

Technische Universität Darmstadt

Fachbereich Informatik

**Ein beweisbar sicherer
Pseudozufallsbit-Generator
auf der Basis des
DL-Problems in elliptischen Kurven**

Diplomarbeit

von

MARCUS LIPPERT

28. Dezember 2000

angefertigt bei PROF. DR. J. BUCHMANN
Arbeitsgruppe Theoretische Informatik

Zusammenfassung

Die vorliegende Diplomarbeit beschäftigt sich mit der Implementierung eines kryptographisch sicheren Pseudozufallsbit-Generators auf Basis des DL-Problems in elliptischen Kurven.

Es wird ein Modell vorgestellt, in dem allgemein Pseudozufallsbit-Generatoren definiert und ihre Eignung für alle, auch kryptographische Anwendungen, mit Komplexitätstheoretischen Mitteln bewiesen werden können. Dies geschieht mittels asymptotischer Analyse und unter Verwendung eines uniformen Angreifermodells.

In obigem Modell wird ein Pseudozufallsbit-Generator, der mit elliptischen Kurven arbeitet, definiert und der Beweis seiner kryptographischen Sicherheit erbracht. Dieser Beweis stellt eine Reduktion der kryptographischen Sicherheit auf die Schwierigkeit des DL-Problems in elliptischen Kurven über Primkörpern großer Charakteristik dar.

An diese theoretischen Vorarbeiten schließt sich eine Implementierung des Elliptische-Kurven-Pseudozufallsbit-Generators an. Hierzu wird der Rahmen der asymptotischen Analyse verlassen und konkrete Parameter für elliptische Kurven auf der Basis von Vorgaben des Bundesamtes für Sicherheit in der Informationstechnik gewählt. Die Implementierung erfolgt in der Programmiersprache C++ und wird durch ein in der Programmiersprache Java erstelltes Interface konform zur Java Cryptography Architecture auch Java-Anwendungen zur Verfügung gestellt.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Marcus Lippert
In den Frenzen 2
55218 Ingelheim
Marcus.Lippert@epost.de

Darmstadt, den 28. Dezember 2000

Marcus Lippert

Die in diesem Dokument erwähnten Soft- und Hardwarebezeichnungen sind in den meisten Fällen auch eingetragene Warenzeichen und unterliegen als solche den gesetzlichen Bestimmungen.

Danksagung

Mein Dank gilt allen Mitarbeitern des Fachgebiets Theoretische Informatik, bei denen ich jederzeit für Fragen oder Probleme ein offenes Ohr fand. Insbesondere möchte ich mich bei Prof. Dr. Johannes Buchmann, Dr. Ingrid Biehl, Harald Baier und Markus Ruppert für die Diskussionen, Tips und viel konstruktive Kritik bedanken. Mein Dank gilt auch meiner Familie; besonders natürlich meiner Frau und meinen beiden Kindern Denise und Katharina, die auf ihre Weise sehr zum Gelingen dieser Diplomarbeit beigetragen haben. Schließlich möchte ich mich noch herzlich bei meinem Bruder Andreas dafür bedanken, daß er mir viel Mut zugesprochen und Korrektur gelesen hat.

Inhaltsverzeichnis

1	Einleitung	7
2	Grundlagen	10
2.1	Zufallsbits	10
2.2	Pseudozufallsbits	12
2.3	Komplexitätstheorie	12
2.4	Pseudozufallsbit-Generator	14
2.5	Konstruktion	14
2.6	Schaaren von Oneway-Funktionen	16
2.7	Steigerung der Effizienz	17
3	Elliptische Kurven und diskrete Logarithmen	19
3.1	Grundlagen	19
3.1.1	Gruppen	19
3.1.2	Folgen regulärer Gruppen	22
3.1.3	Körper	23
3.2	Elliptische Kurven	23
3.2.1	Anzahl der Punkte	24
3.2.2	Struktur der Elliptische-Kurven-Gruppen	26
3.2.3	Binärkodierung der Kurvenpunkte	26
3.2.4	Folgen von Elliptische-Kurven-Gruppen	26
3.3	Erzeugung Elliptischer Kurven zum Index i	27
3.3.1	Primkörper	27
3.3.2	Elliptische Kurve	28
3.3.3	Ordnung der Gruppe	28
3.3.4	Elementordnung	28
3.3.5	Faktorisierung der Gruppenordnung	29
3.3.6	Maximalordnung	29
3.3.7	Erzeuger-Tupel	29
3.4	DL-Problem	31
3.4.1	Verfahren zur Berechnung diskreter Logarithmen	32
3.4.2	Kryptographisch starke elliptische Kurven	32
4	Hard-Core-Prädikat	35
4.1	Hauptsatz	35
4.2	Erster Beweisschritt	36

