Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Secure Java Code Interpreter for Public Utility Computing

Roger Lacher

29. Oktober 2004

Diplomarbeit NA-2004-1

April 2004 bis Oktober 2004

Tutor: Thomas Dübendorfer

Co-Tutor: Dr. Germano Caronni

Supervisor: Prof. Bernhard Plattner

Zusammenfassung

Public Utility Computing bezeichnet ein Szenario, in welchem ein Anwender den grösstmöglichen Teil seiner IT-Infrastruktur an öffentliche Dienstanbieter auslagert. Dabei stellt die Exposition der Programme und Daten des Dienstbenutzers an den Dienstanbieter eines der Hauptprobleme dar.

In dieser Arbeit werden notwendige Massnahmen für eine sichere und vertrauliche Ausführung von Java Programmen auf einer öffentlichen Plattform beschrieben. Die aufgezeigte Lösung basiert auf kryptographischen Algorithmen und zuverlässiger, manipulationssicherer Hardware zur Sicherstellung von Vertraulichkeit und Integrität der Daten und Programme. Der Bytecode sensitiver Java-Methoden wird verschlüsselt und in einem neu angefügten Methodenattribut innerhalb der bestehenden Struktur eines Classfiles untergebracht. Die Eingabedateien des Programms werden durch den Benutzer verschlüsselt und während der Programmausführung durch das Anwenderprogramm selbst entschlüsselt. Sämtliche Dateien und Classfiles werden beim Einlesen für die Programmausführung anhand von MD5-Hashes auf ihre Integrität hin überprüft. Ein Konfigurationsfile erlaubt die sichere Übergabe der Schlüssel und Hashwerte sowie zusätzlicher benutzerdefinierter Werte.

Der Bytecode verschlüsselter Methoden wird erst zum Zeitpunkt der Programmausführung entschlüsselt. Die Entschlüsselung erfolgt segmentweise in einen sicheren Speicherbereich der manipulationssicheren Hardware. Zugriffe auf den Methodencode werden in das entschlüsselte Segment umgelenkt. Die Java VM überwacht den Programmzähler, um festzustellen, ob ein neues Segment entschlüsselt werden muss.

Vertraulichkeit und Integrität der von der Laufzeitumgebung während der Programmausführung verwalteten Datenstrukturen werden ebenfalls mittels segmentation Verschlüsselung und Hashwerten gewährleistet.

Als Proof of Concept wurden ein Werkzeug zur Verschlüsselung von Classfiles entwickelt und eine Java VM so modifiziert, dass sie die verschlüsselten Klassen ausführen kann.

Die Arbeit zeigt, dass die vorgeschlagene segmentweise Entschlüsselung und Ausführung von verschlüsseltem Java Bytecode realisierbar ist. Ebenfalls zeigt sie, dass eine segmentweise Verschlüsselung und Entschlüsselung der Java Laufzeit-Stacks zwischen einem ungeschützten und einem sicheren Bereich des Arbeitsspeichers realisierbar ist. Bei einem Programm mit mehreren Threads muss sichergestellt werden, dass das momentan entschlüsselte Segment in den ungeschützten Speicherbereich zurückverschlüsselt wird, bevor das System einen neuen Thread zur Ausführung ansetzt.

Abstract

Public Utilty Computing is the name of a scenario where an user out-sources as much of his IT infrastructure to a public provider as he possibly can. The disclosure of the user's code and data is considered as one of the major problems of this approach.

This work aims at defining a set necessary measures for a secure and trusted execution of Java programs on a public platform. The solution presented is based on cryptographic algorithms and trusted, tamper resistant hardware to ensure the confidentiality and integrity of the code and data. The bytecode of a sensitive Java method is encrypted and saved in a newly created attribute of the method preserving the classfile's overall structure. The user encrypts the input data of the his program himself. The user's program itself decrypts them during it's execution. All input data and classfiles are checked for integrity upon loading into the platform for execution by using MD5 hashes. A configuration file allows us to pass the encryption keys and hashes as well as additional, user defined key-value pairs to the program.

The encrypted method bytecode is not decrypted until program execution. Decryption occurs segment by segment inside a secure memory region on the tamper proof device. All access to the method code is redirected to the decrypted segment. The Java execution environment monitors the program counter to check, if a new segment needs to be decrypted.

The confidentiality and integrity of the data structures managed by the runtime environment are also ensured by encryption and hashing.

As a proof of concept, a tool for encryption of classfiles has been written and an existing Java VM modified to enable the execution of the encrypted bytecode.

This work shows, that the proposed segment-wise decryption and execution of encrypted Java bytecode is feasible. It also shows, that a segment-wise segment-wise encryption and decryption of the Java runtime stacks between an unproteced and a secure memory region is feasible. For a program with multiple threads it must be ensured that the currently decrypted segment is encrypted back to the unprotected memory area before the system schedules a new thread for execution.

Inhaltsverzeichnis

1 Einleitung **7**

1.1 Public Utility Computing . . . 7
1.2 Ziel der Arbeit . . . 8
1.3 Begriffe . . . 9
1.4 Notation . . . 9

2 Verwandte Arbeiten **10**

2.1 Mobile Computing . . . 10
2.2 Obfuscated Software . . . 10
2.3 Tamper Resistant Software . . . 10
2.4 Rechnen mit verschlüsselten Funktionen . . . 11
2.5 Secure Multi-Party Computation . . . 11
2.6 Tamper Resistant Hardware . . . 12

3 Konzept und Design **13**

3.1 Ablauf Benutzung PUC Rechendienst . . . 13
3.2 Angriffsszenarien . . . 14
3.2.1 Passive Angriffe . . . 14
3.2.2 Aktive Angriffe . . . 15
3.3 Systemanforderungen . . . 16
3.3.1 Sicherheitsanforderungen . . . 16
3.3.2 Allgemeine Entwicklungsziele . . . 18
3.4 Voraussetzungen . . . 18
3.4.1 Hardwarearchitektur . . . 19
3.4.2 Initialisierung und Zertifizierung . . . 21
3.4.3 Datenfluss . . . 22
3.5 Designentscheidungen . . . 23
3.5.1 Granularität der Verschlüsselungseinheiten . . . 23
3.6 Zusammenfassung der Abwehrmassnahmen . . . 24

4 Spezifikation **25**

4.1 Programm- und Datenvorbereitung . . . 25
4.1.1 Ablauf Programmvorbereitung . . . 25
4.1.2 Verschlüsselung der Programmeinheiten . . . 25
4.1.3 Schlüsselgenerierung für Methodenverschlüsselung . . . 28
4.1.4 Verschlüsselung Benutzerdaten . . . 29

4.1.5 Integrität von Programm- und Eingabedateien 29
4.1.6 Verschlüsselungswerkzeug 29
4.2 Programm- und Datenübertragung 32
4.3 Schlüsselaustausch 32
4.4 Sichere Programmausführung 33
4.4.1 Initialisierung der Java VM 33
4.4.2 Entschlüsselung des Benutzerschlüssels 34
4.4.3 Benutzerdefinierte Schlüssel-Werte-Paare 34
4.4.4 Integritätsprüfung beim Laden von Dateien 34
4.4.5 Entschlüsseln und Ausführen sicherer Methoden 36
4.4.6 Sichere Methodenausführung 36
4.4.7 Laufzeitdaten 38
4.5 Übertragung der Resultate und Validation 40
4.6 Sicherheits-Policy für die PUC-Plattform 41

5 Implementierung **42**
5.1 Verschlüsselungswerkzeug 42
5.1.1 Benutzte Libraries und Software 42
5.1.2 Übersicht Module 42
5.2 Modifikation Java VM 42
5.2.1 Benutzte Libraries und Software 42
5.2.2 Übersicht Module 44
5.2.3 Code 44
5.2.4 Stack 44
5.2.5 Heap 44
5.3 Implementierungsstand 46

6 Validation und Ergebnisse **47**
6.1 Validation 47
6.1.1 Testcases 47
6.1.2 Testdurchführung 49
6.1.3 Testergebnisse 50
6.2 Bemerkungen 50
6.2.1 Code 50
6.2.2 Stack 52
6.2.3 JNI 53
6.3 Vervollständigung der Implementierung 53

7 Zusammenfassung **56**
7.1 Schlussfolgerungen 56
7.2 Weiterführende Arbeiten 57
7.2.1 Modularisierung 57
7.2.2 PUC in Supernetzen 57

A Original Problem Statement **59**

Abbildungsverzeichnis

1 PUC Szenario für Rechendienst 8
2 Ablauf PUC Dienstbenutzung 13
3 Java Runtime Plattform als sicheres Modul 20
4 Datenfluss 23
5 Entschlüsselung eines Codesegments 38
6 Verschlüsselung / Entschlüsselung eines Stack Segments 39
7 Abstand für Opcode Rewriting 51

Tabellenverzeichnis

1 Speicherbereiche 21
2 Datenstrukturen einer Java VM 22
3 Massnahmen gegen Angriffe 24
4 Programm- und Datenvorbereitung 26
5 Programm- und Datenübertragung. 32
6 Sichere Programmausführung 33
7 Übertragung der Resultate und Validation 40
8 PUC System Policy 41
9 Klassen des Verschlüsselungswerkzeugs 43
10 Modifizierte Module JamVM 45
11 Implementierungsstand 46
12 Testcases 48
13 Konstanten 49
14 Native Methoden 54
15 Cache Algorithmen für Modularisierung 57

Algorithmenverzeichnis

1 Integritätsprüfung beim Laden von Dateien 35
2 Laden und Initialisieren einer Klasse 37
3 Methodenausführung 39

1 Einleitung

1.1 Public Utility Computing

Utility Computing bezeichnet ein Szenario, in welchem ein Anwender die von ihm benötigten IT-Ressourcen und Dienste - Netzwerk, Speicher, Rechenleistung - je nach Bedarf bezieht. Von *Public* Utility Computing (PUC) wird gesprochen, wenn die entsprechenden Dienste von einem oder mehreren miteinander konkurrenzierenden, öffentlichen Anbietern offeriert werden.

Der Benutzer wählt unter den zur Verfügung stehenden Anbietern je nach seinen Bedürfnissen den günstigsten, grössten, schnellsten oder billigsten aus und überantwortet ihm die Erbringung des geforderten Dienstes. Die angebotenen Dienste sollen bezüglich Verfügbarkeit, Performanz und Sicherheit vorgegebene Anforderungen erfüllen. Die Vorteile für ein Unternehmen sind Kosteneinsparungen bei Investitionen und Unterhalt, garantierte Servicequalität, Verbesserung der Betriebseffizienz und verbesserte Anpassung an sich verändernde Anforderungen. Interessant ist PUC also vor allem für kleine und mittlere Unternehmen (KMU), die sich die Investitionen in eine teure IT-Infrastruktur und ihren Unterhalt nicht leisten können oder wollen.

Ein Szenario für einen PUC Rechendienst und seine Benutzung könnte so aussehen:

> Alice ist eine mittelgrosse Bank, welche an der Börse täglich grössere Volumen handelt. Sie betreibt zur Durchführung der Tagesendverarbeitung der von ihr getätigten Transaktionen ein kleines Rechenzentrum. Die getätigten Käufe und Verkäufe werden gegeneinander aufgerechnet und der Bestand bewertet. Zusätzlich führt Alice, um das Exposure ihrer Anlagen zu berechnen, einige spezielle Berechnungen auf den von ihr gehaltenen Positionen durch. In den entsprechenden Programmen steckt viel eigenes Know-How, das sie nicht einfach preisgeben möchte.
>
> Um die mit dem Betrieb und Unterhalt des Rechenzentrums verbundenen Kosten zu senken beschliesst Alice, die Berechnung der Tagesendverarbeitung zukünftig an den PUC-Anbieter Bob zu delegieren. Bob verfügt über ein Rechenzentrum. Seine Umgebung ist von Viktor, dem Alice vertraut, zertifiziert worden. Deshalb weiss Alice, dass die Vertraulichkeit ihrer Daten und Programme bei Bob durch technische Massnahmen gewährleistet ist.

Das beschriebene Szenario des PUC-Rechendienstes ist in Abbildung 1 dargestellt. Eve und Mallory versuchen, Alices vertrauliche Daten zu lesen oder sogar zu manipulieren. Bob offeriert seine Rechendienste auch an weitere Kunden.

Der Begriff *Utility* Computing leitet sich von den bekannten *Utilities* Wasser, Strom und Telefon her.

PUC ist zu einem guten Teil bereits heutzutage nicht mehr nur Vision. Sun bietet mit der "N1 Utility Computing Software" eine Möglichkeit an, um mehrere Maschinen zu einem grossen, virtuellen Server zu vereinigen und plant, die entsprechende Rechenleistung zu einem Preis von 1 USD pro Stunde pro Prozessor zu verkaufen.

Internet Service Provider (ISP) offerieren Dienste wie Mailboxen und Speicherplatz - oft sogar kostenlos. Auch die Idee, die Berechnung eines Programmes zu delegieren, ist nicht neu und erinnert an die Anfangszeiten der IT mit ihren zentral verwalteten Hostsystemen. Mit den Peer-to-Peer Netzwerken sind aber auch neue Formen der Delegation und Distribution der Programmausführung entwickelt worden - ein

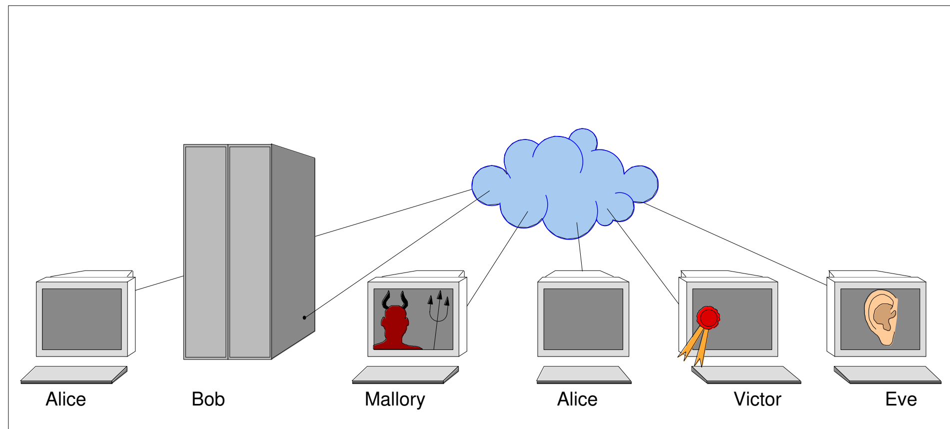


Abbildung 1: PUC Szenario für Rechendienst

bekanntes Beispiel ist das SETI[1] Projekt, wo die Analyse der Radioteleskopdaten auf unzähligen Heimcomputern verteilt durchgeführt wird.

Ein weiteres Beispiel für real existierendes PUC sind Virtual Private Networks (VPN). Die Verantwortung für den Aufbau, die Wartung und den Unterhalt eines standortübergreifenden Unternehmensnetzwerkes wird dabei auf einen oder mehrere ISPs verteilt. Der Anwender nutzt und bezahlt nur die von ihm effektiv benötigte Leistung. Verfügbarkeit und Performanz werden vom Anbieter durch entsprechendes Design des Netzwerks gewährleistet und vertraglich in Service Level Agreements festgehalten. Die Vertraulichkeit und Integrität der Benutzerdaten werden durch kryptographische Verfahren sichergestellt.

Dem PUC verwandte Konzepte sind unter Bezeichnungen wie Computing On Demand, Adaptive Enterprise, Business On Demand oder Grid Computing bekannt.

Das Hauptproblem am Public Utility Computing ist die Exposition der Programme und Daten des Dienstbenutzers an den Dienstanbieter.

Sicherheitsbedenken gehören denn auch zu den von Unternehmen in Studien am häufigsten geäusserten Einwänden gegenüber dem Konzept (Siehe Michael Kallus: Utility Computing in Warteposition [17]). Die Unternehmen fürchten, bei der Delegation einer Berechnung wertvolle Daten oder Algorithmen preiszugeben.

1.2 Ziel der Arbeit

Ziel der vorliegenden Arbeit ist es, das mit der Exposition von Benutzercode und Benutzerdaten an den Dienstanbieter verbundene Sicherheitsproblem für einen PUC Rechendienst zu lösen.

Als Proof of Concept werden für eine Java Laufzeitumgebung Tools zur Programmverschlüsselung erstellt und eine bestehende Java Virtual Machine (VM) so abgeändert, dass diese die verschlüsselten Programme ausführen kann, ohne dabei Programmcode und Daten des Benutzers zu exponieren.

Die Hauptaufgaben sind:

- Spezifikation der Verschlüsselung des Java Bytecodes.

[1] Search for Extraterrestrial Intelligence

- Spezifikation der Modifikationen an der Java VM für die Ausführung des verschlüsselten Bytecodes.
- Implementierung eines Werkzeugs für die Verschlüsselung von Java Bytecode.
- Implementierung der spezifizierten Modifikationen für die Java VM.
- Dokumentation der Arbeit.

1.3 Begriffe

Dienst: Die von einem Anbieter in einem PUC Szenario zur Verfügung gestellte Dienstleistung. In dieser Arbeit bezeichnet Dienst oder PUC-Dienstleistung immer Rechendienstleistung *(Computation)*.

Plattform: Infrastruktur (Hardware, Software) mittels welcher der Dienst eines Anbieters erbracht wird.

Anbieter: Ein an einem PUC Szenario Beteiligter, der einen Dienst - zum Beispiel einen Rechendienst - anbietet und erbringt.

Benutzer: Ein an einem PUC Szenario Beteiligter, der einen Dienst - zum Beispiel einen Rechendienst - in Anspruch nimmt.

Angreifer: Bekannt aus der Kryptographie. Im PUC Szenario ein Beteiligter, der versucht, an nicht für ihn bestimmte Informationen zu gelangen oder die Diensterbringung zu stören, indem er aktiv in das Protokoll eingreift.

Lauscher: Bekannt aus der Kryptographie. Im PUC Szenario ein Beteiligter, der ohne aktiv in das Protokoll einzugreifen versucht, an nicht für ihn bestimmte Informationen zu gelangen.

Benutzerprogramm: Das Programm, das ein Benutzer einem PUC-Anbieter zur Ausführung überträgt.

Benutzerdaten: Die für die Ausführung des Benutzerprogramms notwendigen Eingabedaten und die von diesem produzierten Ausgabedaten.

Protokoll: Ein Protokoll beschreibt eine Folge von Aktionen, welche zur Durchführung einer bestimmten Aufgabe durch zwei oder mehr beteiligte Parteien ausgeführt werden müssen. Ein kryptographisches Protokoll ist ein Protokoll, das kryptographische Mittel einsetzt.

