Diplomarbeit

Analyse, Verifikation und Realisierung kryptographischer Protokolle für flexible Zugriffe auf medizinische Datenbanken

Universität Karlsruhe
Fakultät für Informatik
Institut für Algorithmen
und Kognitive Systeme

Erik-Oliver Blaß

Betreuer: Prof. Dr. Thomas Beth
Betreuender Mitarbeiter: Dipl. Inform. Jörg Moldenhauer

8. Oktober 2001

Erklärung

Ich versichere, diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Karlsruhe, den 8. Oktober 2001

(Erik-Oliver Blaß)

Kurzfassung

Der Umgang mit sensiblen Patientendaten stellt hohe Anforderungen an deren Vertraulichkeit und Integrität. Für einen entfernten, elektronischen Zugriff auf solche Daten und deren elektronische Verarbeitung sind besondere Sicherheitsvorkehrungen nötig, um Geheimhaltung und Schutz vor Mißbrauch zu gewährleisten. Ein Problem dabei sind insbesondere die zeitlich veränderlichen Zuständigkeiten und Verantwortlichkeiten für Patientendaten im klinischen Alltag.

In dieser Arbeit werden auf bestehenden kryptographischen Komponenten basierende Protokolle zum Zugriff auf medizinische Daten entwickelt, die diese Anforderungen gezielt umsetzen. Dazu gehören die gesicherte Speicherung der medizinischen Daten in speziell hierfür entwickelten Datenbanken sowie eine funktionsspezifische Rechtverwaltung und Zugangskontrolle mittels sogenannter Rollen.

Die erarbeiteten Protokolle und das Umfeld ihres Einsatzes werden auf verschiedenen Ebenen analysiert und ihre Sicherheit formal bewiesen. Mit einer Implementierung in Java wird die Einsatzfähigkeit der Protokolle gezeigt. Die Verwendung von CORBA-Kommunikationsmechanismen erlaubt die Einbindung in ein verteiltes medizinisches Informationssystem, das vom IAKS im Rahmen des Teilprojekts Q6 des SFB 414 „Informationstechnik in der Medizin“ als Prototopy entwickelt wird.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einleitung

Das Institut für Algorithmen und kognitive Systeme (IAKS [1]) der Universität Karlsruhe arbeitet im Sonderforschungsbereich 414 der Deutschen Forschungsgemeinschaft (DFG) am Datenschutz und der Systemsicherheit in medizinischen Informationssystemen. Ein Szenario hierbei ist die elektronische Verwaltung von Patientendaten, d. h. ein rechnerbasiertes Abspeichern und ein entfernter Zugriff auf diese.

Elektronisch gespeicherte medizinische Daten eines Patienten haben als Ersatz für die klassische Papier-Patientenakte einige Vorteile: Sie bieten neben verwaltungstechnischen oder abrechnungstechnischen Vorzügen eines einzelnen Krankenhauses noch die Möglichkeit des computer-unterstützten, eventuell weltweiten Zugriffs auf komplette Krankengeschichten und somit einen enormen Vorteil für neue Diagnosen und Behandlungen. Denkbar sind außerdem Dinge wie das Anlegen von großen Statistiken bestimmter Krankheitsbilder über einen langen Zeitraum hinweg mit anonymen Patientendaten oder der internationale Austausch und das gegenseitige Helfen bei Problemfällen.

Auf der anderen Seite muß der elektronische Zugriff auf die persönlichen Daten eines Patienten entsprechend gesichert werden. Unberechtigtes Lesen, Ändern oder Löschen einer Akte kann nicht nur die wichtige Intimsphäre des Patienten verletzen, sondern auch fatale Auswirkungen auf Leib und Leben haben. Auch darf der entsprechend autorisierte behandelnde Arzt nur Zugang zu den für ihn relevanten Patienteninformationen haben. Bei einer Grippebehandlung sind mögliche Allergien relevant, ein vorhergehender Aufenthalt in einer Psychiatrie jedoch nicht. Die genauen Rahmenbedingungen und Anforderungen im medizinischen Umfeld sind in Abschnitt 1.2 beschrieben.

1.1 Aufgabenstellung

In dieser Arbeit geht es um die Frage, wie Protokollabläufe für lesende, schreibende und löschende Datenzugriffe auf eine medizinische Datenbank mit sensitiven Daten kryptographisch sicher durchgeführt werden können.

Im Fokus der Arbeit steht die Analyse von solchen Protokollen zum Zugriff auf medizinische Datenbanken im Hinblick auf die noch zu beschreibenden Anforderungen.

Den zweiten Schwerpunkt bildet die Realisierung der Protokolle. Sie sollen die Grundlage für ein prototypisches medizinisches Informationssystem bilden, das derzeit am IAKS im Rahmen des Projekts Q6 des SFB 414 entwickelt wird. Die Implementierung soll in Java erfolgen und Kommunikationsabläufe zwischen beteiligten Protokollinstanzen über CORBA [28] abgewickelt werden.

Beschreibung des Szenarios

Im behandelten Szenario sollen beliebige medizinische Daten, d. h. Daten über Patienten, Krankengeschichten, prinzipiell alles, was in klinischem Umfeld an Daten anfallen kann, elektronisch abgelegt und wieder zugreifbar gemacht werden. Benutzer dieses Systems sind z. B. Ärzte, Schwestern oder auch Patienten selbst.

Zugriffe dieser Benutzer auf den Datenbestand können entweder *statischer* oder *dynamischer* Struktur sein [2]. Bei statischen Systemen sind einzelne Datensätze fest an bestimme Personen gebunden. Nur die Person, die den Datenbestand angelegt und verschlüsselt hat, kann in der Regel darauf zurückgreifen. Die Zugriffsfreigabe auf den angelegten Datenbestand für Dritte stellt insofern ein technisches Problem dar, da nur der Erzeuger der Daten im Besitz des entsprechenden Schlüssels zum Dechiffrieren ist.

Gerade im medizinischen Umfeld treten jedoch häufig Fälle ein, die eine etwas kompliziertere Zuordnung benötigen. Behandelt beispielsweise ein Stationsarzt einen Patienten, wird jedoch am nächsten Tag auf Grund eines Schichtwechsels von einem Kollegen abgelöst, so muß der neue Kollege auch auf die Patientenakte zugreifen können. Der bisherige Arzt darf jedoch die Krankengeschichte des Patienten dann nicht mehr oder nur sehr eingeschränkt weiterverfolgen. Welcher Arzt gerade genau welche Möglichkeiten des Zugriffs hat, wird durch verschiedene Regeln entschieden, die abhängig vom Dienstplan, der aktuellen Situation, z. B. einem Notfall, und der eigentlichen Funktion des Arztes ausgewertet werden müssen. Der Zugriff auf die persönlichen Daten geschieht somit dynamisch - die häufig wechselnden Zuständigkeiten sind durch die statischen Strukturen nicht zu erfassen.

Ein grundlegendes Prinzip des Systementwurfs ist daher zunächst die Vergabe von *Rollen* an Benutzer, bzw. das Auftreten der Benutzer innerhalb einer Rolle.

Im System selbst handeln verschiedene Benutzer, d. h. im Sinne von verschiedenen Individuen, z. B. „Christian Müller“ oder „Hans Schmidt“, idealerweise nur innerhalb einer Rolle, so z. B. in den Rollen „Chefarzt“ oder „Stationsarzt“. Benutzer treten im System nicht mehr als Personen oder Individuen auf, sondern funktionsorientiert. Geheimnisse wie Schlüssel zum Entschlüsseln von Daten können damit bestimmten Funktionen zugeordnet werden, etwa je ein Schlüssel für die Rollen „Chefarzt“ und „Stationsarzt“. Ein Benutzer, der sich am System anmeldet, bekommt dynamisch eine Rolle zugeordnet oder bewirbt sich bei einem zentralen *Rollenserver* um eine Rolle. Dieser Server entscheidet nach bestimmten Regeln, wie den Vorgaben eines vorher erstellen Dienstplans, oder nach Maßnahmen der Notfallbehandlung, ob ein Benutzer zu diesem Zeitpunkt eine Rolle einnimmt. Danach tritt der Benutzer im System nur noch unter dieser Rolle auf. Er bekommt vom Rollenserver die Schlüssel zum Entschlüsseln der entsprechenden Daten zugeordnet. Alle Transaktionen (Zugriffe) auf den Datenbestand geschehen, wie oben erwähnt, im Namen dieser Rolle. Der Benutzer kreiert sich dabei einen sogenannten *Rollenproxy*, der für ihn den Datenzugriff stellvertretend im Namen seiner Rolle, sowie sämtlichen kryptographischen Protokolle transparent durchführt. Zu einem späteren Zeitpunkt kann der Nutzer diese Rolle, und damit das Geheimnis, freiwillig wieder abgeben oder wird dazu durch ein Regelsystem gezwungen.

Das Schema dieser Zugriffsregelung ist in Abbildung 1.1 dargestellt. Von fundamentaler Bedeutung für dieses Schema ist neben dem Rollenserver auch der *Rechteserver*. Alle Transaktionen werden von dieser Entität aus genehmigt respektive abgelehnt. Will ein Benutzer bzw. die ihn repräsentierende Rolle durch den Rollenproxy eine Transaktion durchführen, so benötigt er dazu zunächst eine entsprechende Autorisierung durch den Rechteserver. Anhand dieser Autorisierung kann später die angesprochene Datenbank verifizieren, ob der aktuelle Zugriff der Rolle legal ist. In Abbildung 1.1 ist bereits die Position der zu analysierenden Protokolle innerhalb des Systems zu erkennen. Sie werden zwischen Benutzern/Rollen und den Datenbanken vermitteln, um Anfragen sicher durchzuführen. Genauso werden die gestrichelten Verbindung zu sichern sein. Dies wird in Kapitel 3 ausführlich erklärt.

Das Auftreten von Benutzern als Rollen im System steuert bereits in gewissem Maße auch zur Sicherheit bei. Es müssen keine Rechte bestimmt oder Sicherheitsvorkehrungen mehr für jeden einzelnen Benutzer getroffen werden, sondern nur noch für Rollen. Dies hilft bei Änderungen, die am Rechtesystem durchgeführt werden müßten, wenn neue Benutzer dem System hinzugefügt werden oder alte Benutzer das System verlassen, siehe hierzu [21].

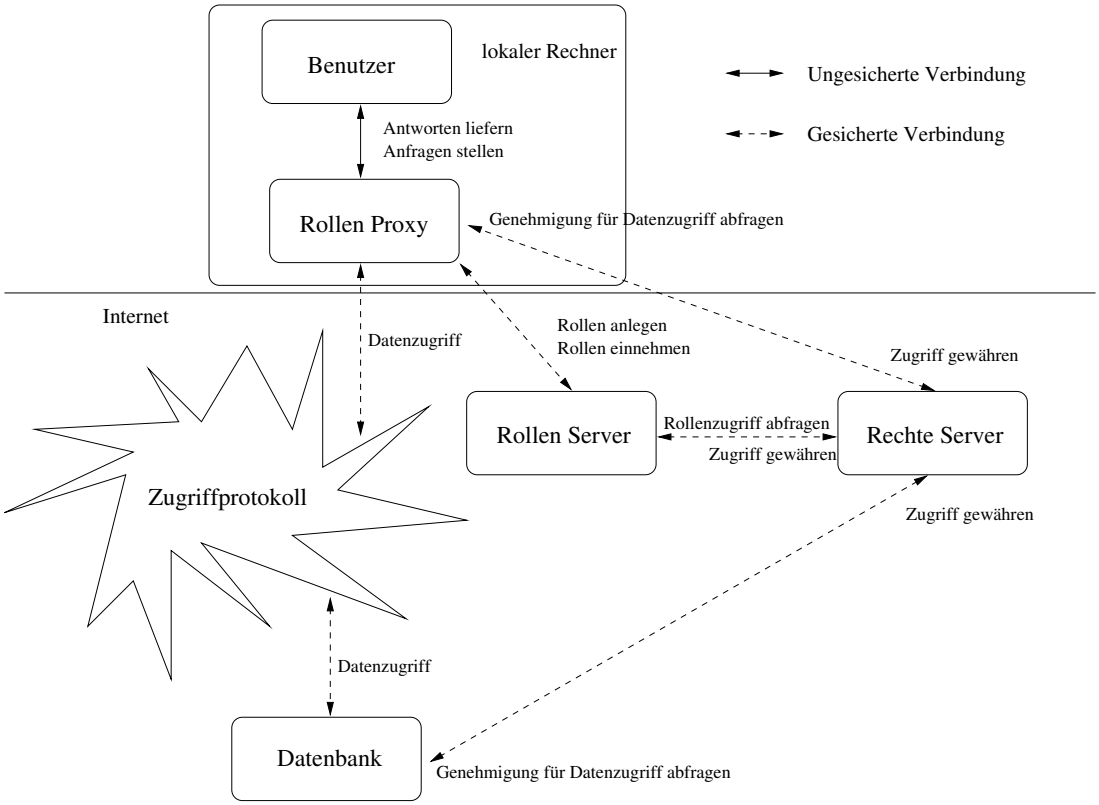

Abbildung 1.1: Schema des Datenzugriffs im Szenario

1.2 Sicherheitsrelevante Anforderungen

Die Arbeiten [2] und [21] geben eine Übersicht über die sicherheitsrelevanten Anforderungen in medizinischen Informationssystemen.

Zunächst gelten für alle gespeicherten oder übertragenen (Patienten-)Daten die vier klassischen kryptographischen Anforderungen, wie sie allgemein für alle sicheren Systeme vorausgesetzt werden:

1. **Vertraulichkeit**
   Diese Anforderung ist die offensichtlichste der vier. Es geht um die Geheimhaltung der Informationen gegenüber unbefugtem Mitlesen während einer Datenübertragung oder beim unberechtigten Zugriff direkt auf den Speicherort der Daten. In Falle klinischer Informationssysteme muß beispielsweise sichergestellt sein, daß nur der entsprechend autorisierte Arzt die angeforderten Informationen über einen Patienten lesen darf. Dies macht nicht nur eine Form von Rechtesystem erforderlich, z. B. einen Rechteserver, der verwaltet, welcher Arzt genau wie zugreifen darf, sondern auch die Notwendigkeit der Verschlüsselung/Chiffrierung der eigentlichen Daten. Ziel dieser

Vertraulichkeitsanforderung ist es, daß ein potentieller Angreifer aus dem zufälligen oder unberechtigten Mitlesen der chiffrierten Nutzdaten keinerlei Rückschlüsse auf die eigentlichen Nutzdaten ziehen kann [31].

2. **Integrität**
   Es muß sichergestellt sein, daß Daten unverändert vom Sender zum Empfänger gelangen. Die verschlüsselten gesendeten Daten müssen gleich den entschlüsselten empfangenen Daten sein. Auf keinen Fall darf durch unabsichtliche Übertragungsfehler oder gar absichtliche Manipulation während der Übertragung der Inhalt der Daten unbemerkt verändert werden. Sind Daten verändert worden, so muß dies den Benutzern sofort ersichtlich sein. Es ist von außerordentlicher Wichtigkeit, daß nicht autorisierte Personen keine Patientendaten verändern können.

3. **Authentizität**
   Im Vordergrund der Authentizität steht die Frage nach dem Nachweis der Echtheit bzw. der Herkunft von Daten. Benutzer A bzw. Rolle A muß seine Identität und seine Daten über ein Verfahren authentifizieren, so daß später andere Benutzer überprüfen können, daß diese Daten wirklich genau von A stammen. Im klassischen Briefverkehr erkennt der Empfänger eines Briefes z. B. anhand der Unterschrift, daß dieser Brief vom Absender stammt. Bei elektronischer Datenübertragung gibt es ähnliche Mechanismen, digitale Signaturen (siehe Abschnitt 2.2), die - unter bestimmten Voraussetzungen - genauso arbeiten, wie die handschriftlichen. Änderungen an Patientendaten dürfen nicht nur von ganz bestimmten Benutzern gemacht werden, sondern es muß später ersichtlich sein, wer die Änderungen gemacht hat.

4. **Verbindlichkeit**
   Ein weiteres Problem ist die Verbindlichkeit von Daten. Hat ein Sender A Daten an Empfänger B gesendet, so ist dies für A verbindlich. A kann danach nicht mehr abstreiten (*non repudiation*), diese Daten gesendet zu haben und muß für diese Daten haften. Verbindlichkeit und Authentizität hängen miteinander zusammen, sind jedoch nicht äquivalent. Kennt B die Unterschrift von A, so erkennt B zwar ein von A signiertes Dokument als authentisch an, kann jedoch nicht gegenüber einem dritten C beweisen, daß dieses Dokument von A stammt - außer C kennt ebenso die Unterschrift von A. A kann gegenüber C leugnen, dieses Dokument je unterschrieben zu haben.

Spezielle Anforderungen im medizinischen Bereich

Speziell für das in dieser Arbeit untersuchte medizinisch/klinische Umfeld gelten zusätzliche Anforderungen, die berücksichtigt werden müssen:

1. **Unbeobachtbarkeit**
   Im geplanten System werden mehrere Computer über wahrscheinlich abhörbaren Leitungen viele Daten übertragen. Es soll einem Angreifer nicht möglich sein, aus dem Fluß der Daten irgendwelche Rückschlüsse auf deren Inhalt zu ziehen. Greift der Rechner eines Arztes auf eine Datenbank zu, so darf nicht klar sein, ob es sich hierbei um einen Lese- oder Schreibzugriff handelt, sondern nur, daß es sich um einen Zugriff handelt. Auch das mehrmalige Zugreifen auf ein und denselben Datensatz darf nach außen hin nicht als solches erkannt werden.

2. **Nichtverfolgbarkeit**
   Außerdem existiert das Bedürfnis nach Nichtverfolgbarkeit des Datenflusses. Es muß schwierig sein, zurück zu verfolgen, wer wann welche Zugriffe getätigt hat.

3. **Anonymisierung und Pseudonymisierung**
   Unter Umständen sollen die Namen der Patienten pseudonymisiert werden, beispielsweise zum Anlegen von Statistiken. Hier werden die Daten des Patienten benötigt, nicht jedoch der Name. Er wird hinter einem Pseudonym verborgen. Im Gegensatz zur kompletten Anonymisierung ist die Pseudonymisierung gegebenenfalls jedoch wieder umkehrbar. Gewinnt ein Arzt aus einer pseudonymisierten Akte zufällig eine neue bedeutende Erkenntnis zum Wohl des Patienten, so muß aus dem Pseudonym der richtige Name abgeleitet werden können.

4. **Erhalt der Sicherheit bei Manipulation einzelner Server und Stationen**
   Wie in Abbildung 1.1 dargestellt, wird der Datenzugriff über mehrere *Stationen* verwirklicht. Eine Station kann ein eigener Computer sein, es können aber auch mehrere Stationen auf einem Computer arbeiten. Stationen sind z. B. die Datenbank, der Rollenserver, der Regelserver etc. Problematisch ist, wenn eine solche Station von einem Angreifer korrumpiert wird. Beispielsweise könnte der Rollenserver von einem Eindringling unter seine Kontrolle gebracht werden. Eine wichtige grundsätzliche Anforderung ist, daß das gesamte Zugriffs-System solange sicher gegen einen möglichen Mißbrauch bleiben soll, solange nur maximal eine einzelne Station korrumpiert wird. Ist also nur der Rollenserver manipuliert, so sind die geheimen Patientendaten noch nicht verloren. Bei einem Zugriff auf eine korrumpierte Station muß das System einen Fehler erkennen.

Kapitel 2

Kryptographie

Dieses Kapitel soll zunächst ein kurze Einführung in grundlegende kryptographische Techniken, wie symmetrische und asymmetrische Chiffren, Zufallszahlen, sowie Signaturen und Zertifikate, die in Kapitel 3 Verwendung finden und die der Klarheit der Analyse dienen. Auch die Verfahren wie Shared-Secrets oder kommutative Chiffren, auf denen die Sicherheit der Protokolle basieren, werden hier zum besseren Verständnis von Kapitel 3 bereits erklärt. Schließlich wird die formale Protokoll-Analysemethode *Logic of Authentication* beschrieben

2.1 Chiffren

Das Verschlüsseln bzw. Chiffrieren von Daten dient in erster Linie der Vertraulichkeit, also dem Schutz vor unberechtigtem Mitlesen von Geheimnissen. Jedes Chiffriersystem arbeitet nach folgendem Prinzip [39]:

- Gegeben sei ein Klartext M, also die zu verschlüsselnden Daten,
- eine Verschlüsselungsfunktion E, die aus dem Klartext M durch Anwenden von E auf M ein Chiffrat $C = E(M)$ produziert,
- eine Entschlüsselungsfunktion D, die aus dem Chiffrat C wieder den Klartext $M = D(C)$ errechnet. Natürlich sollte $D(E(M)) = M$ gelten.
- Sinn und Zweck einer Verschlüsselung ist, daß es unmöglich bzw. sehr schwierig und aufwendig ist, aus der Kenntnis von Chiffrat C zurück auf Klartext M zu schließen. Das Berechnen von $E(M)$ sowie $D(C)$ ist allerdings mit geringem Aufwand verbunden.

Da bei dieser Form von Verschlüsselungssystemen die Sicherheit von der Geheimhaltung von E und D abhängt[1], sind die Funktionen E und D meistens

[1] ist einem Angreifer D bekannt, so kann er $M = D(C)$ berechnen

über (mindestens) einen zusätzlichen Parameter k parametrisiert. k ist der geheime Schlüssel des Chiffriersystems. Es sollte gelten $k_1 \neq k_2 \rightarrow E_{k_1}(M) \neq E_{k_2}(M)$. Dies hat zur Konsequenz, daß mehrere Benutzer dasselbe öffentlich bekannte Chiffriersystem benutzen können, ohne gegenseitig vertrauliche Daten mitzulesen, solange sie nicht im Besitz des geheimen Schlüssels sind. Die Sicherheit einer solchen parametrisierten Chiffre hängt dann nur vom Schlüssel und nicht von der Geheimhaltung des Chiffriersystems ab, es sei denn, die Chiffre an sich hätte eine ausnutzbare Schwäche.

Je nach Schlüsseltyp unterscheidet man Chiffren in

- **Symmetrische Chiffren**
  Hier gibt es, wie oben bereits beschrieben, nur einen Schlüssel k zum Parametrisieren von E und D. Es gilt $D_k(E_k(M)) = M$ und $E_k(D_k(C)) = C$. Der Schlüssel k ist geheim. Die Sicherheit einer symmetrischen Chiffre hängt von der Geheimhaltung von k ab.

- **Asymmetrische Chiffren**
  Bei asymmetrischen oder auch *Public-Key* Chiffren wird mit zwei zusammengehörenden Schlüsseln gearbeitet. Ein *offentlicher* Schlüssel e zum Verschlüsseln, sowie ein geheimer oder *privater* Schlüssel d zum Entschlüsseln von Daten. Aus dem öffentlichen Schlüssel kann man nicht bzw. nur mit unvertretbar hohem Aufwand den privaten Schlüssel errechnen. Bei asymmetrischen Chiffren gilt: $E_e(M) = C$, $D_d(C) = M$. Der öffentliche Schlüssel heißt öffentlich, weil er ohne Risiko veröffentlicht werden kann - d. h. er steht jedem Interessierten zur Verfügung. Hat A ein Schlüsselpaar aus zusammengehörenden e und d, so kann er e an jeden anderen Nutzer B verteilen. Will B nun A geheime Daten schicken, so chiffriert B $E_e(M) = C$ und schickt C an A, der dies mit seinem geheimen d entschlüsselt.

- **Hybridsysteme**
  Da asymmetrische Chiffren deutlich langsamer sind als symmetrische, werden häufig Hybridsysteme, eine Kombination aus symmetrischen und asymmetrischen Chiffren, eingesetzt. Dabei wird zunächst mit dem Public-Key Algorithmus ein symmetrischer Schlüssel chiffriert, der dann als Parameter eines symmetrischen Algorithmus zur Verschlüsselung der eigentlichen Nutzdaten verwendet wird.

2.1.1 AES

Das National Institute of Standards and Technology (*NIST*) hat auf der Suche nach einem offiziellen Nachfolger des veralteten Data Encryption Standard (DES [23]) mehrere verschiedene Verschlüsselungs-Algorithmen auf ihre Sicherheit, Geschwindigkeit und Flexibilität hin überprüft und schließlich den Algo-

rithmus Rijndael ausgewählt. Eine Beschreibung über den Aufbau und die Arbeitsweise von Rijndael findet sich in [16]. Rijndael wird Ende 2001 der offizielle Nachfolger von DES und heißt dann *Advanced Encryption Standard* (AES [9]). AES wird sowohl im öffentlichen als auch im privaten Datenverkehr zur Verschlüsselung von sensitiven Daten dienen. Der Rijndael/AES Algorithmus verschlüsselt Daten(-blöcke) mit einer Größe von 128 Bits. Er ist ein Beispiel für eine symmetrische Chiffre, denn es gibt nur einen geheimen Schlüssel zur Ver- und Entschlüsselung. Rijndael gilt bisher als sicher: Der effizienteste Weg Rijndael zu brechen ist nach heutigem Ermessen das Erraten des richtigen Schlüssels. Bei einer Schlüssellänge zwischen 128 und 256 Bit gibt es bis zu 2^{256} verschiedene mögliche Schlüssel, die ein Angreifer probieren müßte. Dies ist eine sogenannte *Brute-Force* Attacke, wobei der Angreifer sukzessive alle möglichen Schlüssel ausprobiert.

Zum Vergleich: die Anzahl der Atome in unserer Galaxie wird auf nur 2^{233} geschätzt. Selbst bei einer Schlüssellänge von 128 Bit bräuchte ein Supercomputer, der eine Millionen Schlüssel pro Sekunde probieren kann, immernoch 10^{25} Jahre [4]. AES ist nicht nur sehr sicher, sondern auch portabel, schnell und effektiv. Außerdem kann er effizient auf Chipkarten implementiert werden [5]. Der Rijndael-Code paßt in 52 Bytes. Er ist im Gegensatz zu anderen symmetrischen Algorithmen, wie IDEA [4] flexibel in seiner Schlüssellänge und es existieren keine Patentprobleme. Jeder darf AES benutzen.

Auf Grund dieser Vorteile wird er in dieser Arbeit bei der Implementierung der entwickelten Protokolle als symmetrische Chiffre verwendet.

2.1.2 RSA

Rivest, Shamir und Adlemann haben 1978 den nach ihnen benannten, sehr erfolgreichen Public-Key Algorithmus *RSA* vorgestellt. Aufbau und Arbeitsweise sind beschrieben in [33]. Zum Erzeugen von öffentlichem und privatem Schlüssel werden zwei möglichst gleichgroße Primzahlen p und q gewählt und

$$n = pq$$

berechnet. Der öffentliche Schlüssel e wird zufällig aus dem Bereich

$$3 \leq e < n$$

gewählt und zusammen mit n veröffentlicht. e hat die Eigenschaft:

$$ggT(e, (p-1)(q-1)) = 1.$$

Der private Schlüssel d ist das Inverse zu $e \; mod \; (p-1)(q-1)$ und kann durch den erweiterten euklidischen Algorithmus

$$eggT(e, (p-1)(q-1))$$

ermittelt werden. Für das Chiffrieren eines Klartextes M wird das Chiffrat C durch

$$C = M^e \ mod \ n$$

berechnet. Das Entschlüsseln funktioniert über

$$M = C^d \ mod \ n.$$

Ein Angreifer ist nicht im Besitz von p oder q, sondern kennt nur n. Um von n und e auf d zu schließen, würde er p und q (die Primfaktorzerlegung von n) für den erweiterten Euklid brauchen. Die Sicherheit von RSA basiert nach bisherigen Erkenntnissen auf dem Problem, für sehr große Zahlen eine Primfaktorzerlegung zu finden. Große Zahlen sind Zahlen in der Größenordnung von etwa 2^{2048}. Das ist eine Zahl mit 617 Ziffern. Die Faktorisierung einer solchen Zahl mit den bisher bekannten mathematischen Techniken, wie dem *speziellen Zahlkörpersieb*, dauert selbst auf Großrechnern etwa 10^6 Jahre. Allerdings werden in Zukunft mit Sicherheit neue, schnellere Algorithmen zum Faktorisieren entwickelt werden.

RSA ist weit verbreitet und gut untersucht. Die Schlüssellängen und damit auch die zu faktorisierenden Zahlen sind beliebig vergrößerbar. Da RSA außerdem in der Lage ist, Signaturen (s. Abschnitt 2.2) zu erstellen, wird diese Chiffre bei der Implementierung des Protokolls als Public-Key Algorithmus eingesetzt.

2.2 Digitale Signaturen

Wie in Abschnitt 1.2 bereits angedeutet, werden *digitale Signaturen* zum Sicherstellen von Authentizität in den Protokollen benötigt. Digitale Signaturen werden genauso oder ähnlich wie handschriftliche Unterschriften arbeiten. Sie werden, fest mit einem Dokument verbunden, einem „Verifizierer“ den Beweis liefern, daß ein bestimmter Autor dieses Dokument unterschrieben hat - mit den entsprechenden Konsequenzen (*Authentizität*, siehe 3). Zudem soll das unterschriebene (elektronische) Dokument nach der Unterschrift nicht mehr veränderbar sein (*Integrität*). Außerdem darf ein Autor eine Unterschrift möglichst nicht mehr abstreiten können (*Verbindlichkeit*).

Asymmetrische (Public-Key) Chiffren eignen sich zum Anfertigen und Überprüfen von Signaturen ganz besonders. Wenn Benutzer A ein Dokument M unterschreiben will, und B diese Signatur zu dem Dokument überprüfen will, so geschieht dies folgerndermaßen:

1. Der öffentliche Schlüssel e von A ist jedermann (also auch B) bekannt. A errechnet $C = D_d(M)$ mit Hilfe seines privaten Schlüssels d. C ist die Unterschrift von A zum Dokument M.

2. A sendet das Tupel (M, C) an B, also das Dokument selbst und die dazu passende Unterschrift.

3. B will die Unterschrift überprüfen und testet, ob $E_e(C) = M$. Ist dies der Fall, so kann B sicher sein, daß A Dokument M unterschrieben hat.

Da der eigentliche Signaturvorgang (1.) mit Public-Key Systemen beim Unterschreiben von großen Datenmengen meist sehr langwierig ist, beschränkt man sich oftmals auf die Unterschrift des *Hash-Wertes* eines elektronischen Dokumentes. Der Hash-Wert eines elektronischen Dokumentes ist die Ausgabe einer (kryptographischen) Hash-Funktion H, mit dem Dokument als Eingabe. Eine Hash-Funktion berechnet aus einer beliebig langen Eingabe eine Art Zusammenfassung oder *Fingerabdruck*, d.h. aus der langen Eingabe wird eine wesentlich kürzere aber doch das Dokument fast eindeutig repräsentierende Ausgabe erzeugt. Kryptographische Hash-Funktionen sind „Einweg-Funktionen“, d. h. aus einem Hash-Wert ist es nur sehr schwierig oder unmöglich zurück auf die Eingabe der Hash-Funktion zu schließen.

Der erzeugte kurze Hashwert $h = H(M)$ wird dann von A unterschrieben, also $C = D_d(h)$, und wiederum das Tupel (M, C) an B gesendet. Auf der anderen Seite berechnet B ebenso zunächst den Hashwert h_2 aus M und überprüft dann die Signatur C durch Berechnen von $E_e(C) \stackrel{!}{=} h_2$. Ist diese wie erwartet, so kann B annehmen, daß entsprechend A den Hash-Wert und damit auch das Dokument unterzeichnet hat. Diese Methode der Signatur ist natürlich wesentlich unsicherer als die obige, denn es kann beim Erzeugen von Hash-Werten natürlich zu sogenannten Kollisionen kommen: Werden Dokumente mit der Länge mehrerer Mega-Byte auf 128 Bit Fingerabdrücke abgebildet, so gibt es mit Sicherheit mehrere verschiedene Dokumente, die den selben Hash-Wert besitzen. B kann dann nicht wirklich sicher sein, daß A auch M und nicht zufällig ein M_2 unterschrieben hat. Es könnte nämlich sein, daß gilt: $H(M) = h = H(M_2)$. Hat ein Angreifer eine Unterschrift für ein h eines „harmlosen“ Textes, und findet er ein oder mehrere M_2 mit obigen Eigenschaften, so kann er B die M_2 mit der alten Unterschrift untergeschoben - B wird sie als von A signiert akzeptieren.

