifizieren. Erst nach der erfolgreichen Verifikation ist die weitere Bearbeitung der einzelnen Aufgaben auszuführen.

Diese Lösung ermöglicht die sichere Kommunikation innerhalb der PKI. Aber bei der Bearbeitung der *fremden* (von einer anderen PKI) Produkten (s. Figure 17) ist man auf folgende Probleme gestossen:

1. Alle Daten, die *p7*-Datei enthält, sind **Java-Objekte** und werden deswegen von den in anderen Sprachen (z.B C++) implementierten Komponenten nicht akzeptiert.
2. Serialisierungsproblem: Die Struktur der zu verarbeitenden Daten muss den vom Programm (z.B. CMA) erwarteten Strukturen entsprechen. Das Serialisierungs-API versucht diesem Problem mit einem Versionierungsmechanismus zu begegnen. Dazu enthält das Interface *Serializable* eine long-Konstante *serialVersionUID*, in der eine Versionskennung zur Klasse gespeichert wird. Sie wird beim Aufruf von *writeObject* (d.h. bei der Serialisierung ) automatisch berechnet und stellt einen Hashcode über die wichtigsten Eigenschaften der Klasse dar. So gehen beispielsweise Name und Signatur der Klasse, implementierte Interfaces sowie Methoden und Konstruktoren in die Berechnung ein. Selbst triviale Änderungen wie das Umbenennen oder Hinzufügen einer öffentlichen Methode verändern die *serialVersionUID*. Sollte z.B. ein Zertifikat von einer fremden in Java implementierten CA erstellt werden, so wird ihm bei der Serialisierung eine *serialVersionUID* vergeben,

die sich sicherlich von der `serialVersionUID` eines FlexiTrust-Zertifikates unterscheidet. Die FlexiTrust CMA akzeptiert ausschließlich Produkte mit ihr bekannten `serialVersionUID`. D.h. ein fremdes Produkt wird von CMA nur dann bearbeitet, wenn die CMA über die dieses Produkt implementierende Klasse verfügt (muss mit allen anderen Klassen in einer *jar-Datei* enthalten sein). Diese Voraussetzung kann in der Praxis nicht immer gewährleistet werden, was die Kommunikation zwischen der FlexiTrust und anderen PKIs kompliziert.

#### 4.1.2 Soll-Zustand

**XML:** Das Transferprotokoll verwendet die *extensible markup language XML* als Format für die Transferdateien (z.B. `p7.xml` im unseren Beispiel). Diese Sprache verfügt über eine Reihe von Eigenschaften, die bei dem Protokollentwurf und Implementierung eine entscheidende Rolle spielen:

1. Datenportabilität ermöglicht die Bearbeitung von den *fremden* Produkten (s. Figure 17)
2. Eine flexible, erweiterbare, menschenlesbare Datenstruktur vereinfacht den Einsatz von neuen Produkten sowie die Anpassung von alten.
3. Alle XML-Dokumente können mit Hilfe von verschiedenen kryptographischen Algorithmen signiert und verschlüsselt werden. D.h. in unserem Falle, dass alle Transferdateien sowie die einzelnen Produkte mit einer Signatur zu versehen sind, was zu der sicheren Kommunikation zwischen den PKI-Komponenten beiträgt.

**Protokoll:** Die Funktionsweise des Protokoll wird an dem oberen Beispiel geschildert. Als erstes beantragt die RA bei der CA ein Zertifikat für eine Person und hat deswegen folgende Daten zu übertragen:

1. Schlüsselpaar. Es ist in einem mit PKCS#12-Format standardisierten ([9]) Software token enthalten.
2. Name dieser Person.
3. Passwort für den Rückruf des Zertifikaten (*revocationPassword*).
4. E-mail Adresse dieser Person. Das zu erstellende Zertifikat ist an diese Person per E-mail zu verschicken.

Alle diese Informationen werden in der Form einer XML-Datei (s. Figure 18) auf eine Diskette gespeichert und der CA zugeliefert. Dabei werden alle zu übertragende Daten in ein XML-Element namens *application* eingeschlossen. Dieses Element wird im unseren Beispiel signiert, kann aber auch keine Signatur enthalten, da sein Vater-Element *request* auf jedem Fall mit einer Signatur zu versehen ist. Zu diesem Zeitpunkt hat das Element *cert* noch kein Inhalt und wird in diesem Beispiel als *EMPTY* definiert.

```
<request>
  <application profile=PKCS12>
    <PFX>StringValue</PFX>
    <p12Pass>StringValue</p12Pass>
    <revocPass>7c4a8d09...8941</revocPass>
    <cert></cert>
    <subjectDN>CN=Alice,OU=OrgUnitName,O=OrgName,C=DE</subjectDN>
    <email>alice@orgunit.orgname.de</email>
    <Signature><!-- signature elements ... --> </Signature>
  </application>
  <Signature><!-- signature elements ... --> </Signature>
</request>
```

Figure 18: RA-Anfrage

In dem zweiten Schritt verifiziert die CA sowohl die ganze RA-Anfrage als auch das *application*-Element. An dem *profile*-Attribut erkennt sie ihre Aufgabe und erstellt mit Hilfe ihr zugeteilter Daten ein Zertifikat für diese Person. Im Anschluss wird dieses in einer neuen XML-Datei (s. Figure 19) an die CMA übergeben. Dabei erhält das Attribut *profile* einen neuen Wert *CRT*, woran die CMA einen Antrag auf die Zertifikatveröffentlichung erkennt. Nach der erfolgreichen Verifikation dieser Anfrage veröffentlicht die CMA das Zertifikat in LDAP, schickt eine E-mail mit dem Zertifikaten an seinen Inhaber sowie speichert das Passwort in ihre interne Datenbank. Die CMA kann auch eine *fremde*-Anfrage anderer CA problemlos bearbeiten, solange sie den öffentlichen Schlüssel dieser CA besitzt.

**Anfrage-Profile:** Die Syntax des Anfrage-Produktes ist dem Beispiel aus Figure 20 zu entnehmen. Zur Gewährleistung der maximalen Flexibilität hat seine Struktur den folgenden Anforderungen zu genügen:

1. Jede Anfrage wird in einem XML-Element gespeichert, das bei allen Produkten als *request* bezeichnet wird.

```

<request>
  <application profile=CRT>
    <revocPass>7c4a8d09...8941</revocPass>
    <cert>Base64 encoding of a certificate</cert>
    <email>alice@orgunit.orgname.de</email>
    <Signature><!-- signature elements ... --> </Signature>
  </application>
  <Signature><!-- signature elements ... --> </Signature>
</request>

```

Figure 19: CA-Anfrage

2. Jede Anfrage muss digital signiert werden. Die über das *request*-Element berechnete Signatur wird in seinem Kind-Element *Signature* gespeichert.
3. Jede Anfrage muss in der Lage sein, gleichzeitig mehrere gleichartige oder verschiedene Aufgaben an den Kommunikationspartner zu übergeben. Eine Anfrageerweiterung ist jedoch nur vor der Signaturberechnung möglich. Die Syntax der einzelnen Aufgaben wird im nächsten Kapitel betrachtet.

```

<request>
  <application1 profile=profileName>
    <element1>value</element1>
    <element2>value</element2>
    <!-- ... more elements -->
    <signature> <!-- signature elements ...--> </signature>?
  </application1>
  <application2 profile=profileName>
    <element1>value</element1>
    <element2>value</element2>
    <!-- ... more elements -->
    <signature> <!-- signature elements ...--> </signature>?
  </application2>
  <!-- .... more applications -->

  <signature> <!-- signature elements ...--> </signature>
</request>

```

Figure 20: XML Syntax eines Anfrage-Produktes.

```

<application profile=profileName>
  <element1>value</element1>
  <element2>value</element2>
  <!-- ... more elements -->
  <parameter1 paramKey1=paramValue1></parameter1>?
  <parameter2 paramKey2=paramValue2></parameter2>?
  <!-- ... more parameters -->
  <signature> <!-- signature elements ...--> </signature>?
</application>

```

Figure 21: XML Syntax eines Aufgabe-Produktes.