1.4 Notation

`Diese Schrift wird für Pseudocode, Programmlistings und Datenstrukturen verwendet.`

2 Verwandte Arbeiten

2.1 Mobile Computing

Ähnliche Sicherheitsprobleme wie sie im PUC auftauchen sind aus dem Bereich des Mobile Computing seit längerem bekannt. In einem Mobile Computing Szenario bewegt sich ein Programm, ein sogenannter Agent, durch ein Netzwerk von Host zu Host. Sowohl die Frage nach dem Schutz eines Hosts vor einem potentiell gefährlichen Agenten [23, 9], als auch umgekehrt die Frage nach dem Schutz des Agenten und der Geheimnisse, die er mit sich trägt, vor einem potentiell gefährlichen Host wurden untersucht. Diese zweite Fragestellung ist analog zu dem hier aufgeworfenen Problem der Exposition von Benutzercode und Daten in einer PUC Umgebung.

Als Massnahmen gegen diese Gefährdung eines Agenten wurden *Obfuscated Software, Tamper Resistant Software* und *Secure Multi-Party Computation* vorgeschlagen.

2.2 Obfuscated Software

Mit Obfuscated Software wird Programmcode bezeichnet, der schwierig zu disassemblieren ist. Dabei werden Techniken wie Verschleierung des Programmflusses, Verschleierung von Bezeichnern und Verschlüsselung von Konstanten eingesetzt. Der Bytecode einer Java Klasse lässt sich aufgrund seiner klaren und relativ einfachen Struktur einfach analysieren und mittels eines Decompilers in den Originaltext zurückverwandeln[2]. Das hat dazu geführt, dass speziell für Java auch kommerzielle Obfuscators angeboten werden.

Obfuscated Software löst das Code- und Datenexpositionsproblem gegenüber dem PUC Dienstanbieter nicht zufriedenstellend. Die Rekonstruktion des Programmcodes ist nicht unmöglich sondern lediglich aufwändiger. Ausserdem könnte der Dienstanbieter mit zusätzlichen Programmen während der Programmausführung immer noch die Laufzeitdatenstrukturen des ausgeführten Programms beobachten.

2.3 Tamper Resistant Software

Nickerson et al. [19] definieren als Tamper Resistant Software Programmcode, welcher nicht nur schwierig zu disassemblieren ist, sondern auch nicht unbemerkt modifiziert werden kann. Jede unerlaubte Modifikation des manipulationsresistenten Programmcodes soll zu einem Programmabbruch führen, bei welchem keine der im Programmcode enthaltenen Informationen preisgegeben wird. Nickerson et al. beschreiben eine Codiermaschine zur Erzeugung solchen Codes, ohne jedoch auf die Details ihrer Implementierung genauer einzugehen.

Mambo et al. [15] definieren Tamper Resistant Software als Programmcode, der so beschaffen ist, dass es schwierig *(hard)* ist, ihn zu modifizieren oder Informationen, welche in ihm enthalten sind, auszulesen. Sie schlagen vor, Programmcode durch Ersetzungen, Vermischen und Einfügen von Dummy-Code so zu bearbeiten, dass der resultierende Programmcode zwar funktional identisch, aber für einen Angreifer schwierig zu lesen ist. Mit dieser Definition und dem entsprechenden Lösungsansatz bewegen sich Mambo et al. eher im Bereich der Obfuscated Software.

Die existierenden Ansätze für Tamper Resistant Software lösen das Code- und Datenexpositionsproblem beim PUC Dienstanbieter nicht zufriedenstellend. Die Rekonstruktion des Programmcodes ist nicht unmöglich sondern lediglich aufwändiger. Ausserdem könnte der Dienstanbieter mit zusätzlichen Programmen während

[2] Eine ausführliche Behandlung dieses Themas bietet Godfrey Nolan: Decompiling Java. [20]

der Programmausführung immer noch die Laufzeitdatenstrukturen des ausgeführten Programms beobachten.

2.4 Rechnen mit verschlüsselten Funktionen

Sander und Tschudin haben gezeigt, wie Mobile Agents durch eine Erweiterung des Rechnens mit verschlüsselten Daten auf das Rechnen mit verschlüsselten Funktionen vor einem bösartigen Host geschützt werden können. Ihr Szenario einer nicht-interaktiven Berechnung mit verschlüsselten Funktionen ist dem PUC Szenario sehr ähnlich[21]:

> "Alice has an algorithm to compute a function f. Bob has an input x and is willing to compute $f(x)$ for her, but Alice wants Bob to learn nothing substantial about f. Moreover, Bob should not need to interact with Alice during the computation for $f(x)$."

Die von Sander und Tschudin vorgeschlagene Lösung basiert darauf, ein homomorphes[3] Verschlüsselungsschema E auf die Funktion f anzuwenden. Bob berechnet den Wert der transformierten Funktion $g = E(f)$ auf dem transformierten input $y = E(x)$ und schickt Alice das Resultat zurück, welche es entschlüsselt. Sander und Tschudin können ein auf einer Exponentialabbildung basierendes, zumindest additiv homomorphes Verschlüsselungsschema aufzeigen.

Das Rechnen mit verschlüsselten Funktionen ist keine Lösung für das Expositionsproblem beim PUC, weil die Klasse der mittels algebraisch homomorpher Verschlüsselungsschemata sicher berechenbaren Funktionen zur Zeit auf Polynome und rationale Funktionen beschränkt ist. Es können also nicht beliebige Anwenderprogramme auf diese Weise gesichert werden.

2.5 Secure Multi-Party Computation

Als Secure Multi-Party Computation wird ein kryptographisches Protokoll bezeichnet, welches die Berechnung von Funktionen zwischen zwei oder mehr Teilnehmern erlaubt. Das Protokoll ist so beschaffen, dass jede der beteiligten Parteien ausser dem Input, den sie selbst beisteuert und dem Resultat der Berechnung keine anderen Informationen erlangen kann. Ein Anwendungsfall für eine sichere Berechnung mit mehreren Parteien ist zum Beispiel die geheime Abstimmung[4].

Das Rechnen mit verschlüsselten Funktionen kann als Spezialfall der Secure Multi-Party Computation betrachtet werden. Cachin et al. [7] sowie Algesheimer et al. [5] haben Secure Multi-Party Computation Protokolle für den Schutz mobiler Agenten aufgezeigt.

Für das Public Utility Computing sind diese Secure Multi-Party Computation Protokolle nicht geeignet, weil auch hier die Klasse der berechenbaren Funktionen zur Zeit noch beschränkt ist. Es können also nicht beliebige Anwenderprogramme auf diese Weise gesichert werden. Zudem entspricht das Setting mit mehr als zwei beteiligten Parteien nicht dem in dieser Arbeit angenommenen.

[3] Etwas unscharf definiert heisst eine Verschlüsselungsfunktion $E : R \to S$ homomorph, wenn sich aus $E(x)$ und $E(y)$ die Terme $E(x + y)$ (additiv homomorph) und $E(xy)$ (multiplikativ homomorph) einfach berechnen lassen.

[4] Eine Beschreibung von Secure Multi-Party Computation Protokollen findet sich in [22] und [10].

2.6 Tamper Resistant Hardware

Berechnungen, bei welchen die Vertraulichkeit der Daten von grosser Wichtigkeit ist, finden heute allenthalben statt. Ein Geldautomat zum Beispiel sollte die Überprüfung der PIN-Code Eingabe eines Bankkunden auf eine Art und Weise durchführen, die garantiert, dass der Code nicht von einem unbefugten Lauscher mitgelesen werden kann. Insbesondere darf es auch einem Servicemitarbeiter, der den Automaten wartet, nicht möglich sein, die Bestandteile des Automaten welche diese PIN-Code-Überprüfungen durchführen zu manipulieren oder sogar auszutauschen.

Für derartige Anwendungen kommt sogenannte manipulationsresistente Hardware zum Einsatz, wie sie z.B. in kryptographischen Modulen verwendet wird. Kryptographische Module sind zuverlässige Hardwarekomponenten *(Trusted Devices)*, welche oftmals zusätzlich mit einem Schutz vor physischer Manipulation ausgestattet sind. Die Anforderungen an solche Geräte wurden vom National Institute for Software and Technology (NIST) im Federal Information Processing Standard FIPS-140 definiert [18]. FIPS-140 beschreibt 4 Sicherheitsstufen für kryptographische Module, welche vom einfachen Nachweis von Manipulationen bis zu aktiver Ausnullung sensitiver Daten bei Feststellung einer Manipulationsversuchs reichen.

Hardware, welche die in FIPS-140 -1 und -2 spezifizierten Anforderungen ganz oder teilweise erfüllt, wurde von verschiedensten Herstellern entwickelt. Als Beispiel seien hier einzig die kryptographischen Coprocessoren von IBM aufgeführt[6, 11, 25, 24, 26, 1].

Der Einsatz von verlässlicher, manipulationsresistenter Hardware wird von manchen als einzige Lösung für den Schutz eines Benutzerprogramms gegenüber einem böswilligen Host angesehen[5].

[5] Vgl. Bennet S. Yee: A Sanctuary for Mobile Agents. [27].

3 Konzept und Design

3.1 Ablauf Benutzung PUC Rechendienst

Um den Rahmen der Arbeit abzustecken und zu möglichen Angriffsszenarien zu gelangen wird der Ablauf der Benutzung eines PUC Rechendienstes in dieser Arbeit zunächst in die in Abbildung 2 gezeigten Schritte unterteilt.

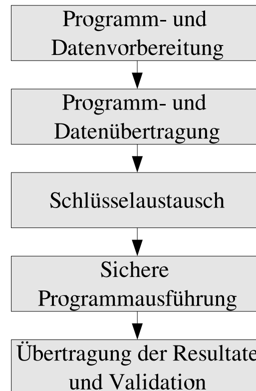


Abbildung 2: Ablauf PUC Dienstbenutzung

Programm- und Datenvorbereitung: Die Programme und Daten müssen vom Dienstbenutzer für die Übertragung und die Ausführung auf der Plattform des Dienstanbieters vorbereitet werden. Das Programm muss getestet und auf der Plattform des Anbieters lauffähig sein. Die Programme und Daten müssen vor Angriffen auf Vertraulichkeit und Integrität geschützt werden.

Programm- und Datenübertragung: Die Programme und Daten müssen vom Dienstbenutzer zum Dienstanbieter übertragen werden. Eine solche Übertragung kann mittels einer Netzwerkverbindung stattfinden.

Schlüsselaustausch: Dienstbenutzerin und Dienstanbieter müssen einen Weg finden, die Schlüssel für die Programme und Daten auszutauschen. Der Austausch der Schlüssel hat direkt zwischen dem Dienstbenutzer und der PUC Plattform zu erfolgen, ohne dass der Betreiber der Plattform die Schlüssel kennt.

Sichere Programmausführung: Die Programme und Daten sollen auf der Plattform sicher, das heisst unter Wahrung von Integrität und Vertraulichkeit, ausgeführt werden.

Übertragung der Resultate und Validation: Am Ende der Programmausführung muss die Programmausgabe zum Dienstbenutzer übertragen werden.

Diese Arbeit befasst sich ausschliesslich mit den Schritten Programm- und Datenvorbereitung und sichere Programmausführung. Nicht behandelt wird das Problem der Authentizität der übermittelten Programme, Eingabedaten und Ausgabedaten.

3.2 Angriffsszenarien

Angriffe auf eine PUC Umgebung können analog zu Angriffen auf einen verschlüsselten Kommunikationskanal in *passive* und *aktive Angriffe* unterteilt werden.

3.2.1 Passive Angriffe

In der Kryptanalyse ist von einem passiven oder Lausch-Angriff auf eine verschlüsselte Kommunikation die Rede, wenn der Angreifer die Kommunikation selbst nicht beeinflusst. Der passive Angriff gefährdet also weder die Übermittlung der Daten an sich, noch deren Integrität, sondern lediglich ihre Vertraulichkeit. Die Lauscherin, Eve, versucht durch reines Abhorchen der Kommunikationskanäle den Verschlüsselungsalgorithmus ganz oder teilweise zu brechen und den Klartext der zwischen den Kommunikationspartnern Alice und Bob übermittelten Nachrichten zu reproduzieren. Im schlimmsten Fall ermittelt sie den geheimen Schlüssel und ist somit in der Lage, die gesamte Kommunikation zu verstehen.

Die Kryptanalyse unterscheidet vier Arten von passiven Angriffen[22]:

1. Ciphertext-only Angriff
   Der Chiffretext mehrerer Nachrichten ist dem Angreifer bekannt.

2. Known-plaintext Angriff
   Chiffre- und Klartext mehrerer Nachrichten sind dem Angreifer bekannt.

3. Chosen-plaintext Angriff
   Der Angreifer kann sich den Chiffretext zu mehreren Klartextblöcken generieren lassen.

4. Chosen-ciphertext Angriff
   Der Angreifer kann sich den Plaintext zu mehreren Chiffretextblöcken generieren lassen.

Bei Chosen-plaintext und Chosen-ciphertext Angriffen werden jeweils noch die adaptiven Varianten unterschieden, bei welchen der Angreifer seine Auswahl variieren kann, indem er die Ergebnisse der vorangehenden Verschlüsselung respektive Entschlüsselung berücksichtigt.

Ein erfolgreicher passiver Angriff wird unter Umständen von den Kommunikationspartnern nicht einmal bemerkt.

In einer PUC Umgebung richtet sich ein passiver Angriff gegen Programmcode und Daten der Dienstbenutzerin. Ein solcher Angriff kann schon bei der Benutzerin, während der Programm- und Datenübertragung oder erst während der Programmausführung beim Dienstanbieter stattfinden. In dieser Arbeit wird der Begriff des passiven Angriffs für einen Angriff, bei welchem nicht nur die Kommunikation sondern auch die PUC-Plattform selbst beobachtet wird, gebraucht.

Für die sichere Ausführung werden Programme und Daten verschlüsselt. Sofern ausreichend starke kryptographische Verfahren benützt werden, sind damit die Programme und Daten während der Übermittlung ausreichend geschützt. Für die Gewährleistung der Integrität der Übertragung, der Authentizität der Daten und des Kommunikationspartners sowie der Nicht-Abstreitbarkeit von Herkunft und Erhalt

können zusätzlich gesicherte[6] Verbindungen, Signaturen und Message Authentication Codes verwendet werden.

Zwei mögliche Angriffe auf die Vertraulichkeit der Programme und Daten sind:

1. Analyse der Bytecodestruktur
   Denkbar ist ein Angriff, welcher auf der Analyse der Bytecodestruktur basiert. Aus der Kenntnis der Bytecodestruktur eines Java Classfiles können gewisse Teile eines verschlüsselten Classfiles (magic number, major version, minor version) erraten werden und erlauben so einen Known-plaintext Angriff. Die Kenntnis der Signatur einer Methode gibt Aufschlüsse über ihr Code Attribut (max_locals, return instruction) und erlaubt ebenfalls einen Known-plaintext Angriff.

2. Beobachtung der Laufzeitdaten während der Programmausführung
   Die grösste Exposition erfahren Programmcode und Daten während der Programmausführung auf der Plattform des Dienstbenutzers. Falls es einer Lauscherin gelingt, mittels eines Trojaners, durch Instrumentierung von Systemsoftware oder mit anderen Mitteln den Arbeitsspeicher der PUC Umgebung auszulesen, hätte sie schon gewonnen. Spätestens für die Ausführung der Programme müssen nämlich Code und Daten entschlüsselt werden. Möglichkeiten zur Beobachtung der Laufzeitdaten bieten sich insbesondere auch einem Systemadministrator beim Dienstleister mit Root-Berechtigung.

Der zweite Angriff widerspiegelt noch einmal das fundamentale Problem der Exposition von Programmen und Daten beim Public Utility Computing. Ein solcher Angriff kann nur durch die Annahme eines Moduls, in welchem die Berechnungen von der Aussenwelt abgeschirmt ablaufen können, entschärft werden.

3.2.2 Aktive Angriffe

Als aktiver Angriff auf eine verschlüsselte Kommunikation wird in der Kryptanalyse ein Angriff bezeichnet, bei welchem der Angreifer, Mallory, die Möglichkeit hat, die Übertragung der Daten zu beeinflussen. Dies macht den aktiven Angreifer mächtiger als den passiven.

In einer PUC Umgebung kann ein aktiver Angriff auf Programme und Daten nicht nur während der Übertragung sondern auch während der Ausführung stattfinden. Die grösste Angriffsfläche für einen aktiven Angriff bietet sich also wieder beim Dienstanbieter selbst. Durch einen aktiven Angriff sind die Vertraulichkeit und Integrität von Benutzerprogramm und Benutzerdaten sowie die korrekte Ausführung des Benutzerprogramms überhaupt gefährdet. In dieser Arbeit wird der Begriff des aktiven Angriffs für einen Angriff, bei welchem neben der Kommunikation auch die Datenstrukturen der Ausführungsumgebung selbst modifiziert werden, gebraucht.

Im folgenden sei Alice Dienstbenutzerin, Bob Dienstanbieter.

Einige mögliche Angriffe sind:

1. Impersonation.
   Ein Angreifer der die Verbindung zwischen Alice und Bob kontrolliert, z.B. ein Mitarbeiter von Bob, schafft es, gegenüber Alice die Rolle von Bobs Plattform

[6] Eine gesicherte Verbindung *(secured channel)* ist, im Gegensatz zu einer sicheren Verbindung *(secure channel),* eine nicht physisch geschützte Verbindung, verunmöglicht aber einem Angreifer dennoch, Information einzufügen, zu löschen, zu lesen, umzuordnen [16].

und gegenüber der Plattform die Rolle von Alice einzunehmen ("Man-in-the-Middle" Angriff). Auf diese Weise erhält er von beiden Seiten alle notwendigen Informationen und kann die gesamte Kommunikation zwischen den beiden nicht nur abhören sondern auch verändern. Die Authentizität der Kommunikationspartner ist verletzt.

2. Austauschen von Teilen des Programms oder der Daten
Ein Angreifer der die Verbindung zwischen Alice und Bob kontrolliert, z.B. ein Mitarbeiter von Bob, kann Teile des Benutzerprogramms oder das gesamte Benutzerprogramm durch seine eigenen Programme auszutauschen. Die Integrität der Programme und Daten ist verletzt. Im einfachsten Fall erzeugt er gezielt oder willkürlich Fehlfunktionen. Ein versierterer Angriff bestünde darin, eigenen Code einzuschleusen, welcher während der Programmausführung Informationen sammelt. Falls der Angreifer das gesamte Programm durch ein eigenes Programm austauscht, erfährt er zwar damit nichts über Alices Geheimnisse, immerhin würde er aber die benutzte Rechenzeit von Alice bezahlt bekommen.

3. Replay Attack
Ein Angreifer der die Verbindung zwischen Alice und Bob kontrolliert, z.B. ein Mitarbeiter von Bob, setzt das Programm mehrmals zur Ausführung an. Die Authentizität ist verletzt. Dadurch schadet er Alice, die für die zusätzliche Rechenzeit bezahlen muss.

