Machbarkeitsstudie Gruppensignaturverfahren

Studienarbeit

HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT II
INSTITUT FÜR INFORMATIK

eingereicht von: Tobias Mühl

Betreuer: Dr. Wolf Müller

eingereicht am: 03.12.2012

Inhaltsverzeichnis

1. Abstract

Viele sicherheitsrelevanten Anwendungen in der Informatik werden mit Hilfe bekannter Public-Key-Verfahren realisiert. Ein Berechtigungs- oder Authentizitätsnachweis ist in den meisten Verfahren jedoch nur unter der Offenlegung der Identität des Teilnehmers möglich. Um beispielsweise Zugang zu geschützten Ressourcen zu erlangen, weist sich ein Anfragender aus und legt damit seine Identität offen.

Mit Hilfe von Gruppensignaturverfahren ist es möglich, die Mitgliedschaft in einer Gruppe nachzuweisen, ohne die Identität preiszugeben. Auf der anderen Seite ist es im Bedarfsfall möglich, zu einer gegebenen Signatur die Identität eines Mitglieds aufzudecken.

In dieser Arbeit wird das in [CG04] beschriebene Verfahren in seinen verschiedenen Ausbaustufen implementiert und auf Praxistauglichkeit untersucht. Die verschiedenen Ausbaustufen werden untereinander verglichen, insbesondere nach Aufwands- und Leistungskriterien. Ebenso werden Probleme und Schwachstellen angesprochen, die verfahrens- oder implementierungsbedingt sein können. Die Arbeit basiert im Wesentlichen auf dem genannten Paper von Camenisch und Groth [CG04] sowie auf der Diplomarbeit von Torsten Zimmermann [Zim09]. Erstgenannte Arbeit stellt die Algorithmen für die Verfahren vor. In [Zim09] werden die drei Ausbaustufen zusammen mit anderen Verfahren auf theoretischer Ebene auf Effizienz und Geschwindigkeit hin untersucht und verglichen. In der vorliegenden Arbeit sollen nun die praktischen Auswirkungen beschrieben werden.

2. Einleitung

2.1. Übersicht Gruppensignaturen

Digitale Signaturen erlauben es, die Urheberschaft eines Dokuments zu bestimmen. Der Urheber erstellt zu einem Dokument eine digitale Signatur und verknüpft dadurch seine Identität mit dem Dokument. Es ist im Folgenden jedem Dritten möglich, die Urheberschaft zu verifizieren, indem er die Signatur überprüft. Auf der anderen Seite ist es dem Ersteller der Signatur nicht möglich, seine Urheberschaft abzustreiten. Diese Eigenschaft einer digitalen Signatur nennt man Nichtabstreitbarkeit.

Im Unterschied zu herkömmlichen Signaturverfahren wird bei Gruppensignaturen nachgewiesen, dass der Ersteller eines Dokuments Mitglied einer Gruppe ist: Gruppensignaturverfahren erlauben den Mitgliedern im Namen der Gruppe Signaturen zu erstellen. Sie weisen damit ihre Mitgliedschaft nach, bleiben innerhalb der Gruppe jedoch anonym und agieren unter dem Pseudonym der Gruppe. Außenstehenden ist es möglich, diese Signaturen auf Echtheit zu prüfen und damit zu verifizieren. Nicht möglich ist es jedoch, auf Basis einer gegebenen Signatur ein einzelnes Mitglied der Gruppe zu identifizieren. Darüber hinaus gibt es einen Gruppenmanager, dem es möglich ist, im Bedarfsfall eine Gruppensignatur zu „öffnen“ und damit die Anonymität des Urhebers innerhalb der Gruppe aufzuheben.

Ein Anwendungsfall wäre ein Raum in einem Unternehmen, zu dem nur bestimmte Angestellte Zutritt haben. An der Zugangstür authentisieren sich die berechtigten Angestellten mittels einer Chipkarte. Allerdings soll die Identität des authentifizierten Mitarbeiters geschützt werden. Es darf durch Auslesen der Chipkarte keine Identifizierung möglich sein. Bei Missbrauch übernehmen der Betriebsrat und die Geschäftsführung gemeinsam die Rolle des Gruppenmanagers. Sie können gemeinsam die im Chipkartenlesegerät signierten Anmeldungen auslesen und die Signatur zur Missbrauchszeit öffnen. Dadurch wird die Identität des entsprechenden Mitarbeiters offengelegt.

2.1.1. Eigenschaften von Gruppensignaturverfahren

Das erste Gruppensignaturverfahren wurde 1991 von Chaum und van Heyst in [CvH91] vorgestellt. Seitdem gab es mehrere Verfahren, die sich in ihrer Leistungsfähigkeit unterscheiden.

Die Verfahren lassen sich grob danach kategorisieren, ob sie eine *statische*, *halbdynamische* oder *volldynamische* Gruppenstruktur besitzen. Statische Gruppensignaturverfahren besitzen eine feste Anzahl an Mitgliedern. Um ein Mitglied in die Gruppe aufzunehmen oder aus der Gruppe zu entfernen, ist es notwendig, das Schema mit neuen Schlüsseln erneut aufzusetzen. Ist ein Verfahren *volldynamisch*, so ist es möglich, zur Laufzeit Mitglieder aufzunehmen und auszuschließen. Wenn Mitglieder zur Laufzeit entweder ausgeschlossen oder aufgenommen werden können, so spricht man von einem *halbdynamischen* Gruppensignaturverfahren.

Weitere Merkmale zur Unterscheidung verschiedener Gruppensignaturverfahren sind nach [BMW03] *Volle-Rückverfolgbarkeit* und *Volle-Anonymität* bzw. *Anonymität*.

Volle-Rückverfolgbarkeit: Ohne das Mitgliedsgeheimnis eines Mitglieds darf es nicht möglich sein, eine Signatur in dessen Namen zu erstellen, die beim Öffnen auf dieses Mitglied zurückzuführen ist. Das gilt insbesondere auch dann, wenn eine beliebige Zahl Mitglieder der Gruppe sowie der Gruppenmanager zusammenarbeiten. Auf der anderen Seite darf keine Signatur im Namen der Gruppe erstellt werden, die nicht durch Offenlegung zweifelsfrei einem Mitglied zuzuordnen ist.

Volle-Anonymität: Ein Angreifer, ungeachtet ob er Mitglied der Gruppe ist oder nicht, soll nicht in der Lage sein, zwischen zwei Signaturen zu unterscheiden. Weder darf er zwei Signaturen verschiedener Mitglieder als solche erkennen, noch zwei Signaturen ein und desselben Mitglieds miteinander verknüpfen können. Selbst wenn das Geheimnis eines Mitglieds bekannt ist, darf es dem Angreifer nicht gelingen, eine Signatur diesem Mitglied zuzuordnen. Ist das gewährleistet, besitzt ein Gruppensignaturverfahren *Volle-Anonymität*.

Anonymität: Als abgeschwächte Form der *Vollen-Anonymität* fungiert die Eigenschaft der *Anonymität*. Sie wird in [CG04] eingeführt, um ein Verfahren mit *Markierungsfähigkeit* zu ermöglichen, in dem Signaturen ausgeschlossener Mitglieder im Nachhinein ihre Gültigkeit verlieren.

Eine formale Definition von *Voller-Rückverfolgbarkeit*, *Voller-Anonymität* und *Anonymität* findet sich in [CG04, S. 3-4].

Eine andere Definition der Anforderungen an ein Gruppensignaturverfahren sind die Eigenschaften: *Korrektheit*, *Anonymität*, *Nichtverknüpfbarkeit*, *Schwache Unfälschbarkeit*, *Starke Unfälschbarkeit*, *Rückverfolgbarkeit* und *Koalitionsresistenz*. Sie werden in [ACJT00] eingeführt und erläutert. Bellare, Micciancio und Warinschi [BMW03] argumentieren jedoch, dass diese Eigenschaften bereits in *Voller-Rückverfolgbarkeit* und *Voller-Anonymität* enthalten sind.

2.1.2. Aktuelle Gruppensignaturverfahren

Eines der nach wie vor wichtigsten Gruppensignaturverfahren ist das halbdynamische von [ACJT00], das als Grundlage für viele andere Verfahren gilt. Daneben werden in [Zim09] noch das generische Verfahren [Ge05] sowie die auf *bilinearen Gruppen* und *Pairings* basierenden Verfahren [BBS04] und [DP06] genannt. Die Verfahren unterscheiden sich in ihren Eigenschaften, sind aber je nach Anwendungsfall allesamt als effektiv zu bezeichnen. Ein Vergleich findet sich in [Zim09, S. 107ff]. Dort wird ebenfalls gezeigt, dass die Gruppensignaturschemata nach [CG04] zurzeit zu den flexibelsten und effizientesten gehören.

Eines der neuesten Verfahren wurde von [LPY] vorgestellt. Es verbessert u. a. den Revocation-Mechanismus. Teilnehmer müssen im Vergleich zu dem Verfahren [CG04] ihre privaten Schlüssel nicht aktualisieren, sobald ein Mitglied ausgeschlossen wurde.

2.2. Aufgabenstellung

Auch wenn es inzwischen zahlreiche Arbeiten zu Gruppensignaturverfahren gibt, so sind diese fast ausschließlich auf dem theoretischen Gebiet zu finden[1].

Aufgabe dieser Arbeit ist es, ein Gruppensignaturverfahren zu implementieren und auf seine Praxistauglichkeit hin zu bewerten.

Implementiert wurde das Gruppensignaturverfahren von Camenisch und Groth aus [CG04], das bereits in [Zim09] genauer untersucht wurde. Es bietet sich vor allem an, weil es zum gegenwärtigen Zeitpunkt eines der leis-

[1] Für Beispiele theoretischer Arbeiten siehe Quellenverzeichnis. Arbeiten, die sich praktisch mit Gruppensignaturen auseinandersetzen, sind vor allem im technischen Umfeld angesiedelt: Ein Beispiel hierfür ist [Pos10]

tungsfähigsten Verfahren ist[2] und in drei Ausbaustufen vorliegt. Durch Umsetzung aller drei Schemata konnte das Verfahren abgestuft nach verschiedenen Leistungsmerkmalen bewertet werden.