2.2.1 SHA-1

Der Secure Hash Algorithm [26] ist ein vom NIST und der National Securicy Agency (NSA) gemeinsam entwickeltes Hash-Verfahren. SHA-1 wurde ursprünglich als sichere Hash-Funktion für ein neues Signaturverfahren, den Digital Signature Algorithm (DSA) entwickelt, hat sich allerdings auch unabhängig davon durchgesetzt. Der Algorithmus produziert für jede Eingabe der Länge $< 2^{64}$ Bit einen Hash-Wert der Länge 160 Bit. Es gibt in dem Algorithmus selbst bisher keine bekannte Schwachstelle. Dies bedeutet für einen eine Kollision suchenden

Angreifer, daß er zu einem bekannten M etwa 2^{159} andere M_2 ausprobieren muß, bis gilt $H(M) = H(M_2)$, das ist sehr aufwendig. Um das Erzeugen von Signaturen deutlich zu beschleunigen - mit einem überschaubar begrenztem Nachteil für die Sicherheit - wird in der Implementierung SHA-1 zum Hashen der Daten benutzt.

2.2.2 Zertifikate und Zertifizierungsstellen

Abbildung 2.1: Struktur eines X.509 Zertifikat

Die Überprüfung digitaler Signaturen geschieht folgendermaßen: In einem Public-Key System könnte jeder Benutzer bzw. jede Rolle allen anderen Benutzern seinen öffentlichen Schlüssel zukommen lassen. Die Idee ist eigentlich gut, weil sie elek-

tronisch einfach durchsetzbar ist - der öffentliche Schlüssel wird auch elektronisch vor der eigentlichen Kommunikation, in deren Verlauf eine Signatur erzeugt und überprüft wird, übertragen. Bevor A seine Daten und die Signatur an B schickt, sendet er vorneweg seinen öffentlichen Schlüssel an B. Dies allerdings ist sehr gefährlich: ein Angreifer könnte den öffentlichen Schlüssel von A abfangen und stattdessen seinen eigenen öffentlichen Schlüssel an B schicken. Dieses Szenario heißt *Man in the Middle*-Angriff, siehe Abbildung 3.3. B würde, da er eigentlich keine Möglichkeit hat, genau zu überprüfen, von wem der öffentliche Schlüssel ist, den öffentlichen Schlüssel des Angreifers zum Verifizieren empfangener Unterschriften benutzen.

Um diesem Problem entgegenzutreten werden anstatt einfacher öffentlicher Schlüssel *Zertifikate* eingesetzt. Ein Zertifikat ist eine Art digitaler Ausweis. Auf einem Zertifikat stehen neben dem öffentlichen Schlüssel des Inhabers auch sein Name und weitere Daten, wie z. B. eine Gültigkeitsdauer, siehe Abbildung 2.1. Zertifikate werden von einer vertrauenswürdigen Instanz digital unterzeichnet [4]. Jeder, der im Besitz des öffentlichen Schlüssel dieser Instanz, auch *Certificate Authority* (CA) oder RootCA genannt, ist, kann ein Zertifikat verifizieren und damit die Gültigkeit der Zuordnung öffentlicher Schlüssel und zugehöriger Eigentümer des Schlüssels überprüfen. Vertrauen die Teilnehmer des Protokolls der CA, so können sie auch den von der CA unterschriebenen Zertifikaten vertrauen.

Zwei Probleme bleiben natürlich. Wie gelangen die Teilnehmer des Protokolls sicher an den öffentlichen Schlüssel der CA, ohne der *Man in the Middle*-Gefahr, und wie lassen sich neue Teilnehmer im System ihren neuen öffentlichen Schlüssel bei der CA zu einem Zertifikat zertifizieren. Beide Probleme lassen sich bei einem initialen Benutzer-Registriervorgang so lösen: Ein neuer Benutzer muß persönlich bei der CA erscheinen und sich so - auf kryptographisch sicherem Wege - sein Zertifikat unterschreiben und sich den öffentlichen Schlüssel der CA aushändigen lassen.

Diese Art der Vertrauensbildung über eine *Zertifizierungskette* - vertraut man der CA, so vertraut man auch den Benutzerzertifikaten - ist sicherlich fraglich, denn schließlich kann selbst die CA korrumpiert sein. Sie hat sich in der Praxis aber durchgesetzt: SSL und damit der komplette elektronische Zahlungsverkehr im Internet basiert auf dieser Form des Vertrauens. Daher werden in dieser Implementierung Zertifikate benutzt, und zwar X.509v3 Zertifikate [32].

2.3 Shared-Secrets

Shared-Secrets behandeln das grundsätzliche Problem, wie man ein Geheimnis derart zerlegen und auf k verschiedene Personen aufteilen kann, daß nur diese

k Personen zusammen und nicht einzelne bzw. bis zu $k-1$ Personen das Geheimnis zusammensetzen können. Viele Konzepte von Shared-Secrets sind vom US-amerikanischen Militär motiviert. Ein Anwendungsbeispiel ist das berühmte Starten einer Atomrakete: eine einzelne Person alleine kann mit ihren Codes die Rakete nicht starten, nur wenn zwei Personen (z. B. der Präsident und der General) im Einverständnis ihre Codes in ein Kontrollsystem eingeben, wird die Rakete gestartet. Mit Shared-Secrets sind sogar (k, n)-*Threshold Schemes* (Schwellwertverfahren) möglich. Das Geheimnis wird hierbei in insgesamt n Teile *(Shares)* zerlegt und kann dann restauriert werden, sobald mindestens k verschiedene Teile davon zusammengefügt werden. Eine recht umfassende Übersicht über die Theorie und Praxis von Shared-Secret-Systemen geben [12] und [13].

Das in dieser Arbeit entwickelte Protokoll aus Abschnitt 3.2 benötigt jedoch eine einfachere Variante ohne Schwellenwertverfahren, die dem Atomraketen-Problem gleicht. Für einen Zugriff auf chiffrierte Daten soll sich ein Benutzer einen Schlüssel von zwei verschiedenen Schlüsselservern besorgen. Jeder Server liefert dabei einen Teil des Schlüssel, der Benutzer kann die Teile zusammensetzen und erhält so den kompletten Schlüssel. Die erste Idee, die vielleicht sehr naheliegt, ist den kompletten Schlüssel, z. B. mit 128-Bit Breite, in genau zwei 64-Bit breite Teile zu zerlegen und auf jeweils einen Schlüssel-Server zu speichern. Die Idee ist gut, denn bekommt ein potentieller Angreifer nur Zugriff auf einen der Teilschlüssel, so kann er damit den gesamten Schlüssel nicht rekonstruieren und damit auch auf die mit dem gesamten Schlüssel chiffrierten Daten nicht zugreifen. Die Anzahl der Versuche, die der Angreifer bei einer Brute-Force Attacke betreiben müßte, wäre bei entsprechend sicherem Chiffriersystem allerdings nur noch etwa 2^{64}, um die restlichen 64 Schlüssel-Bit zu errechnen (s. Abschnitt 2.1.1). Ein solches Shared-Secret-System heißt dann *nicht perfekt*: Aus dem Wissen eines Teils des Geheimnisses kann man Informationen über das komplette Geheimnis sammeln.

Es gibt jedoch ein *perfektes*, wirksameres, d. h. für den Angreifer aufwendiger zu brechenendes Shared-Secret System, das Gus Simmons in [12] vorstellt. Bei diesem System werden zwei Zahlen a und b zufällig von einem endlichen Zahlenstrahl aus dem Intervall $0 \leq a, b < 2^{128}$ gewählt. Die Summe $c = a + b \bmod 2^{128}$ ist das Geheimnis, also hier der geheime Schlüssel, mit dem Daten verschlüsselt werden und a, b sind die Shares, die in unserem Falle auf die beiden Schlüssel-Server verteilt werden. Kann nun ein Angreifer ein einzelnes Share a oder b erlangen, so muß er das andere Share konstruieren. Dies kostet ihn allerdings im Gegensatz zu obigem Verfahren diesmal 2^{128} Versuche, da alle 2^{128} Zahlen auf dem Zahlenstrahl gleich wahrscheinlich sind. Der Angreifer hat also durch das Wissen über ein Teilgeheimnis keinerlei Vorteil in Bezug auf das Gesamtgeheimnis. Bei 128-Bit Schlüsselbreite gibt es 2^{128} verschiedene Schlüssel. Der Angreifer kann sogar ein beliebiges a' aus dem Zahlenstrahl auswählen und versuchen c zu errechnen, denn es gibt für jedes a' ein b', mit $a' + b' = c \bmod 2^{128}$. Das Errechnen von b' kostet allerdings wieder etwa 2^{127} Versuche.

2.4 Kommutative Chiffren

Im Verlauf des zweiten Protokolls aus Abschnitt 3.3 müssen Daten umverschlüsselt werden. Dazu werden *kommutative Chiffren* benötigt:

1. Ein Klartext M wird mit Schlüssel k_1 zu $C_1 = E_{k_1}(M)$ verschlüsselt.

2. C_1 wird nochmals mit einem zweiten Schlüssel k_2 zu

   $C_2 = E_{k_2}(C_1) = E_{k_2}(E_{k_1}(M))$ weiterverschlüsselt.

3. Es ist jetzt nötig, durch $D_{k_1}(C_2) = E_{k_2}(M) = C_3$ die erste Verschlüsselung zu lösen. Der originale Klartext M soll einfach über $M = D_{k_2}(C_3)$ rekonstruiert werden. Dies sind Anforderungen!

Für die Chiffre (E, D) muß folglich gelten $C = (M \otimes k_1) \otimes k_2 = (M \otimes k_2) \otimes k_1$, wobei $\otimes$ für „wird verschlüsselt mit“ steht. Eine Chiffre, die obige Anforderungen erfüllt, heißt *kommutative Chiffre*. Leider sind im allgemeinen symmetrische oder asymmetrische Chiffren wegen Einbußen der Sicherheit nicht ohne weiteres kommutativ. Zum Beispiel könnte RSA durch Benutzen eines gemeinsamen Modulus zu einer kommutativen Chiffre umfunktioniert werden, jedoch ist RSA dann leicht brechbar (siehe [41]). Auch andere Systeme, wie z. B. das 3-Wege-Exponieren [31] sind zwar kommutativ aber unsicher. Hinzukommt, daß die möglichen Public-Key-Systeme mit Exponieren in endlichen Körpern arbeiten, was sehr aufwendig und langsam ist. AES mit einer Schlüssellänge von 128 Bit ist um den Faktor 1,8 schneller als RSA mit einer Schlüssellänge von 2048 Bit, wobei die Schlüssellägen vergleichbare Sicherheit bieten.

2.4.1 One-Time-Pad

Es gibt aber auch wirksame kommutative Chiffren, die hinsichtlich ihrer Effizienz keine Einschränkungen aufweisen. Beispielsweise eignet sich das einfache bitweise „exklusive Oder“ *XOR*. Als Schlüssel k_1 und k_2 für *XOR* werden dann entsprechend dem Klartext lange *One-Time-Pads* verwendet. Ein One-Time-Pad ist eine einmalige Zufallszahlenfolge (siehe Abschnitt 2.5), die mindestens die gleiche Länge wie der zu verschlüsselnde Klartext hat. Der One-Time-Pad ist das einzige bisher bekannte Chiffriersystem für das bewiesen wurde, daß es keinen Angriff dagegen geben kann [39]. Der One-Time-Pad als Schlüssel steht und fällt natürlich mit seiner „Zufälligkeit“ bzw. Pseudo-Zufälligkeit, deshalb muß darauf ein großes Augenmerk liegen. Aus diesem Grund wird im nächsten Abschnitt die Theorie zum Erzeugen von Zufallszahlen gezielt diskutiert.

2.4.2 Drei-Wege-XOR

Es gibt eine Attacke gegen XOR, wenn es als kommutative Chiffre eingesetzt wird. Dieses *Drei-Wege-XOR* [31] zwischen den Benutzern A und B stellt sich folgendermaßen dar:

A		B
$c_1 = m \oplus k_A$		
	$\xrightarrow{c_1}$	
		$c_2 = c_1 \oplus k_B = m \oplus k_A \oplus k_B$
	$\xleftarrow{c_2}$	
$c_3 = c_2 \oplus k_A = m \oplus k_B$		
	$\xrightarrow{c_3}$	
		$m = c_3 \oplus k_B$

Dabei ist m der Klartext und k_A sowie k_B sind zwei *One-Time-Pads*.

Die folgende Rechnung widerlegt, daß das Versenden von c_1, c_2 und c_3 sicher ist, denn der Klartext läßt sich einfach aus der Verknüpfung der Chiffrate rekonstruieren:

$$\begin{aligned} c_1 \oplus c_2 \oplus c_3 &= (m \oplus k_A) \oplus (m \oplus k_A \oplus k_B) \oplus (m \oplus k_B) \\ &= m \oplus m \oplus m \oplus k_A \oplus k_A \oplus k_B \oplus k_B \\ &= m \end{aligned}$$

Ist ein *Big-Brother*, ein Angreifer, der sämtlichen Netzverkehr abhören kann, in der Lage, alle Chiffrate mitzulesen, so hat er dadurch die Möglichkeit, m zu rekonstruieren. In Abschnitt 3.3 wird eine Maßnahme zur Verteidigung gegen dieses Drei-Wege-XOR beschrieben.

2.5 Zufallszahlen

In jedem kryptographisch arbeitenden System werden Zufallszahlen benötigt. Sei es zum Generieren von geheimen symmetrischen oder asymmetrisch Schlüsseln, oder, wie in dieser Arbeit, zum Erzeugen eines One-Time-Pads.

Das Problem ist allerdings, Zufallszahlen zu generieren, die wirklich „zufällig“ sind. Man spricht hier von *echten Zufallszahlen*. Echter Zufall bedeutet, daß unabhängig von allen vorherigen Ausgaben für die Wahrscheinlichkeit p der nächsten Bit-Ausgabe eines Zufallszahlen-Generators immer gilt: $p(0) = \frac{1}{2}$ und $p(1) = \frac{1}{2}$. Das heißt, es ist gleichwahrscheinlich, ob das nächste Bit eine 0 oder eine 1 wird.

Obwohl diese Eigenschaft sehr einfach klingt, sind Generatoren, die echte Zufallszahlen liefern, selten und teuer. Bekannte Beispiele für echte Zufallszahlen-Generatoren sind z. B. Geiger-Zähler, die zufällige atomare Zerfallsprozesse erfassen oder Mikrophone, die Meeresgeräusche aufnehmen und daraus Zahlen produzieren.

Da echter Zufall für die meisten Anwendungen zu teuer oder nicht einsetzbar ist, werden in der Praxis sogenannte *Pseudozufallszahlen* eingesetzt. Pseudozufallszahlen sind keine echten Zufallszahlen, mit obigen Eigenschaften, sondern nur Zahlen, die zufällig aussehen. Sie sind nicht wirklich zufällig, sondern werden nach einer bestimmten Vorschrift berechnet und sind damit, bei bekannter Vorschrift, auch vorhersagbar.

Pseudozufallszahlen sind bei unbekannter Berechnungsvorschrift, oder unbekanntem Startwert (s. unten) innerhalb einer bestimmten Periode, d. h. Länge der Generatorausgabe scheinbar zufällig und nicht vorhersagbar. Danach aber wiederholen sie sich gemäß der Periode. Pseudozufallszahlen-Generatoren (PRNG[1]) arbeiten meist mit einem kleinen initialen Zufallswert, der mehr oder weniger echten Zufall repräsentiert, wie z. B. die aktuelle Systemzeit oder zufällige Mausbewegungen des Benutzers. Aus diesem Initialen Wert (engl. *Seed*) wird dann die ganze Pseudo-Zufallsfolge abgeleitet. Viele Pseudo-Generatoren sind anfällig gegen bestimmte Attacken, daher wird für die One-Time-Pads unseres Zyklischen Protokolls eine von [17] zunächst kryptanalysierte und danach verbesserte Version des Pseudozufallszahlen-Generators des DSA-Standards aus [27] verwendet.

Zuerst zur Beschreibung der Arbeitsweise des DSA-Generators: Dargestellt ist die Berechnung der nächsten Pseudo-Zufallszahl $output[i]$ zum Zeitpunkt i.

1. Der PRNG hat einen internen Zustand X_i, dieser wird am Anfang ($i = 0$) mit dem Seed initialisiert.

2. Der PRNG akzeptiert in jedem Schritt i eine optionale Eingabe W_i, diese sollte möglichst echter Zufall sein und dem System Entropie hinzufügen

3. Es wird berechnet:

   - $output[i] = hash(W_i + X_i \ mod \ 2^{160})$
     $hash$ bezeichnet eine HAsh-Funktion, beispielsweise *SHA-1*.
   - $X_{i+1} = X_i + output[i] + 1 \ mod \ 2^{160}$

$output[i]$ ist die nächste Zufallszahl. Da die nächste Zufallszahl vom Berechnen von *output*, also in erster Linie von der Qualität der Hash-Funktion abhängt, ist die ausgegebene Pseudo-Zufallszahlenfolge $output[i]$ bei entsprechend guter Hash-Funktion wie z. B. *SHA-1* schwer von einer echten zu unterscheiden. Die

[1] Pseudo-Random-Number-Generator

kryptographische Analyse dieses Generators wird Abschnitt 4.1 vorweggenommen, da aus der Analyse heraus ein neuer Generator entsteht, der den bekannten Attacken standhält.

Im Folgenden wird beschrieben, wie sich der DSA-PRNG bei typische Generator-Attacken aus [17] verhält und gegebenenfalls verbessert werden kann.

Falls ein Angreifer per Zufall an den initialen Seed, oder zu irgendeinem Zeitpunkt i an den internen Zustand X_i gelangt, so kann er daraus bei diesem Generator im Gegensatz zu vielen anderen keinen Gewinn erzielen, da durch das wiederholte Eingeben weiterer Entropie W_i der Angreifer die nächsten Ausgaben und internen Zustände des Generators nicht vorherberechnen kann. (Gäbe es die Entropie W_i nicht, so könnte er alle Ausgaben vorhersagen.)

Input based Attack

Problematisch ist jedoch eine sogenannte *Input Based Attack*, bei der der Angreifer die Eingaben W_i beliebig verändern kann. Dies ist nicht unrealistisch. Wird z. B. die Systemzeit als zusätzliche Entropie verwendet, und kann der Angreifer diese festlegen, so wird durch das Setzen von $W_i = W_{i-1} - output[i-1] - 1$ die Ausgabe $output[i]$ konstant gehalten. Der Angreifer erreicht dadurch, daß immerwieder die gleiche Zahl vom PRNG ausgegeben wird.

Da für den internen Zustand X_i gilt $X_i = X_{i-1}+\text{output}[i-1]+1$, ist

$$\begin{aligned} output[i] &= hash(W_i + X_i \ mod \ 2^{160}) \\ &= hash(W_{i-1} - output[i-1] - 1 + X_{i-1} + output[i-1] + 1 \ mod \ 2^{160}) \\ &= hash(W_{i-1} + X_{i-1} \ mod \ 2^{160}) \\ &= output[i-1]. \end{aligned}$$

Auf diese Weise wird zwar der interne Zustand X_i nicht erraten, aber die Ausgabe des PRNG festgesetzt.

Eine einfache Abhilfe gegen die *Input Based Attack* bietet hier das Hashen sämtlicher Eingabe-Entropien mit Hilfe einer kryptographischen Einweg-Funktion, also $W_i = hash(Entropie)$. Der Angreifer kann keine Entropie bestimmen, die gehasht sein gewünschtes W_i ergibt.

Filling in the Gaps

Ein anderes Problem ist die *Filling in the Gaps*-Attacke: der Angreifer besitzt hier X_i, X_{i+2} und $output[i+1]$. Er benötigt $output[i]$, das *Gap*, die Zufallszahl, die vor der aktuellen Zahl vom Generator produziert wurde. Er kann dies über

$$output[i] = X_{i+2} - X_i - 2 - output[i+1]$$

berechnen. Abhilfe für diese Sicherheitslücke bringt hier das Hashen der Ausgabe, vor der Berechnung von X_{i+1}:

$$X_{i+1} = X_i + hash(output[i] + W_i) \ mod \ 2^{160}.$$

Mit den beschriebenen Verbesserungen scheint der DSA-PRNG sehr sicher zu sein und allen Attacken zu widerstehen. Er wird in dieser Arbeit nicht nur als Generator für Schlüsselpaare verwendet, sondern auch zum Erzeugen von One-Time-Pads.

Es können mit diesem Generator keine symmetrischen Schlüssel länger als 160-Bit sicher erzeugt werden, denn der SHA-Algorithmus hasht alle Entropie-Eingaben auf 160-Bit. Dies bedeutet, es steht für die Erzeugung eines geheimen Schlüssels auch nicht mehr Entropie als 160 Bit zur Verfügung. Das Erzeugen längerer Schlüssel erhöht nicht die Sicherheit. Ähnliches gilt für das Generieren von asymmetrischen Schlüsseln, z. B. beim Erzeugen von Primzahlen beim RSA-Algorithmus. Auch hier kann nur mit einer Unsicherheit von 160 Bit gerechnet werden. Allerdings sind 160 Bit ausreichend sicher, wie bereits mehrfach erwähnt.

Die Güte der generierten Zufallszahlen wird in Abschnitt 4.1.3 mit dem sogenannten FIPS 140-1 Test untersucht.

2.6 Logic of Authentication

Ein Protokoll ist eine Kette von Aktionen und Vorschriften, an der mehrere Parteien beteiligt sind. Bei den Aktionen handelt es sich zumeist um das Versenden von Nachrichten. Nach dem erfolgreichen Durchführen aller Aktionen ist ein bestimmtes Protokollziel erreicht. Ein kryptographisches Protokoll dient meist dem Austausch geheimer Informationen oder der gegenseitigen Authentifizierung von Protokollteilnehmern.

In [19] wurde mit der *Logic of Authentication* oder *BAN-Logik*[1] eine formale Logik zur Analyse kryptographischer Protokolle entworfen. Aufgabe dieser Logik ist es, die Protokollziele genauer zu identifizieren sowie zu überprüfen, welche Informationen die Teilnehmer miteinander ausgetauscht haben bzw. glauben, miteinander ausgetauscht zu haben. Der Zentrale Begriff der BAN-Logik ist der *Glaube* der Teilnehmer an bestimmte wahre oder unwahre Tatsachen. Die Teilnehmer des Protokolls können Nachrichten oder Informationen vertrauen und diesen glauben. Glaubt ein Protokoll-Teilnehmer einer Information, heißt das allerdings noch lange nicht, daß diese wirklich gültig ist. Er kann sich dabei unter Umständen geirrt haben.

Der erste Schritt bei der Analyse eines Protokolls mit der BAN-Logik ist es, initiale Annahmen über die Protokollumgebung, sowie das Wissen, bzw. den Glauben

[1] Nach M. Burrows, M. Abadi und R. Needham

der am Protokoll beteiligten Parteien zu identifizieren und entsprechend der Logik zu formulieren. Solche sog. *Assumptions* können z. B. die Annahmen über den Besitz öffentlicher, privater bzw. geheimer Schlüssel der verschiedenen Protokollteilnehmer sein. Es wird bestimmt, *welcher* Teilnehmer in Besitz von *welchem* Schlüssel ist, und *welchem* anderen Teilnehmer er *was* glaubt.

Im zweiten Schritt werden die einzelnen Nachrichten des Protokolls in eine idealisierte, formalisierte Form gebracht und nacheinander von der Logik interpretiert.

Analog zu einem Expertensystem gibt es eine Menge von Ableitungsregeln und eine Menge von Fakten bzw. *Prädikaten*, nämlich die formalisierten Nachrichten des Protokolls und die initialen Annahmen. Iterativ werden dann auf jede Nachricht die Ableitungsregeln angewandt und somit neue Prädikate erzeugt. Läßt sich keine weitere Ableitungsregel mehr anwenden, so ist die BAN-Verifikation schließlich beendet. Aus der dann vorhandenen großen Menge von abgeleiteten Prädikaten versucht man die Protokollziele (*Goals*) zu finden. Diese sind ebenso im vorhinein formalisierte Prädikate, die den gewollten Schlüsselaustausch oder die durchgeführte Authentifizierung, das Ziel des Protokolls, beschreiben. So wäre z. B. „Teilnehmer A glaubt, daß K der geheime Schlüssel zwischen A und Teilnehmer B ist.“ ein Protokollziel. Werden solche Prädikate gefunden, so erfüllt das Protokoll zunächst einmal zumindest seine Aufgabe.

Es ist weiterhin interessant, die Menge der abgeleiteten Prädikate zu analysieren und mögliche Schwachstellen zu finden. Außerdem kann das Protokoll mit anderen, schwächeren Annahmen und veränderten Nachrichten getestet und optimiert werden. Dadurch können u. U. überflüssige Nachrichten oder Voraussetzungen erkannt und entfernt werden.

2.6.1 Notation der Logic of Authentication

Im Hinblick auf Lesbarkeit und die später folgenden Analysen in Abschnitt 4.2, wird hier die Prädikatschreibweise aus [8] vorgestellt. Im folgenden bezeichnet A eine *Entität*. Entitäten sind die einzelnen Teilnehmer des Protokolls. X stellt irgendeine beliebige Information oder einen Sachverhalt dar. K oder K_a sind Schlüssel, wobei K einen symmetrischen und K_a einen asymmetrischen Schlüssel bezeichnet.

Prädikate

In Tabelle 2.1 sind die grundlegenden BAN-Prädikate aufgelistet. Prädikate $P(Q)$ sind Aussagen über Eigenschaften. Das Prädikat $P(Q)$ ist die Aussage, daß Q die Eigenschaft P erfüllt.

believes(A, X) A glaubt die Information X, d. h. A handelt so, als wäre X wahr. Dies ist das zentrale Prädikat der Logik.

sees(A, X) A sieht die Information X. Dies geschieht z. B. durch das Empfangen einer Nachricht mit Inhalt X.

said(A, X) A hat, zu einem Zeitpunkt der (längeren) Vergangenheit, einmal X in einer Nachricht verschickt.

fresh(X) Die Information X, bzw. der Sachverhalt X ist aktuell, beispielsweise vor kurzer Zeit von jemandem gesendet worden. Zeit wird in der BAN-Logik in zwei Teile eingeteilt: *Vergangenheit* und *Gegenwart*. Die Gegenwart ist der Zeitpunkt der gerade aktuellen Ausführung des Protokolls. Vergangenheit alles, was zeitlich davor liegt. Damit kann man sich gegen *Replay*-Attacken, bei denen Nachrichten aus vergangenen Protokoll-Durchläufen erneut verwendet werden, schützen. *fresh(X)* gilt, wenn X davor in noch keiner Nachricht verschickt wurde. In der Praxis wird die Auswertung von *fresh()* meist durch Zeitstempel realisiert.

public_key(K_a, A) K_a ist der öffentliche Schlüssel von A, den dazu passenden geheimen Schlüssel K_a^{-1} kennt nur A, bzw. eine von A später autorisierte Person.

inv(K_a, K_a^{-1}) Der Schlüssel K_a^{-1} ist bzgl. der Chiffre invers zum öffentlichen Schlüssel K_a. K_a^{-1} ist der private Schlüssel.

shared_key(K, A, B) Der Schlüssel K ist ein geheimer symmetrischer Schlüssel zwischen A und B. Nur A und B, bzw. von ihnen autorisierte Teilnehmer kennen K.

controls(A, X) A ist zuständig für den Sachverhalt X, und *kontrolliert* ihn, d. h. A ist eine Autorität in Bezug auf X. Zum Beispiel müssen in manchen Protokollen geheime Schlüssel von einer dafür zuständigen Autorität generiert werden. Mit diesem Prädikat gesteht man A die Fähigkeit zum Generieren guter geheimer Schlüssel zu.

encrypt(X, K)	X ist mit dem Schlüssel K verschlüsselt worden. Aus dem Chiffrat kann X ohne Kenntnis von K nicht rekonstruiert werden.

Tabelle 2.1: Prädikate der BAN-Logik

Ableitungsregeln der BAN-Logik haben die Form:

$$\frac{P_1, P_2, \ldots, P_n}{P'}.$$

Die Schreibweise bedeutet, daß aus der Gültigkeit der Prädikate $P_1, \ldots, P_n$ die Gültigkeit des Prädikats P' folgt.Tabelle 2.2 führt einige der grundlegenden Ableitungsregeln der BAN-Logik auf.

Glaubt A, daß K ein symmetrischer Schlüssel zwischen A und B ist, und empfängt A ein mit K verschlüsseltes Datum, so glaubt A, daß B dieses geschickt hat.

$$\frac{\mathit{believes}(A, \mathit{shared\_key}(K, A, B)), \mathit{sees}(A, \mathit{encrypt}(X, K))}{\mathit{believes}(A, \mathit{said}(B, X))}$$

Für öffentliche Schlüssel gilt ähnliches. Glaubt A, daß K_b der öffentliche Schlüssel von B ist, und empfängt A eine mit K_a^{-1} verschlüsselte Nachricht, so glaubt A, daß diese von B stammt.

$$\frac{\mathit{believes}(A, \mathit{public\_key}(K_b, B)), \mathit{sees}(A, \mathit{encrypt}(X, K_b^{-1}))}{\mathit{believes}(A, \mathit{said}(B, X))}$$

Glaubt A, daß X aktuell ist, und daß B den Sachverhalt X gesendet hat, so glaubt A, daß B auch an X glaubt.

$$\frac{\mathit{believes}(A, \mathit{fresh}(X)), \mathit{believes}(A, \mathit{said}(B, X))}{\mathit{believes}(A, \mathit{believes}(B, X))}$$

Diese Regel ist gefährlich: Sie impliziert, daß jeder Sender auch wirklich das glaubt, was er verschickt. Ein Sender *lügt* also nicht. Man spricht von *Honesty*, wenn kein Protokollteilnehmer lügt. In den Erweiterungen zur BAN-Logik wird auf dieses Problem eingegangen.

Wenn A glaubt, daß B Kontrolle über X hat, und A glaubt, daß B an X glaubt, so glaubt auch A an X.

$$\frac{believes(A, controls(B, X)), believes(A, believes(B, X))}{believes(A, X)}$$

Tabelle 2.2: Grundlegende BAN-Ableitungsregeln

Die Menge der BAN Regeln wird vervollständigt durch Regeln zum Auftrennen von Nachrichten. Glaubt A z. B. einer Nachricht $X = X_1, X_2, \ldots, X_n$, so glaubt A auch allen einzelnen $X_1, X_2, \ldots, X_n$. Eine Übersicht über die vollständige Regelmenge findet sich in [19] oder [20].

Beispielableitung

Zur Verdeutlichung wird im Folgenden eine Ableitung nach BAN-Regeln demonstriert.

Gegeben sei das Prädikat

$$believes(A, shared\_key(K, A, B)).$$

Protokoll-Entität A glaubt, daß K ein geheimer Sitzungsschlüssel zwischen A und Entität B ist.
Nun schickt B die Nachricht

$$encrypt(X, K)$$

an A. Dies bedeutet, daß ein neues Prädikat,

$$sees(A, encrypt(X, K))$$

der Menge der Prädikate hinzugefügt wird.
Daher kann beispielsweise die erste oben vorgestellte Regel angewendet werden, welche ein weiteres Prädikat

$$believes(A, said(B, X))$$

erzeugt.
Die Menge der Prädikate nach dem Verschicken der Nachricht besteht demnach aus den Prädikaten:

$$believes(A, shared\_key(K, A, B)),$$

$$sees(A, encrypt(X, K)),$$

$$believes(A, said(B, X)).$$

Sukzessiv wächst auf diese Weise die Menge der Prädikate. Kommen neue Prädikate hinzu, können weitere Regeln angewendet werden, die wiederum neue Prädikate produzieren. Nach dem Versand einer Nachricht, werden solange Regeln auf die Prädikate angewendet, bis keine Regel mehr anwendbar ist. Dann wird mit der nächsten Nachricht fortgefahren.

2.6.2 RVLogik, eine Erweiterung der BAN-Logik

Wird ein Protokoll erfolgreich mit der BAN-Logik verifiziert, so ist dieses Protokoll noch lange nicht wirklich sicher. Bei der initialen Festlegung der *Assumptions* kann der Verifizierer genauso von falschen Voraussetzungen ausgegangen sein, wie bei der Abstraktion und Idealisierung der jeweiligen Nachrichten. Die Assumptions in der *realen Welt* können ganz andere sein, als der Verifizierer angenommen hat. Beispielsweise wird in [8] beschrieben, wie bei der Durchführung der ursprünglichen BAN-Analyse aus [19] für das Public-Key-Protokoll nach Needham-Schroeder von einer falschen Idealisierung ausgegangen wurde, die lange unentdeckt geblieben ist. Genauso können die gesuchten Protokollziele falsch formuliert werden. Daneben fehlt die Möglichkeit, negative Aussagen zu überprüfen. Es ist ohne Weiteres nicht möglich, zu testen, ob A ein bestimmtes X nicht glaubt.