4. Überschreiben von Arbeitsspeicherbereichen.
Ein Angriff auf die korrekte Programmausführung besteht darin, während der Programmausführung willkürlich oder gezielt Bereiche des Arbeitsspeichers der PUC Plattform zu überschreiben. Die Änderung eines einzigen Bits im Arbeitsspeicher der Ausführungsumgebung kann zu einem Programmabbruch führen. Ein Bug in der Systemsoftware der PUC Umgebung kann dieser Kategorie Angriffe zugerechnet werden, ebenso Umwelteinflüsse wie Stromausfälle, Feuchtigkeit usw. welche das zuverlässige Funktionieren der Plattform gefährden.

5. Modifikation der Systemsoftware.
Auch bei Einsatz manipulationsresistenter Hardware, wie im Abschnitt über passive Angriffe beschrieben, bieten sich einem aktiven Angreifer immer noch viele Möglichkeiten. Bob könnte die Systemsoftware der Plattform so abändern, dass sie während der Programmausführung laufend Daten sammelt. Das Einschleusen eines solchen Trojaners kann also auch beim Einsatz von manipulationsresistenter nicht ausgeschlossen werden.

3.3 Systemanforderungen

Im folgenden werden die Anforderungen, die an eine PUC Plattform gestellt werden müssen, definiert. Sie sind in dem Sinne maximal, als sie die mit ihrer Realisierung verbundenen Kosten nicht berücksichtigen. Vor einer Realisierung ist eine genaue Analyse der Kosten einer Massnahme im Verhältnis zu ihrem Nutzen respektive zum möglichen Schaden bei Unterlassung der Umsetzung notwendig.

3.3.1 Sicherheitsanforderungen

Oberste Priorität in der Liste der Anforderungen nimmt die Sicherheit ein. Die PUC Plattform soll ein maximales Mass an Sicherheit gegenüber physischen und

logischen Angriffen bieten. Der vom Benutzer gelieferte Code sowie seine Daten dürfen unter keinen Umständen in die Hände eines Angreifers gelangen oder von einem solchen unbemerkt modifiziert werden. Die in diesem Zusammenhang erhobenen Forderungen heissen:

- **Vertraulichkeit:** Die Vertraulichkeit der Benutzerprogramme und Benutzerdaten muss zu jedem Zeitpunkt gewährleistet sein.
- **Integrität:** Die Integrität der Benutzerprogramme und Benutzerdaten muss zu jedem Zeitpunkt gewährleistet sein.

Diese allgemeinen Anforderungen lassen sich spezifischer in Bezug zu den in Abschnitt 3.2 geschilderten Angriffsszenarien setzen.

Vertraulichkeit
Zur Wahrung der Vertraulichkeit wird in dieser Arbeit folgender Ansatz vorgeschlagen:

Der Benutzer kann sensitive Teile seines Programmcodes markieren und verschlüsseln. Die entsprechenden Programmteile werden erst während der Programmausführung entschlüsselt. Diese Programmteile sollen durch Kryptanalyse weder vor noch während der Programmausführung rekonstruiert werden können. Die Ausführung des entschlüsselten Codes findet in einer geschützten Umgebung statt.

Der Benutzer kann die Eingabedaten des Programms verschlüsseln. Er kann die Schlüssel, die sein Programm zur Entschlüsselung der Daten benötigt, dem Programm sicher übermitteln und mit denselben oder weiteren Schlüsseln auch die Ausgabe des Programms verschlüsseln. Der Benutzer kann die Programmteile, welche die Ver- und Entschlüsselung seiner Daten bewerkstelligen, markieren. Die Ausführung dieser Programmteile findet so statt, dass weder die Schlüssel noch die Daten selbst unverschlüsselt beobachtet werden können.

Es müssen bewährte kryptographische Verfahren in sicheren Implementierungen verwendet werden. Die Verfahren und ihre Implementierungen müssen ein hohes Mass an Resistenz gegenüber Angriffen bewiesen haben.

Integrität
Zur Wahrung der Integrität wird in dieser Arbeit folgender Ansatz vorgeschlagen:

Die Plattform überprüft sowohl Programme als auch Daten des Benutzers vor der Ausführung auf ihre Integrität. Damit können Angriffe durch Manipulation der Programme und Eingabedaten abgewehrt werden. Die Integrität der Programme und Daten garantiert die korrekte Programmausführung. Ebenfalls werden auf diese Weise Integritätsverletzungen der Systemsoftware festgestellt.

Während der Programmausführung stellt die Plattform sicher, dass keine Manipulationen an ihren Laufzeitdatenstrukturen stattfinden können. Die Plattform muss Integritätsverletzungen ihres Arbeitsspeichers feststellen können. Falls die gleichzeitige Ausführung mehrerer Benutzerprogramme möglich ist, müssen die verschiedenen Benutzerprozesse strikt voneinander getrennt sein.

Die Plattform muss in der Lage sein, auf Einflüsse zu reagieren, welche ihr zuverlässiges Funktionieren beeinflussen können, wie Umwelteinflüsse oder physische Manipulationsversuche. Die verwendeten Hardwarekomponenten müssen ein hohes Mass an Resistenz gegenüber physischer Manipulation aufweisen.

Zusätzlich muss die Forderung nach Sicherstellung Authentizität der Programme und Daten erhoben werden. Dieses Problem bewegt sich aber ausserhalb des hier behandelten Szenarios. Auf Massnahmen zur Authentizitätssicherung wird deshalb hier nicht eingegangen.

Die skizzierten Ansätze werden in den folgenden Kapiteln detaillierter ausformuliert. Die Ausführung eines Programms in einer Umgebung, welche alle obengenannten Anforderungen erfüllt wird im weiteren *sichere Programmausführung* genannt. Ein Programm, welches für eine sichere Programmausführung vorbereitet wurde, wird *sicheres Programm* genannt.

3.3.2 Allgemeine Entwicklungsziele

Für die zu entwickelnden Werkzeuge sowie die PUC Plattform sollen folgende zusätzlichen Entwicklungsziele mit absteigender Priorität verfolgt werden:

- **Standardtreue:** Die notwendigen Änderungen an der Java Virtual Machine sollen möglichst minimal sein. Die notwendigen Modifikationen an Java Source- und/oder Classfiles (Tagging für die Verschlüsselung) sollen so sein, dass keine Sprachspezifikationen verletzt werden. Der Versuch, ein sicheres Programm auf einer nicht modifizierten Java VM auszuführen soll eine klar definierte Fehlermeldung, welche auf die Verschlüsselung des Programmes hinweist, generieren.
- **Allgemeinheit:** Es sollen keine unnötigen Einschränkungen bezüglich der ausführbaren Programme und Methoden gelten.
- **Flexibilität und Modularität:** Idealerweise sollte für jeden Teil eines Programmes, von der kleinsten Einheit bis zum ganzen Programm, Grad und Art der Verschlüsselung gewählt werden können.
- **Performanz:** Die Modifikationen an der Java Maschine sollen die Ausführungsgeschwindigkeit der VM möglichst nicht beeinträchtigen.
- **Bedienungsfreundlichkeit:** Die Werkzeuge sollen einfach zu benutzen sein. Das Verschlüsseln und Signieren von Code durch den Benutzer soll mittels weniger Befehle möglich sein. Das Laden und Ausführen von Benutzerprogrammen auf der Plattform des Dienstanbieters soll möglichst einfach zu bewerkstelligen sein.

Im Rahmen des Proof of Concept werden diese Anforderungen nur bedingt berücksichtigt.

3.4 Voraussetzungen

In diesem Kapitel werden die Annahmen, von welchen für Spezifikation und Implementierung ausgegangen wird, und die Anforderungen welche mit diesen Annahmen abgedeckt werden, beschrieben. Die Realisierung der skizzierten Hardwarearchitektur ist nicht Teil dieser Arbeit, muss aber als Grundlage für die präsentierte Lösung Erwähnung finden.

3.4.1 Hardwarearchitektur

Die im Abschnitt 2 beschriebenen, heute bekannten Verfahren und Protokolle alleine können die an ein PUC Szenario gestellten Sicherheitsanforderungen nicht erfüllen oder decken nicht wirklich die gestellten Anforderungen ab. Die hier präsentierte Lösung für das PUC beruht unter anderem auf dem Einsatz von Tamper Resistant Hardware.

Kryptographische Module
Die Sicherheitsanforderungen welche in Abschnitt 3.3.1 an eine PUC Plattform erhoben wurden, sind vergleichbar mit den Anforderungen, die vom NIST Standard FIPS 140 an kryptographische Module gestellt werden. Als Modell für die Realisierung einer PUC Plattform kann deshalb die Architektur eines kryptographischen Moduls, z.B. eines kryptographischen Coprocessors, dienen.

Ein solches kryptographisches Modul ist im allgemeinen ein mehr oder weniger komplexer, auf einem separaten Gerät untergebrachter Computer für kryptographische Berechnungen, der mit seinen Umsystemen nur über wenige klar definierte logische und physische Schnittstellen[7] kommuniziert.

Kryptographische Module werden oft vor physischen Angriffen durch Sensoren und Schaltkreise geschützt. Damit sollen die Manipulation von Modulkomponenten oder gar der Austausch des ganzen Moduls verhindert werden. Die Sensoren sind in der Lage, extreme Änderungen der Umweltbedingungen, welche die zuverlässige Funktionsweise des Moduls gefährden könnten, festzustellen und darauf zu reagieren. Die Komponenten des Moduls sind vor elektromagnetischen Interferenzen ausreichend geschützt.

Für eine effiziente Durchführung der kryptographischen Berechnungen sind die entsprechenden Algorithmen kryptographischer Module oftmals in Hardware realisiert.

Beim Initialisieren durchläuft das Modul eine Reihe von Selbsttests um Integritätsverletzungen an Hard- oder Sofwarekomponenten feststellen zu können. Falls beim Initialisieren Systemsoftware über die Verbindung zu den Umsystemen geladen werden muss, wird deren Integrität geprüft.

Smith et al. [25, 24, 26, 1] sowie Arnold et al. [6] beschreiben in ihren Arbeiten detailliert Aufbau und Funktionalität von kryptographischen Coprozessoren.

PUC Plattform Für den weiteren Verlauf dieser Arbeit wird für die PUC Plattform die in der in Abbildung 3 schematisch dargestellte Architektur angenommen. Die Java VM wird als *Blackbox mit beschränkten Ressourcen* betrachtet, welche Java Programme zuverlässig ausführen kann.

Die Java VM initialisiert sich und läuft vollständig innerhalb eines manipulationsresistenten Moduls, welches nur über einen einzigen Bus mit seiner Umgebung, in der Abbildung als Host bezeichnet, kommuniziert[8]. Dieses im folgenden PUC-Modul

[7] Vgl. FIPS 140-2, Kapitel 4.2: Cryptographic Module Ports and Interfaces [18].

[8] Die Auslagerung der Runtime ist sowohl als VM, die z.B. als Applikation auf einem sicheren Coprocessor läuft, als auch als Hardware Implementierung einer Java Runtime denkbar. Im ersten Fall läuft der VM Prozess vollständig innerhalb des sicheren Moduls. Der Vorteil einer Virtuellen Maschine ist klar ihre Adaptibilität an veränderte Spezifikationen sowie die Möglichkeit, Patches für Fehler einspielen zu können. Für eine Realisierung könnten VM Executable, Klassenbibliotheken und Cryptosupport entweder in Hardware gegossen, im sicheren persistenten Speicher des PUC-Moduls gespeichert oder von einem Loader auf dem PUC-Modul unter Überprüfung ihrer Integrität über die Verbindung zum Host geladen werden.

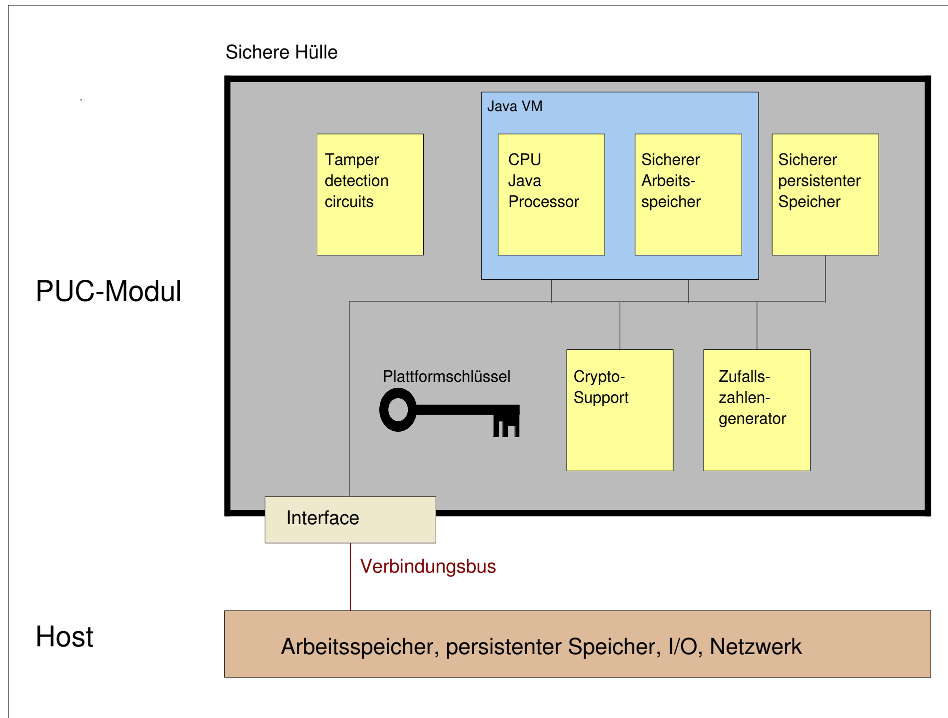


Abbildung 3: Java Runtime Plattform als sicheres Modul

genannte Gerät verfügt über einen eigenen Arbeitsspeicherbereich und über einen persistenten Speicher für die Speicherung des Boostrapcodes und der Systemsoftware.

Über den Bus hat die Java VM Zugriff auf Festspeicher und Arbeitsspeicher des Hostsystems.

Als logische Schnittstelle kennt das Modul nur einen Befehl, der die Ausführung eines Java Programmes, das auf dem persistenten Speicher des Hostsystems abgelegt ist, startet, einen Befehl, der diese Programmausführung abbricht, einen Befehl, der das Modul in den Originalzustand zurücksetzt, sowie die vom Controller gesendeten Signale für die Kommunikation zwischen Host und Modul. Alle anderen, nicht definierten Anfragen über diesen Kanal führen augenblicklich zum Abbruch der laufenden Programmverarbeitung.

Das Modul verfügt über einen privaten Schlüssel *(Plattformschlüssel)*, mit dem es Daten, welche mit dem zugehörigen öffentlichen Schlüssel verschlüsselt worden sind, entschlüsseln kann.

Zusätzlich können im Modul ein Generator für Zufallszahlen und eine Hardwareunterstützung für kryptographische Algorithmen untergebracht sein.

Speicherbereiche Für die folgenden Betrachtungen interessiert lediglich der Aspekt der beschränkten Speicherressourcen des PUC-Moduls.

Es soll deshalb vereinfachend die Existenz von 4 Speicherbereichen mit den in Tabelle 1 beschriebenen Eigenschaften angenommen werden. Die Java VM hat Zugriff auf alle 4 Speicherbereiche. In den als *sicher* bezeichneten Bereichen sind die Benutzerprogramme und Benutzerdaten vor physischen und logischen Angriffen auf

Tabelle 1: Speicherbereiche

<table>
<tr><td><u>ungeschützter Arbeitsspeicher</u>
(Host)
• ungeschützt
• Grösse unbeschränkt
• Daten sind verschlüsselt</td><td><u>sicherer Arbeitsspeicher</u>
(Modul)
• sicher
• Grösse beschränkt
• Daten sind unverschlüsselt</td></tr>
<tr><td><u>ungeschützter persistenter Speicher</u>
(Host)
• ungeschützt
• Grösse unbeschränkt
• Daten sind verschlüsselt</td><td><u>sicherer persistenter Speicher</u>
(Modul)
• sicher
• Grösse beschränkt
• Daten sind unverschlüsselt</td></tr>
</table>

Integrität und Vertraulichkeit geschützt. In den als *ungeschützt* bezeichneten Bereichen sind müssen Benutzerprogramme und Benutzerdaten durch Verschlüsselung vor Angriffen geschützt werden.

Die physische Sicherheit wird durch die Manipulationsresistenz des Moduls, auf welchem diese Speicherbereiche untergebracht sind, gewährleistet.

Für die Gewährleistung der logischen Sicherheit werden folgende Annahmen getroffen:

1. Das Modul reagiert nur auf die in Abschnitt 3.4.1 beschriebenen Befehle.
2. Ausser dem Programm eines einzelnen Dienstbenutzers werden gleichzeitig keine anderen Programme auf dem Modul ausgeführt.

Die physisch sicheren Speicherbereich sind beschränkt und auf jeden Fall kleiner als die ungeschützten.

3.4.2 Initialisierung und Zertifizierung

Um gegenüber einem Benutzer garantieren zu können, dass seine Plattform alle obengeannten Voraussetzungen erfüllt, muss ein Anbieter diese Plattform notwendigerweise von einer vertrauenswürdigen Drittpartei zertifizieren lassen. Es wird die Existenz einer Drittpartei angenommen, der sämtliche Beteiligten funktional[9] vertrauen.

Die Schlüsselgenerierung für die Plattform hat unter Aufsicht einer Zertifizierungsinstanz so zu erfolgen, dass

[9] [16]: D.h. es wird angenommen, dass die Drittpartei ehrlich und fair ist, aber dass sie nicht die privaten Schlüssel der Beteiligten kennt. (*functionally trusted* im Gegensatz zu *unconditionally trusted*)

- der generierte private Schlüssel nur der Plattform selbst bekannt ist und von niemandem während oder nach der Generierung ausgelesen werden kann,
- der öffentliche Schlüssel der Plattform als wirklich zum Private Key der Plattform gehörend eindeutig identifiziert und zertifiziert werden kann.

3.4.3 Datenfluss

Eine Java VM Implementierung verwaltet zur Laufzeit die in Tabelle 2 dargestellten Datenstrukturen.

