Sicherheit in Mobile-Agenten-Systemen

Von der Fakultät Informatik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Fritz Hohl

aus Wangen/Allgäu

Hauptberichter:	Prof. Dr. Kurt Rothermel
Mitberichter:	Prof. Dr. Winfried Lamersdorf
Tag der mündlichen Prüfung:	26. April 2001

Institut für Parallele und Verteilte Höchstleistungsrechner (IPVR)
der Universität Stuttgart

2001

Inhaltsverzeichnis

Abkürzungsverzeichnis

BTX	Bildschirmtext
CED	Computing with Encrypted Data
CPU	Central Processing Unit
DES	Data Encryption Standard
DSS	Digital Signature Scheme
EEF	Evaluation of Encrypted Functions
FTP	File Transfer Protocol
HTML	Hypertext Markup Language
ISO	International Standardization Organisation
LAN	Local Area Network
MD4	Message Digest 4
MD5	Message Digest 5
PKCS1	Public-Key Cryptography Standard #1
PRAC	Partial Result Authentication Code
RAM	Random Access Memory
RAM	Random Access Machine
RMI	Remote Method Invocation
ROM	Read-only Memory
RSA	Rivest-Shamir-Adleman (-Algorithmus)
SHA-1	Secret Hash Algorithm 1
TLBP	Time-limited Blackbox Protection
URL	Uniform Resource Locator
WAN	Wide Area Network
WWW	World-Wide Web

Zusammenfassung

Mobile Agenten sind Programminstanzen, die in der Lage sind, sich selbstständig zwischen verschiedenen, eventuell fremden, Ausführungsumgebungen zu bewegen und, unter Ausnutzung lokaler Ressourcen, Aufgaben zu erfüllen. Die vorliegende Arbeit beschäftigt sich mit der Frage der Sicherheit zwischen mobilen Agenten und deren Ausführungsumgebungen. Die Aufgabenstellung umfasst zwei Teilbereiche. Im Bereich des Schutzes der Ausführungsumgebung vor Angriffen durch mobile Agenten werden die möglichen Angriffe und die existierenden Lösungsansätze vorgestellt. Der zweite Teilbereich umfasst den Schutz mobiler Agenten vor Angriffen durch ihre Ausführungsumgebung. Hier werden ebenfalls zunächst die möglichen Angriffe geschildert. Weiter wird ein Modell der Wirtsmaschine und des Angreifers erarbeitet, das es erlaubt, die möglichen Angriffe zu illustrieren. Unter Benutzung einer neuen Kategorisierung werden dann verwandte Arbeiten erläutert. Im Bereich des Schutzes vor einzelnen Angriffen werden existierende Verfahren untersucht, die bestimmte Angriffe gegen einen mobilen Agenten durch Vergleich mit Referenzzuständen entdecken können. Durch eine Kombination dieser Verfahren wird dann ein neues Verfahren entwickelt sowie diskutiert. Um mobile Agenten vor allen Angriffen zu schützen, wird danach das Gesamtproblem des Schutzes mobiler Agenten auf ein kleineres Problem reduziert indem die sog. Blackbox-Eigenschaft mobiler Agenten angenommen wird. Darauf folgend werden zwei Verfahrensklassen beschrieben, die diese Eigenschaft gewährleisten sollen. Für die Klasse der nichtinteraktiven Auswertung von verschlüsselten Funktionen werden drei existierende Mechanismen beschrieben. Um einen Nachteil der existierenden Verfahrensklasse zu überwinden, wird dann eine neue Verfahrensklasse vorgestellt, die aber eine Zeitbeschränkung des erzielten Schutzes aufweist. Abschließend wird ein neues Protokoll beschrieben, das den sog. Blackbox-Test, einen Angriff, der auch bei Annahme der Blackbox-Eigenschaft möglich ist, verhindert.

Abstract

Mobile agents are program instances that are able to migrate between different, even unknown execution environments, and to fulfil tasks using local resources. This work examines the area of security between mobile agents and their execution environments. This area comprises two sub-areas. In the area of protection of the execution environment from attacks by malicious agents the possible attacks and the existing protection approaches are presented. The second sub-area comprises the protection of mobile agents from attacks by their execution environments. First, again the possible attacks are presented. Furthermore, a model of the host machine

and the attacker is given that can illustrate these attacks. Using a new categorization, related approaches were examined. For the area of preventing single attacks, existing mechanisms are examined that are able to detect some attacks against a mobile agent by comparing the execution with a reference state. Then, a new approach is presented that results from a combination of these mechanisms. To protect mobile agents from all attacks, then the complete problem of protecting mobile agents is reduced to a smaller problem by assuming the so-called blackbox property of mobile agents. Then, two classes of mechanisms are described that achieve this property. For the class of non-interactive evaluation of encrypted functions, three existing mechanisms are described. To circumvent a disadvantage of the existing class of mechanisms, a new class is presented. This class shows a time-limitedness of the achieved protection. Finally, a new protocol is described that prevents the so-called blackbox test, an attack that is possible even for agents showing the blackbox property.

1 Einleitung

Mobile Agenten sind, in erster Näherung, Programminstanzen, die in der Lage sind, sich selbständig von einem Rechner innerhalb eines Netzwerks zu einem anderen Rechner zu bewegen (zu *migrieren*). Als Programminstanzen bestehen mobile Agenten aus Code und aus konstanten sowie variablen Daten, wobei ein Agent die variablen Daten zur Laufzeit erweitern oder verringern kann. Mobile Agenten können mit anderen mobilen Agenten, dem Agentensystem und anderen Partnern im Netzwerk kommunizieren. Weil mobile Agenten kommunizieren können, und weil sie sich zur Ausführung ihres Programmes nicht unbedingt bewegen müssen, kann man diese Technik auch für eine Client-Server-artige Interaktion einsetzen, bzw. Kombinationen aus solchen Interaktionen und lokalen Interaktionen nach einer Migration benutzen. Daher können mobile Agenten als eine Erweiterung des Client-Server-Modells verstanden werden.

Aus technischer Sicht werden für mobile Agenten eine ganze Reihe von möglichen Vorteilen geltend gemacht, wobei diese zum Teil qualitativer und zum Teil quantitativer Natur sind. Qualitative Vorteile sind u.a. ein einfacheres Fehlermodell, asynchrones Arbeiten, dynamisches programmgesteuertes Verlagern von Funktionalität, und Ersetzung passiver durch aktive Elemente. Als quantitative Vorteile ergeben sich u.a.: Verringerung der Kommunikationskosten, Erhöhung des Verarbeitungsdurchsatzes und Verminderung von Kommunikationsverzögerungen.

Anwendungsgebiete für mobile Agenten werden zurzeit vor allem in den Bereichen Informationssuche, mobile Endgeräte, aktive elektronischer Post, Verteilung von Berechnungen, aktive Netzwerke, Netzwerkmanagement und elektronischer Handel gesehen, aber der Anwendungsbereich ist im Prinzip unbegrenzt.

In einer zunehmend stark vernetzten Umgebung, in der viele bisher nicht-automatisierte Aspekte des Lebens wie z.B. der Handel und die Unterhaltung von der Straße weg hinein in die Rechner verlagert werden, spielt der Aspekt der Sicherheit eine immer wichtigere Rolle. Dies gilt in besonderem Maße für Mobile-Agenten-Systeme, weil diese zum einen die Grundlage für den elektronischen Handel bilden könnten. Zum anderen ist die Automatisierung bisher durch Menschen durchgeführter Handlungen (z.B. das Suchen einer Information oder der Kauf einer Eintrittskarte) mittels mobiler Agenten gerade die Hauptmotivation für den Einsatz mobiler Agenten. Daher ist der Aspekt der Sicherheit für mobile Agenten auch wesentlich für die Akzeptanz dieser Technologie.

Unglücklicherweise stürzt aber gerade der Aspekt der Mobilität eine der (oftmals impliziten) Grundannahmen im Bereich der Sicherheit um, nämlich die Identität von Programmbenutzer und Programmausführer und damit auch die implizite

Annahme der Existenz einer vertrauenswürdigen Ausführungsumgebung für Programme.

Herkömmliche, d.h. nicht-mobile Programme werden von Benutzern deshalb ausgeführt, weil sich diese Vorteile aus der Ausführung versprechen, d.h. die Programme sollen im Auftrag dieser Benutzer arbeiten. Mit dem Aufkommen von Viren, Würmern und trojanischen Pferden, also Programmen die gewollt zum Schaden des Benutzers arbeiten, entstand das Bewusstsein vom Risiko, fremde Programme auf dem eigenen Rechner auszuführen.

Durch den breiten Einsatz von mobilem Code (wie z.B. Java Applets) in den letzten fünf Jahren erhöhte sich diese Gefahr noch, weil der Zyklus "Software finden, auf eigenen Rechner bewegen, installieren, starten" automatisiert wurde. Er kann sogar unbeabsichtigt angestoßen werden, z.B. durch das Laden einer HTML-Seite in einem WWW-Browser.

Mobile Agenten weisen diese Eigenschaft ebenfalls auf, da auch sie zu den Mobile-Code-Systemen gehören. Im Fall der mobilen Agenten werden die mobilen Programme jedoch nicht einmal mehr unbedingt für einen Benutzer des Rechners ausgeführt, auf dem das Programm läuft, sondern unter Umständen sogar für einen fremden Benutzer, der nicht mehr der organisatorischen Kontrolle des benutzten Rechnersystems unterliegt.

Die Trennung von Programmbenutzer und Programmausführer birgt jedoch nicht nur Gefahren für diejenige Partei, die einen mobilen Agenten ausführt. Auch der Programm*benutzer*, genauer: der Auftraggeber des Agenten kann durch Angriffe geschädigt werden. Man kann sich beispielsweise in einer kommerziellen Anwendung vorstellen, dass ein solcher Agent elektronisches Geld oder kryptografische Schlüssel transportiert, und dass es im Interesse des Auftraggebers liegt, dass z.B. der Programmausführer diese Daten nicht lesen kann. Dieses Problem, der Schutz mobiler Agenten vor Angriffen durch den Programmausführer, entsteht nicht nur durch die Mobilität von Agenten; er ist gleichermaßen das Resultat der Tatsache, dass mobile Agenten einen individuellen Zustand besitzen. Dies ist bei mobilem Code nicht immer der Fall. Daher stellt dieser Aspekt eine Besonderheit im Vergleich zur allgemeinen Sicherheitsproblematik mobilen Codes dar.

Da Mobile-Agenten-Systeme auch Mobile-Code-Systeme sind und als solche ebenfalls Eigenschaften herkömmlicher verteilter Systeme aufweisen, werden glücklicherweise nicht an allen Stellen neue Sicherheitsmechanismen benötigt. Gerade was den Schutz des ausführenden Rechners (bzw. dessen Betreiber) angeht, können viele traditionelle Sicherheitselemente wie Verschlüsselung, Authentifizierung und Autorisierung modifiziert oder sogar unverändert übernommen werden. Ein Ziel der Forschung im Bereich Sicherheit bei mobilen Agenten ist außerdem häufig, solche

Sicherheitsmechanismen zu finden, die Unterschiede zu herkömmlichen verteilten Systemen aufheben, um solchermaßen mit bisherigen Systemen gleichgestellt, deren Sicherheitsmechanismen ebenfalls benutzen zu können.

Im Gegensatz zu den meisten anderen Fragestellungen im Bereich der Sicherheit reicht es für Mobile-Agenten-Systeme aber nicht aus, Schutzmechanismen nur zu finden. Da agentenbasierte Anwendungen in der Regel ohne Verlust der Funktionalität (nicht aber anderer Eigenschaften wie Fehlerverhalten und Performanz) auch Client-Server-basiert konzipiert werden können, dürfen die Kosten der gefundenen Mechanismen die Vorteile der Benutzung mobiler Agenten nicht aufwiegen.

Die Forschung im Bereich der Sicherheit in Mobile-Agenten-Systemen (siehe [SMA] für eine Bibliographie dieses Gebietes) konzentrierte sich am Anfang auf den Aspekt des Schutzes der ausführenden Partei vor Angriffen durch den ausgeführten mobilen Agenten. Die Resultate dieser Forschung zeigen, dass es bereits heute möglich ist, Verfahren zur Lösung der wichtigsten Fragestellungen zu realisieren. Der Aspekt des Schutzes mobiler Agenten vor Angriffen durch die ausführende Partei galt zu Anfang als unlösbares Problem, für das allenfalls Lösungen in Teilbereichen existierten. Auch heute gibt es keine universelle Lösung dieses Problems außerhalb des Einsatzes spezialisierter Hardware oder der Einbindung von Mobile-Agenten-Systemen in spezielle Organisationsformen.

Die vorliegende Arbeit beschäftigt sich daher im Besonderen mit der Frage unter welchen Bedingungen ein mobiler Agent vor Angriffen durch die ausführende Partei geschützt werden kann. Zur Beantwortung dieser Frage erfolgt daher zuerst eine Schilderung der möglichen Angriffe. Weiter wird ein Modell der Wirtsmaschine und des Angreifers erarbeitet, das es erlaubt, die möglichen Angriffe zu illustrieren. Unter Benutzung einer neuen Kategorisierung werden dann verwandte Arbeiten erläutert. Im Bereich des Schutzes vor einzelnen Angriffen werden existierende Verfahren untersucht, die bestimmte Angriffe gegen einen mobilen Agenten durch Vergleich mit einer Referenzausführung entdecken können. Durch eine Kombination dieser Verfahren wird dann ein neues Verfahren entwickelt sowie diskutiert. Um mobile Agenten vor allen Angriffen zu schützen, wird danach das Gesamtproblem des Schutzes mobiler Agenten auf ein kleineres Problem reduziert indem die sog. Blackbox-Eigenschaft mobiler Agenten angenommen wird. Darauf folgend werden zwei Verfahrensklassen beschrieben, die diese Eigenschaft gewährleisten sollen. Für die Klasse der nicht-interaktiven Auswertung von verschlüsselten Funktionen werden drei existierende Mechanismen beschrieben. Um einen Nachteil der existierenden Verfahrensklasse zu überwinden, wird dann eine neue Verfahrensklasse vorgestellt, die aber eine Zeitbeschränkung des erzielten Schutzes aufweist. Abschliessend wird ein neues Protokoll beschrieben, das den sog. Blackbox-Test,

einen Angriff, der auch bei Annahme der Blackbox-Eigenschaft möglich ist, verhindert.

1.1 Aufbau der Arbeit

Nachdem in Kapitel 2 mobile Agenten mobile Agenten eingeführt, das dahinterstehende Programmierparadigma erläutert und die Umsetzung dieses Paradigmas beschrieben wird, befasst sich Kapitel 3 mit der Strukturierung des Gesamtbereichs der Sicherheit in Mobile-Agenten-Systemen. Nachdem kurz einzelne Kategorien des Gesamtproblems gestreift werden, befasst sich Kapitel 4 näher mit dem Schutz der Ausführungsumgebungen (oder *Wirte*) vor böswilligen Agenten. Kapitel 5 und die nachfolgenden Kapitel sind dann dem Schutz von mobilen Agenten gegenüber böswilligen Wirten gewidmet. Nachdem zunächst auf die möglichen Angriffe gegen mobile Agenten eingegangen wird sowie ein Modell der Wirtsmaschine und des Angreifers erarbeitet wird, beschäftigt sich Kapitel 6 mit den verwandten Arbeiten auf diesem Gebiet. Dazu wird in diesem Kapitel eine neuartige Kategorisierung dieser Ansätze entwickelt und dann, unter Benutzung dieses Hilfsmittels, die Ansätze geschildert. Kapitel 7 ist einer speziellen Familie von Ansätzen, nämlich solchen, die Referenzzustände benutzen, gewidmet. Nachdem zunächst existierende Ansätze dieser Familie erläutert und analysiert werden, wird ein neuer Ansatz dieser Familie dargelegt, der bestehende Ansätze kombiniert. War dieses Kapitel noch Ansätzen vorbehalten, der eine eng begrenzte Anzahl an Angriffen verhindert, beschäftigt sich Kapitel 8 mit einer Klasse von Ansätzen, die alle Angriffe zu verhindern suchen. Dazu wird die Idee der Blackbox-Eigenschaft eingeführt, die per definitionem einige Angriffe verhindert. Es wird gezeigt, dass es möglich ist, auf dieser Eigenschaft aufbauend, alle bis auf einen Angriff zu verhindern. Danach werden Ansätze untersucht, die die Blackbox-Eigenschaft gewährleisten sollen. Zuerst werden drei Ansätze erläutert, die unter dem Begriff „Nicht-interaktive Auswertung von verschlüsselten Funktionen“ zusammengefasst sind. Danach wird der neue Ansatz eines zeitbeschränkten Blackbox-Schutzes vorgestellt. In Kapitel 9 wird ein neues Protokoll vorgestellt, das sog. Blackbox-Tests verhindert, Angriffe, die von der Blackbox-Eigenschaft nicht verhindert werden. Kapitel 10 fasst die Arbeit zusammen und schildert die Anwendbarkeit der in dieser Arbeit vorgestellten Ergebnisse auf andere Gebiete der Informatik.

2 Mobile Agenten

Mobile-Agenten-Systeme sind spezielle Mobile-Code-Systeme, die ein besonderes Programmierparadigma anbieten. Daher wird in diesem Kapitel der Begriff des Mobile-Agenten-Systems aus einer Erläuterung des Begriffs Mobile-Code-System erklärt. Das Programmierparadigma von Mobile-Agenten-Systemen, das in Kapitel 2.2 beschrieben wird, kann auf verschiedene Weise technisch umgesetzt werden. Da für den Bereich der Sicherheit eine solche Umsetzung als Grundlage der Diskussion dienen muss, wird in Kapitel 2.2.2 eine einfache Architektur für ein Mobile-Agenten-System beschrieben. Schließlich wird in Kapitel 2.2.3 auf die Vorteile und die Probleme von Mobile-Agenten-Systemen eingegangen. Diese Vorteile werden für die spätere Diskussion von Sicherheitsmechanismen deshalb wichtig, weil sich diese auch unter Verwendung von Sicherheitsmechanismen erhalten sollen. Ein Mechanismus, der eine vorteilhafte Eigenschaft von mobilen Agenten unterbindet, beschneidet daher die möglichen Anwendungsbereiche dieser Technik.

2.1 Mobiler Code

Der Begriff “Mobiler Code” bezeichnet Systeme, die in der Lage sind, Code automatisch von einem Rechner auf einen anderen Rechner zu transferieren, zu installieren und auszuführen. Mobile-Code-Systeme umfassen eine breite Palette an

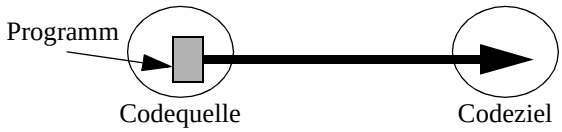


Bild 1: Parteien in Mobile-Code-Systemen

existierenden und möglichen Systemen, die, außer dem Aspekt des transportierten Programmes, wenig Gemeinsamkeiten haben. Man kann Mobile-Code-Systeme nach dem Aspekt einteilen, welche der Parteien “Codequelle” und “Codeziel” die Bewegung des Codes aus technischer Sicht initiiert.

In “Pull”-Systemen holt das Codeziel den Code von der Quelle und führt ihn aus. Das bekannteste Beispiel für ein solches System sind Java Applets, bei denen ein Web-Browser Java-Klassen von einem Web-Server anfordert und dann startet, wobei die Aufrufparameter aus HTML-Seiten genommen werden, die ebenfalls von Web-Servern geladen werden.

In “Push”-Systemen lädt die Codequelle das Programm auf einen anderen Rechner und lässt dieses dort starten. Beispiele für solche Systeme sind Postscript, bei dem

Programme z.B. auf Drucker geschickt und dort ausgeführt werden, sowie der “Remote Evaluation”-Ansatz [SG90], ein Kommunikationsmechanismus, bei dem neben den benötigten Parametern auch der benötigte Code auf einen anderen Rechner bewegt wird.

2.2 Mobile Agenten

Mobile-Agenten-Systeme sind spezielle Mobile-Code-Systeme, die ein besonderes Programmierparadigma anbieten (im Gegensatz zu Mobile-Code-Systemen allgemein, die verschiedene Paradigmen benutzen). Dieses Paradigma wird im folgenden Abschnitt beschrieben. Kapitel 2.2.2 wird dann die Umsetzung des Programmierparadigmas beschreiben, Kapitel 2.2.3 die Vorteile, und Kapitel 2.2.4 die Herausforderungen in Mobile-Agenten-Systemen.

2.2.1 Das Programmierparadigma

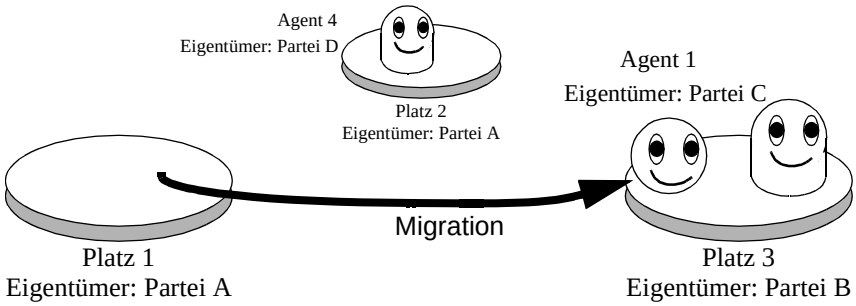


Bild 2: Modell im Programmierparadigma

Aus der Sicht des mobilen Agenten ist die Welt eingeteilt in einzelne *Plätze*, d.h. abgegrenzte Gebiete, auf denen ein Agent existieren und zwischen denen er migrieren kann. Eine Migration findet immer genau zwischen zwei Orten statt. Ein Agent, der auf einem Platz ausgeführt wird, heißt aktiv, ein Agent, der sich in der Migration befindet, oder aus sonstigen Gründen nicht ausgeführt wird, passiv. Agenten können mit anderen Agenten lokal oder entfernt, sowie Partnern außerhalb des Mobile-Agenten-Systems kommunizieren. Ein mobiler Agent, der bereits von einem Ort gestartet, aber noch nicht auf dem Zielort angekommen ist, befindet sich im “Transit”, und kann dann weder kommunizieren noch aktiv sein. Agenten und Plätze haben lebenslange, systemweit eindeutige Namen.

Agenten arbeiten für eine auftraggebende Partei, Plätze werden von Betreiberparteien unterhalten, wobei es Überschneidungen zwischen diesen Parteien geben kann, aber nicht muss. Agenten können die Betreiberparteien von Plätzen in Erfahrung bringen, ebenso kann die Betreiberpartei eines Platzes die auftraggebende Par-

tei eines Agenten in Erfahrung bringen, der auf diesen Platz migrieren will. Bisweilen wird in der Literatur auch noch der Urheber des Agenten, also der Programmierer unterschieden. Da diese Unterscheidung jedoch nur für solche Fragestellungen Sinn macht, bei der die Frage nach juristischer Haftung des Programmierers gegenüber dem Auftraggeber eine Rolle spielt oder bei der etwa mit der Partei des Programmierers Vertrauen assoziiert wird (“lasse die Migration eines Agenten zu, wenn er von der Firma WorldAgent programmiert wurde“) und diese Fragestellungen in dieser Arbeit nicht behandelt werden, wird im Folgenden auf diese Unterscheidung verzichtet.

2.2.2 Realisierung mobiler Agenten

Das im letzten Abschnitt beschriebene Programmierparadigma kann auf verschiedene Weise technisch realisiert werden, wobei die “Push”-Variante des Codetransfers benutzt wird. Diese Umsetzung kann andere, einfachere Mobile-Code-Systeme benutzen oder modifizieren (z.B. “Remote Evaluation”), oft wird der Codetransfer jedoch durch das Mobile-Agenten-System selbst realisiert, wobei meistens Sprachen wie Java und Tcl eingesetzt werden, die ein dynamisches Einfügen von neuem Code schon auf der Ebene der Laufzeitumgebung erlauben. Um die Grundlage für eine Diskussion der Sicherheit in Mobile-Agenten-Systemen zu haben, soll nun eine mögliche, einfache Architektur für ein Mobile-Agenten-System beschrieben werden, die aber bereits so allgemein ist, dass alle möglichen Problematiken zugelassen werden.

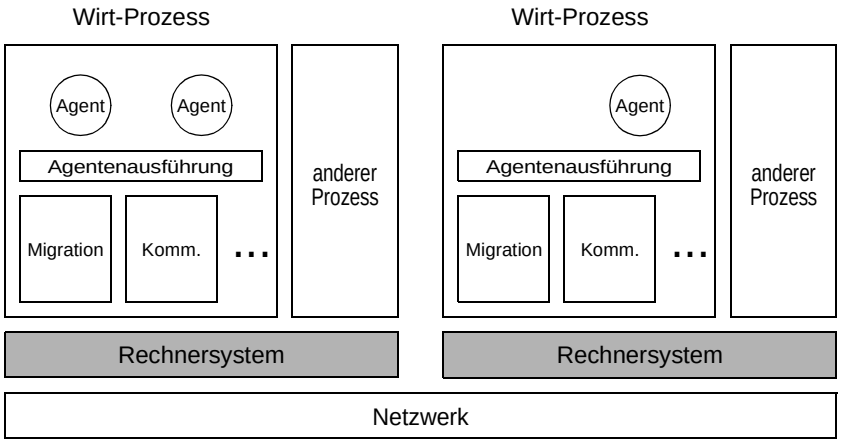


Bild 3: Allgemeine Architektur eines Mobile-Agenten-Systems

Die Architektur ist zunächst die eines beliebigen verteilten Systems, in dem unabhängige Rechnersysteme über ein Netzwerk verknüpft sind. Auf diesen Rechnersystemen laufen Prozesse, die lokal über das Betriebssystem und global über das Netzwerk kommunizieren. Einige dieser Prozesse auf einem Rechnersystem erlauben die Ausführung mobiler Agenten. Diese sogenannten “Wirts”-Prozesse, oder kurz Wirte (engl. host), sind dafür zuständig, mobile Agenten auszuführen, migrationswillige Agenten an andere Wirte zu verschicken und migrierende Agenten entgegenzunehmen. Weiterhin erlauben sie es, mobile Agenten miteinander und mit anderen Prozessen außerhalb des Mobile-Agenten-Systems kommunizieren zu lassen. In der Regel bieten Wirte darüber hinaus weitere Funktionalitäten an, die für diese Arbeit jedoch nicht von Interesse sind. Zuletzt erlaubt es ein Wirt, dass bestimmte Agenten auf Ressourcen des Rechnersystems, wie z.B. ein Dateisystem, zugreifen können. Auch wenn es vom Programmierparadigma nicht gefordert wird, wollen wir außerdem annehmen, dass ein Agent in der Lage ist zu bestimmen, auf welchem Rechnersystem sich welcher Wirt befindet.

Die Frage nach den beteiligten Parteien in der vorgestellten Architektur erlaubt es nicht nur, die Parteien des Programmierparadigmas zu identifizieren, also den Auftraggeber des Agenten und den Betreiber des Wirts, sondern auch den Betreiber des zugrundeliegenden Rechnersystems und andere Benutzer des Rechnersystems im Sinne des benutzten Betriebssystems. Im Bereich der Literatur der Sicherheit in Mobile-Agenten-Systemen wird auf die Unterscheidung der Betreiber und Benutzer eines Rechnersystems allerdings verzichtet, wenn nicht gerade Aspekte der Abrechnung zwischen Diensteanbietern und -inanspruchnehmern behandelt werden. Dies macht Sinn, weil Prozesse in bisherigen Betriebssystemen kaum vor dem Betreiber eines Rechnersystems geschützt werden können, und umgekehrt, der Betreiber eines Wirts als Benutzer eines Rechnersystems ein Interesse daran hat, dass dem Rechnersystem nicht in seinem Namen geschadet wird. Als letzte „Partei“ lässt sich ein möglicher Angreifer identifizieren, der das Netzwerk abhören, auf das Netzwerk schreiben, bzw. im Netzwerk transportierte Daten modifizieren kann. Dieser Angreifer ist natürlich keine erwünschte Partei, muss aber als separater Teilnehmer modelliert werden.

Die vorgestellte Architektur spezifiziert die Relation zwischen Agent und Wirt zunächst nur sehr grob. In Kapitel 5.6 wird daher ein genaueres Modell der Ausführung von Agenten durch den Wirt vorgestellt, das insbesondere die Einflussmöglichkeiten des Wirts auf die Ausführung von Agenten deutlich macht.

2.2.3 Vorteile mobiler Agenten

Der Einsatz mobiler Agenten verspricht eine Reihe von qualitativen und quantitativen Vorteilen gegenüber Client-Server-Systemen. Qualitative Vorteile sind:

Höhere Fehlertoleranz

Wenn auf die Möglichkeit entfernter Kommunikation verzichtet wird, und Interaktion ausschließlich durch Migration und anschließender lokaler Kommunikation realisiert wird, kann man eine höhere Fehlertoleranz erreichen. Dies ist zum einen in der Verringerung der Abhängigkeit von der Verfügbarkeit des Netzwerks begründet. Für eine Interaktion von *n* Schritten muss das Netzwerk im Client-Server-Fall für jeden der *n* Schritte verfügbar sein. Migriert der Agent und arbeitet lokal, dann muss das Netzwerk nur für die Migration (und eine eventuelle Rückmigration) verfügbar sein. Zum anderen muss der Ursprungsrechner nicht für die Dauer der Interaktion verfügbar sein (wie das bei Client-Server-Verarbeitung der Fall ist), sondern ebenfalls nur während der Hin- und Rückmigration.

Asynchrones Arbeiten

Im Gegensatz zu den Client-Server-basierten Anwendungen, bei denen der Client eine Verbindung zum Server während der gesamten Interaktion unterhält (z.B. bei FTP), benötigt ein mobiler Agent keine dauerhafte Kommunikationsverbindung. Der Rechner, der einen mobilen Agenten erzeugt, muss nur für die Dauer der Migration und zum Entgegennehmen der Ergebnisse (z.B. durch eine Rückmigration) mit einem anderen Rechner verbunden bleiben. Diese Eigenschaft erhöht zum einen die Fehlertoleranz, wenn die Fehlerhäufigkeit von der Dauer der Verbindung abhängt, zum anderen hat diese Eigenschaft in einer Umgebung Vorteile, bei der der Benutzerrechner mobil ist, und mit dem restlichen Netzwerk nur über eine teure, schmalbandige Leitung verbunden ist, die mehr Übertragungsfehlern unterliegt.

Dynamisches programmgesteuertes Verlagern von Funktionalität

Mobile Agenten als Mobile-Code-Einheiten erlauben es, Funktionalität, sprich Code, dynamisch an Stellen (also auf Rechner) zu bringen, auf denen diese Funktionalität vorher nicht vorhanden war. Diese Art der Installation funktioniert im Gegensatz zur heutigen Praxis programmgesteuert, d.h. ohne Zutun eines menschlichen Benutzers, und diese Funktionalität steht dann über das Agentensystem allgemein zur Verfügung.

Ersetzen passiver durch aktive Elemente

Dadurch, dass man bisher passive Elemente wie z.B. Text-Emails durch mobile Agenten ersetzt, ist mehr Funktionalität möglich. So kann ein mobiler Agent, der als Email fungiert, z.B. die Nachricht auch anzeigen oder konvertieren. Diese Eigenschaft wird vor allem im Anwendungsgebiet “Aktive Netzwerke” ausgenutzt, wo die Einheiten der Datenübertragung, also Pakete, auch Code tragen können, der auf den Routern ausgeführt wird.

Die meisten der genannten Vorteile könnten dabei auch mit einer Mobile-Code-Middleware erreicht werden, die keine speziellen Mobile-Agenten-Konstrukte anbietet. Allerdings gibt es zur Zeit keine Middleware, die unabhängig von einer bestimmten Anwendung die Benutzung von mobilem Code anbietet, wie es bei Mobile-Agenten-Systemen der Fall ist. Das scheint mit der Verbreitung von Java zwar erstaunlich, aber bei näherer Untersuchung erweist sich die “weite Verbreitung” nur einerseits als weite Verbreitung einer Pull-Code-Plattform bei Web-Clients (sprich Applets, die zudem weitreichenden Sicherheitsrestriktionen unterliegen), und andererseits als die Verbreitung von Java-Interpretern als bereitliegendem Werkzeug zur Abarbeitung von Java-Programmen. Letztere erlauben es natürlich nicht einfach, dass fremder Java-Code auf ein Rechnersystem bewegt wird, zumal die Codetransferfunktionen nicht Teil von Java selbst sind, sondern von Anwendungen, z.B. Browsern, implementiert werden.

Neben den qualitativen Vorteilen gibt es auch einige mögliche quantitative Vorteile:

Verringerung der Kommunikationskosten

Wenn n Kommunikationsinteraktionen über ein Netzwerk mit jeweils d Dateneinheiten durch eine Hinmigration eines Agenten der Transportgröße *t* und eine Rückmigration, die zusätzlich noch m Resultatsdaten bewegt, ersetzt wird, verringern sich die Kommunikationskosten, wenn $n \cdot d > 2 \cdot t + m$ ist. Das ist z.B. dann der Fall, wenn viele Interaktionen stattfinden, bei denen der Agent aus den Rohdaten d weniger Resultatsdaten m generiert.

Verringerung der Latenzzeit

In durch die Lichtgeschwindigkeit begrenzten Netzwerken und einer großen Distanz zwischen zwei Rechnern gibt es eine nicht unterschreitbare Kommunikationsverzögerung. Wenn bei der Kommunikation zwischen zwei Rechnern die Interaktionen seriell nacheinander erfolgen müssen, können diese *Latenzzeiten* den Flaschenhals der Verarbeitung bilden. Wenn man einen der Kommunikationspartner durch Migration lokal zum anderen Partner bewegen kann, verringern sich die Latenzzeiten nicht nur graduell, sondern um mindestens eine Größenordnung.

Diese quantitativen Vorteile gelten dabei für mobile Agenten keineswegs immer, sie hängen stark von der Charakteristik der Anwendung, der Rechnersysteme und des Netzwerks, sowie den aktuellen Gegebenheiten ab (siehe [SS97] für eine Diskussion dieses Aspektes).

2.2.4 Herausforderungen des Konzepts mobiler Agenten

Die Benutzung mobiler Agenten erzeugt bisweilen auch Probleme, die zum jetzigen Zeitpunkt noch nicht alle gelöst sind:

Mobile Agenten können zu schnell sein

Will man Agenten mithilfe von Nachrichten steuern oder beispielsweise den aktuellen Aufenthaltsort eines mobilen Agenten mittels Nachrichten in Erfahrung bringen, stösst man auf das Problem, dass Agenten sich bisweilen mit derselben Geschwindigkeit bewegen wie die Nachrichten selbst. Ist dies der Fall (etwa, weil ein Agent sehr kurz ist), dann ist es schwer, in pathologischen Fällen sogar unmöglich, den Agenten mit der Nachricht einholen zu können. Dieses Phänomen wird durch den Umstand begünstigt, dass es keine notwendige Lokalität der Migrationen gibt, d.h. dass die Folgemigration eines Agenten auf einem „benachbarten“ Rechner erfolgt. Damit ist ohne weiteres Wissen kaum vorherzusagen, wohin ein Agent als nächstes migrieren wird, bzw. wo eine Nachricht einen Agenten erwarten könnte.

Keine direkte Kontrolle

Dadurch, dass ein mobiler Agent auf einem anderen Rechner ausgeführt wird, ist es ohne weitere Maßnahmen schwierig, mobile Agenten zu kontrollieren. Falls beispielsweise ein Agent terminiert werden soll, kann dem Agenten eine Terminierungsaufforderung geschickt werden, aber nur der Agent selbst und der ausführende Wirt sind in der Lage, dieser Aufforderung nachzukommen, jedoch nicht der Sender dieser Aufforderung. Eine weitere Problematik, die aus dem Problem der fehlenden direkten Kontrolle, folgt, ist die Frage der zuverlässigen Ausführung von Agenten. Ohne weitere Maßnahmen ist nicht garantiert, dass ein Agent überhaupt, komplett oder genau einmal ausgeführt wird.

Sicherheit

Wie sich in den folgenden Kapiteln zeigen wird, stellt der Schutz des Wirts vor böswilligen mobilen Agenten kein grundsätzliches Problem dar. Lösungen für das umgekehrte Problem jedoch, den Schutz mobiler Agenten vor böswilligen Wirten, sind bisher noch nicht bis zur praktischen Einsetzbarkeit entwickelt worden. Diese sind aber die unabdingbare Voraussetzung für einen Einsatz mobiler Agenten in offenen Systemen (in denen jeder einen Wirt betreiben kann), falls man den Eigentümern von Agenten durch Angriffe auf diese Agenten schaden kann. Das ist vor allem in Szenarien des elektronischen Handels der Fall.

3 Sicherheit in Mobile-Agenten-Systemen - Eine Strukturierung

In diesem Kapitel wird das Gebiet der Sicherheit in Mobile-Agenten-Systemen in Teilbereiche strukturiert. Es wird unterschieden zwischen der Sicherheit des verwendeten Rechnersystems, der Sicherheit der verwendeten Technologien, sowie verschiedenen Teilbereichen der Sicherheit zwischen Agenten und Wirten.

3.1 Sicherheit des verwendeten Rechnersystems

Als Betriebssystemprozess ist ein Wirt auch ein Element eines Rechnersystems und unterliegt damit auch allen Angriffen, denen auch andere Anwendungen unterliegen können. Dazu zählen u.a. Einbruchsversuche in das Rechnersystem ohne Verwendung des Mobile-Agenten-Systems und „Denial-of-Service“-Angriffe, die sich allgemein gegen Dienste auf einem Rechnersystem richten. Da sich diese Angriffe nicht aus der Existenz eines Mobile-Agenten-Systems ergeben, werden sie in dieser Arbeit nicht behandelt. Trotzdem gilt natürlich auch hier, dass sich die Gesamtsicherheit eines Systems aus der Sicherheit der Einzelkomponenten ergibt. Wenn also ein Rechnersystem Sicherheitslücken aufweist, können damit auch einige Schutzmechanismen des Agentensystems, die in einigen Bereichen auf der Annahme beruhen, dass das zugrundeliegende Rechnersystem sicher ist, außer Kraft gesetzt werden. Interessanterweise wird durch die Annahme der Existenz von möglicherweise böswilligen Wirten, die mobile Agenten angreifen, in Kapitel 5 für die Sicherheit des Agenten die Frage unwichtig, ob ein Angreifer einen Wirt von außen unter seine Kontrolle gebracht hat, oder ob der Betreiber des Wirts der Angreifer ist.

3.2 Sicherheit der verwendeten Technologien

Mobile-Agenten-Systeme verwenden in der Regel Programmiersprachen und andere Technologien, um die benötigte Funktionalität bereitzustellen. Sind diese Technologien unsicher, dann wird auch das Agentensystem als Ganzes Sicherheitsprobleme in diesem Bereich haben. Ein gutes Beispiel für diesen Aspekt ist die Programmiersprache Java, die weite Verwendung im Bereich der mobilen Agenten findet. Immer wieder tauchen Meldungen über Sicherheitsprobleme in einzelnen Implementierungen von Java auf. Diese Probleme können es den Beteiligten in einem Mobile-Agenten-System erlauben, andere Parteien anzugreifen. Da diese Problemstellung ebenfalls nicht von der Verwendung mobiler Agenten, sondern von der speziellen Implementierungsversion abhängt, wird diese Fragestellung in dieser Arbeit ebenfalls nicht behandelt. Wir nehmen daher im Folgenden immer an, dass die verwendeten Technologien sicher sind.

3.3 Sicherheit von Agenten und Wirten

Der Bereich der Sicherheit von Agenten und Wirten lässt sich unterteilen in das Gebiet der Attacken durch entfernte Angreifer bzw. in das der Attacken durch Agenten und Wirten. Unter Benutzung dieser Unterteilung werden in diesem Abschnitt die wichtigsten Fragestellungen und eventuelle Lösungsmöglichkeiten durch existierende Verfahren in den einzelnen Teilbereichen skizziert.

3.3.1 Sicherheit gegen Attacken durch entfernte Angreifer

Da Wirte untereinander und auch Agenten unter Benutzung ihrer Wirte über ein möglicherweise unsicheres Netzwerk kommunizieren, kann es dazu kommen, dass Wirte und Agenten über dieses Netzwerk angegriffen werden.

Angriffe gegen Agenten

Mögliche Angriffe umfassen hier Angriffe gegen die Kommunikation eines Agenten mit Parteien außerhalb des eigenen Wirts, Angriffe gegen die Migration eines Agenten sowie „Denial-of-Service“-Angriffe durch entfernte Angreifer. Bis auf die letzte Angriffsart können alle diese Angriffe durch den Wirt verhindert werden. Da sich hierbei keine neuen Anforderungen aus der Mobilität der Agenten ergeben (auch migrierende Agenten sind aus Sicht der Wirte nur sicher zu übertragende Daten), können bestehende Mechanismen wie Verschlüsselung, Integritätssicherung und Authentifizierung eingesetzt werden. Unter „Denial-of-Service“-Angriffen werden hier nur solche verstanden, die sich nicht gegen den Wirt richten, und von diesem unerkannt als reguläre Kommunikation an den Agenten weitergereicht werden. Auch hierbei gilt, dass sich durch die Migrationsfähigkeit der Agenten keine neuen Anforderungen ergeben, und das eventuell bestehende Präventionsmechanismen auch hier eingesetzt werden können. Allerdings gibt es zurzeit wenig solche Gegenmaßnahmen gegen „Denial-of-Service“-Angriffe in verteilten Systemen allgemein.

Angriffe gegen Wirte

Mögliche Angriffe umfassen hier Angriffe gegen die Kommunikation eines Wirts mit Parteien außerhalb dieses Wirts sowie „Denial-of-Service“-Angriffe durch entfernte Angreifer. Da Wirte im Allgemeinen auch für die Migration und die Kommunikation der ausgeführten Agenten zuständig sind, betreffen diese Angriffe oft auch Agenten. Allerdings passen alle diese Angriffe in das herkömmliche Sicherheitsschema verteilter Systeme, weswegen sie einfach durch Anwendung bestehender Verfahren zur Verschlüsselung, Integritätssicherung und Authentifizierung verhindert bzw. entdeckt werden können. Auch hier gilt, dass Mechanismen zum Schutz

vor „Denial-of-Service“-Angriffen, so sie existieren, ohne Änderungen eingesetzt werden können.

3.3.2 Sicherheit gegen Attacken durch lokale Agenten und Wirte

Nicht nur durch die Benutzung eines unsicheren Netzwerks, auch schon zwischen Agenten auf demselben Wirt und zwischen Agenten und Wirten kann es zu Angriffen kommen.

Agent - Agent

Hierbei handelt es sich um mögliche Angriffe zwischen zwei Agenten. Diese lassen sich in technische und anwendungsspezifische Angriffe unterscheiden. Technische Angriffe benutzen Sicherheitslöcher in der Implementierung des Wirts, um direkt auf Ressourcen eines anderen Agenten zuzugreifen, anwendungsspezifische Angriffe versuchen im weitesten Sinne auf der Ebene der Anwendungsinteraktion zwischen zwei Agenten zu “betrügen”, etwa in dem Sinne, dass ein Dienstanbieter das Geld eines Dienstnutzers nimmt, ohne eine adäquate Leistung zu erbringen.

Anwendungsspezifische Angriffe sind ein allgemeines Problem gleichberechtigter Partner, die interagieren, etwa zum Zwecke des elektronischen Handels; es ist nicht spezifisch für mobile Agenten. Daher können hier, sofern es sich nicht sowieso um anwendungsspezifische Probleme handelt, auch dieselben Mechanismen eingesetzt werden wie in Anwendungen ohne mobile Agenten. Ein gutes Beispiel sind die Protokolle zur Bezahlung mit elektronischem Geld, bei denen es z.T. sogar verschiedene Ausbaustufen gibt, je nachdem, welche möglichen Angriffe seitens der Partner verhindert werden sollen, bzw. welche Anonymität der Kunde wünscht (siehe [SET97]).

Bei technischen Angriffen kann es, in Abhängigkeit von der verwendeten Architektur und den Sicherheitslücken der Wirtsimplementierung, für einen Agenten z.B. möglich sein, auf den anderen Agenten über ungeschützte Speicherbereiche direkt zuzugreifen und dessen Daten auszuspähen oder zu manipulieren. Programmiersprachen, die die Benutzung von mobilem Code vorsehen (z.B. Java oder Tcl), stellen hier oft bereits Mechanismen bereit, die zur Abschottung der einzelnen Agenten benutzt werden können und so nicht durch das Agentensystem bereitgestellt werden müssen.

Agent - Wirt

Den Schwerpunkt dieser Arbeit bildet die Sicherheit zwischen Agent und Wirt. Wie sich zeigen wird, entstehen hier neue Probleme durch den Umstand, dass ein Programm auf einem fremden Rechner ausgeführt wird. Während sich in einem Unterbereich, dem *Schutz des Wirts vor Angriffen durch die ausgeführten mobile Agenten*

auch Techniken einsetzen lassen, die bereits auf dem Gebiet des mobilen Codes Verwendung finden, gibt es im anderen Unterbereich, dem *Schutz des mobilen Agenten vor Angriffen durch den ausführenden Wirt*, wenige Vorarbeiten. Das Gebiet des Schutzes des Wirts vor Angriffen durch die ausgeführten mobilen Agenten wird Gegenstand des nächsten Kapitels sein. Das Gebiet Schutz des mobilen Agenten vor Angriffen durch den ausführenden Wirt wird den Gegenstand der Kapitel 5 bis Kapitel 9 bilden.

4 Schutz des Wirts gegenüber böswilligen Agenten

In diesem Kapitel wird die eine Richtung des Bereichs der Sicherheit zwischen Agenten und Wirten, nämlich der Schutz des Wirts vor Angriffen durch die ausgeführten mobilen Agenten behandelt. Zuerst wird dazu die Problemstellung näher erläutert und es werden einzelne Angriffsklassen aufgezählt. Danach werden exemplarisch einige Ansätze vorgestellt, die Teile dieses Problems lösen.

4.1 Die Problemstellung

Wenn sich die Betreiberpartei eines Wirts von der Auftraggeberpartei eines mobilen Agenten unterscheidet, kann es zu Angriffen des Wirts durch den mobilen Agenten kommen. Für den Betreiber eines Wirts ist es wichtig, diese Angriffe zu verhindern. Ist dies nicht möglich, wird er kaum einen solchen Wirt anbieten wollen. Daher berührt dieses Problem auch in hohem Maße die Akzeptanz der Mobile-Agenten-Technologie.

Ein sehr ähnliches Problem taucht auch bei anderen Mobile-Code-Systemen auf, z.B. beim Einsatz von Java-Applets in WWW-Browsern. In diesem speziellen Fall gibt es allerdings erstens den Unterschied, dass das Applet für den Betreiber der Ausführungsumgebung ausgeführt wird. Der Interessenkonflikt bei Applets besteht daher zwischen der Auftraggeber/Betreiberpartei und der *Programmierer*partei. Dies führt vor allem zu möglichen Angriffen gegen das Rechnersystem des Betreibers, nicht aber gegen die Ausführungsumgebung an sich, während Angriffe gegen den Wirt an sich durchaus Angriffsziele mobiler Agenten sein können. Ein zweiter Unterschied besteht darin, dass Applets nicht von einem Browser zu einem anderen Browser migrieren. Drittens sind Applets keine von außen addressierbaren Einheiten und kommunizieren in Folge dessen auch nicht direkt mit anderen Applets. Insgesamt konstituiert das Problem der sicheren Ausführung von Applets also ein Teilproblem des Problems der sicheren Ausführung von mobilen Agenten, so dass für manche Problembereiche bestehende Verfahren übernommen werden können.

Ebenfalls verwandt in einigen Aspekten ist das Problem des Schutzes des Wirts gegenüber böswilligen Agenten mit dem Schutz eines Mehrbenutzersystems vor den auf diesem System ausgeführten Benutzerprozessen. Auch hier beschränken sich die Angriffe auf solche gegen das Rechnersystem (das mit der Ausführungsumgebung identisch ist). Ein Unterschied zu Mobile-Agenten-Systemen ist, dass die Benutzergruppe geschlossen ist, und dass die Benutzer (bis auf Hacker) bekannt sind. Insgesamt gilt aber auch hier, dass in einigen Bereichen bestehende Verfahren für den Schutz des Wirts vor böswilligen Agenten übernommen werden können.

4.2 Angriffsklassen

Die generelle Problemstellung lässt sich in einzelne Angriffsklassen abbilden, denen dann einzelne Teilbereiche dieser Problematik, bzw. in Folge, auch einzelne Lösungsansätze entsprechen (siehe Bild 4).

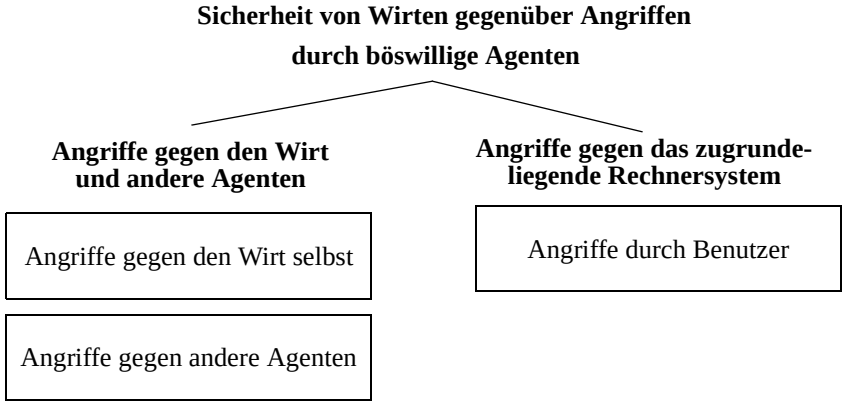


Bild 4: Grobkategorisierung der Angriffsklassen

4.2.1 Angriffe gegen den Wirt und andere Agenten

Der erste Bereich umfasst alle Angriffe gegen den Wirt, also den Teil eines Mobile-Agenten-Systems, der von einer Partei auf einem Rechner betrieben wird. Die meisten Angriffe sind möglich, wenn eine vollständige Kontrolle des Wirts durch den angreifenden Agenten zustande kommt. Dieser Fall sollte nie auftreten, entsteht also i.A. durch einen Fehler der Implementierung eines Systems. Denkbar ist ein solcher Effekt beispielsweise in einer Ausführungsumgebung, in der ein Agent über Zeiger auf die Daten und den Code des Wirts zugreifen kann oder wenn der Agent in der Lage ist, den Wirt durch eigenen Code zu ersetzen. Kontrolliert ein Agent einen Wirt vollständig, so kann der Agenten alles das tun, was auch der Wirt kann.

Ein Unterbereich der Angriffe innerhalb des Agentensystems besteht aus einzelnen Attacken gegen den Wirt selbst. Angriffsklassen in diesem Unterbereich umfassen:

Unberechtigtes Benutzen von Ressourcen

Wirte stellen Agenten Ressourcen zur Verfügung. Diese bestehen zunächst aus Rechenzeit, Speicherplatz und Kommunikationsressourcen, können aber auch aus anderen Dienstleistungen wie z.B. einem Lokalisierungsdienst für Agenten bestehen. Mobile Agenten verbrauchen diese Ressourcen in unterschiedlichem Maße. Die Frage, welcher Verbrauch für welchen Agenten unter welchen Umständen legi-

tim ist, und welcher einen Angriff darstellt, hängt von Kriterien ab, die der Wirt festlegt. Diese Kriterien können klar formulierbar und durch den Agenten abfragbar sein, sie können aber auch nur sehr implizit vorhanden sein. Solche Kriterien könnten z.B. den maximalen Verbrauch einer Ressource pro Agent umfassen oder eine Obergrenze, die von der Partei des Auftraggebers abhängt. Einige Agentensysteme werden unter Umständen sogar eine regelrechte Abrechnung des Ressourcenverbrauchs vorsehen, d.h. dass die Auftraggeber eines Agenten Geld für dessen Verbrauch aufwenden müssen.

„Denial-of-Service“-Angriffe gegen den Wirt

Ein Angriffsziel eines Agenten kann es sein, dass ein Wirt nicht mehr in der Lage ist, existierende oder neue Agenten auszuführen, z.B. um damit auszuschließen, das Konkurrenzangebote ausgenutzt werden können. Diese Art von Attacken werden als „Denial-of-Service“-Angriffe bezeichnet. Dies kann z.B. dadurch geschehen, dass ein oder mehrere Agenten auf einem Wirt versuchen, alle verfügbaren Ressourcen zu verbrauchen.

Ein zweiter Unterbereich der Angriffe innerhalb des Agentensystems besteht aus einzelnen Attacken gegen andere Agenten. Um diese gegen den Bereich der Sicherheit zwischen Agenten abzugrenzen, werden hier nur solche Angriffe betrachtet, an deren Verhinderung ein Interesse des Wirts besteht. Dieses Interesse kann z.B. darin begründet liegen, dass der Wirt seine Dienste gegenüber Agenten kostenpflichtig anbietet (alle Agenten also Kunden sind), oder darin, dass ein Wirt nicht wegen der Angriffe verantwortlich gemacht werden will. Angriffsklassen in diesem Unterbereich umfassen:

„Denial-of-Service“-Angriffe gegen andere Agenten

Dieser Angriff bezeichnet Attacken gegen andere Agenten, bei denen der Angreifer versucht, diese Agenten daran zu hindern, Dienste zu erbringen (z.B. um seine eigene, konkurrierende Dienstleistung monopolartig anbieten zu können). Solche Angriffe werden in der Regel versuchen, den anzugreifenden Agenten zu überlasten oder aber diesen Agenten durch das Ausnutzen bekannter Probleme abstürzen zu lassen.

Maskierung gegenüber anderen Agenten

Ein Agent kann versuchen, sich einem anderen Agenten gegenüber als ein dritter Agent auszugeben, etwa, um sich Leistungen zu erschleichen.

Direkter Zugriff auf Daten anderer Agenten

Da Agenten wichtige Daten wie digitales Geld oder kryptografische Schlüssel mit sich führen können, kann es ein Ziel eines anderen Agenten sein, diese Daten in Erfahrung zu bringen. Ebenso kann es ein Angriffsziel darstellen, bestimmte Daten eines anderen Agenten schreibend zu verändern. Besteht die Möglichkeit, direkt auf die Daten eines anderen Agenten zuzugreifen, sind beide Angriffe möglich.

4.2.2 Angriffe gegen das zugrunde liegende Rechnersystem

Neben den Angriffen, die sich gegen den Wirt, also den Teil des Agentensystems richten, der von einer Partei auf einem Rechner betrieben wird, gibt es auch noch die Gefahr der Angriffe, die sich gegen das *Rechnersystem* richten, auf dem der Wirt betrieben wird. In diesem Fall wird das Mobile-Agenten-System nur als (komfortable) Möglichkeit benutzt, das entsprechende Rechnersystem anzugreifen. Die Abwehr dieser Angriffsklasse ist besonders wichtig, weil sie gewährleistet, dass der Betrieb eines Mobile-Agenten-Systems das restliche Rechnersystem nicht gefährdet. Sind solche Angriffe jedoch möglich, so stellen sie nicht nur eine weitere Möglichkeit dar, ein Rechnersystem anzugreifen. Die (gewollten) Fähigkeiten mobiler Agenten zur Kommunikation und vor allem zur Migration erlauben es dem Angreifer wesentlich bequemer, Angriffe auch gegen andere Rechnersysteme durchzuführen. Er muss sich weniger mit der Heterogenität der Rechnersysteme beschäftigen, und die Migration ist “legal”, d.h. es wird auch keine Maßnahmen geben, eine solche Migration zu unterbinden (im Gegensatz z.B. zu Viren, bei denen schon beim Auftauchen klar ist, dass sie eine Gefahr darstellen).