Daneben geht die BAN-Logik von bestimmten kryptographischen Idealen aus, z. B. der perfekten Sicherheit der im Protokoll benutzten Chiffriersysteme. Die Information X aus $encrypt(X, K)$ kann wirklich nur derjenige lesen, der K besitzt, unabhängig von der gewählten Schlüssellänge von K oder der Sicherheit der Chiffre, die $encrypt()$ zugrunde liegt. Genauso können böswillige oder kompromittierte Protokollteilnehmer, die neben dem *Big-Brother*-Abhören des kompletten Netzverkehrs auch noch Insider-Informationen besitzen nur schwer simuliert werden. Hierfür wären mehrere Simulationsdurchläufe mit verschiedenen Szenarien für kompromittierte Teilnehmer notwendig.

Die genannten Probleme fordern aus diesen Gründen einen bewußten Umgang mit der BAN-Logik und lassen sich nicht durch Veränderungen der Logik beheben. Der Verifizierer muß entsprechend vorsichtige und gewissenhafte Idealisierungen und treffende Annahmen bestimmen, sowie die Ergebnisse einer Analyse entsprechend interpretieren.

Für andere Nachteile gibt es allerdings Erweiterungen, wie die *ReVere-Logik* (kurz *RVLogik*) aus [8], die die Analysemethoden verstärken und so die Fehlersuche in Protokollen ohne großen Mehraufwand sicherer macht.

Explizite Interpretationsregeln

Im allgemeinen entsteht durch das Formalisieren der Nachrichten ein Unterschied zwischen der Bedeutung der tatsächlichen Nachricht, die laut Protokollspezifikation geschickt wird, sowie der formalisierten Nachricht, die während der Analyse verschickt wird.

Die RVLogik versucht durch das explizite Angeben von Nachrichten-Interpretationen diesen Unterschied zu minimieren. Ein Beispiel: empfängt Entität A von B einen Schlüssel K, so wird dies auf

$$believes(A, said(B, K))$$

formalisiert. Im tatsächlichen Protokoll soll A aber jetzt K als gemeinsamen Schlüssel von A und B interpretieren. Eine derartige Formulierung ist in der BAN-Logik so ohne weiteres nicht möglich. Die RVLogik jedoch erlaubt eine Interpretationsregel, die im Klartext lauten würde: „empfängt A von B irgendein K, so glaubt A, daß dieses K ein gemeinsamer Schlüssel zwischen A und B ist". Formal läßt sich das ausdrücken durch

$$\frac{believes(A, said(B, K))}{believes(A, shared\_key(K, A, B))}.$$

Allgemein erweitert die RVLogik BAN um die Regeltypen

$$(1)\quad \frac{believes(A, said(B, N))}{believes(A, said(B, N'))}$$

und

$$(2)\quad \frac{sees(A, N)}{sees(A, N')},$$

wobei N eine konkrete formale Nachricht aus dem Protokollablauf ist und N' eine Formel, die die Interpretation der Nachricht repräsentiert. In N' können Variablen aus N vorkommen (siehe obiges Beispiel). Die Regeln vom Typ (1) dienen der Interpretation aller authentischen Nachrichten, da hier A bereits weiß, daß die Nachricht von B kommt. Hingegen interpretieren Regeln vom Typ (2) Nachrichten unbekannten Ursprungs. Die RVLogik legt einige Rahmenbedingungen für Interpretationsregeln fest, z. B. daß auf jede formale Nachricht innerhalb des Protokolls höchstens eine Interpretationsregel zutrifft, oder daß in N' das Schlüsselwort *encrypt* nicht enthalten sein darf.

Mit der entsprechenden Interpretationsregel bekommt so jede Nachricht eine realistische und doch formalisierte Bedeutung.

Ehrlichkeit

Ein Nachteil der BAN-Logik ist das *Ehrlichkeits (Honesty)*-Problem. Die BAN-Logik fordert, daß der Sender einer Nachricht den Inhalt dieser Nachricht immer selbst glaubt. Dies ist bei modernen Protokollen schwierig, nicht nur, weil ein Sender bewußt lügt und eine falsche Information verschickt, sondern vor allem weil das Weiterschicken (engl. *forwarding*) von empfangenen Nachrichten unbekannten Inhalts so nicht möglich ist. Die RVLogik läßt die erzwungene Forderung nach der Ehrlichkeit der Nachrichten offen. Sie überprüft jedoch während der Analyse jede Nachricht auf die Aufrichtigkeit des Senders. Der Verifizierer muß dann entscheiden, ob sein Protokoll mit einer entsprechend unehrlichen Nachricht einwandfrei arbeitet oder ob er andere, stärkere Annahmen fordern muß. Wiederum gilt, daß trotz der Ehrlichkeit einer Nachricht die Annahmen immer noch gefährlich „gutgläubig“ gewählt worden sein können.

Um die *Honesty*-Eigenschaft für eine Nachricht zu erreichen, muß folgende Bedingung erfüllt sein: für jede Nachricht, die vom Sender durch Chiffrierung mit seinem privaten asymmetrischen oder geheimen symmetrischen Schlüssel unterschrieben wird, soll gelten, daß der Sender zum Zeitpunkt des Sendens an diese glaubt.

Da der Inhalt der Nachrichten für die Protokolle jedoch keine direkte Bedeutung hat - schließlich werden hier nur Byte-Folgen betrachtet - macht es wenig Sinn, an diese zu glauben. Stattdessen muß der Sender an die mögliche Interpretation der Nachricht auf Seiten des Empfängers glauben. Empfängt beispielsweise B eine von A unterzeichnete oder verschlüsselte Nachricht N und interpretiert diese als gemeinsamen geheimen Schlüssel, so muß A auch an die Bedeutung *dieses* gemeinsamen Schlüssel glauben.

Formal prüft die RVLogik die:

> **Honesty-Eigenschaft**
> Für jede Nachricht N, die A verschlüsselt und versendet, sowie für die mögliche Interpretation N' von N des Empfängers muß A zum Zeitpunkt des Verschickens an N' glauben (siehe [8]).

Die RVLogik führt ein neues Prädikat $legit(N)$ ein, an das ein Sender glauben muß, damit seine zu verschickende Nachricht ehrlich ist. Um auf $legit()$ abzuleiten, werden u. a. folgende Regeln benötigt:

$$\frac{believes(A, N'), believes(A, signed(N, N_S, A, B))}{believes(A, legit(N_S))},$$

wobei $signed(N, N_S, A, B)$ bedeutet, daß N_S die aus N verschlüsselte Nachricht

von A an B ist. Zusätzlich gelten

$$\frac{believes(A, believes(B, shared\_key(K, A, B)))}{believes(A, signed(N, encrypt(N, K), A, B))}$$

und

$$\frac{sees(A, public\_key(K_a, A)), inv(K_{a'}, K_a)}{believes(A, signed(N, encrypt(N, K_{a'}), A, B))}.$$

Die restlichen Regeln sind in [8] zu finden.

Um die Honesty-Eigenschaft für Protokolle zu überprüfen, testet die RVLogik nur, ob das Prädikat $believes(A, legit(N))$ für jede Nachricht N, die von A geschickt wird, gilt. Besteht eine Nachricht die Überprüfung auf Ehrlichkeit nicht, so bedeutet dies, daß Sender und Empfänger verschiedene Interpretationen vom Inhalt dieser Nachricht haben. Ob dies problematisch für den weiteren Verlauf des getesteten Protokolls ist, muß der Verifizierer entscheiden.

Geheimhaltung

Ein weiterer Kritikpunkt an der BAN-Logik ist die Geheimhaltung (engl. *Secrecy*). Es wird nicht überprüft, ob durch das Übertragen von Nachrichten vielleicht Informationen veröffentlicht werden, die eigentlich geheim bleiben sollten. Wird z. B. durch einen Fehler ein symmetrischer Schlüssel im Klartext übertragen, so erkennt die BAN-Logik dies nicht als Problem. Die RVLogik stellt ein Werkzeug bereit, um solche Fehler zu vermeiden. Der Sender der Nachricht muß davon überzeugt sein, daß der Empfänger den Inhalt der Nachricht sehen darf. Das Prädikat $maysee(A, X)$ legt fest, daß A ohne weiteres X sehen darf. Ist K ein gemeinsamer Schlüssel von A und B, so gilt $believes(A, maysee(B, K))$ und $believes(B, maysee(A, K))$. Hier gehen sowohl A als auch B davon aus, daß jeweils der andere Teilnehmer aber kein dritter den geheimen Schlüssel K sehen darf.

Eine Nachricht N kann sicherlich gefahrlos übertragen werden, wenn der Sender davon ausgeht, daß jeder und nicht nur der designierte Empfänger sie lesen darf. Formal könnte dies durch den Ausdruck $\forall P,\ believes(A, maysee(P, N))$ beschrieben werden. Zur Vereinfachung verwendet man anstelle des Quantors $\forall P$ einfach nur die Variable I für einen möglichen Angreifer *(Intruder)*. Damit ist außerdem ein Big-Brother Angreifer erfaßt, der Zugriff auf alle Übertragungskanäle hat und so alle Nachrichten mithören kann. Glaubt der Sender, daß dieser potentielle Angreifer die übertragene Nachricht mitlesen darf, weil dadurch kein Geheimnis verraten wird, so ist diese Nachricht zumindest aus Sicht des Senders sicher. Die RVLogik testet folglich für jede Nachricht N mit Sender A, ob $believes(A, maysee(I, N))$ gilt. Unter anderem leiten folgende Regeln $maysee()$ ab:

$$\frac{sees(A, shared\_key(K, B, C))}{believes(A, maysee(B, K))},$$

$$\frac{believes(A, maysee(B, X)), believes(A, public\_key(K, B))}{believes(A, maysee(I, encrypt(X, K)))}$$

und

$$\frac{\begin{array}{c}believes(A, maysee(B, X)), believes(A, maysee(C, X)),\\ believes(A, shared\_key(K, B, C))\end{array}}{believes(A, maysee(I, encrypt(X, K)))}.$$

Durchführbarkeit

Eine weitere Funktion der RVLogik ist das Überprüfen jeder einzelnen Nachricht auf Durchführbarkeit engl. *(Feasibility)*. Das Senden einer Nachricht ist nur dann durchführbar, wenn der Sender im Besitz der einzelnen Komponenten der Nachricht ist. Will der Sender z. B. die Nachricht $encrypt(X, K)$ verschicken, so kann er dies nur, wenn er bereits im Besitz von $encrypt(X, K)$ oder von X und K ist. Der Besitz wird durch die Prädikate $sees()$ oder $has()$ formalisiert.

Erreicht wird diese Überprüfung durch das Prädikat $canProduce(P, X)$, das angibt, daß ein Protokollteilnehmer P die Formel X *produzieren* kann. Für jede Nachricht M mit Sender P muß gelten $canProduce(P, M)$.

Abgeleitet wird das Prädikat durch Regeln der Form:

$$\frac{canProduce(P, X), canProduce(P, K)}{canProduce(P, encrypt(X, K))}.$$

Kapitel 3

Protokolle

Die zwei Hauptaufgaben der zwei vorzustellenden Protokolle sind das Speichern eines Datensatzes eine Datenbank sowie das Lesen dieses Datensatzes aus der Datenbank. Dabei müssen die in Abschnitt 1.2 beschriebenen Anforderungen erfüllt, d. h. bestimmte Sicherheits- und Geheimhaltungskriterien zugesichert werden.

Im Folgenden sollen die zu analysierenden Protokolle beschrieben werden. Da der Mechanismus von Rollenserver und Rechteserver bei beiden Protokollen gleich ist und nicht zur Aufgabe dieser Arbeit gehört, wird auf die bereits in Abschnitt 1.2 vorgestellten Zugriffe auf diese Server nicht mehr im Detail eingegangen. Genausowenig ist das Vorhandensein einer obersten, zentralen Zertifizierungsinstanz CA, oder *RootCA* für den Protokollablauf von Bedeutung.

Im Vordergrund steht die Authentifizierung bzw. der Zugriff auf geheime Schlüssel und Datenbank.

Neben der sprachlichen Beschreibung des Protokollablaufs und den schematischen Darstellungen sind zu jeder der verschickten Nachrichten im Protokoll die entsprechenden formalisierten Nachrichten in BAN- bzw. RVLogik-Notation angegeben, für das erste Protokoll, das *Shared-Protokoll*, in Abschnitt 3.2 und für das zweite Protokoll, das *Zyklische Protokoll*, in Abschnitt 3.3. Mit diesen wird in Abschnitt 4.2 die formale Analyse der Protokolle durchgeführt. Außerdem sind jeweils die initialen Annahmen aufgezählt. Diese stellen die Voraussetzungen *vor* dem Protokolldurchlauf dar.

3.1 Gemeinsamkeiten der entwickelten Protokolle

Sowohl das Shared- als auch das Zyklische Protokoll haben gemeinsam, daß Benutzer unter bestimmen *Rollen* auftreten, wie in Abschnitt 1.1 beschrieben. Diese Rollen müssen zunächst eingenommen werden. Vor Ablauf der Protokolle erzeugt

sich ein Benutzer daher einen Rollenproxy, der für ihn den weiteren Protokolldurchlauf einer Transaktion übernehmen wird. Hierzu meldet sich ein Benutzer beim Rollenserver an und authentifiziert sich unter seinem Benutzernamen oder seinem Zertifikat und einem geheimen Paßwort. Der Rollenserver überprüft die Gültigkeit der Anfrage und übergibt bei entsprechender Zulässigkeit die Geheimnisse wie z. B. geheime Rollen-Schlüssel an den Rollenproxy zurück. Damit hat der Benutzer die Rolle eingenommen.

Verläßt der Benutzer freiwillig oder erzwungenermaßen die Rolle später wieder, so meldet er sich zunächst beim Rollenserver ab. Danach löscht er seinen Rollenproxy mit den Geheimnissen der Rolle. Damit hat der Benutzer die Rolle verlassen, da ihm alle Rollen-spezifischen Informationen fehlen.

Ob ein Benutzer eine Lese-/Schreib-Transaktion überhaupt durchführen darf, wird mit Hilfe des Rechteservers überprüft. Der Rechteserver ist in der Lage zu entscheiden, ob Anfragen von Rollenproxys rechtmäßig sind oder nicht. Vor jeder Transaktion schickt der Rollenproxy die Anfrage nach einer Transaktion codiert an den Rechteserver. Dieser überprüft, ob die Rolle zu dieser Transaktion berechtigt ist und unterschreibt sie gegebenenfalls. Die Unterschrift des Rechteservers wird vom Rollenproxy beispielsweise an die Transferstation, einer Station zum Umverschlüsseln von Daten, die Datenbank oder an den Schlüsselserver als Teil der Transaktion mitgeschickt und dort überprüft.

Der Rollenproxy selbst autorisiert sich beim Rechteserver durch eine gültige Unterschrift unter seine Transaktion. Gültig bedeutet, daß dessen Unterschrift mit dem allgemein bekannten Zertifikat der Rolle übereinstimmt.

Die Rechtevergabe und Autorisierung, sowie die Verwaltung und das Einnehmen von Rollen sind komplexe eigenständige Probleme, auf die in dieser Arbeit nicht weiter eingegangen wird. Hier werden die beschriebenen einfachen Modelle wie Paßwortüberprüfung dafür benutzt. Für weitere Informationen siehe [21].

3.2 Shared Protokoll

Die Teilnehmer des Shared-Protokolls sind neben dem Benutzer, bzw. in unserem Fall dessen Rollenproxy, zwei verschiedene Schlüssel- bzw. Key-Server und eine Datenbank. Jeder Teilnehmer hat sein von der RootCA unterschriebenes Zertifikat und den dazu passenden geheimen Schlüssel sowie das Zertifikat der RootCA selbst, um andere Zertifikate auf Gültigkeit hin zu überprüfen. Die einzelnen Komponenten des Protokolls sind in Abbildung 3.1 dargestellt.

Die grundsätzliche Idee des Shared Protokoll ist, daß der für den Zugriff auf geheime Daten notwendige Schlüssel auf zwei verschiedene Schlüsselserver *Schlüssel-Server1* und *Schlüsselserver2* aufgeteilt wird. Der Datensatz soll nur genau dann erfolgreich gelesen werden können, wenn beide Teil-Schlüssel vorliegen. Dies ver-

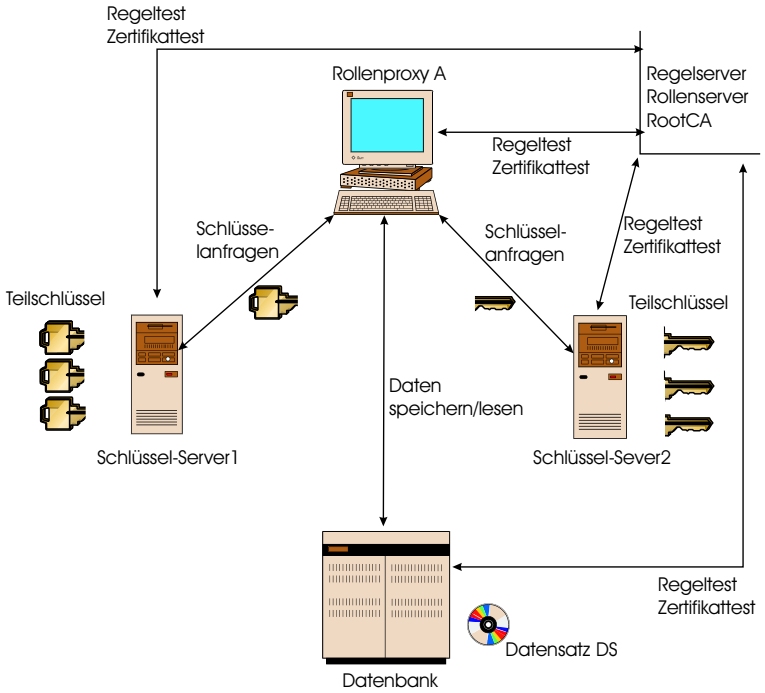

Abbildung 3.1: Übersicht Shared Protokoll

hindert einen zu mächtigen Schlüsselserver, der durch den universalen Zugriff auf alle Schlüssel jeden Datensatz aus der Datenbank entschlüsseln und lesen könnte. Das Aufteilen des Schlüssels in zwei gleiche Teile kann mit der Technik der Shared-Secrets realisiert werden, wie sie in Abschnitt 2.3 beschrieben sind. Der Protokollablauf beim Einfügen eines Datensatzes in die Datenbank gestaltet sich damit wie folgt (vgl. Abbildung 3.1):

1. Der Rollenproxy A holt und überprüft das Zertifikat von Schlüssel-Server1.

2. A verschlüsselt die Anfrage zum Speichern eines bestimmten Datensatzes DS und verschickt diese an Schlüssel-Server1. Dir Anfrage ist nur für Schlüssel-Server1 lesbar.

3. Schlüssel-Server1 generiert und speichert einen Teilschlüssel zum Verschlüsseln von DS und schickt diesen zurück an A, wiederum verschlüsselt und nur für A lesbar .

4. Analog holt und überprüft A das Zertifikat von Schlüssel-Server 2.

5. A verschlüsselt wiederum eine Anfrage nach einem Teilschlüssel für DS und schickt diese an Schlüssel-Server2.

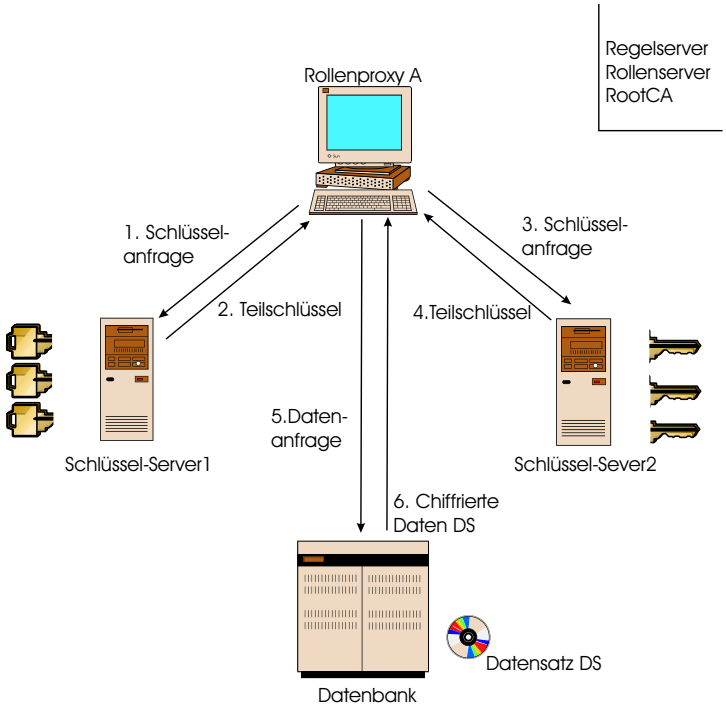

Abbildung 3.2: Reihenfolge der Nachrichten im Shared Protokoll

6. Schlüssel-Server2 generiert einen Teilschlüssel und schickt diesen nur für A lesbar zurück an A.

7. A setzt die erhaltenen Teilschlüssel zusammen und kann nun damit DS chiffrieren.

8. A schickt das bereits chiffrierte DS an die Datenbank, welche DS speichert.

Der Lese-Zugriff auf die Datenbank erfolgt analog. Zunächst fragt der Rollenproxy die beiden Schlüssel-Server nach den Schlüsselteilen und fordert dann den verschlüsselten Datensatz von der Datenbank.

In Abbildung 3.2 sind die einzelnen Nachrichten, die im Protokoll verschickt werden, nochmals in ihrer Reihenfolge angegeben. Weggelassen wurde das Austauschen der Zertifikate, da dies für den eigentlichen Protokollablauf keine Rolle spielt.

3.2.1 Zertifikate

Da alle Zertifikate von der RootCA unterschrieben worden sind und jeder Rollenproxy im Besitz des RootCA-Zertifikats ist, sind *Man in the Middle*-Attacken (siehe Abbildung 3.3) wie nachfolgend beschrieben unmöglich. Der Angreifer fängt

das Zertifikat, das der Schlüssel-Server an den Rollenproxy schicken will, ab und schickt stattdessen sein eigenes. Empfängt der Rollenproxy das Zertifikat des Angreifers und glaubt, daß dieses eigentlich vom Schlüssel-Server stammt, so ist damit der Angreifer in der Lage, sämtliche Kommunikationsvorgänge zwischen Schlüssel-Server und Rollenproxy abzuhören und sogar zu verändern. Verschlüsselt der Rollenproxy eine Anfrage nur für den Schlüssel-Server, so verschlüsselt er dies in Wirklichkeit für den Angreifer, der diese Anfrage entschlüsseln, mitlesen und verändert an den Schlüssel-Server weiterschicken kann. Der Schlüssel-Server, der davon ausgeht, daß der Angreifer der legitime Rollenproxy ist, antwortet dann dem Angreifer in seiner Antwort. Der Angreifer kann dieses wiederum mitlesen, verändern und an den Rollenproxy weiterschicken.

Diese Attacke scheitert aber daran, daß sich der Rollenproxy vergewissert, ob das empfangene Zertifikat von der RootCA auf den Schlüssel-Server ausgestellt wurde. Das Zertifikat muß eine gültige Unterschrift der RootCA enthalten. Diese Unterschrift ist mit dem Zertifikat der RootCA überprüfbar, das jedem Rollenproxy zu Beginn des Protokollablaufs vorliegt. Daher ist er gegen die beschriebene Attacke sicher. Wie das RootCA-Zertifikat zu den einzelnen Rollenproxys gelangt, ist hingegen problematisch. Sobald es einfach verschickt wird, verlagert sich das Problem der *Man in the Middle*-Attacke lediglich auf diesen Bereich des Protokolls. Eine einfache Lösung dafür ist, daß jeder Rollenproxy beim einmaligen Registrieren seiner Person, d. h. beim Ausstellen eines Zertifikats auf seinen Namen, das RootCA-Zertifikat über einen unabhängigen Kanal erhält, z. B. von der ausstellenden Zertifizierungsstelle auf Diskette. Auf diese Weise wird sichergestellt, daß ein Rollenproxy mit dem entsprechenden Schlüssel-Server oder der Datenbank kommuniziert. Umgekehrt stellt sich die Frage, wie sich ein Rollenproxy einem Schlüssel-Server gegenüber authentifiziert. Dazu muß der Rollenproxy seine Anfragen mit seinem privaten Schlüssel unterschreiben, um damit seine Absicht (engl. *intention*) zu bekräftigen. Diese Absicht können der Schlüssel-Server und die Datenbank mit dem Zertifikat des Rollenproxys überprüfen, welches dieser den einzelnen Anfragen beifügt. Da sowohl der Schlüssel-Server als auch der Datenbank-Server eine Rechteüberprüfung über einen Rechte-Server durchführen (siehe Abschnitt 1.2), sind weder ein falsches Zertifikat, noch der daraus resultierende *Man in the Middle*-Angriff kritisch. Der Angreifer würde nur an Daten gelangen, die er aufgrund seiner Rechte legal einsehen könnte.

3.2.2 Formalisierte Nachrichten

Die eigentliche Analyse des Shared Protokoll mit Hilfe der BAN-Logik kann auf einen Schlüssel-Server reduziert durchgeführt werden, da man für die formale Verifikation davon ausgeht, daß die Technik der Shared-Secrets sicher ist. Inwieweit der Mechanismus des Zusammensetzens und Verbergens von einzelnen Teilgeheimnissen sicher ist, wurde bereits in Abschnitt 2.3 erläutert. Im Folgenden

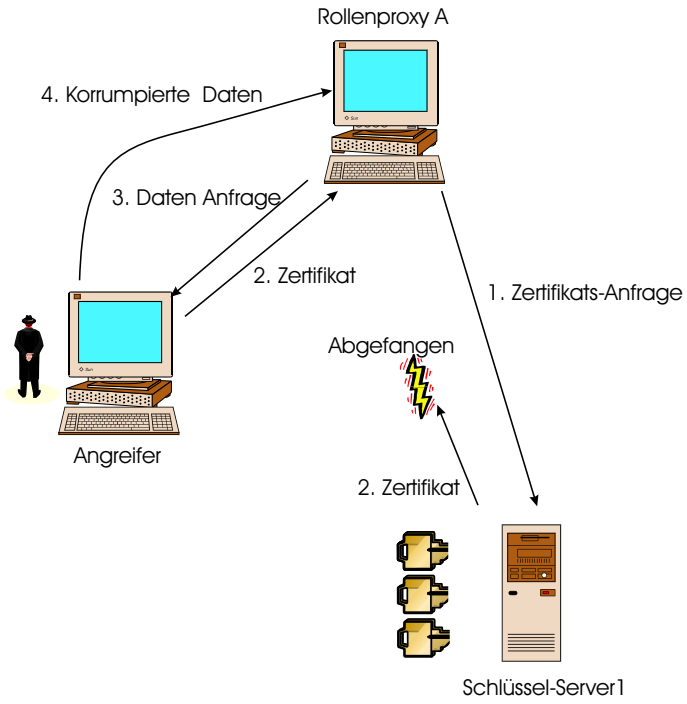

Abbildung 3.3: Man in the Middle

sollen die in Abbildung 3.2 dargestellten Nachrichten formalisiert werden.

Initiale Annahmen

Zunächst werden dazu die initialen Annahmen spezifiziert.

A und B seien die Rollenproxys des Protokolls. A legt einen Datensatz in der Datenbank ab, und B will diesen Datensatz danach lesen. Das Prädikat $has(A, X)$ bedeutet, daß A im Besitz einer Bitfolge X ist. Um X verschicken zu können, muß für einen Sender S $has(S, X)$ gelten. Das Schlüsselwort any bezeichnet alle Protokollteilnehmer. So bedeutet z. B. das Prädikat $believes(any, X)$, daß alle Protokollteilnehmer an X glauben.

Damit ergeben sich die Annahmen:

$$
\begin{aligned}
&inv(K_A, K_A^{-1}),\\
&believes(A, fresh(K_A^{-1})),\\
&believes(any, public\_key(K_A, A)),\\
&sees(any, public\_key(K_A, A)),\\
&believes(any, fresh(K_A)).
\end{aligned}
$$

Rollenproxy A ist im Besitz seines öffentlichen Schlüssels K_A und eines dazu pas-

senden geheimen Schlüssels K_A^{-1}. Alle glauben, daß K_A der öffentliche Schlüssel von A ist, und alle kennen K_A. K_A ist außerdem für alle *aktuell*. K_A^{-1} ist für A aktuell.
Die Korrektheit dieses Zustands wird durch Zertifikate erreicht. Alle Teilnehmer glauben an den öffentlichen Schlüssel eines Rollenproxys, weil er Teil eines von der RootCA unterschriebenen Zertifikats ist. Der Schlüssel ist aktuell, da Zertifikate üblicherweise mit einem Verfallsdatum ausgestattet sind und nach dessen Ablauf ihre Gültigkeit verlieren.

Vergleichbare Annahmen gelten für den zweiten Rollenproxy B:

$$
\begin{aligned}
&inv(K_B, K_B^{-1}),\\
&believes(B, fresh(K_B^{-1})),\\
&believes(any, public\_key(K_B, B)),\\
&sees(any, public\_key(K_B, B)),\\
&believes(any, fresh(K_B)),
\end{aligned}
$$

den betrachteten Schlüssel-Server KMS:

$$
\begin{aligned}
&believes(any, public\_key(K_{KMS}, KMS)),\\
&sees(any, public\_key(K_{KMS}, KMS)),\\
&believes(any, fresh(K_{KMS}))
\end{aligned}
$$

und die beteiligte Datenbank DB:

$$
\begin{aligned}
&believes(any, public\_key(K_{DB}, DB)),\\
&sees(any, public\_key(K_{DB}, DB)),\\
&believes(any, fresh(K_{DB})).
\end{aligned}
$$

Aus einem Prädikat der Form $public\_key(K, A')$ folgt immer, daß A' auch im Besitz des geheimen Schlüssels passend zu K ist. Der geheime Schlüssel wird allerdings nur dann explizit benannt, wenn er gebraucht wird. Da sowohl Datenbank als auch Schlüssel-Server den Schlüssel außer zum impliziten Entschlüsseln nicht brauchen, wird er eben nicht formalisiert. A und B müssen allerdings mit ihrem geheimen Schlüssel Signaturen anfertigen, wie später noch gezeigt wird.

Protokollablauf

Basierend auf den beschriebenen Annahmen lassen sich nun die einzelnen Nachrichten formalisieren. Während des Protokollablaufs haben die Nachrichten die Form $A \rightarrow B : X$. Dies bedeutet, daß Rollenproxy A die Nachricht X an B verschickt.

Der nachfolgenden Aufzählung der verschickten Nachrichten ist dieses Schema zugrundegelegt:

- formalisierte Nachricht und deren Beschreibung
- vorgeschlagene Annahmen zum Erreichen der Ehrlichkeits-, Geheimhaltungs- und Durchführbarkeits-Anforderungen und deren Beschreibung
- explizite Interpretationsregel und deren Beschreibung

Zu den Nachrichten:

1. *Nachricht*:
   $A \rightarrow KMS : encrypt(encrypt(sessionkey, K_{KMS}), K_A^{-1})$

   A schickt an den Schlüssel-Server eine Nachricht mit einem Sitzungsschlüssel $sessionkey$, der zum weiteren Verschlüsseln zwischen A und KMS verwendet wird. Da die eigentliche Schlüsselanfrage von A an KMS genauso geheim wie der Sitzungsschlüssel $sessionkey$ für die Übertragung ist aber keinerlei Bedeutung für den Protokollablauf hat, kann sie weggelassen werden. A unterzeichnet die Nachricht, damit KMS sicher sein kann, daß diese Nachricht auch wirklich von A stammt. Sie wird mit dem öffentlichen Schlüssel von KMS verschlüsselt, so daß nur KMS sie lesen kann.

   Annahmen:
   Um diese Nachricht erfolgreich, also glaubhaft und sicher, verschicken zu können, sind folgende Annahmen nötig:

   $has(A, sessionkey)$,
   $believes(A, fresh(sessionkey))$,
   $sees(A, shared\_key(sessionkey, A, KMS))$,
   $believes(A, shared\_key(sessionkey, A, KMS))$,
   $believes(KMS, controls(A, shared\_key(sessionkey, A, KMS)))$,
   $believes(KMS, fresh(shared\_key(sessionkey, A, KMS)))$

   A ist im Besitz und glaubt an den geheimen Schlüssel $sessionkey$ zwischen A und KMS. KMS vertraut A, daß dieser gemeinsame geheime Schlüssel erstellt. Außerdem glaubt KMS, daß die Sitzungsschlüssel alle noch gültig sind. Dies kann u. a. erreicht werden, indem dem Sitzungsschlüssel ein Zeitstempel beigefügt wird.