**Aufgabe-Profile:** Das Beispiel aus Figure 21 stellt nur eine sehr abstrakte Syntax-Definition für die Aufgabe-Produkte. Dabei wird jedes Aufgabe-Produkt in einem XML-Element namens *application* enthalten, dessen Attribute *profile* den Aufgabentyp beschreibt. Die meist benutzten Aufgabentypen müssen vordefiniert zur Verfügung gestellt werden. Jeder besitzt eine erweiterbare Anzahl von XML-Elementen, die für diese Aufgabe notwendige Informationen enthalten. Es müssen bei der Implementierung Massnahmen getroffen werden, die den flexiblen Einsatz von neuen Typen sowie die Erweiterung der bestehenden um neue Elemente ermöglichen. Jedes Aufgabe-Produkt kann zusätzliche Informationen als Parameter-Elemente speichern. Jede einzelne Aufgabe kann im Anschluss signiert werden. Der Einsatz von verschiedenen Techniken bei der Signaturberechnung der Anfrage und der einzelnen Aufgaben (z. B. DSA und ECDSA [1]) erhöht das Sicherheitsgrad der Kommunikation.

## 4.2 Implementierung

Die Implementierung des Protokoles erfolgte in der Programmiersprache Java von Sun Microsystems. Zu den wichtigsten Eigenschaften dieser Sprache zählen Objektorientierung, weite Verbreitung, breite Unterstützung mit Bibliotheken sowie die Plattformunabhängigkeit. Besonders die letzte Eigenschaft sowie die Datenportabilität von XML ermöglichen den flexiblen Einsatz des Protokoles bei den verschiedenen PKIs.

Dieses Kapitel stellt einen Überblick über die Implementierung des Protokoles dar. Eine komplette Dokumentation wurde mit Hilfe der Dokumentationsschnittstelle *JavaDoc* erstellt. Die dadurch entstandenen HTML-Dateien stellen alle Informationen über den implementierten Klassen zur Verfügung (z.B. Klassenvariablen, Klassenkonstruktoren, Methoden usw.) und vereinfachen somit den Einsatz sowie die Weiterentwicklung des Paketes.

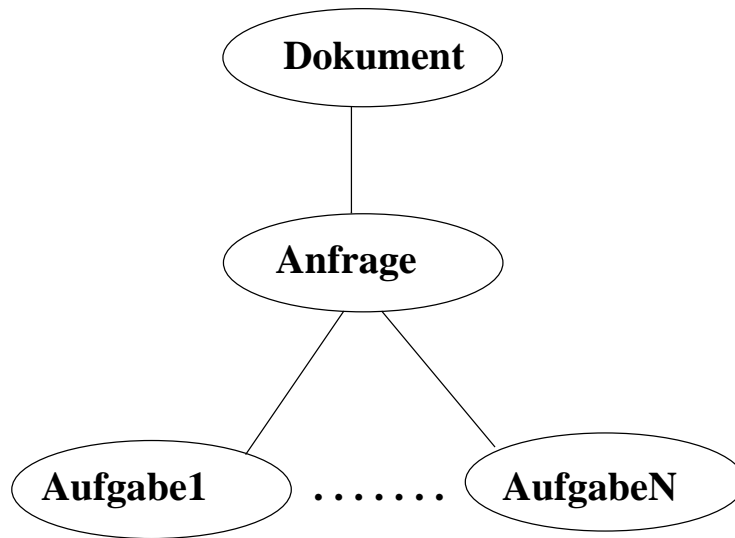


Figure 22: DOM-Baumstruktur für die P7.xml-Anfrage

#### 4.2.1 Übersicht

Die Transferprotokol-Software besteht aus drei Klassen-Paketen:

- Das Paket **de.tud.cdc.flexiTrust.cma.ca** enthält Klassen zum Erzeugen *p7.xml*-Dateien und wird als *P7Creator* bezeichnet.
- Das Paket **de.tud.cdc.flexiTrust.cma.cma** enthält Klassen zum Parsen und Bearbeiten von empfangenen XML-Dateien. Aufgrund seiner Funktionalität wird dieses Paket auch *P7Importer* genannt.
- Das Paket **de.tud.cdc.flexiTrust.cma.samples** enthält den Beispiel-Code zum Einsatz der zwei oberen Pakete.

#### 4.2.2 P7Creator

Bei der Implementierung aller XML-Produkte wird das Document Object Model **DOM** ([10, Seite67]) verwendet. Diese Technik benutzt die *Node*-Schnittstelle als Basisschnittstelle für alle XML-Strukturen. Das ganze XML-Dokument stellt somit einen Baum dar, dessen Knoten die einzelnen XML-Elemente sind. Das Einfügen eines neuen Elementen in dieses Dokument entspricht dem Einfügen eines Kindknoten in den Baum und deswegen leicht ausführbar.

Alle Anfrage- und Aufgabeprodukte werden auch in Form der XML-Knoten gehalten, wobei ein Anfrage-Knoten mehrere Aufgabeprodukte als

Kinderelemente enthalten kann (s. Figure 22). Diese Struktur ermöglicht einfache Anfrageerweiterung um neue Aufgaben, die allerdings nur vor der Signaturberechnung stattfinden kann.

Zu den Klassen dieser Komponente gehören:

**Request:** Diese Klasse implementiert Methoden zum Erzeugen, Signieren und Verifizieren eines Anfrageproduktes (s. Kapitel 4.1.2). Bei der Signaturberechnung wird die Wahl der kryptographischen Technik konfigurierbar gehalten.

**Profile:** Diese Klasse ist eine obere Klasse für alle Aufgabeprodukte und erlaubt das Einfügen von neuen Aufgabentypen sowie die Erweiterung der alten um neue Elemente.

Der Klassen-Konstruktor **Profile(String profile)** benutzt den String-Parameter *profile*, um den neuen oder bereits bestehenden Aufgabentyp zu definieren (s. Kapitel 4.1.2).

Die Klassenmethode **void addElement(String name, String value)** erweitert den Aufgabe-Knoten um ein neues Element und wird von allen unteren Klassen benutzt.

Diese Klasse implementiert auch Methoden zum Signieren und Verifizieren der einzelnen Aufgabe-Knoten die ähnlich den entsprechenden Methoden der Klasse **Request** verschiedene kryptographische Techniken unterstützen.

**X509CrtProfile:** Diese Klasse ist eine Unterklasse von **Profile** und erzeugt einen Aufgabe-Knoten für ein **X509Certificate** von der Form :

```
<application profile="CRT">
  <revocPass>StringValue</revocPass>
  <certificate>StringValue</certificate>
</application>
```

Dabei enthält das XML-Element *certificate* die Base64-Kodierte Darstellung des Zertifikaten.

Diese Klasse hat **String revocPass** und **X509Certificate cert** als Klassenvariablen und verfügt über verschiedenen Set- und Get- Methoden für diese Elemente.

Die Klassen **X509CrlProfile**, **X509P11Profile** und **X509P12Profile** sind weitere Unterklassen von **Profile**. Sie implementieren Aufgabe-Objekten für eine X509Crl, ein Hardware-Token ([9, PKCS#11]) und ein Software-Token ([9, PKCS#12]). Die XML-Syntax für diese Aufgabe-Produkte ist dem Anhang B zu entnehmen.

### 4.2.3 P7Importer

Das Paket `de.tud.cdc.flexiTrust.cma.cma` beinhaltet folgende Klassen:

**XMLRequest:** Diese Klasse implementiert Methoden zu Extrahieren von den Anfrage-Knoten aus P7.xml-Dateien. Nach erfolgreicher Verifikation kann die Anzahl der Aufgaben in diesem Knoten ermittelt werden. Jeder einzelne Aufgabe-Knoten ist dann per seine Position (Nummer) in der Anfrage anzusprechen.

**XMLProfile:** Mit dieser Klasse werden einzelne Aufgaben aus dem Anfrage-Knoten extrahiert und ggf. verifiziert.

Für die vordefinierten Aufgaben (wie z.B. **X509CrtProfile**) implementiert diese Klassen Methoden zum Extrahieren ihrer einzelnen Elementen. So erzeugt z.B. die Methode **X509Certificate** `getCertificate()` ein X509-Zertifikat. Dafür parst sie den Aufgabe-Knoten für das Element *certificate*, extrahiert seinen String-Wert und umwandelt diesen mit Hilfe *Base64-Dekodierung* in einen Byte Array. Daraus wird mit der Klasse `java.security.cert.X509Certificate` ein Zertifikat erzeugt, dessen Inhalt dem in Aufgabe-produkt kodierten Zertifikaten entspricht.

