Sicherheitsdienste für Mobile-Agenten-Anwendungen in elektronischen Märkten

und Implementierung eines Prototyps in Java

Diplomarbeit

am
Institut für Algorithmen und Kognitive Systeme
Fakultät für Informatik
Universität Karlsruhe (TH)

und am
IBM Zürich Forschungslaboratorium
CH-8803 Rüschlikon, Schweiz

von

Jürgen Bohn

Betreuer: Prof. Dr. Th. Beth
Dr. W. Geiselmann

Ausgabe: September 1999
Abgabe: Februar 2000

Sicherheitsdienste für Mobile-Agenten-Anwendungen in elektronischen Märkten

und Implementierung eines Prototyps in Java

Diplomarbeit

am
Institut für Algorithmen und Kognitive Systeme
Fakultät für Informatik
Universität Karlsruhe (TH)

und am
IBM Zürich Forschungslaboratorium
CH-8803 Rüschlikon, Schweiz

von

Jürgen Bohn

Matr.-Nr. 0827165
jjbohn@topmail.de

Betreuer: Prof. Dr. Th. Beth
Dr. W. Geiselmann

Ausgabe: September 1999
Abgabe: Februar 2000

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

Mit der weltweit rasant zunehmenden Vernetzung von Rechnern und der Tendenz zu verteilten Systemen rückt auch ein neues Konzept in das Blickfeld der Informatik: das Mobile-Agenten-Paradigma. Während der Schutz verteilter Systeme und verteilter Kommunikation bereits seit längerer Zeit Gegenstand der Forschung gewesen ist und zahlreiche Lösungsansätze existieren, sind die Sicherheitsaspekte der mobilen Agenten heute weitgehend ungelöst. Obwohl sich die Verwendung mobiler Agenten in vielen Bereichen als sehr erfolgversprechend erweist, wird deren Einsatz aufgrund der Sicherheitsproblematik mit Skepsis betrachtet, und die Agententechnologie findet nur zögerlich Einzug in Gebiete, in denen Sicherheit eine kritische Rolle spielt, wie das beispielsweise im elektronischen Handel der Fall ist.

1.1 Ziel dieser Arbeit

In dieser Arbeit sollen nun die Sicherheitsbedürfnisse mobiler Agenten durchleuchtet und entsprechende Schutzmaßnahmen identifiziert werden. *Ziel* ist die Erstellung einer Bibliothek von konkreten Sicherheitsdiensten, die praktische Unterstützung bei der Entwicklung sicherer Agentensysteme gewähren, insbesondere im Hinblick auf die Realisierung elektronischer Märkte. Um die Machbarkeit der entwickelten Konzepte zu demonstrieren, sollen die Sicherheitsdienste außerdem exemplarisch implementiert und im Rahmen eines prototypischen Agentensystems eingesetzt werden.

1.2 Ergebnis

In dieser Arbeit wurde eine Palette von *generischen Sicherheitsdiensten* für mobile Agenten entworfen und ausgearbeitet. Die Analyse der Sicherheitsproblematik führte weiter zum *Entwurf eines sicheren Protokolls*, das zum Schutz der Agentendaten die Verwendung von sicherer Hardware auf unsicheren Plattformen vorschlägt. Als Leitbeispiel wurde durchgehend – stellvertretend für Anwendungen des elektronischen Handels – die preisvergleichende Produktrecherche gewählt. Dieses Szenario dient als Grundlage eines in der Programmiersprache Java implementierten *prototypischen Comparison Shopping Agentensystems.* Eine Auswahl der Sicherheitsdienste wurde ebenfalls in Java realisiert und unter Berücksichtigung des entworfenen sicheren Protokolls in den Prototypen

integriert. Die dabei benötigte Funktionalität der sicheren Hardware wurde durch die *Programmierung eines JavaCard-Applets* und dessen Installation auf einer SmartCard bereitgestellt. Damit konnte der Nutzen sowie die Durchführbarkeit der vorgestellten Ideen und Konzepte durch eine voll funktionsfähige prototypische Realisierung in Java erfolgreich untermauert werden.

1.3 Kapitelübersicht

Das **Kapitel 2** führt in die Thematik der mobilen Agenten ein. Zuerst werden die charakteristischen Eigenschaften und Merkmale der mobilen Agenten erläutert, die Vorzüge des Mobile-Agenten-Paradigma aufgezeigt und dessen Abgrenzung zu konkurrierenden Konzepten vorgenommen. Im Anschluss daran werden die Sicherheitsaspekte und Probleme mobiler Agentensysteme behandelt und potentielle Angriffe und Schutzmechanismen besprochen.

In **Kapitel 3** werden die Begriffe des elektronischen Handels und Marktplatzes erklärt, deren starke Affinität zur Agententechnologie motiviert und die dabei gestellten Anforderungen an agentenbasierte Anwendungen aufgeführt. Die besondere Eignung Mobiler Agentensysteme für Anwendungen im elektronischen Handel wird anhand der agentenbasierten preisvergleichenden Produktrecherche belegt, die im weiteren Teil der Arbeit als Leitbeispiel und Implementierungsgrundlage dient.

Kapitel 4 ist der Beschreibung der IBM Aglets Agentenplattform gewidmet, die bei der Implementierung des Prototypen verwendet wurde.

In **Kapitel 5** werden die elementaren Sicherheitsbedürfnisse mobiler Agenten identifiziert und strukturiert. Daraus werden anschließend systematisch die benötigten Sicherheitsdienste abgeleitet und entsprechende Datenstrukturen und Verfahren bestimmt.

Darauf aufbauend wird im **6. Kapitel** eine Weiterentwicklung des sicheren Protokolls von Karjoth et al. [KAG98] zum Schutz der Rechenergebnisse und Daten mobiler Agenten vorgestellt, das unter Verwendung von Trusted Hardware bisher ungelöste Sicherheitprobleme beseitigt und gleichzeitig neue Geschäftsmodelle und Anwendungsgebiete erschließt.

Im **7. Kapitel** werden die implementierten Sicherheitsdienste vorgestellt und Wege und Möglichkeiten zur Realisierung weiterführender, problemspezifischer Dienste aufgezeigt. Danach werden die Grundzüge der durchgeführten JavaCard-Programmierung beschrieben und die bereitgestellten Operationen aufgelistet.

Die Architektur und die Implementierung des Comparison-Shopping-Prototypen folgen in **Kapitel 8**.

Abschließend bietet das **Kapitel 9** eine Zusammenfassung der Arbeit und einen kurzen Ausblick.

Kapitel 2

Mobile Agenten

In diesem Kapitel werden die Begriffe *Agent*, *Software Agent* und *Mobiler Agent* erklärt sowie die Abgrenzung des letzteren gegenüber den „gewöhnlichen“ Agenten erläutert. Anschließend werden die Bedeutung und die Vorteile des Mobilen-Agenten-Paradigmas besprochen. Am Ende des Kapitels wird der Leser in die Sicherheitsproblematik der mobilen Agenten eingeführt.

2.1 Software Agenten

Der Begriff *Agent* hat bereits im alltäglichen Sprachgebrauch eine Vielzahl von Bedeutungen, vom Versicherungsvertreter bis zum Spion, und ebenso verhält es sich bei den sogenannten *Software Agenten* in der Informatik. Man findet in der Literatur [RH98, Vig98, VT97, Whi96] zahlreiche Ausführungen und Definitionen darüber, was einen programmierten Agenten, einen Software Agenten, auszeichnet, die in den wesentlichen Punkten mit den unten aufgeführten typischen Eigenschaften übereinstimmen. Aus der Sicht des Endbenutzers ist ein Software Agent ein Maschinenprogramm, das von diesem bestimmte Aufgaben übernimmt und diese in seinem Auftrag in gewissem Umfang selbständig und unabhängig erledigt. Der Endbenutzer wird in diesem Fall zum *Agentenbesitzer*. Eine genauere Beschreibung liefert die systemnähere Betrachtung eines Agenten als *Software-Objekt* mit folgenden Charakteristika:

- Ist eingebettet in eine *Ausführungsumgebung*.
- Besitzt folgende *notwendigen* Eigenschaften:
  - *Unabhängigkeit*: Plant und führt selbständig Aktionen durch (*Autonomie*).
  - *Zielorientierung*: Ist auf das Lösen einer bestimmten Aufgabe ausgerichtet.
  - *Intelligenz*: Verfügt über gewisse Wissensbasis, die seine Entscheidungsfindung beeinflusst.
  - *Reaktivität*: Nimmt Veränderungen in der Umgebung wahr und reagiert entsprechend darauf.
  - *Zeitliche Kontinuität*: Wird ununterbrochen ausgeführt.
- Kann zusätzlich eine Untermenge der folgenden *orthogonalen* Eigenschaften besitzen:

- *Kommunikationsfähigkeit*: Kommuniziert mit anderen Agenten oder Rechnern.
- *Mobilität*: Kann seinen Ausführungsort zu einem anderen Rechner verlegen (*aktiv* oder *passiv*).
- *Lernfähigkeit*: Passt sein Verhalten an zuvor gemachte Erfahrungen an (*Adaptivität*).
- *Glaubwürdigkeit*: Erscheint dem Endbenutzer oder der Ausführungsumgebung vertrauenswürdig.
- *Nichtabstreitbarkeit*: Getroffene Entscheidungen und vollzogene Aktivitäten sind bindend für den Agentenbesitzer und damit verbundene Konsequenzen sind nicht abstreitbar.

2.2 Das Mobile-Agenten-Paradigma

Im Zeitalter der Vernetzung und der verteilten Systeme hat sich die Klasse der *Mobilen Agenten* als eigenständiges Gebiet herauskristallisiert. Die Stärke des *Mobilen-Agenten-Paradigmas* liegt darin, dass es den Entwurf, die Implementierung und die Wartung verteilter Anwendungen wesentlich einfacher und intuitiver gestaltet. Wie sich bei der objekt-orientierten Programmierung das *Objekt* als elementare Einheit beim Entwurf und der Implementierung komplexer Aufgabenstellungen anbietet, so lassen sich in verteilten Systemen die autonomen, miteinander kooperierenden Parteien in natürlicher und verständlicher Weise als *mobile Agenten* modellieren.

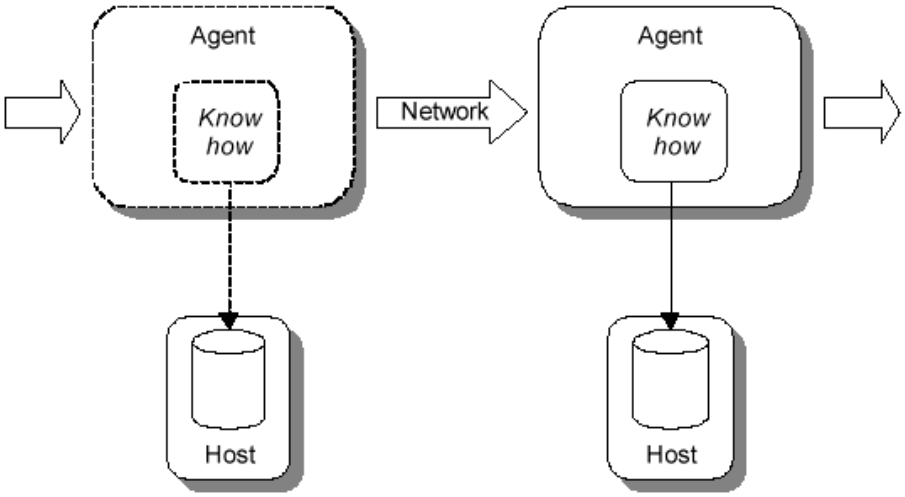


Abbildung 2.1: Das Mobile-Agenten-Paradigma.

Im Wesentlichen unterscheiden sich *mobile* Agenten von gewöhnlichen, *stationären* Agenten in der Hinsicht, dass sie nicht fest an das System gebunden sind, in dem sie mit ihrer Ausführung begonnen haben. Vielmehr erlaubt ihnen ihre Mobilität, zu anderen Systemen zu migrieren, um dort vor Ort eine zu lösende Aufgabe durch Interaktion mit anderen Objekten oder unter Benutzung lokaler Datenbestände effizienter und besser (bzgl. der Funktion) zu erledigen. Mit dem mobilen Agenten wandert das Wissen zur Erledigung der Aufgaben (Know-How) über das Netzwerk von Host zu Host, d.h. von Plattform zu Plattform, wie in Abbildung 2.1 dargestellt. Mobile Agenten lassen sich auch in Hinblick auf den Grad ihrer Mobilität unterscheiden: Man spricht von *starker Mobilität*, wenn der Transfer von Agenten sowohl den auszuführenden Programmcode, die Daten als auch den

zuletzt gültigen Ausführungszustand (engl. *execution state*) beinhaltet [Ple99]. Dies ist zum Beispiel bei Java nur bedingt möglich, da die Java Virtual Machine (JVM) keinen Zugriff auf den Ausführungsstack (Befehlsstack) erlaubt. Werden nur der Programmcode und die Daten des Agenten übertragen, so bezeichnet man das als *schwache Mobiliät*. Soll der aktuelle Ausführungszustand zumindest teilweise erhalten bleiben, muß dieser zuvor explizit in Zustandsvariablen gesichert und nach der Migration ensprechend manuell wiederhergestellt werden.

Warum Mobile Agenten? Es gibt zum jetzigen Zeitpunkt keine Anwendung, für die der ausschießliche Einsatz von Mobilen Agenten als zwingend erscheint. Zahlreiche Problemstellungen in der Praxis lassen sich beispielsweise durch das Client/Server-Modell unter Verwendung von Remote Procedure Calls (RPC) und durch asynchronen Nachrichtenaustausch (*Messaging*) realisieren.

Dennoch bieten Mobile Agenten einen *entscheidenden Vorteil*: Während für jeden Einzelfall herkömmliche Lösungsansätze ohne Gebrauch mobiler Agenten existieren, so stellt das Mobile-Agenten-Paradigma ein Rahmenwerk zu Verfügung, welches alle diese Fälle gleichzeitig abdeckt und behandelt [CHK97]. Das heißt, die eigentliche Stärke des Mobile-Agenten-Paradigmas liegt nicht in der Eignung für ein spezielles Teilgebiet, sondern in der Integration all der Vorteile der verschiedenen Einsatzmöglichkeiten innerhalb eines einzigen Ansatzes. Was diese kumulierte Stärke ausmacht, wird im nachfolgenden Abschnitt erläutert.

2.2.1 Vorteile Mobiler Agenten

Die folgenden Punkte spiegeln die Vorteile mobiler Agenten wider [CHK97, LO98]:

1. **Reduktion der Netzlast** durch Zusammenführung von Agent und Daten bzw. Interaktionspartner.
2. **Überwindung der Netzlatenz** durch *Ausführung vor Ort* unter Umgehung von üblichen Verzögerungen bei Überlastsituationen oder „langsamen“ Verbindungen.
   - **Verbesserte Client/Server-Echtzeitanwendungen**: Zeitliche Verzögerungen werden minimiert durch Reduktion des Kommunikationsaufwandes zwischen Client und Server.
3. **Asynchronität und Autonomie**: Der Agent wird abgeschickt, arbeitet unabhänging (autonom) von der Verfügbarkeit des Ursprungsrechners und kehrt zu gegebenem Zeitpunkt nach Beendigung der Tätigkeit zurück.
4. **Intelligentere Suchanfragen und Informationsbeschaffung**:
   - Es sind mittels programmierter Agenten beliebig komplexe und benutzerspezifische Suchanfragen realisierbar, die *vor Ort* Datenbestände durchforsten, indizieren oder zusammenfassen.
   - Suchanfragen können außerdem mit einem Gedächtnis (Zustandsspeicher) versehen werden und benutzerabhängige Kontextinformationen bzw. Wissen vorhergehender Operationen speichern und nutzen.
   - Es lassen sich auch andere Aktionen definieren, die im Anschluss an eine Suche die erhaltenen Resultate weiterverarbeiten oder andere Tätigkeiten (bzw. Agenten) anstoßen.

5. **Protokollwandlung und -einkapselung**: Mobile Agenten können bei Bedarf zwischen inkompatiblen Rechnern bzw. Protokollen vermitteln oder situationsbedingt Kommunikationskanäle öffnen, die auf proprietären Protokollen basieren. Dadurch erspart man sich die aufwendige Anpassung bereits vorhandener bzw. die Neuinstallation zusätzlicher (zeitweise) benötigter Protokolle.

6. **Robustheit und Fehlertoleranz**: Dienste können durch redundanten Einsatz (bezüglich Anzahl und Funktion) unabhängig agierender und untereinander kommunizierender mobiler Agenten fehlertolerant realisiert werden. Dadurch wird insgesamt die Robustheit des übergeordneten Gesamtsystems gesteigert, und die Anfälligkeit für Ausfälle bei punktuellen Angriffen (engl. *single points of failure*) reduziert.

7. **Natürliche Modellierung nebenläufiger Prozesse**: Der Agent bietet sich als elementare Einheit bei der Modellierung nebenläufiger Prozesse und Systeme an.

8. **Unterstützt heterogene Umgebungen**: Ein Agentensystem ist unabhängig von der unterliegenden Hardware, der Transportschicht und idealerweise auch vom Betriebssystem (z.B. durch Java-basierte Ausführungsplattformen) und somit hervorragend geeignet für heterogene Netze und Systeme.

9. **Besondere Eignung für E-Commerce Anwendungen**:

   (a) *Benutzerfreundliche Schnittstellen* und Dienste für Endbenutzer: Schnittstellen zu Server-Anwendungen können dynamisch verändert oder aktualisiert werden bzw. für verschiedene Benutzergruppen (beispielsweise Experten, Anfänger) maßgeschneidert werden.

   (b) *Autorisierung und Authentifikation*: Unterschiedliche Zugriffsrechte, verschiedene Sichten auf Daten oder Variationen im Funktionsangebot eines Agenten für bestimmte Endbenutzer können durch das Agentensystem feinkörnig und verteilt modelliert werden: Zuteilung von entsprechend konfigurierten Agenten, Rechte werden vom Agenten mitgeführt und müssen bei Bedarf nicht von zentraler Stelle eingeholt werden (dezentraler Ansatz, fehlertolerant, vgl. Punkt 6).

   (c) *Plattformübergreifende Lösungen* (vgl. Punkt 8).

   (d) *Dynamische Integration von bereits vorhandenen proprietären Systemen (engl.* legacy systems*)* (vgl. Punkt 5).

   (e) *Robustere und leistungsfähigere Transaktionen*:

   - Der Prozessstatus wird nur im Agenten mitgeführt, und nicht im Client oder Server gespeichert, und dort laufend aktualisiert.
   - *Bessere Skalierbarkeit* als bei RPC-basierte Transaktionen dank asynchronem Verhalten.
   - *Geringerer Overhead bei sicheren Transaktionen,* da eine einmalige sichere Übertragung des Agenten ausreicht statt ggf. eine Vielzahl von zu sichernden Transaktionen durchzuführen.

   (f) *Grad der Benutzerinteraktion ist beliebig wählbar* bis hin zu vollkommen autonomen, asynchron agierenden Agenten (vgl. Punkt 3).

2.2.2 Abgrenzung zu etablierten Paradigmen

Wie die aufgezeigten Vorteile mobiler Agenten belegen, tritt das Mobile-Agenten-Paradigma aus guten Gründen zunehmend in Konkurrenz zu den bereits etablierten Paradigmen verteilter (Netzwerk-)Systeme, dem Client/Server-Modell und dem *Code-On-Demand*-Paradigma, die hier kurz vorgestellt und vom Ansatz der mobilen Agenten abgegrenzt werden.

Client/Server-Paradigma. Heutzutage hat sich im Zusammenhang mit der gemeinsamen Nutzung von Resourcen und dem Einsatz verteilter Rechnersysteme in Netzwerken in großem Maße das *Client/Server*-Modell durchgesetzt. Als grundlegende Elemente dienen Server-Prozesse, die *Resourcen* verwalten, und Client-Prozesse, die Zugriff auf geteilte Resourcen benötigen [RR96]. Client/Server-Beziehungen sind geprägt durch Client-Prozesse, die sich Ressourcen durch Senden von *Aufträgen* an den ensprechenden Server teilen, der diese Aufträge an Stelle der Clienten abarbeitet und ggf. eine *Antwort* (Ergebnis) zurückschickt. Der Server verwaltet sozusagen das *Wissen* und ist zuständig für die Erbringung von *Diensten*; der Client hingegen benötigt nur geringe Rechenleistung und ausreichend *Intelligenz,* um den passenden Dienst auszuwählen und anzusprechen. Das *Client/Server*-Paradigma ist heute in verteilten Systemen am weitesten verbreitet; die dabei verwendeten Techniken sind beispielsweise CORBA (Common Object Request Broker Architecture), der entfernte Prozeduraufruf RPC (Remote Procedure Call) und der entfernte Methodenaufruf RMI (Remote Method Invocation) in Java.

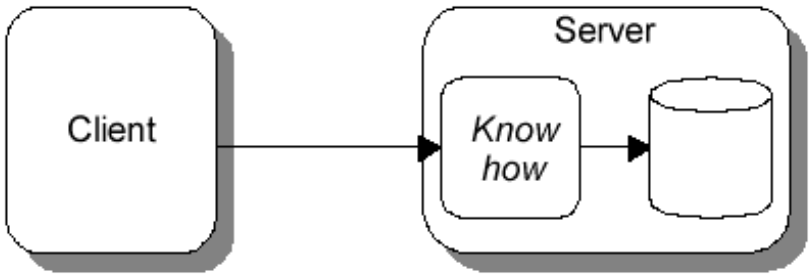


Abbildung 2.2: Grundprinzip des Client/Server-Konzepts.

Im Gegensatz zum Mobile-Agenten-Paradigma ist das Client/Server-Modell sehr stark *transaktionsorientiert,* und der Schwerpunkt liegt auf der Übertragung von Daten und entfernten Aufrufen, wobei das Dienstangebot auf ein feste Menge von Dienstprimitiven festgelegt ist. Um zur Laufzeit dynamisch Dienste anpassen zu können, müssen verwaltungsintensive Verzeichnisdienste oder noch mächtigere Konzepte wie z.B. der *Trader* realisiert werden [Kel93]. Außerdem haben RPC-basierte Systeme einen ausgeprägt synchronen Charakter. Um den Grad von Asynchronität zu erhöhen, ist zusätzlich eine komplexe Mittelschicht (*middleware*) vonnöten, die z.B. mittels persistenten Warteschlangen einen asynchronen Nachrichtenaustausch erlaubt. Abbildung 2.3 stellt das RPC-Konzept dem Mobilen-Agenten-Ansatz gegenüber.

Code-On-Demand-Paradigma. Mit dem zunehmenden Erfolg des World Wide Webs (WWW) als weltumspannendem Informationssystem im Internet hat auch das *Code-On-Demand*-Paradigma an Bedeutung gewonnen: Der Maschinencode wird nicht fest installiert, sondern bei Bedarf entweder durch den Web-Browser heruntergeladen und lokal ausgeführt (Java *Applets*, siehe Abbildung 2.4) oder an den entfernten Server gesendet und dort gestartet (*Servlets*).

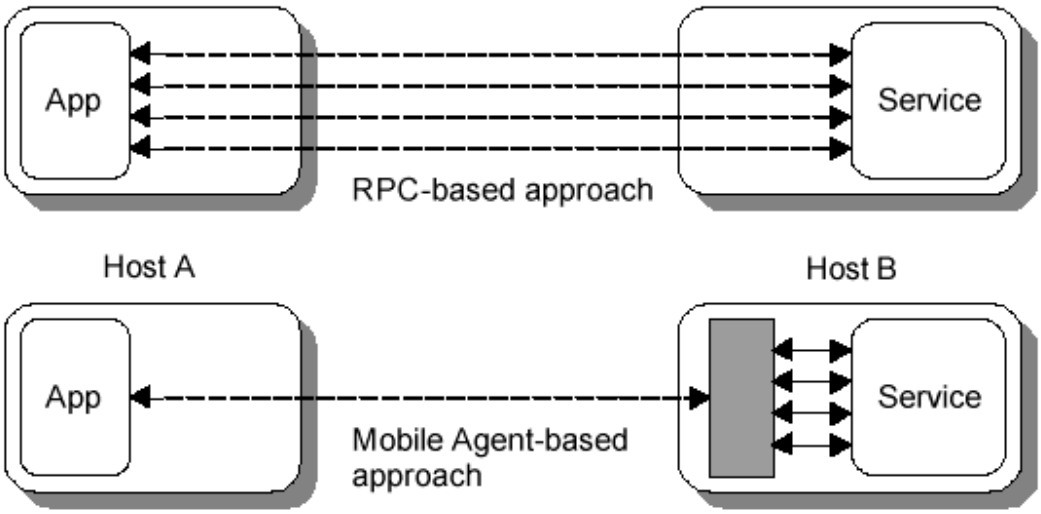


Abbildung 2.3: Vergleich des RPC-Konzeptes mit dem mobilen Agenten Ansatz.

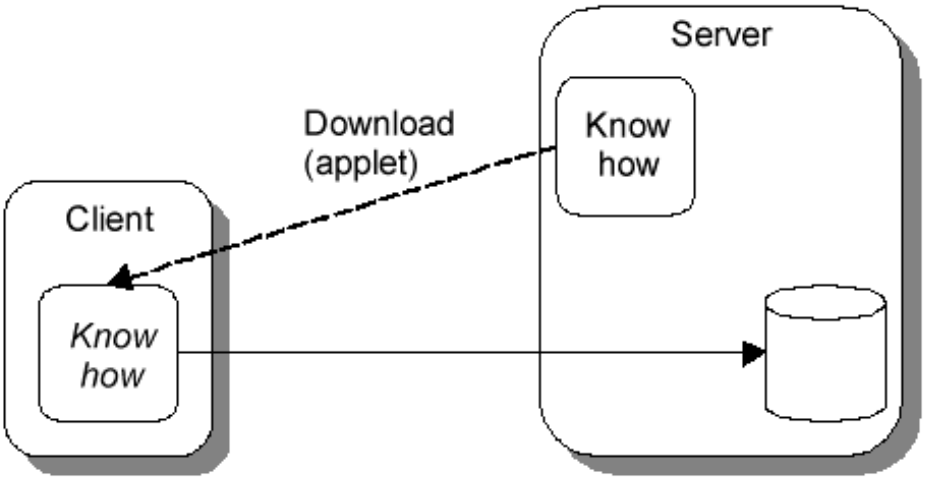


Abbildung 2.4: Applets als Beispiel für das Code-On-Demand-Paradigma.

Das heißt, in beiden Fällen ist nur ein unidirektionaler Programmfluss gegeben, und die Partnerbeziehung ist stets auf eine eins-zu-eins Beziehung zwischen (Web-)Servern und Clienten (Web-Browser) beschränkt.

2.2.3 Einsatzgebiete

White hat bereits 1996 in [Whi96] verschiedene Einsatz-Szenarien mobiler Agenten aufgezeichnet: Mobile Agenten als persönliche Terminassistenten, die (1) Termine verwalten und abmachen, (2) nützliche Informationen beschaffen (wie z.B. Fernsehprogramme, Verkehrsaufkommen, Aktienkurse), (3) im Namen des Besitzers bestimmte Tätigkeiten übernehmen (wie beispielsweise Theaterkarten bestellen, Überweisungen tätigen, Aktien unter gewissen Umständen kaufen oder verkaufen usw.). Weitere Anwendungsgebiete sind auch

- Fortgeschrittene *Informationsrecherche* [RGK97],
- *Produktrecherche*: (Preis-)Vergleich ähnlicher Produkte verschiedener Anbieter (engl. *comparison shopping*),
- verteilte agentenbasierte *Routingalgorithmen* [DD98],
- *Störungsmanagement*: Fehlertolerante Verfahren zur Erkennung von gezielt herbeigeführten Störungen und unerlaubtem Eindringen bei elektronischen Systemen (*intrusion detection*),
- *Workflow Anwendungen*, in denen z.B. verschiedene Tätigkeiten aufeinanderfolgender Aktivitäten durch enstprechende Agenten bereitgestellt und die dabei zu manipulierenden Daten stets mitgeführt werden.

2.3 Sicherheitsaspekte Mobiler Agentensysteme

Die Sicherheit von Agentensystemen ist – mit Blickpunkt auf den Schutz der Rechnerplattform und der Kommunikation – in zahlreichen Veröffentlichungen bereits ausgiebig diskutiert worden, siehe zum Beispiel [BGS98, Che98, HLPS98, MEJ99]. Dabei stehen Maßnahmen zur Resourcenverwaltung und -nutzung sowie zur Zugangskontrolle (engl. *access control*) im Vordergrund. Das Hauptaugenmerk dieser Arbeit richtet sich jedoch auf das weitgehend ungelöste und daher besonders kritische Problem von Sicherheit und Integrität auf Seiten der mobilen Agenten selbst. Eine grobe Klassifizierung der bekannten Schutzmechanismen liefert Tabelle 2.1.

2.3.1 Grundsätzliche Sicherheitsbedürfnisse Mobiler Agenten

Grundsätzlich sind folgende Sicherheitsdienste zum Schutz von Agenten erforderlich oder wünschenswert:

- *Authentifikation*
  - der Benutzer des Agentensystems (Wer hat Zugriff?),
  - der beteiligten Rechner (Agentenplattformen) untereinander (Ist mein Gegenüber wirklich der, für den ich ihn halte?),

  - des Programmcodes (Vertrauenswürdige Implementierung?) und
  - der Agenten selbst (Kann ein verantwortlicher Besitzer zugeordnet werden?).
- *Integritätskontrolle:* Wurde der Programmcode des Agenten oder seine Daten manipuliert?
- *Geheimhaltung:* Wie kann ich meine Daten vor dem Zugriff Dritter schützen?
- *Autorisierung:* Welche Ausführungsrechte werden dem Agenten zugebilligt?
- *Unabstreitbarkeit* von Nachrichten und Aktionen. Wer ist der Urheber?
- *Buchführung* über sicherheitsrelevante Vorkommnisse (engl. *auditing*) zur späteren Auswertung: Wer hat wann welche Aktion ausgeführt?
- *Anonymität* und Schutz der *Privatsphäre*: Können Aktivitäten einem bestimmten Teilnehmer ohne dessen Wissen oder ausdrückliche Zustimmung zugeordnet in einem Benutzerprofil erfasst werden?

2.3.2 Angriffe gegen Mobile Agenten

Sobald mobile Agenten jedoch ihren (hier als sicher betrachteten) Ursprungsrechner verlassen, sind sie Ziel potentieller Angriffe. Darunter fallen:

- **Ausspionieren** oder **Manipulation** von
  - Code,
  - Daten,
  - Kontrollfluss,
  - Kommunikation zwischen Agenten,
  - Interaktion mit anderen Agenten.
- **Abfangen**: Das Abfangen von Agenten bei deren Migration zur nächsten Rechnerplattform bzw. *Überwachen des Migrationsverhaltens,* um Rückschlüsse auf Art und Umfang der Agentenaktivitäten und somit indirekt auf das übergeordnete Benutzerverhalten zu erhalten.
- **Maskierung** (engl. **masquerading**):
  Durch Maskierung der eigenen Identität versucht ein Rechner (oder eine andere Partei), dem Agenten gegenüber eine andere Identität vorzuspiegeln, um den Agenten zu bestimmten Handlungen zu verleiten oder geheime Daten zu erschleichen.
- **Dienstverweigerung** (engl. **denial of service**):
  Die Ausführung des Agenten kann (zum eigenen Vorteil) gezielt verweigert oder verzögert werden. Fehlerhafte oder korrumpierte Agenten können ebenfalls durch störende Interaktion andere Agenten blockieren.
- **Fehlerhafte Programmausführung**:
  Statt den Ablauf im Agentenprogramm zu ändern wird der Kontrollfluss zum Zeitpunkt der Ausführung des Programmcodes auf dem Host direkt manipuliert, um das Programmverhalten gezielt zu beeinflussen, z.B. durch das Verändern von Speicher- oder Registerwerten.

- **Verfälschte Systemaufrufe**:
  Systemaufrufe werden gezielt maskiert oder manipuliert, um dem Agenten falsche Tatsachen vorzuspiegeln (z.B. falsche Rechneradresse) und damit seine korrekte Ausführung zu kompromittieren.

Da sich der Agent bei seiner Ausführung gewissermaßen dem gastgebenden Rechner anvertrauen muss, liegen dort auch die meisten potentiellen Gefahrenstellen und Angriffsmöglichkeiten.

2.3.3 Schutzmaßnahmen

Jede Plattform stellt für sich eine *Sicherheitspolitik* (engl. *security policy*) auf, die z.B. die beteiligten Entitäten bestimmt und deren Rechte und Pflichten definiert. Eine Vielzahl von Sicherheitsmechanismen beugen dem Missbrauch von Resourcen und unlauterem Verhalten von Seiten der Agenten vor, siehe dazu [TK98]. Eine umfangreiche Analyse von Maßnahmen zum Schutz von Agentenplattformen liefern Jansen und Karrygiannis in [JK99] sowie Pleisch in [Ple99], wie beispielsweise *signierter Code*, *beweismitführender Programmcode* oder das Sandkastenmodell (engl. *sandbox model*). Die Probleme des *Schutzes der Migration* (Übertragung des Agenten von einer Plattform zur nächsten) und der Kommunikation bei mobilen Agenten sind weitestgehend gelöst. *Sichere Kommunikation* kann mittels bekannter Verfahren zur Authentifikation und mittels Verwendung sicherer Kanäle bewerkstelligt [BGS98, BM95, Nee93], die *sichere Übertragung* von Agenten durch entsprechende sichere Protokolle erreicht werden, entweder direkt durch das zuständige Transferprotokoll, wie z.B. das *Agent Transfer Protocol* in [Kar98], oder über sichere Punkt-zu-Punkt-Verbindungen mittels der Etablierung kryptographisch geschützter Kanäle.

Schutz der Mobilen Agenten. Eine potentielle Gefahrenquelle für mobile Agenten stellen andere Agenten dar, die in derselben Agenten-Umgebung ausgeführt werden. Um eine sichere Ausführung von Agenten zu gewährleisten, werden diese entweder in *isolierten Adressräumen* ausgeführt, was auch als *isolierte Ausführung* bezeichnet wird. Oder die Kapselung und der Schutz privater Daten wird durch Einhaltung einer strengen Typisierung (engl. *strong typing*), wie beispielsweise in der Programmiersprache Java geschehen, erzwungen. *Proxy-basierte Zugangsmechanismen* können außerdem einen direkten Zugriff auf den Agenten erschweren und Manipulationen am Agenten-Objekt selbst verhindern.

Die größte zu bewältigende Gefahr für mobile Agenten bleibt das Auftreten bösartiger Rechnerplattformen, von denen die Mehrzahl der Angriffe aus Abschnitt 2.3.2 ausgehen. Das Einschleusen von fehlerhaften oder bösartigen Klassen wird durch die strikte Zuweisung von genau einem *Klassen-Lader* (engl. *class loader*) je Agent verhindert. Hierbei ist jedoch zu berücksichtigen, dass auf unsicheren Plattformen dieser Mechanismus selbst Gegenstand von Manipulationsversuchen sein kann, und dessen Zuverlässigkeit daher nicht immer gewährleistet ist. Manipulationen am Agenten selbst sind kaum zu verhindern. Mit Hilfe von *Signaturen* kann zumindest die Integrität der zum Zeitpunkt der Initialisierung festgeschriebenen unveränderlichen Komponenten eines Agenten vor dessen Ausführung überprüft werden. Zur *Authentisierung* werden digitale Signaturen eingesetzt, die in der Regel auf asymmetrischen Schlüsselpaaren (öffentlicher und privater Schlüssel) basieren, z.B. mittels dem El Gamal- oder dem RSA-Signaturverfahren nach Rivest, Shamir und Adleman [ElG85, RSA78]. Dies setzt jedoch auch das Vorhandensein einer entsprechenden *Public-Key Infrastruktur* (PKI) voraus. Damit geheime Daten des Agenten nicht von unbefugten Dritten eingesehen

werden können, finden die bekannten kryptographischen Verfahren zur *Verschlüsselung* Anwendung. Je nach Eignung und Bedarf werden die effizienteren symmetrischen Verschlüsselungsalgorithmen (z.B. Data Encryption Standard DES [NIS93]) oder die im Funktionsumfang mächtigeren asymmetrischen Public-Key Verfahren (z.B. RSA) verwendet. Um aber auch das unbefugte selektive Zerstören oder Ersetzen von Datenfragmenten durch Dritte zu verhindern, sind umfassendere Schutzmaßnahmen vonnöten. Die derzeit wichtigsten Ansätze dabei werden in Tabelle 2.1 klassifiziert und im Folgenden erläutert.

Kategorie	Verfahren	Prakt. Eignung	Literatur
Verbergen der Funktionsweise	Verschlüsselte Funktionen	(noch) ungeeignet	[ST98a, ST98b, ST98c]
Dummy-Objekte	Applikationsspezifische Detection Objects	bedingt	[Mea97]
Fehlertoleranz	Replikation von Servern und Agenten, Mehrheitsentscheid	bedingt	[Ple99, Sch97]
Sofware-Blackbox	Umordnungsalgorithmus und Gültigkeitsinterval	bedingt	[Hoh98]
Sichere Protokolle	Public-Key-Infrastruktur und Hash-Ketten	gut	[KAG98]
Hardware-Blackbox	Trusted Hardware: Smart-Cards, Spezialhardware	gut	[Fün99]

Tabelle 2.1: Bekannte Ansätze zum Schutz mobiler Agenten gegen Spionage und Daten-Manipulation in unsicheren Umgebungen.

Verschlüsselte Funktionen. In diese Richtung zielen die noch überwiegend theoretischen Ausführungen von Sander und Tschudin [ST98b, ST98c]. Mittels *verschlüsselter Funktionen* (*Encrypted Functions, Function Hiding*) soll die korrekte Ausführung von Agenten auf unsicheren, evtl. bösartigen Plattformen ohne die Verwendung von Spezialhardware ermöglicht werden. Die entwickelten Ansätze sind jedoch zur Zeit noch auf Polynome und rationale Funktionen beschränkt und scheinen nur in sehr begrenztem Maße geeignet zu sein, die gestellten Aufgaben in punkto Sicherheit zu lösen.

Detection Objects. Meadows schlägt die Verwendung von sogenannten *Detection Objects*, Erkennungsobjekten, vor [Mea97]; das sind anwendungsspezifische Dummy-Objekte, deren alleinige Funktion darin liegt, unerlaubte Manipulationsversuche an den Daten (Objekten) mobiler Agenten zu erkennen. Die Grundidee ist, dass sowohl der Agent als auch die ausführende Plattform nicht wissen, welche Datenobjekte die Erkennungsobjekte darstellen. Im Falle eines Angriffes werden mit sehr hoher Wahrscheinlichkeit auch die Dummy-Objekte modifiziert und diese Modifikationen später durch Objektvergleich festgestellt. Voraussetzung ist jedoch, dass die *Detection Objects* nicht von den eigentlichen Objekten unterschieden werden können, auch nicht bei mehreren Durchläufen, und dass sie das Verhalten und die Entscheidungsfindung von Kooperationspartnern auf fremden Plattformen nicht beeinflussen.

Fehlertoleranz. Andere Überlegungen folgen dem Ansatz der *Fehlertoleranz*. Sie setzen dabei einerseits auf eine redundante Ausführung durch den gleichzeitigen Einsatz replizierter Agenten,

die sich nach jeder Bearbeitungsstufe durch *Abstimmung* auf ein gemeinsames gültiges Ergebnis einigen. Die Grundprinzipien sind also *Replikation* von Agenten und Mehrheitsentscheid (*Voting*) [Sch97]. Die Nebenbedingungen dabei sind jedoch u.a. die Verfügbarkeit von Replikationen des zu besuchenden Servers; ebenso ist deren unlautere Kooperation nicht auszuschliessen, so dass erhaltene Resultate nicht zwingendermaßen verlässlich sind. Die geforderte Redundanz an (identischen) Servern ist zudem teuer und in der Praxis oft nicht gegeben.
Pleisch untersucht in [Ple99] auch die Steigerung der allgemeinen Zuverlässigkeit von Mobile-Agenten-Systemen durch deren fehlertolerante Auslegung. In [RB94] wird die sichere Replikation von Diensten besprochen, auch mit Bezug zur Schwellenwert-Kryptographie (engl. *threshold cryptography*), bei der Geheimnisse oder Funktionen von einer Gruppe von Benutzern geteilt werden [DDFY94, Des94, Sha79].

Software Blackbox. Hohl hat ein Verfahren entwickelt, das mobile Agenten für eingeschränkte Zeiträume zur *sicheren software-basierten Blackbox* werden lässt [Hoh98]. Es beruht auf Algorithmen, die durch geschickte Zerlegung und Rekomposition von Code-Fragmenten des Agenten bestimmte Vorgänge verschleiern (engl. *mess-up algorithms*). Dadurch wird angestrebt, dass zumindest innerhalb eines festgelegten Zeitraumes die Analyse und sinnvolle Manipulation des Agenten verhindert wird, und dass geheime Daten im zeitlichen Gewährleistungsintervall nicht unerlaubt für anderweitige Zwecke missbraucht werden können. Gegenstand solcher Verschleierungsaktionen sind Variablen, Schlüssel und der Kontrollfluss selbst. *Schwachstellen* sind hierbei jedoch, dass die Güte und Effektivität der verwendeten Verschleierungs-Algorithmen nur schwer ermittelt werden kann, und dass die Wahl des zeitlichen Gültigkeitsintervalls – die insbesondere im Hinblick auf die asynchrone, zeitlich entkoppelte Arbeitsweise mobiler Agenten – unter Umständen erhebliche Probleme bereiten kann.

Sichere Protokolle. Ein protokollbasiertes Verfahren zum Schutz sich frei bewegender Agenten wurde von Yee [Yee97] vorgestellt und durch Karjoth et al. verbessert [KAG98]. Eine Ausprägung des Algorithmus basiert auf dem Vorhandensein einer Public-Key-Infrastruktur (PKI) mit eindeutig zuweisbaren digitalen Unterschriften für jeden beteiligten Server (engl. *per-server digital signatures*). Eine Alternative dazu ist ein Mechanismus, der ohne eine solche PKI auskommt und stattdessen die zu schützenden Daten auf Seiten des Agenten mittels *Hash-Ketten* sichert. Das Protokoll erlaubt, die Integrität des Agenten auch unterwegs auf unsicheren Rechnern zu überprüfen, seine Daten gegen Manipulation zu schützen und sie geheimzuhalten. Mit Hilfe einer Public-Key-Infrastruktur wird außerdem die Nichtabstreitbarkeit der Herkunft von gesammelten Daten realisiert. Damit lässt sich der Zustand eines Agenten samt Datenbestand bis zum Besuch des ersten bösartigen Rechners schützen. Anschließend sind jedoch z.B. Replay-Angriffe möglich, in deren Verlauf der Agent durch eine ältere, zuvor gespeicherte Kopie ersetzt wird, so dass die dazwischen gesammelten Daten unbemerkt entfernt werden können.

Vertrauenswürdige, sichere Hardware. Der zur Zeit einzige Weg, die Ausführung mobiler Agenten auf unsicheren, möglicherweise bösartigen Rechnerplattformen gegen Manipulation und Ausspionieren zu schützen, ist die Verwendung von sicheren, vertrauenswürdigen Geräten (*Trusted Hardware*). Diese Geräte erscheinen bezüglich Ihrer Funktionsweise nach außen hin als ein schwarzer Kasten (*Blackbox*). Kritische Operationen des Agenten können nun so ausgelegt werden, dass sie nur innerhalb der speziellen sicheren Geräte ausführbar sind, und sich somit dem Einflussbereich des zugehörigen Rechners entziehen. Die Bandbreite derartig vertrauenswürdiger, sicherer Geräte

reicht von nur kreditkartengroßen SmartCards über komplette geschützte Rechner bis hin zur teuren Spezialhardware.

Weitere Literatur zu sicherer Hardware: Fünfrocken beschreibt in [Fün99] die Verwendung von SmartCards zum Schutz mobiler Agenten, die in die World Wide Web (WWW) Umgebung integriert werden. Anderson and Kuhn diskutieren ausführlich verschiedene Aspekte der Fälschungssicherheit von sicheren Geräten [AK96]. Allgemeiner behandelt Chess in [Che98] ausführlich die Sicherheitsaspekte von Systemen, die auf mobilem Programmcode basieren.