Der Kanon der möglichen Angriffe gegen das vom Wirt benutzte Rechnersystem umfasst alle Angriffe, die einem Benutzer gegen das benutzte Rechnersystem möglich sind, u.a. auch alle bereits genannten Angriffe (wie Maskierung, „Denial-of-Service“, Lesen vertraulicher Daten usw.), nur jetzt nicht gegen den Wirt, sondern gegen das benutzte Rechnersystem, andere Rechnersysteme sowie Benutzer auf diesen Systemen gerichtet.

Zur Verhinderung solcher Angriffe sind normalerweise andere Mechanismen notwendig als bei den Angriffen gegen den Wirt. Der Grund hierfür liegt in der Tatsache begründet, dass es bei Angriffen gegen ein Rechnersystem nicht sehr schwer ist, berechtigten von unberechtigtem Gebrauch zu unterscheiden - ein mobiler Agent soll im Allgemeinen keinen direkten Zugriff auf die Ressourcen eines Rechnersystems haben, sondern ausschließlich über den Wirt agieren. Daher können alle entsprechenden Aktionen eines mobilen Agenten, die erkennbar sind, entweder von vornherein ausgeschlossen oder zur Laufzeit verwehrt werden. Auch hier gilt aber der Grundsatz, dass nicht erkennbare Aktionen, die sich z.B. aufgrund von Implementierungsfehlern ergeben, ein großes Problem darstellen.

4.3 Ansätze auf diesem Gebiet

Bereits heute gibt es Ansätze, die Teilprobleme des Schutzes des Wirts vor böswilligen Agenten lösen. Eine vollständige Auflistung dieser Ansätze würde an dieser Stelle zu weit führen, da der Fokus dieser Arbeit auf dem Schutz mobiler Agenten vor böswilligen Wirten liegt. Jedoch sollen im Folgenden einige Ansätze genannt werden, die exemplarisch für die Vielzahl der existierenden Systeme und Verfahren stehen.

4.3.1 Isolierung einzelner Agenten

Dieser Ansatz verfolgt das Ziel sicherzustellen, dass Agenten nicht direkt, d.h. z.B. über direkte Speicherzugriffe, auf andere Agenten und den Wirt zugreifen können. Dazu werden entweder Mechanismen der benutzten Programmiersprache oder des Betriebssystems verwendet. Ersteres ist bei allen Java-basierten Systemen der Fall da Java solche Mechanismen anbietet), z.B. bei Mole [BHR98], oder beim Aglets-System [LO98]. Letzteres ist im Mobile-Agenten-System Ara [PS97] der Fall. In diesem System wird jeder Agent unter Aufruf einer eigenen Interpreterinstanz als Prozess des Betriebssystems initialisiert. Da Prozesse in den meisten Betriebssystemen gegeneinander isoliert sind, kann der Angreifer z.B. keine Sicherheitslöcher in der verwendeten Programmiersprache verwenden.

4.3.2 Abschottung der Ressourcen

Um sicherzustellen, dass Agenten nur kontrolliert Zugriff auf Ressourcen erhalten, kann das “Sandbox”-Verfahren eingesetzt werden, das z.B. in der Sprache Java realisiert ist. Dazu werden einfach alle Möglichkeiten der Sprache, auf Ressourcen jenseits von Rechenzeit und Hauptspeicherverbrauch zuzugreifen, über eine zentrale Komponente (in Java das Objekt `SecurityManager`) geleitet, die jede solche Anforderung ablehnen oder ihr zustimmen kann. Solange keine Sicherheitslöcher in der Implementierung vorliegen, kann so z.B. durch einen Ressourcenkontrollmechanismus der Zugriff und der Verbrauch von Ressourcen geregelt werden. Bei den Mobile-Agenten-Systemen, die Java verwenden, wird meist auch dieser Mechanismus benutzt, z.B. in Mole [BHR98] oder im Aglets-System [LO98].

4.3.3 Authentifikation von Agenten

Um die Identität von Agenten zu verifizieren, muss ein Wirt die Agenten, die er empfängt, authentifizieren. Ist ein Agent authentifiziert, so heißt das nicht unbedingt, dass er einen Wirt nicht angreifen wird, aber der Wirt kann sich dann im Angriffsfall unter Umständen an den Auftraggeber oder den Ersteller eines Agenten wenden. Die Authentifizierung eines Agenten durch den Wirt können dann auch andere Agenten benutzen, um Interaktionspartner auf demselben Wirt zu authentifi-

zieren, sofern der Wirt vertrauenswürdig ist, d.h. korrekt arbeitet. Die Authentifizierung eines Agenten durch den empfangenen Wirt kann auf zwei Arten geschehen. Entweder wird ein bestehendes Verfahren benutzt, bei dem der Agent die Kenntnis eines Geheimnis nachweist (z.B. in [Röl99], wo X.509-Verfahren benutzt werden). Oder die Authentifikation erfolgt durch die Verifikation der digitalen Signatur des Agenten. Die Signatur erfolgt dabei vom Auftraggeber des Agenten. Da die Identität eines Agenten vor allem von der Identität des Auftraggebers abhängt und die Verifikation dieser Signatur sicherstellt, dass der Auftraggeber ein Geheimnis (in der Regel einen geheimen Schlüssel) kennt, reicht dieses zweite Verfahren bei mobilen Agenten aus. Dieses Verfahren wird z.B. von [CGH95] vorgeschlagen, und in [KT99] implementiert.

4.3.4 Ressourcenkontrolle

Bei der Kontrolle von Ressourcen kann man zwei Bereiche unterscheiden. Eine qualitative Kontrolle erlaubt es dem Wirt zu bestimmen, *ob* ein bestimmter mobiler Agent Zugang zu einer Ressource bekommt. Zu den qualitativ kontrollierten Ressourcen zählen vor allem die sicherheitssensitiven Ressourcen wie etwa der direkte Zugang zum Betriebssystem, zum Dateisystem oder zum Wirt-Management. Auch der Zugang zu einem Wirt selbst, also die Frage, ob ein bestimmter Agent überhaupt auf einen Wirt migrieren darf oder nach der Migration ausgeführt wird, zählt zu den qualitativ kontrollierten Ressourcen. Die quantitative Kontrolle erlaubt es dagegen dem Wirt zu bestimmen, *wieviel* einer Ressource ein Agent verbrauchen darf. Zu den so kontrollierten Ressourcen zählen typischerweise Rechenzeit, Speicherplatzverbrauch und die Kommunikation.

Um den Zugang zu und den Verbrauch von Ressourcen kontrollieren zu können, existieren verschiedene Verfahren. *Politiken* (engl. policies) erlauben die Formulierung von Regelsystemen, die es bestimmten Agenten erlauben, qualitativ oder quantitativ kontrollierte Ressourcen zu benutzen. Dabei kommen zur quantitativen Kontrolle meist nur feste Obergrenzen zum Einsatz. Ein typisches Beispiel für einen solchen Ansatz ist [KLO98]. Dort können solche Entscheidungen auf der Basis der Identität des Agenteneigentümers, der Identität des Agentenerstellers, dem vorherigen Wirt, dessen Betreiber sowie der Identität des Agenten selbst getroffen werden, wobei diese Elemente durch UND-, ODER- und NICHT-Operationen verknüpft werden können. Kontrollierte Ressourcen in diesem Ansatz umfassen das lokale Dateisystem, das Netzwerk, das Fenstersystem, Rechenzeit und Speicherplatz sowie Operationen auf anderen Agenten.

Einen anderen Ansatz zur qualitativen und quantitativen Ressourcenkontrolle geht das frühe System Telescript, bei dem Ausweise (engl. permits oder tickets) statt Politiken benutzt werden (siehe [TV96]). Ausweise werden dabei von den Agenten-

eigentümern oder den Wirten ausgestellt. Sie erlauben es, den maximalen Verbrauch von Ressourcen durch Agenten zu limitieren bzw. die Rechtmäßigkeit einer Ressourcenanforderung zu beweisen. Zur Kontrolle des Zugangs mobiler Agenten auf Wirten bzw. in Gruppen von Wirten, den sog. *Regionen*, wurden aber auch hier Politiken benutzt.

In Mobile-Agenten-Systemen, bei denen Benutzer für den Verbrauch von Ressourcen Geld zahlen, wird die Frage der quantitativen Kontrolle durch eine Protokollierung und Abrechnung des Ressourcenverbrauchs ersetzt, d.h. jeder Agent darf beliebig viele sicherheitsunkritische Ressourcen verbrauchen, solange er sie bezahlt. Ein kommerziell betriebenes Mobile-Agenten-System, das einen Verbrauch in dieser Art und Weise abrechnet, gibt es zurzeit nicht. Eine solche Protokollierungs- und Abrechnungskomponente wird aber in [Trä99] für das Mobile-Agenten-System Mole beschrieben. In [GM94] ist davon die Rede, dass es im System Telescript möglich war, Aktionen Teleclicks genannte Einheiten kosten zu lassen. Diese Teleclicks sollten dann durch echtes Geld bezahlt werden.

Wenn keine geldwerte Abrechnung benutzt werden soll und es eher das Ziel ist, die Ressourcen eines Wirts gerecht auf die vorhandenen Agenten aufzuteilen, können andere Mechanismen genutzt werden. Eine Möglichkeit ist der Einsatz von Scheduling-Verfahren (siehe z.B. [LP99]) wie bei Betriebssystemen, eine andere die Verwendung von künstlicher „Systemwährung“ (siehe [BKR98]). Die Idee hinter diesem letzten Ansatz ist, dass eine Monopolisierung von Ressourcen dadurch vermieden werden kann, dass jeder Agent initial gleich viel dieser “Währung” bekommt und dass der Verbrauch von Ressourcen gewisse Beträge dieser Währung kostet. Der Preis kann dabei durch mehrere alternative Verfahren ausgehandelt werden. Wenn z.B. das Ziel eines solchen Verfahrens eine gewisse Lastbalancierung ist, dann könnte der Preis für die Benutzung einer Ressource dynamisch mit der Auslastung der Ressource steigen.

4.3.5 „Proof-Carrying“-Code

Um schon vor einer Ausführung eines Agenten beweisen zu können, dass dessen Aktionen nicht gegen eine durch den Wirtsbetreiber spezifizierte Sicherheitspolitik verstoßen, stellten [NL98] ein Verfahren vor, das sie “Proof-Carrying-Code“nannten. Dazu erstellt der Eigentümer einen digitalen “Beweis” über den Auswirkungen der Agentenausführung, der mit dem Agenten mitgeschickt und automatisch verifiziert werden kann.

4.4 Zusammenfassung

Zusammenfassend lässt sich sagen, dass das Problem des Schutzes des Wirts gegenüber böswilligen Agenten keine grundsätzlich ungelösten Probleme bereithält,

jedenfalls keine, die sich speziell aus der Problematik bei mobilen Agenten ergeben. Überschneidet sich dieses Problem mit dem des Schutzes der Ausführungsumgebung von mobilem Code oder des Rechnersystems vor Benutzerprozessen (z.B. bei der Isolierung einzelner Agenten), so lassen sich die bestehenden Ansätze aus diesem Gebiet in der Regel anwenden. Andere Probleme, wie Ressourcenkontrolle und Authentifizierung lassen sich durch Erweiterung bestehender Verfahren lösen. Nicht gelöst ist bis jetzt das Problem der „Denial-of-Service“-Attacken, sowohl gegen Agenten als auch gegen den Wirt. Wie sich zuletzt in jüngster Zeit zeigte (siehe z.B. [CT00]), stellen diese Art von Angriffen auch in herkömmlichen netzbasierten Anwendungen ein großes, ungelöstes Problem dar. Da hier keine besonderen Probleme aufgrund der Mobilität der Agenten zu erwarten sind, können zukünftige Lösungsansätze auch in Mobile-Agenten-Systemen verwendet werden.

Nachdem in diesem Kapitel die eine Richtung des Sicherheitsbereichs Agent-Wirt, nämlich der Schutz des Wirts vor Angriffen durch die ausgeführten mobilen Agenten behandelt wurde, soll nun in den folgenden Kapiteln die Gegenrichtung untersucht werden. Dabei wird es sich zeigen, dass auf diesem Gebiet, dem Schutz mobiler Agenten vor Angriffen durch den ausführenden Wirt, mehr zu erarbeiten ist. Der Grund hierfür besteht darin, dass dieses Problem spezifisch für mobilen Code ist, der einen individuellen Zustand mit sich führt, also insbesondere auch für mobile Agenten. Da dieser Aspekt bei den bisherigen Spielarten mobilen Codes kaum eine Rolle gespielt hat, gibt es dazu auch wenig Vorarbeiten aus anderen Forschungsgebieten.

5 Schutz von mobilen Agenten gegenüber böswilligen Wirten

Da der Betreiber eines Wirts und der Auftraggeber eines mobilen Agenten getrennte Parteien sind, kann es zu Angriffen des Wirts gegen den Agenten kommen. Auch hier kann es wieder zwei Fälle geben: entweder befindet sich der mobile Agent nicht auf dem angreifenden Wirt, oder aber er wird vom angreifenden Wirt ausgeführt. Der erste Fall lässt sich wieder auf das bekannte Problem des Angriffs einer autonomen Partei gegen eine andere über ein Netzwerk zurückführen und birgt keine Besonderheiten seitens des Mobilitätsaspektes. Der Wirt ist in diesem Sinne einfach ein weiterer Interaktionspartner, wie etwa ein Agent auf einem anderen Wirt. Daher findet hier ebenfalls das in Kapitel 3.3.2 Gesagte Verwendung und wird nicht weiter diskutiert. Der Fall jedoch, dass ein Agent von dem Wirt angegriffen wird, der ihn eigentlich korrekt ausführen soll (“malicious host”), tritt erst dann auf, wenn ein Programm *von* einer anderen Partei ausgeführt wird als derjenigen, *für* die das Programm ausgeführt wird. Dieser Fall hat nicht nur für mobile Agenten und mobilen Code Bedeutung, auch in herkömmlichen Umgebungen wird teilweise Code für eine andere Partei ausgeführt. Wenn beispielsweise Teile eines Anwendungsprogrammes dazu da sind, zu verhindern, dass das Programm illegalerweise kopiert wird oder in einer anderen Umgebung als der vereinbarten benutzt wird, geschieht dies zum Nutzen des Programmerstellers, also einer anderen Partei als der des Benutzenden. In diesem Sinne handelt es sich hier ebenso um den Schutz des *Auftraggebers* vor Angriffen gegen seinen Agenten durch den ausführenden Wirt.

Auch in heutigen Systemen ist es meistens der Fall, dass ein Programm *von* einer anderen Partei (und damit auch für dieselbe) ausgeführt wird als der, die das Programm *erstellt* hat. In einigen Fällen hat der Programmersteller das Interesse, gewisse Angriffe des Programmausführenden zu verhindern. Diese Angriffe umfassen meist den Versuch, die dem Programm zugrunde liegenden Mechanismen aus dem Programm zu extrahieren (“Re-engineering”), etwa weil diese Mechanismen von kommerziellem oder sonstigem Interesse sind. Diese Problematik greift auch in Mobile-Agenten-Systemen, weil auch hier eine weitere Partei einen Agenten erstellt hat, und sie ihre Interessen gegenüber den anderen Parteien (z.B. dem Auftraggeber des Agenten und dem Betreiber des ausführenden Wirts) vertreten will. In diesem Sinne stellt dieser Aspekt den Schutz des *Programmerstellers* vor Angriffen gegen seinen Agenten durch den ausführenden Wirt oder den Auftraggeber dieses Agenten dar. Auch hier trägt der Mobilitätsaspekt nichts neues zur Problemstellung bei, weshalb dieser Aspekt hier nicht weiter vertieft werden soll. Allerdings ist durchaus zu prüfen, ob die Mechanismen, die das in dieser Arbeit behandelte Problem angehen, nicht auch zur Verhinderung von Re-engineering-Angriffen einge-

setzt werden können. Auf diesen Aspekt wird kurz in der Zusammenfassung (siehe Kapitel 10) eingegangen werden.

Im Folgenden wird daher unter “Schutz des Agenten vor böswilligen Wirten” immer der erste Aspekt verstanden, nämlich der Schutz des Auftraggebers vor Angriffen gegen seinen Agenten durch den ausführenden Wirt.

5.1 Ein Beispiel

Im Folgenden sollen nun solche Angriffe durch ein Beispiel illustriert werden. Das Beispiel besteht aus einem mobilen Agenten, der aus einer Liste von elektronischen Läden, die ein bestimmtes Gut (in diesem Fall Rosen) anbieten, denjenigen ermitteln soll, der den geringsten Preis für dieses Gut verlangt. Die verschiedenen Läden sind durch stationäre Agenten auf verschiedenen Wirten realisiert. Es wird angenommen, dass die Läden ein gemeinsames Interesse mit den Wirten haben, auf denen sie sich befinden, d.h. ein Wirt ist daran interessiert, dass der Kauf in “seinem” Laden durchgeführt wird. Der Auftraggeber des Agenten ist bereit, den Agenten den Betrag gleich in Form von elektronischem Geld an den billigsten Laden übergeben zu lassen, solange dieser Betrag unter einem vom Auftraggeber festgelegten Wert liegt. Der Agent besteht aus einem Datenteil und Code. Der Datenteil könnte wie in Bild 5 aussehen:

```
Address home = “PDA, sweet PDA”
Money wallet = 20.00
float maximumprice = 20.00$
good flowers = 10 red roses
Address shoplist[] = empty list
int shoplistindex = 1
float bestprice = 20.00$
Address bestshop = empty
```

Bild 5: Variablen des Beispielagenten

Die Variable `home` enthält die Adresse des Heimat-Wirts, dem nach der Rückkehr des Agenten ein eventuelles “Restgeld” zurückgeliefert wird. In `wallet` ist das elektronische Geld gespeichert, `maximumprice` gibt die vom Auftraggeber bestimmte Preisobergrenze an, `flowers` repräsentiert eine Spezifikation des zu kaufenden Gutes. Diese vier Variablen werden mit der Erzeugung des Agenten belegt und sind dann (bis auf `wallet`) konstant. Die nächsten vier Variablen werden für die Ausführung des Agenten dynamisch belegt. Die Variable `shoplist` enthält eine Liste der zu besuchenden Läden, `shoplistindex` ist ein Zeiger in diese Liste, der den nächsten zu besuchenden oder den gerade besuchten Laden

anzeigt. In `bestshop` steht eine Adresse des Ladens mit dem bisher billigsten Preis, der in `bestprice` abgelegt wird.

Der Code des Agenten ist in Bild 6 als Java-ähnlicher Pseudocode abgebildet.

```
1  public void startAgent() {
2
3    shoplist = getTrader().
4           getProvidersOf(“SellFlowers”);
5    shoplistindex = 0;
6    while (shoplistindex < shoplist.length) {
7       go(shoplist[++shoplistindex]);
8       if (shoplist[shoplistindex].
9                 askprice(flowers) < bestprice) {
10          bestprice = shoplist[shoplistindex].
11                   askprice(flowers);
12          bestshop = shoplist[shoplistindex];
13      }
14   }
15   // remote buy if price was below maximumprice
16   if (bestshop != empty) {
17          buy(bestshop,flowers,wallet);
18   }
19   // go home and deliver wallet
20   go(home);
21   if (location.getAddress() = home) {
22          location.put(wallet);
23   }
24 }}
```

Bild 6: Code des Beispielagenten

Der obige Agent ist für ein Mobile-Agenten-System gedacht, das transparente Migration (“strong migration”) unterstützt, d.h. der gesamte Zustand des Agenten samt Ausführungszustand wird automatisch auf den nächsten Wirt transferiert und der Agent macht dort mit der auf den Migrationsbefehl folgenden Zeile weiter.

Zuerst holt sich der Agent über einen Trader eine Liste mit Läden, die den gewünschten Dienst (“SellFlower”) anbieten (siehe Bild 6, Zeilen 3 und 4). Danach wird `shoplistindex` auf 0 gesetzt, um mit Zeile 7 zum ersten Laden migrieren zu können. Die Hauptschleife zwischen Zeile 6 und Zeile 14 arbeitet nach und nach die ganze Liste mit den Läden ab. Zuerst erfolgt eine Migration zum nächsten

Laden (Zeile 7), in den Zeilen 8 bis 13 werden `bestshop` und `bestprice` auf die aktuellen Werte des aktuellen Ladens gesetzt, wenn dessen Preis niedriger als der bisher niedrigste ist. Nach der Hauptschleife ist ab Zeile 15 die Liste der Läden abgearbeitet, d.h. alle Preisangebote eingeholt. In Zeile 17 wird daher dem Laden mit dem niedrigsten Preis insgesamt die Geldbörse übergeben. Um die Anforderung zu erfüllen, dass auch der niedrigste Preis unterhalb einer gewissen Obergrenze sein muss, wurde `bestprice` bereits in der Initialisierung auf `maximumprice` gesetzt, so dass nur Angebote gespeichert wurden, die billiger sind. In Zeile 16 wird daher nur noch abgeprüft, ob in `bestshop` die Adresse eines Ladens gespeichert ist, oder ob kein Laden einen Preis unterhalb der Obergrenze anbieten konnte. Ab Zeile 20 migriert der Agent zu seinem Heimat-Wirt und übergibt ihm die Börse mit dem nicht ausgegebenen Geld.

Ein Preisvergleichs- und Einkaufsagent wie eben vorgestellt ist das klassische Beispiel im Bereich der Sicherheit in Mobile-Agenten-Systemen, weil sich an ihm die meisten Angriffe demonstrieren lassen. Gleichzeitig ist eher unklar, warum eine solche Anwendung gerade durch mobile Agenten realisiert werden sollte. Die Sicherheitsproblematik entsteht nur durch die Verwendung von mobilen Agenten (d.h. sie tritt bei einer Client-Server-artigen Implementierung nicht auf), und Vorteile, die diese Nachteile für diese einzelne Anwendung aufwiegen könnten, sind nicht ersichtlich.

Um anhand dieses Beispiels mögliche Angriffe gegen mobile Agenten erläutern zu können, ist es hilfreich, zunächst zwischen den *Angriffszielen* des Angreifers und den *Angriffsmethoden* zu unterscheiden, die er einsetzen kann.

5.2 Mögliche Angriffsziele gegen mobile Agenten

Das Angriffsziel bezeichnet den Zweck, den ein Angriff gegen einen mobilen Agenten haben soll (*Was* will der Angreifer erreichen?). Angriffe gegen mobile Agenten können ein oder mehrere Ziele aus einer ganzen Vielzahl von möglichen Zielen haben. Diese Ziele hängen von der Motivation des Angreifers und den Eigenschaften des angegriffenen Agenten ab. Grundsätzlich lassen sich die möglichen Angriffsziele in drei Klassen unterteilen: Kenntnis eines Datums des Agenten, Modifikation eines Datums des Agenten sowie Änderung des Verhaltens eines Agenten. Diese Klassen gruppieren einzelne Angriffsziele, wobei es bisweilen durchaus abstraktere Ziele geben mag (z.B. schade dem Eigentümer des Agenten), die sich dann in einzelne Angriffsziele unterteilen lassen.

Kenntnis eines Datums des Agenten

Diese Klasse umfasst alle Angriffe deren Ziel es ist, Kenntnis eines Datenelements des Agenten zu erlangen. Dies gilt vor allem für Datenklassen, bei denen schon die

Kenntnis dieser Daten ausreicht, um dem Auftraggeber zu schaden. In unserem Beispiel trifft das für den Inhalt der Variablen `wallet` zu, die digitales Geld enthält, und deren Wert sich daher allein aus der Kenntnis dieser Daten ergibt. Auch die Kenntnis anderer Datenklassen kann für einen Angriff eingesetzt werden, selbst wenn diese Daten keinen Wert an sich besitzen. In unserem Beispiel könnte der Angreifer z.B. die Kenntnis des Inhalts der Variablen `maximumprice` oder `bestprice` benutzen, um seinen Laden ein Angebot abgeben zu lassen, das nur minimal besser als diese Werte ist, obwohl er normalerweise einen weit geringeren Preis verlangen würde.

Modifikation eines Datums des Agenten

Diese Klasse umfasst alle Angriffsziele, die Datenelemente des Agenten in ihrem Wert verändern. Ein solches Angriffsziel kann es beispielsweise sein, Hinweise in einem Agenten darauf zu löschen, dass ein bestimmer Wirt besucht wurde.

Änderung des Verhaltens eines Agenten

Die letzte Klasse umfasst alle Angriffsziele, die das Verhalten eines Agenten in einer bestimmten Art und Weise ändern wollen. Ein solches Angriffsziel kann z.B. sein, den Agenten in genau dem Laden einkaufen zu lassen, den der Angreifer bestimmt. Grundsätzlich kann man diese Klasse in gewünschte Verhaltensänderungen einteilen, die dauerhaft sind, sich also auch noch auf den folgenden Wirten auswirken und temporäre Verhaltensänderungen, die nur auf dem angreifenden Wirt zum Tragen kommen. Ein Ziel kann es beispielsweise sein, den Agenten an der weiteren Durchführung seiner Aufgaben zu hindern.

5.3 Mögliche Angriffsmethoden gegen mobile Agenten

Angriffsmethoden bezeichnen den Weg, den ein Angriff zum Erreichen des Angriffsziels nimmt (*Wie* geht der Angreifer vor?). Um ein übergeordnetes Angriffsziel zu erreichen, ergeben sich bisweilen Unterziele, die dann bereits den Charakter von Angriffsmethoden haben, weil sie den Weg bezeichnen das übergeordnete Angriffsziel zu erreichen. Diese Unterziele lassen sich dann entweder direkt durch die Anwendung einzelner Methoden oder durch das Erreichen weiterer Unterziele erreichen.

Angriffsmethoden lassen sich in die zwei Hauptkategorien „Leseangriffe“ und „Modifikationsangriffe“ unterteilen (siehe Bild 7), wobei diese sowie die Unterkategorien wie gesagt, nicht das Ziel eines Angriffs angeben, sondern nur die Methode. Eine Angriffsmethode beispielsweise, die eine Modifikation der Ausführung benutzt, kann durchaus das Ziel haben, Kenntnis eines Datums zu erlangen

(etwa wenn man die Ausführung so ändert, dass der Agent zum Kaufen gebracht wird, und so das elektronische Geld nach außen gibt).

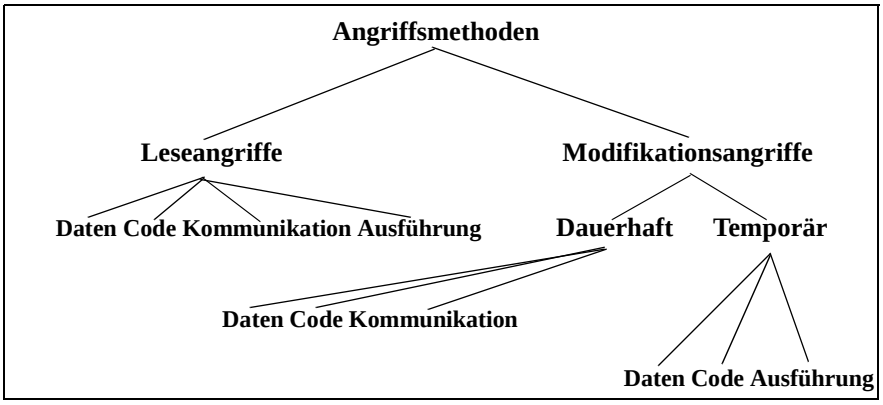


Bild 7: Kategorien von Angriffsmethoden

Die Kategorie der Leseangriffe lässt sich weiter in die vier Unterkategorien „Leseangriffe gegen Daten“, „Leseangriffe gegen Code“, „Leseangriffe gegen die Kommunikation“ sowie „Leseangriffe gegen die Ausführung“ unterteilen. Die Kategorie der Modifikationsangriffe lässt sich zunächst in die zwei Unterkategorien „dauerhafte Modifikationsangriffe“ und „temporäre Modifikationsangriffe“ einteilen. Dauerhafte Modifikationsangriffe wirken sich auch noch auf den folgenden Wirten aus, während temporäre Modifikationsangriffe nur auf dem angreifenden Wirt zum Tragen kommen. Temporär lassen sich alle die Element modifizieren, die nicht auch sofort bei Modifikation außerhalb des angreifenden Wirts sichtbar sind, also Daten, Code und Ausführung des Agenten. Daher lassen sich temporäre Modifikationsangriffsmethoden weiter in entsprechende Unterkategorien aufteilen. Entsprechendes gilt für die dauerhaften Modifikationen, wobei hier noch die Kommunikation des Agenten hinzukommt. Eine dauerhafte Modifikation der Ausführung eines Agenten ist hingegen nicht möglich, weil ein Wirt nur die Ausführung eines Agenten auf diesem Wirt beeinflussen kann.

Während die Kategorien der Angriffsmethoden durchaus aufzählbar sind, gilt selbiges nicht für die Angriffsmethoden selbst. Ein Datum eines Agenten kann z.B. durch das „Scannen“ des Datenbereichs (siehe unten) gefunden werden. Ein Schutzalgorithmus kann dieses Vorgehen durch das Aufteilen einzelner Datenelemente verhindern, aber ein Angreifer kann darauf wieder mit einer neuen Angriffsmethode reagieren, die bestimmte Eigenschaften des Schutzalgorithmus benutzt, usw. ad infinitum.

Im Folgenden sollen nun die einzelnen Kategorien von Angriffsmethoden beleuchtet werden. Dabei sollte beachtet werden, dass einzelne dieser Methoden durchaus diesselben Effekte haben können und damit denselben Angriffszielen dienen können.

Lesen der Agentendaten

Diese Kategorie enthält all diejenigen Angriffsmethoden, die Kenntnis eines oder mehrerer Datenelemente des Agenten zu erlangen suchen. Vier solcher Methoden sollen diese Kategorie illustrieren.

Direkter Zugriff auf Variablen

In einem Agenten, dessen genaue Struktur dem Angreifer bekannt ist, kann der Inhalt eines Datenelements über dessen Namen oder seine Position erreicht werden.

Scannen des Datenbereiches

Ist die genaue Struktur des Agenten unbekannt, liegen aber Informationen über das Format vor, in dem Daten abgelegt werden, besteht ein möglicher Angriff im „Scannen“ des Datenbereichs. Dazu wird versucht, das Vorliegen eines Elements in einem bestimmten Format zu erkennen. Dieses Vorgehen kann durchaus auch falsche Elemente hervorbringen (also solche, bei denen zwar das Format stimmt, die aber nur eine Fehlinterpretation anderer Elemente sind). Daher bedingt es dieser Angriff, dass es eine Möglichkeit geben muss, die Elementhypothesen zu überprüfen.

Blackbox-Tests

Ein Wirt kann einen Agenten auch angreifen, ohne die innere Struktur des Agenten zu benutzen, d.h. er kann den Agenten als *Blackbox* sehen. Dann kann der Angreifer versuchen, durch Tests Eigenschaften dieser Blackbox abzuleiten. Ein Blackbox-Test ist ein Angriff gegen einen mobilen Agenten durch einen Wirt, bei dem der Agent mehrere Male mit variierenden Eingabeparametern ausgeführt wird. Dies kann parallel oder sequentiell geschehen. Nach jeder Ausführung beobachtet der Angreifer den Effekt des Tests. Diese Effekte können in expliziten Resultaten wie z.B. Ausgabedaten bestehen, oder in charakteristischen “Aktivitätsmustern”. Das Ziel eines Blackbox-Tests ist es also, die Eingabeparameter zu finden, die zu einem bestimmten Effekt führen, oder aber, bestimmte Eigenschaften des Agenten zu erfahren. Der Bereich der Blackbox-Test-Angriffe wird in Kapitel 9 näher untersucht.

Kenntnis der Rückgabewerte beim Aufruf von Systemfunktionen

Da der Wirt Systemfunktionen berechnet, kennt er naturgemäß auch die so erzeugten Rückgabewerte. Dies kann z.B. bei den im vorherigen Abschnitt erwähnten kryptografischen Protokollen problematisch sein, da gerade die Kenntnis einer spe-

zifischen vom Agenten verwendeten Zufallszahl der entscheidende Schritt sein kann, Sicherheitsmaßnahmen zu unterlaufen.

Lesen des Agentencodes

Unter dieser Kategorie fallen alle Angriffsmethoden, die das Ziel haben, Kenntnis vom Code des Agenten zu erlangen. Die Kenntnis des Quellcodes des Agenten ist nicht unmittelbar ein verhindernswertes Gut, da wir uns ja auf den Schutz des Auftraggebers des Agenten beschränken. Wie wir jedoch später sehen werden, kann es ein sekundäres Schutzziel sein, dessen Erreichen verhindert, dass andere Angriffe stattfinden können, etwa Leseangriffe. Die Kenntnis des Codes erlaubt es dem Angreifer, Wissen über die Vorgehensweise des Agenten sowie die genaue Strukturierung des Programmes und der verwendeten Daten zu erhalten. Der Code des Agenten muss durch den Wirt soweit lesbar sein, wie er für die Ausführung benötigt wird. In unserem Beispiel ist das bis auf einige “then”-Zweige für den Großteil des Programms der Fall. Wenn es nur wenige verschiedene Klassen von Agenten gibt bzw. wenn diese Agenten aus Standardkomponenten erstellt sind, ist es durchaus wahrscheinlich, dass ein Wirt schon aus dem äußeren Verhalten des Agenten auf dessen Klasse bzw. seine Komponenten schließen kann und damit den genauen Code eines Agenten zur Gänze kennt.

Falls der Code des Agenten nicht sowieso ungeschützt ist, und daher dem Angreifer zur Gänze zur Verfügung steht, besteht eine Angriffsmethode darin, den Agenten so auszuführen, dass möglichst viel Code ausgeführt werden muss. Da die ausgeführten Codeteile lesbar sein müssen, kann so ein zumindest partielles Bild des Codes erstellt werden.

Lesen der Kommunikation des Agenten mit anderen

Da die Kommunikation des Agenten mit anderen Agenten über das Rechnersystem, auf das der Wirt Zugriff hat, oder gar über den Wirt selbst abgewickelt wird, hat dieser weitreichende Möglichkeiten der Einflussnahme auf dieser Ebene. Er kann z.B. die Kommunikation abhören, selbst wenn andere Angriffe nicht möglich sind. Wenn wir in unserem Beispiel annehmen, dass der Agent die Kaufaktion nicht lokal, sondern über entfernte Kommunikation tätigt, dann wäre der Wirt z.B. in der Lage, das elektronische Geld zu kopieren und auszugeben.

Lesen der Ausführung des Agenten

In diese Kategorie fallen alle Angriffsmethoden, die das Ziel haben, Eigenschaften der tatsächlichen Ausführung des Agenten zu erfahren und die von Leseangriffen gegen Daten unterscheidbar sind. Hierzu zählen insbesondere Angriffsmethoden, die den tatsächlichen Kontrollfluss erkennen können, d.h. z.B. welche Zweige einer

“if-then-else”-Operation ausgeführt werden. Das ist nicht weiter schwierig, weil diese Kenntnis die Voraussetzung zur Ausführung des Agenten ist.

Für die Kenntnis des Kontrollflusses gilt dasselbe wie für die Kenntnis des Codes: Obwohl kein unmittelbares Schutzziel, mag ein Mechanismus, der diese Kenntnis verhindert, andere Angriffe zumindest erschweren. Umgekehrt kann die Kenntnis über den Kontrollfluss bereits andere Informationen über den Agenten enthüllen. Beispielsweise kann man im Beispielagenten aus der Kenntnis des konkreten Kontrollflusses Erkenntnisse über den Inhalt der Variablen `bestprice` gewinnen (weil dann, je nach Angebot, Zeile 10 ausgeführt wird oder nicht). Auch hier gilt, dass ein Wirt den Kontrollfluss des Agenten in dem Grade kennen muss, wie er ihn zur Ausführung des Agenten benötigt.

Dauerhafte Modifikation der Agentendaten

Diese Kategorie enthält Angriffsmethoden, die Agentendaten dauerhaft, d.h. auch noch auf den folgenden Wirten sichtbar, modifizieren.

Direkte Manipulation von Daten

Wenn der Angreifer die Struktur, die Semantik und den Ort einer Information kennt, dann ist er in der Lage, diese Information zu modifizieren. In unserem Beispiel könnte das der Wirt benutzen, um die Liste der zu besuchenden Läden zu kürzen, nachdem er das Angebot seines Ladens in `bestprice` bzw. `bestshop` gespeichert hat.

Rückgabe falscher Werte beim Aufruf von Systemfunktionen

Ein Agent als Programm ruft bisweilen Systemfunktionen auf, die vom Wirtssystem bereitgestellt werden. In den Zeilen 3 und 4 unseres Beispiels fragt der Agent nach einer initialen Liste mit Läden, die das gewünschte Gut anbieten. Hier kann der Wirt angreifen und eine Liste zurückgeben, die nur den Laden des Wirts enthält. Neben diesen direkten Angriffen birgt die Möglichkeit der Kontrolle der Systemfunktionen durch den Wirt jedoch auch noch die Gefahr, dass Schutzmechanismen unterlaufen werden können, die auf die Korrektheit solcher Werte angewiesen sind. Für einige kryptografische Protokolle beispielsweise ist es notwendig, eine Zufallszahl zu bekommen, oder aber die genaue Systemzeit. Durch die Kontrolle beider Werte können derartige Protokolle dann nicht mehr unmodifiziert zum Schutz gegen den Wirt eingesetzt werden.

Dauerhafte Modifikation des Agentencodes

Wenn der Wirt in der Lage ist, den Code des Agenten unbemerkt dauerhaft zu modifizieren, kann er dessen Verhalten ändern, oder Viren oder trojanische Pferde einfügen. Auf unser Beispiel angewandt könnte dann ein Angreifer das Verhalten des

Agenten so ändern, dass er immer einen bestimmten Laden, unabhängig vom Preis, bevorzugt.

Dauerhafte Modifikation der Kommunikation des Agenten mit anderen

Neben dem Abhören ist der Wirt unter Umständen auch in der Lage die Kommunikation des Agenten mit anderen zu modifizieren. Damit kann es dem Wirt z.B. möglich sein, im Namen des Agenten zu agieren, oder sich als den Kommunikationspartner auszugeben. Wenn unser Einkaufsagent daher die Kaufaktion über eine entfernte Kommunikation ausführt, dann könnte der Wirt beispielsweise den Kauf an einen anderen Laden umleiten.

Temporäre Modifikation der Agentendaten

Unter diese Kategorie fallen alle Angriffsmethoden, die Daten des Agenten auf dem angreifenden Wirt modifizieren, solange diese Modifikationen nicht auch auf dem nächsten Wirt sichtbar sind (d.h. entweder vor der Migration wieder zurückgenommen werden oder nicht weitertransportiert werden).

Temporäre Modifikation des Agentencodes

Unter diese Kategorie fallen Codemanipulationen, die spätestens zur Weitermigration des Agenten wieder zurückgenommen werden, d.h. nur während der Ausführung auf dem angreifenden Wirt wirken. Die Effekte dieser Angriffsmethoden sind von einer temporären Modifikation der Art der Agentenausführung ununterscheidbar, aber es mag jeweils einfacher sein, die eine oder die andere Angriffsmethode durchzuführen.

Temporäre Modifikation der Agentenausführung

Diese Kategorie enthält alle Angriffsmethoden, die den Agenten in einer anderen Art und Weise ausführen, als dies ein nicht-angreifender Wirt tun würde. Dies kann verschiedene Gebiete der Programmausführung umfassen, z.B. den Kontrollfluss.

Modifikation des Kontrollflusses

Selbst wenn der Wirt keinen direkten Zugang zu den Daten eines Agenten hat, kann er das Verhalten des Agenten dadurch beeinflussen, dass er den Kontrollfluss modifiziert. In unserem Beispiel könnte der Angreifer den Agenten dazu bringen, den Laden des Angreifers zu bevorzugen, indem er den Kontrollfluss der While-Schleife in Zeile 6 und den der if-Anweisung in Zeile 16 ändert.

Das Vorgehen bei dieser Art von Angriff kann entweder in einer Modifikation der Laufzeitumgebung (also des Wirt-Systems) im Vergleich zu einer Referenzumgebung bestehen, oder aber in einer temporären Modifikation des Agentencodes, die

bei der Migration auf den nächsten Wirt wieder rückgängig gemacht wird und so im Nachhinein nicht mehr entdeckbar ist.

Ausführungsverweigerung

Ein Wirt kann die Ausführung eines Agenten komplett oder für eine gewisse Zeit verweigern. Dies kann dann ein Angriff sein, wenn z.B. ein Wirt von einem zeitlich begrenzten Angebot eines anderen Blumenladens weiß, und den Agenten einfach so lange aufhält, bis dieses Angebot ausgelaufen ist. Dieser Angriff kann auch benutzt werden, um die Spuren eines anderen Angriffs zu verwischen, etwa wenn der Wirt das elektronische Geld des Agenten kopiert und ausgegeben hat. Da bei einigen Systemen, die elektronisches Geld bereitstellen, ein zweites Ausgeben des gleichen Geldes zur Offenlegung der Zahlungsvorgänge führen würde, muss der Wirt den Agent nach dem Angriff nur löschen um nicht in diese “Falle” zu geraten. Ein Problem bei diesem Angriff ist allerdings, dass es auch das Recht eines Wirts sein kann, die Ausführung eines Agenten zu verweigern, etwa, weil dieser versucht, den Wirt anzugreifen, oder weil er aus technischen Gründen nicht ausgeführt werden kann. Wenn es keine Mechanismen zur Fehlertoleranz gibt, kann es außerdem sein, dass ein Agent nicht ausgeführt wird, weil der Wirt zusammenbricht.

5.4 Der Einfluss der Organisationsstruktur auf die Sicherheit

Nicht in allen Anwendungsszenarien sind böswillige Wirte ein Problem, das mit technischen Mitteln angegangen werden muss. Oft erlaubt es die Organisationsstruktur des Agentensystems, Angriffe gegen mobile Agenten auszuschließen oder zumindest unwahrscheinlich zu machen. Diese Organisationsstrukturen lassen sich nach der Frage unterteilen, wer mobile Agenten und Wirte betreiben darf. Die folgenden drei Strukturen sollen dabei nur die mögliche Bandbreite demonstrieren, weitere Zwischenstufen sind denkbar.

5.4.1 Zentraler Betreiber

In geschlossenen Systemen wird das gesamte System nur von einer zentralen Partei betrieben, d.h. Agenten und Wirte können nur von dieser einen Partei betrieben werden. In einem solchen geschlossenen System kann es natürlich auch zu Angriffen kommen, aber diese sind dann die interne Angelegenheit einer Partei. Der Sicherheitsbedarf einer solchen geschlossenen Anwendung ist relativ gering; alles, was ausgeschlossen werden muss, ist, dass fremde Agenten in das System gelangen bzw. dass mobile Agenten, die zum System gehören, auf fremde Wirte umgeleitet werden. Mittels Authentifikation sind diese Anforderungen aber leicht lösbar. Die Anwendungsbereiche für geschlossene Systeme sind im Wesentlichen Intranet-

Anwendungen wie Netzwerk- und Systemmanagement, parallele Berechnungen sowie Informationssuche im Intranet.

5.4.2 Zentraler Betreiber und dritte Parteien

In plattformartigen Systemen werden alle Wirte nur von einer Partei betrieben. Agenten dagegen können von beliebigen dritten Parteien eingebracht werden. Auch in einem solchen System sind Angriffe nicht ausgeschlossen, aber weil die zentrale Betreiberpartei des Systems normalerweise eine vertrauenswürdige Institution ist, können die anderen Parteien davon ausgehen, dass Angriffe seitens des Wirts nicht im Interesse dieser Partei liegen (weil diese einen Ruf wahren muss). Gegenüber geschlossenen Systemen entsteht ein zusätzlicher Sicherheitsbedarf nach Maßnahmen, die den Wirt vor möglichen Angriffen durch die Agenten schützen, sowie solchen, die Agenten voreinander schützen. Mit plattformartigen Systemen lassen sich im Prinzip alle Anwendungen betreiben, die einzige Schwierigkeit liegt in solchen Anwendungen, die bereits bestehende Systeme (sogenannte *legacy*-Systeme) integrieren müssen, weil sich diese wahrscheinlich nur sehr schwer als mobile Agenten in das System einbringen lassen (gegen eine externe Anbindung dieser Systeme wie das im BTX-System der Fall war, spricht natürlich nichts). In der Praxis weisen plattformartige Systeme jedoch zwei Probleme auf: erstens muss es eine Partei geben, die willens ist, eine solche Plattform zu betreiben. Zweitens muss eine solche Plattform eine gewisse Startmasse aufweisen, um überhaupt benutzt zu werden (eine Diskussion dieser Probleme erfolgt in Kapitel 6.3).

Plattformartige Systeme eignen sich gut für Mobile-Agenten-Systeme, die als kommerzielle Dienste angeboten werden, d.h. bei denen die Ausführung von mobilen Agenten abgerechnet wird, und in deren Rahmen auch dritte Parteien kommerzielle Dienste an andere Agenten anbieten können, die über die zentrale Betreiberpartei abgerechnet werden können. In die Kategorie der plattformartigen Systeme gehört auch das bisher einzige System, das mobile Agenten kommerziell anbot: Der PersonaLink-Dienst [Mob96] war ein System, das von AT&T angeboten wurde, und auf der Mobile-Agenten-Technologie Telescript von General Magic beruhte. Er erlaubte es den Benutzern, auf der Basis von Telescript aktive Nachrichten an andere Benutzer zu schicken.

5.4.3 Nur dritte Parteien

In offenen Systemen schließlich können sowohl mobile Agenten als auch Wirte von beliebigen dritten Parteien eingebracht werden. Der zusätzliche Sicherheitsbedarf gegenüber plattformartigen Systemen ist der Schutz der Wirte vor Angriffen durch andere Wirte und der Schutz von Agenten vor Angriffen durch Wirte. Da dies die

allgemeinste Form eines Mobile-Agenten-Systems darstellt, sind auch alle Anwendungen damit möglich.

5.5 Die Alternative: Client-Server

Wenn das Anwendungsszenario unbekannte Wirte vorsieht, von denen man nicht weiß, ob sie einen mobilen Agenten angreifen, muss das Problem technisch gelöst werden. Dies kann dadurch geschehen, dass Sicherheitsmechanismen für mobile Agenten benutzt werden, oder aber dadurch, dass statt mobilen Agenten eine Technologie mit weniger Angriffsmöglichkeiten durch den Wirt verwendet wird: die Client-Server-Technik. Dies entspricht dem Fall, dass ein Agent nicht auf unbekannte Wirte migriert sondern ausschließlich auf einem oder mehreren vertrauenswürdigen Wirten ausgeführt wird bzw. von dort aus entfernt kommuniziert.

Dieses Vorgehen wird nicht in allen Fällen möglich sein. Wenn

- kein vertrauenswürdiger Wirt zur Verfügung steht
- es zwar einen solchen gibt, dieser aber keine dauernde Verbindung zu anderen Wirten aufbauen kann (z.B. falls er sich auf einem mobilen Gerät befindet)
- diese Verbindungen bzw. der Wirt nicht performant genug sind um die Anforderungen der Anwendung zu erfüllen

muss der Agent auch auf unbekannte Wirte migrieren.

Falls es aber möglich ist, die Client-Server-Alternative zu benutzen, bedeutet dies auch, dass dann die Verwendung von mobilen Agenten unter Einsatz der Sicherheitsmechanismen Vorteile gegenüber der Benutzung der Alternative bieten muss. Diese Vorteile können qualitativer oder quantitativer Natur sein, aber sie müssen eventuelle Nachteile (etwa eine gewisse Verschlechterung der Leistung) zumindest aufwiegen. Die Betrachtung der Vor- und Nachteile sowie deren quantitative Einschätzung sind unter Umständen anwendungsspezifisch oder hängen sogar von dynamischen Gegebenheiten wie etwa der Menge an übertragenen Daten ab. Sie stellt aber einen Unterschied zu anderen Gebieten im Bereich der Sicherheit in Verteilten Systemen dar, bei denen es häufig keine alternativen Technologien gibt, allenfalls alternative Algorithmen.

5.6 Ein Modell der Wirtsmaschine und des Angreifers

Obwohl die im letzten Kapitel geschilderten Angriffsmöglichkeiten des Wirts gegenüber dem ausgeführten Agenten intuitiv klar erscheinen, ist es nicht ohne weiteres möglich, alle Fragen im Zusammenhang möglicher Angriffe und Gegenmaßnahmen zu beantworten.

Dies soll an einem kleinen Beispiel demonstriert werden. Dazu nehmen wir an, dass ein Aspekt des Beispielagenten aus Kapitel 5.1 geschützt werden soll. Der Agent führt eine Kaufoperation aus, die, abstrahiert, in Bild 8 abgebildet ist. Der erste Parameter dieser Operation, A, soll das elektronische Geld enthalten, das zum Kauf benötigt wird. Da es die Kenntnis des Inhalts der Variablen A dem Angreifer erlauben würde, das elektronische Geldstück zu kopieren und auszugeben, soll diese Variable vor Leseangriffen im Speicher geschützt werden. Dazu soll der Inhalt der Variablen A, B und C in Teile zerlegt werden, die in neu angelegten Variablen gespeichert werden, so dass ein Angreifer den entsprechenden Bitstring nicht mehr im Agenten finden kann.

```
buy(A, B, C)
```

Bild 8: Originaler Code

Bild 9 zeigt eine Möglichkeit, den Inhalt von A wieder aus einer solchen Zerlegung zurückzugewinnen. Eine solche Zerlegung benutzend kann jetzt die originale Kaufoperation in eine geschützte transformiert werden, die diese Zerlegung für den Kauf rückgängig macht.

```
A = (v230 XOR v194)+ v233
B = (v193 Shift v231) * v190
C = (v191 - v234) Shift v232
```

Bild 9: Transformation

In einem solchermaßen geschützten Agenten gibt es nun keine einzelne Variable mehr, die das elektronische Geld enthält. Die Fragen aber, ob nun der Angreifer diesen Bitstring nicht wieder finden kann und ob es weitere Maßnahmen gibt, diesen Angriff zu verhindern, sowie Überlegungen zur Performanz solcher Maßnahmen, können nicht diskutiert werden ohne genauer auf die Ausführungsumgebung einzugehen. Was benötigt wird, um mögliche Angriffe näher zu untersuchen und Schutzmechanismen zu diskutieren, ist also ein genaues Modell der Ausführung des Agenten durch den Wirt. Allerdings ist das Verarbeitungsmodell aus Kapitel 2.2 nicht detailliert genug, um das spezifische Angriffspotential darzustellen. Daher wird ein weiteres Modell benötigt, das in der Lage ist, die Möglichkeiten eines angreifenden Wirts aufzuzeigen und als Grundlage zu weiteren Untersuchungen zu dienen. Ein solches Modell muss detailliert genug sein, um real verwertbare Erkenntnisse liefern zu können, und allgemein genug, um dem Wirt alle möglichen Angriffe zu erlauben.

```
buy((v230 XOR v194)+ v233,
      ,(v193 Shift v231) * v190
      (v191 - v234) Shift v232)
```

Bild 10: Resultierender Code

Um ein adäquates Modell zu erhalten, muss erst einmal analysiert werden, wer bzw. was der Angreifer ist und welche Anforderungen an das Wirtsmodell sich daraus ergeben.

5.6.1 Wer oder was ist der Angreifer?

Um uns ein Bild vom Angreifer zu machen, müssen wir zwei Fälle unterscheiden. Im ersten Fall kennt der Angreifer den Typ des anzugreifenden Agenten nicht. Daher sind entweder nur sehr einfache Angriffe möglich, wie das automatische Erkennen von Dateneinheiten (z.B. elektronischem Geld) durch das Suchen nach bestimmten Formaten, oder ein menschlicher Angreifer muss den Agenten erst analysieren, d.h. ein mentales Modell davon erzeugen. Zwar kann dieser menschliche Angreifer computerunterstützte Werkzeuge zur Analyse benutzen, aber Menschen arbeiten im Vergleich zur Ausführungsgeschwindigkeit von Rechnern sehr langsam. Dies ist deshalb von Bedeutung, weil ein Agent unter normalen Bedingungen durchaus seine Aufgabe auf einem Wirt beendet haben kann, bevor der menschliche Angreifer dazu kommt, die letzte Programmzeile gelesen zu haben. Zwar ist der Angreifer durchaus in der Lage, die Ausführung des Agenten für eine gewisse Zeit zu verzögern, aber man kann sich leicht einen Mechanismus vorstellen, der nach einer bestimmten Zeit Maßnahmen (wie ein Umleiten einer Kopie des Agenten auf einen anderen Wirt) zum Schutz des Agenten ergreift, die zu kurz für eine manuelle Analyse ist. Der Unterschied in der Verarbeitungsgeschwindigkeit von Menschen und Computern kann grundsätzlich nicht verändert werden (im Gegenteil, die relative Geschwindigkeit von Rechnern nimmt von Jahr zu Jahr zu).

Im zweiten Fall kennt der Angreifer den genauen Typ des Agenten zum Zeitpunkt der Ausführung. Dieser Fall wird vor allem dann auftreten, wenn es in einem Agentensystem nur wenige Agententypen gibt. Dann wird es möglich sein, den Typ z.B. über das Verhalten des Agenten zu bestimmen. Den genauen Typ des Agenten zu wissen, erlaubt es aber auch, im Vorfeld erstellte Analysen der bekannten Agententypen zu benutzen, insbesondere solche, die eine Feinspezifikation des Agenten erzeugen (z.B. mittels Reengineering-Techniken). Mittels einer solchen Feinspezifikation kann dann sogar ein Agent im Quellcode erzeugt werden, der die gleiche Spezifikation erfüllt (falls der Quellcode eines Standardagenten nicht sowieso vorliegt). Wenn nun aber die Feinspezifikation oder der Quellcode eines Agenten schon vor der Ausführung bekannt ist, kann ein Angriff automatisiert erfolgen, d.h. in Form eines *Angriffsprogrammes* vorliegen, da die Analyse des Codes schon im Voraus erfolgen kann. Ein solcher Angriff kann ein sehr begrenztes Ziel haben, z.B. die Kenntnis des Inhalts einer bestimmten Variablen. Da ein solcher Angriff als Programm vorliegt, kann er (im Vergleich zum manuellen Angriff) natürlich sehr schnell und sogar parallel durch mehrere Programme erfolgen.

Da der zweite Fall den ersten enthält (wenn ein Schutz möglich ist, falls der Angreifer den Typ des Agenten kennt, ist dieser Schutz auch möglich, falls dieser Typ unbekannt ist) wollen wir uns im Folgenden auf die Verhinderung des zweiten Falls

beschränken. Der Wirt greift also den mobilen Agenten durch ein oder mehrere Programme an, die in der Lage sind, die Ausführung des Agenten zu kontrollieren.

Nachdem wir jetzt wissen, wer bzw. was der Angreifer ist, muss jetzt analysiert werden, welche Anforderungen an das Wirts- bzw. Angriffsmodell zu stellen sind.

5.6.2 Anforderungen

Die ersten sechs Anforderungen resultieren aus den Möglichkeiten eines Angreifers, der auf den Rechner und die Ausführungsumgebung des auszuführenden Agenten weitestgehenden Zugriff hat. Um die Allgemeingültigkeit des Wirts- bzw. Angriffsmodells nicht zu beschneiden, wurden diese Anforderungen analog zu den möglichen Angriffsmethoden gewählt. Dadurch ergeben sich Anforderungen, die, wie bei den Methoden, eventuell die gleichen Angriffsziele erfüllen. Da man sich jedoch vorstellen kann, dass ein Schutzmechanismus eine Angriffsmethode verhindert, nicht jedoch eine zweite, die ebenfalls das Angriffsziel erfüllt, müssen beide Angriffsmethoden im Wirts- bzw. Angriffsmodell durchführbar sein. Die letzte Anforderung resultiert außerdem aus den Möglichkeiten, die sich dem mobilen Agenten in geeigneten Umgebungen zum Schutz vor Angriffen bieten.

