FACHHOCHSCHULE KÖLN
UNIVERSITY OF APPLIED SCIENES COLOGNE
ABTEILUNG GUMMERSBACH

FACHBEREICH INFORMATIK

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom - Informatiker (FH)

Ein Klassenlader unter Berücksichtigung von Sicherheits- und Kryptographieaspekten unter Java

Irmgard Barbara Büscher
St- Engelbert-Str. 22
51519 Odenthal

1. Prüfer: Prof. Dr. rer. nat. Heide Faeskorn - Woyke
2. Prüfer: Prof. Dr. rer. nat. Holger Günther

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Thematische Grundlage der vorliegenden Diplomarbeit ist eine von der Firma S|4|M Solution for Media gegebene Aufgabenstellung.
Für das Modul JCool der Firma S|4|M Solution for Media sollte ein Klassenlader geschrieben werden, der geänderte oder neue Klassen des Programmes JCool über eine Socketverbindung von einem Server liest und auf dem Client-System speichert. Für die Beantwortung der Client-Anfragen sollte der bestehende Server für das JCool-Login erweitert werden. Der ganze Prozess des Updatens der Klassendateien sollte dabei den Lauf des Programmes JCool nicht beeinflussen und deshalb im Hintergrund stattfinden. Auch war die Integrität der Daten während der Übertragung und der autorisierte Zugriff auf den Server ein Schwerpunkt der Aufgabenstellung.

Zur Zeit muss jede neue Version des Modul JCool bei den Kunden (den Werbeagenturen) entweder vom Kunden oder vom Auftraggeber des JCoolClass-Loaders neu installiert werden. Da es aber für den Auftraggeber ein sehr großer Aufwand ist, über ganz Deutschland hinweg bei den Kunden das Modul JCool zu aktualisieren, oder nicht sichergestellt ist, dass die Aktualisierung vom Kunden durchgeführt wurde, entschied sich der Auftraggeber für eine automatische Aktualisierung des Moduls JCool.

Ziel dieser Diplomarbeit ist die Entwicklung eines benutzerdefinierten netzwerkorientierten Klassenladers für das Modul JCool unter der besonderen Berücksichtigung der Sicherheit. Dazu werden zum einen die Sicherheitsmerkmale von Java und die Grundlagen der Kryptographie mit der Beschreibung einzelner Verschlüsselungstechniken erläutert, und zum anderen werden die Techniken zur Realisierung eines Klassenladers, der die Klassen über eine Socketverbindung lädt, besprochen.

Im Teil "Das Modul JCool“ wird ein Überblick über das zugrunde liegende Programm gegeben und die Struktur des Programmes vorgestellt.
Im darauf folgendem Kapitel "Java Architektur und Java Sicherheit “ werden die Architektur, die Arbeitweise des ClassLoaders in der Java Virtual Machine sowie die Sicherheitsaspekte von Java beschrieben.
Darauf folgt mit dem Kapitel "Kryptographie" ein Hauptschwerpunkt der Arbeit, da die Unversehrtheit der geladenen Klassen grundsätzlich sichergestellt sein muss. In dem Kapitel werden die Grundlagen der Kryptographie unter Berücksichtigung einzelner Verschlüsselungstechniken erläutert.

Das anschließende Kapitel "Sockets in Java“ behandelt die Möglichkeit, in Java Socketverbindungen herzustellen, um Client-Server Anwendungen zu implementieren.
Danach werden im Kapitel "Programmrelevante Klassen der Java API" die wichtigsten Klassen der Java API besprochen, die für die Umsetzung des JCool-ClassLoaders herangezogen wurden.
Im abschließenden Kapitel "Der JCoolClassLoader" wird der Programmablauf der clientseitigen sowie serverseitigen Anwendung unter Erwähnung der besonderen Sicherheitsmaßnahmen vorgestellt.

2. Das Modul JCool

Das Modul JCool, das von der Softwarefirma S|4|M Solution for Media entwickelt wurde, ermöglicht Werbeagenturen Online Buchungen von Werbezeiten vorzunehmen.

2.1. Überblick über das Modul JCool

Werbeagenturen wickeln ihre Buchungen für die Kunden normalerweise über die Dispositionsabteilung der Sender oder über einen Vermarkter (wie z.B. IP Deutschland) ab. JCool wird als so genanntes "Goodie" den Werbeagenturen zur Verfügung gestellt und ermöglicht diesen, einen direkten Zugriff auf die verfügbaren Werbeinseln, inklusive Buchen und Stornieren zu haben. Buchungen und andere datenverändernde Zugriffe werden nicht direkt durchgeführt, sondern asynchron durch einen Buchungsprozess auf dem Server bearbeitet. Dazu werden die Daten in einer Request-Tabelle auf der Sybase-Datenbank zwischengespeichert, damit eine maximale Sicherheit gewährleistet werden kann. Die Weiterverarbeitung der Transaktionen geschieht anschließend mit dem Buchungsserver des Produktes S|4|AdSales der Firma S|4|M.
Diesen Zusammenhang verdeutlicht folgende Grafik[1]:

[1] http://www.s4m.de/deutsch/portfolio/JCool.html (16.09.2002)

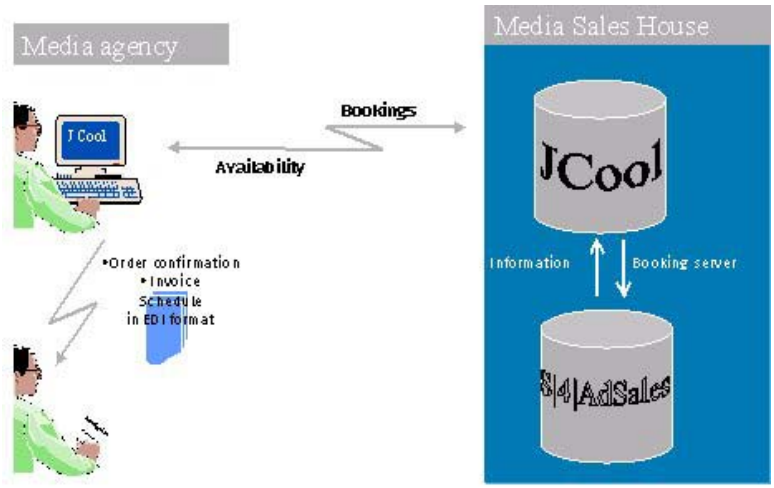


Abbildung 2.1 : JCool Online-Kommunikation

JCool ist eine Java Applikation, die die heutigen Standards wie Kontextmenüs, Drag&Drop, Farbgebung und Benutzerführung unterstützt wie folgende Abbildung[2] zeigt:

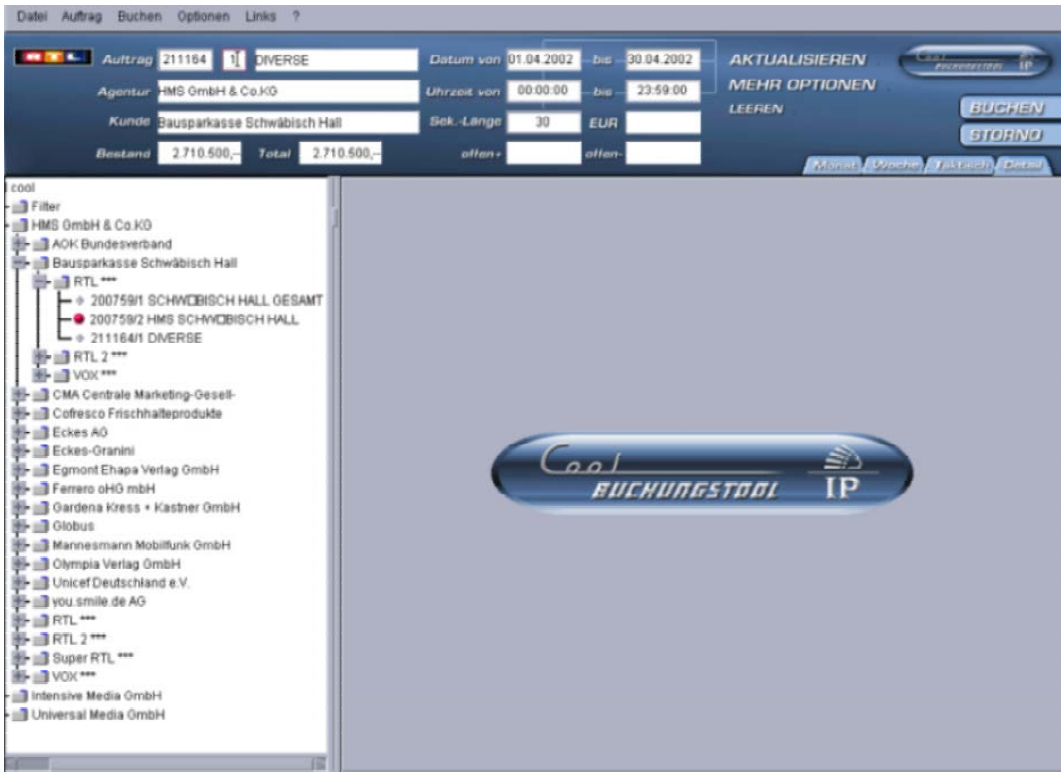


Abbildung 2.2 : Grafische Oberfläche des Moduls JCool

Zur grafischen Gestaltung werden sowohl Klassen der Java AWT und Java Swing, als auch Klassen eines kommerziellen Anbieters genutzt, die in dem Package "Argent" liegen. Für den sicheren Serverzugriff während des Logins

[2] Screenshot der Programmoberfläche von JCool

wird die RSA-Verschlüsselung[3] verwendet. Die Kryptoklassen stammen wie die grafischen Klassen aus einem kommerziellen Package "is". Die beiden genannten kommerziellen Packages und andere, die im nächsten Kapitel erwähnt werden, beeinflussen den Aufbau des JCoolClassLoaders, da die Klassen dieser Packages grundsätzlich von dem Client-System geladen werden sollen.

2.2. Packagestruktur des Moduls JCool

Ein wesentlicher Punkt, der den Aufbau des Klassenladers beeinflusst, ist die Vorgabe, dass nur Klassen des eigentlichen JCool-Programmes bei Bedarf dynamisch während der Laufzeit von dem Server geladen werden sollen. Alle anderen Klassen aus kommerziellen Packages oder aus Packages, die keiner Aktualisierung bedürfen, sollen dynamisch vom Client-System geladen werden. Der Kern des Moduls JCool besteht aus ca. 240 Klassen, die in der folgenden grün gefärbten Package-Struktur integriert sind.

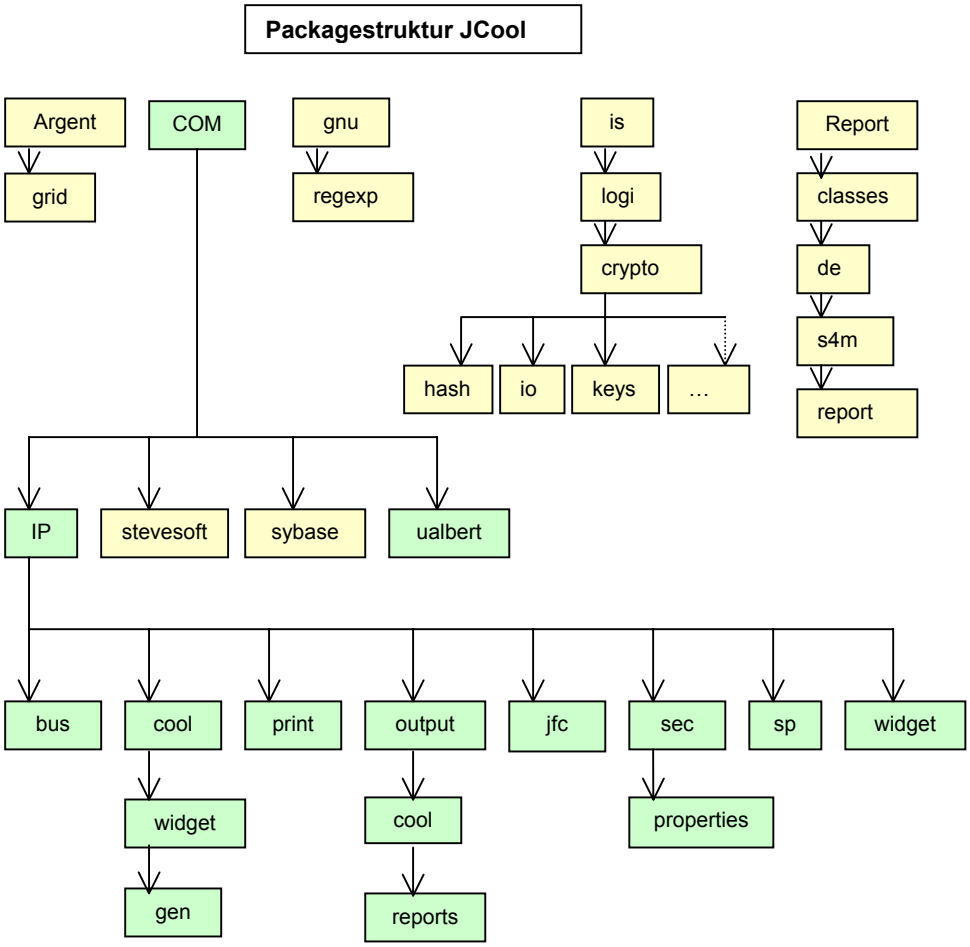


Abbildung 2.3 : Packagestruktur des Moduls JCool

[3] Der RSA-Algorithmus wird ausführlich im Kapitel 4.3.4 besprochen.

Die gelb gefärbte Package-Struktur stellt die kommerziellen Packages dar, bzw. diejenigen, die keiner Aktualisierung bedürfen.
So verbirgt sich hinter der gelben Packages-Struktur, die mit dem Package-Namen "is" beginnt das Kryptopackage. Die Packages "stevesoft" und "Report" enthalten Klassen, die für das Drucken benötigt werden und im Programmablauf erst recht spät geladen werden.
Da das Modul JCool während der Laufzeit sowohl auf Klassen aus den grün und gelb gefärbten Packages zugreift, muss der JCoolClassLoader mit Hilfe einer Abfrage entscheiden, ob die Klassen zu den Packages gehören, die immer vom Client-System geladen werden, oder zu den Packages, die aktualisiert werden können. Diese Unterscheidung ist wichtig, weil eine einmal auf bestimmten Weg durch den Klassenlader geladene Klasse auch alle anderen Klassen auf dem gleichen Weg lädt, auf die die vorher geladene Klasse verweist. Der Klassenlader unterscheidet zwar prinzipiell, ob es sich um Systemklassen oder um selbstdefinierte Klassen handelt, aber er kann nicht entscheiden, ob die selbstdefinierten Klassen vom Client-System oder vom Server geladen werden sollen. Das Problem wurde gelöst, indem eine Abfrage sicherstellt, dass nur Klassen aus dem Package COM.ip oder COM.ualbert aus dem Fileordner oder über die Socketverbindung geladen werden. Im Kapitel über den JCoolClassLoader wird die Realisierung dieser Problematik im Zusammenhang mit dem Gesamtkontext dargestellt.

3. Java Architektur und Java Sicherheit

Java wurde am 23. Mai 1995 als neue, objekt-orientierte und plattformunabhängige Programmiersprache von der Firma Sun eingeführt.
Zurückzuführen ist Java auf die Sprache Oak, die 1991 von Bill Joy, James Gosling und Mike Sheridan im so genannten Green-Projekt entwickelt wurde. Das Projekt hatte das Ziel, eine einfache und plattformunabhängige Programmiersprache zu kreieren, die auf allen Betriebssystemen und Plattformen lauffähig ist. Seit der Einführung von Java kamen verschiedene neue Versionen auf dem Markt. Java wurde mit der Einführung der Version 1.2 in Java 2 umgenannt. Die Realisierung des JCoolClassLoaders wurde mit der Version 1.3.1 umgesetzt. Die folgenden Unterkapitel beschäftigen sich mit der Java Architektur und mit der Java Sicherheit.

3.1. Die Java Architektur

Die Java Architektur besteht im Grunde aus vier in Beziehung stehenden Teilen:

- der Java Programmiersprache
- dem Class-File-Format (.class)
- der Java API[4]
- der Java Virtual Machine (JVM)

Während des Schreibens und Ablaufens eines Java-Programmes werden alle vier Teile benutzt. Der Source-File des Programmes wird in der Java Programmiersprache geschrieben und anschließend in das Class-File-Format compiliert. Während des Programmablaufs werden die im Source-File genutzten System-Ressourcen aus der Java API von der JVM aufgerufen.
Die Java API und die JVM bilden zusammen die Java-Plattform oder auch das Java Runtime System genannt. Java Programme sind auf verschiedenen Computersystemen aufgrund der Java-Plattform ablauffähig.
Die folgende Grafik[5] verdeutlich insbesondere den Unterschied zwischen Übersetzungs-Umgebung und Laufzeit-Umgebung:

[4] application programming interface
[5] Bill Venners: Inside the Java 2 Virtual Machine, USA: McGraw-Hill Companies 1999, S. 5

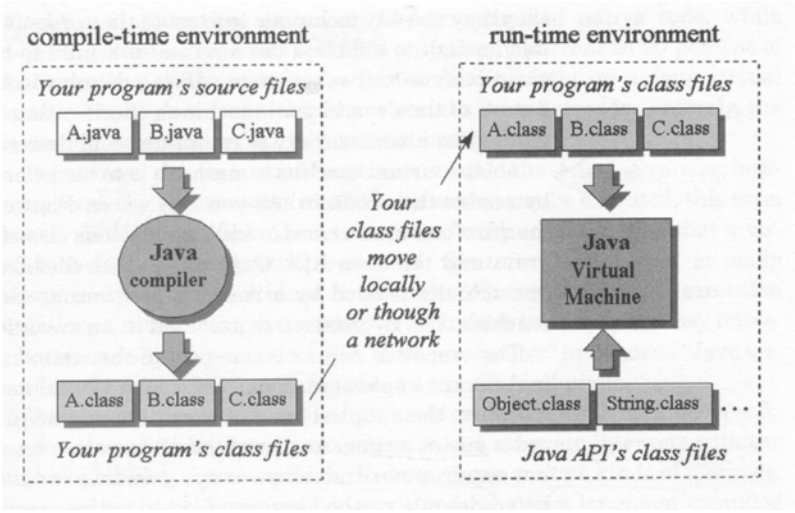


Abbildung 3.1 : Java Programm Umgebung

Die JVM ist der Grundstein für die Netzwerkorientierung von Java, welche die drei wesentlichen Punkte von Java unterstützt:

- Plattformunabhängigkeit
- Sicherheit
- Netzwerk Mobilität

Die JVM ist ein abstrakter Computer, dessen Hauptaufgabe es ist, Klassen zu laden, und deren Bytecode auszuführen. Die JVM enthält dafür einen Class-Loader, der Klassen, wie in der folgender Grafik[6] zu sehen ist, vom Java Programm und von der Java API lädt.

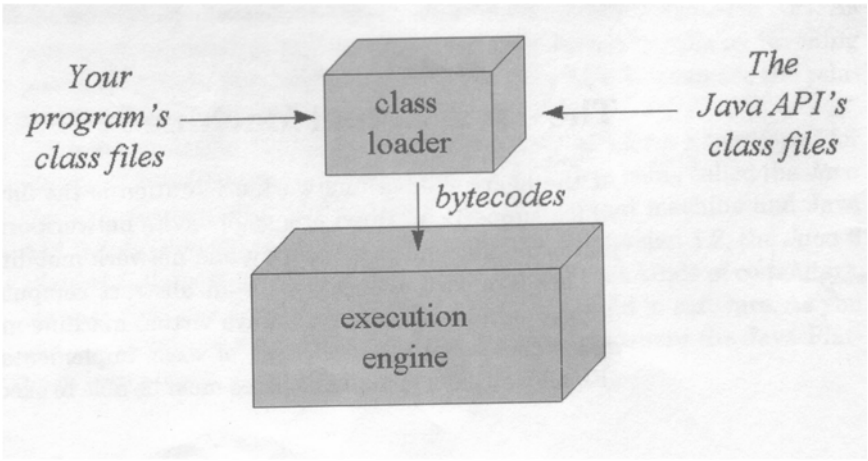


Abbildung 3.2 : Basis der JVM

[6] Bill Venners: Inside the Java2 Virtual Machine, USA: McGraw-Hill Companies 1999, S. 6

3.1.1. Die ClassLoader Architektur

Sowohl für die Sicherheit, als auch für die Netzwerkmobilität der JVM spielt die Architektur des ClassLoaders eine wichtige Rolle.
In der Realität können mehr als ein ClassLoader in der JVM enthalten sein. So kann der ClassLoader Block in der Abbildung 3.2 mehrere ClassLoader enthalten. Die JVM besitzt eine flexible ClassLoader Architektur, die es Java Anwendungen ermöglicht, Klassen auf selbstdefiniertem Wege zu laden. Dadurch kann eine Java Applikation zwei Arten von ClassLoader nutzen:

- den Bootstrap-ClassLoader[7]
- und den User-Defined-ClassLoader (wie der JCoolClassLoader)

Der Bootstrap-ClassLoader ist Teil der JVM Implementation und insgesamt einmal vorhanden. Der Bootstrap-ClassLoader lädt Klassen, die in der Java API enthalten und die in der Umgebungsvariable CLASSPATH angegeben sind, von der lokalen Festplatte. Da der Bootstrap-ClassLoader keine Klassen über das Netzwerk laden kann, wurde in Java die Möglichkeit gegeben, dass eine Java Anwendung einen User-Defined-ClassLoader einsetzen kann, der die Klassen auf Benutzer definierte Art lädt, z.B. über ein Netzwerk oder über das Internet. Java bietet keine Möglichkeit direkt auf den Bootstrap-ClassLoader zuzugreifen, dazu muss die abstrakte Klasse *'ClassLoader'*[8] der Java API überschrieben werden. Ein Bootstrap-ClassLoader ist also ein wesentlicher Teil der JVM, User-Defined-ClassLoader sind aber nicht Teil der JVM.
User-Defined-ClassLoader sind in Java geschrieben und in das Class-File-Format compiliert. Sie werden in die JVM geladen und wie jedes andere Objekt instanziert. Mit Hilfe des User-Defined-ClassLoaders ist es also möglich, Java Anwendungen dynamisch während der Laufzeit zu erweitern, und Class-Files z.B. von einem Netzwerk oder von einer Datenbank herunterzuladen. Für jede Klasse, die geladen wird, behält die JVM die Spur, von welchem ClassLoader - User-Defined-ClassLoader oder Bootstrap-ClassLoader - die Klasse geladen wurde. Wenn eine geladene Klasse auf eine andere Klasse verweist, verlangt die JVM, dass die referenzierte Klasse von demselben ClassLoader geladen wird, wie die ursprüngliche Klasse.
Im vorigen Kapitel wurde dieser Sachverhalt schon kurz angesprochen.
Da die JVM dieses Verfahren zum Laden der Klassen nutzt, können Klassen nur andere Klassen sehen, die mit demselben ClassLoader geladen wurden.

---

[7] Bootstrap (englisch) = Urlader (deutsch)

Auf diese Weise ermöglicht es die Java Architektur, innerhalb einer Java Anwendung vielfache Namensräume zu kreieren. Jeder ClassLoader in einer laufenden Anwendung hat seinen eigenen Namensraum, der mit den Klassennamen belegt ist, die mit dem jeweiligen ClassLoader geladen wurden. Eine Java Anwendung kann viele User-Defined-ClassLoader, entweder von der gleichen Klasse oder von verschiedenen Klassen, instanziieren. Die Anwendung kann außerdem so viele User-Defined-ClassLoader kreieren wie nötig. Klassen, die von verschiedenen ClassLoader geladen wurden, liegen in verschieden Namensräumen und können keinen Zugriff auf die anderen Klassen erhalten, es sei denn, dass dieser Zugriff explizit erlaubt wird. Auf diese Weise kann die ClassLoader-Architektur von Java dazu genutzt werden, jede Interaktion zwischen Klassen, die von verschiedenen Quellen geladen wurden, zu kontrollieren. Man kann also vorbeugend verhindern, dass feindlicher Code den Zugriff auf freundlichen Code erhält. Eine der bekanntesten dynamischen Ausführer von Code ist der Web-Browser, der User-Defined-ClassLoader nutzt, um die Klassen für ein Applet über das Netzwerk zu laden.
Auch das dieser Arbeit zugrundeliegende Programm, der JCoolClassLoader, nutzt die durch die flexible ClassLoader Architektur gegebene Möglichkeit, gezielt Klassen über ein Netzwerk zu laden.

3.2. Die Java Sicherheitsarchitektur

Die Java Sicherheitsarchitektur wird als ein Schlüssel für die Netzwerkorientierung von Java angesehen. Die Sicherheit ist besonders wichtig, weil die Gefahr von Attacken für Computer, die an einen Netzwerk angeschlossen sind, sehr hoch ist. Eine besondere Gewichtung erhält die Sicherheit, wenn Software, gleich ob Applets oder eine Java-Anwendung, über das Netz geladen und ausgeführt wird. Denn ohne diese Sicherheit wäre der Weg für böswilligen Code geöffnet.
Die Java Sicherheitsarchitektur ist darauf fokussiert, Endnutzer vor böswilligen Code oder wohlwollenden Code mit Fehlern zu schützen, die z.B. über das Netzwerk geladen werden.
Ein Teil der Sicherheit von Java wird bereits durch die Datenkapselung ermöglicht, die von der Objektorientiertheit von Java unterstützt wird. Während des Compilierens tritt ein weiterer Sicherheitsaspekt zutage. Der Sourcecode wird in

[8] In Kapitel 6.1 'ClassLoader' wird gezeigt, wie ein User-Defined-ClassLoader implementiert werden kann.

das Class-File-Format mit der Extension .class compiliert, die den Bytecode enthält, und gleichzeitig auf syntaktische Korrektheit überprüft.
Die .class Datei enthält über dem compilierten Code hinaus noch Informationen über das Source-File, die der Bytecode-Verifier auswertet.
Die wesentlichen Komponenten der Sicherheitsarchitektur, die den jeweiligen Sicherheitsmodellen der Java Versionen zugrunde liegen und die Teil der JVM sind, sind der Bytecode-Verifier, der ClassLoader und der Security Manager. Die folgende Abbildung[9] verdeutlicht den Zusammenhang der hauptsächlichen Komponenten der Sicherheitsarchitektur, die in den anschließenden Unterkapiteln erläutert werden.