Kapitel 3

Elektronische Märkte und Agenten

Im vorliegenden Kapitel werden die Möglichkeiten und der Nutzen von mobilen Agentensystemen zur Realisierung elektronischer Märkte und zur Unterstützung von elektronischem Handel untersucht und bereits existierende Ansätze kurz vorgestellt. Anschliessend wird ein konkretes agentenbasiertes Protokoll zur *preisvergleichenden Produktsuche* und Einholung bindender Angebote (engl. *comparison shopping*) auf *elektronischen Marktplätzen* eingeführt, das den praktischen Einsatz mobiler Agenten illustriert.

3.1 Elektronischer Handel

Elektronischer Handel (engl. *electronic commerce* , kurz *e-commerce*) kann kurz und bündig als ein *Austausch von Waren, Diensten oder Besitzgütern jeglicher Art über ein elektronisches Medium* definiert werden. Der Grundstein dafür wurde schon vor Jahrzehnten gelegt: So wurde beispielsweise mit dem Electronic Data Interchange-Standard (EDI -Standard) die Grundlage für den elektronischen Datenaustausch [TZ93] zwischen kooperierenden Unternehmen und Organisationen geschaffen. In Europa sind auch seit Jahren elektronische Onlinesysteme erfolgreich im Einsatz, wie z.B. Minitel in Frankreich oder Bildschirmtext (BTX) in Deutschland, wenn auch mit abnehmender Relevanz.

Der Quantensprung im elektronischen Handel erfolgte mit dem Wandel des Internets von einem nationalen (amerikanischen) Forschungsnetz zu dem weltumspannenden Netzwerk schlechthin. Mit der Öffnung für Geschäftskunden und Privatpersonen in der ganzen Welt wurde ein neues kommerzielles Massenmedium geschaffen.

Die damit verbundene *wirtschaftliche Bedeutung* ist immens: Laut Andersen Consulting [CEF+98] bezifferte sich 1998 der weltweite Umsatz von elektronischem Handel im Internet auf über 10 Milliarden US-Dollar. Für das Jahr 2002 wird ein Anstieg auf über 350 Milliarden US-Dollar prognostiziert. Der Wandel der Wirtschaft insgesamt zu einer elektronischen Wirtschaft (engl. *electronic economy* , kurz *e-economy*) wird angekündigt, in die sich der elektronische Handel eingliedere und deren zentrales Charakteristikum das Vorhandensein von Firmen aller Größenordnungen mit elektronischer Ausrichtung sei, wie in Abbildung 3.1 illustriert. Eine solche Entwicklung soll in den Vereinigten Staaten zum Beispiel bereits im Jahre 2003 vollzogen sein.

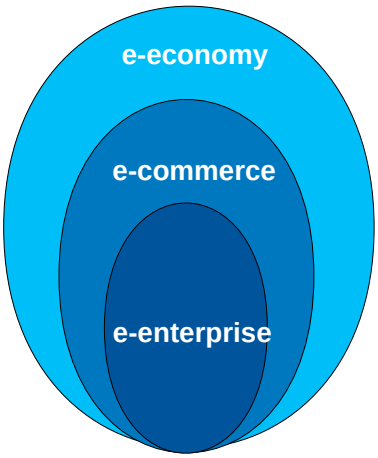


Abbildung 3.1: Integrale Bestandteile einer elektronischen Wirtschaft (*e-economy*)

3.2 Elektronische Handelssysteme und Märkte im Internet

Mit weltweit bereits mehr als 100 Millionen angeschlossenen Benutzern und einer steigender Tendenz [CEF+98] ist das Internet drauf und dran, sich zu einem riesigen, grenzenlosen Zukunftsmarkt zu entwickeln, zu einem allumspannenden Einkaufs- und Dienstleistungsmarkt. Nach der Einführung der ersten marktreifen, kreditkarten-basierten Zahlungssysteme im Internet in den 90er Jahren in den U.S.A. gibt es heute bereits eine Vielzahl elektronischer Handelssysteme auch außerhalb Nordamerikas. Elektronische Zahlungssysteme wurden u.a. bereits in Deutschland und der Schweiz weiterentwickelt und im Rahmen von Pilotprojekten erprobt [Boh98]. Auch im Börsengeschäft werden bereits elektronische Systeme erprobt, die in Echtzeit Kundenaktionen wie das Überwachen von Kursverläufen oder Kaufen/Verkaufen von Aktien ermöglichen.

Die Vorteile und Geschäftsgelegenheiten des elektronischen Handels haben zu einer weiten Verbreitung und steigenden Akzeptanz – sowohl bei Geschäftsführern als auch bei den privaten Internetbenutzern – geführt. Die Notwendigkeit, in diesem Sektor verstärkt aktiv zu werden, würde nach Andersen Consulting in Europa von der Mehrheit der Unternehmer erkannt, wenn auch die tatsächliche Umsetzung noch zögerlich voranschreite und generell eher eine abwartende Haltung vorherrschend sei.

Wichtige Faktoren für elektronische Handelssysteme sind zum einen gesellschaftliche oder soziale Aspekte (Akzeptanz, Vertrauen) und wirtschaftliche Aspekte (Durchführbarkeit, Wirtschaftlichkeit, Eignung neuer Geschäftsmodelle) andererseits.
Auf Seiten der Informatik steht die Frage nach der *Sicherheit* und *Vertraulichkeit* solcher Systeme im Vordergrund; die Bereiche Kommunikation, Netzwerke und verteilte Systeme sind schon seit Jahrzehnten Gegenstand der Forschung und dementsprechend besser durchdrungen. Zwar ist das Gebiet der klassischen Kryptographie bereits intensiv untersucht worden, z.B. in Bezug auf Verschlüsselungs-, Signatur oder Hashverfahren. Jedoch eröffnen sich mit der Technlologie der mobilen Agenten auch neue, bislang unbekannte Problemfelder, denen man sich erst in den letzten Jahren ausgiebiger zu widmen begonnen hat, wie z.B. der Schutz mobilen Maschinencodes oder verteilter Daten.

Agentenbasierte Elektronische Marktplätze Es gibt heute schon eine stattliche Anzahl von *e-commerce*-Systemen, die jedoch in Bezug auf *Funktionalität* und *Wirkungskreis* noch stark eingeschränkt sind. Sie unterstützen den Kunden entweder bei der Produktvermittlung (engl. product brokering, *was* will ich kaufen) und bei der Händlervermittlung (engl. merchant brokering, *wo* soll ich es kaufen), oder bei der Transaktionsabwicklung (engl. negotiation, *wie* wird der Kauf abgewickelt) [GM98]. Eine vergleichende Studie über bereits verfügbare agentenbasierte elektronische Handelssysteme findet sich in [GMM98].

Auf der Kundenseite liegen den Anwendungen als Schnittstelle in der Regel mit der *HyperText Markup Language* (HTML) erstellte Seiten zugrunde, die mittels Web-Browsern angezeigt und bearbeitet werden können. Die Anfragen an die Händler-Server werden über das *Hypertext Transaction Protocol* (HTTP) Protokoll abgewickelt. Dabei ist die Zahl von Händlern stets auf eine feste, vom jeweiligen System festgelegte Vorauswahl begrenzt. Nur die Systeme zur Transaktionsabwicklung sind wirklich agentenbasiert und erlauben das temporäre vom-Netz-Gehen des Benutzers, unterstützen also die Asynchronität. Beispiele dafür sind das Web-basierte Multiagentensystem Kashbah sowie die elektronischen Auktionsanbieter (engl. *electronic auctions*) *Auction Bot* oder *Auction Web*. Dabei ist jedoch der Grad der erforderlichen Interaktion mit dem Benutzer zur Erledigung der Aufgabe sehr hoch.

In Japan wird außerdem ein elektronischer Marktplatz in der realen Welt betrieben [Miy98]: Ausgehend von dem auf dem IBM Aglet Toolkit basierenden Protoypen *TabiCAN*wurde ein Rahmenwerk von abstrakten Funktionen und ein Software-Paket mit Namen *e-Marketplace* entwickelt.

Wie bereits im Abschnitt 2.2.1 aufgezeigt sind Mobile-Agenten-Systeme besonders für den Einsatz im elektronischen Handel und zur Realisierung von elektronischen Marktplätzen geeignet. Sie können dem Endkunden eine Vielzahl von Aufgaben erleichtern und Tätigkeiten wie z.B. das Suchen, Vergleichen, Kaufen oder Verkaufen von Produkten komfortabler gestalten oder gar vollständig automatisieren. Zusätzlich bieten sich Agenten als eigenständige Einheit und Handlungsträger (Akteur) an, da sich reale Rollen in der Wirklichkeit sehr oft in natürlicher Weise auf entsprechende Agenten im Modell abbilden lassen. Zum Beispiel lässt sich die Rolle eines einkaufenden Kunden direkt auf einen einkaufenden Agenten übertragen und die zu besuchenden Händler wiederum durch Händleragenten repräsentieren, mit denen der Kundenagent wie im richtigen Leben interagiert.

Als Leitbeispiel für eine Ausprägung von elektronischem Handel im Internet und für die Betrachtung der Sicherheitsproblematik wird im Folgenden eingehender auf Systeme zur vergleichenden Produktrecherche eingegangen, den sog. *Comparison Shopping*Anwendungen, mit Schwerpunkt auf deren agentenbasierte Realisierung.

3.3 Anforderungen an agentenbasierte Anwendungen im E-Commerce

Wie bereits festgestellt gilt es, mobile Agenten auf unsicheren, eventuell bösartigen Plattformen gegen Angriffe zu schützen. Es stellt sich hier die Frage, welche Objekte oder Daten dabei im elektronischen Handel als kritisch (gefährdet) einzustufen sind, und daher in der Hauptsache als Zielobjekt von Attacken in Frage kommen. Die zur Zeit vorherrschenden Einsatzgebiete elektronischer (agentenbasierter) Handelssysteme sind (1) *Auktionen*und (2) *preisvergleichende Produktrecherche*(Comparison Shopping). Andere mit dem E-Commerce verwandte Anwendungen, die von ihrer

Natur her eher den Charakter von *Punkt-zu-Punkt-Beziehungen* haben, basieren eher auf dem altbekannten Client/Server-Modell (vgl. Abschnitt 2.2.2), das sich in der Praxis bewährt hat. Ein Beispiel dafür ist das (3) *Homebanking*, also das Erledigen von Bankgeschäften von zu Hause aus über eine (zeitweilige) Netzwerkverbindung. In der Regel wird dabei auf dem jeweiligen Netzwerkprotokoll ein sicherer Kanal etabliert, z.B. über einer vorhandenen Internet-Verbindung oder direkt nach Einwahl per Modem und Aufbau einer zeitweiligen Punkt-zu-Punkt Protokollverbindung (PPP). Die große Anzahl von (4) *Online-Verkaufsläden*[1] im World Wide Web werden auch überwiegend mit klassischer (Client/Server) Technologie betrieben, z.B. unter Verwendung von HTML-Masken und CGI-Skripten, die Anfragen an unterliegende Datenbank-Systeme oder proprietäre Anwendungen weiterleiten.

Damit können die grundlegenden Aktionen, die Agenten in Rahmen von elektronischem Handel ausführen, wie folgt identifiziert werden:

- **Produktrecherche**:
  Sammeln von produkt- bzw. dienstleistungsbezogenen Informationen.
- **Produktvergleich**:
  Bewertung bzw. Preisvergleich von Produkten bzw. Dienstleistungen.
- **Produkterwerb**:
  Plazieren von Kaufangeboten bzw. Anzeige von Kaufinteresse gemäß einer vom Agentenbesitzer gewählten und i.d.R. geheimzuhaltenden Strategie.

Dabei sind für die teilnehmenden Parteien verschiedene Aspekte von unter Umständen orthogonaler Bedeutung:

- **Dienstnehmer (Kunde)**
  - *Anonymität* bei Aktionen, die keine Preisgabe der Identifikation des Kunden erfordern (z.B. allgemeine Preisauskünfte[2]).
  - *Schutz der Privatsphäre* – kein Missbrauch von verfügbaren Kundendaten.
  - *Integrität* der übermittelten Daten – Erkennung und Prävention von Manipulation oder Zerstörung von Daten.
  - *Geheimhaltung* der dem Agenten mit auf den Weg gegebenen sensitiven Informationen (z.B. Verhandlungsstrategie) bzw. der vertrauenswürdigen gesammelten Daten.
  - *Qualität und Vertrauenswürdigkeit* der erhaltenen Resultate – wurde wirklich eine umfassende Erhebung durchgeführt oder wurden schlechtere Angebote erfunden (Mondangebote) oder bessere entfernt, um ein spezielles (nicht-optimales) Angebot hervorzuheben?
  - *Zuverlässigkeit* und *Verfügbarkeit* des entsprechenden Agentensystems – ist das System tatsächlich rund um die Uhr einsatzbereit oder gibt es oft Funktionsstörungen?
  - *Preistransparenz* und *Vergleichbarkeit* von Angeboten.

[1] Hier handelt es sich zu einem großen Teil um elektronische Verkaufstheken von Anbietern, deren Kerngeschäft (noch) die herkömmlichen Läden und Einkaufszentren in der realen Welt bilden, beispielsweise Buchläden oder Computerhändler mit zusätzlicher Internet-Präsentanz als Extra-Service.

[2] Im Gegensatz zu kundenspezifische Sonderangeboten, die ggf. eine Authentisierung des Kunden erforderlich machen.

  - *Unabstreitbarkeit* von Angeboten.
  - *Korrektheit* bzw. *Gültigkeit* von erhaltenen Informationen.

- **Dienstgeber (Händler)**
  - *Zuverlässigkeit* und *Verfügbarkeit* des Systems – keine Verdienstausfälle oder Abwanderung von Kunden zu Konkurrenzsystemen.
  - Längerfristige *Kundenbindung* und Aufbau eines *Kundenstamms.*
  - Kundenneigungen und Präferenzen durch *Kundenprofile* vor der Konkurrenz erkennen, ggf. für Direktwerbung oder bedarfsgesteuerte/zielgruppenorientierte Angebotsspezialisierung (**keine** *Anonymität*).
  - *Fairness* – keine Bevorzugung von bestimmten Händlern.
  - Unter Umständen **keine** volle *Preistransparenz* oder *Vergleichbarkeit* von Angeboten.
  - Hohe *Akzeptanz* beim Kunden.
  - *Unabstreitbarkeit* von Äufträgen und Kundenaktionen.

- **Diensterbringer (Systembetreiber)**
  - Hohe *Akzeptanz* bei Kunden und Händlern.
  - Hoher *Standardisierungsgrad* reduziert Wartungsaufwand und Störanfälligkeit.
  - *Schutz der Plattformen* (gegen Angriffe von außerhalb oder innerhalb durch bösartige Agenten).
  - *Hohes Sicherheitsniveau* von getätigten Aktionen, Feststellbarkeit von *Zuständigkeiten* bzw. Zuordnung von getätigten Aktionen zu verantwortlichen Parteien (z.B. falls Diensterbringer auch Schlichterfunktion hat).

Der *Schutz der Privatsphäre* und der Umgang mit kundensensiblen Daten ist primär Angelegenheit von vertraglichen Abstimmungen und festzulegenden Teilnahmebedingungen für die einzelnen Parteien im System. Der *Schutz der Systemhardware* liegt im Zuständigkeitsbereich der jeweiligen Betreiber, genauso wie *Verfügbarkeit* und *Zuverlässigkeit* technische Betriebsaspekte darstellen. Die *Sicherheit der Plattform* ist Aufgabe des Agentensystems selbst, das entsprechende Schutzmaßnahmen und Mechanismen vorsehen muss. *Akzeptanz* ist eine nur indirekt über die anderen Parameter wie Sicherheit und Verfügbarkeit beeinflussbare Größe und nicht von technischer Natur. *Fairness* stellt eine Anforderung an die administrative Leitung des elektronischen Handelssystems, und Maßnahmen in diesem Bereich können sich auch in der technischen Realisierung des Agentensystems (Agentenplattform) widerspiegeln (faire Protokolle, organisatorische Überwachungsmechanismen, Transparenz der Plattformdienste für alle Teilnehmer). Die *Unabstreitbarkeit* von Aktionen oder Angeboten kann durch Realisierung einer geeigneten Public-Key-Infrastruktur erreicht werden. Hierbei verpflichten sich die Teilnehmer vertraglich, dass mit ihrem geheimen Schlüssel signierte Nachrichten verbindlichen Charakter besitzen und etwaige Kaufentscheidungen oder Angebote unabstreitbar sind. Damit lassen sich auch Garantien bzgl. der Korrektheit und *Gültigkeit von Auskünften* an anfragende Agenten modellieren, was aber nicht Gegenstand dieser Arbeit ist. Der Schutz der *Anonymität* von Kunden oder teilnehmenden Parteien generell wird wesentlich durch die Art der Beziehung zwischen Händlern und Kunden auf Protokollebene bestimmt. Dabei spielen jedoch die Aspekte der *Geheimhaltung* von Informationen und des *Integritätsschutzes* von Daten im System ebenfalls eine Rolle. Dies betrifft auch in besonderem Maße die mobilen Agenten selbst. Das heißt, die Agenten müssen durch entsprechende Dienste in die Lage versetzt

werden, die Geheimhaltung und Unversehrtheit der Daten und/oder des Programmcodes selbst garantieren bzw. zumindest überprüfen zu können. Diesbezüglich wurden in der Forschung bereits verschiedene Ansätze diskutiert, wie in Tabelle 2.1 kurz zusammengefasst.

Da zum jetzigen Zeitpunkt keine ausschließlich softwarebasierten Verfahren zum sicheren Zugriff auf geheime Daten in unsicheren Umgebungen zur Verfügung stehen, wie in Abschnitt 2.3.3 bereits beschrieben, ist eine abhörsichere und gegen Manipulationen gefeite Auswertung geheimer Daten (zum Treffen von ggf. sicherheitskritischen Kaufentscheidungen) nur mittels der Verwendung sicherer Hardware möglich.

3.4 Agentenbasiertes Comparison Shopping

Eine typische und naheliegende Anwendung von mobilen Agenten im elektronischem Handel ist der Einsatz von **preisvergleichenden Agenten**, die im Auftrag ihrer Besitzer (der Kunden) selbständig aus einer an sich unüberschaubaren Anzahl von Händlern und Angeboten die in Frage kommenden (oder günstigsten) Angebote ausfindig machen. Die englische Bezeichnung für solche Agenten lautet ***Comparison Shopping Agents***, und wie bei anderen Begriffen im Bereich der Informatik hat auch dieser Ausdruck sein deutschsprachliches Pendant beinahe verdrängt.

Ausgehend vom Grundmodell wird ein agentenbasierter Ansatz vorgestellt, und die Sicherheitsfragen werden diskutiert. Die Grundzüge eines Agentensystems zur preisvergleichenden Produktrecherche umfassen im Wesentlichen zwei Arten von Agenten: Den vom Kunden beauftragten Suchagenten (oder Einkaufsagenten, je nach Befugnis) und eine Anzahl von Verkaufsagenten, die auf Anfrage Auskünfte über aktuelle Händlerangebote geben. Der prinzipielle Ablauf von Comparison Shopping Anwendungen wurde bereits in [RGK97, Vig97, Whi96] skizziert, und ist in etwa der folgende:

1. Der Kunde instruiert den mobilen Agenten mit der Beschreibung der zum Kauf gewünschten Ware.

2. Anschließend wandert der Agent vom Rechner des Kunden zu einem elektronischen Marktplatz und erhält die Adressen aller Händler, die die gewünschte Ware anbieten, z.B. mittels eines Verzeichnisdienstes.

3. Der Agent besucht nun nacheinander alle elektronischen Ladentheken, die jeweils in unterschiedlichen räumlichen Lokationen untergebracht sein können. Nach Angabe der Produktbeschreibung gibt der elektronische Laden ein Preisangebot zurück, das der Agent speichert.

4. Nachdem eine gewisse Menge von Angeboten eingeholt wurde, entscheidet der Agent selbständig,

   (a) zu der Örtlichkeit des Kunden heimzukehren und die gesammelten Ergebnisse zu präsentieren;

   (b) oder alternativ dazu direkt eine Kaufentscheidung zu treffen, falls er über die entsprechende Autorität verfügt und das System solche Aktionen erlaubt. Danach kann er entweder terminieren oder zum Kunden zurückkehren, um ihn über die getroffene Auswahl zu informieren.

Der Einsatz *umgangssprachlicher Produktbeschreibungen* im Modell ist wenig geeignet - einerseits laufen diese dem Bestreben zuwider, die durch den Agenten zu transportierende Datenmenge klein zu halten, andererseits bedingen sie einen erhöhten Verarbeitungsaufwand auf Seiten der elektronischen Läden, wobei eine inkorrekte Interpretation ebenfalls nicht ausgeschlossen werden kann.

Gemeinsame Ontologie. In der Praxis stellt sich in Hinblick auf die Produktbeschreibung die Frage nach einer gemeinsamen *Ontologie*: Agent und elektronische Läden sollen die gleiche Sprache sprechen bzw. die des jeweiligen Verhandlungspartners unmissverständlich interpretieren können. Jeder Agent wird in diesem Zuge mit einer komprimierten Produktbeschreibung ausgestattet, die von allen Teilnehmern des Comparison-Shopping-Systems eindeutig interpretiert werden kann. Diese Produktbeschreibungen werden z.B. in einer *Ontologie-Datenbank* verwaltet und durch einen *Ontologie-Server* zugänglich gemacht.

Im Idealfall sind der *Geltungsbereich* der verwendeten Ontologie global und die Produktbeschreibungen einheitlich für alle elektronischen Märkte. In der Realität ist aber zu erwarten, dass dies nur eingeschränkt für Untergruppen von Märkten der Fall sein wird. Gründe dafür sind nicht zuletzt unterschiedliche Neigungen und Kaufverhalten in verschiedenen Wirtschaftsräumen sowie inkompatible Gesetzesgrundlagen, die den Vertrieb bestimmter Güter oder Dienstleistungen in gewissen Ländern verbieten oder einen genau vorgeschriebenen Umgang damit erfordern. Folglich werden meiner Ansicht nach zumindest im Anfangsstadium elektronischer Märkte lokale Marktverbunde und Händlerringe vorherrschend sein.

In Hinblick auf die *Positionierung* des Ontologie-Dienstes lassen sich dabei zwei Fälle unterscheiden:

A) Der Kunde erzeugt und instruiert seinen mobilen Agenten auf seiner eigenen Plattform (Abb. 3.2).
   In diesem Fall besorgt sich der Kunde zuvor selbst den entsprechenden Produktcode durch Interaktion mit einem Ontologie-Server.

B) Der Kunde hat keinen direkten Kontakt zu mobilen Agenten sondern stellt seine Anfrage über eine Benutzerschnittstelle, die hier als *Zugangsportal* bezeichnet wird. Wie in Abbildung 3.3 gezeigt, bietet das Portal eine HTML-Web-Schnittstelle im World Wide Web an, mit der der Kunde über eine sichere HTTP-Verbindung kommuniziert: Das Zugangsportal bestimmt mit Hilfe der Ontologie-Datenbank und der Benutzerinteraktion den entsprechenden exakten Produktbezeichner, erzeugt den Agenten und sendet die Resultate z.B. per verschlüsselter Email an den Kunden zurück. Dadurch bleibt auch der asynchrone Charakter der Kunden-Agenten-Beziehung erhalten.

Die größte Herausforderung bei der Verwendung der Agententechnologie stellt jedoch die Frage der Sicherheit mobiler Agenten dar. Wie bereits festgestellt, ist der Schutz der Daten und der Integrität mobiler Agenten trotz zahlreicher Forschungsansätze (vgl. Kapitel 2.3.3) ein bisher noch nicht zufriedenstellend gelöstes Problem. Die *Gefahr* liegt darin, dass bösartige Händlerplattformen versuchen, passierende Agenten zu ihrem eigenen Vorteil zu manipulieren, um z.B. Angebote von Konkurrenten zu korrumpieren oder zu vernichten. Die *Dienstverweigerung* durch Rechner kann nicht verhindert werden. Zumindest aber lässt sich erreichen, dass ein bösartiger Händler sein Angebot nur dann einbringen kann, wenn gleichzeitig alle vorher gesammelten Angebote erhalten bleiben [KAG98], und eine Verweigerung der Zusammenarbeit damit für den entsprechenden

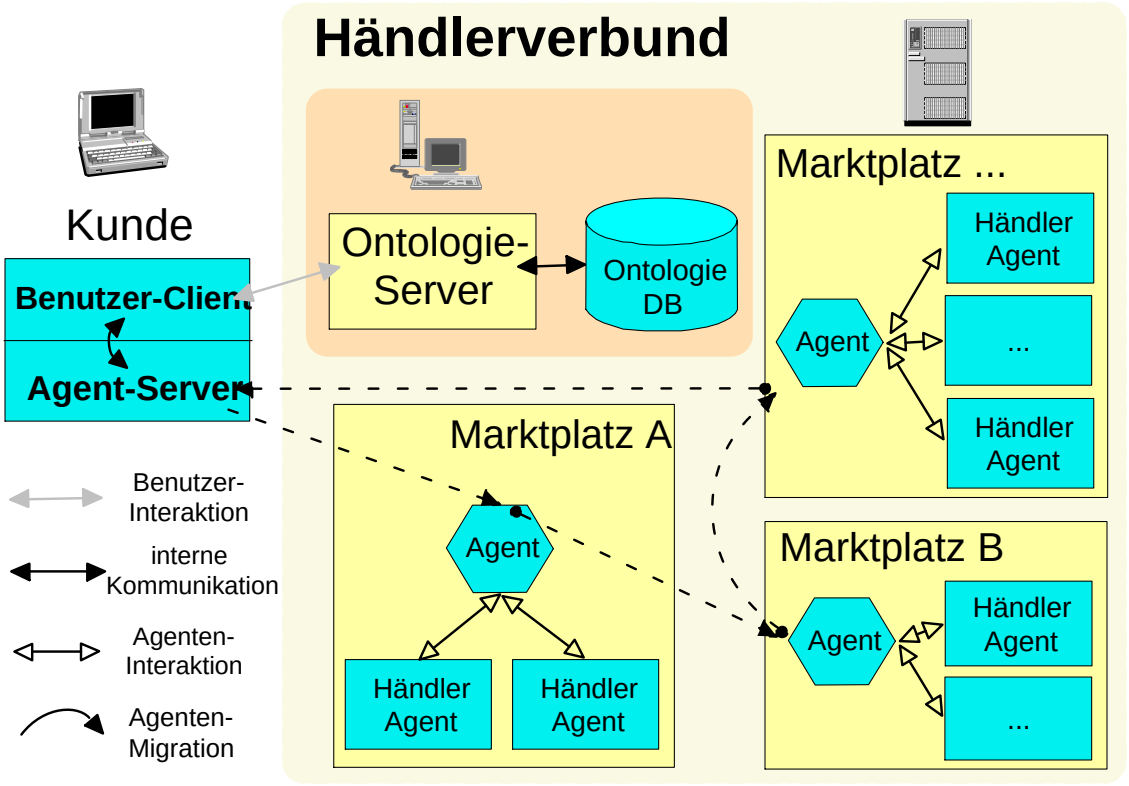


Abbildung 3.2: Modell für ein Comparison-Shopping-Agentensystem mit lokaler Benutzerschnittstelle

Händler nicht wünschenswert ist. Mit Hilfe einer Reihe weiterer Maßnahmen lässt sich die Sicherheit von Comparison-Shopping-Agenten noch zusätzlich erhöhen, wie später gezeigt wird. Zu guter Letzt kann mit Hilfe von sicherer Hardware sogar erreicht werden, dass der Agent nicht nur Angebote sammelt, sondern auch autonom einen bindenden Geschäftsabschluss abwickeln kann (siehe Kapitel 6.7.2).

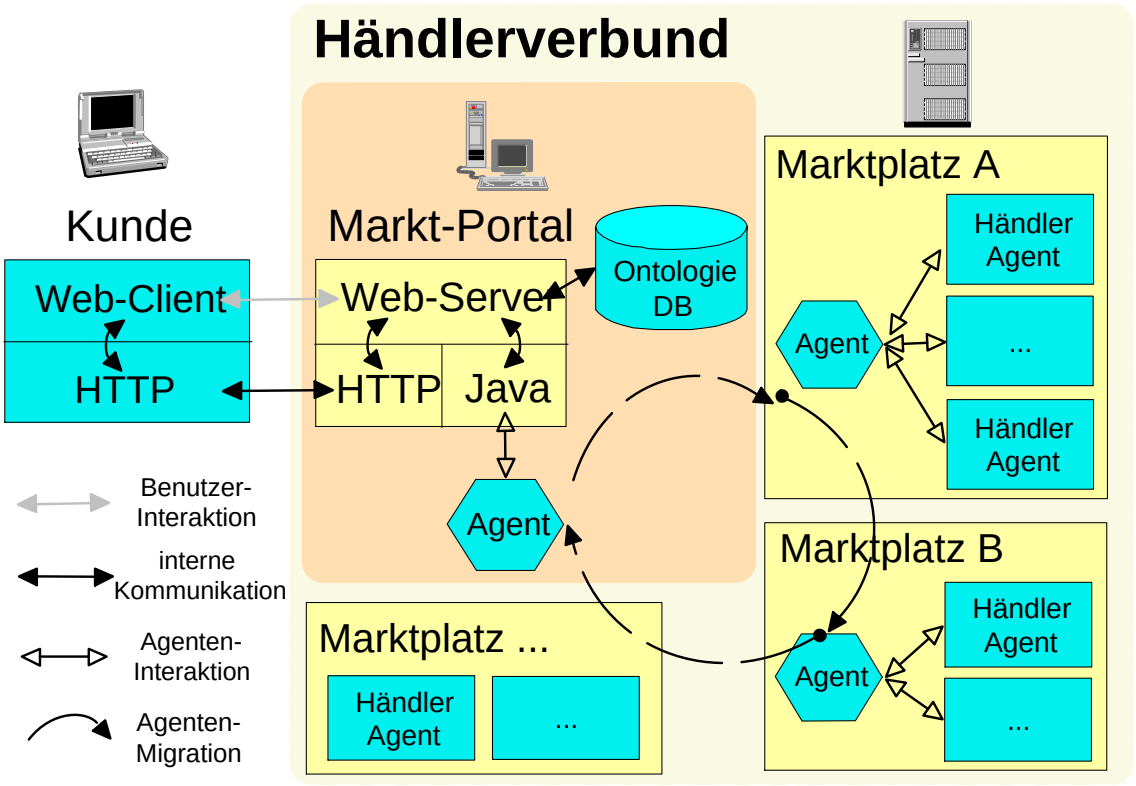


Abbildung 3.3: Modell für ein Comparison-Shopping-Agentensystem mit Web-Schnittstelle als Zugangsportal

Kapitel 4

Die Aglets Agentenplattform

In diesem Kapitel wird das Agentensystem vorgestellt, auf welchem die in dieser Arbeit vorgestellten Sicherheitsdienste prototypisch realisiert wurden. Im Mittelpunkt stehen „leichtgewichtige“ Agenten, die als *Aglets* bezeichnet werden. Aglets sind Java-Objekte, die sich selbst inklusive ihres aktuellen Datenbestandes im Internet von einer Agentenplattform zur nächsten bewegen können, um dort eine bestimmte Tätigkeit zu beginnen oder fortzusetzen. Das Aglet-Prinzip kann als Erweiterung und Generalisierung des Applet- und Servelet-Prinzips aufgefasst werden: Aglets werden – analog zu Applets auf Webservern – auf dezidierten *Aglet-Servern* ausgeführt. Eine ausführliche Beschreibung des Aglet-Systems findet sich in [LO98]. Die nachfolgende Beschreibung beschränkt sich auf die Teile, die zum Verständnis der in dieser Arbeit entwickelten Sicherheitsdienste und des Prototyen notwendig sind. Die Aglets Agentenplattform wurde am IBM Forschungslabor in Tokyo entwickelt und ist unter [IBMa] im Internet erhältlich.

4.1 Das Aglets Rahmenwerk

Das Aglets Rahmenwerk beschreibt

1. ein einfaches und verständliches Modell für Java-basierte mobile Agenten-Anwendungen,
2. das ohne vorherige Modifikation der zugrundeliegenden Java Virtual Machine auskommt,
3. dessen Architektur auf Wiederbenutzung von Komponenten und Erweiterbarkeit ausgelegt ist und
4. das die nahtlose Integration existierender Web-Technologien ermöglicht.

Es stellt zudem Werkzeuge und Java Pakete bereit, die

5. die strukturierte Programmierung benutzerspezifischer Agenten erlauben,
6. dynamische Kommunikation und Interaktion zwischen Agenten unterstützen und
7. die Migration zu anderen Agentenplattformen zur Erledigung spezifischer Aufgaben ermöglichen.

4.2 Architekturüberblick

Die Aglets Architektur besteht aus zwei Schichten, deren Funktionalität durch zwei entsprechende Programmierschnittstellen (Application Programming Interface, API) zur Verfügung gestellt wird:

1. **Aglets Laufzeitsystem** (*runtime layer*):
   Diese Schicht spiegelt die Implementierung der *Aglet-API* wider und legt neben der Realisierung der Grundfunktionalität (Erzeugung, Kontrolle, Migration, Serialisierung von Aglets, Laden und Übertragen von benötigten Klassen, Referenzenverwaltung, Speicherbereinigung (*garbage collection*)) das Verhalten der Systemkomponenten (`AgletProxy`, `AgletContext` usw.) fest.

2. **Aglets Kommunikationsschicht** (*communication layer*):
   Hauptaufgabe der Kommunikationsschicht ist die Übertragung bzw. der Empfang von zuvor serialisierten Agenten. Zudem unterstützt sie die Kommunikation zwischen Agenten und erlaubt deren (entfernte) Verwaltung. Die Kommunikations-API wurde vom Standard MASIF (*Mobile Agent System Interoperability Facility*) der *Object Management Group* (OMG) [OMG] abgeleitet.

Die *Laufzeitkomponente* untergliedert sich weiter in drei wesentliche Bestandteile:

- Persistenzmanager `PersistenceManager` – Speichern auf persistenten Medien und Wiederherstellen von serialisierten Agenten (Programmcode und Zustand/Daten).
- Cachespeicher-Manager `CacheManager` – Verwaltung und Caching des Bytecodes von Agenten.
- Sicherheitsmanager `SecurityManager` – Schutz der Agentenplattform und der Agenten gegen bösartige Parteien. Es gibt systemweit genau eine unveränderliche Instanz des *Sicherheitsmanagners*, der die Zugriffsrechte aller sicherheitsrelevanten Operationen überwacht.

Abbildung 4.1 zeigt, wie die APIs der beiden Schichten miteinander in Verbindung stehen.

Diese Komponenten können jeweils an entsprechende Anforderungen und Umgebungen angepasst werden, zum Beispiel im Zusammenspiel mit Web-Servern (modifizierte Persistenzmanager) oder Web-Browsern (Benutzung des browserinternen Standard-Sicherheitsmanagers). Die Standard-Implementierung der Kommunikationsschicht ist das sogenannte Agent Transfer Protocol (ATP), das auf dem HTTP-Protokoll aufbaut. Es ist ein anwendungsnahes Protokoll für die Übertragung von mobilen Agenten und für die entfernte Inter-Agenten-Kommunikation, und es unterstützt außerdem den Austausch von Nachrichten. Die Programmierschnittstelle der *Kommunikationsschicht* orientiert sich am *OMG-Standard Mobile Agent System Interoperability Facility* **MASIF**, um Interoperabilität zwischen verschiedenen Agentensystemen zu unterstützen.

4.3 Sicherheit von Aglet-Systemen

Karjoth, Lange und Oshima schlagen in [KLO98] ein *Sicherheitsmodell für Aglets* vor, und in der aktuellsten Version des IBM Aglets Software Development Kits (ASDK 1.1b1) wurden laut der Aglets Spezifikation [OKO98] folgende Sicherheitsaspekte bereits realisiert:

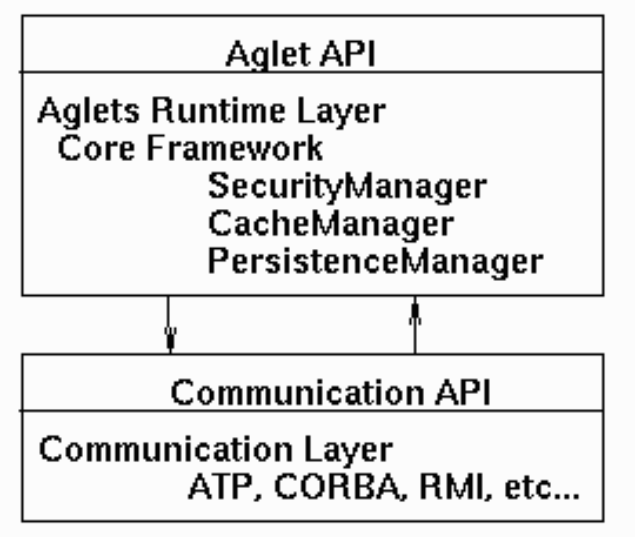


Abbildung 4.1: Architektur der Aglets Agentenplattform: Aglet-API und Kommunikations-API

- Authentifikation von Benutzern und Domänen,
- zuverlässige Kommunikation zwischen den Servern innerhalb einer Domäne und
- feinkörnige Autorisierung ähnlich der des Java 1.2 Sicherheitsmodells.

Sicherheitsmechanismen für den expliziten Schutz der mobilen Agenten selbst wurden nicht spezifiert und sind im Aglet-Baukasten nicht verfügbar.

Die in Abschnitt 2.3.3 aufgezählten Schutzmechansimen lassen sich grob in *anwendungsnahe* und *betriebssystemnahe Dienstklassen* aufteilen, da sie nur zum Teil von der tatsächlich verwendeten Agentenplattform abhängig sind. Durch die Plattform selbst werden im Idealfall die *betriebssystemnahen Sicherheitsprobleme* behoben, zum Beispiel durch

- starke Typisierung, die illegale Speicherzugriffe zur Laufzeit unterbindet,
- getrennte Klassenlader mit Caching und Integritätsprüfung zum Schutz gegen das Einschleusen korrumpierter Klassen und
- restriktive, proxy-basierte Zugriffsmechanismen auf Agentenobjekte.

Außerdem werden die Aglets bei der Ausführung in getrennten Adressräumen gehalten, um gegenseitige Interferenz auszuschließen.

Als von ihrem Charakter her anwendungsorientierte Schutzmaßnahmen und (bis auf die Implementierungsdetails) vom zugrundeliegenden Agentensystem unabhängig können jedoch die

- Protokolle zur Autorisierung und Authentisierung der am System teilnehmenden Entitäten, die
- Protokolle zur Sicherung der Privatsphäre und Anonymität der beteiligten Personen oder Haushalte und die

- Algorithmen und Verfahren zum Schutz der Integrität und Daten der mobilen Agenten

gewählt und durch entsprechende abstrakte Dienste realisiert werden.

Die in dieser Arbeit entworfenen Sicherheitsdienste für Anwendungen im elektronischen Handel haben diesbezüglich noch einen höheren Abstraktionsgrad und fallen somit ebenfalls in die zuletzt beschriebene plattformunabhängige Dienstklasse.

Kapitel 5

Sicherheitsdienste für Mobile Agenten im Elektronischen Handel

In diesem Kapitel werden – ausgehend von den in der Programmiersprache Java bereits vorhandenen Schnittstellen für generische kryptographische Dienste – weiterführende abstrakte Dienste und Datentypen definiert, mit Augenmerk auf den Schutz von mobilen Agenten im elektronischen Handel. Zuerst werden wesentliche Elemente der als Norm für die Programmiersprache Java etablierten Java Security API[1] vorgestellt. Anschließend werden grundlegende Sicherheitsdienste für mobile Agenten identifiziert und motiviert.

5.1 Sicherheit in Java

Die der Programmiersprache Java beigefügten Bibliotheken enthalten bereits ein *Sicherheitspaket* mit Namen `java.security`. Die in diesem Paket zur Verfügung gestellten Sicherheitsdienste beschränken sich dabei in der Hauptsache auf *kryptographische Grundfunktionen* wie Signieren, Ver- und Entschlüsseln sowie die Berechnung von Pseudozufallszahlen und Hashwerten (vgl. Tabelle 5.1) und definieren grundlegende Datentypen (vgl. Tabelle 5.2). Ferner werden weitere Pakete zum Gebrauch von *Zertifikaten* (`java.security.cert`) für algorithmenspezifischere Schnittstellen und Parameterspezifikationen angeboten. Um auch Implementierungen anderer Hersteller oder zukünfige kryptographische Verfahren später bei Bedarf dynamisch eingliedern zu können, wurden sogenannte *Diensterbringer-Schnittstellen* definiert, die sogenannten *Service Provider Interface*s (SPI). Damit können z.B. den Klassen `SecureRandom`, `KeyPairGenerator` oder `MessageDigest` durch Implementierung der entsprechenden abstrakten Service-Provider-Klassen `SecureRandomSpi`, `KeyPairGeneratorSpi` oder `MessageDigestSpi` eigene, alternative Realisierungen des jeweiligen Dienstes zu Verfügung gestellt werden.

Das kryptographische Erweiterungspaket *Java Cryptography Extension* (JCE) bietet weitere Dienste, wie z.B. den Schlüsselaustausch oder das `Cipher`-Objekt mit entsprechenden *Stream*-Klassen (vgl. Tabelle 5.3). Dieser Programmcode unterliegt den US-amerikanischen Exportrestriktionen (siehe Kapitel 5.7). Es gibt jedoch in der Zwischenzeit auch (kommerzielle) JCE-Implementierungen außerhalb der Exportverbotszone, z.B. das Paket IAIK-JCE der Technischen Universtität Graz in Österreich [IAI99]. Analog zum Java 1.2 Sicherheitspaket können auch hier

[1] Bestandteil des Java 2 Standards, Java^{TM} 2 Platform, Standard Edition, v1.2 (J2SE).

Klasse	Beschreibung
`GuardedObject`	Objekt, das den Zugriff auf ein anderes Objekt kontrolliert.
`KeyPair`	Objekt, das ein Schlüsselpaar enthält.
`KeyPairGenerator`	Objekt zum Erzeugen von Schlüsselpaaren.
`KeyStore`	Transiente Schlüssel und Zertifikatssammlung im Arbeitsspeicher.
`MessageDigest`	Bietet allgemeine Funktionalität kryptographischer Hashfunktionen.
`SecureRandom`	Kryptographisch sicherer Pseudozufallszahlen-Generator.
`Signature`	Grundfunktionen elektronischer Signaturverfahren.
`SignedObject`	Erlaubt die Erzeugung signierter Objekte mit geschützter und verifizierbarer Integrität zur Laufzeit.

Tabelle 5.1: Auswahl kryptographischer Dienstklassen der Java 1.2 Security API

Schnittstelle	Beschreibung
`Guard`	Wächter, der den Zugriff auf andere Objekte überwacht.
`Key`	Allgemeinste Schnittstellendefinition für alle Schlüsselklassen.
`Principal`	Abstrakte Definition im System teilnehmender Entitäten.
`PrivateKey`	Ein privater Schlüssel.
`privilegedAction`	Eine mit privilegierten Rechten ausgeführte Aktion (bzw. Berechnung).
`PublicKey`	Ein öffentlicher Schlüssel.

Tabelle 5.2: Kryptographische Schnittstellen der Java 1.2 Security API

Schnittstellendefinitionen	Beschreibung
`SecretKey`	Ein geheimer (symmetrischer) Schlüssel.

Klassenübersicht (Auswahl)	Beschreibung
`Cipher`	Bietet Funkionalität einer kryptographischen Chiffre zum Ver- und Entschlüsseln von Daten.
`CipherInputStream`	Eingabestrom, dessen Daten beim Auslesen durch die zugehörigen Chiffre bearbeitet werden.
`CipherInputStream`	Ausgabestrom, dessen Daten vor dem effektiven Schreiben durch die zugehörigen Chiffre bearbeitet werden.
`KeyAgreement`	Realisiert Protokoll zur Schlüsselvereinbarung und zum Schlüsselaustausch.
`KeyGenerator`	Schlüsselgenerator für (symmentrische) Schlüssel.
`Mac`	Realisiert Message Authentication Code (MAC) (Nachrichten-Authentizierungscode).
`SealedObject`	Erlaubt das kryptographische Versiegeln eines Objektes, um die Vertraulichkeit der Daten zu gewährleisten.
`SecretKeyFactory`	Fabrik für geheime Schlüssel.