### 4.2.4 Code-Beispiel

Der Einsatz einzelner Methoden der `P7Creator` und `P7Importer` kann am folgenden Szenario geschildert werden: Eine CA erstellt eine neue CRL, speichert ihre Base64-Kodierte Darstellung in einer Datei namens `beispiel.crl` und beauftragt die CMA diese in LDAP zu veröffentlichen .

Die Implementierung dieses Beispiels kann in folgende Schritte unterteilt werden:

1. Die CA extrahiert die Base64-kodierte Darstellung aus `beispiel.crl` und speichert diese in einem byte array zur weiteren Bearbeitung:

```
File crlFile = new File('beispiel.crl');
FileInputStream in = new FileInputStream(crlFile);
byte [] encodedCrl = new byte [in.available()];
in.read(encodedCrl);
in.close();
```

2. Es wird mit dem Konstruktor `X509CrlProfile(byte [] rawCrl, String revocPass)` einen Aufgabe-Knoten erzeugt:

```
X509CrlProfile crlProfile = new X509CrlProfile(encodedCrl, "12345");
```

3. Die erzeugte Aufgabe wird signiert:

```
Document signedProfile = crlProfile.signProfile(privateKey, certificate);
```

4. Die CA erzeugt eine neue Anfrage:

```
Request caRequest = new Request();
```

5. Dem Anfrage-Knoten wird die signierte Aufgabe als Kind-Element zugefügt:

```
caRequest.addProfile(crlProfile);
```

6. Die vollständige Anfrage wird signiert:

```
caRequest.signRequest(privateKey, certificate);
```

7. Das entstandene Anfrageprodukt wird in der P7.xml-Datei auf einer Diskette gespeichert und der CMA übergeben:

```
String xmlFile="P7.xml";  
caRequest.getXML(xmlFile);
```

8. Die CMA erzeugt aus der Transferdatei den Anfrage-Knoten:

```
XmlRequest request = new XmlRequest("P7.xml");
```

9. Die Anfrage wird mit dem öffentlichen Schlüssel von CA verifiziert.

```
if (request.verify(publicKey)){ \** weitere Bearbeitung *\}
```

10. Im Falle einer erfolgreichen Verifikation wird die Aufgabenanzahl ermittelt (1 im unseren Beispiel) und einen Aufgabe-Knoten erzeugt:

```
int profileNumber = request.getRequestLength();  
XmlProfile profile = new XmlProfile(request,0);
```

11. Es wird überprüft, ob der Aufgabe-Knoten signiert ist und anschliesslich verifiziert:

```
if (profile.isSigned()){profile.verify(certificate)}
```

12. Im Anschluss wird aus dem Inhalt des Aufgabe-Knotens eine X509Crl erzeugt und in der Datei cma.crl in der CMA-Umgebung intern gespeichert:

```
profile.getCrl("cma.crl")
```

Die in der cma.crl enthaltene CRL ist wieder Base64-kodiert und kann von der CMA sowohl in der DB als auch in LDAP veröffentlicht werden.

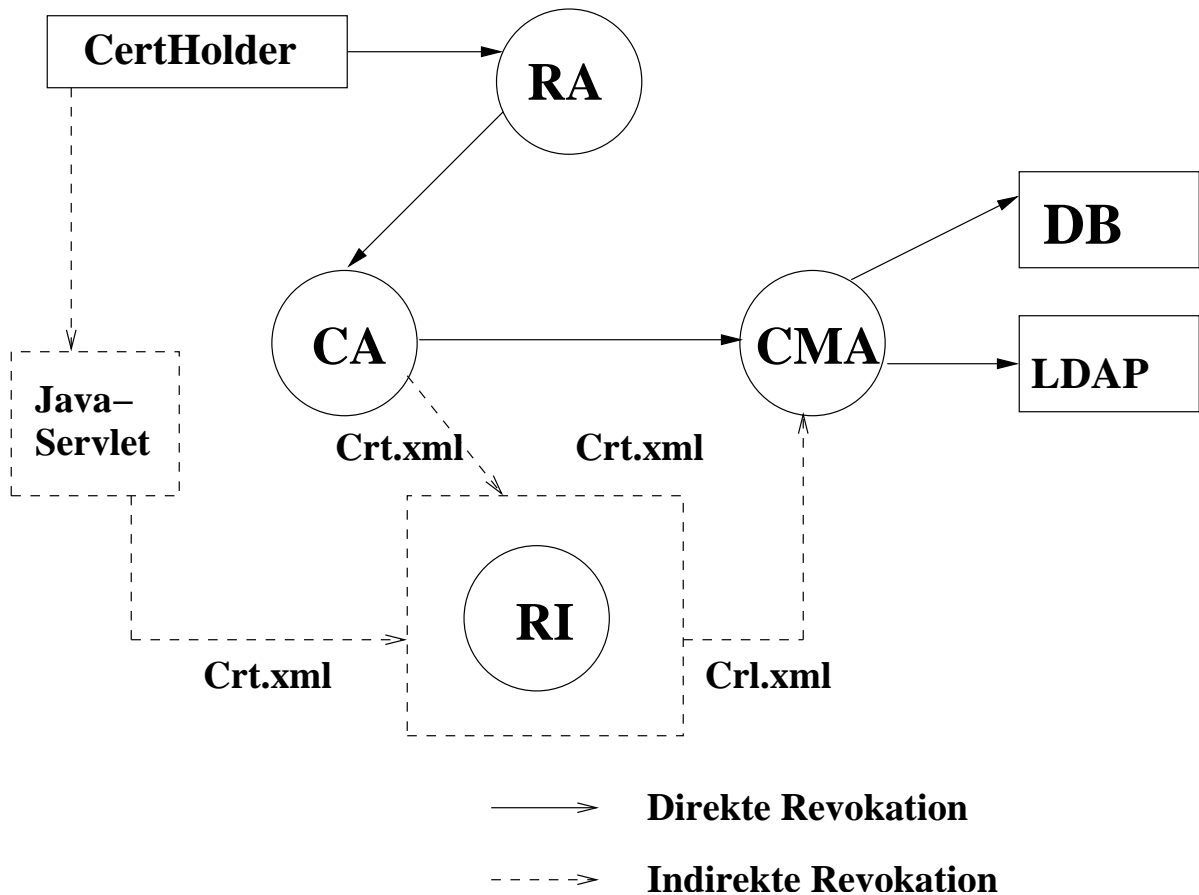


Figure 23: Zertifikatenrevokation: Direkte und Indirekte Methode

## 5 Revokation-Komponente

### 5.1 Entwurf

#### 5.1.1 Anforderungen an die Software

**Ist-Zustand:** Zu Beginn dieser Arbeit hat man bei dem FlexiTrust-Projekt die direkte Methode zur Zertifikaten-Revokation verwendet (s. Figure 23). Dem Over-issued Verfahren zufolge (s. Kapitel 3.3) wurde eine neue CRL einmal pro einer festgesetzten (aber konfigurierbaren) Periode durch die FlexiTrust-CA erstellt. Dabei kann der ganze Revokationsablauf in folgende Schritte unterteilt werden:

1. Zur Revokation seines Zertifikaten setzt sich der CertificateHolder in Verbindung mit der RA.

2. Die FlexiTrust-RA überprüft die persönlichen Daten des Antragstellers (z.B. Name, Vorname, RevocationPassword) und im Falle einer erfolgreichen Verifikation reicht die Revokationsanfrage an die FlexiTrust-CA weiter.
3. Die CA erstellt eine neue CRL, signiert sie digital und gibt an die FlexiTrust-CMA weiter. Sollte der Zeitpunkt der Revokation mit dem Zeitpunkt der Erstellung einer neuen CRL nicht übereinstimmen (was in der Praxis sehr oft zutrifft), so wird auf jedem Fall eine neue "unplanmässige" CRL erstellt.
4. Die FlexiTrust-CMA veröffentlicht die neue erstellte CRL in LDAP. Sie kann diese aber auch in der DB für ihre internen Verwaltungszwecke abspeichern.