4.2.1	höchstwertige Bits	36
4.2.2	Übersicht	37
4.2.3	Kreuzkorrelation	38
4.2.4	Schätzen der Kreuzkorrelation	40
4.2.5	Erweiterungen	42
4.3	Zweiter Beweisschritt	43
5	Konstruktionen	45
5.1	Diskretes Exponentieren als Oneway-Permutation	45
5.1.1	Verwendung spezieller Kurven	46
5.1.2	Twist	46
5.2	Definition des Pseudozufallsbit-Generators	48
6	Realisierung	50
6.1	Vorgaben	50
6.1.1	Wahl der Programmiersprache	50
6.1.2	Java Cryptography Architecture (JCA)	51
6.1.3	JNI	52
6.1.4	Die Bibliothek LiDIA	53
6.1.5	Die Klasse <code>gcm_complex_multiplication</code>	53
6.2	Design und Implementierung	54
6.3	Beschreibung der Komponenten	55
6.3.1	<code>ECPrng</code>	55
6.3.2	<code>ThreadSeed</code>	55
6.3.3	<code>ECPrng_impl</code>	56
6.3.4	Erzeugung der Kurven	57
6.4	Laufzeiten	58
6.5	Ausblick	58
A	Java Quellcode	60
A.1	<code>cdc.ECPrng.java</code>	60
B	C++ Quellcode	67
B.1	<code>impl.h</code>	67
B.2	<code>types.h</code>	69
B.3	<code>ecprng_native.cpp</code>	69
B.4	<code>ecprng_impl.cpp</code>	71
B.5	<code>generate_curve.cpp</code>	76

Abbildungsverzeichnis

4.1	Schätzen der Korrelation	41
4.2	Berechnung von Logarithmen	42
6.1	Zusammenspiel von SecureRandom und SecureRandomSpi . .	52
6.2	Verteilung der Komponenten	54
6.3	Arbeitsweise von ThreadSeed	56

Tabellenverzeichnis

6.1	Klasse SecureRandomSpi	51
6.2	Verwendete Klassen aus LiDIA	53
6.3	Verwendete Methoden von <code>gex_complex_multiplication</code> . .	54
6.4	Liste der Komponenten	55
6.5	Schnittstelle der Klasse <code>ECPrng_impl</code>	56
6.6	Timings	58

Kapitel 1

Einleitung

In den letzten Jahren hat die Kryptographie in unserer Gesellschaft immer mehr an Bedeutung gewonnen und ist nun auf dem besten Wege, Einzug in unseren Alltag zu halten. Wesentlich dazu beigetragen hat die starke Verbreitung des Internets und die damit verbundene Flut an Daten, die täglich weltweit über digitale Netze transportiert wird. Damit geht das Bedürfnis der Nutzer einher, Daten sicher im Netz versenden zu können, sei es um die Vertraulichkeit persönlicher Informationen zu wahren oder um geschäftliche Transaktionen verbindlich und sicher abzuwickeln. Entsprechende Schlagworte sind unter anderem 'starke Verschlüsselung' und 'digitale Unterschrift'. Die Infrastruktur, die dies alles leisten soll, setzt sich aus einzelnen Bausteinen zusammen, die unter anderem auf kryptographischen Basistechnologien und -konzepten fußen.