Anforderung 1: Der Angreifer kann den Datenteil des mobilen Agenten lesen und verändern.

Da ein Angreifer den Speicher des Rechnersystems durchsuchen und modifizieren kann, wenn er Systemprivilegien hat, muss er im Modell in der Lage sein, zu jedem Zeitpunkt auf den aktuellen Stand des Datenteils des Agenten in der gleichen Weise zuzugreifen. Dieser Teil kann Variableninhalte, Konstanten oder beliebige Bitstrings umfassen. Da der Angreifer den transportierten Code sieht und die Anweisungen, die der Wirt auszuführen hat, kennt er zumindest zu diesen Zeitpunkten auch die Art des Zugriffs auf einzelne Daten.

Anforderung 2: Der Angreifer kann den Code des Agenten lesen und (temporär) verändern.

Letztendlich ist auch Code nur ein spezielles Datenkonstrukt, daher gilt auch für Code die Anforderung 1 bezüglich Les- und Änderbarkeit. Auch hier gilt wieder, dass der Angreifer über die gesamte Ausführung den Codeteil eines Agenten lesen und manipulieren kann, dass aber erst zum Zeitpunkt der tatsächlichen Ausführung einzelne Operationen wirklich in der verwendeten Form existieren müssen. Änderungen des Codes eines Agenten sind immer nur während der Ausführung auf einem Wirt möglich, insbesondere wird modifizierter Code nicht weitermigriert. Diese Einschränkung resultiert daher, dass es möglich ist, Code, der konstant über der Lebensdauer des Agenten ist, mittels digitaler Signaturen durch den Agenteneigentümer auf Modifikationen hin zu überprüfen.

Anforderung 3: Der Angreifer kann den Ausführungszustand lesen und verändern.

Der Angreifer ist nicht nur in der Lage, die “expliziten” Bestandteile eines Agenten, also Code und Daten zu kontrollieren, sondern insbesondere auch die Elemente der Ausführungsumgebung, die benutzt werden, um den Agenten auszuführen. Diese Elemente umfassen z.B. Parameter- und Rückgabestapel, den Programmzähler oder die Register. Diese sind durch den Angreifer in beliebiger Weise manipulierbar und er kann durch sie u.a. auf Werte zugreifen, die nicht explizit im Agenten vorkommen, sondern beispielsweise nur das Zwischenergebnis der Berechnung eines Ausdrucks im Programm sind. Durch diesen Zugriff ist aber auch vor allem die Art und Weise kontrollierbar, wie der Agent abgearbeitet wird.

Anforderung 4: Der Angreifer kann die Ausführungsweise des Agenten verändern.

Die Ausführungsumgebung eines Programmes realisiert eine Regelmenge, die bestimmt, wie ein Programm in einer bestimmten Programmiersprache ausgeführt wird. Einige Angriffe könnten andere Regeln benutzen als die, die eigentlich für eine Programmiersprache verbindlich sind. Beispielsweise könnte ein Angreifer immer den einen Zweig einer “if”-Anweisung ausführen lassen, unabhängig davon ob die “if”-Bedingung wahr oder falsch ist (z.B. weil gerade diese Anweisung einen Sicherheitsmechanismus realisiert). Daher muss der Angreifer im Modell in der Lage sein, diese Regelmenge zu kontrollieren.

Anforderung 5: Der Angreifer kann das Ergebnis von Aufrufen von Systemfunktionen kontrollieren.

Mobile Agenten sind als Programme passive Einheiten. Das bedeutet u.a., dass sie ihre Umgebung nur über das Laufzeitsystem erfahren können, zumindest jedoch, dass der Wirt diesen Aspekt kontrollieren kann. Wenn ein Agent die aktuelle Systemzeit wissen oder eine Zufallszahl produzieren will, muss er die Hilfe des Wirts mittels Aufruf einer Systemfunktion in Anspruch nehmen. Damit hat der Wirt die Kontrolle über das Wissen des Agenten über die Umgebung. Wenn man sich vorstellt, dass z.B. viele kryptografische Verfahren im Bereich der Authentifizierung Zufallszahlen oder die Systemzeit benötigen, um nicht umgangen zu werden, kann man sich vorstellen, dass auch dieser Aspekt sicherheitskritisch ist. Für unser Modell heißt das, dass der Angreifer die Resultate solcher Systemfunktionen beliebig manipulieren können muss.

Anforderung 6: Der Angreifer kann die Kommunikation des Agenten mit dritten Parteien lesen und verändern.

Ein Agent kann mit einer dritten Partei nur dann kommunizieren, wenn er die Ausführungsumgebung oder das Rechnersystem benutzt, das der Angreifer kon-

trolliert. Daher muss der Angreifer in der Lage sein, solche Kommunikation mit Dritten zu lesen bzw. zu modifizieren.

Anforderung 7: Das Modell soll es dem Agenten erlauben, Code berechnen zu können.

Einige Schutzmechanismen sehen vor, Teile des Codes während der Ausführung zu entschlüsseln. Weiter kann man sich Schutzmechanismen vorstellen, die mit selbstmodifizierendem Code arbeiten. Um diese Möglichkeiten auch im Angriffsmodell zu haben, muss es eine Möglichkeit geben, Agentencode aus Daten zu berechnen.

Bevor nun das Modell eines angreifenden Wirts (das sog. Angriffsmodell) vorgestellt wird, soll erst ein Teilmodell beschrieben werden, das die Aspekte der normalen, d.h. nicht-angegriffenen Ausführung eines mobilen Agenten beschreibt. Dieses Modell der Wirtsmaschine wird im Folgenden kurz Wirtsmodell genannt.

5.6.3 Ein Wirtsmodell

Unser Wirtsmodell realisiert eine konventionelle Von-Neumann-Architektur, da diese noch immer die Grundlage der allermeisten Computer bildet.

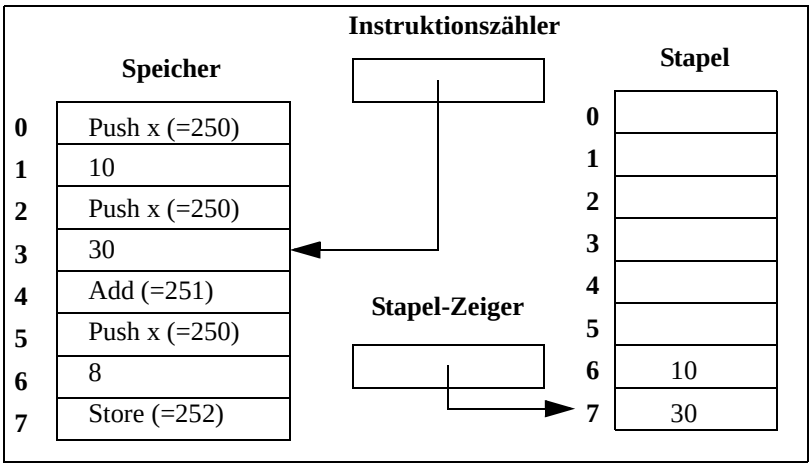


Bild 11: Modell eines Wirts

Unser Wirtsmodell besteht aus einer Liste von Speicherzellen, einem Instruktionszähler, einem Stapel samt Zeiger, sowie einem Instruktionssatz. Der Instruktionssatz erlaubt Sprünge im Programm an beliebige Register, das Berechnen von Werten, sowie das Speichern von Werten in Register. Jede Speicherzelle, die von 0

beginnend adressiert werden, kann eine beliebige Zahl eines begrenzten Zahlenraumes enthalten. Wenn die nächste Instruktion vor der Ausführung steht, wird die Zahl der Speicherzelle, auf die der Instruktionszähler zeigt, als Instruktion verstanden und ausgeführt. Ein Lesen des Instruktionszählers ist nicht möglich, ebensowenig wie ein direktes Schreiben desselben, außer über den Umweg des Sprungs. Einige Instruktionen enthalten literale Daten und können aus mehreren Speicherzellen bestehen. Ein Beispiel dafür ist die Speicherzelle 2 in Bild 11. Dort steht ein Wert (250), der als “Push X”-Instruktion interpretiert wird. Diese Instruktion legt den numerischen Wert X auf dem Stapel ab. Diese Instruktion umfasst daher noch eine weitere Speicherzelle, in der der auf den Stapel zu legende Wert (in diesem Fall 30) steht. Nach Ausführung einer Instruktion wird der Instruktionszähler um die Anzahl der durch die letzte Instruktion benutzten Speicherzellen erhöht, sofern die Instruktion nicht selbst den Instruktionszähler verändert hat (was etwa bei Sprüngen der Fall ist). Danach wird die nächste Instruktion ausgeführt, und so weiter. Die Benutzung eines Stapels erlaubt die komfortable Nutzung von Unterprogrammen und damit auch von Programmbibliotheken.

Instruktionen des Wirtsmodells

Ein Agent kann seine Umgebung nicht selbst wahrnehmen, ebenso wenig, wie er unter Umgehung der Laufzeitumgebung mit Kommunikationspartnern interagieren kann. Er muss zu diesen Zwecken immer Funktionen des Wirts benutzen. Solch ein Aufruf einer Systemfunktion kann wie in Bild 12 aussehen:

```
callsystem <address of host memory cell>
```

Stack before execution: <parameter1> .. <parameter n>
Stack after execution: <result of procedure>

Bild 12: Aufruf einer Systemfunktion

Systemfunktionen können von Agenten dazu benutzt werden, mit externen Partnern zu kommunizieren oder Umgebungswerte zu bekommen. Bild 13 zeigt einige der möglichen Systemfunktionen.

`callsystem 100`
Stack before execution: v p
Stack after execution: --
Comment: send value `v` to partner `p`

`callsystem 200`
Stack before execution: p
Stack after execution: v
Comment: receive value `v` from partner `p`

`callsystem 300`
Stack before execution: --
Stack after execution: t
Comment: request current system time `t` in milliseconds

Bild 13: Beispiele für Systemfunktionen

Der Befehlssatz der Agentenmaschine enthält neben diesen Systemfunktionen die üblichen Kommandos für stapelbasierte Architekturen wie etwa mathematische Funktionen, Kontrollinstruktionen sowie Instruktionen zur Kontrolle des Stapels.

5.6.4 Angriffsmodell

Nachdem nun das Modell der Wirtsmaschine erläutert wurde, kann nun der angreifende Wirt modelliert werden. Wie das Wirtsmodell das Agentenprogramm ausführt, führt das Angriffsmodell das Angriffsprogramm aus. Gemäß unseren Anforderungen aus Kapitel 5.6.2 muss das Angriffsprogramm dabei in der Lage sein, die Ausführung des Agenten zu kontrollieren. Daher kann das Angriffsmodell auf das Wirtsmodell während der Ausführung des Agenten zugreifen. Da zudem die Systemfunktionen, die ein Agent ausführen kann, im Angriffsmodell angesiedelt

sind, kann der Angreifer auch diesen Aspekt kontrollieren.

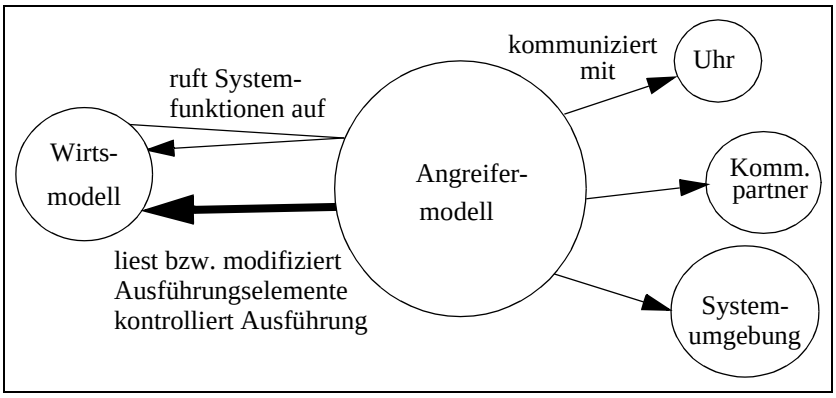


Bild 14: Architektur des Angriffsmodells

Instruktionen des Angriffsmodells

Zusätzlich zu den Instruktionen, die das Wirtsmodell kennt, besitzt das Angriffsmodell weitere Befehle, die den Zugriff auf die Elemente des Wirtsmodels erlauben (siehe Bild 15):

`fetchagent <address of agent memory cell>`
Stack before execution: --
Stack after execution: v
Comment: request value of memory cell <address> of agent

`storeagent <address of agent memory cell>`
Stack before execution: v
Stack after execution: --
Comment: store value v in memory cell <address> of agent

`fetchstackagent <address of agent stack cell>`
Stack before execution: --
Stack after execution: v
Comment: request value of stack cell <address> of agent

`storestackagent <address of agent stack cell>`
Stack before execution: v
Stack after execution: --
Comment: store value v in stack cell <address> of agent

`readstackpointer`
Stack before execution: --
Stack after execution: i
Comment: request value of the stack pointer of the agent

`writestackpointer`
Stack before execution: v
Stack after execution: --
Comment: store value v in the stack pointer of agent

Bild 15: Zugriffsinstruktionen auf das Wirtsmodell

Generelle Funktionsweise des Angriffsmodells

Zu Beginn der Ausführung des Agenten, also direkt nach der Migration, wird das Wirtsmodell mit dem Agentencode geladen. Das Angriffsmodell wird mit dem Angreiferprogramm geladen. Dieses Programm wurde von der angreifenden Partei

aus Parametern wie dem Typ des Agenten, dem Agentencode sowie den Angriffszielen (wie z.B. den Namen der Variablen, deren Werte gesucht sind) erzeugt.

Vor jeder Instruktion, die im Wirtsmodell ausgeführt wird, läuft folgender Algorithmus auf der Angriffsmaschine ab:

1. Dekodiere die nächste Instruktion der Agentenmaschine
2. Speichere auf dem Stapel der Angreifermaschine:
   - die nächste Instruktion der Agentenmaschine
   - die Parameter dieser Instruktion
3. Berechne den nächsten Wert des Instruktionszählers der Agentenmaschine und speichere diesen Wert auf dem Stapel der Angreifermaschine
4. Führe das Angriffsprogramm der Angreifermaschine aus
5. Führe als nächste Instruktion auf der Agentenmaschine diejenige aus, die samt Parametern auf dem Stapel der Angreifermaschine liegt
6. Setze den Instruktionszähler der Agentenmaschine auf den Wert, der auf dem Stapel der Angreifermaschine liegt
7. Beginne wieder bei 1.

Dieser Algorithmus kann als ein Teil des Angriffsprogrammes im Angriffsmodell implementiert werden, oder aber im Interpreter des Angriffsmodells (das dann damit etwas anders funktioniert als die Wirtsmaschine).

5.6.5 Ein Beispiel

Um das Modell des Wirts und des Angreifers zu illustrieren, wird das Beispiel aus Kapitel 5.6 benutzt. Das Ziel des zu untersuchenden Schutzmechanismus war es, zu verhindern, dass Variableninhalte im Klartext im Speicher stehen, weil sonst ein Angreifer solche Werte durch simples “Pattern Matching” erkennen könnte.

```
Fetch 10        # 10 = Variable A
Fetch 11        # 11 = Variable B
Fetch 12        # 12 = Variable C
Call 40         # 40 = buy
```

Bild 16: Originalcode

Zu diesem Zweck wird ein Transformationsschema für Variableninhalte benutzt, das den Inhalt einer Variablen in Teile aufsplittet und diese über neue Variablen verteilt (Bild 16 zeigt den originalen Code, Bild 17 eine beispielhafte Transfor-

```
A = (<230>XOR<194>)+<233>
B = (<193> Shift <231>) * <190>
C = (<191> - <234>) Shift <232>
40 = <500> + <501> - <150>
```

Bild 17: Transformation

mation, und Bild 18 den aus dem Schutzmechanismus resultierenden Code, der den originalen ersetzt).

Die Fragen, die nun mithilfe des Angriffsmodells beantwortet werden sollen, lauten: Kann der Angreifer den Variableninhalt nun nicht mehr lesen? Falls doch, wie kann der Angriff automatisiert werden und wie aufwändig ist der Angriff?

Wir nehmen an, dass der Angreifer den Typ des Agenten kennt und damit den Originalcode vor Anwendung des Schutzmechanismus. Er kennt aber die angewendete Transformation nicht (die erst bei Erzeugung des konkreten Agenten angewandt wird). Leider spielt aber nun die Unkenntnis der konkreten Transformation keine Rolle, da der Mechanismus ein schwerwiegendes Problem aufweist: selbst Variableninhalte, die nicht in der originalen Form im Speicher aufbewahrt werden, treten während der Verarbeitung als Klartext auf. Sobald der Funktionsaufruf vor der Ausführung steht, liegen die Parameter dieses Aufrufs in der originalen Form auf dem Stapel. Um einen Leseangriff auszuführen reicht es daher, auf die “Call”-Instruktion zu warten, und dann die Parameter vom Stapel zu lesen. Ein Angriffsprogramm, das dieses Vorgehen implementiert, zeigt Bild 19:

```
Fetch 230
Fetch 194
Xor
Fetch 233
Add
Fetch 190
Fetch 193
Fetch 231
Shift
Multiply
Fetch 232
Fetch 191
Fetch 234
Sub
Shift
Fetch 500
Fetch 501
Add
Fetch 150
Sub
Call
```

Bild 18: Resultierender Code

```
# attack main routine
08  dup                 # duplicates the top stack value
10  push “Call 40”      # push numeric value of statement on stack
13  equals?             # compare the two top stack values
14   if true goto 17
    else goto 20        # if stack value is true..
17  ...                 # the attack target value now is on the stack
18  ...                 # money can now be spent
20  return              # end of main routine
```

Bild 19: Angriffsprogramm

Der erste Befehl dupliziert den obersten Wert auf dem Stapel (der die nächste Instruktion der Agentenmaschine darstellt). Der zweite Befehl legt das numerische Äquivalent eines “Call 40”-Befehls auf den Stapel. Dann wird dieser Wert mit dem Duplikat des ersten Wert verglichen. Falls beide Werte gleich sind, kann in Zeile 17 auf den zu lesenden Wert (z.B. das elektronische Geld) auf dem Stapel zugegriffen werden. Schließlich wird das Angriffsprogramm in Zeile 20 beendet und die nächste Instruktion der Agentenmaschine vom Stapel gelesen und ausgeführt (Zeile 8 hatte uns diesen Wert ja kopiert).

Das Angriffsprogramm ist relativ effizient: es benötigt im Wesentlichen einen Vergleich pro ausgeführter Agenteninstruktion.

5.6.6 Diskussion des Modells

Das oben vorgestellte Modell erfüllt die Anforderungen aus Kapitel 5.6.2. Der Angreifer kann über Befehle den Datenteil des Agenten lesen und verändern (Anforderung 1). Da der Code ebenfalls im Datenteil des Agenten vorliegt, kann damit ebenso der Code gelesen und verändert werden (Anforderung 2). Durch den Zugriff auf den Instruktionszähler und den Stapel kann der Angreifer auch den aktuellen Ausführungszustand lesen und verändern (Anforderung 3). Anforderung 4, die Möglichkeit, die Ausführungsweise des Agenten ändern zu können, wird durch den Mechanismus erfüllt, der es dem Angriffsprogramm erlaubt, vor jeder Ausführung eines Befehls beliebigen Code auszuführen und damit Einfluss auf die Abarbeitung dieses Befehls zu nehmen. Anforderung 5 kann sogar auf zweifache Weise erfüllt werden. Einmal stellt der Wirt den aufgerufenen Code selbst bereit. Zum zweiten kann er auch über den Mechanismus zur Erfüllung von Anforderung 4 Einfluss nehmen. Da auch die Kommunikation eines Agenten mit dritten Parteien durch die Benutzung von Systemfunktionen zustande kommt, und der Wirt wie eben dargestellt diese Systemfunktionen kontrolliert, ist auch Anforderung 6 erfüllt. Anforderung 7 schließlich ist erfüllt, weil das Programm des Agenten im Speicherbereich des Agenten vorliegt, und weil der Agent selbst diesen Speicherbereich verändern kann.

In Kapitel 5.6.2 forderten wir von einem geeigneten Wirts- und Angriffsmodell, dass es detailliert genug sein soll, um Erkenntnisse liefern zu können, und abstrakt genug, um dem Wirt alle möglichen Angriffe zu erlauben. Darüberhinaus sollte es dem Wirt alle Techniken zum Angriff erlauben, die ihm in einem normalen Rechnersystem zur Verfügung stehen, und es sollte dem Agenten alle Schutzmechanismen erlauben, die in einem solchen Rechnersystem machbar sind. In diesem Sinne ist das Angriffsmodell “generisch”, d.h. auf alle möglichen Agentensysteme anwendbar. Der Nachteil dieser Allgemeinheit ist aber auch, dass es in einigen Agentensystemen einfacher ist, mobile Agenten anzugreifen, als es im Modell

scheint. Der Grund für diese Kluft liegt u.a. darin begründet, dass die Programmiersprachen wie Java, Tcl oder C++, die in heutigen Systemen Verwendung finden, dafür ausgelegt sind, die Programmerzeugung möglichst einfach und die Programmausführung möglichst schnell zu machen. Zu diesem Zweck versuchen diese Sprachen völlig zu Recht, möglichst viele Informationen über ein Programm bereitzustellen. Das Ziel des Schutzes von mobilen Agenten vor böswilligen Wirten dagegen verlangt, dass der Wirt möglichst wenig über den Agenten in Erfahrung bringen kann. Ein Beispiel für diesen Umstand ist das “static typing”. Dieser Begriff umschreibt den Umstand, dass es in bestimmten Sprachen möglich ist, den Typ jeder Variablen zur Übersetzungszeit festzulegen. Offensichtlich ist es jedoch einfacher, den Wert einer Variablen zu verstehen, wenn der Angreifer den Typ kennt, anstatt dass er ihn erst herausfinden muss. In diesem Sinne beschreibt das Angriffsmodell auch eine Umgebung für mobile Agenten, die besser geeignet ist, mobile Agenten zu schützen, als die existierenden Umgebungen.

Das in diesem Kapitel vorgestellte Wirts- und Angriffsmodell erlaubt nur ein Angriffsprogramm bzw. einen gleichzeitig ausgeführten Prozess. Einige Mobile-Agenten-Systeme erlauben die Verwendung mehrerer Prozesse pro Agent. Wenn dieser Aspekt für zu untersuchende Sicherheitsmechanismen wichtig ist, muss daher das Wirts- bzw. das Angriffsmodell erweitert werden. Da der Performanzaspekt für den Bereich der Sicherheit nicht im Vordergrund steht, kann diese Erweiterung einfach dadurch erreicht werden, dass die Anzahl der Wirtsmaschinen sowie die Anzahl der Angriffsmaschinen auf die gewünschte Anzahl vervielfacht wird. Neue Befehle erlauben die Kommunikation der Wirtsmaschinen sowie der Angriffsmaschinen untereinander und die Kontrolle einer beliebigen Wirtsmaschine durch eine beliebige Angriffsmaschine. Bis jetzt gibt es allerdings noch keine Sicherheitsmechanismen, die auf der Existenz von mehreren Agentenprozessen beruhen.

Teile diesen Kapitels wurden [Hoh98c] entnommen.

6 Verwandte Arbeiten

In diesem Kapitel sollen mögliche Lösungsansätze des Problems des Schutzes mobiler Agenten vor Angriffen durch böswillige Wirte kategorisiert und bestehende Lösungsansätze vorgestellt werden.

6.1 Ansätze außerhalb von Mobile-Agenten-Systemen

Wie in Kapitel 5 dargestellt, tritt das Problem der böswilligen Wirte bisher nur in Programmen auf, die Code enthalten, der für eine andere als diejenige Partei ausgeführt wird, die das Programm eigentlich für sich selbst ausführt. Das ist vor allem dann der Fall, wenn diese Codeteile einen Kopierschutz oder Ausführungsrestriktionen implementieren. Mit dem Aufkommen von digital vertriebenen Mediendokumenten wie Musikstücken oder Videofilmen gewinnt dieser Aspekt auch dann an Bedeutung, wenn nicht das Programm Gegenstand des kommerziellen Interesses ist, sondern die damit verarbeiteten Daten. So gibt es z.B. Programme zum Abspielen von DVDs (Digital Versatile Disc), die einen zur Dekodierung von verschlüsselten Inhalten notwendigen Schlüssel im Programm enthalten.

Die bisherigen Versuche zur Verhinderung der Aufhebung oder Entfernung dieser Programmteile (bekannt als “Cracking”) waren bisher nicht besonders wirksam. Bis jetzt war es immer möglich, die Schutzmechanismen zu brechen, indem ein menschlicher Programmierer diese Teile analysiert und deaktiviert hat (“Re-engineering”), ohne dass dazu Quellcode notwendig war. Da offensichtlich keine grundsätzlichen Mechanismen gefunden worden sind, dieses Vorgehen zu verhindern, sondern sich das Vorgehen darauf beschränkt hat, Angriffe dadurch möglichst schwierig zu machen, dass die Schutzmechanismen (mit wechselndem Erfolg) geheim gehalten werden (“security by obscurity”), gibt es naturgemäß auch nur wenig Literatur zu diesem Thema. Eine Ausnahme bildet [Auc96], der Techniken beschreibt, die gegen einen menschlichen Angreifer gerichtet sind, und die Verwürfelung (“obfuscation”) und teilweise Ver- bzw. Entschlüsselung von Code beschreibt. Dieser Ansatz bietet allerdings nicht nur keinen theoretisch abgesicherten Schutz, sondern kann auch keine experimentelle Überprüfung der erzielten Schutzleistung vorweisen.

6.2 Kategorisierung der Lösungsansätze

Die Lösungsansätze zum Schutz mobiler Agenten vor Angriffen böswilliger Wirte lassen sich nach folgendem Schema grob klassifizieren (siehe Bild 20).

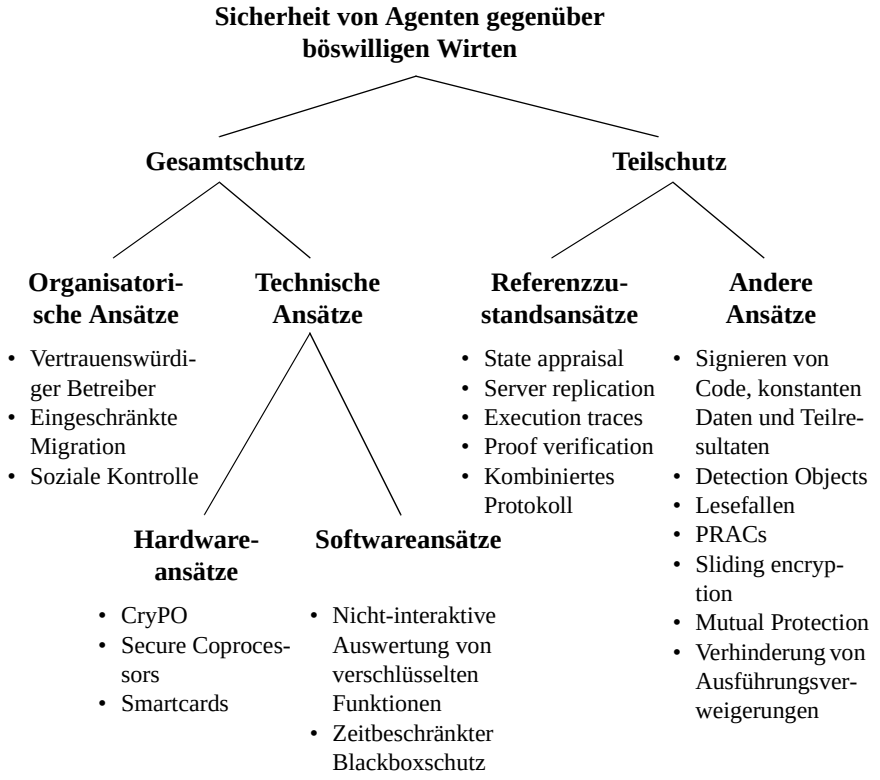


Bild 20: Kategorien der Lösungsansätze zum Schutz von Agenten

Je nachdem, ob im Wesentlichen alle oder nur eine Teilmenge der Angriffe abgewehrt werden sollen, kann man die Ansätze zunächst in Gesamtschutz- bzw. Teilschutzansätze gliedern.

6.3 Gesamtschutzansätze

Gesamtschutzansätze haben das Ziel, fast alle Angriffe zu verhindern. Diese Ansätze lassen sich nach der Frage unterteilen, ob die Grundlage des Schutzes organisatorischer oder technischer Art ist.

6.3.1 Organisatorische Ansätze

Organisatorische Ansätze erreichen den Schutz des Agenten dadurch, dass sie die Migration eines Agenten auf solche Wirte beschränken, die als vertrauenswürdig angenommen werden, bei denen also ein Angriff gegen einen Agenten nicht im Interesse der Wirt-Partei liegt. Das bedeutet nicht, dass Agenten nicht auch auf vertrauenswürdigen Wirten angegriffen werden können oder tatsächlich angegriffen werden, aber dann liegt es im Interesse der Wirt-Partei, den Schaden gegenüber dem Agenteneigentümer wieder gutzumachen. In der Filiale einer Ladenkette kann es wie in jedem anderen Laden dazu kommen, dass der Käufer zuwenig Wechselgeld herausbekommt, aber wenn die Kette dadurch einen Schaden an ihrem Ruf befürchtet, wird sie selbst darauf achten, dass solche Vorgänge nicht vorkommen, bzw. den Käufer für den entstandenen Schaden entschädigen.

Vertrauenswürdiger Betreiber

Angriffe sind vor allem dann zu erwarten, wenn der Eigentümer eines Agenten die Vertrauenswürdigkeit der Betreiberpartei der Wirte nicht einschätzen kann. Wenn aber umgekehrt das gesamte Agentensystem, d.h. alle Wirte von einer Partei betrieben werden, die zudem eine bekannte Institution darstellt, kann der Eigentümer davon ausgehen, dass es nicht im Interesse der Wirt-Partei liegt, einen Agenten anzugreifen. Das bedeutet nicht, dass sich die Interessen beider Parteien vollkommen decken müssen (schließlich ist z.B. die Frage des Datenschutzes auch bei nicht-elektronischem Handel ein Konfliktbereich zwischen Kunden und Firmen), aber die Sicherheit, die ein Kunde bei einem bekannten Betreiber erwarten kann, ist die gleiche wie bei den nicht-elektronischen Geschäftsbeziehungen. Obwohl es keine expliziten Aussagen darüber gibt, kann angenommen werden, dass dieses Sicherheitsmodell die Grundlage für das “PersonaLink”-System [Mob96] war, das von AT&T betrieben wurde und auf der Telescript-Technologie [GM96] der Firma General Magic beruhte (PersonaLink war kein kommerzieller Erfolg und wurde 1997 vom Markt genommen).

Eingeschränkte Migration

Einige Ansätze (z.B. [FGS96]) versuchen das Problem dadurch zu lösen, dass sie die Existenz von vertrauenswürdigen Wirten postulieren und es mobilen Agenten nicht gestatten, auf andere als vertrauenswürdige Wirte zu migrieren. Falls die Migration erfolgt, um Ressourcen lokal in Anspruch zu nehmen und diese auch auf vertrauenswürdigen Wirten existieren, lässt sich dieser Ansatz auf einen Auswahlmechanismus der Zielwirte reduzieren. Falls notwendige Ressourcen nur auf nicht vertrauenswürdigen Wirten vorliegen, muss entfernte Kommunikation zu diesen Wirten benutzt werden (diesen Ansatz gibt es auch in einer Variante, die dem Schutz des Wirts dient, und bei dem ein Wirt nur solche Agenten annimmt, die vor-

her auf keinem nicht vertrauenswürdigen Wirt gewesen sind). Eine Variante dieses Ansatzes benutzt einfache mobile Agenten statt entfernter Kommunikation mit dem Hintergrund, dass „One-Hop“-Agenten, also solche die sich nur von einem Heimatwirt zu einem anderen Wirt und dann wieder zurück bewegen, oft so einfach gestaltet sind, dass der fremde Wirt kein Interesse hat, diesen Agenten anzugreifen (weil er keine sensitiven Daten mit sich führt, und die Ausführung oft im Interesse des besuchten Wirts liegt).

Soziale Kontrolle

Zwei der Probleme des letzten Ansatzes sind, dass Agenteneigentümer apriori Wirtsbetreibern vertrauen müssen, und dass Vertrauen eine 1-zu-1-Beziehung zwischen zwei Parteien ist. Ein anderer Ansatz [RJ96] ersetzt daher das Konzept des Vertrauens durch das Konzept des Rufs (“reputation”), das als eine allgemeingültige Variante von Vertrauen angesehen werden kann. Agenten, die gute Erfahrungen mit einem Wirt gemacht haben, können dessen guten Ruf mehren, während Agenten mit schlechten Erfahrungen zur Minderung des Rufs beitragen können. Wirte, die betrügen, können so mit einem “sozialen Bann” belegt werden, der bewirkt, dass kein Agent mehr auf diesen Wirt migriert.

Bewertung

Der Nachteil des Vertrauenswürdiger-Betreiber-Ansatzes besteht in genau der Eigenschaft, die ihn auch sicher macht: Wenn man nur eine Betreiberpartei zulassen kann ist das entstehende System auf der Ebene der Wirte nicht offen. Damit kann ein solches System nicht völlig inkrementell entstehen - es benötigt eine kritische Startmasse, damit sich Benutzer finden. Diese Startmasse besteht aber aus einem Dienste- oder Informationsangebot, das groß genug sein muss. Falls die Betreiberpartei diese Masse nicht selbst bereitstellen kann, benötigt sie andere Parteien, die diese Daten bereitstellen. Diese lassen sich aber nur finden, wenn die Menge der Benutzer groß genug ist, womit man sich wieder am Anfang der Problemstellung befindet. Weiter stellt sich die Frage nach der Motivation der Betreiberpartei, eine solche Infrastruktur aufzubauen. Die Antwort wird in den meisten Fällen in einem kommerziellen Interesse dieser Partei bestehen, was wiederum bedeutet, dass andere Parteien Geld einsetzen müssen, um an diesem System teilzunehmen, womit wieder das erste Problem eine Rolle spielt.

Der Vorteil des zweiten Ansatzes ist, dass er immer möglich ist (weshalb er häufig wieder als ad-hoc-Idee in die Diskussion geworfen wird). Falls es nur einen vertrauenswürdigen Wirt gibt (z.B. den Heimatwirt), dann fällt dieser Ansatz mit der stationären Client-Server-Alternative (siehe Kapitel 5.5) zusammen. In jedem Fall gilt, dass diese Änderung des generellen Verarbeitungsmodells kompatibel zu den

Anforderungen der Anwendung sein muss, d.h. dass durch Verwendung dieses Ansatzes keine Eigenschaft eines Mobile-Agenten-Systems wegfallen darf, die von der Anwendung gefordert wird (so könnte man sich z.B. vorstellen, dass eine Datenbank auf einem unbekannten Wirt fordert, dass ihre Daten nur lokal abgefragt werden dürfen, weil sie die Kontrolle über die Rohdaten behalten will). Immerhin würde es eine Systemkonfiguration, bei der sich ein vertrauenswürdiger Wirt in der "Nähe" eines beliebigen Wirts befindet (z.B. im selben LAN), einem Agenten erlauben, eine "fast-lokale" Kommunikation mit einem Wirt aufzubauen. Die Nachteile des zweiten Ansatzes sind, dass Agenteneigentümer a-priori Wirtsbetreibern vertrauen müssen, und dass Vertrauen eine 1-zu-1-Beziehung zwischen zwei Parteien ist, sowie die Frage, ob die Existenz eines ganzen Netzes von vertrauenswürdigen Wirten realistisch ist. Das Problem ist, dass "Vertrauen" in eine Partei erstens zeitabhängig und zweitens aufgabenbezogen sein kann. Im Allgemeinen wird man z.B. dem Wirt einer bekannten Fluglinie vertrauen, aber nicht mehr unbedingt dann, wenn der Agent eine vergleichende Preisliste anderer Fluglinien benutzt. Aber selbst wenn die Frage des Vertrauens beantwortet werden kann, bleibt die Frage, warum vertrauenswürdige Wirte ihre Dienste (und damit auch Rechenzeit und Speicherverbrauch) anbieten sollen, vor allem für Anwendungen, die sich nicht unbedingt auf ihre Ressourcen beziehen. Diese Frage ist nach wie vor ungeklärt, oder wird mit der Forderung nach einem kommerziellen Agentensystem beantwortet, das aber dann auch die bereits dargestellten Probleme (wie z.B. die kritische Startmasse) hat. Zuwenig vertrauenswürdige Wirte bedeuten auch viele Migrationsrestriktionen, und damit unter Umständen eine Situation, bei der die Verwendung mobiler Agenten nicht mehr viele Vorteile bietet.

Der dritte Ansatz schließlich hat den Vorteil, dass er ein allgemeingültiges "Vertrauensmaß" benutzt. Seine Nachteile liegen in dem Umstand, dass ein "bösartiges" Verhalten eines Wirts erst (manuell) festgestellt werden muss, darin, dass der Regulationsvorgang unter Umständen zu langsam für den Gebrauch in Gemeinschaften von Programmen ist (es ist innerhalb weniger Sekunden möglich, einen neuen Dienst anzukündigen, seine Kunden "auszurauben" und seinen Wirt wieder zu schließen). Zudem wird ein neuer Angriff, "Rufmord" ermöglicht, bei dem eine Partei eine andere dadurch ausschließt, dass sie sich durch ein oder mehrere Agenten über schlechtes Verhalten beschwert. Solange solches Verhalten nicht automatisch überprüft werden kann, sind dann Widersprüche nur durch Eingriff eines Menschen, d.h. sehr langsam, zu lösen.

6.3.2 Technische Ansätze

Technische Gesamtschutzansätze zielen darauf ab, einen vollständigen Schutz des Agenten zu erreichen, und zwar idealerweise so, dass die Schutzmechanismen

transparent für den Programmierer sind, er sich also um den Sicherheitsaspekt bei der Programmierung nicht (oder kaum) kümmern muss. Diese technischen Ansätze lassen sich in zwei Gruppen teilen. Hardware-Ansätze sehen spezielle Hardware im Wirt vor, während Software-Ansätze keine besonderen Anforderungen an die Architektur des Wirt-Systems stellen.

Hardwareansätze

Ein grundlegender Ansatz besteht darin, zumindest Teile des Agenten von einer vertrauenswürdigen, autonomen Hardware ausführen zu lassen, die von außen, d.h. vom Wirt-Betreiber nicht manipuliert werden kann. Dieser Ansatz erklärt sich aus zwei Sicherheitsproblemen mobiler Agenten. Erstens sind Agenten nur passive Entitäten, die vom Wirt nach Belieben ausgeführt werden können. Zweitens kann die Wirt-Partei alles am ausführenden Wirt-System bis hinunter zur Hardware zum Zwecke des Angriffs modifizieren. Vertrauen in den Wirt wird bei diesen Ansätzen durch Vertrauen in den Hersteller der Hardware ersetzt. Dies stellt zwar keinen prinzipiellen Unterschied dar, aber erlaubt es die Frage nach dem Vertrauen in viele, möglicherweise unbekannte Wirtsbetreiber durch das Vertrauen in wenige, bekannte Hardwarehersteller zu ersetzen.

CryPO

In [Wil99] und [WSB99] wurden eine Hardware-Architektur und ein Protokoll vorgestellt, die es erlauben, einen mobilen Agenten sicher innerhalb eines autonomen, vertrauenswürdigen Moduls auszuführen. Das Modul besteht aus einer CPU, RAM, ROM und nicht-flüchtigem Speicher. Seine Hauptaufgabe ist es, eine sichere Ausführungsumgebung für mobile Agenten bereitzustellen. Das Modul besitzt einen eigenen privaten Schlüssel, der vom Modul selbst generiert wird, und keiner anderen Partei, auch nicht dem Wirtsbetreiber oder dem Modul-Hersteller bekannt ist. Das CryPO-Protokoll erlaubt es, einen mobilen Agenten vom Wirt in das Modul zu laden und dort ausführen zu lassen.

Secure Coprocessors

In [Yee99] wird ein Ansatz kurz anskizziert, der es wie der CryPO-Ansatz erlaubt, mobile Agenten innerhalb eines “versiegelten” autonomen Hardware-Moduls auszuführen.

Smartcards

Statt Hardware einzusetzen, in denen der gesamte mobile Agent sicher ausgeführt wird, benutzt [Fün99] Smartcards, also modifikationssichere Schaltkreise, die relativ geringe Rechen- und Speicherkapazitäten haben, aber dafür wesentlich billiger sind. Da sich aus diesen Gründen die komplette Agentenausführung auf der Smartcard verbietet, werden mobile Agenten durch den Programmierer in sicherheitssen-

sitive und -unkritische Abschnitte zerlegt. Die sicherheitskritischen Abschnitte werden auf der Smartcard ausgeführt, die anderen auf dem normalen Wirt.

Bewertung

Der große Vorteil der ersten beiden Ansätze ist, dass hier der Schutz des Agenten gegeben ist, schließlich wird dem mobilen Agenten ja auch ein eigener, sicherer Rechner spendiert. Die Frage, ob ein solches Modul wirklich sicher gegen physikalische Angriffe von außen gemacht werden kann, ist nicht in aller Absolutheit zu beantworten, allerdings kann man sich gut vorstellen, dass solche Module durch den Einsatz aufwendigerer Verfahren so sicher gemacht werden können, dass ein Angriff zumindest sehr kostenintensiv ist. Die Nachteile aller drei Verfahren bestehen zum einen darin, dass sie den Wirt-Betreiber nicht nur Geld kosten, sondern auch schwer in ein existierendes Rechnersystem zu integrieren sind. Das verschärft die Frage nach der Motivation von Wirt-Betreibern, die durch die höhere Installationsschwelle noch mehr die Kosten gegen einen zu erwartenden Bedarf bzw. den Nutzen abwägen müssen. Der zweite Nachteil ist, dass solche Module einen Flaschenhals im Gesamtsystem darstellen können, und dass dieser Flaschenhals nicht so einfach durch den Einsatz von Standardkomponenten zu beseitigen ist bzw. das Gesamtsystem dann nicht mehr mit den normalen Mitteln skalierbar ist. Die zusätzlichen Nachteile des Einsatzes von Smartcards sind die bisher unbeantworteten Fragen, ob sich mobile Agenten überhaupt sinnvoll in sicherheitssensitive und -unkritische Teile aufspalten lassen sowie welche Angriffe durch die Kommunikation zwischen beiden Teilen (die den Angriffen durch den Wirt ebenfalls ausgesetzt ist) möglich sind.

Softwareansätze

Die Ansätze, die von spezieller Hardware absehen und einen Schutz durch eine reine Softwarelösung erreichen wollen, lassen sich zurzeit alle unter dem Begriff “Blackbox”-Schutz kategorisieren. Der mobile Agent wird, aus einer Spezifikation, z.B. in Form von bestehendem Code, in eine andere Form umgewandelt, die immer noch ausführbar bleibt, die aber wesentliche Aspekte vor dem ausführenden Wirt verbirgt. Damit ist dieser mobile Agent für den Wirt eine “Blackbox” geworden, von der er nur noch die Eingabe und eventuelle Ausgaben sehen kann (wobei diese nicht unbedingt im Klartext vorliegen müssen), sowie Zustandsänderungen, die der Wirt zwar unterscheiden, aber nicht in ihrer Bedeutung zuordnen kann. Die einzige Option, die dem Wirt verbleibt ist den Agenten ganz, teilweise oder gar nicht auszuführen. Die Ansätze im Bereich Blackbox-Schutz werden in Kapitel 8 gesondert behandelt.

6.4 Teilschutzansätze

Neben den Verfahren, die mobile Agenten im Wesentlichen vor allen Angriffen schützen wollen, gibt es weitere Ansätze, die Schutz vor einzelnen Angriffsklassen oder Angriffen bieten. Zurzeit bietet dieses Teilgebiet kein einheitliches Bild, immerhin lässt sich eine Klasse von Verfahren herauskategorisieren, die zur Aufdeckung von Angriffen "Referenzzustände" benutzen.

6.4.1 Ansätze, die "Referenzzustände" benutzen

Einige Angriffe wirken sich auf den Zustand eines Agenten aus, der mit der Migration auf den nächsten Wirt transportiert wird. Diese Zustände sind damit ein beobachtbares "Ergebnis" der Ausführung eines Agenten auf einem Wirt. Wenn man es nun schafft, eine "Referenzausführung" zu berechnen, also dieselbe Ausführung eines Agenten auf einem "Referenzwirt", d.h. einem Wirt, der garantiert keinen Angriff startet, kann man den so entstehenden "Referenzzustand" mit dem Ergebniszustand auf einem Wirt vergleichen und so einige Angriffe feststellen. Ansätze, die solche Referenzzustände benutzen, werden in Kapitel 7.4 vorgestellt.

6.4.2 Andere Ansätze

Neben den Ansätzen, die Referenzzustände benutzen, gibt es eine ganze Reihe weiterer Verfahren, die aber noch nicht so zahlreich sind, dass sie sich adäquat kategorisieren lassen. Einige dieser Ansätze sollen nun kurz skizziert werden.

Signieren von Code

Da auch bei anderen Mobile-Code-Systemen das Problem besteht, dass Code von unbekannten Knoten geladen wird, existieren seit einiger Zeit Mechanismen, die sicherstellen, dass dieser Code nicht unbefugt manipuliert wird. Dieselben Mechanismen kann man auch in Mobile-Agenten-Systemen einsetzen, wenn der Code eines mobilen Agenten über seine Lebensdauer konstant ist (falls sich der Code wie die Daten dynamisch aus der Ausführung des Agenten ergibt, kann dieses Verfahren nur auf die konstanten Teile des Codes angewandt werden). Dazu unterschreibt der Autor des Agenten den Code digital, wobei ein beliebiger Signaturalgorithmus eingesetzt werden kann (wie z.B. DSS [NIS94]). Ein Wirt, der einen Agenten empfängt, prüft daher vor der Ausführung diese Signatur, und stellt so für sich sicher, dass kein anderer Wirt diesen Code modifiziert hat.

Signieren der konstanten Teile des Agenten

Auf derselben Idee wie das Signieren von Code beruht auch die Idee des Signierens der konstanten Teile des Agenten durch den Eigentümer des Agenten. Wie in Kapitel 8.7 ausgeführt werden wird, umfassen diese Teile den Typ (bzw. die Klassen) des

Agenten, seine Id, eine Referenz auf den Eigentümer sowie andere konstante Daten. Durch die Signatur wird sichergestellt, dass kein Wirt diese Daten unbemerkt modifiziert weitergeben kann.

'Detection Objects'

Um Modifikationsangriffe aufzudecken, schlägt [Mea97] vor, einzelne Daten, die unter den anderen versteckt werden und die von einer korrekten Ausführung nicht verändert werden, als Modifikationsindikatoren einzusetzen. Wenn beispielsweise eine Liste von Preisen erfragt wird, kann der Eigentümer dem Agenten bereits beim Start solche Indikationsdaten mitgeben. Nachdem der Agent zurückgekehrt ist, kann der Eigentümer dann prüfen, ob diese Indikatoren noch unverändert vorhanden sind. Der Nachteil dieser Methode ist, dass es nicht sicher ist, dass ein Modifikationsangriff wirklich die Indikatoren ändert. Probleme, die vor einer Benutzung dieser Technik gelöst werden müssen, betreffen die Frage, wer und wie solche Indikationsdaten plausibel erzeugen kann, und wie man verhindern kann, dass der Angreifer herausfindet, welches diese Daten sind.

Lesefallen

In der gleichen Weise, in der Detection Objects Modifikationsangriffe aufzeigen können, kann man sich auch andere “Fallen” vorstellen, Daten also, die nur dazu dienen, bei Gebrauch den Benutzer als Angreifer eines mobilen Agenten aufzudekken. Dies ist dann möglich, wenn es eine dritte Partei gibt, bei der man solche Datenelemente “einlösen” muss, und die in der Lage ist, Fallen von echten Datenobjekten zu unterscheiden. Ein Beispiel dafür könnten bestimmte Arten von elektronischem Geld sein. Wenn ein Agent verschiedene Stücke elektronischen Geldes mit sich führt, könnten einige davon Fallen sein. Versucht ein Angreifer, diese bei der Bank einzulösen, löst er Alarm aus und kann als Angreifer erkannt werden. Damit ein Angreifer solche Fallen nicht erkennt, muss es möglich sein, Fallen von echten Daten ununterscheidbar zu machen. Wenn ein Agent z.B. nur bestimmte Geldstücke ausgibt, könnte ein Angreifer versuchen, den Agenten soweit auszuführen, dass er erkennt, welche diese Geldstücke sind. Damit kann diese Technik vermutlich nur dann Verwendung finden, wenn es sich um Daten handelt, die ein Agent zwar mit sich führt, aber nicht verwendet, zumindest aber nicht mit einer Ausführung, die eine deutbare Ausgabe erzeugt, als solche offenlegt.

PRACs

In [Yee99] wird eine Technik vorgestellt, die “Partial Result Authentication Codes” (oder PRAC) genannt wird, und die es erlaubt, Daten, die ein Agent auf einem Wirt berechnet hat, vor Modifikationen auf nachfolgenden Wirten zu schützen. Dazu berechnet der Eigentümer vor dem Start des Agenten eine Liste mit geheimen Ein-

malschlüsseln. Auf jedem Wirt verwendet der Agent einen dieser Schlüssel, um die Integrität des aktuellen Resultats zu sichern und löscht diesen Schlüssel vor der Migration auf den nächsten Wirt. Da nur der Eigentümer eine Kopie der Liste hat, kann dieser nach der Rückkehr des Agenten die Einzelresultate verifizieren.

Signieren von Teilresultaten

In [KAG98] wird die Technik der PRACs dahingehend weiterentwickelt, dass eine Kette von verschlüsselten Hash-Werten verwendet wird, die die bisherigen Resultate, das neue Resultat sowie das nächste Migrationsziel umfasst. Außerdem werden die Resultate mit dem öffentlichen Schlüssel des Agenteneigentümers verschlüsselt, so dass die Resultate auf den nachfolgenden Wirten auch vor Leseangriffen geschützt sind.

„Sliding Encryption“

Einen weiteren Ansatz, um die Daten, die ein Agent im Laufe seiner Ausführungen sammelt, zu verschlüsseln, beschreiben [YY97]. Die Besonderheit dieses Ansatzes stellt die Tatsache dar, dass kleine inkrementell hinzugefügte Datenmengen in kleinen Schlüsseltextmengen resultieren, während bei traditionellen Verfahren die kleinsten Schlüsseltextmengen von der Größe des verwendeten Schlüssels sind.

„Mutual Protection“

Ein weiterer Ansatz (siehe [Rot99]) resultiert aus der Idee, dass man wichtige Entscheidungen nicht nur durch einen Agenten auf einem Wirt trifft und sensitive Daten nicht nur in einem Agenten, sondern in zwei gespeichert werden. Kommt es jetzt durch die Ausführung eines Agenten zu einer Situation, dass eine solche Entscheidung getroffen oder ein sensitives Datum (z.B. Geld) benötigt wird, dann erfolgt eine Kommunikation des Agenten mit dem anderen Agenten auf einem anderen Wirt, wobei der zweite Agent Plausibilitätschecks macht und wichtige Ereignisse aufzeichnet. Dazu wird angenommen, dass es unwahrscheinlich ist, dass zwei Wirte zusammenarbeiten, wenn einer davon zufällig ausgewählt wird.

Verhindern von Ausführungsverweigerungen

Um zu verhindern, dass ein Agent verschwindet, weil ein Wirt einen Agenten nicht (mehr) ausführen kann (im Falle eines technischen Problems) oder will (im Falle eines Angriffs oder aus sonstigen Gründen), können mehrere Mechanismen eingesetzt werden. Mechanismen, die den Aspekt des technischen Problems angehen (z.B. [RS98]), beispielsweise den Ausfall des Wirt-Rechners, nehmen meist an, dass der Wirt, falls er wieder einsatzbereit ist, gutwillig ist und zu kooperieren versucht. Daher können diese Mechanismen nicht ohne weiteres zur Verhinderung eines entsprechenden Angriffs eingesetzt werden. Allerdings könnte man einige

dieser Mechanismen so modifizieren, dass sie sich, um einen Ausfall festzustellen, nicht auf die Angaben des Wirts verlassen. Ein solches modifiziertes Verfahren könnte z.B. ein Zeitintervall definieren, innerhalb dessen sich der Agent vom nächsten Wirt aus melden muss, so dass ein Angriff als gegeben angenommen wird, falls dies nicht geschieht. Die Länge dieses Intervalls kann großzügig bemessen sein, da es in diesem Fall nicht auf Effizienz, sondern nur auf Effektivität ankommt.

Im Bereich der Verfahren, die sowohl gegen technische Probleme als auch gezielte Angriffe helfen (also auch byzantinische Fehler erlauben), findet sich vor allem der "Server replication"-Ansatz [MRS96]. Dieser Ansatz geht davon aus, dass für jede Ausführungssitzung n mögliche verschiedene Wirte existieren, und dass der Agent auf allen Wirten repliziert ausgeführt wird. Daher können mit diesem Ansatz $n/2$ Wirte, die eine Ausführung verweigern, oder sogar manipulieren, toleriert werden. Eine genauere Untersuchung dieses Ansatzes findet sich in Kapitel 7.4. Um nur eine Ausführungsverweigerung zu verhindern, könnte man diesen Ansatz so modifizieren, dass für n zu tolerierende Wirte $n+1$ Wirte pro Ausführungssitzung eingesetzt werden.

Um eine böswillige Ausführungsverweigerung zu verhindern, setzt [Ros99] in einem Protokoll namens MASP (Mobile Agent Security Protocol) ein Verfahren ein, das dort "Progress Management" genannt wird. In diesem Verfahren schickt ein Wirt nach Beendigung einer Ausführungssitzung eine unterschriebene Nachricht an einen früheren Wirt ab (wobei dieser eine Entfernung von $m+1$ Migrationen hat). Sobald dieser die Nachricht erhält, kann er den von ihm an den nächsten Wirt geschickten Agenten vergessen. Ist nun ein Wirt bösartig, setzt also eine Ausführungsverweigerung ein, dann kann er zwar noch eine Nachricht an einen früheren Wirt schicken, aber es erfolgt keine Nachricht vom folgenden Wirt an dessen früheren Wirt. Da Wirte auf diese Nachrichten mithilfe eines Zeitgebers (engl. timer) warten, können diese nach Ablauf des Zeitgebers den Agenten an eine vertrauenswürdige Stelle, den sogenannten "Trusted Monitor", schicken. Dieser sendet dann den Agenten an den Wirt, der eigentlich auf den böswilligen Wirt folgen sollte. Der Fall, dass durch dieses Verfahren mehrere Instanzen desselben Agenten entstehen können, wird nicht behandelt.

Bewertung

Eine zusammenfassende Bewertung der Ansätze in diesem Unterkapitel fällt schwer, da die vorgestellten Verfahren verschiedene Angriffe abwehren und unterschiedliche Techniken einsetzen. Die Signierungsverfahren schließen sehr wichtige Angriffe aus und sind mit einfachen Mitteln zu erreichen; sie bilden daher die Grundlage einer ganzen Reihe von Systemen. Die Methoden, um Teilresultate vor Modifikationen oder Leseangriffen auf nachfolgenden Wirten zu schützen, erfüllen

ihren Zweck, decken aber nur einen kleinen Teil der möglichen Angriffe ab und sind nur für einen geringen Teil der möglichen Anwendungen notwendig. Detection Objects und Lesefallen sind nur erste Ansatzideen und bedürfen der weiteren Ausarbeitung, bevor an eine Bewertung geschweige denn einen Einsatz gedacht werden kann. Mutual Protection schließlich scheint erhebliche Kommunikationskosten zwischen den beiden Agenten zu verursachen und muss erst noch in den Kosten, vor allem gegen die Client-Server-Alternative abgeschätzt werden. Die Verfahren zur Verhinderung der Ausführungsverweigerung stellen zurzeit nur Teilaspekte von Mechanismen dar, die eine andere Hauptzielsetzung haben, und daher nicht auf diesen Fall hin optimiert sind. Allen diesen Ansätzen ist aber gemein, dass nur einzelne Angriffe verhindert bzw. entdeckt werden, und es nur wenig Anwendungen gibt, die allein durch einen solchen Ansatz geschützt werden können.