Der Revokationsalgorithmus selbst sowie die Datenverwaltung zwischen den einzelnen PKI-Komponenten sind hier sehr einfach, was diese Methode sehr attraktiv für den Einsatz in der Praxis macht. Sie weist aber auch eine Reihe der schwachen Punkten auf:

1. Eine sofortige Revokation ist nicht immer möglich: Der Datentransfer zwischen RA und CA erfolgt mit Hilfe eines Administrators und z.B. nachts nicht bei jeder PKI möglich.
2. Es wird nur eine Revokationsmethode (Over-issued CRLs) implementiert, die aus den in Kapitel 3.3 genannten Gründen nicht immer effizient ist.
3. Im Falle einer *Hierarchical* PKI, was das FlexiTrust-Projekt u.a. unterstützt, erstellt jede CA ihre eigenen CRLs. Der Relaying Party muss das Repository (bzw. verschiedene Repositories) für jeden einzelnen CRL-issuer abfragen, wenn eine Zertifikatenkette zu verifizieren ist.

**Soll-Zustand:** Die Aufgabe dieser Diplomarbeit war Entwicklung und Implementierung einer neuen Komponente, die die FlexiTrust-CA um die Revokationsaufgaben entlastet und mit anderen PKI-Komponenten mittels des (im Kapitel 4 beschriebenen) Transferprotokoles kommuniziert. Hierbei wurden an die neue hinzukommende Komponente folgende Anforderungen gestellt:

1. Der Zeitpunkt der Revokation soll nicht durch die CA, sondern durch den CertificateHolder bestimmt werden, da er alleine das Risiko bei Verwendung eines ungültigen Zertifikaten trägt. Zu diesem Zwecke ist ein HTML-Formular zu implementieren, das die Revokationsaufträge der CertificateHolder jede Zeit annehmen kann.

2. Es sollen mehrere Revokationsmethoden implementiert werden. Die Wahl einer bestimmten Methode hängt sehr stark von der konkreten PKI-Realisierung (s. Kapitel 3) ab und soll deswegen durch den PKI-Administrator erfolgen.
3. Die Revokation-Komponente (RI) soll Multi-CA-fähig sein. Die Revokationen sollen durch die indirekte CRLs erfolgen (s. Kapitel 3.2), die gleichzeitig Zertifikaten von verschiedener Ersteller beinhalten.

**Anwendungsszenarien:** Aus Figure 23 sind zwei verschiedenen Szenarien: Revokation durch einen CertificateHolder und Revokation durch eine CA zu entnehmen.

In dem ersten Fall hat der CertificateHolder einen Formular im Internet auszufüllen, dabei sind u.a. die Zertifikatennummer und das Revokationspassword anzugeben. Das Java-Servlet überprüft die Eingangsdaten, erstellt eine signierte Transfer-Datei *Crt.xml* (s. Kapitel 4) und reicht diese an die Revokation-Komponente weiter. Diese kümmert sich um die Revokation, indem ein *Crl.xml* Produkt erstellt und an die CMA übergeben wird.

Bei der zweiten Variante bearbeitet die Revokation-Komponente Aufträge verschiedener CAs in einer Multi-CA-fähigen PKI. Jede CA-Anfrage wird wieder in Form einer *Crt.xml* an die RI übergeben und genauso wie in dem ersten Fall weiter bearbeitet.

Die Revokation-Komponente soll die beiden Variante implementieren, die Wahl der Revokationsmethode ist dabei dem CertificateHolder zu überlassen.

### 5.1.2 Revokation-Servlet

Das Revokation-Servlet ermöglicht eine sofortige Revokation zu dem durch CertificateHolder definierten Zeitpunkt. Die Revokation ist in diesem Fall erst nach dem erfolgreichen Ablauf drei folgenden Schritten möglich:

**Revokationsdaten-Eingabe und Überprüfung.** Mittels eines HTML-Formulares werden die Nummer des zu revozierenden Zertifikaten *serialNumber*, der Name seines Erstellers *issuerDN*, der Revokationsgrund und das Revokationspassword eingegeben. Zur Überprüfung dieser Daten erstellt das Servlet eine Verbindung zu dem Datenbank- oder LDAP-Server (die Wahl des Servers ist konfigurierbar und dem PKI-Administrator zu überlassen) und sucht nach dem Zertifikaten, das durch die Kombination von *serialNumber* und *issuerDN* eindeutig identifiziert wird. Der Revokation-Antrag ist sofort nach der Zertifikatensuche aufgrund folgender Fehler abzuweisen:

1. Das Zertifikat mit der eingegebenen Nummer existiert nicht,
2. Das Zertifikat mit der eingegebenen Nummer wird von einer anderen (als angegeben) CA erstellt,
3. Das Zertifikat mit den eingegebenen *serialNumber* und *issuerDN* ist gefunden, das Revokation-Passwort ist aber falsch.

**Erzeugen einer XML-Anfrage.** Im Falle einer erfolgreichen Suche wird aus dem gefundenen Zertifikaten ein Anfrage-Produkt erzeugt (s. Kapitel 4), mit dem privaten Schlüssel vom Servlet signiert und in einer Transfer-Datei gespeichert. Zu der eindeutigen Identifizierung erhält jede solche Datei als Name den Zeitpunkt ihrer Erstellung (einen Zeitvermerk) von der Form: **YYYY.MM.DD.at.HH.MM.SS.xml**.

**Übertragung der Revokation-Anfrage.** Die vom Servlet erzeugte und signierte Revokation-Anfrage wird zur weiteren Verarbeitung via Internet der RI übergeben. Die erfolgreiche Revokation wird dem CertificateHolder durch das Servlet zu dem selben Zeitpunkt bestätigt.

### 5.1.3 RI-Dämon

Die Revokation-Komponente stellt ein Hintergrundprozess (RI-Dämon) dar, der ständig auf die Arbeit wartet. Hierzu überwacht der RI-Dämon ein Eingangsverzeichnis nach Crt.xml Transferdateien und liest diese dann ein. Die Bearbeitung einer Transferdatei erfolgt durch die RI-Komponente in folgenden Schritten:

1. Die Signatur der Anfrage wird verifiziert. Dafür muss die RI im Besitz der Zertifikaten von Java-Servlet und aller zu der PKI gehörenden CAs sein. Die weiteren Revokationsschritte sind nur nach der erfolgreichen Verifikation möglich.
2. Das Aufgabe-Produkt wird aus der Datei extrahiert und, wenn es nötig ist, verifiziert (s. Kapitel 4).
3. Die aus dem Crt.xml extrahierten Informationen werden in einer internen Datenbank gespeichert.
4. Die bearbeitete Transferdatei wird aus dem Eingangsverzeichnis gelöscht.
5. Es wird eine neue CRL erstellt und signiert.
6. Es wird ein Crl.xml Produkt erstellt und signiert (s. Kapitel 4).

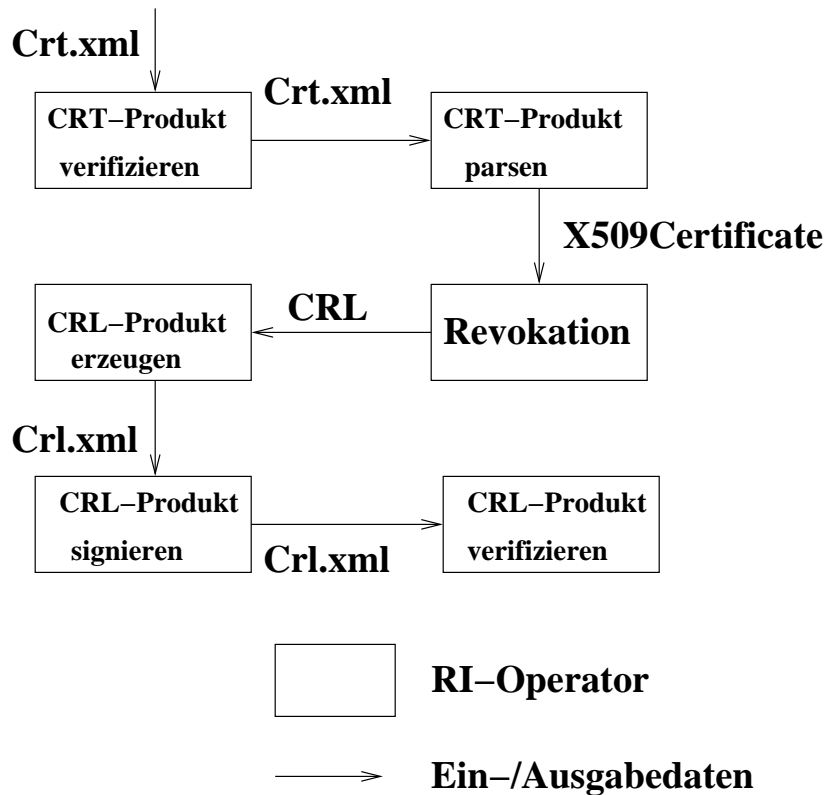


Figure 24: Bearbeitung einer Revokation-Anfrage durch Operatoren

7. Die Crl.xml Transferdatei wird zur weiteren Übergabe an die CMA im Ausgangsverzeichnis gespeichert.

Der RI-Dämon kann von der Kommandozeile gestartet und gestoppt werden. Die Änderung der Revokation-Methode kann nur statisch vor dem RI-Starten vorgenommen werden.