Tabelle 2: Datenstrukturen einer Java VM

Name	**Inhalt**	**Gültigkeitsbereich**
Heap	Instanzen von Klassen, Arrays *(Private Memory)*	VM (alle Threads)
Method Area	Runtime Constant Pool *(Shared Memory)*, Methodencode	VM (alle Threads)
Stack	Frames	Thread
PC	Programmzähler	Thread
Runtime Constant Pool	Konstanten, Feldreferenzen (Symboltabelle)	Klasse
Frame	Lokale Variable (Array), Operand Stack, Referenzen auf Runtime Constant Pool der Klasse	Methode
Lokale Variable	Lokale Variable eines Methodenaufrufs. Werden für die Parameterübergabe benutzt.	Frame
Operand Stack	LIFO Speicher für Operanden von Instruktionen	Frame

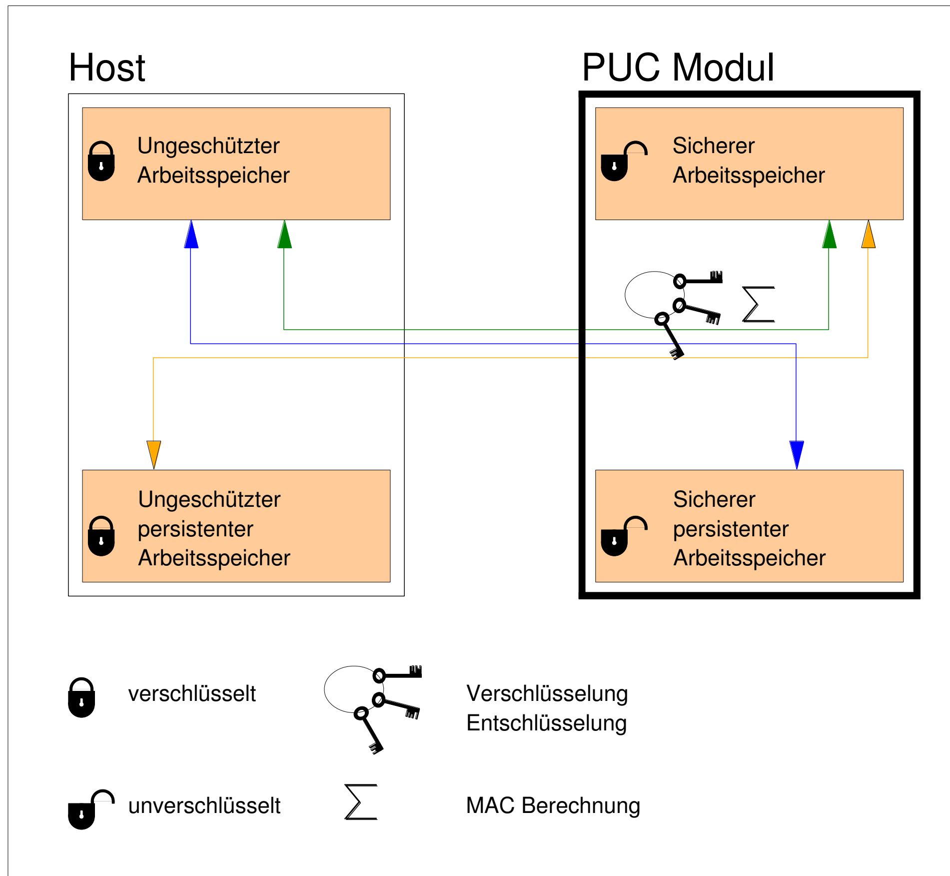
Bei Ausführungsstart reklamiert die VM vom Betriebssystem einen Speicherbereich, den sogenannten *Heap,* den sie selbst verwaltet. Der Heap wird bei Programmstart unterteilt in die *Method Area* und einen Bereich für Objekte und Arrays. In der Method Area legt die VM Methodencode und Symboltabelle *(Runtime Constant Pool)* einer initialisierten Klasse ab. Während der Programmausführung erzeugte Objekte und Arrays werden im restlichen Heap abgelegt. Die Speicherallozierung dieser Objekte besorgt einzig die VM selbst - ein Java Programm kann nicht direkt auf Speicherbereiche zugreifen.

Jeder VM Thread verfügt über einen eigenen *Stack* und einen eigenen *Programmzähler*. Auf dem Stack legt die VM die *Frames* ab, welche den Array der *Lokalen Variablen* und den *Operand Stack* eines Methodenaufrufs beinhalten.

Abbildung 4 stellt für die PUC Plattform den Datenfluss zwischen den in Abschnit 3.4.1 definierten Speicherbereichen dar.

Bei Start der Programmausführung liegen Benutzerdaten und -programme in verschlüsselter Form im sicheren persistenten Speicher des Hosts. Das PUC Modul prüft sie beim Einlesen auf Integrität.

Abbildung 4: Datenfluss



Während der Ausführung bildet die Java VM Klassen und Daten auf ihre eigenen Datenstrukturen ab. Wegen der beschränkten Grösse des sicheren Arbeitsspeichers können nicht alle während der Laufzeit von der VM erzeugten Datenstrukturen im sicheren Arbeitsspeicher Platz finden. Die Daten werden vom und zum sicheren Arbeitsspeicher entschlüsselt und verschlüsselt, um ihre Vertraulichkeit zu wahren.

Um die Integrität der auf den ungeschützten Arbeitsspeicher ausgelagerten Datenstrukturen zu überprüfen und zu gewährleisten berechnet das PUC Modul schlüsselabhängige Hashwerte (MAC) für die ausgelagerten Daten. Diese werden bei einem Lesezugriff überprüft.

3.5 Designentscheidungen

3.5.1 Granularität der Verschlüsselungseinheiten

Der bis hierhin skizzierte Lösungsansatz beruht auf manipulationsresistenter Hardware, Verschlüsselung und Hashwertberechnung. Als erstes muss die Frage nach der Granularität der Verschlüsselung für die Java Programme beantwortet werden, also die Frage, welche Teile eines compilierten Java Programmes verschlüsselt werden sollen.

Tabelle 3: Massnahmen gegen Angriffe

Angriff	**Massnahme**
Analyse der Bytecodestruktur	Verschlüsselung
Beobachtung der Laufzeitdatenstrukturen	Manipulationsresistente Hardware und Verschlüsselung
Impersonation	- (Angriff auf Authentizität)
Austauschen von Programmteilen	Verschlüsselung und Einweg-Hashwerte
Replay Attack	-
Überschreiben von Arbeitsspeicherbereichen	Manipulationsresistente Hardware und MACs
Modifikation der Systemsoftware und Trojaner	Manipulationsresistente Hardware, MACs undAusführung nur eines Benutzerprogramms zur selben Zeit

Als Verschlüsselungseinheit für die Programme bietet sich zunächst das Java-Programmarchiv (.jar File) als ganzes an. Diese radikale Lösung besticht im ersten Augenblick durch ihre Einfachheit; allerdings werden dadurch die Anforderungen *Flexibilität* und *Standardnähe* nicht zufriedenstellend abgedeckt. Ein so verschlüsseltes Programm würde auf einer Standard VM nicht zu der erwünschten sauberen Fehlermeldung führen. Dasselbe gilt für die Verschlüsselung ganzer Classfiles.

Ein weiterer Aspekt ist der Aufwand, der für die Verschlüsselung getrieben werden muss. Falls das Programm des Dienstbenutzers sehr umfangreich ist, die Anzahl und der Umfang der sensitiven Methoden aber klein, so macht es keinen Sinn, mehr Daten als notwendig zu verschlüsseln. Die Ausführung sicherer Methoden wird aufgrund des notwendigen zusätzlichen Entschlüsselungsschrittes auf jeden Fall langsamer sein als die Ausführung nicht verschlüsselter Methoden.

Die Frage lässt sich am Besten aus der Komplexität, die sich für die Programmentwicklung ergibt, beantworten. Vom Standpunkt eines Entwicklers aus wäre es wünschenswert, beliebige Abschnitte (Blöcke) innerhalb des Codes als geschützt markieren zu können, analog zu *synchronisierten* Blöcken oder Transaktionen bei Datenbankabfragen. Objekte und Daten, die innerhalb eines sicheren Blocks kreiert und benutzt werden dürfen ebensowenig exponiert werden, wie der Methodencode dieses Blocks.

Für das Proof of Concept wird als Verschlüsselungseinheit die einzelne Methode gewählt. Damit kann dasselbe Mass an Flexibilität gewährleistet werden wie mit beliebigen Blöcken. Für Alice ergibt sich mit diesem Ansatz die Möglichkeit, die durch die Entschlüsselung notwendige Verlängerung der Laufzeit ihres Programms auf der Plattform des Dienstanbieters - und damit ihre Kosten - auf ein Minimum zu beschränken.

3.6 Zusammenfassung der Abwehrmassnahmen

Mit dem gewählten Lösungsansatz wird den in Abschnitt 3.2 geschilderten Angriffen mit den in Tabelle 3 aufgezeigten Massnahmen begegnet.

- Angriff durch Analyse der Bytecodestruktur

4 Spezifikation

In diesem Kapitel werden ein Werkzeug zur Programm- und Datenvorbereitung sowie die notwendigen Modifikationen an der Java Virtual Machine spezifiziert für die Ausführung verschlüsselter Java Klassen spezifiziert. Dabei wird von den in Abschnitt 3.4 getroffenen Voraussetzungen und Designentscheiden ausgegangen.

Eine ganz oder teilweise verschlüsselte Klasse als *sichere Klasse,* eine ganz oder teilweise verschlüsselte Methode als *sichere Methode* bezeichnet. Das Werkzeug zur Programm- und Datenvorbereitung wird vereinfachend *Verschlüsselungswerkzeug* genannt.

4.1 Programm- und Datenvorbereitung

4.1.1 Ablauf Programmvorbereitung

Tabelle 4 stellt die für die Programm- und Datenvorbereitung auszuführenden Schritte dar.

4.1.2 Verschlüsselung der Programmeinheiten

Verschlüsselungsalgorithmus
Für die Verschlüsselung der Programmeinheiten wird ein symmetrischer Verschlüsselungsalgorithmus verwendet. Symmetrische Algorithmen sind im allgemeinen wesentlich schneller als asymmetrische Algorithmen. Viele symmetrische Algorithmen sind so ausgelegt, dass sie einfach und effizient in Hardware realisiert werden können.

Als Verschlüsselungsalgorithmus für die Programmeinheiten wird für das Proof of Concept DES im CBC Modus mit Schlüssellänge $L = 8$ gewählt. Der Initialisierungsvektor IV hat den festen Hexadezimalwert $0xcafebabe$.

Markieren und Compilieren sicherer Klassen
Eine Methode, deren Code verschlüsselt werden soll, wird im Sourcetext mit dem Präfix `secure_` markiert. Die Methode `transferMoney()` muss also in `secure_transferMoney()` umbenannt werden.

Ein Java Sourcefile mit als sicher bezeichneten Methoden wird zunächst wie gewohnt compiliert. Die Verschlüsselung der sicheren Methoden durch das Verschlüsselungswerkzeug erfolgt erst nach dem Compilieren, direkt auf dem Bytecode. Der Name der Methode wird vom Verschlüsselungswerkzeug nicht verändert.

Bytecodestruktur sicherer Klassen
Für den verschlüsselten Code einer sicheren Methode wird der Methodenstruktur ein neues Attribut `CryptoCode` angefügt. Der verschlüsselte Methodencode wird in diesem Attribut gespeichert.

Der ursprüngliche Methodentext wird ersetzt durch den Methodentext einer vorgegeben Methode, welche eine konfigurierbare Meldung ausgibt und die Programmausführung abbricht.

Die Erweiterung eines Classfiles um zusätzliche Attribute ist erlaubt. Ebenfalls ist es nicht explizit verboten, die Attribute einer Methode oder gar die Attribute des `Code` Attributes selbst um zusätzliche Attribute zu erweitern. Damit bleibt die Classfile Struktur sicherer Klassen erhalten.

Tabelle 4: Programm- und Datenvorbereitung

Hauptakteur: Dienstbenutzerin Alice

Vorbedingungen: Alice verfügt über ein lauffähiges Java Programm P bestehend aus den Classfiles $C_1 ... C_n$ und den Eingabedateien $D_1 ... D_m$, das die Ausgabedateien $D_n ... D_s$erzeugt.

1. Alice wählt einen Benutzerschlüssel K_{user}.
2. Alice markiert die zu schützenden Methoden der Classfiles $C_1 ... C_n$.
3. Alice generiert die Klassenschlüssel $K_{C_1} ... K_{C_n}$.
4. Alice verschlüsselt die in Schritt 2 markierten Classfiles $C_1 ... C_n$mit $K_{C_1} ... K_{C_n}$. Das Verschlüsselungswerkzeug erzeugt aus dem Methodennamen und dem Klassenschlüssel für jede zu verschlüsselnde Methode einen eigenen Methodenschlüssel und verschlüsselt den Methodentext damit.
   Algorithmus: DES-CBC.
5. Alice generiert die Datenschlüssel $K_{D_1} ... K_{D_m}$.
6. Alice verschlüsselt die Eingabedateien $D_1 ... D_m$mit Schlüsseln $K_{D_1} ... K_{D_m}$.
   Algorithmus: DES-CBC.
7. Alice berechnet Hashwerte $M_{C_1} ... M_{C_n}$für die Classfiles und $M_{D_1} ... M_{D_m}$für die Eingabedateien.
   Algorithmus: MD5.
8. Alice schreibt die Klassenschlüssel $K_{C_1} ... K_{C_n}$, die Datenschlüssel $K_{D_1} ... K_{D_m}$sowie die Hashwerte $M_{C_1} ... M_{C_n}$und $M_{D_1} ... M_{D_m}$in eine Konfigurationsdatei `data.pf`.
9. Alice legt die zusätzlichen von ihrem Programm benötigten Datenschlüssel $K_{D_n} ... K_{D_s}$ in der Konfigurationsdatei `data.pf` ab.
10. Alice verschlüsselt die Konfigurationsdatei mit dem Benutzerschlüssel K_{user}.
    Algorithmus: DES-CBC.
11. Alice verschlüsselt den Benutzerschlüssel K_{user}mit dem öffentlichen Schlüssel der Plattform K_{pub}.
    Algorithmus: RSA.
12. Alice speichert den verschlüsselten Benutzerschlüssel in einer Schlüsseldatei `key.ef`.

Im folgenden werden die Strukturen für `method_info` und `CryptoCode` Attribut definiert. Dabei bezeichnen die Typen `u1`, `u2` und `u4` vorzeichenlose 1, 2 und 4-Byte Grössen.

Die Struktur einer Methode bleibt gegenüber ihrer Struktur im ursprünglichen Classfile unverändert; es erhöht sich lediglich die Anzahl der Methodenattribute um 1 für das neue Attribut `CryptoCode`.

```
method_info {
  u2 access_flags;
  u2 name_index;
  u2 descriptor_index;
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

Die Struktur des Methodenattributes `CryptoCode` entspricht derjenigen des ursprünglichen Methodenattributs `Code`.

```
CryptoCode_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u1 info[attribute_length];
}
```

`attribute_name_index` ist ein gültiger Index in den `ConstantPool` der Klasse. Der Eintrag an dieser Stelle muss eine `UTF8_Info` Struktur, die den String "CryptoCode" repräsentiert, sein. Dieser Eintrag muss vom Verschlüsselungswerkzeug erzeugt werden. `attribute_length` bezeichnet die Länge des verschlüsselten Attributes ohne die ersten 6 Bytes, welche `attribute_name_index` und `attribute_length` enthalten.

Der `info` genannte Teil des Methodenattributs `CryptoCode` enthält den eigentlichen Methodencode, die Fehlerbehandlungstabelle sowie weitere Attribute:

```
info {
  u2 max_stacks;
  u2 max_locals;
  u4 code_length;
  u1 code[code_length];       <-Nur dieser Teil ist verschlüsselt!
  u2 exception_table_length;
  {     u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
  } exception_table[exception_table_length];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

Der eigentliche Methodencode ist im `u1`-array `code` gespeichert. Um die für das Proof of Concept notwendigen Anpassungen an der Java VM zur Ausführung sicherer Methoden in Grenzen zu halten, wird nur dieser Teil des Methodenattributes verschlüsselt.

Die Verschlüsselung des gesamten `info[]`-arrays hätte für die Modifizierung der VM neben dem Anpassen der Funktionen für das Einlesen des Methodencodes umfangreichere Anpassungen für die Allozierung der Frames sowie für das Exception Handling zur Folge.

Das Methodenattribut `CryptoCode` kann wie das Methodenattribut `Code` selbst wieder die Attribute `LineNumberTable` und `LocalVariableTable` enthalten. Die Länge dieser Attribute wird auf Null gesetzt, ihr Inhalt wird entfernt.

Falls ein Verschlüsselungsalgorithmus verwendet wird, bei dem die Länge von Klartext und Chiffretext nicht übereinstimmen, so muss der Wert von `code_length` entsprechend angepasst werden. Das Verschlüsselungswerkzeug ist dafür verantwortlich, dass die Länge des verschlüsselten Attributes nicht die Maximallänge eines Attributes ($2^{32} Bytes$)überschreitet.

Ersetzen ursprüngliches Code Attribut durch vorgegebenen Methodentext

Der ursprüngliche Methodencode soll durch den Code einer vorgegebenen Methode ersetzt werden. Eine Java VM welche die sicheren Methoden nicht als solche erkennen kann wird versuchen, diese Methode auszuführen. Die Methode soll einen Hinweis auf der Konsole ausgeben und die Ausführung abbrechen. Dabei sind folgende Punkte zu beachten.

- Die Signatur der ursprünglichen Methode muss erhalten bleiben. Andernfalls wird bei der Verifikation des Classfiles ein `java.lang.ClassFormatError` generiert.
- Beim Ersetzen der ursprünglichen Methode werden die notwendigen Einträge für den vorgegebenen Methodentext im Constant Pool erzeugt. Andernfalls wird bei der Verifikation des Classfiles ein `java.lang.ClassFormatError` generiert.
- Die für die vorgegebenen Methode notwendigen Einträge im Constant Pool sollen, falls die Methode in einer Klasse mehrmals eingefügt werden muss, nur einmal erzeugt werden. Damit wird der Constant Pool nicht unnötig erweitert.
- Die Werte von `max_stack` und `max_locals` der vorgegebenen Methode sind so anzupassen, dass sie mit der Signatur der ersetzten Methode in Einklang sind. Andernfalls wird bei der Verifikation des Classfiles ein `java.lang.VerifyError` generiert.

Das Verschlüsselungswerkzeug ist dafür verantwortlich, dass weder die maximale Anzahl Einträge im Constant Pool (2^{16}) noch die maximale Anzahl Attribute einer Methode (2^{16}) überschritten wird[10].

4.1.3 Schlüsselgenerierung für Methodenverschlüsselung

Um nicht alle sicheren Methoden mit demselben Schlüssel zu verschlüsseln, wird folgender Mechanismus für die Schlüsselgenerierung verwendet:

$$\begin{array}{rl} Schlüssel\,für\,Klasse\,C: & K_C \\ Schlüssel\,für\,Methode & \\ mit\,Signatur\,M\,in\,Klasse\,C: & K_C^M = Left_L[H(M\|K_C)] \end{array}$$

Der Klassenschlüssel $K_C$wird durch einen Zufallsgenerator erzeugt. Jede Methode wird mit einem eigenen Methodenschlüssel K_C^M verschlüsselt. Dabei ist Heine

[10] Wegen des Hinzufügens der Attribute und Einträge im Constant Pool gelten für zu verschlüsselnde Klassen geringfügig andere Limitationen für die Anzahl möglicher Einträge in diesen ConstantPool als für nicht zu verschlüsselnde. Das dürfte aber für die meisten Applikationen kein Problem darstellen. Siehe Lindholm [14] für die Limitationen der Java VM.