Die vorliegende Diplomarbeit beinhaltet einen solchen Baustein, nämlich die Erzeugung von Pseudozufallszahlen oder, etwas allgemeiner, von pseudozufälligen Bitfolgen.

Zufallbitfolgen kommen in fast allen kryptographischen Protokollen zum Einsatz, zum Beispiel bei der Erzeugung von Schlüsseln und Parametern für die symmetrische und asymmetrische Verschlüsselung.

Die Verfahren zur rechnergestützten Gewinnung von Zufallszahlen sind jedoch nicht erst im Rahmen kryptographischer Anwendungen entstanden, sondern umfassen ein weit größeres Verbreitungsgebiet. Die ersten Anwendungen von Zufallszahlen-Generatoren in Rechnersystemen waren im Bereich stochastischer Simulationen angesiedelt; Vor allem in der Physik, aber auch in vielen anderen naturwissenschaftlichen und technischen Disziplinen, werden zum Simulieren realer Vorgänge in Computersystemen große Mengen an Zufallszahlen benötigt. Nicht zuletzt deswegen finden sich auf praktisch jeder Rechnerplattform entsprechende Programmbibliotheken, die (Pseudo-)Zufallszahlen-Generatoren bereitstellen. Diese sind in der Regel auf die Bedürfnisse dieser Anwendungsgebiete hin ausgelegt.

Die Anforderungen innerhalb kryptographischer Verfahren sind jedoch anderer Art. Die in eben genannten Programmpaketen enthaltenen Generatoren sind hier in der Regel gar nicht oder nur mit Einschränkung verwendbar. Der wesentliche Unterschied besteht bei kryptographischen Verfahren darin, daß die erzeugten Bitfolgen *sicher* im Sinne von *nicht vorhersagbar* sein sollen.

Im Folgenden soll ein kurzer Überblick über einige Begrifflichkeiten und Aspekte

im Zusammenhang mit Zufallszahlenerzeugung und darauf basierend eine Eingrenzung des Themas dieser Diplomarbeit gegeben werden.

Als erstes stellt sich natürlich die Frage nach einer Definition von (Pseudo-) Zufallszahlen. In einem allgemeinen Lexikon¹ findet sich etwa folgende Beschreibung:

Zufallszahlen, in Abhängigkeit von zufälligen Prozessen (z.B. radioaktiver Zerfall) oder durch geeignete Rechenvorschriften hergestellte Zahlenfolgen, die sich in vielerlei Hinsicht wie eine zufällige Zahlenfolge verhält (daher auch *Pseudo-Zufallszahl* genannt). Zufallszahlen werden z.B. für Modellrechnungen physikalischer Prozesse benutzt (→ Monte-Carlo-Methode).

Bemerkenswert ist zunächst, daß sich Zufall nicht auf die Zahlenfolgen an sich, sondern auf ihren Erzeugungsprozess bezieht. Eine Objekt selbst kann nach diesem Verständnis nicht zufällig sein², sondern nur zufällig erzeugt werden.

Wir betrachten demnach Prozesse, die Objekte aus einer (endlichen) Grundgesamtheit möglicher Objekte auswählen, und die Wahrscheinlichkeitsverteilung, mit der dies geschieht.

Einen Prozeß, bei dem zu jedem Zeitpunkt jedes mögliche Objekt unabhängig von den bisher erzeugten Objekten mit gleicher Wahrscheinlichkeit auftritt, werden wir *echt gleichverteilt* nennen. Wenn wir bei der Erzeugung eines Schlüssels für ein Verschlüsselungsverfahren garantieren können, daß jeder mögliche Schlüssel mit gleicher Wahrscheinlichkeit gewählt wird, ist es einem potentiellen Angreifer nicht möglich, zusätzliche Informationen über diesen Schlüssel zu gewinnen. Ein solcher Erzeugungsprozeß wäre sicher. Da die Gleichverteilung in gewisser Weise ideal ist, stellt sie ein geeignetes Maß zur Beurteilung tatsächlicher Verfahren dar.