Tabelle 5.3: Auswahl kryptographischer Dienste der Java 1.2 Cryptography Extension

durch Ausprogrammieren von Diensterbringer-Klassen eigene Implementierungen der eigentlichen Programmierschnittstelle transparent unterfüttert werden. So gibt es zu den Klassen `Cipher`, `KeyAgreement`, `KeyGenerator`, `Mac` und `SecretKeyFactory` entsprechende Service-Provider-Klassen (`CipherSpi` usw.).

5.2 Elementare Sicherheitsdienste für Mobile Agenten

Die Sicherheit mobiler Agenten in unsicheren Umgebungen erfordert den Schutz des *Zustandes* des Agenten und der *Ausführung* auf der unsicheren Plattform.

Der Zustand eines Agenten wird durch den auszuführenden **Maschinencode** und die mitgeführten **Daten** bestimmt.

Den Begriff *Schutz des Agenten* grenzen wir auf die Bereiche **Integrität** und **Vertraulichkeit** ein. Er hat dabei unterschiedliche Bedeutung, je nachdem, ob auf den Zustand bzw. die Ausführung des Agenten Bezug genommen wird. In Abbildung 5.1 werden verschiedene Ausprägungen aufgeführt und jeweils dafür typische Schutzmechanismen mit praktischem Beispiel aufgezählt.

Während der Maschinencode eines Agenten zur Laufzeit einen *statischen* Charakter besitzt, lassen sich die Daten bezüglich ihrer Persistenz nach als *statisch* oder *dynamisch* klassifizieren.

Das dynamische Laden von Maschinencode und die damit verbundene Sicherheitsproblematik ist nicht Bestandteil dieser Arbeit. Hierbei treten zusätzliche Sicherheitsrisiken in Hinblick auf die Korrektheit und Gutartigkeit der dynamisch eingebundenden Bibliotheken auf. Es stellen sich dabei außerdem die Fragen der Programmverifikation, Maschinencode-Zertifizierung und sicheren (starken) Typentrennung.

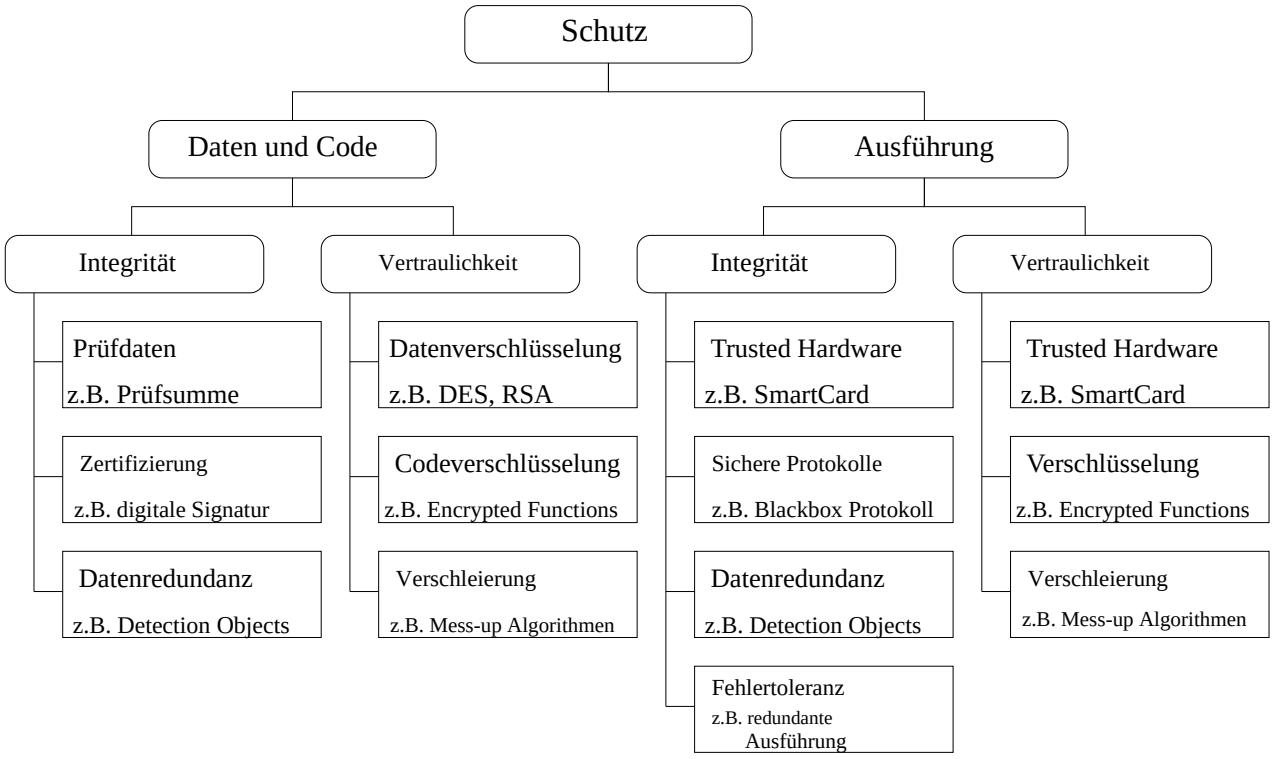


Abbildung 5.1: Klassifikation von Schutzmechanismen für mobile Agenten

Eine weitere Untergliederung des Datenanteils ist nach semantischen Gesichtspunkten möglich:

a) *Nutzdaten* **ND**:
Alle Daten und Informationen, die zur Erledigung primären Aufgabe des Agenten benötigt bzw. gesammelt werden.
Beispiel: Gesammelte Angebote im Comparison Shopping.

b) *Metadaten* **MD**:
Diese Daten tragen nur indirekt zur Lösung der durch den Agenten zu bearbeitenden Aufgabenstellung bei. Sie werden (im dynamischen Fall) durch Auswertung der Nutzdaten regelmäßig aktualisiert und dienen dem Agenten als Wissensbasis und strategische Entscheidungsgrundlage.
Beispiele: Favoritenliste (*Hotlist*) der auf jeden Fall zu besuchenden Händlern; minimale und maximale Anzahl zu sammelnder Angebote.

c) *Schutzdaten* **SD**:
Die Schutzdaten tragen nicht zur Erfüllung der eigentlichen Agententätigkeit bei, sondern dienen ausschließlich dem Schutz der Integrität und Vertraulichkeit der Nutzdaten, Metadaten oder gar des Maschinencodes selbst.
Von Protokollen verwaltete Schutzdaten werden unten auch als *Protokolldaten* **PD** bezeichnet.
Beispiele: Verschlüsselung der Angebote mit öffentlichem Schlüssel des Agenteneigentümers; Kette von Hashwerten über alle gesammelten Angebote.

Die auf den identifizierten Datenarten operierenden Parteien des Mobilen-Agenten-Szenarios lassen sich in drei Kategorien einteilen:

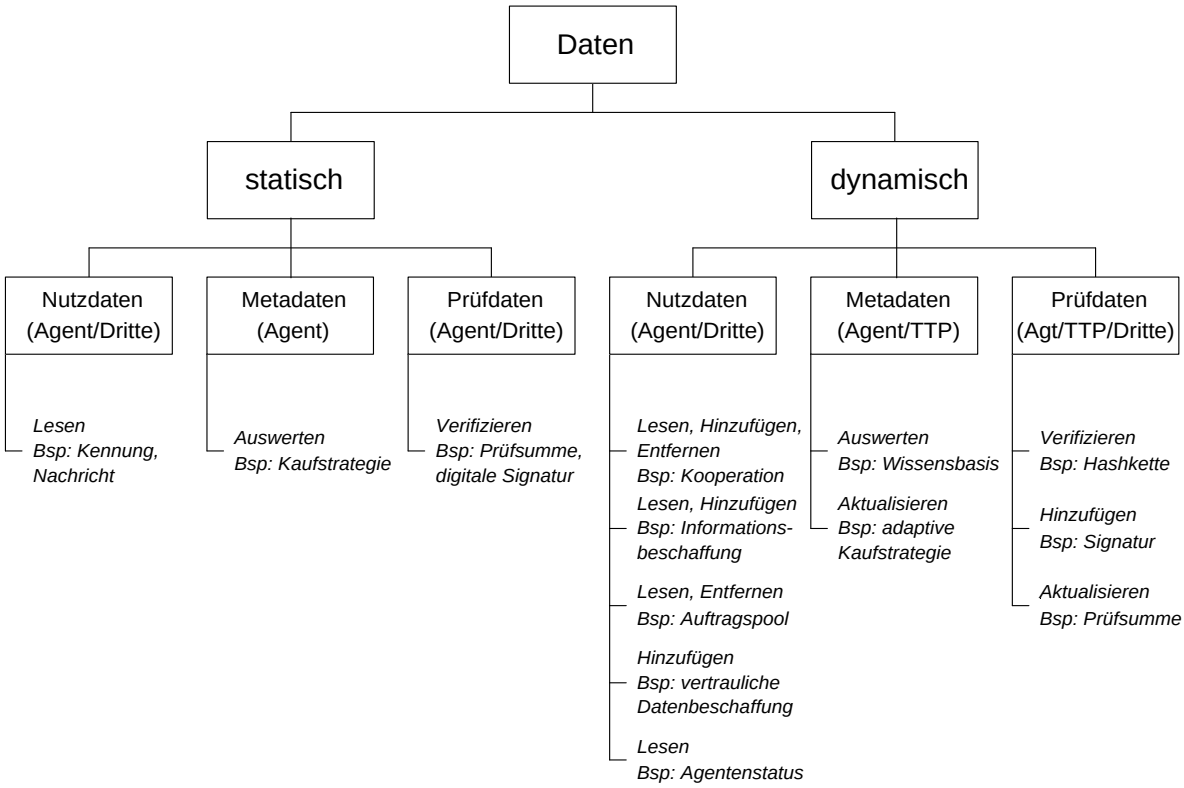


Abbildung 5.2: Klassifikation der Daten mobiler Agenten und Identifikation der darauf operierenden Dienstprimitiven.

- `Agent`: Der mobile Agent selbst.
- `Trusted Third Party` (`TTP`): Von allen teilnehmenden Parteien als vertrauenswürdig anerkannte Instanz. Diese kann auch lokal in unzuverlässigen und unsicheren Umgebungen in Form von sicherer, vertrauenswürdiger Hardware (`Trusted Hardware`) verfügbar sein, die von der TTP ausgegeben und verwaltet wird.
- `Dritte`: Alle anderen am System beteiligten Parteien, wie beispielsweise die ausführende Plattform, andere Agenten usw.

Eine baumartige Gliederung der Daten mobiler Agenten erfolgt in Abbildung 5.2. Die eingeklammerten Begriffe in den untersten Knoten (Blättern) nennen die Parteien, die auf den jeweiligen Datentyp zugreifen. Darunter werden in einer Liste die anfallenden Primitiven identifiziert, die typischerweise auf den zugehörigen Daten operieren. Bei den dynamischen Nutzdaten wurden die Primitiven darüberhinaus zu Gruppen zusammengefasst. Die getroffene Einteilung wird anhand von Beispielen kurz motiviert.

Davon lassen sich – wie in Tabelle 5.4 vorgeführt – die grundlegenden Datentypen und **Sicherheitsdienste** für mobile Agentenanwendungen ableiten:

- Sichere Dienste für den Umgang mit invarianten, statischen Daten (Tab. 5.4, 1. Teil):

Datenart	Operation	sicherer Dienst	Java Dienstklasse
statisch			
Nutzdaten	Lesen	Nur-Lese-Container	`ReadOnlyContainer`
Metadaten	Auswerten	Sichere Berechnung	`SecureComputation`
Prüfdaten	Auswerten	kryptograph. Primitive	`java.security.*`
dynamisch			
Nutzdaten	Lesen, Hinzufügen, Entfernen	abschließbarer Container	`LockableContainer`
	Lesen, Hinzufügen	Ohne-Entfernen Cont.	`AppendOnlyContainer`
	Lesen, Entfernen	Ohne-Hinzufügen Cont.	`RemoveOnlyContainer`
	Hinzufügen	Nur-Hinzufügen Cont.	`SecretAppendOnlyContainer`
	Lesen	Dyn. Nur-Lese-Container	`DynamicReadOnlyContainer`
Metadaten	Auswerten	Sichere Berechnung	`SecureComputation`
	Aktualisieren	Sichere Berechnung	`SecureComputation`
		kryptogr. Primitive	`java.security.*`
Prüfdaten	Verifizieren	kryptogr. Primitive	`java.security.*`
	Hinzufügen	kryptogr. Primitive	`java.security.*`
	Aktualisieren	kryptogr. Primitive	`java.security.*`
		Sichere Berechnung	*Trusted Hardware* Lösung

Tabelle 5.4: Grundlegende Sicherheitsdienste für mobile Agentenanwendungen.

1. *Lesen von Nutzdaten* → Container mit ausschließlichem Lesezugriff.
   ⟶ `ReadOnlyContainer`
2. *Auswerten von Metadaten* → Container mit ausschließlichem Lesezugriff bzw. sichere, vertrauliche Auswertung (Berechnung).
   ⟶ `ReadOnlyContainer` bzw. `SecureComputation`
3. *Auswerten von Prüfdaten* → Verifizieren von statischen Prüfeigenschaften und Integritätsbedingungen.
   ⟶ allgemeine kryptographische Primitive oder `SecureComputation`

Der `ReadOnlyContainer` kann auch als Gruppenoperation ausgelegt werden, `GroupReadOnlyContainer`, der nur einer Untermenge von Anwendern Lesezugriff gestattet (siehe Abschnitt über sichere Containerklassen in Kapitel 5.3.2).

- Sichere Dienste für den Umgang mit veränderlichen dynamischen Daten (Tab. 5.4, 2. Teil):
  - Auf den Nutzdaten operierende Dienste:
    1. *Lesen*, *Hinzufügen* und *Entfernen* → Container mit wahlfreiem Zugriff, der bei Bedarf kryptographisch verschlossen werden kann.
       ⟶ `LockableContainer` oder auch die direkte Verwendung des `IntegrityStack`
    2. *Lesen* und *Hinzufügen* → Container mit Lesezugriff, dem neue Objekte hinzugefügt, aber nicht entnommen werden können.
       ⟶ `AppendOnlyContainer` oder auch direkt `PushOnlyIntegrityStack`

3. *Lesen* und *Entfernen* → Container mit Lesezugriff, dem Objekte entnommen, aber nicht hinzugefügt werden können.
   ⟶ `RemoveOnlyContainer`
4. *Hinzufügen* → Container, der nur das Hinzufügen neuer Objekte erlaubt, nicht aber das Auslesen oder Entfernen von Objekten.
   ⟶ `SecretAppendOnlyContainer`
5. *Lesen* → Container mit ausschließlichem Lesezugriff für Dritte, dessen Inhalt sich aber in Abhängigkeit des Agentenzustandes oder der internen Operationen dynamisch zur Laufzeit verändern kann.
   ⟶ `DynamicReadOnlyContainer`

Alle nutzdatenbezogenen Dienste lassen sich außerdem als *Gruppenoperationen* realisieren, d.h. dass der Zielbenutzerkreis auf eine Untergruppe von Anwendern eingeschränkt wird. Daraus ergeben sich dann die Dienste `GroupLockableContainer`, `GroupAppendOnlyContainer`, `GroupRemoveOnlyContainer` usw. (siehe Gruppencontainerklassen in Abschnitt 5.3.2).

– Auf den Metadaten operierende Dienste:

1. *Auswerten* → Metadatenobjekt, das die sichere Auswertung bzw. Berechnung von Daten erlaubt, ohne jedoch die im Objekt selbst enthaltene Datenbasis (*Wissensbasis*) zu modifizieren.
   ⟶ `SecureComputation` bei vertraulicher Datenbasis, sonst `ReadOnlyContainer`
2. *Aktualisieren* → Metadatenobjekt, das die sichere und vertrauliche Aktualisierung der im Objekt selbst enthaltenen Datenbasis erlaubt, ohne Rückschlüsse auf Art und Inhalt der gespeicherten Daten zu ermöglichen.
   ⟶ `SecureComputation`
3. *Auswerten* und *Aktualisieren*→ Metadatenobjekt, das (1.) die sichere Auswertung von gesammelten Daten und (2.) die vertrauliche und sichere Aktualisierung der im Objekt selbst enthaltenen Wissensbasis ermöglicht.
   ⟶ `SecureComputation`

– Typische Operationen auf den Prüfdaten:

1. *Verifizieren* → Verifizieren von dynamisch veränderlichen Eigenschaften anhand bestimmter Integritätsbedingungen oder Kontrolldaten.
   ⟶ allgemeine kryptographische Primitive oder `SecureComputation`
2. *Aktualisieren* → Aktualisieren von bereits existierenden Prüfdaten nach Zustandsänderungen oder Aktionen des Agenten.
   ⟶ `SecureComputation`
3. *Hinzufügen* → Kumulatives Sammeln von Zustandskontrolldaten: Hinzufügen von neuen, zusätzlichen, nicht-manipulierbaren Informationen, anhand derer zu einem späteren Zeitpunkt die Integrität der Nutz- bzw. Metadaten getestet überprüft werden kann. Das führt im Allgemeinen zu streng monoton wachsendem Prüfdatenaufkommen, ist aber sicherheitstechnisch einfacher handzuhaben und zu schützen als die Operation *Aktualisieren*, da bereits existierende Daten nicht in unsicherer Umgebung ausgewertet werden müssen oder entfernt (ersetzt) werden können.
   ⟶ allgemeine kryptographische Primitive oder `SecureComputation`
4. *Verifizieren* und *Aktualisieren* → Kombination der beiden bereits beschriebenen Operationen, die den restriktiveren Schutzmechanismus der sicheren Berechnung

erfordert.
$\longrightarrow$ `SecureComputation`

5. *Verifizieren* und *Hinzufügen* $\rightarrow$ Kombination der beiden bereits beschriebenen Operationen, die den restriktiveren Schutzmechanismus der sicheren Berechnung erfordert.
$\longrightarrow$ `SecureComputation`

Damit sind die erforderlichen elementaren Sicherheitsdienste für mobile Agenten identifiziert. Sie stellen *Grundbausteine* für den Entwurf sicherer Agentensysteme auf anwendungsnaher Ebene dar und dienen als Grundlage für die Erstellung problemspezifischer, komplexerer Sicherheitsdienste und die Entwicklung sicherer Protokolle.

Generische providerunabhängige Dienste. Neu in dieser Arbeit ist auch die Tatsache, dass die vorgestellten Dienste generischer Natur sind. Gemäß dem bereits für das Java 2 Sicherheitspaket eingeführten *Service Provider Implementation*-Konzept (siehe *SPI*, Kapitel 5.1) ist die Realisierung streng von der Konzeption getrennt, und es können verschiedene Implementierungen desselben Dienstes existieren, die jeweils je Dienstimplementierung eine eindeutige Kennung besitzen.
Dies ermöglicht dem Benutzer später – unabhängig von der Implementierung und Konzeption des Agenten – durch Wahl eines entsprechenden *SPI*s bei der Initialisierung des abstrakten Sicherheitsdienstes die tatsächlich zu verwendende Implementierung festzulegen. Im Falle von kryptographischen Operationen also z.B. den Algorithmus, etwa „RSA“, und den Hersteller (*Provider*), etwa „ACME_JAVACARD“, um explizit eine bestimmte JavaCard-Realisierung des Dienstes auszuwählen.

Es ist dadurch also möglich, für einen Agenten parametrisch zu entscheiden, entweder nur Sicherheitsdienste mit einer konkreten Implementierung mittels sicherer Hardware zuzulassen (insofern das System das unterstützt), oder eine bevorzugte softwarebasierte Implementierung zu verwenden.

5.3 Beschreibung der Sicherheitsdienste

Die in diesem Abschnitt beschriebenen elementaren generischen Sicherheitsdienste und abstrakten Datentypen eignen sich aufgrund ihres hohen Abstraktionsgrades nicht nur für Anwendungen im E-Commerce, sondern realisieren einen generellen Schutz für mobile Agenten auch in anderen Einsatzbereichen. Darüberhinaus dienen diese Dienste als Grundlage für die Entwicklung anwendungsspezifischer Sicherheitsdienste, z.B. im Bereich des elektronischen Handels, wie nachfolgend ebenfalls skizziert wird.

5.3.1 Integritätsgeschützter Stack

Der integritätsgeschützte Stack stellt eine Erweiterung der von Devanbu und Stubblebine in [DS98] vorgestellten integritätsgeschützten und resourcenlimitierten Stack-Implementierung auf verteilte Systeme dar. Kernpunkt dabei ist die *minimale Verwendung von sicherer Hardware*, die ausschließlich zur geschützten Verwaltung und Berechnung der (zum Teil) geheimen Kontrolldaten eingesetzt wird. In diesem Sinne muss nur ein Bruchteil der im Stack enthaltenen Daten in der sicheren Hardware (Blackbox) für Prüfzwecke bearbeitet werden, während die unter Umständen sehr umfangreichen restlichen Daten im externen Speicher gehalten werden können. Die sicherheitskritische

Datenstruktur (IDU, *Integrity Data Unit*) enthält dabei nur den geheimen Anker und den zuletzt berechneten Wert der Hashkette, mit deren Hilfe sich die Integrität der externen Daten überprüfen lässt, wie in Abbildung 5.3 demonstriert wird.

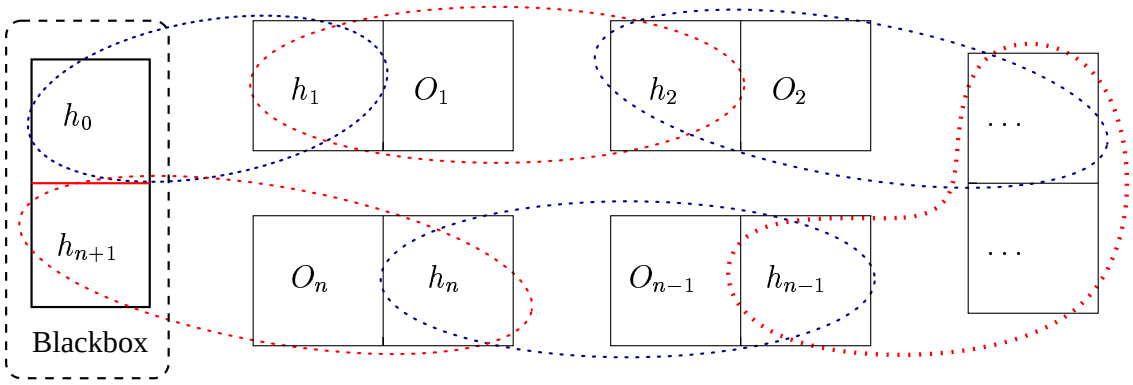


Abbildung 5.3: Aufbau der integritätsschützenden Hashkette nach Stubblebine [DS98]: h_0 ist der geheime Startwert (Verankerung), h_{n+1} der Abschluss (Terminierung). Die rekursive Berechung der Hashwerte h_i erfolgt über den durch die gestrichelten Ellipsen umfassten Daten. $< h_0, h_{n+1} >$ ist das kritische Datentupel (IDU), das zur Überprüfung der Hashkette benötigt und nur innerhalb der sicheren Hardware ausgewertet und aktualisiert wird.

Der integritätsgeschützte Stack steht in zwei Ausprägungen zu Verfügung:

- **`IntegrityStack`**
  Verteilt realisierter Stack, dessen Integrität nach dem von Devanbu und Stubblebine entworfenen Protokoll geschützt wird. Die Korrektheit des Protokolls wird in [DS98] bewiesen.
- **`PushOnlyIntegrityStack`**
  Eine Spezialisierung des `IntegrityStack`, indem nur Objekte auf dem Stack abgelegt, nicht aber entnommen werden dürfen. *Anwendungsbeispiel*: Verwendung bei der Realisierung des `AppendOnlyContainers`, siehe Kapitel 5.3.2.

5.3.2 Sichere Containerklassen

Die Mehrzahl der in Kapitel 2.2.3 skizzierten Einsatzgebiete mobiler Agenten verlangt den Umgang mit persönlichen Daten ihres jeweiligen Benutzers (z.B. Agenten als Adressassistenten) bzw. das Beschaffen oder den Transfer evtl. sensitiver Informationen (Theaterkarten bestellen, Bankgeschäfte erledigen, Produktrecherche durchführen). Diesem Umstand wird durch eine breite Palette von Container-Dienstklassen Rechnung getragen werden, die die Vertraulichkeit und Integrität der Daten schützen. Die entworfenen Klassen können dabei grundsätzlich in allgemeine und gruppenorientierte Containerklassen eingeteilt werden. Also in Containerklassen *ohne Zugriffsbeschränkung* für individuelle Benutzer und in eine weitere Klasse mit *eingeschränkter Zielgruppe.*

Containerklassen ohne Zugriffsbeschränkung

- Klasse `LockableContainer`
  Dieser Container kann beliebige serialisierbare Objekte aufnehmen, und lässt sich mit Hilfe

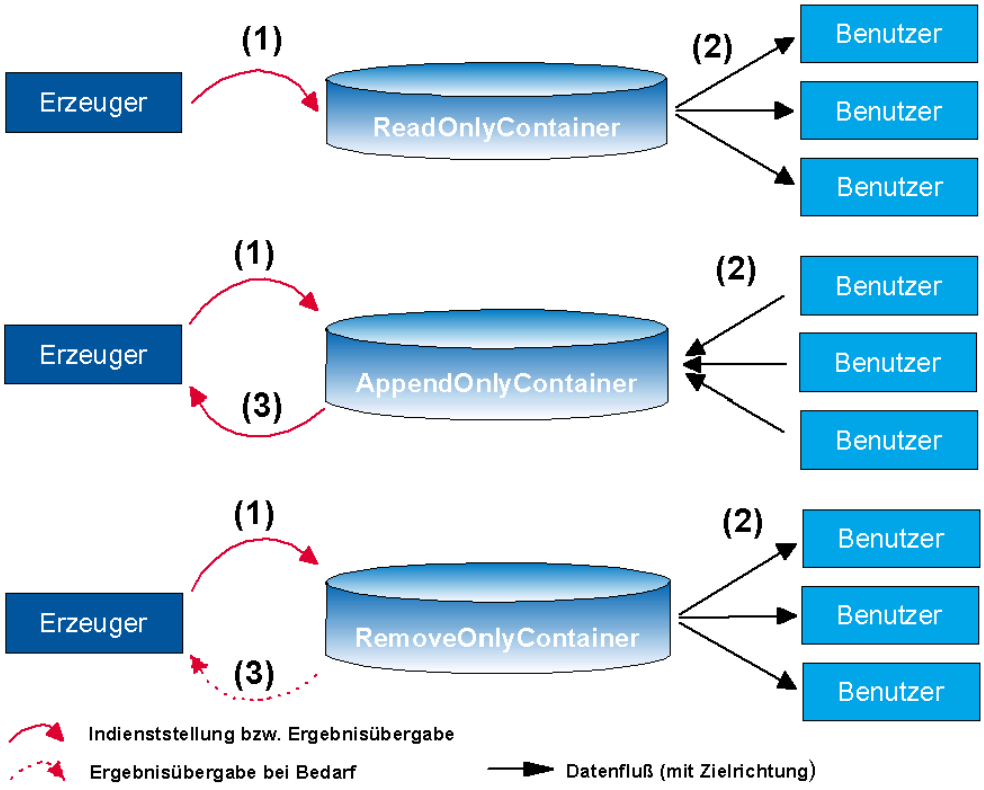


Abbildung 5.4: Allgemeine Containerklassen.

eines geheimen Schlüssels bei Bedarf entweder auf- oder abschließen, d.h. der Inhalt wird entweder ver- oder entschlüsselt. Der Zugriff auf darin abgelegte Elemente erfolgt durch Angabe eines Objekt-Bezeichners. Der *LockableContainer* stellt eine Verallgemeinerung des `SealedObjects` dar (vgl. Abb. 5.3).

- Klasse `ReadOnlyContainer`
  Der Inhalt dieses Containers wird zum Zeitpunkt der Erzeugung versiegelt, so dass er zwar jederzeit von jedem gelesen, aber nicht geändert werden kann. Manipulationsversuche können eindeutig erkannt werden.
- Klasse `RemoveOnlyContainer`
  Dieser Container erlaubt nur das Lesen und Entfernen von Objekten, z.B. von Aufträgen oder begrenzten Ressourcen.
- Klasse `AppendOnlyContainer`
  Dieser Container erlaubt nur das Anhängen von Informationen an den bereits vorhandenen Datenbestand. Die Daten können ohne Einschränkung gelesen werden.
- Klasse `SecretAppendOnlyContainer`
  Ein `AppendOnlyContainer`, bei dem die Daten verschlüsselt werden und nur vom Erzeuger des Containers selbst (oder von einem zum Zeitpunkt der Erzeugung festgelegten Benutzer) gelesen werden können.

Abbildung 5.4 veranschaulicht die Verwendung einer Auswahl dieser Containerklassen. Die gewundenen Pfeile markieren die Erzeugung (Indienststellung) bzw. Auswertung (Ergebnisrückgabe)

eines Containers, die geradlinigen Pfeile symbolisieren die Zugriffe darauf. Die zeitliche Abfolge der Schritte wird durch die Zahlen in runden Klammern angezeigt.

Containerklassen mit eingeschränkter Zielgruppe: Die nachfolgenden Klassen schränken die Funktionalität der verschiedenen Containerarten auf einen festzulegenden Benutzerkreis ein.

- Klasse `GroupLockableContainer`
  Dieser Container erlaubt das gemeinsame Bearbeiten von (geheimen) Daten durch eine bestimmte Gruppe von Personen. Beim Erzeugen des Containers wird der geheime (symmetrische) Schlüssel mit dem öffentlichen Schlüssel jedes gewünschten Gruppenmitglieds verschlüsselt und mit dem Container weitergegeben.
- Klasse `GroupReadOnlyContainer`
  `ReadOnlyContainer` mit eingeschränktem Benutzerkreis.
- Klasse `GroupRemoveOnlyContainer`
  `RemoveOnlyContainer` mit eingeschränktem Benutzerkreis.
- Klasse `GroupAppendOnlyContainer`
  `AppendOnlyContainer` mit eingeschränktem Benutzerkreis.
- Klasse `GroupSecretAppendOnlyContainer`
  `SecretAppendOnlyContainer` mit eingeschränktem Benutzerkreis.

In Abbildung 5.5 wird der Einsatz verschiedener Gruppencontainer graphisch dargestellt. Die Zahlen in den runden Klammern legen die zeitliche Abfolge der Schritte fest.

5.3.3 Sichere Berechnungen auf unsicheren Plattformen

Durch Einsatz einer *Trusted Hardware* wird die *sichere Berechnung* unter Verwendung vertraulicher Informationen ermöglicht, auch auf ansonsten unsicheren und unzuverlässigen Plattformen.

Dienste zur sicheren Berechnung lassen sich Ihrer Art nach in zwei Klassen einteilen:

a) *Statischer* sicherer Berechnungsdienst:
   Der Dienst wird fest vom Systemadministrator auf der Hardware installiert. Alle Benutzer teilen sich denselben Dienst.

b) *Dynamischer* sicherer Berechnungsdienst:
   Jeder Anwender kann zeitweilig seinen eigenen benutzerspezifischen Dienst auf der sicheren Hardware zur sicheren Ausführung installieren. Jeder Benutzer führt nur genau seinen eigenen Dienst aus und ist für diesen Zeitraum alleiniger Nutzer der sicheren Hardware.

Die statische Variante ist sicherheitstechnisch leichter zu handhaben, da die angebotenen Sicherheitsdienste nicht von Dritten manipuliert werden können und die Ausführung streng dem realisierten Sicherheitsmodell folgt.

Die dynamische Variante erlaubt dem Benutzer verändernden Zugriff auf die sichere Hardware. Falls die sichere Hardware weitere sichere (System-)Dienste realisiert, so muss zusätzlich sichergestellt werden, dass benutzerinstallierte Dienste isoliert ablaufen und die Ausführung anderer Dienste

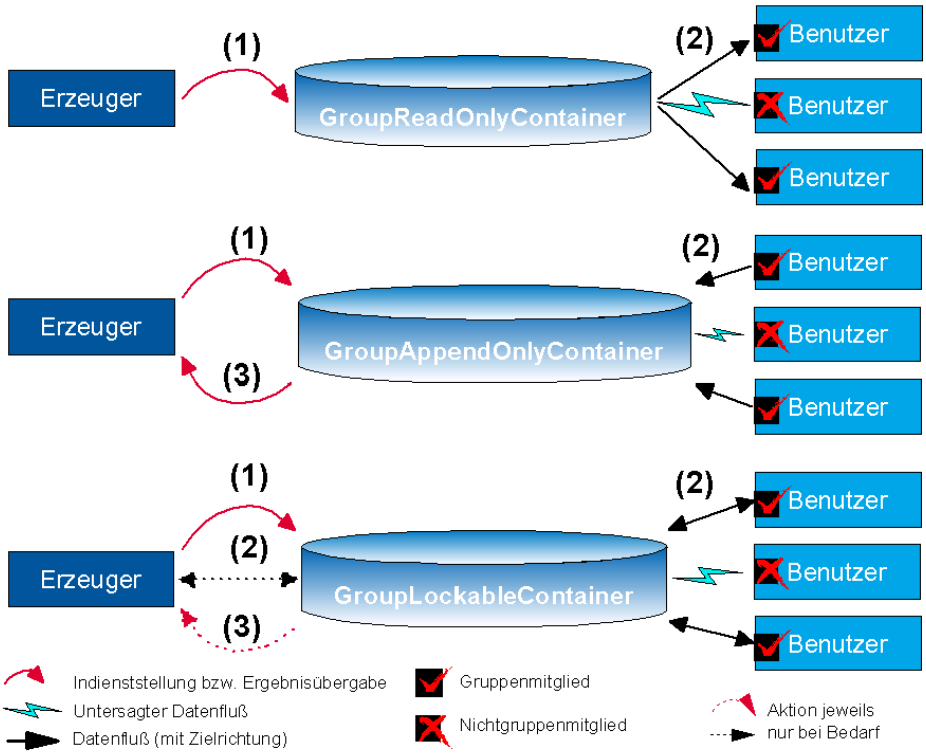


Abbildung 5.5: Auf bestimmte Benutzergruppen eingeschränkte Containerklassen.

weder direkt noch indirekt beeinflussen können, also z.B. müssen die Adressräume streng getrennt werden und der Missbrauch von begrenzten Resourcen auf der Trusted Hardware verhindert werden (*Denial-of-Service* Angriff). Die Sicherheit dynamischer, auf Trusted Hardware implementierter sicherer Dienste erfordert weitere Forschung.

In dieser Arbeit werden nur statische sichere Berechnungsdienste, auf SmartCards als Trusted Hardware installiert, betrachtet, da im Comparison Shopping Modell nur fest installierte Dienste benötigt werden, wie zum Beispiel die Dienste des sicheren Stacks in Kapitel 5.3.1, auf den verschiedene Containerklassen aus dem Kapitel 5.3.2 aufsetzen. Der *prinzipielle Aufbau* eines sicheren *statischen* Berechnungsdienstes in Java umfasst zum einen die *abstrahierte Schnittstelle* zur unterliegenden sicheren Implementierung, die die tatsächliche Abbildung auf eine konkrete sichere Hardware (z.B. SmartCard) versteckt, zum anderen die (abstrahierten) *Berechnungsobjekte*, falls die Diensterbringung bestimmte Eigenschaften (Methoden oder Felder) an die Objekte selbst stellt. Werden im Berechnungsdienst nur bereits vorhandene Systemdatentypen verwendet, so enfällt das Berechnungsobjekt.

- `interface SecureComputationService`
  Dienst zur sicheren Ausführung und Berechnung von Operationen innerhalb einer sicheren Hardware, der die sichere Auswertung geheimer Daten erlaubt.

- `abstract class ComputationObject`
  Abstrakte Klassendefinition für die Implementierung von Objekten, die im `SecureComputationService`-Dienst verwendet werden sollen. Damit werden die auf Seiten eines Objektes zur Berechnung innerhalb der Trusted Hardware erforderlichen Methoden und Felder

festgelegt. Die Implementierung der Methoden in der Hardware und im externen Software-Objekt dürfen sich somit unterscheiden, und der Transfer der Objekte von und zur Karte kann durch Zerlegung in von der SmartCard unterstützte (flache) Datentypen erfolgen.

Damit können durch die Ausführung in der Trusted Hardware sowohl interne Metadaten unbeobachtet und sicher aktualisiert, als auch neue unverschlüsselte (öffentliche) Daten an den Agenten zurückgeliefert werden. Zum Beispiel kann dem Agenten als Ergebnis jeder kritischen Berechnung mitgeteilt werden, ob der vertrauliche Metadaten-Zustand noch konsistent ist bzw. ob die vertrauliche Überprüfung kritischer Prüfdaten auf einen integren Agentenzustand schließen lässt. Dadurch kann der Agent zumindest auf der nächsten gutartigen Plattform seine Mission abbrechen oder entsprechende Maßnahmen ergeifen, z.B. heimkehren und Meldung erstatten.

5.4 Anwendungsnahe E-Commerce-Dienste

Ausgehend vom Dienst zur sicheren Berechnung aus Kapitel 5.3.3 werden hier exemplarisch zwei Ausprägungen vorgestellt, die sich an typischen E-Commerce-Aufgabenstellungen orientieren.

Sicheres Treffen von vertraulichen strategischen Entscheidungen. Auf unsicheren Plattformen sollen Entscheidungen anhand einer geheimzuhaltenden Strategie getroffen werden. Dazu werden sowohl ein verschlüsseltes Strategieobjekt als auch eine Menge von Eingabeparametern an den Dienst weitergereicht, der eine Menge von Ausgabeparametern zurückliefert, deren Zustandekommen aber außerhalb des Einflussbereiches der evtl. bösartigen Plattform liegt.

- `interface SecureStrategyEvaluation`
  Dienst zur sicheren Auswertung von Strategieobjekten durch die unterliegenden, idealerweise auf sicherer Hardware ausgeführten, kryptographischen Operationen.
- `interface Strategy`
  Schnittstellendefinition, der die einzelnen Implementierungen von Strategieobjekten genügen müssen, damit eine Auswertung durch den `SecureStrategyEvaluation`-Dienst erfolgen kann.

Die vertrauenswürdige Auswertung von strategischen Entscheidungen auf unsicheren Rechnerplattformen erfordert die Verfügbarkeit von Trusted Hardware.

Je nach Anwendungsgebiet wird eine entsprechende, aufgabenspezifische Strategie implementiert, die dem definierten `Strategy`-Interface folgt. Anwendungsbeispiele für solche Klassen sind Strategieimplementierungen für elektronische Auktionen (`AuctionBettingStrategy`) oder für Comparison Shopping Systeme (`BuyingStrategy`).

Dienste für den sicheren Umgang mit Angeboten im Comparison Shopping. In bestimmten Fällen ist es durchaus vorteilhaft, sich nur das jeweils beste Angebot zu merken und die anderen zu vergessen. Dies trifft beispielsweise dann zu, wenn die Anzahl der zu sondierenden Angebote insgesamt sehr hoch und das Datenaufkommen pro Angebot sehr groß ist. Wenn alle in solch einem Szenario aktiven Agenten jeweils sämtliche erhaltenen Angebote speicherten, so stellten sie aufgrund ihrer Datengröße bei der Migration selbst eine erhebliche Belastung des vorhandenen Netzes

dar. Ist die Anzahl der Agenten zudem beträchtlich, so dass z.B. tausende zur gleichen Zeit auf einer Plattform aktiv seien, dann würde dort auch die Verfügbarkeit der (limitierten) Resourcen zu einem beträchtlichen Problem.

- `interface SecureBestOfferEvaluation`
  Dienst zur sicheren Auswertung von besten Angeboten, also `BestOffer`-Objekten, durch die unterliegenden, idealerweise auf sicherer Hardware ausgeführten, kryptographischen Operationen.
- `abstract class BestOffer`
  Abstrakte Klasse, die im `SecureBestOfferEvaluation`-Dienst als Typ der Eingabeparameter verwendet wird, und die die eigentlichen angebotstypischen Vergleichsfunktion realisiert.

Sichere Interaktion und Kommunikation. Der Dienst zur sicheren Berechung erlaubt ebenfalls eine sichere Kommunikation und Interaktion im folgenden Sinne: Agenten können Nachrichten und darin eingekapselt Daten austauschen, die auch auf unsicheren Plattformen nicht eingesehen werden können. Durch die sichere Auswertung geheimer Nachrichten können beispielsweise auch geheime Strategien zusammenarbeitender Agenten koordiniert und ohne Einsicht der Wirtsrechner aktualisiert werden. Eine mögliche Realisierung wird in Kapitel 7.3 aufgezeigt.

5.5 Geschützte Reiseroute

Ein weiteres allgemeineres Problem mobiler Agenten stellt der Schutz der Reiseroute und Wegewahl des Agenten dar. Bei einer festen Route soll diese unterwegs nicht manipuliert werden können. Ist die Wegfindung dageben dynamisch, so sollen bereits gewählte Ziele nicht von einem nachfolgenden Rechner unbemerkt entfernt werden können.

- `SecureSeqItinerary`
  Diese Klasse stellt die zustandsgeschützte Version der mit dem IBM ASDK gelieferten Klasse `SecureSeqItinerary` dar, die eine feste Wegsequenz mit auszuführenden Aktionen an den jeweiligen Stationen definiert.
- `SecureSeqPlanItinerary`
  Analog zu `SecureSeqItinerary` ist diese Klasse eine sichere Implementierung von `SecureSeqPlanItinerary`, die eine feste Wegsequenz definiert mit Nachrichten, die beim Erreichen einer neuen Plattform an den Agenten geschickt werden sollen.

Damit ergibt sich z.B. auch die Realisierung einer *Hotlist* von Marktplätzen, die der mobile Agent auf jeden Fall besuchen soll, und die von bösartigen Rechnern nicht unentdeckt modifiziert werden kann. Falls ein Zielknoten zeitweise unerreichbar ist, kann wie folgt reagiert werden:

(a) Der Besuch wird aufgeschoben, d.h. der Zielrechner wird in der Wegeliste z.B. an das Ende gesetzt. Sind keine Knoten mehr erreichbar, so kehrt der Agent zum Ursprungsrechner zurück.

(b) Der Rechner, der einen Ausfall des nächsten Folgeknoten feststellt, sendet eine signierte „Rechner X unerreichbar“ Nachricht an den Agenten, der diese an Stelle eines Angebots speichert und diese Plattform auslässt. Bei vorgetäuschten angeblichen Ausfällen durch einen betrügenden Rechner kann dieser somit anhand der Auftrittshäufigkeit im Vergleich zu den Meldungen anderer Rechner entlarvt werden.

5.6 Weiterführende Dienste

Die oben eingeführten Sicherheitsdienste sind grundlegender Natur. Darauf aufbauend können bei Bedarf verfeinerte Ausprägungen derselben eingeführt werden, wie z.B.

- Klasse `ForwardLinkedAppendOnlyContainer`
  Alle Elemente dieses `AppendOnlyContainer`s enthalten neben der Verkettung mit dem Vorgänger auch einen impliziten Vorwärtsverweis auf das nachfolgende Element.
  Vorteil: Dieser Vorwärtsverweis kann z.B. die Zieladresse der nächsten zu besuchenden Plattform oder ein Hashwert über die nächsten n Zielplattformen auf dem Weg darstellen. Für $n > 1$ können bei dynamischer Wegewahl (mindestens um n Stationen vorausschauend) konspirative Replay-Attacken der in Abbildung 6.1 gezeigten Art erheblich erschwert bzw. verhindert werden. Genauer: Unter der Annahme, dass ein Agent unterwegs jede Plattform nur einmal besucht, müssen mindestens n bösartige Plattformen zusammenarbeiten und konsekutiv besucht werden, um vorwärtsgerichtete Replay-Attacken zu ermöglichen.
  Nachteil: Es stellt sich die Frage, wie man zwischen einer tatsächlich nicht erreichbaren und einer in Folge eines Betrugs als nicht erreichbar deklarierten Plattform unterscheiden kann, und wie man mit solchen Fällen umgeht.

- Klasse `DynamicReadOnlyContainer`
  Der Inhalt dieses Containers ist dynamisch veränderlich und abhängig vom Zustand des Agenten. Zustandsänderungen können durch Dritte nicht direkt hervorgerufen, sondern nur ausgelesen werden. Die Änderungen können durch interne Auswertung der Nutz-, Meta- und Prüfdaten erfolgen, z.B. in Kombination mit einem von `SecureComputation` abgeleiteten Dienst zur sicheren Zustandsänderung.