#### 5.1.4 Operatoren

Alle oben beschriebenen Bearbeitungsschritte werden von separaten Modulen implementiert, den sogenannten Operatoren. Jede Operator nimmt die Eingabedaten, bearbeitet diese und gibt an den nächsten Operator (s. Figure 24). Die Reihenfolge dieser Operatoren wird durch eine Konfiguration-Datei vor dem Starten der RI festgelegt. Um die Basisfunktionen abzudecken, enthält RI Operatoren für die folgenden Aufgaben:

- Over-issued CRL-Revokation (s. Kapitel 3.3)
- Delta CRL-Revokation (s. Kapitel 3.4)

- Sliding window Delta CR-Revokation (s.Kapitel 3.5)
- Verifikation einer Transferdatei
- Signieren einer Transferdatei
- Erzeugen einer Crl.xml Transferdatei
- Parsen einer Crt.xml Transferdatei und Speicherung der extrahierten Informationen in der internen Datenbank

Für neue Aufgaben (z. B. Einführung einer neuen Revokationsmethode) können weitere Operatoren entwickelt und integriert werden.

## 5.2 Implementierung

Die Revokation-Komponente besteht aus einer Web-Applikation, mit deren Hilfe über einen Webbrowser die Formulare (Revokation-Aufträge) ausgefüllt werden, und einem Modul (Revokation-Instanz genannt), das sich um die Revokationen kümmert. Die Implementierungen dieser beiden Komponenten werden in weiteren erläutert.

### 5.2.1 Revokation-Servlet

**Übersicht:** Die Revokation Web-Applikation besteht aus einem HTML-Formular zur Beantragung einer Revokation per Internet `Revocation.html` und einem Revokation-Servlet `CertRevocation.jsp` (s. Kapitel 5.1.2).

**Java-Servlet:** Die Bearbeitung einer Revokation-Anfrage durch das Java-Servlet kann in folgende Schritte unterteilt werden:

1. Eingabe von den Revokation-Parametern (s. Kapitel 5.1.2) mittels POST-Methode:

```
String issuerDN = request.getParameter("IssuerDN");
String serialNumber = request.getParameter("serialNumber");
String revocPass = request.getParameter("revocPassword");
revocPass.trim();
String revocReason = request.getParameter("revocation reason");
```

2. Einlesen der Konfiguration-Parametern. Der Pfad zu der Konfigurationsdatei für diese Web-applikation wird in einem *Web Application Deployment Descriptor*, einer Datei namens `web.xml` (s. Anhang) angelegt:

```

<!-- Path for config.xml -->
<init-param>
    <param-name>config</param-name>
    <param-value>../webapps/ri/etc/config.xml</param-value>
</init-param>

```

Das Java-Servlet ermittelt die Konfigurationsdatei mittels zwei folgenden Befehle:

```

/* gets path to the configFile */
ServletConfig Config = getServletConfig();
String configFile = Config.getInitParameter("config");

```

Die Konfiguration-Parametern werden im Anschluss mit der Klasse **ConfigParser** (s. Kapitel 5.2.2.1) eingelesen:

```

ConfigParser parser = new ConfigParser(new File(configFile));
String SearchLocation = parser.getSearchLocation();

```

Dabei legt der Parameter **SearchLocation** fest, ob eine Datenbankverbindung oder eine LDAP-Verbindung zu erstellen ist.

3. Eingangsdatenverifikation. Im Falle der Datenbanksuche erfolgt diese mit Hilfe der Klasse **DBRevocationInfo**. Die LDAPsuche benötigt dazu die Klasse **LdapCertificateInfo** (s. Kapitel 5.2.2.1)
4. Im Falle einer erfolgreicher Verifikation wird ein X509CrtProfile erzeugt und signiert. (s. Kapitel 4)
5. Die neue erstellte Revokation-Anfrage wird in einer XML-Datei gespeichert. Da keine Revokation verloren gehen darf, muss die Revokation-Komponente in der Lage sein, jede eingehende Transferdatei eindeutig zu identifizieren. Ausserdem darf der Inhalt der Transferdateien durch das Betriebssystem vor der Revokation nicht überschrieben werden. Um diese Anforderungen zu gewährleisten, wird jede Transferdatei mit dem Zeitvermerk im Namenfeld versehen:

```

/* write request to XML-File with the timestamp as a filename */
java.util.Date date = new java.util.Date();
String year = new Integer(date.getYear()+1900).toString();
String month = new Integer(date.getMonth()+1).toString();
String day = new Integer(date.getDate()).toString();

```

```

String hour = new Integer(date.getHours()).toString();
String minute = new Integer(date.getMinutes()).toString();
String second = new Integer(date.getSeconds()).toString();
String timestamp = year+"."+month+"."+day+"at"+hour+"."+minute+"."+second;
xmlFile = dir+timestamp+".xml";
req.getXML(xmlFile);

```

**KeyStores:** Dem Java-Servlet stehen zur Speicherung seiner Schlüssel ein `RSAKeyStore` und ein `DSAKeyStore` zur Verfügung. Im Falle einer Verifikation wird das Signaturalgorithmus direkt aus der Signatur ermittelt. Vor dem Signieren einer Transferdatei ist dieses aus der Konfigurationdatei zu entnehmen:

```

String signatureAlgorithm = parser.getSignatureAlgorithm();
String digestAlgorithm = parser.getDigestAlgorithm();

```

Zum Extrahieren eines Schlüssels (z.B. eines Zertifikaten) aus einem KeyStore sind dann folgende Schritte vorzunehmen:

1. Einlesen des Pfades zu dem KeyStore aus der Konfigurationdatei:

```
keystoreFileName = parser.getRSAKeystore();
```

2. Einlesen von `certificateAlias` (einem Vermerk, das dieses Zertifikat in dem KeyStore eindeutig identifiziert) aus der Konfigurationdatei:

```
keystoreFileName = parser.getRSACertAlias();
```

3. Erzeugen einer Java-Instanz `KeyStore`:

```
KeyStore publicKeystore = KeyStore.getInstance("JKS");
```

4. Beladen dieser Instanz mit dem Inhalt vom `RSAKeystore`.

```

File keystoreFile = new File(keystoreFileName);
FileInputStream fis = new FileInputStream(keystoreFile);
publicKeystore.load(fis, null);
fis.close();

```

5. Extrahieren des `X509Zertifikaten`:

```

X509Certificate certificate =
    (X509Certificate)publicKeystore.getCertificate(certificateAlias);

```

Dabei ist die Klasse `KeyStore` in dem Paket `java.security.*` und die Klasse `X509Certificate` in dem Paket `java.security.cert.*` enthalten.