Einweg-Hashfunktion und $\|$ die Konkatenationsfunktion. Der von der Hashfunktion berechnete Wert wird auf die Schlüssellänge L des Verschlüsselungsalgorithmus gekürzt.

Der PUC Plattform müssen nur die Klassenschlüssel K_C übergeben werden. Die Plattform kennt den verwendeten Einweg-Hashalgorithmus und leitet daraus und aus den Methodennamen, auf die sie über den unverschlüsselten Constant-Pool der Klasse Zugriff hat, selbst die Methodenschlüssel ab.

Als Initialisierungsvektor für die Verschlüsselung wird ein konstanter Wert der Länge L verwendet. Der Initialisierungsvektor wird dem verschlüsselten Methodencode vorangestellt. Die modifizierte VM weiss das und beginnt mit der eigentlichen Entschlüsselung des Codes erst nach dem Einlesen des Initialisierungsvektors.

4.1.4 Verschlüsselung Benutzerdaten

Eingabedateien für das Benutzerprogramm müssen vom Benutzer verschlüsselt werden, bevor das Verschlüsselungswerkzeug aufgerufen wird. Die Wahl der Schlüssel, Algorithmen und Verschlüsselungsstärken für die Verschlüsselung einer Eingabedatei bleibt dem Benutzer überlassen. Der Benutzer integriert die Entschlüsselung der Eingabedatei für die Bearbeitung während der Programmausführung in sein Programm. Die entsprechenden Entschlüsselungsmethoden müssen als sichere Methoden deklariert werden.

Die benutzerdefinierten Datenschlüssel können der VM im Konfigurationsfile übergeben werden. Sie werden dem Benutzerprogramm während der Ausführung als Systemeigenschaften zur Verfügung gestellt und können mittels `System.getProperty("userkey")` ausgelesen werden.

4.1.5 Integrität von Programm- und Eingabedateien

Sämtliche Dateien - seien es Java Classfiles, Eingabedateien oder auch Systembibliotheken - müssen beim Laden auf ihre Integrität überprüft werden.

Für die Integritätsprüfung wird die Einweg-Hashfunktion MD5 verwendet. Das Werkzeug für die Programm- und Datenvorbereitung berechnet für jedes im Programmpaket enthaltene Classfile und jede Eingabedatei einen solchen MD5 Hashwert und speichert ihn in der Konfigurationsdatei ab[11].

Die Hashes für verschlüsselte Klassen werden *nach deren Verschlüsselung* berechnet, weil die Entschlüsselung der Methodentexte erst nach dem Laden der Klasse bei deren Ausführung erfolgt.

4.1.6 Verschlüsselungswerkzeug

Das Verschlüsselungswerkzeug

- erzeugt die Klassenschlüssel $K_{C_1} ... K_{C_n}$ für die einzelnen Classfiles des Programms

[11] Eine schlüsselabhängige Generierung von Hashwerten ist nicht notwendig, weil die im nächsten Schritt erfolgende Verschlüsselung der Konfigurationsdatei es einem Angreifer verunmöglicht, Dateien oder Classfiles zu verändern, ihre Hashwerte neu zu berechnen und in der Konfigurationsdatei zu ersetzen.

- erzeugt die Methodenschlüssel K_C^M für die als sicher markierten Methoden aller Classfiles C
- verschlüsselt die als sicher markierten Methoden aller Classfiles (DES-CBC)
- erzeugt für alle Classfiles und Eingabedateien Einweg-Hashwerte (MD5)
- erlaubt die sichere Übergabe zusätzlicher Benutzerdaten (Key-Value Paare, z.B. Datenschlüssel K_{D_j})
- erzeugt die Konfigurationsdatei `data.pf` mit Klassenschlüsseln, Hashwerten und Key-Value Paaren
- verschlüsselt das Konfigurationsfile mit dem Benutzerschlüssel K_{user} (DES-CBC)
- verschlüsselt den symmetrischen Schlüssel mit dem Public Key K_{pub} der Plattform (RSA)
- speichert den verschlüsselten Benutzerschlüssel im File `key.ef`

Die Verschlüsselung und Vorbereitung der markierten Klassen eines Verzeichnisses soll mittels eines einzigen Aufrufs erfolgen. Die Usage des Tools ist:

```
encryptiontool -s SymmetricKeyFile -p PlatformPublicKeyFile
[-v key:value] [-v key:value] [...] file

Verschlüsselt Classfiles und erzeugt MACs für Dateien.
Wenn file ein Verzeichnis ist, werden rekursiv alle Classfiles
unterhalb dieses Verzeichnisses verschlüsselt und die
MACs für alle Dateien unterhalb dieses Verzeichnisses
berechnet. Schlüssel, MACs und zusätzliche Key-Value Paare
werden im Konfigurationsfile "DATA.PF" abgelegt, welches mit
dem symmetrischen Schlüssel aus "SymmetricKeyFile"
verschlüsselt wird.

Optionen
    -s SymmetricKeyFile
       File in welchem der Benutzerschlüssel K_user für
       die symmetrische Verschlüsselung des DATA.PF
       Files liegt.
    -p PlatformPublicKeyFile
       File in welchem der öffentliche Schlüssel K_pub
       der Platform für die Verschlüsselung des
       symmetrischen Schlüssels liegt.
    -v key:value
       Benutzerdefiniertes Schlüssel-Werte-Paar
```

Die generierte Konfigurationsdatei besteht aus je einem Eintrag für jede Datei und einem Eintrag `MetaData` in welchem die zusätzlichen vom Benutzer definierten Key-Value-Paare geschützt übergeben werden können. Die Struktur des Konfigurationsfiles:

```
ClassFile Encryption Configuration
Created by EncryptionTool V.01

Name: secure/secure_class1.class
Class-Key: [encrypted user key for class1]
Hash: [MD5 of secure_class1]

Name: secure/secure_class2.class
Class-Key: [encrypted user key for class2]
...

Name: common/class3.class
...

Name: input/input1.txt
Hash: [MD5 of input1]
...


Name: MetaData
#Any additional, user defined Key-Value pairs
User-Key-1: [encrypted user key for data]
Algorithm: DES-OFB
```

Für jedes verschlüsselte Classfile wird der benutzerdefinierte Klassenschlüssel im Parameter `Class-Key` übergeben.

Für jede Datei wird der mittels des MD5-Algorithmus berechnete Hashwert gespeichert. Dieser Wert wird beim Laden der Datei von der PUC Umgebung überprüft[12]. Die Frage nach der Integrität der data.pf Datei selbst wird in Abschnitt 4.4.4 behandelt.

Der Eintrag MetaData kann für die sichere Übergabe beliebiger zusätzlicher Werte benutzt werden. Für jedes Schlüssel-Werte Paar in MetaData erzeugt die PUC Plattform eine Systemeigenschaft mit demselben Namen, welche den entsprechenden Wert enthält.

[12]Das Java SDK stellt das Tool *jarsigner* zur Verfügung. Dieses Tool hat ebenfalls den Zweck, die Integrität eines Java Archivs zu garantieren, indem Einweg-Hashfunktionen für die Files des Archivs berechnet und die Hashes signiert werden. Das Tool kann für PUC aber nicht angewendet werden, weil es einmalig die Integrität des Archivs als Ganzes prüft, und danach davon ausgeht, dass die Integrität des Archivs auf der Festplatte gewährleistet ist. Für PUC muss gerade die Integrität der Dateien auf der Festplatte des Dienstanbieters in Frage gestellt werden. Zusätzlich benötigt jarsigner einen öffentlichen Benutzerschlüssel.

4.2 Programm- und Datenübertragung

Tabelle 5: Programm- und Datenübertragung.

Hauptakteur: Alice

Nebenakteure: Bob, Angreifer

Vorbedingungen: Alice hat ihr Programm P gemäss 4.1 vorbereitet.

1. Alice schickt Programm P bestehend aus Classfiles $C_1...C_n$, Eingabedateien $D_1...D_m$, Konfigurationsdatei data.pf und Schlüsseldatei key.ef an Bob.
2. Alice teilt Bob die Synopsis für den Aufruf ihres Programmes P mit.
3. Bob legt Alices Dateien in einem Verzeichnis auf einer Festplatte ab, welches für die Java VM erreichbar ist.

Die Übertragung der Programmdateien, Benutzerdaten und des Konfigurationsfiles stellen kein besonderes Problem dar und können mittels der gängigen Verfahren für Dateiübertragungen bewerkstelligt werden.

4.3 Schlüsselaustausch

Der vom Benutzer gewählte Schlüssel K_{user}wird für die Übergabe mit dem öffentlichen Schlüssel der Plattform K_{pub}verschlüsselt. Damit entfällt der Schritt eines Schlüsseltauschs zwischen Plattform und Benutzer.

4.4 Sichere Programmausführung

Tabelle 6: Sichere Programmausführung

Hauptakteur: VM der PUC-Plattform

Vorbedingungen: Bob hat von Alice ein lauffähiges Programm P bestehend aus den Classfiles $C_1...C_n$ und den Eingabedateien $D_1...D_m$, mit zugehöriger Konfigurationsdatei `data.pf` und Schlüsseldatei `key.ef` erhalten. Die Programm- und Eingabedateien liegen auf einem für die Java VM erreichbaren Verzeichnis.

1. Bob startet die Programmausführung. Die Syntax des Programmaufrufs ist ihm von Alice mitgeteilt worden.
2. Die VM durchläuft einen Initialisierungsprozess.
3. Die VM extrahiert mithilfe ihres privaten Schlüssels K_{priv}aus der Schlüsseldatei `key.ef` Alices Benutzerschlüssel K_{user}.
4. Die VM entschlüsselt mithilfe des Benutzerschlüssels K_{user} die Konfigurationsdatei `data.pf` und liest die von Alice definierten, zusätzlichen Schlüssel $K_{D_n}...K_{D_s}$ sowie weitere benutzerdefinierte Schlüssel-Werte-Paare ein.
5. Beim Laden eines Classfiles C_i oder beim Lesen einer Eingabedatei D_j extrahiert die VM mithilfe des Benutzerschlüssels K_{user}den zugehörigen Hashwert und überprüft damit zunächst die Integrität der Datei.
   Falls eine Integritätsverletzung einer Klasse oder Eingabedatei festgestellt wird, wird die Programmausführung abgebrochen.
6. Bei jedem Methodenaufruf prüft die VM zunächst, ob es sich um eine verschlüsselte Methode handelt.
   (a) Eine unverschlüsselte Methode wird von der VM unverändert ausgeführt.
   (b) Eine verschlüsselte Methode wird von der VM *sicher* ausgeführt.
7. Alices Programm erzeugt die mit den Schlüsseln $K_{D_n}...K_{D_s}$ verschlüsselten Ausgabedateien $D_n...D_s$.

Der Ablauf für die sichere Programmausführung ist in Tabelle 6 dargestellt. Die sichere Ausführung einer einzelnen Methode wird in Abschnitt 4.4.6 spezifiziert.

Die Massnahmen für die Wahrung der Integrität und Vertraulichkeit der Laufzeitdatenstrukturen werden in Abschnitt 4.4.7 spezifiziert.

4.4.1 Initialisierung der Java VM

Im Initialisierungsprozess liest die modifizierte Java VM den Private Key der Plattform K_{priv} ein.

Für das Proof of Concept müssen die sicheren Speicherbereiche für Code, Stack, Heap und allenfalls zuätzlich notwendige Buffer alloziert werden[13]. Der private Schlüssel der Plattform wird für das Proof of Concept aus einer Datei ausgelesen.

4.4.2 Entschlüsselung des Benutzerschlüssels

In diesem Schritt entschlüsselt die PUC-Plattform mithilfe ihres privaten Plattformschlüssels K_{priv} den Benutzerschlüssel K_{user} aus der `key.ef` Datei. Der Benutzerschlüssel wird für eine Programmausführung nur einmal entschlüsselt und danach im sicheren Speicher gespeichert.

4.4.3 Benutzerdefinierte Schlüssel-Werte-Paare

Für das Einlesen der benutzerdefinierten Schlüssel-Werte-Paare muss die Java VM die Konfigurationsdatei `data.pf` entschlüsseln und parsen. Für das Entschlüsseln der Konfigurationsdatei wird ein Buffer im sicheren Speicher der PUC Plattform alloziert. Die entschlüsselte Konfigurationsdatei muss für das Parsen vollständig innerhalb dieses Buffers liegen. Falls die Datei zu gross ist, um in einem Mal in diesen Buffer entschlüsselt zu werden, müssen Entschlüsselung und Parsen blockweise ausgeführt werden. Die Grösse des sicheren Buffers wird auf 4 KiByte festgesetzt.

Für jedes Schlüssel-Werte Paar in MetaData erzeugt die PUC Plattform eine Systemeigenschaft, auf welche das Benutzerprogramm mittels `System.getProperty("propertyname")` zugreifen kann. Auf diese Weise kann für eine Eingabedatei D der benutzerdefinierte Datenschlüssel $K_D$übergeben werden.

Das Überschreiben der Standard-Systemeigenschaften durch benutzerdefinierte Werte darf nicht erlaubt sein.

4.4.4 Integritätsprüfung beim Laden von Dateien

In Algorithmus 1 ist der Ablauf der Integritätsprüfung beim Laden einer Datei beschrieben. Wird beim Laden einer Datei eine Integritätsverletzung festgestellt so wird die Programmausführung abgebrochen.

[13]Eine reale PUC-Plattform müsste diese Bereiche nicht allozieren, sondern würde in diesem Schritt Integritätstests der Hard- und Softwarekomponenten durchlaufen.

Alg. 1 Integritätsprüfung beim Laden von Dateien

1. Laden einer Datei
   (a) Hashwert für Datei aus `data.pf` entschlüsseln und zwischenspeichern
   (b) Kopiere Datei in Arbeitsspeicher
      i. a priori: sicherer Arbeitsspeicher
      ii. falls zu gross für sicheren Arbeitsspeicher: Lagere die Datei oder Teile davon auf Arbeitsspeicher des Hostsystems aus unter Berechnung von Laufzeithashes für die Sicherstellung der Integrität der ausgelagerten Teile
   (c) Berechne den Hashwert der Datei
   (d) Überprüfe Hashwert
   Falls der berechnete Hashwert nicht mit dem Hashwert aus dem `data.pf` File übereinstimmt: Abbruch der Programmausführung

Vor Benutzung der Datei berechnet die Java VM mittels eines vorgegebenen Einweg-Hash-Algorithmus einen Hashwert für die Datei. Dieser Wert wird verglichen mit dem Hashwert, der vom Dienstbenutzer für die entsprechende Datei mitgeliefert wurde. Falls die beiden Hashwerte nicht übereinstimmen, wurde die Datei manipuliert und die Programmausführung wird abgebrochen.

Die entsprechenden Funktionalitäten müssen in das I/O Subsystem der PUC Umgebung integriert werden. Das I/O Subsystem der PUC Plattform lädt ein Classfile oder eine Eingabedatei via den sicheren Arbeitsspeicher. Falls die Datei zu gross ist für den sicheren Arbeitsspeicher, wird sie unter Erzeugung von schlüsselabhängigen Einweg-Hashes in den ungeschützten Arbeitsspeicher ausgelagert (siehe Integrität der Laufzeitdaten, Abschnitt 4.4.7). Dateien, welche nicht als Ganzes für die Überprüfung in den ungeschützten Arbeitsspeicher eingelesen werden können, müssen blockweise geladen und die Hashwerte blockweise nachgerechnet werden. Das Verschlüsselungswerkzeug muss die entsprechenden Limitationen der PUC Plattform kennen und solche Hashwerte für einzelne Blocks erzeugen.

Integrität des data.pf File
Weil die PUC-Plattform den Benutzerschlüssel K_{user} während der Ausführung behält, muss die `data.pf` Datei nicht vor Integritätsverletzungen geschützt werden. Falls ein Angreifer das `data.pf` File manipuliert, z.B. durch Austauschen eines Blocks, so liefert die Entschlüsselung mittels des Benutzerschlüssels K_{user} fehlerhafte Klassennamen, falsche Klassenschlüssel, falsche Hashwerte oder überhaupt sinnlose Daten. Alle diese Fehler führen aber bei der PUC Ausführungsumgebung zum Abbruch des Programms.

Bemerkung: Für einen Angreifer ist es möglich, Klassen des Programms zu erweitern oder neue Klassen zum Programm hinzuzufügen, eine Konfigurationsdatei `data.pf` mit allen Einweg-Hashes und erfundenen Klassenschlüsseln zu generieren, diese Konfigurationsdatei mit einem "eigenen" Benutzerschlüssel zu verschlüsseln und diesen Benutzerschlüssel mit dem Public Key der Plattform in eine `key.ef` Datei abzulegen. Dieser Angriff ermöglicht aber lediglich die Analyse und Ausführung der nicht als sicher deklarierten Methoden der Klassen und nicht den Aufruf oder die Entschlüsselung sicherer Methoden oder die Entschlüsselung der Eingabedateien.

4.4.5 Entschlüsseln und Ausführen sicherer Methoden

Laden und Initialisieren einer sicheren Klasse
Das Laden einer sicheren Klasse unterscheidet sich nicht wesentlich vom Laden einer nicht verschlüsselten Klasse. Runtime Constant Pool und Methodencode werden in der Method Area abgelegt. Das CryptoCode-Attribut einer verschlüsselten Methode wird zunächst in verschlüsseltem Zustand in die Method Area kopiert.

Das Linken der Klasse kann unverändert erfolgen, sofern die von der VM Spezifikation an dieser Stelle verlangte Verifikation des Bytecodes übersprungen wird. Das bedeutet, dass der von der Java VM Spezifikation vorgeschriebene Prozess an dieser Stelle nicht korrekt eingehalten wird. Der Versuch, den verschlüsselten Methodencode ohne Entschlüsselung zu verifizieren würde zwangsläufig zu einem `ClassFileVerificationError` Fehler führen[14]. Für eine Verifizierung müsste der Methodencodes zu diesem Zeitpunkt bereits entschlüsselt werden.

Das Initialisieren der statischen Felder, das Auflösen der symbolischen Referenzen des Constant Pool und die Überprüfung der Zugriffskontrolle bleiben unverändert.

Als letzter Schritt erfolgt die Initialisierung der Klasse, d.h. Initialisierung der Felder und Ausführen der statischen Initialisierungsmethoden. Falls eine statische Initialisierungsmethode als sichere Methode deklariert ist, so muss sie sicher ausgeführt werden (siehe Abschnitt 4.4.6, Sichere Methodenausführung).

Der Ablauf ist in Algorithmus 2 dargestellt.