- Klasse `SelectiveReadOnlyContainer`
  Ein `ReadOnlyContainer`, bei dem Informationen explizit für den jeweiligen Empfänger verschlüsselt wurden. Für jeden Empfänger stehen also entsprechende Lese-Slots zu Verfügung.

- Klasse `WeakAppendOnlyContainer`
  Dieser Container erlaubt nur das Anhängen von Informationen an den bereits vorhandenen Datenbestand. Der Schutz der Daten erfolgt über eine herkömmliche Prüfkette. Eine Implementierung liegt vor.

Denkbar sind auch weiterführende Dienste mit Bezug zum elektronischen Handel, die die folgenden Schnittstellen erfüllen. Für diese Schnittstellen wurden aber zum Teil wegen des begrenzten zeitlichen Rahmens der Diplomarbeit keine Implementierungen angefertigt.

- Interface `CipherLockable`
  Legt den Funktionsumfang fest, den Klassen bereitzustellen haben, deren Inhalt (kryptographisch durch Verschlüsselung) abschließbar sein soll. Die Containerklasse `LockableContainer` implementiert z.B. diese Schnittstelle.

- Interface `TimeLockable`
  Definiert die Schnittstelle für Objekte, die mittels eines passenden signierten Zeitstempels freigeschalten werden. Ziel ist, einen zeitweiligen Schutz der Vertraulichkeit zu erreichen, etwa bis Informationen entweder bezahlt oder veraltet sind (z.B. aktuelle Börsennachrichten). Die Auswertung und der Vergleich der Zeitstempel könnten dabei durch einen sicheren Dienst von einer vertrauenswürdigen dritten Instanz (*Trusted Third Party*, TTP) vorgenommen werden oder zusammen mit dem sicheren Stack aus Kapitel 5.3.1 auf einer von einer dritten Partei ausgegebenen SmartCard realisiert sein. Wurde im Rahmen dieser Arbeit nicht implementiert.

- Interface `TokenLockable`
  Legt die Schnittstelle für Objekte fest, die mittels eines zeitunabhängigen, aber ggf. von einer gewissen Autorität (etwa eine Trusted Third Party) signierten, Tokens freigeschaltet werden. Wurde ebenfalls nicht implementiert.

Anwendungsbeispiele für diese Schnittstellen sind Angebotsklassen, die entweder mit einem Zeitschloss ausgestattet sind und die TimeLockable-Schnittstellendefinition erfüllen, also z.B. eine Klasse `TimeLockedOffer`, bzw. Angebotsobjekte, die über einen Freigabetoken kontrolliert werden, etwa eine Klasse `TokenLockedOffer`, die die Schnittstelle `TokenLockable` implementiert.

5.7 Problematik der Exportrestriktionen

Die strikten Exportrestriktionen für kryptographische Software in Nordamerika laufen dem Gedanken des grenzüberschreitenden Marktes auf dem Internet entgegen, und stellen ein nicht unwesentliches Hindernis für weltweiten elektronischen Handel dar: Oft muss deswegen – insbesondere bei Produkten aus den Vereinigten Staaten oder Canada – zweigleissig gefahren werden: Eine sichere US-Version und eine abgespeckte Version für den Rest der Welt. Dies hat zur Folge, dass auch Software, die weltweit vom selben Hersteller erhältlich ist (z.B. Web-Browser), sich national bezüglich der verwendeten kryptographischen Module u.U. erheblich unterscheidet. Diese Problematik beeinflusste erwartungsgemäß auch die Konzeption und Entwicklung des Comparison-Shopping-Prototyps. Bestimmte Funktionen der Java Cryptography Extension mussten als Stubs realisiert werden, weil wegen des Exportverbots die freie Version nicht verfügbar war und ansonsten nur kommerzielle Versionen erhältlich gewesen sind. Die Exportrestriktionen wirken sich kontraproduktiv auf die Vereinheitlichung von Standards aus und laufen dem Wunsch nach Interoperabilität der Sicherheitsdienste zuwider, da der Ausweg oft in proprietären kryptographischen Lösungen gesucht wird. Das führt letztendlich zu einer Einschränkung des heterogenen Charakters der mobilen Agenten.

Anmerkung: Kurz vor Ablauf der Diplomarbeit wurden Anfang Februar die US-Exportbestimmungen gelockert. Dies konnte bei der Erstellung der Diplomarbeit und der dabei angefertigten Implementierungen nicht mehr berücksichtigt werden.

5.8 Vergleich mit anderen Arbeiten

Wie gezeigt, gibt es bereits mehrere Arbeiten zum Thema Sicherheit von mobilen Agentensystemen, wie beispielsweise die Arbeiten von Hohl über softwarebasierte Blackbox-Sicherheit [Hoh98] oder Sanders und Tschudin zum Thema Encrypted Functions [ST98a]. Dabei handelt es sich aber um Forschungsansätze, die heute noch keine in der Praxis brauchbaren Werkzeuge für die Entwicklung sicherer Agentenanwendungen bereitstellen. Oder es werden konzeptuelle, protokollorientierte Verfahren zum Schutz von Daten mobiler Agenten vorgestellt, wie z.B. bei Karjoth et al. [KAG98].

Andere Arbeiten gehen nur am Rande auf Sicherheitsdienste ein und weisen zum Teil konzeptionelle Mängel auf, wie z.B. bei Karnik (siehe weiter unten), oder erkennen die eigentliche Problematik nicht vollständig und schlagen fehlerhafte oder unpraktikable Lösungen vor, wie z.B. die Veröffentlichung von Corradi et al (siehe den Protokollvergleich in Kapitel 6.8).

Eine ausführliche, klärende Behandlung der Sicherheitsaspekte mobiler Agenten, wie in dieser Arbeit geschehen, ist daher angebracht. Zudem werden zum ersten Mal generische Sicherheitsdienste vorgestellt, die den transparenten Einsatz von unterliegender sicherer Hardware erlauben und wegen ihrer abstrakten Natur direkt in beliebige (Java-basierte) Agentensysteme integriert werden können. Die in Kapitel 7 vorgestellte Implementierung in Java entspricht dem intuitiven Problemverständnis und kann unmittelbar in der Praxis Anwendung finden.

In seiner Diplomarbeit mit dem Titel „Sicherheit von mobilen Agenten auf elektronischen Märkten“ [Man98] stellt Mandry elektronische Märkte in Verbindung mit mobiler Agententechnologie vor. Er diskutiert die Sicherheitsproblematik von Agentensystemen und gibt einen vergleichenden Überblick über die existierenden Lösungsansätze, ohne jedoch konkrete Sicherheitsdienste genauer auszuführen oder zu spezifizieren.

Fünfrocken untersucht den Einsatz von mobilen Agenten durch Integration in die existierende Webtechnologie auch unter Einbeziehung von SmartCards [Fün99], aber nicht mit Blick auf allgemeine Agentensysteme und generische Sicherheitsdienste.

Karnik geht im Rahmen seiner Dissertation zum Thema „Sicherheit von Mobilen-Agenten-Systemen“ [Kar98] in Kapitel 8 nur knapp auf den Schutz der Agenten selbst ein. Er beschränkt sich dabei auf die Einführung dreier sicherer Datentypen, von denen einer eine wesentliche konzeptionelle Schwachstelle aufweist, wie unten gezeigt wird. Außerdem sind die Datentypen sehr speziell und unterstützen nicht die transparente Verwendung sicherer Hardware. Bei den Datenstrukturen handelt es sich die folgenden Klassen:

- Klasse `ReadOnlyContainer` und `AppendOnlyContainer`
  Containerklassen, deren prinzipieller Einsatz semantisch den oben beschriebenen gleichlautenden Datentypen entspricht (vgl. Kapitel 5.3.2). Der Entwurf und die Realisierung entscheiden sich jedoch wesentlich, und es wird weiter gezeigt, dass Karnik’s Ausprägung des `AppendOnlyContainer` **unsicher** ist.
- Klasse `TargetedState`
  Dieser Klasse entspricht bezüglich ihrer Funktionalität der Container `SelectiveReadOnlyContainer`, der in Abschnitt 5.6 vorgestellt wurde.

Karniks Realisierung des `AppendOnlyContainers` weist aber eine *wesentliche Schwäche* auf: Die verwendete Prüfsumme (Feld `checkSum`) zum Wahren der Integrität der Datenstruktur kann von einem Angreifer unentdeckt manipuliert werden. Ein erfolgreicher, nicht-detektierbarer Angriff auf den `AppendOnlyContainers` wird im Anhang C auf Seite 110 vorgeführt.

Kapitel 6

Comparison Shopping Protokoll mit sicherer Hardware

Der hier vorgestellte Ansatz ist ein sicheres Protokoll zum Schutz der Integrität und Geheimhaltung der Daten mobiler Agenten auf unsicheren Agentenplattformen. Er leitet sich in seiner Grundform von dem von Asokan, Gülcü und Karjoth in [KAG98] vorgestellten Protokoll P1 ab. Während das letztere aber vollständig auf softwarebasierte kryptographische Operationen auch auf unsicheren, eventuell bösartigen Agentenplattformen setzt, wird in dieser Ausprägung stattdessen der händlerseitige Einsatz sicherer Hardware angenommen. Das Protokoll eignet sich insbesondere für typische E-Commerce-Systeme wie beispielsweise das Comparison Shopping oder für elektronische Auktionen, da es ein bisher nach Ansicht des Autors unerreichtes Maß an Sicherheit bereitstellt. Durch einfache Modifikation wird erreicht, dass nur auf Seiten der Händler bzw. Läden sichere Hardware benötigt wird, nicht aber auf Seiten der Kunden (oder Systembenutzer), was die Kosten für den Benutzer und den administrativen Aufwand insgesamt in Grenzen hält.

Zuerst wird eine kurze Beschreibung der sicheren Protokolle von Karjoth et al. gegeben, und daraufhin die Verwendung von sicherer Hardware motiviert. Im Anschluss daran folgt die formale Beschreibung des *BlackBox Comparison Shopping Protokolls* und die Diskussion praktischer Aspekte bezüglich des Geschäftsmodells und der Administration der Trusted Hardware. In Kapitel 6.5 werden weitergehende Maßnahmen gegen konspirierende Plattformen und Replay-Angriffe besprochen. Danach werden Protokollvarianten und Weiterentwicklungen vorgestellt, die u.a. die sichere autonome Entscheidungsfindung mobiler Agenten in E-Commerce-Szenarien erlauben. In diesem Zusammenhang steht das Konzept der *Entscheidungsboxen* im Mittelpunkt. Daran anknüpfend folgt die Realisierung nichtabstreitbarer, bindender Kaufentscheidungen als Erweiterung des präsentierten Protokolls. Es folgt der Vergleich des Blackbox-Protokolls mit dem TTP-Protokoll von Corradi et al, und abschließend die Einordnung des Blackbox-Protokolls in eine allgemeine Klassifikation von Sicherheitsdiensten.

6.1 Sichere Protokolle von Karjoth et al.

Karjoth et al. stellen als erste Variante das *Publicly Verifiable Chained Digital Signature* Protokoll vor (P1), das ausgehend von Yee's *Per-Server Digital Signature* Protokoll [Yee97] mit Hilfe einer Hashkette die gesammelten Angebote jeweils mit der Identität des folgenden elektronischen

Ladens verknüpft. Eine weitere Variante ist das verkettete Digitale Signatur Protokoll (*Chained Digital Signature Protocol*, P2) mit vorwärtsgerichtete Vertraulichkeit (*Forward Privacy*), die durch das Vertauschen der Reihenfolge des Signierens und Verschlüsselns von Angeboten erreicht wird. Dadurch wird nicht bekannt, welche Läden zuvor welche Angebote gemacht haben.

Diese beiden Protokolle benötigen eine vorhandene Public-Key Infrastruktur (PKI). Für den Fall, das keine PKI gegeben ist, werden weitere sichere Protokollvarianten besprochen, die u.a. auf den Gebrauch von Hashketten und die Berechnung von *Message Authentication Codes* (MAC) zurückgreifen.

Notation. An dieser Stelle werden die zum Verständnis der Protokollbeschreibungen notwendigen Begriffe geklärt. Die zur formalen Beschreibung der nachfolgenden Protokolle verwendete Terminologie wird in Table 6.1 zusammengefasst, die verwendeten kryptographischen Primitiven werden in Tabelle 6.2 beschrieben. Außerdem gelte, dass jeder Teilnehmer aus einer digitalen Signature $\mathcal{SIG}_{S_i}(m)$ kann m und die Identität von S_i extrahieren kann. Die Verschlüsselung ist probabilistisch und sicher gegen Chosen-Plaintext-Angriffe. Die kryptographische Einwegfunktion ist kollisionsfrei und gibt keinerlei Informationen preis.

Tabelle 6.1: Im Blackbox Protokoll verwendete Notation.

$A = S_0 = S_{n+1}$	Agentenbesitzer (Auftraggeber, Kunde).
$B = B_i, 1 \leq i \leq n$	Sichere Hardware (Blackbox).
$S_i, 1 \leq i \leq n$	Händler (Shop).
$o_i, 1 \leq i \leq n$	Angebot (Objekt) von S_i.
$O_i, 1 \leq i \leq n$	Von S_i signiertes Angebot o_i, mit Verschlüsselung für A.
$\overrightarrow{O}_n$	Die Kette der gesammelten Angebote der Läden $S_1, S_2, \ldots, S_n$.
$\overrightarrow{h}_n$	Die Hashkette $h_1, h_2, \ldots, h_n$ ohne Verankerung h_0 und Abschluss h_{n+1}.
h_0	Initialer (geheimer) Identifikator, von A gewählt.
$h_i, 1 \leq i \leq n+1$	Hashkette zum Integritätsschutz der Angebote $O_1, O_2, \ldots, O_n$.

Tabelle 6.2: Kryptographische Notation.

$\mathcal{PK}_X$	Öffentlicher Public-Key Schlüssel von Teilnehmer X (Entschlüsseln/Verifizieren).
$\mathcal{SK}_X$	Privater Public-Key Schlüssel von Teilnehmer X (Verschlüsseln/Signieren).
$[m]_X$	Mit dem öffentlichen Schlüssel von Partei X verschlüsselte Nachricht m.
$[m]_{X^{-1}}$	Mit dem privaten Schlüssel von Partei X verschlüsselte Nachricht m.
$\mathcal{SIG}_X(m)$	Mit dem privaten Schlüssel $\mathcal{SK}_X$ von Teilnehmer X signierte Nachricht m.
$\mathcal{K}$	Symmetrischer Einmalschlüssel.
$\mathcal{H}(m)$	Kollisionsfreie, kryptographische Einwegfunktion (z.B. SHA-1 [NIS95]).
$Alice \rightarrow Bob$: m	Alice sendet Bob die Nachricht m.
$\overrightarrow{O}_{i-1} \mid\mid O_i$	Um Angebot O_i erweiterte Angebotsliste mit result. Länge i.
$\overrightarrow{h}_{i-1} \mid\mid h_i$	Um Hashwert h_i erweiterte Hashkette mit resultierender Länge i.

Ein Vektor $\overrightarrow{O_n}$ ist eine Folge von n verschlüsselten und signierten Angeboten $o_i,\ 1 \leq i \leq n$.

$$O_i \ := \ [\mathcal{SIG}_{S_i}(o_i)]_A \text{ für } 1 \leq i \leq n \tag{6.1}$$

Eine *Verkettungsrelation* der n Angebote aus $\overrightarrow{O_n}$ wird durch die Berechnung einer rekursiven Folge von Hashwerten bestimmt, wie für den sicheren Stack nach Stubblebine in Abbildung 5.3 gezeigt. Die Vorschrift zur Berechnung der Hashwerte lautet

$$h_{i+1} := \begin{cases} \mathcal{H}(h_0) & \text{für } i = 0 \\ \mathcal{H}(h_i, O_i) & \text{für } 1 \leq i \leq n \end{cases} \tag{6.2}$$

Die Werte h_0 und h_{n+1} werden jeweils in einer ausschließlich durch die sichere Hardware zugänglichen Datenstruktur gehalten, mit n als aktuelle Anzahl gesammelter Angebote. Jedem verschlüsselten und signierten Objekt O_i entspricht also ein Hashwert h_i, der sich aus den vorhergehenden Objekten und Hashwerten bestimmt. Der initiale Hashwert h_0 entspricht der geheimen *Verankerung* der Hashkette, h_{n+1} ist deren *Abschluss*. h_0 wird entweder zufällig durch die Hardware oder explizit vom Erzeuger gewählt, um die Datenstruktur eindeutig zu identifizieren und sie von vorhergegangenen Agentendurchläufen zu unterscheiden; h_{n+1} wird in Abhängigkeit von den gesammelten Nutzdaten (Angeboten) sukzessive berechnet. Der Vektor $\overrightarrow{h_n}$ enthalte alle übrigen, nicht durch die Hardware verwalteten Hashwerte der Hashkette, also

$$\overrightarrow{h_n} := \{h_1, \ldots, h_n\} \tag{6.3}$$

Der Begriffsbildung von [KAG98] folgend ist die Angebotskette *gültig*, wenn die Verkettungsrelation an jedem Kettenglied bis einschließlich O_k hält. Das unerlaubte Entfernen oder Einfügen von Elementen führt zu einer Verletzung der Kettenrelation für alle der Manipulationsstelle nachfolgenden Elemente. Jeder Händler S_i besitzt oder hat Zugriff auf eine Blackbox B_i. Alle Blackboxen sind bezüglich ihrer Funktionalität gleich, d.h. $B_i = B_j$ für $i \neq j$, und sie teilen sich auch dasselbe Public-Key Schlüsselpaar $(\mathcal{PK}_B, \mathcal{SK}_B)$. Sie stellen damit sozusagen eine über das System verteilte sichere Hardware dar.

Vertrauensmodell. Die Aufgabe des Systems ist, Agenten von Rechner zu Rechner wandern zu lassen und jeweils vor Ort unmittelbar Ergebnisse zu erhalten und zu speichern. Die Angebote von vorhergehenden Läden werden dabei geheimgehalten, damit sie nachfolgende Preisauskünfte nicht beeinflussen[1], und werden somit als voneinander unabhängig angesehen. Auch werden keine Aussagen über die Korrektheit und Art der Berechnung der erhaltenen Angebote auf einer besuchten Plattform gemacht. Das primäre Ziel liegt vielmehr darin, die durch den Agenten gesammelten Resultate möglichst gut zu schützen und zu garantieren, dass etwaige Manipulationen und unerlaubte Änderungen in jedem Fall zweifelsfrei erkannt werden.

Grad der Mobilität. Das erwähnte Protokoll berücksichtigt den allgemeinsten Fall von Agenten-Mobilität, d.h. die *uneingeschränkte Beweglichkeit*:
Der mobile Agent kann beliebige Plattformen besuchen; auch solche, die unter Umständen zur Zeit der Indienststellung dem Agentenbesitzer (oder Erzeuger) noch nicht bekannt waren, und die Reihenfolge der Visiten wird frei gewählt. Dagegen reicht im Prototyp ein restriktiverer Mobilitätgrad aus, um die wesentlichen Sicherheitsprobleme aufzuzeigen. Hier wird der Agent vom Agentenbesitzer zum Zeitpunkt der Inbetriebnahme mit einem festen Wegeplan (engl. *itinerary*) versehen.

Das Protokoll bietet die **Lösung folgender Probleme**:

[1] Die Möglichkeit von direkten Preisanfragen durch einen Händler bei anderen Händlern wird dabei nicht in Betracht gezogen.

- *Geheimhaltung von Daten:* Nur der Agentenbesitzer kann ein Angebot o_i eines Ladens S_i entschlüsseln.
- *Nichtabstreitbarkeit:* Der Laden S_i kann sein zuvor gemachtes Angebot o_i nicht abstreiten.
- *Vorwärtsgerichtete Vertraulichkeit:* Keine der Identitäten der Läden, die ein Angebot gemacht haben, kann von Dritten enthüllt werden.
- *Starke vorwärtsgerichtete Integrität:* Keines der geschützten Angebote O_k kann bei Laden S_m manipuliert werden, mit $k < m$.
- *Öffentlich prüfbare vorwärtsgerichtete Integrität:* Jeder kann die Integrität des Angebotes o_i überprüfen, indem er die Hashkette bis O_i nachrechnet und auf Korrektheit testet.
- *Resistenz gegen das unerlaubte Einfügen von Daten:* Kein Angebot kann an Stelle i unerlaubterweise eingefügt werden.
- *Resistenz gegen das unerlaubte Abschneiden von Daten:* Die Kette von gesammelten Angeboten kann nur dann an Stelle i abgeschnitten werden, wenn der Laden S_i mit dem Angreifer konspiriert (siehe Replay-Attacke in 2.3.3).

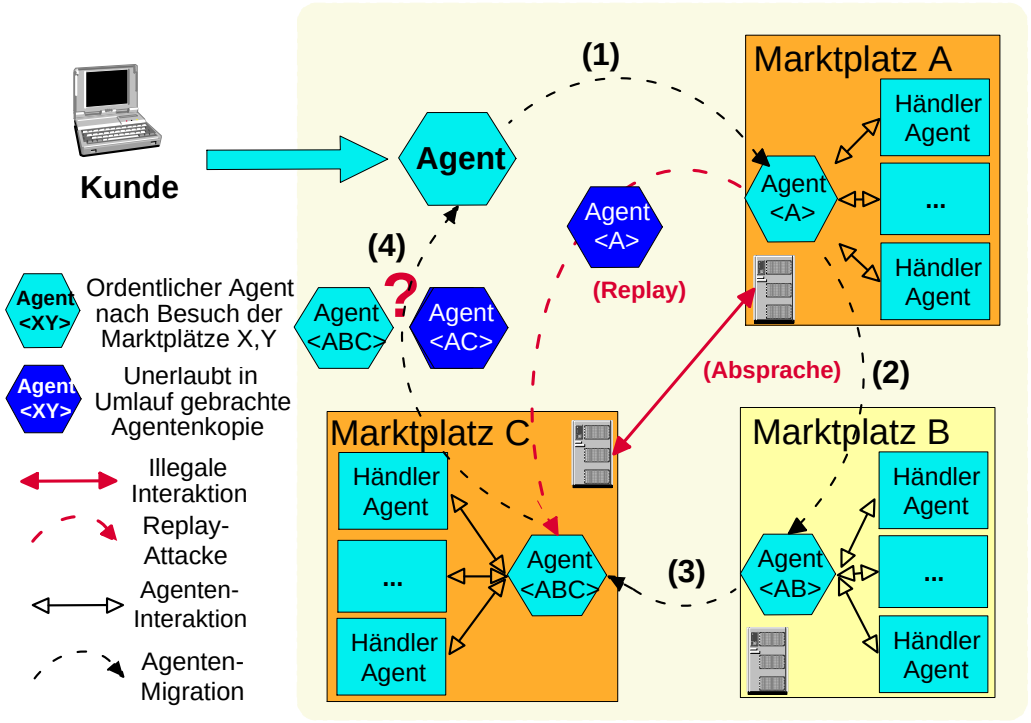


Abbildung 6.1: Grundproblematik von Mobilen-Agenten-Anwendungen in Comparison Shopping Systemen: Replay-Attacke

Sicherheitsbetrachtung Der Agent wird vom Besitzer auf die Reise geschickt, besucht nacheinander die spezifizierten Händler und kehrt anschließend zur Herkunftsplattform zurück. Das Ziel der Angriffe hierbei sind die vom Agenten gesammelten Angebote der konkurrierenden Händler. Das grundlegende Sicherheitsproblem dabei lässt sich durch ein Szenario mit drei zu besuchenden

Marktplätzen skizzieren, wie in Abbildung 6.1 aufgezeichnet. Der Agent erreicht den ersten bösartigen Markplatz A (1), der den Agenten bedient, aber eine Kopie desselben aufbewahrt. Der Agent wandert weiter (2) zum nächsten Rechner (Marktplatz B), und gelangt anschließend zum Marktplatz C. Marktplätze A und C können nun konspirieren, um die Angebote der günstigen Konkurrenten auf Marktplatz B durch einen sogenannten **Replay-Angriff** zu vernichten:

A sendet C die Kopie des Agenten, der den angekommenen Agenten mit Daten aus A und B, Schreibweise <A,B>, damit ersetzt. Somit enthält der Agent am Ende nur die Daten <A,C> statt <A,B,C>. Dieser Angriff kann bei frei beweglichen Agenten nicht verhindert werden, wenn die Schutzmechanismen nur einseitig auf dem Agenten realisiert werden. Dagegen können die folgenden Attacken verhindert oder entdeckt werden: Abschneiden von gültigen Daten oder Ersetzen von gültigen durch gefälschte Daten.

Folgende Aspekte werden dadurch jedoch *nicht gelöst*:

- Die garantiert ungestörte Ausführung wesentlicher Agentenfunktionen (Schutz des Kontrollflusses),
- der geschützte und verborgene Zugriff auf geheime Daten (Geheimnisse des Agenten, z.B. geheimer Schlüssel) und
- das sichere Treffen kritischer Entscheidungen (z.B. Kauf- oder Strategieentscheidungen).

Die Erfüllung dieser Anforderungen ist insbesondere im Bereich des elektronischen Handels eine kritische Angelegenheit, die wesentlichen Einfluss auf das Vertrauen in derartige elektronische Handelssysteme und deren Marktdurchdringung ausübt.

6.2 Vorteile des Einsatzes von Trusted Hardware

Der Einsatz sicherer Hardware hat den Vorteil, durch das Schaffen von „neutralen“ Zonen auf ansonsten unsicheren Plattformen auch die bisher nicht gelösten Fragestellungen in Bezug auf Sicherheit und Vertraulichkeit addressieren zu können:

1. Schutz des Kontrollflusses bei der Ausführung kritischer Operationen im Agenten.
2. Exklusiver Zugriff und sichere Auswertung geheimer Daten[2], die dem Agenten zum Zeitpunkt der Initialisierung oder unterwegs mitgegeben wurden.
3. Erweiterter Schutz der Interaktion von und Kommunikation zwischen Agenten.
4. Nichtabstreitbarkeit von Aktionen und Entscheidungen des Agenten (und der damit evtl. verbundenen Verantwortung/Haftung des Agentenbesitzers.)
5. Erhöhter Schutz für proprietären Maschinencode und Algorithmen.

Es ist bekannt, dass ein strikter verlässlicher Langzeitschutz für Maschinencode oder geheime Daten nicht realisierbar ist, da mittels Brute-Force Angriffen nach gewisser Zeit jede Art von Verschlüsselung gebrochen werden kann [BGW97], zumal sich in der Praxis die üblichen Schlüsselängen aus

[2] Zum Beispiel geheime Einmalschlüssel, die nach Erledigung der Arbeit verworfen werden.

Performanzgründen oft am unteren Ende der empfohlenen, als sicher geltenden Längen bewegen[3]. Solche Angriffe sind jedoch teuer in Hinblick auf Rechenzeit und Resourceneinsatz (Prozessorleistung, Speicheraufwand, Entwicklung von Spezialhardware) und je nach der Stärke des kryptographischen Verfahrens entweder praktisch unmöglich oder bei begrenztem Wert der Informationen nicht rentabel. Ebenso ist es in vielen Fällen ausreichend, einen zeitlich begrenzten Schutz zu gewähren, so etwa beim sicheren Auswerten von exklusiven Börseninformationen und kurzfristigen Prognosen, die nur für einen sehr kurzen Zeitraum einen Wissensvorsprung darstellen und danach wertlos werden.

6.3 Das Blackbox Comparison Shopping Protokoll

In diesem Kapitel wird nun eine Weiterentwicklung des von Karjoth et al. entworfenen Protokolls (Variante P1) vorgestellt, das *Blackbox Comparison Shopping Protokoll*. Durch die Verwendung sicherer Hardware ist es möglich, praktische Lösungsansätze für die ungelösten Probleme aus Kapitel 6.1 vorzustellen. Zuerst werden die Grundzüge des Basis-Protokolls erläutert und die Vorzüge diskutiert. Im Anschluss daran wird ein denkbares Geschäftsmodell skizziert, und die mit dem Einsatz von Trusted Hardware verbundenen Chancen und Erweiterungsmöglichkeiten werden aufgezeigt.

6.3.1 Formale Protokollbeschreibung

Wie bereits erwähnt, wird das Protokoll anhand der Comparison Shopping Problematik vorgestellt. Der Lebenszyklus eines mobilen Agenten kann dabei in vier Phasen eingeteilt werden: (1) *Erzeugung und Initialisierung* durch den Agentenbesitzer, (2) *Migrationsphase,* (3) *Interaktionsphase* und die (4) *Entscheidungsphase und Außerdienststellung* nach Erledigung der Aufgabe und Migration zur letzten Zieladresse (in der Regel die Heimatadresse). Das Zusammenwirken der Phasen wird in Abbildung 6.2 gezeigt.

Der Prozess der Entscheidungsfindung oder Datenauswertung ist nicht Gegenstand dieser Arbeit. Anstöße zur Realisierung autonomer Agentenentscheidungen werden im Kapitel 6.7 gegeben. Gemäß der Klassifikation in Kapitel 5.2 kann der Zustand eines Agenten durch das Tupel $\langle\, \text{Nutzdaten } \mathbf{ND},\ \text{Metadaten } \mathbf{MD},\ \text{Protokolldaten } \mathbf{PD} \,\rangle$ beschrieben werden, mit Blick auf die auftretenden Datenarten. In unserem Agentenbeispiel werden *keine* Metadaten benötigt, da sich der Agent auf das Sammeln von Angeboten beschränkt und keine Metaoperationen (wie z.B. die Auswertung des besten Angebotes) darauf ausführt. Die Sichtung der Daten erfolgt erst nach der Rückkehr des Agenten zu seinem Ursprungsrechner. Deshalb wird in unserem Fall der Agentenzustand – unter Vernachlässigung der Metadaten – durch das Tupel $\langle\, \text{Nutzdaten } \mathbf{ND},\ \text{Protokolldaten } \mathbf{PD} \,\rangle$ bestimmt. Sei n die Anzahl vorhandener, zu schützender Datenobjekte O_i, mit $1 \leq i \leq n$, die zusammen den Nutzdatenanteil **ND** bilden, dann gilt

$$\mathbf{ND_n} = \overrightarrow{O_n} := \{O_1, \ldots, O_n\} \tag{6.4}$$

[3]So wurde beispielsweise lange Zeit der Gebrauch von 512bit Schlüssellänge beim RSA-Verfahren als ausreichend sicher betrachtet und propagiert, wogegen heute nach neuesten Erkenntnissen mind. 768bit empfohlen werden. Shamir behauptet z.B. dass mit Hilfe des von ihm theoretisch entworfenen Twinkle-Geräts [Sha99, Sil99] in der Größenordnung von einigen Tagen auch 512bit RSA-Moduli faktorisiert werden könnten, und unabhängig davon wurde als Teil eines Wettbewerbs eine 512bit RSA-Modulus erfolgreich [Cer99] mit Hilfe eines verbesserten Zahlensieb-Faktorisierungsalgorithmus faktorisiert.

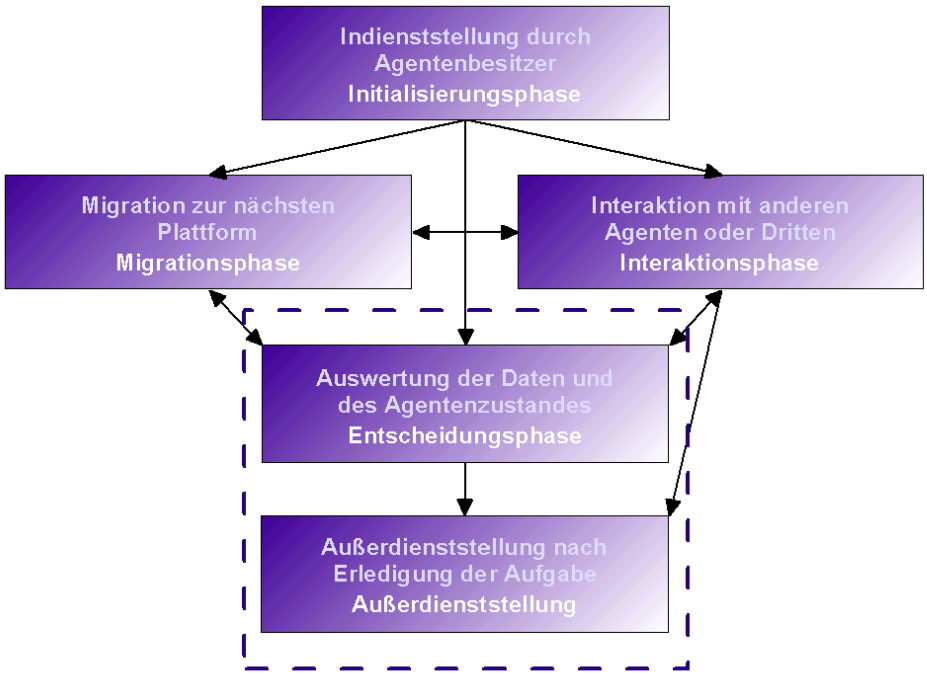


Abbildung 6.2: Lebenszyklus und Phasen mobiler Agenten in Comparison Shopping Systemen. In der Regel fallen die Entscheidungs- und Außerdienststellungsphase zusammen, was durch die gestrichelten Linien angedeutet wird.

Die *kritischen* protokollrelevanten Daten KPD beim Einsatz der Hashkette nach Abbildung 5.3, deren Aktualisierung sicher innerhalb der Blackbox vorgenommen wird, werden im verschlüsselten Zustand mit IDU (vgl. sicherer Stack nach Stubblebine, Kapitel 5.3.1) bezeichnet. Die kritischen Protokolldaten sind gegeben durch

$$\mathbf{KPD_n} = \{h_0,\ h_{n+1}\}, \tag{6.5}$$

und nach Verschlüsselung mit dem öffentlichen Blackboxschlüssel gilt

$$\mathbf{IDU_n} := [KPD_n]_B = [h_0,\ h_{n+1}]_B. \tag{6.6}$$

Die Vertraulichkeit der Daten wird durch Verschlüsselung mit dem öffentlichen Schlüssel der Blackbox gewährleistet. Durch den Einsatz von Trusted Hardware wird sichergestellt, dass die kritischen Protokolldaten nicht unerlaubt manipuliert werden können, die zur Prüfung der Identität des Stacks und der Korrektheit der Hashkette unbedingt notwendig sind. Die *unkritischen* protokollbezogenen Daten UPD, die extern gehalten und eingesehen werden dürfen, sind die n Hashwerte h_i, mit $1 \leq i \leq n$, die in der Protokollbeschreibung mit $\overrightarrow{h_n}$ bezeichnet werden:

$$\mathbf{UPD_n} = \overrightarrow{h_n} := \{h_1, \ldots, h_n\} \tag{6.7}$$

Eine unerlaubte Manipulation dieser Daten wird immer entdeckt, solange die kritischen Protokolldaten KPD unversehrt sind. Damit ergibt sich der Protokolldatenanteil PD zu

$$\begin{aligned} \mathbf{PD_n} &:= \mathbf{KPD_n} \bigcup \mathbf{UPD_n} \\ &= \{h_0, h_{n+1}\} \bigcup \{h_1, \ldots, h_n\} \\ &= \{h_0, \ldots, h_{n+1}\} \end{aligned} \tag{6.8}$$

Der Protokollablauf ist durch folgende formalisierten Schritte festgelegt:

1. **Initalisierung des Agenten bei S_0:**
   Der Agentenbesitzer wählt den geheimen Wert h_0 (Gl. 6.9) und berechnet daraus h_1 (Gl. 6.10). h_1 ist der erste extern gehaltene Hashwert der Hashkette und dient gleichzeitig als Identifikationsmerkmal (*Label*), das die Angebotskette eindeutig dem Agenten zuordnet und mit h_{label} bezeichnet wird:

$$\begin{aligned} h_0 &:= \text{initialer Zufallswert} && (6.9)\\ h_1 &:= h_{\text{label}} = \bar{\mathcal{H}}(h_0) && (6.10)\\ \overrightarrow{O_0} &:= \emptyset \text{ (leere Menge)} && (6.11)\\ \overrightarrow{h_0} &:= \emptyset && (6.12)\\ \text{IDU}_0 &:= [h_0,\ h_1]_B && (6.13) \end{aligned}$$

2. **Migration von Plattform S_{i-1} zu Händler S_i:**

$$\begin{aligned} S_i \longrightarrow S_{i+1}: \quad & \text{IDU}_\text{i},\ \overrightarrow{h_i},\ \overrightarrow{O_i} \\ = \ & [h_0,\ h_{i+1}]_B,\ \{h_1,\ldots,h_i\},\ \{O_1,\ldots,O_i\} \end{aligned}$$

3. **Interaktion mit Händler S_i, $i \geq 1$:**

$$\begin{aligned} S_i \longrightarrow \text{Blackbox}: \quad & O_i,\ \text{IDU}_\text{i} = [h_0,\ h_i]_B \\ \text{Blackbox} \longrightarrow S_i: \quad & \text{IDU}_{\text{i+1}} = [h_0,\ h_{i+1}]_B \text{ mit } h_{i+1} := \mathcal{H}(h_i, O_i) \\ S_i \text{ berechnet}: \quad & \overrightarrow{O_i} := \overrightarrow{O_{i-1}} \mid\mid O_i \text{ und} \\ & \overrightarrow{h_i} := \overrightarrow{h_{i-1}} \mid\mid h_i \text{ mit } h_i := \mathcal{H}(h_{i-1}, O_{i-1}) \end{aligned}$$

4. **Heimkehr zum Agentenbesitzer A (Ursprungsplattform S_0):**
   Sei n die Anzahl der gesammelten Angebote. Die Integrität der gesammelten Daten $\overrightarrow{h_i}$ wird durch Nachrechnen der Hashkette verifiziert. Auf der Karte werden nur der Anfangswert (Verankerung bzw. Label) und der Endwert (Abschluss) der Hashkette auf der sicheren Hardware überprüft. Dazu wird die verschlüsselte IDU-Datenstruktur kartenintern entschlüsselt, d.h. deren Inhalt liegt nur innerhalb der Blackbox im Klartext vor. Der verbleibende Teil der Hashkette, gegeben durch $\overrightarrow{h_i}$, kann im Hauptspeicher des Hosts gemäß der Vorschrift in Gleichung 6.2 extern nachgerechnet werden.

$$\begin{aligned} S_0 \longrightarrow \text{Blackbox}: \quad & O_n,\ h_1,\ h_n,\ \text{IDU}_\text{n} \\ \text{Blackbox verifiziert}: \quad & \text{Label u. Verankerung } h_1 = \mathcal{H}(h_0) \text{ sowie} \\ & \text{Abschluss der Hashkette } h_{n+1} = \mathcal{H}(h_n, O_n) \\ \text{Blackbox} \longrightarrow S_0: \quad & \texttt{Verifikationsergebnis} \\ \text{Letzter Schritt erfolgreich}: S_0 \text{ verifiziert}: \quad & h_{i+1} = \mathcal{H}(h_i, O_i) \text{ für } 1 \leq i < n \end{aligned}$$

6.3.2 Sicherheitsbetrachtung

Die Sicherheit der Hashkette beruht auf dem strikten Schutz der kritischen Protokolldaten KPD, die immer nur innerhalb der Blackbox entschlüsselt, ausgewertet und aktualisiert werden. Der vorgehaltene Abschluss h_{n+1} verhindert die unentdeckte Manipulation (Entfernen, Hinzufügen, Verändern)

von Datenobjekten: Die Neuberechnung der Hashkette ergibt im Falle eines verändernden Eingriffes aufgrund der Kollisionsfreiheit der kryptographischen Hashfunktion einen von h_{n+1} verschiedenen Endwert $\widehat{h_{n+1}}$, so dass gilt: $\widehat{h_{n+1}} \neq h_{n+1}$. Als eindeutiger Identifikator der Hashkette dient der Hashwert der geheimen Kettenverankerung h_0: $h_{label} := \mathrm{Hash}(h_0)$, der öffentlich bekanntgemacht wird. Es ist offensichtlich, dass h_1 und h_{label} identisch sind. Die nur dem Erzeuger der Datenstruktur bekannte geheime Verankerung h_0 verhindert, dass die Hashkette durch eine andere konsistente Hashkette mit anderen Datenobjekten und anderer Verankerung ersetzt wird.

Anderson und Kuhn [AK96] machen darauf aufmerksam, dass eine Reihe von Angriffen bekannt ist, um sichere Hardware und gesicherte Daten rückwärts zu entwickeln (engl. *reverse engineering*), zu manipulieren oder auch nur auszuspionieren.

Im vorgestellten Comparison Shopping Protokoll ist es daher außerordentlich wichtig, dass

- die sichere Hardware sorgfältig gegen physikalische Manipulation geschützt wird,
- die kritischen Teile der Trusted Hardware regelmäßig ersetzt und inspiziert werden (z.B. Austausch des geheimen privaten Blackbox-Schlüssels),
- alle teilnehmenden Parteien ihre Rechte und Verantwortungen aktiv anerkennen und sich auf die Teilnahmebedingungen und die Einhaltung des Protokolls verpflichten, und dass
- Missbrauch, Betrug und Manipulation konsequent verfolgt werden.

6.4 Realisierungsaspekte und Geschäftsmodell

Im Blackbox-Szenario stellt sich die Frage, wer für die Ausgabe und Verwaltung der sicheren Hardware verantwortlich zeichnet. Aus diesem Grunde wird hier eine dritte, vertrauenswürdige Partei eingeführt, die als *Blackbox–Behörde* bezeichnet werden soll. Diese Organisation definiert und überwacht die Teilnahmebedingungen im Comparison Shopping Szenario und übernimmt die administrativen Aufgaben und Pflichten. Sie kann beispielsweise aus einem Zusammenschluss von Händlern bestehen, die einen Händlering formen und mit anderen Händlerringen in Konkurrenz treten. Die vertrauenswürdige Blackbox–Behörde ist wesentlicher Bestandteil eines Comparison Shopping *Geschäftsmodells*, das im Folgenden skizziert und erläutert wird.

Alle Händler, die einen elektronischen Laden für Comparison Shopping Agenten anbieten wollen, treten der vertrauenswürdigen Blackbox–Behörde bei.

Blackbox–Behörde. Die Blackbox–Behörde gibt die sichere Blackbox-Hardware aus und ist zuständig für deren Verwaltung und regelmäßige Wartung, wozu unter anderem der regelmäßige Austausch des geheimen Public-Key Schlüsselpaares gehört, das sich alle Blackboxen teilen. Die Menge der Blackboxen stellt somit quasi eine einzige, auf die Händler physikalisch verteilte sichere Hardware dar.

Verteilung der Blackbox-Schlüssel Der öffentliche Blackbox-Schlüssel $\mathcal{PK}_B$ ist frei verfügbar, sowohl für Händler als auch für Kunden (Agentenbesitzer). Der private Blackbox-Schlüssel $\mathcal{SK}_B$ wird den registrierten Händlern in Form einer Hardware-Karte (z.B. SmartCard) zum Einbau in die Blackbox zur Verfügung gestellt. Der Schlüssel ist geheim und für die Teilnehmer im Comparison

Shopping Modell nicht zugänglich. Sowohl die Blackbox als auch die Schlüsselkarte bleiben im Besitz der Blackbox–Behörde und werden dem Händler nur für die Dauer seiner Mitgliedschaft ausgehändigt.

Schutz des privaten Blackbox-Schlüssels. Die Schlüsselkarten werden durch die Blackbox-Behörde in regelmäßigen Zeitabständen ausgewechselt. Alte, ungültige Schlüsselkarten müssen von den Händlern zurückgegeben werden. Bei dieser Gelegenheit werden die Karten auf Spuren physikalischer Einwirkung hin untersucht und die Funktionsfähigkeit wird getestet. Manipulationsversuche und Betrug stellen Vergehen dar, die gerichtlich verfolgt und geahndet werden. Mit der Teilnahme am Comparison Shopping System verpflichtet sich der Händler dazu, keine Veränderungen an der sicheren Hardware vorzunehmen, und erkennt etwaige Strafen und Bußgelder im Falle der Vertragsverletzung als rechtmäßig an. Die Ersetzung der Schlüsselkarten ist notwendig, um kryptographischen Angriffen gegen den geheimen privaten Blackbox-Schlüssel vorzubeugen. Dies gewährt ein Höchstmaß an Sicherheit, da bei entsprechend großen Schlüssellängen und regelmäßigen Schlüsseländerungen kryptoanalytische Angriffe oder vollständige Suchen nicht praktikabel sind. Physikalische Attacken gegen die Blackbox-Hardware selbst können nicht verhindert werden – befinden sich die Geräte doch in fremder Hand – und stellen die größte Gefahr dar. Wir nehmen jedoch an, dass etwaige physikalische Veränderungen an den Karten irreversibler Natur sind, und folglich beim Austausch der Karten das Vergehen detektiert und bestraft wird. Der Fall ist vergleichbar mit Stromverbrauchszählern, die in privaten Haushalten installiert sind und dennoch Eigentum der Energieversorgungsunternehmen bleiben. Manipulationen sind möglich, werden aber bei Routineuntersuchungen durch die Betreiberfirma entdeckt und geahndet, was zur Abschreckung von Nachahmungstätern in der Regel hohe Bußgelder nach sich zieht, die den Betrug unrentabel werden lassen.