   Interpretationsregel:
   Die passende explizite Interpretationsregel für die Ehrlichkeits-Überprüfung lautet:

   $$\frac{believes(KMS, said(A, encrypt(Key, K_{KMS})))}{believes(KMS, said(A, shared\_key(Key, A, KMS)))}$$

   Empfängt KMS einen Schlüssel von A, so wird dieser als gemeinsamer geheimer Schlüssel interpretiert.

Außerdem bleibt an dieser Stelle noch anzumerken, daß die Kommunikation zwischen KMS und A chiffriert sein muß, damit ein Big-Brother nicht beide Teilschlüssel hintereinander abfangen und so das Shared-Secret zusammensetzen kann.

2. *Nachricht:*
   $KMS \rightarrow A : encrypt(datakey, sessionkey)$

   KMS schickt den Schlüssel *datakey* zum Verschlüsseln des Datensatzes an A. Der Schlüssel *datakey* ist durch den Sitzungsschlüssel *sessionkey* chiffriert. Da außer KMS nur A den Sitzungsschlüssel kennt, kann A erkennen, daß diese Nachricht vom Schlüssel-Server stammt.

   Annahmen:

   $$\begin{aligned}
   &believes(KMS, maysee(A, datakey)),\\
   &believes(KMS, maysee(KMS, datakey)),\\
   &believes(KMS, believes(A, shared_key(sessionkey, A, KMS)))\\
   &has(KMS, datakey),\\
   &believes(KMS, fresh(datakey)),\\
   &believes(KMS, shared_key(datakey, A, B)),\\
   &believes(A, controls(KMS, shared_key(datakey, A, B))),\\
   &believes(A, fresh(shared_key(datakey, A, B)))
   \end{aligned}$$

   KMS geht also davon aus, daß A den Schlüssel *sessionkey* als gemeinsamen geheimen Schlüssel betrachtet. Damit ist diese Nachricht ehrlich und geheim.

   Interpretationsregel:

   $$\frac{believes(A, said(KMS, Key))}{believes(A, said(KMS, shared\_key(Key, A, B)))}$$

   A interpretiert somit *datakey* als gemeinsamen geheimen Schlüssel zwischen A und B.

3. *Nachricht:*
   $A \rightarrow DB : encrypt(encrypt(adbsessionkey, K_{DB}), K_A^{-1})$

   Das Verschicken des Datensatzes von A an die Datenbank DB muß in zwei Teilen geschehen. Zunächst wird (in dieser Nachricht) der Sitzungsschlüssel *adbsessionkey* zwischen A und DB ausgetauscht, dann der verschlüsselte Datensatz, der diesmal für das Protokoll von Bedeutung ist. A signiert diese Nachricht, so daß die Datenbank überprüfen kann, daß tatsächlich A sie versandt hat.

Annahmen:

$has(A, adbsessionkey)$,
$believes(A, fresh(adbsessionkey))$,
$sees(A, shared\_key(adbsessionkey, A, DB))$,
$believes(A, shared\_key(adbsessionkey, A, DB))$,
$believes(DB, controls(A, shared\_key(adbsessionkey, A, DB)))$,
$believes(DB, fresh(shared\_key(adbsessionkey, A, DB)))$.

Sowohl A als auch DB akzeptieren *adbsessionkey* als gemeinsamen Sitzungsschlüssel. Dabei gesteht die Datenbank DB dem Rollenproxy A zu, daß er *adbsessionkey* generieren kann.

Interpretationsregel:

$$\frac{believes(DB, said(A, encrypt(key, K_{DB})))}{believes(DB, said(A, shared\_key(key, A, DB)))}$$

Empfängt DB von A einen Schlüssel, so wird dieser als gemeinsamer Schlüssel interpretiert.

4. *Nachricht:*
   $A \rightarrow DB : encrypt(encrypt(dataset, datakey), adbsessionkey)$

   Erst jetzt schickt A den verschlüsselten Datensatz an die Datenbank. Er wird nochmals mit dem gerade ausgetauschten Sitzungsschlüssel verschlüsselt, damit die Datenbank erkennen kann, daß die Nachricht von A gesendet wurde. Es würde ausreichen, diese Nachricht nur mit dem geheimen Schlüssel von A zu signieren, jedoch würde es einem Angreifer ermöglichen zu erkennen, daß A den Datensatz gegebenenfalls auch an weitere Datenbanken schickt. Der in der Datenbank gespeicherte Datensatz wird durch $encrypt(dataset, datakey)$ repräsentiert.

   Annahmen:

   $has(A, dataset)$,
   $believes(A, fresh(dataset))$,
   $believes(A, maysee(A, dataset))$,
   $believes(A, maysee(B, dataset))$,
   $believes(A, believes(DB, shared\_key(adbsessionkey, A, DB)))$,
   $believes(A, encrypt(dataset, datakey))$.

   A ist der Auffassung, daß B den Datensatz lesen darf und außerdem, daß die Datenbank *adbsessionkey* als gemeinsamen geheimen Schlüssel akzeptiert hat. Letzteres muß nicht unbedingt stimmen, denn A hat keine Bestätigung, daß DB diesen Schlüssel so akzeptiert. Dies stellt jedoch kein Problem dar.

Interpretationsregel:

$$\frac{sees(DB, encrypt(encrypt(data, datakey), adbsessionkey))}{sees(DB, encrypt(data, datakey))}$$

Die Datenbank interpretiert die empfangene Nachricht nicht weiter sondern nimmt sie nur zur Kenntnis.

5. *Nachricht:*
   $B \rightarrow KMS : encrypt(encrypt(sessionkey2, K_{KMS}), K_B^{-1})$

   Analog zur Nachricht in Schritt 1 schickt jetzt B eine Schlüsselanfrage an den Schlüsselserver. B möchte den entsprechenden Teilschlüssel zum späteren Dekodieren des Datensatzes. Auch hier wird die Nachricht nur durch das Versenden des Sitzungsschlüssels $sessionkey2$ repräsentiert.

   Annahmen:

   $$\begin{aligned}
   &has(B, sessionkey2)\,,\\
   &believes(B, fresh(sessionkey2))\,,\\
   &sees(B, shared_key(sessionkey2, B, KMS))\,,\\
   &believes(B, shared_key(sessionkey2, B, KMS))\,,\\
   &believes(KMS, controls(B, shared_key(sessionkey2, B, KMS)))\,,\\
   &believes(KMS, fresh(shared_key(sessionkey2, B, KMS)))
   \end{aligned}$$

   Ebenso glaubt B an die Gültigkeit des gemeinsamen geheimen Schlüssels $sessionkey2$ zwischen B und KMS. KMS vertraut B, daß dieser den Sitzungsschlüssel $sessionkey2$ korrekt generiert.

   Interpretationsregel:

   $$\frac{believes(KMS, said(B, encrypt(key, K_{KMS})))}{believes(KMS, said(B, shared\_key(key, B, KMS)))}$$

   Glaubt der Schlüssel-Server, daß B einen Schlüssel key geschickt hat, so interpretiert er diesen als gemeinsamen geheimen Sitzungsschlüssel.

6. *Nachricht:*
   $B \rightarrow KMS : encrypt(datakey, sessionkey2)$

   Der Schlüssel-Server antwortet B mit dem chiffrierten $datakey$.

   Annahmen:

   $$\begin{aligned}
   &believes(KMS, maysee(B, datakey))\,,\\
   &believes(KMS, believes(B, shared_key(sessionkey2, B, KMS)))\,,\\
   &believes(B, controls(KMS, shared_key(datakey, A, B)))\,,\\
   &believes(B, fresh(shared_key(datakey, A, B)))
   \end{aligned}$$

KMS geht davon aus, daß B auch an den geheimen Sitzungsschlüssel $sessionkey2$ glaubt. Außerdem darf B den $datakey$ sehen. Auf der anderen Seite glaubt B, daß KMS in der Lage ist, ein solches $datakey$ zu senden.

Interpretationsregel:

$$\frac{believes(B, said(KMS, key))}{believes(B, said(KMS, shared\_key(key, A, B)))}$$

Wenn B glaubt, daß der Schlüssel-Server ihm ein Datum key zuschickt, dann interpretiert er key als das Shared-Secret, mit dem der Datensatz entschlüsselt wird. Das Shared-Secret wird hier als gemeinsamer geheimer Schlüssel zwischen A und B formalisiert.

7. *Nachricht:*
   $B \rightarrow DB : encrypt(encrypt(bdbsessionkey, K_{DB}), K_B^{-1})$

   Der Rollenproxy B fordert nun von der Datenbank den Datensatz an. Hierzu wird wiederum ein Sitzungsschlüssel $bdbsessionkey$ ausgetauscht.

   Annahmen:

   $has(B, bdbsessionkey)$,
   $believes(B, fresh(bdbsessionkey))$,
   $believes(B, shared\_key(bdbsessionkey, B, DB))$,
   $sees(B, shared\_key(bdbsessionkey, B, DB))$,
   $believes(DB, controls(B, shared\_key(bdbsessionkey, B, DB)))$,
   $believes(DB, fresh(shared\_key(bdbsessionkey, B, DB)))$

   Es gelten erneut die Annahmen für den Austausch von Sitzungsschlüsseln. DB glaubt, daß B einen solchen generieren kann und daß dieser aktuell ist.

   Interpretationsregel:

$$\frac{believes(DB, said(B, encrypt(key, K_{DB})))}{believes(DB, said(B, shared\_key(key, B, DB))}$$

   Die Datenbank interpretiert einen von B empfangenen Schlüssel als entsprechenden Sitzungsschlüssel zwischen B und DB.

8. *Nachricht:*
   $DB \rightarrow B : encrypt(encrypt(dataset, datakey), bdbsessionkey)$

   Schließlich schickt die Datenbank den durch den ausgetauschten Sitzungsschlüssel $bdbsessionkey$ verschlüsselten Datensatz
   $encrypt(dataset, datakey)$ an Rollenproxy B. Diese zusätzliche Verschlüsselung ist notwendig, um die Anforderung nach Geheimhaltung zu erfüllen: Da die in $dataset$ übertragenen Daten mit $datakey$ chiffriert sind, könnte die Nachricht von DB zwar lediglich unterzeichnet und damit auf die

weitere Verschlüsselung verzichtet werden. Jedoch wäre es einem Angreifer auf diese Weise möglich nachzuvollziehen, an welche verschiedenen Rollenproxys das Datum *encrypt*(*dataset*, *datakey*) geschickt wird und folglich Rückschlüsse auf die Wichtigkeit dieses Datensatzes zu ziehen.

Annahmen:

$$believes(DB, encrypt(dataset, datakey)),$$
$$believes(DB, maysee(B, encrypt(dataset, datakey))),$$
$$believes(DB, maysee(DB, encrypt(dataset, datakey)))$$

Interpretationsregel:

$$\frac{sees(B, encrypt(encrypt(dataset, datakey), bdbsessionkey))}{sees(B, dataset)}$$

Für diese Interpretation kann die Datenbank jedoch keinerlei Verantwortung übernehmen, da diese den Datensatz nicht entschlüsseln kann. Es ist zu erwarten, daß der *RVChecker* hier die Ehrlichkeit bemängeln wird.

3.3 Zyklisches Protokoll

Am Zyklischen Protokoll nehmen der durch den Rollenproxy vertretene Benutzer, eine *Transferstation* sowie eine Datenbankstation teil. Der Begriff *Station* soll hier im Gegensatz zum herkömmlichen *Server* verwendet werden, da Anfragen an Stationen gesendet, dort abgearbeitet und abschließend an weitere Instanzen weitergeleitet werden. Station sind abgeschlossene unabhängige logische Einheiten. Ob eine Station einen physikalischen Rechner darstellt, oder ob mehrere Stationen auf einem solchen Rechner parallel arbeiten, bleibt offen.

Wie beim Shared Protokoll ist jeder Teilnehmer im Besitz seines von der obersten Zertifizierungsstelle (RootCA) unterschriebenen Zertifikats, dem dazu passenden geheimen Schlüssel und dem Zertifikat der RootCA selbst zum Überprüfen anderer Zertifikate. Der Aufbau des Zyklischen Protokolls ist in Abbildung 3.4 dargestellt.

Beim Zyklischen Protokoll wird die Sicherheit der Daten durch Mehrfach- bzw. Umverschlüsselung der Daten erreicht. Die Aufgabe des Umverschlüsselns übernimmt dabei die Transferstation. So wird beispielsweise bei einer Lese-Transaktion vom Benutzer bzw. dessen Rollenproxy der Protokollablauf initiiert, indem der Rollenproxy eine Anfrage an die Datenbank stellt. Die Datenbank schickt die Daten zum Umverschlüsseln weiter an die Transferstation und diese zurück an

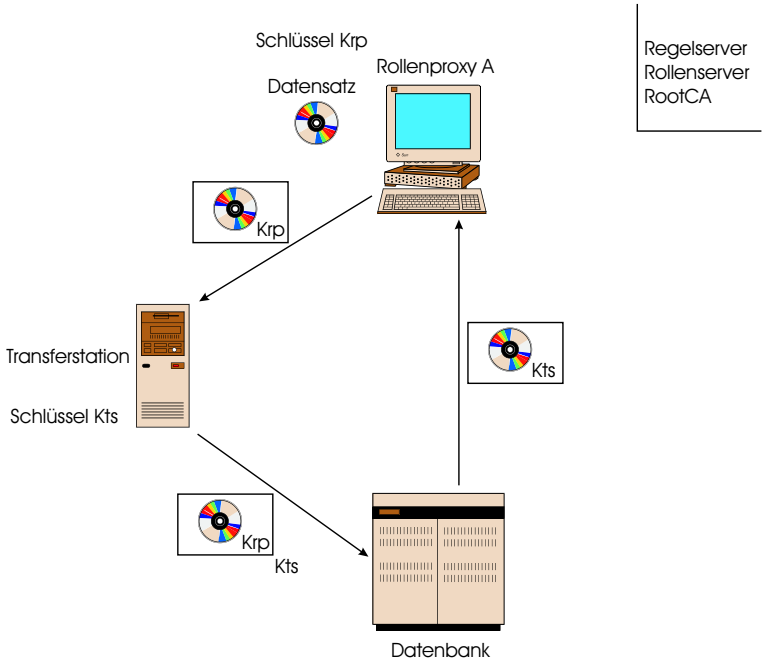

Abbildung 3.4: Übersicht Zyklisches Protokoll

den Rollenproxy. Umgekehrt schreibt ein Rollenproxy ein Datum in die Datenbank, wobei er den Datensatz zunächst an die Transferstation schickt, die ihn dann umverschlüsselt und weiter an die Datenbank leitet. Ein Datensatz läuft in beiden Fällen *zyklisch* in einem *Workflow* vom Initiator über mehrere Stationen hinweg zurück zum Initiator.

3.3.1 Ablauf

Der genaue Ablauf beim Einfügen eines Datensatzes durch einen Benutzer bzw. dessen Rollenproxy A hat folgende Gestalt:

1. A holt und überprüft die Zertifikate der Transferstation TS und der Datenbank DB.

2. A verschlüsselt den zu speichernden Datensatz mit dem geheimen Schlüssel K_{RP} und schickt das erzeugte Chiffrat an die Transferstation TS. Bisher kennt nur A den Schlüssel K_{RP}.

3. Die Transferstation TS holt und überprüft das Zertifikat der Datenbank DB. TS verschlüsselt das von A empfangene Chiffrat weiter mit ihrem geheimen Schlüsse K_{TS} und schickt das Ergebnis-Chiffrat wiederum weiter an die Datenbank DB. Nur TS kennt den Schlüssel K_{TS}.

4. A schickt den Schlüssel K_{RP} an die Datenbank DB. Diese kann den mit Schlüsseln K_{RP} und K_{TS} verschlüsselten Datensatz mit Hilfe des empfangenen Schlüssels K_{RP} derart entschlüsseln, daß der Datensatz nur noch mit dem Schlüssel K_{TS} chiffriert vorliegt.

Eine genaue Betrachtung des Protokolls macht deutlich, daß der Rollenproxy A den Schlüssel K_{RP} in Schritt 4 mit einem geheimen Sitzungsschlüssel zwischen A und DB chiffrieren muß. Eine kompromittierte Transferstation TS, die den Datenverkehr zwischen den Teilnehmern abhört, könnte sonst in den Besitz von K_{RP} gelangen. TS wäre dann in der Lage, das in Schritt 2 empfangene Chiffrat zu entschlüsseln, und erhielte somit den Original-Datensatz. Dasselbe gilt für eine kompromittierte Datenbank. Genauso muß der in Schritt 2 verschickte und nur mit dem Schlüssel K_{RP} verschlüsselte Datensatz noch zusätzlich mit einem Sitzungsschlüssel zwischen A und TS verschlüsselt werden, da sonst eine kompromittierte Datenbank DB beim Empfang des Schlüssels K_{RP} in Schritt 4 den in Schritt 2 empfangenen Datensatz rekonstruieren könnte.

Das Auslesen eines Datensatzes, der wie im vorigen Abschnitt beschrieben, gespeichert wurde, kann für einen Benutzer bzw. dessen Rollenproxy B mit Hilfe der folgenden Protokollschritte durchgeführt werden. Der Einfachheit wegen wurde hier auf die Beschreibung des Austausches und die Überprüfung von Zertifikaten verzichtet.

1. B kennt einen eigenen geheimen Schlüssel $K_{RP'}$ und schickt ihn an die Datenbank DB.

2. DB verschlüsselt den gespeicherten und bisher nur mit dem Schlüssel K_{TS} verschlüsselten gespeicherten Datensatz zusätzlich mit dem empfangenen Schlüssel $K_{RP'}$ und schickt das Chiffrat an die Transferstation TS.

3. TS kennt den Schlüssel K_{TS} und dechiffriert das durch $K_{RP'}$ und K_{TS} verschlüsselte Chiffrat zu einem rein durch $K_{RP'}$ verschlüsselten Chiffrat. Dieses wird an B weitergeschickt.

4. B kann nun den Datensatz entschlüsseln, da er im Besitz des Schlüssels $K_{RP'}$ ist.

Analog zum Speichervorgang in Schritt 3 muß die Übertragung zusätzlich durch einen geheimen Sitzungsschlüssel zwischen B und TS gesichert werden, da sonst eine kompromittierte Datenbank DB, die in Schritt 1 in Besitz des Schlüssels $K_{RP'}$ kommt, den Datensatz entschlüsseln könnte. Der Schlüssel $K_{RP'}$ aus Schritt 1 muß ebenso verschlüsselt werden, damit eine kompromittierte Transferstation in Schritt 3 den Datensatz nicht dechiffrieren kann.

Aus der Beschreibung des Protokolls wird ersichtlich, daß zum Verschlüsseln der Daten mit den Schlüsseln K_{TS} und $K_{RP'}$ eine kommutative Chiffre, wie sie in Ab-

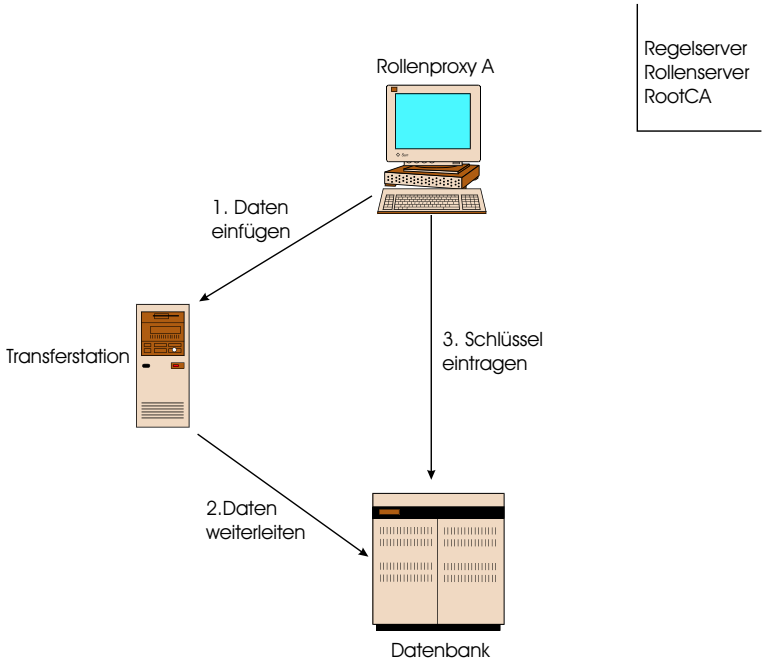

Abbildung 3.5: Reihenfolge der Nachrichten im Zyklischen Protokoll

schnitt 2.4 beschrieben wurde, nötig ist. Wenn der Versand der Nachricht aus Schritt 4 beim Speichern bzw. Schritt 3 beim Lesen mit einer zusätzlichen Chiffre geschützt wird, so ist dies eine wirksame Verteidigung gegen den 3-Wege-XOR-Angriff aus Abschnitt 2.4.2. Demzufolge kann XOR mit One-Time-Pad-Schlüsseln als kommutative Chiffre in diesem Protokoll sicher eingesetzt werden.

3.3.2 Formalisierte Nachrichten

Wie beim Shared Protokoll werden in Abbildung 3.5 die einzelnen Protokoll-Nachrichten ohne den notwendigen Austausch der Zertifikate dargestellt. Dies spielt keine Rolle für die Analyse des Protokolls und ist nur für die spätere Implementierung wichtig, damit *Man in the Middle* Attacken aus Abschnitt 3.2.1 verhindert werden.

Wieder sind A und B die Benutzer bzw. die Rollenproxys des Protokolls. A legt einen neuen Datensatz in der Datenbank ab, den B hinterher ausliest. Zunächst folgen die initialen Annahmen des Zyklischen Protokolls in der Notation der RV-Logik.

$$sees(any, public\_key(K_{DB}, DB)),$$
$$believes(any, public\_key(K_{DB}, DB)),$$
$$believes(any, fresh(K_{DB})),$$

$has(A, K_{RP})$,
$believes(A, fresh(K_{RP}))$,
$has(A, dataset)$,
$has(TS, K_{TS})$,
$believes(TS, fresh(K_{TS}))$,
$believes(TS, shared\_key(K_{TS}, TS, TS))$,
$has(B, K_{RP'})$,
$sees(B, shared\_key(K_{RP'}, B, DB))$
$sees(any, public\_key(K_B, B))$,
$believes(any, public\_key(K_B, B))$,
$believes(any, fresh(K_B))$,

Der öffentliche Schlüssel der Datenbank ist jedem bekannt und jeder glaubt an dessen Gültigkeit. Dasselbe gilt für den *public-key* von B. Die Zertifikate der anderen Protokollteilnehmer spielen hier keine Rolle. Bereits beim Shared Protokoll wurde die Korrektheit des Mechanismus bewiesen, bei dem mit Hilfe von öffentlichen Schlüsseln Sitzungsschlüssel geheim übertragen werden können, siehe Abschnitt 3.2.2. Wie später noch gezeigt wird, ist hier nur beim Verschicken des Schlüssels K_{RP} vom Rollenproxy A an die Datenbank ein Sitzungsschlüssel nötig. Daher wird dieser durch das Chiffrieren mit dem öffentlichen Schlüssel K_{DB} ersetzt.

A ist im Besitz des gültigen Schlüssels K_{RP} und eines zu speichernden Datensatzes *dataset*. Die Transferstation kennt den gültigen Schlüssel K_{TS} und interpretiert diesen als nur ihr bekannten geheimen Schlüssel.

B wiederum verfügt über den zweiten Schlüssel $K_{RP'}$. Dieser wird für den Lesezyklus benötigt. Außerdem sieht B diesen auch als gemeinsamen geheimen Schlüssel zwischen B und der Datenbank an.

Die einzelnen Nachrichten sind abermals von der Form $P \rightarrow Q : X$, d. h. Station A verschickt die Nachricht X an Station Q.

Hier folgen die einzelnen Nachrichten:

1. *Nachricht:*
   $A \rightarrow TS : encrypt(dataset, K_{RP})$

   Der Rollenproxy A schickt den Datensatz *dataset*, der in der Datenbank abgelegt werden soll, mit dem Schlüssel K_{RP} chiffriert an die Transferstation. In der Implementierung wird dieses Chiffrat nochmals mit einem Sitzungsschlüssel zwischen A und TS gesichert, damit eine eventuell kompromittierte und mitlesende Datenbank zu dem Zeitpunkt, bei dem sie von A den Schlüssel K_{RP} erhält, nicht den Datensatz entschlüsseln kann. Die Existenz solcher kompromittierten oder mitlauschenden Protokollteilnehmer kann durch die Logik des *RVCheckers* nicht ausgedrückt bzw. über-

prüft werden und wird daher im Folgenden nicht weiter beachtet. In Abschnitt 4.2.2 werden diese Fälle deshalb separat betrachtet.

Annahmen:
Die Voraussetzungen für Geheimhaltung, siehe Abschnitt 2.6.2, sind

$$believes(A, encrypt(dataset, K_{RP})),$$
$$believes(A, maysee(A, dataset)),$$
$$believes(A, shared\_key(K_{RP}, A, DB)),$$
$$believes(A, shared\_key(K_{RP}, A, A)).$$

Das Prädikat $believes(A, maysee(B, dataset))$ ist hier oder später beim Empfang des Chiffrats bei B für die Geheimhaltung nicht notwendig.

Interpretationsregel:

$$\frac{sees(TS, encrypt(dataset, K_{RP}))}{sees(TS, encrypt(dataset, K_{RP}))}.$$

Dies bedeutet, daß TS das Chiffrat nicht weiter interpretiert bzw. nicht weiter interpretieren kann. Daher muß A nur an das Chiffrat glauben (siehe Annahmen), um die geforderte Ehrlichkeit zu erreichen.

2. *Nachricht:*
$TS \rightarrow DB : encrypt(encrypt(dataset, K_{RP}), K_{TS})$
Die Transferstation verschlüsselt das empfangene Chiffrat weiter mit dem Schlüssel K_{TS}. Danach wird das Ergebnis an die Datenbank geschickt.

Annahmen:

$$believes(TS, maysee(TS, encrypt(dataset, K_{RP}))),$$
$$believes(TS, shared\_key(K_{TS}, TS, TS))$$

Die Transferstation TS glaubt, daß K_{TS} ein „gemeinsamer“ Schlüssel ist. Dieser wird aber nur zum zum eigenen Ver- und Entschlüsseln verwendet. Genauso ist sie davon überzeugt, daß sie das Chiffrat sehen darf.

Interpretationsregel:

$$\frac{sees(DB, encrypt(encrypt(dataset, K_{RP}), K_{TS}))}{sees(DB, encrypt(encrypt(dataset, K_{RP}), K_{TS}))}.$$

Mit obigen Annahmen wird die Eigenschaft der Geheimhaltung zum Weitersenden an die Datenbank gewährleistet. Allerdings reichen die Angaben für eine Überprüfung auf Ehrlichkeit nicht aus, da die Datenbank das empfangene Chiffrat nicht entschlüsseln kann. Der *RVChecker* wird bei der Überprüfung eine entsprechende Meldung liefern (siehe Abschnitt 4.2), um die fehlende Ehrlichkeit zu bemängeln. Dies ist sinnvoll, da der Inhalt der von TS verschickten Nachricht in keinster Weise von TS überprüfbar ist. TS schickt diese Nachricht nur weiter (*forwarding*) an die Datenbank.

3. *Nachricht:*
   $A \rightarrow DB : encrypt(K_{RP}, K_{DB})$
   Nachdem die Datenbank von der Transferstation den mit den Schlüsseln K_{RP} und K_{TS} chiffrierten Datensatz empfangen hat, bekommt sie nun den noch ausstehenden Schlüssel K_{RP} vom Rollenproxy. Damit kann sie vom Chiffrat den Schlüssel K_{RP} entfernen und speichert nur noch $encrypt(dataset, K_{TS})$ ab. Da diese Nachricht verschlüsselt werden muß, wird sie mit dem öffentlichen Schlüssel der Datenbank chiffriert.

   Annahmen:

   $$believes(A, maysee(A, K_{RP})),$$
   $$believes(A, maysee(DB, K_{RP})).$$

   Zum Erreichen der Geheimhaltung muß A annehmen, daß neben ihm auch die Datenbank den geheimen Schlüssel K_{RP} sehen darf. Es wurde bereits beim Shared Protokoll gezeigt, daß mit Hilfe von richtigen Sitzungsschlüsseln und öffentlichen Schlüsseln Authentizität erreicht werden kann. Es sind keine weiteren Annahmen für die Ehrlichkeit nötig.

   Interpretationsregel:

   $$\frac{sees(DB, encrypt(K_{RP}, K_{DB}))}{sees(DB, shared\_key(K_{RP}, A, DB))}.$$

   Empfängt die Datenbank einen chiffrierten Schlüssel, so interpretiert sie diesen als geheimen gemeinsamen Schlüssel zwischen ihr und A.

   Damit ist das Einfügen eines Datensatzes in die Datenbank beendet.

4. *Nachricht:*
   $B \rightarrow DB : encrypt(K_{RP'}, K_{DB})$
   Rollenproxy B will jetzt den zuvor gespeicherten Datensatz aus der Datenbank auslesen. Dazu schickt er einen zweiten Schlüssel $K_{RP'}$ an die Datenbank. Hiermit wird der Lesezyklus initiiert. Der zweite Schlüssel $K_{RP'}$ selbst muß analog zu K_{RP} beim Schreibvorgang (s.o.) auch verschlüsselt werden. In der Implementierung geschieht dies mit einem Sitzungsschlüssel zwischen B und DB. Alternativ kann hierfür der öffentliche Schlüssel der Datenbank genutzt werden.

   Annahmen:

   $$believes(B, K_{RP'})$$
   $$believes(B, maysee(DB, K_{RP'})).$$

   B geht davon aus, daß die Datenbank den Schlüssel $K_{RP'}$ dechiffrieren und sehen kann. Das genügt zur Zusicherung von Ehrlichkeit für diese Nachricht.

   Interpretationsregel:

$$\frac{sees(DB, encrypt(K_{RP'}, K_{DB}))}{sees(DB, K_{RP'})}$$

Die Interpretationsregel ist an dieser Stelle wieder überflüssig, da sie selbst schon ein Teil des BAN-Regelwerks ist. Da der *RVChecker* jedoch zu jeder Nachricht eine explizite Angabe einer Interpretationsregel benötigt, muß die Regel beigefügt werden.

5. *Nachricht:*
   $DB \rightarrow TS : encrypt(encrypt(dataset, K_{RP'}), K_{TS})$
   Die Datenbank schickt den nun wieder mit zwei Schlüsseln gesicherten Datensatz *dataset* weiter an die Transferstation. Da keinerlei Verantwortung für den Inhalt der Nachricht übernommen wird, zumal die Datenbank den Inhalt von *dataset* gar nicht kennt bzw. mit dem verschlüsselten *dataset* nichts anfangen kann, ist Ehrlichkeit für diese Weiterleitung nicht erreichbar. Man beachte, daß die verschickte Nachricht zunächst mit $K_{RP'}$ und dann mit K_{TS} verschlüsselt ist und nicht umgekehrt, wie ursprünglich angenommen. Dies liegt an der Tatsache, daß der *RVChecker* die Eigenschaft zur Geheimhaltung im umgekehrten Falle nicht ableiten kann, bzw. umfangreichere Modifikationen als in Abschnitt 5.9 nötig sind.

   Annahmen:

   $$believes(DB, maysee(TS, encrypt(dataset, K_{RP'}))),$$
   $$believes(DB, shared\_key(K_{TS}, TS, TS)).$$

   DB ist der Überzeugung, daß die Transferstation das Chiffrat aus *dataset* und zweitem Schlüssel $K_{RP'}$ sehen darf - die Transferstation sollte im Besitz des Schlüssels K_{TS} sein. Das zweite Prädikat ist eine Formalisierung des Sachverhalts, daß der Schlüssel K_{TS} ein geheimer Schlüssel allein der Transferstation ist.

   Interpretationsregel:

   $$\frac{sees(TS, encrypt(encrypt(dataset, K_{RP'}), K_{TS}))}{sees(TS, encrypt(dataset, K_{RP'}))}.$$

   Die Interpretationsregel für diese Nachricht hat wieder keine richtige Aussage. Ehrlichkeit kann damit nicht nachgewiesen werden.

6. *Nachricht:*
   $TS \rightarrow B : encrypt(encrypt(dataset, K_{RP'}), K_B)$
   Die letzte Nachricht des Zyklischen Protokolls ist der Versand des Datensatzes von der Transferstation an Rollenproxy B. Formal gesichert ist der Datensatz hierbei durch den zweiten Schlüssel $K_{RP'}$ sowie den öffentlichen

Schlüssel von B. Die doppelte Sicherung ist wieder aus den oben angesprochenen Problemen notwendig. In der Implementierung wird anstatt des öffentlichen Schlüssels von B ein geheimer Sitzungsschlüssel zwischen B und der Transferstation verwendet. Da Rollenproxy B im Besitz dieses Schlüssels $K_{RP'}$ ist, sollte er damit *dataset* erfolgreich rekonstruieren können. Damit ist der Lesezyklus beendet. Da die Transferstation den Inhalt des versendeten Chiffrats nicht kennt, kann Ehrlichkeit abermals nicht erreicht werden.