7 Ansätze auf der Basis von Referenzzuständen

In diesem Kapitel werden wir uns mit Ansätzen beschäftigen, die den Endzustand eines mobilen Agenten nach Ausführung auf einem Wirt mit dem Zustand vergleichen, den der gleiche Agent nach Ausführung auf einem vertrauenswürdigen Wirt besitzt. Wir werden die Idee solcher Referenzzustände aus einer allgemeinen Betrachtung der Frage ableiten, wie Angriffe definiert werden können. Um zu untersuchen was ein Angriff ist, benötigen wir ein Ausführungsmodell eines mobilen Agenten, das auf den von außen beobachtbaren Eigenschaften der Ausführung beruht. Teile diesen Kapitels finden sich in [Hoh00].

7.1 Ausführungsmodell

Im Gegensatz zum Wirtsmodell aus Kapitel 5.6 modelliert dieses Modell die Ausführung eines Agenten über mehrere Wirte.

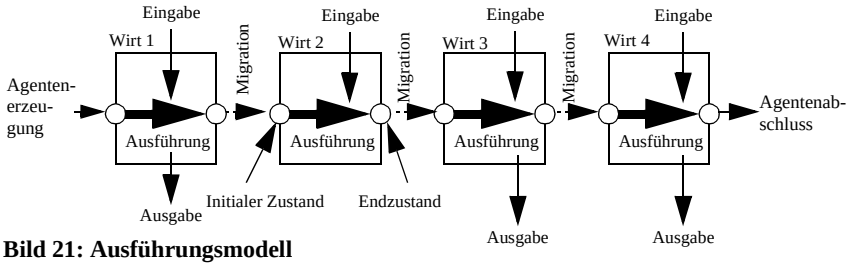


Bild 21: Ausführungsmodell

Um seine Aufgabe zu erfüllen, migriert der Agent entlang einer Sequenz von Wirten (vergl. Bild 21). Der Wirt, auf dem der Agent ankommt, nimmt den Anfangszustand des Agenten und beginnt eine Ausführungssitzung. In dieser Sitzung führt der Wirt den Agenten mithilfe des Codes und anhand der Eingabedaten aus und berechnet so einen Endzustand, wobei eine Sequenz von Ausgaben erzeugt wird. Die Eingabe umfasst alle Daten, die “von außen” an den Agenten geschickt werden, d.h. sie beinhaltet sowohl die Kommunikation mit Partnern auf anderen Wirten als auch diejenige, die mit dem Ausführungswirt selbst stattfindet oder von diesem vermittelt wird. Insbesondere enthält sie auch die Resultate des Aufrufs von Systemfunktionen, wie z.B. Zufallszahlen oder die aktuelle Systemzeit. Alle Eingaben stammen damit vom Wirt selbst, oder werden über diesen transportiert. Entsprechendes gilt für die Ausgaben. Sobald der Agent auf einen anderen Wirt migriert oder beendet wird, ist die Ausführungssitzung beendet, und der Endzustand des Agenten auf dem Startwirt wird zum Anfangszustand des Agenten auf dem Zielwirt. Im Folgenden wird angenommen, dass die konstanten Teile eines mobilen Agenten (z.B. der

Code) durch den Eigentümer signiert sind, und dass deren Integrität jederzeit von allen Parteien prüfbar ist.

7.2 Angriffe als Verhaltensunterschiede

Anhand der Beispiele aus Kapitel 5.2 bzw. Kapitel 5.3 ist intuitiv klar, was mit dem Begriff "Angriff gegen einen mobilen Agenten durch einen böswilligen Wirt" gemeint ist, aber bisher wurde nicht versucht, ihn explizit zu definieren.

Da mit dem Begriff "Angriff" eine Verletzung der Erwartungen des Programmierers oder Eigentümers eines Agenten verbunden werden, kann man diesen Begriff informal als „*Verhaltensunterschied*" zwischen einem angreifenden Wirt und einem nicht-angreifenden Wirt, dem sog. *Referenz-Wirt* definieren.

Bezogen auf Modifikationsangriffe ist damit klar, dass ein Wirt sich wohl verhält, wenn er die Daten eines Agenten nur in der Weise behandelt, die von einem nicht-angreifenden Wirt vorgegeben ist. Was das gleiche Verhalten im Bezug auf Leseangriffe bedeutet, muss erläutert werden. Dass ein Wirt sensitive Daten eines Agenten kennt, ist zunächst nicht von Übel. Erst wenn diese Kenntnis benutzt wird, um gegen den Agenten oder dessen Eigentümer vorzugehen, entstehen Probleme. Ein Vorgehen jenseits der bloßem Kenntnis eines Datums erfordert aber wieder Aktionen, d.h. ein bestimmtes Verhalten, das sich dann wieder vom Verhalten eines Referenzwirts unterscheidet. Das Problem ist nun, das sich dieses böswillige Verhalten zeigen kann, nachdem der Agent weitermigriert ist, während ein sich auf Modifikationsangriffe bezogenes Verhalten vor der Migration äußert. Dieses Problem löst sich aber schnell auf, wen man sich überlegt, dass die Voraussetzung für die Ausnutzung der Kenntnis eines Datums wieder die Kenntnis ist, und dass es eine Aktion eines Wirts darstellt, diese Kenntnis dem Wissen des Wirts hinzuzufügen. Da ein Referenzwirt diese Aktion aber nicht durchführen wird, würde diese Aktion seitens des angreifenden Wirt wieder einen Verhaltensunterschied darstellen. Wenn ein Referenz-Wirt die Ausführung eines Agenten mit der Löschung (wieder eine Aktion) des Agenten abschließt, kann diese Kenntnis nach der Ausführung nicht mehr wiedergewonnen werden, und es reicht, das Verhalten der beide Wirte während der Ausführung zu vergleichen.

Definition: ***Referenz-Wirt***

> *Ein* Referenz-Wirt *ist ein Wirt, der einen ausgeführten Agenten nicht angreift und sich somit wie erwartet verhält. Ein Referenz-Wirt schließt die Ausführung des Agenten (nach einer eventuellen Migration oder ähnlichen Aktionen) mit der Löschung des Agenten ab.*

Um einen Vergleich des Verhaltens zu erlauben, müssen beide Wirte von denselben Voraussetzungen ausgehen. d.h. denselben Agenten ausführen, dieselben Ressourcen benutzen, diesselben Ereignisse während der Ausführung des Agenten bekommen usw. Wenn wir in der Lage wären, den Verhaltensunterschied zwischen einem zu untersuchenden und einem Referenz-Wirt zu messen, dann könnten wir Angriffe zwar nicht verhindern, aber doch zumindest feststellen.

Diese Auffassung von Angriffen spiegelt den Aspekt der Mutwilligkeit von Angriffen absichtlich nicht wieder, umfasst also auch abweichendes Verhalten aufgrund von unbeabsichtigten Fehlern, die z.B. durch eine fehlerhafte Interpretation der Spezifikationen oder durch technische Fehler hervorgerufen werden. Zum einen ist es schwer, Absichten zu erkennen, zum anderen können diese Mechanismen dann auch auftretende Fehler behandeln.

Nehmen wir zunächst an, der Wirt führt maximal einen Agenten pro Zeiteinheit aus, und führt darüberhinaus keine weiteren Aufgaben aus. Nehmen wir an, der Agent wäre deterministisch. Nehmen wir an, der Code wird durch die Ausführung eines Agenten nicht geändert, und bleibt daher für einen Agenten konstant. Nehmen wir weiter an, der Wirt unterteile sich in einen Wirtszustand und in einen zustandslosen Wirtskern. Der *Wirtszustand* bestehe aus den variablen Teilen, der *zustandslose Wirtskern* aus dem Rest. Nehmen wir an, ein Agenten kommuniziert ausschließlich unter Benutzung der Kommunikation des Wirts. Nehmen wir schließlich an, die Tätigkeit des Wirts lasse sich in Schritte einteilen, wobei ein Schritt immer eine der folgenden Aktionen sowie die damit verbundenen Daten enthält:

- die Änderung des Agentenzustands (samt neuen Agentenzustand)
- die Änderung des Wirtszustands (samt neuem Wirtszustand)
- den Empfang eines Datums von außerhalb des Wirts (samt Datum)
- die Sendung eines Datums nach außerhalb des Wirts (samt Datum)

Die Eingaben an einen Agenten bestehen damit im Empfang eines Datums und der Änderung des Agentenzustands, die Ausgaben in der Sendung eines Datums durch den Wirt.

Mittels dieser Annahmen lässt sich definieren:

Definition: ***Genaues Verhalten eines Wirts während einer Ausführungssitzung***

> *Das* genaue Verhalten eines Wirts während einer Ausführungssitzung *ist die Abfolge von Schritten, die ein Wirt nach Erhalt eines Agenten bis zur Beendigung der Ausführung durchführt.*

Eine mögliche Definition eines Angriffs wäre nun:

These 1: ***Angriff***

> *Ein* Angriff *ist ein Unterschied im genauen Verhalten eines Wirts während eines Ausführungssitzung zwischen einem zu untersuchenden Wirt (bestehend aus dem initialen Wirtszustand* ***Z*** *und einem zustandslosen Wirtskern* ***A****) und einem Referenzwirt (bestehend aus dem gleichen initialen Wirtszustand* ***Z*** *und einem anderen zustandslosen Wirtskern* ***R****).*

Das Problem dieser Definition besteht darin, dass die Abfolge von Schritten zum Teil nicht gleich sein muss, und trotzdem kein Angriff vorliegt. Eine Wirtsimplementierung könnte beispielsweise bei mehreren Eingaben an den Agenten erst alle sammeln und dann den Zustand des Agenten ändern, sofern der Inhalt einer Eingabe nicht vor der nächsten Eingabe verarbeitet wird. Eine andere könnte den Agentenzustand direkt nach jeder Eingabe ändern.

Man könnte nun versuchen, dieses Problem durch Angabe einer geeigneten Äquivalenzrelation auf Abfolgen von Schritten zu lösen, aber letztendlich reicht es, dass man die Reihenfolge der Kommunikationsschritte und die jeweils letzten Zustandsänderungen betrachtet.

Definition: ***Verhalten eines Wirts während einer Ausführungssitzung***

> *Das* Verhalten eines Wirts während einer Ausführungssitzung *ist die Abfolge von Kommunikationsschritten, die ein Wirt nach Erhalt eines Agenten bis zur Beendigung der Ausführung durchführt, sowie des jeweils letzten Zustandsänderungsschritts des Agenten bzw. des Wirts.*

Eine Angriffs lässt sich somit definieren als:

Definition: ***Angriff***

> *Ein* Angriff *ist ein Unterschied im Verhalten eines Wirts während eines Ausführungssitzung zwischen einem zu untersuchenden Wirt (bestehend aus dem initialen Wirtszustand* ***Z*** *und einem zustandslosen Wirtskern* ***A****) und einem Referenzwirt (bestehend aus dem gleichen initialen Wirtszustand* ***Z*** *und einem anderen zustandslosen Wirtskern* ***R****).*

Leider stehen einige Probleme vor einer Anwendbarkeit eines entsprechenden Detektionsverfahrens von Angriffen. Die Überprüfung der Äquivalenz der Verhalten von zu überprüfendem Wirt und Referenzwirt bedarf vierer Dinge: Des initialen Zustands des Agenten, des initialen Zustands des Wirts, der Sequenz von Ein- und Ausgaben des Wirts, des Endzustands des Agenten, sowie des Endzustands des Wirts. Der initiale und der Endzustand des Agenten liegen explizit während der Migrationen hin zum und weg vom zu überprüfenden Wirt vor; sie sind also verfügbar. Da der (initiale bzw. finale) Wirtszustand unter Umständen ganze Datenbanken

umfasst, und die Ausführung des Referenzwirts auf einer vertrauenswürdigen, also einer anderen Maschine vonstatten gehen müsste, würde dieses Vorgehen den Transfer einer sehr großen Menge an Daten vorschreiben. Abgesehen davon könnte es der zu prüfende Wirt vielleicht nicht akzeptabel finden, diese Daten an dritte Parteien weiterzugeben. Außerdem kann niemand die Übereinstimmung der transferierten Daten mit denen der Datenbank überprüfen, da es gerade die Eigenschaft eines Wirts ist, dass man dessen Innenleben von außen nicht zuverlässig erforschen kann. Ähnliches gilt für die Kommunikation des Wirts. Zwar ist es durchaus möglich, die Ein- und Ausgaben des Agenten aufzuzeichnen, weil dieser kooperieren will, aber es scheint nicht möglich zu sein, ohne Kooperation des Wirts dessen Kommunikation aufzuzeichnen.

Wie wir fesgestellt haben, ist eine Detektion aller Angriffe mit dem vorgestellten Ansatz nicht möglich, weil es nicht möglich ist, *in* die Ausführung eines Agenten durch einen Wirt zu blicken. Einige der vorgestellten Auswirkungen dieser Ausführung lassen sich jedoch von außen beobachten. Diese erlauben zwar keine vollständige Entdeckung von Angriffen, ermöglichen es aber, bestimmte Angriffe zu detektieren.

7.3 Referenzzustände

Während es also schwierig ist, das Verhalten eines Wirts gegen das eines Referenz-Wirts zu messen, gilt dies nicht für die beobachtbaren Auswirkungen dieses Verhaltens. Bezogen auf einen Agenten resultieren diese Auswirkungen vor allem in den Endzuständen der Agenten nach einer Ausführungssitzung auf einem Wirt. Was man daher tun kann, ist, den Endzustand eines Agenten auf einem zu prüfenden Wirt mit dem Endzustand auf einem Referenz-Wirt zu vergleichen. Daher definieren wir:

Definition: ***Referenzzustand***

> *Ein* Referenzzustand *ist die Menge aller variablen Teile eines mobilen Agenten nach der Ausführung auf einem Referenz-Wirt.*

Um zu einem verwertbaren Vergleich des Referenzzustandes mit dem Endzustand auf dem zu prüfenden Wirt zu kommen, ist es notwendig, dass für beide Ausführungen dieselbe Eingabe benutzt werden kann. Wie im Ausführungsmodell erläutert, umfasst diese Eingabe auch die Resultate des Aufrufs von Systemfunktionen wie etwa Zufallszahlen, aber z.B. nicht die Resultate von Funktionen, die Teil des Agentencodes sind, da diese als Teil der normalen Ausführung eines Agenten berechnet werden können.

Wenn wir nun in der Lage sind, den Unterschied zwischen dem Referenzzustand und dem zu prüfenden Endzustand zu festzustellen, können wir fast alle Angriffe feststellen, die sich auf den Endzustand auswirken. Diese Angriffe umfassen Schreibe- bzw. Modifikationsangriffe gegen die variablen Teile eines Agenten sowie einige Angriffe, bei denen der Code des Agenten nicht gemäß der Spezifikation ausgeführt wird. Der Umstand, dass nicht alle solchen Angriffe entdeckt werden, sondern nur solche, die sich im Zustand des Agenten auswirken, ist nicht nur eine Einschränkung dieses Ansatzes, sondern kann auch in manchen Fällen von Vorteil sein. Wenn z.B. der Wirt den Code des Agenten temporär ändert, um dessen Effizienz zu optimieren, aber zu den gleichen Resultaten gelangt, muss er nicht befürchten, des Angriffs gegen den Agenten bezichtigt zu werden. Mit diesem Ansatz nicht festgestellt werden können Leseangriffe und Angriffe, bei denen der Wirt (von dem Eingaben stammen, oder der sie übermittelt hat) diese Eingaben modifiziert oder unterdrückt.

7.4 Bestehende Ansätze

In diesem Abschnitt werden wir vier bestehende Ansätze analysieren, die Referenzzustände benutzen, um Angriffe durch Wirte festzustellen. Dazu werden diese Ansätze erst beschrieben und kurz diskutiert, danach werden wir die Ansätze nach Kriterien wie dem Zeitpunkt der Überprüfung und den benutzten Vergleichsdaten klassifizieren.

Ansatz 1: „State appraisal“

Farmer, Guttman und Swarup beschreiben in [FGS96] einen “state appraisal”-Ansatz, der die Plausibilität eines Zustands als ersten Schritt nach der Ankunft eines Agenten auf einem Wirt testet. Dieser Mechanismus betrachtet dabei nur den reinen Zustand. Der Testmechanismus wird durch eine Prozedur realisiert, die Teil des Agenten ist, und zu Beginn ausgeführt wird. Sie kann z.B. aus einer Menge von Bedingungen bestehen, die bei der Ankunft erfüllt sein müssen. In diesem Fall besteht kein expliziter Referenzzustand. Statt werden „Meta-Regeln“ über den initialen Zustand in den Bedingungen kodiert. Die Bedingungen werden durch den Programmierer formuliert, der Relationen zwischen bestimmten Datenelementen kennt. Der Test wird vom Ziel-Wirt durchgeführt, und es ist in dessen Interesse, den Test durchzuführen, da dieser nur gültige, d.h. nicht angegriffene Agenten ausführen will (die dem Wirt sonst Schaden könnten). Wenn der Wirt den Test nicht durchführt (z.B. weil er mit dem angreifenden Wirt zusammenarbeitet), können keine Angriffe entdeckt werden.

Die Frage, welche Angriffe denn erkannt werden, wenn der Test durchgeführt wird, hängt von der Mächtigkeit des benutzten Test-Formalismus ab. Wenn z.B. Bedin-

gungen benutzt werden, die nur Boolesche und numerische Operatoren enthalten können (d.h. wenn die benutzten Testkonstrukte nicht Turing-mächtig sind), gibt es Berechnungen, die nur durch den Code des Agenten ausgeführt, aber nicht durch die Bedingungen nachvollzogen werden können. Das bedeutet, dass nicht alle Berechnungen durch diese Art von Bedingungen getestet werden können. Der Plausibilitätstest betrachtet nur den initialen Zustand, nicht aber auch die Eingaben an den Agenten während dessen Ausführung. Daher kann es auch hier Angriffe geben, die nicht entdeckt werden können. Man kann sich z.B. einen Agenten vorstellen, der Preisangebote durch entfernte Kommunikation einholt, den billigsten Preis berechnet und die übrigen Preise löscht. Dann kann der Wirt die Ausführung und/oder die Angebote beliebig modifizieren, ohne entdeckt zu werden, da es unmöglich ist, eine Inkonsistenz in der Ausführung zu bemerken, ohne auf die eingeholten Preise zurückgreifen zu können.

Ansatz 2: „Server replication“

In [MRS96] schlagen Minsky et al. vor, Mechanismen zur Fehlertoleranz auch dazu zu verwenden, Angriffe durch böswillige Wirte festzustellen. Die Grundannahme hierbei ist, dass es für jede Ausführungssitzung (die in diesem Ansatz “Stage” heißt) eine Menge von unabhängigen, d.h. von verschiedenen Parteien betriebene, replizierte Wirte gibt. Repliziert bedeutet in diesem Zusammenhang, dass die Wirte dieselbe Menge an Ressourcen, vor allem dieselben Daten, anbieten. Jeder Ausführungsschritt wird parallel von der Menge der replizierten Wirte ausgeführt. Nach der Ausführung stimmen die Wirte über die Ergebnisse, d.h. die Agentenzustände, ab. Dies geschieht dadurch, dass die Zustände an alle replizierten Wirte der nächsten Ausführungssitzung geschickt werden. Jeder Wirt vergleicht die erhaltenen Zustände und benutzt denjenigen, der am häufigsten errechnet wurde. Wenn man annimmt, dass alle Wirte, die den Agenten nicht angreifen, zu demselben Ergebnis kommen, dann kann dieser Ansatz selbst $\langle n/2 - 1 \rangle$ böswillige Wirte tolerieren. Der Referenzzustand wird hier also aus einer Menge von Ausführungen auf unbekannten Wirten über die Mehrheit der Stimmen berechnet. Da die replizierten Wirte parallel arbeiten, müssen externe Eingaben an den Agenten an alle replizierten Wirte parallel geschickt werden (Eingaben, die direkt von einem Wirt kommen, werden in gleicher Weise auch auf den anderen replizierten Wirten erzeugt, da ja alle Ressourcen gleich sind).

Dieser Ansatz kann alle Angriffe, die in einem veränderten Zustand resultieren, feststellen, wobei selbst gemeinsame Angriffe von weniger als $n/2$ Wirte pro Ausführungssitzung toleriert werden können.

Ansatz 3: „Execution traces“

```
10  read(x)
11  y=x+z
12  m=y+1
13  k=cryptInput
14  m=m+k
```

Bild 22: Code-Fragment

Neben dem Testen der inhärenten Integrität von Zuständen und dem Vergleichen von Zuständen, die durch parallele Ausführung entstehen, besteht die dritte grundlegende Idee, Referenzzustände zu benutzen, darin, den ausführenden Wirt ein Ausführungsprotokoll oder “trace” erstellen zu lassen. In [Vig98] stellt Vigna einen Ansatz vor, der diese Idee benutzt, um es einem Agenteneigentümer zu ermöglichen, die Ausführungssitzungen seines Agenten auf verschiedenen Wirten zu überprüfen, wenn ein Angriff vermutet wird. Zu diesem Zweck erstellt jeder Wirt ein Ausführungsprotokoll, das wie in Bild 23 dargestellt, aufgebaut ist.

```
10  x=5
11
12
13  k=2
14
```

Bild 23: Trace des Code-Fragmentes

Ein Ausführungsprotokoll besteht aus Paaren `(n,s)` bei denen `n` einen Bezeichner einer ausgeführten Anweisung bezeichnet. Falls diese Anweisung den Zustand des Agenten durch Einbeziehung von Daten außerhalb des Agenten verändert (d.h. eine Eingabe benutzt), bezeichnet `s` die Liste mit Variablennamen-Wert-Paaren, die den Inhalt dieser Variablen nach Ausführung dieser Anweisung wiedergeben.

Nach der Ausführungssitzung berechnet der Wirt einen Hash-Wert des Ausführungsprotokolls und einen des Endzustandes, signiert diese digital und schickt diese zusammen mit dem Agenten an den nächsten Wirt. Das Ausführungsprotokoll selbst wird vom Wirt gespeichert. Der Agent wird so auf jedem Wirt ausgeführt, bis der Agent wieder zurück zu seinem Heimatwirt migriert. Nun kann sich der Eigentümer des Agenten entscheiden, ob er die Ausführung überprüfen will oder nicht. Im Falle eines Verdachts fordert der Eigentümer das Ausführungsprotokoll vom ersten Wirt an, auf den der Agent migriert ist. Der Eigentümer berechnet den Hash des Protokolls und vergleicht diesen mit dem, den der nächste Wirt vom ersten erhalten hat. Sind diese identisch, ist sichergestellt, dass der erste Wirt dieses Ausführungsprotokoll erzeugt hat. Danach wird der Agent erneut beginnend vom Startzustand auf dem ersten Wirt ausgeführt. Falls eine Eingabe von außen benutzt wird, wird diese dem Ausführungsprotokoll entnommen. Nachdem der Agent wieder ausgeführt worden ist, wird der Hash des Endzustandes mit demjenigen verglichen, den der zweite Wirt bekommen hat. Falls beide identisch sind, konnte die Ausführung nachvollzogen werden, und der Prüfprozess wird mit dem zweiten Wirt bzw. den folgenden Wirten fortgeführt, bis alle Ausführungen geprüft sind oder eine Unstimmigkeit entdeckt wurde. Der Fall, dass ein Wirt vorgibt, einen anderen Agentenzu-

stand erhalten zu haben als denjenigen, den er vom vorherigen Wirt in Wirklichkeit erhalten hat, wird dadurch ausgeschlossen, dass ein Wirt die Hashes, die er erhalten hat, digital signiert an den Sender zurückschickt.

Offensichtlich wird mit jeder Ausführungssitzung ein neues Protokoll erzeugt, das mit jedem ausgeführten Schritt wächst. Um die Protokolle zu verkleinern schlägt Vigna daher vor, die Anweisungsbezeichner im Protokoll wegzulassen. Es gibt aber noch andere Gründe, warum diese weggelassen werden können. Erstens kann man argumentieren, dass ein korrekter Endzustand wichtiger ist als ein korrekter Ausführungsweg. Zweitens beweisen die Anweisungsbezeichner keineswegs, dass der Endzustand durch die Serie der bezeichneten Anweisungen entstanden ist. Es ist jederzeit möglich, eine korrekte Liste mit Bezeichnern zu generieren und diese mit den korrekten oder modifizierten Eingaben zu versehen. In diesem Fall wird ein Angriff erst dann festgestellt, wenn der nachgerechnete Endzustand mit dem angegebenen Endzustand verglichen wird, nicht aber früher, wenn die Bezeichner überprüft werden. Daher werden die Bezeichner auch von einem Sicherheitsstandpunkt aus nicht benötigt.

Dieser Ansatz entdeckt alle Angriffe, die zu einem modifizierten Endzustand führen solange der Wirt nicht in der Lage ist, die Eingabe zu modifizieren. Der Ansatz erlaubt es, den Wirt festzustellen, der einen Angriff beging, nicht aber, die Unterschiede zum korrekten Agentenzustand festzustellen, da nur Hashes der Endzustände gespeichert werden.

Ansatz 4: „Proof Verification“

In [Yee99] wird ein Ansatz vorgestellt, der den Begriff des Beweises (“Proof”) der korrekten Ausführung benutzt. Wie im letzten Ansatz auch besteht ein Beweis aus Ausführungsinformation und dem Endzustand, und soll beweisen, dass die Ausführung durch den Wirt korrekt war. Die Idee ist nun aber, dass es einen effizienteren Weg gibt, die Ausführung zu prüfen, als sie noch einmal nachzuvollziehen. Dazu wird die Existenz von holographischen Beweisen aufgezeigt, die in der Lage sind zu beweisen, dass ein Ausführungsprotokoll existiert, das zu einem bestimmten Endzustand führt, indem nur konstant viele Bits des Beweises geprüft werden. Ein Protokoll, das diese Idee aufgreift, wird in [BMW98] beschrieben. In diesem Protokoll werden alle Beweise zum Eigentümer des Agenten geschickt, der die Beweise prüft, nachdem der Agent seine Aufgabe beendet hat. Nachdem zuerst nur bekannt war, dass der holographische Beweis eine Länge von $l^d \ \langle d > 0 \rangle$ hat, wobei l die Länge des Ausführungsprotokolls ist, bewiesen Biehl, Meyer und Wetzel in [BMW98] dass Beweise existieren, die sogar sublinear bzw. polylogarithmisch bezüglich der Länge der Agentenlaufzeit (d.h. bezüglich der Anzahl der abgearbeiteten Schritte) sind.

Das Problem dieses Ansatzes besteht darin, dass z.Zt. nur NP-harte Algorithmen bekannt sind, die in der Lage sind, holographische Beweise zu erzeugen. Aus diesem Grund wird dieser Ansatz noch nicht praktisch eingesetzt und wird daher auch im weiteren nicht weiter betrachtet.

7.5 Bewertung der bestehenden Ansätze

Um die Stärke des erzielten Schutzes von Ansätzen, die Referenzzustände benutzen, untersuchen zu können, müssen die gemeinsamen Merkmale der präsentierten Ansätze extrahiert und deren Beziehung untersucht werden. Diese sind:

7.5.1 Prüfzeitpunkt

Gegen den Referenzzustand kann entweder

- nach jeder Ausführungssitzung oder
- nachdem der Agent seine Aufgabe beendet hat

geprüft werden.

Da es das Ziel der Ansätze ist, den angreifenden Wirt festzustellen und weil solche Wirte überall entlang der Migrationsroute des Agenten auftreten können, bedeutet die Wahl der zweiten Alternative auch, dass erstens die zurückgelegte Migrationsroute in einer sicheren, d.h. vor Modifikationen sicheren Art und Weise abgespeichert sein muss. Dies kann entweder durch das inkrementelle Ablegen der Route im Zustand, durch Kommunikation mit einem vertrauenswürdigen Wirt oder durch einen a priori festliegenden Routenplan geschehen. Zweitens müssen dann die Referenzdaten für jede Ausführungssitzung gespeichert werden, weil sonst die Ausführung nicht nachgeprüft werden kann.

Im Prinzip könnte man auch in kleineren Intervallen als den Ausführungssitzungen prüfen, aber da der Wirt durch das parallele Ausführen eines korrekten und eines angegriffenen Agenten jederzeit das Verhalten eines korrekten Agenten nach außen simulieren kann, ist dieses Vorgehen nicht aussagekräftig.

7.5.2 Benutzte Referenzdaten

In Abhängigkeit vom Prüfzeitpunkt können auch die zum Vergleich benutzten Referenzdaten variieren. Nach einer Ausführungssitzung oder nach Vollzug der Gesamtaufgabe kann eine beliebige Kombination aus Anfangszustand, Endzustand, Eingabe, Ausgabe, Ausführungsprotokoll und benutzten Wirt-Ressourcen verwendet werden.

7.5.3 Benutzter Prüfalgorithmus

Unabhängig vom benutzten Prüfzeitpunkt kann jeder der folgenden Algorithmen verwendet werden. Die Algorithmen markieren dabei allerdings nur einige Punkte auf einer kontinuierlichen Skala von möglichen Algorithmen.

Regeln

Dieser Begriff bezeichnet einfache (d.h. nicht Turing-mächtige) Mechanismen, die es erlauben, z.B. Nachbedingungen in Form von Logik erster Ordnung zu formulieren (z.B. `ausgegebenesGeld + restGeld = initialesGeld`). Wie bereits in Kapitel 7.4 argumentiert wurde, können solche Mechanismen zwar nicht alle Angriffe feststellen, sind aber einfach zu formulieren und zu testen. Regeln können alle beschriebenen Referenzdaten benutzen.

Beweise

Der Begriff “Beweise” bezeichnet in diesem Zusammenhang Repräsentationen von Ausführungsprotokollen, die leichter zu prüfen sind als die Protokolle selbst. Beweise benötigen keine weiteren Referenzdaten, da sie alle relevanten Daten bereits enthalten.

Nachrechnen

Das Ziel des Nachrechnens ist es, einen Referenzzustand zu erzeugen, mit dem der zu prüfende Zustand verglichen werden kann. Wie bei Regeln und Beweisen kann auch hier der Prüfprozess automatisiert werden, d.h. durch Systemfunktionen unterstützt werden. Dazu werden einfach die Endzustände beider Ausführungen und eventuell zusätzlich die Ausführungsprotokolle verglichen. Aus diesem Grunde benötigt dieser Algorithmus an Referenzdaten Eingaben, den Anfangszustand, sowie das Ausführungsprotokoll oder den Endzustand. Die Mächtigkeit des Ansatzes hängt davon ab, wieviele der Referenzdaten verwendet werden. Wird beispielsweise kein Ausführungsprotokoll verwendet, kann der Wirt die Eingaben modifizieren. Aber selbst wenn die Eingaben im Protokoll zur Prüfung verwendet werden, kann nicht einfach festgestellt werden, ob alle Eingaben auch wirklich verzeichnet sind.

Man kann einwenden, dass es unmöglich ist, die Bedingungen der ursprünglichen Ausführungen für die Prüfung wiederherzustellen, da diese Bedingungen z.B. “race conditions” im Fall von parallelen Threads umfassen können. Als Beispiel für diesem Umstand kann man sich vorstellen, dass ein Agent eine Liste aus einer Eingabe berechnet, in der die Reihenfolge der Elemente vom Timing von zwei Threads abhängt, die der Agent benutzt. Damit kann diese Liste nicht einfach Element für Element mit der Liste einer anderen Ausführung verglichen werden, da die andere Liste dieselben Elemente in anderer Reihenfolge enthalten kann. Um dieses Pro-

blem, und das Problem, dass Eingaben authentifiziert werden sollten, zu lösen, benötigen wir einen mächtigeren Algorithmus.

Beliebiger Code

Dies ist der mächtigste Algorithmus, da er alle vorherigen enthält und weitere Verfahren erlaubt wie z.B. eine spezifische Methode zum Vergleichen von Endzuständen oder die Möglichkeit, die Kommunikationspartner nach den empfangenen Nachrichten zu fragen. Da der genaue Algorithmus durch den Programmierer festgelegt wird und damit nicht vorher bekannt ist, kann das System hier nur sehr grundlegende Unterstützung anbieten, d.h. die Möglichkeit, beliebige Programme zu den Prüfzeitpunkten auszuführen. Auch hier können alle möglichen Referenzdaten benutzt werden.

Diskussion

Die Kombination dieser Merkmale öffnet einen Raum an möglichen Ansätzen, die diese Merkmale kombinieren, der wesentlich mehr Möglichkeiten umfasst als die vorgestellten Verfahren (siehe auch Tabelle 1 in Kapitel 7.8). Wenn wir den Programmierer entscheiden lassen wollen, welcher Schutzmechanismus für seine spezifische Anwendungen angemessen ist, müssen wir ihm die Möglichkeit geben, aus dem gesamten Ansatzraum auszuwählen statt ihm nur einen einzelnen Mechanismus anzubieten.

7.6 Stärken und Schwächen der Ansätze

Wie bereits dargestellt, können Ansätze, die Referenzzustände benutzen, nicht alle Angriffe feststellen. In diesem Unterkapitel werden wir den erzielten Schutz untersuchen, Anwendungen identifizieren, die nicht geschützt werden können, und mögliche Erweiterungen diskutieren.

7.6.1 Erzielter Schutz

Der erzielte Schutz hängt von den im letzten Abschnitt untersuchten Merkmalen ab, d.h. von Prüfungszeitpunkt, den benutzten Referenzdaten sowie dem Prüfalgorithmus. Ein Mechanismus, der nur geringen Schutz liefert, benutzt nur die schwächsten Merkmale d.h. er prüft die einzelnen Ausführungen erst nach Beendigung der Gesamtaufgabe, benutzt pro Prüfung nur den Endzustand und verwendet nur einfache Regeln zur Prüfung. Aufgrund dieser Charakteristik werden Angriffe nicht sofort erkannt, und der angegriffene Agent wird danach noch von weiteren Wirten ausgeführt. Es ist zwar möglich zu erkennen, wer schuld daran ist, dass der Agent unter Umständen unerwünschte Reaktionen zeigt (wie z.B. 1000 statt 10 Rosen zu kaufen), aber es ist schwierig, diese Reaktionen zu kompensieren, vor allem wenn sie auf ehrlichen Wirten stattgefunden haben. Da nur Regeln benutzt werden, kön-

nen viele Angriffe nicht erkannt werden. Wenn beispielsweise eine Regel prüft, ob das initiale Geld gleich der ausgegebenen Summe plus dem Rest ist, kann ein unerwünschter Kauf nicht entdeckt werden. Obwohl dieser Mechanismus sehr effizient arbeitet und die Ausführung des Agenten nicht verzögert, ist er vom Sicherheitsstandpunkt aus gesehen nicht sehr wirkungsvoll.

Ein Mechanismus, der alle sich ihm Möglichkeiten nutzt, prüft nach jeder Ausführungssitzung, benutzt alle möglichen Referenzdaten und erlaubt beliebige Programme als Prüfalgorithmus. Wenn ein Wirt einen Agenten empfängt, kann er im Falle einer erfolgreichen Prüfung sicher sein, dass der Zustand des Agenten zu dessen Eingaben passt. Da der Mechanismus auch ein Nachrechnen der letzten Ausführung erlaubt, kann diese Ausführung nachvollzogen werden. Wenn es der Prüfalgorithmus zusätzlich erlaubt, die Eingaben der Kommunikationspartner verifizieren zu lassen, kann auch dieser Aspekt geschützt werden. Offensichtlich ist dieser Mechanismus also sicherer als der obige. Der Preis für diese erhöhte Sicherheit ist aber der Mehraufwand an Berechnung und Kommunikation: Erstens verdoppeln sich grob die Berechnungskosten bei einmaligem Nachrechnen, und zweitens muss ein Zustand pro Migration mehr sowie die Eingaben transportiert werden

Auch wenn Angriffe festgestellt werden können, erhebt sich allerdings die Frage nach den Konsequenzen. Nur wenn juristische, organisatorische oder soziale Schritte gegen einen Angreifer unternommen werden können, sind Schemata wie die in diesem Kapitel untersuchten sinnvoll. Obwohl diese Überlegungen durchaus die Frage nach der Gesamtsicherheit betreffen, stehen sie doch außerhalb des Themas dieser Arbeit. Nichtsdestotrotz verdienen sie weitere Untersuchung.

7.6.2 Nicht-entdeckbare Angriffe

Inkorrekte Ausführungen (also Angriffe), die sich von korrekten Ausführungen nicht durch den Endzustand unterscheiden, können mit den vorgestellten Schutzmechanismen nicht entdeckt werden. Vor allem Leseangriffe liegen damit außerhalb des zu schützenden Bereichs, also Angriffe, die nur darauf abzielen, den Inhalt von Variablen zu erfahren, da diese Angriffe keine Spuren hinterlassen. Weitere Angriffe, die nicht festgestellt werden können, sind solche, bei denen der Wirt die Eingaben an den Agenten unbemerkt modifizieren oder unterdrücken kann, sowie solche, bei denen der Angreifer den Agenten zu ungewollten Aktionen zwingt (z.B. zum Kauf einer Ware), dann aber eine korrekt ausgeführte Version des Agenten weitergibt. Letzterer Angriff entspricht zwar einem bestimmten Leseangriff, verlangt aber unter Umständen weniger Wissen über die innere Struktur des Agenten. Insgesamt müssen also zusätzlich noch andere Mechanismen benutzt werden, wenn ein Agent umfassend geschützt werden soll.

7.6.3 Mögliche Erweiterungen

Um zu verhindern, dass Wirte Einfluss auf die zur Prüfung benutzte Eingabe eines Agenten nehmen können, kann man digital signierte Eingaben benutzen, deren Integrität dann die prüfende Stelle sicherstellen kann. Um die Unterdrückung von Eingaben durch den Wirt gegenüber dem Agenten auszuschließen kann man ein Verfahren anwenden, das sicherstellt, dass alle Interaktionspartner im Nachhinein gefunden werden können.

Eine korrekte Ausführung eines Agenten besteht nicht nur im Erzeugen des Referenzzustandes; auch die *Ausgaben* eines zu prüfenden Agenten sollten denen des Referenzagenten entsprechen. Daher könnte man weitere Verfahren entwickeln, die es gestatten, die Kommunikationspartner nach den erfolgten Ausgaben des Agenten zu fragen.

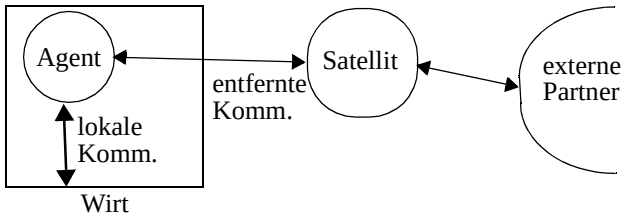


Bild 24: Satellitenkonzept

Wenn man beide Aspekte, den der korrekten Eingabe und den der korrekten Ausgabe, zusammennimmt und auch die Korrelation zwischen beiden als schützenswert ansieht, kommt man auf eine zu prüfende Kommunikationshistorie (bei der der Wirt selbst immer bezüglich des von ihm erzeugten Teils der Kommunikation lügen kann). Eine Möglichkeit, diese Historie zu prüfen, könnte die Einführung eines vertrauenswürdigen Stellvertreters sein, der auf einem anderen Wirt liegt, und über den jegliche externe Kommunikation mit dem Agenten indirekt abgewickelt wird bis auf die mit dem Wirt selbst, die nicht kontrollierbar ist. Gleichzeitig ist diese Einschränkung aber sogar aus Effizienzgesichtspunkten wünschenswert, weil dann lokale Interaktion nicht verteuert wird (auch wenn durch die Notwendigkeit der Aufzeichnung auch der lokalen Kommunikation natürlich immer noch ein Mehraufwand gegenüber ungeschützten Agenten entsteht).

Erhält man nun durch den Einsatz dieses „Satellitenkonzeptes“ eine Kommunikationshistorie aus Sequenzen von Ein- und Ausgaben, findet die Überprüfung derselben dadurch statt, dass, etwa beim Nachrechnen, die Eingaben in die Nachrechnung einbezogen werden und die so erhaltenen Ausgaben gegen die Ausgaben in der Kommunikationshistorie geprüft werden.

Die Benutzung eines Kommunikationsstellvertreters oder “Satelliten” erzeugt natürlich Mehraufwand, aber er entsteht nur dann, wenn eine entfernte Kommunikation stattfindet, und erhält die Vorteile der Lokalität von Agent und Wirt.

7.7 Systemunterstützung für Schutzmechanismen

In diesem Abschnitt wird ein Framework präsentiert, das die Implementierung von Sicherheitsmechanismen unterstützt, die Referenzzustände benutzen. Es beruht dabei auf der Unterstützung der Merkmale, die in Kapitel 7.5 gefunden wurden. Die generelle Idee ist, den Programmierer selbst den eigentlichen Prüfalgorithmus implementieren zu lassen und die grundlegenden Funktionalitäten, wie das Signieren der Referenzdaten bereitzustellen. Obwohl es für das Mobile-Agenten-System Mole implementiert wurde, kann das im Folgenden vorgestellte Schema für beinahe jedes in Java implementierte Agentensystem verwendet werden, das schwache Migration* unterstützt und den Aufruf von Prozeduren durch den Wirt im Rahmen der Ausführung des Agenten zulässt (siehe [Hoh00]). Dies ist für die meisten Systeme der Fall (siehe [MAL] für die Charakteristika der meisten Agentensysteme). Die nachfolgende Beschreibung des Frameworks erfolgt anhand der oben erwähnten Merkmale.

Prüfzeitpunkt

Um die verschiedenen Möglichkeiten des Prüfzeitpunkts zu unterstützen, benötigen wir verschiedene Rückrufprozeduren (engl. *callbacks*), die nach der Ankunft auf einem neuen Wirt bzw. nach Beendigung der Gesamtaufgabe bei Ankunft auf dem Heimatwirt aufgerufen werden (siehe Bild 25). Die erste Rückrufprozedur heißt

* Bei der schwachen Migration werden nur der Code- und der Datenzustand, nicht aber der Ausführungszustand transportiert.

`checkAfterSession`, der zweite `checkAfterTask`. Er wird durch den letzten Wirt, der den Agenten ausführt, aufgerufen. Dies ist zumeist der Heimatwirt.

Rückrufprozeduren im Agenten

`checkAfterSession()`
This method is called by the host as the first action when arriving

`checkAfterTask()`
This method is called by the last host

Vom Agenten implementierte Interfaces

`InitalStateRequester`
declares need for initial state

`ResultingStateRequester`
declares need for resulting state

`InputRequester`
declares need for input

`ExecutionLogRequester`
declares need for execution log

`ResourceRequester`
declares need for host resources

Bild 25: Rückrufprozeduren und Schnittstellen des Frameworks

Benutzte Referenzdaten

Um verschiedene Referenzdaten zu unterstützen, müssen wir nur zwei Dinge tun: Erstens müssen wir sicherstellen, dass wir am Ende einer Ausführungssitzung die benötigten Daten in einer Form haben, die es uns erlaubt, diese Ausführung zu prüfen. Zweitens müssen wir sicherstellen, dass diese Daten zu den Wirten, auf denen die Prüfung stattfindet, transportiert werden. Letzteres ist in Mobile-Agenten-Systemen äußerst einfach. Alles, was wir tun müssen, ist, diese Daten im Datenteil des Agenten zu speichern, da dieser automatisch bei der Migration auf den nächsten Wirt transportiert wird. Ersteres ist dagegen etwas schwieriger. Der Anfangs- und der Endzustand stellen kein Problem dar, da diese sowieso bei der Migration entstehen und transportiert werden. Replizierte Ressourcen werden einfach als Daten dem Datenteil hinzugefügt (auch wenn diese unter Umständen sehr groß sein können). Um die Eingabe oder das Ausführungsprotokoll zu erstellen, gibt es zwei mögliche Wege. Entweder werden diese Informationen durch die Java Virtual Machine (JVM)

gesammelt, die als Ausführungsumgebung z.B. Zugriff auf die Zeilennummern der Anweisungen hat. Oder aber sie werden von in den Agenten eingefügten Code gesammelt, der entweder automatisch oder manuell erzeugt wird. Manuell erzeugter Code hat den Vorteil, dass der Programmierer dann damit das effizienteste Datenformat erzeugen kann, wenn auch der Prüfalgorithmus von ihm manuell erzeugt wird.

Schließlich müssen wir noch eine Möglichkeit vorsehen, die Referenzdaten auszuwählen, die zum Prüfen benutzt werden sollen. Falls die Referenzdaten durch manuelles Instrumentieren des Codes erzeugt werden, wird auch die Auswahl durch den Programmierer in den Code implementiert. Falls eine automatische Instrumentierung erfolgt, müssen die benötigten Referenzdaten spezifiziert werden. Dies kann durch die Deklaration der Implementierung verschiedener Interfaces wie `InitalStateRequester`, `ResultingStateRequester`, `InputRequester`, `ExecutionLogRequester`, und `ResourceRequester` geschehen (siehe Bild 25), analog zur Verwendung von `Cloneable` in Java.

Prüfalgorithmus

Da die “beliebiger Code”-Alternative sowohl die mächtigste Variante und damit die anderen Alternativen enthält, reicht es, diese Möglichkeit anzubieten. Dies wird zum Prüfungszeitpunkt durch die Ausführung von Code erreicht, den der Programmierer des Agenten geschrieben hat. Falls eine der anderen Alternativen benutzt werden soll, kann dies entweder der Programmierer selbst implementieren oder er kann zu diesem Zweck die allgemeingültigen Prozeduren mit speziellen überladen. So können Regeln beispielsweise durch einen speziellen Regelmechanismus unterstützt werden oder direkt als Programmanweisungen formuliert werden. Um Beweise zu unterstützen, müsste man erst die genaue Struktur des Beweises analysieren. Wenn dieser aus Daten besteht, könnten diese einfach mittransportiert werden und man muss dann nur noch die Prüfroutinen, durch die Möglichkeit, Code auszuführen, mitführen. Falls der Beweis (eventuell zusätzlich) aus Code besteht, kann dieser ebenfalls leicht durch den dargestellten Mechanismus eingebunden werden.

Die Unterstützung für das Nachrechnen kann auf mehreren Ebenen stattfinden. Ein Problem stellt hierbei die Frage dar, wie man vom Original-Code zum Code für das Nachrechnen kommt. Erstens muss der Original-Code ein zweites Mal ausgeführt werden, wobei die Eingaben aus den Referenzdaten kommen. Zweitens können Ausgaben unterdrückt werden, da sie für die Prüfung nicht benötigt werden (eine Ausnahme stellt die Erweiterung aus Kapitel 7.6.3 dar, bei der auch die Ausgaben zur Prüfung benutzt werden). Drittens muss der so erzeugte Endzustand mit dem zu prüfenden Endzustand in einer Weise verglichen werden, die es dem Programmierer

erlaubt, diese selbst zu erstellen (aufgrund der diskutierten Probleme mit einem automatischen Vergleich in Kapitel 7.5.3). Lösungen dieses Problems umfassen eine modifizierte Ausführungsumgebung (z.B. JVM), die in der Lage ist, statt der normalen Eingabeinteraktionen Referenzdaten zu verwenden, eine Kopie des Originalcodes, die automatisch um die benötigten Aktionen (zweite Ausführung, Ausgabeunterdrückung und Zustandsvergleich) erweitert wird, sowie eine Kopie des Originalcodes, die manuell vom Programmierer instrumentiert wird. Letztere Möglichkeit wurde gewählt, um einen beispielhafte Schutzmechanismus mithilfe dieses Frameworks zu implementieren, der gleichzeitig einen neuen Ansatz im Bereich der Referenzzustände darstellt (siehe nächsten Abschnitt).

7.8 Ein neuer Ansatz

Die Ansätze in Kapitel 7.4 stellen nur einige Möglichkeiten aus dem Spektrum an Ansätzen dar, die Referenzzustände benutzen können. Unterteilt man diese wie in [Wil99] nach den Kriterien “In welchem Fall wird geprüft?”, “Wann wird geprüft?”, und “Welche Referenzdaten werden benutzt?”, und “Wie hoch ist der Berechnungsaufwand?“ und nimmt man die Ansätze “Server replication” und “Proof verification” hinzu, erhält man folgende Tabelle:

Ansatz	**Prüfungs-anlass**	**Prüfungs-zeitpunkt**	**Benutzte Referenzdaten**	**Berech-nungs-aufwand**
„State Appraisal“	in jedem Fall	nach jeder Sitzung	nur Endzustand	gering[a]
„Server replication“	in jedem Fall	nach jeder Sitzung	Zustände, Eingaben	hoch[b]
„Execution traces“	bei Verdacht	nach Gesamtausführung	Zustände, Eingaben	mittel[c]
„Proof verification“	in jedem Fall	nach jeder Sitzung	Zustände, Eingaben	sehr hoch[d]

Tabelle 1: Vergleich existierender Ansätze

a. Es muss die Prüfroutine abgearbeitet werden.
b. Agenten werden n-fach ausgeführt und transportiert, n >= 3.

c. Im Verdachtsfall wird der Agent im Wesentlichen noch einmal ausgeführt. Jeder besuchte Wirt muss den Trace speichern, bis der Agent beendet wird.
d. Die Berechnung eines Beweises ist NP-hart.

Die Frage ist nun, ob es noch andere Ansätze innerhalb dieses Spektrums gibt, die Vorteile gegenüber den bestehenden aufweisen. Will man einen Ansatz, der in den ersten drei Kategorien das Maximum bietet (Prüfung in jedem Fall, nach jeder Sitzung, Benutzung von Anfangs- und Endzuständen sowie Eingaben), und keinen hohen oder sehr hohen Aufwand besitzt, muss ein neues Verfahren gefunden werden.

Da es unser Ziel ist, den Traces-Ansatz so zu modifizieren, dass die Prüfung in jedem Fall stattfindet (also ohne einen Verdacht haben zu müssen), könnte man einfach den Heimatwirt jede Ausführungssitzung prüfen lassen. Das Problem dieses ad-hoc-Ansatzes ist aber, dass sich dieser Fall nicht nur vom Modell her nicht mehr vom Client-Server-Ansatz unterscheiden würde (der Heimatwirt führt den gesamten Agenten aus und bekommt Eingaben entfernt von einigen Servern), sondern dass er sogar noch teurer als dieser Ansatz wäre, weil der Agent ja noch auf diesen Servern ausgeführt werden müsste.

Ein besserer Ansatz ist es daher, die Idee aus dem “State appraisal”-Ansatz der Prüfung einer Ausführungssitzung auf dem nächsten Wirt zu nehmen, und diese mit der generellen Idee des Nachrechnens einer Ausführung aus dem “Traces”-Ansatz zu koppeln. Dieser Ansatz wird das Problem haben, das Angriffe zweier oder mehr kollaborierender, aufeinanderfolgender Wirte nicht entdeckt werden können. Wie wir später sehen werden, kann das Verfahren jedoch auch für das Entdecken von Angriffen von mehreren kollaborierenden Wirten erweitert werden.

7.8.1 Der Ansatz

Die Idee des Ansatzes besteht darin, das Prüfverfahren des “Traces”-Ansatzes, also das Nachrechnen von Ausführungssitzungen unter Benutzung der Eingaben während dieser Sitzung zu nehmen, und es auf dem nächsten Wirt, der besucht wird, durchzuführen (siehe Bild 26). Der nächste Wirt führt diese Prüfung deshalb durch, um nicht selbst vom nächsten Wirt eines Angriffs bezichtigt werden zu können. Die

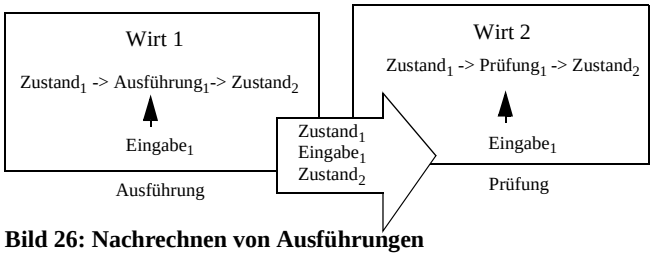


Bild 26: Nachrechnen von Ausführungen

Durchführung dieser Prüfung wird durch das Signieren eines Zustandes gegenüber dem darauffolgenden Wirt dokumentiert, kann also auch nicht mehr abgestritten werden. Wie bereits in Kapitel 7.1 diskutiert, bestehen die Eingaben aus der Gesamtheit aller vom Agenten entgegengenommen Kommunikation mit dem Wirt und dritten Parteien. Es wird dabei nicht darauf Rücksicht genommen, ob der nächste Wirt vertrauenswürdig ist oder nicht, um das Ziel zu erreichen, jede Ausführungssitzung auf dem *nächsten* Wirt zu prüfen. Wie in Kapitel 7.8.6 diskutiert werden wird, ist es mit einem prüfenden Wirt nur möglich, einen angreifenden Wirt zu entdecken. Die Erweiterung des nun folgenden Algorithmus auf die Entdeckung von n angreifenden Wirten durch den Einsatz von n Prüfungen wird in Kapitel 7.8.7 diskutiert. Wir werden mögliche Ausgaben des Agenten im Folgenden nicht betrachten. In Kapitel 7.8.8 wird dieser Aspekt aber diskutiert werden.

Unter *vertrauenswürdigen* (engl. trusted) Wirten (siehe [Hoh99]) werden dabei solche verstanden, von denen der Agent im Voraus weiß, dass diese den Agenten nicht angreifen werden. Vertrauenswürdige Wirte sind zudem bereit, Prüfungen und andere Dienstleistungen für den Agenten durchzuführen. Daher werden solche vertrauenswürdigen Wirten von Agenten entweder deshalb besucht, weil der Agent diesen Besuch für die Durchführung seiner Aufgabe benötigt, oder weil er diese Prüfdienste in Anspruch nehmen will. Wenn wir annehmen, dass ein Agent auf seinem Heimatwirt startet und seine Aufgabe am Ende auch wieder auf diesem beendet, kann man daher den ersten und den letzten Wirt der Migration eines Agenten als vertrauenswürdig einstufen. Wir setzen im Folgenden voraus, dass für beliebige Wirte auf sichere Art und Weise ermittelt werden kann, ob ein Agent einen Wirt für vertrauenswürdig hält oder nicht. Es wird in Kapitel 7.8.3 gezeigt werden, dass diese Annahme erfüllt werden kann.

Ist ein Wirt nicht vertrauenswürdig, wird er als *nicht-vertrauenswürdig* (engl. untrusted) eingestuft. Nicht-vertrauenswürdige Wirte werden besucht, weil der Agent diesen Besuch für die Durchführung seiner Aufgabe benötigt (etwa um lokal vorhandene Ressourcen zu benutzen). Auch ein nicht-vertrauenswürdiger Wirt kann Prüfungen durchführen wollen, aber nur dann, wenn er dadurch eine Verantwortlichkeit für einen Schaden am Agenten bzw. dessen Eigentümer abwenden kann. Auch solche Prüfungen können natürlich wieder Gegenstand der Angriffe durch den nicht-vertrauenswürdigen Wirt sein.