Zur Untersuchung der Tauglichkeit für den praktischen Einsatz sollen in der vorliegenden Arbeit zum einen Probleme bei der Umsetzung in einer aktuellen Hochsprache - in diesem Fall Java - dokumentiert werden. Zum anderen - und hierin besteht die hauptsächliche Aufgabe der Arbeit - soll die Performance der Implementierung auf aktueller Hardware herangezogen werden. Dazu wurden Messungen der Laufzeiten der verschiedenen Funktionen des Verfahrens angestellt.

Da für die Messungen ein Monitoring der Anwendung nötig ist, wird zugleich als Nebenaspekt auf die Mechanismen zur Laufzeitüberwachung der Anwendung eingegangen. Dabei werden Design-Patterns objekt-orientierter Programmierung verwendet und beschrieben.

Das Gruppensignaturverfahren und das Monitoring wurden in Java entwickelt. Dabei kamen verschiedene Werkzeuge (z. B. Maven) und Frameworks (z. B. Spring) zum Einsatz. Diese sind jedoch nicht Gegenstand der vorliegenden Arbeit und werden daher hier nicht beschrieben.

[2] siehe Vergleich mit anderen Verfahren in [Zim09]

3. Gruppensignaturverfahren nach Camenisch und Groth

Jan Camenisch und Jens Groth veröffentlichten 2004 [CG04] ein eigenes Gruppensignaturverfahren, das auf dem Verfahren von Ateniese et al. [ACJT00] basiert. In drei verschiedenen Ausbaustufen (Σ_1, Σ_2 und Σ_3) verbessert es das Verfahren [ACJT00] vor allem hinsichtlich der Geschwindigkeit[1]. Zudem ist [ACJT00] halbdynamisch, während [CG04] mit den Ausbaustufen Σ_2 und Σ_3 zwei volldynamische Gruppensignaturverfahren vorstellen. Dafür ist jedoch in Σ_2 und Σ_3 die *Volle-Rückverfolgbarkeit* eingeschränkt und in Σ_3 zusätzlich die *Volle-Anonymität*. Diese Einschränkungen sind aber nicht unbedingt von Nachteil; sie sind sogar bewusst gewählt. So besteht der Unterschied der beiden höheren Ausbaustufen gerade darin, dass die *Volle-Anonymität* in Σ_2 durch die Markierungsfähigkeit in Σ_3 ersetzt wird. Dadurch ist es in Σ_3 möglich, bereits erstellte Signaturen von ausgeschlossenen Mitgliedern im Nachhinein für ungültig zu erklären.

In [Zim09] findet sich ein ausführlicher Sicherheits- und Effizienzvergleich[2] u. a. der Verfahren Σ_1, Σ_2, Σ_3 und [ACJT00]. Die Tabelle 3.1 ist [Zim09, S. 111] entnommen.

3.1. Beschreibung der Verfahren

Entsprechend der Definition von Gruppensignaturen[3] existieren in allen drei Ausbaustufen des Verfahrens Mitglieder, Verifizierer ein *Trusted-Public-Directory* (*TPD*) und ein Gruppenmanager GM.

Dem Gruppenmanager ist es möglich, Signaturen zu öffnen, d. h. zu deanonymisieren. Mit seiner Hilfe ist es in Σ_2 und Σ_3 außerdem möglich, neue Mitglieder in die Gruppe aufzunehmen. Analog dazu kann er in Σ_2 und Σ_3

[1] „Our scheme is considerably faster than the state of the art scheme in [ACJT00]. Moreover, in our scheme the protocol to join the group only takes two rounds.“ [CG04, S. 1]

[2] Siehe: [Zim09, S. 110-116]

[3] [Zim09, S. 29ff]

	Gruppenstruktur	$\mathcal{TTP}^a$	$\mathcal{MM}$,$\mathcal{D}^b$	Volle-Anon.	Volle-Rück.
Σ_1	statisch	✓		✓	✓
Σ_2	volldynamisch	×	✓	✓	(✓)
Σ_3	volldynamisch	×	✓	(✓)	(✓)
[ACJT00]	halbdynamisch	×	✓	✓	✓

[a] *Trusted-Third-Party*

[b] *Mitgliedschaftsmanager* bzw *Deanonymisierer*. Gibt an, ob die Aufgaben des Gruppenmanagers auf einen *Mitgliedschaftsmanager* und einen *Deanonymisierer* aufgeteilt werden können

Tabelle 3.1.: Vergleich von Struktur- und Sicherheitseigenschaften aus [Zim09, S. 111]

bestehende Gruppenmitglieder ausschließen oder in Σ_3 deren bestehende Signaturen sogar für ungültig erklären (Markierungsfähigkeit).

In dem *Trusted-Public-Directory* werden die öffentlich zugänglichen Informationen verwaltet. Das ist vor allem der öffentliche Schlüssel. Zudem finden sich hier in Σ_2 und Σ_3 auch die Informationen für die übrigen Mitglieder einer Gruppe, um nach dem Ausschluss eines Mitglieds ihre Zertifikate anzupassen. In Σ_3 enthält das *TPD* zudem eine *Certificate-Revocation-List* (*CRL*), eine Liste mit Signaturen, die für ungültig erklärt wurden.

Die Rolle des Verifizierers kann von jeder beliebigen Person übernommen werden, unabhängig davon, ob sie Mitglied der Gruppe ist oder nicht. Die Aufgaben des Verifizierers wurden in der Implementierung für diese Arbeit daher auf das *Trusted-Public-Directory* übertragen und können von jedem aufgerufen werden.

Wie im vorangegangenen Teil erwähnt, ist das Verfahren in seiner ersten Ausbaustufe Σ_1 statisch. Es wird einmal mit einer festen Gruppengröße initialisiert. Sollen neue Mitglieder in die Gruppe aufgenommen oder bereits zugehörige ausgeschlossen werden, so muss das Verfahren erneut aufgesetzt werden. Alle bestehenden Signaturen sind für das neue Schema nicht mehr gültig. Zudem benötigt das Verfahren zusätzlich zum *Trusted-Public-Directory* als neutrale Instanz eine *Trusted-Third-Party* (*TTP*), die die Initialisierung des Verfahrens übernimmt. Insbesondere fertigt sie dabei die Zertifikate der Mitglieder inklusive deren privater Schlüssel an. Die Rolle der *TTP* darf in keinem Fall mit der des Gruppenmanagers zusammengelegt werden.

Die Ausbaustufe Σ_2 ist ein volldynamisches Verfahren. Die Anzahl der Mitglieder wird nicht mehr beim Aufsetzen der Gruppensignatur festgelegt. Stattdessen können zur Laufzeit neue Mitglieder durch den Gruppenmanager aufgenommen und bestehende ausgeschlossen werden. Die Aufnahme neuer Mitglieder passiert in einem Verfahren, an dem der Gruppenmanager und das neue Mitglied beteiligt sind. Die privaten Schlüssel werden bei dem Verfahren lokal bei dem Mitglied erstellt. Daher besitzt der Gruppenmanager keine Informationen, mit denen er im Namen des Mitglieds Signaturen erzeugen könnte (*Volle-Rückverfolgbarkeit*). Aus diesem Grund wird ab der zweiten Ausbaustufe keine *Trusted-Third-Party* mehr benötigt.

Zusätzlich zu den Erweiterungen in Σ_2 wird in Σ_3 die *Volle-Anonymität* zugunsten einer Markierungsfähigkeit aufgegeben. Durch den Gruppenmanager können bestehende Signaturen eines Mitglieds markiert und damit für ungültig erklärt werden. Das wird durch die Veröffentlichung eines dem Gruppenmanager bekannten Teils des Mitgliedszertifikats in der *Certificate-Revocation-List* realisiert.

3.1.1. Verwendete Algorithmen

Die Sicherheit des Gruppensignaturverfahrens von Camenisch und Groth basiert auf dem diskreten Logarithmus-Problem. Es verwendet für die Signierung das in vorhergehenden Arbeiten von Camensich zusammen mit Lysyanskaya entwickelte Signaturverfahren: das Camenisch-Lysyanskaya-Signaturverfahren[4] (CL-Verfahren). Die Verschlüsselung wird durch eine modifizierte Variante des ElGamal-Verschlüsselungsverfahrens realisiert.

Spezieller RSA-Modul

Ein spezieller RSA-Modul $n = p \cdot q$ ist das Produkt zweier sicherer Primzahlen p und q.

Eine Primzahl p bezeichnet man als sicher, wenn sie sich in $p = 2p' + 1$ zerlegen läßt, wobei p' ebenfalls eine Primzahl ist (p' wird auch *Sophie-Germain-Primzahl* genannt).

Camenisch-Lysyanskaya-Signaturverfahren

Das CL-Verfahren bietet die Möglichkeit *Commitments* zu signieren. Damit können Teilnehmer Dokumente signieren, ohne deren Inhalt zu erfahren. Es

[4] siehe [CL02]

basiert auf der Erstellung eines speziellen RSA-Moduls mit quadratischen Restklassen. Es gibt einen privaten und einen öffentlichen Schlüssel.

Nur der Besitzer des privaten Schlüssels kennt die Faktorisierung des RSA-Moduls n. Es ist damit nur ihm möglich, effizient das multiplikativ Inverse e^{-1} mod n zu einer Primzahl e zu errechnen. Er bildet einen Ausdruck a, in dem das zu signierende *Commitment* ein Bestandteil ist. Es ist ihm nun möglich, ein y zu finden, so dass $y \equiv a^{e^{-1}} \bmod n$. Ein Verifizierer kann die Gültigkeit der Signatur überprüfen, indem er den Ausdruck $y^e \equiv a \bmod n$ auf seine Richtigkeit testet.