Annahmen:

$$believes(TS, maysee(B, dataset)),$$
$$believes(TS, shared\_key(K_{RP'}, B, DB)).$$

Die Transferstation geht davon aus, daß B den Datensatz entschlüsseln kann, und daß der zweite Schlüssel $K_{RP'}$ geheim zwischen B und der Datenbank ist; damit wird die Geheimhaltung erreicht.

Interpretationsregel:

$$\frac{sees(B, encrypt(dataset, K_{RP'}))}{sees(B, encrypt(dataset, K_{RP'}))}.$$

Diese Interpretation ist wieder ohne Aussage.

3.4 Geschwindigkeit und Bewertung

Geschwindigkeit

Zum Vergleich der beiden Protokolle wurden Geschwindigkeitsmessungen an der Implementierung durchgeführt. Die Ergebnisse sind in Tabelle 3.1 abgebildet. Getestet wurden zwischen einem und 500 aufeinanderfolgenden Schreibzugriffen über das jeweilige Protokoll. Die Zeiten sind in Sekunden angegeben.

Jeder der Protokollteilnehmer lief auf einem eigenen, dedizierten Rechner, d. h. für das Shared Protokoll jeweils Rechner für die beiden Schlüsselserver, die Datenbank und den Benutzer. Für das Zyklische Protokoll standen sowohl der Transferstation, der Datenbank als auch dem Benutzer eigene Rechner zur Verfügung.

Interessant ist, daß bei beiden Protokollen Schreibvorgänge von einem bzw. zwei Durchläufen vergleichbar länger dauern, als die umfangreicheren Durchläufe, danach die Zeit aber linear zur Anzahl der Durchläufe steigt. Dies ist dadurch zu erklären, daß der Start der *Java-Virtual-Machine* und die einmaligen Initialisierung der Programme sehr rechenintensiv sind.

Durchläufe	Shared	Zyklisch
1	31s	23s
2	65s	43s
10	124s	72s
50	643s	361s
100	1358s	720s
500	6861s	3584s

Tabelle 3.1: Geschwindigkeit der beiden Protokolle

Bewertung

Das Zyklische Protokoll erweist sich als das klar schnellere von beiden. Bei größeren Durchläufen ist es fast doppelt so schnell wie das Shared Protokoll. Dies kann auf die geringere Anzahl von verschickten Nachrichten und die damit verbundenen Verschlüsselungsvorgänge zurückgeführt werden.

Bei gleichbleibender Anzahl von zu übertragenden Datensätzen aber steigender Datensatz-Größe ist mit umgekehrten Ergebnissen zu rechnen: Durch das zweimalige Verschlüsseln der Daten von Datenbank *und* Transferstation wird ein Protokolldurchlauf über das Zyklische Protokoll langsamer sein als über das Shared Protokoll. Meßergebnisse liegen dazu jedoch nicht vor.

Das Zyklische Protokoll erlaubt es, gegenüber dem Shared Protokoll zusätzliche Zwischenstationen - ähnlich der Transferstation - in Abläufe mit einzubeziehen. Diese Zwischenstationen könnten beispielsweise weitere Umverschlüsselungen durchführen.

Das Shared Protokoll ist bei Einsatz eines weiteren Schlüsselservers redundant gegen einen Ausfall eines Schlüsselservers gesichert. Die Technik der 2-aus-3-Schwellwert-Verfahren würde dies ermöglichen (siehe Abschnitt 2.3).

Momentan kann nur auf einzelne Datensätze zugegriffen werden. Eine gleichzeitige Auswahl mehrerer Datensätze zur Verringerung der Kommunikation ist noch nicht möglich, da dafür mehrere Datensätze mit einer Form von *Gruppenschlüssel* gleichzeitig chiffriert sein müßten. Es ist unklar, inwieweit dies die Sicherheit beeinträchtigt.

3.5 Update-Mechanismus

Der folgende Mechanismus ist Teil der beiden Protokolle aus Abschnitt 3.2 und Abschnitt 3.3. Er ist keine notwendige Eigenschaft oder Anforderung gewesen, sondern hat sich im Laufe der Implementierung als sehr sinnvoll erwiesen. Die Protokolle sind von diesem Mechanismus nicht abhängig.

Schreibt ein Benutzer bzw. dessen Rolle einen Datensatz in eine Datenbank hinein, so ist er oftmals daran interessiert zu erfahren, ob sich dieser Datensatz zu irgendeinem Zeitpunkt ändert. Dies ist der Fall, wenn er z. B. von einem anderen Benutzer geändert oder überschrieben wurde. Es ist grundsätzlich sinnvoll, daß sich Rollen für bestimmte Datensätze registrieren und eine Nachricht erhalten, wenn sich der Datensatz ändert (*Update*).

Der Update-Mechanismus besteht nur aus zwei Nachrichten. In der ersten Nachricht schickt Rollenproxy A die Bitte nach Registrierung für einen Datensatz DS an die Datenbank DB. Im Falle eines *Updates* von DS schickt DB an alle registrierten Rollenproxys eine Signal-Nachricht.

1. $A \rightarrow DB : encrypt(K_{A,DB}, K_{DB})$
   A schickt einen Sitzungsschlüssel, sowie den Namen des zu beobachtenden Datensatzes an die Datenbank. Chiffriert ist diese Anfrage mit dem öffentlichen Schlüssel der Datenbank. Für die Analyse unwichtige Informationen, wie der Name des Datensatzes, die Unterschrift der Rolle unter die Anfrage, oder die Unterschrift des Zugriffsserver fehlen.

2. $DB \rightarrow A : encrypt(dataset, K_{A,DB})$
   Dies ist die Nachricht, die im Falle einer Änderung des Datensatzes von der Datenbank an A geschickt wird. Der Name des Datensatzes wird verschlüsselt mit dem vorher ausgetauschten Sitzungsschlüssel.

Analytisch sind beide Nachrichten wenig anspruchsvoll. Wenn vorausgesetzt wird, daß alle Teilnehmer die Zertifikate der anderen Teilnehmer kennen, dann sind diese beiden Nachrichten sicher. Nachricht 1 wird durch den öffentlichen Schlüssel von DB geschützt. Nur DB kann das Chiffrat entschlüsseln und den geheimen Sitzungsschlüssel $K_{A,DB}$ lesen. Mit diesem Schlüssel wird dann Nachricht 2 geschützt, nur A kann den Datensatz dechiffrieren.

Die einzige Gefahr geht von einer kompromittierten Datenbank aus. Diese könnte einem registrierten Benutzer falsche Informationen über Updates senden. Die Geheimnisse in der Datenbank werden dadurch allerdings nicht offengelegt, da für den Lesezugriff des Datensatzes aus der Datenbank heraus immernoch der normale Protokollablauf nötig ist. Dieser Angriff ist eine DoS-Attacke (siehe Abschnitt 4.5).

Kapitel 4

Analyse

Wie in [31] beschrieben wird, ist es hilfreich, die Analyse eines kryptographischen Systems in mehrere logische Ebenen einzuteilen. Die einzelnen Ebenen beschreiben verschiedene Abstraktionsstufen bei der Entwicklung des Systems, in denen Fehler auftreten können (siehe Abbildung 4.1). Ebene für Ebene kann so ein Kryp-

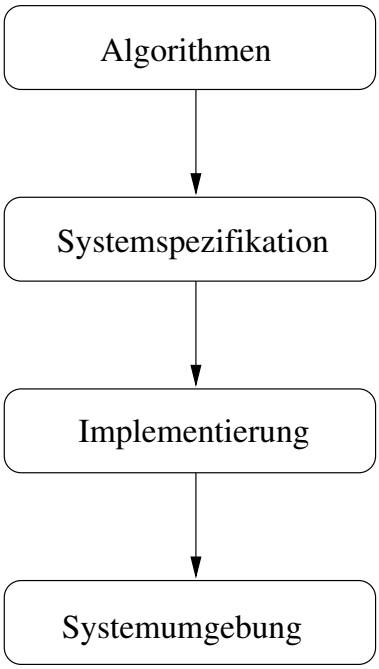

Abbildung 4.1: Aufteilung der Analyse in Ebenen der Entwicklung

tosystem systematisch und gezielt nach Fehlern und Sicherheitslücken untersucht werden. Es werden folgende Ebenen unterschieden:

1. Die *algorithmische* Ebene beschreibt die einzelnen Algorithmen und Mecha-

nismen, die im späteren System verwendet werden. Hier wird untersucht, ob Verschlüsselungsmechanismen wie beispielsweise RSA oder AES algorithmische Schwächen haben, aber auch ob Mechanismen wie Shared-Secrets sicher sind. Darauf wird in Abschnitt 4.1 eingegangen.

2. Mit der *Systemspezifikation* wird das Verhalten des Systems angegeben. Insbesondere werden hier die Funktionsweise, Zustände und Reaktionen des Systems zu jedem Zeitpunkt festgelegt. Dies beinhaltet auch, welche Algorithmen zum Erreichen der vorgeschriebenen Funktionalität eingesetzt werden. Hier liegt der Schwerpunkt bei der Analyse der in dieser Arbeit vorgestellten Protokolle (siehe Abschnitt 4.2).

3. Bei der *Implementierung* wird die Spezifikation in eine Programmiersprache umgesetzt. Falsche Programmierung durch Flüchtigkeit oder mangelnde Sachkenntnis, sowie eine falsche Umsetzung der Spezifikation sind hier große Fehlerquellen (siehe Abschnitt 4.4).

4. Die *Systemumgebung* ist die Umgebung, auf der das fertige Kryptosystem zur Laufzeit arbeiten wird. Von Bedeutung sind das Betriebssystem, Laufzeitumgebungen wie die *Java-Virtual-Machine* (JVM) und auch die eingesetzte Hardware. Probleme, die auf dieser Ebene auftreten können, sind besonders vielfältig. Nur einige Beispiele sind die Abstrahlung der Hardware, Schwächen der JVM oder Sicherheitslücken des Betriebssystems.

   Da die Protokolle in Java implementiert sind, stehen das benutzte Betriebssystem und auch die zugrundeliegende Hardware nicht fest. Die Analyse in Abschnitt 4.3 beschränkt sich daher auf die Risiken der JVM.

In Abschnitt 4.5 folgt eine Beschreibung möglicher Denial of Service-Attacken. Diese Angriffe sind prinzipiell auf jeder Ebene möglich, schwierig vorherzusehen und zu vereiteln.

4.1 Verwendete Chiffren

Zunächst folgt die Analyse der in der Implementierung verwendeten Algorithmen AES, RSA und SHA-1 sowie des Shared-Secrets Mechanismus. Ferner wird der entworfene Pseudo-Zufallszahlen-Generator einem bekannten Test auf Zufälligkeit unterworfen. Dies ist eine Analyse auf der *Algorithmen*-Ebene.

4.1.1 AES, RSA und SHA-1

Wie bereits in Abschnitt 2.1.1 angedeutet, gibt es bis zum jetzigen Zeitpunkt noch keine bekannte algorithmische Schwäche von AES. Dies bedeutet, daß die Chiffre

AES theoretisch bei ausreichender Schlüssellänge sicher gegen Angriffe ist. Die einzige Attacke ist das Erraten des richtigen Schlüssels, bzw. das Ausprobieren aller möglichen Schlüssel, die Brute-Force-Attacke. Selbst die kleinste vorgesehene Schlüssellänge von nur 128 Bit zwingt die Brute-Force-Attacke allerdings zum Scheitern: Sogar ein speziell für diesen Zweck gebauter Computer, ähnlich dem bekannten *DES-Cracker* der Electronic Frontier Foundation (EFF), nur mit einer entsprechend deutlich stärkeren Leistung, würde immernoch 149 Billionen Jahre [24] brauchen.

Ähnliches gilt für die Public-Key Chiffre RSA. Hier ist jedoch ihre Schwäche bekannt. Die Sicherheit des Algorithmus basiert auf dem schwierigen Problem, große Zahlen in Primfaktoren zu zerlegen. Ist das Zerlegen einer großen Zahl in ihre Primfaktoren effizient berechenbar, so kann auch RSA effizient gebrochen werden. Daher nimmt man sehr große Zahlen, die in einem potentiellen Angriff zerlegt werden müßten, z. B. in der Größenordnung von 2^{2048}. Eine Primfaktorzerlegung dieser Zahl dauert etwa 10^{14} Mips-Jahre [4]; das bedeutet, daß ein Computer, der eine Millionen Instruktionen pro Sekunde abarbeiten kann, die Zahl in 10^{14} Jahren faktorisiert haben wird. Der schnellste Großrechner der Welt [40] kann derzeit etwa 10^8-Mips ausführen. Er würde zum erfolgreichen Faktorisieren einer 2048 Bit Zahl demnach etwa 10^6 Jahre benötigen.

Auch der benutze Hash-Algorithmus SHA-1 ist bisher noch nicht erfolgreich angegriffen worden. Damit ist die zur Zeit einzige mögliche Attacke M aus einem Hash-Wert $H(M)$ zu errechnen, die Brute-Force-Attacke aus Abschnitt 2.2.1: das Durchprobieren aller 2^{160} möglichen M. Auch dies ist mit der heutigen Computertechnik unmöglich.

Replay-Attacken

Bei einer *Replay*-Attacken kopiert der Angreifer die gesendeten Chiffrate und schickt sie zu einem späteren Zeitpunkt erneut („replay“) an den Empfänger. Das Protokoll wird dadurch unterwandert, da der Empfänger wahrscheinlich nicht erkennt, daß die *Replay*-Chiffrate Kopien bereits abgearbeiteter Daten sind. Er wird sie u. U. erneut bearbeiten. Abhilfe schafft hier das Einfügen von sogenannten *Zeitstempeln* (engl. *Timestamps*). Versendete Chiffrate enthalten hierbei zusätzlich noch die Zeit, zu der sie abgeschickt wurden. Ist diese Zeit nach Auffassung des Empfängers zu weit in der Vergangenheit, so wird er sie nicht bearbeiten. Liegt sie gar in der Zukunft, so kann er sicher sein, daß eine Manipulation vorliegt - oder falsch synchronisierte Uhren.

4.1.2 Shared-Secrets

Das im Abschnitt 2.3 beschriebene System für Shared-Secrets zweier Zahlen eines Zahlenstrahls $0 \leq a, b \leq 2^{128}$ ist perfekt. Die Kenntnis eines Teilgeheimnisses (Share)bringt zum Erlangen des Gesamtgeheimnisses keinerlei Vorteil. Es gibt keinen Angriff gegen diese Form von Shared-Secrets. Werden die Shared-Secrets wie vorgesehen als Schlüssel eingesetzt, so kostet das Durchprobieren aller möglichen Schlüssel selbst bei Kenntnis eines Teilschlüssels immernoch 2^{128} „Probier"-Schritte, was viel zu aufwendig ist.

4.1.3 FIPS 140-1

Der in Abschnitt 2.5 entworfene und in der Implementierung eingesetzte Zufallszahlen-Generator soll nun analysiert werden.

Ein guter Hinweis auf die (Pseudo-)Zufälligkeit eines PRNG ist das Bestehen des FIPS 140-1 Tests aus [25]. Dieser Test überprüft einige statistische Eigenschaften einer 20000 Bit langen PRNG-Zahlenfolge. Durch die Tests soll die resultierende Pseudo-Zufallsfolge einer echten Folge möglichst ähneln.

- **Mono-Bit-Test**
  Sei X die Anzahl der Einsen in der Zahlenfolge. Es soll gelten: $9654 < X < 10346$.

  Dieser Test überprüft, ob etwa gleichviele Einsen und Nullen in der Folge vorkommen. Dies ist bei einer echten Zufallsfolge zu erwarten.

- **Poker-Test**
  Teile die 20000 Bit Folge in 5000 4-Bit lange Folgen. Zähle und speichere die Vorkommen der 16 möglichen verschiedenen 4-Bit-Folgen. Sei $f(i)$, $i = 0 \ldots 15$, die absolute Häufigkeit der $i-$ten Folge. Berechne:

  $$X = \frac{16}{5000}(\sum_{i=0}^{15} f^2(i)) - 5000$$

  Es soll nun $1,03 < X < 57,4$ gelten.

  Der Test überprüft, ob alle 16 möglichen 4-Bit langen Folgen etwa gleichverteilt in der Folge vorkommen. Auch dies ist bei einer echten Zufalls-Folge zu erwarten.

- **Runs-Test**
  Ein Run ist eine Folge von aufeinanderfolgenden gleichen Bits, entweder nur Einsen oder nur Nullen. Es werden alle Einser-Runs der Längen 1 bis 6

gezählt, genauso alle Nuller-Runs der Längen 1 bis 6. Die Anzahl der Runs der entsprechenden Längen müssen innerhalb der in Tabelle 4.1 abgebildeten Intervalle liegen.

Länge des Runs	Anzahl der Vorkommen
1	2267 - 2733
2	1079 - 1421
3	502 - 748
4	223 - 402
5	90 - 223
6 (und mehr)	90 - 223

Tabelle 4.1: FIPS 140-1 Runs Test

Bei einer echten Zufalls-Folge ist zu erwarten, daß Einsen und Nullen binomialverteilt sind. Es wird versucht, Diese Eigenschaft durch den Runs-Test und den nachfolgenden Long-Runs-Test zu erzwingen [39].

- **Long-Runs-Test**
  In den Zahlenfolgen dürfen weder Einser-Runs noch Nuller-Runs der Länge 34 oder größer (*Long-Runs*) auftreten.

Das IAIK (siehe [15]) hat eine Java-Implementierung der FIPS 140-1 Tests entwickelt, mit der der PRNG dieser Arbeit geprüft wurde. Bei einer gesetzten Seed-Breite von 160 Bit wurden die Tests bei jedem der 20 Durchläufe bestanden.

Die ist kein Beweis für seine kryptographische Stärke, aber wie angedeutet ein guter Hinweis auf eine hohe Güte der generierten Pseudozufallszahlen.

Die NIST empfiehlt, für kritische Anwendungen nur solche Generatoren zu benutzen, die den FIPS Test bestehen.

4.1.4 Bewertung

Aufgrund der Ergebnisse der vorigen Abschnitte läßt sich sagen, daß die in den Protokollen verwendeten Algorithmen zum Chiffrieren von Daten, also auch die Mechanismen wie der Zufallszahlen-Generator auf der *algorithmischen* Ebene sicher sind. Zum heutigen Zeitpunkt existieren keine Verfahren und Mittel, um einen erfolgreichen Angriff durchzuführen.

4.2 Logic of Authentication

Der folgende Abschnitt beschäftigt sich mit der nächsten logischen Ebene, der Spezifikations-Ebene. Die Analyse der beiden Protokolle geschieht auf dieser Ebene mit der BAN Logic of Authentication. Ziel ist eine Überprüfung auf Erfüllung der angegebenen Spezifikation.

4.2.1 Grundlagen

Die eigentliche Analyse mit dem *RVChecker* aus Abschnitt 5.9 gestaltet sich nach dem Formulieren der Annahmen, Nachrichten und Interpretationsregeln aus Kapitel 3 sehr einfach. Die einzelnen Formeln müssen nur noch in einen Java-Quelltext geschrieben und übersetzt werden. Der *RVChecker* kann den erzeugten *class*-File dann analysieren. Hierbei werden nicht nur die angegebenen Protokollziele überprüft sondern auch für jede Nachricht die drei Eigenschaften

- Durchführbarkeit *(Feasibility)*,
- Ehrlichkeit *(Honesty)* und
- Geheimhaltung *(Secrecy)*.

Auf diese Eigenschaften wurde umfassend in Abschnitt 2.6.2 eingegangen.

Neben der Kontrolle der Eigenschaften ist eine Überprüfung der formulierten Protokollziele durchzuführen. Die Protokollziele sind in Abhängigkeit vom jeweiligen Protokoll als Prädikate formuliert. Getestet wird der erfolgreiche Austausch eines Geheimnisses oder das Verhindern der Offenlegung eines solchen.

4.2.2 Erforderliche Anpassungen

Beim Zyklischen Protokoll gibt es jedoch noch ein Problem bezüglich der Durchführbarkeits-Eigenschaft. Bevor das Zyklische Protokoll analysiert werden kann, sind daher zwei Modifikationen am Regelwerk der RVLogik nötig, um den Umgang mit kommutativen Chiffren zu ermöglichen. Dazu wird in dieser Arbeit der komplett in Java geschriebene *RVChecker* um zwei neue Regeln ergänzt, und damit das Zyklische Protokoll analysiert. Das Hinzufügen der Regel ist einfach zu realisieren, da dies direkt im Quell-Code durchgeführt werden kann.

Die zusätzlichen Regeln haben im Detail folgendes Aussehen:

1. Bei der ersten Regel muß die Durchführbarkeits-Überprüfung für kommutative Chiffren erweitert werden. Kann ein Protokoll-Teilnehmer ein Chiffrat, das zuerst mit Schlüssel K_1 und dann mit Schlüssel K_2 verschlüsselt worden ist, verschicken, so kann er auch das Chiffrat verschicken, das zuerst

mit K_2 und dann mit Schlüssel K_1 verschlüsselt worden ist. Formal lautet diese Regel:

$$\frac{canProduce(P, encrypt(encrypt(X, K1), K2))}{canProduce(P, encrypt(encrypt(X, K2), K1))}.$$

Da doppelte Verschlüsselung im Zyklischen Protokoll nur bei kommutativen Chiffren auftritt, arbeitet der *RVChecker* mit dieser Regel weiterhin korrekt.

2. Die zweite Regel gibt an, daß ein Empfang eines Datums mit doppelter Verschlüsselung wie z. B. zuerst mit K_1 und dann mit K_2 chiffriert äquivalent zum Empfang desselben Datums mit zuerst K_2 und dann K_1 Chiffrierung ist. Empfängt ein Protokollteilnehmer ein so verschlüsseltes Datum, so kann er rein mit den Regeln der RVLogik bei Besitz des Schlüssels für das Innere Chiffrat die zugehörige Verschlüsselung nicht aufheben.

   Erforderlich ist eine Regel der Form:

$$\frac{sees(P, encrypt(encrypt(X, K1), K2))}{sees(P, encrypt(encrypt(X, K2), K1))}.$$

   Bekommt Empfänger P das zuerst mit $K1$ und dann mit $K2$ verschlüsselte Chiffrat, so empfängt er selbiges auch zuerst mit $K2$ und dann mit $K1$ verschlüsselt.

   Abermals bleibt die Korrektheit des *RVCheckers* unberührt, da doppelte Verschlüsselung im Zyklischen Protokoll nur bei kommutativen Chiffren auftritt.

4.2.3 Shared Protokoll

Als erstes soll das Shared Protokoll analysiert werden. Dazu wird der Durchlauf des *RVCheckers* betrachtet. Angegeben sind die Ausgaben beim Durchlauf der Nachrichten sowie die Protokollziele des *RVCheckers*. Die jeweiligen Kommentare erläutern die Bedeutung der einzelnen Ausgaben. Die initialen Annahmen sowie die Regeln zur expliziten Interpretation der Nachrichten entsprechen den denen aus Abschnitt 3.2.2. Bei den Protokollzielen handelt es sich um die Frage, ob ein Angreifer in den Besitz des geheimen Datensatz gelangt ist oder ob er Zugang zum Shared-Secret erlangt hat. Daneben wird überprüft, ob der Austausch des Datensatzes zwischen den beiden legitimen Protokoll-Teilnehmern erfolgreich war.

`KMS` bezeichnet den Schlüsselserver, `Ka1` den öffentlichen Schlüssel von `A`, `KMSCert` den öffentlichen Schlüssel von `KMS`, `datakey` den Shared-Secret Schlüssel, `adbsessionkey` den Sitzungsschlüssel zwischen `A` und der Datenbank `database`,

`sessionkey` den Sitzungsschlüssel zwischen `A` und `KMS`, `sessionkey2` der Sitzungsschlüssel zwischen `B` und `KMS`.

Analyse der Nachrichten

Schließlich die eigentliche Ausgabe des *RVCheckers* während der Analyse:

```
Message 1: A -> KMS: encrypt(encrypt(sessionkey, KMSCert),
     Ka1)
Generating consequence closure.......
Message 1 passes feasibility check.
Message 1 passes honesty check.
Message 1 passes secrecy check.
```

Nachricht 1 ist vom *RVChecker* verschickt worden. Die Konsequenzen (`consequence`) dieser Nachricht werden der Menge der Prädikate hinzugefügt, und der Abschluß mit Hilfe der Logik-Regeln hierüber gebildet. Dies bedeutet, daß solange Regeln auf die Menge der Prädikate angewandt werden, bis dabei keine neuen Prädikate mehr entstehen. Beispielsweise ist eine Konsequenz dieser Nachricht, daß KMS das von A verschickte Chiffrat sieht. In die Menge der Prädikate wird demnach ein Prädikat der Form $sees(KMS, ...)$ eingefügt.

Nachdem keine weiteren Prädikate mehr erzeugt werden können, werden die drei genannten Eigenschaften überprüft. Danach wird in der Liste der Prädikate nach dem entsprechenden $canProduce()$ gesucht, um damit die Durchführbarkeit der Nachricht zu beweisen. Dies ist hier der Fall. Die Ausgabe `passes feasibility check` drückt dies aus. Analog werden mit den Regeln zur expliziten Interpretation die Ehrlichkeit (*Honesty*) und Geheimhaltung (*Secrecy*) erfolgreich verifiziert.

Die Nachrichten 2 bis 7 produzieren vergleichbare Ausgaben im *RVChecker*, womit gezeigt wird, daß die Verifikation erfolgreich war.

```
Message 8: database -> B: encrypt(encrypt(datensatz, datakey),
           bdbsessionkey)
Generating consequence closure......
Message 8 passes feasibility check.
Message 8 fails honesty check!
Message 8 passes secrecy check.
```

Die Ausnahme bildet Nachricht 8, bei der die Datenbank das Geheimnis an B schickt. Hier scheitert die Überprüfung auf Ehrlichkeit.

Das Scheitern des Tests auf Ehrlichkeit war jedoch nach Abschnitt 3.2.2 zu erwarten. Die Datenbank kann für die explizite Interpretation von B bzgl. dieser Nachricht keine *legit*-Regel ableiten, siehe Abschnitt 2.6.2. Zur Erinnerung: B

interpretiert den Inhalt der Nachricht nach dem Entschlüsseln als `datensatz`. Hierfür kann die Datenbank nicht einstehen. Dies ist jedoch nicht problematisch.

Protokollziele

Die wichtigsten Protokollziele für das Shared Protokoll lauten:

- A und B besitzen den Datensatz, aber der potentiellen Angreifer I nicht
- Das Shared-Secret ist nur A und B bekannt
- Die Sitzungsschlüssel sind nicht allgemein bekannt, sondern nur den entsprechenden Teilnehmern

```
Protocol Goals:
===============
1. has(I, datensatz): false
2. has(B, datensatz): true
3. has(A, datensatz): true
[...]
6. has(B, datakey): true
7. has(A, datakey): true
8. has(I, datakey): false
[...]
11. has(I, sessionkey): false
12. has(I, sessionkey2): false
[...]
```

I kennt demnach nicht den geheimen Datensatz und auch nicht das Shared-Secret. I ist auch nicht im Besitz der Sitzungsschlüssel. B und A haben den Datensatz über das Shared-Secret erfolgreich ausgetauscht.

Ergebnis

Mit Hilfe des *RVCheckers* ist demnach nachgewiesen, daß das Protokoll unter den angegebenen Voraussetzungen sicher ist. Zunächst ist das Protokollziel, d. h. der erfolgreiche Austausch das Datensatzes von A nach B über den Schlüsselserver, erreicht worden, und die Eigenschaften der RVLogik wurden erfolgreich verifiziert. Außerdem ist gezeigt worden, daß ein potentieller Angreifer in Form eines alles mitlesenden *Big-Brothers* nicht in den Besitz des geheimen Datensatzes respektive der Sitzungsschlüssel gelangen kann. Falls alle Protokollteilnehmer wie von ihnen verlangt handeln, ist die Spezifikation des Protokolls bewiesen sicher.

Kompromittierte Teilnehmer

Die bisherige Analyse geht davon aus, daß alle Protokollteilnehmer wie von ihnen verlangt korrekt handeln. Eine Gefahr geht von kompromittierten Protokoll-Teilnehmern aus, die mit dem *RVChecker* nicht erfaßt werden kann. Im folgenden werden daher die möglichen Fälle eigens betrachtet.

1. **kompromittierter Teilnehmer A oder B**
   Versucht ein Benutzer des Systems einen unrechtmäßigen Zugriff zu starten, beispielsweise A einen unrechtmäßigen Schreibzugriff, so wird dies durch die Regelserver oder Zugriffsserver verhindert. Nur wenn A sich beim Regelserver für jeden Zugriff mit einer zu seinem Zertifikat passenden Unterschrift anmeldet, bekommt er dessen Zustimmung (*Approve*). Dies gilt sowohl für den eigentlichen Schreibvorgang in die Datenbank, als auch für die Anfrage nach dem Shared-Secret vom Schlüsselserver. Schlüsselserver und Datenbank überprüfen die Korrektheit des Zugriffes von A über die Unterschrift des Regelservers. Damit kann ein falschspielender Protokollteilnehmer wie A oder auch B nicht auf für ihn geheime Daten zugreifen.

2. **kompromittierter Schlüsselserver**
   Ist einer der beiden Schlüsselserver in Feindeshand, so kann man damit rechnen, daß er Zugriff auf alle bei ihm gespeicherten Shared-Secrets hat. Dadurch alleine ensteht jedoch noch kein Vorteil, da diese „halben“ Shared-Secrets ohne die andere Hälfte des anderen Schlüsselservers wertlos sind (siehe Abschnitt 2.3). Es ist genauso aufwendig, das komplette Shared-Secret zu erraten, wie bei Besitz des einen Teils das andere. Um Zugang zur anderen Hälfte zu erreichen, müßte sich der kompromittierte Schlüsselserver erfolgreich beim anderen Schlüsselserver autorisieren. Genau hierfür ist jedoch eine gültige Unterschrift des Regelservers nötig, die nur dann ausgestellt wird, wenn die entsprechende Berechtigung existiert. Der Schlüsselserver kann dem Benutzer ein falsches Shared-Secret schicken, was dazu führt, daß dieser den Datensatz später nicht erfolgreich dechiffrieren kann. Dies ist eine Form von *Denial of Service*, worauf kurz in Abschnitt 4.5 eingegangen wird.

3. **kompromittierte Datenbank**
   Eine kompromittierte Datenbank hat Zugriff auf alle Chiffrate. Alle Chiffrate sind mit dem Shared-Secret verschlüsselt, so daß die Datenbank genau wie ein außenstehender Angreifer eine Attacke auf ein solches Chiffrat starten müßte.

   Auch die Datenbank kann dem Benutzer einen falschen oder kaputten Datensatz schicken, welchen dieser dann nicht entschlüsseln kann, siehe Abschnitt 4.5.

Das Protokoll bleibt also auch bei einem kompromittierten Protokollteilnehmer noch sicher, jedoch kann ein Protokollziel, nämlich der Austausch des Datensatzes, nicht mehr erreicht werden.

Im Falle, daß mehrere Stationen übelgesinnt sind und auch noch zusammenarbeiten, beispielsweise die beiden Schlüsselserver, oder Datenbank und Benutzer oder Rollenproxy, ist die Sicherheit nicht mehr gewährleistet.

4.2.4 Zyklisches Protokoll

Analog zum vorigen Abschnitt wird die Ausgabe des *RVCheckers* beim Zyklischen Protokoll genauer interpretiert. Auch hier sollen die drei Eigenschaften aus Abschnitt 2.6.2, sowie die Protokollziele überprüft werden.

`A` bezeichnet wieder den Rollenproxy von Benutzer A. Dieser will einen Datensatz in die Datenbank `database` schreiben. Wie im Zyklischen Protokoll vorgesehen, geschieht dies über eine Umverschlüsselung durch die Transferstation `TS`. `B` ist der Rollenproxy des Benutzers B, der den zuvor abgespeicherten Datensatz aus der Datenbank über die Transferstation auslesen möchte. `K1` entspricht dem Schlüssel K_{RP} aus Abschnitt 3.3, `K2` entspricht K_{TS}. Die restliche Notation stimmt ebenfalls mit der aus dem vorigen Abschnitt überein.