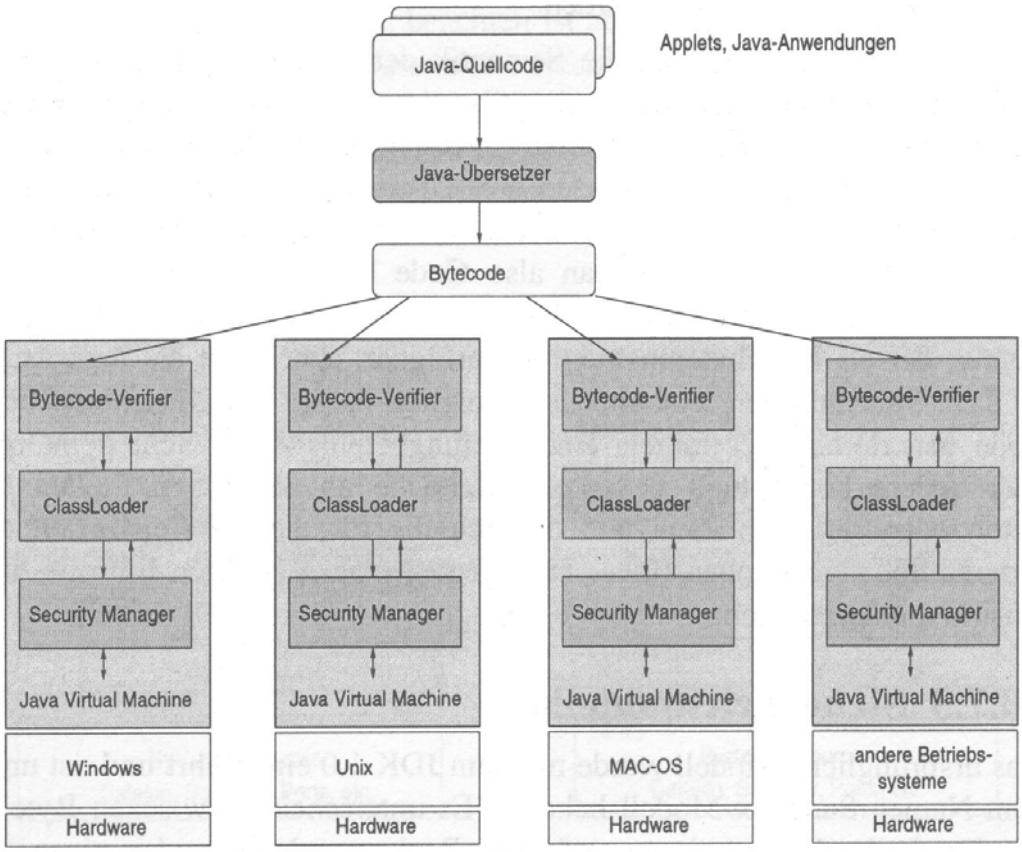


Abbildung 3.3 : Die Java Sicherheitsarchitektur

[9] C. Eckert: IT-Sicherheit, München: Oldenburger Wissenschaftsverlag GmbH, S. 429

3.2.1. Der Bytecode-Verifier

Nach den bereits durchgeführten Analysen während des Compilierens, ist es die Aufgabe des Bytecode-Verifiers sicherzustellen, dass nur korrekter Bytecode auf der JVM ausgeführt wird. Der Bytecode-Verifier wird deshalb auch als das Tor zur JVM bezeichnet.
Die Bytecodeüberprüfung erfolgt in vier Schritten, von denen im ersten die syntaktische Korrektheit nach dem Sprachstandard für Java-Klassen sichergestellt wird. Die Struktur der Java Klassen beinhaltet u.a. Informationen zur Version der Klasse, zu übergeordneten Klassen, zu Methoden und zu Attributen.
Die Integrität der Klasse wird im zweiten Schritt geprüft. Der Bytecode-Verifier untersucht dabei, ob die überprüfte Klasse von einer gültigen Superklasse abgeleitet wurde. Im dritten Schritt erfolgt der Aufruf des komplexesten Bestandteils des Bytecode-Verifiers. Dabei wird der Bytecode vor dessen Ausführung nochmals auf unzulässige Instruktionen überprüft. Unter anderem, ob bei der Ausführung des Codes undefinierte Zustände auftreten können.
Eine Datenflussanalyse überprüft in Schritt 4 nacheinander für jede Instruktion im Bytecode, ob z. B. auf dem Stack der JVM genügend Elemente vom richtigen Datentyp zur Verfügung stehen und ob bei Zugriffen auf lokale Variablen ebenfalls keine falsche Typen verwendet werden.
Besonders wichtig bei der Datenflussanalyse ist das Aufspüren möglicher Überläufe des Stapelspeichers. Denn andere Laufzeitumgebungen in anderen Programmiersprachen überprüfen nicht, ob es zu Überläufen des Stapelspeichers während der Ausführung kommen kann, wodurch immer ein Unsicherheitsfaktor gegeben ist.

3.2.2. Der ClassLoader

In seiner Standardeinstellung unterscheidet der ClassLoader nur zwischen vertrauenswürdigen und nicht vertrauenswürdigen Klassen. Je nach der Version der verwendeten JVM hängt es ab, welche Klassen in welche Kategorie fallen.
Im Grunde trägt der ClassLoader auf drei Wegen zur Sicherheit bei:

1. Beugt bösartigen Code vor, sich in wohlwollenden Code einzumischen
2. Schützt die Grenze zu den vertrauenswürdigen Class-Libraries
3. Plaziert den Code in Kategorien (protection domains), die festlegen, welche Aktionen der Code ausführen kann.

Diese Punkte leistet der ClassLoader unter anderem, indem er Namensräume[10] für verschiedene ClassLoader zur Verfügung stellt. Zum Beispiel erhält jedes nichtvertrauenswürdige Applet oder jede nichtvertrauenswürdige Java-Anwendung einen eigenen ClassLoader und einen eigenen Namensraum und kann nur auf die eigenen Methoden und die zur Verfügung gestellten Systemmethoden zugreifen. Namensräume tragen also zur Sicherheit bei, weil sie Klassen, die mit verschiedenen ClassLoadern geladen worden sind, vollständig voneinander abschotten.

Ein sehr wichtiger Punkt, warum der ClassLoader zur Sicherheit beiträgt, ist, dass beim Laden in die Laufzeitumgebung immer erst die lokal vorliegenden Klassen geladen werden. Danach erst wird auf den Code aus dem Netzwerk zugegriffen. Durch dieses Vorgehen wird sichergestellt, dass die Basisklassen der Laufzeitumgebung nicht durch gleichnamige Klassen z.B. eines Applets überschrieben werden können. Es werden also die Grenzen der vertrauenswürdigen Libraries geschützt.

Für das Laden der mit dem JDK installierten Basisklassen sorgt der Bootstrap-ClassLoader, der auch als Primordial-ClassLoader oder System-ClassLoader bezeichnet wird. Die Basisklassen werden von dem Bytecode-Verifier nicht mehr geprüft, weil diese vertrauenswürdigen Klassen grundsächlich als sicher gelten. Wie bereits erwähnt, ist es möglich, Code von User-Defined-ClassLoader laden zu lassen. Im JDK 1.1 ist dieser User-Defined-ClassLoader in der Regel eine Instanz der Klasse *AppletClassLoader*. Das JDK der Java-2-Plattform stellt für die Implementation eines User-Defined-ClassLoaders neben der abstrakten Klasse *ClassLoader* auch die neuen Java-Klassen *SecureClassLoader* und *URLClassLoader* zur Verfügung.

3.2.3. Der Security-Manager

Der Security-Manager ist die zentrale Komponente der Sicherheitsarchitektur. Die Aufgabe des Security-Manager besteht darin, alle als potentiell gefährlich eingestuften Operationen von nicht vertrauenswürdigen Klassen zu überwachen und einzuschreiten, falls eine Verletzung der Sicherheitspolitk des lokalen Systems vorliegt. Bei allen zur Laufzeit durchzuführenden Kontrollen, so z.B. Zugriffsversuche auf sicherheitskritische Systemressourcen, befragt die JVM den Security-Manager. Es werden z.B. Aufrufe von Methoden, die einen Dateizugriff

[10] Die Namensräume wurden bereits in Kapitel 3.1.1. 'Die ClassLoader Architektur' eingeführt.

oder einen Netzwerkzugriff durchführen, Zugriffe auf Betriebssystemdienste oder die Definition eines neuen ClassLoaders kontrolliert. Der Security Manager ist grundsätzlich eine Implementierung der abstrakten Basisklasse *SecurityManager*. Sobald z.B. ein Applet ausgeführt wird, wird der Security-Manager automatisch gestartet. Bei lokal ausgeführten vertrauenswürdigen Programmen wird der Security-Manager nicht automatisch aktiviert. Seit Java 2 besteht aber die Möglichkeit, bei lokalen Anwendungen die gleichen Eingrenzungen geltend zu machen, wie es für nichtvertrauenswürdige Anwendungen aus dem Netz gilt.

3.3. Die Java Sicherheitsmodelle

Seit der Einführung von Java ist das Sicherheitsmodell bereits mehrfach verändert worden. Die Komponenten der Sicherheitsarchitektur, sie wurden im vorherigen Kapitel vorgestellt, wurden dabei entsprechend dem jeweiligen Sicherheitsmodell angepasst.

3.3.1. Das Sicherheitsmodell von JDK 1.0

Das erste Sicherheitsmodell, das mit dem JDK 1.0 eingeführt wurde, ist unter dem Namen Sandbox-Modell bekannt. Dieser traditionelle Sicherheitsentwurf beugt in erster Linie Risiken vor, indem er jedweden böswilligen Code den direkten Zugriff auf den Rechner verbietet. Dabei unterscheidet das Sicherheitsmodell von dem JDK 1.0 zwischen Bytecode (als Applet oder Java-Anwendung), der von einem entfernten Rechner geladen wird und nur in einer extrem eingeschränkten Rechteumgebung, der genannten Sandbox, ausgeführt werden kann, und den vertrauenswürdigen lokalen Java-Anwendungen. Der lokale Code hat alle Rechte, die auf dem System befindlichen Ressourcen zu nutzen. Nichtvertrauenswürdigem Code wird der Zugriff auf Systemressourcen nur gewährt, wenn diese Zugriffsrechte explizit in den ACL's[11] der JVM erteilt werden. Dazu musste ein eigener Custom-Security-Manager geschrieben werden. In der folgenden Abbildung[12] ist sehr gut der Unterschied von entfernten und lokalen Code im Bezug zu den Zugriffsrechten auf die Systemressourcen zu erkennen:

[11] Ein ACL ist eine Zugriffskontrollliste. Listeneinträge identifizieren Benutzer oder Benutzergruppen, sowie die Rechte, die eingeräumt werden.

[12] C. Eckert: IT-Sicherheit, München: Oldenburger Wissenschaftsverlag GmbH, S. 431

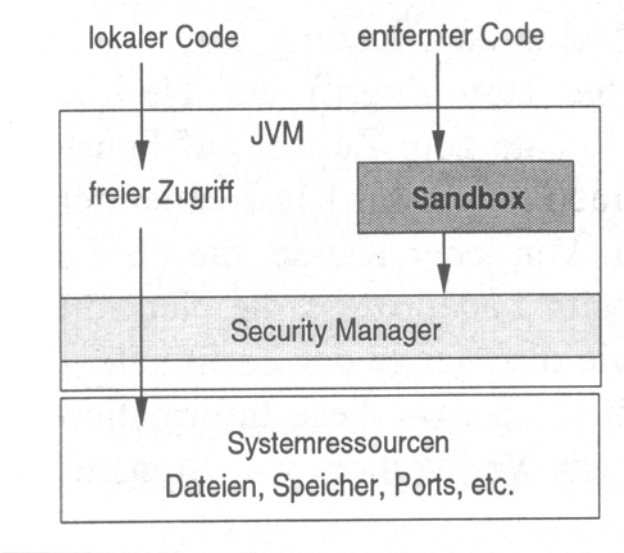


Abbildung 3.4 : Das Sicherheitsmodell von JDK 1.0

3.3.2. Das Sicherheitsmodell von JDK 1.1

Durch die extremen Begrenzungen des Sicherheitsmodells von JDK 1.0 wurde aber verhindert, dass sinnvolle Client-Server-Anwendungen, die auf eine wechselseitige Beziehung begründet sind und die auf den Zugriff von Systemressourcen abhängig sind, geschrieben werden können. Mit signierten[13] Applets wurde eine gewisse Flexibilität geschaffen, die es ermöglichte, kooperierende Anwendungen zu konstruieren. Den signierten Applets wurden die gleichen Rechte zugeordnet, die für lokalen Code galten.

Eine weitere Neuerung im JDK 1.1 war das jar-Kommando, das alle zu einem Applet gehörenden Klassen und Ressourcen (z.B. Bilder) und die Signatur des Applets in Jar-Dateien mit der Extension .jar zusammenfasst.

Eine gleichzeitige Verschärfung des Sicherheitskonzeptes für lokale Anwendungen wurde ebenfalls durchgeführt. Es wurden nur noch Java-Anwendungen uneingeschränkter Zugriff auf die Systemressourcen gewährt, die in einer Umgebungsvariable mit dem Namen *CLASSPATH* eingetragen wurden. Neben der Signaturmöglichkeit im JDK 1.1 wurde auch die Möglichkeit eingeführt, Zertifikate in Java zu erstellen. Es wurde dazu das Kryptography API bereitgestellt.

Die folgende Abbildung[14] veranschaulicht das Sicherheitsmodell vom JDK 1.1:

---

[13] Die digitale Signatur wird im Kapitel 4.6 Digitale Signatur vorgestellt.

[14] C. Eckert: IT-Sicherheit, München: Oldenburger Wissenschaftsverlag GmbH, S. 431

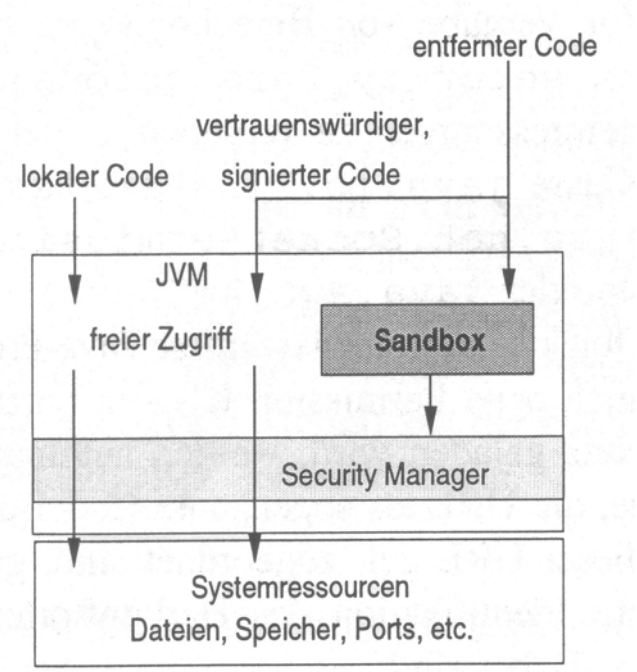


Abbildung 3.5 : Das Sicherheitsmodell von JDK 1.1

Es wurden zwar in der Version des JDK 1.1 viele Verbesserungen durchgeführt, aber die Rechtevergabe war nach wie vor recht grobgranular.

3.3.3 Das Sicherheitsmodell von Java 2

Obwohl das JDK 1.1 viele Verbesserungen zur ersten Version aufwies, war das Problem der minimal flexiblen Rechteverwaltung weiterhin gegeben. Ab dem JDK 1.2 bzw. der Java-2-Plattform wird Code gleich welcher Herkunft, ob mit oder ohne Signatur, der herrschenden Sicherheitspolitik unterworfen.
Die gewünschte Sicherheitspolitik kann dazu vom Systemadministrator oder vom Benutzer in die Textdatei java.policy geschrieben und festgelegt werden.
Wenn keine Sicherheits-Police festgelegt wird, greift die JVM auf die Standardstrategie, die dem Sandbox-Modell entspricht, zurück. In die Sicherheits-Police werden die Berechtigungen, die an den Bytecode gegeben werden, geschrieben. Dabei wird die Rechtevergabe vom Herkunftsort (durch die URL festgelegt) des Bytecode und von verschiedenen Signaturen abhängig gemacht. Für die Vergabe von Berechtigungen wurde eine neue abstrakte Klasse *java.security.Permissions* eingeführt, die in der nochmals erweiterten Kryptography API zu finden ist. Damit eigene Rechte an Systemressourcen festgelegt werden können, werden Subklassen von der abstrakten Klasse *java.security.Permissions* bereitgestellt. Die Klasse java.net.SocketPermission enthält z.B. Rechte zum Zugriff auf Netzdienste.

Eine weitere bedeutende Änderung zum JDK 1.1 ist die Einführung von Schutzbereichen. Diese Schutzbereiche (protection domain) ordnen Klassen gleicher Herkunft eine oder mehrere Berechtigungen (Permission) zum Zugriff auf bestimmte Ressourcen zu. Domänübergreifende Aktionen können nur über vertrauenswürdigen Systemcode erfolgen oder diese Berechtigung (Permission) muss explizit den beteiligten Domänen gegeben werden. Die JVM erzeugt zur Laufzeit die Domänen (Schutzbereiche) dynamisch. Dabei werden alle Klassen, die mit dem gleichen Schlüssel verschlüsselt wurden oder die gleiche URL besitzen, in einer Domäne zusammen gefasst. Ein Beispiel für eine Domäne kann die Domäne für nicht vertrauenswürdigen Code sein, die Sandbox-Domäne. Ein weiteres Beispiel ist die System-Domäne, die den gesamten Code des JDK enthält, sie hat als einzige Domäne die Berechtigung, auf die Systemressourcen direkt zuzugreifen.

Die nachfolgende Abbildung[15] veranschaulicht die Ausführungen:

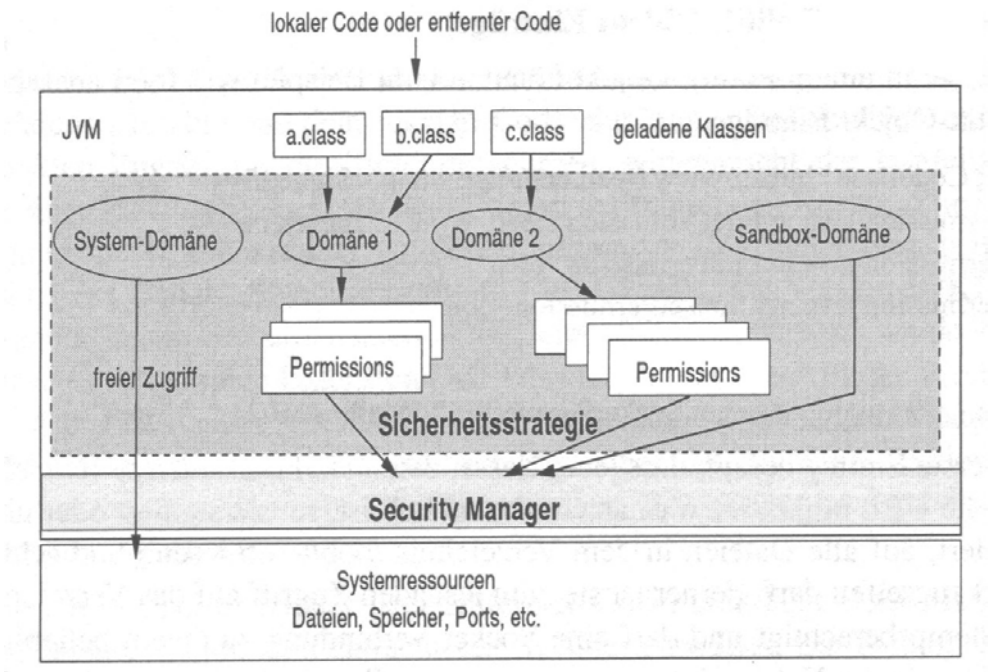


Abbildung 3.6 : Das Sicherheitsmodell von Java 2

Die Zugriffskontrolle ist nicht mehr fest in der Programmlogik des Security Managers verankert, sondern wird dem so genannten Access Controller übergeben. Der Access Controller ermöglicht eine feingranulare und vor allem flexiblere Zuteilung von Zugriffsrechten. Es besteht nun die Möglichkeit, in Abhängigkeit von der Herkunft (URL) oder dem Unterzeichner gezielt Berechtigungen, so zum Be-

[15] C. Eckert: IT-Sicherheit, München: Oldenburger Wissenschaftsverlag GmbH, S. 434

spiel Schreibrechte auf das Dateisystem, einer bestimmten Java-Anwendung einzuräumen.

Obwohl das Sicherheitsmodell von Java 2 ein sehr hohes Maß an Sicherheit bietet, ist es in speziellen Fällen notwendig, zusätzlich auf die Sicherheitsaspekte zurückzugreifen, welche die Kryptographie anbietet. Einige Sicherheitsaspekte des JDK z.B. die der digitalen Signatur sind Teil der Kryptographie. Die theoretische Grundlagen der Kryptographie werden im nachfolgenden Kapitel erläutert.

4. Kryptographie

Eines der grundlegenden Themen, das wegen der immer stärker zunehmenden Verbreitung elektronischer Datenverarbeitung und Datenübermittlung in der Bedeutung zugenommen hat, ist das Problem der Sicherheit. Insbesondere bei Internet- und Client – Server-Anwendungen wie der JCoolClassLoader oder das Modul JCool, die in den letzten Jahren stark expandiert haben, hat das Thema Sicherheit und somit die Kryptographie an Bedeutung gewonnen.

4.1. Grundlagen

Die unter dem Bereich der theoretischen Mathematik eingeordnete Kryptologie besteht aus Verfahren, die dazu eingesetzt werden, Informationen durch Transformation der Daten unkenntlich zu machen und zu schützen.
Die Kryptologie wird in zwei Teilbereiche eingeteilt, die Kryptographie und die Kryptoanalyse.
Die Kryptographie befasst sich mit dem Ver- und Entschlüsseln von Daten, dabei wird ein Klartext in einen nicht mehr allgemein lesbaren Chiffretext umgewandelt, der anschließend wieder entschlüsselt wird, um als Klartext lesbar zu sein.

Abbildung 4.1 : Kryptographie

Die Kryptoanalyse beschreibt dagegen Methoden der unbefugten Entschlüsselung von Daten.

Die Kryptoanalyse wird in dieser Arbeit nicht beschrieben, weil es sich um einen eigenständigen Bereich handelt und die konkrete Aufgabenstellung der vorliegenden Arbeit nicht tangiert.

4.1.1. Ziele der Kryptographie

Wurde früher die Kryptographie ausschließlich dazu genutzt, um den Inhalt einer Nachricht unlesbar zu machen, z.B. eine der ältesten Verschlüsselungsalgorithmen die Caesar Verschlüsselung[16], so sind die Ziele der Kryptographie heute umfangreicher.
Die Ziele der Kryptographie lassen sich in fünf Bereiche unterteilen:

- Vertraulichkeit
- Integrität
- Authentifikation
- Verbindlichkeit
- Anonymität

Die Vertraulichkeit bedeutet, dass der Inhalt eines Dokuments nur von dazu befugten Personen gelesen werden kann.
Die Integrität stellt sicher, dass der Inhalt eines Dokuments nicht unbemerkt verändert werden kann.
Die Authentifikation stellt sicher, dass der Urheber eines Dokuments feststellbar sein soll. Kein anderer soll sich als Urheber ausgeben können.
Die Verbindlichkeit sagt aus, dass der Urheber eines Dokuments seine Urheberschaft nicht abstreiten kann.
Die Anonymität steht für die Vertraulichkeit des gesamten Kommunikationsvorgangs, und nicht nur für die Vertraulichkeit des Nachrichteninhalts.

[16] Julius Caesar (100 – 44 v. Chr.) ersetzte die Buchstaben seiner Nachricht durch Buchstaben im Alphabet, die drei Buchstaben später positioniert sind.

4.1.2. Auswahl eines Verschlüsselungsalgorithmus

Die Auswahl, welcher Algorithmus für ein Verschlüsselungsverfahren genutzt werden soll, hängt im wesentlichen nicht davon ab, wie geheim ein solcher Algorithmus gehalten wird, sondern von der verwendeten Schlüssellänge.
Die Auswahl eines Algorithmus sollte sich daher an die folgenden wichtigen Grundregeln der kryptologischen Wissenschaft orientieren:

Die Grundlage des amerikanischen Informationstheorethikers Claude Shannon lautet:

> *Der Feind kennt das benutzte kryptographische Verfahren.*
> *(Shannons Maxime)*

Der belgische Kryptograph Auguste Kerckhoff verfasste die wohl am häufigsten genannte Grundregel der Kryptographie:

> *Die Sicherheit eines kryptographischen Verfahrens beruht allein auf dem Schlüssel, der zum Dechiffrieren benötigt wird.*
> *(Kerckhoffs Maxime)*

Da es demnach ohnehin wenig Sinn macht, einen Algorithmus geheimzuhalten, wird in der kryptologischen Wissenschaft die Veröffentlichung eines Algorithmus als wichtigste Voraussetzung angesehen, um ihn als sicher einschätzen zu können. Denn nur eine Überprüfung durch die gesamte wissenschaftliche Gemeinschaft sowie insbesondere ausgiebige Kryptoanalyse können Schwächen eines Algorithmus hinreichend zuverlässig aufdecken.