ElGamal-Verschlüsselungsverfahren

Das ElGamal-Verschlüsselungsverfahren ist ein asymmetrisches Kryptoverfahren und besitzt daher einen öffentlichen und einen privaten Schlüssel. Es basiert auf dem Diffie-Hellmann-Verfahren.

Das Verfahren benötigt eine geeignete Primzahl p und ein Generator-Element $g \in \{1, \ldots, p-1\}$. Des Weiteren wird eine Zufallszahl $1 \leq a \leq p-1$ erzeugt und der Ausdruck $A = g^a$ mod p gebildet. Der öffentliche Schlüssel besteht nun aus (p, g, A), der private Schlüssel aus a. Aufgrund der Diskerter-Logarithmus-Annahme ist a mithilfe des öffentlichen Schlüssels nicht effizient berechenbar.

Zur Verschlüsselung einer Nachricht $m \in \{1, \ldots, p-1\}$ wird wiederum eine Zufallszahl $1 \leq b \leq p-1$ gewählt. Die verschlüsselte Nachricht ist nun das Tupel (B, c) mit $B = g^b$ mod p und $c = A^b m$ mod p.

Zur Entschlüsselung wird $B^{p-1-a}c$ mod p errechnet. Das Ergebnis ist die Nachricht m.

4. Implementierung

Das Gruppensignaturverfahren in seinen drei Ausbaustufen wurde in Java als Maven-Projekt umgesetzt. Die Implementierung verwendet keine zusätzlichen Crypto-Bibliotheken. Die Mathematik wurde mit Standard Java-Techniken realisiert, insbesondere kam die Klasse `BigInteger` zum Einsatz. Bei der Umsetzung wurde auf Modularisierung Wert gelegt. Das Kernfahren wurde in einzelnen Bausteinen (RSA, CL-Signaturverfahren[1], ElGamal-Verschlüsselungsverfahren) implementiert und zusammengesetzt. Es enthält die gemeinsamen Grundlagen aller drei Ausbaustufen. Σ_1, Σ_2 und Σ_3 wurden darauf aufbauend entwickelt. Darüber hinaus wurde eine strenge Trennung des Verfahrens selbst (der *Business-Logik*) und der Messinstrumente angestrebt.

Am Ende bestand das Projekt aus folgenden Maven Modulen:

opengroupsig-core	Das Kernmodul, in dem die Verfahren umgesetzt sind, die die Basis bilden: CL-Signaturverfahren, ElGamal-Verschlüsselungsverfahren, Spezielle-RSA-Modul-Generierung.
opengroupsig-scheme1	Das Verfahren in der Ausbaustufe Σ_1
opengroupsig-scheme2	Das Verfahren in der Ausbaustufe Σ_2
opengroupsig-scheme3	Das Verfahren in der Ausbaustufe Σ_3
measuring-tools	Monitoring-Tools und -Klassen und deren JMX-Integration
opengroupsig-example	Die Beispielanwendung der implementierten Gruppensignaturverfahren und Monitoring mithilfe der Klassen aus measuring-tools

[1] Camenisch-Lysyanskaya-Signaturverfahren

4.1. Erzeugung eines speziellen RSA-Moduls und der ElGamal-Verschlüsselungsparameter

Das Erzeugen des speziellen RSA-Moduls und der Parameter für das ElGamal-Verschlüsselungsverfahren spielen eine zentrale Rolle beim Aufsetzen des Verfahrens in einer der drei Ausbaustufen sowie beim Aufnehmen von Mitgliedern in Σ_2 und Σ_3. Es müssen jeweils Primzahlen mit speziellen Eigenschaften gefunden werden. Zur Generierung der entsprechenden Primzahlen kommen probabilistische Algorithmen zum Einsatz, die jedoch rechen- und damit zeitintensiv sind.

Ein spezieller RSA-Modul besteht aus zwei sicheren Primzahlen. Eine Primzahl p gilt genau dann als sicher, wenn sie sich als $p = 2p' + 1$ schreiben lässt, wobei p' selbst eine Primzahl ist. Die speziellen Primzahlen werden durch einen Las-Vegas-Algorithmus gefunden: Es werden so lange Primzahlen erzeugt, bis eine die gewünschten Eigenschaften besitzt.

Für das modifizierte ElGamal-Verschlüsselungsverfahren, das in dem Gruppensignaturverfahren Verwendung findet, werden zwei Primzahlen Q und P benötigt, so dass beide die jeweils gewünschte Länge l_Q bzw. l_P besitzen und es gilt: $Q|P-1$.

Um zwei solche Primzahlen zu finden, wird zuerst Q in geeigneter Länge erzeugt. Dann werden weitere Primzahlen generiert und zusammen mit Q zu dem Produkt P' multipliziert, bis die Länge von $P'+1$ genau der Länge l_P entspricht und $P'+1$ eine Primzahl ist. Wächst P' dabei über die Länge l_P hinaus, so startet der Algorithmus von vorne.

```
def function GeneratePrime(bitLength):
  return Primzahl der Länge bitLength
def function Length(number):
  return Anzahl der Stellen der Zahl number im Dualsystem
def function Random(min, max):
  return eine Zufahlszahl z, so dass min <= z <= max

loop
  Q = GeneratePrime(lQ)
  P' = Q
  while Length(P' + 1) < lP
    length_a = Random(3, lP - Length(P'))
    a = GeneratePrime(length_a)
```

```
      P' = P' * a
      if Length(P' + 1) = lP
        if P' + 1 is prime
          P = P' + 1
          return Q, P
        end if
      end if
    end while
  end loop
```

Listing 4.1: Las-Vegas-Algorithmus zum Finden geeigneter Primzahlen P und Q für das ElGamal-Verschlüsselungsverfahren in Pseudocode

Da die Ausführung dieser Algorithmen unter den gegebenen Schlüssellängen mitunter Stunden oder sogar Tage dauerte, wurden sie aus den Messungen herausgenommen und die Gruppensignaturen mit vorberechneten speziellen RSA-Moduli und ElGamal-Parametern ausgeführt.

4.2. Monitoring

Die Kern-Implementierung des Gruppensignaturverfahrens soll durch das Monitoring nicht beeinträchtigt werden. Monitoring und Business-Logik sind strikt getrennt. Das Ziel dabei ist, die eigentliche Anwendung, die gemessen werden soll, durch die Messinstrumente im besten Fall gar nicht zu beeinflussen. Zum einen, um das Messergebnis nicht zu verfälschen, zum anderen damit die Anwendung wiederverwendbar bleibt und schlussendlich, um einer sauberen[2] Software-Architektur gerecht zu werden.

Erreicht wurde das durch Verwendung zweier Prinzipien[3]:

- *Dependency Injection* (DI): ist ein Design-Pattern in der Software-Architektur. Es minimiert die Abhängigkeiten der einzelnen Komponenten einer Software. Die einzelnen Komponenten, in diesem Fall Java-Klassen, sind nicht von der Implementierung der von ihnen benötigten anderen Klassen abhängig, sondern von Schnittstellen. Die Abhängig-

[2] „sauber“ meint in diesem Fall vor allem wartbar, erweiterbar, übersichtlich und wenig fehleranfällig

[3] auch Design-Pattern genannt

keiten werden zur Laufzeit „injiziert“.[4]. Das Framework *Spring*[5] und sein *IoC-Container* wurden verwendet, um zur Laufzeit die Abhängigkeiten aufzulösen.

- *Aspekt Orientierte Programmierung* (AOP): ist ein Programmieransatz, bei der modulübergreifende Prozesse von der eigentlichen Business-Logik getrennt werden[6]. Beispiele hierfür sind Logging, Sicherheitsaufgaben und Monitoring. Diese Prozesse werden Aspekte genannt und an mehreren Stellen der Anwendung „angeheftet“.

Durch *Dependency Injection* ist es möglich, die zu überwachenden Klassen im Gruppensignaturverfahren durch Proxys[7] zu ersetzen ohne den Quellcode des Verfahrens selbst zu ändern. Für jede zu überwachende Klasse wird eine Proxy-Klasse erstellt, die von der überwachten Klasse erbt. Die Methodenaufrufe, deren Laufzeit gemessen werden soll, werden von der Proxy-Klasse überschrieben und um den Aufruf der Parent-Methode wird der Aspekt implementiert, in diesem Fall das Monitoring.

```
public class AnyClass {
  public void doSomething() {
    // does something...
  }
}
```

Listing 4.2: Beispiel einer Klasse mit einer Methode, die überwacht werden soll

Die zugehörige Proxy-Klasse sieht wie folgt aus:

```
public class AnyClassProxy extends AnyClass {
  @Override
  public void doSomething() {
    // some monitoring
    super.doSomething();
    // some further monitoring
```

[4]Das Prinzip ist vielerorts erklärt, weswegen hier auf eine eingehende Beschreibung verzichtet wird. Siehe beispielsweise: [Fow12] oder [Mar09, S. 157]

[5]siehe http://www.springsource.org

[6]*cross-cutting concerns*

[7]Eine Definition von Proxy-Klassen liefert [Fow03, S.203ff]: „The key to the virtual proxy is providing a class that looks like the actual class you normally use but that actually holds a simple wrapper around the real class.“ Siehe ebenfalls: [ES10, S. 75-77]

```
    }
  }
```

Listing 4.3: Proxy für die Klasse `AnyClass`

Das führt jedoch zu zwei Einschränkungen: 1. Die zu überwachenden Klassen müssen vererben. Sie dürfen demnach nicht als *final* deklariert sein. 2. Die Methoden, deren Laufzeit gemessen werden soll, müssen mindestens als *protected* deklariert sein, damit sie für die Kind-Klasse sichtbar sind und überschrieben werden können.