Die *Detektion physikalischer Manipulationen* an den sicheren Geräten (Schlüsselkarten oder Blackbox) ist nicht Gegenstand dieser Arbeit. Ausführliche Informationen hierzu finden sich z.B. in [AK96] oder [Fün99].

Vorstellbare Aufgaben einer Blackbox–Behörde:

1. *Verwaltung und Distribution der Blackbox-Schlüssel:*

   (a) *Auswirkungen für den Kunden:*
   Nachdem ein Kunde (Agentenbesitzer) bei seiner Anmeldung den gültigen öffentlichen Blackbox-Schlüssel erhalten hat, ist der Austausch mit dem neuesten Schlüssel später einfach zu erreichen: Die *Blackbox–Behörde* sendet allen Kunden und Händlern den neuen öffentlichen Blackbox-Schlüssel $\mathcal{PK}_{B'}$ zu, der zuvor mit dem alten privaten Schlüssel $\mathcal{SK}_B$ unterschrieben wurde. Der neue Schlüssel wird durch Prüfen der Signatur $\mathcal{SIG}_B(\mathcal{PK}_{B'})$ authentifiziert. Anschließend wird der alte durch den neuen öffentlichen Schlüssel ersetzt.

   (b) *Auswirkungen auf der Händlerseite:*
   Zusätzlich zum neuen öffentlichen Schlüssel erhalten alle Händler gegen Rückgabe der alten Schlüsselkarte die neuen Karte mit dem privaten Blackbox-Schlüssel. Die alte Karte wird durch die Blackbox–Behörde überprüft und der Händler bei nachgewiesenem Fehlverhalten belangt.

2. *Schlichtungskommission bei Streitfällen:*
   Die im Comparison Shopping System teilnehmenden Parteien sollten sich in Streitfällen zuerst an die *Blackbox–Behörde* wenden, der dann die Rolle einer Schlichtungskommission zufällt. Sie wird dazu mit der Befugnis ausgestattet, Ermahnungen oder Ausweisungen aus dem Handelssystem auszusprechen.

3. *Rechtmäßige Autorität bei gerichtlichen Streitigkeiten:*
   Die *Blackbox–Behörde* wird in Betrugsfällen und bei Gesetzesverstößen als öffentlicher Ankläger fungieren. Sie schützt die Rechte und Interessen ihrer rechtschaffenen Kunden und Händler.

4. *Dienstleistungsgarantiegeber:*
   Eine weitere Aufgabe ist die Gewährleistung und Überwachung der Einhaltung von angebotenen und vereinbarten Dienstleistungen.

5. *Benutzerunterstützung und Support:*
   Die Blackbox-Behörde steht den Teilnehmern bei technischen Schwierigkeiten oder Problemen beratend und helfend zur Seite.

Finanzierung. Wird den Kunden der Zugang zum Händlerring über ein Zugangsportal gewährt – wie in Abb. 3.3 auf Seite 23 gezeigt – das nach Abschluss der preisvergleichenden Recherche die Möglichkeit zum Kauf des günstigsten Produktes anbietet, so führt der betroffene Händler eine bestimmte Gebühr für die erfolgte Transaktion an die Blackbox–Behörde ab. Oder es wird generell für jede elektronische Bestellung ein prozentualer Anteil des Warenwerts an die Organisation abgeführt, falls spätere Online-Bestellungen nicht eindeutig den besuchenden Agenten zugeordnet werden können.

6.5 Maßnahmen gegen konspirierende Plattformen und Replay-Angriffe

Wenn zwei oder mehr bösartige Plattformen miteinander konspirieren, so können sie die zwischen dem ersten und dem letzten Konspirateur gesammelten Daten kontrollieren. Das gleiche trifft für den Fall zu, dass ein Agent dieselbe Plattform (z.B. denselben Händler) zweimal besucht. Eine *Konspiration* ist gegeben, wenn durch das gezielte Austauschen von Daten die unentdeckte Manipulation der Agentendaten ermöglicht wird oder ansonsten nicht zugängliche Informationen aufgedeckt werden. Solche Angriffe sind bei Mobilen Agenten jedoch nur schwer zu verhindern, insbesondere, wenn der Agent alleiniger Träger von Prüfinformationen ist. In diesem Fall gibt es keine Möglichkeit, Attacken nach Abbildung 6.1 (Replay-Angriff) zu erkennen, geschweige denn zu verhindern. Mit Hilfe einiger einfacher Mittel kann die Attraktivität oder gar die Durchführbarkeit solcher Angriffe vermindert werden. Dazu gehören:

A. *Favoritenliste*: Verwalten einer Favoritenliste (engl. *hotlist*) von Läden, die entweder

   (a) über einen hohen Bekanntheitsgrad verfügen[4] und eine große Produktauswahl anbieten und/oder Produkte zu einem bekanntlich guten Preis verkaufen, oder die

[4] D.h. ein hoher Bekanntheitsgrad in der Internetgemeinschaft, wie z.B. bei großen Bücherläden oder Diensterbringern (*Service Provider*).

(b) in der nahen Vergangenheit bei vergleichbaren Einkäufen sehr gute Angebote vorweisen konnten (≈ „Beste Läden“-Cachespeicher).

Eine Auswahl solcher Läden wird dem unveränderlichen Metadatenbestand des Agenten hinzugefügt mit der Bestimmung, von diesen Lokationen auf jeden Fall Angebote einzuholen. Damit wird eine gewisse *Mindestqualität* bezüglich der Produkteigenschaften (i.a. der Preis) gewährleistet, da diese Angebote nicht unbemerkt durch Replay-Attacken entfernt werden können, und das verbleibende Restrisiko wird kalkulierbarer.

B. *Bestätigungstoken*: Jeder besuchte Händler sendet nach Abgabe eines Angebotes einen minimalen Token der Art ⟨*Agentenkennung, Shopkennung, Zeitstempel*⟩ an den Agentenbesitzer (oder an eine stellvertretende Partei)[5]. Diese Token können entweder vorübergehend aufbewahrt oder direkt verarbeitet werden, etwa um eine Prüfsumme zu berechnen, die bei der späteren Auswertung der gesammelten Angebote verifiziert wird. Eine solche Vorgehensweise ist jedoch nur dann sinnvoll, wenn das zu sammelnde Datenaufkommen substantiell merklich größer ist als die Token selbst.

C. *Wegprotokollierung*: Beim Besuch einer neuen Plattform meldet sich der Agent bei einem Wegüberwachungsserver (*Agent Itinery Watchdog*) mittels Nachrichten der Art „Hallo von Agent $AgentenID$ auf Plattform $HostID$ zum Zeitpunkt $Zeitstempel$“. Entweder werden diese Meldungen bei Gelegenheit zum Agentenbesitzer zur Auswertung weitergeleitet, oder sie stehen unterwegs anderen Plattformen zur Verfügung, die damit (stichprobenartig) den Reisezustand der passierenden Agenten verifizieren können. Im Falle von erkannten unerlaubten Datenersetzungen kann entweder

(a) direkt der Agentenbesitzer benachrichtigt, oder

(b) das Vorkommnis einem sogenannten Reputationsserver (engl. *reputation server*) gemeldet werden, der bei häufigeren Klagen über Fehlverhalten bösartige Rechner auf eine „rote Liste“ setzt[6] und Meldung an eine übergeordnete Behörde gibt.

Der Agent selbst könnte mit einer solchen „roten Liste“ der zu ignorierenden Händler versehen werden. Dies erhöht jedoch das Metadatenaufkommen und verschlechtert die Skalierbarkeit bei gesteigerter Komplexität des Einkaufsagentensystems.

D. *n-stufige Vorwärtsverzeigerung*: Soll eine große Anzahl von Plattformen besucht werden, von denen immer mindestens die nächsten n im voraus bekannt sind, so bietet sich eine n-fache Vorwärtsverzeigerung der Daten an. D.h. die Information der noch zu besuchenden Lokationen werden mit den Nutzdaten zusammen im Agenten abgelegt und verschlüsselt. Erfolgreiche Replay-Attacken bedingen dann den Besuch von mindestens n konspirierenden Plattformen in Folge.

Verbesserungen der Sicherheit werden auch durch *fehlertolerante Ansätze* erreicht, bei denen Agenteninstanzen vervielfacht und abschnittsweise parallel ausgeführt werden. An den Verschmelzungspunkten können integritätsverletzende Veränderungen entdeckt werden unter der Annahme, dass zumindest ein Agent eine unterschiedliche Route gewählt hat und nicht korrumpiert wurde. Damit

[5] Alternativ können diese Token an einen unabhängigen Proxy-Server gesendet werden, bei dem der Agentenbesitzer registriert ist; der Proxy leitet die Nachrichten entsprechend weiter und verbirgt damit die Herkunft des Agenten, falls der Agentenbesitzer anonym bleiben möchte.

[6] Eine denkbare Sanktion ist die Streichung des Händlereintrags aus dem Händlerring oder aus dem öffentlichen Register.

wird die Sicherheit jedoch nur vage erhöht, da die Manipulation *aller* Agenten nicht auszuschließen ist.

Es zeigt sich, dass zur Zeit noch *keine zufriedenstellende Lösung* existiert, die Ersetzung eines Agenten durch einen zuvor gemachten Schnappschuss stets zu entdecken.

Es ist ebenfalls ersichtlich, dass ein solcher Schutz zusätzliche Entitäten oder Dienste erforderlich macht, wie z.B. Reputationsserver oder Watchdog-Server. Damit steigen aber auch die administrativen Kosten und die dem Agentensystem inhärente Komplexität mit an.

6.6 Protokollvarianten

Es gibt eine Reihe von Modifikationen, die das oben vorgestellte Protokoll je nach Aufgabe und Intension zurechtschneiden oder erweitern:

1. Der Agentenbesitzer hat keine Blackbox und benutzt symmetrische Verschlüsselung.
2. Der Agent hält immer nur das jeweils beste Angebot in seinem Datenspeicher.
3. Protokollerweiterung zur Realisierung sicherer und autonomer Agentenentscheidungen.
4. Der Agent trifft nichtabstreitbare Kaufentscheidungen.

6.6.1 Agentenbesitzer ohne Blackbox und Public-Key-Schlüsselpaar

Falls der Agentenbesitzer über keine eigenen Public-Key-Schlüssel verfügt und er auf keine Blackbox angewiesen sein soll, so müssen ihm die kritischen Kontrolldaten (IDU) auf anderem Wege zugänglich gemacht werden, damit er den unversehrten Zustand der gesammelten Daten verifizieren kann. Dies geschieht, indem der Abschluss der Hashkette h_{n+1} zusammen mit dem gewählten Anfangswert h_0 als Identifikator nach einer Blackboxoperation mit einem geheimen, mit der Blackbox geteilten Schlüssel $\mathcal{K}$ verschlüsselt wird. Die resultierende Dateneinheit soll mit ADU (*Agent-owner Data Unit*) bezeichnet werden:

$$\mathbf{ADU_n} := [h_0,\ h_{n+1}]_{\mathcal{K}} \tag{6.14}$$

Der Agentenbesitzer wählt deshalb also neben dem geheimen Wert h_0 (Gl. 6.9) auch einen geheimen symmetrischen Einmalschlüssel $\mathcal{K}$. Die kritischen integritätsschützenden Kontrolldaten KPD werden dann neu definiert zu

$$\mathbf{KPD_n} := \{h_0,\ h_{n+1},\ \mathcal{K}\} \tag{6.15}$$

Das heißt, es wird zusätzlich der geheime Einmalschlüssel $\mathcal{K}$ mit aufgenommen. Damit ändert sich die IDU ebenfalls zu

$$\mathbf{IDU_n} := [h_0,\ h_{n+1},\ \mathcal{K}]_B \tag{6.16}$$

Die Verknüpfung der Hashwerte mit den Angeboten und die Realisierung der Hashkette mit dem sicheren Kontrolldatenteil wird in Abbildung 6.3 dargestellt.

Anpassungen am Protokollablauf:

1. **Initalisierung des Agenten bei S_0:**

   Folgende Festlegungen weichen von der in Kapitel 6.3.1 auf Seite 54 beschriebenen Intialisierungsphase ab:

$$\mathrm{IDU}_0 := [\mathrm{KPD}_0]_B = [h_0,\ h_1,\ \mathcal{K}]_B \tag{6.17}$$
$$\mathrm{ADU}_0 := [h_0,\ h_1]_{\mathcal{K}} \tag{6.18}$$

2. **Migration von Plattform S_i zu Händler S_{i+1}:**

   Die Migrationsphase ändert sich durch Hinzunahme der ADU-Struktur ebenfalls:

$$\begin{aligned} S_i \longrightarrow S_{i+1}: \quad & \mathrm{IDU_i},\ \mathrm{ADU_i},\ \overrightarrow{h_i},\ \overrightarrow{O_i} \\ = \ & [h_0,\ h_{i+1},\ \mathcal{K}]_B,\ [h_0,\ h_{i+1}]_{\mathcal{K}},\ \{h_1,\ldots,h_i\},\ \{O_1,\ldots,O_i\} \end{aligned}$$

3. **Interaktion mit Händler S_i, $i \geq 1$:**

   Die Blackbox erzeugt bei jeder Anpassung der IDU-Struktur auch eine aktualisierte ADU-Struktur und gibt beide Datenstrukturen an den Host zurück:

$$\begin{aligned} S_i \longrightarrow \mathrm{Blackbox}: \quad & O_i,\ \mathrm{IDU_i} \\ \mathrm{Blackbox} \longrightarrow S_i: \quad & \mathrm{IDU_{i+1}\ und\ ADU_{i+1}} \\ S_i: \quad & \overrightarrow{O_i} := \overrightarrow{O_{i-1}} \mid\mid O_i \ \text{und} \\ & \overrightarrow{h_i} := \overrightarrow{h_{i-1}} \mid\mid h_i \end{aligned}$$

4. **Heimkehr zum Agentenbesitzer A (Ursprungsplattform S_0):**

   Sei n wiederum die Anzahl der gesammelten Angebote. Mit Kenntnis des selbstgewählten Schlüssels $\mathcal{K}$ und dem Startwert h_0 kann der Agentenbesitzer die gesamte Hashkette selber überprüfen, da er neben $\overrightarrow{h_i}$ auch den Abschlusswert $h_n + 1$ mit der ADU-Struktur erhält:

$$S_0 \text{ verifiziert}: \quad h_{i+1} = \mathcal{H}(h_i, O_i) \text{ für } 0 \leq i \leq n$$

   Um sicherzustellen, dass die richtige ADU-Struktur erhalten wurde und nicht eine Kopie eines früheren Durchlaufs, kann der Agentenbesitzer den h_0-Startwert als Identifikator auf Gleichheit mit dem ursprünglich gewählten und daher bekannten Wert testen.

Auswirkungen auf die Sicherheit. Mit den vorgeschlagenen Modifikationen werden die grundlegenden Protokolleigenschaften nicht berührt, da nur die sicherheitsrelevanten Daten zusätzlich für den Agentenbesitzer in verschlüsselter Form bereitgestellt werden, damit dieser am Ende die Datenverifikation ohne sichere Hardware vornehmen kann.

6.6.2 Nur das beste Angebot speichern

Die meisten Kunden interessieren sich lediglich für das beste verfügbare Angebot. Deshalb ist es in diesem Fall ausreichend, dass der Agent nur das jeweils günstigste Angebot mit sich führt.

Auf der anderen Seite erschwert dies die Entscheidung, ob der Agent die Angebotsrecherche erfolgreich ausgeführt oder ob eine Art von Manipulation stattgefunden hat. Die Gefahr und Bedeutung von *Replay-Attacken* ist nun größer, da keine Vergleichsangebote von anderen Händlern zu

PSfrag replacements

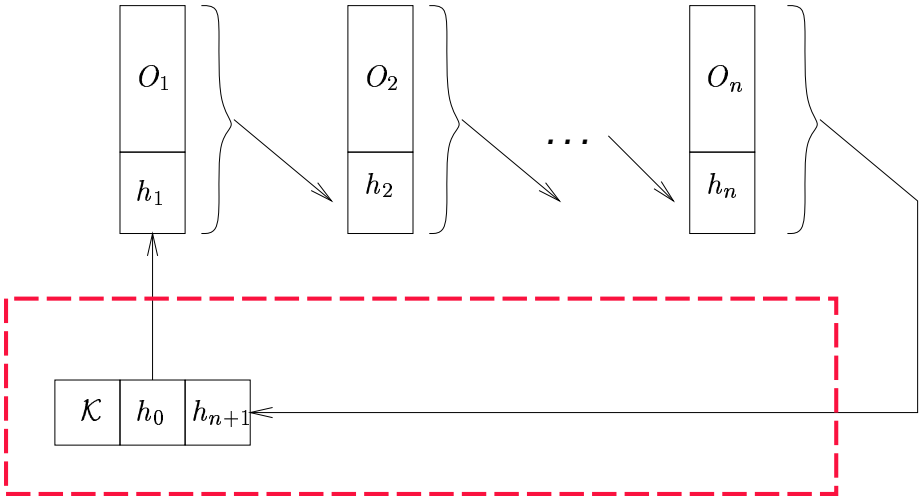


Abbildung 6.3: Sichere Implementierung des Integritätsschutzes für die Angebote o_1 bis o_n durch die Hashkette h_0 bis h_{n+1}, deren Anfang und Abschluss in einer besonderen, nur in der sicheren Hardware zugänglichen Datenstruktur (IDU) geschützt werden.

Verfügung stehen. Ein zusätzliches neues Problem ist die Frage nach der *Qualität* des gefundenen besten Angebotes. Bisher konnte der Agentenbesitzer sich die Palette der Angebote vor Augen führen und für sich selbst entscheiden, ob ihm das beste Angebot daraus gut genug erscheint. Wie kann jetzt sichergestellt werden, dass der Agent tatsächlich das beste Angebot aus einer Menge von echten alternativen Angeboten ausgewählt hat, und nicht aus einer künstlich erzeugten Menge von schlechteren „Mondangeboten"?

Falls daher die mit einem Angebot verbundene Datenmenge klein genug und unkritisch im Hinblick auf Netzwerk- und Rechnerresourcenverbrauch ist, so sollte das herkömmliche Blackbox-Protokoll vorgezogen werden. Ist es jedoch aus irgendeinem Grund erforderlich, die Anzahl der zu speichernden Angebote im Agenten minimal zu halten, etwa wegen einer sehr umfangreichen Datenmenge für einen bestimmten Typ von Angeboten, so kann das ursprüngliche Protokoll wie folgt angepasst werden:

1. *Angebote*: Der Vektor von Angeboten $\overrightarrow{O}_i$ wird durch das beste und von S_{best} signierte, aber unverschlüsselte Angebot $\mathcal{SIG}_{S_{best}}(o_{\text{best}})$ ersetzt.

2. *Hashkette*: Statt der Hashkette $\overrightarrow{h}_i$ wird nun das beste Angebot O_{best} durch eine einfache Prüfsumme h_{best} geschützt, die durch die sichere Hardware berechnet und gesichert wird.

3. Das gerade beste Angebot wird durch die Blackbox mit einem vom Agentenbesitzer gewählten geheimen Schlüssel $\mathcal{K}$ chiffriert.

Eine formale Beschreibung des *BestOffer-Blackbox-Protokolls*:

$$
\begin{aligned}
S_{i-1} \longrightarrow S_i: &\quad [(\mathcal{K}, h_0, h_{best})]_B, [(h_0, \mathcal{SIG}_{S_{best}}(o_{best}))]_{\mathcal{K}} \\
S_i \longrightarrow \text{Blackbox}: &\quad o_i, [(\mathcal{K}, h_0, h_{best})]_B, [(h_0, \mathcal{SIG}_{S_{best}}(o_{best}))]_{\mathcal{K}} \\
\text{Blackbox} \longrightarrow S_i: &\quad [(\mathcal{K}, h_0, h_{newbest})]_B, [(h_0, \mathcal{SIG}_{S_{newbest}}(o_{newbest}))]_{\mathcal{K}} \\
S_i \longrightarrow S_{i+1}: &\quad [(\mathcal{K}, h_0, h_{newbest})]_B, [(h_0, \mathcal{SIG}_{S_{newbest}}(o_{newbest}))]_{\mathcal{K}}
\end{aligned}
$$

Alle Blackboxen teilen sich jeweils ein gemeinsames Public-Key-Schlüsselpaar. Außerdem wird jede Blackbox vor der erstmaligen Benutzung durch den das Gerät besitzenden Händler S_i mit dessen geheimen Schlüssel *personalisiert*, damit das Signieren von Angeboten ebenfalls durch die Blackbox *on-the-fly* erledigt werden kann. Aus einer Signatur lässt sich das signierte Objekt sowie die Identität des Unterzeichners extrahieren. o_i ist das unverschlüsselte und nicht signierte Angebot von Händler S_i. $o_{newbest}$ ist das neue beste Angebot; der Händler, der es ausgegeben hat, wird mit $S_{newbest}$ bezeichnet, und $h_{newbest}$ ist der zugehörige Hashwert. Die verwendeten Begriffe folgen ansonsten der Notation wie in Kapitel 6.1 vorgestellt. Es sollten auch folgende Punkte berücksichtigt werden:

1. Der *Ciphertext* des aufbewahrten besten Angebotes sollte sich *bei jedem Angebotsvergleich* unabhängig vom Ergebnis ändern (z.B. durch probabilistische Verschlüssung des Vergleichssiegers), um keine Rückschlüsse auf den Entscheidungsprozess und dessen Resultat zu erlauben.

2. Der *Vergleich der Angebote* findet *innerhalb der Blackbox* statt, um den Vorgang vor Manipulation auf bösartigen Plattformen zu schützen.

3. Der Agent sollte bestimmte Kontrollinformationen protokollieren, anhand derer die Qualität und Integrität des besten Angebotes eingeschätzt werden kann.

Verifikation des besten Angebots

Zur Verifikation der Integrität und Qualität des gesammelten besten Angebotes können dem Agenten zusätzliche Kontrolldaten (Metadaten) mitgegeben werden, die neben o_{best} auch mit dem geheimen Schlüssel $\mathcal{K}$ chiffriert und von der Blackbox intern aktualisiert werden: $\#Angebote, Durchschnittspreis, Preislimit, Angebotsoberlimit, Angebotsunterlimit$. $\#Angebote$ sei die Anzahl bisher ausgewerteter Angebote, $Durchschnittspreis$ der bisherige Durchschnittspreis, $Preislimit$ das Preislimit für Angebote und $Angebotsunterlimit/Angebotsoberlimit$ die minimale/maximale Anzahl von zu untersuchenden Angeboten vor der Heimkehr des Agenten. Der neue Durchschnittspreis lässt sich einfach aus dem bisherigen Durchschnitt und der Anzahl bereits untersuchter Angebote bestimmen, wie in der Gleichung 6.19 formuliert.

$$neuer\ Durchschnittspreis = \frac{\#Angebote \cdot Durchschnittspreis + Preis(o_i)}{\#Angebote + 1} \tag{6.19}$$

Daneben könnte auch eine weitere Datenstruktur eingeführt werden; ein Preisstack, der jeweils nur die signierten Preise aller bislang verglichenen Angebote protokolliert. Die Verwaltung der Datenstruktur wird wiederum von der sicheren Hardware übernommen. Mit Einträgen der Art ⟨*Händlerkennung, Preis*⟩ ist dann eine Gegenprobe möglich. Das heißt, der Agentenbesitzer kann feststellen, mit welchen Händlern der Agent tatsächlich interagiert hat, und bei Bedarf stichprobenhaft die Gültigkeit der entsprechenden Preise überprüfen, z.B. durch eine direkte Preisanfrage bei dem (nun bekannten) Händler. Diese Mechanismen können zudem mit den in Kapitel 6.5 diskutierten allgemeineren Schutzmaßnahmen kombiniert werden.

6.7 Treffen autonomer Entscheidungen

Ein Wunschziel ist es, die Agenten nicht nur zum Sammeln von Daten zu bewegen, sondern sie in die Lage zu versetzen, *nichtabstreitbare Entscheidungen autonom und sicher zu treffen*. Bislang gibt es keinen bekannten Mechanismus, um die sichere Ausführung von Agenten auf unsicheren Plattformen ohne Einsatz von sicherer, vertrauenswürdiger Hardware zu gestatten.

Im nächsten Kapitel wird der Gedanke des Blackbox-Protokolls noch einen Schritt weiter gedacht: Durch die Einführung von *Entscheidungsboxen* werden sichere und autonome Agentenentscheidungen möglich.

6.7.1 Entscheidungsboxen

Die Grundidee ist das Aufsetzen eines logischen Netzwerkes von sicheren, bzgl. der Funktionalität und Leistung mächtigeren Geräten als die Blackboxen, die hingegen durch günstige Smart-Cards als Niedrigkostenlösung realisiert werden mögen. Diese Geräte entsprechen also von der Leistungsfähigkeit her betrachtet eher einem kompletten Rechner. Sie werden im Folgenden als *Entscheidungsboxen* bezeichnet und bilden als sichere Inseln in einem ansonsten unsicheren Netz den mobilen Agenten eine geschützte Ausführungsplattform an. Die Verwaltung von Blackboxen und Entscheidungsboxen würde durch eine *Blackbox–Behörde* vorgenommen. Der Vorteil liegt darin, dass die mobilen Agenten wichtige Entscheidungen in solchen sicheren Entscheidungsboxen sozusagen unter Ausschluss der Öffentlichkeit vornehmen können. Agenten können ihre gesammelten Daten analysieren und darauf basierend Entscheidungen treffen und weitere Aktionen einleiten. Beispielsweise könnte ein Agent im Comparison Shopping Szenario alle gesammelten Angebote auswerten und gemäß einer festgelegten Einkaufsstrategie daraus das Angebot im Namen seines Besitzers bestellen und kaufen.

Probleme. Solche autonomen Entscheidungen mit direkten finanziellen Konsequenzen erfordern einen hohen Grad an *Sicherheit.* Um präziser zu sein: Der Aufwand und die Ausgaben, die für einen erfolgreichen Angriff gegen das System notwendig sind, müssen mit zunehmendem Wert der gehandelten Objekte in vergleichbarem Maßstab anwachsen, so dass Angriffe nicht rentabel sind. Deswegen müssen in unserem Fall die Entscheidungsboxen *besonders stark* geschützt werden. Weiter stellt sich die Frage nach der *Wirtschaftlichkeit* und der *Finanzierung* eines solchen Systems, die im Folgenden ebenfalls adressiert wird.

Sicherheit. Die *Entscheidungsboxen* werden mit einem eigenen starken Public-Key Schlüsselpaar ausgestattet. Der Zugriff ist strikt beschränkt auf besuchende Agenten. In diesem Zusammenhang stellen Anderson und Kuhn in [AK96] fest, dass die

> [. . .] kritische Frage immer sei, ob ein Gegner unüberwachten Zugriff auf das Gerät erhalte. Ist die Frage mit nein zu beantworten, dann mögen relativ einfache Maßnahmen ausreichen: Der physikalische Zugriff sollte verweigert und die Entscheidungsboxen „bewacht“ werden. So ein Modell ist vergleichbar mit einem Kunden, der keinen physikalischen Zugriff auf die Hardware eines Geldautomaten hat, sondern diesen nur mit Hilfe einer wohldefinierten Benutzerschnittstelle verwenden kann. Dadurch wird

> die Robustheit und Sicherheit des Systems stark erhöht, und es werden Angriffe verhindert, die auf die Manipulation des Gerätes selbst oder auf das Ausspähen von hartverdrahteten Geheimnissen zielen.[7]

Finanzierung. Die *Finanzierung* der Entscheidungsboxen kann auf einer per-Verkaufsabschluss-Basis geregelt werden. Das bedeutet, dass ein Händler für jeden erfolgreichen Verkaufsabschluss, der mittels einer Entscheidungsbox herbeigeführt wird, eine gewisse Gebühr an die Blackbox-Behörde überweist, z.B. über ein internes System für Kleinstzahlungen (*Micropayments*).

Vorteile. Der Vorteil eines solchen Modells ist, dass (1) sich der Auftraggeber aus dem Netzerk ausklinken kann, nachdem der mobile Agent die Heimatplattform verlassen hat, da keine weitere Interaktion mehr notwendig ist. Der Agent kann (2) über einen Proxy eine maskierte Bestätigung der beschlossenen Handelsaktionen an den Agentenbesitzer senden (Quittung). Dadurch lassen sich bei Bedarf die *Privatsphäre* und die *Anonymität* des Auftraggebers schützen. Darüber hinaus können (3) Informationen gezielt zur ausschließlichen Auswertung innerhalb von *Entscheidungsboxen* bestimmt werden, durch Verschlüsselung mit dem öffentlichen Entscheidungsboxenschlüssel. Damit ist es möglich, besonders kritische Geheimnisse explizit nur den Entscheidungsboxen zugänglich zu machen, so dass die sensitiven Daten deswegen sogar im Falle einer erfolgreich manipulierten Blackbox nicht an die Öffentlichkeit gelangen. Zuletzt gestattet das Konzept (4) eine feinkörnige Abrechnung und Buchführung. Werden die Transaktionskosten wie vorgeschlagen von den Händlern getragen, so ergibt sich damit ein *kundenfreundliches Finanzierungsmodell.*

6.7.2 Nichtabstreitbarkeit von Kaufentscheidungen

Mit Hilfe der Entscheidungsboxen können mobile Agenten also auf sichere Art autonome Entscheidungen treffen. Besonders für die Händlerseite ist aber es von großer Wichtigkeit, dass getroffene Kaufentscheidungen einen bindenden Charakter besitzen und vom Kunden später nicht leichtfertig abgestritten werden können – ansonsten könnte den Händlern durch Falschbestellungen oder durch verweigerten Annahmen unter Umständen ein erheblicher finanzieller Schaden entstehen. Die Nichtabstreitbarkeit ist daher eine Schlüsseleigenschaft von agentenbasierten Einkaufsystemen.

Durch eine *systemweite Public-Key Infrastruktur* kann die Nichtabstreitbarkeit von Kaufentscheidungen und anderen Aktionen bewerkstelligt werden. Kunden, die sich am Einkaufsagentensystem beteiligen möchten, melden sich bei einem speziellen Server zur *Kundenregistrierung* an, dem *Customer Registry Server*. Jeder Kunde erhält ein initiales Public-Key Schlüsselpaar. Anschließend erzeugt er sich nun selbst ein eigenes Schlüsselpaar, welches der mobile Agent zum Signieren von Kaufentscheidungen (in Entscheidungsboxen) künftig verwenden soll. Den öffentlichen Schlüssel signiert er mit dem vom Customer Registry Server erhaltenen Schlüssel und mit dem neuen, geheimen Schlüssel. Anschließend leitet er den signierten öffentlichen Schlüssel an einen Customer Registry Server weiter, der den Kundeneintrag in seiner Datenbank aktualisiert und den öffentlichen Schlüssel bei den übrigen Kundenregistrierungsservern als neuen *Entscheidungsschlüssel* bekannt macht. Diese Schlüsselaktualisierung ist für alle am Handelssystem teilnehmenden Parteien bindend.

[7] Anm. des Autors: Zitat wurde aus dem Englischen übersetzt.

Vorteile. Der Kunde kann in selbstgewählten regelmäßigen Zeitabständen seinen Entscheidungsschlüssel bei einem beliebigen Customer Registry Server aktualisieren. Damit ist es bei Verwendung von entsprechenden Schlüssellängen derzeit praktisch nicht möglich, einen solchen Entscheidungsschlüssel kryptographisch zu brechen. Außerdem wird durch den *physikalischen Schutz* der *Entscheidungsboxen* die direkte Manipulation der sicheren Hardware verhindert und die Bandbreite der Angriffe auf kostenintensive und unpraktikable kryptographischen Methoden reduziert.

Ergebnis. Mit dem Konzept der *Entscheidungsboxen* und der Realisierung der Nichtabstreitbarkeit getroffener Entscheidungen (z.B. durch das Modell der *Customer Registry Server*) können mobile Agenten in die Lage versetzt werden, sicher und autonom im Namen des Besitzers bindende Entscheidungen zu treffen.

Die dabei eingesetzte Trusted Hardware bietet erstens einen sehr hohen Grad an Sicherheit und Schutz für mobile Agenten und bringt zweitens eine Vielzahl von weiteren Vorteilen mit sich, beispielsweise die sichere Interaktion und Datenauswertung, bindende Entscheidungen durch den Einsatz von Entscheidungsboxen und damit die Erschließung neuer Geschäftsbereiche.

6.8 Vergleich des Blackbox-Protokolls mit dem TTP-Protokoll von Corradi et al

In [CCMS99] stellen Corradi, Cremonini, Montanari und Stefanelli das TTP-Protokoll zum Schutz der Integrität mobiler Agenten vor. Dabei wird, wie der Protokollname andeutet, eine *Trusted Third Party* (TTP) eingesetzt. Es wird gezeigt, dass das Protokoll zusätzlich zu seiner Beschränkung auf geschlossene Netze oder Intranetze auch konzeptuelle Fehler und erhebliche Mängel aufweist.

Das TTP-Protokoll soll die Sicherung der Integrität des Agentenzustandes (Anwendungsdaten, Maschinencode und Initialisierungsdaten sowie Protokolldaten) unter Verwendung einer Trusted Third Party garantieren. Die Idee dabei ist, dass der mobile Agent nach jedem Besuch einer neuen Plattform und dem Sammeln zusätzlicher Daten zur TTP zurückkehrt, die für den Agenten einen *Message Integrity Code* (`MIC`) (etwa eine „Prüfsumme zur Wahrung der Nachrichtenintegrität“) speichert und die Gültigkeit des jeweils neuen `MIC`s überprüft.
Verwendete *Bezeichnungen* sind: Konstante Initialisierungsdaten KID, Protokolldaten PD, veränderliche Agentendaten AD, Leerzustand $void$ als Initialisierungswert für Variablen.

Protokollablauf: Kernpunkt im Protokoll ist die *Trusted Third Party*. Diese berechnet beim Besuch i des Agenten immer den neuen `MIC`, also $MIC_i \overset{TTP}{\rightsquigarrow} MIC_{i+1}$, und legt diesen im Agenten und in der TTP ab[8].

Der vollständige Ablauf lautet:

1. *Initialisierung*:
   (a) $AD_1 :=$ void; $PD_1 := init$; $MIC_1 := \mathrm{Hash}(KID, PD_1)$
   (b) $PD_1 := PD_1 + MIC_1$;

[8] Auszug aus *Mobile Agents and Security: Protocols for Integrity*, Kapitel 3.1, Seite 180: „A new digest MIC_{i+1} is computed and recorded on both the TTP and the agent.“

Der Agentenzustand wird durch das Tupel (KID, PD_1, AD_1) beschrieben.

2. *Datenaufnahme* bei Plattform/Host P_i:
   $AD_i := \text{Daten}(P_i)$;
   Der Agentenzustand ist nun (KID, PD_i, AD_i), wobei AD_i statt $void$ die gesammelten Daten enthält.

3. *Zustandssicherung und Prüfung* beim i-ten TTP-Besuch:

   (a) Vergleich ${MIC_i}^{Agent}$ mit ${MIC_i}^{TTP}$: ${MIC_i}^{Agent} \stackrel{?}{=} {MIC_i}^{TTP}$

   (b) Neue Daten in Protokolldaten übernehmen: $PD_{i+1} := PD_i + AD_i$.

   (c) `MIC` berechnen: $MIC_{i+1} := Hash(KD, PD_{i+1})$;

   (d) Neue Protokolldaten um `MIC` erweitern: $PD_{i+1} := PD_{i+1} + MIC_{i+1}$;

   (e) Das nächste Datenfeld initialisieren: $AD_{i+1} :=$ void;

   Der Agentenzustand ist nun $(KID, PD_{i+1}, AD_{i+1})_{i+1}$.

Es wurde nicht genau spezifiziert und ist deshalb unklar, wie die TTP die `MIC`s speichert. Dennoch wird behauptet, dass der Agent seine TTP frei wählen könne und nicht immer dieselbe besuchen müsse[9]. Diese Aussage ist bereits *widersprüchlich*, da gespeicherte `MIC`-Werte beim Aufsuchen anderer TTPs dort nicht verfügbar sind. Es würde einen erheblichen zusätzlichen Aufwand erfordern, wenn sichergestellt werden müsste, dass alle TTP jeweils die `MIC`-Werte aller Agenten kennen. Dies ist offensichtlich nicht gewollt, da man sich auf den im Agenten abgelegten `MIC` verlässt („embodied `MIC`“). Diese *Inkonsistenz* führt zum **Brechen** des integritätsschützenden `MIC`s:
Plattform P_k beschafft sich einen Schnappschuss des Agentenzustands mit Index $j < k$, also (KID, PD_j, AD_j) und MIC_j, entweder durch Speichern während eines vorausgegangenen Besuchs oder durch Konspiration mit einem anderen, zuvor vom Agenten besuchten Host. Die Plattform P_k kann beliebige Daten ab Index j des Agenten austauschen oder löschen, ohne entdeckt zu werden – alle dazu benötigten Daten sind bekannt:
KID, PD_j, und MIC_j. Der Angreifer wählt die Daten AD_i mit $j \leq i \leq k$ beliebig und berechnet die Schritte 3b, 3c und 3d rekursiv, wobei er auch nach Wunsch bestimmte Datenfragmente AD_i auslassen kann.

Geht der Agent weiter zu einer noch nicht besuchten TTP, so kann diese die Manipulation nicht erkennen, da die echten `MIC`s nicht zur Kontrolle bereitstehen, und die Integritätsverletzung bleibt *unentdeckt*. Gilt sogar, dass die TTP sich nur den letzten gültigen `MIC` merkt, dann kann der Angreifer P_k alle Daten AD_i mit $j \leq i < k$ manipulieren und anschließend AD_k so lange variieren (*Chosen Plaintext* Angriff [BGW97]), bis die Berechnung von ${MIC_k}'$ den alten Wert MIC_k ergibt und Gleichheit im Test 3a gilt. Das heißt, die Manipulation wird nicht einmal von der bereits besuchten TTP entdeckt. Dieser Angriff ist aber nur bei verhältnismäßig kleiner Länge n [in bit] der verwendeten Hashwerte praktikabel.

Bewertung: Trotz der Verwendung einer TTP bietet dieses Protokoll keinen verlässlichen Schutz der Integrität von Daten und Code der Agenten. Außerdem können die Berechnungen der Agenten auf fremden Plattformen noch immer gestört werden (Angriff auf den Kontrollfluss). Weiterer

[9] Auszug aus *Mobile Agents and Security: Protocols for Integrity*, Abschnitt 3.1, Seite 181: „We also point out that the agent is not forced to always deal with the same TTP but, by means of the embodied `MIC`, is free to exploit different TTPs.“

Schwachpunkt ist die Tatsache, dass der `MIC` jedes Mal über dem gesamten, monoton wachsenden Protokolldatenbestand vorgenommen werden muss, der wegen des rekursiven Aufbaus in Schritt 3b und 3b alle gesammelten Agentendaten mit einschließt. Ebenfalls problematisch ist, dass dieses Protokoll fehlertolerante Einsätze von Agenten verhindert: Sollen unterwegs identische Kopien des Agenten erzeugt werden (*Forking*), um fehlertolerant und parallel weiterzuarbeiten, dann führt das zu Problemen mit der Verwaltung der `MIC`s bei der TTP. Dass dieses in sich unschlüssige Protokoll auf zwei verschiedenen Konferenzen akzeptiert und veröffentlicht wurde, ist ein Beleg dafür, dass die Vielschichtigkeit der Sicherheitsproblematik mobiler Agenten sogar von Experten verkannt wird und heute vielfach noch nicht voll verstanden ist.

Vergleich mit dem Blackbox-Protokoll: Das in Kapitel 6.3 vorgestellte Blackbox-Protokoll hingegen löst auch diese Probleme, ist beliebig skalierbar und schützt vor Angriffen gegen den Agenten unter der Nebenbedingung, dass alle relevanten Daten nur im Agenten selbst gespeichert werden und keine externe Protokollierung von Kontrolldaten stattfindet. In diesem Fall bleibt als einzige Gefahr das Auftreten von *Denial-of-Service*- und *Replay*-Angriffen, die generell unter dieser Nebenbedingung nicht verhindert werden können. Alle anderen Arten von Konspirationen mit dem Ziel der teilweisen Datenmanipulation werden jedoch stets aufgedeckt.

6.9 Klassifikation der Sicherheitsdienste

Wie wir gesehen haben, reicht der Sicherheitsbegriff von grundlegenden kryptographischen Operationen (wie z.B. Verfahren zur Verschlüsselung oder zur digitalen Signatur) über zunehmend komplexere, zusammengesetzte Sicherheitsdienste bis zu sicheren, anwendungsorientierten Protokollen, wie es beim Blackbox-Protokoll der Fall ist. Es zeigt sich, dass die Dienste in einem *hierarchischen Verhältnis* zueinander stehen und komplexere Dienste jeweils auf einfacheren Sicherheitsdiensten aufsetzen. Daher lassen sich die Sicherheitsdienste für mobile Agenten gemäß ihrer strukturellen Komplexität und dem Grad der Spezialisierung wie folgt klassifizieren:

1. *Kryptographische Dienstprimitive*:
   Allgemeine kryptographische Grundoperationen.

2. *Elementare Sicherheitsdienste*:
   Allgemeine, auf kryptographischen Primitiven aufbauende anwendungsunabhängige Dienste.

3. *Anwendungsspezifische Sicherheitsdienste*:
   Auf kryptographischen Primitiven und elementaren Sicherheitsdiensten aufbauende Dienste mit höherer struktureller Komplexität und Spezialisierung auf die Problemstellung bestimmter praktischer Anwendungen. Es bestehen also Abhängigkeiten zu konkreten Anwendungen.

4. *Anwendungsspezifische, sichere Protokolle*:
   Im Gegensatz zu den kryptographischen Primitiven und den Sicherheitsdiensten definieren sichere Protokolle keine neuen Dienstklassen, sondern legen für ein konkretes Anwendungsproblem *Verhaltensregeln* für die teilnehmende Parteien fest und *spezifizieren Art (Format) und Umfang der auszutauschenden Daten*. Erst mit der Realisierung des Protokolls findet eine Abbildung auf tatsächliche Dienste statt. Dabei wird dann auf die Sicherheitsdienste und Primitiven der Klassen 1 bis 3 zurückgegriffen.

Kapitel 7

Implementierungen und Anwendungsbeispiele

In diesem Kapitel werden besondere Aspekte in Bezug auf die Realisierung und Implementierung der präsentierten Sicherheitsdienste besprochen. Im ersten Teil werden die sicheren Datenstrukturen exemplarisch vorgestellt. Anschließend wird in Kapitel 7.2 die Programmierung der JavaCard geschildert und die Einordnung des entwickelten Blackbox Applets ins Gesamtkonzept dieser Arbeit vorgenommen. Zum Abschluss werden in Kapitel 7.3 zwei praktische Beispiele für sichere Berechnungsdienste beschrieben.

7.1 Implementierte Sicherheitsdienste für Mobile Agenten

Hier werden exemplarisch die Implementierung einiger der oben eingeführten Dienste vorgestellt, denen im Rahmen dieser Arbeit auch im Zusammenhang mit dem Einsatz sicherer Hardware und der Integration im Comparison-Shopping-Prototypen besondere Bedeutung zukommt. Ergänzend werden weiterführende Dienste umrissen und mögliche Implementierungen kurz skizziert.