Analyse der Nachrichten

In Nachricht 1 und 2 hat A den Schreibvorgang initiiert, in dem er seinen mit `K1` chiffrierten Datensatz an die Transferstation schickt:

```
Message 1: A -> TS: encrypt(dataset, K1)
Generating consequence closure....
Message 1 passes feasibility check.
Message 1 passes honesty check.
Message 1 passes secrecy check.
Message 2: TS -> database: encrypt(encrypt(dataset, K1),
              K2)
Generating consequence closure.....
Message 2 passes feasibility check.
Message 2 fails honesty check!
Message 2 passes secrecy check.
```

Die Nachricht besteht die Tests der drei Eigenschaften. In Nachricht 2 scheitert die Überprüfung auf Ehrlichkeit. Hierbei schickt die Transferstation das inzwischen doppelt verschlüsselte Chiffrat weiter an die Datenbank. Wie in Abschnitt 3.3 schon erwartet, kann die Transferstation keine entsprechende Regel zur Erfüllung

der Ehrlichkeits-Eigenschaft ableiten, da ihr angemessene Voraussetzungen bzw. Annahmen zum Glauben an den Inhalt des Chiffrats fehlen. Das negative Ergebnis dieser Überprüfung spielt jedoch hier noch keine Rolle.

```
Message 3: A -> database: encrypt(K1, Kdatabase)
Generating consequence closure.....
Message 3 passes feasibility check.
Message 3 passes honesty check.
Message 3 passes secrecy check.
```

Mit dieser 3. Nachricht ist der Schreib-Zyklus beendet. *A* schickt hierbei schließlich den Schlüssel `K1` an die Datenbank, damit diese ihn von dem zuvor empfangenen Chiffrat entfernen kann. In der Datenbank ist jetzt der Datensatz mit dem Schlüssel `K2` chiffriert gespeichert.

Jetzt kann der Lese-Zyklus von *B* beginnen:

```
Message 4: B -> database: encrypt(K3, Kdatabase)
Generating consequence closure.....
Message 4 passes feasibility check.
Message 4 passes honesty check.
Message 4 passes secrecy check.
Message 5: database -> TS: encrypt(encrypt(dataset,
                          K3), K2)
Generating consequence closure......
Message 5 passes feasibility check.
Message 5 fails honesty check!
Message 5 passes secrecy check.
```

B hat den Lese-Vorgang gestartet, indem er den Schlüssel `K3` an die Datenbank schickt. In der Implementierung muß *B* zweifelsohne auch noch den Namen bzw. eine Art Identifizierung des Datensatzes an die Datenbank schicken, doch diese ist für die Analyse des Protokolls nicht von Bedeutung. Der übertragene Schlüssel ist, wie in Abschnitt 3.3 gefordert, mit einem Sitzungsschlüssel bzw. dem öffentlichen Schlüssel der Datenbank chiffriert.

Die Datenbank schickt den doppelt verschlüsselten Datensatz in Nachricht 5 an die Transferstation. Wieder wird der Test auf Ehrlichkeit nicht bestanden, da auch hier, wie erwartet, keine Möglichkeit zur Interpretation des eigentlichen Inhalts besteht.

```
Message 6: TS -> B: encrypt(encrypt(dataset, K3), KB)
Generating consequence closure......
Message 6 passes feasibility check.
Message 6 fails honesty check!
```

```
Message 6 passes secrecy check.
```

Damit ist das Zyklische Protokoll beendet. In der letzten Nachricht ist der Datensatz nur noch mit `K3` und dem verlangten Sitzungsschlüssel (hier: öffentlicher Schlüssel von B) chiffriert an B gesendet worden. Der Sender Transferstation kann keinerlei Verantwortung für den Inhalt seiner verschickten Nachricht übernehmen, folglich das negative Ergebnis für den Test auf Ehrlichkeit.

Protokollziele

Schließlich folgen die wichtigsten Protokollziele.

- Das Protokoll erfüllt die Anforderungen, nämlich den Datenaustausch des Datensatzes `dataset`. B ist bei Protokollende im Besitz dieses Datensatzes.
- Ein potentieller Angreifer I, der den kompletten Netzverkehr abhören kann, hat keinen Zugriff auf den geheimen Datensatz.
- Die Datenbank oder die Transferstation sind nicht in den Besitz des Datensatzes gelangt.

```
Protocol Goals:
===============
1. has(A, dataset): true
2. has(B, dataset): true
[...]
6. has(TS, dataset): false
7. has(database, dataset): false
[...]
10. has(I, dataset): false
```

B hat erfolgreich den Datensatz aus der Datenbank lesen und entschlüsseln können. Sowohl I als auch Datenbank und Transferstation haben keinen Zugriff auf den Datensatz.

Ergebnis

Damit ist auch für den zyklischen Fall mit dem *RVChecker* bewiesen, daß das Protokoll unter den angegebenen Annahmen sicher ist. Die Protokollziele sind erreicht worden, welche der Austausch des Datensatzes sowie dessen Geheimhaltung waren. Die Nachrichten erfüllen alle die Eigenschaften *Durchführbarkeit* und *Geheimhaltung*. Die Nachrichten, die die Überprüfung auf Ehrlichkeit nicht bestehen, sind identifiziert und waren zu erwarten. Sofern alle Protokollteilnehmer laut Spezifikation handeln, stellt dies kein Problem dar.

Kompromittierte Teilnehmer

Wie im letzten Abschnitt muß allerdings untersucht werden, welche Konsequenzen kompromittierte Protokoll-Teilnehmer nach sich ziehen. Dies ist mit dem *RVChecker* nicht zu verifizieren. Es gibt drei Fälle:

1. **kompromittierter Teilnehmer A oder B**
   Ein Benutzer bzw. dessen Rollenproxy braucht, ebenso wie beim Shared Protokoll, die Autorisierung des Zugriffsservers für den entsprechenden Datenzugriff. Nur Rollenproxys, die im Besitz der gültigen Unterschriften für einen Zugriff sind, können diesen durchführen. Somit können böswillige Benutzer A oder B nicht auf für sie nicht bestimmte Daten zugreifen.

2. **kompromittierte Transferstation**
   Ist die Transferstation (alleine) kompromittiert, so verfügt sie nur über das Chiffrat $encrypt(dataset, K_{RP})$ oder $encrypt(dataset, K_{RP'})$. Da die Schlüssel K_{RP} und $K_{RP'}$ jedoch nur verschlüsselt übertragen werden, kann sie aus dem Besitz des Chiffrats keinerlei Gewinn erzielen.

   Um an einen dieser Schlüssel zu gelangen bzw. diesen der Datenbank selbst zu übergeben, muß die Transferstation wie jeder Rollenproxy die Zustimmung des Zugriffsservers erreichen.

   Somit bleibt der Transferstation nur die Möglichkeit, dem Benutzer oder der Datenbank einen falsch verschlüsselten Datensatz zu schicken. Auch dies ist eine *Denial of Service*-Attacke, siehe dazu Abschnitt 4.5.

3. **kompromittierte Datenbank**
   Die Datenbank empfängt neben dem Chiffrat
   $encrypt(encrypt(dataset, K_{RP}), K_{TS})$ auch den Schlüssel K_{RP}. Sie ist also ausschließlich im Besitz von Chiffraten vom Typ $encrypt(dataset, K_{TS})$. Wäre die Nachricht $encrypt(dataset, K_{RP'})$ von der Transferstation zum Rollenproxy B nicht noch durch einen zusätzlichen Sitzungsschlüssel chiffriert, so könnte die Datenbank dieses Chiffrat entschlüsseln und käme in den Besitz von *dataset*.

   Schließlich kann die Datenbank fehlerhafte Chiffrate an die Transferstation schicken, was wiederum in die Kategorie *Denial of Service*-Attacken aus Abschnitt 4.5 gehört.

Damit ist das Protokoll auch noch bei einem kompromittierten Protokollteilnehmer sicher. Wie beim Shared Protokoll ist aber u. U. der normale Austausch eines Datensatzes nicht mehr möglich.

Falls mehrere Stationen böswillig sind, und diese zusammenarbeiten, so ist die Sicherheit im Zyklischen Protokoll nicht mehr gegeben. Arbeiten z. B. Transferstation und Datenbank zusammen, so kann die Transferstation der Datenbank

den Schlüssel K_{TS} übergeben, oder umgekehrt, die Datenbank der Transferstation den Schlüssel K_{RP}.

4.3 Laufzeitumgebung

Die in dieser Arbeit entwickelten und analysierten Protokolle sind die Grundlage eines medizinischen Informationssystems zum Zugriff und zur Verwaltung von Patientendaten. Das Informationssystem wird als Java-Anwendung oder als Java-Applet [35] in einem Web-Browser den Umgang mit solchen Patientendaten ermöglichen. Die Aufgabe der Protokolle hierbei ist der Schutz und die Sicherung des Datenbank-Zugriffs. Zwangsläufig müssen die implementierten Java-Protokoll-Klassen bei Daten-Transaktionen in der *Java-Virtual-Machine* (JVM) des Benutzers ausgeführt werden. Die JVM ist die Laufzeitumgebung des Browsers, in der alle Java-Applets ausgeführt werden. Dies kann problematisch sein, da böswillige Benutzer u. U. so durch Schwächen in der JVM einen unrechtmäßigen Zugriff auf z.B. geheime Schlüssel bekommen könnten. Auf der anderen Seite sind ordentliche Benutzer gefährdet, die etwa durch Angreifer veränderte Versionen der Protokoll-Applets untergeschoben bekommen.

Welche Gefahren im Umgang mit den Protokoll-Klassen bestehen und wie die JVM und der Benutzer sich schützen können, welche Probleme im Umgang mit CORBA auftreten, soll im Folgenden beschrieben werden.

4.3.1 Vom Quell-Code zur JVM

Zunächst jedoch eine Darstellung der einzelnen Vorgänge, die nötig sind, um ein Java-Programm auf einem Computer auszuführen, siehe Abbildung 4.2.

Der Java-Quell-Code wird vom Java-Compiler in einen plattform-unabhängigen Byte-Code übersetzt. Der Compiler generiert dabei sogenannte *class* Dateien. Die Dateien müssen später entweder auf der lokalen Festplatte des Benutzers oder aber auf einem Web-Server beispielsweise des Krankenhauses zur Verfügung stehen, damit sie von der JVM geladen werden können. Nach dem Laden interpretiert die JVM den Byte-Code und übersetzt ihn Schritt für Schritt in die plattformspezifische Maschinensprache.

4.3.2 Risiken und Schutzmechanismen der JVM

Es sind zwei Typen von Bedrohungen zu unterscheiden:

1. Ein legitimer Benutzer wird durch den falschen Code eines Angreifers bedroht.

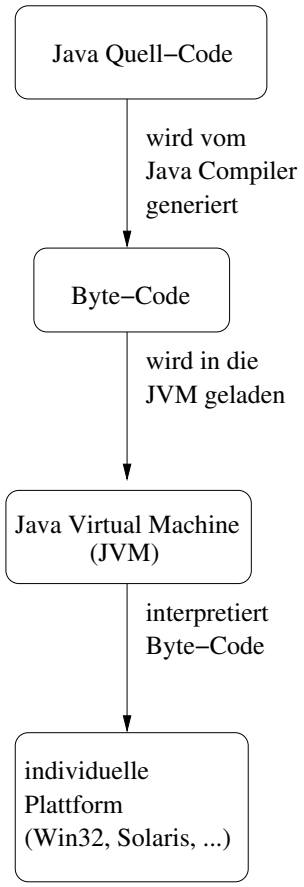

Abbildung 4.2: Vom Quell-Code zur individuellen Plattform

2. Ein böswilliger Benutzer versucht Schwächen innerhalb des Codes oder der JVM auszunutzen.

Ein rechtmäßiger Benutzer muß die Java-Klassen in seine JVM laden. Dies geschieht entweder über das Netzwerk oder über seine lokale Festplatte. In jedem Fall ist es möglich, daß ein Angreifer die zu ladenden Dateien manipuliert hat, und so manipulierter Code in der JVM des Benutzers ausgeführt wird.

Der Angreifer kann hierbei entweder seinen eigenen „regulären“ Java-Quell-Code compiliert haben und den resultierenden Byte-Code verwenden, oder aber illegalen Byte-Code produzieren. Illegaler Byte-Code ist Byte-Code, der nicht der Spezifikation der JVM entspricht, d. h. die JVM kann diesen Byte-Codes keine Anweisungen zuordnen. Es ist jedoch möglich, daß durch illegalen Byte-Code Sicherheits-Lücken in der JVM ausgenutzt werden. Die Risiken beim Ausführen von falschem oder fehlerhaftem Byte-Code sind z. B. ein unberechtigter Datei-Zugriff des Programms, das Auslesen von geheimen Paßwörtern und Schlüsseln oder das Übertragen von Geheimnissen an den Computer des Angreifers.

Um sich dagegen zu schützen, wird vor dem Ausführen eines Java-Programms dessen Byte-Code durch den *Verifier* der JVM inspiziert [34]. Der *Verifier* prüft den Byte-Code auf Korrektheit bezüglich der Java-Spezifikation, auf Typen-Sicherheit, d. h. auf fehlerfreie Zuweisungen und *Type-Casts*, sowie auf das korrekte Referenzieren von Objekten. Trotz des *Verifiers* hat es in der Vergangenheit dennoch eine Menge erfolgreicher Angriffe gegen die JVM gegeben [36]. Daher existiert eine weitere Möglichkeit zum Schutz vor falschem Code, das *Signieren* von *class*-Dateien. Ein Entwickler kann seine compilierte Byte-Code *class*-Datei mit Mechanismen ähnlich zu Abschnitt 2.2 elektronisch signieren. Von dem Byte-Code des Programms wird ein Hash-Wert errechnet, und dieser mit dem geheimen Schlüssel des Herausgebers chiffriert. Der Benutzer des Programms empfängt nicht nur die *class*-Datei, sondern auch die Unterschrift. Automatisch bildet die JVM den Hash-Wert der *class*-Datei und überprüft mit Hilfe des öffentlichen Schlüssels des Herausgebers, ob die Unterschrift gültig ist. Die JVM zeigt dem Benutzer bei authentischer Signatur an, von wem das Programm geschrieben wurde. Daraufhin kann der Benutzer entscheiden, ob er das Programm ausführen möchte, oder nicht. Außerdem kann er speziell für dieses Programm bestimmen, welche Rechte es auf seinem Computer besitzen soll: beispielsweise ob Dateien gelesen oder geschrieben werden dürfen, oder ob Netzwerkverbindungen zu fremden Computern erlaubt sind, siehe [18]. Es gibt eine Vielzahl von Einstellungsmöglichkeiten zur Zugriffskontrolle. Die JVM kann die Unterschriften überprüfen, da ihr die Zertifikate der rechtmäßigen Herausgeber bzw. einer obersten Zertifizierungsstelle bekannt sind. Die Technik hierfür ist in Abschnitt 2.2.2 beschrieben.

Es ist demnach wichtig, daß die entwickelten *class*-Dateien des Programms signiert werden.

Bei der zweiten Form von Bedrohung besteht die Gefahr, daß ein böser Nutzer versucht, Geheimnisse des Programms zu bewahren und zu kopieren. Alle Geheimnisse, das sind insbesondere die geheimen Schlüssel der Rollen, sind in der JVM als Teil des Programms im RAM gespeichert. Der Rollenproxy (eine Java-Klasse, und damit eine *class*-Datei) verwaltet sie automatisch ohne Interaktion mit dem Benutzer. Muß der Benutzer eine Rolle abgeben, so ist wichtig, daß er danach keinen Zugriff mehr auf die Geheimnisse dieser Rolle hat. Deshalb löscht der Rollenproxy beim Verlassen der Rolle alle damit zusammenhängenden abgespeicherten Geheimnisse aus dem Speicher. Das ist jedoch problematisch: ein wirkliches „Löschen“ oder Überschreiben eines Datums gibt es in Java nicht. Der Variablen, die ein Geheimnis enthält, kann nur ein neuer Wert zugewiesen werden, was nicht bedeutet, daß der alte Wert dadurch überschrieben wird. Im Gegenteil, der alte Wert verbleibt solange im Speicher, bis die „Garbage-Collection“ [22] ihn zum Überschreiben freigibt. Gleichermaßen ist es für einen Benutzer möglich, sich grundsätzlich eine Kopie von Objekten oder des gesamten Speicherbereichs

der JVM zu erstellen, und so die Geheimnisse vor dem Löschen zu bewahren.

Kann ein Angreifer geheime Schlüssel aufbewahren, so ist er zwar nicht mehr in der Lage eigene Transaktionen durchzuführen - wir erinnern uns: dafür war die Unterschrift eines Zugriffsservers notwendig - aber er kann u. U. die Kommunikation anderer Protokollteilnehmer abhören.

Insofern ist festzuhalten, daß Geheimnisse oder dessen Kopien im Speicher des böswilligen Benutzers vorliegen. Offen bleibt, inwieweit es realistisch ist, die Geheimnisse aus dem Speicherbereich zu extrahieren. Dieses Problem ist allerdings nicht typisch für Java bzw. die JVM, sondern gilt für jedes Programm, das auf einem Computer ausgeführt wird. Einen hundertprozentigen Schutz dagegen gibt es nicht. Der Entwickler eines Java-Programms kann nur einige vorbeugende Maßnahmen treffen, die es einem potentiellen Angreifer erschweren, Geheimnisse aus dem Speicherbereich des Programms bzw. der JVM auszulesen.

In [11] wird ein Regelkatalog vorgestellt, der das Kopieren oder Auslesen von Speicherbereichen verkompliziert. Die Regeln wurden in der vorliegenden Implementierung soweit möglich auf die Rollenproxy-Klasse angewandt. Dadurch wird erzwungen, daß ein Angreifer mit einem Speicher-Monitor in dem viele Mega-Byte großen Speicherbild nach dem Geheimnis suchen muß. Das ist sehr aufwendig.

Ein weiteres Problem stellt die komplette Ersetzung der JVM durch einen Angreifer dar. Dies ist nicht abwegig, da die JVM meistens aus dem Internet bezogen wird, und ein Angreifer dadurch in der Lage ist, den *download* zu kompromittieren. Der Angreifer hätte so Zugriff auf sämtliche Systemkomponenten, könnte nach Belieben auch ohne Wissen des Benutzers agieren und ohne Probleme z. B. die geheimen Rollen-Schlüssel kopieren. Die Benutzer müssen daher darauf achten, daß ihre JVMs von entsprechend gesicherten Orten installiert werden.

4.3.3 CORBA

Die *Common Object Request Broker Architecture* ist die Spezifikation einer Architektur zur Zusammenarbeit von Computer-Programmen über Netzwerke [28]. Mit CORBA wird eine Schnittstelle definiert, mit der verschiedene Programme interagieren können, die in verschiedenen Sprachen geschrieben wurden und auf unterschiedlichen Rechnern mit verschiedenen Betriebssystemen laufen. CORBA ermöglicht es Clients, Anfragen transparent an ein Server-Objekt zu senden, wobei der Server sowohl auf dem gleichen Rechner als auch auf einem entfernten Rechner laufen kann. CORBA übernimmt dabei die Aufgabe, das geforderte Server-Objekt zu finden, die entsprechende Methode aufzurufen und das Ergebnis an den Client zurück zu liefern. Dabei wird automatisch eine Konvertierung verschiedener Repräsentationen von Datentypen durchgeführt [29].

Auf die weitere Funktionsweise von CORBA, bzw. dessen genauen Aufbau und Struktur soll im Folgenden nicht eingegangen werden, da es für die kryptographische Betrachtung nicht weiter nötig ist. Wesentlich ist, daß sämtliche Kommunikation in den implementieren Protokollen über die in Java eingebettete CORBA-Funktionalität abgewickelt wird.

CORBA ist unsicher: Sämtliche Daten werden im Klartext übertragen. Dies ist allerdings nicht weiter tragisch, da in den implementierten Protokollen wo angebracht alle zu übertragenen Daten verschlüsselt werden. CORBA übernimmt so nur die Aufgabe des eigentlichen Transports der verschlüsselten Daten.

Das Suchen nach Server-Objekten geschieht in CORBA ebenso mit Klartext-Zeichenketten, es gibt keine wirkliche Authentifizierung. Dies bedeutet, daß das Objekt, das sich zuerst unter einem frei bestimmbaren Namen im CORBA-System anmeldet, auch unter diesem registriert wird. Es ist dementsprechend für den Client gefährlich, davon auszugehen, daß das angesprochene Server-Objekt tatsächlich das angeforderte ist. Als Konsequenz wird in der Implementierung immer überprüft, ob das Zertifikat des Server-Objekts (bzw. des zugehörigen Protokollteilnehmers) wirklich dem geforderten Objekt gehört, und ob es gültig ist.

Es gibt inzwischen Implementierungen der Sicherheits-Erweiterungen von CORBA, *CORBASec*, siehe [30], die chiffrierte Verbindungen zwischen Protokollteilnehmern ermöglichen. Allerdings können damit Umverschlüsselungen und kommutative Chiffren wie im Zyklischen Protokoll benötigt nicht realisiert werden, bzw. nur, wenn die einzelnen Datentypen bereits mit der kommutativen Chiffre chiffriert als Bytefolgen vorliegen. Dadurch bietet CORBASec keinen Vorteil gegenüber der eigenen Implementierung, sondern würde im Gegenteil den Aufwand für Kommunikation und das Management eines Systems nur noch unnötig erhöhen.

4.4 Implementierung

Das Finden von Fehlern im Quell-Code ist eine altbekannte Schwierigkeit. Neben den Fehlern, die die spezifizierte Funktionalität beeinträchtigen und die meist durch intensives Testen gefunden werden, sind insbesondere Fehler gefährlich, die nur in Ausnahmesituationen auftreten [31]. Dies ist darin begründet, daß das System normalerweise nur mit vorher spezifizierten Eingaben und Zuständen rechnet, sicherheitskritische Systeme aber auf alle Eingaben und Zustände korrekt reagieren müssen. Es ist bei der Entwicklung schwierig, dem Quell-Code etwaiges Versagen bei u. U. noch unbekannten Ausnahmen anzusehen.

Die Programmiersprache Java hilft, einige klassische Fehler bei der Software-Entwicklung zu verhindern [11]. Folgende Eigenschaften der Sprache ermöglichen

das:

- Java bietet Typsicherheit. Dies ist die wohl wichtigste sicherheitstechnische Eigenschaft der Sprache. Unsaubere Zuweisungen oder Type-Casts, die in anderen Sprachen wie *C* oder *C++* häufig zu Fehlern führen, sind in Java verboten. Operationen können auf Objekten nur dann ausgeführt werden, wenn diese Operationen auch für dieses Objekt gültig sind.

- Array-Grenzen werden überprüft. Bei jedem Zugriff auf ein Array wird kontrolliert, ob der angegebene Index gültig ist. Dadurch werden das sonst so häufige Überschreiten von Puffergrenzen und das damit verbundene Überschreiben kritischer Bereiche vermieden.

- Variablen müssen, bevor sie benutzt werden können, immer zuerst initialisiert werden. Dies verhindert das Verwenden von nicht initialisierten Variablen mit unvorhergesehenem Inhalt.

- Es existiert keine explizite Freigabe von Speicher. Werden Objekte nicht mehr länger benötigt, dann entfernt der *Garbage Collector* diese automatisch aus dem Speicher. Es bedarf dafür keines expliziten Programmierbefehls. Dadurch kann es nicht zu Problemen durch das Referenzieren von bereits freigegebenen Objekten kommen. Außerdem wird so Speicher effizienter verwaltet. Der *Garbage Collector* errechnet, wann auf ein Objekt nicht mehr zugegriffen werden kann, und gibt es dementsprechend anschließend frei.

- Innerhalb von Java-Objekten existiert eine Zugriffsbeschränkung. Variablen können als `public, private` oder `protected` deklariert werden. Mit `public` darf jedes Objekt auf diese Variable zugreifen, mit `protected` nur Methoden der eigenen Klasse bzw. die abgeleiteter Klassen, und mit `private` dürfen nur die klasseneigenen Methoden auf diese Variable zugreifen. Dadurch kann schon bei der Implementierung ein rudimentärer Schutz und eine Abstufung kritischer Informationen bestimmt werden.

- Java sieht grundsätzlich das Auftreten von Ausnahmen *(Exceptions)* vor. Bei vorher definierten Fehlern oder bei bisher unbekannten Fehlern oder Eingaben werden solche Ausnahmen ausgelöst, auf die der Entwickler in seinen eigenen Routinen reagieren kann. Es ist demnach sehr sinnvoll, daß der Entwickler beim Auftreten von unbekannten *Exceptions* eine Routine zur Verfügung stellt, die beispielsweise einen kontrollierten Programm-Abbruch oder gar ein normales Weiterarbeiten ermöglicht.

4.5 Denial of Service

Bei *Denial of Service* (DoS)-Attacken handelt es sich um eine Kategorie von Angriffen, bei der die Verfügbarkeit von Betriebsmitteln oder Diensten das Hauptziel des Angreifers ist. Der Angreifer versucht den legalen Zugriff rechtmäßiger Benutzer auf diese Dienste und Betriebsmittel zu verhindern.

Im Folgenden werden verschiedene Typen von DoS-Angriffen beschrieben, ihre Gefahren für die Protokolle sowie einige mögliche Verteidigungsmaßnahmen aufgezeigt. Dabei werden die von ihnen ausgehenden Gefahren für die Protokolle sowie mögliche Verteidigungsmaßnahmen entwickelt.

Denial of Service Angriffe lassen sich in folgendes Schema einteilen [6].

1. Verbrauch von begrenzten Ressourcen
2. Verändern oder Zerstören von Konfigurations-Einstellungen
3. Physikalische Zerstörung

4.5.1 Verbrauch begrenzter Ressourcen

Der erste Fall ist die wohl häufigste DoS-Attacke. Ein Angreifer verbraucht einen Großteil einer begrenzten Ressource so auf, daß legalen Benutzern diese Ressource entweder gar nicht oder nur in sehr begrenztem Maße zur Verfügung steht. Das klassische Beispiel ist das Aufbrauchen von Netzwerkkapazität durch *Flooding*-Attacken. Der Angreifer überschüttet ein Netzwerk mit Datenpaketen derart, daß die Kommunikation legitimer Nutzer stark verlangsamt und eingeschränkt wird. Auch einzelne Computer bzw. dessen Betriebssysteme können Ziel einer Flooding-Attacke sein: Ein Computer wird mit eingehenden Datenpakten stark überhäuft, mit dem Ziel, daß für ordentliche Pakete keine Rechenkapazität mehr zur Verfügung steht. Diese Angriffe sind sehr real und in der Vergangenheit häufig aufgetreten. Andere Formen dieses DoS-Typs sind z. B. das Füllen von Plattenplatz. Ein böswilliger (Protokoll-)Benutzer kann eine Menge Datensätze in die Datenbank hineinschreiben und diese u. U. sogar komplett füllen, so daß kein Platz mehr für andere Benutzer zur Verfügung steht. Genauso kann ein Angreifer immer und immer wieder Anfragen an die Zugriffsserver oder anderen Protokollteilnehmer stellen, die zwar wegen des Fehlens der entsprechenden Berechtigungen zurückgewiesen werden aber dennoch wertvolle Rechenzeit und Netzwerk-Performance kosten. Alle Dienstgeber wie Schlüssel-Server, Transferstation, Datenbank usw. sind anfällig für diese Attacken. Sogar der Benutzer selbst ist dadurch gefährdet: Der Rollenproxy, der in der JVM des Benutzer läuft, kann von Datenbanken *update*-Nachrichten über veränderte Datensätze empfangen. Ein Angreifer kann hier dem Benutzer schaden, indem er eine Menge

gefälschter *update*-Nachrichten produziert. Der empfangende Benutzer erkennt diese zwar als ungültig an, muß aber dennoch Rechenzeit dafür ausgeben.

Der Schutz gegen solche DoS-Attacken ist problematisch, da man schwierig zwischen legalen aber stark benutzten Ressourcen und Ressourcen, die unter Angriff stehen, unterscheiden kann. Mögliche Verteidigungsmaßnahmen gegen Netzwerkattacken sind beispielsweise Netzwerkfilter oder entsprechend konfigurierte Router. Das Vollschreiben von Festplattenplatz kann durch das Verwenden von Techniken wie *Quota* begrenzt werden.

Ähnlich dem böswilligen Aufbrauchen von Ressourcen verhält es sich mit kompromittierten Protokollteilnehmern, wie in Abschnitt 4.2.3 und Abschnitt 4.2.4 angedeutet. Ist einer der Protokollteilnehmer kompromittiert, so kann er, wie z. B. in Abschnitt 4.2.3 bewiesen, zu keinem Zeitpunkt das Geheimnis erlangen, jedoch ordentlichen Benutzern falsche Informationen zuspielen. Ist die Datenbank kompromittiert, so kann sie dem Benutzer falsche Datensätze zuschicken. Der Benutzer merkt dies natürlich, da sich der Datensatz nicht mit dem zu benutzenden Schlüssel dechiffrieren läßt. Dasselbe gilt für die Transferstation. Auch sie kann nur ein falsch umgeschlüsseltes Datum an den Benutzer versenden und ihn damit „verwirren“. Beim Shared Protokoll kann ein kompromittierter Schlüssel-Server ein falsches Share an den Benutzer schicken. Dies hat zur Konsequenz, daß der resultierende Schlüssel falsch ist und verhindert das rechtmäßige Entschlüsseln des Datensatzes durch den Benutzer.

Die Verteidigung ist auch hier problematisch. Es ist falsch, aufgrund nicht-entschlüsselbarer Datensätze auf eine DoS-Attacke zu schließen, beispielsweise sind Übertragungsfehler viel wahrscheinlicher. Bei korrupten Datensätzen oder fehlerhaften *update*-Signalen sollte zunächst von Übertragungsfehlern ausgegangen werden. Erst wenn sich falsche Nachrichten häufen, besteht die Gefahr einer DoS-Attacke. Ein zentraler Monitor könnte die Anzahl falscher oder korrupter Nachrichten zählen und bei Überschreitung einer Schwelle Alarm auslösen.

4.5.2 Verändern der Konfiguration

Ein falsch konfiguriertes System kann nicht einwandfrei arbeiten. Demnach ist die Konfiguration eines Systems Ziel von Angriffen. Schafft es ein Angreifer beispielsweise, die Routing-Tabelle eines Computers zu manipulieren, so ist der Computer nicht mehr funktionstüchtig. Wird die Registry eines Windows-NT Rechners verändert, so ist dies nicht mehr benutzbar. Solche Attacken sind nur durchführbar, wenn der Angreifer im Vorfeld schon bestimmte Erfolge erzielt hat, z. B. das Erlangen von Administrator-Rechten. Daher sollte der Zugang zu Rechnern streng kontrolliert und die Paßwörter von Administrator-Konten regelmäßig geändert werden.

4.5.3 Physikalische Zerstörung

Bei der physikalischen Zerstörung sind Netzwerkkabel, Router und ganze Computer das Ziel. Nach einer Zerstörung durch einen Angreifer muß Ersatz beschafft werden. Auch der einfach Diebstahl von Hardware ist alltäglich. Dies ist mit langen Ausfallzeiten und Wiederbeschaffungskosten verbunden. Fällt eine große Datenbank aus, so sind alle darin gespeicherten Datensätze verloren. Kann ein Angreifer einen Schlüsselserver übernehmen und alle Teilschlüssel löschen, so kann kein legaler Benutzer mehr auf Daten zugreifen.

Dementsprechend sollten Bereiche mit kritischer Hardware gegen den unbefugten Zutritt gesichert werden und regelmäßig Backups angelegt werden.

Denial of Service-Attacken sind in der letzten Zeit sehr häufig geworden, da sie für den Angreifer wenig aufwendig, aber dafür meistens sehr effektiv sind. Sie sind schwierig, wenn nicht gar unmöglich zu verteidigen. Eine erfolgreiche Attacke bedeutet jedoch oftmals nicht das Offenlegen eines Geheimnisses, sondern nur das Verhindern der Benutzung des Systems. Nach einem DoS-Angriff kann deshalb meistens die normale Arbeit wieder aufgenommen werden, da keine Geheimnisse verloren gegangen sind.

Kapitel 5

Implementierung

Die Implementierung der Protokolle, der dazu benötigten Server und Clients sowie der Werkzeuge erfolgte vollständig mit dem *Java-Development-Kit (JDK) Version 1.3* [35].

Im Folgenden wird auf ausgewählte Teile des entwickelten Quell-Codes eingegangen, um einige kritische Techniken zu verdeutlichen, die von entscheidender Bedeutung für die Umsetzung der Protokolle in ausführbare Programme sind.