Um die Performance der Gruppensignaturverfahren zu bestimmen, wurde die Laufzeit der wichtigen Methoden durch die Proxys gemessen. Die Informationen wurden an eine zentrale Monitorklasse weitergegeben, den *OpenGroupSigMonitor*. Der *OpenGroupSigMonitor* stellte die Schnittstelle zu JMX bereit. Somit konnten die gemessenen Ausführungszeiten jederzeit bei laufender Anwendung beobachtet werden.

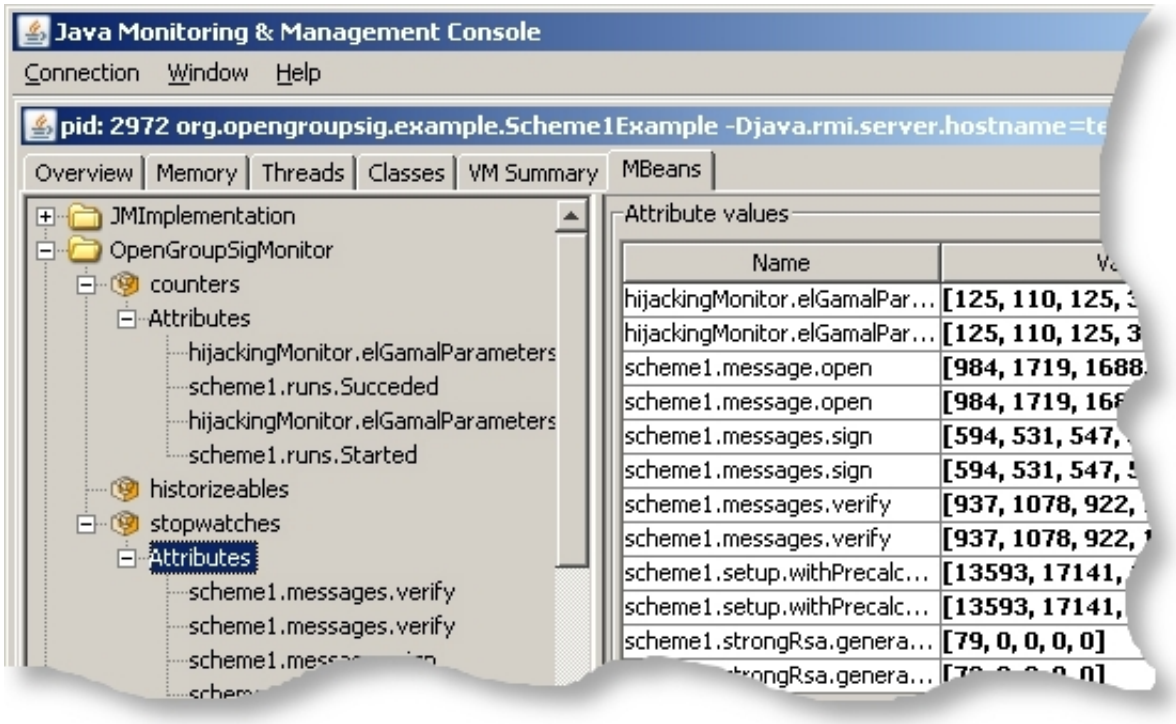


Abbildung 4.1.: Monitoring-Beispiel der Anwendung zur Laufzeit mit Hilfe eines JMX-Clients, hier: JConsole

4.3. Probleme der Implementierung und des Verfahrens

Erstellen von Gruppensignaturen mit veralteter Revisionsnummer

Um in den Ausbaustufen Σ_2 und Σ_3 ein Mitgliedschaftszertifikat zurückzuziehen, wird der Wert w des öffentlichen Schlüssels G_{Public} mit dem Wert w_j des Mitgliedschaftszertifikats $M_{Cert}(j)$ überschrieben[8]. Da der Wert w sowohl beim Signieren als auch beim Verifizieren eine Rolle spielt, sind Signaturen immer an eine Revision des öffentlichen Schlüssels gebunden. Damit alle bestehenden Signaturen ihre Gültigkeit behalten, muss in einer Signatur die Revisionsnummer von G_{Public} enthalten sein, mit der die Signatur erstellt wurde. Zu jeder Revisionsnummer sind im *Trusted-Public-Directory* die öffentlichen Schlüssel bzw. w gespeichert. Jede Signatur muss mit der Revision des öffentlichen Schlüssels verifiziert werden, mit der sie erstellt wurde.

Da die Revisionsnummer durch das Mitglied in die Signatur geschrieben wird, gibt es keine Überprüfung, ob die angegebene Revision tatsächlich die zu dem jeweiligen Zeitpunk aktuelle ist. Ein potentieller Angreifer, der ein Mitgliedschaftszertifikat besitzt, das einmal gültig war, inzwischen jedoch zurückgezogen wurde, kann so weiterhin Signaturen erstellen, die als valide erkannt werden. Er schreibt dazu eine Revision in die Signatur, in der sein Mitgliedschaftszertifikat noch gültig war. Einem Verifizierer ist es nicht möglich festzustellen, dass die Signatur mit einer Revision erstellt wurde, die zu dem Erstellungszeitpunkt keine Gültigkeit mehr besaß.

Es ist für die Praxistauglichkeit der Gruppensignaturen auch notwendig, dass Signaturen mit einer nicht mehr aktuellen Revision erstellt werden können. Andernfalls müsste jedes Mitglied, bevor es eine Signatur erstellen kann, vom zentralen *Trusted-Public-Directory* die Information erfragen, ob eine neue Revision vorhanden ist[9]. Andersherum ist es nicht praktikabel, dass eine zentrale Instanz[10] die einzelnen Mitglieder umgehend informiert, sobald eine neue Revision vorliegt[11]. In beiden Fällen müssen die Mitglieder in ständigem Kontakt zu dieser zentralen Instanz stehen, insbesondere zum *Trusted-Public-Directory*, möchten sie weiterhin Nachrichten signieren können. Das führt zu einem Teilverlust der Anonymität der Mitglieder und läßt keine

[8] siehe [CG04, S. 6, „Revocation“] und [Zim09, S. 59, 4.26 „Revoke“]
[9] Pull-Verfahren
[10] Auch in diesem Fall könnte diese Rolle das *Trusted-Public-Directory* übernehmen
[11] Push-Verfahren

Offlinenutzung des Gruppensignaturverfahrens zu.

Existiert jedoch kein Zwang, immer die neueste Revision zu verwenden, um Signaturen zu erstellen, so gibt es für Mitglieder im Umkehrschluss aber auch keine Notwendigkeit, ihre Revision überhaupt zu aktualisieren.

Inkrementelles Aktualisieren auf eine neue Revision

Möchte ein Mitglied sein Zertifikat auf die neueste Revision aktualisieren, so kann das nur inkrementell geschehen. Bezeichnet r die neueste Revisionsnummer und ein Mitglied M_i befindet sich auf Revision $r - 2$, so muss das Mitglied erst auf Revision $r - 1$ aktualisieren, bevor es sein Zertifikat auf Revision r erneuern kann.

Wäre es zulässig direkt auf die aktuellste Version zu springen, so könnte ein ausgeschlossenes Mitglied wieder ein gültiges Zertifikat erlangen, indem es eine Revision überspringt, nämlich die, in der sein Zertifikat zurückgezogen wurde. Es muss bei der Aktualisierung eines Mitgliedschaftszertifikats auf eine neue Version durch das *Trusted-Public-Directory* darauf geachtet werden, dass dieser Vorgang schrittweise erfolgt. Das wird erreicht, indem der zu einer Revision r gehörende Wert w nur ausgeliefert wird, wenn durch das anfragende Mitglied eine gültige Signatur für Revision $r - 1$ vorgelegt wird[12].

Ein ausgeschlossenes Mitglied hat trotzdem die Möglichkeit, wieder ein gültiges Zertifikat zu erhalten, wenn ein noch aktives Mitglied mit ihm kooperiert. Dazu wartet das Mitglied eine Revision ab, die größer ist als die, in der das ehemalige Mitglied ausgeschlossen wurde, und teilt diesem den aktuellen Wert w mit, den es von dem *Trusted-Public-Directory* erhalten hat. Mit Hilfe von w berechnet das ausgeschlossene Mitglied seinen zugehörigen Wert w_j. Dadurch ist es einer zentralen Instanz nicht möglich zu überprüfen, ob das Mitgliedschaftszertifikat tatsächlich inkrementell aktualisiert wurde.

[12] Diese Sicherheitsvorkehrungen sind in der vorliegenden Implementierung der Ausbaustufen Σ_2 und Σ_3 jedoch nicht umgesetzt, da sie für das Laufzeitverhalten, dass in dieser Arbeit beobachtet werden soll, nur insofern eine Rolle spielen, als dass ein Mitglied u.U. die Methode UpdateToLatestRevision mehrfach aufrufen muss, falls es eine oder mehrere Revisionen übersprungen hat

Iterieren über die *Certificate-Revocation-List* beim Verifizieren in Ausbaustufe Σ_3

Um in Ausbaustufe Σ_3 eine Signatur zu überprüfen, muss zunächst festgestellt werden, ob die Signatur durch den Gruppenmanager GM für ungültig (durch Revoke) erklärt wurde[13]. Dazu hält das *Trusted-Public-Directory* eine *Certificate-Revocation-List* (CRL) vor. Der Verifizierer einer Signatur iteriert zunächst über sämtliche Einträge s_j der CRL und überprüft, ob die Signatur mit einem der in der CRL enthaltenen Werte erstellt wurde. Danach kann die eigentliche Verify-Methode aufgerufen werden. Dadurch wächst die Laufzeit des gesamten Verifizierungsprozesses je nach Implementierung mit der Größe der CRL.