7.8.2 Das Protokoll

In diesem Abschnitt soll nun das Protokoll beschrieben werden, das den im letzten Abschnitt beschriebenen Ansatz implementiert. Dazu werden wir uns zunächst die Protokollteile ansehen, die jeder Wirt in einer allgemeinen Konfiguration abzuwikkeln hat. Diese Konfiguration besteht aus einer Liste von Wirten, die ein mobiler

Agent nacheinander besucht. Um so generelle Aussagen wie möglich zu treffen, werden wir nur annehmen, dass der erste und der letzte Wirt vertrauenswürdig sind (z.B. weil der Agent von seinem Heimatwirt startet und auf diesen am Ende wieder zurückkehrt), alle anderen Wirte aber nicht-vertrauenswürdig. Nachdem wir so ein Protokoll entwickelt haben, entwickeln wir dieses zu einer optimierten Variante, die es erlaubt, spezifischen Nutzen aus zwischendurch besuchten vertrauenswürdigen Wirten zu ziehen.

Basis-Protokoll

Im Folgenden bedeutet $sign_x(y)$ die *Signatur* der Nachricht y durch den Wirt x. Diese soll dabei nicht die Nachricht y selbst enthalten.

1.1 compute $state_2$
1.2 transfer agent code to next host(s) if needed
1.3 add $state_2$ to message m
1.4 send m,$sign_1$(m) to next host

Bild 27: Protokoll für den ersten Wirt

Der erste Wirt (also der, von dem der Agent am Anfang startet) erzeugt den initialen Zustand, transferiert gegebenenfalls den Code und übermittelt den initialen Zustand des Agenten an den nächsten Wirt. Wie auch in allen folgenden Protokollteilen wird die sich so ergebende Nachricht als Ganzes vom übermittelnden Wirt signiert an den nächsten Wirt weitergegeben, so dass die Authentizität gewährleistet ist. Bild 27 enthält den Pseudocode für den ersten Wirt.

Der zweite Wirt bekommt die Nachricht vom ersten und überprüft zunächst, ob der Agent den vorherigen Wirt als vertrauenswürdig einstuft. Falls er dies nicht tun würde, könnte sich ein beliebiger, nicht-vertrauenswürdiger Wirt als erster Wirt ausgeben und einen manipulierten Agenten in das System einschleusen. Da der zweite Wirt annimmt, dass der erste Wirt vertrauenswürdig ist, finden bestimmte Prüfungen, wie wir gleich sehen werden, nicht statt. Damit könnte ein angreifender Wirt die Sicherheitsmechanismen des Verfahrens unterlaufen. Diese Prüfung auf Vertrauenswürdigkeit findet sich im nun folgenden Pseudocode für den zweiten Wirt nicht, weil die Mechanismen zur Ermittlung vertrauenswürdiger Wirte erst in Kapitel 7.8.3 eingeführt werden.

Zuerst wird nun die Signatur der Gesamtnachricht, die nur aus dem Ausgangszustand $state_2$ besteht, verifiziert. Falls diese Signatur nicht verifiziert werden kann, wird der erste Wirt benachrichtigt. In diesem Fall kann der erste Wirt sich entweder entschließen, seine Nachricht noch einmal zu senden oder aber den Agenten zu

beenden. Dieser Aspekt berührt allerdings nicht die Frage nach der Sicherheit eines Agenten durch einen Wirt, und wird hier nicht weiter vertieft.

2.1 get $state_2$,$sign_1(state_2)$ from previous host and check signature
2.2 if not true, COMPLAIN and STOP
2.3 compute $state_3$ from $state_2$ using $input_2$
2.4 transfer agent code to next host(s) if needed
2.5 add $sign_1(state_2)$, $sign_2(state_3)$ to message m
2.6 add $input_2$, $state_2$, $state_3$ to message m
2.7 send m,$sign_2$(m) to next host

Bild 28: Protokoll des zweiten Wirts

Falls die Signatur verifiziert werden konnten, führt der zweite Wirt den Agenten unter Berücksichtigung einer eventuellen Eingabe aus (diese Eingabe kann aus mehreren Teilen bestehen und von mehreren Partnern kommen, siehe dazu auch die Diskussion dieses Aspektes in Kapitel 7.1). Der durch die Ausführung entstehende, bzw. *resultierende* Zustand des Agenten wird dann durch den Wirt signiert. Diese Signatur, der resultierende Zustand, der Ausgangszustand sowie dessen Signatur durch den ersten Wirt und schließlich die Eingabe werden, signiert durch den zweiten Wirt, an den dritten übermittelt (siehe Bild 28). Falls ein Codetransfer notwendig ist, wird auch dieser durchgeführt.

3.1 get message from previous host and check signature
3.2 if not true, COMPLAIN and STOP
3.3 get $state_{i+1}$, $sign_i(state_{i+1})$ from message
3.4 check $sign_i(state_{i+1})$
3.5 if not true, COMPLAIN and STOP
3.6 get $input_i$, $state_i$, $sign_{i-1}(state_i)$ from message
3.7 check $sign_{i-1}(state_i)$
3.8 if not true, COMPLAIN and STOP
3.9 compute $state_{i+1}$ from $state_i$ using $input_i$
3.10 check if $state_{i+1}$ from line 3.9 and $state_{i+1}$ received from predecessor differ
3.11 if true, COMPLAIN and STOP
3.12 compute $state_{i+2}$ from $state_{i+1}$ using $input_{i+1}$
3.13 transfer agent code to next host(s) if needed
3.14 add $sign_i(state_{i+1})$, $sign_{i+1}(state_{i+2})$ to message m
3.15 add $input_{i+1}$, $state_{i+1}$, $state_{i+2}$ to message m
3.16 send m,$sign_{i+1}$(m) to next host

Bild 29: Protokoll für den Wirt i+1

Vom dritten bis zum vorletzten Knoten wird das Protokoll verwendet, das in Bild 29 beschrieben wird. Zuerst wird die Signatur der Gesamtnachricht und die des jetzigen Ausgangszustands ($state_{i+1}$) verifiziert. Dieses Vorgehen erscheint auf den ersten Blick überflüssig zu sein, da der letzte Wirt ja bereits diesen Zustand im Rahmen der Gesamtnachricht signiert hat. Der Grund für dieses Vorgehen liegt jedoch darin begründet, dass nur die Signatur des Ausgangszustandes an den nächsten Wirt gesendet werden muss, und dass die Verifikation der Gesamtsignatur auch immer den Transfer aller signierten Inhalte bedingt. Falls eine der beiden Signaturen nicht verifiziert werden kann, wird der erste Wirt benachrichtigt.

Zusätzlich zur Verifikation dieser zwei Signaturen wird jetzt auch noch die Signatur des vorherigen Zustands, $state_i$, verifiziert. Diese Signatur wurde auf dem vorletzten Wirt erzeugt, und vom letzten Wirt durchgereicht, nachdem dieser die Signatur ebenfalls verifiziert hat. Dadurch kann der aktuelle Wirt nach Zeile 3.8 in Bild 29 vertrauenswürdig sein, dass dieser Zustand durch Berechnung auf dem vorletzten Wirt erzeugt wurde, und dass der letzte Wirt glaubt, dass dieser Zustand korrekt ist. Dies kann entweder deshalb geschehen sein, weil der vorletzte Wirt vertrauenswürdig ist oder weil der letzte Wirt die Ausführung nachgerechnet hat (wenn der vorletzte Wirt nicht-vertrauenswürdig war). Dieser Glauben wird dadurch dokumentiert, dass der letzte Wirt die Signatur weitergereicht, und dies durch Signierung der letzten Gesamtnachricht bestätigt hat.

Der aktuelle Wirt prüft dann die Berechnung des vorherigen Wirts durch erneute Ausführung des Agenten unter Benutzung der Eingabe $input_i$, und durch Vergleich des dergestalt resultierenden Zustands mit dem vom letzten Wirt übermittelten Zustand $state_{i+1}$. Wenn diese Zustände differieren, hat der letzte Wirt den Agenten angegriffen. In diesem Fall "beschwert" sich der aktuelle Wirt, d.h. er schickt den Agenten und alle anderen Daten zum Eigentümer des Agenten. Dadurch wird der Eigentümer in die Lage versetzt, diesen Fall zu prüfen und gegebenenfalls juristische oder sonstige Schritte einzuleiten. Da der Eigentümer in der Lage ist, sowohl den vom letzten Wirt signierten Zustand, als auch einen Referenzzustand zu präsentieren (den er, wenn er dem aktuellen Wirt nicht vertrauen will, selbst noch einmal erzeugt), kann ein Angriff nicht nur bewiesen werden, sondern es können auch die Folgen aufgezeigt werden.

Falls alle Prüfungen jedoch erfolgreich durchgeführt werden konnten, führt der aktuelle Wirt den Agenten aus, d.h. errechnet den resultierenden Zustand $state_{i+2}$ aus dem Ausgangszustand $state_{i+1}$ und der Eingabe $input_{i+1}$. Danach wird wieder Code, falls benötigt, auf den nächsten Wirt übertragen. Schließlich signiert der aktuelle Wirt den resultierenden Zustand und der Gesamtnachricht wird die Signatur des letzten Wirts über den Startzustand beigefügt, so dass der nächste Wirt wieder die Integrität dieses Zustandes überprüfen kann, bevor er die Ausführung auf dem aktu-

ellen Wirt nachrechnet. Die Gesamtnachricht wird noch durch $input_{i+1}$, $state_{i+1}$, und $state_{i+2}$ erweitert, vom aktuellen Wirt signiert, und an den nächsten Wirt gesendet.

4.1 get message from previous host and check signature
4.2 if not true, COMPLAIN and STOP
4.3 get $state_{i+1}$, $sign_i(state_{i+1})$ from message
4.4 check $sign_i(state_{i+1})$
4.5 if not true, COMPLAIN and STOP
4.6 get $input_i$, $state_i$, $sign_{i-1}(state_i)$ from message
4.7 check $sign_{i-1}(state_i)$
4.8 if not true, COMPLAIN and STOP
4.9 compute $state_{i+1}$ from $state_i$ using $input_i$
4.10 check if $state_{i+1}$ from line 4.9 and $state_{i+1}$ received from predecessor differ
4.11 if true, COMPLAIN and STOP
4.12 compute $state_{i+2}$ from $state_{i+1}$ using $input_{i+1}$

Bild 30: Protokoll des letzten Wirts i + 1

Der letzte Wirt schließlich (der ebenfalls als vertrauenswürdig angenommen wurde) muss nur noch die Berechnungen seines Vorgängers analog zum letzten Protokoll durchführen und eine eventuelle letzte Ausführung des Agenten vornehmen (siehe Bild 30).

Optimiertes Protokoll

Bei Verwendung des Basis-Protokolls werden nicht nur Ausführungen auf nicht-vertrauenswürdigen, sondern auch auf vertrauenswürdigen Wirten zwischen dem ersten und dem letzten Wirt geprüft. Da in der Realität (im Gegensatz zum Beispiel) solche vertrauenswürdigen Wirten vorkommen können, und die Prüfungen von

Ausführungen auf vertrauenswürdigen Wirten unnötig sind, kann eine optimierte Protokollvariante entwickelt werden, die solche unnötigen Prüfungen weglässt.

5.1 get message from previous host and check signature
5.2 if not true, COMPLAIN and STOP
5.3 get $state_{i+1}$, $sign_i(state_{i+1})$ from message
5.4 check $sign_i(state_{i+1})$
5.5 if not true, COMPLAIN and STOP
5.6 if (predecessor is untrusted) then
5.7 get $input_i$, $state_i$, $sign_{i-1}(state_i)$ from message
5.8 check $sign_{i-1}(state_i)$
5.9 if not true, COMPLAIN and STOP
5.10 compute $state_{i+1}$ from $state_i$ using $input_i$
5.11 check if $state_{i+1}$ from line 5.10 and $state_{i+1}$ received from predecessor differ
5.12 if true, COMPLAIN and STOP
5.13 compute $state_{i+2}$ from $state_{i+1}$ using $input_{i+1}$
5.14 transfer agent code to next host(s) if needed
5.15 add $sign_i(state_{i+1})$, $sign_{i+1}(state_{i+2})$ to message m
5.16 add $input_{i+1}$, $state_{i+1}$, $state_{i+2}$ to message m
5.17 send m,$sign_{i+1}$(m) to next host

Bild 31: Protokoll für einen nicht-vertrauenswürdigen Wirt i + 1

Das Protokoll für den ersten Wirt bleibt auch in der optimierten Variante dasselbe. Danach wird aber in Abhängigkeit davon, ob der Agenten einen Wirt für vertrauenswürdig oder nicht-vertrauenswürdig hält, ein anderes Protokoll verwendet. Aufgrund der Annahme, dass es dem Agenten sicher möglich ist, dieses festzustellen, kann jeder Wirt diese Entscheidung treffen. Ein nicht-vertrauenswürdiger Wirt kann natürlich bewusst eine falsche Entscheidung treffen. Wie wir gleich sehen werden, wird dieses Vorgehen aber durch den nächsten Wirt aufgedeckt werden.

Bild 31 beschreibt das Protokoll für einen *nicht-vertrauenswürdigen* Wirt i+1. Im Vergleich zum Protokoll in Bild 30 wird der Prüfteil (Zeilen 5.7 bis 5.12) nur dann ausgeführt, wenn der vorherige Wirt nicht-vertrauenswürdig ist. Diese Entscheidung ist aufgrund der Annahme über die Existenz der Entscheidungsfunktion möglich. Hat der Vorgänger das falsche Protokoll ausgeführt, fehlen die in Zeile 5.7 benötigten Nachrichtenelemente, und die Prüfung in Zeile 5.8 schlägt fehl.

Bild 32 zeigt das Protokoll für einen *vertrauenswürdigen* Knoten i+1. Verglichen mit dem Protokoll für einen nicht-vertrauenswürdigen Knoten, fehlt hier nur der Transfer von $sign_i(state_{i+1})$, $input_{i+1}$, and $state_{i+1}$ auf den nächsten Wirt. Dies sind

genau die Elemente, die für eine Prüfung auf einem nachfolgenden Wirt benötigt werden würden. Da der Wirt vertrauenswürdig ist, muss aber keine Prüfung durchgeführt werden. Auch hier gilt das im letzten Paragraf Gesagte bezüglich der Feststellbarkeit, ob der Vorgänger vertrauenswürdig oder nicht-vertrauenswürdig ist.

6.1 get message from previous host and check signature
6.2 if not true, COMPLAIN and STOP
6.3 get $state_{i+1}$, $sign_i(state_{i+1})$ from message
6.4 check $sign_i(state_{i+1})$
6.5 if not true, COMPLAIN and STOP
6.6 if (predecessor is untrusted) then
6.7 get $input_i$, $state_i$, $sign_{i-1}(state_i)$ from message
6.8 check $sign_{i-1}(state_i)$
6.9 if not true, COMPLAIN and STOP
6.10 compute $state_{i+1}$ from $state_i$ using $input_i$
6.11 check if $state_{i+1}$ from line 6.10 and $state_{i+1}$ received from predecessor differ
6.12 if true, COMPLAIN and STOP
6.13 compute $state_{i+2}$ from $state_{i+1}$ using $input_{i+1}$
6.14 transfer agent code to next host(s) if needed
6.15 add $sign_{i+1}(state_{i+2})$ to message m
6.16 add $state_{i+2}$ to message m
6.17 send m,$sign_{i+1}$(m) to next host

Bild 32: Protokoll für einen vertrauenswürdigen Wirt i + 1

Am Ende muss das Protokoll für den letzten Knoten nur die Ausführungen von nicht-vertrauenswürdigen Wirten prüfen. Es besteht aus dem Protokoll für einen vertrauenswürdigen Knoten i+1 ohne die letzten vier Zeilen (siehe Bild 32).

7.8.3 Ermittlung vertrauenswürdiger Wirte

In diesem Abschnitt wollen wir zeigen, dass die Annahme erfüllt werden kann, dass auf sichere Art und Weise ermittelt werden kann, ob ein bestimmter Agent einen Wirt für vertrauenswürdig oder nicht-vertrauenswürdig hält.

Die Frage, ob ein bestimmter Wirt für einen bestimmten Agenten als vertrauenswürdig oder nicht-vertrauenswürdig eingestuft wird, muss im Protokoll an zwei Stellen beantwortet werden. Ein Wirt selbst muss wissen, wie er eingestuft wird, um die richtige Protokollvariante auszuführen und ein prüfender Wirt muss die Einstufung seines Vorgängers kennen, um die richtigen Prüfungen durchzuführen. Wie bereits ausgeführt, nützt es im ersten Fall dem Wirt nicht, zu betrügen. Würde ein

nicht-vertrauenswürdiger Wirt die Protokollvariante für einen vertrauenswürdigen Wirt benutzen, würde dies auf dem nächsten Wirt festgestellt werden, da dann Nachrichtenelemente fehlen würden. Um nicht eines Angriffs bezichtigt zu werden haben daher alle Wirte ein Interesse daran, die richtige Protokollvariante zu benutzen. Im zweiten Fall hat der Wirt ebenfalls ein Interesse an der richtigen Antwort, da sie sich auf den vorherigen Wirt bezieht, und wir zunächst annehmen, dass es keine gemeinsamen Angriffe von zwei oder mehr Wirten gibt (diese Einschränkung wird in Kapitel 7.8.7 aufgehoben). Insgesamt lässt sich also sagen, dass wenn eine solche Einstufung abgefragt wird, der jeweils aktuelle Wirt ein Interesse daran hat, die korrekte Antwort zu bekommen, und diese Ermittlung nicht beeinträchtigen wird.

Für ein solches Ermittlungsverfahren gibt es zwei Möglichkeiten. Entweder es wird ein vertrauenswürdiger Wirt (z.B. der Heimatwirt) um eine solche Information gebeten oder der Agent entscheidet selbst (und fragt einen vertrauenswürdigen Wirt nur in wenigen Fällen). Im ersten Fall ist eine entfernte Kommunikation vom aktuellen Wirt zum vertrauenswürdigen Wirt notwendig (die auf jedem Wirt entstehenden zwei Fragen können gleichzeitig gestellt werden). Dies ist ein Nachteil, weil es der Idee der Asynchronizität von mobilen Agenten widerspricht, gleichwohl ist dieses Verfahren sicher, wenn es über einen sicheren Kanal abgewickelt wird. Im zweiten Fall führt der Agent Zertifikate einiger der Wirte mit sich, von denen er weiß, dass er sie eventuell besuchen will. Diese Zertifikate umfassen den Namen eines oder mehrerer Wirte sowie deren Einschätzung (vertrauenswürdig bzw. nicht-vertrauenswürdig) und sind vom Eigentümer des Agenten unterschrieben. Falls der Agent keine anderen Wirte besucht, reicht ein solches Zertifikat aus, um alle Anfragen zu beantworten, da ein signiertes Zertifikat nicht unerkannt manipuliert werden kann. In diesem Fall gibt es auch keinen Mehraufwand, da die konstanten Teile eines Agenten in jedem Fall vom Eigentümer unterschrieben werden. Falls der Agent unvorhergesehene Wirte besucht (und diese Möglichkeit will man sich nicht verbauen), stellt sich die Frage, wie entschieden wird, ob ein Wirt vertrauenswürdig oder nicht-vertrauenswürdig ist.

Wenn diese Frage nur vom Heimatwirt beantwortet werden kann, muss dieser in jedem Fall gefragt werden (er antwortet mit einem Zertifikat, das die angefragten Wirte umfasst). Dann scheint es zunächst das Problem zu geben, dass ein böswilliger Wirt Zertifikate löschen kann, um die Migration auf diese Wirte zu verhindern, aber die Zertifikate können ja jederzeit auf den jeweiligen Wirten neu angefordert werden.

Wenn die Frage, ob ein unvorhergesehener Wirt vertrauenswürdig oder nicht-vertrauenswürdig ist, nicht nur vom Heimatwirt, sondern auch vom Agenten beantwortet werden kann, dann kann dies der Agent tun, ohne befürchten zu müssen, vom

aktuellen Wirt angegriffen zu werden (weil, wie bereits festgestellt wurde, der jeweils aktuelle Wirt kein Interesse an diesem Angriff hat). Der Code als konstanter Teil des Agenten kann auch von keinem Wirt manipuliert werden, weil er, wie alle konstanten Teile, vom Eigentümer des Agenten signiert wurde.

7.8.4 Kosten des Protokolls

Da das Protokoll pro Ausführungssitzung arbeitet, also pro Ausführung eines Agenten auf einem Wirt, werden auch die Kosten zunächst pro Sitzung angegeben. Die maximalen Zusatzkosten, die durch das Protokoll entstehen (d.h. wenn maximal viele nicht-vertrauenswürdige Wirte besucht werden) betragen:

- 2 Signaturverifikationen
- 1 Signaturerzeugung
- 1 Ausführung
- 1 Zustandsvergleich
- 1 Transport der Eingabe
- 1 Transport eines Zustands
- 2 Transporte von Signaturen
- der zusätzliche Signierungsaufwand für die signierten Daten, der bei der Signatur der Gesamtnachricht anfällt

Diese Zahlen gelten für den Fall, dass der Code des Agenten konstant über die Lebenszeit des Agenten ist. Wird das Protokoll nicht eingesetzt, betragen die Kosten für einen Agenten: 1 Ausführung, 1 Transport eines Zustands, 1 Transport des Codes. Zusätzlich sollte der zu migrierende Agent in jedem Fall durch den sendenden Wirt signiert, und durch den empfangenden Wirt verifiziert werden. Diese Anforderung resultiert aus dem allgemeinen Sicherheitsbedarf bei zu schützenden mobilen Agenten und kann daher nicht zu den Zusatzkosten des Protokolls gerechnet werden.

Falls angenommen werden kann, dass alle Zustände dieselbe Größe haben, dass die Ausführungen immer dieselbe Zeit benötigen, und dass der Transport des Codes genausoviel Zeit in Anspruch nimmt wie die Erzeugung, Verifikation und der Transport der Signaturen sowie der Vergleich der Zustände, dann verdoppelt das Protokoll grob die Kosten im schlimmsten Fall (das ist für eine Sitzung auf einem nicht-vertrauenswürdigen Wirt der Fall, der auf einen anderen nicht-vertrauenswürdigen Wirt folgt). Da das Protokoll nicht die Migration beeinflusst, gilt diese grobe Verdoppelung auch für die Kosten einer kompletten Ausführung eines Agenten über seiner Lebenszeit. Obige Annahmen gelten natürlich keineswegs in jedem Fall; wie wir in Kapitel 7.8.9 sehen werden, scheint diese grobe Abschätzung zumindest für die gemessenen Fälle im Wesentlichen zuzutreffen.

Im praktischen Einsatz können die Zusatzkosten jedoch kleiner sein. Erstens muss die Ausführung nicht auf Eingaben warten (die das Ergebnis einer länger dauernden Berechnung sein können). Zweitens kann die Prüfung einer Ausführung Ausgabeoperationen auslassen (da eine wirkliche Interaktion nicht notwendig ist). Drittens kann ein Zustand als Differenz zu einem anderen Zustand übertragen werden. Umgekehrt können die Zusatzkosten für das Protokoll jedoch auch höher sein. Wenn die Eingabe sehr groß ist (etwa, weil eine ganze Datenbank durchsucht wird), kann dieser Anteil die anderen überwiegen. In Kapitel 7.8.9 werden die Zusatzkosten für verschiedene Fälle gemessen.

7.8.5 Weitere Optimierungen

Abgesehen von der Ausnutzung des Unterschieds zwischen vertrauenswürdigen und nicht-vertrauenswürdigen Wirten gibt es weitere Möglichkeiten, das Protokoll zu optimieren. Wie erwähnt, besteht eine dieser möglichen Optimierungen darin, einen Zustand als eine Differenz zu einem zweiten Zustand zu transportieren. Da im Protokoll immer zwei aufeinanderfolgende Zustände transportiert werden und es wahrscheinlich ist, dass diese gemeinsame Elemente haben, kann damit eine Reduktion der zu transportierenden Daten erreicht werden. Eine weiterer Optimierungsansatz verringert die transportierte Eingabe. Wie wir gesehen haben, speichert der “Traces”-Ansatz nicht die wirkliche Eingabe, sondern das Resultat der Anweisung, die diese Eingabe bekommt. Obwohl diese Methode aus Sicherheitsaspekten nicht in jedem Fall wünschenswert erscheint, ist sie doch unter Umständen nützlich, um die transportierten Eingabedaten zu verringern. Wenn wir uns eine Anweisung vorstellen, die die URLs in einem HTML-Dokument findet, kann man sich vorstellen, dass weniger Daten anfallen, wenn nur die URLs, nicht aber das ganze Dokument transportiert werden müssen. Der umgekehrte Fall, dass eine solche Anweisung mehr Daten produziert, als sie als Eingabe bekommen hat, ist jedoch ebenfalls vorstellbar (wenn diese Anweisung beispielsweise eine Dekomprimierungsfunktion darstellt). Daher könnte sich ein in dieser Richtung optimiertes Protokoll von Fall zu Fall entscheiden, ob die tatsächliche Eingabe oder das Resultat der darauf arbeitenden ersten Anweisung transportiert werden soll.

Weitere Optimierungen können aus dem Umstand resultieren, dass die Ausführungs- und Prüfphasen weder vollkommen sequentiell noch auf einem einzelnen Wirt berechnet werden müssen. Der Grad der möglichen zeitlichen Entkoppelung dieser Phasen hängt dabei von der Frage ab, ob es möglich bzw. wünschenswert ist, einen Agenten auszuführen, ohne zu wissen, ob der vorherige Wirt den Agenten angegriffen hat oder nicht. Zwei Antworten auf diese Frage sind möglich. Wenn die Auswirkungen einer Ausführungssitzung ausgeglichen werden können (entweder durch ein transaktionales Rollback oder eine Kompensierung), kann der Wirt mit

einem “Commit” einer Ausführungssitzung warten, bis die Prüfung erfolgreich durchgeführt wurde. Wenn die Auswirkungen nicht ausgeglichen werden können, muss der Wirt mit der Ausführung zumindest der ersten Anweisung warten, die nicht ausgeglichen werden kann, bis die Prüfung abgeschlossen ist.

Schließlich kann die Wiederausführung unter Umständen teilweise durch schnellere Prüfungsverfahren ersetzt werden. Techniken wie Invarianten und Zustandsprüfung mögen vielleicht nicht generell alle möglichen Angriffe entdecken; sie können aber in einigen Anwendungsfällen durchaus ausreichend sein. Das Problem dieses Optimierungsverfahrens ist, dass nicht bekannt ist, ob die teilweise Ersetzung eines Nachrechenverfahrens durch effizientere Verfahren automatisch erfolgen kann. Um die die Prüfung zu optimieren, könnte es aber ein erster Ansatz sein, den Programmierer entscheiden zu lassen, wie die Prüfung genau zu erfolgen hat, z.B. durch die Unterstützung von Rückrufprozeduren, wie in Kapitel Kapitel 7.7 vorgestellt.

7.8.6 Diskussion des Protokolls

Das in diesem Kapitel beschriebene Protokoll erfüllt die Anforderungen, die wir eingangs an ein neues Verfahren gestellt haben (Prüfung in jedem Fall, nach jeder Sitzung, höherer Grad der Entdeckung von Angriffen, kein hoher Aufwand). Im Vergleich zum “Traces”-Ansatz ergeben sich aber zwei Nachteile.

Ein Nachteil besteht darin, dass eine Eingabe nicht vor den Prüfwirt geheimgehalten werden kann. Das kann vor allem dann ein Problem sein, wenn es keine zusätzlichen Verfahren gibt, Daten vor Leseangriffen durch solche Wirte zu schützen, die diese Daten verarbeiten müssen. Solche Verfahren würden es einem Wirt weiterhin erlauben, Daten zur Ausführung zu benutzen, es aber gleichzeitig verhindern, dass der Wirt die Semantik der verarbeiteten Daten kennt. Wenn kein solches Verfahren eingesetzt werden kann (und zurzeit gibt es nur wenige Ansätze, die darauf hinweisen, das solche Verfahren existieren könnten, siehe Kapitel 8.6), können allgemein alle Wirte all diejenigen Eingabedaten lesen, die sie zur Prüfung vom vorherigen Wirt erhalten. Da dies nur bei nicht-vertrauenswürdigen Wirten ein Problem darstellt (vertrauenswürdige Wirte greifen den Agenten ja nicht an), da in der optimierten Version des Protokolls nur ein nicht-vertrauenswürdiger Wirt seinem Nachfolger die Eingabe zur Prüfung übergibt, und da eine Prüfung nur auf dem nächsten Wirt stattfindet, bezieht sich dieses Problem maximal auf die Eingabe an den Agenten, die auf dem vorherigen Wirt stattfand. Allerdings sind von diesen Daten auch ohne Verwendung des Protokolls einige für den nächsten Wirt lesbar, diejenigen nämlich, die ohnehin zum darauffolgenden Wirt transportiert werden. Wenn es jedoch Verfahren zum Schutz vor Leseangriffen gibt, kann eine “geschützte Form” der Eingabe für die Prüfung verwendet werden (im Fall von

[ST98] beispielsweise muss diese dann vor dem Transport speziell “verschlüsselt” werden).

Der schwerwiegendere Nachteil ist das Problem, dass Angriffe von zwei oder mehr aufeinanderfolgenden Wirten, die zusammenarbeiten, nicht entdeckt werden können (es reicht nicht aus, dass zwei beliebige Wirte kollaborieren). Wenn der zweite Wirt einen resultierenden Zustand signiert, der aus einem Angriff des ersten Wirts entsteht, kann der dritte Wirt diesen Angriff nicht entdecken. Um diese Kollaborationsangriffe zu verhindern kann das Protokoll aber erweitert werden (siehe nächster Abschnitt).

7.8.7 Tolerierung von Kollaborationsangriffen

Wenn man mehr als einen Wirt für die Prüfung verwendet, kann das Protokoll so erweitert werden, dass n böswillige, kollaborierende, aufeinanderfolgende Wirte toleriert werden können. Zu diesem Zweck wird jede Ausführungssitzung durch n Prüfungen auf anderen Wirten verifiziert. Das Vorgehen folgt dabei dem Verfahren, dass beim “Server replication”-Ansatz benutzt wird (siehe Kapitel 7.4). Der Unterschied liegt zum einen darin, dass nicht die Ausführung repliziert wird, sondern die Prüfung, und darin, dass nicht $2 \cdot n$ Wirte pro Sitzung benötigt werden, sondern nur $n + 1$. Das liegt daran, dass es zur Entdeckung eines Angriffs nicht wichtig ist, zu wissen, welcher Zustand der korrekte ist; solange auch nur ein Wirt zu einem anderen Ergebnis kommt, kann ein Angriff entdeckt werden. Nach einer solchen Entdekkung kann dann der Agenteneigentümer nachrechnen, welche Partei Recht hatte.

7.8.8 Einbeziehung von Ausgaben des Agenten

Das vorgestellte Protokoll (wie auch die existierenden Ansätze aus Kapitel 7.4) bezieht die Ausgaben eines mobilen Agenten nicht in die Prüfung der Ausführung eines mobilen Agenten ein. Der Grund hierfür liegt darin zu suchen, dass es nicht einfach ist, die Ausgaben eines mobilen Agenten zu prüfen, da diese sich nicht unbedingt in Änderungen des Zustands des Agenten widerspiegeln (dies ist nur bei Interaktionen mit externen Partnern der Fall, bei denen sich die Nachrichten des Partners in Abhängigkeit von den Nachrichten des mobilen Agenten ändern).

Mit einem Verfahren wie dem in Kapitel 7.6.3 vorgestellten Satellitenkonzept ist es möglich, die Kommunikation eines Agenten mit seiner Außenwelt (abgesehen von der Kommunikation mit seinem Wirt) aufzuzeichnen. Daher ist es auch einfach möglich, auch die Ausgaben des Agenten in eine Prüfung miteinzubeziehen. Dazu könnte man entweder die Ausgaben des Agenten während einer Ausführungssitzung als Paket an den prüfenden Wirt schicken, oder aber den Agenten während der Prüfung in einem Prüfmodus erneut mit dem Satelliten kommunizieren lassen, der dann die Ausgaben nicht an externe Partner weitergibt, sondern diese mit den tat-

sächlich stattgefundenen Ausgaben vergleicht. Im Falle einer Diskrepanz kann dann der Satellit diese Tatsache an den Prüfwirt melden. Welche dieser Möglichkeiten (Versand der Ausgaben oder Prüfung auf dem Satellit) dann benutzt wird, könnte sogar dynamisch mittels einer Berechnung der zu erwartenden Kommunikationskosten (wie sie z.B. in [SS97] diskutiert wird) erfolgen.

Die Benutzung des Satellitenkonzepts hätte sogar noch einen weiteren Vorteil: Da auch die externen Eingaben an einen Agenten als Teil der Kommunikation erfasst werden, könnte der Wirt diese Eingaben nicht mehr vor einer Prüfung manipulieren.

7.8.9 Messungen

Um die Kostenschätzungen aus Kapitel 7.8.4 zu evaluieren, wurde das Protokoll prototypisch für einen generischen mobilen Agenten implementiert. Der Agent migriert entlang eines Pfades von drei Wirten, wobei der erste und der letzte vertrauenswürdig sind, der zweite dagegen nicht-vertrauenswürdig. Eine Ausführungssitzung eines Agenten kann durch zwei Werte charakterisiert werden. Der erste Wert, “Zyklen”, steht für die Anzahl an Summationen von 1000 Integer-Werten. Diese Summation emuliert die Berechnungsteile eines Agenten. In den Messungen wurde ein “Zyklen”-Wert von 1 bzw. 10000 benutzt, um die Fälle “keine Berechnungen” bzw. “viele Berechnungen” zu simulieren. Der zweite Parameter bestimmt die Anzahl an Eingabeelementen für die Ausführung des Agenten. Jedes dieser Elemente besteht dabei aus einem String der Länge 10 Zeichen. In den Messungen wurden entweder 1 oder 100 solche Eingabeelemente benutzt. Da diese Werte vier verschiedene Möglichkeiten ausmachen, wurden vier Agenten erstellt, die jeweils eine dieser Wertkombinationen benutzten. Diese Agenten wurden auf zwei verschiedene Weisen ausgeführt: ungeschützt (also ohne das Protokoll zu benutzen, aber als Ganzes signiert und verifiziert), und geschützt durch das Protokoll.

Die Messungen wurden für das Mobile-Agenten-System Mole [BHR98] implementiert, das Java als Programmiersprache benutzt. Als Sicherheitsbibliothek wurde IAIK-JCE 2.0 [IAI99] verwendet, die eine reine Java-Implementierung verschiedener kryptografischer Algorithmen anbietet. Digitale Signaturen wurden mit dem DSA-Verfahren dieser Bibliothek erstellt, wobei eine Schlüssellänge von 512 Bits benutzt wurde.

	Signatur	Zyklen	Rest	**Gesamt**
1 Eingabe, 1 Zyklus	209	2	93	**304**
100 Eingaben, 1 Zyklus	409	3	153	**564**
1 Eingabe, 10000 Zyklen	217	27158	93	**27468**
100 Eingaben, 10000 Zyklen	400	27235	155	**27789**

Tabelle 2: Gemessene Zeiten für ungeschützte Agenten in [ms]

	Signatur	Zyklen	Rest	**Gesamt**
1 Eingabe, 1 Zyklus	237 (1.1)	3 (1.7)	345 (3.7)	**584 (1.9)**
100 Eingaben, 1 Zyklus	560 (1.4)	4 (1.5)	670 (4.4)	**1234 (2.2)**
1 Eingabe, 10000 Zyklen	235 (1.1)	36353 (1.3)	341 (3.7)	**36929 (1.3)**
100 Eingaben, 10000 Zyklen	472 (1.2)	36272 (1.3)	1983 (12.8)	**38727 (1.4)**

Tabelle 3: Gemessene Zeiten für geschützte Agenten in [ms]

Tabelle 2 zeigt die gemessenen Zeiten für die vier ungeschützten Agenten. Tabelle 3 zeigt die entsprechenden Zeiten für die geschützten Agenten. Die Zahlen in Klammern in Tabelle 3 geben an, um welchen Faktor diese Zeiten von den Zeiten in Tabelle 2 abweichen. Die letzte Spalte zeigt in beiden Tabellen die Gesamtlaufzeiten an, d.h. die Zeit, die der Agent benötigte, um vom Start auf dem ersten Knoten zu dem Ende auf dem letzten Knoten zu kommen. Die Zeiten in der “Signatur”-Spalte bezeichnen die Zeiten, die gebraucht wurden, um die Signatur der Gesamtnachricht zu erstellen und zu verifizieren. Die “Zyklen”-Spalte bezeichnet die Zeit,

die benötigt wurde, um die Integer-Werte zu summieren. Die “Rest”-Spalte enthält die Zeiten für alle anderen Aktionen des Agenten.

In der für die Messungen benutzten Konfiguration führt ein ungeschützter Agent seine Hauptprozedur dreimal aus (wegen der drei besuchten Wirten), ein geschützter Agent hingegen viermal, da eine Wiederausführung wegen der Prüfung benötigt wird. Daher liegen die Faktoren der “Zyklen”-Kolumne in Tabelle 3 um den Wert 1,3. Die Werte in der “Signatur”-Kolumne ändern sich ebenfalls nur gering, wenn das Protokoll benutzt wird, da der Signierungsprozess nur mehr Daten verarbeiten muss. In der “Rest”-Kolumne muss das Protokoll solche Dinge tun wie das Vergleichen, Signieren und Verifizieren von einzelnen Zuständen. Daher ist dieser Wert für einen geschützten Agenten wesentlich höher (um einen Faktor von ca. 4).

Bei den Gesamtlaufzeiten liegen die Faktoren um die Werte 1,3 bis 1.4 für die zwei Agenten, die viel berechnen (wobei die Summation über 95% der Gesamtlaufzeit ausmacht) bis hin zu den Faktoren 1,9 und 2,2 für die zwei Agenten ohne große Berechnungen. Da es für die Kosten des Protokolls allein auf die Einflussfaktoren Länge der Ausführung und Gesamtgröße der Eingaben ankommt, und diese beiden Faktoren durch die generischen Agenten modelliert wurden, lassen sich diese Ergebnisse auch auf reale mobile Agenten anwenden. Insgesamt bestätigen diese Messungen also die Vermutung, das die Anwendung des Protokolls bei einer Migrationsfolge, die fast nur nicht-vertrauenswürdige Wirte enthält, zu einem Mehraufwand von ungefähr einem Faktor 2 führt (da die Hauptprozedur dann doppelt so häufig aufgerufen wird).

Da in den Messungen nur lokale Migrationen benutzt wurden (also solche innerhalb eines Rechners), fiel kein Code-Transfer bei der Migration an. Falls ein solcher Transfer notwendig ist, würden die Faktoren etwas sinken, da dieser für geschützte und ungeschützte Agenten die gleiche Zeit benötigt. Die gemessenen Zeiten wurden ohne Just-In-Time-Compiler ermittelt. Durch die Benutzung eines solchen reduzieren sich die Zeiten auf 60% für die ersten beiden Agenten und auf 50% für die letzten beiden Agenten.

Teile diesen Kapitels wurden [Hoh99] und [Hoh00] entnommen.

8 Blackbox-Schutz

Nachdem bis jetzt nur Teilschutzansätze behandelt wurden, also Verfahren, die einen mobilen Agenten nur vor bestimmten Ansätzen schützen, betrachten wir nun Verfahren, deren Anliegen es ist, den Agenten möglichst vor allen Angriffen zu schützen. Nachdem organisatorische Ansätze bereits in Kapitel 6.3.1 untersucht wurden, und von den technischen Ansätzen in Kapitel 6.3.2 bereits diejenigen vorgestellt wurden, die spezielle Hardware verwenden, wenden wir uns nun genauer den Verfahren zu, die einen technischen Gesamtschutz allein durch Softwaremaßnahmen zu erreichen suchen.

Nachdem dieses Problem zu Anfang als unlösbar galt*, entwickelten sich ab etwa 1997 zwei Lösungsrichtungen, die unter dem Stichwort “Blackbox-Schutz” zusammengefasst werden können. Teile dieses Kapitels wurden [Hoh97] und [Hoh98b] entnommen.

8.1 Die Idee

Die grundlegende Idee des Blackbox-Ansatzes ist es, einen beliebigen Ursprungsagenten zu nehmen und durch eine Konvertierung einen äquivalenten Agenten (d.h. einen Agenten, der dieselbe Funktionalität bietet) zu erzeugen, dessen Struktur nicht mehr dem Ursprungsagenten entspricht, der aber weiter ausführbar ist. Die Konvertierung wird dabei durch einen Parameter konfiguriert (siehe Bild 33), so dass ein Angreifer nicht in kurzer Zeit alle möglichen Konvertierungen erzeugen und so einen konvertierten Agenten, die *Blackbox*, erkennen bzw. zurückkonvertie-

* Die Autoren von [FGS96] merken an, dass es unmöglich sei, Agenten vor Leseangriffen durch den Wirt zu schützen und empfehlen daher, den Agenten keine geheimen Daten transportieren zu lassen. [CHK97] finden es unmöglich, Agentenangriffe zu verhindern, es sei denn man benutzt sichere Hardware, und [Ord96] behauptet, dass es nicht möglich sei, die Daten eines Agenten zu schützen, da ein Wirt diese Daten ändern und den modifizierten Agenten weiterleiten kann.

ren kann.

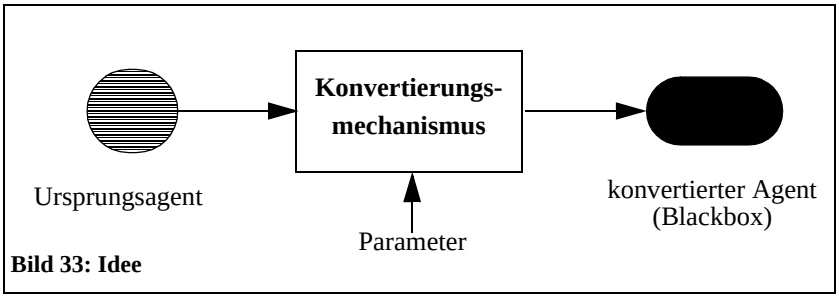


Bild 33: Idee

Als *Blackbox* wird ein Agent dann bezeichnet, wenn es nicht möglich ist, dass die Datenelemente und Codeteile eines Ursprungsagenten in der Blackbox erkannt werden können. Damit geht einher, dass der Angreifer die Werte der Datenelemente des Ursprungsagenten nicht bestimmen kann, und dass diese Datenelemente und Codeteile auch nicht temporär gezielt modifiziert werden können (eine dauerhafte Modifikation wird ja bereits durch eine Signierung des Agenten bzw. seines Codes verhindert). Weiter ist damit sichergestellt, dass der Code nicht, entgegen der Spezifikation, in einer Weise ausgeführt werden kann, so dass der Angreifer einen bestimmten Effekt erzielt.

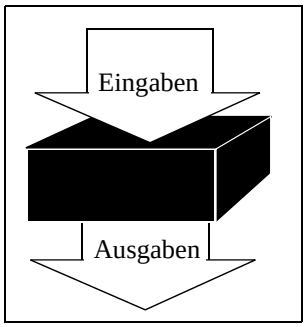


Bild 34: Blackbox

Ist ein Agent eine Blackbox, so kann ein Angreifer im Wesentlichen nur noch Eingaben in und Ausgaben aus der Blackbox beobachten (Bild 34). Er kann zwar noch Zustandsänderungen des Agenten wahrnehmen, diese aber nicht mehr bestimmten Änderungen einzelner Datenelemente des Ursprungsagenten zuordnen. Daher ist eine Blackbox ein Agent, der gegenüber dem ausführenden Wirt einen Teil seiner Autonomie wiedererlangt hat und der diese Autonomie nutzen kann, um andere Angriffe zu verhindern.

Mit dieser Definition lässt sich auch erklären, warum der im Beispiel zum Angriffsmodell in Kapitel 5.6.5 vorgestellte (und durch ein Angriffsprogramm gebrochene) Schutzmechanismus keinen Blackboxschutz erzeugt: die Datenelemente des Ursprungsagenten lassen sich (auf dem Stapel) immer noch erkennen.

8.2 Eigenschaften der Blackbox

Durch die Konvertierung in eine Blackbox ist also nach der Definition des Begriffs ein mobiler Agent entstanden, der vor bestimmten Angriffen geschützt ist. Um festzustellen, für welche der in Kapitel 5.3 identifizierten Angriffe dies der Fall ist, muss zunächst festgestellt werden, wie der ursprüngliche und der konvertierte Agent unter dem Gesichtspunkt der möglichen Angriffe zusammenhängen. Das Problem ist, dass in Kapitel 5.3 immer nur von einem Agenten die Rede ist, während wir nun zwischen dem konvertierten Agenten, der dem Angreifer vorliegt und der ursprünglichen Version, die im konvertierten Agenten "versteckt" ist, unterscheiden müssen.

Gegenüber der Außenwelt verhalten sich beide Agenten identisch, d.h. das Interaktionsverhalten ist gleich, auch Systemaufrufe erfolgen bei beiden Agenten in derselben Weise. Damit erscheint auch das Verhalten des Wirts für beide Agenten gleich. Unterschiedlich ist die äußere Form (die ja den Schutz ausmacht). Bezüglich der drei Elemente eines mobilen Agenten bedeutet das für den Code und den Ausführungszustand, dass sie beliebig unterschiedlich sein können, solange das äußere Interaktionsverhalten gleich bleibt. Für den Datenzustand gilt eine Art Inklusionsbeziehung. Für jedes Datenelement aus dem Datenzustand des Ursprungsagenten gilt, dass eine (Agenteninstanz- und Datenelement-spezifische) Funktion existiert, die dieses Datenelement aus dem Datenzustand der Blackbox dieses Ursprungsagenten erzeugen kann. Natürlich kann eine Blackbox auch noch weitere Daten enthalten, aber diese tragen nicht zum Interaktionsverhalten bei (das ja gleich bleibt). Da sich eine Blackbox von ihrem Ursprungsagenten unterscheidet (sonst wäre ein Schutz nicht zu gewährleisten) erfolgt auch die Ausführung beider Agenten durch den Wirt in verschiedener Weise.

Generell verhindert werden können die Angriffsmethoden "Dauerhafte Modifikation von Code" und "Ausführungsverweigerung". Wie in Kapitel 6.4.2 diskutiert, ist dies im ersten Fall durch Signierung des Codes möglich, im zweiten Fall durch verschiedene Mechanismen wie z.B. modifizierte Fehlertoleranzverfahren. Daher fallen diese beiden Angriffsmethoden in jedem Fall weg. Damit bleiben von den Angriffen aus Kapitel 5.3 9 Angriffsmethoden übrig. Alle diese Angriffsmethoden sind auf den konvertierten Agenten möglich, eben weil der Wirt z.B. jedes Datum im Speicher lesen können muss, wenn er es zur Ausführung benötigt.

Der durch den Blackbox-Schutz erreichte Effekt ist aber nun, dass von den (möglichen) Angriffsmethoden gegen den konvertierten Agenten nicht auch automatisch Angriffsmethoden gegen den Ursprungsagenten ausgehen. Tatsächlich sind nun, bezogen auf den Ursprungsagenten, folgende Angriffsmethoden ausgeschlossen:

Lesen der Agentendaten

Durch die Konvertierung ist die Relation zwischen Datenelementen des Ursprungsagenten und der Blackbox nicht mehr oder nicht mehr einfach nachzuvollziehen. Daher können die Werte der Datenelemente durch den Wirt nicht mehr (einfach) gelesen bzw. kopiert werden.

Zielgerichtete Modifikation von Agentendaten

Aus obigem Grund ist es auch schwer bzw. unmöglich, Datenelemente temporär oder dauerhaft zu modifizieren. Wenn der Angreifer die Semantik der Datenelemente in der Blackbox nicht kennt (weil ihm die Relation zu den Elementen des Ursprungsagenten verborgen bleibt, deren Semantik ihm bekannt ist), dann kann er die Datenelemente auch nicht gezielt ändern.

Lesen des Codes

Wie bei den Daten ist dem Angreifer auch hier die Relation zwischen den Codeelementen der Blackbox und denen des Ursprungsagenten unbekannt. Allerdings kann angenommen werden, dass der Angreifer den Ursprungscode kennt.

Inkorrektes Ausführen von Code

Ohne die Kenntnis der Relation der Codeelemente von Ursprungsagent und Blackbox, ist es unwahrscheinlich, den Code der Blackbox gezielt inkorrekt ausführen zu können.

Lesen der Ausführung des Agenten

Wie bei den Daten sieht der Angreifer zwar z.B. den konkreten dynamischen Kontrollfluss der Blackbox, kann diesem aber keine Semantik im Hinblick auf den Ursprungsagenten zuordnen.

Temporäre Modifikation der Agentenausführung

Wie bereits erwähnt, kann man Angriffsmethoden im Gebiet der “Ausführungsverweigerung“, wie in Kapitel 6.4.2 diskutiert, durch verschiedene Mechanismen wie z.B. modifizierte Fehlertoleranzverfahren verhindern. Für das Gebiet der „Modifikation des Kontrollflusses“ gilt, dass es, wie auch bei den Daten, schwer fällt, den Kontrollfluss gezielt zu modifizieren, wenn die Semantik desselben bezogen auf den Ursprungsagenten unbekannt ist.

Nachdem man von den 9 noch möglichen Angriffsmethoden also die sieben abzieht, die durch die Blackbox-Eigenschaft verhindert werden, bleiben noch drei übrig, die noch nicht abgedeckt sind. Wie wir im nächsten Abschnitt sehen werden, können einige davon durch den Einsatz weiterer Schutzmechanismen verhindert werden.

8.3 Verhinderung weiterer Angriffsmethoden

Von den drei Angriffsmethoden, die sich, aufbauend auf der Blackbox-Eigenschaft, verhindern lassen, können sich einige durch Anwendung bekannter Maßnahmen ausschließen lassen.

Verschleierung der Identität (Maskerade)

Insbesondere für eine Blackbox sind diejenigen Angriffsmethoden ein Problem, bei denen ein Wirt vorgibt, ein anderer zu sein, weil die Blackbox in der Lage ist, vertrauliche Daten zu speichern, und diese erst auf bestimmten Wirten zu offenbaren. Für eine Migration von einem vertrauenswürdigen Wirt aus kann dieses Problem auch ohne Blackbox-Schutz einfach durch eine Verschlüsselung des Gesamtagenten mit dem öffentlichen Schlüssel des Zielwirts gelöst werden, jedoch nicht für eine Migration von einem nicht-vertrauenswürdigen Wirt aus.

Eine einfache 1-Wege-Authentifikation des Wirts durch den Agenten reicht nicht aus, weil sonst zwei verschiedene Arten von Wiederholungsangriffen (engl. replay attacks) gegen die Authentifizierung möglich wären. Die erste Art des Angriffs, der

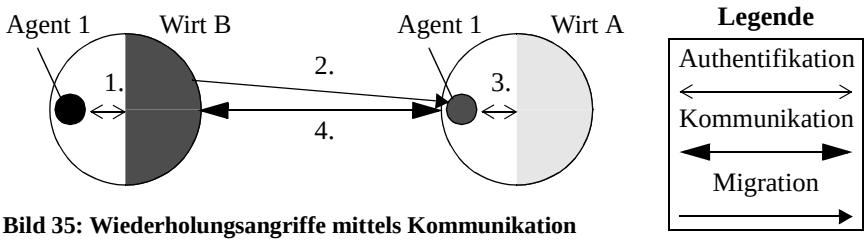


Bild 35: Wiederholungsangriffe mittels Kommunikation

sog. “man in the middle“-Angriff, besteht darin, dass ein sich maskierender Wirt einen speziellen Agenten auf den eigentlichen Zielwirt schicken könnte, um so die Authentifizierung des zu schützenden Agenten mit dem eigentlichen Zielwirt stattfinden lassen zu können (siehe Bild 35). Hier maskiert sich der Wirt B gegenüber dem Agenten 1 als Wirt A. Dazu schickt Wirt B einen Agenten an Wirt A, der sich als Agent 1 ausgibt, in Wirklichkeit jedoch nur als eine Art Zwischenstation fungiert, um es dem angreifenden Wirt zu erlauben, gegenüber dem zu schützenden Agenten als der eigentliche Zielwirt aufzutreten. Wenn nun der richtige Agent 1 eine Authentifikationsanfrage an den Wirt B schickt, etwa ein Datum, das der Wirt mit seinem geheimen Schlüssel verschlüsseln soll, dann schickt Wirt B diese Anfrage an seinen speziellen Agenten auf Wirt A. Der speziellen Agent schickt diese Anfrage (als ob er Agent 1 wäre) an Wirt A. Wirt A beantwortet diese Anfrage nun korrekt, indem er das Datum mit seinem geheimen Schlüssel verschlüsselt und

an den speziellen Agenten zurückgibt. Diese sendet die Antwort von Wirt A an Wirt B, der sie an Agent zurückgibt, ganz so, als ob sie von Wirt B generiert worden sei. Nachdem Agent 1 die Antwort überprüft hat (etwa indem er die Antwort mit dem öffentlichen Schlüssel von Wirt A entschlüsselt und den erhaltenen Wert mit dem Anfragedatum vergleicht), glaubt Agent 1, dass der augenblickliche Wirt Wirt A ist. Die zweite Art des Angriffs, ebenfalls eine Variation des “man in the middle“-

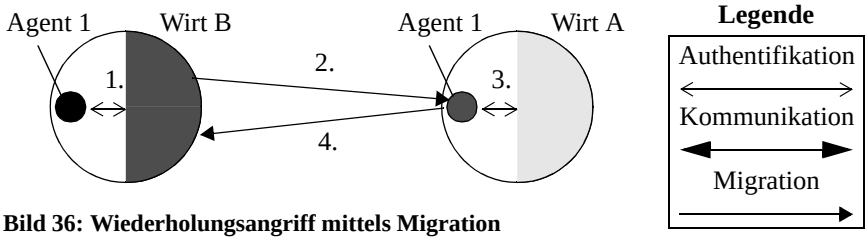


Bild 36: Wiederholungsangriff mittels Migration

Angriffs, besteht darin, dass der angreifende Wirt einen Agenten mit der Authentifikationsanforderung des zu schützenden Agenten auf den eigentlichen Zielwirt schickt, der den Authentifikationsvorgang aufzeichnet und zurückmigriert. Jetzt kann sich der angreifende Wirt mittels der aufgezeichneten Nachricht analog zur ersten Art des Angriffs authentifizieren (siehe Bild 36).

Um sicherzustellen, dass sich ein Wirt nicht als ein anderer maskieren kann, muss daher ein Zwei-Wege-Authentifikationsverfahren benutzt werden, das sicherstellt, dass auch der Agent der ist, der er vorgibt zu sein. Wird jedoch ein 2-Wege-Authentifikationsverfahren wie etwa das des ISO-Authentication-Frameworks [ITU87] verwendet, das auf der Kenntnis von geheimen Schlüsseln beruht, können die „man in the middle“-Angriffe nicht unbedingt verhindert werden. Der zweite Wiederholungsangriff, der Angriff mittels Migration, kann dann nicht verhindert werden, wenn der spezielle Agent den eigentlichen Agenten enthält und diesen für die Authentifizierungsphase aktiviert. Ist dieses Vorgehen nicht möglich, kann der zweite Angriff verhindert werden, weil der Agent wieder zurück- und wieder hinmigrieren müsste, um die Authentifikation zu vollenden. Der erste Angriff ist jedoch möglich, wenn der Agent auch schon während der Authentifizierung kommunizieren dürfte. Der Schutz vor einem solchen Angriff besteht daher darin, dass der Wirt einen Agenten nicht kommunizieren lassen darf, bevor beide Seiten authentifiziert wurden, es also sichergestellt ist, dass wirklich nur der zu authentifizierende Agenten über den geheimen Schlüssel verfügt. Da bei diesen Verfahren eine Identität dadurch verifiziert wird, dass der zu identifizierende Partner über einen geheimen

Schlüssel verfügt, könnte sich ein Wirt dann maskieren, wenn er in Besitz des geheimen Schlüssels eines Agenten gelangen könnte.