## 5.2.2 Revokation-Komponente

**Übersicht:** Die Revokation-Komponente besteht aus folgenden Java-Klassenpaketen:

1. Das Paket **de.tud.cdc.flexiTrust.ri** enthält die Klasse **RIDemon**, das Hauptprogramm des RI-Dämons.
2. Das Paket **de.tud.cdc.flexiTrust.ri.util** enthält die Klassen:
  - ConfigParser**, die Methoden für den Zugriff auf XML-Konfigurationsdaten implementiert.
  - TaskCRL**. Diese Klasse implementiert eine Aufgabe für die Klasse **java.util.Timer**. Diese Klasse wird von dem Over-issued CRL Operator (s. Kapitel 5.1.4) aufgerufen.
  - TaskDeltaCRL**. Diese Klasse implementiert eine Aufgabe für die Klasse **java.util.Timer**. Diese Klasse wird von dem Delta CRL Operator (s. Kapitel 5.1.4) aufgerufen.
  - TaskSlidingWindow**. Diese Klasse implementiert eine Aufgabe für die Klasse **java.util.Timer**. Diese Klasse wird von dem Sliding window Delta CRL Operator (s. Kapitel 5.1.4) aufgerufen.
3. Das Paket **de.tud.cdc.flexiTrust.ri.crl** enthält die Klasse **IndirectCrl**, die Methoden zur Erzeugung verschiedener CRLs (z.B. DeltaCRL, Full-CRL usw.) implementiert.
4. Das Paket **de.tud.cdc.flexiTrust.ri.operators** enthält die Klassen :
  - Operator**, eine abstrakte Klasse für alle RI-Operatoren,
  - OverIssued**, eine Unterklasse von Operator, die Over-issued Revokationmethode implementiert,
  - Delta**, eine Unterklasse von Operator, die Delta CRL Revokationmethode implementiert,
  - SlidingWindow**, eine Unterklasse von Operator, die Sliding window Delta Crl Revokationmethode implementiert,
  - Verifier**, eine Unterklasse von Operator, die die Verifikationsmethoden für die Transferdateien implementiert,
  - Signer**, eine Unterklasse von Operator, die Methoden zur Signaturberechnung für die Transferdateien implementiert,
  - CrtParser**, eine Unterklasse von Operator, die Methoden zum Parsen einer Crt.xml Transferdatei implementiert und

**CrlCreator**, eine Unterklasse von **Operator**, die Methoden zum Erzeugen einer **Crl.xml** Transferdatei implementiert.

5. Das Paket **de.tud.cdc.flexiTrust.ri.db** besteht aus den Klassen:

**DBRevocationInfo**, die von dem Revokation-Servlet benutzt wird und Methoden zur Überprüfung der Revokationsdaten implementiert,

**DB**, die alle Datenbankzugriffsmethoden implementiert.

6. Das Paket **de.tud.cdc.flexiTrust.ri.ldap** enthält die Klasse **LdapCertificateInfo**, die von dem Revokation-Servlet benutzt wird und Methoden zur Überprüfung der Revokationsdaten implementiert.

**Revokationsmethoden:** Für die Implementation von verschiedenen Revokationsmethoden benötigt man in der Praxis verschiedene CRLs. Die Klasse **de.tud.cdc.flexiTrust.ri.crl.IndirectCrl** implementiert Methoden für drei Basistypen: `void createBaseCRL()`, `void createDeltaCRL()` und `void createCompleteCRL(X509Crl deltaCRL)`. Dabei sind zur Erzeugung jedes CRL-Typs folgende Schritte vorzunehmen:

1. Als erstes muss eine `java.cert.X509Crl`-Instanz erzeugt werden:

```
/* creating a new crl */
X509Crl crl = new X509Crl(new Name("CN=riIssuer, O=test, C=de"), date);
```

2. Bei der erzeugten CRL werden **this-** und **nextUpdate** Felder (s. Kapitel 2.3) gesetzt:

```
crl.setThisUpdate(date);
crl.setNextUpdate(nextUpdate);
```

3. Das Signatur-Algorithmus wird aus der Konfigurationdatei entnommen und der CRL zugefügt:

```
String crlSignatureAlgorithm = parser.getCrlSignatureAlgorithm();
codec.x509.AlgorithmIdentifier mySigAlgId =
    new codec.x509.AlgorithmIdentifier(crlSignatureAlgorithm);
crl.setSignatureAlgorithm(mySigAlgId);
```

4. Die CRL wird um die CRL Entries erweitert, wobei jede solche Entry einem zu revozierenden Zertifikaten entspricht:

```

crlEntry = new CRLEntry();
crlEntry.setSerialNumber(new BigInteger(certificateNumber));
crlEntry.setRevocationDate(date);
/* add crlEntry to the crl */
crl.addEntry(crlEntry);

```

5. Extensions werden zugefügt: Alle drei CRLtypen sind indirekte CRLs (s. Kapitel 3.2) und enthalten somit eine entsprechende Extension :

```

/* add Issuing Distribution Point Extension */
IssuingDistPoint idp = new IssuingDistPoint();
idp.addIndirectCrl(true);
crl.addExtension(idp);

```

Bei der Implementierung der Delta CRL Methode müssen alle CRLs die CRLNumber Extension enthalten (s. Kapitel 2.3):

```

CRLNumber crlNumber = new CRLNumber(Integer.toString(newBaseCRLNumber));
crl.addExtension(crlNumber);

```

Die Delta CRLs unterscheiden sich von allen anderen CRLtypen durch die DeltaCRLIndicator Extension:

```

/* adds DeltaCrlIndicator to deltaCrl */
DeltaCRLIndicator dci =
    new DeltaCRLIndicator( Integer.toString(this.crlSerialNumber));
deltaCrl.addExtension(dci);

```

6. Nach dem kompletten Erzeugen einer CRL wird diese von der Revokation-Komponente signiert:

```

/* sign crl */
sigEng = Signature.getInstance(crlSignatureAlgorithm);
sigEng.initSign(privateKey);
sigEng.update(crl.getTBSCertList(mySigAlgId));
crl.setSignature(sigEng.sign());

```

7. Sofort nach dem Signieren der CRL ist eine Verifikation erforderlich:

```

/* verify crl */
crl.verify(publicKey);

```

### 5.2.3 Benutzte Bibliotheken

Die Revokation-Komponente greift auf eine Reihe von Java-Klassenbibliotheken zurück, die Basisfunktionalität aus verschiedenen Bereichen implementieren. Alle diese Bibliotheken werden in Form von JAR-Dateien bei der Installation der RI-Software mit installiert.

**Jave-Plattform** RI wurde in der Version JDK 1.4.1 für die Java2 Standard Edition entwickelt.

**Ant** Während der Entwicklung sowie bei der Installation der Anwendungen wurde Ant (entwickelt vom Jakarta-Projekt der Apache Software Foundation) eingesetzt.

**Tomcat** Für die Installation und Ausführen von Java-Servlets wird Tomcat Servlet Container (entwickelt vom Jakarta-Projekt der Apache Software Foundation) eingesetzt.

**ASN.1-Codec** Zum Umgang mit den ASN.1-Datenstrukturen wird das Codec Paket des Fraunhofer-Instituts verwendet, das ebenfalls vom Fachgebiet gepflegt wird.

**FlexiProvider** Die RI benutzt die kryptographischen Algorithmen der auf dem Fachgebiet entwickelte auf der Java Cryptography Architecture (JCA/JCE) basierte Bibliothek für kryptographische Algorithmen FlexiProvider.

**LDAP** Der LDAP-Zugriff geschieht mittels im Rahmen des FlexiTrust-Projektes entwickelte Bibliothek ldap.

**Datenbank** Als Datenbank kommt MySQL zum Einsatz, so dass ein JDBC-Treiber hierfür benötigt wird.

**Apache XML Security** Zum Berechnen und Verifizieren von Digitalen Signaturen wird XML Security Bibliothek (entwickelt vom XML Security Projekt der Apache Software Foundation) eingesetzt.

## 5.3 Betriebsanleitung

Die RI-Software besteht aus zwei Bibliotheken ri.war, die alle Klassen für die Web-Anwendung beinhaltet und ri.jar, zu der alle Klassen der Revokation-Komponente selbst gehören. Da die Web-Anwendung auf die RI-Verzeichnisse greift, die Installation der beiden Komponenten erforderlich.

### 5.3.1 Installation der Web-Applikation

Die Web-Anwendung kann auf zwei Wege geliefert werden:

Als bereits installierte Anwendung. Es wird Tomcat Servlet Container mitgeliefert. Die Anwendung-Software ist dort bereits installiert.

Als ri.war-Datei. Diese muss in das Verzeichnis

```
$Tomcat_Home/webapps
```

vom Tomcat Servlet Container installiert werden. Beim Starten von Tomcat wird diese Datei automatisch entpackt und installiert.