4.2. Symmetrische Verschlüsselung

Bei der symmetrischen Verschlüsselung wird sowohl zum Chiffrieren als auch zum Dechiffrieren derselbe Schlüssel verwendet. Dieser muss beiden Kommunikationspartnern (Sender und Empfänger) bekannt sein.

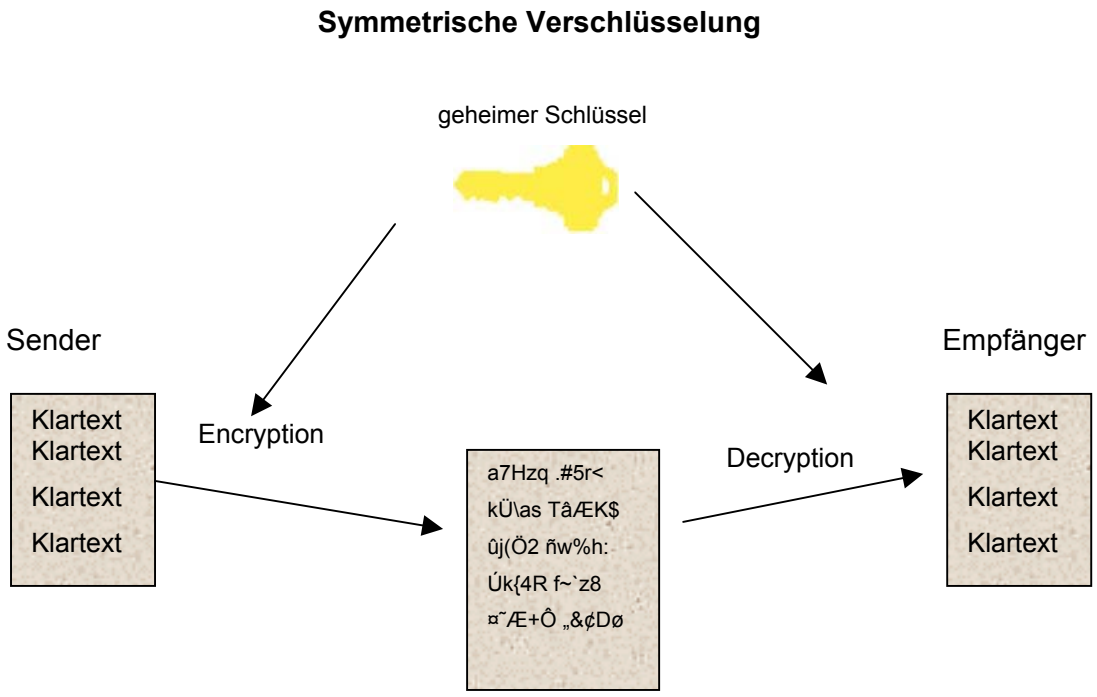


Abbildung 4.2 : Symmetrische Verschlüsselung

Voraussetzung ist, dass dem Sender und dem Empfänger der gemeinsame Schlüssel bekannt ist. Möchte der Sender nun ein Klartextdokument an den Empfänger schicken, verschlüsselt er das Klartextdokument mit dem gemeinsamen geheimen Schlüssel und einem Verschlüsselungsalgorithmus, und sendet das chiffrierte Dokument an den Empfänger. Der Empfänger kann das chiffrierte Dokument nun mit dem gleichen Schlüssel entschlüsseln und erhält wieder das Klartextdokument.

Ein zentraler Punkt der symmetrischen Verschlüsselung ist, dass der Schlüssel vor dem Zugriff von unbefugten Dritten zur Wahrung der Vertraulichkeit geschützt bzw. geheim gehalten werden muss. Besonders kritisch ist deswegen der Vorgang des Schlüsselaustauschs zwischen Sender und Empfänger.
Ein großer Vorteil der symmetrischen Verschlüsselung ist, dass durch die relativ kurzen Schlüssellängen, die zwischen 56 und 256 Bit liegen, eine hohe Verschlüsselungsgeschwindigkeit gegeben ist. Zwar wird häufig gesagt, dass die

kurze Schlüssellänge die Sicherheit der Algorithmen herabsetzt, aber auch der verwendete Algorithmus, der genutzt wird, beeinflusst die Sicherheit.
Hierzu wird an späterer Stelle der DES-Algorithmus und der IDEA-Algorithmus näher betrachtet.

Im allgemeinen wird bei der symmetrischen Verschlüsselung zwischen Stromverschlüsselung und Blockverschlüsselung unterschieden.

4.2.1. Stromchiffrierung

Algorithmen, die mit der Stromchiffrierung arbeiten, erzeugen aus einem kontinuierlichen Strom von Klartextzeichen und einem Chiffrierschlüssel einen kontinuierlichen Strom von Chiffretextzeichen.

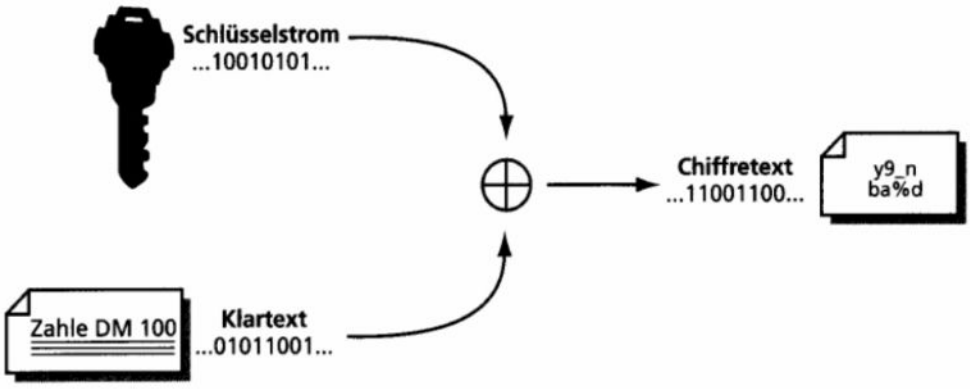


Abbildung 4.3 : Stromchiffrierung[17]

Die Vernam-Chiffrierung[18] ist die einfachste Form der Stromchiffrierung, und obwohl sie als nicht mehr sicher gilt, wird sie in einigen Algorithmen noch verwendet. Denn wenn sich der Schlüsselstrom fortlaufend ändert, ist eine relativ hohe Sicherheit gegeben. Bei der Vernam-Chiffrierung wird jeweils ein Bit des Schlüssels und ein Bit des Klartextes XOR verknüpft und zu einem Bit Chiffretext addiert. Zur Verschlüsselung werden Pseudozufallszahlen verwendet, die die Schlüsselfolge darstellen. Der Schlüsselstrom wird auch One-Time-Pad genannt, wenn jedes Klartextbit mit einem zufälligen Schlüsselbit chiffriert wird.

---

[17] Richard E. Smith: Internet-Kryptographie, Bonn: Addison-Wesley-Longman 1998, S. 53

[18] Die Vernam-Chiffrierung wurde Anfang des zwanzigsten Jahrhunderts zur Verschlüsselung von Fernschreibern entwickelt.

Man unterscheidet im Allgemeinen zwischen synchronen und selbst-synchronisierenden Stromchiffren. Bei den synchronen Stromchiffren wird der Schlüsselstrom separat generiert. Die selbst-synchronisierenden Stromchiffren erzeugen die Schlüsselfolge dagegen selbst aus den Daten.

Der wohl bekannteste Algorithmus, der die Stromchiffrierung verwendet, ist der RC4 (Rivest Chiper 4) Algorithmus, der von Ron Rivest bei der RSA Data Security entwickelt wurde. Der RC4 hat eine variable Schlüssellänge, die bis zu 2048 Bit betragen kann, üblicher ist aber eine Schlüssellänge von 128 Bit. Der RC4 Algorithmus erzeugt Pseudozufallszahlen, deren Periode von über 10^{100} , d.h. erst nach 10^{100} Verschlüsselungen tritt eine Wiederholung des Schlüsselstroms auf, eine Kryptoanalyse stark erschweren. Der RC4 Algorithmus gehört auch zu den wenigen Algorithmen, die kommerziell genutzt werden. Zu erwähnen ist auch noch der SEAL – Algorithmus (Software-optimized Encryption Algorithm), der von Phil Rogaway und Den Coppersnüth 1993 bei IBM entwickelt wurde. Der SEAL verwendet mit dem sicheren Hashalgorithmus SHA-1[19] initialisierte Lookuptabellen. Eine Besonderheit des SEAL ist, dass ein bestimmter Datenblock *i* im Voraus berechnet werden kann, ohne alle Blöcke vor *i* zu ver- bzw. entschlüsseln, dazu muss nur die Position des Blocks bekannt sein.

4.2.2. Blockchiffrierung

Bei der Blockchiffrierung wird der Klartext in gleich große Datenblöcke aufgeteilt, gängige Größen sind dabei 64 Bit oder 128 Bit große Blöcke. In der Regel wird dabei der letzte Block, wenn nötig, ausgedehnt (padding). Diese Datenblöcke werden mit einem Schlüssel zu gleichgroßen Chiffretextblöcken verschlüsselt. Die folgende Abbildung stellt dieses Verfahren dar.

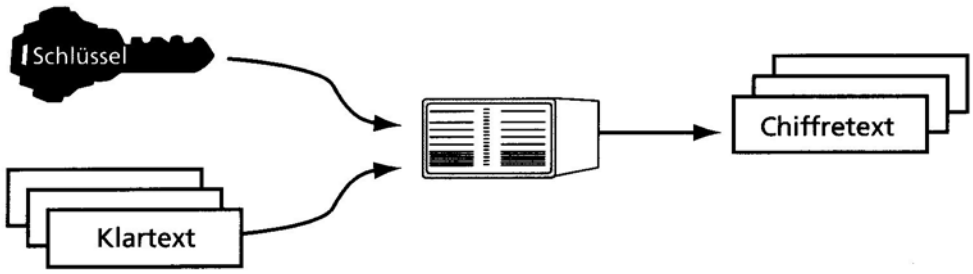


Abbildung 4.4 : Blockchiffrierung[20]

[19] Der SHA-1 wird unter dem Unterkapitel Hashfunktionen behandelt.
[20] Richard E. Smith: Internet-Kryptographie Bonn: Addison-Wesley-Longman 1998, S. 54

Eine Bedingung bei den Blockchiffrieralgorithmen ist, dass bei der Chiffrierung möglichst keine Gesetzmäßigkeiten bei den Chiffretextblöcken zu erkennen ist. Denn erkennbare Gesetzmäßigkeiten bieten Codebrechern die Möglichkeit den Chiffretext zu knacken.
Die gängigsten angewendeten symmetrischen Verschlüsselungsverfahren sind Blockchiffren.
Dazu gehören der DES, Triple-DES, AES (Rijndael), IDEA, Blowfish, Twofish, MARS und der Serpent .
In den folgenden Abschnitten werden der DES-, Tripel-DES- und IDEA-Algorithmus näher erläutert.

4.2.2.1. DES-Algorithmus

Der wohl bekannteste symmetrische Algorithmus unter den Blockchiffren ist der DES (Data Encryption Standard). Der DES ist aus der Zusammenarbeit von IBM und der NBC (National Security Agency) entstanden und ging aus dem ursprünglich von IBM Projekt Lucifer[21] hervor. 1977 wurde der DES in den USA als ANSI-Standard normiert. Ein großes Einsatzgebiet vom DES war und ist im Chipkartenumfeld zu finden. Der Standard wurde seit 1983 alle fünf Jahre neu zertifiziert, da aber wegen der kurzen Schlüssellänge die Sicherheit nicht mehr gegeben war, lief die letzte Zertifizierung 1998 aus. Der DES arbeitet mit Feistel-Netzwerken[22] und gehört zu der Gruppe der Feistel-Chiffren, zu denen auch die Algorithmen RC5, RC6, Blowfish, Twofish und MARS gehören. Dabei wird der in Blöcke zerlegte Klartext blockweise in zwei Teilblöcke geteilt, danach in mehreren Runden jeweils verschlüsselt und im Anschluss wieder zusammengesetzt. Dabei wird für jede Runde ein neuer Schlüssel generiert. Der Chiffretext setzt sich dann aus den verschlüsselten Teilblöcken zusammen. Der Chiffretext kann mit der gleichen Methode, aber mit dem inversen Schlüssel, wieder in Klartext umgewandelt werden.

Der DES ist ein Blockchiffre der mit 64 Bit Datenblöcken arbeitet. Der Schlüssel hat ebenso eine Länge von 64 Bit, wobei aber jedes achte Bit ein Paritätsbit zur Fehlerkorrektur ist, und der Schlüssel im Endeffekt eine Länge von 56 Bit hat. Jeder 64 Bit Klartextblock hat 16 Runden zu durchlaufen.

---

[21] Das Projekt Lucifer wurde in den 70'er Jahren von einem Kryptographenteam bei IBM durchgeführt. Das Ziel des Projektes war, Verschlüsselungstechnologien zu finden, die effizient arbeiten und leicht in Hardware zu implementieren sind.

Der Ablauf einer DES Verschlüsselung stellt sich folgendermaßen dar:

Aus dem geheimen Schlüssel wird der Rundenschlüssel erzeugt. Das geschieht, indem nach der Schlüsselpermutation der 56 Bit Schlüssel in jeder Runde in zwei Hälften von je 28 Bit geteilt wird, und abhängig vom Rundenindex 1 oder 2 Bits nach links geshiftet wird. Danach werden die zwei 28 Bit Teile wieder zusammengefügt und mit einer Kompressionspermutation, bei der bestimmte Bits nach einer Tabelle vertauscht werden - 8 Bits bleiben dabei unberücksichtigt - zu einem 48 Bit Block langen Teilschlüssel K(i) gefertigt. Es ist möglich, die Teilschlüssel z.B. für längere Texte zu speichern.

Der 64 Bit lange Klartextblock wird in zwei Hälften (L,R) zu je 32 Bit geteilt. Der L(i)-Teil bleibt zuerst unverändert, auf den R-Teil wird eine so genannte Expansionspermutation durchgeführt, bei der die Bits von R untereinander vertauscht werden, dabei treten einzelne Eingabebits mehrfach in der Ausgabe aus. Das daraus entstandene 48 Bit lange R(i) -Teil wird danach in jeder Runde bitweise XOR mit dem Rundenschlüssel K(i) verknüpft. Das entstandene Resultat wird in 8 Blöcke zu je 6 Bit aufgeteilt und dient als Eingabe für die S-Boxen[23]. Die jeweiligen S-Boxen liefern jeweils 4 Bit zurück, also insgesamt 32 Bit. Dieser nun erfolgte Schritt wird als Konfusion[24] bezeichnet und stellt die maßgebliche Operation zur Sicherheit des DES dar. Anschließend wird auf die 32 Bit die P-Box-Permutation angewandt, die eine einfache Bitvertauschung nach einer festgelegten Tabelle vornimmt. Dies wird wie bei der Eingangspermutation als Diffusion[25] bezeichnet. Als letztes in jeder Runde, außer der letzten, wird dieses Ergebnis mit L(i) XOR verknüpft und bildet nun die nächsten 32 Bit R(i). Das eigentliche unveränderte R(i) bildet dann den nächsten L(i). Nachdem 16 Runden durchlaufen sind, wird eine Abschlusspermutation durchgeführt, die invers zu der Eingangspermutation ist. Das entstandene Ergebnis ist der Chiffretextblock. Die einzelnen Chiffretextblöcke werden dann zum Chiffretext zusammengefügt.
Die folgende Abbildung verdeutlicht die Ausführungen.

---

[22] Horst Feistel hat die Feistel-Netzwerke in den 70'er Jahren im Rahmen des schon erwähnten Projektes Lucifer bei IBM entwickelt.

[23] Bei den S-Boxen handelt es sich um Funktionen, die eine m-n Abbildung realisieren. S-Boxen sind feste Bestandteile von Feisel-Netzen. Der Aufbau der Boxen ist von DES fest vorgegeben.

[24] Prinzip der Konfusion nach Shannon: Konfusion bedeutet, dass der Zusammenhang von Klartext, Schlüssel und Chiffretext so komplex wie möglich gehalten wird, damit die Kryptoanalyse erschwert wird.

[25] Prinzip der Diffusion nach Shannon: Mit Diffusion wird die Verteilung der in einem Klartext enthaltenen Regelmäßigkeiten im zugehörigen Geheimtext beschrieben.

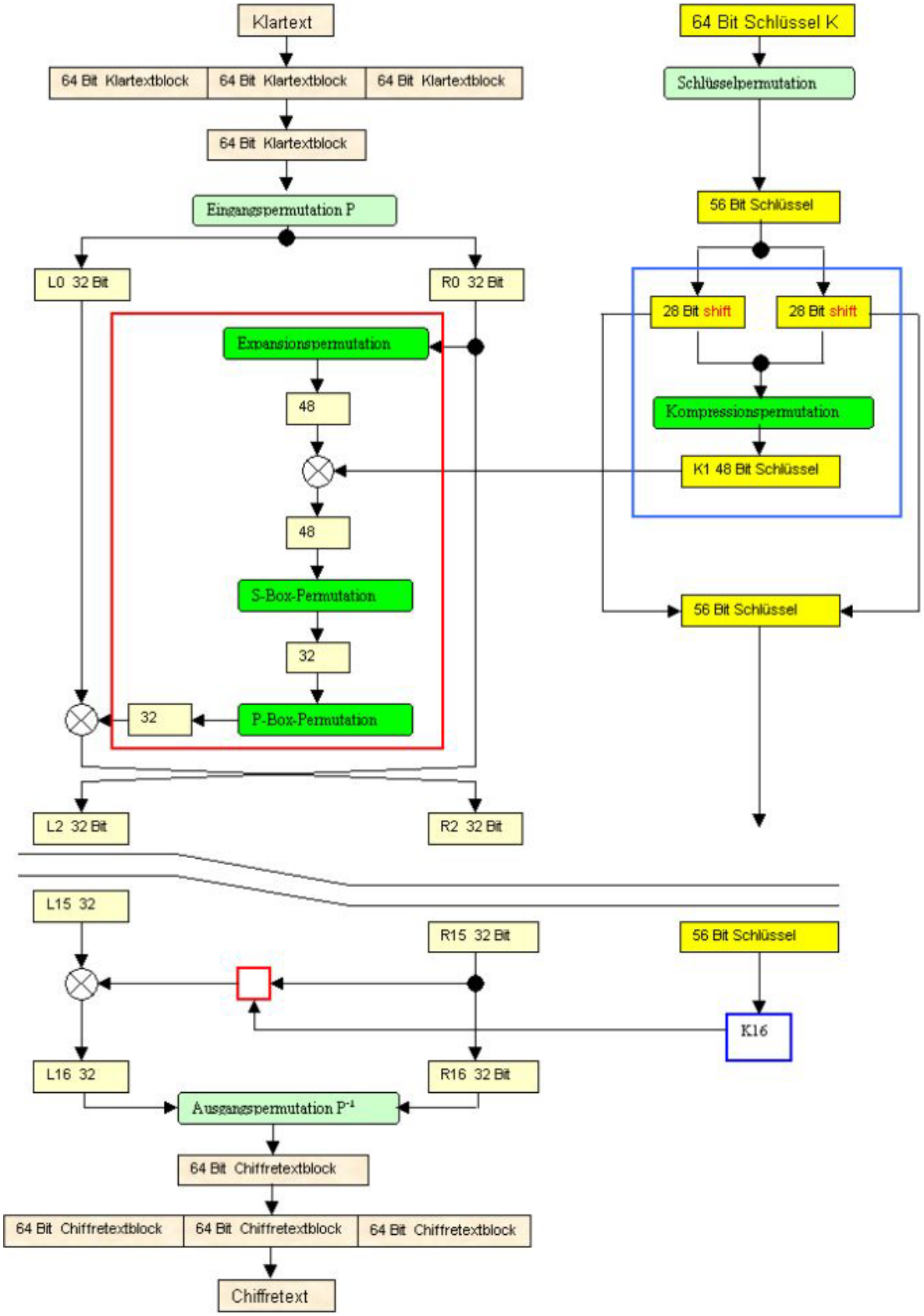


Abbildung 4.5 : Ablauf der DES-Verschlüsselung

4.2.2.2. Tripel-DES-Algorithmus

Da das einfache DES-Verfahren wegen der kurzen Schlüssellänge als nicht mehr sicher eingestuft wurde, wird als ein Nachfolger des DES der Tripel-DES genannt.

Von dem Tripel-DES gibt es verschiedene Varianten, einmal mit einer Schlüssellänge von 112 Bit (2*56 Bit) und einer Schlüssellänge von 168 Bit (3*56 Bit).

Bei dem Tripel-DES wird der Klartext dreimal mit zwei oder drei unterschiedlichen Schlüsseln verschlüsselt.

Bei der Variante mit dem 112 Bit Schlüssel, also zwei 56 Bit Schlüssel, wird zuerst der Klartextblock mit dem ersten Schlüssel verschlüsselt, dann mit dem zweiten entschlüsselt und dann wieder mit dem ersten Schlüssel verschlüsselt.

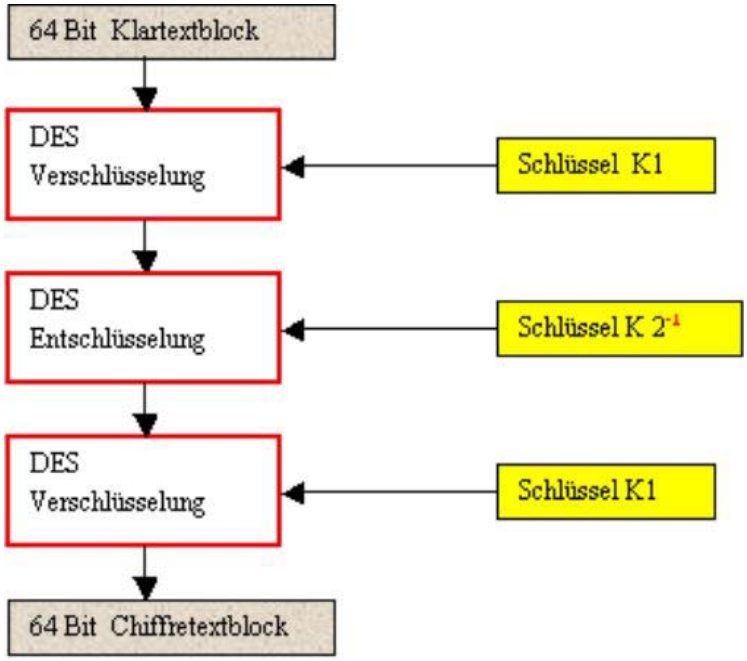


Abbildung 4.6 : Tripel-DES 112 Bit Schlüssel

Bei der zweiten Variante mit dem 168 Bit Schlüssel, also drei 56 Bit Schlüssel, wird der Klartextblock zuerst mit dem ersten Schlüssel verschlüsselt, dann mit dem zweiten Schlüssel entschlüsselt und dann wieder mit dem dritten Schlüssel verschlüsselt.

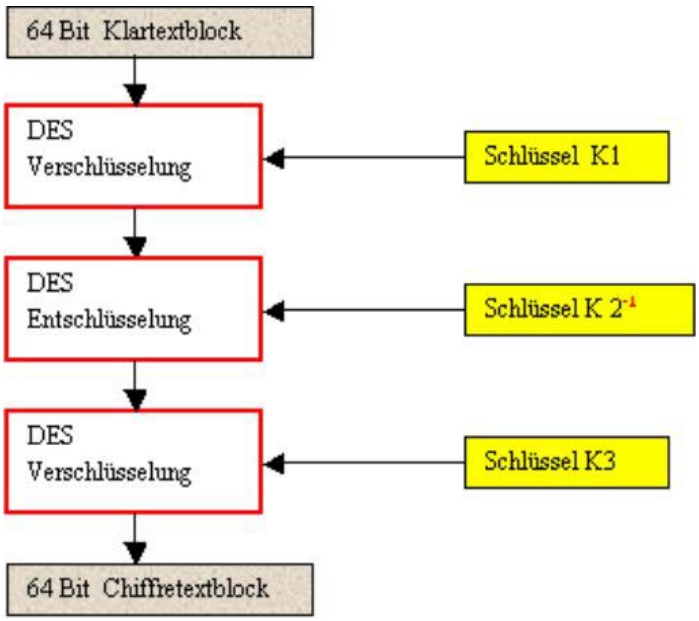


Abbildung 4.7 : Tripel-DES 168 Bit Schlüssel

Diese Verfahren sind auch als EDE (Encryption-Decryption-Encryption)- Verfahren bekannt. Durch die beschriebenen Verfahren wird der Tripel-DES für größere Dokumente aber zu langsam. Krypto-Programme benutzen deshalb modernere und vor allem schnellere Algorithmen.