7.1.1 Integritätsgeschützter Stack

Die Implementierung der integritätsgeschützten Stacks aus Kapitel 5.3.1 erfolgte in zwei Abstraktionsstufen. In der ersten Stufe fußen die abstrakten Stackklassen auf einer allgemeinen Schnittstelle, dem Interface `StackCryptoServices`, das allgemeine Methoden zur Bereitstellung der Stackfunktionalität definiert. Die zweite Stufe stellen Implementierungen dieser Schnittstelle dar. Hier werden die sicherheitskritischen Funktionen verborgen und realisiert. Dabei findet die Abbildung der allgemeinen Stackprimitiven auf konkrete kryptographische Softwarebibliotheken oder sichere Hardware statt.

- Klasse `IntegrityStack`
  Der `IntegrityStack` realisiert einen verteilten Stack, dessen Integrität durch den Einsatz von sicherer Hardware auch auf unsicheren Plattformen zweifelsfrei überprüft werden

kann. Der Verteilungsaspekt ergibt sich z.B. beim Einsatz von SmartCards zur Ausführung der kritischen Operationen bei verteilten Rechnern. Alle SmartCards verfügen über ein gemeinsames Public-Key Schlüsselpaar, so dass durch Verschlüsseln der integritätsschützenden Daten mit Hilfe des öffentlichen SmartCard-Schlüssels diese nur innerhalb dieser SmartCards entschlüsselt und ausgewertet werden können. Diese Betrachtung zieht die Sicherheit der SmartCard selbst nicht in Zweifel.
Anwendungungsbeispiel: Last-In-First-Out-Datenstruktur (LAFO) mit absolutem, d.h. unkompromittierbarem Integritätsschutz.

- Klasse `PushOnlyIntegrityStack`
  Diese Ausprägung des integritätsgeschützten Stacks erlaubt nur push-Operationen und lesenden Zugriff auf alle gespeicherten Objekte, aber nicht das Entfernen einzelner Einträge. Das Auslesen der Daten erfolgt durch direkten Zugriff auf die Stack-Elemente über die `Vector`-Schnittstelle (Klasse `java.util.Vector`), von der sich die Stackklasse (Klasse `java.util.Stack`) ableitet. Werden die kritischen Stackoperationen auf sicherer Hardware ausgeführt, also zum Beispiel auf einer SmartCard, so führen etwaige erzwungene unerlaubte `Pop`-Operationen auf der zugrundeliegenden Basisklasse `java.util.Stack` zu einem inkonsistenten Zustand, da die Trusted Hardware keine `Pop`-Operationen zulässt und die kritischen Kontrolldaten nicht entsprechend aktualisiert. Die integritätssichernde Datenstruktur entspricht der in Kapitel 6.3 beschriebenen *Integrity Data Unit* (IDU). Beim `PushOnlyIntegrityStack` mit Hardware-Unterstützung können also keine effektiven Manipulationen auftreten, die beim Prüfen der Stack-Integrität nicht entdeckt würden.
  Anwendungungsbeispiel: Basisdatenstruktur für die Konstruktion komplexerer Datentypen. Zum Beispiel Verwendung in der Implementierung der Containerklassen `AppendOnlyContainer` und `SecretAppendOnlyContainer`.

- Schnittstelle `StackCryptoServices`
  Analog zum SPI-Konzept in Java (siehe Kapitel 5.1) werden durch die `StackCryptoServices`-Schnittstelle die zur Realisierung der sicheren Stackklassen benötigten sicheren Stackprimitive definiert.

- Schnittstellenimplementierung `StackCryptoServicesImpl`
  Diese Klasse stellt eine konkrete Implementierung der Schnittstelle `StackCryptoServices` zu Verfügung. Sie basiert auf den kryptographischen Systemdiensten in Java (Paket `java.security`) und zum Teil auf der *Java Cryptography Extension*, die im Kapitel 5.1, „Sicherheit in Java“, vorgestellt werden.

- Schnittstellenimplementierung `BlackboxCryptoServicesImpl`
  Die Klasse `BlackboxCryptoServicesImpl` implementiert ebenfalls die Schnittstellenklasse `StackCryptoServices`. Sie legt die Abbildung der Schnittstellendienste auf die Dienste des auf der JavaCard implementierten Blackbox-Applets fest. Die Programmierung der JavaCard wird in Kapitel 7.2 beschrieben.

7.1.2 Implementierung der Containerklassen

- Klasse `AppendOnlyContainer`
  Der `AppendOnlyContainer` setzt auf dem `PushOnlyIntegrityStack` auf; zum Anhängen von Informationen werden `Push`-Operationen zugelassen, das unentdeckte Entfernen von Elementen ist aber nicht möglich. Das folgende Codebeispiel gibt einen Einblick

in die Implementierung des `AppendOnlyContainers` und zeigt die Verwendung des sicheren Stacks zur Realisierung des sicheren Containers:

```
class AppendOnlyContainer
  implements java.io.Serializable {

  private PushOnlyIntegrityStack appendStack = null;

  [...]

  public AppendOnlyContainer(StackCryptoServices secureStackEngine)
    throws StackIntegrityException
  {
     // Initialisierung der Stack-Klasse wahlweise mit soft-
ware-
     // oder hardwarebasierter Stack-Engine:
     this.appendStack = new PushOnlyIntegrityStack(secureStackEngine);
  };


  public synchronized void append(Object obj)
    throws java.io.IOException
  {
     // Hinzuf"ugen von neuen Daten:
    appendStack.push((Serializable)obj);
  };


  public final synchronized void verify()
    throws SecurityException
  {
     // Verifikation der Integrit"at des Datenstacks.
    try {
      appendStack.verify();
    }
    catch (Exception ex) {
      throw new SecurityException("AppendOnlyContainer: veri-
fication failed: "
                                  + ex.getMessage());
    }
  };

  [...]
}
```

- Klasse `SecretAppendOnlyContainer`
  Die Klasse `SecretAppendOnlyContainer` erweitert die Klasse `AppendOnly-Container` insofern, als alle dem Container hinzugefügten Objekte zusätzlich mit einem zu Beginn festgelegten und im Containerobjekt gespeicherten öffentlichen Public-Key Schlüssel

verschlüsselt werden. Dies hat zur Konsequenz dass nur der Inhaber des passenden geheimen Schlüssels die gesammelten Objekte dechiffrieren und somit lesen kann.

Anwendungungsbeispiel: Behälter zum Sammeln von Preisangeboten. Nur für den Besitzer des passenden Schlüssels ist der Inhalt lesbar, die übrigen Parteien können nur blind Angebote anfügen.

- Klasse `WeakAppendOnlyContainer`
  Wird eine ausschließlich softwarebasierte Variante des Containers benötigt, so muss zum Schutz gegen das Abschneiden von Daten eine herkömmliche Prüfkette verwendet werden, die bei jeder Hinzufüge-Operation von Dritten nicht kopierbare oder erzeugbare Prüfdaten an die Kette anhängt. Wird die Integrität der Daten mit der Hashkette nach Abbildung 5.3 auf Seite 37 geschützt, so kann zusätzlich zur Verwendung des PushOnlyIntegrityStacks folgende fortlaufende Prüfsumme berechnet werden, wenn das Objekt O_i bei Anbieter S_i durch den Agenten A entgegengenommen wird:

$$\text{checksum} \;=\; [\text{checksum},\; \mathcal{SIG}_{S_i}(h_i)]_A \tag{7.1}$$

  Die Notation folgt der des Blackbox-Protokolls in Kapitel 6.3.

- Klasse `ReadOnlyContainer`
  Ein Vektor von Objekten wird bei der Initialisierung des Containers mit einem geheimen Public-Key-Schlüssel signiert, so dass später eine Manipulation der Daten durch Prüfen der Signatur mit Hilfe des passenden öffentlichen Schlüssels erkannt wird. Die Realisierung verwendet dazu die Klasse `SignedObject` aus dem Sicherheitspaket `java.security` von Java 2. Die Integrität der gesammelten Daten wird durch die Integritätsprüfung des integritätsgeschützten Stacks überwacht. Das unbemerkte Entfernen von Daten aus dem Container, etwa durch eine erzwungene POP-Operation, wird durch eine zusätzliche Prüfsumme verhindert. In Beispiel 7.1.2 wird die Grundstruktur des `AppendOnlyContainers` in java-orientiertem Pseudo-Code aufgezeigt. Falls das Clonen der Datenstruktur erlaubt sein soll, kann zusätzlich auch das Interface `java.lang.Cloneable` implementiert werden.

  Anwendungungsbeispiel: Adressliste (Liste von URLs (*Uniform Resource Locators*)), bei der Einträge hinzugefügt, jedoch nicht entfernt werden dürfen, und die an fremden Rechnern lesbar sein soll.

- Klasse `LockableContainer`
  Das Speichern und Entnehmen von Objekten erfolgt analog zur Benutzung eines `Hashtable`-Objekts. Die Klasse leitet sich jedoch *nicht* von der Klasse `java.util.Hashtable` ab. Der kryptographische Schutz wird mit Hilfe der Klasse `SealedObjects` aus dem JCE Paket (siehe 5.3) realisiert. Aufgrund der strengen Exportrestriktionen der Vereinigten Staaten von Amerika für kryptographische Bibliotheken wurde diese Klasse als Stub implementiert, der offiziellen Java-Dokumentation und Spezifikation folgend [Sun99c].

  Anwendungungsbeispiel: Ein Behälter, dem beliebig mit einem Identifikator bezeichnete Objekte entnommen und hinzugefügt werden können, und der sich kryptographisch abschließen (verschlüsseln) lässt. Dieser Behälter kann an Dritte weitergereicht werden, indem der mit dem öffentlichen Public-Key Schlüssel des Zielempfängers verschlüsselte geheime Containerschlüssel mitübergeben wird. Bei jedem Verschließvorgang kann ein neuer geheimer (Einmal-)Schlüssel verwendet werden.

7.1.3 Realisierungsansätze für die Gruppencontainer

Die Gruppencontainerklassen vereinfachen dem Programmierer die Entwicklung von Kooperationsbeziehungen zwischen Agenten und dritten Parteien. Diese Dienste sind bei der Betrachtung der Sicherheitsaspekte mobiler Agenten nebenrangig, da sie nur eine Verfeinerung und Spezialisierung der implementierten „einfachen“ Containerklassen darstellen; sie wurden in dieser Arbeit nicht implementiert. Im Folgenden werden jedoch Möglichkeiten zur Realisierung solcher Gruppencontainer kurz beleuchtet.

Die Bestimmung der Zielgruppe kann auf verschiedene Weise realisiert werden:

a) *Statische* Festlegung *zum Zeitpunkt der Erzeugung* des Containers:
Die Authentifizierung der Gruppenmitglieder kann über deren Public Key Schlüsselpaar erfolgen.

   – Autorisierung durch Kenntnis des privaten Schlüssels: Zu Beginn werden alle öffentlichen Schlüssel der gewählten Gruppenmitglieder mit im Container abgespeichert. Später wird der Zugriff auf die Funktionen des Containers nur gestattet, wenn sich der Benutzer mit Hilfe eines passenden geheimen Schlüssels ausweisen kann. Bei Manipulation des Kontrollflusses des Agenten auf einem unsicheren, bösartigen Rechner bietet dies jedoch nicht ausreichend Schutz, da z.B. der Kontrollfluss beim Schlüsselvergleich manipuliert werden kann.
   – Mit öffentlichem Schlüssel geschütztes geteiltes Geheimnis: Bei der Initialisierung des Gruppencontainers wird ein geheimer (symmetrischer) Schlüssel erzeugt und mit dem öffentlichen Schlüssel jedes gewählten Gruppenmitglieds verschlüsselt und mit abgespeichert. Außerdem werden mit diesem Schlüssel die zu lesenden Nutzdaten (z.B. im `GroupReadOnlyContainer`) oder die Prüfdaten (z.B. im `GroupAppendOnlyContainer`) verschlüsselt. Die legale Benutzung des Containers erfordert später die Kenntnis des geheimen symmetrischen Schlüssels, ohne den entweder die Nutzdaten nicht gelesen oder die Prüfdaten nicht aktualisiert werden können, so dass eine Verifikation des Containerzustandes nach unerlaubter Manipulation entdeckt wird.

b) *Dynamische* Festlegung der Gruppenzugehörigkeit *zur Laufzeit*:

   – Mit offentlichem Schlüssel geschütztes geteiltes Geheimnis:
   Analog zum statischen Fall mit der Neuerung, dass es jedem Gruppenmitglied erlaubt ist, entweder (a) den Gruppenumfang neu festzulegen und ggf. einen neuen geheimen Schlüssel zu erzeugen und für die jeweiligen Mitglieder in verschlüsselter Form abzulegen, oder (b) nur neue Mitglieder in die Gruppe aufzunehmen und für diese den existierenden geheimen Schlüssel mit deren öffentlichen Schlüssel chiffriert mit abzuspeichern (z.B. in einem `AppendOnlyContainer`). Das setzt gegenseitiges Vertrauen zwischen allen Gruppenmitgliedern voraus, und ein bösartiger Gruppenteilnehmer kann die Gruppenzusammensetzung korrumpieren.
   – Eine dynamische Verwaltung des Gruppenumfangs über *Shared Secrets*:
   Mit Hilfe der Technik des *Secret Sharing* [Sha79] kann eine anspruchsvollere dynamische Verwaltung der Gruppenzugehörigkeiten und des Gruppengeheimnisses bewerkstelligt werden, bei der Gruppenänderungen oder Containermanipulationen z.B. die Zustimmung einer Untermenge der Gruppenteilnehmer erforderlich macht. Dies erhöht einerseits die Kontrollmöglichkeiten und reduziert die Gefahr des Missbrauchs. Andererseits erfordert es den Gebrauch von Secret Sharing Protokollen und Verhandlungen

zwischen beteiligten Gruppenmitgliedern, was einen Kommunikationsaufwand mit sich bringt und die Flexibilität bei der Containerverwendung einschränkt. Hier stellt sich also die Frage der Verhältnismäßigkeit, und in den meisten Fällen wird die einfache statische oder die einfache dynamische Realisierungsvariante genügen.

Ein effizienter Zugriff auf den geteilten Schlüssel ist mittels einer Hashtabelle möglich, mit dem jeweiligen öffentlichen Schlüsselobjekt eines Gruppenmitglieds als Hashtabellenschlüssel.

7.2 Programmierung der JavaCard

In diesem Kapitel werden die wesentlichen Bestandteile der JavaCard-Implementierung erläutert. Eine Beschreibung der Eigenschaften und des Aufbaus der IBM JavaCard findet sich in [BBE99a]. Weitere Informationen zum Thema JavaCard und SmartCard finden sich sowohl unter [BBE+99b] als auch im Internet, etwa `www.opencard.org` [Ope] oder auf den JavaCard-Seiten der Firma Sun Microsystems [Sun99b]. Besonders hilfreich bei der Entwicklung und Programmierung der JavaCard waren das Buch *SmartCard Application Development Using Java* von Hansmann et al. [HNSS99], der *Java Card Applets Developers Guide* von Sun Microsystems [Sun99a] sowie einige Artikel zum Thema JavaCard im Internet unter `www.javaworld.com` [Jav].

7.2.1 JavaCard Blackbox Applet

Das `Blackbox`-Applet stellt die zur Realisierung der sicheren Stackklassen aus Kapitel 5.3.1 benötigten Funktionen auf einer SmartCard bereit. Die vom Applet unterstützte Befehlsstruktur wird zusammen mit den zugehörigen Operationen in Tabelle 7.1 und Tabelle 7.2 dargestellt. Der Austausch von Daten und Befehlen mit der JavaCard erfolgt über sogenannte *Application Protocol Data Units*, Dateneinheiten des Anwendungsprotokolls, im Folgenden kurz als **APDU** bezeichnet. Die ersten vier Bytes sind dabei fest definiert, die restlichen Bytes variieren je nach APDU-Typ, d.h. je nachdem, ob Nutzdaten geschickt und/oder Ergebnisdaten empfangen werden sollen. Das erste Byte enthält immer den Code für die gewählte Befehlsstruktur (Byte `CLA`), das zweite den Instruktionscode (Byte `INS`), und Byte drei und vier zwei optionale Parameter (Byte `P1` und `P2`). Alle Befehle, die an das Blackbox Applet auf der JavaCard gesendet werden, tragen den Kommando-Code `CLA = 0xB0`[1]. Durch Setzen des gewünschten Instruktionscodes `INS` und gegebenenfalls der erforderlichen Wahlparameter (`P1` und `P2`) wird auf der Karte die spezifizierte Operation ausgeführt. Das aufrufende Programm auf dem Host muss bei mehrstufigen Operationen (z.B. das Hashen großer Objekte auf der Karte) die jeweilige Aufrufsemantik berücksichtigen. Die Rückgabe von Ergebnissen und Daten erfolgt über eine sogenannte *Response APDU*, kurz **RAPDU**. Eine genaue Beschreibung der Formate und APDU-Typen findet sich beispielsweise in [Sun99a].

Eine Instanz eines sicheren Stacks wird durch das Tupel $(PushOnlyFlag, h_0, h_{n+1})$ in eindeutiger Weise beschrieben. Dieses Datentupel wird im weiteren als Stack-**IDU** oder einfach `IDU` (*Integrity Data Unit*) bezeichnet, da diese Daten zur Integritätsprüfung unbedingt notwendig sind: Der Schalter $PushOnlyFlag$ legt fest, ob Elemente aus dem Stack entfernt werden dürfen oder nicht (und wird auf der Karte bei `Pop`-Aufrufen ausgewertet); h_{n+1} ist der Abschluss der über den Daten berechneten Hashkette und h_0 deren geheime Verankerung. Die vom Blackbox-Applet verwaltete Stack-IDU hat in der Sicherheitsdienste-API die

[1] 0xB0 entspricht der hexadezimalen Notation – zu lesen ist also $B0_{16}$.

Klasse `agents.security.stack.IntegrityDataUnit` zum Pendant. Das Public-Key Schlüsselpaar der Blackbox sei $(PubKey_B, PrivKey_B)$. In der prototypischen Implementierung werden dabei 512bit RSA Schlüssel verwendet, die nicht mehr länger unter die US-Exportrestriktionen fallen. Die mit der auf der Karte verfügbaren kryptographischen Bibliothek *CryptoZ* erzeugten Schlüssel haben dabei das folgende Format:

- *Private-Key 512bit RSA*: 130 Bytes. Der private Schlüssel enthält sowohl den geheimen 512bit Exponenten d sowie den Modulus m: $PrivKey_B := (d, m)$
- *Public-Key 512bit RSA*: 67 Bytes. Der öffentlichem Exponent ist fest gewählt zu $e = 3$. Damit lässt sich der öffentliche Schlüssel auch aus dem geheimen Schlüssel konstruieren: $PubKey_B := (3, m)$. In der Implementierung werden jedoch der Einfachheit halber beide Schlüssel auf der Karte gehalten, da sie bei jeder Stack-Operation auf der Smart-Card benötigt werden.

Als Hashalgorithmus wird SHA-1 eingesetzt. Die Blockgröße beim Hashen wird durch die Kryptobibliothek auf 64 Bytes festgelegt. Der Aufruf von SHA-1 mit vollen 64-byte-Blöcken versetzt die SHA-1-Engine in den *Update*-Modus; beim Hashen von weniger als 64 Bytes Nutzdaten wird der Hashwert fixiert und der Hashvorgang terminiert. Dieser Sachverhalt spiegelt sich in Tabelle 7.1 im Vorhandensein zweier verschiedener Hash-Operationen – das heißt einer Ein-Schritt- (*Single-Pass*) und einer Mehr-Schritt-Operation (*Multi-Pass-Operation*) – wider.

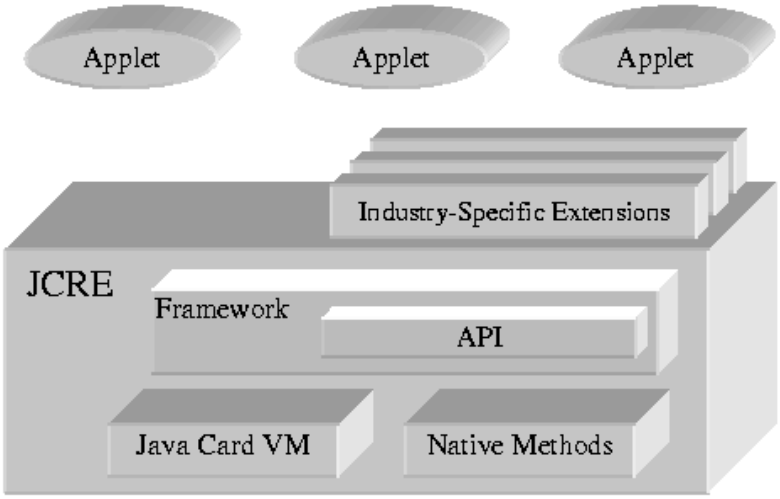


Abbildung 7.1: Java Card Architektur: Applets und Java Card Runtime Environment

Der Entwicklungs- und Installationsprozess von Applets für die IBM JavaCard umfasst folgende Stufen:

1. Übersetzen des Applet-Quellcodes zu *Java Bytecode* (.class).
2. Konvertieren des Java Bytecodes zu plattform-abhängigem *Kartenbytecode* (engl. *Compiled Applet* `CAP`) (.cap).
   Dieser Kartenbytecode wird also außerhalb der Karte (*Off-Card*) erzeugt und später auf der *Java Card Virtual Maschine* (`JCVM`, `VM`, siehe Abb. 7.1) in der Karte (*On-Card*) ausgeführt. Die `JCVM` ist dabei Bestandteil der Ausführungsumgebung der Java Card, der sogenannten

Operation (Byte INS)	**Argument 1** (Byte P1)	**Argument 2** (Byte P2)	**Beschreibung**
0x20	PUSHONLY*	WITHNONCE*	*Neuen Stack erzeugen.* Rückgabewert: verschlüsselte Stack-IDU.
0x25	NOPARAM	NOPARAM	*Stack-Bezeichner (Label) abfragen,* der in der Karte aus der zuvor entschlüsselten Stack-IDU extrahiert wird. Rückgabewert: Stack-Label (20 Bytes).
0x30			*Element auf dem Stack ablegen* (push).
	INIT	NOPARAM	Blackbox Applet mit verschlüsselter Stack-IDU initialisieren, die intern dechiffriert und zeitweilig gespeichert wird.
	INIT_UPDATE	NOPARAM	Erste 44 Bytes des neuen Objektes O_{n+1} zur Bildung des ersten 64-byte Hashblocks mit h_{n+1}.
	UPDATE	NOPARAM	Nächsten Objektdatenblock dazuhashen.
	FINALIZE	NOPARAM	Letzten Datenblock dazuhashen, Hashvorgang abschließen (fixieren), IDU aktualisieren und verschlüsseln.
	FINALIZE	INIT	Pop-Operation direkt nach INIT abschließen, da das gesamte Objekt O_{n+1} in eine APDU passt. Rückgabewert: aktualisierte Stack-IDU.
0x40			*Element vom Stack herunternehmen* (pop).
	INIT	NOPARAM	Blackbox Applet mit verschlüsselter Stack-IDU initialisieren, die intern dechiffriert wird.
	UPDATE	INIT	h_n extrahieren und ersten Datenblock hashen.
	UPDATE	NOPARAM	Nächsten Objektdatenblock dazuhashen.
	FINALIZE	NOPARAM	Letzten Datenblock dazuhashen und Hashvorgang abschließen (fixieren), Hashwert mit h_{n+1} überprüfen und ggf. IDU aktualisieren und verschlüsseln.
	FINALIZE	INIT	Push-Operation direkt nach INIT abschließen, da h_n und Objekt O_n in eine APDU passen. Rückgabewert: aktualisierte Stack-IDU.
0x50			*Integrität des Stacks überprüfen.*
	INIT	NOPARAM	Blackbox Applet mit verschlüsselter Stack-IDU initialisieren, die intern dechiffriert wird.
	LABEL	NOPARAM	Stack-Label überprüfen.
	UPDATE	INIT	h_n extrahieren und ersten Datenblock hashen.
	UPDATE	NOPARAM	Nächsten Objektdatenblock dazuhashen.
	FINALIZE	NOPARAM	Letzten Datenblock dazuhashen und Hashvorgang abschließen (fixieren), Hashwert mit h_{n+1} überprüfen und Ergebnis in RAPDU ablegen.
	FINALIZE	INIT	Überprüfung direkt nach INIT vornehmen, da h_n und Objekt O_n in eine APDU passen. Rückgabewert: VERIFICATION_SUCCESSFUL oder VERIFICATION_FAILED.

Tabelle 7.1: Befehlsumfang des Blackbox Applets 1. Teil (Befehlscode CLA = 0xB0).

Operation (Byte `INS`)	**Argument 1** (Byte `P1`)	**Argument 2** (Byte `P2`)	**Beschreibung**
0x10			*Blackbox-Schlüsselpaar initialisieren.*
	CONTINUE	PRIVATEKEY	Erste Hälfte des privaten 512bit RSA Blackbox-Schlüssels auf die Chipkarte laden. (65 Bytes)
	FINALIZE	PRIVATEKEY	Zweite Hälfte des privaten 512bit RSA Blackbox-Schlüssels auf die Chipkarte laden. (65 Bytes)
	FINALIZE	PUBLICKEY	Öffentlicher 512bit RSA Blackbox-Schlüssel auf Chipkarte laden (67 Bytes).
0x15	NOPARAM	NOPARAM	*Öffentlichen Blackbox-Schlüssel abfragen.* Rückgabewert: öffentlicher Blackbox-Schlüssel.
0x77	NOPARAM	NOPARAM	*Einen einzigen Datenblock hashen* (SHA-1). Rückgabewert: SHA-1 Hashwert (20 Bytes).
0x79			*Einen kontinuierlichen Datenstrom hashen* (bzw. Objekt beliebiger Größe auf der Karte hashen).
	INIT	NOPARAM	Hash-Engine initialisieren und ersten Datenblock hashen.
	UPDATE	NOPARAM	Nächsten Datenblock hashen.
	FINALIZE	NOPARAM	Letzten Datenblock hashen und Hashvorgang abschließen (fixieren). Rückgabewert: SHA-1 Hashwert (20 Bytes).

Tabelle 7.2: Befehlsumfang des Blackbox Applets 2. Teil (Befehlscode `CLA` = `0xB0`).

Java Card Runtime Environment `JCRE`, die zusätzlich Karten-API, Karten-Klassen und native Dienste integriert (Abb. 7.2).

3. Signieren des *Kartenbytecodes* nach dem *Visa OpenPlatform* Standard (`VOP`) (.scap).

4. Installieren der Applet-SCAP-Datei auf der Karte (EEPROM).

Schritt 3 ist nicht Teil der Java Card 2.1 Spezifikation, bei der gemäß Abbildung 7.2 die CAP-Datei direkt auf die Karte geladen wird. Die Benutzung der Java Card und die praktischen Aspekte wie das Laden von Applets auf die Karte wird zum Beispiel in [HNSS99] beschrieben und an dieser Stelle nicht detaillierter ausgeführt.

7.2.2 Blackbox Verbindungsmanager

Die Befehlsaufrufe an das Karten-Applet müssen auf dem Host in entsprechende APDUs (siehe Kapitel 7.2.1) transformiert werden. Komplexere Befehle erstrecken sich dabei über den Austausch mehrerer solcher Protokolldatenpakete und haben den Charakter einer *Sitzung*. Um Java-Anwendungen von den Feinheiten der Kartenbenutzung zu verschonen und um Verbindungen zum Kartenterminal und zur JavaCard komfortabler zu verwalten, wurde ein *Verbindungsmanager* für den Blackbox-Zugriff implementiert.

Die Klasse `BlackboxConnection` bietet den Java-Anwendungen eine einfache Schnittstelle zur Verbindungsverwaltung und Interaktion mit der JavaCard an. Zudem werden einfache, aus

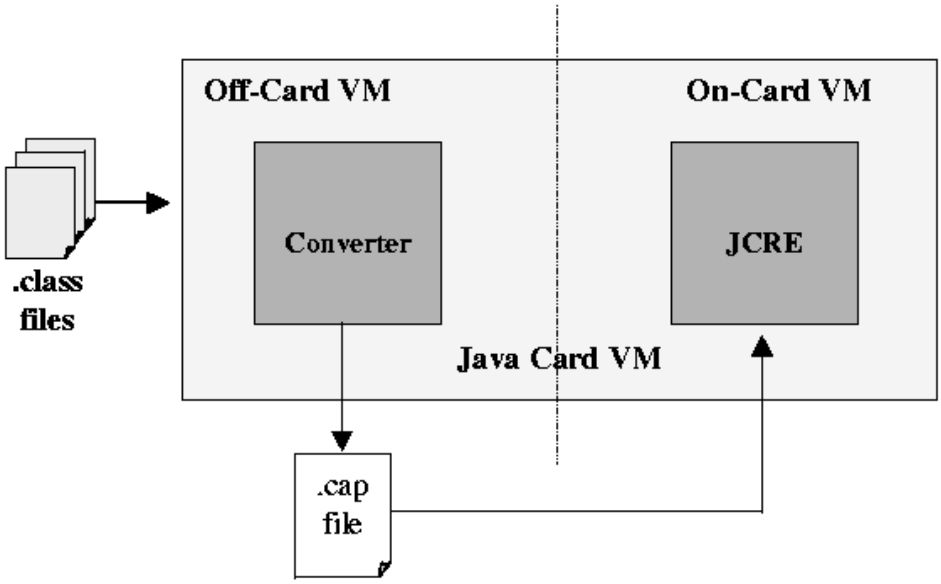
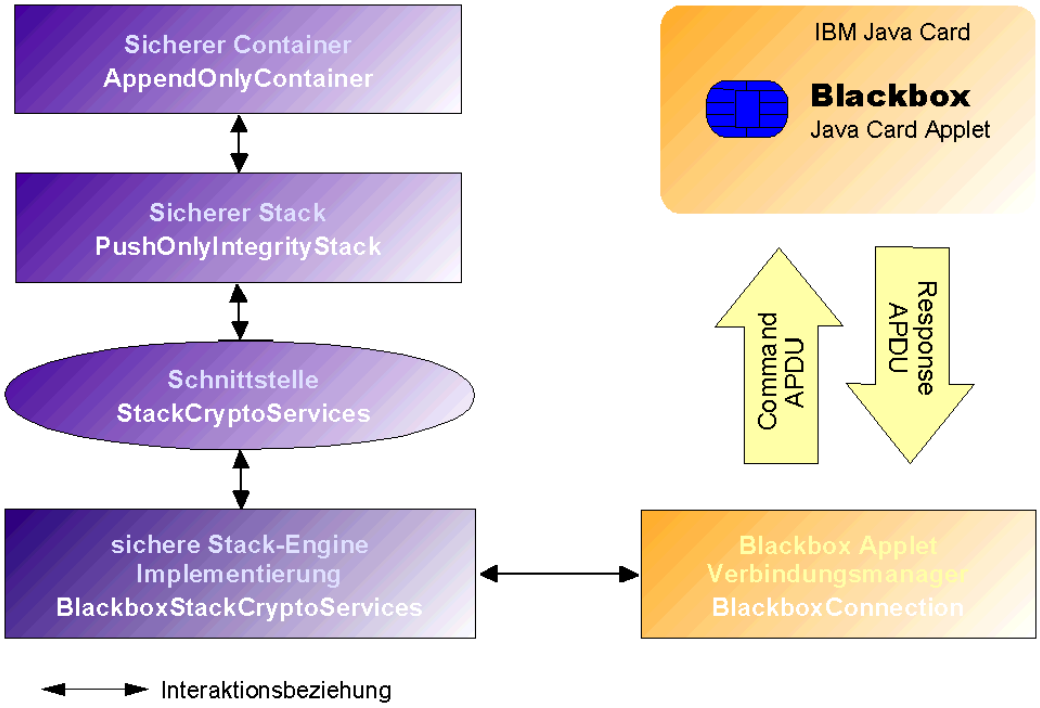


Abbildung 7.2: Konvertierung und Installation von JavaCard Applets

Abbildung 7.3: Stufenweise Integration der JavaCard Blackbox-Stackimplementierung in den abstrakten Sicherheitsdienst `AppendOnlyContainer`.



verschiedenen Blackbox-Primitiven zusammengesetzte Dienste bereitgestellt, die von Blackbox-Anwendungen häufiger benötigt werden.

Die angebotenen Dienstprimitiven sind im einzelnen

1. Verbindungsverwaltung:
   - `connect`
     Verbindungsaufbau mit dem Kartenterminal und Selektion des Applets.
   - `disconnect`
     Verbindungsabbau zur Freigabe des Kartenterminals.
   - `isConnected`
     Abfrage des Verbindungszustandes.
2. Blackbox Schlüsselmanagement:
   - `uploadKeys`
     Neues Public-Key-Schlüsselpaar auf die Karte laden (Blackbox Applet).
   - `getPublicKey`
     Den öffentlichen Public-Key-Schlüssel des Blackbox Applets abfragen.
3. Sichere Stack-Primitive:
   - `createStack`
     Einen neuen Stack erzeugen, d.h. eine neue Stack-IDU generieren (siehe Kapitel 7.2.1).
   - `pushStack`
     Ein Objekt auf dem Stack ablegen (und IDU aktualisieren).
   - `popStack`
     Ein Objekt vom Stack herunternehmen (und IDU aktualisieren).
   - `getLabel`
     Die Kennung des Stacks (Label) zurückgeben (aus IDU extrahiert).
   - `verifyLabel`
     Eine gegebene Stack-Kennung verifizieren (mit Label in IDU vergleichen).
   - `verifyStack`
     Die Verankerung (und damit die Stackkennung) und den Abschluss der Hashkette auf der Karte überprüfen (IDU auswerten und Hashwerte nachrechnen).

Das Zusammenspiel der verschiedenen Komponenten wird in Abbildung 7.3 am Beispiel des `AppendOnlyContainers` aus Kapitel 5.3.2 mit unterliegender Blackbox-Stack-Implementierung gezeigt.

7.2.3 Fortführung und Weiterentwicklung

Das entwickelte JavaCard-Applet ist voll funktionsfähig und stellt alle erforderlichen Funktionen bereit, die zur Realisierung der sicheren Stackklassen (`IntegrityStack` und `PushOnlyIntegrityStack`) und indirekt der sicheren Containerklasse `AppendOnlyContainer` benötigt werden. Die Implementierung ist von prototypischem Charakter, und in diesem Zusammenhang möchte ich hier folgende Ideen und Möglichkeiten zur Weiterentwicklung und Fortführung der praktischen Arbeit geben, die wegen des engen Zeitrahmens der Diplomarbeit nicht in die Umsetzung einfließen konnten:

1. Die Stack-IDU (siehe Kapitel 7.2.1) vor jeder karteninternen Verschlüsselung um n zufällige Bytes erweitern, um eine *probabilistische Verschlüsselung* zu erreichen, so dass sich auch das Chiffrat zweier zeitlich auseinanderliegender aber zustandsgleicher IDUs unterscheidet.
2. Realisierung eines `karteninternen Zustandsautomaten`, anhand dessen unerlaubte Sprünge bzw. Auslassungen bei mehrschrittigen Operationen (d.h. Austausch mehrerer APDUs im Zuge derselben Operation) erkannt und behandelt werden können.
3. *Optimierung des Datentransfers* zwischen Host und SmartCard.
4. *Generalisierung* der Daten- und Befehlssyntax, um beispielsweise variable Schlüssellängen zu unterstützen.
5. Behandlung von *Nebenläufigkeit*. Das Kartenterminal kann zu einem Zeitpunkt immer nur von einer Anwendung benutzt werden. Konkurrierende Zugriffe auf das Blackbox-Applet wurden bisher nicht modelliert. Dies ist ggf. auch Aufgabe des Verbindungsmanagers.
6. Unterstützung des *OpenCard-Standards*. Die vorliegende Implementierung benutzt direkt die JavaCard API, die einen direkten Zugriff auf die Kartenfunktionalität erlaubt. Eine Implementierung nach dem OpenCard-Standard erfordert die Programmierung zahlreicher zusätzlicher Funktionen und Methoden, erhöht aber den Grad der Interoperabilität und Portabilität.

Vielseitigere und flexiblere Lösungen für die Punkte 2 und 4 lassen jedoch auch den Programmumfang des Applets mit anwachsen. Dem kann zum Beispiel durch das Entfernen der im Appletcode eingebauten Prüfroutinen und Plausibilitätstests teilweise entgegengewirkt werden. Nach oben hin ist die Programmgröße jedoch durch den EEPROM-Speicher auf der SmartCard begrenzt, der bei zur Zeit gängigen Karten bei 16 kBytes anzusiedeln ist. Die *Ausführungsdauer* für einschrittigen Kartenoperationen liegt bei der aktuellen Implementierung deutlich unter einer Sekunde. Die Antwortzeit einer Push-Operation für ein ca. 1000 Bytes umfassendes Datenobjekt (16 Update-APDUs erforderlich) liegt zwischen ein und zwei Sekunden. Dabei wird jedoch für jedes empfangene Paket auf der Karte eine Prüfsumme über die erhaltenen Kommandodaten berechnet. Ohne die Prüfsummenberechnung liegt die Ausführungsdauer unter einer Sekunde.

7.2.4 Entwicklungsumgebung und Werkzeuge

Die Programmierung des JavaCard Applets erfolgte unter dem Betriebsystem Windows NT 4.0 (Service Pack 5) mit EmacsNT als Editor und Java Entwicklungsumgebung. Nach anfänglichen Problemen unter Windows NT mit Kartenlesegeräten der Firma XAC Automation [XAC99] – im Batchmodus reagierte die Karte auf schnell aufeinanderfolgende Kartenbefehle mit Fehlermeldungen – wurde ein Kartenlesegerät der Firma Towitoko [Tow] mit Bezeichnung `CHIPDRIVE micro` erfolgreich und ohne Störungen eingesetzt. Die Kartenlesegeräte wurden jeweils am seriellen Port eines IBM Personal Computers 300XL angeschlossen. Die Programmiersprache für die Kartenprogrammierung war `Java` in der Version `SUN Java Development Kit 1.1 (JDK 1.1)`. Als SmartCard wurde die am IBM Forschungslabor in Rüschlikon [IBMb] entwickelten IBM JavaCard, die *IBM Smart-e-Card* (ROM-Maske 2), eingesetzt. Die JavaCard enthält einen kryptographischen Koprozessor und einen *echten* Zufallszahlengenerator. Die *IBM JavaCard* verfügt z.B. über 1200 Bytes flüchtigen Speicher (*Random Access Memory*, RAM), 16 kBytes reprogrammierbarer Nur-Lese-Speicher (*Electrically Erasable Programmable Read-only Memory*, EEPROM) und 32 kBytes Nur-Lese-Speicher (*Read-only Memory*, ROM), wobei nach Angaben von IBM 14kByte des EEPROM-Speichers für Applets frei seien [BBE99a].

Die verwendeten kryptographische Funktionen wurden von der ebenfalls am ZRL entwickelten `CryptoZ`-Bibliothek bereitgestellt [BBE99a]. Aus exportrechtlichen Gründen werden zur Public-Key Verschlüsselung RSA [RSA78] mit max. 512bit Schlüssellänge verwendet; als kryptographische Hashfunktion diente der Secure Hash Algorithm No. 1 (SHA-1) [NIS95].

Blackbox-Applet Testapplikation. Zum Testen der Blackbox-Appletfunktionen auf der JavaCard stand keine Simulationsumgebung zu Verfügung. Stattdessen wurde der konfigurierbare Lisp-Editor Gnu-EMACS eingesetzt; alle erforderlichen Arbeitsschritte wurden in einem `Makefile` spezifiziert und konnten direkt aus dem Editor heraus ausgeführt werden: Übersezten der Quelldateien, Erzeugen der `CAP`- und `SCAP`-Dateien aus den `CLASS`-Dateien, Laden der `SCAP`-Datei auf die Karte, Löschen der `SCAP`-Datei von der Karte und Ausführung des in Java implementierten graphischen Testwerkzeuges `BlackboxGUI`.

Jede der durch das Blackbox Applet realisierten Funktionen wurde mit Hilfe des Testprogramms `BlackboxGUI` ausgeführt und getestet. Dazu wurden kartenintern Prüfsummen über Zwischenergebnisse und Kommandodaten berechnet, diese parallel dazu extern nachgerechnet und nach der Ergebnisrückgabe verifiziert. Außerdem wurden zahlreiche Plausibilitätstests innerhalb wie außerhalb der Karte vorgenommen, und im SmartCard-Applet mit einer breiten Anzahl von Ausnahmemeldungen (Fehlercodes) gearbeitet, die genaue Rückschlüsse auf fehlerhaft konstruierte APDUs gestatteten. Das Testwerkzeug zeigt bei der Ausführung von Operationen stets die aktuell gesendeten Kommandoparameter und korrespondierenden Ergebniswerte an und protokolliert die Test- und Debugmeldungen in einem Textfenster.

7.3 Sichere Ausführung und Berechnung

In diesem Kapitel werden zwei praktische Beispiele für sichere Berechnungsdienste beschrieben.

7.3.1 Geschützte Interaktion mittels Trusted Hardware

Die geschützte Interaktion von Agenten ist ein Anwendungsbeispiel der in Kapitel 5.3.3 besprochenen sicheren Berechnung unter Einsatz von Trusted Hardware. Mit dem zusätzlichen Hardwareeinsatz wird die *Interaktion* zwischen Agenten untereinander oder zwischen Agent und Dritten geschützt. Eine Realisierungsmöglichkeit, die dies leistet, wird nachfolgend ausführlich demonstriert.

Ein dabei verwendetes sicheres Hardware-Gerät wird im Folgenden kurz als *Blackbox* (Plural *Blackboxen*) bezeichnet. Alle Verschlüsselungsvorgänge seien probabilistisch, d.h. zwei Verschlüsselungsvorgänge desselben Klartexts ergeben zwei voneinander völlig unabhängige und verschiedene Chiffrate.

Voraussetzungen:

Alle Blackboxen haben das gleiche Public-Key-Schlüsselpaar, also denselben geheimen Schlüssel $PrivKey_B$, und den bekannten öffentlichen Schlüssel $PubKey_B$. Ein Agent A_i, der die verfügbaren Blackboxen für sichere Kommunikation und Interaktion nutzen möchte,

1. erzeugt einen geheimen Schlüssel $SecretKey$, und
2. eine zufällige Kennung r (*Nonce*)
3. erzeugt ein Interaktionsobjekt, hier $myMsg^i$,
4. verschlüsselt dieses und die Kennung zusammen mittels $SecretKey$ zu

$$\overline{myMsg^i}_r := Enc_{SecretKey}(myMsg, r), \tag{7.2}$$

5. und verschlüsselt $SecretKey$ und r mittels dem öffentlichen Schlüssel der Blackbox $PubKey_B$, also

$$\overline{SecretKey_r} := Enc_{PubKey_B}(SecretKey, r). \tag{7.3}$$

Sichere Interaktion innerhalb von Blackboxen

Die Blackbox bietet mobilen Agenten einen Dienst mit Namen $evaluate$ an, der folgendes leistet:

Als *Eingabeparameter* erhält der Dienst die Objekte $\overline{SecretKey_r}$, $\overline{myMsg_r}$ des Agenten, und zusätzlich das erhaltene Interaktionsobjekt, entweder

(a) verschlüsselt mit $SecretKey$ und zusammen mit derselben Kennung r als $\overline{receivedMsg_r}$, oder

(b) unverschlüsselt als $receivedMsg$ (ohne die Kennung).

Die *Signatur* des Dienstes lautet also $evaluate(\overline{SecretKey_r}, \overline{myMsg_r}, \overline{receivedMsg_r})$ bzw. $evaluate(\overline{SecretKey_r}, \overline{myMsg_r}, receivedMsg)$.

Die *Auswertungsschritte* innerhalb der Blackbox sind dann:

1. Die Blackbox erhält $SecretKey$ und r durch Entschlüsseln von $\overline{SecretKey_r}$ mit Hilfe ihres geheimen Schlüssels $PrivKey_B$.
2. Das Objekt $myMsg$ selbst wird intern mittels $SecretKey$ entschlüsselt und die zugehörige Kennung r wird verglichen.