Man unterscheidet zwei grundsätzliche Arten von solcher Prozesse: Die einen machen sich physikalische oder andere Effekte zunutze und erzeugen die Zufallsbits “wie aus dem Nichts”. Ihr Verhalten ist (im Idealfall) in keiner Weise vorhersagbar oder reproduzierbar. In der Literatur heißen diese Verfahren Zufallsbit-Generatoren³. Wir werden in dieser Arbeit nur diese idealen Generatoren so benennen. Die andere Art, die sogenannten Pseudozufallsbit-Generatoren, sind im Gegensatz dazu deterministische Algorithmen. Sie erhalten einen Bitstring, den sogenannten Zufallskeim (engl. seed), als Eingabe und berechnen daraus eine längere Bitfolge. Die Ausgabe ist, wenn man den Seed kennt, in keiner Weise zufällig. Man kann jedoch zeigen, daß das Ausgabeverhalten dieser Algorithmen bei Nichtkenntnis des Seeds zumindest bezüglich einiger Eigenschaften ähnlich ist wie bei den Zufallsbitgeneratoren.

Diese Diplomarbeit beschäftigt sich mit Pseudozufallsbit-Generatoren, von denen man unter gewissen Annahmen nachweisen kann, daß die von ihnen erzeugten Pseudozufallsbitstrings ohne Einschränkungen in *allen* Anwendungen anstelle echter Zufallsbitstrings verwendet werden können, also auch in kryptographischen. Ziel ist es, einen solchen Pseudozufallsbit-Generator auf Basis elliptischer Kurven bereitzustellen und zu implementieren.

¹dtv-Lexikon in 20 Bänden, erschienen 1990 im Deutschen Taschenbuch Verlag

²Dies hängt allerdings davon ab, welche Theorie des Zufalls man zugrunde legt. Solomonov, Kolmogorov und Chaitin haben einen Ansatz basierend auf der Berechenbarkeitstheorie, der es erlaubt die Zufälligkeit eines Objekts zu definieren. Vgl. dazu [Gol99, Abschnitt 1].

³Wir gebrauchen den Begriff ‘Generator’ im Sinne einer Realisierung eines Prozesses

Hier nun eine Übersicht über die folgenden Kapitel:

- In Kapitel 2 wird das Modell eingeführt, welches es erlaubt, solche kryptographisch sicheren Generatoren einzuführen. Insbesondere wird hier festgelegt, welche Möglichkeiten bezüglich der Berechnungskapazität einem möglichen Angreifer zugestanden werden. Wir nehmen die Existenz sogenannter Oneway-Funktionen, d.h. Funktionen, die nicht effizient invertiert werden können, an und führen dann das sogenannte Hard-Core-Prädikat ein, das die Oneway-Eigenschaft auf einzelne Bits überträgt. Mit diesen beiden Primitiven ist es dann möglich, kryptographisch sicherer Pseudozufallsbit-Generatoren zu definieren.
- Kapitel 3 bietet eine knappe Einführung in die Theorie der elliptischen Kurven. Wir werden elliptische Kurven als endliche, Abelsche Gruppen auffassen, was es uns ermöglicht, die diskrete Exponentiation und ihre Umkehrfunktion, den diskreten Logarithmus, bereitzustellen. Die diskrete Exponentiation kann man als Oneway-Funktion auffassen, wenn diskrete Logarithmen in diesen Gruppen nicht effizient berechnet werden können (DL-Problem).
- Das Kapitel 4 führt Hard-Core-Prädikate für die diskrete Exponentiation ein. Diese werden das Ausgabeverhalten des am Ende implementierten Generators beschreiben.
- Schließlich werden wir im Kapitel 5 die zuvor bereitgestellten Konzepte in der Konstruktion eines Pseudozufallsbit-Generators vereinen und die letzten theoretischen Vorarbeiten vor der Implementierung abschließen.
- Das Kapitel 6 beschreibt zu guter Letzt die Implementierung des Generators. Themen sind hier die Wahl der Programmiersprache(n), die Integration des Generators mit der Softwarebibliothek LiDIA, die Integration in die Java Cryptography Architecture und Belange der Laufzeitoptimierung.