Ein 2-Wege-Authentifikationsverfahren, das diese Schwäche nicht aufweist, und auch mehr über den zu authentifizierenden Agenten aussagt, besteht in einem zweistufigen Verfahren nach erfolgter Migration.

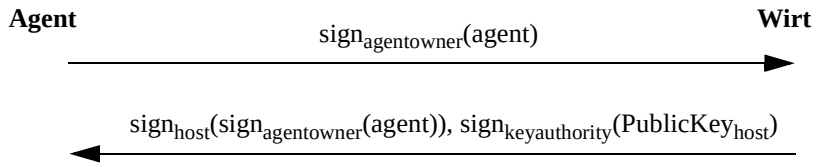


Bild 37: Spezielles 2-Wege-Authentifikationsverfahren

Dieses Verfahren ist in Bild 37 abgebildet. Zu beachten ist, dass die Kommunikation zwischen Agenten und Wirt lokal ist, und nicht abgehört werden kann. In der ersten Phase authentifiziert der Wirt den Agenten durch eine Verifizierung der Agentensignatur. Damit wird sichergestellt, dass der Agent genau der ist, der vom Agenteneigentümer losgeschickt wurde, d.h. z.B. bezüglich des Codes nicht geändert wurde. In Bild 37 ist das durch einen Kommunikationsakt zwischen Agent und Wirt symbolisiert, aber der Wirt kann dieses Datum, das Teil des transportierten Agenten ist, natürlich auch lesen und verifizieren, ohne den Agenten ausführen zu müssen. Wie bereits erwähnt, ist der Agent vom den Agenteneigentümer signiert. Die Verifikation (nicht dargestellt) erfolgt also unter Benutzung des öffentlichen Schlüssels des Agenteneigentümers. In der zweiten Phase verschlüsselt der Wirt diese Signatur (oder einen Hash-Wert der Signatur) mit seinem geheimen Schlüssel und schickt dieses Datum zusammen mit einem Zertifikat des Wirtsschlüssels an den Agenten. Solche Schlüsselzertifikate sind von einer vertrauenswürdigen Instanz (der sog. „key authority“) signiert. Der Agent kennt den öffentlichen Schlüssel dieser Instanz oder kann ihn auf sichere Weise in Erfahrung bringen und kann damit den öffentlichen Schlüssel des Wirts verifizieren. Der Agent entschlüsselt das Datum mithilfe des öffentlichen Schlüssels des Wirts und vergleicht den so erhaltenen Wert mit seiner Signatur (bzw. dem enthaltenen Hash-Wert). Sind beide gleich, kann der Agent sicher sein, dass das Datum von diesem Wirt stammt. Die Nachricht kann auch nicht durch einen Wiederholungsangriff von einem anderen Wirt kommen, weil der Agent, der diese Nachricht erhalten hat, nur die Blackbox selbst sein kann (wegen der Verifizierung der Signatur). Falls die Blackbox also die Nachricht nicht selbst an andere Wirte “verrät”, kann sie kein fremder Agent aufzeichnen.

Abhören der Kommunikation mit anderen

Da ein Agent durch den Blackbox-Schutz vor gezielten Angriffen durch den Wirt vor Veränderung und ungewolltem Informationsverlust geschützt ist, kann dieser Aspekt durch konventionelle Verfahren wie Verschlüsselung dann abgedeckt werden, wenn diese Verfahren nicht auf Geheimnissen beruhen, die der Wirt erzeugt (etwa Zufallszahlen). Für bestehende Public-Key-Verschlüsselungsverfahren, die auf Zertifikaten zur Schlüsselübermittlung beruhen, gilt diese Einschränkung, da Zertifikate nicht durch Wiederholungsangriffe angreifbar sind.

Modifizieren der Kommunikation mit anderen

Wie im letzten Abschnitt gilt auch hier, dass konventionelle Verfahren zur Integritätssicherung, etwa sichere Hash-Verfahren, ohne Probleme benutzt werden können, solange keine Anrufe (engl. challenge) durch den Wirt generiert werden müssen. Dies gilt z.B. für sichere Hash-Verfahren, aber es kann auf höheren Interaktionsebenen (wie einem E-Commerce-Protokoll) vorkommen, dass durch Benutzung von Anrufen sichergestellt werden soll, dass keine Nachricht wiedereingespielt werden kann. In diesem Fall müssen Anrufe, die durch den Wirt erzeugt werden müssten oder von diesem kontrolliert werden können, durch andere Verfahren erzeugt werden.

Blackbox-Tests

Die Verhinderung von Blackbox-Tests ist Gegenstand von Kapitel 9. Das Problem, das sich hier, im Gegensatz zu den anderen drei Angriffen, stellt, ist, dass dieser Angriff bisher kaum außerhalb von Mobile-Agenten-Systemen vorkam, und daher grundsätzlich neue Schutzverfahren entwickelt werden mussten.

8.4 Nicht verhinderbare Angriffsmethoden

Zuletzt bleiben zwei Angriffe übrig, von denen bis jetzt nicht bekannt ist, ob man sie verhindern kann oder nicht. Sollten eines Tages auch diese zwei Angriffe ausschließbar sein, ist es möglich, einen transparenten Schutz eines beliebigen Agenten zu ermöglichen, falls ein solcher Agent in eine Blackbox konvertiert werden kann.

8.4.1 Kenntnis der Rückgabewerte bei Systemfunktionen

Durch die Trennung von ausführender und ausgeführter Partei kann eine Ausführungsplattform bezüglich der Sicherheit nicht mehr unbesehen Dienste für einen mobilen Agenten erbringen, wenn diese Dienste für den Agenten sicherheitsrelevant sind. Ein gutes Beispiel hierfür sind Zufallszahlen oder die Systemzeit, die in einigen Protokollen für Sicherheitszwecke eingesetzt werden, und deren Kenntnis als geheim vorausgesetzt wird. Normalerweise werden solche Daten durch Aufruf

einer Systemfunktion generiert, weil in traditionellen Client-Server-Systemen Programm und Ausführungsumgebung eine organisatorische Einheit bilden. Wenn aber in Mobile-Agenten-Systemen die Wirte, also die Ausführungsumgebungen genau die Angreifer sind, vor denen man sich schützen will, ist ein solches Vorgehen nicht sicher, weil die Wirte die Rückgabewerte nach dem Aufruf der jeweiligen Funktion natürlich kennen. Will man dieses verhindern, müssen diese Rückgabewerte auf anderen, vertrauenswürdigen oder zumindest nicht mit dem aktuellen Wirt kollaborierenden Wirten erzeugt werden, oder sogar durch den Agenten selbst. Falls dieses Vorgehen möglich ist (wie wir im nächsten Abschnitt sehen werden, lassen sich nicht alle Werte auf anderen Wirten erzeugen), ist es dann sicher, wenn die Kommunikation nicht durch den Wirt abgehört werden kann (siehe Kapitel 8.3).

Ein zweites Problem stellen Bibliotheken dar, die zwar keine speziellen Ressourcen des Wirts benutzen (etwa Funktionen, die große Primzahlen bereitstellen), über die aber sicherheitsrelevante Informationen fließen. Wenn ein Agent beispielsweise das RSA-Verfahren benutzt, um Daten vor dem Wirt zu verbergen, dann darf der Wirt auch nicht die Zahlen sehen, die als Schlüssel benutzt werden. Selbst kleinere Teilwerte, die im Verfahren benutzt werden, können den Suchraum für den Angreifer so einschränken, dass es für diesen kein Problem darstellt, den Verschlüsselungsschutz zu brechen. Daher müssen, sofern diese Daten im Klartext durch den Wirt sichtbar sind, die entsprechenden Bibliotheken im Agenten vorhanden sein und dürfen nicht als Teil des Wirts benutzt werden.

8.4.2 Rückgabe falscher Werte bei Systemfunktionen

Neben dem Problem der *Kenntnis* der Rückgabewerte beim Aufruf von Systemfunktionen ist auch das Problem der Rückgabe *falscher* Werte bis jetzt nicht gelöst (tatsächlich gibt es auch kaum Arbeiten, die dieses Problem thematisieren). Zwar kann man zwischen Rückgabewerten unterscheiden, die auch durch andere Wirte und solchen, die nur durch den aktuellen Wirt erbracht werden können, aber letztere sind in der Regel nicht überprüfbar. Damit bleibt nur, sicherheitsrelevante Werte durch sichere, externe Wirte zu beziehen und die anderen durch den aktuellen Wirt.

8.5 Ansätze, Blackboxes zu realisieren

Es gibt zwei Ansätze, Blackboxes zu realisieren. Im folgenden Kapitel werden drei Verfahren beschrieben (“Mobile Cryptography”, “Funktionsverschlüsselung durch Fehlerkorrekturcodes” und “Code Padding”), die auf der Idee der nicht-interaktiven Auswertung von verschlüsselten Funktionen beruhen. In Kapitel 8.7 wird ein neuer Ansatz, der zeitbeschränkte Blackbox-Schutz, beschrieben, der einen wesentlichen Nachteil der ersten drei Ansätze, nämlich die Restriktion der Kommunikation des Agenten, nicht aufweist.

8.6 Nicht-interaktive Auswertung von verschlüsselten Funktionen

Das Problem der direkten Verarbeitung verschlüsselter *Daten* (engl. computing with encrypted data, CED) beschäftigt sich mit der Frage, wie eine Partei A einer Partei B eine Funktion f, die Daten d enthält, schicken kann, die die Partei B auf einer Eingabe x als Parameter dieser Funktion auswertet und das Ergebnis y wieder zurück zu A schicken kann, ohne dass B Kenntnis der Daten d haben kann. Die Daten d sind also in gewisser Weise “verschlüsselt”.

Ein verwandtes Problem ist das der Auswertung von verschlüsselten Funktionen (engl. evaluation of encrypted functions, EEF). Hier sollen nicht nur die Daten d vor der Kenntnis durch B geschützt werden, sondern auch noch die Funktion f. Diese Funktionen können recht einfach sein, z.B. rationale Polynomfunktionen, oder aber Turing-mächtig, z.B. Boolesche Schaltkreise. [AFK89] präsentierten ein Verfahren, mit dem es möglich ist, das EEF-Problem auf das CED-Problem zu reduzieren. Dazu beschrieben sie Boolesche Schaltkreise als Eingaben für einen universellen Booleschen Schaltkreis, der dann als Funktion f dient.

Der Bezug zum Schutz mobiler Agenten vor böswilligen Wirten liegt in dem Umstand begründet, dass durch EEF-Verfahren geschützte Einheiten Blackboxes sind, falls diese Verfahren Turing-mächtige Funktionen zulassen. Auf diesen Umstand wurde (ohne den Begriff Blackbox zu verwenden) das erste Mal durch [ST98] hingewiesen, die das generelle Verfahren zudem so modifizierten, dass A und B nicht mehr während der Ausführung häufig kommunizieren müssen (wie das noch bei [AFK89] der Fall war), sondern die Interaktion dem Mobile-Agenten-Schema genügt, d.h. A die Funktion auf B schickt (Migration) und B das Ergebnis A mitteilt (eine Art Rückmigration). Diese Art eines EEF-Verfahrens wird in [ST98] *nicht-interaktiv* genannt.

Wenn man allerdings die generelle Problemstellung dieser Verfahren mit der typischen Arbeitsweise von mobilen Agenten vergleicht, fällt auf, dass die geschützten Einheiten während ihrer Ausführung nicht kommunizieren. Der Grund hierfür liegt in der Annahme begründet, dass es sich ursprünglich um zu schützende *Funktionen* handelte, die als solche nicht kommunizieren. Wie wir später sehen werden, führt dieses “Sparschwein”-Modell (Daten können nur am Anfang eingebracht werden, nur der Besitzer ist in der Lage, mit seinem Schlüssel wieder Daten zu entnehmen) dann zu Problemen, wenn mobile Agenten dieses generelle Schema durchbrechen und z.B. mit dem Wirt interagieren.

Im Folgenden werden die drei Verfahren beschrieben, die die Idee der nicht-interaktive Auswertung von verschlüsselten Funktionen verfolgen.

8.6.1 Mobile Cryptography

In [ST97] und [ST98] verwiesen Sander und Tschudin zum ersten Mal auf den Zusammenhang zwischen den Ansätzen der Auswertung von verschlüsselten Funktionen und dem Problem des Komplettschutzes mobiler Agenten vor Angriffen durch den Wirt. Aufgrund der Möglichkeit der Reduktion des EEF-Problems auf das CED-Problem beschäftigte sich die Arbeit mit der Frage, wie ein Verfahren zur Lösung des CED-Problems aussehen könnte, das nicht-interaktiv ist, das also keine Kommunikation zwischen A und B während der Ausführung benötigt. Das Verfahren in [AFK89] verlangt ein hohes Maß an Interaktion, und gerade dieser Aspekt widerspricht der häufig geforderten Eigenschaft mobiler Agenten der asynchronen Verarbeitung.

In ihrem Ansatz prägten Sander und Tschudin den Begriff des “verschlüsselten Programms”, d.h. eines Programms, das direkt auf verschlüsselten Daten arbeiten kann ohne dazu die Daten entschlüsseln zu müssen. Dazu soll konzeptuell ein beliebiges Programm einer Transformation unterworfen werden, die die Operationen in andere Operationen verwandelt, die direkt auf den verschlüsselten Daten arbeiten. Statische Daten des Programms werden in eine verschlüsselte Form gebracht. Code und Daten können dann von einer anderen Partei ausgeführt werden, ohne dass diese in der Lage ist, die verarbeiteten Daten zu lesen oder zu manipulieren. Dazu kann die ausführende Partei eine Eingabe x benutzen, die vom Agenten verarbeitet wird. Das Ergebnis dieser Verarbeitung ist ein Resultat y, das ebenfalls verschlüsselt ist (weil die Operationen im Raum der verschlüsselten Daten bleiben). Das Resultat y wird dann schließlich dem Auftraggeber geschickt, der in der Lage ist, y zu entschlüsseln, weil er die Verschlüsselung kennt.

“Verschlüsselung” als Begriff ist hier also nicht mit der klassischen Verschlüsselung gleichzusetzen, die etwa DES oder RSA bietet. Diese Verfahren erlauben es zwar auch, einen Agenten zu verschlüsseln, aber das Ergebnis wäre mit hoher Wahrscheinlichkeit nicht einmal ausführbarer Code, geschweige denn äquivalent zum Ursprungsagenten. “Verschlüsselungs-”funktionen, die stattdessen benötigt werden, müssen anderen Anforderungen genügen. Zum einen müssen die Operationen, durch die das Verschlüsselungsverfahren die Originaloperationen ersetzt, direkt auf den verschlüsselten Daten arbeiten können. Zum anderen muss die Verschlüsselungsfunktion selbst, die Klartextdaten in Schlüsseldaten umwandelt, eine Falltürfunktion sein (sonst könnte ein Angreifer aus der Kenntnis der Verschlüsselungsfunktion die Entschlüsselungsfunktion ableiten).

In [ST98] zeigten die Autoren zwei Verfahren, die CED für rationale Polynomfunktionen über Z/NZ-Ringen ermöglicht.

Die Vorteile dieses Ansatzes sind:

- Die Berechnungskomplexität der Schutzstärke ist beweisbar
- Der Schutz ist zeitlich nicht beschränkt

Die Nachteile dieses Ansatzes sind:

- Für den speziellen Fall der Verwendung der Verschlüsselungsverfahren für rationale Funktionen gibt es noch eine Reihe von möglichen Fällen, in denen die Sicherheit des Verfahrens nicht gegeben ist
- Die Kosten des Verfahrens sind unklar; schon das beim Übergang von CED zu EEF in Frage kommende Verfahren von [AFK89] hat laut [ST98] exponentiellen Platzbedarf
- Zurzeit ist zumindest für diesen speziellen Ansatz noch nicht gezeigt, dass beliebige Programme, also Turing-mächtige Sprachen schützbar sind (das Verfahren von [AFK89] ist zwar anwendbar, aber es verlangt ein hohes Maß an Interaktion, ist also nicht nicht-interaktiv)
- Das Interaktionsmodell des Agenten erlaubt es nicht, dass der Agent während der Ausführung kommuniziert

Die Ausdehnung dieses Ansatzes auf Turing-mächtige Sprachen ist zurzeit Gegenstand der Forschung. Sobald dieses geschehen ist, sind derart geschützte Agenten Blackboxes.

Der vierte Nachteil, die Tatsache, dass eine Interaktion des Agenten nicht vorgesehen ist, wird in diesem Ansatz nicht explizit behandelt. Es ist ohne weiteres denkbar, dass verschlüsselte Daten kommuniziert werden können, ebenso, dass verschlüsselte oder Klartextdaten in Folge einer Interaktion mit anderen durch den Agenten verarbeitet werden, ohne dass dies zu Sicherheitsproblemen führt. Was aber auf dem momentanen Stand als mindestens problematisch erscheint, ist eine Kommunikation des Agenten mit anderen (z.B. dem Wirt), bei der der Agent Klartextdaten ausgibt, die aus den verschlüsselten Daten errechnet werden. Dazu wäre eine Entschlüsselungsfunktion notwendig, die Teil des Agenten sein müsste. Dies wiederum würde sehr wahrscheinlich Angriffe durch den Wirt erlauben, der diese Funktion unter Umständen benutzen könnte, um den Verschlüsselungsschutz des Agenten aufzuheben.

8.6.2 Funktionsverschlüsselung durch Fehlerkorrekturcodes

Aufbauend auf den Arbeiten von Sander und Tschudin wird in [LM99a] und [LM99b] ein ähnlicher Ansatz präsentiert, der allerdings auf der Verwendung von binären Fehlerkorrekturcodes der Länge n beruht. Die Autoren diskutieren einige Angriffsmöglichkeiten, kommen aber zu dem Schluss, dass das Verfahren für genügend lange Codes sicher ist. In [LM99b] wird als Anwendungsbeispiel ein Boole-

scher Schaltkreis mit a Eingängen und b Ausgängen betrachtet, wobei die zu übertragende Datenstruktur die Größe $2^a \times n$ hat.

Aufgrund der Ähnlichkeit des Ansatzes zum "Mobile-Cryptography"-Ansatz gelten auch für dieses Verfahren alle Vorteile und der Nachteil der Beschränkung auf Agenten, die nicht kommunizieren. Gegenüber dem ersten Ansatz hat die Verwendung von Fehlerkorrekturcodes jedoch den Vorteil, etwas effizienter bezüglich der Größe der zu übertragenden Daten zu sein, keine wesentlichen Probleme mit möglichen Angriffen zu haben und direkt auf Turing-mächtige Sprachen (Boolesche Schaltkreise) anwendbar zu sein.

8.6.3 Code Padding

Ist es in den letzten beiden Ansätzen noch vorgesehen, dass der Agent Eingaben durch den Ausführenden erhält, beschäftigt sich [Baz98] mit der Frage, wie man eine Funktion f mit den Daten d ohne die Möglichkeit einer bekannten Eingabe vor Lese- und Modifikationsangriffen durch den Ausführenden schützen kann. Die ausführende Partei wird nur noch benutzt, um den Agenten auszuführen. Da auch dieses Modell keine Klartextausgaben zulässt, ist es bezüglich der damit möglichen Anwendungen noch weiter eingeschränkt.

Das generelle Schema besteht darin, zwischen einzelne Anweisungen des Originalagenten automatisch weitere Anweisungen einzufügen, so dass ein Angreifer nicht feststellen kann, welcher Programmfluss relevant ist. Damit ist ein Angreifer vor das Problem gestellt zu entscheiden, welche Anweisungen zum eigentlichen Agenten gehören, und welche nicht. Damit also u.a. auch welche Daten relevant sind, und welche nur zur Verwirrung des Angreifers berechnet werden. In [Baz98] wird allerdings dieses generelle Schema nicht ausgeführt, sondern ein Verfahren präsentiert, das dieses Schema auf den Fall von Polynomfunktionen über Z_m anwendet.

8.7 Zeitbeschränkter Blackbox-Schutz

In diesem Kapitel wird ein neues Verfahren beschrieben, um Blackboxes zu erzeugen. Im Gegensatz zu den Ansätzen in Kapitel 8.6 sieht es das verwendete Modell vor, dass der Agent auch Klartextausgaben während seiner Ausführung vornehmen kann. Teile dieses Kapitels wurden [Hoh97] und [Hoh98b] entnommen.

8.7.1 Die Idee

Um zu einem Angriffsprogramm zu kommen, muss ein menschlicher Angreifer den Code des anzugreifenden Agenten lesen und verstehen. Dieser Vorgang erlaubt es dem Angreifer ein mentales Modell des Agenten zu erstellen. Ausgehend von diesem mentalen Modell und einem gegebenen Angriffsziel (etwa der Kenntnis des

Inhalts einer bestimmten Variablen) kann dann ein entsprechendes Angriffsprogramm geschrieben werden. Die Analyse durch den menschlichen Angreifer benötigt Zeit und kann kaum schneller gemacht werden, zumindest bleibt die Größenordnung der benötigten Zeit gleich. Wie bereits in Kapitel 5.6.1 diskutiert, ist es wahrscheinlich, dass ein Angreifer einem Agenten seinen Code zuordnen kann (weil es z.B. nur wenige Typen von Agenten gibt). Damit hat der Angreifer die Möglichkeit, eine Analyse eines Agenten vor dessen Eintreffen durchzuführen, das Angriffsprogramm zu schreiben, und dann dieses Angriffsprogramm anzuwenden, wenn ein Agent dieses Typs ankommt.

Die zentrale Idee des in diesem Kapitel beschriebenen Ansatzes ist nun die “Entwertung” dieser vorab erstellen Analyse durch eine Konvertierung des Agenten in eine Form, die immer noch ausführbar bleibt, und auch (im Wesentlichen) dasselbe tut wie der ursprüngliche Agent, die aber anders aufgebaut ist. Diese Konvertierung wird bei der Erzeugung der Agenteninstanz durch sogenannte “Verwürfelungsverfahren” vorgenommen. Die Konvertierung muss so beschaffen sein, dass der Angriff schwer durchführbar wird, d.h. der so erreichte Schutz darf nicht einfach durch ein spezialisiertes Angriffsprogramm (das immer noch das ursprüngliche Angriffsziel anstrebt, nicht aber unbedingt die komplette Rückkonvertierung des Agenten) brechbar sein.

Die Erfahrung zeigt, dass es einem *menschlichen* Angreifer letzten Endes immer gelingen wird, einen solchen Schutz zu brechen. Zwar könnte der erreichte Schutz gegen spezialisierte Angriffsprogramme einen gewissen Aufwand seitens des Angreifers erfordern, aber schon durch Berechnungen im Stundenbereich brechbar sein. Daher kann nicht von der (im Kryptografie-Bereich vorherrschenden) Annahme ausgegangen werden, dass der Schutz “für immer” ausreicht, sondern es wird angenommen, dass der erreichte Schutz nur über ein bestimmbares Zeitintervall ab der ersten Migration auf einen nicht-vertrauenswürdigen Wirt aufrechtzuerhalten ist. Daher tragen derart geschützte Agenten ein *Verfallsdatum*, das das Ende dieses Intervalls kennzeichnet. Nach diesem Verfallsdatum gilt der Agent als angreifbar, und daher als ungültig, d.h. eine Ausführung dieses Agenten oder eine Interaktion mit demselben geschieht auf eigene Gefahr.

Solchermaßen geschützte Agenten sind Blackboxes, aber nur für eine bestimmte Zeit (siehe Bild 38). Durch die Zeitbeschränkung ändert sich auch das generelle Ausführungsmodell: ein Agent kann nicht mehr beliebig lange unterwegs sein, sondern muss seine Aufgabe innerhalb einer bestimmten Frist erledigen. Bevor wir zu den Verwürfelungsverfahren kommen, soll daher im nächsten Abschnitt auf die Frage eingegangen werden, welche Auswirkungen diese Zeitbeschränkung hat.

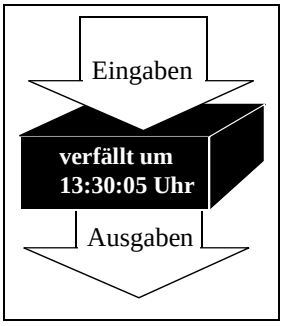


Bild 38: Zeitbeschränkte Blackbox

8.7.2 Was ändert sich durch eine solche Zeitbeschränkung?

Die Zeitbeschränkung durch das Schutzverfahren wirkt sich auf zweierlei Weisen auf einen mobilen Agenten aus. Einmal beschränkt sich zunächst die Zeit, die der Agent für die Durchführung seines Auftrages zur Verfügung hat. Falls die Zeit zur Erfüllung des Auftrages nicht ausreicht, muss es möglich sein, den Agenten wieder "aufzuladen", d.h. sein Verfallsdatum neu zu setzen. Dieser Aspekt ist Gegenstand des Kapitel 8.7.6. Neben dem Effekt auf die Zeit zur Durchführung wirkt sich die Zeitbeschränkung jedoch auch auf die Mechanismen aus, die sicherstellen sollen, dass der Schutz gewährleistet wird. Um diesen Sicherheitsaspekt der Zeitbeschränkung zu verdeutlichen, werden im Folgenden vier Interaktionsszenarien betrachtet.

Keine Kommunikation mit einer dritten Partei

In diesem Szenario kommunizieren weder der Agent noch der Wirt mit einer dritten Partei. Da keine Kommunikation stattfindet, resultieren aus diesem Szenario keine Forderungen an die Schutzmechanismen. Selbst wenn der Wirt den Agenten angreift, hat dieser Angriff keinen Effekt, da die "Außenwelt" nicht miteinbezogen wird.

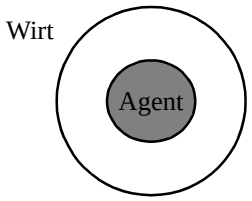


Bild 39: Keine Kommunikation

Kommunikation nur mit vertrauenswürdigen Partnern

In diesem Szenario kommuniziert der Agent nur mit einer vertrauenswürdigen dritten Partei, also einer, die den Agenten nicht angreift. Daher können der Agent und sein Kommunikationspartner einen sicheren Kommunikationskanal etablieren, wie er in Kapitel 8.3 diskutiert wurde. Da der

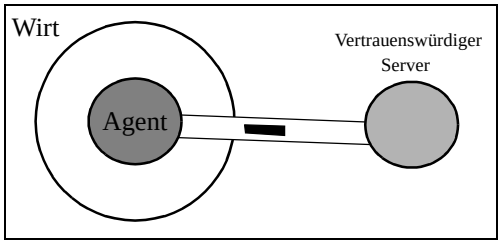


Bild 40: Kommunikation nur mit vertrauenswürdigem Partner

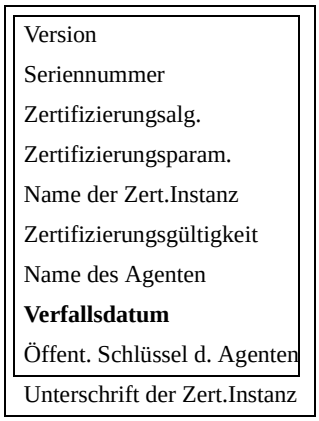


Bild 41: Erweiteres Schlüsselzertifikat

Wirt den Agenten nach dem Verfallsdatum angegriffen haben kann, und so gegenüber dem Partner vorgeben könnte, der Agent zu sein, muss der Kommunikationspartner feststellen können, ob der Agent noch gültig, oder bereits verfallen ist. Dies kann dadurch geschehen, dass Schlüsselzertifikate erweitert werden, die zum Aufbau des sicheren Kommunikationskanals benutzt werden. Die Erweiterung besteht dabei lediglich aus dem Verfallsdatum, so dass an dieser Stelle kein wesentlicher Mehraufwand entsteht (siehe Bild 41).

Kommunikation mit nicht-vertrauenswürdigen Partnern

In diesem Szenario kommuniziert der Agent entweder mit einer nicht-vertrauenswürdigen dritten Partei oder mit dem Wirt, der nicht-vertrauenswürdig sein kann (siehe Bild 42).

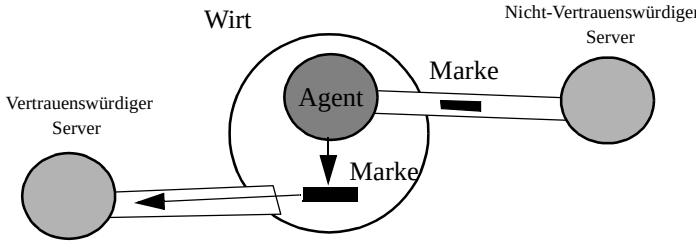


Bild 42: Kommunikation mit nicht-vertrauenswürdigen Partnern

Wir können dabei zwei Arten von zu kommunizierenden Daten unterscheiden: “Marken” und andere Daten.

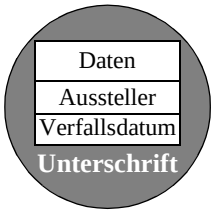


Bild 43: Marke

Marken (engl. token) sind „Inhaberausweise“, d.h. Dokumente, die denjenigen, der sie vorweisen kann, berechtigen, etwas zu tun. Beispiele für Marken sind elektronische Münzen (die den Inhaber berechtigen, eine bestimmte Summe Geldes auszugeben), kryptografische Schlüssel (die den Inhaber berechtigen, sich als ein bestimmtes Objekt auszuweisen, in dessen Namen zu unterschreiben, oder verschlüsselte Nachrichten an dieses Objekt zu lesen) oder andere Berechtigungsausweise (engl. capabilities oder tickets), die den Inhaber dazu berechtigen z.B. eine Datei modifizieren zu dürfen. Marken enthalten neben spezifischen Daten wie z.B. dem Namen der zu modifizierenden Datei und eine Referenz auf den Aussteller. Marken sind durch den Aussteller unterschrieben. Das Problem bei Marken ist, dass ein Angreifer diese zum Nachteil des Auftraggebers des Agenten benutzen kann, obwohl sie durch einen Angriff nach Ablauf des Verfallsdatums in dessen Hände gelangten. Daher müssen auch Marken Verfallsdaten tragen, die es verhindern, dass diese noch nach Erreichen des Verfallsdatums des Agenten gültig sind (siehe Bild 43). Zweckmäßigerweise stimmen daher die Verfallsdaten der Marken und des sie transportierenden Agenten überein. Jeder, der eine Marken zur Benutzung oder Einlösung erhält, muss daher anhand des Verfallsdatums prüfen, ob die Marken noch gültig ist oder nicht. Damit diese Prüfung stattfinden kann, muss es eine korrekte, globale Zeit bei allen Parteien geben, d.h. dass synchronisierte Uhren für diesen Ansatz notwendig sind. Dabei ist die Kenntnis der aktuellen Zeit für die

die Marken transportierende Partei (z.B. der Agent) nicht notwendig. Es reicht, wenn die die Marke ausstellende und die die Marke einlösende Partei diese Zeit kennen. Der Aussteller benötigt die Zeit, um das Verfallsdatum einzutragen, der Empfänger muss das Verfallsdatum gegen das aktuelle Datum vergleichen um keine ungültige Marke zu bekommen. Zwar hindert ihn niemand, auch eine ungültige Marke anzunehmen, aber er wiederum kann diese Marke dann nicht mehr einlösen.

Einige mögliche Marken haben einen Schutzbedarf, der zeitlich größer ist als das durch das Verfallsdatum eines Agenten bestimmte Zeitintervall. Dies kann für existierende Marken der Fall sein, wenn sich bei diesen keine Zeitbegrenzung einbauen lässt oder z.B. bei den geheimen Schlüsseln der Agenteneigentümer ein Problem sein, die im Allgemeinen eines unbegrenzten Schutzintervalls bedürfen. Marken dürfen daher nur von Agenten transportiert werden, wenn sie ein ausreichend großes Schutzintervall anbieten können.

Umgekehrt bedeutet die nun notwendige Zeitbeschränkung der Gültigkeit von mit Agenten transportierten Schlüsseln auch eine Änderung der Semantik der mit diesen Schlüsseln assoziierten Berechtigung. Zeitbeschränkte Schlüssel erlauben insbesondere keinen zeitlich unbegrenzten Schutz der Privatheit von verschlüsselten Daten. Das bedeutet dass alle an der verschlüsselten Kommunikation beteiligten Parteien davon ausgehen müssen, dass die verschlüsselten Daten nach der Verfallszeit des Schlüssels öffentlich lesbar sein können.

Neben Marken transportiert ein Agent jedoch meist auch andere Daten. Beispiele für diese Kategorie sind einfache Zahlenwerte wie z.B. der maximale Preis im Beispiel in Kapitel 5.1. Ist ein Agent eine zeitbeschränkte Blackbox, so wird davon ausgegangen, dass der Wirt diesen Wert vor dem Verfallsdatum weder lesen noch modifizieren kann. Lese- und Schreibangriffe nach dem Verfallsdatum sind zwar möglich, aber sie haben keinen Effekt mehr, weil diese ohne gültigen Agent nicht mehr zum Nachteil des Agentenauftraggebers verwendet werden können.

Migration eines Agenten

Das letzte Szenario umfasst die Möglichkeit, explizit Information durch den Agenten zu übermitteln: die Migration des Agenten zum nächsten Wirt (siehe Bild 44). Das Problem dabei besteht darin, dass der Agent vom letzten Wirt nach dem Verfallsdatum angegriffen sein könnte. Mit der Benutzung von signiertem Code ist es zwar ausgeschlossen, dass der Angriff diesen Code manipuliert haben könnte, aber Angriffe gegen den variablen Zustand sowie Leseangriffe sind möglich.

Daher muss der empfangende Wirt sicherstellen, dass der ankommende Agent gültig ist, d.h. dass das Verfallsdatum noch nicht abgelaufen ist. Da wir bereits annehmen, dass der Agent durch eine Signatur geschützt ist, und diese Signatur über den

konstanten Teilen des Agenten erfolgen wird, kann das Verfallsdatum als ebenfalls konstanter Wert ebenfalls in diese Signatur einbezogen sein. Der empfangende Wirt muss daher nur diese Signatur verifizieren und das Verfallsdatum gegen das aktuelle Datum vergleichen. Nicht gültige Agenten werden nicht angenommen; gegebenenfalls kann der Agentenauftraggeber über dieses Ereignis informiert werden.

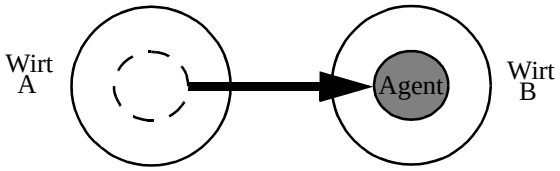


Bild 44: Migration

Wie wir gesehen haben, ist es also möglich, die meisten Effekte der Zeitbeschränkung des Blackbox-Schutzes zu kompensieren. Wie kann aber nun ein solcher Schutz realisiert werden?

8.7.3 Realisierung zeitbeschränkten Blackbox-Schutzes

In diesem Abschnitt wird auf Ansätze zur Erzeugung von Verwürfelungsverfahren eingegangen, die den Kern von Mechanismen zum zeitbeschränkten Blackbox-Schutz bilden.

Verwürfelungsverfahren

Zunächst muss erst einmal der Begriff des Verwürfelungsverfahrens definiert werden. Dazu benötigen wir den Begriff einer Umwandlungsfunktion.

Definition: ***Umwandlungsfunktion***

> Sei C die Menge aller Programme, $c \in C$ das Programm des Ursprungsagenten und $c^B \in C$ das Programm der Blackbox. Sei S die Menge aller Zustände eines Agenten. Ein Zustand $s \in S$ besteht dabei aus einer Menge von Elementen d, die Paare von jeweils einem Bezeichner und einem Wert darstellen. Sei $s_{inital} \in S$ der Anfangszustand des Ursprungsagenten und $s^B_{initial} \in S$ der Anfangszustand der Blackbox, sowie $s_{final} \in S$ der Endzustand des Ursprungsagenten und $s^B_{final} \in S$ der Endzustand der Blackbox. Sei I_i die Abfolge von Eingabe-/Ausgabeereignissen eines Ursprungsagenten $(c,s_{initial})$ während der Ausführung auf einem Wirt i und I^B_i die Abfolge

von Eingabe-/Ausgabeereignissen der Blackbox $(c^B, s^B_{initial})$ während der Ausführung auf demselben Wirt i.

Damit berechnet der Ursprungsagent auf einem Wirt i aus einem Anfangszustand s_{inital} einen Endzustand s_{final}, wobei er eine Abfolge von Eingabe-/Ausgabeereignissen I_i erzeugt.

Dual dazu berechnet die Blackbox auf einem Wirt i aus einem Anfangszustand $s^B_{initial}$ einen Endzustand s^B_{final}, wobei sie eine Abfolge von Eingabe-/Ausgabeereignissen I^B_i erzeugt.

Eine *Umwandlungsfunktion* ist dann eine Funktion f auf $C \times S \rightarrow C \times S$, wobei gilt:

(1) $f(c, s_{initial}) = (c^B, s^B_{initial})$

(2) $\forall(d \in s_{initial}) \quad \exists \bar{f}_d \quad \exists(e_d \in s^B_{initial}) \quad \bar{f}_d(e_d) = d$, wobei d ein Element eines Zustands (in diesem Fall des Anfangszustands des Ursprungsagenten) und $\bar{f}_d$ eine Funktion auf Datenelementen aus Zuständen ist

(3) $I_i = I^B_i$

(4) $\forall(d \in s_{final}) \quad \exists \bar{f}_d \quad \exists(e_d \in s^B_{final}) \quad \bar{f}_d(e_d) = d$, wobei d ein Element eines Zustands (in diesem Fall des Endzustands des Ursprungsagenten) und $\bar{f}_d$ eine Funktion auf Datenelementen aus Zuständen ist

Umgangssprachlich bedeutet das, dass eine Umwandlungsfunktion eine Funktion ist, die den Ursprungsagenten, der aus Code und initialem Zustand besteht, nimmt und in eine Blackbox, bestehend aus neuem Code und neuem initialem Zustand, umwandelt. Dabei muss gelten, dass sich beide Agenten im äußeren Verhalten bis auf Zeitaspekte nicht unterscheiden (siehe auch die Diskussion in Kapitel 7.2). d.h. dass sich die Abfolge von Eingabe-/Ausgabeereignissen nicht unterscheidet. Die Blackbox mag sich im inneren Aufbau unterscheiden, von außen ist sie nicht unterscheidbar. Weiterhin muss gelten, dass die Datenelemente aus dem initialen Zustand des Ursprungsagenten auch im initialen Zustand der Blackbox zu finden sein müs-

sen (siehe die Eigenschaften der Blackbox in Kapitel 8.2). Die Blackbox kann zwar mehr Datenelemente beinhalten, aber zu jedem Datenelement des Anfangszustandes im Ursprungsagenten muss es eine „Entschlüsselungsfunktion“ geben, die dieses Element wieder aus dem Anfangszustand der Blackbox herausholen könnte. Das Gleiche muss auch für den Endzustand beider Agenten gelten. Jedes Datenelement aus dem Endzustand des Ursprungsagenten muss sich auch im Endzustand der Blackbox wiederfinden lassen.

Diese Definition spezifiziert zwar, dass eine Umwandlung stattfindet (bzw. stattfinden kann, denn auch eine Umwandlung in den gleichen, also den Ursprungsagenten ist nach dieser Definition eine Umwandlung), aber noch nicht, dass die Blackbox gegen bestimmte Angriffe geschützt ist. Wir benötigen daher eine weitergehende Definition.

Definition: ***Verwürfelungsverfahren***

Jeder Zustand besteht aus einer Menge von Elementen, die Paare von jeweils einem Bezeichner und einem Wert darstellen. Sei x der Bezeichner eines Elements $d \in s_{initial}$, w der dazu gehörende Wert (damit ist $d = (x,w)$). Sei weiter v eine Datenlesefunktion, so dass $v(x, s_{initial}) = w$. Gegeben sei ein Ursprungsagent $(c,s_{initial})$ sowie eine Blackbox $(c^B,s^B_{initial})$ dieses Ursprungsagenten.

Ein Verwürfelungsverfahren f ist dann eine Umwandlungsfunktion, für die innerhalb eines Zeitintervalls t und für einen bestimmbaren „Angriffsaufwand“ $e(k)$ gilt:

(1) Gegeben sei der Bezeichner x eines Elements $d \in s_{initial}$. Dann gilt:
$\neg\exists r(c, c^B, s^B_{initial}, x) = w$.

(2) Gegeben sei der Bezeichner x eines Elements $d \in s_{initial}$ mit $d = (x,w)$ sowie ein weiterer Wert w'. Dann gilt:
$\neg\exists m(c, c^B, s^B_{initial}, x, w') = s^B_{initial}'$, so dass
$f(c, s_{initial}') = (c^B,s^B_{initial}')$ mit $v(x, s_{initial}') = w'$ für $w \neq w'$, wobei $v(x, s_{initial}) = w$

(3) Gegeben sei der Bezeichner x eines Elements $d \in s_{final}$ mit $d = (x,w)$ sowie ein weiterer Wert w'. Dann gilt:

$\neg\exists m(c, c^B, s^B_{final}, x, w') = s^B_{final}'$, so dass

$f(c, s_{final}') = (c^B, s^B_{final}')$ mit $v(x, s_{final}') = w'$ für $w \neq w'$, wobei $v(x, s_{final}) = w$

(4) Sei a die Ausführungsfunktion (z.B. ein Interpreter einer Programmiersprache) eines Agenten, die unter Benutzung des Programms und eines Anfangszustands die Abfolge von Eingabe-/Ausgabeereignissen I_i dieses Agenten erzeugt. Da ein Verwürfelungsverfahren auch eine Umwandlungsfunktion ist, muss gelten: $a(s_{initial}, c) = a(s^B{}_{initial}, c^B) = I_i$, d.h. die Abfolge von Eingabe-/Ausgabeereignissen bei Ursprungsagent und Blackbox ist gleich.

Gegeben sei nun eine beliebige, vom Angreifer ausgewählte Abfolge von Eingabe-/Ausgabeereignissen $I_i' \neq I_i$. Dann gilt:

$\neg\exists p(c, c^B, s^B_{initial}, I_i') = c^{B,}$ mit $a(s^B{}_{initial}, c^{B,}) = I_i'$

Umgangssprachlich bedeutet das, dass

- **Datenleseangriffe**, also das Bestimmen eines Elements des Zustands unter Benutzung eines Bezeichners dieses Datums im Ursprungsagenten (z.B. ein Variablenname) nicht möglich sind. In (1) wird daher gefordert, dass es keine Funktion gibt, die zu einem solchen Bezeichner nur aus dem Zustand der Blackbox heraus den Wert ermitteln kann.
- dauerhafte, gezielte **Datenmodifikationsangriffe** (also solche, bei denen der Angreifer den Wert eines Datums beliebig wählen kann) nicht möglich sind. In (2) wird daher gefordert, dass es keine Funktion gibt, die eine gezielte Modifikation des Anfangszustands bezgl. eines gewünschten Wertes eines bestimmten Zustandselementes erlaubt, falls dieses Element Teil des Anfangszustandes ist. In (3) wird dasselbe für den Fall gefordert, dass das Element Teil des Endzustands ist (in (3) um den Fall abzudecken, dass die Ausführung des Agenten neue Elemente erzeugt, in (2) für den Fall, dass ein Element im Laufe der Ausführung gelöscht wird).
- keine temporären **Codemodifikationsangriffe** möglich sind, siehe dazu auch die Überlegungen in Kapitel 8.2. Nachdem in (1), (2) und (3) bereits Angriffe gegen Daten abgedeckt sind, können Angriffe, die den Code temporär ändern, nur noch das Interaktionsverhalten modifizieren. In (4) wird daher gefordert,

dass es keine Funktion gibt, die eine Codemodifikation erlaubt, die eine gewünschte Interaktionsänderung zur Folge hat.

Die obige Definition bedingt dabei nicht, dass eine Blackbox vollständig unangreifbar ist. Nicht nur, dass die Eigenschaften nur für ein bestimmtes Schutzintervall gelten müssen, sie gelten auch nur unterhalb eines bestimmten „Angriffsaufwandes" seitens des Angreifers. Dieser Angriffsaufwand $e(k)$ sollte dabei günstigerweise von einem durch den Blackbox-Ersteller frei wählbaren Parameter k, etwa einer Schlüssellänge, abhängen, so dass auch der Angriffsaufwand wählbar ist. Trotzdem gilt, dass ein Angreifer eine Blackbox auch innerhalb des Schutzintervalls angreifen kann, wenn er mehr als diesen Aufwand aufbringt. Es ist daher für den Schutz des Agenten wichtig, den Zusammenhang von Angriffsaufwandsfunktion und Schutzintervall zu klären.

Zusammenhang von Schutzintervall und Angriffsaufwand

Schutzintervall und Angriffsaufwand hängen von der Leistungsfähigkeit der Angreifermaschine ab. Wir definieren daher:

Definition: ***Angriffsleistung***

Sei $P = e(k)$ *der Aufwand eines Angriffs in einer Einheit [e] für einen festen Parameter* k. *Sei* t *ein Zeitintervall in Sekunden. Dann ist die Angriffsleistung* E *definiert als* $E = \frac{P}{t}$ *in* $\frac{[e]}{[s]}$.

Das bedeutet, dass wenn eine Angreifermaschine eine Leistung von 1 $\frac{[e]}{[s]}$ hat, und der Aufwand 100 [e] beträgt, dann das Schutzintervall 100 [s] beträgt. Wird statt dessen eine zweite Angreifermaschine mit einer Leistung von 2 [e] benutzt, verringert sich das Schutzintervall auf 50 [s]. Eine konkretere Beispielrechnung findet sich im übernächsten Abschnitt.

Um für einen gegebenen Angriffsaufwand ein Schutzintervall zu berechnen, muss man also die Leistung der Angreifermaschine kennen. Dies ist natürlich nicht möglich, aber man kann z.B. die erwartete Leistung der Angreifermaschine verwenden, die Leistung des besten bekannten Computersystems, oder, für besonders grossen Schutz, die Leistung eines imaginären Rechnersystems, das frühestens in einigen Jahren realisierbar sein wird.

Bestimmung der Angriffsaufwandsfunktion

Was jetzt noch fehlt, ist die Definition der Einheit [e]. Mit dieser Definition eng verknüpft ist die Frage, wie die Angriffsaufwandsfunktion berechnet wird. Gemäß der Definition eines Verwürfelungsverfahrens muss die Angriffsaufwandsfunktion bestimmbar sein (es mag durchaus Verfahren geben, für die man zwar zeigen kann, dass eine solche Funktion existiert, deren Funktion aber nicht bestimmt wurde). Diese Funktion leitet sich dabei davon von dem Umstand ab, dass das Verwürfelungsverfahren so beschaffen ist, dass ein Angriff einen ganz bestimmtes Problem lösen muss, um die Verwürfelung zu brechen. Dies erfolgt analog zu den existierenden kryptografischen Verfahren, die das Problem z.B. der Entschlüsselung von Daten auf das Berechnen eines bestimmten mathematischen Problems zurückführen. Wie in der Kryptografie kann man dann die besten bisher bekannten oder die theoretisch besten Algorithmen bezüglich ihres Aufwandes untersuchen.

Die Ausführung des Angriffsalgorithmus besteht aus der Abarbeitung einer Anzahl von verschiedenen Operationen. Auf konkreten Rechner verhalten sich die Ausführungszeiten verschiedener Operationen zueinander unterschiedlich. Verkompliziert wird dieser Aspekt außerdem noch durch die Tatsache, dass verschiedene Rechnerarchitekturen erlauben, bestimmte Operationen parallel auf dem Prozessor auszuführen. Da innerhalb dieser Arbeit diese sehr komplexen Zusammenhänge, die auf verschiedenen Rechner auch noch unterschiedlich sein können, nicht modelliert werden können, wird für alle Operationen die Einheit [e] gezählt. In [e] wird dabei nur die Anzahl der Operationen gezählt; die Ausführungszeit einer Operation ist bereits in der Angriffsleistung modelliert.

Eine Beispielrechnung

Um zu illustrieren, wie die Berechnung eines Schutzintervalls ablaufen könnte, soll dieses Vorgehen anhand eines hypothetischen Beispiels erläutert werden. Nehmen wir an, wir finden ein Verwürfelungsverfahren, das auf der Einfügung künstlicher If-Anweisungen beruht (ähnlich, wie es das Angriffsverhinderungsverfahren in Kapitel 8.7.4 macht). Dieses Verfahren habe eine Angriffsaufwandfunktion $e(k) = 2^k$ wenn k die Anzahl einzufügender If-Anweisungen ist. Nehmen wir an, wir wählen $k = 100$, fügen also 100 If-Anweisungen ein. Dann ist $e(100) = 2^{100} = 1,27 \cdot 10^{30} \cdot [e]$ der zu erbringende Angriffsaufwand.

Nehmen wir als Angriffsleistung einen Wert von 10 G $\frac{[e]}{[s]}$, also $10^{10}\frac{[e]}{[s]}$ für die Angreifermaschine an*. Dann beträgt das Schutzintervall $10^{20} \cdot [s]$, das sind mehrere Billionen Jahre, ein Wert, der auch durch massivste Parallelisierung und auch

nach mehreren Jahrzehnten Rechenleistungssteigerung nach dem Moore'schen Gesetz nicht auf ein in einem Menschenleben erwartbares Niveau herabgesetzt werden kann.

Würde das Verwürfelungsverfahren nur eine Aufwandsfunktion von $e(k) = k^2$ haben, dann würde das Schutzintervall bei $k = 100$ und derselben Angriffsleistung nur noch eine Mikrosekunde betragen...

Durchführung der Verwürfelung

Um die Verwürfelung durchzuführen, spezifiziert der Programmierer

- die zu benutzende Angriffsleistung,
- den maximalen Angriffsaufwand sowie
- die Länge des gewünschten Schutzintervalls,

wobei der Agenteneigentümer vor der Erzeugung der Blackbox diese Werte modifizieren kann (aber nicht muss). Das System berechnet aus Angriffsleistung und Angriffsaufwand das mögliche Schutzintervall und vergleicht es gegen das spezifizierte Schutzintervall. Ist das errechnete Schutzintervall größer oder gleich dem spezifizierten, wird das Verwürfelungsverfahren verwendet, um eine Blackbox zu generieren, sonst wird eine Fehlermeldung zurückgegeben. Im letzten Fall kann der Agenteneigentümer neue Werte für Angriffsleistung, Angriffsaufwand und Schutzintervall wählen.

Es ist vorstellbar, dass es mehrere Verwürfelungsverfahren gibt, die das System einsetzen kann (im Gegensatz zur traditionellen Kryptografie müssen sich Sender und Empfänger des Agenten nicht auf ein gemeinsames Verfahren einigen, so dass jedes System zu jedem Zeitpunkt beliebige Verwürfelungsverfahren anbieten kann). In diesem Fall könnte das System für jedes Verwürfelungsverfahren ein Schutzintervall berechnen, und dann zufällig eines der Verwürfelungsverfahren auswählen, deren errechnetes Schutzintervall über dem spezifizierten liegt.

---

* Es ist aufgrund den im letzten Abschnitt aufgezählten Problemen schwierig, ein abstraktes Maß für die benötigte Rechnerleistung zu finden. Wenn man das Maß „Dhrystone Mips" nimmt, einen synthetischen Benchmark, der vor allem die CPU-Leistung bei bestimmten String-Operationen misst, und wenn man die Einheit 1 Mips, die sich auf die Leistung der VAX 11/780 als einer Maschine bezieht, die per definitionem 10^6 Operationen pro Sekunde leistete, wörtlich nimmt, dann haben heutige CPUs, etwa ein Intel Pentium III-500 oder ein Motorola PowerPC G4+-733 etwa 1400 Mips.

Einbeziehung des Schutzaufwandes

Ein Problem kann in dem Mehraufwand liegen, den das Verwürfelungsverfahren zur Laufzeit des Agenten benötigt. Unter dem Aspekt, dass manche Anwendungen auf der Basis mobiler Agenten nur Effizienzvorteile gegenüber der gleichen Client-Server-basierten Lösung bieten (vergleiche auch die Diskussion dieses Aspektes in Kapitel 5.5), kann ein solch harter Schutz diese Vorteile aufheben oder sogar in Nachteile verwandeln. Daher kann der durch Verwürfelung entstehende Mehraufwand nicht beliebig hoch sein. Um diese Überlegungen miteinzubeziehen, muss ein Verwürfelungsverfahren nicht nur eine Angriffsaufwandsfunktion $e(k)$ spezifizieren, sondern auch noch eine Schutzaufwandsfunktion $g(k)$, die ebenfalls vom Parameter k abhängt.

Nun kann der Programmierer (bzw. der Agenteneigentümer) zusätzlich noch

- den maximalen Schutzaufwand

spezifizieren. Bei der Berechnung des Schutzintervalls erhält das System über $P = e(k)$ (P ist der maximale Angriffsaufwand) ein k, über das mittels $g(k)$ ein Schutzaufwand errechnet werden kann. Nun kann auch ein Vergleich des errechneten Schutzaufwandes mit dem spezifizierten darüber entscheiden, ob eine Verwürfelung zu diesen Bedingungen möglich ist, und (im Falle mehrerer Verwürfelungsverfahren) welche Verfahren dafür eingesetzt werden können.

Einbeziehung wirtschaftlicher Überlegungen

Oft haben Angriffe einen monetären Hintergrund (d.h. der Angreifer will sich wirtschaftliche Vorteile verschaffen). Auf diesem Hintergrund ist es interessant, sich auch die wirtschaftlichen Aspekte von Angriffen und Schutzmechanismen zu verdeutlichen. Erwartet der Angreifer von einem Angriff einen wirtschaftlichen Wert der Größe m, dann wird er keine Angriffskosten in Kauf nehmen, die größer oder gleich diesem Wert sind (falls es sich um einen Angreifer handelt, der kostenrechnerische Überlegungen anstellt). Da der Programmierer oder der Agenteneigentümer die wirtschaftlichen Werte, die ein Agent mit sich trägt, kennt, könnte man nun diese Werte mittels einer Funktion $k(m)$ in einen maximalen Angriffsaufwand überführen, so dass statt diesem Aufwand der maximale Verlust spezifiziert werden könnte. Auch diese Funktion hängt dynamisch vom Stand der Technik ab, und könnte durch den Programmierer oder den Agenteneigentümer spezifiziert werden.

Wie könnten Verwürfelungsverfahren gefunden werden?

Die abzuwehrenden Angriffe gegen Blackboxes sind sehr unterschiedlich und zum Teil unabhängig voneinander. Selbst wenn beispielsweise bei einem Angriffsziel “maximaler Preis” die Daten gegen Leseangriffe geschützt sind, können Modifikationsangriffe dazu führen, dass der Wirt diesen Preis kennt (indem der Wirt einfach

einen Wert in die Variable schreibt). Ein möglicher Weg, Verwürfelungsverfahren zu entwickeln, besteht daher zunächst darin, die Vielzahl der möglichen Angriffe auf einen einzigen Angriff zu reduzieren, ganz ähnlich wie bei den Ansätzen der nicht-interaktive Auswertung von verschlüsselten Funktionen in Kapitel 8.6 die Vielzahl der Angriffe auf Lese- und Modifikationsattacken reduziert wird. Dann muss, wie bei kryptografischen Verfahren, ein hartes mathematisches Problem gefunden werden, dessen Lösung von beweisbar hoher Komplexität ist, und ein Verwürfelungsverfahren muss entwickelt werden, das erreicht, dass Angriffe dieses Problem lösen müssen. Im Vergleich zu Verschlüsselungsverfahren haben Verwürfelungsverfahren zwei Vorteile, die die Entwicklung eines solchen Verfahren vereinfachen könnten:

- Es gibt keine Notwendigkeit, die Verwürfelung auf Empfängerseite wieder aufzuheben, da das äußere Verhalten des Agenten unverändert bleibt und daher eine eventuell notwendige Kommunikation des Agenten mit vertrauenswürdigen Wirten auch in verwürfeltem Zustand möglich ist.
- Die Identität, die Reihenfolge, die Parameter, eventuell sogar das genaue Verfahren der angewendeten Verwürfelungsverfahren kann geheim bleiben und muss keinem Empfänger mitgeteilt werden.