[13] siehe [CG04, S. 9, „Full Revocation“] und [Zim09, S. 66, „FullRevoke“]

5. Die Gruppensignaturverfahren in der Praxis

5.1. Aufwand der Implementierung

Die Gruppensignaturen sind auf Grundlage von [CG04] und [Zim09] mit Java realisierbar. Die Implementierung aller drei Ausbaustufen im Rahmen dieser Arbeit umfasste letztendlich 45 Java-Klassen, die sich wie folgt aufteilen:

Modul	Java-Klassen	LoC[a]
Kern-Modul (opengroupsig-core)	7 Klassen	489
Ausbaustufe Σ_1 (opengroupsig-scheme1)	9 Klassen	471
Ausbaustufe Σ_2 (opengroupsig-scheme2)	14 Klassen	625
Ausbaustufe Σ_3 (opengroupsig-scheme3)	15 Klassen	717

[a] „Lines of Code“, gemessen mit Eclipse Metrics Plugin, siehe: http://metrics2.sourceforge.net/

Tabelle 5.1.: Umfang der einzelnen Module der Implementierung

Durch die von Java bereitgestellten Bordmittel, insbesondere die Klassen `BigInteger` und `SecureRandom`, sind die mathematischen Grundlagen bereits vorhanden.

5.2. Performanceanalyse

5.2.1. Messverfahren

Um die Praxistauglichkeit der Signaturverfahren hinsichtlich der Performance zu bewerten, wurde jede Ausbaustufe 150-mal auf heute üblicher Hardware ausgeführt. In einem Durchlauf wurden die einzelnen Funktionen je-

des Schemas ausgeführt.

Ausbaustufe Σ_1: Setup, Sign, Verify, Open.
Ausbaustufe Σ_2: Setup, Join, JoinServerSide, Sign, Verify, Open, Revoke, UpdateToLatestRevision.
Ausbaustufe Σ_3: Setup, Join, JoinServerSide, Sign, Verify, Open, Revoke, FullRevoke, UpdateToLatestRevision, VerifyARevokedSignature.

Die Schritte UpdateToLatestRevision, VerifyARevokedSignature und JoinServerSide werden in [CG04] nicht als eigene Methoden vorgestellt. Sie werden hier bei der Performanceanalyse aber dennoch separat betrachtet.

Die Revoke-Methode markiert bei beiden Ausbaustufen Σ_2 und Σ_3 das Zertifikat eines Mitglieds als ungültig und erhöht die Revisions-Nummer. Der weitaus größere Rechenaufwand erfolgt erst in dem Schritt UpdateToLatestRevision. Diese Methode muss von jedem Teilnehmer als Folge des Rückrufs eines Zertifikats ausgeführt werden. Das Update auf die neueste Revision wird lokal bei jedem verbleibenden Mitglied durchgeführt. Damit ist die Gruppengröße für den Zertifikatsausschluss unerheblich. Um die hierfür benötigte Laufzeit zu bewerten, muss der gesamte Prozess des Zertifikatswiderrufs betrachtet werden. Dieser besteht aus dem Aufruf von Revoke selbst sowie vor allem aus UpdateToLatestRevision. Somit wird der gesamte Prozess nicht zentral abgearbeitet, sondern ist auf die Mitglieder verteilt. Die Performance wird maßgeblich von der Rechenkapazität jedes einzelnen Mitglieds bestimmt[1].

In den Ausbaustufen Σ_2 und Σ_3 findet ein Teil der Berechnungen des Join-Verfahrens bei dem Mitglied statt, das der Gruppe hinzutreten möchte. Der andere Teil wird vom Gruppenmanager berechnet. Entsprechend müssen Werte zwischen beiden Parteien während des Verfahrens ausgetauscht werden. JoinServerSide bezeichnet den Teil des Join-Prozesses, der vom Gruppenmanager berechnet wird. Die restliche Laufzeit von Join entfällt auf das Mitglied und nimmt damit lokale Rechenzeit in Anspruch.

Die Testumgebung besaß folgende Ausstattung: Java 1.6.0_24 VM, Intel Core2 T5600 @1,83 Ghz, 2 GB DDR2 Ram, Ubuntu 11.04 mit Linux Kernel

[1] Neben der lokal vorhandenen Rechenkapazität spielt die Geschwindigkeit der Kommunikation mit dem *Trusted-Public-Directory* eine Rolle. Vom *Trusted-Public-Directory* erfragt das Mitglied zunächst die neue Revision mit der zugehörigen Variable E_j und führt danach lokal die Berechnungen mit E_j durch. Siehe [Zim09, S. 59-60], [CG04, S. 6]

2.6.38-15-generic

Gemessen wurde die durchschnittliche Laufzeit jeder Methode der drei Ausbaustufen. Dabei wurde jedes Verfahren mit einer Schlüssellänge[2] von 1024 Bit, 2048 Bit und 4096 Bit ausgeführt[3]. Da es sich bei der Ausbaustufe Σ_1 um ein statisches Verfahren handelt, wurde das Schema in jedem Lauf zusätzlich einmal mit 10 Mitgliedern und einmal mit 1000 Mitgliedern erzeugt. Eine tabellarische Auflistung der Messungen findet sich in Anhang A.

5.2.2. Bewertungskriterien

Ziel dieser Arbeit ist es u. a. die Laufzeiten zu bewerten und zu entscheiden, ob das Verfahren bei heute üblicher Rechenkapazität praxistauglich ist. Eine wichtige Rolle spielen dabei die Aspekte, wer eine Methode aufruft und in welcher Phase, d. h. wo und wann die Rechenlast für eine Operation stattfindet.

Verteilung der Methodenaufrufe

Rechenoperationen werden entweder zentral von einer *Trusted-Third-Party*[4], vom Gruppenmanager oder von den einzelnen Mitgliedern ausgeführt. Es kann davon ausgegangen werden, dass für zentrale Rechenoperationen im Allgemeinen eine weitaus stärkere Infrastruktur mit wesentlich mehr Rechenleistung bereit steht, als es von den einzelnen Mitgliedern zu erwarten ist. Deshalb sind Methoden, die lokal bei den einzelnen Mitgliedern berechnet werden müssen, kritischer zu bewerten als Operationen, die zentral berechnet werden können. Lokale Operationen werden in den meisten Anwendungsfällen auf durchschnittlicher Desktop-Hardware, auf Smartphones oder anderen Embedded Systems laufen. Zentrale Rechenoperationen hingegen werden für gewöhnlich von Server-Hardware bewältigt. Zudem hat der Betreiber einer Gruppensignatur-Infrastruktur die Verantwortung für ein funktionierendes und performantes System. Mehrkosten an der zentralen Infrastruktur sind daher eher akzeptabel als auf Seite der Miglieder, die in der Rolle des Benutzers zu sehen sind.

[2] Parameter l_n, siehe [CG04, S.6 „Parameters“]

[3] Um die Sicherheit der Gruppensignatur zu gewährleisten, wird jedoch eine Schlüssellänge von 2048 Bit empfohlen. siehe: [CG04, S. 6]

[4] nur bei Ausbaustufe Σ_1 vorhanden

Ziel einer funktionierenden Gruppensignatur-Implementierung ist es, für den Benutzer möglichst transparent zu arbeiten. Das bedeutet, die Verwendung darf sich für die Mitglieder nicht nachteilig bemerkbar machen. Das System ist benutzerseitig auf möglichst vielen Endplattformen verfügbar; es bringt minimale Ressourcen-Anforderungen mit sich.

Häufigkeit des Methodenaufrufs

Generell gilt: Je häufiger eine Methode aufgerufen wird, desto wichtiger ist, dass sie eine kurze Laufzeit besitzt. Je nach Anwendungsfall des Gruppensignaturverfahrens können sich jedoch unterschiedliche Prioritäten ergeben.

Beispiel 1: Zur Steigerung der Integrität eines Chats werden alle Nachrichten mit einer Gruppensignatur unterschrieben. Beim Empfang einer Nachricht kann jedes Mitglied sofort deren Authentizität anhand der Signatur feststellen. Es ist davon auszugehen, dass deutlich mehr Nachrichten in einem Chat geschrieben (und somit auch verifiziert) werden, als Teilnehmer hinzukommen oder den Chat verlassen. Gehen wir davon aus, es wurde für diesen Chat eine dynamische Gruppensignatur verwendet, dann werden die Methoden Sign und Verify deutlich häufiger aufgerufen als Join, Revoke und UpdateToLatestRevision. Sie sind damit in diesem Beispiel kritischer.

Beispiel 2: Hier betrachten wir das öffentliche Verkehrsnetz der Busse und Bahnen. Anstelle von Fahrkarten erhalten die Fahrgäste am Automaten ein Mitgliedschaftszertifikat eines dynamischen Gruppensignaturschemas. Werden sie von Fahrkartenkontrolleuren überprüft, können sie durch Erstellen einer Gruppensignatur nachweisen, dass sie berechtigt sind, die Bahn oder den Bus zu benutzen. Es kommt in Großstädten häufiger vor, dass ein Fahrgast einen Fahrschein zieht bzw. dieser Fahrschein nach der Fahrt oder Ablauf einer Frist seine Gültigkeit verliert, als dass ein Fahrgast kontrolliert wird. Daher sind für die Performance des gesamten Systems die Laufzeiten von Join und Revoke wichtiger als Sign und Verify.

Phase des Methodenaufrufs

Ziel ist ein reibungsloser Betrieb der Gruppensignatur, die zur Laufzeit möglichst geringe Geschwindigkeitseinbußen verursacht. Daher sind lange Ausführungszeiten bei Methoden, die häufig und im laufenden Betrieb aufgerufen werden, weniger akzeptabel als bei statischen Methoden, die in die Phase der Initialisierung fallen. Die Methoden Sign und Verify sind besonders relevant, da sie die Hauptfunktionalität der Gruppensignatur umsetzen.