Kapitel 2

Grundlagen

In kryptographischen Anwendungen ist es an vielen Stellen notwendig, Objekte zufällig zu wählen. Alle diese Objekte werden in Rechnersystemen durch endliche Bitfolgen repräsentiert. Wir werden uns deshalb allgemein mit der (pseudo-)zufälligen Erzeugung solcher Bitstrings beschäftigen. Ein wesentliches Interesse bei kryptographischen Verfahren gilt ihrer Sicherheit. Doch was bedeutet Sicherheit in Bezug auf Zufallsbit-Generatoren?

Betrachten wir folgendes Beispiel: Ein PC soll eine sichere Verbindung zu einem Server aufbauen. Zu diesem Zweck erzeugt der Server einen geheimen n Bit langen Schlüssel, der dem PC über einen sicheren Kanal mitgeteilt wird. Mit diesem Spruchschlüssel werden nun sämtliche Daten, die PC und Server austauschen, mittels einer symmetrischen Chiffre verschlüsselt. Wir wollen annehmen, daß ein Außenstehender keine Möglichkeit hat, den verwendeten Schlüssel selbst auszuspähen. Wenn nun das Verfahren zur Erzeugung des Spruchschlüssels an sich *keine weiteren Informationen* über den Schlüssel preisgibt, können wir es getrost als sicher bezeichnen. Ein Angreifer kann dann nur noch alle möglichen Schlüssel ausprobieren. Wir können dann erwarten, daß er im Schnitt die Hälfte der möglichen Schlüssel (das sind $2^{(n-1)}$ viele!) testen muß, um den richtigen zu finden.

Das Beispiel dieser Rechner-zu-Rechner-Kommunikation ist bewußt so ausgewählt, man kann hier noch einen anderen Effekt beobachten: Der Server erzeugt den Spruchschlüssel auf Anfrage. Ein Angreifer hat also prinzipiell die Möglichkeit, sehr viele, aus Sicht eines Erzeugungsprozesses, aufeinanderfolgende Spruchschlüssel anzufordern, um diese auf Informationen hin zu untersuchen.

2.1 Zufallsbits

In idealer Weise (bezogen auf die Sicherheit) leisten diese Aufgabe (echte) Zufallsbit-Generatoren, die wie folgt definiert sind (vgl. [AMV96, Definition 5.1]):

Definition 2.1.1 (Zufallsbit-Generator)

Ein Zufallsbit-Generator (ZBG) ist ein Gerät oder Verfahren, daß Folgen statistisch unabhängiger, gleichverteilter Bits erzeugt.

Betrachten wir einen solchen ZBG G , der Bitstrings der Länge n erzeugt. Dann ist für alle $x \in \{0, 1\}^n$ zu jedem Zeitpunkt die Wahrscheinlichkeit, daß G den String x

erzeugt wie folgt gegeben:

$$\Pr[G = x] = \frac{1}{2^n} \quad (2.1)$$

Dies ist gleichbedeutend damit, daß jedes erzeugte Bit unabhängig von allen bisher erzeugten Bits mit der gleichen Wahrscheinlichkeit $\frac{1}{2}$ eine '1' oder eine '0' wird. So erzeugte Bitstrings werden in der Literatur oft auch echte Zufallsbitstrings genannt. Sie sind quasi der Idealfall.

Es stellt sich natürlich jetzt die Frage, ob es solche echten Zufallsbit-Generatoren überhaupt gibt.

In der Praxis benutzt man bestimmte Effekte, um daraus Bitfolgen zu erzeugen. Man kann hier zwei Arten von Generatoren unterscheiden.

Die einen sind "in Hardware gegossen" und beruhen meist auf physikalischen Effekten. Ein Beispiel hierfür ist der radioaktive Zerfall bestimmter Elemente. Hierbei wird die Zeitdauer zwischen dem Zerfall zweier Atomkerne als zufällig angesehen. Sie erfordern meist erheblichen zusätzlichen Hardwareaufwand und sind anfällig gegenüber Defekten. Dennoch wird viel Aufwand in ihre Entwicklung gesteckt und man findet sie mittlerweile sogar integriert in Prozessoren und Smartcards. Dies liegt daran, daß sie in der Regel wesentlich schneller sind, als die nachfolgend beschriebenen.