Die Frage, ob es überhaupt Verwürfelungsverfahren gibt, die auf beliebige Agenten anwendbar sind, konnte in dieser Arbeit nicht geklärt werden. Kann man die Einschränkung treffen, dass Agenten keine Klartextdaten kommunizieren, so kann diese Frage bejaht werden, weil die Verfahren zur nicht-interaktiven Auswertung von verschlüsselten Funktionen aus Kapitel 8.6 als Blackbox-Schutz realisierende Verfahren natürlich auch zeitbeschränkt sind, wobei das Schutzintervall beliebig groß sein kann.

Was gefunden wurde, sind Teilschutz-Verfahren, d.h. solche, die im Wesentlichen eine Angriffsmethode verhindern. Würde man das Problem des Blackbox-Schutzes auf diese Angriffsmethoden reduzieren können, könnte man diese Verfahren wie oben beschrieben als Kern eines Verwürfelungsverfahrens benutzen. Eine solche Reduktion gelang, wie gesagt, in dieser Arbeit nicht. Auch wenn diese Verfahren nur gegen einzelne Angriffsmethoden schützen, lässt sich an ihnen illustrieren, wie Verwürfelungsverfahren vorgehen könnten. Im Gegensatz zu Verwürfelungsverfahren kann man diese einzelnen Verfahren als Angriffsverhinderungsverfahren bezeichnen.

8.7.4 Angriffsverhinderungsverfahren (AVV)

Wir wollen in diesem Abschnitt vier Angriffsverhinderungsverfahren betrachten. Dazu werden wir die Angriffsmethode beschreiben, die sie verhindern sollen, das

Verfahren zur Verhinderung dieser Angriffsmethode sowie ihre Angriffsaufwandsfunktionen.

Variablenrekomposition

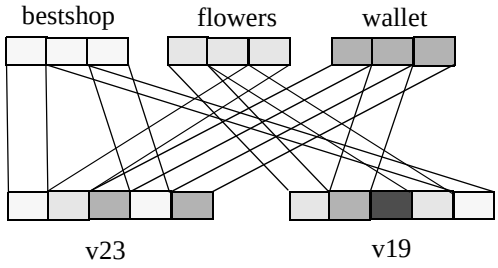
Das Ziel dieses AVVs ist es, Daten vor Leseangriffen zu schützen. Dazu nimmt dieses Verfahren die Programmvariablen, zerteilt jede Variable in Segmente und erzeugt neue Variablen als Kombination dieser Segmente.

Bild 45: Originaler Variablenzugriff

```
5 buy(bestshop,flowers,wallet)
6 go(home)
```

Der Zugriff auf die Ursprungsvariablen wird dann auf diese neue Speicherungsstruktur angepasst. In Bild 45 wird der originale Variablenzugriff auf die drei Variablen `bestshop`, `flowers` und `wallet` beschrieben. Bild 46 zeigt die Segmentierung dieser Variablen und die Rekomposition dieser Segmente in die zwei neuen Variablen `v23` und `v19`.

Bild 46: Variablenrekomposition



Der neue Variablenzugriff ist in Bild 48 abgebildet, wobei die Konvertierungsfunktionen in Bild 47 genutzt werden, um die originalen Inhalte aus den neuen Variablen zurückzukonvertieren.

Bild 47: Konvertierungsfunktionen

```
public Address c7(Bitstring b)
public Good c4(Bitstring b)
public Money c3(Bitstring b)
public Address c34(Bitstring b)
```

Das Ergebnis des AVVs ist eine neue Variablenstruktur, die keine einfache Erkennung der Originalvariablen erlaubt, und bei der die Variablennamen keine Rückschlüsse auf den Inhalt erlauben und im Speicher auch keine Klartextdaten mehr auftauchen.

Bild 48: Neuer Variablenzugriff

```
5  buy(c7(v23[0]+v19[4]+v23[3])
      ,c4(v19[0]+v19[3]+v23[1]),
      c3(v23[2]+v19[1]+v23[4]))
6  go(c34(v21[4]+v19[2]+v21[2]))
```

Als “Komplexitätsparameter” k bietet sich hier die Anzahl der Segmente pro Variable an.

Analyse des erzielten Schutzes

Leider ist der notwendige Aufwand zum Brechen dieses AVVs und damit auch $e(k)$ von O(1). Der Grund dafür liegt in einem trivialen möglichen Angriff. Der Inhalt aller so geschützten Variablen liegt nämlich im Klartext als Parameter auf dem Stapel vor (um unser Angriffsmodell aus Kapitel 5.6 zu verwenden), sobald die Operation `buy` ausgeführt wird. Daher kann der Angreifer diese Werte dann direkt vom Stapel lesen. Damit ist $e(k) = \#o$, wobei $\#o$ die Anzahl der ausgeführten Operationen bis Erreichen von `buy` ist.

Dies ist jedoch nicht die einzige Angriffsmöglichkeit. Es ist u.a. auch möglich, das neue Variablenlayout dadurch zu ermitteln, dass der Zugriff auf die einzelnen Bytes oder Bits statistisch über Operationen ausgewertet werden, von denen im Originalcode bekannt ist, dass sie einzelne Originalvariablen benutzen.

Iterative Verbesserung des Verfahrens

Beide Angriffe sind nur dann möglich, wenn sich die Prozedurenstruktur des Originalprogrammes in der Blackbox wiederfinden lässt. Eine mögliche Verbesserung des Verfahrens wäre daher die Auflösung der originalen Struktur. Dies ist allerdings nur dann möglich, wenn keine externen Prozeduren aufgerufen werden. In unserem Beispiel könnte schon *buy* eine solche externe Prozedur sein. Allerdings spielt bei externen Prozeduren der erste Angriff kaum eine Rolle, da die Parameter dieses Aufrufs ohnehin im Klartext nach außen gegeben werden, d.h. der Wirt bekommt diese Parameter ohnehin übergeben. Bei der Auflösung interner Prozeduren wäre es dann das Ziel eines weiteren AVVs, alle Prozeduren eines Agenten in eine Hauptprozedur zu vereinigen. Dies ist für Programme, die keine Rekursionen besitzen, durch Umwandlung des Programmes in einen gerichteten azyklischen Graphen und dessen Serialisierung möglich. Bei Programmen mit Rekursion muss dann ein eigener Stapel auf Programmebene benutzt werden.

Konvertierung von Kontrollflussanweisungen in werteabhängige Sprünge

Ein Problem beim Schutz eines Agenten kann ein zu statischer Kontrollfluss sein, d.h. einer, der im Wesentlichen schon im Ursprungsagenten, d.h. vor der Ausführung der Blackbox analysiert werden kann. Um dieses zu verhindern kann man den Kontrollfluss eines Programmes von Ausdrücken abhängig machen, in denen neue Variablen vorkommen.

Bild 49: Originaler Code

```
if (a(b) < c) {
   b = s(d(e) + f);
}
```

Ein Beispiel für diese Konvertierung von Kontrollflussanweisungen in werteabhängige Sprünge ist die Konvertierung einer if-Anweisung in eine Sequenz von Sprüngen, deren Bedingungen von Variablenwerten abhängen. In Bild 49 finden wir ein Beispiel für eine solche If-Anweisung, in Bild 50 dann den konvertierten Code, der aus der Anwendung des Verwürfelungsverfahrens entstanden ist.

Bild 50: Konvertierter Code

```
z= 0
DO
   if (z=0) then t1 = a(b); z=1; continue;
   if (z=1) then t2 = t1 < c; z=2; continue;
   if (t2) then t3 = d(e); z=3; continue;
   if (z=3) then t4 = t3 + f; z=4; continue;
   if (z=4) then b = t4; z=5; continue;
   if (z=5) then break;
LOOP
```

Analyse des erzielten Schutzes

Auch in diesem konkreten Beispiel kann der Schutz relativ leicht gebrochen werden. Der Angriff hier ist die Errechnung der originalen Anweisung. Die Methode besteht darin, die Sprünge nacheinander aufzulösen, wodurch die originale If-Anweisung in der einschließenden DO-LOOP-Schleife entsteht, die dann wegoptimiert werden könnte. Der Aufwand für diesen Angriff ist von O(k), wobei *k* die Anzahl der If-Anweisungen darstellt. Der Parameter k in diesem Beispiel ist frei wählbar, da beliebig viele überflüssige If-Anweisungen (die durch Benutzung entsprechender Variablen auch nicht wegoptimiert werden können) eingefügt werden können.

Iterative Verbesserung des Verfahrens

Das Problem bei diesem Beispiel ist, dass die Variablenwerte, von denen der Programmfluss abhängt, sehr einfach sind, und dass sie literal im Code stehen. Ein verbessertes Verfahren müsste daher komplexere Ausdrücke verwenden, die zusätzlich noch von Werten abhängig sein könnten, die erst zur Laufzeit zur Verfügung stehen (z.B. durch nachfolgendes “Externe Schlüssel”-Verfahren).

Externe Schlüssel

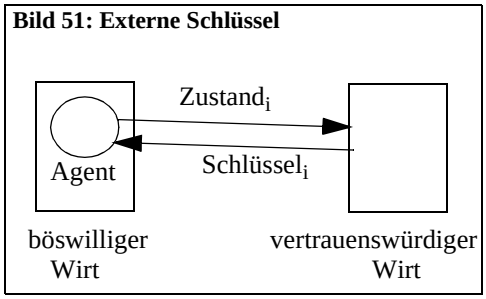


Bild 51: Externe Schlüssel

Wenn der Schutz eines Agenten nur von Informationen abhängt, die komplett im Agenten enthalten sind, ist ein Angreifer in der Lage, den Agenten zu analysieren, sobald er auf dem Wirt angekommen ist. Wenn wir aber z.B. Teile des Agenten verschlüsseln können, oder kleine Informationen finden, die unerlässlich dazu sind, den Agenten zu analysieren, können wir die Schlüssel bzw. diese Informationen auf externen, vertrauenswürdigen Stellen ablegen. Der Agent kann diese Informationen dann bei Bedarf unter Angabe des aktuellen Zustands von dieser vertrauenswürdigen Stelle anfordern (siehe Bild 51). Die vertrauenswürdige Stelle prüft, ob der Agent diese Informationen in diesem Zustand benötigt, und gibt sie erst dann frei. Das Resultat dieses Verfahrens ist, dass der Wirt erst dann in der Lage ist, Codeteile zu analysieren, wenn die entsprechenden Informationen von der vertrauenswürdigen Stelle freigegeben wurden. Ein Beispiel für Informationen, die sich so ablegen lassen, stellen die fett gedruckten Werte in Bild 50 dar. Diese Werte verbinden die If-Anweisungen und konstituieren erst den möglichen Kontrollfluss. Sind diese Werte nicht im Agenten enthalten, kann der Angreifer den Agenten nicht vollständig analysieren und den Agenten nicht vor der Ausführung angreifen.

Analyse des erzielten Schutzes

Der gewünschte Effekt wird erreicht, sofern sichergestellt werden kann, dass der Wirt nicht die Zustände errechnen kann, die dazu führen, dass die vertrauenswürdige Stelle die Schlüssel bzw. Informationen freigibt. Aber die Frage ist, was damit gewonnen ist. Sobald ein Abschnitt vom Wirt ausgeführt werden soll, müssen alle notwendigen Daten vorliegen. Danach kann die Analyse dieses Abschnittes sofort beginnen. Spätestens bei Beendigung der Ausführung liegen dem Wirt alle Informationen über die Teile des Agenten vor, die er ausgeführt hat. Die Brechbarkeit dieses Ansatzes kann zwar errechnet werden, aber der Wirt muss diesen Angriff eigentlich nicht durchführen, um seine Angriffsziele zu erreichen. Damit ist $e(k) = \#o$, wobei $\#o$ die Anzahl der ausgeführten Operationen bis zum Erreichen der gewünschten Information ist

Verstecken von If-Anweisungen

Anweisungen im Ursprungsprogramm wie `if` und `while`, die den Kontrollfluss betreffen, stellen eine große Herausforderung in Bezug auf den Schutz des Agenten

dar, da sie sehr deutlich Stellen im Code markieren, an denen Entscheidungen gefällt werden, auch im Bezug auf Sicherheit. Wenn ein Agent beispielsweise entscheidet, ob er einem Wirt vertrauen will oder nicht, kulminiert ein unter Umständen komplexes Verfahren in einer `if`-Anweisung. Selbst wenn ein Angreifer die Bedingung dieser Anweisung nicht lesen kann, so genügt doch schon eine Manipulation des binären Kontrollflusses, um, unabhängig von der korrekten Auswertung der Bedingung, das Entscheidungsverfahren auszuhebeln. Ein AVV sollte sich daher mit der Frage beschäftigen, wie diese Kulminationspunkte versteckt bzw. vor inkorrekter Ausführung geschützt werden können.

Eine Möglichkeit, dies zu erreichen, besteht darin, eine If-Anweisung im Originalprogramm durch mehrere, verschachtelte If-Anweisungen zu ersetzen. Bild 52 zeigt eine If-Anweisung des Ursprungsagenten, die in eine Endlosschleife eintritt, wenn `bedingung` eintritt. Wir nehmen an, dass die Endlosschleife so komplex ist, dass sie vom Angreifer im Programm nicht als solche erkannt wird. Der Angreifer sieht aber nach einiger Zeit, dass sich der Agent in einer solchen Endlosschleife befindet und geht die If-Anweisungen im Programm solange durch, bis er eine findet, so dass der Agent bei Evaluierung zu “false” oder “true” nicht mehr in die Endlosschleife gerät.

Bild 52: Originale Anweisung

```
10 if (bedingung) then do loop
```

Um dieses Vorgehen zu erschweren, wurde in Bild 52 die originale If-Anweisung durch drei andere ersetzt. Wendet der Angreifer nun das gleiche Verfahren wie bisher an, findet er keine einzelne If-Anweisung mehr, deren Umgehung den Agent nicht in die Endlosschleife geraten lässt. Wenn in Bild 53 die If-Anweisung in Zeile 10 auf true gesetzt wird, tritt die Endlosschleife in Zeile 10 ein. Wenn die Anweisung auf false gesetzt wird, tritt die Endlosschleife in Zeile 11 ein. Erst durch eine Kombination von true- und false-Werten bei diesen drei Bedingungen kann die Endlosschleife umgangen werden.

Bild 53: Ersetzung

```
bedingung1 = bedingung;
bedingung2 = not(bedingung1)
10 if (bedingung1) then do loop
11 if (not(bedingung2)) then a=true
12 if (bedingung1 == a) then
    <do nothing>
else
   do loop
endif
```

Analyse des erzielten Schutzes

Wenn der Angreifer weiter versucht, durch “An-” bzw. “Abschalten” von Kombinationen von If-Anweisungen den gewünschten Effekt zu erzielen, ist der Aufwand zum Brechen dieses Verfahrens von $O(2^k)$, wenn k die Anzahl der eingefügten If-

Anweisungen ist. Leider gibt es für den Angreifer eine Reihe weiterer Möglichkeiten, den Agenten nicht in die Endlosschleife geraten zu lassen, die unter Umständen von geringerer Komplexität sind. Er könnte z.B. berechnen, welche If-Anweisungen von welchen Bedingungen abhängig sind, und dann das Auftreten dieser Bedingung zu “true” oder “false” evaluieren.

Iterative Verbesserung des Verfahrens

Statt direkt im Code Maßnahmen gegen den Wirt zu ergreifen, wenn ein Angriff erkannt wird, und damit dem Angreifer die Möglichkeit zu geben, diese Maßnahmen zu unterlaufen, könnte ein Agent seinen aktuellen Zustand an eine vertrauenswürdige, externe Stelle melden, die genug Wissen hat, um aus diesem Zustand auf einen Angriff schließen und dann die geeigneten Maßnahmen ergreifen zu können. Damit der Wirt nicht schon an der Tatsache, dass der Agent mit einem externen Partner kommuniziert, erkennt, dass ein Angriff erkannt wurde, ist es notwendig, dass der Agent an dieser Stelle immer mit dem Partner kommuniziert, egal ob ein Angriff gemeldet wird oder nur zufällige Daten versendet werden. Ein solches Verfahren kann in das Protokoll zur Verhinderung von Blackbox-Tests integriert werden, das in Kapitel 9 vorgestellt werden wird.

Kombination einzelner AVVs

Auch wenn die obigen Verwürfelungsverfahren nicht hart genug für einen Einsatz sind, zeigen sie doch zumindest, dass es nicht unbedingt ein einziges AVV geben muss, dass den kompletten Schutz eines mobilen Agenten zu gewährleisten hat. Statt dessen kann eine *Kombination* existierender AVVs dazu beitragen, dass der erzielte Schutz sogar größer ist als die Summe des Schutzes jedes einzelnen Verfahrens, zumal die Reihenfolge und die Frage der benutzten Verfahren als weiteres zufälliges Element eine Analyse erschweren kann.

8.7.5 Weitere Aspekte des Einsatzes von Verwürfelungsverfahren

Die Existenz von Verwürfelungsverfahren jenseits des Sparschweinmodells ist eine offene Frage, die in dieser Arbeit nicht gelöst werden konnte. Um dennoch die noch fehlenden Mechanismen des Ansatzes des zeitbeschränkten Blackbox-Schutzes zu erarbeiten wurde im Folgenden angenommen, dass solche Verwürfelungsverfahren existieren. Werden zukünftig Verwürfelungsverfahren zur Verfügung stehen, kann dann ein solcher Schutz realisiert werden.

8.7.6 Erzeugung und Wiederaufladung von Blackboxes

Nachdem wir im Folgenden annehmen wollen, dass geeignete Verwürfelungsverfahren existieren, wenden wir uns der Frage zu, wie mithilfe dieser Verfahren

Blackboxes erzeugt werden, und ob und wie man diese auch nach Ablauf des Verfallsdatums “erneuern” kann.

Erzeugung von Blackboxes

Nachdem der Ursprungsagent durch den Eigentümer erzeugt wurde, wird aus der zu benutzenden Angriffsleistung und dem maximalen Angriffsaufwand das Verfallsdatum dadurch berechnet, dass das errechnete Schutzintervall zum Migrationszeitpunkt addiert wird (falls das errechnete Schutzintervall größer oder gleich dem gewünschten Schutzintervall ist). Das Verfallsdatum wird in den Agenten sowie die Marken eingetragen, die dann digital von den Ausstellern signiert werden müssen. Jetzt wird der Agent durch das Verwürfelungsverfahren samt Daten (also auch inklusive der Marken) in die Blackbox umgewandelt. Schließlich wird der Agent über den konstanten Daten vom Eigentümer signiert. Jetzt ist die Blackbox bereit für die Migration.

Erneuern von Blackboxes

Unter Umständen reicht das Schutzintervall eines Agenten nicht aus, um die Aufgabe des Agenten zu erfüllen. Es ist daher die Frage, ob und wie ein Blackbox-geschützter Agent erneuert werden kann. Aufgrund der Natur der Verwürfelungsverfahren könnte ein beliebiger Wirt auch auf einem verwürfelten Agenten eine weitere Verwürfelung anwenden, da zu der Konvertierung keine weitergehende Kenntnis der inneren Semantik des Agenten notwendig ist. Diese zweite Verwürfelung könnte im Prinzip von jedem Wirt durchgeführt werden, der nicht mit einem der möglichen Angreifer kooperiert. Leider gilt das nicht für das Verfallsdatum und die Signatur des Agenten und seiner Marken, die nicht von jedermann erstellt werden dürfen, sondern nur von einer Partei, der der Eigentümer des Agenten vertraut. Ein neues Verfallsdatum bedingt allerdings, dass sich die Identität des Agenten ändern muss, da das Verfallsdatum Teil der Identität sein muss. Weiterhin bedeuten neue Verfallsdaten in den Marken, dass diese durch neue Marken ersetzt werden müssen. Im Falle von Marken, die nicht durch eine dritte Partei ausgegeben werden, müssen daher die alten gegen neue Marken ausgetauscht werden (z.B. die alten digitalen Geldscheine durch neue). Marken, die der Eigentümer erzeugt hat (z.B. Schlüssel), müssen von diesem oder durch von ihm beauftragte Parteien neu erzeugt werden. Eine Alternative zur Neuerzeugung des kompletten Agenten besteht darin, den Zustand eines Agenten in eine neue Agenten-”hülle” zu injizieren, und so einen neuen Agenten zu erzeugen, der den Zustand des alten benutzt. Der Vorteil dieser Alternativer besteht darin, dass er die Verzögerung vermeidet, die notwendig ist, um den alten Agenten neu zu verwürfeln.

8.7.7 Kosten des zeitbeschränkten Blackbox-Schutzes

Die Verwendung des zeitbeschränkten Blackbox-Schutzes erzeugt einen Mehraufwand gegenüber ungeschützten Agenten. Dieser Mehraufwand hängt vor allem von den verwendeten Verwürfelungsverfahren ab und kann daher in dieser Arbeit nicht beziffert werden. Auch hier gilt aber das in Kapitel 5.5 Gesagte, nämlich dass in vielen Fällen eine auf mobilen Agenten beruhende Lösung von den Gesamtkosten her mit einer Client-Server-Lösung konkurriert. Um einen Überblick über die Kosten zu geben, die durch zeitbeschränkten Blackbox-Schutz entstehen, können vier Kostenklassen unterschieden werden.

Kosten bei der Erzeugung eines Agenten

Die ersten Kosten entstehen bei der Konvertierung des Agenten in die Blackbox, also im Wesentlichen durch Anwendung der Verwürfelung. Diese Kosten tragen allerdings nichts zum Mehraufwand bei der Ausführung bei. Vielmehr verzögern sie die Zeit, die zwischen dem Wunsch des Eigentümers und der ersten Migration des Agenten vergeht. Sollte es möglich sein, zwischen einen Mehraufwand bei der Erzeugung und dem bei der Ausführung eines Agenten zu wählen, so sollte letzteres zugunsten von ersterem optimiert werden.

Kosten bei der Übertragung

Diese Kosten entstehen dadurch, dass ein geschützter Agent größer ist als ein ungeschützter, weil er z.B. auch die Bibliotheken transportieren muss, die er nicht vom Wirt benutzen will. Ein Beispiel für eine solche Bibliothek ist das J/Crypto-Paket der Firma Baltimore Technologies, das Implementierungen von DES, RSA, SHA-1 und MD5 bereitstellt und aus etwa 200 KB Java-Code besteht. Ein weiterer Faktor sind die Vergrößerungen des Agenten selbst, die durch die Verwürfelung erzeugt wird.

Kosten zur Ausführungszeit

Die Kosten zur Ausführungszeit ergeben sich aus dem Mehraufwand, der durch den Einsatz der Verwürfelungsverfahren entsteht, aus dem Mehraufwand, der entsteht, wenn nicht die optimierten Programmbibliotheken des Wirts benutzt werden (sondern solche, die der Agent selbst transportiert), und aus dem Mehraufwand, der entsteht, wenn zusätzlich externe Kommunikation zum Schutz mobiler Agenten eingesetzt werden muss (wie dies bei externen Schlüsseln der Fall ist).

“Kosten” die dadurch entstehen, dass Optimierungen nicht benutzt werden können

Aufgrund der Eigenschaften des Blackbox-Schutzes kann es dazu kommen, dass Systemmechanismen nicht benutzt werden können, die dazu dienen, die Effizienz

einzelner Aspekte zu erhöhen. Ist dies der Fall, so müssen die nicht erzielten Ausführungsverbesserungen ebenfalls als Kosten verbucht werden. Ein Beispiel für eine solche Optimierung stellt [HKB97] dar, in dem ein System beschrieben wird, mit dem es möglich ist, einzelne Codeblöcke von benachbarten Servern zu transferieren, anstatt diese vom Ausgangswirt beziehen zu müssen. Da Blackbox-geschützte Agenten nicht modular sind, sondern pro Agent ganz verschiedenen Code benutzen, können Optimierungen wie diese nicht eingesetzt werden.

9 Ein Protokoll zur Verhinderung von Blackbox-Tests

Wie wir in Kapitel 8.3 gesehen haben, lassen sich, aufbauend auf der Blackbox-Eigenschaft, andere Angriffe verhindern. Ein spezieller Angriff gegen Blackbox-geschützte Agenten wurde bisher jedoch noch nicht behandelt. Dieser Angriff verwendet kein Wissen über die innere Struktur von mobilen Agenten und kann daher auch nicht durch die Blackbox-Eigenschaft verhindert werden; im Gegenteil: er sieht einen Agenten ebenfalls als geschlossene Box und versucht lediglich, durch Variation der Eingaben in die Blackbox und Beobachtung der Ausgaben und anderer Effekte aus der Blackbox, Wissen über die innere Struktur abzuleiten. Es ist daher notwendig, diesen Angriff, den Blackbox-Test, durch einen eigenen Mechanismus abzuwehren. Im Prinzip wird dieser Mechanismus auf alle Blackbox-geschützten Agenten anwendbar sein, aber wir werden in diesem Kapitel annehmen, dass das Time-limited-Blackbox-Verfahren verwendet wird. Diese Annahme erweist sich als vorteilhaft, da mit der Existenz eines Verfallsdatums klar ist, wann welche Daten zu löschen sind.

Auch in diesem Kapitel verwenden wir wieder die Notation von vertrauenswürdigen und nicht-vertrauenswürdigen Wirten. Nicht-vertrauenswürdige Wirte greifen den Agenten unter Umständen an, während vertrauenswürdige Wirte keine Angriffe starten. Agenten werden auf vertrauenswürdigen Wirten maximal einmal (at most once) gestartet, da Fehler oder Zugriffskontrollmechanismen einen Agenten beenden bzw. dessen Ausführung verhindern können. Nicht-vertrauenswürdige Wirte dagegen können versuchen, Agenten mehrere Male auszuführen, um so durch Blackbox-Tests Informationen über die innere Struktur des Agenten zu erhalten.

Teile diesen Kapitels wurden [HR99] entnommen.

9.1 Blackbox-Test-Angriffe

Ein Blackbox-Test ist ein Angriff gegen einen mobilen Agenten durch einen Wirt, bei dem der Agent mehrere Male mit variierenden Eingabeparametern ausgeführt wird. Dies kann parallel oder sequentiell geschehen. Nach jeder Ausführung beobachtet der Angreifer den Effekt des Tests. Diese Effekte können in expliziten Resultaten wie z.B. Ausgabedaten bestehen, oder in charakteristischen “Aktivitätsmustern”. Das Ziel eines Blackbox-Tests ist es, die Eingabeparameter zu finden, die zu einem bestimmten Effekt führen, oder aber bestimmte Eigenschaften des Agenten zu erfahren. Das folgende Beispiel soll einen solchen Angriff illustrieren.

9.1.1 Ein Beispiel

Als Beispiel soll uns ein Agent dienen, der eine Reise buchen soll, die aus einer Buchung eines Fluges, der Reservierung eines Hotelzimmers, sowie dem Mieten eines Autos besteht. Die Datenstrukturen und die Prozedur `start()`, die bei der Ankunft auf einem Wirt aufgerufen wird, könnten folgendermaßen aussehen (Prozeduren, die die Migration und andere Aspekte verwirklichen, wurden aus Gründen der Übersichtlichkeit weggelassen):

```
1    public class TourBookingAgent {
2    private Price maxhotelprice = $200;
3    private Price hotelprice;
4    private Data hoteldata = “St.Malo-Fra,b,tv,lbed:5.10.00,6.10.00,7.10.00”;
5    private Record hotelbooking, carbooking, flightbooking;
6    private Data hotelinformation;
7    private Price maxcarflightprice = $1000;
8    private Data flightdata = “STR-Ger,STM-Fra,5.10.00,Arrival:10.00,Business”;
9    private Data cardata = “St.Malo-France, 5.10.00, BMW”;
12   public start() {
13       ServiceProvider sp = node.getProviderFor(hoteldata);
14       hotelinformation = sp.getGeneralInformation();
15       hotelprice = sp.getOffer();
16       if (hotelprice <= maxhotelprice) {
17           hotelbooking = sp.book(hotelprice); }
19       ServiceProvider spc = node.getProviderFor(cardata);
20       Price carprice = spc.getOffer();
21       ServiceProvider spf = node.getProviderFor(flightdata);
22       Price flightprice = spf.getOffer();
24       if ((carprice + flightprice) <= maxcarflightprice) {
25           carbooking = spc.book(carprice);
26           flightbooking = spf.book(flightprice); }
27   } }
```

In diesem Agent bestimmt die Variable `maxhotelprice` den maximalen Betrag, den der Agenteneigentümer bereit ist, für die Reservierung des Hotelzimmers zu zahlen. Wenn wir annehmen, dass der Angreifer in der Lage ist, den Wert dieser Variablen in Erfahrung zu bringen, kann er dieses Wissen benutzen, um den Preis für das Zimmer zu maximieren, indem er dem Agenten das Zimmer zu einem Preis anbietet, der nur wenig unter dem maximalen Betrag liegt (auch wenn der tatsächliche Preis darunter liegt). Wenn der Angreifer den Code des Agenten kennt, weiß er, dass wenn das Angebot durch den Agenten akzeptiert wird, der Agent die Prozedur `book()` aufrufen wird. Daher kann der Angreifer nun einen Blackbox-Test starten, der die Hauptprozedur mit immer geringeren Preisen für ein Hotelzimmer aufruft,

bis der Agent `book()` aufruft. Der Test kann dabei immer bei Zeile 19 gestoppt werden, d.h. die Prozedur muss nicht einmal vollständig ausgeführt werden.

9.2 Verhindern von Blackbox-Tests

Um einen Mechanismus zu erarbeiten, der Blackbox-Tests verhindert, sehen wir uns zunächst ein erweitertes Ausführungsmodell eines Agenten an. In Erweiterung des Modells von Kapitel 7.1 modelliert es die Ausführung bezüglich der Ein- und Ausgaben *innerhalb* einer Ausführungssitzung genauer.

9.2.1 Ausführungsmodell

Auch in diesem Modell besteht die Gesamtausführung des Agenten aus Ausführungssitzungen auf mehreren Wirten, zwischen denen der Agent migriert. Ein Agent kann dabei einen Wirt durchaus mehrere Male im Verlauf der Gesamtausführung besuchen. Um verschiedene Ausführungssitzungen auf demselben Wirt zu unterscheiden, wird im Folgenden der Begriff des *Migrationszählers* (engl. hop count) verwendet, der vor jeder Migration inkrementiert wird.

Eine Ausführungssitzung auf einem Wirt überführt eine initiale Agenteninstanz durch Einbeziehung einer Liste von Eingabeereignissen in eine finale Agenteninstanz, wobei eine Liste von Ausgabeereignissen erzeugt wird. Da sich bei einer Agenteninstanz nur der variable Zustand des Agenten über der Zeit ändert, lassen sich diese Instanzen durch Zustände beschreiben (siehe Bild 54). Nachdem der finale Zustand (S_{final}) erreicht wurde, ist die Ausführungssitzung beendet, und der Agent terminiert oder migriert auf den nächsten Wirt.

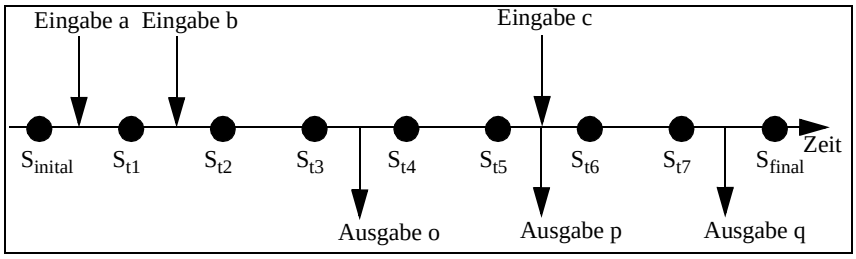


Bild 54: Ausführungsmodell auf einem Wirt

Zu jedem Zeitpunkt t_i kann der Angreifer entscheiden, ob der aktuelle Zustand S_{ti} verschieden vom vorherigen Zustand ist. Während der Ausführungssitzung kann ein Eingabeereignis auftreten, durch das der Agent einen Eingabewert x als Parameter bekommt. Eingaben können aus externer Kommunikation mit dem Wirt oder dritten Parteien bestehen, oder aus Systembibliotheksaufrufen durch den Agenten (siehe

Kapitel 7.1 für eine Diskussion dieses Aspektes). Wann immer der Angreifer eine Aktion des Agenten außerhalb der Blackbox beobachten kann, stellt dies ein Ausgabeereignis dar, das manchmal mit einem Wert y verknüpft ist. Wenn ein Agent aktiv um Eingaben bittet, z.B. durch ein “getNextMessage”-Befehl, stellt dies ebenfalls eine solche Aktion und damit ein Ausgabeereignis dar. Daher tauchen bisweilen Eingabe- und Ausgabeereignisse zusammen auf. Wenn ein Agent Eingaben erhält, ohne diese angefordert zu haben (z.B. indem ein anderer Agent eine Prozedur des ersten Agenten entfernt aufruft), kommen Eingabeereignisse ohne Ausgabeereignis vor.

9.2.2 Die Idee

Ein Blackbox-Test-Angriff besteht darin, dass der Angreifer dieselbe Agenteninstanz nimmt und in mehreren Ausführungssitzungen nur die mit den Eingabeereignissen assoziierten Werte variiert. Ein vertrauenswürdiger Wirt dagegen wird maximal eine Agenteninstanz unter Verwendung einer Liste von Eingabeereignissen verarbeiten.

Die Möglichkeit, Blackbox-Tests allein durch Mechanismen innerhalb des Agenten zu verhindern, besteht deshalb nicht, weil ein solcher Angriff einen Agenten wieder vom initialen Zustand aus startet, und in diesem Zustand keine Information aus der Ausführungssitzung, die danach folgt, vorhanden sein kann.

Ein Weg, Blackboxtests zu verhindern, besteht darin, das Vorkommen von mehr als einer Ausführungssitzung zu verhindern, die die gleiche Migrationsnummer und den gleichen Wirt-Identifikator (Wirt-Id) besitzt. Da eine solche Migrationsnummer und der Identifikator des nächsten Wirts innerhalb des Agenten erzeugt werden und der Agent über die Blackbox-Eigenschaft im Inneren nicht manipulierbar ist, bezeichnet eine Kombination aus Migrationsnummer und Wirt-Identifikator, also `(HopCount,HostId)`, eine Ausführungssitzung. Wenn es gelingt, diese Kombination pro Ausführungssitzung bei einer vertrauenswürdigen Registratur durch sichere Kommunikation zu speichern bzw. deren Gültigkeit zu erfragen, wäre das Problem gelöst. Um Wiederholungsangriffe zu verhindern, würde eine solche sichere Kommunikation mit den heutigen Mitteln die Möglichkeit zur Erzeugung von nicht-deterministischen Zufallszahlen auf der Seite des Agenten benötigen. Die Erzeugung von nicht-deterministischen Zufallszahlen in einer nicht-vertrauenswürdigen Umgebung ist aus heutiger Sicht aber ohne den Einsatz sicherer Hardware sehr schwierig (wenn nicht gar unmöglich), da der Erzeugungsprozess Werte oder Ereignisse benötigt, die von der nicht-vertrauenswürdigen Ausführungsumgebung geliefert werden müssen.

Ein zweiter Weg, Blackbox-Tests zu verhindern besteht darin, Ausführungssitzungen mit der gleichen `(HopCount,HostId)`-Kombination zwar zu erlauben,

gleichzeitig aber zu verlangen, dass sie dieselbe Liste von Eingabeereignissen benutzen. Damit können zwar weiterhin mehrere Ausführungen derselben Agenteninstanz vorkommen. Da diese jedoch dieselbe Eingabe bekommen, verhalten sie sich auch vollkommen identisch, falls sie deterministisch sind. Damit kann ein Angreifer keinen Informationsgewinn aus Blackbox-Tests bekommen, und dieser Angriff ist sinnlos. Um sicherzustellen, dass zur Ausführung einer Agenteninstanz auf einem Wirt und unter derselben Migrationsnummer dieselbe Liste von Eingabeereignissen benutzt wird, benötigen wir eine vertrauenswürdige Komponente. Diese Komponente, die *Registratur*, muss daher auf einem vertrauenswürdigen Wirt platziert werden. Das nun folgende Protokoll realisiert diese Idee.

9.3 Das Protokoll

Bevor wir zum Protokoll zwischen Agent und Registratur kommen, untersuchen wir nun kurz, welche Vorkehrungen auf Seiten des Agenten getroffen werden müssen.

9.3.1 Vorkehrungen auf Seiten des Agenten

Auf der Seite des Agenten müssen wir in der Lage sein, einzelne Eingabeereignisse zu identifizieren und gleiche Listen von Eingabeereignissen zu unterscheiden, die verschiedene assoziierte Werte besitzen. Solch ein Identifikationsschema kann wie folgt realisiert werden. Wir nehmen an, dass jede Eingabeoperation einen Identifikator besitzt, der diese Operation innerhalb des Agentenprogramms eindeutig identifiziert. Dies kann z.B. dadurch erreicht werden, dass jeder Eingabeoperation eine Sequenznummer auf der Basis der statischen Zeilennummern im Quellcode erhält. Im Kontext einer Programmausführung kann dann ein Eingabeereignis durch eine Kombination (`i_stmt#`, `n`) identifiziert werden, bei der n die n-te Ausführung dieser Operation darstellt, die durch `i_stmt#` identifiziert wird.

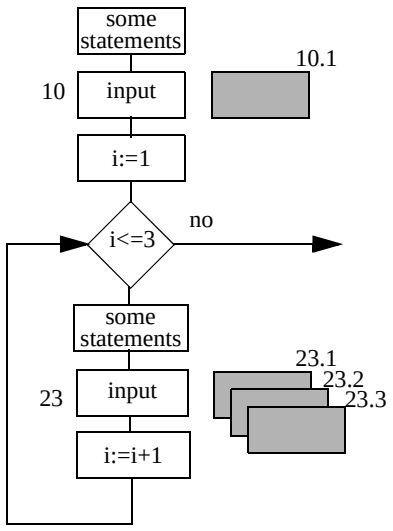


Bild 55: Schema zur Identifikation der Befehle

In Bild 55 gibt es zwei Eingabeoperationen, die durch die Zeilennummern 10 und 23 identifiziert werden. Die Eingabeoperation 10 wird einmal ausgeführt und

erzeugt ein Eingabeereignis 10.1, während die Eingabeoperation 23 innerhalb der Schleife dreimal ausgeführt wird, und somit die Ereignisse 23.1 bis 23.3 erzeugt.

Wenn nun ein Angreifer eine Agenteninstanz mehr als einmal ausführt, erfahren die verschiedenen Ausführungen Eingabeereignisse mit demselben Identifikator. Zwei Eingabeereignisse zweier Ausführungen einer Agenteninstanz werden daher als korrespondierend bezeichnet, wenn sie denselben Identifikator haben. Wenn das Beispielprogramm aus Bild 55 *n*-mal ausgeführt wird, gibt es maximal *n* korrespondierende Eingabeereignisse für jede Eingabeoperation.

Eine Eingabeoperation kann daher als *geschützt* betrachtet werden, wenn zu Eingabeereignissen dieser Operation korrespondierende Ereignisse dieselben Eingabewerte liefern. Der Agent ist daher vor Blackbox-Tests dann geschützt, wenn alle Eingabeoperationen geschützt sind (wie wir noch sehen werden, ist diese Forderung zu stark; es reicht, wenn einige Eingabeoperationen geschützt sind).

Ein Agent beinhaltet u.a. die folgenden Daten:

- AgentId: Der Identifikator des Agenten
- Exp: Das Verfallsdatum des Agenten
- LocId: Der Identifikator des aktuellen Wirts
- Hop: Der Migrationszähler
- Der öffentliche Schlüssel einer Zertifizierungsstelle
- Reg: PlaceId --> (RegistryId;PubKey)
  Eine Liste mit Zuordnung von Wirt-Identifikatoren zu Registraturen samt deren öffentlicher Schlüssel

Eine Kombination (`i_stmt#`,`n`) wird *lokaler Eingabeereignis-Identifikator (Local_InteractionId)* genannt, da er ein Eingabeereignis im Kontext einer Ausführungssitzung eindeutig identifiziert. Ein Eingabeereignis wird global durch ein 4-Tupel (`AgentId`, `Exp`, `VisitId`, `Local_InteractionId`) identifiziert. `VisitId` besteht aus einer Kombination (`LocId`,`Hop`) und identifiziert eine Ausführungssitzung des Agenten `AgentId` auf `LocId`. Daher wird dieses 4-Tupel als *globaler Eingabeereignis-Identifikator (GID)* bezeichnet. Die Liste `Reg` schränkt in dieser Form die Migration auf solche Wirte ein, die einen Eintrag in dieser Liste besitzen. Mit Maßnahmen, wie sie in Kapitel 7.8.3 vorgestellt wurden (Agent bittet Heimatwirt um weitere Einträge oder ist in der Lage, Einträge selbst zu berechnen), lässt sich diese Einschränkung aber aufheben.

9.3.2 Protokoll zwischen Agent und Registratur

Um das Protokoll zu realisieren, registriert ein Agent Eingabeereignisse bei einer Registratur. Die Registratur antwortet auf eine Registrierung genau dann mit einer positiven Antwort, falls ein solches Ereignis in einer solchen Ausführungssitzung

bisher noch nicht vorgekommen ist oder das Ereignis mit denselben Eingabewerten bereits registriert wurde. Ein Agent setzt seine Ausführung nur dann fort, wenn er eine positive Antwort auf einen Registrierungsversuch bekommt. Registraturen sind auf vertrauenswürdigen Wirten lokalisiert, und jeder vertrauenswürdige Wirt kann ein oder mehrere Registraturen aufnehmen. Mithilfe der Liste `Reg` aus dem letzten Abschnitt ist gewährleistet, dass ein Agent während einer Ausführungssitzung immer dieselbe Registratur verwendet.

Bild 56 stellt das Registrierungsprotokoll für Eingabeereignisse dar:

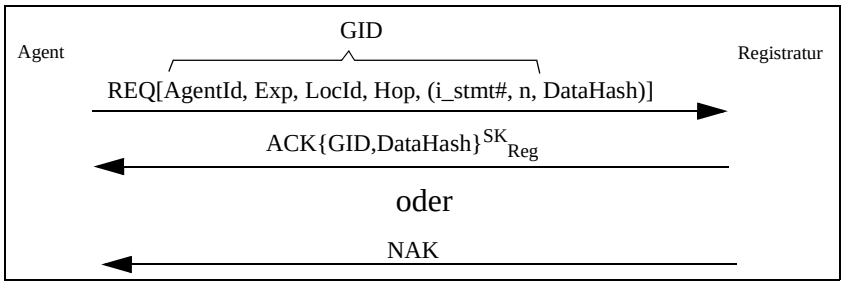


Bild 56: Registrierungsprotokoll

Die Registrierungsanforderung, REQ, wird an die Registratur geschickt. Sie enthält den globalen Eingabeereignis-Identifikator (GID), sowie den Hash der Eingabewerte, `DataHash`. Der Hash-Wert wird als eine Art spezialisierte Zufallszahl verwendet, die charakteristisch für die Eingabewerte ist. Daher darf es nicht viele verschiedene Eingabewerte geben, die im selben Hash-Wert resultieren, zumindest sollten die anderen Eingabewerte schwierig zu berechnen sein. Diese Anforderungen erfüllt der Einsatz sicherer Hash-Verfahren wie MD4 oder MD5 zur Errechnung des Hash-Wertes.

Nach Eingang der Anforderung entscheidet die Registratur, ob sie eine positive oder negative Antwort schicken soll. Im ersten Fall wird eine positive Bestätigung (ACK) zurückgeschickt, die dieselben Daten wie die Anforderung enthält und durch die Registratur signiert wurde. Im zweiten Fall wird eine einfache negative Antwort (NACK) zurückgeschickt.

Weder die Registrierungsanforderung noch die negative Antwort werden geschützt. Die Anforderung muss nicht geschützt werden, da der Agent die Integrität dieser Anforderung dadurch überprüfen kann, dass er die Signatur der positiven Antwort verifizieren kann bzw. dann den Inhalt der Antwort mit seiner Anforderung vergleicht. Gleichzeitig authentifiziert die Signatur auch die Registratur. Jeder Angrei-

fer, sei es der Wirt, oder eine andere Partei, kann die Anforderung auf eine Weise modifizieren, dass die Registratur die Anforderung negativ beantwortet und somit der Agent seine weitere Ausführung abbricht. Dies jedoch ist zumindest für den Wirt immer eine Option (z.B. durch die Unterdrückung jeglicher Interaktion des Agenten). Die negative Antwort wird aus denselben Gründen nicht geschützt. Wenn „Denial-of-Service“-Angriffe durch *andere* Parteien als den Wirt verhindert werden sollen, muss sich der Agent gegenüber der Registratur authentifizieren und alle Antworten müssen von der Registratur signiert werden.

9.3.3 Vorkehrungen für die Registratur

Die Aufgabe der Registratur ist es, zu entscheiden, welche Antwort auf eine Registrierungsanfrage zu erfolgen hat. Eine positive Antwort wird dabei zurückgegeben, wenn:

- das Verfallsdatum (`Exp`) der Anfrage noch nicht abgelaufen ist und
- es keinen Eintrag mit demselben globalen Eingabeereignis-Identifikator und einem anderen Hash-Wert (`DataHash`) gibt

Diese Strategie bedingt die Speicherung aller akzeptierten Anfragen bis es das Verfallsdatum erlaubt, diese Einträge wieder zu löschen. Das Verfallsdatum bestimmt also, wann ein Eintrag in der Registratur gelöscht werden kann. Daher muss es auf einen Wert gesetzt werden, der garantiert, dass der Angreifer nach diesem Datum den Agenten nicht mehr angreifen kann. Die Bestimmung dieses Wertes hängt damit direkt mit der Frage zusammen, wie lange ein Agent im System gültig ist. Wenn ein Agent unbegrenzte Zeit gültig ist, z.B. weil der Blackbox-Schutz keine Zeitrestriktionen besitzt, kann eine Registratur keinen Eintrag vergessen, da, selbst wenn der Agent bereits terminiert ist, ein Angreifer eine lokal gespeicherte Kopie benutzen könnte. Im Fall der Benutzung eines durch das Zeitbeschränkte-Blackbox-Verfahren geschützten Agenten ist ein solches Verfallsdatum bereits Teil des Agenten, da das Verfahren einen Schutz nur bis zum Verfallsdatum gewährleisten kann.

9.3.4 Authentifikation

Um trotz des Registrierungsprotokolls Blackbox-Tests durchführen zu können, könnte ein angreifender Wirt sich bei einer anstehenden Migration als nächster Wirt maskieren, mit dem Effekt, dass der Agent ein zweites Mal (und ein drittes Mal, usw.) ausgeführt werden kann. Um dies zu verhindern, muss daher eine Zweiwege-Authentifizierung durchgeführt werden, wie sie in Kapitel Kapitel 8.3 vorgestellt wurde.

9.4 Programmierschnittstelle

Die Frage, wann ein Eingabeereignis registriert werden soll, kann auf zwei Arten beantwortet werden. Eine Möglichkeit besteht darin, dies unmittelbar nach der Ausführung einer Eingabeoperation zu tun (d.h. sobald die Eingabewerte übermittelt wurden). Dies kann durch automatische Verfahren realisiert werden, z.B. durch Instrumentierung des Agentencodes, also die automatische Einfügung von Befehlen zur Registrierung nach Eingabeoperationen. Damit kann ein Angreifer keine Informationen aus der Agentenausführung jenseits derjenigen extrahieren, die durch eine einzige, reguläre Ausführung resultieren, weil eine zweite Ausführung mit anderen Daten abgeblockt wird, bevor eine Reaktion auf diese Eingabedaten erfolgen kann. Leider bedingt diese Lösung den Transfer einer Anfrage- und einer Antwortnachricht pro Eingabeoperation. Wenn wir aber unser Beispiel untersuchen, stellen wir fest, dass es nicht notwendig ist, nach jeder Eingabe zu registrieren:

```
12    public start() {
13        ServiceProvider sp = node.getProviderFor(hoteldata);
14        hotelinformation = sp.getGeneralInformation(NotProtected);
15        hotelprice = sp.getOffer(Protected);
16        if (hotelprice <= maxhotelprice) {
17            hotelbooking = sp.book(hotelprice); }
19        ServiceProvider spc = node.getProviderFor(cardata);
20        Price carprice = spc.getOffer(ProtectedExplicit);
21        ServiceProvider spf = node.getProviderFor(flightdata);
22        Price flightprice = spf.getOffer(ProtectedExplicit);
23        register();
24        if ((carprice + flightprice) <= maxcarflightprice) { ... } }
```

Eingabeoperationen kommen u.a. in den Zeilen 14, 15, 20 und 22 vor. Ein Angreifer könnte die Reaktion des Programms auf die Eingabe in Zeile 15 benutzen, um den Wert der Variablen `maxhotelprice` zu erfahren, daher muss die Eingabeoperation in Zeile 15 auf jeden Fall durch eine Registrierung geschützt werden. Der Angreifer kann außerdem die Eingaben in Zeile 20 und 22 benutzen, um den Wert der Variablen `maxcarflightprice` zu erfahren. Daher müssen auch diese Eingabeoperationen geschützt werden. Im Gegensatz zu diesen Operationen kann die Eingabeoperation in Zeile 14 nicht dazu benutzt werden, Informationen über den Agenten zu erfahren, da der Eingabewert nur gespeichert, aber nicht verarbeitet wird. Daher muss diese Operation auch nicht geschützt werden.

Der Programmierer kann durch Anfügen eines *Protected*-Attributes spezifizieren, welche Eingabeoperationen geschützt werden sollen. Geschützte Eingabeoperationen werden implizit registriert, d.h. im Agentencode ist dafür kein Registrierungsaufruf notwendig. Die implizite Registrierung wird nach der Eingabeoperation

durchgeführt, da ab diesem Zeitpunkt Effekte wie Aktivitätsmuster oder Ausgabeereignisse beobachtet werden könnten.

In einigen Situationen ist es effizienter, die Registrierung eines Eingabeereignisses zu verschieben, und den Programmierer entscheiden zu lassen, wann diese Registrierung durchgeführt werden soll. In unserem Beispiel ist das in Zeile 20 und 22 möglich, da eine Reaktion auf diese Eingaben erst in Zeile 24 erfolgt, ein Angreifer vorher also keine Informationen über den Agenten ableiten kann. In diesem Fall wird den Eingabeoperationen das Attribut *ProtectedExplicit* angefügt. Wenn eine solche Eingabeoperation ausgeführt wird, wird die Registrierungsanforderung nur lokal (d.h. im Agenten oder im Wirt) gespeichert. Der Programmierer kann dann bestimmen, wann die eigentliche Registrierung durchgeführt wird, indem er die *register*-Operation aufruft. Wenn diese Operation ausgeführt wird, werden alle bis dahin lokal gespeicherten Registrierungsanforderungen in einer Nachricht an die entsprechende Registratur gesandt, die wiederum mit einer positiven oder negativen Nachricht antwortet. Zu diesem Zweck muss die Anforderungsnachricht so erweitert werden, dass sie mehrere lokale Eingabeereignis-Identifikatoren enthalten kann.

9.5 Protokollimplementierung

Das Protokoll wurde als reine Java-Anwendung implementiert (siehe [Fri98] für eine genauere Beschreibung der Implementierung). Die Implementierung besteht aus zwei Teilen: dem durch ein Blackbox-Verfahren geschützten Agenten und der Registratur. Beide Komponenten wurden als eigenständige Anwendungen realisiert; die Kommunikation zwischen ihnen erfolgt durch Java RMI. Die Hauptverarbeitungsschritte, die bei einer Registrierungsanfrage abgearbeitet werden, sind in Bild 57 dargestellt.

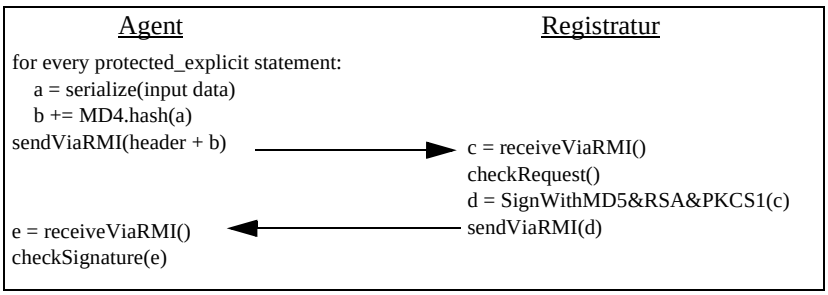


Bild 57: Hauptverarbeitungsschritte

Zur Bildung der Hash-Werte der Eingabedaten wurde dabei MD4 verwendet, zur Signierung der Antworten der Registratur wurde eine Kombination aus MD5, RSA

und PKCS1 mit einer Schlüssellänge von 1024 Bit verwendet. Unter Verwendung dieser Implementierung wurde der Mehraufwand gemessen, der durch die Verwendung des Protokolls entsteht. Zu diesem Zweck wurde der Agent auf einem Rechner lokal platziert, während die Registratur ca. 100 km entfernt installiert wurde, wobei beide Rechner über eine WAN-Verbindung (B-WIN) vernetzt waren. Die Umlaufzeit dieser Verbindung lag während der Messung etwa bei 7 ms (gemessen mit dem Unix-Werkzeug *ping*), die Anzahl der Router zwischen beiden Rechnern lag bei 10.

Für die Messung wurden drei verschiedene Eingabeobjekte verwendet:

1. ein leerer String mit einer serialisierten Länge von 7 Bytes
2. ein String mit einer serialisierten Länge von 1031 Bytes
3. ein weiterer String mit einer serialisierten Länge von 10025 Bytes

Da Eingabeoperationen, die mit dem `protected`-Attribut versehen sein sollten, durch die Angabe eines `protected_explicit`-Attributes sowie eine anschließende `register`-Operation implementiert wurden, wurden ausschließlich Serien von durch `protected_explicit` und anschließender `register`-Operation geschützten Eingabeoperationen gemessen. Die erste von drei gemessenen Eingabeserien bestand aus einem Eingabeobjekt, die zweite aus 5 Eingabeobjekten und die dritte aus jeweils 10 Eingabeobjekten. Jede Serie wurde dabei 100 Mal gemessen. Aus diesen 100 Messungen wurde jeweils der Mittelwert berechnet. Die so erhaltenen Resultate finden sich in Tabelle 4.

Tabelle 4: Mehraufwand für das Protokoll

Eingabeobjekt	Anzahl der Eingabe-operationen mit „protected_ explicit“	**Benutzung des Protokolls in [ms]**	*Benutzung des Protokolls in [ms] korrigierte Werte*	**Alternativ-er Fall in [ms]**	*Alternativer Fall in [ms] korrigierte Werte*
leerer String	1	**192**	*73*	**145**	*23*
leerer String	5	**208**	*86*	**713**	*113*
leerer String	10	**217**	*97*	**1441**	*231*
1k String	1	**217**	*94*	**155**	*34*
1k String	5	**279**	*158*	**816**	*181*
1k String	10	**360**	*238*	**1610**	*350*
10k Object	1	**385**	*259*	**393**	*240*
10k Object	5	**1362**	*1230*	**1951**	*1271*
10k Object	10	**2146**	*2014*	**4127**	*2707*

Die dritte Spalte gibt die Mehrzeiten wieder, die für das Protokoll aufgewendet wurden. Obwohl diese Zeiten absolut gesehen relativ hoch erscheinen, müssen diese gegen den Zeitbedarf von Alternativen verglichen werden.