Der offizelle Nachfolger des DES ist der AES (Advanced Encryption Standard), der auf den Rijndael-Algorithmus[26] beruht. Der Rijndael-Algorithmus wurde im Jahre 2000 vom National Institute of Standards und Technology (NIST) zum Sieger des von ihnen ausgeschriebenen Wettbewerbes zur Findung eines neuen Standards ausgewählt.
AES ist ein symmetrisches Verschlüsselungsverfahren, das 128-, 192- und 256 Bit Schlüssel unterstützt. Damit werden Brute-Force-Angriffe[27] auf Jahre so gut wie unmöglich gemacht. Der 56 Bit Schlüssel von DES konnte mit entsprechender Hardware nach einigen Stunden geknackt werden. Ein Vorteil des AES ist unter anderem, dass er nicht patentiert ist und so kostenlos eingesetzt werden kann.

[26] Der Rijndael-Algorithmus wurde von Joan Daemen und Vincent Rijmen entwickelt.
[27] Bei Brute-Force-Angriffen werden alle möglichen Schlüssel durchprobiert. Bei kurzen Schlüsseln ist es bei der jetzigen Rechenleistung der Computer möglich, diese in einigen Stunden herauszufinden.

4.2.2.3. IDEA-Algorithmus

Der IDEA (International Data Encryption Algorithm) wurde 1990 von Xuejia Lai und James Massey an der Eidgenössischen Technischen Hochschule in Zürich entwickelt. Der IDEA ist wie der DES ein Blockchiffre und arbeitet mit Blöcken von 64 Bit aber mit einem 128 Bit Schlüssel[28]. Der IDEA-Algorithmus mit dem 128 Bit Schlüssel bietet sogar größere Sicherheit als der RSA-Algorithmus[29] mit einer Schlüssellänge von 1024 Bit. Der IDEA durchläuft acht Runden, benutzt aber keine Permutation oder Substitution, sondern drei einfache Rechenoperationen,

- XOR-Verknüpfung, 
- Addition modulo 2^{16} und 
- Multiplikation modulo $2^{16}+1$, 

wodurch die Implementation in Hard- und Software einfacher und schneller erfolgen kann. Da die Rechenoperationen auf 16 Bit Teilblöcken erfolgen, ist der IDEA insbesondere auf 16 Bit Prozessoren sehr schnell. Der IDEA ist in etwa doppelt so schnell wie der DES.

Der Ablauf der IDEA Verschlüsselung stellt sich folgendermaßen dar:

Vor der Ausführung werden aus dem 128 Bit Schlüssel 52 Teilschlüssel zu 16 Bit erzeugt. 6 Schlüssel für jede Runde und 4 für die Ausgabetransformation. Der 128 Bit Schlüssel wird dabei zuerst in 8 Teilblöcke zu 16 Bit zerlegt, dies sind die ersten 8 Teilschlüssel. Danach wird der 128 Bit Schlüssel 25 Bit nach links geshiftet und daraus die nächsten 8 Teilschlüssel erzeugt. Das geht solange, bis 52 Teilschlüssel erzeugt sind. Von den letzten acht erzeugten Teilschlüsseln werden die vier Teilschlüssel mit dem höchstwertigen Bits gewählt. In der folgenden Abbildung ist der Sachverhalt dargestellt.

---

[28] Durch den 128 Bit Schlüssel ist der IDEA-Algorithmus geschützt vor Brute-Force-Angriffen.

[29] Das RSA-Verfahren wird in dem Kapitel "Asymmetrische Veschlüsselung" näher erläutert.

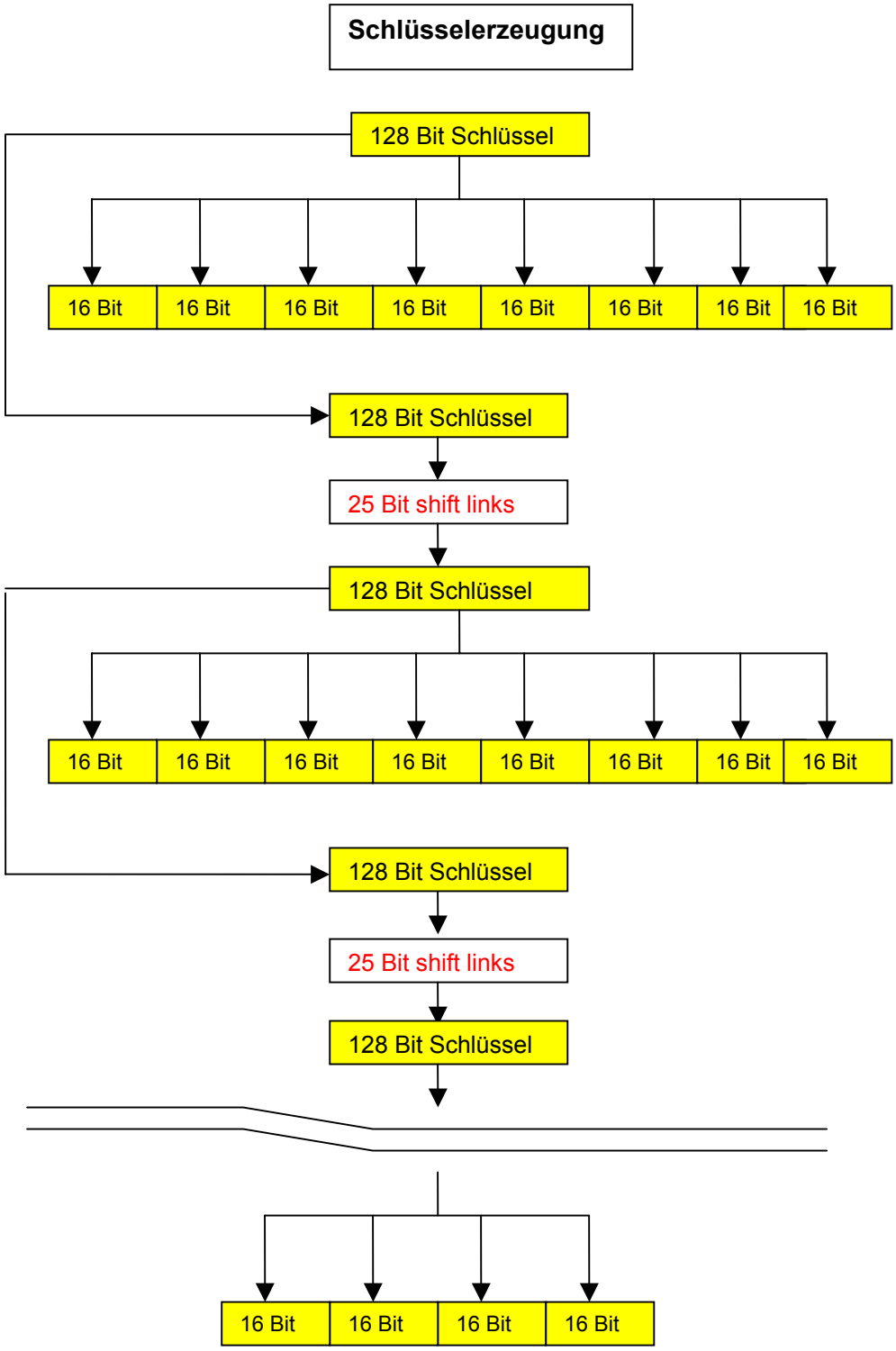


Abbildung 4.8: Erzeugung der 52 IDEA-Teilschlüssel

Für eine Verschlüsselungsrunde wird der 64 Bit Klartextblock in vier Teilblöcke zu 16 Bit zerlegt und danach miteinander und mit den 6 Teilschlüsseln verknüpft, wobei die genannten Rechenoperationen genutzt werden. Als Besonderheit sei noch erwähnt, dass bei der Multiplikation statt der Zahl 0 die Zahl 2^{16} zu verwenden ist, dadurch ist die Multiplikationsoperation immer umkehrbar, da ansonsten der Chiffretext nicht entschlüsselt werden kann.

Am Ende jeder Runde werden der zweite und der dritte Teilblock vertauscht. Nach der letzten Runde werden die vier Teilböcke in einer Ausgabetransformation mit den letzten vier Teilschlüsseln verknüpft und zum 64 Bit Chiffretext zusammengefügt. Den Ablauf der IDEA-Verschlüsselung in einzelnen Schritten zeigt die folgende Abbildung:

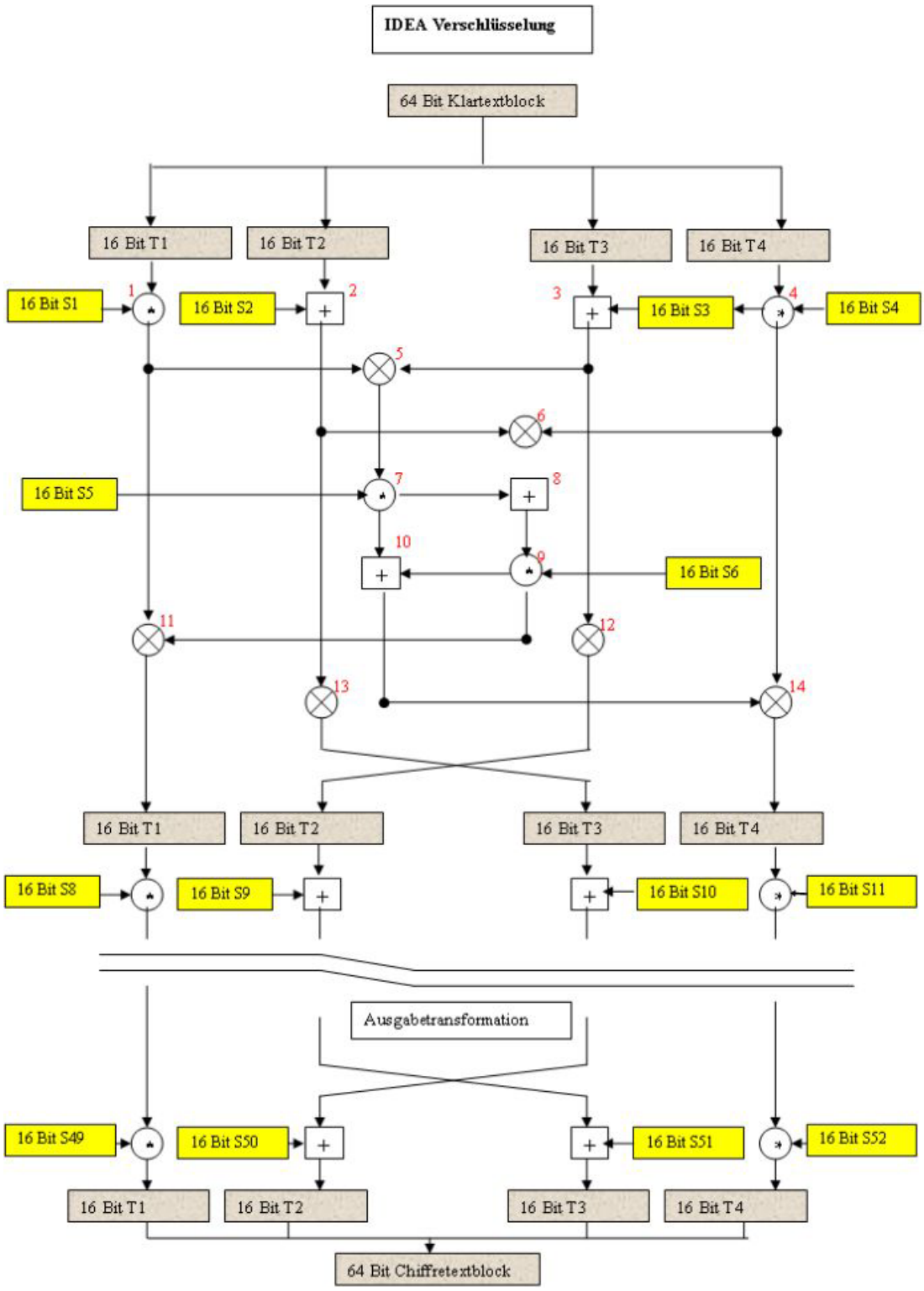


Abbildung 4.9 : Ablauf der IDEA-Verschlüsselung

Die Entschlüsselung läuft mit dem gleichen 128 Bit Schlüssel und dem gleichen Algorithmus ab, allerdings wird die Inverse für die einzelnen Teilschlüssel benötigt.

Der IDEA-Algorithmus wird auch, insbesondere wegen seiner Schnelligkeit, bei dem Softwarepaket PGP (Pretty Good Privacy) verwendet, das besonders bei E-Mail Verschlüsselung eingesetzt wird.

4.3. Asymmetrische Verschlüsselung

Die asymmetrische Verschlüsselung, auch Public Key Verfahren genannt, wurde 1976 von Whitfiled Diffie und Martin Hellmann konzipiert, um damit dem größten Problem der symmetrischen Verschlüsselung, dem Austausch zweier Schlüssel, entgegenzuwirken.
Seinen Namen hat die asymmetrische Verschlüsselung aufgrund der Tatsache, dass zum Verschlüsseln und zum Entschlüsseln von Nachrichten verschiedene Schlüssel benutzt werden, die voneinander nicht[30] abgeleitet werden können. Bei der asymmetrischen Verschlüsselung kann sowohl der Absender als auch der Empfänger je ein Schlüsselpaar bestehend aus einem geheimen (Private-Key) und einem öffentlichen Schlüssel (Public-Key) haben.
Die folgende Abbildung stellt eine Möglichkeit der Nutzung asymmetrischer Verfahren dar:

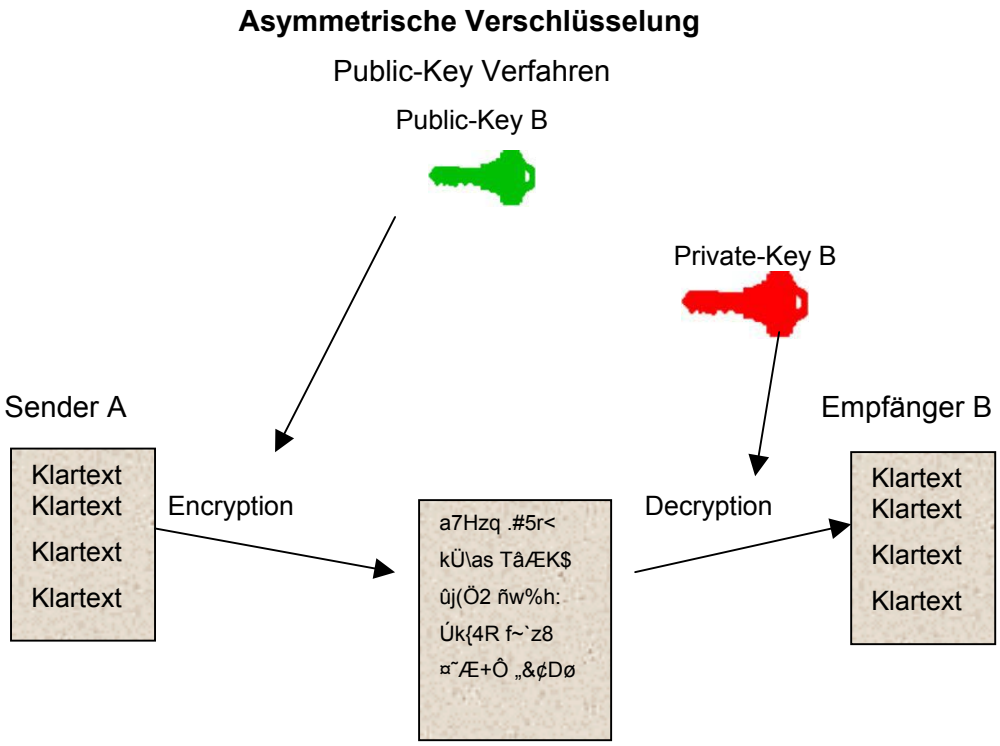


Abbildung 4.10 : Asymmetrische Verschlüsselung

Möchte nun der Sender ein geheimes Dokument an den Empfänger schicken, so verschlüsselt der Sender das Klartextdokument mit einem Verschlüsselungsalgorithmus und benutzt als Schlüssel den öffentlichen Schlüssel des Empfängers. Nachdem der Empfänger das chiffrierte Dokument erhalten hat, dechiffriert er es mit dem entsprechenden Entschlüsselungsalgorithmus und seinem privaten Schlüssel. Die Vertraulichkeit des Dokumentes ist somit gewahrt, da ein Dokument, wenn es einmal mit dem Public-Key verschlüsselt ist, ausschließlich nur mit dem dazugehörigen Private-Key entschlüsselt werden kann.

Eine weitere Möglichkeit, die einige Public-Key-Verfahren bieten, ist die Möglichkeit der Authentifizierung, die in der anschließenden Abbildung dargestellt wird. Hierbei wird ein Klartextdokument mit dem Private-Key verschlüsselt und kann nur mit dem Public-Key wieder entschlüsselt werden. Entschlüsselt nun der Empfänger den Chiffretext mit dem Public-Key, weiß er genau, dass dieses Dokument tatsächlich nur vom Sender stammen kann, der es mit seinem Private-Key verschlüsselt hat. Dieses Verfahren nennt man auch digitale Unterschrift.

[30] Zumindest in der heutigen Zeit ist es noch nicht möglich, aus einem der Schlüssel den anderen zu generieren.

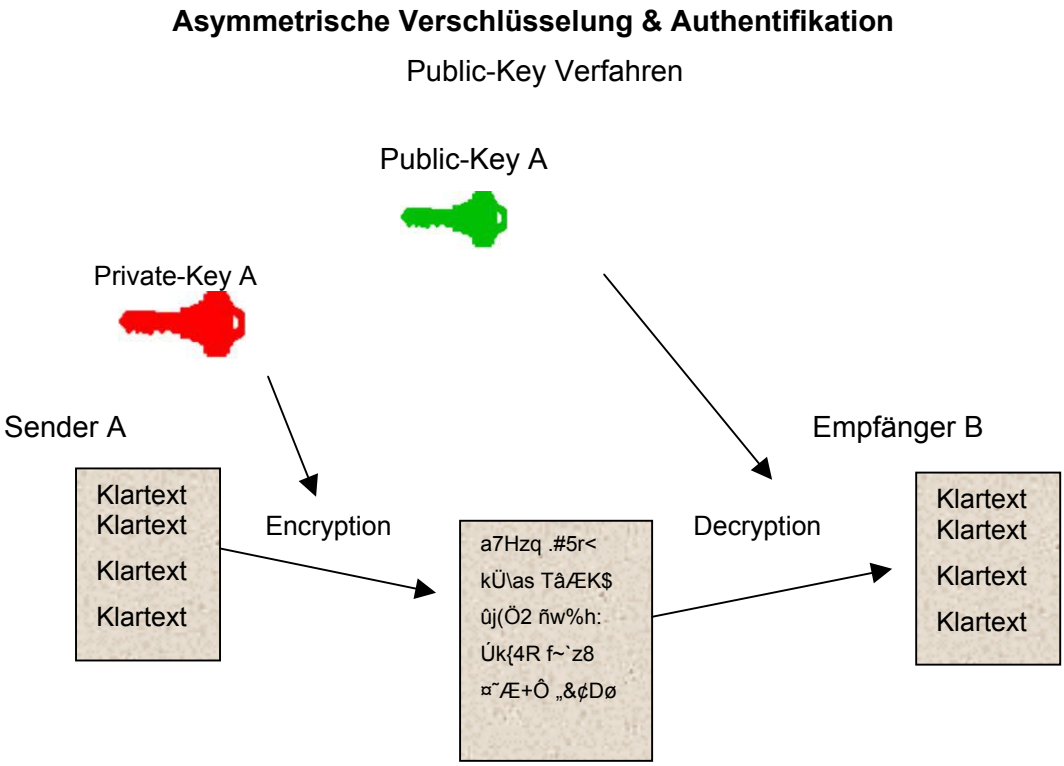


Abbildung 4.11 : Asymmetrische Verschlüsselung und Authentifikation

Der Vorteil der asymmetrischen Verschlüsselung ist, dass die Schlüsselübergabe ohne große Bedenken möglich ist, da mit den bisherigen technischen und zeitlichen Möglichkeiten der Private-Key nicht aus dem Public-Key gewonnen werden kann. Allerdings wird immer wieder betont, dass die absolute Sicherheit nach bisherigem Kenntnisstand nicht bewiesen werden kann.
Ein Nachteil der asymmetrischen Verschlüsselung ist, dass im Gegensatz zur symmetrischen Verschlüsselung extrem große Schlüssellängen benötigt werden, und dass alle bekannten Verfahren im Gegensatz zur symmetrischen Verschlüsselung einen überaus hohen Rechenaufwand haben und dadurch sehr langsam und für große Dokumente nicht geeignet sind.

Ein überaus wichtiger Punkt in Bezug auf die Sicherheit der asymmetrischen Verschlüsselung ist die Authentizität des öffentlichen Schlüssels. Es muss sichergestellt sein, dass sich hinter der Person, die den öffentlichen Schlüssel bereitstellt, tatsächlich die Person verbirgt, die sie vorgibt zu sein.
Die Authentizität kann durch Instanzen[31] sichergestellt werden, die bestätigen können, dass ein bestimmter Public-Key zu einer bestimmten real existierenden Person gehört. Dazu werden von den Instanzen Zertifikate ausgestellt. Die

asymmetrischen Verfahren beruhen auf Einwegfunktionen. Das sind Funktionen, deren Umkehrfunktion gar nicht oder nur mit einem extrem hohen Aufwand gefunden werden können. Viele asymmetrischen Verfahren werden auch als so genannte Einwegfunktionen mit Falltür bezeichnet, weil die benutzte Einwegfunktion nur mit der Falltür, dem zweiten Schlüssel, entschlüsselt werden kann. Zu den asymmetrischen Verschlüsselungsalgorithmen gehören der
Diffie-Hellmann, ElGamal, DSS, DAS, RSA und die Elliptische Kurven.
Die genannten Algorithmen werden in den folgenden Kapiteln näher besprochen.

4.3.1. Diffie-Hellmann Algorithmus

Der Diffie-Hellmann Algorithmus wurde 1976 von Whitfield Diffie und Martin Hellmann entwickelt und gilt als Meilenstein in der Kryptographie, weil es der erste Public-Key-Algorithmus war. Das Verfahren eignet sich zum Austausch von Schlüsseln über unsichere Kanäle. Mit dem Diffie-Hellmann Algorithmus können zwei oder mehrere Teilnehmer einen gemeinsamen Schlüssel erzeugen, der zur symmetrischen Verschlüsselung genutzt werden kann.
Das Verfahren wird als besonders sicher eingestuft, weil es auf der Schwierigkeit beruht, den diskreten Logarithmus[32] einer Zahl bezüglich eines großen Modulus zu berechnen.
Nachfolgend ist das Verfahren der Schlüsselerzeugung erläutert:

Beide Kommunikationsteilnehmer (A und B) wählen eine große Primzahl p und eine Zahl s, dabei muss s < p sein, s und p stellen den öffentliche Teil des Schlüssels dar. Für p gilt zusätzlich die Bedingung (p -1)/2 ist ebenso eine Primzahl. Zusätzlich wählen beide Teilnehmer eine große Zufallszahl a_A und a_B, die den geheimen Teil des Schlüssels darstellen. Danach berechnen A und B die Zahlen b_A und b_B ($b_A = s^{a_A} \bmod p$, $b_B = s^{a_B} \bmod p$) und schicken diese Zahl dem jeweiligen Kommunikationspartner zu. Mit diesen Zahlen kann der gemeinsame Schlüssel k berechnet werden

---

[31] So genannte Instanzen sind unter anderem Trustcenter, Trusted Third Center oder Zertifizierungsdienstanbieter.

[32] "*Das diskrete Logarithmusproblem ist die Aufgabe, zu der gegebenen Zahl y den diskreten Logarithmus $x=\log_a y \bmod n$ zu berechnen, also das x zu bestimmen, für das gilt, $y \equiv a^x \bmod n$.* " C. Eckert: IT-Sicherheit, München: Oldenburg Wissenschaftsverlag GmbH 2001, S. 210

($k_A = b_B^{a_A} \bmod p$, $k_B = b_A^{a_B} \bmod p$), denn für k gilt: $k_A = k = k_B$.