Zunächst wird die Implementierung der benutzten kryptographischen Verfahren wie der Shared-Secrets-Methode oder des Pseudo-Zufallszahlengenerators vorgestellt und danach auf die Realisierung übergeordneter Instanzen (Rollenproxy, Datenbank und Schlüsselserver, etc.), deren Kommunikation über CORBA und den Update-Mechanismus eingegangen. Abschließend werden die ergänzenden Änderungen am RVChecker beschrieben.

5.1 iSaSilk

Hinsichtlich des Einsatzes kryptographischer Methoden definiert Java mit der *Java-Cryptography-Architecture (JCA)* [38] eine Schnittstelle für den Umgang mit Verschlüsselungs-, Hash- und Authentifikations-Funktionen. Die Implementierungen der Verschlüsselungsfunktionen selbst sind jedoch nicht Teil des JDKs. Dies hat mehrere Gründe. Zum einen unterliegt damit das JDK nicht den Exportbeschränkungen der USA, die verbieten, daß bestimmte Chiffren mit großer Schlüssellänge in andere Länder exportiert werden. Zum anderen erfolgt die Benutzung von Kryptographie-Methoden unabhängig von den darunterliegenden Implementierungen. Der Programmierer benutzt lediglich kryptographische Primitive. Verschiedene *Cryptographic Service Provider* bieten kommerzielle und frei verfügbare Implementierungen von einem oder mehreren kryptographischen Diensten an. Der Quell-Code des Programmierers bleibt unabhängig von der

Implementierung des Providers. Der Provider kann zu jedem Zeitpunkt durch einen anderen ausgetauscht werden. Außerdem ist es möglich, mehrere Provider, die beispielsweise unterschiedliche Teilmengen der JCA bereitstellen, gleichzeitig zu benutzen. Nach der Registrierung der Provider bei der JVM kann der Programmierer explizit festlegen, welche Provider welche kryptographische Funktion ausführen sollen.

Die Technische Universität Graz hat mit *ISASILK* [14] eine sehr umfangreiche Implementierung der JCA entwickelt. Diese besticht durch ihre Fülle von kryptographischen Diensten, wie den aktuellen Algorithmen AES, RSA, DSA, oder SHA-1. Sie bietet darüberhinaus noch die Verwaltung von *X.509*-Zertifikaten sowie den statistischen Test FIPS 140-1 an. Die große Anzahl von implementierten Diensten haben in dieser Arbeit ISASILK den Vorzug gegenüber anderen Implementierungen wie denen von Sun [37], von ABA [3] oder Cryptix [7] gegeben, auch wenn diese im Gegensatz zu ISASILK frei verfügbar sind.

5.2 Shared-Secrets

Das in Abschnitt 2.3 beschriebene Modell für Shared Secrets benötigt einen endlichen Zahlenstrahl, den man durch den Ring $\mathbb{Z}_n$ der ganzen Zahlen modulo n darstellen kann. Das Zusammensetzen von zwei Secrets a und b zum Geheimnis s geschieht dann einfach durch $s = a + b \ mod \ n$. Dabei sollte n sehr groß gewählt werden, beispielsweise in der Dimension $n = 2^{128}$. Derart große Zahlen können in Java mit elementaren Datentypen nicht dargestellt werden. Stattdessen wird eine Klasse `BigInteger` angeboten.

Die selbst entwickelte Klasse `Secret`, implementiert Shared-Secrets in Java. Sie arbeitet daher mit `BigInteger` als Darstellung für Teilgeheimnisse. Die öffentlichen Methoden der Klasse `Secret` sind:

```
public BigInteger getSecret()

public static BigInteger assembleSecrets (Vector secrets)

public static byte[] encryptWithSecrets (Vector secrets,
    byte[] stream)

public static byte[] decryptWithSecrets (Vector secrets,
    byte[] stream)
```

Bei der Generierung einer neuen Instanz von `Secret` wird ein Teilgeheimnis in Form einer zufälligen `BigInteger`-Zahl $mod \ 2^{128}$ erzeugt, auf das mit `getSecret` zugegriffen werden kann. Die kryptographischen Funktionen `encryptWithSecrets` und `decryptWithSecrets` erwarten als Parameter einen

`Vector` von `BigInteger`, die als einzelne Teilgeheimnisse interpretiert werden. Beide Chiffriermethoden benutzen die Funktion `assembleSecrets`, die die einzelnen Teilgeheimnisse zusammenknüpft und daraus einen symmetrischen AES-Schlüssel konstruiert. Mit diesem Schlüssel können dann die Daten in `stream` chiffriert werden.

5.3 Generieren von Zufallszahlen

Bei der Implementierung der Protokolle in dieser Arbeit wurde der PRNG aus Abschnitt 2.5 zur Generierung von Zufallszahlen verwendet.

Realisiert wird der PRNG in der neuen Klasse `MISRandom`. Der interne Zustand des Generators wird in der Variablen `MISSeed` gespeichert. Die Abfrage einer neuen Zufallszahl geschieht über die Methode `getNextByte`.

```
private static BigInteger MISSeed;

public byte getNextByte() {
        //get new entropy
5       MessageDigest sha = MessageDigest.getInstance
                                                ("SHA");
        sha.update(getEntropy());
        BigInteger entropy = (new BigInteger (sha.
                                            digest()));
```

Zunächst wird von einem *Entropy*-Objekt, das die Klasse bei ihrer Initialisierung vom Konstruktor übergeben bekommt, neue Entropie erzeugt.

Diese wird wie beschrieben zum alten Zustand des Generators addiert und ausgegeben. Dabei generiert die Methode aus dem `BigInteger` ein einzelnes Ausgabe-Byte durch eine XOR-Verknüpfung aller Bytes des `BigInteger`:

```
10      //compute new random byte
        sha.reset();
        sha.update((entropy.add(MISSeed).mod(modulus)).
            toByteArray());
        byte[] output = sha.digest();
15      BigInteger bigout = (new BigInteger (output));

        byte result=0;
        for (int i=0;i<output.length;i++)
            result = (byte)(result^output[i]);
20
        sha.reset();
        sha.update(bigout.add(entropy).toByteArray());
        bigout = new BigInteger(sha.digest());
        MISSeed = (MISSeed.add(bigout).mod(modulus));
```

```
25                  return result;
            }
```

Schließlich wird dieses Byte nach einem weiteren Hashen ausgegeben.

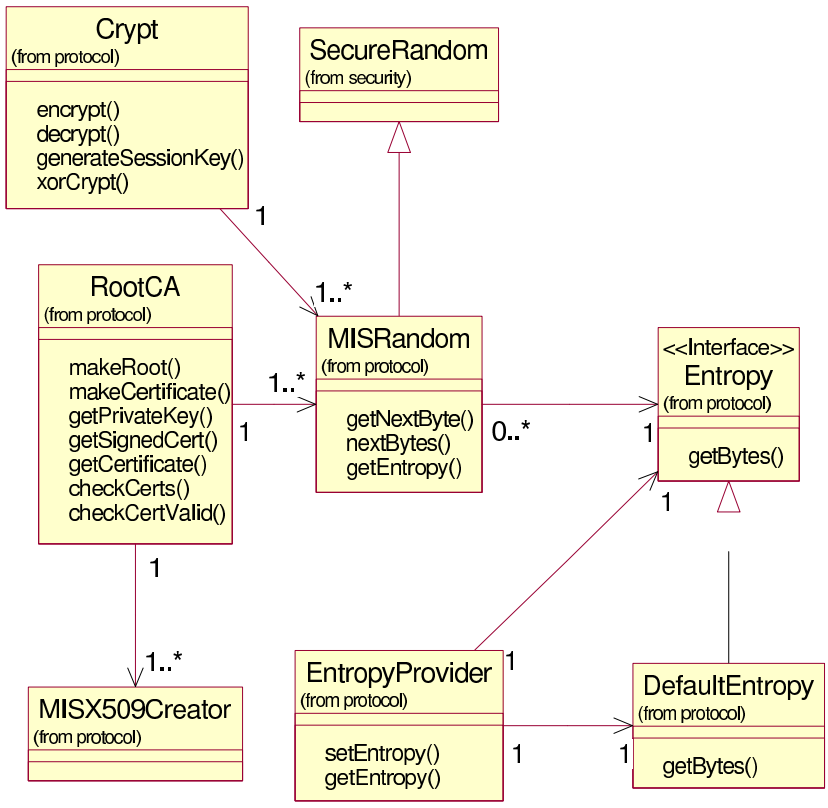

Abbildung 5.1: UML-Diagramm des Zufallszahlengenerators

Der komplette Entropie-Mechanismus ist neu entwickelt. Klassen vom Typ `Entropy` sind über die Methode `getBytes` in der Lage, Entropie zu erzeugen. Die Klasse `DefaultEntropy` ist vom Typ `Entropy` und versucht, aus den Tastaturanschlägen des Benutzers Entropie zu erzeugen. Mit `EntropyProvider` werden mehrere `Entropy`-Objekte verwaltet. Es ist möglich, gezielt eine bestimmte Form der Entropie-Gewinnung auszuwählen. Der Programmierer kann die zu benutzende `Entropy` selbst bestimmen, indem er entweder eine eigene `Entropy` Klasse schreibt oder die vorhandene bereits implementierte `DefaultEntropy` benutzt.

Auf diese Weise können dem System neue Formen der Entropie hinzugefügt werden.

Die Erweiterbarkeit über beliebige `Entropy`-Objekte gestattet beispielsweise auch das Erfassen von Mausbewegungen auf einer graphischen Benutzeroberfläche. Gegenüber dem Auslesen von Tastenanschlägen haben derartige Verfahren den Vorteil, deutlich bessere Zufallszahlen zu generieren, da Menschen aufgrund verschiedener Handhaltungen oder Buchstabenvorlieben nicht alle Tasten gleichverteilt auswählen.

Der Zusammenhang zwischen den Klassen `MISRandom`, `Entropy` und `EntropyProvider` sowie deren Abhängigkeit von zusätzlichen Protokollklassen ist im UML-Diagramm in Abbildung 5.1 dargestellt.

`MISRandom` ist eine Unterklasse der Java-Bibliothek-Klasse `SecureRandom` und damit von allen elementaren kryptographischen Methoden als PRNG benutzbar. `MISRandom` benutzt zum Erzeugen von (Pseudo-)Zufall Klassen vom Typ `Entropy` und zwar die, die vom `EntropyProvider` zur Verfügung gestellt werden. Eine solche Klasse ist die Klasse `DefaultEntropy`.

Die neuen Klassen `Crypt` und `RootCA`, die u. a. für die Generierung von Sitzungsschlüsseln oder Zertifikaten zuständig sind, benutzen `MISRandom`.

5.4 ProcessQueryWrapper

Die entwickelte Klasse `ProcessQueryWrapper` dient dem Benutzer als Schnittstelle für Zugriffe auf Datensätze. Idealerweise sollte der Benutzer von einer vorhandenen Kryptographie-Schicht, über die seine Transaktionen laufen, nichts mitbekommen. Er übergibt seine gewünschte Transaktion, d. h. eine Schreib-, Lese- oder Löschanfrage, an ein `ProcessQueryWrapper`-Objekt, und dieses übernimmt sämtliche weitere Arbeit für ihn. Der `ProcessQueryWrapper` bekommt vom Benutzer als Eingabe ein Objekt vom Typ `QueryData`, mit dem die Anfrage spezifiziert wird. `QueryData` besitzt folgende Attribute:

```
protected String command;
protected Marker marker;
protected byte[][] dataBlocks;
```

Eine Anfragespezifikation besteht demnach aus einem Kommando (z. B. *Schreiben*, *Lesen* oder *Löschen*), einem Marker zur Angabe des betreffenden Datensatzes in Form einer URL und beliebigen Datenblöcken, in denen zu übertragene Daten übergeben werden.

Die einzige Methode, die der Benutzer für eine Transaktion aufruft, ist `processQuery`.

```
public QueryData processQuery (QueryData input)
```

Er übergibt der Methode ein `QueryData` und bekommt ein `QueryData` mit der Antwort zurückgeliefert. Der `ProcessQueryWrapper` analysiert das Kommando und initiiert damit einen Protokolldurchlauf mit dem Rollenproxy.

Mit `ProcessQueryWrapper` kann der Benutzer sich außerdem noch für *Updates* registrieren (siehe Abschnitt 5.6).

5.5 Rollenproxy

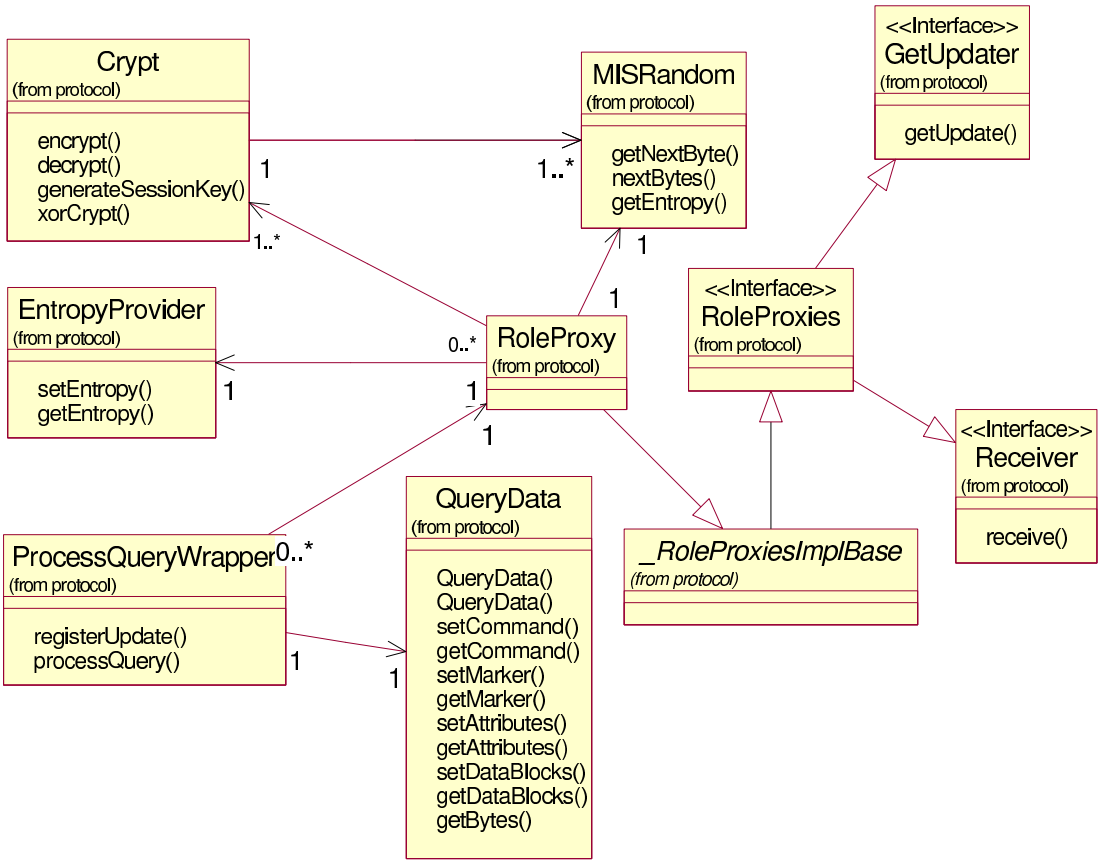

Abbildung 5.2: UML-Diagramm des Rollenproxys

Der Rollenproxy ist in der neuen Klasse `RoleProxy` implementiert und stellt damit das zentrale Objekt der Protokolle dar. Er übernimmt für den Benut-

zer sämtliche kryptographischen Aufgaben, implementiert die Kommunikation und die benutzerseitige Verwaltung der Rollen. Der Rollenproxy wird über den `ProcessQueryWrapper` gesteuert. Der Zusammenhang ist im UML-Klassendiagramm in Abbildung 5.2 dargestellt.

Über einen `EntropyProvider` bekommt der Rollenproxy Zugang zur aktuellen Entropie. Damit kann er die Funktionen zum Generieren von Schlüsseln und zum Chiffrieren der Klasse `Crypt` nutzen. Da jeder Rollenproxy Ziel der Update-Nachrichten von Rollenservern sein kann, muß er die entsprechende Methode `getUpdate` zu deren Empfang bereitstellen. Empfängt der Rollenproxy eine gültige Update-Nachricht, so werden die Geheimnisse der Rolle, z. B. die geheimen Schlüssel, aktualisiert. Für den Einsatz im Zyklischen Protokoll muß außerdem beim Lese-Zugriff auf die Datenbank eine Methode `receive` zum Empfang des mit $K_{RP''}$ (siehe Abschnitt 3.3.2) chiffrierten Datensatzes bereitstehen. Mit $K_{RP''}$ wurde der geheime Schlüssel zwischen Rollenproxy und Datenbank bezeichnet. Die Methode `receive` wird von der Transferstation aufgerufen.

5.5.1 Crypt

Der Rollenproxy macht ausführlichen Gebrauch von Objekten der Klasse `Crypt`. Diese Klasse bietet Chiffrier- und Dechiffrier-Operationen zum Verschlüsseln von Datensätzen an und ist in der Lage, Sitzungsschlüssel zu generieren. Dafür stellt `Crypt` folgende Methoden zur Verfügung:

```
public static synchronized byte[] encrypt (byte[]s, Key pk)
public static synchronized byte[] decrypt (byte[]s, Key pk)
public synchronized static SecretKey generateSessionKey()
public synchronized static byte[] xorCrypt (byte[] daten,
                                            byte[] MISSessionKey)
```

Mit `encrypt` und `decrypt` können asymmetrische (RSA) und symmetrische (AES) Verschlüsselungen durchgeführt werden. Die Methoden können anhand des Parameters `Key` erkennen, ob es sich um eine RSA oder AES Verschlüsselung handeln soll. Mit `generateSessionKey` wird ein neuer symmetrischen Sitzungsschlüssel generiert. Die Methode `xorCrypt` implementiert die XOR-Chiffre für das Zyklische Protokoll. Die Eingabe ist hier nicht wie bei den Methoden `encrypt` und `decrypt` ein Schlüsseltyp, sondern eine zweite Byte-Folge, die den *One-Time-Pad* enthält (siehe Abschnitt 2.4).

5.5.2 Rollenverwaltung

Über den Rollenproxy kann sich der Benutzer unter verschiedenen Rollen anmelden oder freiwillig abmelden.

```
    public void registerMe (RoleServer roleserver, String RoleID,
                                                    String password)

    public String unregisterMe (RoleServer roleserver, String
5                                          RoleID, String password)
```

Dem Benutzer bietet der Rollenproxy dafür die Methoden `registerMe` und `unregisterMe` an. Der Rollenserver entscheidet anhand des mitgelieferten Paßwortes, ob die geforderte Rolle eingenommen werden kann. Das ist eine einfache Form der Überprüfung. In zukünftigen Implementierungen soll der Rollenserver Zugriff auf Dienstpläne haben und dadurch mit komplexeren Regeln Entscheidungen zur Rollenvergabe treffen. Außerdem soll erzwungenes Abgeben von Rollen möglich werden: Endet beispielsweise die Schicht eines Arztes, so wird ihm in der Regel zwangsläufig die Rolle entzogen, die er zur Ausübung seines Dienstes innehatte.

5.5.3 Durchführung der Protokolle

Unter der Voraussetzung, daß sich der Benutzer einen Rollenproxy generieren konnte und eine gültige Rolle eingenommen hat, funktioniert die Abarbeitung eines Protokollzugriffs folgendermaßen: Der Benutzer erzeugt zur Spezifikation seiner Anfrage ein `QueryData`-Objekt sowie eine Instanz von `ProcessQueryWrapper`, mit der die Anfrage durchgeführt wird. Das `ProcessQueryWrapper`-Objekt interpretiert den Anfrage-Spezifikation und ruft entsprechende Methoden im Rollenproxy zur Durchführung der Transaktion auf. In `RoleProxy` sind dafür folgende Methoden definiert:

```
    public Dataset doCyclicRead (String database, String index)
    public String doCyclicInsert(String database, Dataset data)

    public Dataset doSharedRead (String DataBaseServerName,
10                                                  String index)
    public String doSharedInsert(String DataBaseServerName,
                                                    Dataset data)
```

`doCyclicRead` und `doSharedRead` lesen mit Zyklischem, bzw. Shared Protokoll aus einer Datenbank den Datensatz mit der ID `index` aus. Umgekehrt schreiben `doCyclicInsert` und `doSharedInsert` in eine Datenbank hinein. Die `read`-Methode liefern einen Datensatz vom Typ `Dataset`, eine Form von `Markern`. Die `insert`-Methoden erwarten `Datasets` als Parameter. Die Konvertierung von `Marker` nach `Dataset` ist Aufgabe des `ProcessQueryWrapper`.

Die Transaktionen werden nur dann von der Datenbank, den Schlüsselservern oder Transferstationen bearbeitet, wenn die besprochenen Zugriffsberechtigungen gültig sind.

5.5.4 Sicherheit der RoleProxy-Klasse

Die Sicherheit von Java-Objekten kann durch einige programmiertechnische Verfahren erheblich verbessert werden. In [11] werden einige dieser Verfahren vorgestellt, die auch bei den Implementierungen in dieser Arbeit eingesetzt wurden. Am Beispiel von `RoleProxy` soll die Wirkungsweise der Verfahren beschrieben werden.

Kopieren von Objekten

Damit `RoleProxy`-Objekte, und damit ihre Geheimnisse nicht zu einfach kopiert werden können, wird die Methode `clone` in der Klasse überschrieben. Diese Methode legt nach Sprachspezifikation eine byteweise Kopie des Objektes im Speicher der JVM an. Mit folgendem Programm-Code ist dies nicht mehr möglich:

```
public final Object clone() {
    throw new java.lang.CloneNotSupportedException();
}
```

Anstelle einer Kopie des Objekts anzulegen, tritt beim Aufruf von `clone` jetzt eine `Exception` auf, d. h. es wird kein Kopie erzeugt.

Serialisieren von Objekten

Das *Serialisieren* von Objekten wird ebenso verboten. Serialisieren bedeutet, daß der Zustand des Objektes, das sind alle Attribute und alle Methoden, in eine Folge von Bytes geschrieben werden. Ein Angreifer könnte beispielsweise ein `RoleProxy`-Objekt serialisieren und so einfach in dem resultierenden Byte-Strom nach geheimen Schlüsseln suchen. Dies erspart ihm wesentlichen Aufwand, da das Byte-Array für ein Objekt im Vergleich zum kompletten Speicher der JVM relativ klein ist. Die Methode, die dazu überschrieben wird, heißt `writeObject`.

Deserialisieren von Objekten

Noch gefährlicher als das Serialisieren ist das *De-serialisieren* eines Objektes. Hier wird aus einer Byte-Folge wieder ein gültiges Objekt rekonstruiert. Dieser Vorgang ist deshalb so gefährlich, weil ein Angreifer auf irgendeine Art an eine Byte-Folge als Repräsentation eines `RoleProxy`-Objekts gekommen sein könnte, und diese dann modifiziert hat. Durch das Deserialisieren wird aus der modifizierten Byte-Folge ein u. U. bösartiger Rollenproxy. Um das zu verhindern wird `readObject` überschrieben.

```
    private final void writeObject(ObjectOutputStream out)
                                        throws java.io.IOException {
        throw new java.io.IOException("Object cannot be
        serialized");
    }

    private final void readObject(ObjectInputStream in)
                                        throws java.io.IOException {
        throw new java.io.IOException("Class cannot be
        deserialized");
    }
```

Wie im vorigen Beispiel wird wieder nur eine Ausnahme produziert.

5.6 Update-Mechanismus

Neben dem Generieren von Anfragen hat der Benutzer noch die Möglichkeit, sich bei Datenbanken für bestimmte Datensätze zu registrieren (siehe Abschnitt 3.5). Registrieren bedeutet, daß die Datenbank dem registrierten Benutzer ein Benachrichtigungs-Signal schickt, wenn sich der Zustand des Datensatzes geändert hat. Das Registrieren für solche *Updates* geschieht mit der Methode `registerUpdate` im `ProcessQueryWrapper`:

```
public void registerUpdate(QueryData query, GetUpdater
                                                toregister)
```

Der Benutzer übergibt der Methode ein `QueryData`, mit dem die Datenbank und der enthaltene Datensatz spezifiziert wird. Bei dieser Datenbank wird der Benutzer für diesen Datensatz registriert. Der zweite Parameter ist ein vom Benutzer implementierter *GetUpdater*, der über die Methode `getUpdate` aufgerufen wird, wenn der Datensatz verändert wurde.

Zum Empfang und zur Verarbeitung der Änderungsbenachrichtigung implementiert der Benutzer die abstrakte Methode `getUpdate`, so daß er sich in `registerUpdate` als Empfänger `toregister` registrieren kann.

```
void getUpdate (byte[] sessionKey, byte[] update)
```

In `sessionKey` steht ein ausgehandelter und sicher übertragener Sitzungsschlüssel, mit dem die Information `update` mit dem Namen des geänderten Datensatzes chiffriert wurde.

Nachdem so die Information über einen geänderten Datensatz ausgetauscht wurde, kann der Datensatz entsprechend der Protokolle neu gelesen werden.

5.7 Benutzen der Protokolle

Das Benutzen der vom Rollenproxy unabhängigen Protokollabläufe, wie über Transferstation und Datenbank, wurden nach dem Entwurfsmuster *Dekorierer* entworfen [10]. Ein Dekorierer (engl. *Wrapper*) paßt die Schnittstelle eines anderen Objektes für den Zugriff des Benutzers an. Der Dekorierer übersetzt dazu die Anfragen des Benutzers und leitet sie an das Objekt weiter.

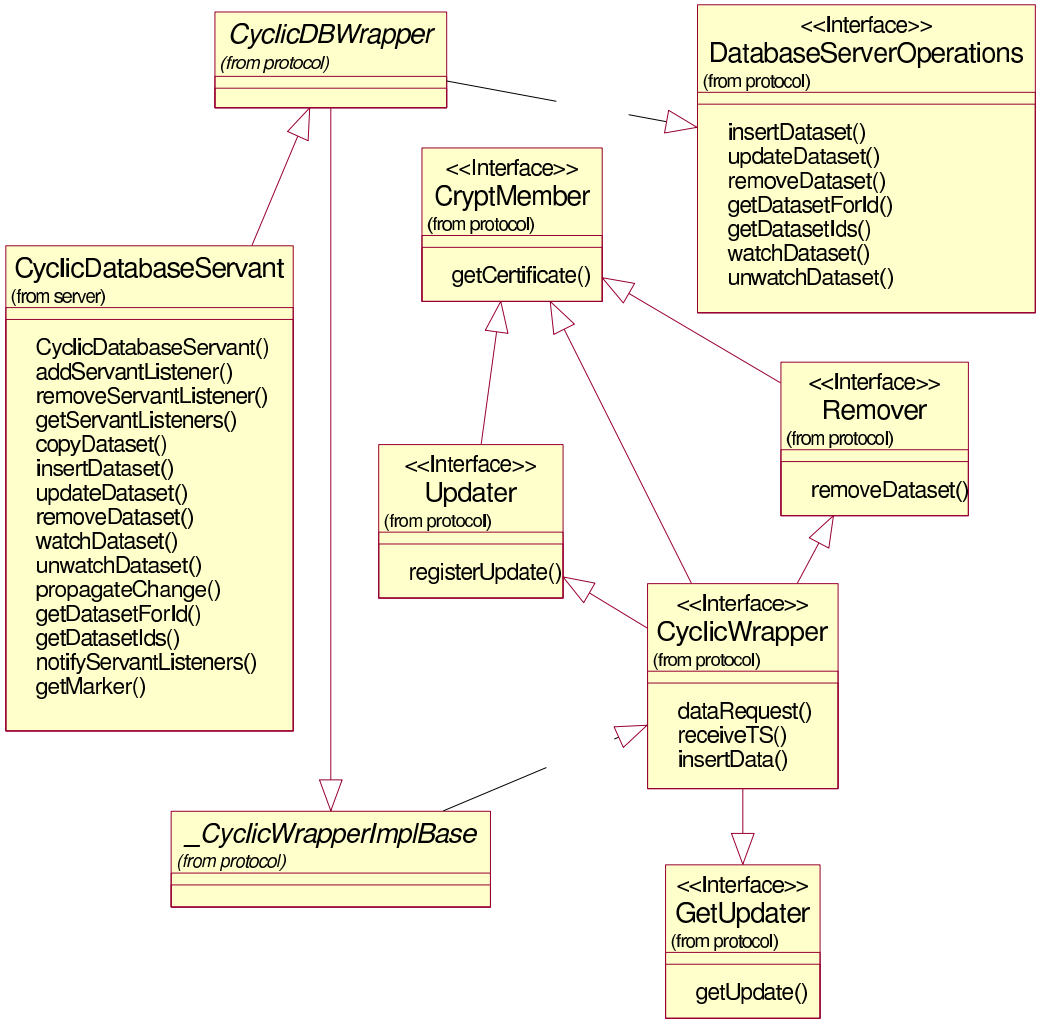

Abbildung 5.3: UML-Diagramm des zyklischen Datenbank-Servers

Die Datenbank `CyclicDatabaseServant` aus Abbildung 5.3 ist beispielsweise für das Speichern und Laden von Datensätzen zuständig und wird von der Klasse `CyclicDBWrapper`, dem Dekorierer, umschlossen: der Benutzer ruft nicht die Methoden vom `CyclicDatabaseServant` auf, sondern die vom logisch darüberliegenden `CyclicDBWrapper`. Der Wrapper ist für das Übersetzen der Kryptographie-

Aufrufe aus `CyclicWrapper` in Aufrufe aus `DatabaseServerOperations` zuständig, die der `CyclicDatabaseServant` implementieren muß. Jeder `CyclicWrapper` hat ein eigenes Zertifikat (Klasse `CryptMember`), kann Benutzer für Updates (Klasse `Updater`) registrieren, oder selbst vom `CyclicDataBaseServant` über Updates benachrichtigt werden (Klasse `GetUpdater`).

Dieses Entwurfsmuster wurde gewählt, da so die verschlüsselten Methoden in vorhandene Datenbankimplementationen integriert werden konnten, und die kryptographischen Protokollmethoden unabhängig von den darunter liegenden Datenbanken sind. Der *CyclicDBWrapper* und analog auch der *SharedDBWrapper* des Shared Protokolls greifen auf Methoden von Objekten des Typs `DataBaseServerOperations` zu.

Die Klasse `_CyclicWrapperImplBase` wird durch CORBA bereitgestellt und ermöglicht den entfernten Zugriff auf `CyclicDBWrapper`-Methoden.

5.8 CORBA

Die Kommunikation über CORBA (siehe Abschnitt 4.3.3) gestaltet sich aus Java heraus denkbar einfach. Um eine Methode eines „entfernten“ Objekts aufzurufen, wird zunächst über Hilfsklassen eine Referenz auf das Objekt beschafft.

```
CyclicWrapper db2 = CyclicWrapperHelper.narrow(
                    ComManager.resolve(database));
```

In diesem Beispiel ist `CyclicWrapperHelper` die Hilfsklasse für Objekte vom Typ `CyclicWrapper`. Im ersten Schritt wird über die Methode `narrow` und einem symbolischen Namen, der in `database` gespeichert ist, das `CyclicWrapper`-Objekt lokalisiert. Es ist grundsätzlich möglich, CORBA-Objekten Klartext-Namen zuzuordnen, und darüber auf sie zuzugreifen. Nach dem ersten Schritt ist `db2` eine Referenz auf das angeforderte CORBA-Objekt.

Der Aufruf der entfernten Methode geschieht im zweiten Schritt transparent, als wäre die Methode Teil eines lokalen Objektes.
Beispiel:

```
X509Certificate dbcert = new X509Certificate (
                         db2.getCertificate());
```

Damit wird die Methode zum Übertragen des Zertifikats auf dem entfernten CORBA-Objekt aufgerufen und das Ergebnis in `dbcert` gespeichert.

5.9 Änderungen am RVChecker

Der *RVChecker* ist ein komplett in Java geschriebener *Analysierer* für die *ReVere*-Logik. Als Eingabe liest er eine formale Protokoll-Beschreibung in Form von Java-Code ein, analysiert diese und gibt die Ergebnisse der Analyse aus. Das komplette Regelwerk ist in Java geschrieben worden und kann daher problemlos erweitert werden. Dies ist, wie in Abschnitt 4.2.2 beschrieben, zur Analyse des Zyklischen Protokoll notwendig.

Zunächst zur Regel 1 aus Abschnitt 4.2.2, die es einem Sender ermöglicht, eine kommutative Chiffre zu erzeugen. Die Regel lautet:

$$\frac{canProduce(P, encrypt(encrypt(X, K1), K2))}{canProduce(P, encrypt(encrypt(X, K2), K1))}.$$

Nach Java übersetzt ergibt dies:

```
protected Rule produces_kommu() {
    Vector premises = new Vector();
    premises.addElement(canProduce(P,encrypt(encrypt(X, K1),
    K2)));
    return new Rule(premises, canProduce(P,encrypt(encrypt(X,
    K2), K1)));
}
```

In der Liste `premises` werden die Bedingungen gespeichert, die erfüllt sein müssen, damit diese Regel (`Rule`) angewandt wird.