4.4.6 Sichere Methodenausführung

Berechnung des Methodenschlüssels
Der Methodencode einer verschlüsselten Methode wird erst zum Zeitpunkt ihrer Ausführung entschlüsselt. Die Methodenschlüssel werden von der PUC Plattform aus dem Konfigurationsfile abgeleitet. Sobald die VM versucht, eine sichere Methode auszuführen, wird das Konfigurationsfile mittels des Benutzerschlüssels K_{user} entschlüsselt und der Klassenschlüssel der Klasse, zu der die Methode gehört, gesucht. Sobald der Klassenschlüssel gefunden ist, berechnet die VM aus dem Methodennamen und dem Klassenschlüssel den Methodenschlüssel nach dem in 4.1.3 beschriebenen Verfahren.

Ausführung der sicheren Methode
Für die sichere Ausführung wird der Code einer sicheren Methode segmentweise vom ungeschützten Arbeitsspeicher in den sicheren Arbeitsspeicher *(sicheres Codesegment)* entschlüsselt[15]. Zugriffe auf den Programmzähler werden entsprechend Offset korrigiert, d.h. für eine sichere Methode wird nicht der Instruktionscode aus dem verschlüsselten Methodencode, sondern der entsprechende, entschlüsselte Instruktionscode aus dem sicheren Codesegment zurückgegeben.

Der korrigierte Programmzähler wird für jede Instruktion neu aus dem Originalprogrammzähler abgeleitet. Sobald der korrigierte Programmzähler aus dem sicheren Codesegment zeigt, muss das nächste Codesegment entschlüsselt werden.

[14] Der Verifikationsschritt ist bei der JamVM sehr viel weniger strikt implementiert als z.B. bei Sun's Java VM.

[15] In dieser Arbeit wird die Bezeichnung *Segment* verwendet, weil der Begriff *Block* in einem kryptographischen Kontext missverständlich sein könnte. Ein Segment kann mehr als einen Block beinhalten.

Alg. 2 Laden und Initialisieren einer Klasse

1. Lade Klasse C mit Namen N mittels Classloader L
   Der Classloader übergibt der VM eine Bytecode Darstellung von C
2. Definiere Klasse C aus Bytecode Darstellung von C
   (a) Parse Bytecode und erstelle VM-spezifische Datenstrukturen für Felder, Methoden, Runtime Constant Pool
   (b) Registriere L als initiierenden Loader von C
   Der verschlüsselte Methodencode bleibt während der Definition verschlüsselt und wird erst bei der Ausführung entschlüsselt.
3. Linke Klasse C
   (a) Verifizieren von C: Strukturelle Überprüfung der binären Representation des Classfiles
   Die Verifikation des Methodencodes sicherer Methoden wird übersprungen.
   (b) Vorbereitung von C: Initialisieren der statischen Felder und ihrer Werte
   (c) Auflösung von C: Die symbolischen Referenzen des Constant Pool müssen in konkrete Adressen des Runtime Constant Pool aufgelöst werden.
   (d) Zugriffskontrolle
4. Initialisierung
   (a) Initialisiere statische Felder
   (b) Führe statische Initialisierungsmethoden aus

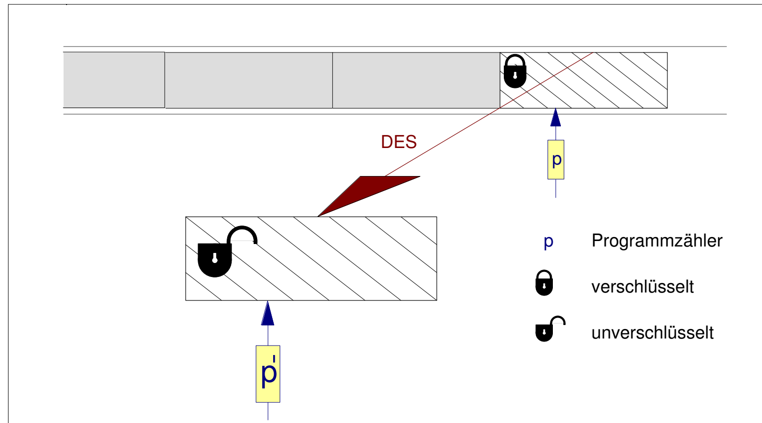


Abbildung 5: Entschlüsselung eines Codesegments

Abbildung 5 veranschaulicht die Entschlüsselung eines Segments vom ungeschützten Arbeitsspeicher in den sicheren Arbeitsspeicher und das Umlenken des Programmzählers p.

Die Funktionalität kann in der Java VM mittels einer Funktion realisiert werden, welche sämtliche Zugriffe auf den Programmzähler ersetzt. Der Pseudocode dieser Methode ist im Algorithmus 3 dargestellt.

4.4.7 Laufzeitdaten

Verschlüsselung der Laufzeitdaten

Stack Für das Proof of Concept wird der gesamte Stack verschlüsselt. Die Verschlüsselung geschieht segmentweise. Es wird für alle Segmente derselbe Schlüssel K_S und einfaches XOR verwendet. Für die Bearbeitung wird das aktuelle Stacksegment in den sicheren Arbeitsspeichers entschlüsselt. Sobald auf ein neues Segment zugegriffen werden soll, muss das momentan verschlüsselte Segment an die richtige Stelle des ungeschützten Arbeitsspeichers zurückverschlüsselt werden. Abbildung 6 veranschaulicht das Entschlüsselung und Verschlüsseln eines Segments vom ungeschützten Arbeitsspeicher in den sicheren Arbeitsspeicher und das Umlenken des Frame Pointers f.

Heap Für das Proof of Concept wird der gesamte Heap für Objekte und Arrays verschlüsselt. Die Verschlüsselung geschieht segmentweise. Es wird für alle Segmente derselbe Schlüssel K_H und einfaches XOR verwendet. Für die Bearbeitung eines Objekts wird das aktuelle Heapsegment in den sicheren Arbeitsspeicher entschlüsselt. Sobald auf ein neues Segment zugegriffen werden soll, muss das momentan verschlüsselte Segment an die richtige Stelle des ungeschützten Arbeitsspeichers zurückverschlüsselt werden.

Alg. 3 Methodenausführung

PC: Programmzähler, zeigt in verschlüsselten Methodencode
B: Basisadresse des sicheren Codesegments
S: Addresse des momentan entschlüsselten Codesegments

1. Für jeden Zugriff auf Programmzähler PC
   (a) Sichere Methode
      i. Ist Segment S in das PC zeigt schon entschlüsselt?
         A. Ja:
            Liefere Instruktion an der Stelle B + (PC - S) zurück
         B. Nein:
            Berechne neues Segment:
            S = (PC / Grösse sicheres Codesegment) * (Grösse sicheres Codesegment)
            Entschlüssle S nach B
            Liefere Instruktion an der Stelle B + (PC - S) zurück
   (b) Normale Methode

      Liefere Instruktion an der Stelle PC zurück

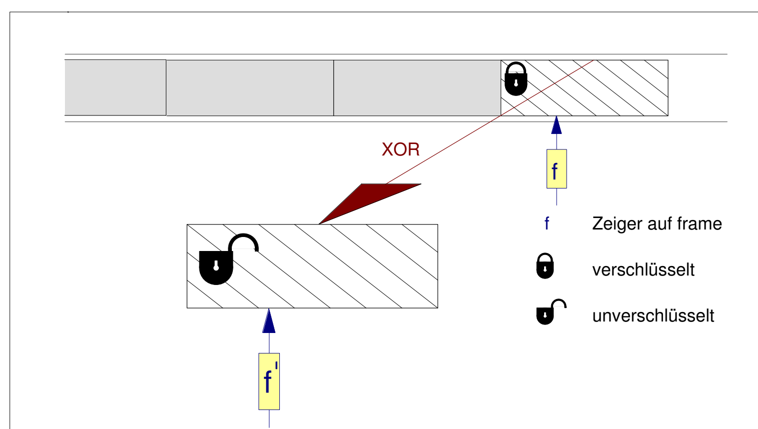


Abbildung 6: Verschlüsselung / Entschlüsselung eines Stack Segments

Integrität der Laufzeitdaten
Um die Integrität der Laufzeitdaten zu wahren, wird jedes Segment, das in den ungeschützten Arbeitsspeicher geschrieben wird, zusätzlich mit einem schlüsselabhängigen Hashwert versehen. Als schlüsselabhängige Hashfunktion wird MAC verwendet. Der Schlüssel für die Prüfsumme wird mittels einer Hashfunktion (MD5) aus einem von der VM zu Beginn der Programmausführung generierten Prozessschlüssel K_P und der Speicheraddresse des Segments erzeugt: $K_S = Left_L[H(A_S \| K_P)]$.

Dabei bezeichnet L die Länge des generierten Schlüssels, A_S die Segmentaddresse.

Um ein Segment zu lesen, kopiert die VM sich dessen Inhalt zunächst in den sicheren Arbeitsspeichers. Dann rechnet sie den schlüsselabhängigen Hashwert nach und vergleicht ihn mit dem gespeicherten Wert. Stimmt der berechnete Wert nicht mit dem berechneten Wert überein, wird die Programmausführung abgebrochen.

Bemerkung 1: Package java.security kennt die Klasse SignedObject. SignedObject erlaubt die Integrität von Java Objekten während der Ausführung zu überwachen, indem eine serialisierte Version dieser Objekte in das SignedObject eingebettet und signiert wird. Damit lassen sich zwar Objekte vor Integritätsverletzungen schützen, der Schutz ist aber für das PUC Szenario unpraktikabel, weil mit diesem Konzept jedes Objekt nach jedem neuen Zugriff signiert werden müsste. Zusätzlich müsste das SignedObject sich selbst signieren.

Bemerkung 2: Die Generierung von Laufzeit-Hashwerten ermöglicht die Überprüfung der Integrität von Dateien, welche zu gross sind um als ganzes in den sicheren Arbeitsspeicher gelesen werden zu können. Vgl. Abschnitt 4.4.4.

4.5 Übertragung der Resultate und Validation

Tabelle 7: Übertragung der Resultate und Validation

Hauptakteur: Bob

Vorbedingungen: Alices Programm P wurde auf Bobs PUC Plattform erfolgreich ausgeführt und hat die Ausgabedateien $D_n ... D_s$ erzeugt.

1. Bob schickt die vom Programm P erzeugten Ausgabedateien $D_n ... D_s$ an Alice zurück.
2. Alice entschlüsselt die Ausgabedateien $D_n ... D_s$ und prüft ihre Richtigkeit.

Für die allfällige Verschlüsselung der Ausgabe ihres Programms benutzt Alice die Datenschlüssel, die sie dem Programm mittels der Konfigurationsdatei `data.pf` übergeben hat. Ebenso generiert Alices Programm selbst eine allfällige schlüsselabhängige Prüfsumme für die Programmausgabe.

Bob schickt Alice die Ausgabedaten über das Netzwerk zurück. Alice prüft ihre Richtigkeit. Der Ablauf ist in Tabelle 7 dargestellt.

Die Gewährleistung der Authentizität der Ausgabedaten stellt ein schwieriges Problem dar, das hier nicht weiter behandelt wird.

Tabelle 8: PUC System Policy

Recht	**Methoden**	**Permission**
Ausführen von Systemprogrammen	java.lang.Runtime.exec()	java.io.FilePermission "{command}", "execute"
Linken von Systembibliotheken	java.lang.Runtime load(), loadLibrary()	java.lang.Runtime-Permission "loadLibrary.{libName}"
Schreiben von System Properties	java.lang.System setProperty(), setProperties()	java.util.PropertyPermission "{key}", "write"
Ändern des Security Managers	java.lang.System setSecurityManager()	java.lang.Runtime-Permission "setSecurityManager"
Ändern von Standard In, Standard Out und Standard Err	java.lang.System setIn(), setOut(), setErr()	java.lang.Runtime-Permission "setIO"
Die Ansprechbarkeit von Feldern eines Objekts ändern	java.lang.reflect. AccessibleObject setAccessible()	java.lang.reflect. ReflectPermission "suppressAccessChecks"
Alle Rechte im Zusammenhang mit Security	java.lang.security.*	java.security. SecurityPermission

4.6 Sicherheits-Policy für die PUC-Plattform

Für die PUC Plattform wird eine System Policy definiert. Die Policy verwehrt primär Rechte, welche Benutzerprogrammen den Aufruf von nativen Funktionen sowie das Laden von Systembibliotheken verwehren. Ebenfalls verwehrt ist das Ändern der Ansprechbarkeit von Feldern via Reflection sowie alle Rechte für Funktionen, welche die Sicherheitseinstellungen ändern.

Zusätzlich verwehrt die Policy das Umleiten der Standardeingabe, Standardausgabe und der Standard-Fehlerausgabe.

Bemerkung: Diese Policy gilt zusätzlich zu der in Abschnitt 3.4.1 getroffenen Annahme, dass immer nur ein Benutzerprogramm gleichzeitig auf der Plattform am Laufen sein soll, und schützt vor allem die Plattform selbst und ihre Systembibliotheken vor einem bösartigen Benutzerprogramm.

5 Implementierung

In diesem Kapitel werden die Details sowie der Stand der Implementierung beschrieben.

5.1 Verschlüsselungswerkzeug

Das Verschlüsselungswerkzeug wurde in Java realisiert.

5.1.1 Benutzte Libraries und Software

Bouncy Castle Crypto API bcprov-jdk14-125.jar
Eine Open Source Implementierung kryptographischer Algorithmen in Java.

http://www.bouncycastle.org [2]:

> "Here at the Bouncy Castle, we believe in encryption. That's something that's near and dear to our hearts. We believe so strongly in encryption, that we've gone to the effort to provide some for everybody."

5.1.2 Übersicht Module

Das Werkzeug besteht aus den in Tabelle 9 beschriebenen Klassen. Eine Implementierung auf der Grundlage des Packages `sun.tools.javap`, welches dem Java Disassembler `javap` zugrunde liegt, wurde in Betracht gezogen, konnte aufgrund der unzureichenden Dokumentation dieses Packages jedoch nicht durchgeführt werden.

5.2 Modifikation Java VM

5.2.1 Benutzte Libraries und Software

JamVM 1.1.4
JamVM ist eine kleine, sehr vollständige Open Source Implementierung der Java 2 Spezifikation in C.

Robert Laugher, http://jamvm.sourceforge.net [12]:

> "JamVM is a new Java Virtual Machine which conforms to the JVM specification version 2 (blue book). In comparison to most other VM's (free and commercial) it is extremely small, with a stripped executable on PowerPC of only ~120K, and Intel 90K. However, unlike other small VMs (e.g. KVM) it is designed to support the full specification, and includes support for object finalisation, the Java Native Interface (JNI) and the Reflection API."

GNU Classpath 0.09
GNU Classpath ist eine Open Source Implementierung der Java Libraries.

http://www.classpath.org [3]:

> "GNU Classpath, Essential Libraries for Java, is a GNU project to create free core class libraries for use with virtual machines and compilers for the java programming language."

Tabelle 9: Klassen des Verschlüsselungswerkzeugs

Klasse	**Beschreibung**
EncryptionTool	Verschlüsselt rekursiv alle Classfiles im angegebenen Verzeichnis und erstellt das data.pf sowie das key.ef File.
EncryptClass	Verschlüsselt ein einzelnes Classfile.
Encrypter	Kapselung der notwendigen kryptographischen Methoden. Encrypter kann symmetrische Schlüssel lesen und erzeugen, Methodenschlüssel berechnen, einen Public Key aus einem X509-Zertifikat auslesen und Dateien und Bytearrays symmetrisch verschlüsseln.
ClassFileRepresentation	Repräsentation eines Classfiles in einer Form, welche den Zugriff auf die einzelnen Elemente dieses Classfiles wie ConstantPool, Methoden, Felder usw. erlaubt. Verschiedene Methoden erlauben das Hinzufügen von Klassenattributen und Einträgen in den ConstantPool.
cp_info	Repräsentation eines ConstantPool-Eintrags des Classfiles. Inner class von ClassFileRepresentation.
attribute_info	Repräsentation eines Attributs des Classfiles. Inner class von ClassFileRepresentation.
field_info	Repräsentation eines Feldes des Classfiles. Inner class von ClassFileRepresentation.
method_info	Repräsentation einer Methode des Classfiles. Inner class von ClassFileRepresentation.

OpenSSL Libraries 0.9.7d
OpenSSL ist eine Open Source Implementierung kryptographischer Algorithmen und Protokolle.

http://www.openssl.org [4]:

> "The OpenSSL Project is a collaborative effort to develop a robust, commercial-grade, full-featured, and Open Source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library. The project is managed by a worldwide community of volunteers that use the Internet to communicate, plan, and develop the OpenSSL toolkit and its related documentation."

Die modifizierte JamVM wird im folgenden als CryptoJVM bezeichnet. Installation und Aufruf der CryptoJVM sind in den INSTALL und README Dateien beschrieben.

5.2.2 Übersicht Module

Tabelle 10 zeigt die für das Proof of Concept modifizierten Dateien der JamVM. Für die Implementierung der verschiedenen Funktionalitäten wurden in diesen Dateien die direkten Zugriffe auf Programmzähler und Stack durch Funktionsaufrufe ersetzt, welche diese Zugriffe für die sichere Ausführung eines Java Programmes überwachen. Die aufgerufenen Funktionen sowie sämtliche weiteren von der CryptoJVM benötigten Funktionen sind im neuen Modul `secure.c` mit der Header Datei `secure.h` implementiert.

5.2.3 Code

CryptoJVM wird defaultmässig nur für das Ausführen von verschlüsseltem Code compiliert.

5.2.4 Stack

Die Verschlüsselung des Stack kann in zwei verschiedenen Modi betrieben werden:

1. Mittels des Compiler Flags SECURE_STACK wird die CryptoJVM für das Ver- und Entschlüsseln des Stacks mit XOR konfiguriert. In dieser Konfiguration wird jeder aus dem Stack ausgelesene und jeder in den Stack geschriebene Wert einzeln entschlüsselt und verschlüsselt. In diesem Modus muss der sichere Arbeitsspeicher für den Stack lediglich so gross sein, wie die grösste Datenstruktur im Stack[16].
2. Der Stack wird segmentweise entschlüsselt und verschlüsselt, wenn zusätzlich zu SECURE_STACK das Flag SECURE_STACK_BLOCKWSE gesetzt wird.

5.2.5 Heap

Die Verschlüsselung der Laufzeitdatenstrukturen für den Heap wurde nicht implementiert.