Wie bereits erwähnt ist das Verfahren des Diffie-Hellman Algorithmus sehr sicher, aber da der Algorithmus keinerlei Authentifizierung der Kommunikationspartner beinhaltet, ist der Algorithmus anfällig gegen Man-in-the-middle Angriffe[33]. Die nachfolgende Abbildung unterstützt die gegebene Ausführung:

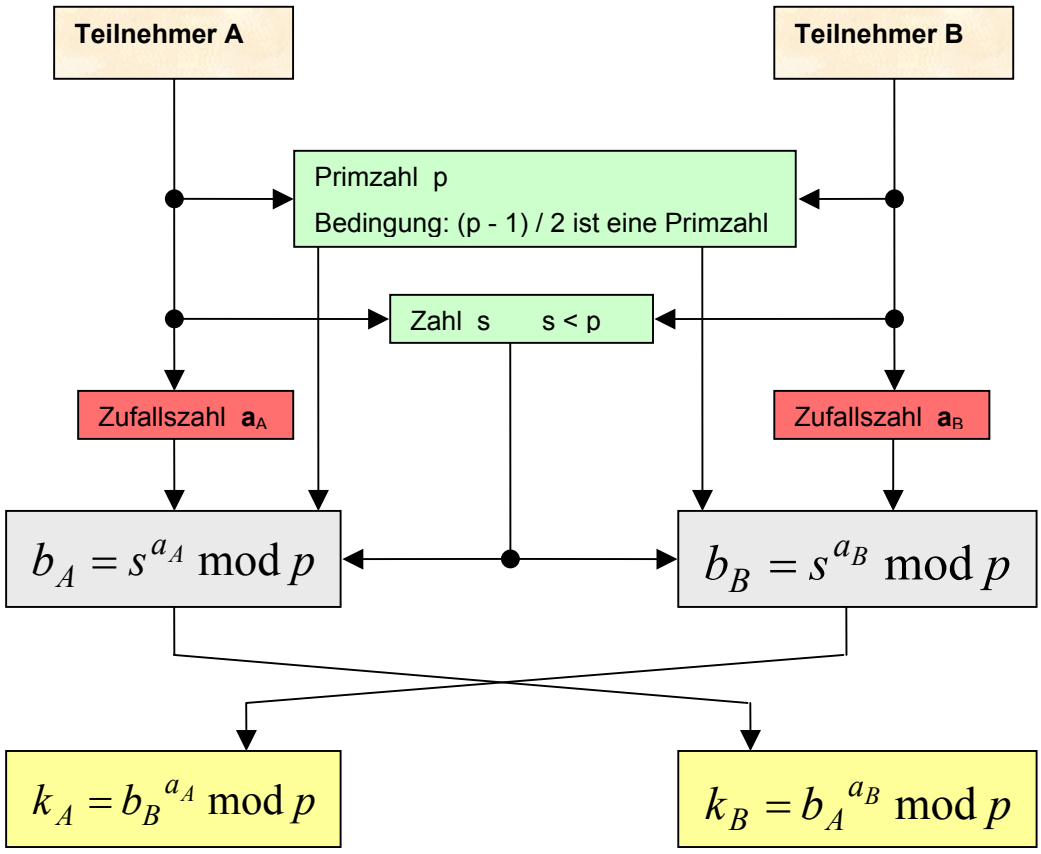


Abbildung 4.12 : Diffie-Hellmann Algorithmus

4.3.2. ElGamal-Algorithmus

Taher ElGamal veröffentlichte sein Verschlüsselungsverfahren im Jahr 1985.

Der ElGamal-Algorithmus als Verschlüsselungsverfahren ist eine Abänderung des Protokolls des Schlüsselerzeugungsverfahrens von Diffie und Hellman. Der ElGamal-Algorithmus ist bei gleicher Schlüssellänge genauso sicher wie der RSA-Algorithmus. ElGamal beruht wie der Diffie-Hellmann Algorithmus auf der

[33] Bei einem Man-in-the-Middle Angriff kann ein Angreifer die Rolle eines Teilnehmers übernehmen.

Schwierigkeit, diskrete Logarithmen zu berechnen. Diffie-Hellmann und ElGamal haben den Vorteil, dass das Verfahren seit Mitte 1997 ohne Lizenzgebühren genutzt werden kann. Deshalb bietet zum Beispiel PGP das Diffie-Hellmann Verfahren als Alternative zu RSA an.

Wie bei allen Verschlüsselungsalgorithmen kommt der Schlüsselgenerierung eine besondere Bedeutung zu. Im Fall des ElGamal-Algorithmus erfolgt die Schlüsselerzeugung folgendermaßen:
Es wird eine große Primzahl p gewählt, wobei (p-1)/2 aus Sicherheitsgründen prim sein sollte. Dann werden noch zwei Zahlen g und x gewählt. Anschließend wird mit den Zahlen y bestimmt ($y \equiv g^x \bmod p$) . Die Zahlen p, g und y stellen den öffentlichen Teil des Schlüssel dar, x ist der geheime Teil des Schlüssels. g, x und y müssen dabei natürliche Zahlen zwischen 1 und p-1 sein.
Um nun einen Klartext m zu verschlüsseln, muss der Sender eine beliebige große Zahl k auswählen, bei der die Bedingung gilt, k muss p-1 relativ prim und kleiner p sein. Ist die Bitfolge des Klartextes m größer als p, muss der Klartext in Blöcke kleiner p zerlegt werden. Danach berechnet der Sender den Wert a und den Chiffretext c und sendet beides zum Empfänger ($a \equiv g^k \bmod p,\ \mathrm{c} \equiv \mathrm{y}^{\mathrm{k}} * m \bmod p$). Die Zahl k wird danach nicht mehr gebraucht, muss aber geheim gehalten werden, weil damit die Nachricht herauszufinden ist.
Zum Entschlüsseln nimmt der Empfänger seinen geheimen Schlüssel x und berechnet den Klartext ($m = c * a^{p-1-x} \bmod p$).

4.3.3. DSA / DSS

Der DSS (Digital Signature Standard) wurde vom NIST in Zusammenarbeit mit der NSA ausgewählt, um einen elektronischen Unterschriftenstandard für die US-Behörden zu erhalten. Der im DSS genutzte Algorithmus ist der DSA (Digital Signature Algorithm). Die Sicherheit des DSA beruht auf der Schwierigkeit, diskrete Logarithmen zu berechnen. Dieser Algorithmus kann nur für Vorgänge mit digitalen Signaturen (und nicht für die Datenverschlüsselung) verwendet werden. Die Schlüssellänge beträgt 512 bis 1024 bit. Die Echtheit des Dokumentes wird dabei durch die Einweg-Hashfunktion SHA-1 überprüft und die Signatur bestätigt. Die Parameter p (Primzahl), q und g sind öffentlich, dabei gelten für die Parameter die Bedingung:

- p muss eine Bitlänge zwischen 512 und 1024 Bit haben
- q muss eine Bitlänge von mindestens 160 Bit haben

Die Generierung einer digitalen Unterschrift mit DSA erfolgt, indem der Sender den Hashwert (H_m) über das zu verschickende Dokument bildet und eine Zufallszahl k generiert, die kleiner q ist. Mit dem Zufallswert k wird der Wert $r=(g^k \bmod p) \bmod q$ berechnet. Mit diesen Werten und dem privaten Schlüssel x wird nun $s=(k^{-1}(H_{(m)}+x*r)) \bmod q$ berechnet. Die Werte s und r bilden zusammen die digitale Signatur und werden dem eigentlichem Dokument angehängt. Der Empfänger verifiziert die Signatur, indem er verschiedene Werte

- $w=s^{-1} \bmod q$
- $u1=(H_{(m)}*w) \bmod q$
- $u2=(r*w) \bmod q$
- $v=((g^{u1} * y^{u2}) \bmod p) \bmod q$

berechnet, wozu er den öffentlichen Schlüssel y des Senders benötigt.
Danach werden die Werte v und r verglichen. Wenn v = r ist, wurde das Dokument nicht verändert.

Für die Parameter p gelten in den nächsten Jahren Zusatzbedingungen. Es wird verlangt, dass für den Zeitraum bis Ende 2005 die Bitlänge des Parameters p mindestens 2048 betragen soll. Für den Zeitraum bis Ende 2004 reicht eine Minimallänge des Parameters von 1024 Bit aus.
Die folgende Ausführung veranschaulicht die Ausführung:

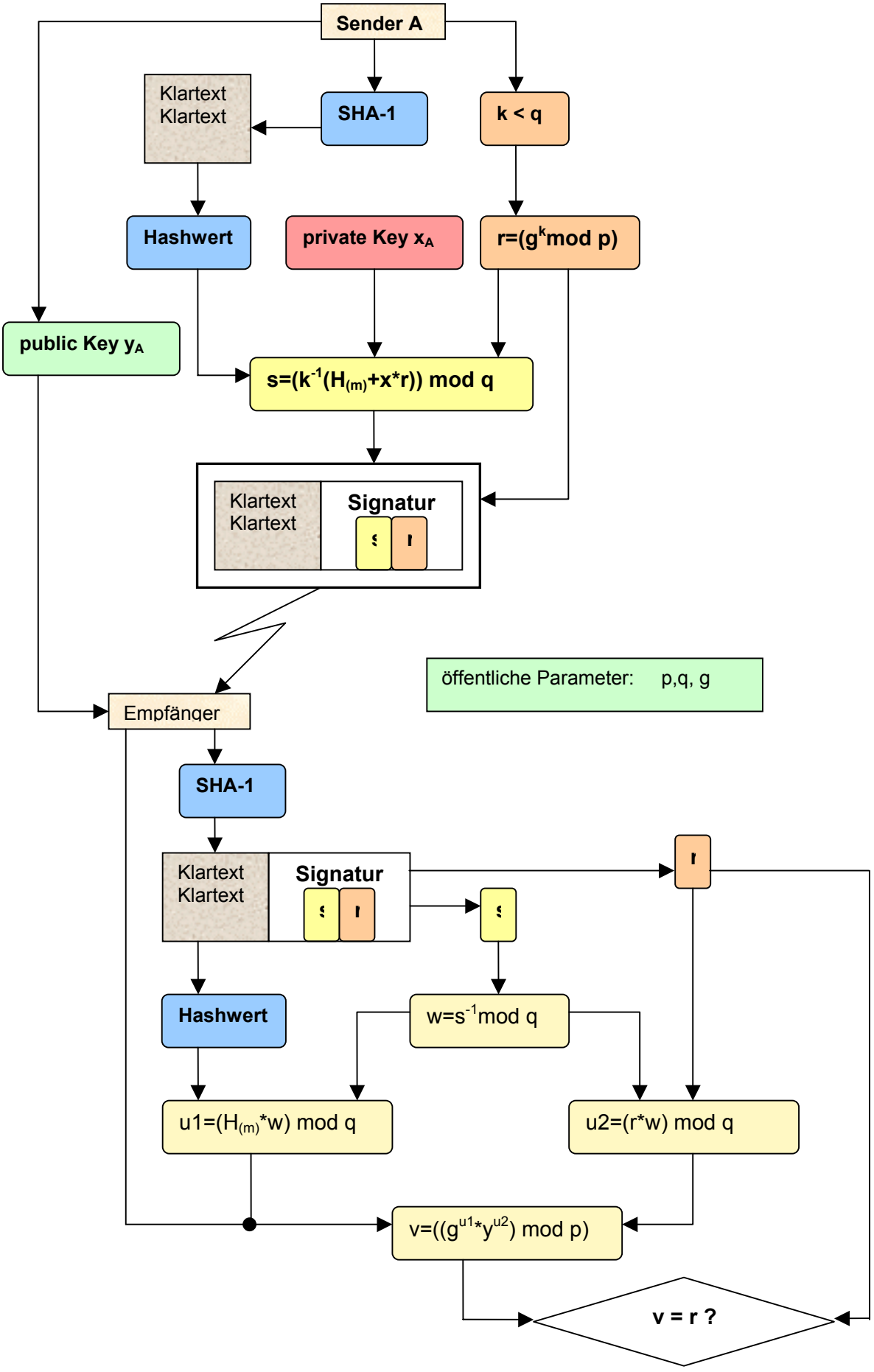

Sender A
Klartext
Klartext
SHA-1
k < q
Hashwert
private Key x_A
$r=(g^k \bmod p)$
public Key y_A
$s=(k^{-1}(H_{(m)}+x*r)) \bmod q$
Klartext
Klartext
Signatur
öffentliche Parameter: p,q, g
Empfänger
SHA-1
Klartext
Klartext
Signatur
Hashwert
$w=s^{-1} \bmod q$
$u1=(H_{(m)}*w) \bmod q$
$u2=(r*w) \bmod q$
$v=((g^{u1}*y^{u2}) \bmod p)$
v = r ?

Abbildung 4.13 : Generierung einer digitalen Unterschrift mit DSA

4.3.4. RSA-Algorithmus

Der RSA-Algorithmus von Rivest, Shamir und Adleman wurde im Jahr 1977 entwickelt. Der RSA-Algorithmus war bis zum 20. September 2000 in den USA patentiert, ansonsten aber frei benutzbar. Das Verfahren beruht auf der Faktorisierung von sehr großen Zahlen. Zurzeit besitzen die Zahlen 300 Dezimalstellen und mehr.

Wie auch bei anderen Algorithmen ist die Schlüsselerzeugung ein wesentlicher Punkt bei dem RSA-Algorithmus. Mit zwei zufälligen, großen Primzahlen p und q wird zuerst eine Zahl n berechnet, n = p*q. n wird auch häufig als Modulzahl bezeichnet. Übliche Längen für die Modulzahl sind 512, 1024 und 2048 Bit, wobei 512 Bit wegen der kurzen Länge nicht mehr eingesetzt werden sollte. Danach wird die Funktion r = (p-1)*(q-1) berechnet, diese gibt die Anzahl der teilerfremden Zahlen an. Für eine weitere Zufallszahl e, die nicht hoch zu sein braucht (um eine schnellere Verschlüsselung zu gewährleisten), muss die Bedingung ggT(r,e) = 1 und e< r gelten. Aus dem vorherigen Berechnungen lässt sich die geheime Zahl d berechnen. Die Zahl d muss dabei die Bedingungen e*d mod r = 1 und d < r erfüllen. Nun bildet (e,n) den öffentlichen Schlüssel und (d,n) den privaten Schlüssel. Zum Verschlüsseln wird das Klartextdokument M in Blöcke < n zerlegt und mit der Funktion $C = M^e \bmod n$ verschlüsselt.
Die Entschlüsselung des Chiffretextes C erfolgt mit der Funktion $M = C^d \bmod n$. In den zwei folgenden Abbildungen wird das Verfahren nochmals anhand eines einfachen Beispiels[34] aufzeigt, die gewählten Zahlen sind für die Realität viel zu klein.

[34] Die Zahlen des Beispiels sind aus Gispert W. Selke: Kryptographie, O'Reilly 2000, S. 67 entnommen.

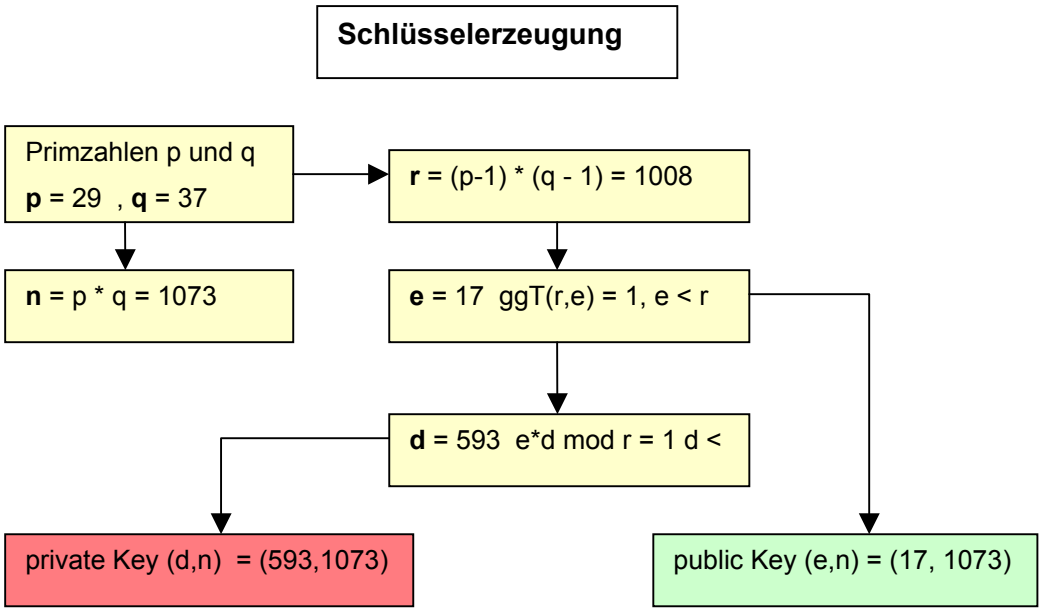


Abbildung 4.14 : Schlüsselerzeugung RSA

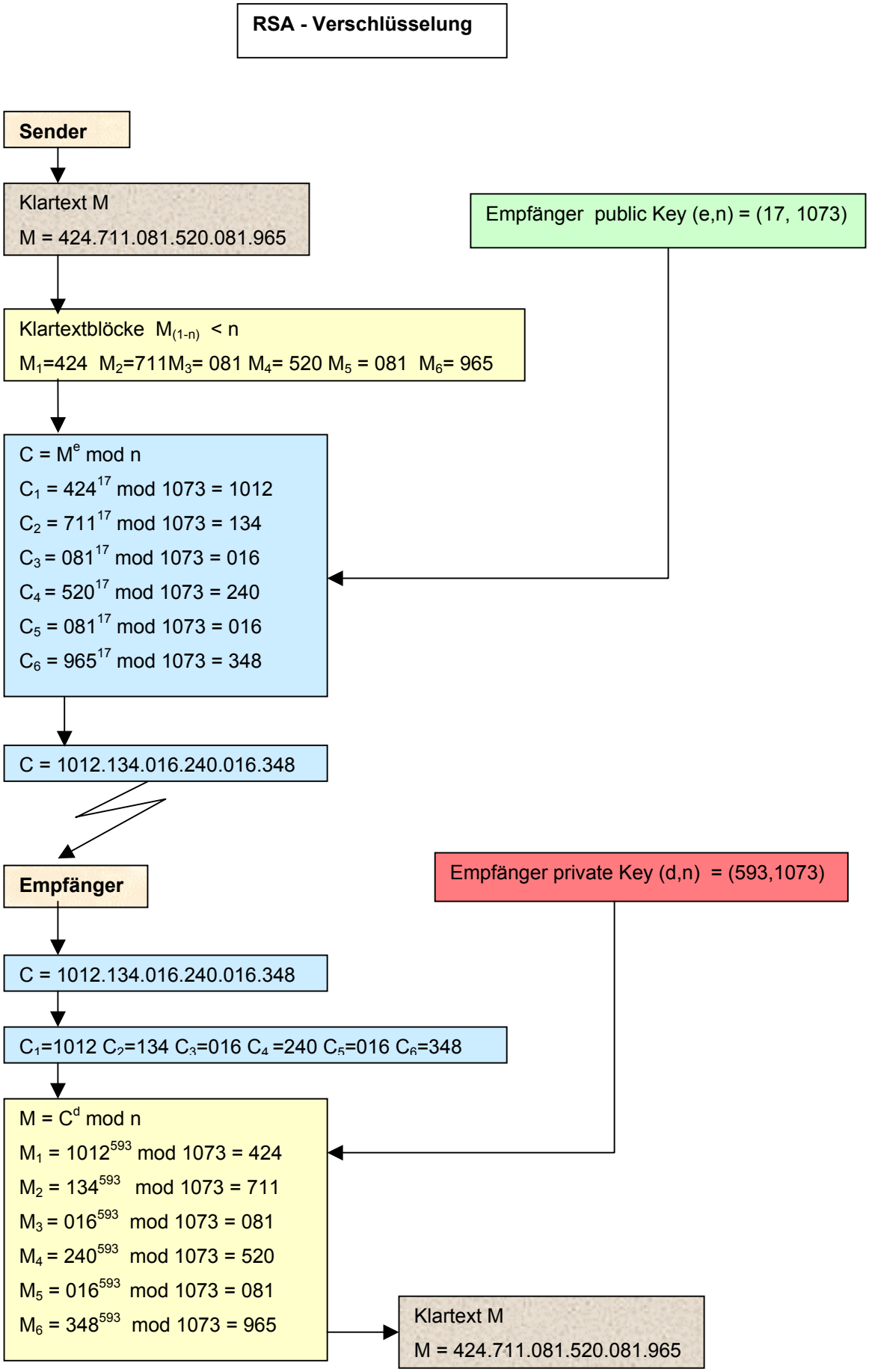


Abbildung 4.15 : RSA Verschlüsselung

Die RSA Verschlüsselung wird in dem bereits erwähnten Softwarepaket PGP (Pretty Good Privacy) verwendet.
Das Modul JCool und der JCoolClassLoader verwenden ebenfalls den RSA-Algorithmus. Dabei werden alle Anfragen des Moduls JCool und des JCoolClassLoaders an den Server mit dem Public-Key des Betreibers verschlüsselt, damit ein sicherer Zugriff auf den Server erfolgen kann. Zur Verschlüsselung wurde der RSA-Algorithmus ausgewählt, weil er zur Zeit als am sichersten gilt, und weil die im Gegensatz zu symmetrischen Verschlüsselungstechniken (z.B. DES-Algorithmus oder IDEA-Algorithmus) schlechte Performance dadurch kompensiert wird, dass immer nur einzelne Wörter verschlüsselt werden müssen. Auch die durch den RSA-Algorithmus gegebene Möglichkeit der Authentifizierung war ausschlaggebend für die Wahl dieses Public-Key Verfahrens.

4.3.5. ECC (Elliptic Curve Cryptography)

Eine hervorragende Alternative zum RSA-Algorithmus stellt der
ECC (Elliptic Curve Cryptography) dar, der an dieser Stelle kurz erwähnt wird. ECC ist ein Public Key Kryptographie-Verfahren auf Grundlage der elliptischen Kurve. Die Verwendung von elliptischen Kurven in der Kryptographie wurde erstmals 1985 von Neal Koblitz und Victor Miller unabhängig voneinander vorgeschlagen. Das dem ECC bzw. den elliptischen Kurven zugrundeliegende mathematische Problem ist, den diskreten Logarithmus in der Punktegruppe einer elliptischen Kurve zu finden. Bei den elliptischen Kurven handelt es sich um eine Menge von Punkten (x, y) in der Ebene, deren Koordinaten eine bestimmte Gleichung erfüllen.

Eine elliptische Kurve lässt sich durch die Gleichung

$$E : y^2 = x^3 + ax + b \quad \text{beschreiben.}$$

Punkte, die diese Gleichung erfüllen, können z.B.

$$P = (x_1, y_1),\ Q = (x_2, y_2) \ \text{sein.}$$

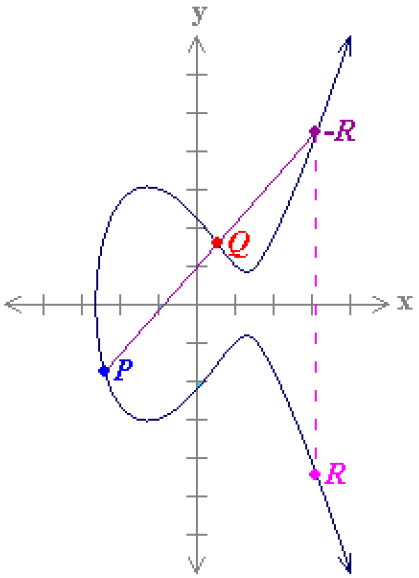


Abbildung 4.16 : Additionsgesetz für elliptische Kurven[35]

Die eigentliche Addition der Punkte P und Q geschieht, indem man eine Verbindungsgerade durch die beiden Punkte zieht. Dabei erhält man einen eindeutigen Schnittpunkt -R der Gerade mit der Kurve. Um die Summe der Punkte P und Q zu erhalten, wird der Schnittpunkt R an der x-Achse gespiegelt (R = P+Q). Die Punktegruppe einer elliptischen Kurve ist eine additive Gruppe (2*P=P+P → k*P). War bei dem diskreten Logarithmusproblem[36] die Bestimmung von x maßgeblich, so ist das diskrete Logarithmusproblem bei elliptischen Kurven, aus zwei gegebenen Punkten Q, P ein k zu bestimmen, das die Bedingung Q=k*P erfüllt.
Ein wesentlicher Vorteil von ECC ist, dass ECC eine sehr hohe Performance im Gegensatz zu anderen Public-Key-Verfahren hat. Dies liegt unter anderem an den kurzen Schlüsseln, die benötigt werden. So entspricht ein 160 Bit langer ECC-Schlüssel von der Sicherheit her einem 1.024 Bit langen RSA-Schlüssel.
Das Einsatzgebiet von ECC ist in Bereichen, in denen besonders Speicher- und Performance-Effizienz von Bedeutung ist. Die Einsatzgebiete sind zum Beispiel bei Chipkarten oder im Mobilfunkbereich. Aber auch das Bundesamt für Sicher-

---

[35] http://www.cryptovision.com/deutsch/service/ppt/ecc.ppt S. 13; cryptovison gmbh 2000 (12.08.2002)
[36] Vergleiche Fußnote 32 S. 39

heit in der Informationstechnik (BSI) setzt ECC zur Zeit schon zur verschlüsselten Datenübertragung zwischen Behörden ein.

4.4. Hybride Verschlüsselung

Ein elementarer Nachteil der asymmetrischen Verschlüsselung ist der hohe Rechenaufwand und somit die schlechte Performance, die sich insbesondere bei der Verschlüsselung von großen Dokumenten negativ bemerkbar macht.
Hingegen ist bei der symmetrischen Verschlüsselung der unsichere Schlüsselaustausch ein großer Nachteil.

Die hybride Verschlüsselung vereinigt nun die Vorteile beider Verfahren. Das sind bei der asymmetrischen Verschlüsselung die sichere Schlüsselübertragung und bei der symmetrischen Verschlüsselung die sehr gute Performance. Es wird deshalb eine Kombination aus symmetrischer und asymmetrischer Verschlüsselung genutzt.

Die Hybride Verschlüsselung läuft, wie in dem folgenden Absatz erläutert ab:
Der Sender A erzeugt einen möglichst zufälligen symmetrischen Schlüssel, den so genannten Session Key, und chiffriert mit diesem seinen Klartext. Den Session Key chiffriert der Sender danach mit dem Public-Key (asymmetrisch) des Empfängers B.
Den Chiffretext schickt der Sender zusammen mit dem chiffrierten Session Key anschließend an den Empfänger B.
Da der Empfänger B den Session Key nicht kennt, muss er zunächst den chiffrierten Session Key (symmetrisch) mit seinem Private-Key (asymmetrisch) entschlüsseln. Den so gewonnenen Session Key kann er nun dazu verwenden, den Chiffretext wieder zu dechiffrieren.