Regel 1 trifft keine Aussage über die Sichtbarkeit vom Geheimnis `X`. `P` kann nur die umgekehrte kommutative Chiffre generieren, er kommt nicht in den Besitz von `encrypt(X, K2)`, da er `K1` nicht kennt.

Regel 2 ermöglicht nach dem Empfang eines kommutativ chiffrierten Datensatzes, diesen zu entschlüsseln:

$$\frac{sees(P, encrypt(encrypt(X, K1), K2))}{sees(P, encrypt(encrypt(X, K2), K1))},$$

was in Java folgendermaßen formuliert wird:

```
protected Rule kommutative_chiffre() {

premises.addElement(sees(P, encrypt(encrypt(X,K1), K2)));
return new Rule(premises, sees(P, encrypt(encrypt(X,K2), K1)));
}
```

5.10 Werkzeuge

In dieser Arbeit sind neben den beiden Protokollen zwei Werkzeuge für den medizinischen Einsatz implementiert worden, die im gewissen Maße unabhängig von den Protokollen sind. Ein Werkzeug behandelt die Möglichkeit, *Annotationen* zu verwalten, das andere Werkzeug pseudonymisiert die Zugriffe von Personen.

5.10.1 Annotationen

Mit *Annotationen* haben Ärzte die Möglichkeit, an einen Patienten-Datensatz mehrere Notizen, Bemerkungen oder Diagnosen anzufügen. Grundsätzlich können ganze Krankengeschichten damit aufgezeichnet werden.

Eine Annotation ist eine Notiz eines Arztes A zu einem Patienten, die digitale Unterschrift von A zu dieser Notiz sowie sein Zertifikat.

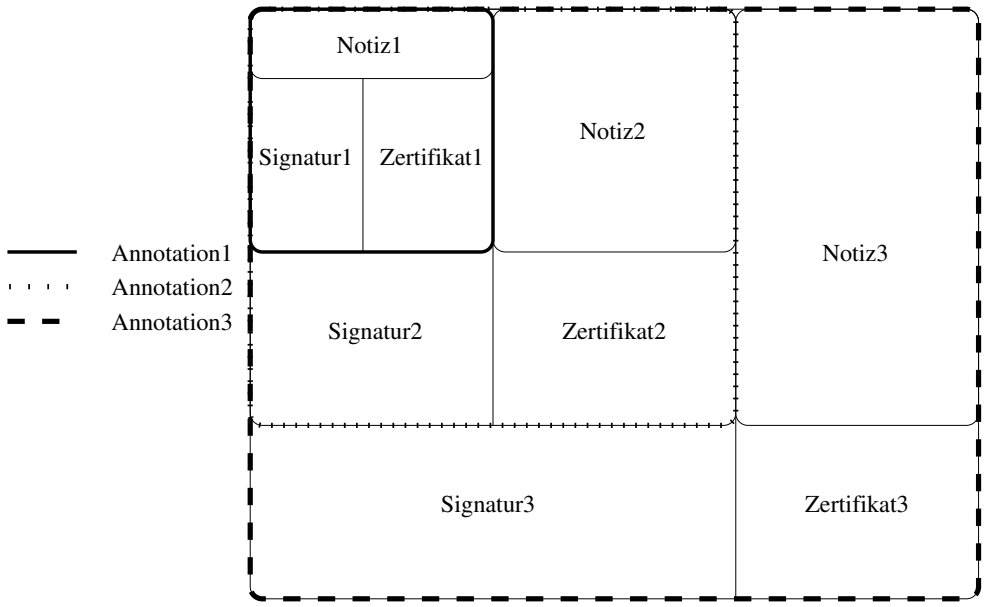

Abbildung 5.4: Kaskade von drei Annotationen

Ein anderer Arzt B kann zu einem späteren Zeitpunkt überprüfen, ob die ihm vorliegende Annotation gültig von A unterschrieben wurde, indem er Unterschrift und Zertifikat zur Notiz der Annotation verifiziert. Danach ist B sicher, daß A die Unterschrift gefertigt hat, A demnach für den Inhalt der Notiz, z. B. eine Fehldiagnose, verantwortlich ist. B selbst kann zu der vorhandenen Annotation eine weitere Notiz oder Diagnose hinzufügen, indem er die komplette vorhandene Annotation und seine neue Notiz signiert. Die neue Annotation besteht dann aus

der alten Annotation, der neuen Notiz, der Unterschrift und dem Zertifikat von B (siehe Abbildung 5.4). Mit der Unterschrift unter der alten Annotation übernimmt B auch Verantwortung über die enthaltenen Notizen. Zumindest billigt er, oder glaubt den Notizen der Ärzte vor ihm. Auf diese Weise können ganze Ketten von Annotationen mehrerer Ärzte zu einem Datensatz entstehen, die folgende Eigenschaften innehaben:

- Sie sind **fälschungssicher**. Wird zu einem Zeitpunkt eine Notiz, oder Unterschrift eines Arztes verändert, passen entsprechende Zertifikate, Notizen und Unterschrift nicht mehr zusammen. Es kann dann davon ausgegangen werden, daß beispielweise die angegebenen Diagnosen nicht stimmen.

- Sie sind **verbindlich**. Ein Arzt kann nicht abstreiten, eine Unterschrift zu einer Annotation erzeugt zu haben. Nur derjenige kann eine Unterschrift zu einer Notiz und einem Zertifikat erzeugen, der im Besitz des dazu passenden geheimen Schlüssels ist. Ein Arzt kann keine Zertifikate fälschen, da diese von einer *Certificate Authority* (siehe Abschnitt 2.2.2) unterschrieben wurden, dessen Unterschrift er nicht fälschen kann.

- Aus dem selben Grund folgt auch, daß Annotationen **authentisch** sind. Überprüft B Zertifikat und Unterschrift zu einer Annotation von A, dann ist er sicher, daß A diese Annotation angefertigt hat.

Ein böswilliger Arzt kann zwar eigene Annotationen erzeugen und signieren, jedoch nur mit seiner Unterschrift und seinem passenden Zertifikat, jeder Betrug würde sofort auffallen.

Struktur einer Annotation

Zunächst zur Klasse `PatientAnnotation`. Dies ist die Repräsentation einer einzelnen Annotation, d. h. einer Notiz, der dazu passenden Unterschrift und dem Zertifikat des Unterschreibers.

```
private String body;
private X509Certificate cert;
private byte[] signature;
```

Dies sind die Attribute jedes `Annotation`-Objektes. Die Zeichenkette `body` beinhaltet die eigentliche Notiz, `cert` das *X.509*-Zertifikat und in `signature` wird die Signatur gespeichert. Über einen Konstruktor kann eine einzelne neue Annotation erzeugt werden. Darüberhinaus existieren in der Klasse Methoden für den Zugriff auf die einzelnen Attribute.

Patientenakten

Wesentlich aufwendiger ist dagegen die Klasse `PatientRecord`. Diese implementiert die einzelnen Operationen, um eine ganze Kette von Annotationen zu verwalten.

```
public class PatientRecord extends Dataset {

  private Vector annotations;
  private String identity;

  public PatientRecord() {
    super(); //create a new PatientRecord
    type = Dataset.PATIENT_RECORD;
    objectName = "PatientRecord";
    annotations = new Vector();
    identity = "";
  }

  public PatientRecord(String id) {
    super();//create a new PatientRecord with specified id
    type = Dataset.PATIENT_RECORD;
    objectName = "PatientRecord";
    annotations = new Vector();
    identity = id;
  }
```

Ein `PatientRecord` besteht demnach aus einer Liste von Annotationen, die verwaltet werden sowie einem Namen. Dieser Name könnte z. B. der Name eines Patienten, dessen Pseudonym, eine Seriennummer oder ähnliches sein. Die beiden Konstruktoren sind nötig, da `PatientRecord` von der Klasse `Dataset` erbt. Auf diese Weise wird eine spätere Integration der Annotation-Liste in die vorhandene graphische Benutzeroberfläche vereinfacht.

Überprüfung der Annotationen in einer Patientenakte

Die folgende Methode `checkSignatures` überprüft, ob die Menge der enthaltenen Annotationen jeweils korrekt signiert wurden. Notwendig ist eine Überprüfung aller Signaturen zu ihren Zertifikaten und den gemachten Notizen. Der Rückgabewert der Methode ist die Nummer der ersten Annotation, bei der die Überprüfung fehlschlug, oder -1, falls alle Annotationen korrekt signiert wurden. Falls ein anderer Fehler, eine unerwartete Ausnahme o. ä. auftritt, ist der Rückgabewert -2.

```
  public int checkSignatures() {
    Signature sig = Signature.getInstance("SHA/RSA");
```

```
25      PatientAnnotation pa = (PatientAnnotation)annotations.
        elementAt(0);
        sig.initVerify(pa.getCertificate().getPublicKey());
        sig.update(pa.getBody().getBytes());
        sig.update(pa.getCertificate().getEncoded());
```

Der erste Eintrag bzw. die erste Annotation in einer Patientenakte muß separat von den anderen Annotationen verifiziert werden, da hier nur die Unterschrift zu dem Zertifikat und der (ersten) Notiz überprüft wird.

```
30      if (sig.verify(pa.getSignature())) {
          //and now go for the rest
          for (int i=1;i<annotations.size();i++) {
            sig = Signature.getInstance("SHA/RSA");
            pa = (PatientAnnotation) annotations.elementAt(i);
35          sig.initVerify(pa.getCertificate().getPublicKey());
    5A
            for (int j=0;j<i;j++) {
              pa = (PatientAnnotation) annotations.elementAt(j);
              sig.update(pa.getBody().getBytes());
40            sig.update(pa.getSignature());
              sig.update(pa.getCertificate().getEncoded());
            }

            pa = (PatientAnnotation) annotations.elementAt(i);
45
            sig.update(pa.getBody().getBytes());
            sig.update(pa.getCertificate().getEncoded());
            if (!sig.verify(pa.getSignature())) return i;
          }
50        return -1; //everything ok
        }
        else return 0;
      }
```

Für jede der Annotationen wird überprüft, ob die Unterschrift über alle vorherigen Annotationen korrekt ist, d. h. für die n-te Annotation werden die Annotationen $2 \ldots n-1$ durchgesehen und das n-te Unterschrift/Zertifikat/Notiz-Paar.

Hinzufügen von Annotationen

Die folgende Methode wird zum Hinzufügen einer neuen Annotation zu einer vorhandenen Liste von Annotationen verwendet. Sie bereitet die nötige Unterschrift insofern vor, daß nur noch die neue Notiz dem Signatur-Mechanismus übergeben werden muß.

```
public static void prepareSig (Vector vec, Signature
```

```
55                                                  signature) {
      Enumeration e = vec.elements();
      while (e.hasMoreElements()) {//run through all annotations
        PatientAnnotation pa = (PatientAnnotation)
                                                    e.nextElement();
60      signature.update(pa.getBody().getBytes());
        signature.update(pa.getSignature());
        signature.update(pa.getCertificate().getEncoded());
      }
    }
```

Damit kann nun eine neue Annotation einfach zu einer vorhandenen Liste hinzugefügt werden. Dies übernimmt die Methode `addAnnotation`

```
65
    public boolean addAnnotation (String anno, byte[] Sig,
    X509Certificate x){

      x.checkValidity();
70    Signature signature =  Signature.getInstance("SHA/RSA");
      signature.initVerify(x.getPublicKey());
      //get all the other annotations and signatures and
                                                  check them
      prepareSig (annotations, signature);
75
      signature.update(anno.getBytes());
      signature.update(x.getEncoded());

      if (signature.verify(Sig)== false ) {
80      System.out.println("Signature not valid!");
        return  false ;
      } else {
        PatientAnnotation p = new PatientAnnotation(x, anno,
                                                      Sig);
85      annotations.add(p);
        propagateChange("annotations");

        System.out.println("Annotation added");
        return  true ;
90    }
    }
```

Zunächst überprüft die Methode natürlich, ob die bisherige Liste der Annotationen korrekt signiert wurde. Erst dann wird von der bisherigen Liste der Annotationen sowie von der neuen Notiz eine Unterschrift generiert.

Die Klasse `PatientRecord` wird durch Methoden für den `Dataset`-Typ und Zugriffsmethoden auf die Attribute vervollständigt.

5.10.2 Pseudonymisierer

Ein Pseudonym ist eine Möglichkeit, Identitäten zu verbergen. Einzelne Individuen sollen im System nicht unbedingt mit ihrem wirklichen Namen oder Identifikations-String agieren, sondern geschützt durch ein Pseudonym, einen Decknamen. Der Unterschied zum anonymen Handeln ist die Möglichkeit, von einer bestimmten autorisierten Instanz in Ausnahmefällen vom Pseudonym zurück auf den tatsächlichen Namen zu schließen.

In bestimmten Situationen ist es wünschenswert, daß Patienten in einer Datenbank nicht unter ihrem richtigen Namen gespeichert sind, sondern unter einem Pseudonym. Beispielsweise kann zu statistischen Zwecken eine Datenbank mit HIV-Patienten angelegt werden, bei der die Namen der einzelnen Patienten nicht von Bedeutung sind. Wird jedoch ein Heilmittel für HIV gefunden, so ist es wichtig, zu den einzelnen Pseudonymen die Namen zu erfahren. Genauso sollen auch Benutzer des Informationssystems im System selbst unter einem Pseudonym auftreten können. Mit Pseudonymen gibt es noch einige Unklarheiten, weshalb diese nicht im Protokoll zum Einsatz kommen. Auf die Unklarheiten wird später eingegangen.

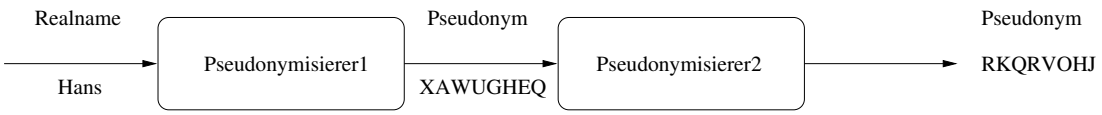

Abbildung 5.5: Pseudonymisieren in zwei Stufen

Mehrstufiges Pseudonymisieren

Das Pseudonymisieren eines Namens in ein Pseudonym wird durch *Pseudonymisierer* durchgeführt. Damit ein einzelner Pseudonymisierer nicht alleine die Zuordnung von Namen in Pseudonyme besitzt, wird der eigentliche Pseudonymisierungs-Vorgang von zwei hintereinandergeschalteten Pseudonymisierern realisiert, siehe Abbildung 5.5. Der Name wird zum Pseudonymisierer 1 geschickt, der den Namen pseudonymisiert und das resultierende Pseudonym an Pseudonymisierer 2 weiterschickt. Pseudonymisierer 2 verarbeitet das empfangene Pseudonym und verarbeitet es wie einen echten Namen, d. h. er produziert daraus ein weiteres Pseudonym. Dieses Ergebnis ist dann das tatsächliche zu benutzende Pseudonym. Die Verbindung der beiden Stationen ist über Sitzungsschlüssel gesichert, so daß kein Außenstehender das Teilpseudonym abfangen kann. Wird eine der beiden Stationen kompromittiert, so kennt der Angreifer nur die Hälfte des Pseudonyms. Die Technik und die Sicherheit sind demnach ähnlich der der Shared-Secrets aus Abschnitt 2.3.

Muß nun zu einem Pseudonym wieder der reale Name gefunden werden, so kann eine entsprechend autorisierte Person Zugang zu *beiden* Pseudonymisieren bekommen und dort jeweils die Zuordnungen überprüfen.

Einsatz von Pseudonymen

Das Handeln von Benutzern unter einem Pseudonym ist in unserem System problematisch, da jeder Benutzer dann zu diesem Pseudonym ein gültiges Zertifikat braucht. Ohne ein gültiges Zertifikat ist keine Authentizität und keine gesicherte Kommunikation möglich. Theoretisch könnte das Zertifikat für das Pseudonym von der zweiten Pseudonymisier-Stationen angefertigt werden, was jedoch grobe Sicherheitseinbußen zur Folge hat. Eine kompromittierte Station 2 könnte beliebig Zertifikate erstellen.

Ein weiteres Problem steht im Zusammenhang mit der Pseudonymisierung von Datensätzen. Sollen beispielsweise alle Datensätze in einer Datenbank jeweils unter Pseudonymen gespeichert werden, so ist es nicht mehr möglich, gültige Unterschriften für Transaktionen zu generieren. Ein Benutzer, der nur den Namen eines pseudonymisierten Datensatzes kennt, kann keine Unterschrift für den Zugriff auf das Pseudonym erzeugen, da er es nicht kennt. Ohne Unterschrift zu einer Transaktion ist es jedoch für Datenbanken unmöglich, zu unterscheiden, ob ein bestimmter Nutzer wirklich diese Anfrage getätigt hat oder nicht.

Grundsätzlich wird die Frage aufgeworfen, ob das Pseudonymisierern von Datensätzen oder Teilen von Datensätzen nicht äquivalent zur dessen Verschlüsselung ist. Der Datensatz steht „pseudonymisiert“ in der Datenbank, was prinzipiell dem herkömmlichen Verschlüsseln ähnelt. Die Pseudonymisierer entsprechen den Schlüsselservern. Insoweit könnte sich Pseudonymisierung erübrigen. Dies muß aber noch genauer erörtert werden.

Implementierung der Pseudonymisierer

Die Implementierung der zweistufigen Pseudonymisierung ist prototypisch. Es sind damit keine Zugriffe über das Protokoll möglich, nur der Mechanismus der Pseudonymisierung über CORBA wird damit gezeigt.

Die Klasse MISPseudo ist die Implementierung eines Pseudonymisierers.

```
private X509Certificate MISCert;
private PrivateKey pk;
private X509Certificate rootcert;

private Hashtable realnamen = new Hashtable();
private Hashtable pseudonyme = new Hashtable();
private int nr;
```

Dies sind soweit die Attribute jedes Pseudonymisierers. Für die kryptographisch sichere Kommunikation und Authentizität werden Zertifikat und geheimer Schlüssel benötigt und das Root-Zertifikat. Die Zuordnung Name nach Pseudonym wird mittels der Hash-Tabelle `realnamen` realisiert. Umgekehrt speichert die Hash-Tabelle `pseudonyme` die Zuordnung Pseudonym nach Name. Anhand der Nummer `nr` erkennt der Pseudonymisierer, ob er der erste oder zweite Pseudonymisierer ist, demnach also sein erzeugtes Pseudonym noch an die nächste Station schicken muß oder nicht.

Infolgedessen gestaltet sich die Arbeit mit den Zuordnungen denkbar einfach, wie folgende Methode `giveOut` zeigt. Sie gibt eine Liste aller gespeicherten Namen und deren Pseudonyme aus.

```
    public void giveOut() {

10      Enumeration keys = realnamen.keys();

        while (keys.hasMoreElements()) {
            String key = (String)keys.nextElement();
            System.out.println("R: "+key+" P: "+(String)
15          realnamen.get(key));
        }

    }
```

Die folgende Methode `createPseudonym` erzeugt ein neues Pseudonym. Dabei wird e neuer String generiert, der aus acht Zeichen besteht. Jedes Zeichen ein zufälliger Großbuchstabe. Auf diese Art existieren insgesamt $26^8 \approx 10^{11}$ verschiedene Pseudonyme.

```
    private String createPseudonym() {
20
            String pseudonym = new String();
            for (int i=1;i<=8;i++) {
                pseudonym = pseudonym.concat(String.valueOf
                ((char) (MISRand.nextInt(26) + 65)));
25          }
      return pseudonym;
    }
```

Die Methode `doPseudonym` führt die eigentliche Pseudonymisierung durch. Dieser Methode wird der zu pseudonymisierende Name verschlüsselt übergeben.

```
    public synchronized void doPseudonym
    (byte[] daten, byte[] sessionkey) throws ProtocolException{
30      try {
            SecretKey k = new SecretKey (Crypt.decrypt
```

```
                (sessionkey, pk), "Rijndael");
                String realname = new String(Crypt.decrypt
                (daten, k));
35
                String pseudonym2 = (String) realnamen.get
                (realname);
```

Der Name wird auf die herkömmliche Art entschlüsselt und dessen entsprechendes Pseudonym in der Hash-Tabelle `realnamen` gesucht. Falls dieser Name schon existiert, dann wird das bereits existierende Pseudonym verwendet, sonst ein neues.

```
                String pseudonym;
                if (pseudonym2==null) {
40                  pseudonym = createPseudonym();
                    realnamen.put(realname, pseudonym);
                    pseudonyme.put(pseudonym, realname);
                } else pseudonym = pseudonym2;
```

Das neu erzeugte Pseudonym wird zusammen mit dem Namen in die beiden Hashtabellen gespeichert.

```
                if (nr==1) {
45                  int nr2 = nr+1;
                    try {
                        Pseudonymizer ps =
                        PseudonymizerHelper.narrow
                        (ComManager.resolve
50                      ("Pseudonymisierer"+nr2));
                        k = (SecretKey) Crypt.generateSessionKey();
                        X509Certificate ot = new X509Certificate
                        (ps.getCertificate());
                        if (RootCA.stripDN(ot).compareTo
55                      ("Pseudonymisierer"+nr2)!=0) {
```

Schließlich muß, falls es sich um Pseudonymisierer 1 handelt, das erzeugte Pseudonym an Pseudonymisierer 2 weitergeschickt werden. Das funktioniert, wie schon gesehen, mit Hilfe von CORBA. Es werden wie üblich die Zertifikate überprüft, dann eine Verbindung aufgebaut und das Pseudonym verschlüsselt weitergegeben. Was dann Pseudonymisierer 2 mit dem fertigen Pseudonym macht, ist noch offen. Er kann keinen `processQuery` im Namen des Pseudonyms durchführen, da, wie angesprochen, keine Zertifikate für dieses Pseudonym existieren.

Kapitel 6

Zusammenfassung

In dieser Arbeit wurde gezeigt, wie mit Hilfe kryptographischer Protokolle der Zugriff auf sensible medizinische Datenbanken gesichert werden kann. Dafür wurden zunächst Anforderungen spezifiziert, die Protokolle und Ablaufpläne für einen netzbasierten Daten-Zugriff leisten müssen. Das waren neben den vier klassischen kryptographischen Anforderungen Sicherheit, Integrität, Authentizität und Verbindlichkeit auch Eigenschaften wie Nichtverfolgbarkeit oder Unverkettbarkeit von Informationen und vor allem das Auftreten von Benutzern in *Rollen*, um funktionsbasierte Zugriffsentscheidungen treffen zu können. Geheimnisse, beispielsweise Daten über Patienten, sollten auch dann noch geschützt sein, wenn einer der Protokollteilnehmer durch einen Angriff kompromittiert wurde.

Basierend auf diesen Anforderungen wurden zwei alternative Protokolle für den Zugriff entwickelt, das *Shared Protokoll* und das *Zyklischen Protokoll*. Die Sicherheit des Shared Protokolls basierte auf dem Verteilen der Zugangs-Schlüssel für Datensätze auf zwei unterschiedliche Schlüsselserver. Im Zyklischen Protokoll sicherte neben der Datenbank eine *Transferstation* den Datenaustausch.

Der Protokollspezifikation folgte eine Verifikation mit der Methode *Logic of Authentication* sowie einer schrittweisen Analyse aller logischen Ebenen des Entwurfs, von der algorithmischen Ebene bis hin zur Implementierung. Beide Protokolle wurden unter bestimmten, angegebenen Voraussetzungen als sicher befunden. Teil dieser Voraussetzungen war beispielsweise, daß jeder Benutzer zu Beginn der Protokolle im Besitz gültiger Zertifikate einer obersten Zertifizierungsstelle sein muß und daß die Zertifizierungsstelle auf keinen Fall kompromittiert sein darf.

Neben den Protokollen wurden Gefahren und Bedrohungen für einzelne Komponenten der Protokolle oder der in den Protokollen benutzten Mechanismen identifiziert und analysiert. Das wesentliche Ergebnis dieser Arbeit ist der Beweis der Sicherheit. Dies bedeutet, daß die geforderte Geheimhaltung von Patientenakten und die Privatsphäre von Patienten mit den Protokollen realisierbar ist.

Die beiden Protokolle wurden vollständig in Java implementiert und können somit in ein *Medizinisches Informationssystem* eingebunden werden.

Ausblick

In zukünftigen Arbeiten muß noch untersucht werden, inwiefern das Pseudonymisieren von Datensätzen, Benutzern oder Zugriffen sich im vorhandenen medizinischen Informationssystem als sinnvoll erweist, d. h. wo die Unterschiede zum konventionellen Verschlüsseln liegen. Genauso muß das Verfahren der Rollen-Einnahme u. a. mittels eines Dienstplans oder des Regel- und Zugriffsservers noch genauer analysiert und entworfen werden. In dieser Arbeit wurden prototypische Formen der Regelverwaltung vorausgesetzt.

In der derzeitigen Entwicklung der Protokolle kann nur auf einzeln spezifizierte Datensätze gleichzeitig zugegriffen werden. Jeder Datensatz ist mit einem eigenen Schlüssel chiffriert. In der Praxis ist es jedoch häufig wünschenswert, eine Menge von Datensätzen gleichzeitig zu erfassen, beispielsweise zum Anlegen von Statistiken. Dafür müssen ohne Herabsetzung der Sicherheit Probleme wie das Verschlüsseln mehrerer Datensätze mit Gruppenschlüsseln gelöst werden.

Auf praktischer Seite ist die derzeit mangelhafte Geschwindigkeit der Java-Implementierung zu verbessern. Durchgeführte Versuche haben gezeigt, daß der Start des Systems sich über einige Minuten hin erstreckt, und auch ein einzelner Schreib/Lese-Protokolldurchlauf entschieden zuviel Zeit kostet. Dies ist ein grundsätzliches Problem der Sprache Java, das wohl nur durch eine aufwendige *native*-Implementierung entsprechender mathematischer oder kryptographischer Methoden verbessert werden kann.

Literaturverzeichnis

[1] *Website des Instituts für Algorithmen und Kognitive Systeme.* `http://iaks-www.ira.uka.de/`.

[2] Alexander Nicklas. *Modellierung und Realisierung gesicherter Datenzugriffe in medizinischen Informationssystemen.* Diplomarbeit, November 2000. IAKS, Universität Karlsruhe.

[3] Australian Business Access Pty Ltd (ABA). *Open JCE.* `http://www.openjce.org`.

[4] Bruce Schneier. *Angewandte Kryptographie: Protokolle, Algorithmen und Sourcecode in C, Addison-Wesley Verlag*, 1996. ISBN 3-89319-854-7.

[5] Bruce Schneier und Doug Whiting. *A Performance Comparison of the Five AES Finalists*, April 2000. Third AES Candidate Conference, to appear.

[6] CERT(R) Coordination Center. *Denial of Service Attacks.* `http://www.cert.org/tech_tips/denial_of_service.html`.

[7] Cryptix(TM). *Cryptix JCE.* `http://www.cryptix.org/products/jce/index.html`.

[8] Darrel Kindred. *Theory Generation for Security Protocols*, 1999. Dissertation, School of Computer Science, Carnegie Mellon University.

[9] Federal Information Processing Standards Publication. *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*, 2001. Draft.

[10] Gamma, Helm, Johson, Vli, Marchand, Muzyka. *Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software*, 2001. Addison-Wesley Verlag, ISBN 3-8273-1862-9.

[11] Gary McGraw und Ed Felten. *Securing Java, Getting Down to Business with Mobile Code*, 1999. ISBN 047131952X.

[12] Gus Simmons. *An introduction to shared secret and/or shared control schemes and their application*, 1992. Contemporary Cryptology, The Science of Information Integrity, IEEE Press, S. 441ff.

[13] Ingemar Ingemarsson und Gus Simmons. *A protocol to set up shared secret schemes without the assistance of a mutually trusted party*, 1991. Advances in Cryptology – EUROCRYPT '90, Lecture Notes in Computer Science, Band 473, S. 266ff.

[14] Insitute for Applied Information Processing and Communications, Graz University of Technology. *Isasilk Toolkit*.
`http://jcewww.iaik.at/products/isasilk/index.php`.

[15] Institute for Applied Information Processing and Communications, University of Graz. *IAIK Java Cryptography Extension (IAIK-JCE)*.
`http://jcewww.iaik.tu-graz.ac.at/`.

[16] Joan Daemen und Vincent Rijmen. *The Rijndael Block Cipher*, 2000. AES proposal,
`http://www.esat.kuleuven.ac.be/~rijmen/rijndael/`.

[17] John Kelsey, Bruce Schneier, David Wagner und Chris Hall. *Cryptanalytic Attacks on Pseudorandom Number Generators*, März 1998. Fast Software Encrpytion, Fifth International Workshop Proceedings .

[18] Li Gong, Marianne Mueller, Hemma Prafullchandra und Roland Schemers. *Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java(TM) Development Kit 1.2*. Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California, Dezember 1997.

[19] Martin Abadi, Michael Burrows und Roger Needham. *A Logic of Authentication: A Formal Semantics for Evaluating Cryptographic Protocols*, February 1990. ACM Transactions on Computer Systems, Band 8, Nr. 1, S. 18ff.

[20] Martin Abadi und Mark R. Tuttle. *A Semantics for a Logic of Authentication*, August 1991. Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, S. 201ff.

[21] Martin Haimerl, Jörg Moldenhauer, Michael Walz und Peer Wichmann. *Lifetime Dynamic Security Protocols*, 2000. interner Bericht, IAKS, Universität Karlsruhe.

[22] Monica Pawlan. *Reference Objects and Garbage Collection*, August 1998.
`http://developer.java.sun.com/developer/technicalArticles/ALT/RefObj/`.

[23] National Bureau of Standards. *Data Encryption Standard*, Januar 1977. Federal Information Processing Standards Pub. 46.

[24] National Institute of Standards and Technology. *ADVANCED ENCRYPTION STANDARD (AES) Fact Sheet*.
`http://csrc.nist.gov/encryption/aes/aesfact.html`.

[25] National Institute of Standards and Technology. *Security Requirements for Cryptographic Modules*, Januar 1994. Federal Information Processing Standard Pub. 140-1.

[26] National Institute of Standards and Technology. *Secure Hash Standard*, April 1995. Federal Information Processing Standard Pub. 180-1.

[27] National Institute of Standards and Technology. *Digital Signature Standard*, Januar 2000. Federal Information Processing Standard Pub. 186-2.

[28] Object Management Group. *The common object request broker: Architecture and specification Tech. Rep. Version 2.0*, Juli 1995.
`http://www.omg.org`.

[29] Object Management Group Corba.ch. *Was ist Corba?*
`http://www.corba.ch/WasIstCorba.html`.

[30] OMG Security Special Interest Group. *Security Service Specification*, Januar 1998.
`http://secsig.omg.org/`.

[31] Peer Wichmann. *Systematische Analyse von Kryptosystemen*, Februar 1998. Dissertation, Universität Karlsruhe.

[32] R. Housley, W. Ford, W. Polk und D. Solo. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*, Januar 1999. RFC 2459.

[33] Ron Rivest, Adi Shamir und Leonard Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, 1978. Communications of the ACM, Band 21, Nr. 2, S. 120ff.

[34] Shelina Gorain. *What's brewing in your browser? A review of Java security - past, present and future*, 1998.
`http://www.dstc.qut.edu.au/MSU/staff/shelina/brewing.html`.

[35] Sun Microsoystems. *What is the Java(TM) Platform.*
`http://www.javasoft.com/nav/whatis/?frontpage-shortcuts`.

[36] Sun Microsystems. *Chronology of security-related bugs and issues, 2/23/01.*
`http://java.sun.com/sfaq/chronology.html`.

[37] Sun Microsystems. *Java(TM) Cryptography Extension (JCE).* `http://java.sun.com/products/jce/index.html`.

[38] Sun Microsystems. *Java(TM) Cryptography Architecture - API Specification & Reference*, Dezember 1999. `http://java.sun.com/j2se/1.3/docs/guide/security/CryptoSpec.html`.

[39] Thomas Beth, Willi Geiselmann und Peer Wichmann. *Datensicherheitstechnik - Signale, Codes und Chiffren II*, Juli 2000. Universiät Karlsruhe, Vorlesungsskript.

[40] University of Mannheim, University of Tennessee. *Top 500 Supercomputer Sites.* `http://www.top500.org/`.

[41] Willi Geiselmann. *Public-Key Kryptographie*, Februar 2000. Institut für Algorithmen und Kognitive Systeme, Universität Karlsruhe, Vorlesungsskript.

Index