Für den alternativen Fall wurden folgende Überlegungen mit einbezogen. Wenn wir die Sicherheit des Agenten nicht garantieren können, muss dieser von einem vertrauenswürdigen Wirt aus arbeiten, ohne zu den nicht-vertrauenswürdigen Interaktionspartnern migrieren zu können. In diesem Fall muss die Kommunikation mit diesen Partnern entfernt erfolgen, d.h. z.B. dass Eingabeparameter an den Agenten und Ausgaben von Agenten serialisiert und verschickt werden müssen. Für den benötigten Aufwand bedeutet dies, dass Zeit für Serialisierung der Parameter und deren Transport über das Netz benötigt wird. Falls diese Daten über ein unsicheres Netzwerk transportiert werden, müssen diese Daten bisweilen verschlüsselt, in jedem Fall aber signiert werden, um zumindest die Integrität dieser Daten zu sichern. Also wird auch noch zusätzlich Zeit für die Signierung der Daten benötigt. Die fünfte Spalte enthält die gemessenen Zeiten für diesen alternativen Fall, bei dem die Eingabedaten über RMI transportiert und mit denselben Methoden signiert werden, wie bei der Verwendung des Protokolls.

Da wir aus praktischen Gründen auch für die Alternative eine reine Java-Implementierung verwenden mussten, korrigierten wir die gemessenen Zeiten, die zur Signierung sowohl bei der Registrierung als auch beim Interaktionspartner im alternativen Fall benötigt wurden, indem wir diese Zeiten durch einen Beschleunigungsfaktor (in diesem Fall ein geschätzter Wert von 10*) teilten. Diese korrigierten Werte sollen eine Annäherung an den Fall darstellen, dass die Signierung effizient durch für die Architektur optimierten Code auf Seiten der Registratur bzw. des Interaktionspartners stattfindet. Die so erhaltenen Werte wurden wieder in die Gesamtzeiten eingerechnet. Diese lassen sich in den Spalten 4 und 6 finden.

Die in der Tabelle dargestellten Zeiten erlauben keinen Vergleich der gesamten benötigten Zeit für die Ausführung eines Agenten im Falle der Migration, des Schutzes durch das Blackbox-Verfahrens und des Protokolls und der gesamten Zeit

* Der Beschleunigungsfaktor gegenüber der Implementierung in einer anderen Programmiersprache (z.B. C), die ein Kompilat erzeugt, hängt stark vom Compiler, dem Problem, sowie den Implementierungen ab. Laut Berichten über vergleichende Implementierungen in Java bzw. C ([And97],[Sch97]) oder C++ ([Gal98]) ist ein Java-Programm 1-8 mal langsamer als ein entsprechendes Programm in C bzw. C++. Ein Faktor von 10 scheint als Obergrenze für einen solchen Faktor daher realistisch zu sein.

im alternativen (Client-Server-)Fall. Dieser Vergleich müsste u.a. den zusätzliche Aufwand miteinbeziehen, um den Agenten zu seinem, möglicherweise nicht-vertrauenswürdigen Wirt, zu transportieren. Dieser Aufwand hängt jedoch in starkem Maße von der Größe des Agenten ab, und damit auch von seiner konkreten Funktionalität. Was ebenfalls noch in die Analyse miteinbezogen werden müsste, ist der zusätzliche Aufwand für die Benutzung des Blackbox-Verfahrens, z.B. für die Verwürfelung des Ursprungsagenten sowie den Mehraufwand für die Ausführung der Blackbox.

Wenn man die Zeiten des Mehraufwands des Protokolls gegenüber der Zeit betrachtet, die die Alternative nur für die notwendige zusätzliche Kommunikation benötigt, fällt auf, dass der Unterschied zur Alternative für ein explizit geschütztes Eingabeereignis gering ist (maximal 62 ms) und sich ab 5 explizit geschützten Eingabeereignissen sogar ins Gegenteil umgekehrt, d.h. die Alternative ist in diesen Fällen sogar mindestens 23 ms (bis hin zu einem maximalen Zeitvorteil von ca. 700 ms im korrigierten Fall) langsamer. Dies ist leicht erklärbar, wenn wir uns überlegen, dass das Protokoll normalerweise weniger Daten über das Netz bewegt als der alternative Fall, da das Protokoll nur einen Hash fester Länge von den Eingabedaten berechnet. Sobald die Zeit für die Berechnung des Hash-Wertes plus einiger zusätzlicher Zeit geringer ist als die Zeit, die zum Transport der Daten mit Java RMI benötigt wird, ist der Gebrauch des durch das Protokoll geschützten Agenten schneller als der alternative Fall.

Unter der Annahme, dass die Benutzung von mobilen Agenten gegenüber der Client-Server-Alternative Effizienzvorteile hat, selbst wenn Blackbox-Verfahren benutzt werden (falls es außer aus Effizienzargumenten noch aus anderen Gründen notwendig sein sollte, mobile Agenten zu benutzen, gibt es ja keine Client-Server-Alternative), erzeugt die Benutzung des Protokolls zur Verhinderung von Blackbox-Tests also maximal einen kleinen Mehraufwand. Der dargestellte Ansatz erscheint also bereits in der augenblicklichen Form als für den praktischen Einsatz geeignet.

9.6 Mögliche Erweiterungen

Das Protokoll kann in drei Richtungen erweitert werden. Erstens sollte eine schnellere charakteristische Funktion aus Performanzgründen die Sichere-Hash-Funktion ersetzen. Ein Weg, dies zu erreichen, besteht darin, den Programmierer diese charakteristische Funktion selbst erstellen zu lassen, da dieser am Besten die Struktur der Eingabedaten kennt. Bei kleinen Eingabedaten könnten die Werte dieser Daten sogar selbst als charakteristische Daten verwendet werden. Zweitens sollte die Registratur durch Verwendung von für die Architektur optimiertem Code effizienter gemacht werden. Drittens sollte ein automatischer Mechanismus die manuelle Einfügung von `protected`- und `protected_explicit`-Attributen bei Eingabe-

operationen ersetzen, um so die transparente Verwendung des Protokolls für den Programmierer zu ermöglichen.

Teile diesen Kapitels wurden [HR99] entnommen.

10 Zusammenfassung und Ausblick

10.1 Zusammenfassung

Mobile Agenten sind Programminstanzen, die in der Lage sind, sich selbstständig von einer Ausführungsumgebung (dem *Wirt*) zu einer anderen Ausführungsumgebung zu bewegen. Als Programminstanzen bestehen mobile Agenten aus Code, konstanten und variablen Daten. Während mobile Agenten im Sinne des Programmiermodells selbständige Einheiten sind, sind sie aus Sicht der technischen Umsetzung passive Einheiten, die erst durch die Ausführung durch den Wirt aktiv werden können. Mobile-Agenten-Systeme als Basis für beliebige Anwendungen bieten eine Reihe von Vorteilen wie ein einfacheres Fehlermodell, asynchrones Arbeiten, echte Parallelität, dynamisches automatisches Bewegen von Funktionalität, Ersetzung passiver durch aktiver Elemente, Verringerung der Kommunikationskosten, Erhöhung des Verarbeitungsdurchsatzes und Verminderung von Kommunikationsverzögerungen.

Neben diesen Vorteilen gibt es bei mobilen Agenten jedoch auch noch Problemfelder, die vor einem Einsatz dieser Technologie in der Praxis stehen. Eines dieser Problemfelder umfasst den Bereich der Sicherheit. Nur wenn sichergestellt ist, dass keine der beteiligten Parteien Gefahr läuft, von anderen Parteien angegriffen zu werden, sind die Beteiligten bereit, Agenten einzusetzen oder Agenten auf ihren Rechnern auszuführen. Dies gilt umso mehr, als einer der wichtigsten möglichen Anwendungsbereiche der elektronische Handel ist. Im Gegensatz zu Client-Server-Systemen taucht bei mobilen Agenten ein neues Sicherheitsproblem auf, die Frage, wie Agenten und Wirte voreinander geschützt werden können.

Die möglichen Angriffe eines böswilligen Agenten gegen seinen Wirt umfassen erstens Angriffe gegen den Wirt als Teil des Mobile-Agenten-Systems selbst. Beispiele für solche Angriffe sind das unberechtigte Benutzen von Ressourcen, „Denial-of-Service“-Angriffe gegen Wirte oder Angriffe gegen andere Agenten, die über die Erlangung der Kontrolle über den Wirt führen. Zweitens umfassen die möglichen Angriffe eines böswilligen Agenten Angriffe gegen das dem Wirt zugrundeliegende Rechnersystem. Diese Angriffe benutzen das Mobile-Agenten-System nur als komfortable Angriffsmöglichkeit. Sie umfassen dabei das mögliche Spektrum der Angriffe eines Rechner gegen andere Rechner bzw. eines Benutzers gegen andere Benutzer auf demselben System. Ansätze, die solche Angriffe verhindern sollen, umfassen Mechanismen in den Bereichen der Isolierung einzelner Agenten, der Abschottung der Ressourcen vor dem Zugriff durch den Agenten, die Authentifikation von Agenten, der Ressourcenkontrolle sowie des “Proof-carrying code”. Insgesamt ähnelt der Schutz des Wirts vor Angriffen durch die ausgeführten

Agenten dem Problem des Schutzes der Ausführungsumgebung vor Mobile-Code-Einheiten, etwa Applets. Letzteres Problem umfasst dabei allerdings im Wesentlichen Angriffe gegen das zugrundeliegende Rechnersystem, nicht jedoch solche gegen die Ausführungsumgebung selbst.

Das Problem des Schutzes mobiler Agenten gegen Angriffe durch einen böswilligen Wirt umfasst Leseangriffe (die gegen die Daten des Agenten oder dessen Kommunikation mit Dritten gerichtet sind), Modifikationsangriffe (gegen Daten, Code oder Kommunikation mit Dritten) sowie das inkorrekte Ausführen von Code. Weiterhin umfassen diese Angriffe die Kenntnis der Werte und die Rückgabe falscher Werte beim Aufruf von Systemfunktionen sowie die Möglichkeit, durch Blackbox-Tests Eigenschaften eines Agenten abzuleiten. Diese Angriffe kommen dabei zur Gänze nur dann vor, wenn die Organisationsstruktur des benutzten Agentensystems offen ist, wenn also jede Partei nicht nur Agenten, sondern auch Wirte in das System einbringen darf. Mechanismen, die Angriffe durch einen böswilligen Wirt verhindern, dürfen dabei die Gesamtanwendung nicht zu “teuer” machen. Der Grund hierfür liegt darin zu suchen, dass es in manchen Fällen möglich ist, die durch die Möglichkeit zur Migration entstandene Sicherheitsproblematik (und die damit verbundenen Kosten) durch die Verwendung eines Client-Server-basierten Ansatzes zu umgehen.

Um das Problem des Schutzes mobiler Agenten vor böswilligen Wirten zu veranschaulichen und eine Basis für die Evaluierung von möglichen Schutzverfahren anzubieten, wurde ein neues Modell der Wirtsmaschine und des Angreifers vorgeschlagen. Dieses Modell erlaubt es dem Angreifer, einen Agenten mittels eines Programmes anzugreifen. Dieses Programm ist in der Lage, nicht nur auf den Code und die Daten des Agenten lesend und schreibend zuzugreifen, sondern auch Einfluss auf die Ausführung zu nehmen.

Um die Vielzahl an existierenden Ansätzen zu ordnen, wurde eine Klassifikation erarbeitet. Auf der obersten Ebene besteht diese aus zwei Klassen. Die eine Klasse von Ansätzen versucht, einen mobilen Agenten vor allen Angriffen zu schützen. Auch hier lassen sich die Ansätze wieder unterscheiden, und zwar nach organisatorischen und technischen Ansätzen. Organisatorische Ansätze versuchen das Problem dadurch zu lösen, dass beliebige Agenten nur noch auf vertrauenswürdige Wirte migrieren dürfen. Technische Ansätze versuchen einen technischen Schutz zu erreichen, so dass die Frage der individuellen Vertrauenswürdigkeit nicht mehr gestellt werden muss. Technische Ansätze schließlich lassen sich aufteilen in Hardware-Ansätze, die spezielle, sichere Module zur Ausführung von Agenten verwenden, und Software-Ansätze, die ohne spezielle Hardware auskommen. Die Software-Ansätze beruhen auf der Idee, einen mobilen Agenten in eine “Blackbox” zu verwandeln. Ein mobiler Agent ist eine Blackbox, wenn der Wirt nicht mehr in

der Lage ist, die innere Struktur des Agenten zu analysieren und zu manipulieren. Die andere Klasse von Ansätzen erlaubt einen Schutz des Agenten nur vor einem Teil der möglichen Angriffe. Aus der Menge dieser Ansätze lassen sich die Ansätze zusammenfassen, die, auf verschiedene Weise, einen Zustand zur Erkennung von Modifikationsangriffen und Angriffen nutzen, die den Code inkorrekt ausführen. Neben dieser Gruppe gibt es noch eine ganze Reihe weiterer Ansätze, die sich z.B. mit der Integritätssicherung von konstanten Daten oder Teilresultaten beschäftigen.

Um das Potenzial von Ansätzen zu untersuchen, die die von “außen” sichtbaren Ergebnisse der Ausführung eines Agenten auf einem Wirt zur Entdeckung von Angriffen benutzen können, wurde, ausgehend von einer allgemeinen Definition von Angriffen gegen Agenten durch ihre Wirte, der Begriff des Referenzzustands eingeführt. Dieser ist als der Zustand eines Agenten definiert, der nach der Ausführung durch einen Wirt entsteht, der den Agenten nicht angreift. Mithilfe eines solchen Referenzzustandes ist es dann möglich, auf einem beliebigen Wirt errechnete Endzustände zu prüfen. Es wurden vier existierende Verfahren analysiert und verglichen, die auf dem Prinzip der Referenzzustände beruhen. Es wurde ein neues Verfahren vorgeschlagen. Dieses Verfahren kombiniert Elemente aus drei der existierenden Verfahren und erlaubt es, Referenzzustände durch Nachrechnen zu erzeugen. Unter der Annahme, dass keine zwei aufeinanderfolgenden Wirte konspirieren findet dieses Nachrechnen dabei immer auf dem nächsten Wirt statt, der vertrauenswürdig sein kann, oder auch nicht, wobei das Verfahren in der Lage ist, Prüfungen wegzulassen, wenn Berechnungen auf vertrauenswürdigen Wirten stattfinden. Der Mehraufwand des Verfahrens wurde auf einen Faktor zwei geschätzt. Mithilfe einer Evaluierung dieses Verfahrens wurde diese Schätzung im Wesentlichen bestätigt. Das Verfahren wurde für das Mobile-Agenten-System Mole implementiert, wobei ein ebenfalls neu entwickeltes Framework benutzt wurde, das in der Lage ist, ein breites Spektrum an Verfahren zu unterstützen, die Referenzzustände verwenden. Zuletzt wurde eine Erweiterung des Verfahrens angerissen, die es erlaubt, die Annahme aufzuheben, dass keine zwei aufeinanderfolgenden Wirte konspirieren.

Gegen mobile Agenten sind viele Angriffe durch einen böswilligen Wirt möglich. Um dennoch einen möglichst kompletten Schutz zu erreichen, wurde gezeigt, dass sich die Gesamtheit der Angriffe im Wesentlichen auf eine kleinere Menge an Angriffen reduzieren lässt. Wird diese kleinere Menge von Angriffen verhindert, kann man auch fast alle anderen Angriffe verhindern. Agenten, bei denen diese kleinere Menge an Angriffen verhindert werden kann, werden in dieser Arbeit als “Blackbox-Agenten” bezeichnet. Dieser Begriff bezeichnet den Umstand, dass Angreifer bei solchermaßen geschützten Agenten nicht mehr in der Lage sind, die innere Struktur des Agenten zu analysieren, und damit anzugreifen. Der Agent

erscheint nur noch als Blackbox, bei der allenfalls Ein- und Ausgaben zu beobachten sind. Zu den Blackbox-Ansätzen kann die Familie von Mechanismen zur nichtinteraktiven Auswertung von verschlüsselten Funktionen gerechnet werden. Diese Ansätze benutzen eine spezielle Verschlüsselung von Code und Daten, bei denen die Ausführungseigenschaft erhalten bleibt. Der Nachteil dieser Ansätze ist der Umstand, dass derart geschützte Agenten nicht im Klartext mit anderen Parteien kommunizieren dürfen. Um diesen Nachteil aufzuheben, wurde ein neuer Ansatz konzipiert, der zeitbeschränkter Blackbox-Schutz genannt wurde. Dieser Ansatz geht davon aus, dass es möglich ist, den Agenten mittels sogenannter “Verwürfelungsverfahren” in eine Form zu bringen, die den Agenten für eine bestimmbare Zeitspanne vor Analysen schützt. Diese Verwürfelungsverfahren verhindern durch ihre Eigenschaften Angriffsmethoden wie das Lesen und Manipulieren von Agentendaten, sowie die zielgerichtete, temporäre Modifikation des Agentencodes und der Ausführung des Agenten. Aufbauend auf diesen Eigenschaften erlauben es dann Kombinationen bekannter Maßnahmen, weitere Angriffsmethoden, nämlich die Verschleierung der Identität, das Abhören und das Modifizieren der Kommunikation des Agenten mit anderen zu verhindern. Da Angriffe nach diesem Zeitintervall als möglich gelten, muss der Agent sowie einige der mit dem Agenten enthaltenen Daten ein “Verfallsdatum” tragen, nach dessen Erreichen nicht mehr mit diesem Agenten interagiert werden darf. In dieser Arbeit konnte weiter gezeigt werden, dass der letzte verbleibende Angriff, der Blackbox-Test, durch Einsatz eines neuen Protokolls verhinderbar ist. Das Ziel von Blackbox-Tests ist es, die Eingabeparameter zu finden, die zu einer bestimmten Reaktion eines Agenten führen oder aus denen sich bestimmte Eigenschaften des Agenten ableiten lassen. Dazu wird ein Agent mehrere Male mit variierenden Eingabeparametern ausgeführt und es werden die erzielte Effekte beobachtet. Dieser Angriff kann durch Einsatz einer externen, vertrauenswürdigen Komponente verhindert werden, bei der ein Agent den aktuellen Zustand an bestimmten Stellen registriert.

10.2 Anwendbarkeit in anderen Gebieten

Die Lösung des Problems des Schutzes mobiler Agenten vor böswilligen Wirten stellt für die Technologie der mobilen Agenten eine wichtige Voraussetzung für den Einsatz in einigen Anwendungsfeldern dar. Eine solche Lösung könnte jedoch auch für Bereiche außerhalb mobiler Agenten eine große Rolle spielen.

Ein unmittelbarer Anwendungsbereich könnte der Schutz kommerzieller Software vor Reengineering und Cracking (vgl. Kapitel 5) sein. Ein Schutz, der die Kenntnis des ausgeführten Codes verhindert (im Falle von „Code Hiding“) oder erschwert (im Falle von Blackbox-Schutz-Mechanismen allgemein), erlaubt es auch, die in einem Programm angewandten Mechanismen und Algorithmen zumindest zu ver-

schleiern. Ein Schutz, der das Lesen und Manipulieren von Daten, sowie das inkorrekte Ausführen von Code verhindert, verhindert auch, dass ein Programm so modifiziert werden kann, dass z.B. Kopierbeschränkungen ausgebaut oder unwirksam gemacht werden können. Dieser Schutz erstreckt sich allerdings nicht auf die Mechanismen außerhalb des Programms.

Schutzmechanismen, die die ausführende Umgebung als “feindliches Territorium” betrachten und in dieser Umgebung ausgeführte Programme schützen können, sind auch gegen den Fall gewappnet, dass eine eigentlich vertrauenswürdige Umgebung z.B. durch einen Hackerangriff böswillig wird. Daher sind in einem derart geschützten System Programme sicherer als in einem ungeschützten System. Die Sicht von der Ausführungsumgebung als feindlicher Hafen, in dem sicher gearbeitet werden kann, kann auch neue Anwendungen eröffnen. Ein Beispiel wäre das “Mitnehmen” einer Arbeitsumgebung auf einen beliebigen Rechner. In einen System, in dem mobile Agenten geschützt werden können, kann eine Ausführungsumgebung auch nicht in einer einfachen Art und Weise die Arbeitsumgebung des Benutzers angreifen.

Dienstleister, die für Kunden die Ausführung kompletter Anwendungen übernehmen (das sog. Application Hosting), könnten weniger vertrauenswürdig sein bzw. würden weniger über den Kunden in Erfahrung bringen können, wenn die Anwendungen geschützt wären.

Aufgrund der Charakteristik von geschützten mobilen Agenten als softwaretechnisch realisierte “trusted computing base”, wie sie sich heute durch die SmartCard-Technik hardwaretechnisch realisieren lässt, könnten in Zukunft mobile Agenten statt SmartCards eingesetzt werden. Dies hätte u.a. den Vorteil, dass sich der Flaschenhals SmartCard durch die Ressourcen des Ausführungsrechners verhindern ließe.

Da die Schutzmechanismen nicht die Absichten der gegnerischen Partei erkennen können, können für sie auch Umstände Angriffe sein, die eher in die Kategorie “Fehler” fallen (ebenso, wie “byzantinische Fehler” als gewollte Angriffe subsumiert werden). Daher können Schutzmechanismen in einigen Fällen auch auf Fehler reagieren, unter Umständen sogar durch Umgehung des Problems, zumindest aber durch Erkennen eines unbeabsichtigten Ereignisses.

10.3 Ausblick

Das Gebiet des Schutzes mobiler Agenten vor Angriffen durch böswillige Wirte enthält noch viele ungelöste Fragen und Aspekte, deren Beantwortung wichtig für das Gebiet der mobilen Agenten ist.

Erstens ist, mangels echter Anwendungen auf der Basis mobiler Agenten, noch nicht klar, welche konkreten Sicherheitsanforderungen es von Seiten der Anwendungen gibt. Dieser Umstand hängt auch damit zusammen, dass es sicherheitssensitive Anwendungen, z.B. im Bereich des elektronischen Handels sicher nie geben wird, wenn nicht vorher klar ist, dass das Thema Sicherheit lösbar ist. Wichtig wären daher Arbeiten, die, anhand von Entwürfen für solche Anwendungen, die Sicherheitsaspekte aus der Bandbreite der in dieser Arbeit untersuchten Fragestellungen herausarbeiten, die für die jeweilige Anwendung wesentlich sind. Anhand solcher Entwürfe würde sich außerdem entscheiden lassen, welchen Mehraufwand die Sicherheit kosten dürfte, indem der Entwurf mit anderen Alternativen (z.B. einem Client-Server-basierten Ansatz) verglichen wird.

In der Frage, wie ein technischer Komplettschutz eines mobilen Agenten mittels Software erreicht werden kann, sind weitere Untersuchungen notwendig. Diese könnten z.B. die Fragestellung bearbeiten, ob es effizientere Verfahren zur nichtinteraktive Auswertung von verschlüsselten Funktionen gibt und ob diese so erweitert werden können, das auch Klartextausgaben möglich sind. Im Bereich des zeitbeschränkten Blackbox-Schutzes müssten Verwürfelungsverfahren gefunden werden, die den Forderungen nach zeitlich begrenztem Schutz genügen.

Neben den technischen Aspekten besitzt das Problem jedoch auch noch ungeklärte juristische Aspekte, die vor einer Benutzung mobiler Agenten in der Praxis stehen. So ist z.B. nicht klar, für welche Fälle der Betreiber eines Wirts oder der Eigentümer eines Agenten haftet. Weiter ist nicht klar, ob der Agenteneigentümer überhaupt erkennen kann, dass er mobile Agenten benutzt, und welche Konsequenzen sich für ihn aus dieser Benutzung ergeben. Erst wenn Fragen wie diese beantwortet sind, können mobile Agenten die Basistechnologie für weltweite Systeme bilden, die elektronischen Handel, Informationssysteme und andere Bereiche vereinen.

Literaturverzeichnis

[And97] Anderson, Ken: Java Performance Issues. Web page. 1997. http://openmap.bbn.com/~kanderso/performance/java/index.html

[Auc96] Aucsmith, David: Tamper resistant software: An Implementation. In: Ross Anderson (Ed.): Information Hiding - Proceedings of the First International Workshop, LNCS 1174, pp. 317-333, Springer-Verlag, 1996

[AFK89] Abadi, Martin; Feigenbaum, Joan; Kilian, Joe: On hiding information from an oracle. In: Journal of Computer and System Sciences, 39(1):21-50, August 1989

[Baz98] Bazzi, Rida: Code Hiding for Mobile Agents Security. Technical Report TR1112998, Department of Computer Science and Engineering, Arizona State University, November 1998

[BHR98] Baumann, Joachim; Hohl, Fritz; Rothermel, Kurt; Straßer, Markus: Mole - Concepts of a Mobile Agent System. World Wide Web, Vol. 1, Nr. 3, pp. 123-137, 1998

[BKR98] Bredin, Jonathan; Kotz, David; Rus, Daniela: Market-based Resource Control for Mobile Agents. In: Proceedings of Autonomous Agents, ACM, pp. 197-204, 1998

[BMW98] Biehl, Ingrid; Meyer, Bernd; Wetzel, Susanne: Ensuring the Integrity of Agent-Based Computations by Short Proofs. In: Kurt Rothermel; Fritz Hohl (Eds.): Mobile Agents, Proceedings of the Second International Workshop, MA'98. pp. 183-194. Springer-Verlag, Germany, 1998

[CGH95] Chess, David; Grosof, Benjamin; Harrison, Colin; Levine, David; Paris, Colin; Tsudik, Gene: Itinerant agents for mobile computing. IBM Research Report RC 20010, IBM, March 1995. http://www.research.ibm.com/massive/rc20010.ps

[CHK97] Chess, David; Harrison, Colin; Kershenbaum, Aaron: Mobile agents: Are they a good idea?. In: Jan Vitek; Christian Tschudin (Eds.): Mobile Object Systems: Towards the Programmable Internet, pp. 25-45. Lecture Notes in Computer Science No. 1222. Springer-Verlag, 1997

[CT00] Luckhardt, Norbert: Verteilte Angriffe auf dem Vormarsch. c't 2/ 2000, S. 45, 2000 http://www.heise.de/ct/00/02/045/

[FGS96] Farmer, William; Guttmann, Joshua; Swarup, Vipin: Security for Mobile Agents: Authentication and State Appraisal. In: Proceedings

of the European Symposium on Research in Computer Security (ESORICS), LNCS 1146, pp. 118-130, Springer-Verlag, 1996

[Fri98] Fritz, Andreas: Realisierung eines vorgegebenen Mechanismus zur Verhinderung von “Testing”-Angriffen gegen “Blackbox”- geschützte Agenten, Diplomarbeit Nr. 1696, Fakultät Informatik, Universität Stuttgart, 1998

[Fün99] Fünfrocken, Stefan: Protecting Mobile Web-Commerce Agents with Smartcards. In: Proceedings of the First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents (ASA/MA’99), pp. 90-102, IEEE Computer Society, 1999

[Gal98] Galyon, Eric: C++ vs Java Performance. Web page. 1998 http://www.cs.colostate.edu/~cs154/PerfComp/index.html

[GM94] General Magic: Telescript Technology: The Foundation for the Electronic Marketplace. General Magic White Paper, 1994

[GM96] General Magic: The Telescript Reference Manual. General Magic, 1996

[HKB97] Hohl, Fritz; Klar, Peter; Baumann, Joachim: Efficient Code Migration for Modular Mobile Agents. In: Proceedings of the 3rd ECOOP Workshop on Mobile Object Systems: Operating System support for Mobile Object Systems (MOS’97), 1997

[Hoh97] Hohl, Fritz: An approach to solve the problem of malicious hosts. Universität Stuttgart, Fakultät Informatik, Fakultätsbericht Nr. 1997/03. http://www.informatik.uni-stuttgart.de/cgi-bin/ncstrl_rep_view.pl?/inf/ftp/pub/library/ncstrl.ustuttgart_fi/TR-1997-03/TR-1997-03.bib

[Hoh98b] Hohl, Fritz: Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts. In: Giovanni Vigna (Ed.): Mobile Agents and Security, pp. 92-113. Springer-Verlag, 1998

[Hoh98c] Hohl, Fritz: A Model of Attacks of Malicious Hosts Against Mobile Agents. In: Proceedings of the ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations, pp. 105 - 120, INRIA, France, 1998

[Hoh99] Hohl, Fritz: A Protocol to Detect Malicious Hosts Attacks by Using Reference States. Bericht Nr. 09/99, Fakultät Informatik. Universität Stuttgart, 1999. http://www.informatik.uni-stuttgart.de/cgi-bin/ncstrl_rep_view.pl?/inf/ftp/pub/library/ncstrl.ustuttgart_fi/TR-1999-09/TR-1999-09.bib

[Hoh00] Hohl, Fritz: A Framework to Protect Mobile Agents by using Reference States. In: Proceedings of The 20th International Conference on

Distributed Systems (ICDCS 2000), pp. 410-417. IEEE Computer Society, 2000

[HR99] Hohl, Fritz; Rothermel, Kurt: A Protocol Preventing Blackbox Tests of Mobile Agents. In: Tagungsband der ITG/VDE Fachtagung Kommunikation in Verteilten Systemen (KiVS'99). Springer-Verlag, 1999

[IAI99] The IAIK JCE project page. http://jcewww.iaik.tu-graz.ac.at/

[ITU87] ITU-T X.509, The Directory - Authentication Framework, November 1987

[KAG98] Karjoth, G.; Asokan, N. ; Gülcü, C.: Protecting the Computation Results of Free-roaming Agents. In: Kurt Rothermel, Fritz Hohl (Eds.): Mobile Agents, Proceedings of the Second International Workshop, MA'98. pp. 195-207. Springer-Verlag, 1998

[KLO98] Karjoth, Günter; Lange, Danny B.; Oshima, Mitsuru: A Security Model for Aglets. In: Giovanni Vigna (Ed.): Mobile Agents and Security. pp 1-14. Springer-Verlag, 1998

[KT99] Karnik, Neeran; Tripathi, Anand: Security in the Ajanta Mobile Agent System, Technical Report, Department of Computer Science, University of Minnesota, May 1999

[LM99a] Loureiro, Sergio; Molva, Refik: Function Hiding Based on Error Correcting Codes. In: Manuel Blum and C. H. Lee (Eds.): Cryptographic Techniques and E-Commerce. Proceedings of the 1999 International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC '99). City University of Hong Kong Press, 1999

[LM99b] Loureiro, Sergio; Molva, Refik: Privacy for Mobile Code. In: Proceedings of distributed object security workshop, OOPSLA'99, Denver, November 1999, 1999

[LO98] Lange, Danny; Oshima, Mitsuru: Programming and Deploying Java Mobile Agents with Aglets. Addison-Wesley, 1998

[LP99] Lal, Manoj; Pandey, Raju: CPU Resource Control for Mobile Programs. In: Proceedings of the First International Symposium on Agent Systems and Applications / Third International Symposium on Mobile Agents (ASA/MA'99), pp. 74-88, IEEE Computer Society, 1999

[MAL] The Mobile Agent List. Web page. http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/mal/mal.html

[Mea97] Meadows, Catherine: Detecting Attacks on Mobile Agents. Accepted paper to the DARPA Workshop on Foundations for Secure Mobile Code Workshop, 26 - 28 March 1997. http://www.cs.nps.navy.mil/research/languages/statements/meadows.ps

[Mob96] Mobilis: Exploring Telescript - mobilis Reader Interview: General Magic's Jim White. Mobilis March 1996. http://www.volksware.com/mobilis/march.96/interv1.htm

[MRS96] Minsky, Yaron; van Renesse, Robbert; Schneider, Fred; Stoller, Scott: Cryptographic support for fault-tolerant distributed computing. In: Proceedings of the Seventh ACM SIGOPS European Workshop, pp. 109-114, 1996

[NIS94] U.S. National Institute of Standards and Technology (NIST): Digital Signature Standard (DSS). FIPS PUB 186, May 1994

[NL98] Necula, George C.; Lee, Peter: Safe, Untrusted Agents Using Proof-Carrying Code. In: Giovanni Vigna (Ed.): Mobile Agents and Security, pp. 61-91. Springer-Verlag, 1998

[Ord96] Ordille, Joann: When agents roam, who can you trust?. In: Proc. of the First Conference on Emerging Technologies and Applications in Communications, Portland, May 1996.
http://cm.bell-labs.com/cm/cs/doc/96/5-09.ps.gz

[PS97] Peine, Holger; Stolpmann, Torsten: The Architecture of the Ara Platform for Mobile Agents. In: Kurt Rothermel; Radu Popescu-Zeletin (Eds.): Mobile Agents. Proceedings of the First International Workshop on Mobile Agents (MA'97). Lecture Notes in Computer Science No. 1219. Springer-Verlag, 1997

[RJ96] Rasmusson, Lars; Jansson, Sverker: Simulated Social Control for Secure Internet Commerce. In: New Security Paradigms '96. ACM Press, 1996

[Röl99] Rölle, Harald: Authentisierung und Autorisierung für das Java/CORBA-Agentensystem MASA. Diplomarbeit, Institut für Informatik, TU München, 1999.
http://wwwhegering.informatik.tu-muenchen.de/common/Literatur/MNMPub/Diplomarbeiten/roel99/HTML-Version/roel99.html

[Ros99] Rosenthal, Peter: MASP - A Mobile Agent Security Protocol. Diplomarbeit. Fakultät Informatik, Universität Mannheim, 1999

[Rot99] Roth, Volker: Mutual Protection of Cooperating Agents, in: Jan Vitek; Christian Jensen (Eds.): Secure Internet Programming, LNCS 1603, pp. 275-288. Springer-Verlag, 1999

[RS98] Rothermel, Kurt; Straßer, Markus: A Fault-Tolerant Protocol for Providing the Exactly-Once Property of Mobile Agents. In: Proc. 17th IEEE Symposium on Reliable Distributed Systems 1998 (SRDS'98), pp. 100-108. IEEE Computer Society, 1998

[Sch97] Schulman, Andrew: Java On the Fly - Grokking Java's Performance Claims. Web page. http://webreview.com/wr/pub/97/07/25/grok/index.html

[SET97] SET specification version 1.0, May 1997. http://www.setco.org/set.html

[SG90] Stamos, J. W.; Gifford, D. K.: Remote evaluation, ACM Transactions on Programmming Languages and Systems 12, 4, pp. 537 - 565, 1990

[SMA] Security in Mobile Agent Systems. Online web bibliography. http://mole.informatik.uni-stuttgart.de/security.html

[ST97] Sander, Tomas; Tschudin, Christian: Towards Mobile Cryptography. Technical Report 97-049, International Computer Science Institute, Berkeley, 1997. http://www.icsi.berkeley.edu/~sander/publications/tr-97-049.ps

[ST98] Sander, Tomas; Tschudin, Christian: Protecting Mobile Agents Against Malicious Hosts. In: Vigna, Giovanni (Ed.): Mobile Agents and Security, Springer-Verlag, 1998. http://www.icsi. berkeley.edu/~sander/publications/MA-protect.ps

[SS97] Straßer, Markus; Schwehm, Markus: A Performance Model for Mobile Agent Systems. In: H. Arabnia (ed.), Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97).Vol II, pp. 1132-1140. CSREA, 1997

[Trä99] Tränkle, Sven: Abrechnung erbrachter Dienstleistungen in Mobile-Agenten-Systemen Diplomarbeit 1750, Fakultät Informatik, Universität Stuttgart, 1999

[TV96] Tardo, Joseph; Valente, Luis: Mobile Agent Security and Telescript. In: IEEE Proceedings of COMPCON '96, 1996

[Vig98] Vigna, Giovanni: Cryptographic Traces for Mobile Agents. In: Giovanni Vigna (Ed.): Mobile Agents and Security, pp. 137-153. Springer-Verlag, 1998

[Wil99] Wilhelm, Uwe: A Technical Approach to Privacy based on Mobile Agents Protected by Tamper-resistant Hardware. PhD Thesis Nr. 1961. Departement D'Informatique, Ecole Polytechnique Federale de Lausanne, 1999

[WSB99] Wilhelm, Uwe G.; Staamann, Sebastian; Buttyán, Levente: Introducing Trusted Third Parties to the Mobile Agent Paradigm. In: Jan Vitek; Christian Jensen (Eds.): Secure Internet Programming, LNCS 1603, pp. 469-492. Springer-Verlag, 1999

[Yee99] Yee, Bennet S.: A Sanctuary for Mobile Agents. In: Jan Vitek; Christian Jensen (Eds.): Secure Internet Programming, LNCS 1603, pp. 261-274. Springer-Verlag, 1999

[YY97] Young, A.; Yung, M.: Encryption Tools for Mobile Agents: Sliding Encryption. In: E. Biham (Ed.): Fast Software Encryption. Proceedings of the 4th International Workshop, FSE'97. LNCS 1267, Springer-Verlag, 1997

Ich erkläre hiermit, dass ich, abgesehen von den ausdrücklich bezeichneten Hilfsmitteln und den Ratschlägen von jeweils namentlich aufgeführten Personen, die Dissertation selbstständig verfasst habe.

(Fritz Hohl)

Security in Mobile Agent Systems

Extended abstract of the Ph.D. thesis
„Sicherheit in Mobile-Agenten-Systemen“
by Fritz Hohl

1 Introduction

Mobile agents are program instances that are able to autonomously move from one execution environment (or *host*) to another one. As program instances, mobile agents consist of code, constant and variable data. Although mobile agents are autonomous entities from the viewpoint of the programming model, they are passive and can be activated only by the host from a technical implementation viewpoint. Mobile agent systems as an application basis offer a number of advantages like a more simple error model, asynchronous operations, dynamic migration of functionality, replacing of passive by active elements, reduction of communication costs, improvement of processing capacity, and reduction of communication delays.

Mobile agents do not only offer advantages; there are also problem areas that have to be solved before this technology can be widely used. One of these problem areas is the security aspect. Only if it is ensured that none of the parties involved risks to be attacked by another party, agents will be employed, and executed, respectively. Security is also important because one of the major intended application areas is electronic commerce which requires the protection of the involved parties.

2 Security areas

Security in mobile agent system can be divided in a number of areas:

a) Security of the used computer system

Being a process of the used computer system, a host can also be a subject of all possible attacks against processes. These attacks include intrusion into the computer system without using the agent system and denial-of-service attacks against the computer system and/or its applications.

b) Security of the used technologies

Mobile agent systems use normally programming languages and other technologies to offer their services. If these technologies are not secure, also the agent system as a whole has security problems. One example of such a technology is the program-

ming language Java. From time to time, security holes of some Java implementations become public. Such holes might allow an attacker to attack a mobile agent system that uses such an Java implementation.

c) Security between agents and hosts

The area of security between agents and hosts can be subdivided into the protection against attacks by remote attackers and such by local attackers.

c.1) Protection against attacks by remote attackers

As hosts and agents, using their hosts, are able to communicate using a potentially insecure network, hosts and agents might be attacked by remote parties.

c.1.1) Attacks against agents

Possible attacks include in this area such against the communication of an agent with a remote party, attacks against the migration of an agent, and denial-of-service attacks against the agent.

c.1.2) Attacks against hosts

Possible attacks include attacks against the communication of hosts with remote parties and denial-of-service attacks against the host. In consequence, such attacks often also concern the agents executed on or migrating to this host.

c.2) Protection against attacks by local agents and hosts

Attacks might happen even in case that no insecure network connection is used. This is the case when agents on the same host or the host attack.

c.2.1) Security between agents

Possible attacks between two agents on the same host include direct attacks that aim at accessing the resources of the other agent and fraud attacks, i.e. such that e.g. aim at taking the money of another agent without providing the service that was agreed upon.

c.2.2) Security between agent and host

This last area can be again subdivided into the protection of the host from attacks by malicious agents executed on that host and the protection of agents from attacks by the malicious executing host.

Although all these areas contribute to the overall security of mobile agent systems, security in the areas a) to c.2.1) for mobile agent systems does not differ from the corresponding problems in other areas of distributed computing. No new aspects arise from the fact of the mobility of programs and the traditional model of secure

nodes that communicate over an insecure network can be applied. Therefore, they are not part of this work.

The area that contains new security problems is area c.2.2. These problems arise from the fact that not only the programmer of a mobile program and the executing party might be separated (like in every mobile code system), but also the executing party and the owner of a mobile agent. Imagine e.g. that a mobile agent carries electronic money in an e-commerce application. An attacking executing party might try to copy this money from the executed agent (the money belongs to the agent owner). This latter problem, the protection of mobile agents from attacks by the executing party does not only arise from the mobility of the agent, but also from the fact that a mobile agent has an individual state. This is not the case for mobile code in general.

Area c.2.2 can be again separated into the security of hosts from attacks by the executed mobile agent and the security of mobile agents from attacks by the executing host.

3 Security of hosts against malicious agents

The possible attacks of a malicious agent against its host include attacks against the host as a part of the mobile agent system. Examples for such attacks are: unauthorized usage of resources, denial-of-service attacks against hosts and attacks against other agents that are achieved by controlling the host. Apart from attacks against the host, also attacks against the used computer system might occur. These attacks use the mobile agent system only as a comfortable attack means. They include all the attacks of computer system against other computer systems and the attacks of users against other users of the same system, respectively. Approaches that shall prevent such attacks include mechanisms in the areas of isolating single agents, of controlling the access of agents to resources, of authenticating agents, of controlling resources, and of proof-carrying code. Protecting hosts from attacks by the executed agents seems to be similar to the problem of protecting an execution environment from mobile code entities, e.g. applets. But the latter problem includes mainly attacks against the used computer system, but rarely also attacks against the execution environment itself.

4 Security of mobile agents against malicious hosts

For the area of attacks of a malicious host against its executed agents, we first can distinguish between attack aims and attack methods. Attack aims specify the purpose of an attack against a mobile agent (what does the attacker wants to achieve?).

There might be a number of aims an attack wants to achieve. These aims depend on the motivation of the attacker and the characteristics of the attacked agents. In general, attack aims can be divided into three areas: knowledge of agent data, modification of agent data and modification of the behavior of an agent. These areas group single attack aims, but there might be more abstract aims of an attack (e.g. hurt the owner of an agent) that are then realized by single attack aims.

Attack methods specify the way an attack takes to reach the attack aim (how does the attacker proceed?). To reach a major aim, sometimes sub aims need to be achieved. These sub aims might already be attack methods as they denote the way to reach the major attack aim, or they can be achieved by using single attack methods. Attack methods can be subdivided into the categories „read attacks“ and „modification attacks“. Remember that attach methods are not equal to attack aims. An attack method that uses data modification might e.g. want to achieve the aim of learning the value of a variable (e.g. by changing the data in way that the agent gives electronic money to the host).

The category „read attacks“ can be subdivided into the sub categories „data read attacks“, „code read attacks“, „communication read attacks“, and „execution read attacks“. The category „modification attacks“ can be subdivided into the sub categories „permanent modification attacks“, and „temporary modification attacks“. Permanent attacks are valid also on following hosts while temporary attacks are effective only on the executing host. Temporary, all elements can be modified that are not visible on the outside during an execution, i.e. data, code, and execution. Permanently, also the communication of the executed agent with other partners might be the target of an attack. A permanent modification of the execution of an agent is not possible because a host cannot influence other hosts, e.g. the way they execute this agent.

While the categories of the attack methods can be listed, this does not hold true for the single attach methods. A value in an agent can be e.g. found by „scanning“ the data area of an agent. There might be a protection mechanism that prevents this scanning by breaking up variables into parts and distributing them in another way. But an attacker might react on that new protections mechanism by a new attack method that uses weaknesses of the protection mechanism. Again, an extended protection mechanism can be designed, and so forth.

Malicious host attack methods include attacks that use the knowledge of return values and the return of incorrect values at system calls, and the possibility to deduce information about an agent by using blackbox tests. All these attacks occur only if the organizational structure is „open“, i.e. if every party is able to introduce not only agents, but also hosts to the system. Mechanisms that prevent attacks by malicious

agents must not „cost“ too much in terms of efficiency. The reason for this requirement can be found in the fact that it is possible, in some situations, to circumvent the security problems (and, therefore, also the associated costs) caused by the migration of agents by the usage of a client-server based approach.

To illustrate the problem of protecting mobile agents from malicious hosts, and to offer a basis for evaluating possible protection mechanisms, a new model of the host machine and the attacker is proposed. This model allows the attacker to execute his attack by means of a program. This program is not only able to read and modify the code and data of the agent, but also to influence the execution of the agent.

In order to categorize the variety of existing protection approaches, a new classification of approaches is introduced. This classification consists on the top level of two classes. The one class of approaches aims at protecting a mobile agent from all attacks. This class can be again subdivided into organizational and technical approaches. Organizational approaches aim at solving the problem by restricting the agent migration to trustworthy hosts. Technical approaches try to achieve a solution on the technical level, so the individual „trustworthiness“ of a host does not need to be determined. Technical approaches again can be subdivided into hardware approaches that use special, secure modules to execute agents, and software approaches that do not need special hardware. The software approaches are based on the idea of transforming a mobile agent into a „blackbox“. A blackbox is a mobile agent if the host is not able to analyze and, therefore, not to manipulate the inner structure of an agent. The other class of approaches allows to protect a mobile agent only from a part of the possible attacks. From this class of approaches, those mechanisms can be summarized that use, in different ways, a state to recognize modification attacks and attacks that execute code in an incorrect way. Apart from this last group, there are a number of other approaches that aim at e.g. ensuring the integrity of constant data or partial results. One of the simplest, yet effective mechanism is the one that allows an agent owner to digitally sign his/her mobile agent. This signature ensures that all constant parts of the agent, especially the code, cannot be altered throughout the execution of an agent at different hosts.

Starting from a general definition of attacks by hosts against their executed agents, the potential of approaches are examined that use the visible results of the execution of an agent for detecting attacks. To that end, the notion of „reference states“ is introduced. Reference states are defined as the state of an agent that results from an execution of an agent by a host that does not attack the agent. Using these reference states, it is possible to verify states that result from executions on any hosts. In this work, four existing mechanisms based on the reference state approach are analyzed and compared. A new mechanism is proposed that combines elements from three of

the existing mechanisms. It allows to generate reference states by „re-executing“ an agent. Assuming that no two following agents conspire, this mechanism verifies the execution of an agent on a host on the following host, regardless of whether this host is trustworthy or not. To optimize the process, verifications of executions on trustworthy hosts can be omitted. The overhead of the proposed mechanism is estimated to be roughly a factor two. When evaluating the overhead, this estimation can be verified. The mechanism was implemented for the mobile agent system Mole using a new framework. This framework supports a broad variety of mechanisms using reference states. Finally an extension of the mechanism is sketched. This extension allows to delete the restriction that no two following hosts must conspire.

Against a mobile agent, many attacks are possible by a malicious host. To allow an as complete protection as possible it is shown that the overall set of attacks can be reduced to a smaller set. If this smaller set of attacks can be prevented, most attacks can be prevented, too. Agents that allow to prevent this smaller set of attacks are called „Blackbox agents“. This term denotes the fact that attackers cannot analyze the inner structure of agents protected in this way. As a consequence, these agents cannot be attacked anymore. The agent appears to be a blackbox where only input to and output from the blackbox can be observed. One of the two blackbox approach classes is called „non-interactive evaluation of encrypted functions“. This class uses a special „encryption“ of code and data that still ensures that the encrypted code can be executed. The disadvantage of this class of approaches consists in the fact that protected agents must not communicate with other parties in cleartext. To circumvent this disadvantage, a new class of approaches is presented that is called „time-limited blackbox protection“. This approach assumes that it is possible to convert the agent using certain „obfuscating algorithms“ into a blackbox which cannot be analyzed for a certain amount of time. These obfuscating algorithms prevent by construction attack methods like reading and modifying agent data, and the directed, temporary modification of agent code and its execution. Based on this characteristics, combinations and applications of known mechanisms allow to prevent further attack methods like masking, and reading and modifying of the communication of an agent with other parties.

Masking can be prevented by a special two-way authentication mechanism that takes place after the arrival of a mobile agent at a host. In a first phase the host authenticates the agent by verifying the agent signature. This ensures that the agent was not modified by another party. The verification uses the public key of the agent owner. In the second phase the host encrypts the signature of the mobile agent (or at least a hash of the signature) and sends the result to the mobile agent locally, together with the key certificate of the public key of the host. These key certificates are signed by a trust-worthy party, the so-called key authority. The agent either

knows the public key of the key authority or is able to learn this key in a secure way. Therefore, the agent is able to verify the key certificate, and, therefore, also the public key of a host. The agent then decrypts the result sent by the host and compares the resulting value with the signature of the agent (or the hash). If both values are equal, the agent can be sure that it communicates with the host identified by the key certificate. The message to the agent from the host cannot be replayed by an attacker as first, the communication of the host with the agent is local (so no network attackers can see the message), and second, only an unmodified agent could have seen the message (as this was checked by verifying the agent signature). As this agent is not interested in attacking itself, no foreign party can have learned the message.

There are several consequences from the now introduced time-limitation. First, of course, the agent cannot use any amount of time to fulfil its task anymore (the same holds for the host). Second, after the expiration date, the agent must neither be accepted anymore for arrival at a host nor for interaction by remote communication. To achieve that, the expiration date needs to be known to hosts and potential interaction partners. To that end, the expiration date needs to be part of the signed data of the agent itself and of the key certificate of that agent. As the potential interaction partners use this key certificate, they know also the expiration date. The third consequence from the time-limitation is the need of some data carried by the mobile agent to include the same expiration date. These data have a value of their own and can be used independently of an agent. This sort of data is therefore called „Tokens“. Examples for tokens are electronic coins, secret keys, tickets and capabilities. As after the expiration date, tokens can be robbed out of the agent, it has to be ensured that these token are given to a third party by a non-attacked agent. This can be ensured by checking the expiration date of tokens and by not accepting the tokens afterwards. As tokens often also carry a digital signature, the expiration date can often be integrated in the structure of the token. Unfortunately this expiration date also changes the semantics of the token. Electronic coins then cannot be valid forever, a secret key does not ensure the privacy of data encrypted with that key forever, but only for a limited amount of time. It is not possible for all tokens to change their semantics in that way (e.g. the secret key of the agent owner needs to be secure for a longer time than the lifetime of an agent). These tokens then must not be carried by a mobile agent protected by a time-limited blackbox protection mechanism.

It is shown that also another remaining attack, the so-called blackbox test, can be prevented by using a new protocol. The aim of blackbox-tests is to find the set of input parameters that lead to a certain reaction of an agent or out of which certain information about the agent can be derived. To that end, an agent is executed several times with varying parameters. Then, the resulting effects are observed. In contrast

to that behavior, a honest host would execute the agent exactly once. The idea of preventing blackbox tests consists in preventing the execution of an agent several times with different parameters. Unfortunately, this cannot be achieved by mechanisms inside the agent because all data that store e.g. the number of executions, are lost as soon as a new copy of this agent is used by the host. Therefore, a third party outside the host needs to be involved. This party operates a registry component on a trusted machine. Using this registry, the agent can register every execution and ask for permission to continue. If the registry sees multiple executions of the same agent, it reject the request by the agent and the agent stops. To do that in a secure way, replay attacks must not be possible. At a replay attack, the host would record the messages from the permission message from the registry to the agent and would replay this message for every new execution of that agent. Normally, replay attacks are prevented by including some secret data selected by the agent and not predictable by the attacker to the request message from the agent to the registry. By repeating this data in the permission and by signing the message, the agent could be sure that only the registry could have answered to exactly this request message. Unfortunately, currently no way is known to produce such data at the agent side that is both secret and non-predictable, even if the agent is blackbox protected. While the agent could carry secret data, the selection process would be determined and therefore predictable by the host. While the agent could ask the host for random data or the system time, these data would be known to the attacking host.

The problem of preventing replay attacks can be solved when the original problem is slightly redefined. We do not need to prevent multiple executions of the same agent in every case, but only such that use different input parameters. If we follow that idea, we can use a secure hash of the input parameters as the data that prevent replay attacks although these data are neither secret nor non-predictable. The agent requests the registration using the hash of the input parameters. In case the registry detects another execution of the same agent with a different set of input parameters, the request is denied, else accepted. To prove that the registry has seen this hash it is sent back in a message signed by the registry. Now the agent can check whether the permission message is ok by verifying the signature of the message and comparing the contained value with the hash sent to the registry. This mechanism still allows to replay permission messages, but only those the registry would have been sent anyway. If there are several executions of the same agent using the same set of input parameters, the attacker does not gain more knowledge about the agent, therefore, the attack is useless. In the protocol that implements that idea, neither the request message nor the rejection message needs to be protected by digital signatures, only the permission message needs to be signed in the presented way.

5 Possible application in other areas

Solving the problem of protecting mobile agents from malicious hosts is an important prerequisite for the application of the mobile agent technology in some areas. Apart from that aspect such a solution might play an important role also in areas outside the mobile agent area.

A direct application area might be the protection of commercial software from reengineering and cracking (see Section 6). A protection mechanism that prevents the knowledge of the semantic of the executed code forever (in the case of „code hiding“) or at least for a certain period of time (in the case of general blackbox protection mechanisms) allows to at least cover up the mechanisms and algorithms used in a program. A protection mechanism that prevents reading and manipulating of data and the incorrect execution of code allows to prevent the modification of a program with the purpose of circumventing or deleting of e.g. copy protection mechanisms. Though these mechanisms do not protect mechanism outside a program.

Protection mechanisms that consider the executing environment as „hostile territory“ and that are able to protect the executed programs in such environments protect also in the case that a trustworthy environment is getting malicious by e.g. an hacker attack. Therefore, programs in such a protected system are more secure than in an unprotected system. The aspect of the execution environment as a hostile harbor in which still a secure execution is possible, also allows new applications. An example for such an application is carrying a working environment on any computer. In an environment where it is possible to protect mobile agents, also working environments cannot be attacked easily.

Application hosting provider that execute complete applications for their customers do not need to be trustworthy or could learn less about these customers, if the applications can be protected in the described way.

As protected mobile agents are software-based „trusted computing bases“, as they are realized currently in hardware by „smart cards“, in the future mobile agents could be used instead of such smart cards. This approach would allow, among other advantages, to circumvent the bottleneck smart card by the resources of the executing computer system.

As the protection mechanisms are not able to recognize the intentions of the potentially attacking party, they might treat also some errors as attacks (in the same way as attacks can be subsumed under „byzantine failures“). Therefore, protection mechanisms might react in some cases also to errors and failures. This reaction can consist in a recognition of an unwanted event, or even in a treatment of the problem.

6 Outlook

The area of the protection of mobile agents from attacks by malicious hosts still contains many unsolved problems and aspects whose solution would be important for the area of mobile agents.

First, due to the lack of real applications based on mobile agents, it is still not yet clear which concrete security requirements can be derived from the applications. This probably also relates to the problem that there will be no security sensitive applications, e.g. in the area of electronic commerce, if it is not known whether the problem of security in open mobile agent systems can be solved. Therefore, it would be important to examine the design of such applications and to derive those security aspects out of the range presented in this work that are relevant for these applications. Additionally, such designs would allow to estimate the maximum possible costs for security mechanisms by comparing these designs with other alternative designs (e.g. client-server-based approaches).

Furthermore, in the area of how a technical, software-based, complete protection of mobile agents can be achieved, more research is needed. This research could examine e.g. the question of whether there are more efficient mechanism for the non-interactive evaluation of encrypted functions, and whether they can be extended in a way that allows plain text interaction. In the area of time-limited blackbox mechanisms obfuscation mechanisms have to be found that satisfy the requirements of protection for at least a limited period of time.

Apart from the technical aspects the security in mobile agent systems has also unsolved legal aspects that have to be solved before mobile agent can be used in reality. One of these aspects is in which cases the operator of a host or the owner of an agent is liable if damages occur. Other aspects are whether a user is able to even know that it is using mobile agents and which implication this usage might have. Mobile agents can be used as the basic technology for worldwide systems that unify electronic commerce, informations systems and other areas only if questions like the ones mentioned above are solved.