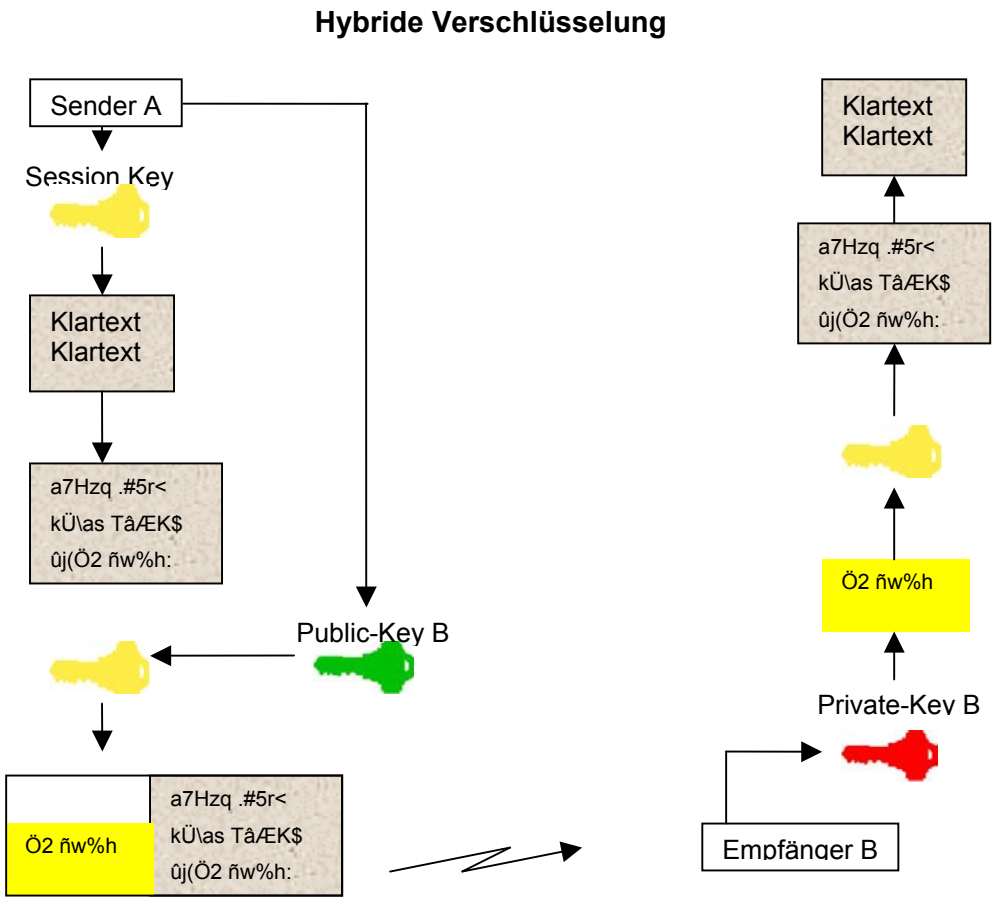


Abbildung 4.17 : Ablauf der hybriden Verschlüsselung

Da nur der symmetrische Schlüssel mit dem asymmetrischen Verfahren verschlüsselt wird, bleibt der Rechenaufwand relativ gering.

4.5. Hashfunktionen

In verschiedenen Kryptosystemen und speziell zur Erstellung der digitalen Signatur spielen Hashfunktion eine wesentliche Rolle.

Eine Hashfunktion ist eine mathematische Funktion H(M), die Eingabedaten M beliebiger Länge auf Ausgabedaten fixer Länge (z. B. 128, 160, 192 oder 256 Bit) abbildet. Der Ausgabewert h wird Hashwert genannt.
In der Kryptographie werden Einweg-Hashfunktionen H(M) verwendet, die aus einem beliebigen Klartext M nach einem vorbestimmten Verfahren eine Prüfzif-

fer[37] h generieren. Die Funktion verwandelt einen Klartext so in eine entsprechende Prüfziffer um, dass auch die kleinste Veränderung des ursprünglichen Texts zu einer vollkommen unterschiedlichen Prüfziffer führt.

Dazu ein kleines Beispiel[38]:
Vor dem Einfügen dieser Zeilen hatte dieses Dokument diese Prüfsumme:
h = d58add7194d08bb6445eef3d4e8f802aafac45ad
und nach dem Einfügen folgende Prüfsumme:
h = c7d1c71a60aadaeaa3b9e5afd0d261a17d39f93a
Wie zu sehen ist, hat sich die Prüfsumme vollständig geändert.

Zu den Forderungen an Einweg-Hashfunktionen gehört es, dass aus der einmal erzeugten Prüfziffer der ursprüngliche Text nicht wieder hergestellt werden kann. Außerdem muss die Einweg-Hashfunktion kollisionsfrei sein, d.h. es darf keine zwei Nachrichten geben, die den selben Hashwert liefern. Daraus folgt, dass die Wahrscheinlichkeit zu einer gegebenen Nachricht M eine andere Nachricht M' zu berechnen mit H(M) = H(M') vernachlässigbar gering sein muss.

Zu den am meisten bekannten Algorithmen gehören folgende kurz erläuterte Hash-Funktionen:

SHA-1 (Secure Hash Algorithm One)
Der SHA-1 ist das zur Zeit wohl beste Verfahren. SHA-1 wurde im Jahr 1993 vom amerikanischen National Institute of Standards and Technology (NIST) als U.S. Standard veröffentlich. Der SHA-1 erzeugt 160 Bit Prüfsummen und verarbeitet Nachrichtenblöcke mit 512 Bit Länge. Wenn eine Nachricht kürzer ist, wird sie nach einer festgelegten Paddingvorschrift auf 512 Bit aufgefüllt. Die Wahrscheinlichkeit, bei dem SHA-1-Algorithmus zwei Nachrichten mit der selben Prüfsummen zu finden, liegt bei 1 zu 2^{80}, und die Wahrscheinlichkeit, eine Nachricht zu einer bestimmten Prüfsumme zu finden, liegt sogar bei 1 zu 2^{160}. Der SHA-1 Algorithmus wird unter anderem in dem DSA (Digital Signature Algorithm) angewendet, der im DSS (Digital Signature Standard) spezifiziert wird.
Im JCoolClassLoader wird die Prüfsumme mit Hilfe der Message-Digest-Klasse von Java und dem SHA-1-Algorithmus berechnet. Dabei werden die Hashwerte der Java-Klassen auf dem Clientrechner mit denen der Java-Klassen auf dem Server verglichen. Sollten die Hashwerte unterschiedlich sein, lädt der JCool-

[37] Prüfziffer = Hashwert = Komprimat
[38] Die Prüfsumme wurde mit Hilfe der Message-Digest-Klasse von Java und dem SHA-1-Algorithmus berechnet.

ClassLoader die Java-Klassen vom Server. Der SHA-1 Algorithmus wurde gewählt, weil die Prüfsumme 160 Bit lang ist und die Wahrscheinlichkeit einer Kollision somit vernachlässigbar gering ist.

MD 5 (Message Digest 5)

Der MD 5 Algorithmus wurde 1991 von Ron Rivest und S. Dusse entwickelt.
Der MD 5 erzeugt aus einem beliebig langen Text eine Prüfsumme von 128 Bit Länge. Der Text wird in Blöcke zu 512 Bit zerlegt und bei Bedarf mit 0 Bits aufgefüllt.
Der MD 5 Algorithmus wurde sehr häufig verwendet. In den letzten Jahren wurden aber Sicherheitsmängel nachgewiesen, da Sicherheitslücken im Algorithmus gefunden wurden. So ließ sich zeigen, dass eine Kollision innerhalb einiger Tage gefunden werden kann.

RIPEMD (RIPE-Message Digest)

RIPEMD wurde im Rahmen des EU-Projektes RIPE (RACE Integrity Primitives Evaluation) von den Kryptographen Antoon Bosselaers, Bart Preneel und Hans Dobbertin entwickelt. Der Hashwert ist entweder 128 Bit (RIPEMD-128) oder 160 Bit (RIPEMD-160) lang.
Die ursprüngliche 128 Bit Variante stellte sich als unsicher heraus. Dies führte zur Entwicklung des wesentlich stärkeren RIPEMD-160 mit 160 Bit. Verwendet wird RIPEMD- 160 in vielen kryptographischen Anwendungen unter anderem auch in PGP.

Die beiden Hashfunktionen SHA-1 und RIPEMD-160 werden mindestens bis Ende 2005 für sicher gehalten und sind für die Anwendung bei digitalen Signaturen nach dem Signaturgesetz geeignet.

4.6. Digitale Signatur

Eine digitale Signatur ist ein Verfahren, dass mit Hilfe von Hashfunktionen und Public-Key-Verfahren die Unveränderheit (Integrität) der Daten und die Identität der Person des Versenders (Authentizität) gewährleistet. Eine digitale Signatur bezeugt aber nicht die Vertraulichkeit der signierten Dokumente. Denn das eigentliche Dokument wird nicht verschlüsselt und ist somit für mögliche Angreifer lesbar. Die Funktionsweise der digitalen Signatur stellt sich folgendermaßen dar:

Als erstes wird mit einer Hashfunktion der Hashwert (Prüfsumme) über das zu verschickende Dokument gebildet. Anschließend wird dieser Hashwert mit dem Private-Key eines asymmetrischen Verschlüsselungsverfahrens verschlüsselt. Dieser chiffrierte Hashwert ist die eigentliche digitale Signatur. Die digitale Signatur wird danach zusammen mit dem Dokument versendet. Um die Unversehrtheit des Dokumentes zu überprüfen, bildet der Empfänger mit der gleichen Hashfunktion den Hashwert über das empfangene Dokument und dechiffriert mit dem Public-Key des Senders die digitale Signatur. Sind die beiden Hashwerte identisch, ist sichergestellt, dass das Dokument unverändert ist und vom Sender stammen muss.

Die folgende Grafik veranschaulicht die Ausführung:

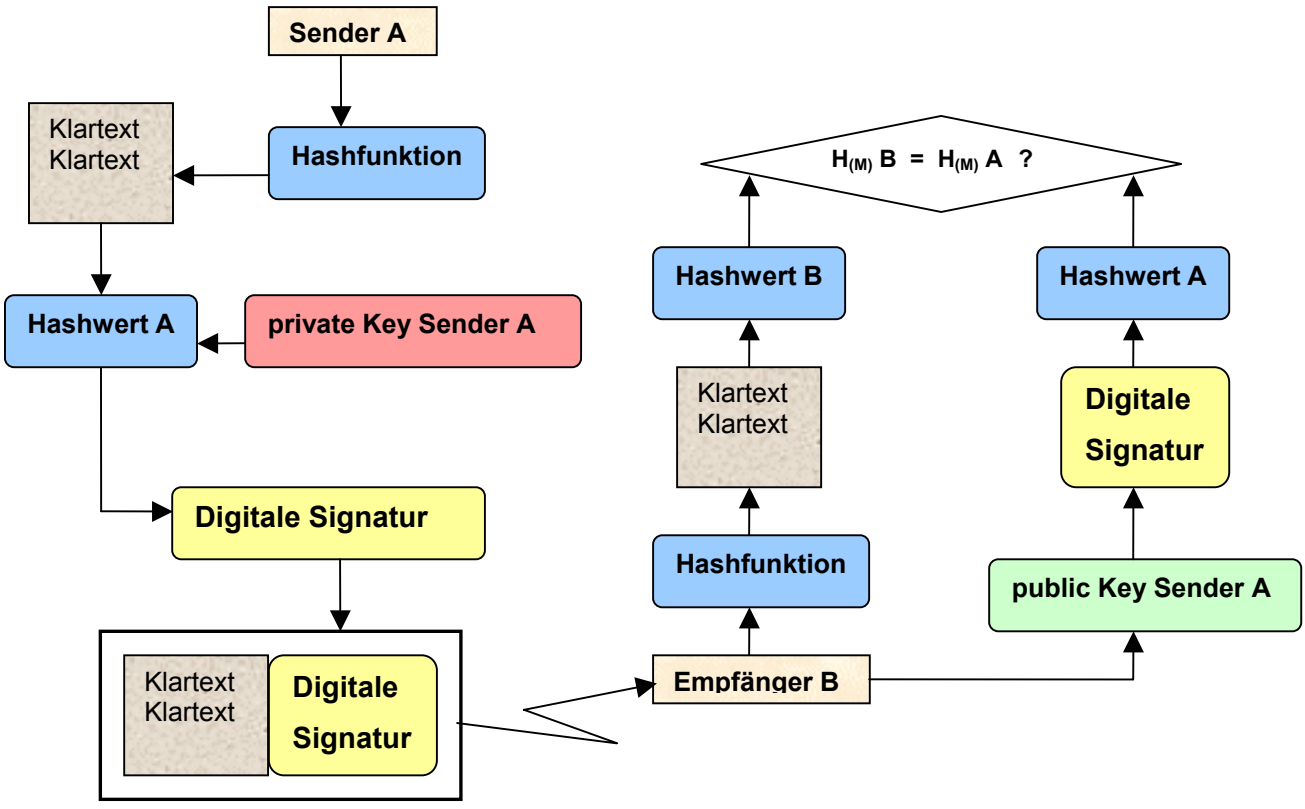


Abbildung 4.18 : Funktionsweise der digitalen Signatur

Wie bereits im Kapitel über asymmetrische Verfahren erwähnt, ist ein Schwachpunkt der Public-Key Verfahren, dass die Authentizität des Besitzers des öffentlichen Schlüssels nicht sicher ist. Damit die Verwendung digitaler Signaturen rechtlich anerkannt werden kann, müssen die jeweiligen Schlüsselpaare (Private- und Public-Key) durch Zertifizierungsstellen den Personen fest zugeordnet werden und weitere Bedingungen des Signaturgesetz (SigG)[39] und der Signaturverordnung (SigV) erfüllen.

Der JCoolClassLoader benutzt einmal während einer Sitzung das Verfahren der digitalen Signatur. Der JCoolClassLoader sendet eine Aufforderung an den Server, das Properties-File mit den Hashwerten der einzelnen Klassen zu schicken. Danach bildet das Server-Programm mit Hilfe der Message-Digest-Klasse von Java und dem SHA-1-Algorithmus den Hashwert über das jCoolServerDigest.properties-File und verschlüsselt diesen chiffrierten Hashwert mit Hilfe des RSA-Algorithmus und dem Private-Key des Anbieters. Das jCoolServerDigest.properties-File und die digitale Signatur werden anschließend an den JCoolClassLoader des Clients gesandt. Dort wird die digitale Signatur mit dem Public-Key des Anbieters entschlüsselt und der Hashwert des empfangenen jCoolServerDigest.properties-File gebildet. Beide Hashwerte werden dann noch verglichen. Ist der Hashwert gleich, so läuft das Programm ungehindert weiter. Sind es aber zwei unterschiedliche Hashwerte, erhält der Benutzer eine Warnmeldung und das Programm bricht ab.

Nachdem die Grundlagen der Kryptographie unter Berücksichtigung einzelner Verschlüsselungstechniken erläutert wurden, die für die Umsetzung eines selbstdefinierten ClassLoaders eingesetzt werden können, wird im folgenden Kapitel die Möglichkeit aufgezeigt, in Java Socketverbindungen herzustellen, um eine Client-Server Anwendung zu implementieren.

---

[39] "Das Gesetz zur digitalen Signatur (Signaturgesetz - SigG) wurde als Artikel 3 des Gesetzes zur Regelung der Rahmenbedingungen für Informations- und Kommunikationsdienste (IuKDG) am 13. Juni 1997 vom deutschen Bundestag beschlossen und trat am 1. August 1997 in Kraft." C. Eckert: IT-Sicherheit, München: Oldenburger Wissenschaftsverlag GmbH, S. 252

5. Sockets in Java

In dem dieser Arbeit zugrundeliegenden Programm, dem JCoolClassLoader, werden die zu aktualisierenden Klassen mit einer Socketverbindung von dem Server über das Netzwerk geladen. Die technischen Grundlagen dazu werden in diesem Kapitel behandelt.

5.1. Grundlagen der Programmierung von Netzwerken

Ganz allgemein versteht man unter einem Netzwerk die Verbindung zwischen zwei oder unbegrenzt vielen Computern, um Ressourcen gemeinsam nutzen zu können und um Daten miteinander auszutauschen. Sowohl zwei Heimrechner, die miteinander verbunden sind, als auch ein firmeneigenes Intranet oder das Internet bilden ein Netzwerk.

5.1.1. Netzwerk-Protokolle

Damit eine einwandfreie Kommunikation in einem Netzwerk stattfinden kann, wurden Protokolle eingeführt, die eine kontrollierte und eindeutige Verbindung und Datenaustausch gewährleisten. Das wohl bekannteste Modell der Datenkommunikation ist das OSI-Referenzmodell[40], das Architekturmodell wurde von der International Standards Organisation (ISO) 1978 entwickelt, und wurde 1983 von der ISO zum Standard erhoben. Daher wird es auch das ISO/OSI-Referenzmodell genannt.

[40] Open Systems Interconnect Reference Model(OSI)

Das OSI-Referenzmodell besteht aus sieben Schichten, auch Layer genannt, bei der jede Schicht eine Funktion repräsentiert, die ausgeführt wird, wenn Daten über das Netzwerk übertragen werden.
In der nebenstehenden Abbildung[41] sind die einzelnen Schichten des ISO/OSI-Referenzmodells mit einer kurzen Beschreibung abgebildet:

Abbildung 5.1 : Das ISO/OSI - Referenzmodell

Das Java-Package *java.net* bietet die Möglichkeit, über ein Netzwerk mit anderen Computern zu kommunizieren. Dabei greift Java auf zwei verschiedene Internet-Protokolle zurück, dem TCP/IP-Protokoll[42] und dem UDP/IP-Protokoll[43]. Der grundlegende Unterschied zwischen beiden Protokollen liegt darin, dass es sich bei dem TCP/IP-Protokoll um eine sichere, streamorientierte bidirektionale Verbindung handelt, und bei dem UDP/IP-Protokoll um eine unsichere, paketorientierte Verbindung, die aber Mitteilungen an mehrere Empfänger erlaubt.
Da der JCoolClassLoader eine sichere Verbindung benötigt, werden in dieser Arbeit nur das TCP/IP-Protokoll und die wesentlichen Klassen des Package *java.net* betrachtet.
Das TCP/IP-Protokoll stellt eine vereinfachte Form des ISO/OSI-Referenzmodells dar. Durch Zusammenfassung einzelner Schichten verfügt das TCP/IP-Protokoll

---

[41] Craig Hunt: TCP/IP Netzwerk-Administration, Köln: O'Reilly 1998, S. 6

[42] Der Name setzt sich aus den Protokollen 'Transmission Control Protocol' (TCP) und dem 'Internet Protocol' (IP) zusammen.

[43] Der Name setzt sich aus den Protokollen 'User Datagramm Protocol' (UDP) und dem 'Internet Protocol' (IP) zusammen.

aber nur über vier. Dabei handelt es sich im einzelnen um die Netzzugangsschicht, die Internetschicht, die Transportschicht und die Anwendungsschicht.

Die nebenstehende Abbildung[44] zeigt diese Schichten des TCP/IP Protokolls mit einer kurzen Beschreibung:

Abbildung 5.2 : Das TCP/IP Protokoll

Die Netzzugangsschicht, auch physikalische Schicht genannt, wird durch die Netzwerkhardware (z.B. Netzwerkkarte) und die Netzwerkmedien (z.B. Ethernet[45]) dargestellt.

Die Internetschicht ist fast ausschließlich dem Internet Protocol (IP) und dessen Kontrollprotokollen z.B. dem ICMP (Internet Control Message Protocol) vorbehalten. Das Internet Protokoll wird als Herzstück von TCP/IP bezeichnet. IP verfügt über die grundlegenden Dienste zur Auslieferung von Paketen. Alle ein- und ausgehenden Daten laufen dabei durch IP.

Die beiden wichtigsten Protokolle der Transportschicht, auch Host-zu-Host-Transportschicht genannt, sind TCP-Protokoll und das UDP-Protokoll.

Die Anwendungsschicht repräsentiert alle Protokolle, die die Transportschicht zur Auslieferung von Daten verwendet. Zu den Anwendungsprotokollen gehört z.B. telnet[46], FTP[47], SMTP[48], HTTP[49] und DNS[50].

---

[44] Craig Hunt: TCP/IP Netzwerk-Administration, Köln: O'Reilly 1998, S. 9

[45] Unter dem Ethernet versteht man eine weit verbreitete, herstellerneutrale Technologie mit der im Lokal Area Network (LAN) Daten übertragen werden können.

[46] Das Network Terminal Protocol bietet den Benutzern die Möglichkeit, sich über das Netzwerk auf entfernten Rechnern einzuloggen.

[47] Das File Transport Protocol wird zum Filetransfer verwendet

[48] Das Simple Mail Transfer Protocol dient dem Mailversand

[49] Das Hypertext Transfer Protocol wird zur Übertragung von Webseiten verwendet.

In Java kann die Kommunikation über das Netzwerk durch verschiedene Abstraktionsebenen realisiert werden. Es wird grundsätzlich zwischen den Abstraktionsebenen von Client-Serveranwendungen mittels Sockets, direkter Austausch bezüglich des HTTP-Protokolls und verteilten Anwendung mittels RMI (Remote Method Invocation) unterschieden.
Im folgenden Kapitel wird auf die Client-Serveranwendungen mittels Sockets eingegangen.

5.2. Sockets

Als Sockets bezeichnet man eine Schnittstelle zur Kommunikation über das Netz (Internet oder Ethernet), die eine kontinuierliche Verbindung ermöglicht. Der Client und der Server sind die Endpunkte der Verbindung, die solange aufrechterhalten wird, bis der Socket geschlossen wird. Der Server wartet dabei, an einem konkreten Port[51], auf die Verbindungsanfrage eines Clients. Stellt der Client nun eine Anfrage an den Server, bauen beide eine eigene Verbindung auf, über die sie kommunizieren können. Der Client erhält während dieses Prozesses eine noch nicht benutzte Portnummer, welche dieser in seinen Socket einbindet, und solange behält, bis die Verbindung geschlossen wird. Während der bestehenden Verbindung können dann Daten gelesen oder geschrieben werden.
Sockets wurden Anfang der achtziger Jahre für die Programmiersprache C entwickelt, als die Universität von Berkeley (Kalifornien) den Auftrag erhielt, unter UNIX das TCP/IP-Protokoll zu implementieren.
Die Java-Plattform unterstützt dabei sowohl TCP-Protokolle als auch UDP-Protokolle, die von Sockets verwendet werden.
Die folgende Abbildung[52] zeigt die Beziehung zwischen Applikationen, Sockets und Ports:

---

[50] Das Domain Name Service erleichtern die Handhabung der IP-Adressen, sie weisen IP-Adressen sprechende Namen zu.
[51] Die Tür zu einem anderem Rechner wird Port genannt. Portnummern sind 16 Bits groß, dadurch kann ein Server bis zu 65535 verschiedene TCP-Verbindungen aufbauen.
[52] Kenneth L. Calvert, Michael J. Donahoo: TCP/IP Sockets in Java, London u.a.: Academic Press 2002, S. 7

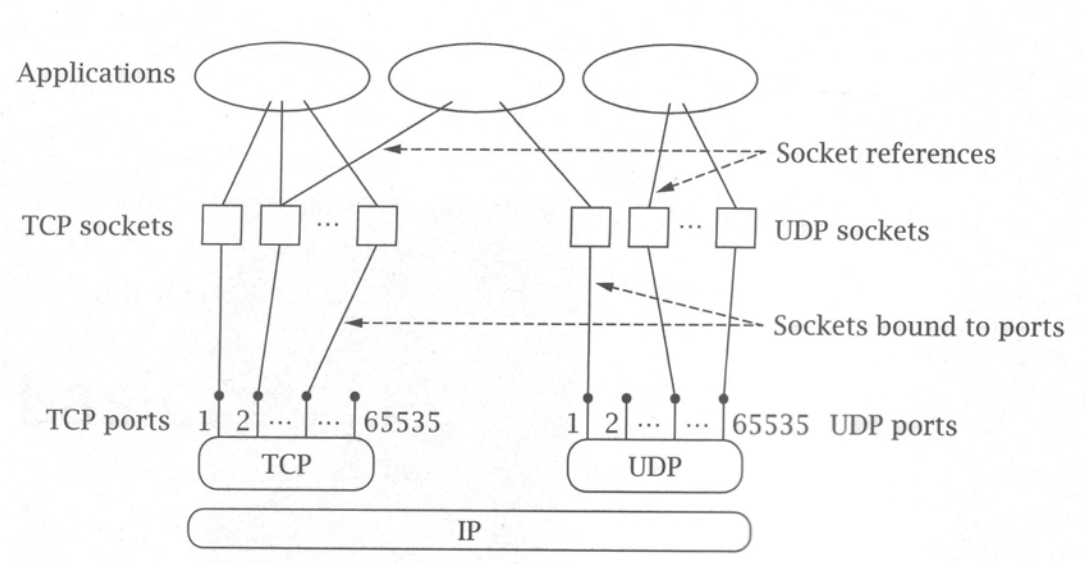


Abbildung 5.3 : Sockets

Das TCP-Protokoll richtet eine sichere Verbindung zwischen zwei Computern ein, wohingegen das UDP-Protokoll als eine unsichere Verbindung gilt. Der Nachteil vom UDP-Protokoll ist, dass die Pakete in zufälliger Reihenfolge eintreffen, und es nicht sicher ist, ob die Pakete überhaupt eintreffen. Deshalb eignen sich Sockets über UDP-Protokolle nur für Anwendungen, die fehlende Anwendungen tolerieren, so z.B. Audio- oder Video-Streams. Der Vorteil vom UDP-Protokoll ist, dass es sehr schnell ist, und dass man damit Mitteilungen an mehrere Empfänger versenden kann. Für den JCoolClassLoader kann die Socketverbindung mittels UDP-Protokoll, wie bereits erwähnt, nicht in Frage kommen, weil die sichere Verbindung eine der Grundlagen des JCoolClassLoaders ist. Der JCoolClassLoaders nutzt die Sockets über das TCP-Protokoll, um eine sichere streambasierte Kommunikation zwischen dem Client-Rechner und dem Server-Rechner herzustellen.