[16] 5*sizeof(u4) = 20 Bytes für ein Frame

Tabelle 10: Modifizierte Module JamVM

Modul	**Beschreibung**	**Beschreibung der Modifikationen**
`alloc.c`	Memory Allocator und Garbage Collection	Abfangen der Zugriffe auf den Stack.
`class.c`	Parsen, definieren und Linken von Klassen	Parsen der sicheren Methoden des CryptoCode Attributes, Abfangen der Zugriffe auf den Stack.
`excep.c`	Exception Handler	Abfangen der Zugriffe auf den Stack.
`execute.c`	Methodenaufruf für Java- und native Methoden	Abfangen der Zugriffe auf den Stack.
`frame.h`	Generierung des Top-Frames eines Methodenaufrufs	Abfangen der Zugriffe auf den Stack.
`interp.c`	Java Interpreter	Abfangen der Zugriffe auf Programmzähler und Stack.
`jam.c`	Initialisierung und `main` Methode	Initialisierungsaufruf für CryptoJVM, Einlesen der zusätzlichen Aufrufparameter.
`jni.c`	Implementierung von JNI	Abfangen der Zugriffe auf den Stack.
`natives.c`	Native Funktionen der VM	Einlesen der benutzerdefinierten Schlüssel-Werte-Paare in die Systemproperties, Abfangen der Zugriffe auf den Stack.
`reflect.c`	Java Reflection	Abfangen der Zugriffe auf den Stack.
`secure.c`	Implementierung der CryptoJVM Methoden	Neues Modul.
`thread.c`	Java Threads	Abfangen der Zugriffe auf den Stack.

5.3 Implementierungsstand

Tabelle 11 zeigt den Stand der im Rahmen des Proof of Concept implementierten Funktionalitäten.

Tabelle 11: Implementierungsstand

Funktionalität	**Implementierung**
Verschlüsselung Code	DES-CBC. Generierung der Methodenschlüssel aus Methodennamen und Klassenschlüssel mit MD5. Der fixe Initialisierungsvektor $0xcafebabe$ wird dem verschlüsselten Methodencode vorangestellt.
Einweg Hashwertberechnung für Dateien	nicht implementiert
Verschlüsselung Konfigurationsdatei	DES-CBC. Der fixe Initialisierungsvektor $0xcafebabe$ wird der verschlüsselten Datei vorangestellt.
Verschlüsselung Benutzerschlüssel	RSA mit 1024 Bit Schlüssel, Public Key wird aus X509-Zertifikat ausgelesen.
Entschlüsselung Benutzerschlüssel	nicht vollständig implementiert
Entschlüsselung Konfigurationsdatei	DES-CBC
Ver-/Entschlüsselung Stack	XOR. Schlüssel wird aus `/dev/random` gelesen.
Ver-/Entschlüsselung Heap	nicht implementiert
Schlüsselabhängige Hashwertberechnung für Stack und Heap	nicht implementiert
Security Policy	nicht implementiert

In der modifizierten Version der JamVM ist der Aufruf von nativen Methoden weiterhin uneingeschränkt möglich, um das Funktionieren der durch die Classpath Library zur Verfügung gestellten Funktionen zu gewährleisten. Im Rahmen des Proof of Concept wurde keine Security Policy für die CryptoJVM realisiert.

6 Validation und Ergebnisse

In diesem Kapitel werden zunächst die durchgeführten Tests präsentiert und danach die gewonnenen Erkenntnisse dargestellt.

6.1 Validation

6.1.1 Testcases

Eigene Testcases
Um die modifizierte JamVM zu testen, wurden die in Tabelle 12 beschriebenen Testcases `Test_1.java` bis `Test_10.java` geschrieben und ausgeführt. Diese geben in wechselseitigem Aufruf von verschlüsselten und unverschlüsselten Methoden die Nachricht "Hello Secure World" aus.

Heaptest
Ein Virtual Machine Performance Test von Sun Microsystems `http://java.sun.com/developer/technicalArticles/Programming/JVMPerf/`. HeapTest ist ein Programm mit mehreren Threads, das die im Aufruf angegebene Anzahl Durchläufen von Speicherallozierungen und Berechnungen durchführt. Mit jedem Durchlauf verschiebt sich dabei die Last mehr von der Speicherallozierung in Richtung Berechnung.

Im Heaptest wurde die statische `main()`-Methode ersetzt durch die Methode `secure_main()` und ihr Aufruf durch `new HeapTest().secure_main()`. Es wurden die folgenden Tests durchgeführt:

1. `$JAVA heaptest/HeapTest 1 1` (1 Thread, 1 Loop)
2. `$JAVA heaptest/HeapTest 2 2` (2 Threads, 2 Loops)

Dabei steht $JAVA als Platzhalter für die verwendete VM (siehe Abschnitt 6.1.2).

weka-3-4
Eine Open Source Data Mining Software der University of Waikato, New Zealand `http://www.cs.waikato.ac.nz/ml/weka/`.

Für das Package wurden die folgenden Methoden als sicher deklariert:

- `buildClassifier`
- `classifyInstance`
- `distributionForInstance`

Mit den verschlüsselten Klassen wurden die folgenden Tests für Klassifikation, Assoziation und Filterung der Weka-Beispieldaten durchgeführt:

1. `$JAVA weka.classifiers.trees.J48 -t $WEKAHOME/data/iris.arff`
2. `$JAVA weka.classifiers.bayes.NaiveBayes -t $WEKAHOME/data/labor.arff`

Tabelle 12: Testcases

Test	Beschreibung	Erwartetes Resultat
Test_1.java	Verschlüssle eine Instanzmethode und führe sie aus.	Korrekte Ausführung der Methode; gleiches Resultat wie bei unverschlüsselter Ausführung.
Test_2.java	Verschlüssle mehrere Instanzmethoden einer Klasse, die sich gegenseitig aufrufen und führe sie aus.	Korrekte Ausführung der Methoden; gleiches Resultat wie bei unverschlüsselter Ausführung.
Test_3.java	Wie Test_2, aber wechselseitiges Aufrufen von verschlüsselten und unverschlüsselten Methoden.	Korrekte Ausführung der Methoden; gleiches Resultat wie bei unverschlüsselter Ausführung.
Test_4.java	Verschlüssle Implementierung einer abstrakten Methode und führe sie aus.	Korrekte Ausführung der Methode; gleiches Resultat wie bei unverschlüsselter Ausführung.
Test_5.java	Wie Test_3, aber mit mehreren Klassen.	Korrekte Ausführung der Methoden; gleiches Resultat wie bei unverschlüsselter Ausführung.
Test_6.java	Verschlüssle eine Methode, welche eine Exception erzeugt, die durch einen try-catch-Block in der Methode abgefangen wird, und führe sie aus.	Korrekte Ausführung der Methode; gleiches Resultat wie bei unverschlüsselter Ausführung.
Test_7.java	Verschlüssle eine Methode, welche eine Exception erzeugt, die nicht durch einen try-catch-Block in der Methode abgefangen wird, und führe sie aus.	Korrekte Ausführung der Methode; gleiches Resultat wie bei unverschlüsselter Ausführung.
Test_8.java	Verschlüssle mehrere Methoden eines Programms mit mehreren Threads und führe das Programm aus.	Korrekte Ausführung der Methoden; gleiches Resultat wie bei unverschlüsselter Ausführung.
Test_9.java	Verschlüssle einige Methoden einer Klasse mit inneren Klassen, welche sich gegenseitg aufrufen.	Korrekte Ausführung der Methode; gleiches Resultat wie bei unverschlüsselter Ausführung.
Test_10.java	Führe eine verschlüsselte oder nicht verschlüsselte Methode aus, welche einen StackOverflowError erzeugt.	Die modifizierte VM verhält sich gleich wie die nicht modifizierte VM und bricht die Programmausführung mit der Meldung "java.lang.StackOverFlowError" ab.
PerformTests .java	Verschlüssle eine statische Methode und führe sie aus.	Korrekte Ausführung der Methode; gleiches Resultat wie bei unverschlüsselter Ausführung.
Alle Tests	Verschlüssle eine Methode, welche eine Methode eines Interfaces implementiert und führe sie aus.	Korrekte Ausführung der Methode; gleiches Resultat wie bei unverschlüsselter Ausführung.

3. `$JAVA weka.classifiers.trees.J48 -t $WEKAHOME/data/iris.arff -m costs -v`
4. `$JAVA weka.associations.Apriori -t $WEKAHOME/data/weather.nominal.arff`
5. `$JAVA weka.filters.supervised.attribute.Discretize -i $WEKAHOME/data/iris.arff -c last`

Dabei steht $JAVA als Platzhalter für die verwendete VM (siehe Abschnitt 6.1.2). $WEKAHOME für das Verzeichnis der Weka-Beispieldaten.

6.1.2 Testdurchführung

Konstanten
Für die Testdurchführung wurden die Konstanten mit den in Tabelle 13 aufgeführten Werten belegt.

Tabelle 13: Konstanten

Konstante	Beschreibung	Wert
DES_KEY_LEN	Länge DES Schlüssels	8 Byte
XOR_KEY_LEN	Länge XOR Schlüssel	8 Byte
CODE_BLOCK_SIZE	Blocklänge für Verschlüsselung Code	8 Byte
STACK_BLOCK_SIZE	Blocklänge für Verschlüsselung Stack	8 Byte
CNFG_BLOCK_SIZE	Blocklänge für Verschlüsselung Konfigurationsdatei	8 Byte
SECURE_CODE_SIZE	Grösse sicherer Arbeitsspeicher Code	512 Byte
SECURE_STACK_SIZE	Grösse sicherer Arbeitsspeichers Stack	512 Byte
SECURE_BUFFER_SIZE	Grösse sicherer Arbeitsspeicher für den Buffer bei Decodierung des Konfigurationsfiles	256 Byte
SECURE_CODE_MARGIN	Abstand vom oberen Ende des sicheren Code Segments für Opcode rewriting	4 Byte
PRIVATE_KEY_FILE	Privater Plattformschlüssel (X509)	"privkey.pem"
RSA_KEY_LEN	Länge des privaten Plattformschlüssels	128 Bytes
SECURE_PREFIX	Kennzeichnung sicherer Methoden	"secure_"

Die Grösse des sicheren Code Blocks muss für aussagekräftige Tests kleiner gewählt werden als der Code der compilierten Methode, welcher meist ziemlich kurz ist. Weil die Entschlüsselung des Benutzerschlüssels mittels des privaten Plattformschlüssels nicht vollständig implementiert ist, wurden bei den Tests der Verschlüsselungs- und Entschlüsselungsschritt für den Benutzerschlüssel weggelassen.

Durchläufe
Zunächst wurden die Tests mit Sun's Java VM und mit der unveränderten JamVM auf den unverschlüsselten Programmen durchgeführt.

Danach wurde jeder Test mit drei verschieden konfigurierten und compilierten CryptoJVMs auf den verschlüsselten Programmen durchgeführt:

1. CryptoJVM_1: Nur Entschlüsseln von Code.
2. CryptoJVM_2: Zusätzliches Ver-und Entschlüsseln des Stack (SECURE_STACK=1).
3. CryptoJVM_3: Segmentweises Ver- und Entschlüsseln des Stack (SECURE_STACK=1 und SECURE_STACK_BLOCKWSE=1).

6.1.3 Testergebnisse

Mit Ausnahme der Programme, die mehrere Threads starten, wurden alle Testprogramme korrekt ausgeführt. Die Programme mit mehreren Threads wurden in der Konfiguration SECURE_STACK teilweise korrekt ausgeführt. In der Konfiguration SECURE_STACK_BLOCKWSE wurde nie ein Programm mit mehreren Threads korrekt ausgeführt.

Die Verschlüsselung des Stack mit sofortigem Zurückschreiben in den verschlüsselten Bereich ist - wie zu erwarten - viel langsamer, als wenn für die Verschlüsselung und Entschlüsselung des Stack ein Segment benutzt wird (Konfiguration SECURE_STACK_BLOCK_WSE).

Die Ergebnisse der Testläufe sind in den testlogs auf der CD dokumentiert.

6.2 Bemerkungen

6.2.1 Code

Opcode Rewriting
JamVM macht für die Verbesserung der Performance von einer Technik Gebrauch, bei welcher der Methodencode während der Ausführung modifiziert wird *(Opcode Rewriting)*.

Bestimmte Instruktionen und ihre Argumente werden während der Ausführung durch sogenannte Pseudoinstruktionen ersetzt. Opcode Rewriting nützt aus, dass die Referenzen auf Felder des Constant Pools nur einmal aufgelöst werden müssen. Bei nachfolgenden Aufrufen derselben Instruktionen werden die bereits aufgelösten Referenzen benutzt, wodurch sich die Performance verbessert.

Opcode Rewriting kann auch für die CryptoJVM zugelassen werden, wenn sichergestellt ist, das die beim Opcode Rewriting ebenfalls umgeschriebenen Operanden an den Addressen $p+1$und $p+2$, wobei p ein Zeiger auf die umgeschriebene Instruktion ist, ebenfalls im entschlüsselten sicheren Stacksegment liegen. In der Implementierung wird deshalb das sichere Code Segment immer nur bis zu einem Abstand SECURE_CODE_MARGIN vor seinem Ende ausgelesen (siehe Abbildung 7). Die Optimierung ist nur für die Dauer gültig, während der das entschlüsselte Code Segment nicht durch ein anderes ausgewechselt wird. Der umgeschriebenen Methodencode wird nicht in den verschlüsselten Methodentext zurückgeschrieben.

Die Optimierung mittels Opcode Rewriting ist in Kapitel 9 der ersten Edition der Java VM Spezifikation[13] beschrieben.

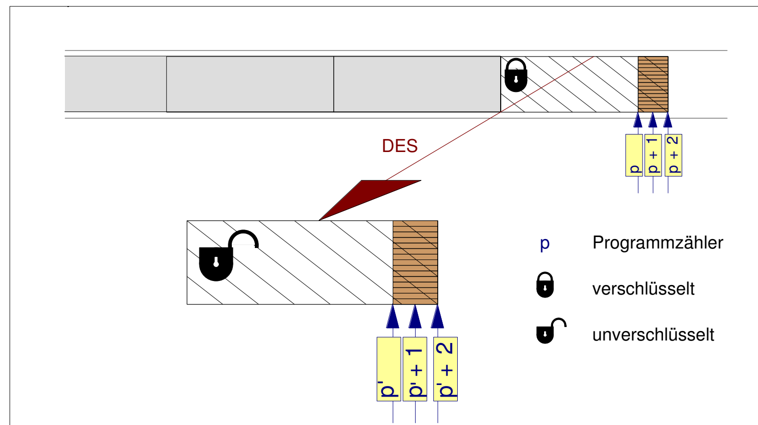


Abbildung 7: Abstand für Opcode Rewriting

Verschlüsselung
In der Implementierung wird, wie in der Spezifikation beschrieben, nur ein Teil des `info`-arrays verschlüsselt. Dadurch lässt sich der Implementierungsaufwand im Rahmen halten. Wenn das gesamte `info`-array oder sogar die gesamte `method_info` Struktur verschlüsselt werden soll, zieht dies umfangreiche Änderungen am VM Code für die Verwaltung der Methodenstrukturen nach sich.

Das in Abschnitt 4.1.3 spezifizierte Verfahren zur Generierung der Methodenschlüssel erzeugt für überladene Methoden innerhalb einer Klasse identische Methodenschlüssel. Falls gewünscht, lässt sich dies durch Verwendung der Methodensignatur anstelle des Methodennamens einfach vermeiden.

Verifikation
Die Java VM Spezifikation verlangt, dass der Code jeder Methode beim Linken einer Klasse verifiziert wird. Dieser Schritt ist für das dynamische Laden von Klassen wichtig, um sicherzugehen, dass die geladene Datei ein gültiges Classfile darstellt. Zum Beispiel wird im Verifikationsschritt für den Code geprüft, ob alle einer Instruktion übergebenen Operanden vom richtigen Typ sind.

Der verschlüsselte Methodencode kann ohne vorherige Entschlüsselung nicht verifiziert werden. Die JamVM Implementierung überspringt - im Gegensatz zu Sun's Java VM - diesen Verifikationsschritt. Das Ausführen verschlüsselter Methoden ist mit einer Verletzung der Spezifikation der Java VM verbunden.

Sicherheit
Mit dem gewählten Ansatz sind in den ungeschützten Speicherbereichen nur die Methodentexte geschützt - statische Konstanten und ihre Werte werden bei Initialisierung einer Klasse unverschlüsselt in den Runtime Constant Pool der Klasse geschrieben. Der Benutzer muss seine Programme so entwickeln, dass keine statischen Konstanten benutzt werden.

Java Exceptions, die beim Ausführen einer Methode auftreten werden weiterhin durch den entsprechenden Exception Handler behandelt. Es liegt in der Verantwortung des Benutzers, Exceptions in seinem Programm so abzufangen, dass bei ihrem Auftreten keine Informationen preisgegeben werden.

6.2.2 Stack

Threads
JamVM benutzt für das Ausführen von Java Programmen mit mehreren Threads die POSIX Threads Implementierung `pthreads`.

Beim Ausführen eines Java Programms startet JamVM auf jeden Fall die folgenden `pthreads`:

- `main_thread`: Der Hauptthread für die Java Ausführung
- `finalizer`: Ausführen der Finalizer Methoden
- `async_gc`: Asynchrone garbage collection

Wenn ein Java Programm selber einen Thread kreiert, startet die VM für diesen Threads einen neuen `pthread`, an den sie den Java Thread anknüpft. Das Scheduling der `pthreads` - und damit der Java Threads - wird dem Betriebssystem überlassen. Der Scheduler kann den momentan laufenden Thread somit zu jedem beliebigen Zeitpunkt unterbrechen. Das kann für das Ver- und Entschlüsseln des Stacks ein Problem darstellen.

Betrachten wir zum Beispiel folgenden Ablauf:

1. Thread A schreibt den Operanden O in den sicheren Stack.
2. Scheduler unterbricht A und startet Thread B.
3. B entschlüsselt das Stacksegment, auf das sein Programmzähler zeigt, in den sicheren Stack.
4. B arbeitet und benutzt den sicheren Stack.
5. Scheduler unterbricht B und fährt mit der Ausführung von A fort.
6. A entschlüsselt das Stacksegment, auf das sein Programmzähler zeigt, in den sicheren Stack.
7. A fährt mit der Programmausführung fort und versucht O zu lesen.

Weil der Scheduler A in Schritt 2 keine Zeit gelassen hat, den Inhalt des sicheren Stack in den ungeschützten Speicher zurückzuschreiben, wird A in Schritt 5 den Wert von O nicht an der erwarteten Adresse finden.

Das Problem lässt sich entschärfen, wenn jede Schreiboperation auf den sicheren Stack zugleich auch in den verschlüsselten Stack schreibt ("write-through"). Das funktioniert aber nur dann zuverlässig, wenn die entsprechende Schreiboperation den Scheduler aussperrt (`pthread_mutex_lock`) und nicht unterbrochen werden kann.

Argumente und Rückgabewerte nativer Methoden
Eine Schwierigkeit bei der Verschlüsselung des Stacks stellen die Aufrufe nativer Methoden dar. JamVM legt bei einem solchen Aufruf die Argumente für die native Methode auf dem Stack ab. Klarerweise erwartet die native Methode ihre Argumente in unverschlüsselter Form. Gleichermassen ist der Rückgabewert einer nativen

Methode primär unverschlüsselt und muss für die Weiterverarbeitung verschlüsselt werden.