   Falls die Entschlüsselung aufgrund eines falschen Schlüssels scheitert bzw. die Kennungen nicht übereinstimmen, so wurde der Agent manipuliert und die Auswertung wird abgebrochen[2], sonst weiter mit dem nächsten Schritt.
3. Falls das $receivedMsg$-Objekt auch (mit $SecretKey$) verschlüsselt ist, wird es ebenfalls entschlüsselt, und die mitgelieferte Kennung mit der geheimen Kennung r auf Gleichheit getestet. Im Fehlerfall wird wie in Punkt 2 verfahren.
4. Die Blackbox führt nun in ihrem geschützten Adressraum die objektspezifische Auswertungsmethode aus. Dabei werden die in $receivedMsg$ enthaltenen Daten in der sicheren Hardware verarbeitet bzw. evaluiert[3]: $myMsg' := myMsg.evaluate(receivedMsg)$

[2] Statt einer Fehlermeldung ist die Rückgabe des unveränderten ursprünglichen Objekts nach probabilistischer Verschlüsselung mit $PubKey_B$ denkbar. Das heißt, das Chiffrat ändert sich in jedem Fall, und es ist nach außen hin nicht ersichtlich, ob intern tatsächlich ein Datenaustausch stattgefunden hat. Oder das Auftreten des Fehlers (Art, Adresse) wird im Objekt zusätzlich protokolliert.

[3] Die Verarbeitung von unverschlüsselten Nachrichtenobjekten ohne Kennung r kann z.B. durch Setzen eines Schalters (Flag) in Objekt $myMsg$ verhindert werden. Die Blackbox verfährt dann wie im Fehlerfall bei Schritt 2.

5. Das resultierende Nachrichtenobjekt $myMsg'$ wird wieder mit $SecretKey$ verschlüsselt und als Rückgabewert and den Agenten übergeben.

Beispielhafter Ablauf der Interaktion aus Sicht der Agenten.

Zwei Agenten, A_j und A_k, können nun ohne Preisgabe von Informationen nach außen hin geheime Daten austauschen oder auswerten. Sie benötigen dazu dieselbe Kennung r als gemeinsames Geheimnis.

1. Es werden n Agenten erzeugt, Bezeichnung A_i mit $i = 1, \ldots, n$, die miteinander sicher interagieren sollen. Alle Nachrichtenobjekte enthalten dieselbe geheime Kennung r.
2. Agent A_i enthält nach Vorschrift 7.2 nun $\overline{myMsg^i{}_r}$ und $\overline{SecretKey_r}$;
3. A_j sendet $\overline{myMsg^j{}_r}$ and A_k.
4. A_k ruft die Schnittstellenmethode $evaluate$ der Blackbox auf, um die sichere Auswertung zu starten. Die Funktionalität (Schnittstelle) der Blackbox werde hier durch das Objekt $blackbox$ zu Verfügung gestellt, der Rückgabewert ist das aktualisierte, verschlüsselte Nachrichtenobjekt $\overline{myMsg^k{}_r}'$:

$$\overline{myMsg^k{}_r}' := \text{blackbox.evaluate}(\overline{SecretKey_r}, \overline{myMsg^k{}_r}, \overline{myMsg^j{}_r}) \tag{7.4}$$

Die so verwalteten Daten können entweder am Ende vom Agentenbesitzer entschlüsselt und ausgewertet werden, da dieser als einziger Außenstehender neben den Blackboxen das Geheimnis $SecretKey$ kennt. Oder die verschlüsselten Nachrichtenobjekte können alternativ durch verschlüsselte *Strategieobjekte* innerhalb der sicheren Hardware ausgewertet werden. Diese sichere Auswertung lässt sich weitgehend analog zum vorgestellten Verfahren der sicheren Interaktion realisieren.

7.3.2 Sichere Strategieauswertung

Ein praktischer Lösungansatz ist die Implementierung des *Strategieobjekt-Auswertungsdienstes* `SecureStrategyEvaluation` analog zur *sicheren Interaktion*, z.B. mittels einer SmartCard als Blackbox mit festem Public-Key-Schlüsselpaar. Durch Verschlüsselung mit dem öffentlichen Blackbox-Schlüssel lässt sich wieder erzwingen, dass die Daten nur innerhalb der Blackbox im Klartext vorliegen. Alle kritischen kryptographischen Operationen werden wiederum durch ein Kartenapplet in der sicheren Hardware ausgeführt und sind damit vor dem nicht vertrauenswürdigen Rechner verborgen. Das Applet ist entweder statisch auf der Karte vorhanden und legt damit die Implementierung der zu verwendenden Klassen fest, oder es wird Agenten unter bestimmten Umständen gestattet, eigene benutzerdefinierte Applets auf die Karte zu laden. Dieser Fall erfordert dann zusätzliche Überlegungen bezüglich der Sicherheit der Karte und der Verlässlichkeit der Applets.

Kapitel 8

Der Comparison Shopping Agenten Prototyp

In diesem Kapitel wird der Aufbau des Comparison-Shopping-Prototyps und die Integration der Sicherheitsdienste und Trusted Hardware besprochen.

8.1 Aufbau und Struktur

Die beiden wesentlichen *Entitäten* des Comparison Shopping Szenarios sind auf der Seite des Kunden der *Einkaufsagent* (engl. *buying agent* oder *comparison shopping agent*) und auf Seiten der Händler der *Verkaufsagent* (engl. *selling agent* oder *shop agent*). Weitere Entitäten sind nicht zwingend erforderlich, bieten sich aber zur Realisierung und Verwaltung der Infrastruktur an, wie beispielsweise eine *Verkaufsagentenfabrik* (engl. *shop factory*) oder eine *Schlüsselzentrale* (engl. *key factory*). Somit wurde das gewählte Comparison Shopping Modell auf die Entitäten Einkaufsagent, Verkaufsagent und Verkaufsagentenfabrik eingeschränkt (siehe Abb. 8.1). Diese reichen aus, um die grundlegenden Sicherheitsprobleme zu demonstrieren. Das *Schlüsselmanagement* ist rudimentär auf das Erzeugen, Speichern und Laden von Schlüsseln begrenzt – auf eine komplett ausgebaute Public-Key-Infrastruktur wurde verzichtet (siehe Paket `agents.security.util` in Appendix A.8). Um die Infrastruktur von Händleragenten auf verschiedenen Agletsplattformen komfortabel von einer Plattform aus einzurichten, kann der Agent `ShopFactory` als Hilfsmittel benutzt werden. Abbildung B.1 auf Seite 103 im Anhang zeigt das graphische Benutzerinterface des `ShopFactory`-Agenten, Abbildung B.2 den Steuerungsdialog der Einkaufsagenten.

Je nachdem, ob ein Agent mit graphischer Benutzeroberfläche oder stattdessen Konstruktor-basiert entworfen wird bzw. ob besondere Sicherheitsvorkehrungen getroffen werden oder nicht, ergeben sich verschiedene *Ausprägungen* der Agentenklassen:

$$\left\{ \begin{array}{c} \text{Ohne} \\ \text{Mit} \end{array} \right\} \text{GUI} \times \left\{ \begin{array}{c} \text{ohne} \\ \text{mit} \end{array} \right\} \text{Sicherheitsdienste} \times \left\{ \begin{array}{c} \text{ohne} \\ \text{mit} \end{array} \right\} \text{TrustedHardware.}$$

Die Anbindung der Verkaufsagenten an eine Händlerdatenbank und die damit verbundenen typischen Funktionen werden durch das Java Paket `aglets.csa.merchant` bereitgestellt (vgl. Anhang B.3). Die auf Kunden- und Händlerseite gemeinsam benötigten Datentypen sind in Paket

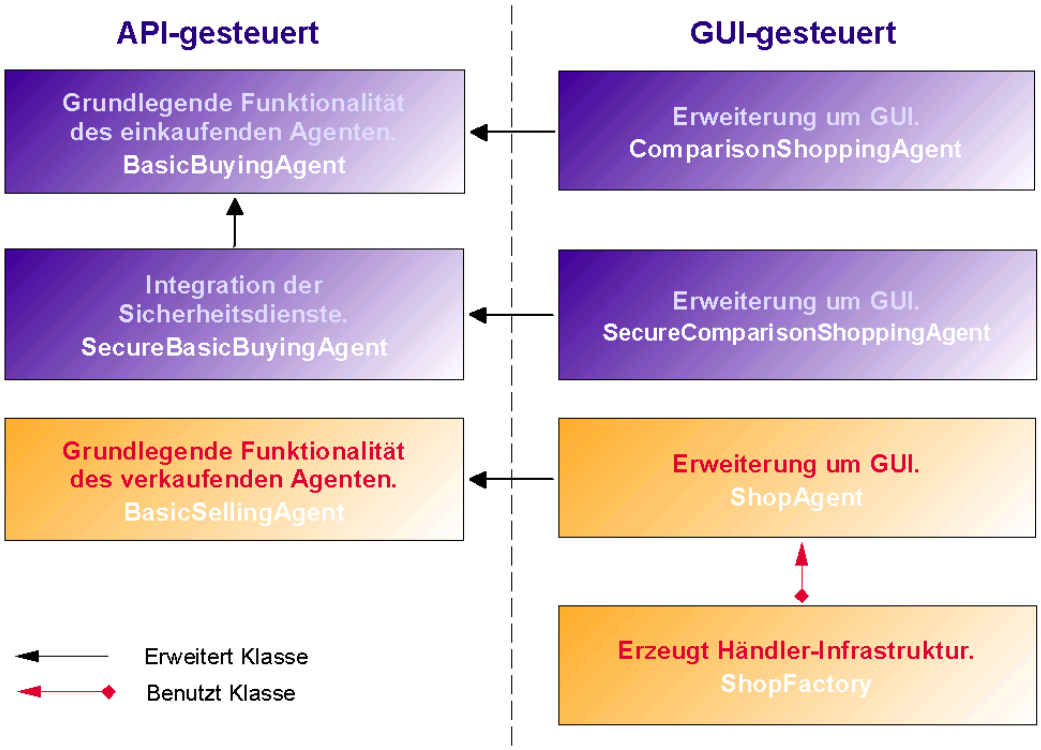


Abbildung 8.1: Entitäten im Comparison Shopping Modell.

`aglets.csa.datatypes` enthalten, das zur Umgehung der in Abschnitt 8.3.1 erwähnten Probleme bei der Klassenumwandlung (*Class Casting*) auf allen Plattformen in den Systemklassenpfad aufgenommen werden sollte. Eine Beschreibung der Klassen und Paketstruktur findet sich im Anhang B.2.

Die verschiedenen Ausprägungen von Einkaufsagenten und die gegenseitigen Abhängigkeiten werden in Abbildung 8.2 dargestellt, die Klassenstruktur des Verkaufsagenten in Abbildung 8.3.

8.2 Integration der Sicherheitsdienste

Die entwickelten und implementierten Sicherheitsdienste werden durch das Java-Paket `agents.security` bereitgestellt (siehe Appendix A). Ihre Integration in den Comparison-Shopping-Prototypen wurde wie folgt vorgenommen: Die anfallenden schützenswerten Daten im mobilen Agenten sind

1. die Produktbeschreibung (Auftragsinformation),
2. die gesammelten Angebote und
3. die festgelegten Wegestationen (engl. *itinerary*).

Auf diese Daten lassen sich die Sicherheitsdienste

1. `ReadOnlyContainer` (nur Lesen, Schutz der Integrität),

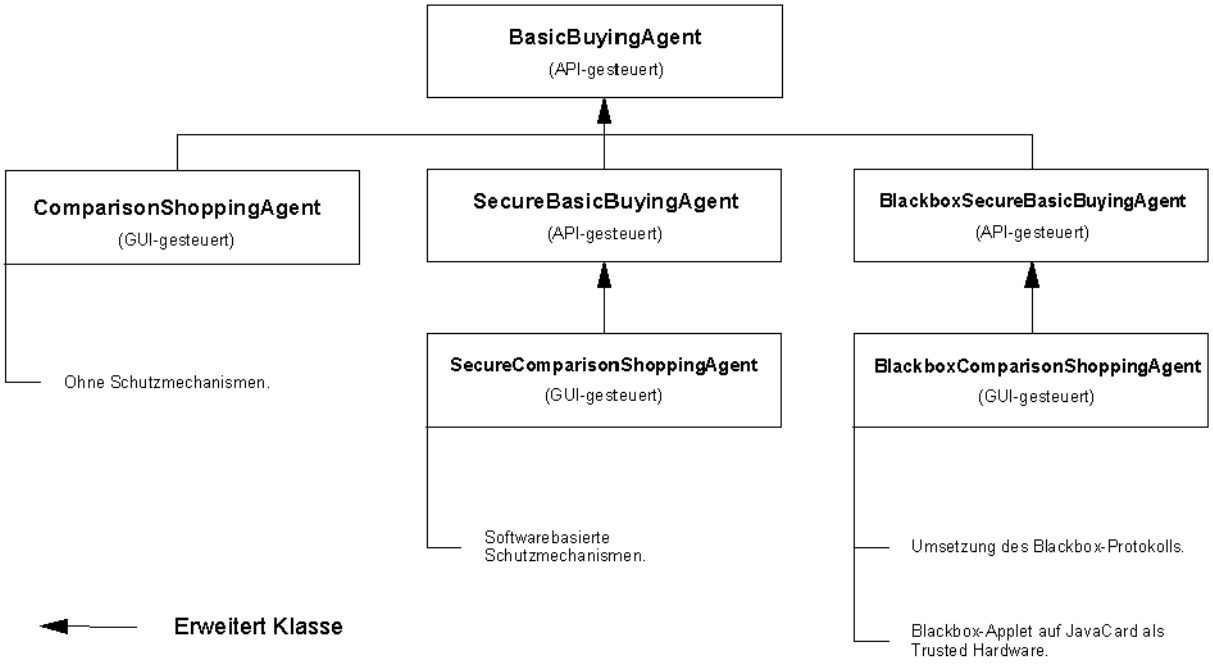


Abbildung 8.2: Vorstellung der verschiedenen Ausprägungen der Einkaufsagenten und der gegenseitigen Abhängigkeitsbeziehungen.

2. `SecretAppendOnlyContainer` (nur Hinzufügen, ohne Lesen, Schutz der Integrität und Vertraulichkeit) sowie

3. `SecureSeqPlanItinerary` als Erweiterung der Klasse `SeqPlanItinerary` (Schutz der Integrität)

direkt eins-zu-eins abbilden. Um das in Kapitel 6.3 vorgeschlagene *Blackbox-Protokoll* zu verwirklichen, wird der `SecretAppendOnlyContainer` mit der SmartCard-Realisierung des sicheren Stacks betrieben. Dadurch wird die Hashkette gemäß dem Protokoll durch die sichere Containerklasse intern realisiert, und die Aktualisierung und Überprüfung der kritischen Protokolldaten erfolgt ausschließlich innerhalb der Blackbox.

Der generische Entwurf der Sicherheitsdienste erlaubt, den `SecretAppendOnlyContainer` transparent zur Implementierung des Prototypen auch ohne Trusted Hardware zu betreiben; stattdessen kann eine reine Software-Implementierung eingesetzt werden. Dabei müssen jedoch wiederum Einbußen in punkto Sicherheit hingenommen werden, die mit Hilfe der sicheren Hardware gerade beseitigt werden sollten, wie in den Kapiteln 2.3 und 6.2 beschrieben.

8.3 Besonderheiten und Stolpersteine

Abschließend werden in diesem Kapitel Besonderheiten und Stolpersteine bei der Implementierung angesprochen.

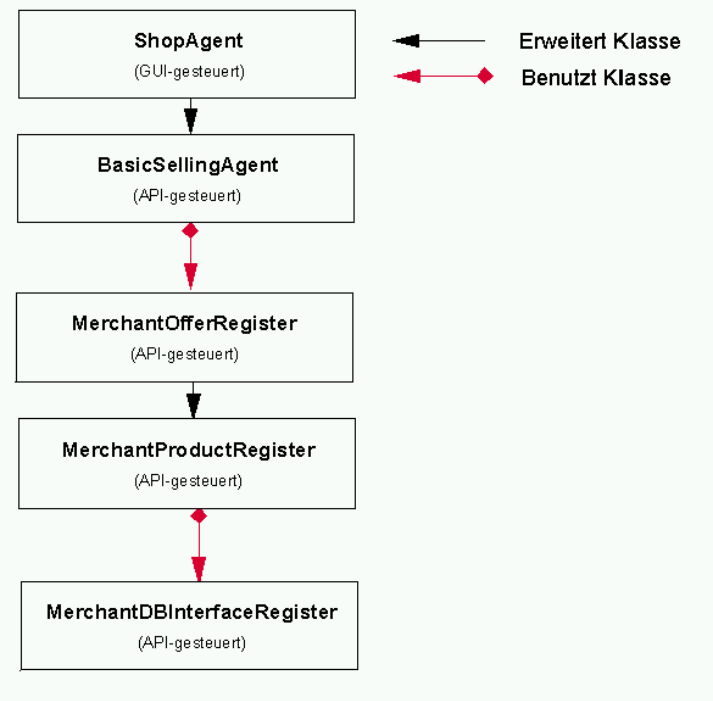


Abbildung 8.3: Klassenstruktur und Abhängigkeitsbeziehungen der Verkaufsagenten.

8.3.1 Class-Casting Konflikte

Bei der Migration von der aktuellen zu einer anderen Plattform tragen Mobile Agenten alle benötigten Klassen mit, die nicht Teil des Systemklassenpfads der Java-Installation (z.B. Sun JDK 1.1) oder des Aglets-Toolkits (IBM ASDK) sind und als `public`, d.h. „öffentlich“ deklariert wurden. Die Verwendung benutzerdefinierter Klassen bringt auf der Aglet-Plattform ein Problem bezüglich der Typenumwandlung mit sich, das nachfolgend beschrieben wird.

Klasse $Class_X$ sei eine solche öffentliche Nicht-Systemklasse, die der Agent $Agent_{fremd}$ bei seiner Migration zu einer anderen Plattform P mitführt. Auf Plattform P existiert ein lokal erzeugter Agent $Agent_{lokal}$.

- $Agent_{fremd}$ migriert zur Plattform P.
- Auf diesem Server wurde exakt dieselbe Klasse $Class_X$ bereits installiert und ebenfalls als öffentliche Nicht-Systemklasse exportiert.
- Im Laufe einer Kooperationsbeziehung zwischen $Agent_{fremd}$ und $Agent_{lokal}$ wird nun eine Objektinstanz der Klasse $Class_X$ über Nachrichtenaustausch durch Einkapselung in ein Nachrichtenobjekt vom Typ $Message$ ausgetauscht. Darin wird die Instanz der Klasse $Class_X$ in eine Instanz der Klasse $Object$ gespeichert, es findet dabei also eine Typumwandlung in die allgemeinste Objektoberklasse in Java statt.
- Der Empfänger will nun die im Nachrichtenobjekt als Argument enthaltene Objektinstanz extrahieren. Er erhält die Objektinstanz zuerst als Instanz der Klasse $Object$, die durch Klassencasting wieder in eine Instanz der Klasse $Class_X$ umgewandelt werden soll. Die Klasse

$Class_X$ ist aber auf dem System als Nicht-Systemklasse bekannt, und das Agletsystem meldet deswegen eine Sicherheitsbeschwerde und erzeugt eine Ausnahmemeldung wegen unerlaubter Klassenumwandlung (*Class Casting Exception*).

Das Problem beruht auf der Tatsache, dass jeder Agent seinen eigenen Klassenlader (engl. *class loader*) besitzt, der verhindert, dass dem Agenten auf fremden Plattformen z.B. korrupte Klassen untergeschoben werden, die die Sicherheit (Integrität oder Vertraulichkeit von Daten/Code) des Agenten korrumpieren und die vorhandenen Sicherheitsmechanismen unterminieren: So soll beispielsweise verhindert werden, dass als `private` deklarierte Felder eines Objektes zugänglich gemacht werden durch das Casten in ein modifiziertes (Unter-)Objekt mit denselben, jedoch als `public` deklarierten Feldern.

Im Aglets-Agentensystem konnte dieses Problem auf folgende Weise umgangen werden:

a) Man stellt die entsprechenden Nicht-Systemklassen als lokale Systembibliothek im Klassenpfad zur Verfügung, entweder (i) als zusätzliche Bibliothek der Java Installation oder (ii) als Teil der ASDK Systembibliothek des Aglets-Toolkits.

b) Man zerlegt das benutzerdefinierte Objekt in einfachere Systemklassen, überträgt diese und erzeugt beim Empfänger ein neues, inhaltlich äquivalentes Objekt der korrespondierenden Klasse, wie diese in dessen Klassenlader enthalten ist bzw. nachgeladen werden kann.

Beide Wege haben Nachteile. Die Variante a) lässt eine feinkörnige Verwaltung benutzerdefinierter Klassen vermissen; die entsprechende Klasse muss auf allen zu besuchenden Systemen als Systemklasse bekannt sein und kann nicht einfach mit dem Agenten mit exportiert werden. Lösung b) ist nur mit Einschränkung zu gebrauchen: So lässt sich beispielsweise ein signiertes Objekt nur bedingt zerlegen und wieder (als neues Objekt) zusammensetzen, ohne die Gültigkeit der Signatur zu verletzen. Der Autor stellt fest, dass es sich hierbei um ein *prinzipielles, nichttriviales Problem* handelt, das die Sicherheit und Integrität von mobilen Agentenanwendungen betrifft. Die im Beispiel auftretende Ausnahmesituation (engl. *exception*) ist ein *Sicherheitsmechanismus* der Agentenplattform, um unerwünschte und unter Umständen *sicherheitsgefährdende Typenumwandlungen* zu unterbinden. Es stellt sich die Frage, wie streng eine Plattform diese Typenumwandlung überwachen und *welche* Klassentypen (z.B. Systemklassen, lokal vorhandene oder frisch importierte Klassen) bei der Typenkonvertierung *wie* behandelt werden sollten. Eine restriktive Politik engt die Flexibilität des Systems (und des Klassenaustauschs) ein, wogegen eine laxe Verfahrensweise Türen für neue Sicherheitslücken öffnet. Dieser Bereich muss in der Zukunft noch intensiv erforscht werden.

Im Comparison-Shopping-Prototypen wurden beide Varianten evaluiert: Die auszutauschende Angebotsklasse `Offer` wurde in der sicheren Variante in den Systempfad aller Aglet-Installationen integriert, um signierte Angebote (`SignedObject(Offer)`) problemlos zwischen Einkaufs- und Verkaufsagenten „huckepack“ als Nutzdatenparameter von Nachrichtenobjekten austauschen zu können. Im Prototypen ohne kryptographische Schutzmaßnahmen wurde der Austausch von (unsignierten) Angeboten erfolgreich über den Weg der Transformation in einfache Systemdatentypen (Array aus Strings) und der Rekonstruktion durch Anwendung eines entsprechenden Konstruktors gelöst. Mit Hilfe der Zerlegung und Rekonstruktion von signierten Objekten, also Instanzen der Klasse `java.security.SignedObject` (Java 2), kann auch im sicheren Prototypen die Variante b) angewandt werden. Da der Aglet-Toolkit jedoch noch nicht Java 2 unterstützt, das jedoch ein wesentlich geschlosseneres Sicherheitsmodell bereitstellt, wurden die daraus benötigten Klassen als Stummel (eng. `stubs`) realisiert und eingebunden (siehe Anhang A.9). Eine Zerlegung und Rekonstruktion der Stummelklassen wurde wegen ihres vorübergehenden Charakters als nicht sinnvoll erachtet.

8.3.2 Änderung des Objektzustandes nach Migration des Agenten

Nach der Migration des Agenten von einem Tahiti Server zu einem anderen auf derselben Maschine (Betriebssystem Windows NT 4.0) änderte sich der Zustand (der serialisierten Darstellung) eines ansonsten finalen Objektes geringfügig: Ein spezielles Charakterzeichen in der `ByteArray`-Repräsentation war an zwei Stellen verändert, wobei das Objekt selbst weiterhin voll funktionsfähig blieb. Dieser Sachverhalt wurde bekannt, weil eine innerhalb der Stub-Klasse `SignedObject` zuvor berechnete (und als korrekt verifizierte) Signatur für das Datenobjekt nach der Rückkehr des Agenten als ungültig betrachtet wurde, obwohl das Objekt selbst nicht aktiv manipuliert worden war. Das Phänomen in Zusammenhang mit der Agentenmigration konnte nicht erklärt werden. Da es sich bei der Klasse jedoch nur um eine Stummelklasse für die nicht verfügbare Klasse SignedObject aus dem Java 1.2 Sicherheitspaket handelte (der Aglets-Toolkit verlangt Java 1.1), ist das Problem nur von geringer Relevanz und wurde an das Aglets-Entwicklungsteam am IBM Forschungslabor in Tokyo weitergeleitet.

Kapitel 9

Zusammenfassung und Ausblick

Zum Schluss sollen die theoretischen und praktischen Beiträge dieser Diplomarbeit zum Thema Sicherheitsdienste für Mobile-Agenten-Anwendungen im elektronischen Handel zusammengefasst werden.

Zunächst wurden die *Sicherheitsbedürfnisse* von mobilen Agenten analysiert und grundlegende Sicherheitsdienste identifiziert. Dies führte zum Entwurf und zur Implementierung *generischer Datenstrukturen* mit hohem Abstraktionsgrad, die die Integrität und Vertraulichkeit von Agentendaten schützen.

Mit Blick auf typische Aufgabenstellungen im elektronischen Handel wurden E-Commerce-spezifische Datenstrukturen entworfen und darüberhinaus Dienste bereitgestellt, welche die ungestörte und unbeobachtete, d.h. *sichere Auswertung vertrauenswürdiger Daten* sowie die *korrekte Ausführung* von kritischen Aktionen in *unsicheren Umgebungen* gestatten. Dazu wurde in dieser Arbeit der Einsatz *sicherer Hardware* vorgeschlagen. Diese Dienste sind *Verfeinerungen* der vorgestellten allgemeinen Sicherheitsdienste für mobile Agenten. Anhand des Leitbeispiels der preisvergleichenden Produktrecherche wurde außerdem formal ein sicheres Protokoll entwickelt, das durch den Einsatz von *Trusted Hardware* die Integrität der Rechenergebnisse mobiler Agenten auf unsicheren Plattformen schützt. Es konnten weiterführende Einsatzszenarien – u.a. die Verwirklichung autonomer und nicht abstreitbarer Agentenentscheidungen – aufgezeigt und ein mögliches Geschäftsmodell umrissen werden. Die Bereitstellung der sicheren Hardware erfolgte exemplarisch mit Hilfe von SmartCard-Technologie: Dazu wurde ein JavaCard Applet zur Ausführung sicherheitskritischer Operationen selbständig entworfen und programmiert. Die resultierenden JavaCard-Dienste lassen sich transparent in die entwickelten Sicherheitsdienste integrieren.

Zur Demonstration der Eignung und Durchführbarkeit der Sicherheitskonzepte wurde das Leitbeispiel auf der Aglets Agentenplattform in der Programmiersprache Java implementiert und in Form des *Comparison Shopping Prototypen* praktisch umgesetzt. Die Realisierung des sicheren Blackbox-Protokolls fand dabei durch die Integration der erarbeiteten Sicherheitsdienste in den Prototypen statt, kombiniert mit dem Einsatz der SmartCard-Implementierung als sichere Hardware.

9.1 Ausblick

Die Auseinandersetzung mit den mobilen Agenten zeigt, dass diese Technologie ein großes Potential für zukünftige Entwicklungen und Einsatzszenarien bietet, nicht nur im elektronischen Handel. In einer Welt der verteilten Systeme sind die Agenten jedoch – wie gezeigt – einer Vielzahl von Gefahren ausgesetzt, und der Aspekt der Sicherheit und des Schutzes mobiler Agenten ist und bleibt *das* Kernproblem. Diese Arbeit erleichert die praktische Realisierung von mobilen Agentensystemen, zeigt neue Richtungen auf und bringt die mobilen Agenten dem praktikablen Einsatz einen Schritt näher. Dennoch ist das Grundproblem, der Schutz der Ausführung von Agenten auf nicht vertrauenswürdigen Plattformen, nicht allgemein gelöst; vielmehr wird durch den empfohlenen Einsatz sicherer Hardware zusätzlich die Einbeziehung einer dritten, vertrauenswürdigen Partei erforderlich. Es ist heute noch nicht bekannt, ob sich für dieses Problem überhaupt eine generelle Lösung finden lässt, oder ob es sich hierbei um eine dem Mobilen-Agenten-Paradigma inhärente Problematik handelt. Aufschlüsse dazu erhofft man sich von der weiteren Forschung, insbesondere von den theoretischen Arbeiten im Bereich der verschlüsselten Funktionen (*Encrypted Functions*). Man darf gespannt sein, was zukünftige Arbeiten ans Tageslicht bringen werden.

Danksagung

Abschließend möchte ich mich bei meinen Kollgegen am IBM Forschungslabor in Rüschlikon herzlich für die freundliche Aufnahme und die abwechslungsreichen Monate im Labor bedanken. Die umfassende Bereitschaft zur Erkundung neuer Wege und Alternativen und das positive Arbeitsklima waren sehr motivierend. Besonderen Dank möchte ich meinem Betreuer bei IBM, Dr. Günter Karjoth, aussprechen, der stets bereitwillig und mit Ausdauer für fachliche Auseinandersetzungen und Diskussionen zu Verfügung stand, und für die fruchtbaren Anregungen bei der Entwicklung des Blackbox-Protokolls.

Weiterer Dank gebührt Prof. Dr. Thomas Beth und Dr. Willi Geiselmann vom Lehrstuhl für Algorithmen und Kognitive Systeme an der Universität Karlsruhe (TH) für die freundliche und problemlose Betreuung.

Anhang A

Pakete und Klassen der implementierten Sicherheitsdienste

In diesem Kapitel werden die wichtigsten entworfenen Klassen und Pakete der Sicherheitsdienste kurz beschrieben. Alle vorgestellten Implementierungen liegen in der Programmiersprache Java vor. Die Programmierung aller Klassen der Sicherheitsdienste umfasst ca. 16000 Zeilen Programmcode. Die vollständige Dokumentation liegt in Form von automatisch erzeugten Klassenbeschreibungen im HTML-Format vor, die mit dem Java Dokumentationswerkzeug `JavaDoc` direkt aus den vollständig kommentierten Quelldateien generiert werden können.

A.1 Paket `agents.security`

Im `agents.security` sind drei allgemeine Schnittstellendefinitionen enthalten, die von diversen anderen Klassen implementiert werden.

1. Interface `CipherLockable`

   Schnittstellendefinition für Objekte, die mittels eines Cipher-Objekts (`java.security.Cipher`) aus Java 1.2 abschließbar sein sollen, das heißt mit Hilfe eines geheimen Schlüssels (Schutz der Vertraulichkeit).
   Anwendungsbeispiel: Die Containerklasse `LockableContainer` implementiert diese Schnittstelle.

2. Interface `TimeLockable`

   Schnittstellendefinition für Objekte, die mittels eines passenden signierten Zeitstempels freigeschalten werden (zeitweiliger Schutz der Vertraulichkeit).

3. Interface `TokenLockable`

   Schnittstellendefinition für Objekte, die mittels eines signierten Tokens freigeschalten und damit durch Entschlüsselung verfügbar gemacht werden (zeitunabhängiger Freischalttoken).

A.2 Paket agents.security.stack

Dieses Paket enthält die entworfenen abstrakten Stack-Klassen, deren sichere Implementierung durch die wohldefinierte Schnittstelle StackCryptoService verborgen wird.

- Klasse IntegrityStack

  Sicherer integritätsgeschützter Stack. Sicherheitskritische Operationen werden durch eine bei der Stack-Initialisierung mitgeliefterte Stack-Engine bereitgestellt, deren Implementierung der Schnittstellendefinition StackCryptoServices folgt.

- Klasse PushOnlyIntegrityStack

  Sicherer integritätsgeschützter Stack, der jedoch nur das Hinzufügen von Objekten zum den Stack erlaubt, aber nicht die Entnahme. Sicherheitskritische Operationen werden durch eine bei der Stack-Initialisierung mitgeliefterte Stack-Engine bereitgestellt, deren Implementierung der Schnittstellendefinition StackCryptoServices folgt.

- Klasse StackCryptoServices

  Schnittstelle für die Implementierung von sicheren Stack-Engines, die den Integritätsschutz der Stackdaten realisieren und die sicherheitskritischen Daten verwalten (siehe *Integrity Data Unit* in Kapitel 5.3.1 auf Seite 36).

Das Paket enthält außerdem einige von Hilfs- und Ausnahmeklassen, die intern bei der Realisierung der Stackklassen Verwendung finden.

A.3 Paket agents.security.stack.javacard

Dieses Paket enthält die im Rahmen der JavaCard-Programmierung erstellten Programme.

- Klasse Blackbox

  Die Klasse Blackbox ist ein JavaCard-Applet-Programm, das auf der SmartCard installiert wird und welches die Funktionalität der in den sicheren Stackklassen und im Blackbox-Protokoll benötigten sicheren Hardware bereitstellt.

- Klasse BlackboxGUI

  Die Anwendung BlackboxGUI diente als Testwerkzeug für das Blackbox-Applet, da keine Simulationsumgebung verfügbar war. Die Steuerung erfolgt mittels graphischem Dialogfenster. Die Kontrollausgaben erfolgen sowohl in einem Protokollfenster als auch auf der Konsole.

- Klasse BlackboxConnection

  Die Klasse BlackboxConnection realisiert einen Verbindungsmanager, der Methoden zur einfachen Benutzung des Blackbox-Applets bereit stellt und die Verbindung zum Kartenterminal verwaltet.

- Klasse JavaCardAccess

  Das Testwerkzeug JavaCardAccess prüft die Verfügbarkeit der benötigten Kartenterminalfunktionen und des Blackbox-Applets.

A.4 Paket agents.security.stack.impl

Das Paket agents.security.stack.impl enthält Implementierungen von Stack-Engines, die der StackCryptoServices-Schnittstelle folgen.

- Klasse StackCryptoServicesImpl

  Softwarebasierte Implementierung der StackCryptoServices-Schnittstelle.

- Klasse StackCryptoServicesBlackboxImpl

  SmartCard-basierte Implementierung der StackCryptoServices-Schnittstelle. Verwendet das Blackbox-JavaCard-Applet über die Vermittlung durch den Verbindungsmanager BlackboxConnection.

A.5 Paket agents.security.container

Das Paket agents.security.container enthält eine Auswahl von Implementierungen der entwickelten generischen sicheren Containerklassen zum Schutz der Integrität und/oder Vertraulichkeit von (Agenten-)Daten.

- Klasse AppendOnlyContainer

  Container, der nur lesenden oder hinzufügenden Zugriff gestattet.

- Klasse LockableContainer

  Abschließbarer Container, der im unverschlossenen Zustand beliebige Operationen auf den Daten erlaubt. Diese Containerklasse benützt die Klasse java.util.Hashtable für das Ablegen und Suchen von Objekten, erweitert diese Klasse jedoch nicht (keine Vererbung).

- Klasse ReadOnlyContainer

  Container, der nur lesenden, aber keinen die verändernden Zugriff ermöglicht.

- Klasse RemoveOnlyContainer

  Container, der nur lesenden oder hinzufügenden Zugriff gestattet.

- Klasse SecretAppendOnlyContainer

  Erweitert die Containerklasse AppendOnlyContainer um automatische Verschlüsselung der abgelegten Daten mit Hilfe eines bei der Erzeugung festgelegten öffentlichen Public-Key-Schlüssels.

- Klasse WeakAppendOnlyContainer

  Bietet dieselbe Funktionalität wie der AppendOnlyContainer, operiert aber mit klassischer rekursiver Prüfsumme für den Einsatz ohne sichere Hardware.

A.6 Paket agents.security.ecommerce

Spezielle E-Commerce Dienste orientieren sich sehr stark an der Anwendung und am zu realisierenden System. Im Zuge des Comparison Shopping Szenarios wurde dabei ein sicherer Containerdienst mit unterliegender sicherer Hardware (JavaCard) implementiert.

Das Paket agents.security.ecommerce enthält Schnittstellendefinitionen für weiterführende Dienste wie z.B. für den Einsatz in elektronischen Auktionen. Sie sind aber eher des beispielhaften Charakters wegen hier aufgeführt. Zwar wurden diese Dienste im Kapitel 5.4 auf Seite 41 und auf der Seite 81 im Kapitel „Sichere Ausführung und Berechnung“ formal entwickelt und beschrieben, eine zusätzliche praktische Implementierung hätte den Rahmen der Diplomarbeit jedoch gesprengt.

Eine mögliche Klassenstruktur zur sicheren Strategieauswertung ist:

- Interface SecureStrategyEvaluation

  Schnittstelle, die die sichere Auswertung von Strategieobjekten definiert.

- Klasse AuctionBettingStrategy

  Implementierung der SecureStrategyEvaluation-Schnittstellendefinition für elektronische Auktionen, z.B. als Niedrigpreisvariante mit SmartCard-Unterstützung, ähnlich der Blackbox-Realisierung.

- ComparisonShoppingStrategy

  Implementierung der SecureStrategyEvaluation-Schnittstellendefinition für Comparison Shopping Agenten, die unterwegs nicht nur Daten sammeln, sondern dabei auch eine bestimmte Vorgehensweise (Strategie) verfolgen, um zu besseren Resultaten zu gelangen.

Ein auf die besprochene Erweiterung des Blackbox-Protokolls zugeschnittener sicherer Dienst, der das jeweils beste Angebot bereits unterwegs bestimmt, könnte durch folgende Klassen realisiert werden:

- Interface SecureComputation

  Abstrakte Dienstspezifikation zur Durchführung sicherer Berechnungen auf unsicheren Plattformen.

- Klasse SecureBestOfferEvaluation

  Sicherer Berechnungsdienst zur Bestimmung des aktuell besten Angebotes auf unsicheren Plattformen. Folgt der SecureComputation-Schnittstelle.

- Klasse SecureBestOffer

  Abstrakter Datentyp, den Angebotsobjekte implementieren müssen, um mit dem SecureBestOfferEvaluation-Dienst ausgewertet werden zu können.

Denkbare Implementierungen von funktionsspezifischen Angebotsklassen im elektronischen Handel sind:

- Klasse `TimeLockedOffer`

  Angebotsobjekt, das mit einem Zeitschloss ausgestattet ist und die TimeLockable-Schnittstellendefinition erfüllt.

- Klasse `TokenLockedOffer`

  Angebotsobjekt, dessen Verfügbarkeit über einen Freigabetoken kontrolliert wird.

A.7 Paket `agents.security.aglets`

Dieses Paket enthält die Erweiterungen der `Itinerary`-Klassen des IBM Aglet-Toolkits um integritätsschützende Maßnahmen.

- Klasse `SecureSeqItinerary`

  Die Aglet-Klasse `com.ibm.agletx.util.SeqItinerary` wurde um ein „Einfrieren“ des Objektzustandes und die Möglichkeit der späteren Zustands-Verifikation erweitert, um die Integrität des Wegwahlobjektes zu schützen.

- Klasse `SecureSeqPlanItinerary`

  Analog zur Klasse `SecureSeqItinerary`: Die Aglet-Klasse `com.ibm.agletx.util.SeqPlanItinerary` wurde um ein „Einfrieren“ des Objektzustandes und die Möglichkeit der späteren Zustands-Verifikation erweitert, um die Integrität des Wegwahlobjektes zu schützen.

A.8 Paket `agents.security.util`

Das Paket `agents.security.util` enthält verschiedene Hilfsdienste, die nicht direkt mit den Sicherheitsdiensten in Verbindung stehen:

- Klasse `IOManagement`

  Die Klasse `IOManagement` bietet einfache Methoden zur Dateiverwaltung an.

- Klasse `KeyManagement`

  Die Klasse `KeyManagement` wird zur Schlüsselverwaltung eingesetzt und erlaubt z.B. das Laden und Speichern von Schlüsseln oder Schlüsselpaaren. Dazu wird intern die Klasse `IOManagement` verwendet.

- Klasse `Util`

  Die Klasse `Util` enthält häufig benötigte Routinen, etwa zur Serialisierung von Objekten oder zum Vergleich oder Kopieren von Bytearrays usw.

A.9 Paket agents.security.stubs

Dieses Paket ist wegen der exportverbotsbedingten Nichtverfügbarkeit der Java Cryptography Extension (JCE) und gewisser Klassen der Java 1.2 Sicherheits-API notwendig geworden.

Es definiert und implementiert z.B. die Klassen SealedObject, Cipher, SignedObject als Stubs, d.h. die Funktionalität wird teilweise nur simuliert (Dummy-Objekte).

A.10 Paket agents.security.impl

In diesem Paket wird eine eigene Java-basierte Kryptobibliothek angeboten, die eine einfachere API als die des java.security-Pakets für in dieser Arbeit häufig benötigte kryptographische Funktionen bereitstellt.

Anhang B

Pakete und Klassen des Comparison-Shopping-Prototyps

In diesem Kapitel werden alle Pakete des Comparison-Shopping-Prototypen aufgelistet und die implementierten Klassen kurz beschrieben. Der Umfang der Programmiertätigkeit zur Realisierung des Prototypen in Java beträgt dabei ca. 12000 Zeilen Programmcode. Eine ausführlichere Beschreibung der Methoden und Eigenschaften liefert die mit dem Werkzeug `JavaDoc` aus dem Quellcode erzeugbare Online-Dokumentation.

B.1 Paket `aglets.csa`

Das Paket `aglets.csa` enthält die verschiedenen grundlegenden Agentenklassen, die im Rahmen des Comparison Shopping Szenarios auftreten. Alle Agenten wurden als IBM Aglets realisiert, in verschiedenen Ausprägungen (z.B. mit und ohne Integration der implementierten Sicherheitsdienste, die Verkaufsagentenfabrik).

- Klasse `BasicBuyingAgent`

  Auf die Grundfunktionen reduzierter Einkaufsagent. Ohne graphische Benutzerschnittstelle (engl. *graphical user interface*, GUI). Keine Sicherheitsdienste integriert.

- Klasse `ComparisonShoppingAgent`

  Mit graphischer Benutzerschnittstelle und Dialogfähigkeiten ausgestatteter Einkaufsagent. Erweitert die Klasse `BasicBuyingAgent`.

- Klasse `SecureBasicBuyingAgent`

  Auf die Grundfunktionen reduzierter Einkaufsagent. Ohne graphische Benutzerschnittstelle (engl. *graphical user interface*, GUI), aber mit integrierten Sicherheitsdiensten (`AppendOnlyContainer`, `SecurePlanSeqItinerary`).

- Klasse `SecureComparisonShoppingAgent`

  Mit graphischer Benutzerschnittstelle, Dialogfähigkeiten und Sicherheitsdiensten versehener Einkaufsagent. Erweitert die Klasse `SecureBasicBuyingAgent`.

- Klasse `BasicSellingAgent`

  Auf die Grundfunktionen reduzierter Verkaufsagent. Ohne graphische Benutzerschnittstelle. Es wurde keine Integration von Sicherheitsdiensten auf Händlerseite vorgenommen, da diese Agenten statischen Charakter besitzen und der Schutz der Einkaufsagenten zu Demonstrationszwecken ausreichend ist.

- Klasse `ShopAgent`

  Um graphische Benutzerschnittstelle und Dialogfähigkeit ergänzter Verkaufsagent. Stellt Erweiterterung der Klasse `BasicSellingAgent` dar.

- Klasse `ShopFactory`

  Verkaufsagentenfabrik. Agent mit graphischem Benutzerdialog zur Erzeugung und Aktivierung von Verkaufsagenten auf verschiedenen zu spezifizierenden Plattformen.

- Interface `CSADefinitions`

  Festlegungen und Konstanten, die sich die Agenten und Objekte im Comparison Shopping Modell teilen.

- Exception `CSAExecutionException`

  Ausnahmeklasse, die sich von der Java-Klasse `java.lang.RuntimeException` ableitet. Sie wird bei unerwarteten Ausnahmesituationen zur Laufzeit geworfen und muss vom Programmierer nicht explizit behandelt werden.

- Exception `CSASecurityException`

  Ausnahmeklasse für sicherheitskritische Ausnahmesituationen. Sie leitet sich von der Java-Klasse `java.lang.SecurityException` und damit auch indirekt von der Klasse `java.lang.RuntimeException` ab. Sie muss daher vom Programmierer ebenfalls nicht explizit im Code behandelt werden.

B.2 Paket `aglets.csa.datatypes`

Dieses Unterpaket enthält die benutzerdefinierten Datentypen, die im Sinne der Comparison Shopping Anwendung von den Agenten neben den Standardpaketen zusätzlich benötigt werden.

- Klasse `Catalogue`

  Allgemeine Katalogklasse. Erweitert `java.util.Hashtable`.

- Klasse `CatalogueEntry`

  Abstrakte Definition eines Katalogeintrages. Legt die erforderlichen Eigenschaften und Methoden von Katalogeinträgen fest, wie z.B. die Darstellung als String oder das Enthalten eines beschreibenden Deskriptors.