5.2.1. Grundlegende Operationen des Client-Socket

Damit Daten von einer Stelle zur anderen geschickt werden können, muss zunächst eine Verbindung zum Server bestehen. Dazu stellt die Java-API mit der Klasse *Socket* im Package *java.net* verschiedene Konstruktoren zur Verfügung.
Der JCoolClassLoader nutzt z.B. den Konstruktor :

- public Socket(String host, int port) throws IOException, throws UnkownHostException

Ein weiterer häufig genutzter Konstruktor ist:

- public Socket(InetAdress address, int port) throws IOException

Zum Aufbau der Verbindung muss an erster Stelle der Hostname angegeben werden. Das kann sowohl als Objekt des Types InetAdresse[53], als auch über den Hostnamen geschehen. Der String des Hostnamens kann sowohl die IP-Adresse, als auch die IP-Nummer enthalten. Im JCoolClassLoader wird z.B. der Hostname als String mit der IP-Nummer angegeben, die vorher aus einem Properties-File geladen wurde.
An zweiter Stelle wird die Portnummer angegeben, dadurch ist eine weitere Möglichkeit gegeben, die Server untereinander zu unterscheiden.
Kann ein Socket nicht geöffnet werden, so wird eine IOException oder UnkownHostException ausgeworfen.
Nach einem erfolgreichen Aufbau der Verbindung kann nun mit dem Server kommuniziert werden. Das geschieht vom Prinzip her genauso, wie man in Dateien schreibt, bzw. liest. Die Socket-Klasse Streams liefert dafür die beiden Methoden getInputStream() und getOutputStream():

- public InputStream getInputStream() throws IOException
- public OutputStream getInputStream() throws IOException

Im JCoolClassLoader wird zur Kommunikation die bequemere Möglichkeit genutzt, die Streams in Reader und Writer zu überführen. Um die binären Daten zu transportieren, werden z.B. die Streams in DataInputStreams und DataOuputStreams geändert.
Nach der erfolgreichen Kommunikation muss die Verbindung mit der Methode

- void close() throws IOException

geschlossen werden.

---

[53] Die Klasse InetAdresse repräsentiert im Grunde eine Internet-Adresse. Das übergebene InetAdress-Objekt kann wiederverwendet werden, womit eine erneute Adressauflösung eingespart wird.

Damit eine Verbindung aufgebaut werden kann, wird aber noch das Gegenstück, der ServerSocket, benötigt. Die wesentlichen grundlegenden Operationen werden im nächsten Kapitel beschrieben.

5.2.2. Grundlegende Operationen des Server-Socket

Auch für die Verwirklichung eines Servers bietet Java die entsprechenden Routinen. Genau genommen bauen Server-Sockets keine eigene Verbindung auf, sondern horchen an ihrem zugewiesenen Port auf Eingaben und Anfragen. Die Verbindung wird mit der Klasse

- public ServerSocket(int port) throws IOException

realisiert.
Dazu wird ein ServerSocket Objekt erzeugt, dem ein Port als Parameter übergeben wird. Der Server wartet nun an dem angegebenen Port auf Anfragen. Mit der Methode

- public Socket accept() throws IOException

der Klasse ServerSocket kann der Server auf eingehende Anfragen reagieren.
Die Methode accept() ist solange blockiert, bis eine Anfrage kommt. Wenn der Verbindungsaufbau erfolgreich war, liefert accept() ein Socket-Objekt, das die gleichen Möglichkeiten (lesende und schreibende Kommunikation) bietet wie das Client-Socket. Um die Verbindung zu schließen, wird das Socket genauso wie bei der Client-Anwendung mit close() geschlossen.
Dieser vorgestellte Server-Socket beinhaltet aber noch nicht die Möglichkeit, mehrere Clients zu bedienen. Ein Server, der bei häufigen Anfragen von Clients aber ständig besetzt ist, ist jedoch für die kommerzielle Nutzung ineffektiv. Zur Beseitigung dieses Problems muss im Hauptprogramm des Servers eine Schleife um accept() gelegt werden, und ein neuer Thread[54] mit dem Verbindungs-Socket als Argument erzeugt werden.

[54] Mit Threads (Fäden) ist die Möglichkeit gegeben, parallel ablaufende Aktivitäten zur Realisierung der Nebenläufigkeit während der Laufzeit zu implementieren.

Der folgende Codeausschnitt verdeutlicht die Aussage:

```
                ServerSocket s = new ServerSocket(4711);
        // Endlosschleife
          while (true)
                {
                Socket socket = s.accept();
                 (new ConnectClientThread( socket)).start();
          }
```

Die Klasse ConnectClientThread erledigt alle anfallenden Aufgaben, nachdem der Thread gestartet wurde. Der Thread wird beendet, sobald der Client die Verbindung trennt. Das Hauptprogramm muss also nur noch auf anfallende Anfragen warten.

Neben den Klassen der Java API, die in diesem Kapitel gesondert behandelt wurden, werden im nächsten Kapitel weitere, für den JCoolClassLoader relevante Klassen behandelt.

6. Programmrelevante Klassen der Java API

In diesem Kapitel werden die relevanten Klassen, die in dem JCoolClassLoader genutzt werden, vorgestellt. Zu den in diesem Kapitel besprochenen Klassen gehört die abstrakte Klasse *ClassLoader* im Package *java.lang*, die Klasse *Properties* im Package *java.util* und die Klasse *MessageDigest* im Package *java.security*.

6.1. ClassLoader

Die Klasse *ClassLoader* ist im Grunde das Herzstück der Java API, um das dynamische Laden von Klassen und die Funktionalität des JCoolClassLoaders zu ermöglichen.
Die API von Java 2 bietet drei verschiedene ClassLoader, die in einer Vererbungshierarchie zueinander stehen, für verschiedene Zwecke an.
Als oberste Klasse in der Vererbungshierarchie steht der Bootstrap-ClassLoader der JVM (Java Virtual Machine). Da von dieser Klasse aber keine weiteren Klassen abgeleitet werden können, bietet die Java API in dem Package *java.lang* die abstrakte Klasse *ClassLoader* an, von der weitere Klassen abgeleitet werden können. Der zweite ClassLoader der Java API ist die Klasse *SecureClassLoader* aus dem Package *java.security*, die von der abstrakten Klasse *ClassLoader* ab-

geleitet wurde. Von der Klasse *SecureClassLoader* ist die Klasse *URLClassLoader* abgeleitet, die sich im Package java.net befindet.
Die folgende Abbildung verdeutlicht die Vererbungshierarchie:

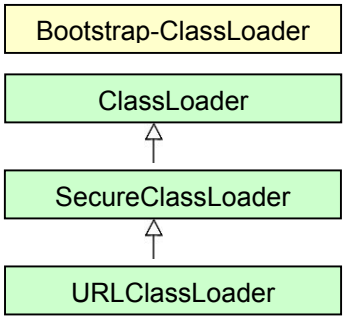


Abbildung 6.1 : Vererbungshierarchie der ClassLoader

Der *URLClassLoader* gibt die Möglichkeit, den expliziten Ort anzugeben, an dem eine Klasse zu finden ist. Diese speziellen Orte werden in Form einer URL-Liste angegeben. Für diesen ClassLoader muss keine Unterklasse abgeleitet werden, nur wenn man z.B. spezielle Verschlüsselungen verwenden möchte, muss dies geschehen. Der *URLClassLoader* kam für die Realisierung des JCoolClass-Loaders nicht in Frage, weil dieser über eine Socketverbindung die Klassen laden sollte.
Durch den *SecureClassLoader* ist die Möglichkeit gegeben, Rechte je nach Herkunftsort der geladenen Klassen zu vergeben. Um diese Klasse zu nutzen, muss eine Unterklasse gebildet werden, weil die Konstruktoren der Klasse *protected* deklariert sind.
Da die Rechte von den Klassen des Moduls JCool, die über die Socketverbindung oder vom File-System während der Ausführung geladen werden, vollkommen gleich sind und auch gleich sein müssen, wurde für den JCoolClassLoader die Möglichkeit genutzt, die abstrakte Klasse *ClassLoader* abzuleiten, in der alle notwendigen Sicherheitsmechanismen hinzugefügt wurden.

Um nun einen eigenen ClassLoader zu schreiben, wird die Methode

- protected Class loadClass(String name, boolean resolve) throws ClassNotFoundException

aus der abstrakten Klasse *ClassLoader* überschrieben. Dazu wird der Methode der Name der angeforderten Klasse als Argument übergeben. Alle Operationen, die zum Laden einer Klasse benötigt werden, müssen in dieser Methode durchgeführt werden.
Im JCoolClassLoader stellt sich der Ablauf der Operationen folgendermaßen dar:

1. Als erstes muss mit der Methode
   - protected final Class findLoadedClass(String name)

   der abstrakten Klasse *ClassLoader* herausgefunden werden, ob die angeforderte Klasse bereits geladen wurde. Ist dies der Fall, kann die Klasse zurückgegeben werden.

2. Wurde die gewünschte Klasse noch nicht geladen, überprüft der JCoolClassLoader, ob es sich bei der angeforderten Klasse um eine Klasse handelt, die zum JCool-Package gehört oder zu denen, die immer vom System geladen werden sollen. Wenn die Klassen zum JCool-Package gehören, werden sie entweder über die Socketverbindung oder vom File-System geladen. Ansonsten geht es mit der vierten Operation weiter.

3. Wurden die Bytes der Klasse mit der zweiten Operation geladen, muss die Methode
   - protected final Class defineClass(String name, byte[] b, int off, int len) throws ClassFormatError

   der abstrakten Klasse *ClassLoader* aufgerufen werden. Diese Methode konvertiert das Array mit den Bytes der angeforderten Klasse in eine Instanz der Klasse *Class*. Fehlt z.B. nur ein Byte, wird eine Exception ausgelöst.

4. Die Methode
   - protected final Class findSystemClass(String name) throws ClassNotFoundException

   lädt alle Systemklassen einschließlich der Klassen, deren Packages im CLASSPATH angegeben werden.

5. Die Methode
   - protected final void resolveClass(Class c)

   wird aufgerufen, wenn das Flag resolve gesetzt ist. Die Methode bewirkt, dass der ClassLoader erneut aufgerufen wird, um alle anderen Klassen zu laden, auf die die geladene Klasse verweist. Die Klassen müssen immer vollständig aufgelöst sein, bevor eine Instanz der Klasse erzeugt werden bzw. eine Methode aufgerufen werden kann.

6. Im letzten Schritt wird anschließend die geladene Klasse mit return zurückgegeben.

Ein Beispiel eines solchen ClassLoaders könnte wie im folgenden Code dargestellt implementiert sein:

```
class CustomerClassLoader extends ClassLoader  {
    protected synchronized Class loadClass(String name, boolean resolve) throws
ClassNotFoundException  {
          // ueberpruefe, ob Klassen schon geladen sind
  Class ret = findLoadedClass(name);                                    // Schritt 1
  //  lade Klassen falls cl == null
  if (ret == null) {
                  try {
              if ( …Customer spezifisch……) {
                  // Lade spezifische Kassen ...
                  byte[] klassenDaten = loadCustomerBytes(name);        // Schritt 2
                  ret = defineClass(name, klassenDaten, 0, klassenDaten.length);  // Schritt 3
                        if (ret == null) {
                     throw new ClassNotFoundException(name);
                   }
                       } else {
                   // Lade Systemklassen ...
                   ret = findSystemClass(name);                         // Schritt 4
                 }
          } catch(ClassNotFoundException classnotfoundexception) {
             } catch(NoClassDefFoundError noclassdeffounderror) {
             } catch(IOException ioexception) {
             }
    if (resolve) {
          resolveClass(ret);                                            // Schritt 5
     }
    return ret;                                                         // Schritt 6
   }
    private byte[] loadCustomerBytes(String name) {
       ……
    }
}
```

Der dargestellte Beispielcode zeigt die beschriebenen Operationsschritte im Kontext zu einer möglichen Implementation.
Um den "CustomerClassLoader" aufzurufen wird die Methode *loadClass(String name, boolean resolve)* nicht direkt aufgerufen, sondern es wird die Methode

- public Class loadClass(String name) throws ClassNotFoundException

der Oberklasse aufgerufen, die ihrerseits die Methode *loadClass(String name, boolean resolve)* aufruft. Der Name der Main-Klasse der Anwendung, die mit

dem "CustomerClassLoader" geladen werden soll, wird dabei als Argument übergeben.

6.2. Properties

Die Klasse *Properties* aus dem Package *java.util*, bietet die Möglichkeit, Datenpaare aus Schlüsseln aus einer Textdatei zu laden und wieder zu speichern. Die Klasse *Properties* ist von der Klasse *Hashtable* abgeleitet und verfügt über alle Möglichkeiten, die auch eine Hashtabelle[55] bietet, mit dem schon genannten Zusatz, die Wertepaare in einer Datei speichern zu können.
Im JCoolClassLoader wird die Möglichkeit, Wertepaare in einer Properties-Datei abzulegen, mehrfach genutzt. Zum einen wird der Hostname und die Portnummer des Servers in einer Properties-Datei gespeichert.
Zum anderen werden in einer Properties-Datei die Hashwerte der JCool-Klassen gespeichert, die auf dem Server liegen, und in einer anderen Properties-Datei werden die Hashwerte der JCool-Klassen gespeichert, die auf dem Client-Rechner liegen. Beide Properties-Dateien werden während des Starts des JCool-ClassLoaders geladen, und es kann so nachgeprüft werden, ob eine Klasse aktualisiert werden muss. Das Speichern der Hashwerte der einzelnen Klassen in einer Properties-Datei hat den großen Vorteil, dass während der Laufzeit viele Arbeitsschritte eingespart werden können, weil nicht zusätzlich für jede Klasse der Hashwert berechnet werden muss. Eine Zeile einer solchen Properties-Datei kann zum Beispiel folgendermaßen aussehen:

COM.ip.cool.widget.gfkTable=90bd61badc8e3fb375f89dcd3c6995a6912058

Der Schlüssel steht vor dem Gleichheitszeichen, in diesem Fall der Name der Klasse, und der Wert steht hinter dem Gleichheitszeichen, dies ist hier der Hashwert (Prüfsumme) der Klasse.
Die wichtigsten Operationen für die Anwendung von Properties, die auch für die Realisierung des JCoolClassLoaders notwendig sind, werden im nächsten Abschnitt vorgestellt.

---

[55] Eine Hashtabelle ist eine Datenstruktur, die es erlaubt, Werte, die einem eindeutigen Schlüssel zugeordnet sind, in einem virtuellen Speicher abzulegen und wieder nachzuschlagen.

Um Properties verwenden zu können, muss als erstes die Klasse Properties instanziert werden. Dazu steht der Konstruktor zur Verfügung:

- public Properties()

Mit Hilfe des Konstruktors ist es möglich, ein leeres Properties-Objekt (auch Properties-Liste genannt) zu erzeugen.
Als nächstes muss diese Properties-Liste gefüllt werden. Das geschieht, indem man entweder mit der Methode

- public Object setProperty(String key, String value)

einen Schlüssel und Wert hinzufügt, oder indem man eine gespeicherte Properties-Textdatei mit Hilfe der Methode

- public void load(InputStream inStream) throws IOException

einliest. Diese Methode lädt eine Properties-Liste aus einem Eingabestrom.
Um auf den Inhalt, also auf die einzelnen Elemente, der Properties-Liste zugreifen zu können, stellt Java die Methoden

- public String getProperty(String key) und
- public String getProperty(String key, String defaultValue)

zur Verfügung. Die erste Methode liefert für den übergebenen Schlüssel den passenden Wert zurück. Bei der zweiten Methode ist die Möglichkeit gegeben, einen Default-Wert anzugeben, falls für den angegebenen Schlüssel kein Eintrag in der Properties-Liste existiert.
Nachdem aufgezeigt wurde, wie man eine Properties-Liste instanziert, sie füllt und die einzelnen Elemente ausliest, fehlt nur noch eine Möglichkeit, diese Liste zu speichern. Dazu muss die Methode

- public void store(OutputStream out, String header) throws IOException

verwendet werden. Diese Methode speichert die Properties-Liste mit Hilfe des Ausgabestroms ab.
Das folgende Codebeispiel zeigt, wie die vorherige Ausführung umgesetzt werden kann:

```
Properties myProperties = new Properties();
try {           // lade Properties-File
        FileInputStream in = new FileInputStream("myPropertiesFile.properties");
        myProperties= new Properties();
        myProperties.load(in);
        in.close();
        String myValue = myProperties.getProperty("key");
        myProperties.setProperty("newKey", "newValue");
        FileOutputStream out = new  FileOutputStream("myPropertiesFile.properties");
        myProperties.store(out, "Pruefsumme der Klassen");
        out.close();
        } catch(IOException e) {
System.out.println("Fehler : " + e);
```

6.3. MessageDigest

Zu den bereits vorgestellten Klassen der Java API, die für die Umsetzung des JCoolClassLoaders bedeutend sind, gehört auch die Klasse *MessageDigest* aus dem Package *java.security*. Diese Klasse ist im JCoolClassLoader nicht nur für das Thema Digitale Signatur von Bedeutung. Mit Hilfe dieser Klasse werden auch die Hashwerte (Prüfsumme, Fingerprints) von allen Klassen des Moduls JCool auf Client- wie auch auf Serverseite gebildet. Diese Hashwerte, die den einzelnen Klassen zugeordnet sind, werden dazu genutzt zu überprüfen, ob eine Klasse des Moduls JCool aktualisiert werden muss. Ein kleines Beispiel wie eindeutig Hashwerte sind, wurde im Kapitel 4.5. Hashfunktionen gegeben.
Mit der Klasse *MessageDigest* des JDK 1.3 können Prüfsummen sowohl mit dem SHA-1-Algorithmus als auch mit dem MD5-Algorithmus gebildet werden. Für die mit der Klasse *MessageDigest* gebildeten Prüfsummen gelten die gleichen Eigenschaften bzw. Bedingungen wie für Hashfunktionen.
Zum einen muss die Wahrscheinlichkeit, dass zwei unterschiedliche Dokumente dieselbe Prüfsumme haben, vernachlässigbar gering sein.
Zum anderen muss es unmöglich sein, aus der berechneten Prüfsumme auf das ursprüngliche Dokument zurückzuschließen.
Um eine Prüfsumme mit der Klasse *MessageDigest* zu berechnen, müssen folgenden Schritte durchgeführt werden:

1. Als erstes muss ein MessageDigest Objekt erzeugt werden. Diese werden nicht direkt instanziiert, sondern dazu wird die statische Methode
    - public static MessageDigest getInstance(String algorithm) throws NoSuchAlgorithmException

aufgerufen. Als Argument wird dabei die Bezeichnung des SHA-1- oder des MD5-Algorithmus angegeben.

2. Mit der Methode
    - public void reset()

   werden vorherige Digest zurückgesetzt.

3. Nachdem nun ein MessageDigest Objekt erzeugt wurde, werden im zweiten Schritt z.B. mit der Methode
    - public void update(byte[] input)

   alle Bytes an das MessageDigest Objekt übergeben.

4. Im dritten Schritt wird mit der Methode
    - public byte[] digest()

   die eigentliche Berechnung durchgeführt. Dazu wird der Algorithmus genutzt, der vorher als Argument mitgegeben wurde. Als Ergebnis erhält man ein Array mit 16 Byte unter Verwendung des MD5-Algorithmus oder ein Array mit 20 Byte unter Verwendung des SHA-1-Algorithmus.

Die Java API bietet unter anderem noch die Methode

- public static boolean isEqual(byte[] digesta, byte[] digestb)

an, um zwei Prüfsummen auf Gleichheit zu überprüfen. Der JCoolClassLoader verwendet diese Möglichkeit aber nicht, weil die Prüfsummen als Strings in den Properties-Files gespeichert sind.

Ein Codebeispiel verdeutlicht die einzelnen Schritte:

```
try {
        MessageDigest messDigest = MessageDigest.getInstance("SHA-1");      // Schritt 1
  // Datei einlesen
  File file = new File(mAbsolutePathName);
  FileInputStream file_input = new FileInputStream(file);
        String dig = "";
        int fileLength = (int) file.length();
        byte[] bytes = new byte[fileLength];
        byte[] buffer = new byte[1024];
        int tempLength = 0;
        int bytesPointer = 0;
        while ((tempLength = file_input.read(buffer)) >= 0) {
                System.arraycopy(buffer, 0, bytes, bytesPointer, tempLength);
        bytesPointer += tempLength;
         }
         messDigest.reset();                                                  // Schritt 2
```

```
            messDigest.update(bytes);                                   // Schritt 3
            byte[] digest = messDigest.digest();                        // Schritt 4
  //  in String ueberfuehren
            for (int i = 0; i < digest.length; i++) {
           dig += Integer.toHexString(digest[i] & 0xFF);
            }
           file_input.close();
} catch(NoSuchAlgorithmException e) { …}
        catch(FileNotFoundException e) {… }
        catch(IOException e) {… }
```

Im JCoolClassLoader wird die Prüfsumme der Klassen des Moduls JCool mit dem SHA-1-Algorithmus berechnet, weil eine Kollision mit dem SHA-1-Algorithmus sehr unwahrscheinlich ist, und weil der MD5-Algorithmus, wie schon im Kapitel über Hashfunktionen angesprochen, als nicht mehr sicher gilt.

Zur Erstellung des Properties-Files mit den Hashwerten einer neuen Version des Moduls JCool wurde ein kleines Zusatzmodul[56] geschrieben. Dabei werden die Klassen rekursiv aus dem File-System ausgelesen, die Prüfsumme wird berechnet und danach in einem Properties-File speichert.

[56] Der Quellcode zum Zusatzmodul befindet sich im Anhang.

7. Der JCoolClassLoader

Der JCoolClassLoader[57] wurde entwickelt, um die Klassen des Moduls JCool während der Laufzeit dynamisch zu laden und zu aktualisieren. Dazu sollen die Klassen entweder von dem Client-System oder von einem Server während der Laufzeit geladen werden. Ein Schwerpunkt bei der Umsetzung der Aufgabenstellung war das Thema der Sicherheit. Es muss sichergestellt sein, dass keine manipulierten Klassen auf das Client-System gelangen bzw. geschrieben werden. Auch muss der autorisierte Zugriff auf den Server sichergestellt sein. Ein weiterer Punkt, der für die Umsetzung der Aufgabenstellung von Bedeutung war, ist die Forderung des Auftraggebers, dass die Aktualisierung des Moduls JCool vollkommen unbemerkt für den Anwender von statten gehen muss, und dass der Ablauf des Moduls JCool in keiner Weise beeinträchtigt wird. Um diese Kriterien umzusetzen, musste erstens eine clientseitige Anwendung geschrieben werden, zweitens musste der Server, der die Anfragen des Moduls JCool bearbeitet, entsprechend erweitert werden. Zu dem Thema Sicherheit wurde sowohl die Java-Sicherheitsarchitektur und das Java Sicherheitsmodell betrachtet, als auch die Grundzüge der Kryptographie unter Berücksichtigung einzelner Verschlüsselungstechniken. Wie der Programmablauf des JCoolClassLoaders mit den in Kapitel 5 und 6 betrachteten Klassen der Java API und den Sicherheitsaspekten stattfindet, wird in den folgenden Unterkapiteln beschrieben.

7.1. Programmablauf der clientseitigen Anwendung

Ziel der clientseitigen Anwendung ist es, mit dem JCoolClassLoader das Modul JCool zu starten, und bei Bedarf die zu aktualisierenden Klassen des Moduls JCool vom Server zu laden und auf das Client-System zu speichern. Damit ein sicherer und autorisierter Zugriff auf den Server erfolgen kann, werden alle Anfragen, die als erstes an den Server gerichtet sind, mit Hilfe der RSA-Verschlüsselung chiffriert. Da die Anfragen an den Server nur einzelne Wörter lang sind, beeinträchtigt der Zeitaufwand für die Verschlüsselung die Performance in keiner Weise.

Für die Realisierung des Ziels unter Berücksichtigung der Sicherheit werden folgende Schritte durchlaufen:

---

[57] Der clientseitige sowie serverseitige Quellcode des JCoolClassLoaders befindet sich im Anhang.

1. Der JCoolClassLoader wird instanziert,

   ClassLoader loader = new JCoolClassLoader();

   und danach wird die Methode

   Class c = loader.loadClass(name);

   mit dem Namen der Main-Klasse des Moduls JCool als Argument überschrieben.

2. Während der Initialisierung werden die Properties-Files
   - security.properties (z.B. Host-Name des Servers)
   - jcoolClientDigest.properties (Hashwerte aller JCool-Klassen auf dem Client-System)
   - jcoolServerDigest.properties (Hashwerte aller JCool-Klassen auf dem Server)

   und die Keys für die RSA-Verschlüsselung geladen. Das File mit den jcoolServerDigest.properties wird über die Socketverbindung geladen. Der Ablauf stellt sich wie folgt dar:

   a) Es wird eine mit dem Public-Key des Betreibers verschlüsselte Anfrage an den Server geschickt. Diese Anfrage signalisiert dem Server, dass die Methode, die für die Anfragen von dem Class-Loader verantwortlich ist, aufgerufen wird.
   b) Danach wird das jcoolServerDigest.properties File angefordert.
   c) Der Client-Socket liest den mit dem Private-Key des Betreibers verschlüsselten Hashwert des Properties-Files.
   d) Der verschlüsselte Hashwert des Properties-Files wird anschließend mit dem Public-Key des Betreibers dechiffriert.
   e) Als nächstes liest der Client-Socket das unverschlüsselte Properties-File ein.
   f) Über das Properties-File wird nun der Hashwert gebildet und mit dem entschlüsselten Hashwert verglichen, der vom Server eingelesen wurde. Sind die Hashwerte verschieden, dass heißt wenn das Properties-File verändert wurde, bricht das Programm mit dem Hinweis auf diese Manipulation ab. Das geschilderte Verfahren entspricht dem im Kapitel 4.5 vorgestellten Verfahren der digitalen Signatur.