Im Allgemeinen lassen sich die Methoden wie folgt unterteilen, berücksichtigt man die Phase und die Häufigkeit des Methodenaufrufs:

besonders zeitkritisch (häufige Ausführung in nahezu Echtzeit): Sign, Verify, VerifyARevokedSignature
mäßig zeitkritisch (gelegentlicher Aufruf): Join, Revoke, FullRevoke, UpdateToLatestRevision
wenig zeitkritisch (seltener Aufruf in Ausnahmefällen): Open
unkritisch (einmalige Ausführung vor Inbetriebnahme): Setup

5.3. Bewertung der Laufzeiten

In der empfohlenen[5] Schlüsselstärke von 2048 Bit benötigen die besonders zeitkritischen Funktionen Sign und Verify ca. 280 ms (in Σ_1 und Σ_2) bzw. 355 ms (in Σ_3). Diese Laufzeiten sind für die meisten Anwendungen auf mindestens durchschnittlicher PC-Hardware ausreichend.

Gerade in offenen Netzen wie dem Internet kommen verstärkt Signaturverfahren und Verschlüsselungstechniken zum Einsatz. Hier ist die Geschwindigkeit der derzeitigen Übertragungskanäle niedrig genug, dass die gemessenen zusätzlichen Latenzen nicht groß ins Gewicht fallen dürften. Die kritische Komponente ist hier der Kommunikationskanal[6].

Für andere Echtzeit-Anwendungen, wie Telefonie oder Videostreaming, können Latenzen in dem Bereich von 280 ms bzw. 355 ms jedoch bereits als störend empfunden werden. Hier gibt es die Möglichkeit, durch Verwendung spezieller Chips, die wichtige kryptografische Funktionen hardwareseitig unterstützen, sowie durch hardwarenahe Programmierung die Latenzen zu verringern[7].

Kleinere und eingebettete Systeme können über schwächere Hardware verfügen. In diesem Fall muss mit höheren Laufzeiten gerechnet werden.

Die Erstellung eines Gruppensignaturschemas mit großer Mitgliederanzahl (1000 Mitglieder) nimmt auf durchschnittlicher Desktop-Hardware, wie

[5] [CG04, S. 6]
[6] Eine 10 MByte große Website bei einer DSL-Geschwindigkeit von 16 MBit/s über das Internet zu übertragen dauert mindestens 5 sec, in der Praxis vermutlich mehr, da mehrere Einzelverbindungen aufgebaut werden müssen.
[7] siehe [Pos10]

sie für die Tests benutzt wurde, u. U. zu viel Zeit in Anspruch. Bei dem statischen Verfahren Σ_1 dauerte das Erstellen des Gruppensignaturschemas mit 1000 Mitgliedszertifikaten über 7,35 min. Bei einer durchschnittlichen Rechenzeit für ein Join auf Server-Seite von 713 ms (Σ_2) bzw 746 ms (Σ_3) würde das Aufnehmen von 1000 Mitgliedern auf dem Testsystem mindestens ca. 11,88 min (Σ_2) bzw. über 12,43 min (Σ_3) benötigen.

Für den in der Einleitung beschriebenen Raum, der nur durch befugtes Personal betreten werden darf, wäre es akzeptabel, das Gruppensignaturschema in den eben genannten Zeiten zu erstellen. Für Anwendungen, die ad hoc eine Gruppensignatur aufsetzen müssen, beispielsweise der bereits beschriebene Chatroom oder ein Videokonferenzsystem, sind die Laufzeiten jedoch zu hoch. Je nach Anwendungsszenario muss die Rolle der *Trusted-Third-Party* (in Σ_1) bzw. des Gruppenmanagers (in Σ_2 und Σ_3) von leistungsstarker Serverhardware übernommen werden.

Durch die Verdoppelung der Schlüssellänge auf 4096 Bit steigen die Laufzeiten aller Funktionen im Durchschnitt um ca. das 5,1-fache. Die Zeiten für Sign und Verify liegen in allen Ausbaustufen deutlich über 1 min. Damit scheiden Schlüssellängen dieser Größen für die meisten Anwendungen aus.

Durch Reduzierung der Schlüssellänge auf 1024 Bit gewinnt man durch die Einbußen an Sicherheit eine gesteigerte Performance. Sign und Verify brauchen im Schnitt nur noch ca. 61 ms (Σ_1 und Σ_2) bzw. ca. 81 ms (Σ_3). Durchschnittlich werden alle Funktionen um das 4,1-fache schneller beendet. Das Aufsetzen von Σ_1 mit 1000 Mitgliedern dauert allerdings immer noch halb so lange (3,64 min), hingegen steigert sich die Geschwindigkeit der Ausführung von Join auf 175 ms bzw. 177 ms (Σ_2 bzw Σ_3). Das Gruppensignaturschema für den Chatroom mit 1000 Mitgliedern wäre damit in unter 3 min auf der Testhardware aufgesetzt.

Je nach Sicherheitsanforderungen läßt sich damit die Performance durch kürzere Schlüssellängen deutlich steigern.

Vergleich der drei Ausbaustufen

Die Laufzeiten der Ausbaustufen Σ_1 und Σ_2 sind nahezu identisch. Das ist erwartungsgemäß, da bei Σ_2 im Vergleich zu Σ_1 bei Sign und Verify lediglich eine Multiplikation mehr ausgeführt werden muss. Die Funktion Open ist sogar identisch. Das Aufsetzen des Schemas dauert bei Σ_2 insgesamt etwas länger (bei 1000 Mitgliedern ca. 11 min zu 7:21 min). Allerdings müssen

	Setup	Join(GM)	Join(M_j)	Sign	Verify	Open	Revoke	FullR.
Σ_1	$k+2$, k			7, 1	4	1		
Σ_2	3	1, 1	2, 2	7, 1	4	1	1	
Σ_3	3	1, 1	2, 2	9, 1	5	1	1	1
[ACJT00]	1	1	12	4 3	4	1		

Tabelle 5.2.: Exponentiationen und Pairings aus [Zim09, S. 113]

nicht mehr alle Mitgliedszertifikate im Voraus erstellt werden. Man erhält also ein volldynamisches Gruppensignaturverfahren ohne relevante zusätzliche Kosten im Vergleich zur statischen Variante.

Die Funktionsaufrufe benötigen beim Vefahren Σ_3 im Schnitt ca. 1,1 mal so viel Rechenzeit wie in Ausbaustufe Σ_2. Während die Ausführung von Join nur ca. 1,01 mal so lange braucht, sind die zeitkritischen Methoden Sign und Verify auf dem Testsystem ca. 1,25 mal so langsam. Damit ist die Markierungsfähigkeit aus Σ_3 relativ teuer; sie verzögert die Ausführung der Gruppensignatur um immerhin ein Viertel. Zudem muss für die Markierungsfähigkeit bei jedem Verify-Aufruf über die *CRL* iteriert werden, was bei Gruppensignaturinstanzen mit vielen Mitgliedern zusätzliche Rechenzeit benötigt[8].

Die Verwendung von Σ_3 hängt wohl davon ab, ob eine Markierungsfähigkeit für das Anwendungsszenario wichtig ist. Andernfalls ist aus Performance-Gründen Σ_2 vorzuziehen. Die Verwendung von Σ_1 bietet hingegen keinen Mehrwert. Einzige Ausnahme ist, es steht eine relativ starke zentrale Infrastruktur zur Verfügung, die die Rolle der *Trusted-Third-Party* übernehmen kann, während Mitglieder nur über sehr schwache Rechenkapazität verfügen. Dann kann es beim Aufsetzen des Gruppensignaturverfahrens hilfreich sein, dass alle Zertifikate zentral erstellt werden, ohne dass clientseitig Rechenkapazität beansprucht wird. Es ist zudem möglich, die Zertifikatserstellung bei Σ_1 zu parallelisieren.

[8] siehe: 4.3 Probleme der Implementierung und des Verfahrens

5.4. Kritik

Die in dieser Arbeit besprochenen Gruppensignaturverfahren von Camenisch und Groth wurden ausschließlich in Java implementiert. Damit kann diese Arbeit keine endgültige Aussage über theoretisch mögliche Geschwindigkeiten der Signaturverfahren treffen. Die Verfahren wurden lediglich exemplarisch implementiert. Abgesehen von der Leistungsfähigkeit der Hardware, auf der die Verfahren ausgeführt werden, ist für die Geschwindigkeit der Implementierung die JVM[9] und insbesondere die Geschwindigkeit der Algorithmen zu finden von Primzahlen und damit verbunden des Zufallszahlengenerators maßgeblich.

Da zur Bestimmung der Laufzeiten einmal die Zeit bei Aufruf und einmal bei Beendigung der jeweiligen Methode gemessen wurde, sei an dieser Stelle ausdrücklich darauf hingewiesen, dass bei den gemessenen Ausführungsgeschwindigkeiten Rechenzeit des Betriebssystems und dessen Schedulers inbegriffen sind. Bei der Bewertung der Praxistauglichkeit der Verfahren ist das aber vernachlässigbar. Zum einen ist der hierdurch entstehende Performanceverlust sehr gering, zum anderen liefe ein Gruppensignaturverfahren unter tatsächlichen Bedingungen in den meisten Fällen ebenfalls auf einem Betriebssystem wie z. B. Linux.

5.4.1. Der Zufallszahlengenerator

Die Kryptographie der Gruppensignaturverfahren basieren auf der Erstellung großer[10] Primzahlen. Häufig sind an diese Primzahlen weitere Bedingungen gestellt. Beispielsweise benötigt das CL-Signaturverfahren zwei Sophie-Germain-Primzahlen und für das ElGamal-Verschlüsselungsverfahren wird eine Primzahl P benötigt, so dass $P - 1$ bestimmte andere Primzahlen in der Primfaktorzerlegung enthält. Um eine sichere Schlüsselgenerierung zu gewährleisten, müssen die Primzahlen, insbesondere mit den eben erwähnten Eigenschaften, mit probabilistischen Algorithmen gefunden werden. Insbesondere ist bei der Primzahlengenerierung der Miller-Rabin-Test relevant. Für die Performance einer Implementierung der Gruppensignaturverfahren ist der verwendete Zufallszahlengenerator deshalb von zentraler Bedeutung. In der für diese Arbeit erstellten Implementierung wurde der von Java bereitgestellte Zufallszahlengenerator `SecureRandom` verwendet.