- Klasse `CatalogueEntryDefinitions`

  Definiert Konstanten und Defaulteinstellungen für Katalogeinträge.

- Klasse `Product`

  Diese Klasse repräsentiert ein Produkt eines Händlers und leitet sich von der abstrakten Katalogeintragsklasse `CatalogueEntry` ab.

- Klasse `ProductDescriptor`

  Die Klasse `ProductDescriptor` verbirgt die Art und Implementierung von eindeutigen Bezeichnern für Produkte.

- Klasse `ProductCatalogue`

  Produktkatalog mit Methoden zum Erzeugen und Verwalten von Produkten. Leitet sich von der allgemeinen Katalogklasse aus `Catalogue` ab.

- Exception `IncompatibleProductsException`

  Ausnahmeklasse für Operationen, die aufgrund inkompatibler Produkte nicht ausführbar sind (z.B. versuchter Vergleich von „Äpfeln“ mit „Birnen“).

- Klasse `Offer`

  Klasse für Händlerangebote. Beinhaltet neben den Produktdaten auch Preisinformation, Händlerdaten und Gültigkeitsinformationen. Erweitert die Klasse `Product`.

- Klasse `OfferCatalogue`

  Angebotskatalog mit Methoden zum Erzeugen und Verwalten von Angeboten. Leitet sich von der Produktkatalogklasse `ProductCatalogue` ab.

- Klasse `OfferVector`

  Erweitert die Klasse `java.util.Vector` um angebotsspezifische Methoden, z.B. Bestimmung des günstigsten Angebotes.

B.3 Paket `aglets.csa.merchant`

Im Unterpaket `aglets.csa.merchant` wird die Anbindung der Shop-Agenten an eine Händlerdatenbank simuliert. Es erfolgt eine schrittweise Abbildung auf eine unterliegende Dummy-Datenbank, die durch die Klasse `MerchantDBInterface` repräsentiert wird:

- Klasse `MerchantOfferRegister`

  Verwaltung der verschiedenen Angebotskataloge. Suchen von Produkten und Kategorien. Schlüsselverwaltung der Händlerschlüssel. Signieren von Angeboten on-the-fly. Bietet dem Händleragenten `ShopAgent` Zugriff auf aktuelle Angebote.

- Klasse `MerchantProductRegister`

  Verwaltung der verschiedenen Produktkataloge. Suchen von Produkten und Kategorien. Benutzt das Datenbankinterface `MerchantDBInterface`.

- Klasse `MerchantDBInterface`

  Diese Klasse simuliert den Zugriff auf eine unterliegende Datenbank. Die Angebots- und Produktdaten sind hartkodiert. Die Preise werden bei der Erzeugung des Objektes jeweils zufällig neu generiert.

B.4 Abbildungen

Dieses Kapitel enthält Screenshots des im Rahmen dieser Diplomarbeit entwickelten und implementierten Comparison-Shopping-Prototyps.

Abbildung B.1 zeigt das Dialogfenster der Verkaufsagentenfabrik `ShopFactory`. In Abbildung B.2 ist der Steuerungsdialog der Einkaufsagenten dargestellt.

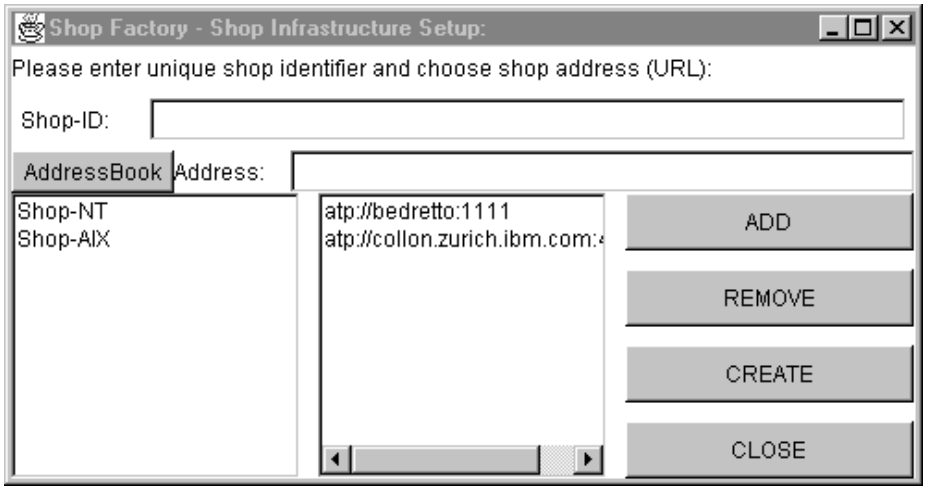


Abbildung B.1: Graphischer Benutzerdialog des `ShopFactory`-Agenten.

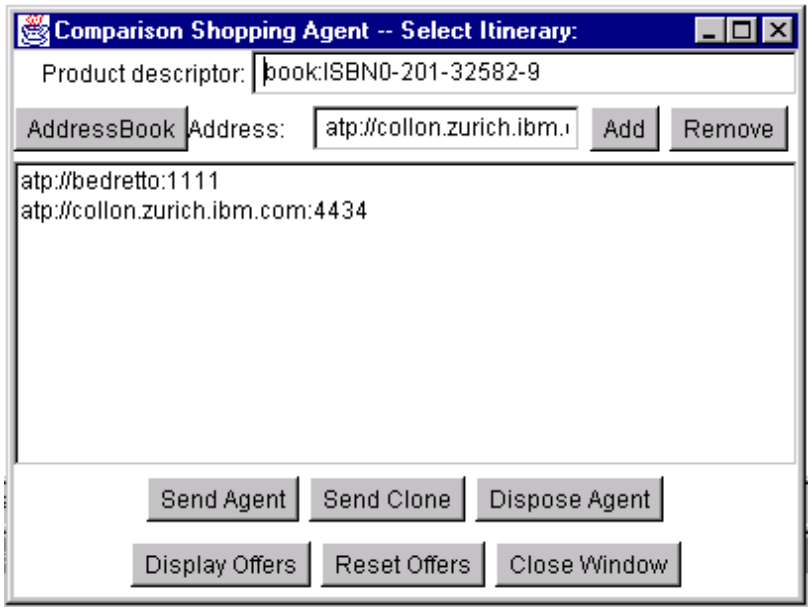


Abbildung B.2: Graphischer Benutzerdialog der Einkaufsagenten.

Abbildung B.3 zeigt die Dialogboxen von verschiedenen Tahiti-Aglet-Servern unter Windows NT und AIX (ganz unten), mit eingeblendetem graphischem Steuerungsdialog des Einkaufsagenten (linke Dialogbox) und dem Kontrollfenster der Verkaufsagenten-Herstellungsfabrik (`ShopFactory`).

Abbildung B.4: Start des sicheren Einkaufsagenten und dessen Steuerung über ein graphisches Dialogfenster. Dazu die Server-Console mit Ausgaben zur Initialisierung des sicheren Stacks auf der SmartCard. Im Hintergrund der Heimat-Agletserver (Port 4434) sowie drei Marktplatz-Server (Port-Adressen 1111, 2222, 3333).

Abbildung B.5 zeigt den Einkaufsagenten während der Verhandlung mit einem Verkaufsagenten auf einer elektronischen Marktplattform. Die zugehörige Server-Console protokolliert die Ausgaben bei der Speicherung des erhaltenen Angebotes im `SecretAppendOnlyContainer` mit unterliegender Blackbox-Stack-Engine.

Abbildung B.6 bildet den Einkaufsagenten während der Verhandlung mit zwei Verkaufsagenten auf einer weiteren elektronischen Marktplattform ab. Ebenfalls ist die Server-Console mit protokollierten Meldungen zum Anhängen der Angebote an den Container und zur erfolgreichen Verifikation des Containerzustandes auf der SmartCard zu sehen.

Abbildung B.7: Nach der Rückkehr des Einkaufsagenten zur Heimatplattform werden die Integrität der gesammelten Angebote mit Hilfe der SmartCard geprüft. Anschließend werden alle Angebote angezeigt und das beste Angebot wird hervorgehoben. Die Server-Console zeigt die protokollierten Ausgaben der erfolgreichen Integritätsprüfung des `AppendOnlyContainers` sowie alle erhaltenen Angebote und das daraus bestimmte beste Angebot.

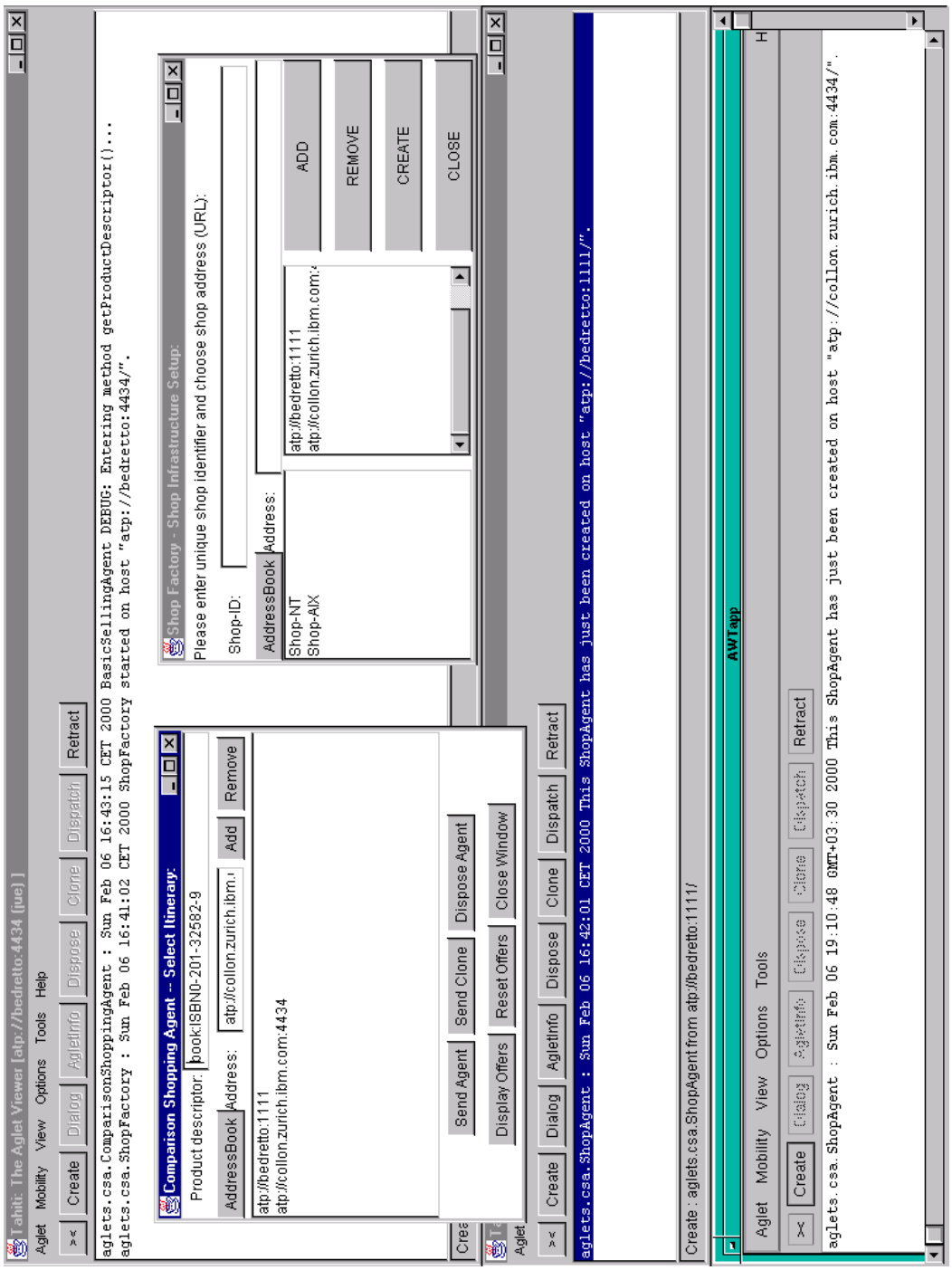


Abbildung B.3: Tahiti Aglet Server unter Windows NT und AIX.

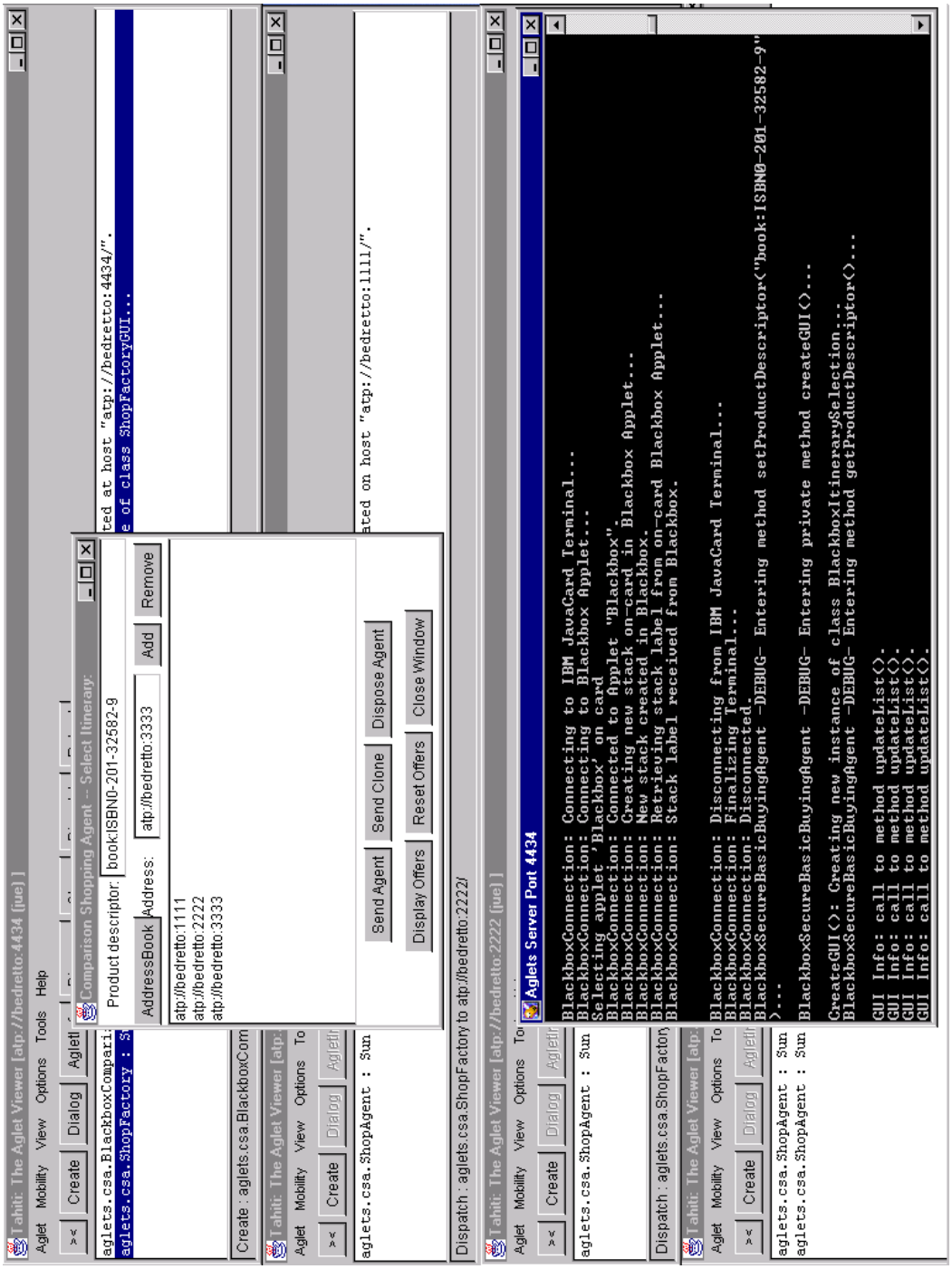


Abbildung B.4: Start und Steuerung des sicheren Einkaufsagenten

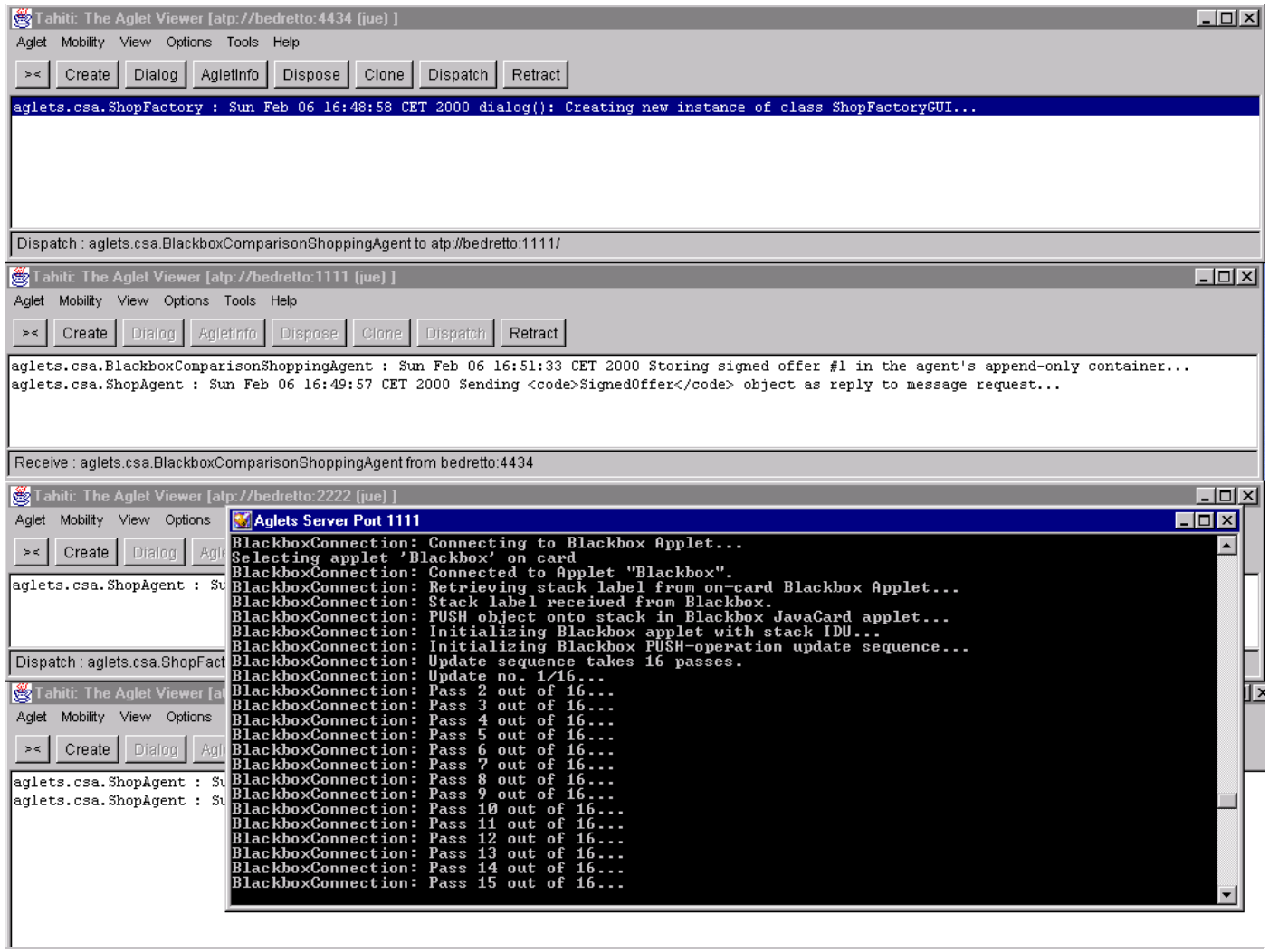


Abbildung B.5: Einkaufsagent während der Verhandlung mit einem Verkaufsagenten.

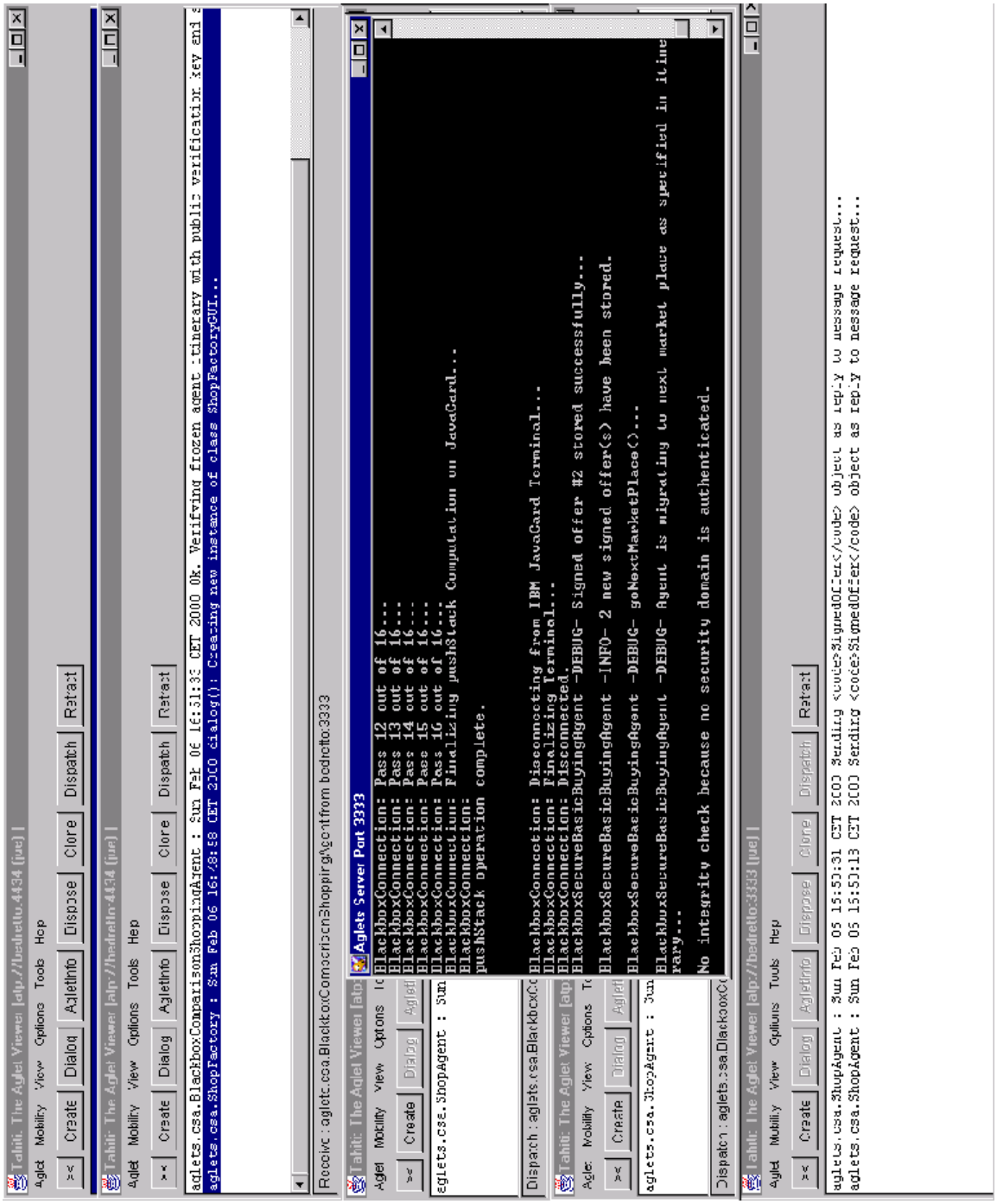


Abbildung B.6: Einkaufsagent während der Verhandlung mit zwei Verkaufsagenten.

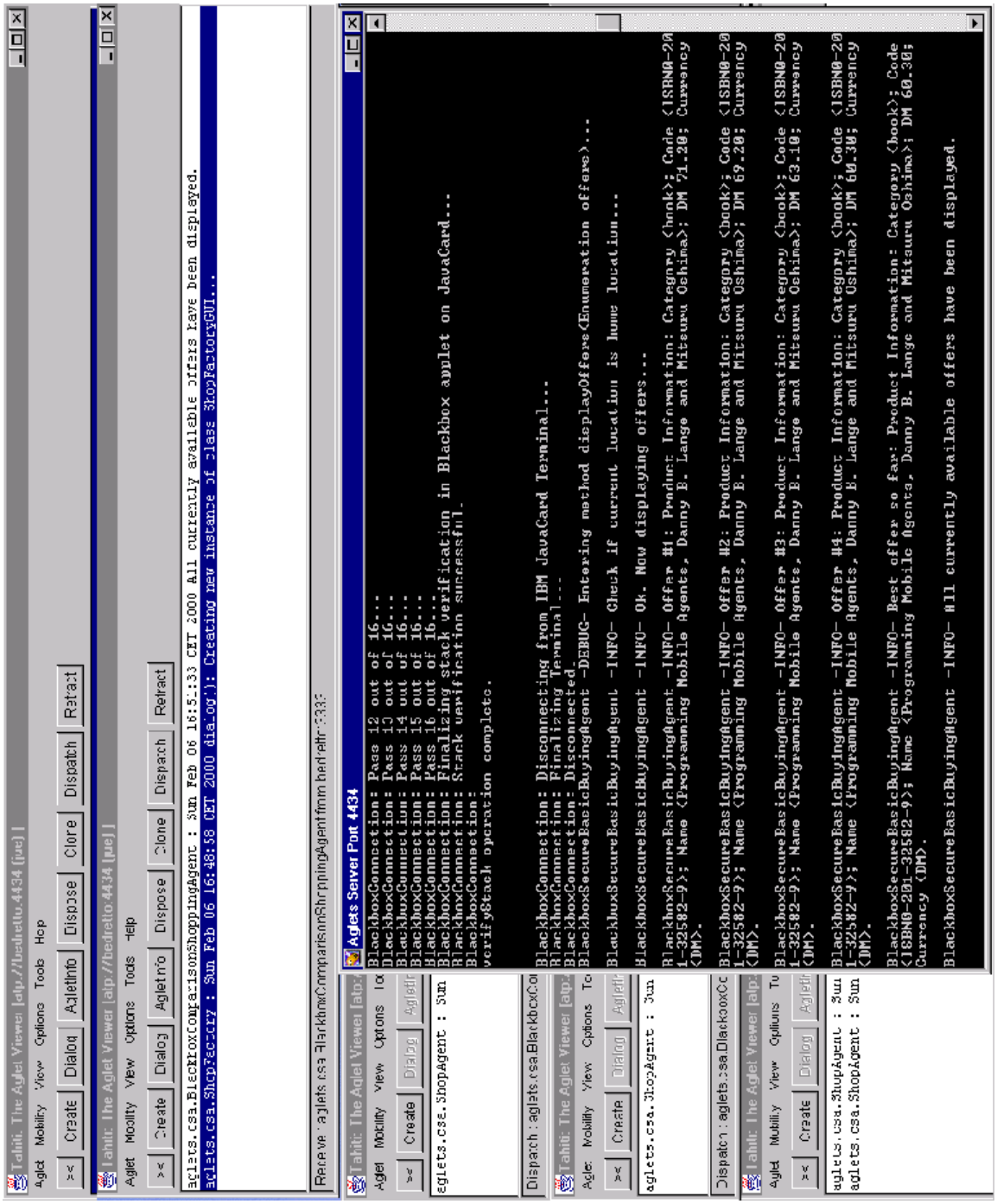


Abbildung B.7: Rückkehr des Einkaufsagenten zur Heimatplattform und Anzeige der Angebote.

Anhang C

Nicht detektierbarer Angriff auf den `AppendOnlyContainer` nach Karnik

Karniks Realisierung des `AppendOnlyContainers` weist eine *wesentliche Schwäche* auf: Die verwendete Prüfsumme (Feld `checkSum`) zum Wahren der Integrität der Datenstruktur kann von einem Angreifer unentdeckt manipuliert werden. Die Berechnung der Prüfsumme des Behälters ist folgende:

- Initialisierung des Containers:
  $checkSum := PubKey_A(nonce)$
  Der geheime Initialisierungswert $nonce$ wird mit dem öffentlichen Schlüssel $PubKey_A$ des Agenten verschlüsselt.
- Anhängen von Objekt X durch Rechner C:
  $checkSum := PubKey_A(checkSum + Sig_C(X) + ID_C)$
  Der neue Prüfsummenwert ergibt sich durch Verkettung des alten Wertes mit der von Rechner C erzeugten Unterschrift für das Objekt X und seiner Identitätskennung ID_C, die wiederum mit dem öffentlichen Schlüssel des Agenten verschlüsselt werden.

Es ist nun aber möglich, nachträglich einzelne Objekte aus dem `AppendOnlyContainer` nach Karnik [Kar98] zu entfernen und die Prüfsumme so anzupassen, dass die Manipulation später nicht entdeckt werden kann:

1. Der Rechner C schafft sich Zugang zu den unverschlüsselten Daten des Agenten, genauer zur aktuellen Prüfsumme des Containers und dem Vektor, in dem die angehängten Objekte gespeichert werden. Daraus bestimmt er den Index des höchsten angehängten Objektes und speichert diese Informationen:

$$\begin{aligned} storedCheckSum &:= checkSum \\ storedVectorSize &:= objs.size() \end{aligned}$$

2. Nun hängt Rechner C gegebenenfalls seine eigenen Daten an den Container an und lässt den Agenten gemäß dessen Wegewahl zur nächsten Plattform migrieren.

3. Nun gibt es zwei Angriffsmuster:

   (a) Der bösartige Rechner C ist ein Einzeltäter:
   Falls der Agent im Laufe seiner Tätigkeit wieder auf dem Rechner C ankommt, kann dieser alle Daten des Agenten unbemerkt entfernen, die seit der Aktion in Punkt 1 an den Container angehängt wurden:

   i. Der Rechner verschafft sich Zugriff auf die Vektoren `objs` (Objekte), `signs` (Unterschriften der Rechner), `signers` (Adressen der unterschriftgebenden Rechner) (zum Beispiel durch direktes Auslesen der Speicherbereiche).

   ii. Nun entfernt er gezielt Datenelemente, deren Indices größer oder gleich dem gespeicherten Wert $storedVectorSize$ sind, die also nachträglich erst angehängt wurden. Dabei kann er mittels der unverschlüsselt gespeicherten Unterschriften und Identitäten in den Vektoren `signs` und `signers` die Objekte den zugehörigen Rechnern (bzw. Parteien) zuordnen[1].

   iii. Ausgehend von der alten gespeicherten Prüfsumme stehen ihm sämtliche zusätzlich benötigten Daten zur Neuberechnung der Prüfsumme zur Verfügung:

   - Die Objekte X_i,
   - die zugehörige Unterschrift $Sig_{R_i}(X_i)$ und
   - die Identität ID_{R_i} des entsprechenden Rechners R_i sowie
   - der öffentliche Schlüssel $PubKey_A$ des Agenten,

   mit Index $i \in I = [storedVectorSize, \ldots, objs.size()] \subset \mathbf{N}$.
   Beschreibe die Indexmenge $\widehat{I} \subset I$ die Menge der nach der Manipulation verbleibenden Objekte X_j mit $j \in \widehat{I}$, dann ergibt sich die Neuberechnung der Prüfsumme $checkSum$ zu:

   A. Erster Schritt für $j_0 = \mathbf{min}(\widehat{I})$:
   $checkSum_{j_0} := PubKey_A(storedCheckSum + Sig_{R_{j_0}}(X_{j_0}) + ID_{R_{j_0}})$

   B. Folgeschritte rekursiv für $j_k \in \widehat{I}$ mit $k = 1, \ldots, \mid \widehat{I} \setminus \{j_0\} \mid$:
   $checkSum_{j_k} := PubKey_A(checkSum_{j_{k-1}} + Sig_{R_{j_k}}(X_{j_k}) + ID_{R_{j_k}})$

   C. Und schließlich den originalen Prüfsummenwert mit dem neuberechneten Wert ersetzen: $checkSum := checkSum_{j_k}$.

   (b) Der bösartige Rechner D konspiriert mit einem Rechner D:
   Rechner C sendet die Informationen $storedCheckSum$ und $storedVectorSize$ an Rechner D, der dann die selben Schritte – wie in Aufzählung 3a für einen Einzeltäter beschrieben – ausführt, um wahlweise unbemerkt Einträge aus dem `AppendOnlyContainer` zu entfernen.

Somit ist der `AppendOnlyContainer` in dieser Ausprägung nicht tauglich für den Einsatz in sicherheitskritischen (E-Commerce-)Anwendungen.

[1] Mit Blick auf das Comparison Shopping Modell kann der Angreifer direkt die Angebote konkurrierender preisgünstigerer Händler feststellen und entfernen.

Literaturverzeichnis

[AK96] Ross Anderson und Markus Kuhn. Tamper Resistance – a Cautionary Note. In *The Second USENIX Workshop on Electronic Commerce Proceedings*, Jgg. 11, Seite 1–11. USENIX Association, Oakland, California, 1996.

[BBE99a] M. Baentsch, P. Buhler und T. Eirich. Java embedded Technology – Zurich (JetZ). `http://www.zurich.ibm.com/technologies/jetz/acs_security_jetz.html`, 1999.

[BBE+99b] M. Baentsch, P. Buhler, T. Eirich, F. Höring und M. Oestreicher. JavaCard – From Hype to Reality. *IEEE Concurrency*, Oct.–Dec. 1999: Seite 36–42, 1999.

[BGS98] S. Berkovits, J. D. Guttman und V. Swarup. Authentication for Mobile Agents. In *Mobile Agents and Security* [Vig98], Seite 114–136.

[BGW97] Thomas Beth, Willy Geiselmann und Peer Wichmann. Public-Key Kryptographie, 1997.

[BM95] C. Boyd und W. Mao. Design and analysis of key exchange protocols via secure channel identification. *Lecture Notes in Computer Science*, 917: Seite 171 ff., 1995.

[Boh98] Jürgen Bohn. Elektronisches Geld. Seminararbeit – `Electronic Commerce` Seminar, Forschungszentrum Informatik, Forschungsbereich Programmstrukturen und Datenorganisation, Prof. Dr. P. Lockemann, Universtiät Karlsruhe (TH), D-76128 Karlsruhe, Juni 1998.

[CCMS99] Antonio Corradi, Marco Cremonini, Rebecca Mantanari und Cesare Stefanelli. Mobile Agents and Security: Protocols for Integrity. 1999.

[CEF+98] Philip Cullum, Vernon Ellis, Glover Ferguson, James Gowers, Steve Johnson und andere. Your Choice. How eCommerce Could Impact Europe's Future. Technical report, Andersen Consulting, 1998.

[Cer99] Certicom. RSA 512-Bit Challenge Factored. `http://www.certicom.com/press/RSA-155.htm`, August 1999.

[Che98] D. Chess. Security Issues in Mobile Code Systems. In *Mobile Agents and Security* [Vig98], Seite 1–14.

[CHK97] D. Chess, C.G. Harrison und A. Kershenbaum. Mobile Agents: Are they a good idea? In Vitek und Tschudin [VT97], Seite 25–47.

[DD98] G. Di Caro und M. Dorigo. Mobile Agents for Adaptive Routing. In *31st Hawaii International Conference on System Science*, Big Island of Hawaii, Januar 1998.

[DDFY94] Alfredo De Santis, Yvo Desmedt, Yair Frankel und Moti Yung. How to Share a Function Securely. In *Proc. 26th Annual ACM Symposium on Theory of Computing (STOC)*, Seite 522–533, 1994.

[Des94] Yvo Desmedt. Threshold Cryptography. *European Transactions on Telecommunications*, 5(4): Seite 449–457, 1994.

[DS98] Premkumar T. Devanbu und Stuart G. Stubblebine. Stack and Queue Integrity on Hostile Platforms. In *IEEE Computer Society Symposium on Research and Privacy*, Seite 198–206. IEEE Press, Oakland, California, Mai 1998.

[ElG85] Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. IEEE Transactions on Information Theory 31/4, 1985.

[Fün99] Stefan Fünfrocken. Protecting Mobile Web-Commerce Agents with Smartcards. Seite 90–102, Oktober 1999. First International Symposium on Agent Systems and Applications (ASA'99) / Third International Symposium on Mobile Agents (MA'99), California, USA.

[GM98] R. Guttman und P. Maes. Agent-mediated Integrative Negotiation for Retail Electronic Commerce. In *Workshop on Agent-mediated Electronic Trading (AMET'98)*, Mai 1998.

[GMM98] R. Guttman, A.G. Moukas und P. Maes. Agent-mediated Electronic Commerce: A Survey. In *Knowledge Engineering Review*, 1998.

[HLPS98] Brant Hashii, Mandoj Lal, Raju Pandey und Steven Samorodin. Securing Systems Against External Programs. *IEEE Internet Computing*, 1089–7801, November 1998.

[HNSS99] Uwe Hansmann, Martin Nicklous, Thomas Schäck und Frank Seliger. *Smart Card Application Development Using Java*. Springer-Verlag Berlin Heidelberg, 1999.

[Hoh98] Fritz Hohl. Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts. *Lecture Notes in Computer Science*, 1419: Seite 92 ff., 1998.

[IAI99] Institute for Applied Information Processing and Communications IAIK. IAIK Java Cryptography Extension (IAIK-JCE). `http://jcewww.iaik.tu-graz.ac.at`, Oktober 1999. IAIK Java Crypto Software Homepage.

[IBMa] IBM Tokyo Research Laboratory. The IBM Aglets Software Development Toolkit (ASDK) Homepage. `http://www.trl.ibm.co.jp/aglets/`.

[IBMb] IBM Zurich Research Laboratory. Homepage. `http://www.zurich.ibm.com`.

[Jav] JavaWorld. Homepage. `www.javaworld.com`.

[JK99] Wayne Jansen und Tom Karygiannis. Technical Report - Mobile Agent Security. Unpublished Draft, 1999.

[KAG98] G. Karjoth, N. Asokan und C. Gülcü. Protecting the Computation Results of Free-Roaming Agents. In *Lecture Notes in Computer Science* [RH98], Seite 195–207.

[Kar98] Neeran M. Karnik. *Security in Mobile Agent Systems.* Dissertation, University of Minnesota, Oktober 1998.

[Kel93] Ludwig Keller. Vom Name-Server zum Trader. Praxis der Informationsverarbeitung und Kommunikation 16/3 (1993), 1993.

[KLO98] Günter Karjoth, Danny B. Lange und Misuru Oshima. A Security Model for Aglets. In *Mobile Agents and Security* [Vig98], Seite 188–205.

[LO98] D.B. Lange und M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley Longman, 1998.

[Man98] Torsten Mandry. Sicherheit von mobilen Agenten auf elektronischen Märkten. Diplomarbeit, Universität GH Essen, Februar 1998.

[Mea97] Catherine Meadows. Detecting Attacks on Mobile Agents. In *Workshop on Foundations for Secure Mobile Code*. Washington, DC 20375, 1997.

[MEJ99] Srilekha Mudumbai, Abdeliah Essiari und William Johnston. Anchor - A Secure Mobile Agent Toolkit. Technical report, Ernest Orlando Lawrence Berkeley Laboratory, 1999.

[Miy98] Shichirou Miyashita. About e-Marketplace. `http://www.trl.ibm.co.jp/aglets/emplace/emplace.html`, 1998.

[Nee93] Roger M. Needham. Cryptography and Secure Channels. In *Distributed Systems*, ACM Press Frontier Series, Seite 531–541. University of Twente The Netherlands, 1993.

[NIS93] National Institute for Standards and Technology NIST. *Data encryption standard (DES)*, Jgg. FIPS PUB 46-2. United States Government Printing Office, Washington, DC, USA, Dezember 1993. `http://csrc.ncsl.nist.gov/fips/`.

[NIS95] National Institute for Standards and Technology NIST. *Secure Hash Standard (SHS)*, Jgg. FIPS PUB 180-1. United States Government Printing Office, Washington, DC, USA, April 1995. `http://csrc.ncsl.nist.gov/fips/`.

[OKO98] Mitsuri Oshima, Günter Karjoth und Kouichi Ono. *Aglets Specification 1.1 Draft*. IBM Research Division, IBM Tokyo, IBM Zürich, Draft 0.65. Auflage, September 1998.

[OMG] Object Management Group OMG. Homepage. `www.omg.org`.

[Ope] OpenCard Consortium. Homepage. `www.opencard.org`.

[Ple99] Stefan Pleisch. State of the Art of Mobile Agent Computing – Security, Fault Tolerance, and Transaction Support. Research Report RZ 3152 (#93198), IBM Zurich Research Laboratory, Juni 1999.

[RB94] Michael K. Reiter und Kenneth P. Birman. How to Securely Replicate Services. *ACM Transactions on Programming Languages and Systems*, 16(3): Seite 986–1009, Mai 1994.

[RGK97] D. Rus, R. Gray und D. Kotz. Transportable Information Agents. In M. Huhns und M. Singh, Hrsg., *Readings in Agents*. Morgan Kaufmann Publishers, 1997.

[RH98] K. Rothermel und F. Hohl, Hrsg. *Second International Workshop on Mobile Agents (MA'98)*, number 1477 in Lecture Notes in Computer Science. Springer-Verlag, 1998.

[RR96] Kay A. Robbins und Steven Robbins. *Practical Unix Programming: A Guide to Concurrency, Communication, and Multithreading.* Prentice Hall, 1996.

[RSA78] Ronald L. Rivest, Adi Shamir und Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2): Seite 120–126, Februar 1978.

[Sch97] F.B. Schneider. Towards Fault-tolerant and Secure Agentry. In *11th Int. Workshop on Distributed Algorithms*, Saarbrücken, Germany, September 1997.

[Sha79] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11): Seite 612–613, November 1979.

[Sha99] Adi Shamir. Factoring large numbers with the TWINKLE device. Eurocrypt'99, Mai 1999. Extended abstract.

[Sil99] Robert D. Silverman. An Analysis of Shamir's Factoring Device, Mai 1999.

[ST98a] Tomas Sander und Christian F. Tschudin. On Software Protection Via Function Hiding. Technical report, International Computer Science Institute, 1947 Center Street, Berkeley, CA 94704, USA, 1998.

[ST98b] Tomas Sander und Christian F. Tschudin. Protecting Mobile Agents Against Malicious Hosts. In *Mobile Agents and Security* [Vig98], Seite 44–60.

[ST98c] Tomas Sander und Christian F. Tschudin. Towards Mobile Cryptography. *IEEE Proceedings of Security & Privacy*, Mai 1998.

[Sun99a] Sun Microsystems. Java Card Applet Developer's Guide, 1999.

[Sun99b] Sun Microsystems. Java Card (TM) Technology. `http://www.javasoft.com/products/javacard/index.html`, 1999.

[Sun99c] Sun Microsystems. Java Platform Documentation. `http://java.sun.com/docs/`, 1999.

[TK98] Anand Tripathi und Neeran Karnik. Protected Resource Access for Mobile Agent-based Distributed Computing. Technical report, Department of Computer Science, Minneapolis, MN 55455, USA, 1998.

[Tow] Towitoko electronics. CHIPDRIVE micro 120 Multifunctional Smartcard Terminal. `http://www.towitoko.de`.

[TZ93] Paula Tallim und J.C. Zeeman. *Electronic data interchange: an overview of EDI standards for libraries.* Number 4 in UDT series on data communication technologies and Standards for Libraries. IFLA International Office for Universal Dataflow and Telecommunications, Ottawa, Canada, 1993.

[Vig97] Giovanni Vigna. Protecting Mobile Agents Through Tracing. In *Third Workshop on Mobile Object Systems*, New York, NY, USA, Juni 1997. Springer-Verlag.

[Vig98] Giovanni Vigna. *Mobile Agents and Security*, Jgg. 1419 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, NY, USA, 1998.

[VT97] J. Vitek und C. Tschudin, Hrsg. *Mobile Object Systems - Towards the Programmable Internet*, number 1222 in Lecture Notes in Computer Science. Springer-Verlag, 1997.

[Whi96] J.E. White. Mobile Agents. In J. Bradshaw, Hrsg., *Software Agents*. AAAI Press and MIT Press, 1996.

[XAC99] XAC Automation Corporation. Stand-alone Smart Card Reader P68. `http://www.xacautomation.com`, 1999.

[Yee97] B.S. Yee. A Sanctuary For Mobile Agents. In *Third Workshop on Mobile Object Systems*, Jgg. Technical Report CS97-537. UC San Diego, April 1997.

Index