3. Die eigentliche Arbeit des JCoolClassLoaders findet nun in der von der Oberklasse aufgerufenen Methode
   loadClass(String name, boolean resolve) statt.

4. In dieser Methode wird als erstes die bereits besprochene Methode *findLoadedClass(String name)* aufgerufen, um zu überprüfen, ob die Klasse bereits geladen wurde.

5. Ist das nicht der Fall, wird ermittelt, ob die angeforderte Klasse zu dem Package der JCool Klassen gehört. Wenn dies nicht der Fall ist, wird die angeforderte Klasse mit der Methode *findSystemClass(String name)* geladen.

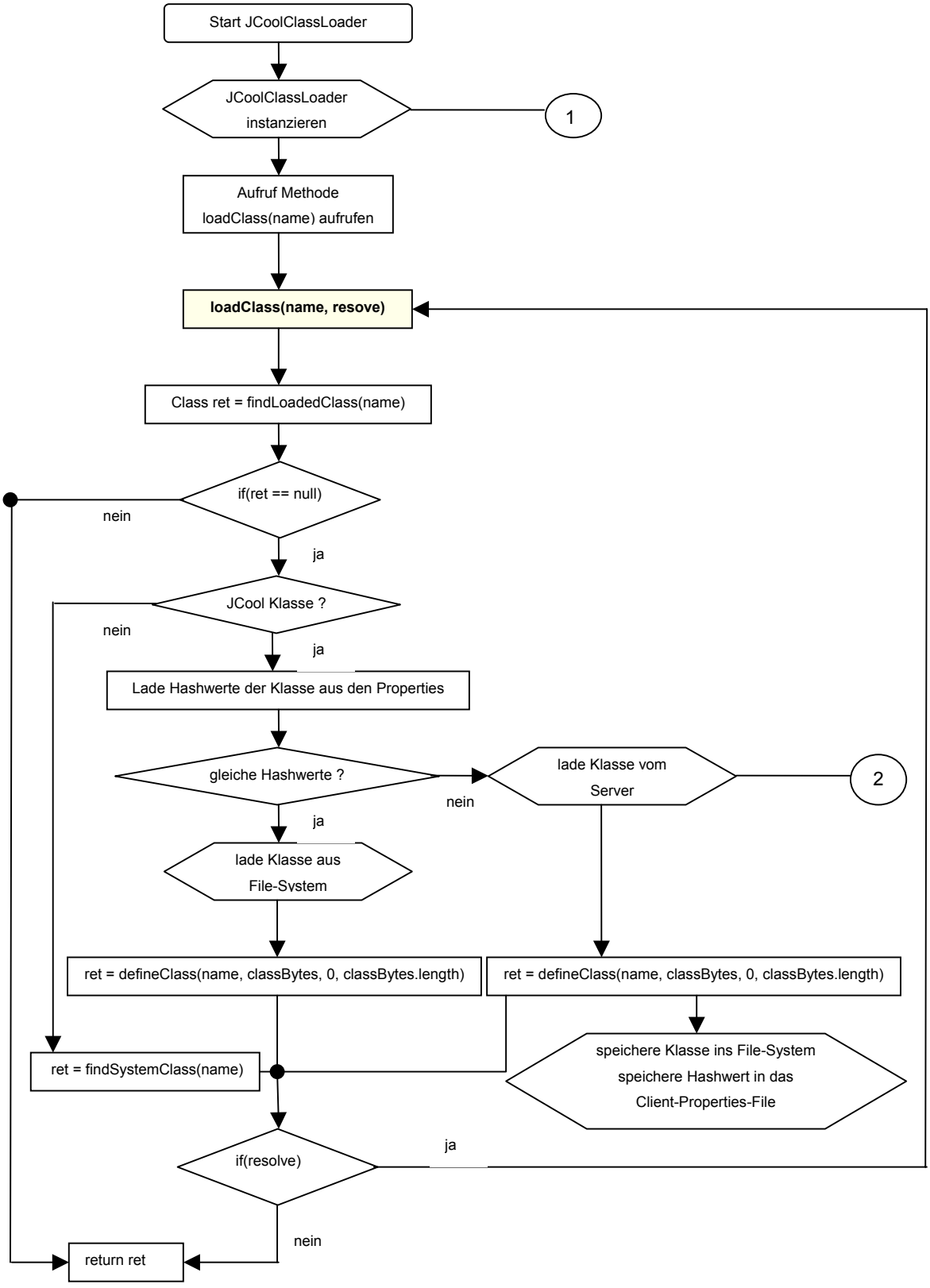
6. Als nächstes werden die Hashwerte der angeforderten Klasse, sowohl die clientseitigen als auch die serverseitigen Hashwerte, geladen und verglichen. Sind die Hashwerte gleich, wird die angeforderte Klasse mit der Methode *private byte[] loadClientFileBytes(String name)* des JCoolClassLoaders vom Client-System geladen. Wenn die Hashwerte nicht gleich sind oder der clientseitige Hashwert für die Klasse nicht existiert, wird die angeforderte Klasse vom Server geladen.
   Dies geschieht wie folgt:

   a) Die Methode *private byte[] loadServerClassBytes(String name, String serverDig)* des JCoolClassLoaders wird aufgerufen. Als Argument wird der Name der angeforderten Klasse und der serverseitige Hashwert übergeben.
   b) Danach wird mit dem Public-Key des Betreibers die verschlüsselte Anfrage an den Server geschickt. Diese Anfrage signalisiert dem Server, dass die Methode, die für die Anfragen von dem ClassLoader verantwortlich ist, aufgerufen wird. Danach wird der Serveranwendung mitgeteilt, dass eine Klasse angefordert wird, und anschließend wird der Name dieser Klasse gesendet.
   c) Anschließend wird die Anzahl der zu übermittelten Bytes und danach die Bytes der Klasse eingelesen.
   d) Über die eingelesene Klasse wird nun der Hashwert gebildet und mit dem der Methode übergebenen serverseitigen Hashwert verglichen. Dieses Verfahren ist notwendig, weil die Klassen unverschlüsselt gesendet werden, und somit theoretisch die Möglichkeit der Manipulation der Klassen besteht. Wenn die beiden

Hashwerte gleich sind, werden die Bytes der Klasse zurückgegeben.

7. Mit der Methode defineClass(String name, byte[] b, int off, int len) wird anschließend die Klasse generiert.
8. Im nächstem Schritt wird die neu geladene Klasse auf dem Client-System gespeichert und der Hashwert der aktualisierten Klasse wird in den jcoolClientDigest.properties File gespeichert.
9. Nach der Überprüfung, ob das Flag resolve gesetzt ist, wird die angeforderte Klasse zurückgegeben.

Die folgender Abbildung veranschaulicht die Ausführungen:


Start JCoolClassLoader
JCoolClassLoader instanzieren
1
Aufruf Methode loadClass(name) aufrufen
loadClass(name, resove)
Class ret = findLoadedClass(name)
if(ret == null)
nein
ja
JCool Klasse ?
nein
ja
Lade Hashwerte der Klasse aus den Properties
gleiche Hashwerte ?
nein
lade Klasse vom Server
2
ja
lade Klasse aus File-System
ret = defineClass(name, classBytes, 0, classBytes.length)
ret = defineClass(name, classBytes, 0, classBytes.length)
ret = findSystemClass(name)
speichere Klasse ins File-System
speichere Hashwert in das Client-Properties-File
if(resolve)
ja
nein
return ret

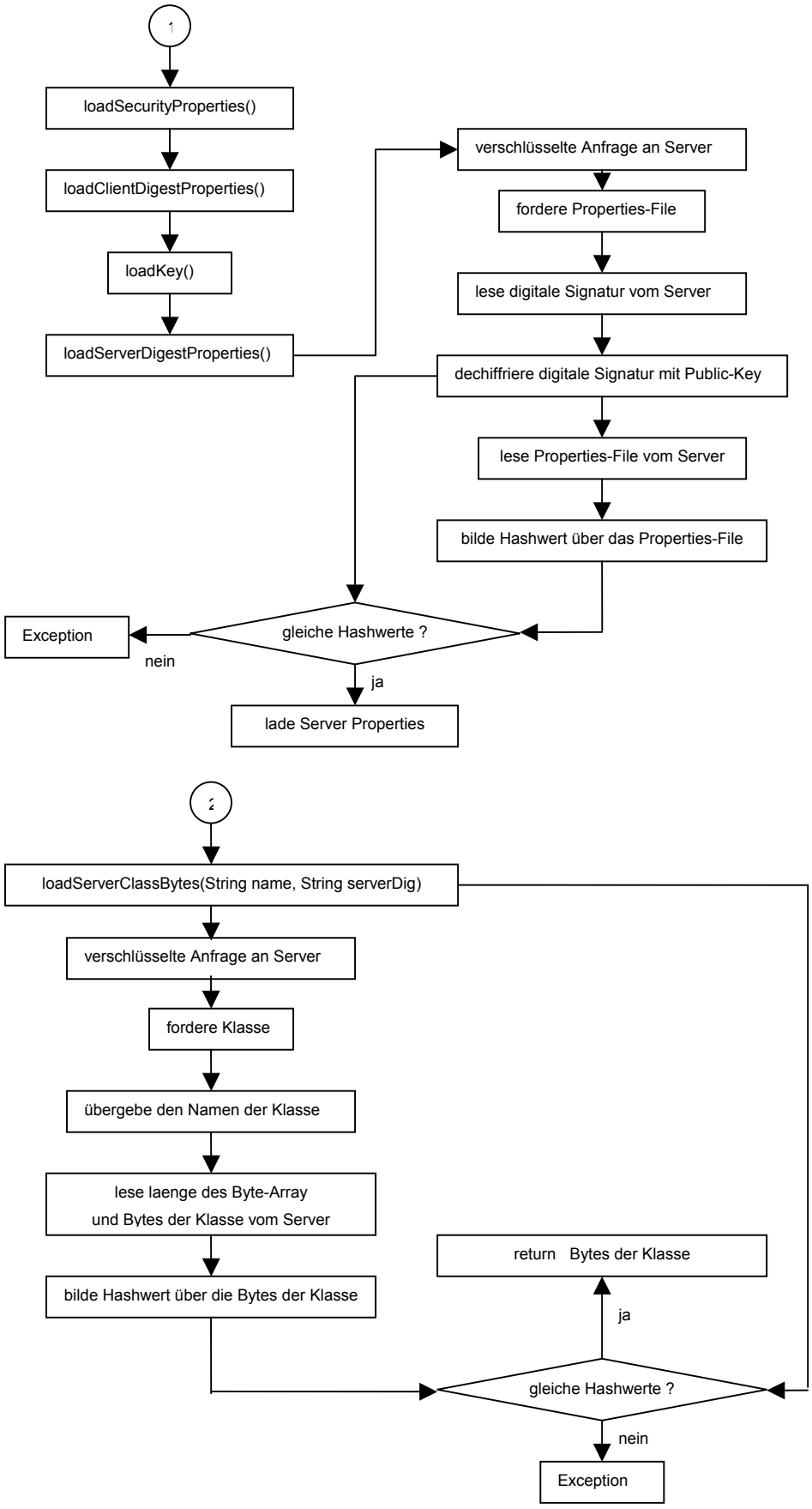


Abbildung 7.1 : Clientseitiger Programmablauf des JCoolClassLoaders

Alle Klassen, die die JVM anfordert, werden über die Methode *loadClass(String name, boolean resolve)* geladen. Somit ist es also gezielt möglich, die dynamisch angeforderten Klassen auf speziell definierten Weg zu laden. Für eine höchstmögliche Sicherheit sorgen zum einen der verschlüsselte Zugriff auf den Server, und zum anderen die digitale Signatur des jcoolServerDigest.properties-Files. Dadurch kann jede vom Server geladene Klasse mittels der Hashwerte auf Unversehrtheit überprüft werden kann. Denn es ist besonders wichtig, dass die Integrität der über den Server geladenen Klassen anhand der Hashwerte überprüft wird. Es muss ausgeschlossen werden, dass manipulierter Code auf das Client-System gelangt.

7.2. Programmablauf der serverseitigen Anwendung

Ziel der serverseitigen Anwendung ist es, die Anfragen des JCoolClassLoaders unter Einbeziehung der Sicherheit zu bearbeiten. Dafür wurde der Server, der die Anfrage des Moduls JCool bearbeitet, entsprechend erweitert. In dieser Arbeit wird nur die Erweiterung bezüglich des JCoolClassLoaders besprochen. Folgende Schritte zur Realisierung dieses Ziels werden durchlaufen:

1. Nachdem der Server für die Anfrage vom JCoolClassLoader einen eigenen Thread gestartet hat, wird der erste eingelesene chiffrierte Stream mit dem Private-Key des Betreibers dechiffriert und überprüft, ob es sich um eine Anfrage des Moduls JCool oder um eine Anfrage des JCoolClassLoaders handelt.
2. Wenn die Anfrage vom JCoolClassLoader stammt, wird die Methode zur Bearbeitung dieser Anfragen aufgerufen und als Argument der dechiffrierte String übergeben.
3. Anschließend liest die Serveranwendung den nächsten gesendeten Stream ein, der signalisiert, ob das jcoolServerDigest.properties File angefordert ist oder eine Klasse des Moduls JCool.
4. Handelt es sich um das Properties-File, werden folgende Schritte durchlaufen:

    a. Das Properties-File wird eingelesen und anschließend wird darüber der Hashwert gebildet.
    b. Nun wird mit dem Private-Key des Anbieters der Hashwert chiffriert und zum Client gesendet.

c. Als nächster Schritt wird nun das Properties-File an den Client gesendet und die Socketverbindung wird geschlossen.

5. Wird aber eine Klasse des Moduls JCool angefordert, sieht der Ablauf folgendermaßen aus:

    a. Der Stream mit dem Namen der angeforderten Klasse wird eingelesen.
    b. Die Klasse wird vom System des Servers eingelesen, und die Anzahl der Bytes an den Client gesendet.
    c. Nun kann die Klasse an den Client gesendet werden, und anschließend wird die Socketverbindung geschlossen.

In folgender Abbildung wird die Ausführung nochmals veranschaulicht:

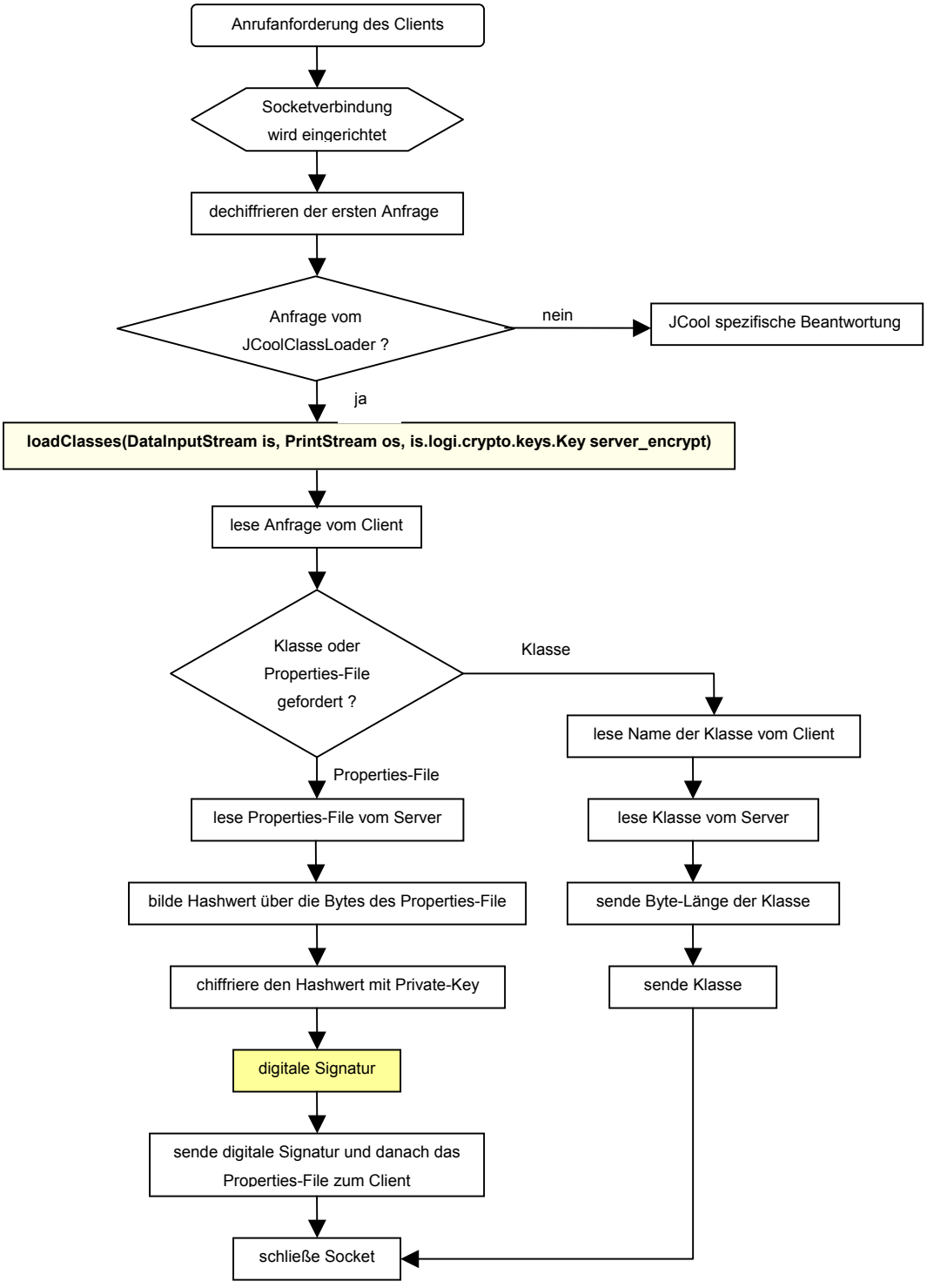


Abbildung 7.2 : Serverseitiger Programmablauf des JCoolClassLoaders

Die Sicherheitsmaßnahmen in der Serveranwendung liegen hauptsächlich in der verschlüsselten Anfrage, dem darin übermittelten Begriff und im Signieren des jcoolServerDigest.properties File begründet.

8. Resümee

In der vorliegenden Diplomarbeit wurden die theoretischen Grundlagen zur Umsetzung eines eigenen dynamischen netzwerkorientierten Klassenladers unter besonderer Berücksichtigung der Sicherheitsaspekte der Programmiersprache Java und der Kryptographie besprochen.

Für die spezielle Umsetzung des JCoolClassLoaders wurde zunächst das Modul JCool und insbesondere die bestehende Packagestruktur betrachtet. Weiterhin wurde der ClassLoader der Java Virtual Machine (JVM) im Zusammenhang zur Java Architektur und die Java Sicherheitsarchitektur sowie das Java Sicherheitsmodell vorgestellt. Ein besonderer Schwerpunkt wurde auf die Grundlagen der Kryptographie gesetzt, da die Kryptographie viele Möglichkeiten für die Sicherheit der Daten bietet. Die Möglichkeit einer Client-Server Anwendung über Sockets in Java und die für die Umsetzung des Programms relevanten Klassen der Java API waren danach Gegenstand der Betrachtung. Als letzter Punkt wurde der Programmablauf des JCoolClassLoaders vorgestellt, in dem unter anderem alle sicherheitsrelevanten Aspekte erläutert wurden.

In dieser Arbeit konnte die Umsetzung der RSA-Verschlüsselung nicht behandelt werden, da es sich bei dem verwendeten Krypto-Package um eine kommerzielle Anwendung handelt und der Zugriff darauf teilweise über Klassen des Moduls JCool erfolgt.

Während der Umsetzung des Programms traten anfänglich Schwierigkeiten auf, die auf die Arbeitsweise des ClassLoaders der JVM und auf die Package-Struktur[58] des Moduls JCool zurückzuführen waren. Denn wenn eine geladene Klasse auf eine andere Klasse verweist, verlangt die JVM, dass die referenzierte Klasse von demselben ClassLoader geladen wird, wie die ursprüngliche Klasse. Das heißt, wenn die erste Klasse z.B. über die Socketverbindung geladen wird, will die JVM die referenzierten Klassen auch über diesem Weg laden. Da aber zusätzlich auch noch zwei Packages, die keiner Aktualisierung bedürfen, innerhalb der Packagestruktur des Moduls JCool liegen, musste durch eine gezielte Abfrage sichergestellt werden, das die angeforderten Klassen über klar definierten Weg geladen werden. Im Kapitel 7.1 'Programmablauf der clientseitigen Anwendung' werden die Programmabläufe, die einem klar definierten Weg vorgeben, dargestellt. Nachdem die Bedeutung der Packagestruktur des Moduls JCool

im Zusammenhang mit dem Verhalten der JVM erkannt wurde, konnte die weitere Umsetzung des Programmes ohne Probleme durchgeführt werden.

Einsetzbar ist der entwickelte Klassenlader mit geringen Änderungen auch für andere Java-Anwendungen. Außerdem besteht die Möglichkeit eine andere Verschlüsselungstechnik einzusetzen. Mit Hilfe von symmetrischen Verschlüsselungstechniken ist es aufgrund der Schnelligkeit der Algorithmen denkbar, die über den Server angeforderten Klassen zu verschlüsseln.

Der JCoolClassLoader befindet sich zur Zeit bei dem Auftraggeber in der Testphase und wird voraussichtlich in den nächsten Monaten bei den Werbeagenturen eingesetzt werden.

[58] Auf die Bedeutung der Package-Struktur wurde in Kapitel 2 und 3 hingewiesen.

Literaturverzeichnis

Gedruckte Literatur

Calvert, Kenneth L.; Donahoo, Michael J.: TCP/IP Sockets in Java, London u.a.: Academic Press, 2002.

Eckert C.: IT-Sicherheit: Konzepte–Verfahren–Protokolle, München: Oldenburger Wissenschaftsverlag GmbH, 2001.

Gerling, Rainer W.: Verschlüsselung im betrieblichen Einsatz, 1. Auflage, Frechen-Königsdorf: Datakontext-Fachverlag GmbH, 2000.

Horstmann, Cay S. ; Cornell, Gary: Core Java 2: Band 1 - Grundlagen, München: Markt und Technik, 1999.

Horstmann, Cay S. ; Cornell, Gary: Core Java 2: Band 2 - Expertenwissen, München: Markt und Technik, 2000.

Hunt, Craig: TCP/IP: Netzwerk-Administration, 2. Auflage, Köln: O'Reilly Verlag, 1998.

Krüger, Guido: GoTo Java 2: Handbuch der Java-Programmierung, 2. Auflage München: Addison-Weslay Verlag, 2000.

Lipp, Peter; u.a.: Sicherheit und Kryptographie in Java: Einführung, Anwendung und Lösungen, München: Addison-Weslay Verlag, 2000.

Selke, Gisbert W.: Kryptographie: Verfahren, Ziele, Einsatzmöglichkeiten, 1. Auflage, Köln: O'Reilly Verlag, 2000.

Smith, Richard E.: Internet–Kryptographie, Bonn: Addison-Weslay-Longman Verlag, 1998.

Venners, Bill: Inside the Java 2 Virtual Machine, Second Edition, USA: McGraw-Hill Companies 1999.

Wobst, Reinhard: Abenteuer Kryptologie: Methoden, Risiken und Nutzen der Datenverschlüsselung, 3. Auflage, München: Addison-Weslay Verlag, 2001.

Internetquellen

Bandouch, Jan: Shared Key Kryptographie (DES, IDEA) und PGP, <http://wwwbrauer.in.tum.de/seminare/web/WS0001/vortrag09.html> (17.07.2002)

Bank, Carsten: Java Sockets, <http://www.stud.uni-siegen.de/carsten.bank/java4/Inhalt.html> (03.09.2002)

Holtkamp, Keiko: Einführung in TCP/IP, <http://www.rvs.uni-bielefeld.de/~heiko/tcpip/inhalt.html> (12.09.2002)

Mekelburg, Hans-G.: Kryptologie: Verschlüsselte Botschaften: <http://home.nordwest.net/hgm/krypto/> (07.07.2002)

Thöing, Christian: Kryptographie: <http://mitglied.lycos.de/cthoeing/crypto/index.htm> (17.07.2002)

o.V.: ECC - Elliptic Curve Cryptography - Einsatz in der Kryptographie - <http://www.cryptovision.com/deutsch/service/ppt/ecc.ppt> cryptovison gmbh 2000, (12.08.2002)

o.V.: Java Virtual Machine (JVM), <http://www.zuggi.de/java/vortrag/jvm.htm> (02.009.2002)

o.V.: Understanding the Java Virtual Machine, <www.quickreferences.yucom.be/P&T/Understanding%20the%20Java%20ClassLoader.pdf> (20.06.2002)

o.V.: JCool: Das Online Tool zur Verwaltung von Werbezeitenhttp://www.s4m.de/deutsch/portfolio/JCool.html (16.09.2002)