[9] Java Virtual Machine
[10] 512 Bit und größer

Der Hauptteil der Laufzeit der Methoden Setup, Join und Sign entfällt auf den Aufruf von `SecureRandom`. Die Klasse erzeugt gegenüber dem anderen Zufallszahlengenerator aus dem Java-Standard-Paket, `Random`, kryptografisch wertvollere Zufallszahlen, d. h. die erzeugten Zufallszahlen sind schwieriger vorauszusagen. Die Tests haben ergeben, dass die Verwendung von `Random` keinen Geschwindigkeitsvorteil bringt[11].

Andere Zufallszahlengeneratoren könnten jedoch u. U. die Performance einer Implementierung steigern. Das ist jedoch nicht Teil dieser Arbeit und müsste gesondert untersucht werden.

5.4.2. Java

Die Gruppensignaturverfahren wurden für diese Arbeit in Java umgesetzt. Implementierungen in anderen Sprachen, wie C/C++, (Object) Pascal, Perl oder Assembler, könnten entscheidend schneller sein. Da Java-Applikationen auf einer Middleware, der JVM, ausgeführt werden, könnten vor allem Implementierungen in Sprachen, die nativ auf der Hardware laufen, einen Vorteil haben, insbesondere C/C++ oder Assembler. Auch das müsste in einer weiteren Arbeit untersucht werden.

Allerdings besitzt Java eine weite Verbreitung[12], was es für eine exemplarische Implementierung prädestiniert. Zudem ist die Java-Plattform, nicht zuletzt aufgrund ihrer langen Entwicklungszeit[13], hinsichtlich Performance optimiert. Java wird bereits für viele sicherheits- und zeitkritische Anwendungen eingesetzt. Damit besitzt eine Implementierung in Java durchaus Aussagekraft für die Praxistauglichkeit von Verfahren.

Entscheidende Geschwindigkeitsvorteile kann auch eine Implementierung direkt auf nativer Hardware bieten, insbesondere wenn die Hardware kryptografische Teil-Funktionen zur Verfügung stellt[14]. In der Praxis ist dieses Szenario sogar wahrscheinlich, da Signaturverfahren in Chipkarten-Lesegeräten, Smartphones und anderen kleinen eingebetteten Systemen verwendet werden könnten.

[11] siehe Laufzeit-Tabellen im Anhang

[12] Im TIOBE-Index belegt Java im September 2012 den zweiten Platz, siehe `http://www.tiobe.com/index.php/content/paperinfo/tpci/`.

[13] seit 1995 auf dem Markt

[14] Wie der in [Pos10] verwendete Microchip von Samsung

5.4.3. Zusätzliche Latenzen

Nicht Teil der Implementierung ist eine Kommunikation zwischen den Parteien der Gruppensignatur: Mitgliedern, Gruppenmanager, *Trusted-Public-Directory* und *Trusted-Third-Party*. In einem realen Szenario finden die Rechenoperationen der Beteiligten in den meisten Fällen nicht auf einer einzigen Plattform statt. Vielmehr sind die Parteien über ein Netzwerk, wie z. B. Ethernet, GSM oder Bluetooth, miteinander verbunden. Dadurch entstehen weitere Latenzen: Die Daten müssen in ein austauschbares Format gebracht, ver- und entschlüsselt sowie übertragen werden.

In den meisten Fällen wird eine Gruppensignatur aber vermutlich in ein System integriert, bei dem bereits eine Kommunikation zwischen den einzelnen Parteien stattfindet.

6. Fazit

In dieser Arbeit wurde ein Gruppensignaturverfahren vollständig implementiert. Dabei wurde sowohl die statische als auch die dynamischen Varianten berücksichtigt. Es wurden Mechanismen gefunden, um die Performance der laufenden Java-Applikation effektiv mit JMX zu überwachen. Die Bewertung der Laufzeiten zeigt, welche Methoden kritisch zu sehen sind und welchen Stellen im Hinblick auf Performance besondere Beachtung geschenkt werden muss, wird das Gruppensignaturverfahren in der Praxis eingesetzt.

Dabei läßt sich die im Zuge dieser Arbeit entwickelte Implementierung durchaus als Framework einsetzen, um Gruppensignaturen in Java-Applikationen zu verwenden. Um die Einsatzfähigkeit zu gewährleisten, müssten vor allem die unter „4.3 Probleme der Implementierung und des Verfahrens“ beschriebenen Schwachstellen behoben werden. Die Sicherheit der Verfahren wurde bereits in [Zim09] und [CG04] theoretisch bewiesen.

Mit insgesamt 45 Klassen und 2302 Lines of Code kann die Anwendung als durchaus klein bezeichnet werden, zumal es trotz der Modularisierung große Redundanzen zwischen den drei Ausbaustufen gibt. Da die Umsetzung zudem auf zusätzliche Crypto-Bibliotheken verzichtet und die Mathematik weitgehend mit `BigInteger` realisiert wurde, ist auch die Komplexität eher gering. Damit wurde gezeigt, dass mit relativ geringem Aufwand und Mitteln ein funktionierendes Gruppensignaturschema entwickelt werden kann.

Anhang

A. Laufzeiten

A.1. Laufzeiten unter Verwendung von `SecureRandom`

Ausbaustufe 1 (10 Mitglieder)

Schlüssellänge	4096 bit	2048 bit	1024 bit
Setup	20413 ms	4398 ms	2147 ms
Sign	1496 ms	280 ms	60 ms
Verify	1517 ms	287 ms	62 ms
Open	1585 ms	303 ms	67 ms

Ausbaustufe 1 (1000 Mitglieder)

Schlüssellänge	4096 bit	2048 bit	1024 bit
Setup	2027833 ms	441106 ms	218500 ms
Sign	1508 ms	279 ms	61 ms
Verify	1537 ms	286 ms	62 ms
Open	1599 ms	304 ms	68 ms

Ausbaustufe 2

Schlüssellänge	4096 bit	2048 bit	1024 bit
Setup	126 ms	33 ms	9 ms
Join	5828 ms	890 ms	175 ms
JoinServerSide	4016 ms	713 ms	233 ms
Sign	1493 ms	279 ms	60 ms
Verify	1513 ms	285 ms	62 ms
Open	1580 ms	303 ms	67 ms
Revoke	0 ms	0 ms	0 ms

UpdateToLatestRevision	177 ms	46 ms	12 ms

Ausbaustufe 3

Schlüssellänge	4096 bit	2048 bit	1024 bit
Setup	127 ms	33 ms	9 ms
Join	5903 ms	902 ms	177 ms
JoinServerSide	4106 ms	746 ms	226 ms
Sign	1786 ms	354 ms	81 ms
Verify	1779 ms	354 ms	81 ms
Open	1847 ms	372 ms	85 ms
Revoke	0 ms	0 ms	0 ms
Full-Revoke	0 ms	0 ms	0 ms
UpdateToLatestRevision	179 ms	46 ms	12 ms
VerifyARevokedSignature	61 ms	16 ms	4 ms