Weil einige elementare Funktionen der GNU Classpath Libraries native Methoden aufrufen, ist das Aufrechterhalten der entsprechenden Methodenaufrufe und Callbacks auch für einen verschlüsselten Stack vital. Ein Beispiel ist die Klasse `java.io.File.java`, welches die meisten Methodenaufrufe an `vm.current.java.io.VMFile.java` delegiert. `VMFile.java` wiederum ruft die statischen nativen Methoden von `native/jni/java-io/java_io_VMFile.c` auf. `java_io_VMFile.c` macht via das Java Native Interface JNI Callbacks in die VM.

6.2.3 JNI

In Abschnitt 3.4.1 wurde als Voraussetzung definiert, dass immer nur ein Benutzerprogramm gleichzeitig auf der Plattform läuft. Grundsätzlich spricht in diesem Szenario nichts dagegen, dass ein Benutzerprogramm JNI für die Ausführung nativer Methoden benutzt, solange diese Methoden nicht dazu missbraucht werden können, einen Trojaner auf der Plattform zu hinterlassen, den Plattformschlüssel auszulesen oder die Eigenschaften der Plattform zu verändern.

In einem Szenario mit mehreren gleichzeitig laufenden Benutzerprogrammen muss ein Übergriff eines Prozesses auf die Datenstrukturen eines anderen Prozesses mithilfe nativer Methoden verhindert werden.

Tabelle 14 listet die in JamVM selbst implementierten nativen Methoden auf, welche von Benutzerprogrammen nicht oder nur eingeschränkt ausgeführt werden dürfen. Die entsprechenden Berechtigungen können durch Installation der in Abschnitt 4.6 definierten System Policy verwehrt respektive erteilt werden.

6.3 Vervollständigung der Implementierung

Verschlüsselung Heap Eine Schwierigkeit bei der Implementierung der Verschlüsselung des Heap stellen unter Umständen die Callbacks nativer Methoden in die Java Runtime via JNI dar. Die an native Methoden via JNI zurückgegebenen Objektreferenzen müssen auf unverschlüsselte Objekte zeigen. Es ist unter Umständen eine Anpassung der JNI Implementierung notwendig.

Hashen der Schlüssel Für eine Verbesserung der Performance können die n zuletzt verwendeten Schlüssel in einem sicheren Speicherbereich (Keystore) gehasht werden. Wird ein Schlüssel nicht im Keystore gefunden, so muss er neu berechnet werden.

Die Anzahl maximaler der Benutzerschlüssel pro Klasse ist beschränkt durch die maximale Anzahl der Methoden einer Klasse. Pro Klasse können maximal 65536 Methoden deklariert werden; es müssen also maximal $1 + 2^{16}$ Schlüssel gespeichert werden können.

Wenn von einer durchschnittlichen Anzahl von 17 Methoden pro Klasse[17] und einem Benutzerprogramm von einer gewissen Komplexität mit mehr als 500 Klassen [18] ausgegangen wird, so erhält man bei einer Schlüssellänge von 64 Bit einen Platzbedarf von $500 * (1 + 17) * 8 = 72'000\, Bytes$für alle Schlüssel.

[17] Ein Durchschnitt aus den Klassen des classpath Projekts: 16.8125 Methoden/Klasse.

[18] weka-3-4: 635 Klassen

Tabelle 14: Native Methoden

Klasse	Methoden	Von Benutzerprogrammen aufrufbar
java.lang.VMRuntime	nativeLoad, nativeGetLibname, setProperty, insertSystemProperties	Nein
java.lang.VMClass	isInstance, isAssignableFrom, isInterface, isPrimitive, isArray, getSuperclass, getComponentType, getName, getDeclaredClasses, getDeclaringClass0, getDeclaredConstructors, getDeclaredMethods, getDeclaredFields, getInterfaces, getClassLoader, getClassModifiers, forName0, forName, loadArray, initialize, throwException, hasClassInitializer	Eingeschränkt. Systemklassen sollten nicht reflektierbar sein.
java.lang.VMThrowable	fillInStackTrace, getStackTrace	Eingeschränkt. Eine Stack Trace für Fehler der Systemklassen sollte nicht ausgegeben werden.
java.lang.reflect. Constructor	constructNative, getMethodModifiers, getFieldModifiers, getAndCheckObject, getPntr2Field, getField, getPrimitiveField, setField, setPrimitiveField	Eingeschränkt. (Nicht alle diese Methoden werden exponiert)
java.lang.reflect.Method	invokeNative	Nein
java.lang.VMThread	currentThread, create, sleep, interrupt, isAlive, yield, isInterrupted, interrupted, setPriority, holdsLock	Eingeschränkt. Nur für vom Benutzerprogramm definierte Threads.

Auswahl der Verschlüsselungsalgorithmen Ein mögliche Erweiterung ist, dass der Benutzer den Algorithmus für die symmetrische Verschlüsselung auswählen kann. Die Parameter für das Festlegen des Verschlüsselungsalgorithmus könnten der PUC-Plattform in der Konfigurationsdatei übergeben werden.

7 Zusammenfassung

7.1 Schlussfolgerungen

Die präsentierten Massnahmen garantieren im Rahmen des vorausgesetzten PUC Szenarios die Integrität und Vertraulichkeit der Programme und Daten eines Dienstbenutzers. Der Einsatz von zuverlässiger, manipulationsresistenter Hardware als Voraussetzung angenommen. Eine Java VM wurde für die Ausführung von verschlüsseltem Java Bytecode modifiziert. Weil sowohl die Integrität als auch die Vertraulichkeit der Programme nicht nur statisch sondern auch zur Laufzeit des Programms garantiert werden muss werden ungeschützte Speicherbereiche zur Laufzeit verschlüsselt und mit MACs versehen. Dies ermöglicht der VM die sichere Ausführung von Aufgaben, welche ihre eigenen sicheren Speicherressourcen übersteigen.

Im speziellen konnte im Rahmen dieses Proof of Concept gezeigt werden:

- Verschlüsselter Java Bytecode lässt sich für die Programmausführung segmentweise in einen sicheren Speicher entschlüsseln und ausführen.
- Der Stack einer Java Ausführung lässt sich verschlüsseln und für den Zugriff segmentweise in einen sicheren Speicher entschlüsseln.

Nicht abschliessend gezeigt werden konnten

- die segmentweise Verschlüsselung und Entschlüsselung des Stack für ein Programm mit mehreren Threads,
- die segmentweise Verschlüsselung und Entschlüsselung des Heap,
- das Erzeugen und Überprüfen von Hashwerten zur Wahrung der Integrität der Laufzeitdaten.

Realisierbarkeit eines PUC Rechendienstes
Ziel dieser Arbeit war, ein Proof of Concept für die vertrauliche Erbringung eines PUC Rechendienstes zu erbringen. Mit den umgesetzten Massnahmen allein kann aber die geforderte Vertraulichkeit noch nicht gewährleistet werden. Erst wenn auch gezeigt werden kann, dass die Integrität der Laufzeitdatenstrukturen in der vorgeschlagenen Weise realisiert werden kann, darf die Ausführung eines Programms auf der in dieser Arbeit beschriebenen Plattform als vertraulich und sicher angesehen werden.

Damit sind aber noch längst nicht alle Probleme im Zusammenhang mit der Realisierung eines PUC Rechendienstes gelöst. Die wichtige Frage nach der Sicherstellung der Authentizität sowohl der Benutzerprogramme und Daten als auch der Programmausgabe bleibt unbeantwortet. Die Frage nach der Zertifizierung der Anbieter ebenfalls. Auch sollte der Dienstanbieter auf die einmalige Ausführung des Benutzerprogramms mit anschliessender vollständiger Vernichtung aller Classfiles und Eingabedateien verpflichtet und überprüft werden können.

Zusätzlich dürfte die Etablierung eines Vertrauensverhältnisses zwischen Dienstbenutzern mit sensitiven Daten und Dienstanbietern eine der grossen nicht technischen Hürden des PUC Szenarios darstellen.

Tabelle 15: Cache Algorithmen für Modularisierung

Read	**Write**
Hit Benutze Datenkopie aus Cache	Hit *write back:* Es wird nur der Cache geschrieben. Der Datenblock wird erst in den Arbeitsspeicher zurückgeschrieben, wenn der Cache Block überschrieben werden soll.
Miss Lese Datenblock + MAC aus Arbeitsspeicher Überprüfe MAC Übergebe Datenblock an Cache	Miss *write allocate:* Der Datenblock wird zuerst in den Cache kopiert: Lese Datenblock + MAC aus Arbeitsspeicher Überprüfe MAC Übergebe Datenblock an Cache
	Zurückschreiben in Arbeitsspeicher Berechne MAC Schreibe Datenblock und MAC aus Cache in den Hauptspeicher zurück

7.2 Weiterführende Arbeiten

7.2.1 Modularisierung

Für eine optimale Performanz sollten die Funktionen für Verschlüsselung und Integritätsprüfung der Laufzeitdaten so nahe an der Hardware als möglich implementiert werden. Das hätte den zusätzlichen Vorteil, dass die sichere Ausführung eines Programms in ungeschützten Speicherbereichen nicht mehr auf die Java VM und Java Programme beschränkt wäre. Ebenfalls ist eine modulare Implementierung der Funktionalität anzustreben. In der für das Proof of Concept gewählten Implementierung innerhalb einer bestehenden VM waren dieser Modularisierung gewisse Grenzen gesetzt.

Vom Ablauf her entspricht das segmentweise Entschlüsseln und Verschlüsseln ungeschützter Speicherbereiche in einen sicheren Speicher den Caching Mechanismus eines Prozessors. Die entsprechenden Algorithmen sind in Tabelle 15 skizziert. Die Hashwertüberprüfung und Generierung sowie die Ver- und Entschlüsselung könnten also eng an diesen Mechanismus angebunden werden. Die schlüsselabhängigen Hashwerte könnten mit den Blöcken mitgeschrieben werden.

Der zusätzliche Speicheraufwand für die Hashes fällt nicht allzusehr ins Gewicht: Bei typischen Cache Grössen von 16 KiByte (L1) und 512 KiByte (L2) kann das Lesen/Schreiben vom und zum Cache z.B. in 4 KiByte Blöcken geschehen. Die Länge eines Hashwerts sollte dabei mindestens 128 Bit betragen. Der Overhead durch das Mitspeichern der Hashes beträgt also $16/(4096 - 16) \simeq 0.003$.

7.2.2 PUC in Supernetzen

Ein Ansatz im Utility Computing ist es, mehrere Rechner zu einem grösseren, virtuellen Server zusammenzuschalten, der die von einem Dienstbenutzer benötigte

Rechendienstleistung erbringen kann[19]. Es wäre interessant, die Möglichkeiten, die sich in einem solchen Verbund von mehreren Rechnern mittels der in Abschnitt 2.5 beschriebenen Secure Multy-Party Computation in Bezug auf die sichere Ausführung beliebiger Benutzerprogramme ergeben, zu untersuchen.

[19]Vgl. Caronni et al.: Supernets and snHubs: A Foundation for Public Utility Computing [8].

A Original Problem Statement

Introduction

Outline

Imagine a scenario where you want to out-source as much of your infrastructure as you possibly can. The concept is called Public Utility Computing (PUC), and is pursued in three forms.
1) Network infrastructure,
2) Storage infrastructure, and
3) Computing infrastructure.

Problem Statement

The work proposed here aims at a proof of concept for security-measures that make the third point worthwhile. The problem is that, if you out-source your computing infrastructure, you expose both your programs and the data they process to the compute service provider. So, what can we do against this exposure?

Task

As a first prototypical proof of concept, we want to change a Java Virtual Machine (written in C) such that the bytecode verifier and interpreter decrypts its methods and their data as it accesses them for execution purposes. To this end, an "encryptor" for Java Bytecode and possibly also input data is needed, and an open source VM needs to be extended with the ability to decrypt and parse the bytecode on the fly.

The task of the student is split in three major subtasks: Specification, Implementation, and Validation.

Specification

Based on previous (unpublished) work at TIK and SUN Labs, a suitable virtual machine (JamVM 1.1.2 by Robert Laugher) was chosen and the basis for a possible design was laid. Using these results, the knowledge of the Java bytecode specification and insights into the chosen Java VM, a more detailed specification of where to implement which functionality in which way within the "de-/encryptor" tools and the Java VM will have to be written and discussed with the supervisors.

There are more than one suitable variants for the implementation of the crypto functionality as it heavily depends on the granularity of encryption units (i.e. single bytecode commands, code blocks, methods, classes, or application). All variants have to be considered in the specification phase, however the focus for the implementation will be on a single variant.

Implementation

Implementation will be done in Java and C on Linux. The code should be documented well enough such that another student can easily understand the core

functionality and extend it. The cryptographic algorithms used will be defined by the supervisors.

Validation

As a validation of the implementation, some sample programs and data that are "out-sourced" in our scenario have to be written or generated. They are first transformed with the "encryptor" tool, then decrypted and run by the extended VM and finally the result data is being decrypted and checked for correctness. Tests must be done to state the correctness of the implementation, and to cleary state to which extent and under what assumptions confidentiality of the data and calculation code (i.e. Java program) can be guaranteed.

Deliverables

The following results are expected:

1. *Specification* Before implementation can start, an exact specification of the algorithms and changes to the Java VM must be written down and discussed with the supervisors.
2. *Implementation of de-/encryptor tools* A tool for de-/encryption of data and Java Byte Code will be written. The code must be correct and feature short and precise comments. It must be usable as a proof of concept.
3. *Extensions to the Java VM* All extensions to the Java VM (written in C) have to be coded and well documented. A short "installation"-readme must be provided along with the code.
4. *Thesis documentation* A concise description of the work conducted in this thesis (task, related work, environment, design decisions and functionality of delivered implementations, results and outlook).

Further optional components are:

- An implementation based on a different granularity of encryption.
- Proposition and/or implementation of specialized cryptographic algorithms that are beneficial for this application.
- A conference paper of about 5-8 pages that summarizes the results of this thesis.
- Generalizations of the algorithms and application to other problem domains.

Documentation and Presentation

A documentation that states the steps conducted, lessons learnt, algorithm design and implementation, major results and an outlook on future work and unsolved problems has to be written. The code should be documented well enough such that it can be extended by another developer within reasonable time. At the end of the thesis, a presentation will have to be given at TIK that states the core tasks and results of this thesis. If important new research results are found, a paper might be written as an extract of the thesis and submitted to a computer network and security conference.

Dates

This thesis starts end of April, 2004 and is finished by end of October, 2004. It lasts approximately one semester as the thesis is done part-time. The student is expected to spend 3-4 person months on the thesis.

An intermediate informal presentation for Prof. Plattner and the supervisors will be scheduled for a date about 5-8 weeks after the thesis has started.

A final presentation at TIK will be scheduled close to the completion date of the thesis.

Informal meetings with the supervisors will be announced and organized on demand.

Supervisors

Dr. Germano Caronni, germano.caronni@inf.ethz.ch, +41 1 632 7326, IFW C41.1
Thomas Dübendorfer, duebendorfer@tik.ee.ethz.ch, +41 1 632 7196, ETZ G64.1

Literatur

[1] *Validating a High-Performance, Programmable Secure Coprocessor*, October 1999.

[2] Bouncy Castle Homepage. http://www.bouncycastle.org, 2004.

[3] GNU Classpath Homepage. http://www.classpath.org, 2004.

[4] OpenSSL Project Homepage. http://www.openssl.org, 2004.

[5] J. Algesheimer, C. Cachin, J. Camenisch, and G. Karjoth. Cryptographic security for mobile code. Technical Report RZ 3302 (# 93348), IBM, 2000.

[6] T. W. Arnold and L. P. V. Doorn. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. *IBM J. RES. & DEV.*, 48, May/July 2004.

[7] C. Cachin, J. Camenisch, J. Kilian, and J. Muller. One-round secure computation and secure autonomous mobile agents. In *LNCS, Automata, Languages and Programming*, pages 512–523, 2000.

[8] G. Caronni, T. Curry, P. S. Pierre, and G. Scott. Supernets and snhubs: A foundation for public utility computing. Technical Report TR-2004-129, SUN Microsystems Laboratories, 2004.

[9] A. Chander, J. Mitchell, and I. Shin. Mobile code security by java bytecode instrumentation. In *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX Ó1*, volume 2, pages 27–40, 2001.

[10] O. Goldreich. *Foundations of cryptography*, volume two Basic Applications. Cambridge University Press, 1st edition, 2004.

[11] IBM Corporation, Department VM9A., Charlotte, NC 28262-8563. *IBM PCI Cryptographic Coprocessor*, sixth edition, 2002. IBM Cryptographic Products. IBM PCI Cryptographic Coprocessor. General Information Manual.

[12] R. Laugher. Jamvm 1.2.0 Homepage. http://jamvm.sourceforge.net, 2004.

[13] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification.* Sun Microsystems, Inc., 2nd edition, 1996. Specification of the Java Virtual Machine.

[14] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification.* Sun Microsystems, Inc., 2nd edition, 1999. Specification of the Java Virtual Machine.

[15] M. Mambo, T. Murayama, and E. Okamoto. A tentative approach to constructing tamper-resistant software. In *Proceedings of the 1997 workshop on New security paradigms*, pages 23–33. ACM Press, 1997.

[16] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 5th edition, 2001. Cryptography and its applications.

[17] Michael Kallus. Utility Computing in Warteposition. CIO: IT Strategie für Manager, http://www.cio.de/index.cfm?PageID=310&cat=det&maid=5699, July 2004.

[18] National Institute of Standards and Technology, Information Technology Laboratory, Gaithersburg, MD 20899-8900. *FIPS PUB 140-2, Security Requirements for Cryptographic Modules*, 2001.

[19] J. Nickerson, S. Chow, H. Johnson, and Y. Gu. The encoder solution to implementing tamper resistant software. CERT/IEEE Information Survivability Workshop, Vancouver, 2001.

[20] G. Nolan. *Decompiling Java.* Apress, 1st edition, 2004.

[21] T. Sander and C. F. Tschudin. Protecting mobile agents against malicious hosts. *Lecture Notes in Computer Science*, 1419:44–64, 1998.

[22] B. Schneier. *Angewandte Kryptographie: Protokolle, Algorithmen und Sourcecode in C.* Addison-Wesley, 1st edition, 1996.

[23] I. Shin and J. C. Mitchell. Java bytecode modification and applet security. Technical report.

[24] M. A. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. Technical Report RC 21102, IBM T.J. Watson Research Center, Yorktown Heights NY 10598, USA, 1998.

[25] S. Smith. Secure coprocessing applications and research issues. Technical Report Los Alamos Unclassified Release LAUR -96-2805, Los Alamos National Laboratory, August 1996.

[26] S. W. Smith, E. R. Palmer, and S. Weingart. Using a high-performance, programmable secure coprocessor. In *Fincancial Cryptography*, pages 73–89, 1998.

[27] B. S. Yee. A sanctuary for mobile agents. In *Secure Internet Programming*, pages 261–273, 1999.