**Verzeichnis-Struktur:** Nach dem Entpacken der ri.war Datei werden im Tomcat folgende RI-Verzeichnisse installiert:

```
$Tomcat_Home/webapps/RI
```

Das Root-Verzeichnis, wo u.a. das Java-Servlet und das HTML-Formular gespeichert sind.

```
$Tomcat_Home/webapps/RI/images
```

Hier werden alle von HTML-Dateien benutzte .jpg-Dateien (Bilder) gespeichert.

```
$Tomcat_Home/webapps/RI/etc
```

Dieses Verzeichnis speichert die Konfigurationsdatei config.xml und alle von dem Servlet benutzten KeyStores.

```
$Tomcat_Home/webapps/RI/WEB-INF
```

Hier wird die Datei web.xml Web-Application Deployment Descriptor (s. Anhang) gespeichert.

```
$Tomcat_Home/webapps/RI/WEB-INF/classes
```

Dieses Verzeichnis ist bei der Revokation-Anwendung leer, da alle RI-Klassen in der ri.jar gespeichert sind. Das Verzeichnis kann allerdings die von der Anwendung benutzten Klassen speichern und bei jeder Web-Applikation erforderlich.

```
$Tomcat_Home/webapps/RI/WEB-INF/lib
```

Hier werden alle Bibliotheken gespeichert, die das Revokation-Servlet zur Bearbeitung von Revokation-Anfragen benötigt.

**Konfiguration:** Zur Installation und Starten von der Web-Anwendung sind keine Konfigurationsschritte erforderlich.

**Starten:** Zum Starten des Servlets muss die Tomcat-Software gestartet werden. Im Anschluss ist das Revokation-Formular mittels eines Web-Browsers unter

`http://Server-IP:8080/RI`

abzurufen.

**Stoppen:** Nach dem Stoppen vom Tomcat kann auf die Web-Anwendung nicht mehr zugegriffen werden.

### 5.3.2 Installation der Revokation-Komponente

Die Installation der RI erfolgt durch die Eingabe in der Kommandozeile von **ant install**. Dabei wird das Verzeichnis flexiTrust-RI angelegt, das u.a. die Skripte zum Starten und Stoppen des RI-Dämons enthält.

**Konfiguration:** Bei der Installation von RI-Software sind keine Konfigurationsschritte erforderlich. Die Konfigurationsdatei (s. Anhang) `config.xml` beinhaltet folgende Informationen:

**KeyStore parameters** Dazu gehören alle Daten, die man beim Zugriff auf ein KeyStore benötigt (z.B. `CertificateAlias`).

**CRL parameters** Die wichtigsten davon sind `crlPeriod`, `deltaCrlPeriod`, `crlSignatureAlgorithm`.

**Crl.xml parameters** Hier werden alle Informationen zum Erzeugen eines X509Crl.xml Produktes (s. Kapitel 4) gespeichert (z.B. `SignaturAlgorithm`).

**RI backend Database parameters** Dazu gehören alle Daten (z.B. IP-Adresse, Datenbank-Treiber) für die Verbindung zu der internen Datenbank (s. Datenbankschema im Anhang)

Alle in der Konfigurationsdatei enthaltenen Daten können statisch vor dem Programmstart geändert werden.

Die Reihenfolge der auszuführenden Operationen wird vom RI-Dämon aus der Datei `operatoren.xml` entnommen.

**Starten:** Zum Starten der Revokation-Komponente ist unter Windows das Skript **run.bat** und unter Solaris/Linux **run.sh** auszuführen.

**Stoppen:** Mit Hilfe der Skripten **stop.bat** unter Windows und **stop.sh** unter Solaris/Linux wird das RI-Dämonprozess beendet.

## 6 Ausblick

Diese Diplomarbeit hat sich in der ersten Linie mit den verschiedenen Revokationsmethoden beschäftigt. Die drei davon kommen bei der Implementierung von Revokation-Komponente zum Einsatz. Die Wahl der Revokationstechnik hängt von der konkreten Revokationsituation einer konkreten PKI ab und ist dem Administrator dieser PKI überlassen. Je mehr Methoden stehen der Revokation-Komponente zur Verfügung, desto besser und flexibler kann sich diese an Bedürfnisse der PKI-Teilnehmer anpassen. Abgesehen von Entwicklung und Implementierung neuer Revokationstechniken, sind noch an der Implementierung der RI einige Dinge zu verbessern:

- Die RunTime-Konfiguration ist erforderlich. Der PKI-Administrator soll in der Lage sein die Revokation-Methode zu wechseln, ohne RI-Dämon zu runterfahren.
- Die Speicherung der Anfrage-Produkten soll zusätzlich zu der DB mit Hilfe einer Hash-Tabelle erfolgen. Dabei wird das Sicherheitsgrad der RI erhöht. Man denke daran: es darf keiner der Revokationsaufträge verloren gehen.
- Die Operatoren-Menge kann um neue Operatoren erweitert werden. Die einzelnen Operatoren können in mehrere kleinere Operationen zerlegt werden: Je kleiner die Granularität einer Operation, desto flexibler diese beim Einsatz in der Praxis ist.

Als eine weitere Aufgabe auf diesem Gebiet kann die Implementierung der neuen Version des XML basierten Transferprotokolles sowie die Integrierung dieses in die FlexiTrust-Software gesehen werden.

Diese Aufgabenliste hat keinen Anspruch auf Vollständigkeit, es werden sich sicherlich noch andere Aufgaben zu einem späteren Zeitpunkt ergeben, wobei die in dieser Arbeit entstandene Version der Revokation-Komponente als Basis für die zukünftigen Entwicklungen gesehen werden kann.

## 7 Anhang

### Anhang A

#### Datenbankschema

```
-----
-- Backend tables for Revocation Authority
-----

use anna_flexitrust;
DROP TABLE IF EXISTS baseCRL;
CREATE TABLE baseCRL(
    serialNumber varchar(30) NOT NULL,
    thisUpdate datetime NOT NULL,
    nextUpdate datetime NOT NULL,
    archiv varchar(5) NOT NULL,
    PRIMARY KEY (serialNumber),
    FOREIGN KEY (serialNumber) REFERENCES certs (crlNumber)
);

DROP TABLE IF EXISTS deltaCRL;
CREATE TABLE deltaCRL (
    baseCRLNumber varchar(30) NOT NULL,
    deltaCRLNumber varchar(30) NOT NULL,
    archiv varchar(5) NOT NULL,
    PRIMARY KEY (deltaCRLNumber),
    UNIQUE(deltaCRLNumber),
    FOREIGN KEY (baseCRLNumber) REFERENCES baseCRL(serialNumber),
    FOREIGN KEY (deltaCRLNumber) REFERENCES certs(crlNumber)
);

DROP TABLE IF EXISTS certs;
CREATE TABLE certs (
    crlNumber varchar(30) NOT NULL,
    revocationdate datetime NOT NULL,
    IssuerDN varchar(255) NOT NULL,
    serialNumber varchar(30) NOT NULL ,
    revocationReason varchar(2)
```

```
);
```

```
DROP TABLE IF EXISTS ri_products;  
CREATE TABLE ri_products (  
    serialNumber varchar(30) NOT NULL,  
    IssuerDN varchar(255) NOT NULL,  
    certificateStatus varchar(10),  
    revocationReason varchar(2),  
    revocationdate datetime NOT NULL
```

```
);
```

## Anhang B

### Web-Application Deployment Descriptor

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>

  <servlet>
    <servlet-name>revocation</servlet-name>
    <display-name>revocation</display-name>
    <description>no description</description>
    <jsp-file>/CertRevocation.jsp</jsp-file>
    <!-- Path for config.xml -->
    <init-param>
      <param-name>config</param-name>
      <param-value>../webapps/ri/etc/config.xml</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>revocation</servlet-name>
    <url-pattern>/CertRevocation.jsp</url-pattern>
  </servlet-mapping>
  <!-- If URL gives a directory but no filename, try index.html -->

  <welcome-file-list>
    <welcome-file>Revocation.html</welcome-file>
  </welcome-file-list>

</web-app>
```

## Anhang C

### Beispiel einer Konfiguration-Datei für die RI

```
<?xml version="1.0"?>

<Revocator>
  <!-- ***** Database parameters -->
  <param-set name="database">
    <param key="serverName" value="jdbc:mysql://130.83.23.182/anna_flexitrust"/>
    <param key="userName" value="root" />
    <param key="userPass" value="HDjwhdwi" />
    <param key="driverName" value="org.gjt.mm.mysql.Driver"/>
    <param key="databaseName" value="anna_flexitrust" />
  </param-set>