A.2. Laufzeiten unter Verwendung von `Random`

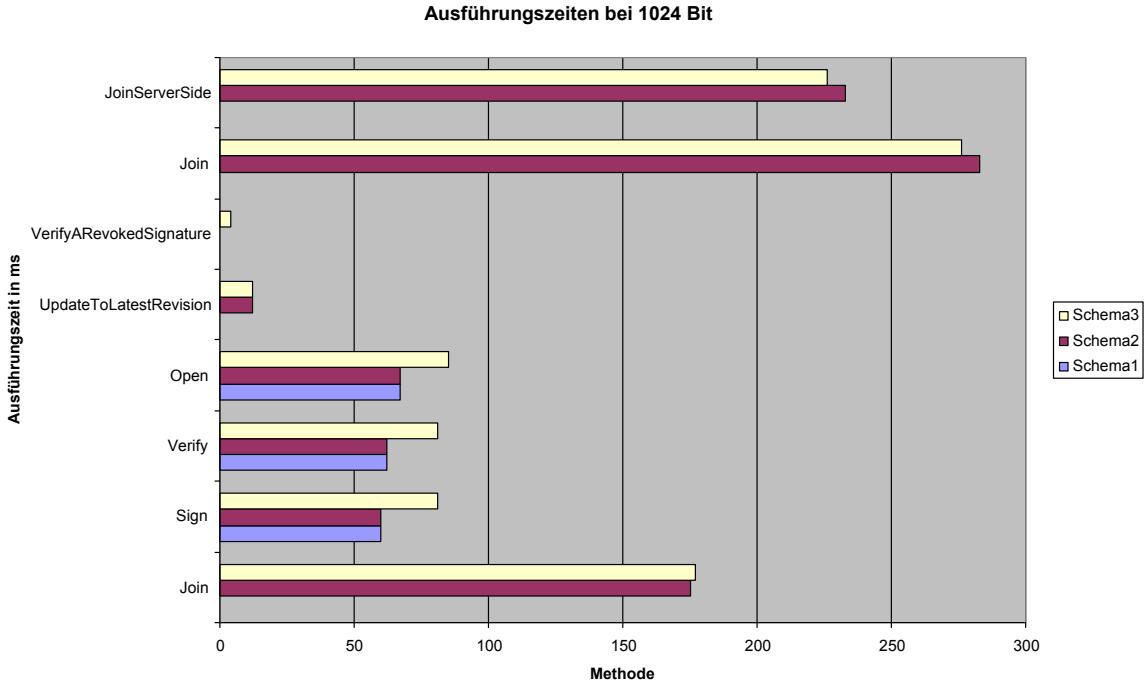
Ausbaustufe 2

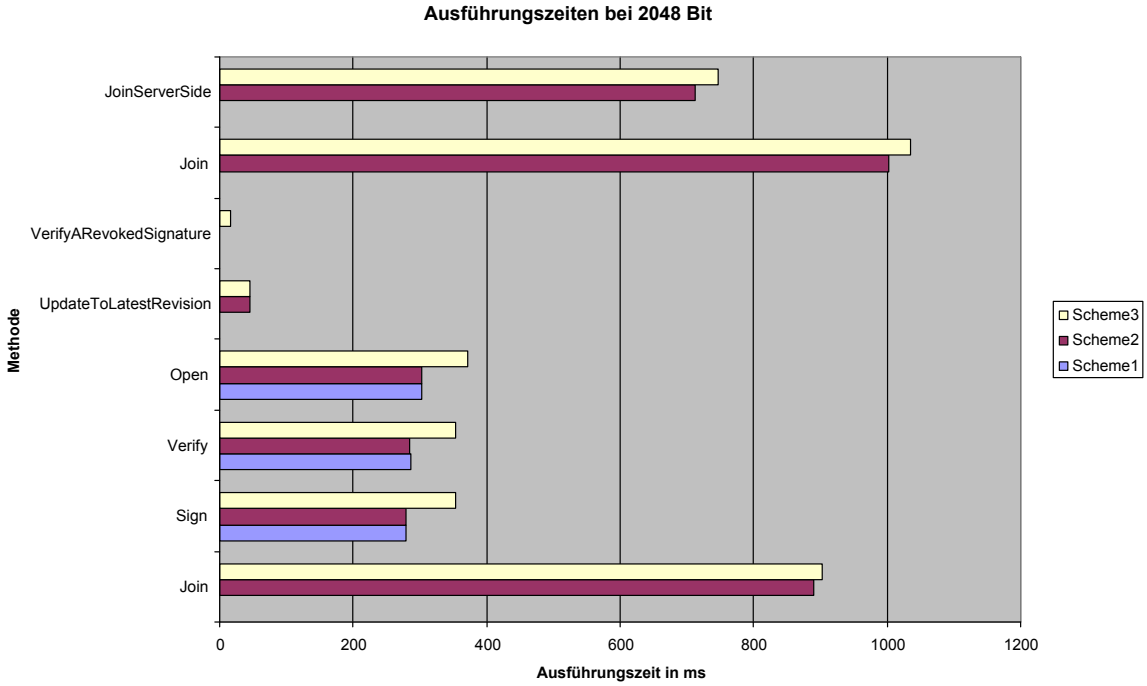
Schlüssellänge	4096 bit	2048 bit	1024 bit
Setup	124 ms	32 ms	8 ms
Join	5935 ms	1038 ms	261 ms
JoinServerSide	3983 ms	746 ms	213 ms
Sign	1494 ms	278 ms	60 ms
Verify	1513 ms	285 ms	62 ms
Open	1582 ms	302 ms	67 ms
Revoke	0 ms	0 ms	0 ms
UpdateToLatestRevision	178 ms	46 ms	12 ms

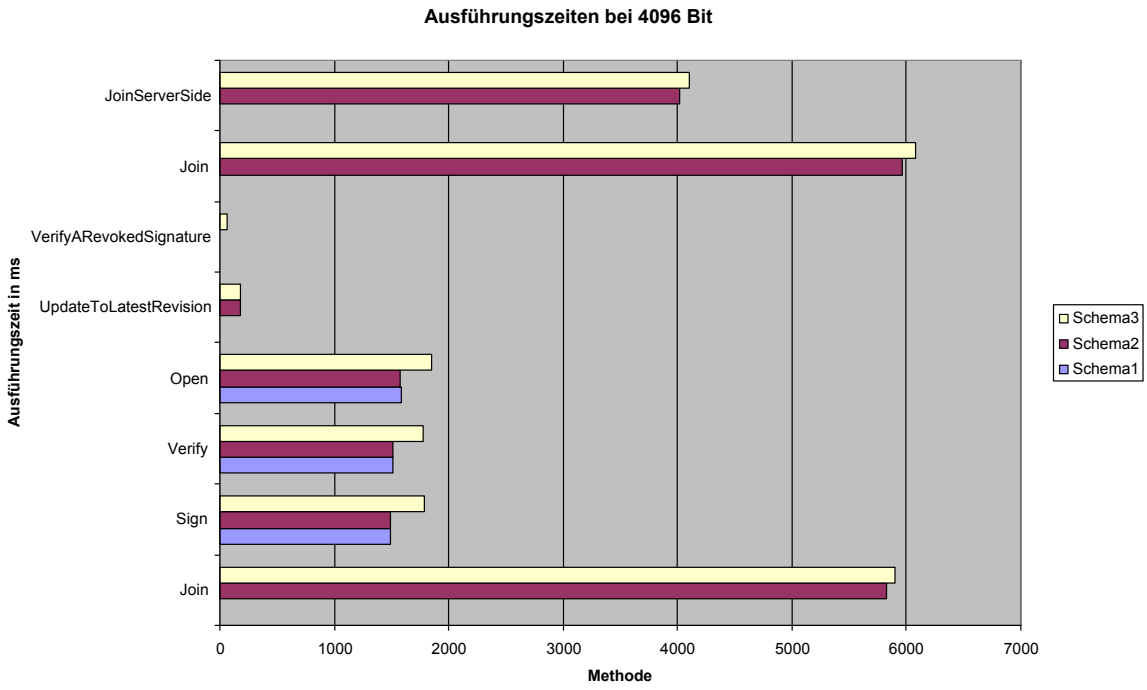
Ausbaustufe 3

Schlüssellänge	4096 bit	2048 bit	1024 bit
Setup	126 ms	32 ms	9 ms
Join	6060 ms	1010 ms	290 ms
JoinServerSide	4088 ms	719 ms	242 ms
Sign	1789 ms	354 ms	80 ms

Verify	1784 ms	353 ms	80 ms
Open	1850 ms	370 ms	85 ms
Revoke	0 ms	0 ms	0 ms
Full-Revoke	0 ms	0 ms	0 ms
UpdateToLatestRevision	179 ms	46 ms	12 ms
VerifyARevokedSignature	61 ms	15 ms	4 ms


Ausführungszeiten bei 1024 Bit
JoinServerSide
Join
VerifyARevokedSignature
UpdateToLatestRevision
Open
Verify
Sign
Join
Ausführungszeit in ms
0
50
100
150
200
250
300
Methode
Schema3
Schema2
Schema1


Ausführungszeiten bei 2048 Bit
JoinServerSide
Join
VerifyARevokedSignature
UpdateToLatestRevision
Open
Verify
Sign
Join
Methode
0
200
400
600
800
1000
1200
Ausführungszeit in ms
Scheme3
Scheme2
Scheme1


Ausführungszeiten bei 4096 Bit
JoinServerSide
Join
VerifyARevokedSignature
UpdateToLatestRevision
Open
Verify
Sign
Join
Ausführungszeit in ms
0
1000
2000
3000
4000
5000
6000
7000
Methode
Schema3
Schema2
Schema1

B. Quellcode

Archiv mit Quelltexten:	opengroupsig.tar.gz
MD5 Hash des Archivs:	`5e0fb486d834de67febb609e1b798dda`
SHA1 Hash des Archivs:	`42b8d0739b91632caf51f5c27febae9bb3e58ae1`

Literaturverzeichnis und Quellenverzeichnis

[ACJT00] Giuseppe Ateniese, Jan Camenisch, Marc Joye, and Gene Tsudik. A pratical and provably secure coalation-resistant group signature scheme. In *Proceedings of Crypto '00, Bd. 1880*. Springer Verlag, 2000.

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In *Advances in Cryptology - CRYPTO 2004*. Springer-Verlag, 2004.

[BMW03] Mihir Bellare, Daniele Micciancio, and Bogdan Warinschi. Foundations of group signatures: Formal definition, simplified requirements and a construction based on general assumptions. In *Advances in Cryptology - Eurocrypt 2003, LNCS 2656*, 2003.

[CG04] Jan Camenisch and Jens Groth. Group signatures: Better efficiency and new theorectical aspects. In *Security In Communication Networks - SCN 2004*. Springer Verlag, 2004.

[CL02] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Proceedings of Crypto 2002*. Springer Verlag, 2002.

[CvH91] David Chaum and Eugène van Heyst. Group signatures. In *Advances in Cryptology - Eurocrypt 1991*. Springer-Verlag, 1991.

[DP06] Cécile Delerablée and David Pointcheval. Dynamic fully anonymous short group signatures. In *International Conference on Cryptology in Vietnam 2006*. Springer-Verlag, 2006.

[ES10] Karl Eilebrecht and Gernot Starke. *Patterns kompakt*. Spektrum Akademischer Verlag, 2010.

[Fow03] Marting Fowler. *Patterns Of Enterprise Application Architecture*. Addison-Wesley, 2003.

[Fow12] Martin Fowler. Inversion of control containers and the dependency injection pattern, August 2012.

[Ge05] He Ge. An effective method to implement group signature with revocation. Technical report, Department of Computer Science and Engineering University of North Texas, 2005.

[LPY] Benoit Libert, Thomas Peters, and Moti Yung. Scalable group signatures with revocation. In *Proceedings of the 31st Annual international conference on Theory and Applications of Cryptographic Techniques*.

[LPY12] Benoît Libert, Thomas Peters, and Moti Yung. Scalable group signatures with revocation. In *Advances in Cryptology - Eurocrypt 2012, LNCS 7323*, 2012.

[Mar09] Robert C. Martin. *Clean Code*. Prentice Hall, 2009.

[Pos10] Tim Postler. Smart Card basierte Berechnung einer Gruppensignatur als Teil einer biometrischen Authentisierung. Diplomarbeit, Universität Paderborn, März 2010.

[Zim09] Torsten Zimmermann. Vergleich beweisbar sicherer effizienter Gruppensignaturverfahren. Diplomarbeit, Justus-Liebig-Universität Gießen, Februar 2009.