  <!-- ***** LDAP parameters -->
  <param-set value="ldap">
    <param key="IP" value="cdc-ws11.cdc.informatik.tu-darmstadt.de" />
    <param key="PORT" value="8389" />
    <param key="ROOT" value="dc=cdc-ws11,dc=cdc,dc=informatik,
                          dc=tu-darmstadt,dc=de" />
    <param key="ldapKeystore" value="../webapps/ri/WEB-INF/lib/ldapservers.ks" />
  </param-set>

  <!-- ***** Revocation Instance parameters ***-->
  <param-set name="revocator">
    <param key="SearchLocation" value="db" />
    <param key="crlDestination" value="../webapps/ri/crls/transfer.crl" />
  </param-set>

  <!-- ***** Signature parameters ***** -->
  <param-set name="signature">
    <param key="signatureAlgorithm" value="SHA1withRSA" />
    <param key="provider" value="FlexiCore" />
    <param key="keystoreFile" value="../webapps/ri/keystore/Firstkeystore.ks"/>
    <param key="certificateAlias" value="crltest" />
    <param key="keystorePass" value="bochka" />
  </param-set>

  <!-- ***** Transfer Protokol parameters *** -->
  <param-set name="p7">
```

```

    <param key="CrtProfileDirectory" value="../flexiTrust-RI/crt/" />
    <param key="CrlProfileDirectory" value="../flexiTrust-RI/crls"/>
    <param key="DSAKeystore" value="../webapps/ri/etc/DSAKeyStore.ks" />
    <param key="certificateAlias" value="dsasign"/>
    <param key="privateKeyPass" value="urodak"/>
    <param key="RSAKeystore" value="../webapps/ri/etc/RSAKeyStore.ks" />
    <param key="certificateAlias" value="rsasign"/>
    <param key="privateKeyPass" value="dunduk"/>
    <param key="signatureAlgorithm" value="1" />
    <param key="digestAlgorithm" value="1" />
</param-set>

<!-- **** RI Demon KeyStore parameters **** -->
<param-set name="ri">
    <param key="DSAKeystore" value="../flexiTrust-RI/etc/DSAKeyStore.ks" />
    <param key="certificateAlias" value="dsasign"/>
    <param key="privateKeyPass" value="urodak"/>
    <param key="RSAKeystore" value="../flexiTrust-RI/etc/RSAKeyStore.ks" />
    <param key="certificateAlias" value="rsasign"/>
    <param key="privateKeyPass" value="dunduk"/>
    <param key="signatureAlgorithm" value="1" />
    <param key="digestAlgorithm" value="1" />
</param-set>

<!-- **** CRL parameters *** -->
<param-set name="crl">
    <param key="crlPeriod" value="30" />
    <param key="deltaCrlPeriod" value="1" />
    <param key="crlSignatureAlgorithm" value="SHA1withRSA" />
    <param key="crlFile" value="../flexiTrust-RI/crls/" />
    <param key="crlKeyStore" value="../flexiTrust-RI/etc/Firstkeystore.ks" />
    <param key="crlCertificateAlias" value="crltest" />
    <param key="crlPrivateKeyPass" value="bochka" />
</param-set>

<!-- **** CRL.xml parameters **** -->
<param-set name="cma">
    <param key="productLocation" value="../flexiTrust-RI/xml/" />
    <param key="productSignatureAlgorithm" value="1" />
    <param key="productDigestAlgorithm" value="1" />
    <param key="productDSAKeystore" value="../flexiTrust-RI/etc/DSAKeyStore.ks"/>
    <param key="productCertificateAlias" value="dsasign"/>
    <param key="productPrivateKeyPass" value="urodak"/>

```

```

    <param key="productRSAKeystore"
        value="../flexiTrust-RI/etc/RSAKeyStore.ks"/>
    <param key="productCertificateAlias" value="rsassign"/>
    <param key="productPrivateKeyPass" value="dunduk"/>
</param-set>

<!-- **** RI backend Database parameters **** -->
<param-set name="database">
    <param key="serverName"
        value="jdbc:mysql://130.83.23.182/anna_flexitrust"/>
    <param key="userName" value="root" />
    <param key="userPass" value="HDjwhdwi" />
    <param key="driverName" value="org.gjt.mm.mysql.Driver"/>
    <param key="databaseName" value="anna_flexitrust" />
</param-set>

<!-- **** RI Timer parameters **** -->
<param-set name="timer">
    <param key="crlPeriod" value="5000" />
</param-set>
</Revocator>

```

# Anhang D

## Struktur des Anfrage-Produktes

```
<!ELEMENT Request ((Profile|X509CrtProfile|X509CrlProfile|
                    X509PKCS11Profile|X509PKCS12Profile)+,Signature?)>
<!ELEMENT Profile (Item+, Parameter?, Signature?)>
<!ATTLIST Profile name CDATA #REQUIRED>
<!ELEMENT Item (Value+)>
<!ATTLIST Item name CDATA #REQUIRED>
<!ELEMENT Value (#PCDATA)>
<!ELEMENT X509CrtProfile (revocPass, certificate, Item?, Parameter?,
                          Signature?)>
<!ELEMENT revocPass (#PCDATA)>
<!ELEMENT certificate (#PCDATA)>
<!ELEMENT X509CrlProfile (crlEntry+, issuer, revocPass, x509Crl, Item?,
                          Parameter?, Signature?)>
<!ELEMENT crlEntry (#PCDATA)>
<!ELEMENT issuer (#PCDATA)>
<!ELEMENT x509Crl (#PCDATA)>
<!ELEMENT X509PKCS11Profile (revocPass, certificate, Item?, Parameter?,
                              Signature?)>
<!ELEMENT certificate (#PCDATA)>
<!ELEMENT X509PKCS12Profile (PFX, revocPass, certificate, p12Pass, Item?,
                              Parameter?, Signature?)>
<!ELEMENT PFX (#PCDATA)>
<!ELEMENT p12Pass (#PCDATA)>
<!ELEMENT Parameter (Value)>
<!ATTLIST Parameter name CDATA #REQUIRED>
<!ENTITY % Signature
        SYSTEM 'http://www.w3.org/TR/xmldsig-core/xmldsig-core-schema.dtd'>
&Signature;
```

## References

- [1] Buchmann, J.: *Einführung in die Kryptographie*. Springer, zweite, erweiterte Auflage, 2001.
- [2] Housley, R., Polk, T.: *Plannig for PKI*. Wiley, 2001.
- [3] Planz, T.: *Entwurf und Implementierung einer Infrastrukturkomponente für ein Java-basiertes Trustcenter*. Diplomarbeit, Technische Universität Darmstadt, Januar 2002.
- [4] Buchmann, J., Ruppert, M., Tak, M.: *FlexiPKI - Realisierung einer flexiblen Public-Key-Infrastruktur*. Technische Universität Darmstadt, Fachbereich Informatik.
- [5] *Information technology - Open Systems Interconnection - The Directory: Authentication framework*. ITU-T Recommendation X.509, August 1997.
- [6] Adams, C., Zuccherato, R.: *A general, flexible Approach to Certificate Revocation*. Entrust, Juni 1998.
- [7] Cooper, D.: *A Model of certificate Revocation*. Computer Security Division, National Institute of Standarts and Technology, Gaithersburg, December 1999.
- [8] Cooper, D.: *A More Efficient Use of Delta-CRLs*. Computer Security Division, National Institute of Standarts and Technology, Gaithersburg, May 2000.
- [9] *Public-Key Cryptography Standards*. RSA Laboratories, <http://www.rsa.com/rsalabs/pkcs/>.
- [10] Box, D., Skonnard, A., Lam, J.: *Essential XML*. Addison-Wesley, 2001.
- [11] Gamma, E., Helm, R., Johnson, R. und Vlissides, J.: *Entwurfsmuster*. Addison-Wesley, 1996.
- [12] Rivest, R.: *Can we eliminate Certificate Revocation Lists?* MIT Laboratory for Computer Science, Cambridge.
- [13] Diffie, W., Hellman, M.: *New Directions in Cryptography*. IEEE Transactions on Information Theory, November 1976.
- [14] Kohnfelder, L.: *Towards a Practical Public-key Cryptosystem*. MIT Laboratory for Computer Science, 1978.

- [15] Rivest, R., Shamir, A., Adleman, L.: *A method for obtaining digital signatures and public-key cryptosystems*. Communications of the ACM, Februar 1978.
- [16] RFC 3280: *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*