Die anderen sind reine Software-Module und beruhen auf Vorgängen im Betriebssystem, die aus Sicht des Generators zufällig sind. Dies können der Inhalt bestimmter Systempuffer oder Effekte im Zusammenspiel von Prozessen oder Threads in einer Multiprocessing-Umgebung sein. Diese Art Generator ist sehr stark von der Betriebssystemumgebung abhängig und unterliegt Effekten, die außerhalb seiner Kontrollmöglichkeiten liegen. Eine Übersicht über solche Generatoren findet sich unter anderem in [AMV96, EEE98, ECS94].

Die Sicherheit beider Arten von Generatoren beruht auf der Annahme, daß die ihnen zugrundeliegenden Vorgänge so komplex sind, daß sie von einem Beobachter nicht in einer vorhersagbaren Weise manipuliert, nachvollzogen oder gar simuliert werden können, womit es unmöglich ist, erzeugte Bitstrings zu reproduzieren.

Als weiteres Indiz für die Sicherheit dienen bestimmte statistische Tests, die Bitstrings bestimmter Länge der Ausgabe des Generators mit statistischen Hilfsmitteln untersuchen. Ein Generator besteht einen solchen Test, wenn der Bitstring sich in diesem Test wie ein echter Zufallsbitstring gleicher Länge verhält. Der Generator gilt als gut, wenn er alle vereinbarten Tests besteht. Diese Vorgehensweise ist jedoch nicht systematisch, es können lediglich bestimmte Defekte aufgedeckt werden. Eine Garantie, daß es nicht doch einen statistischen Test gibt, den der Generator nicht besteht, erhält man so nicht. Es ist also fraglich, ob man solche Generatoren tatsächlich echte Zufallsbit-Generatoren nennen kann. Wir werden später sehen, daß wir diese Art der Generatoren dennoch benötigen, da sie quasi "aus dem Nichts" Zufallsbits erzeugen. Wir werden diese als sogenannte Seed-Generatoren benutzen um einen initialen Zufallsbitstring zu erzeugen. Der wesentliche Unterschied bei unserer Art der Anwendung besteht darin, daß die Ausgabe dieser Seed-Generatoren nicht beobachtet werden kann.

2.2 Pseudozufallsbits

Um die in der Praxis benötigte Menge echter Zufallsbits zu verringern, verwendet man klassischerweise Pseudozufallsbit-Generatoren. Diese deterministischen Algorithmen erzeugen aus kurzen Zufallsbitstrings lange Pseudozufallsbitstrings, die man zumindestens in einigen Anwendungsfällen anstelle gleich langer echter Zufallsbitstrings verwenden kann. Wieder werden hier die bereits weiter oben erwähnten statistischen Tests verwendet, um die Eignung dieser Generatoren für bestimmte Anwendungen zu zeigen.

Im Gegensatz dazu haben Blum und Micali [BM84], Goldwasser [GM84], und Yao [Yao82] eine Theorie initiiert, auf deren Grundlage man Pseudozufallsbit-Generatoren angeben kann, die in *allen* (also auch kryptographischen) Anwendungen echte Zufallsbit-Generatoren ersetzen können. Die Idee dabei ist, die Berechnungskapazitäten eines Beobachters mit in Betracht zu ziehen. Dadurch kann man die Forderung, daß ein Generator *keine* Informationen preisgeben darf, dahingehend abschwächen, daß er keine *für den Beobachter nützlichen* Informationen preisgeben darf. Dieser Ansatz soll Grundlage für den in dieser Arbeit entwickelten Generator sein.

Umfassend beschreibt O. Goldwasser in [Gol95] diese Theorie, die auf der Komplexitätstheorie fußt. Eine Zusammenstellung der Kernaussagen und einen guten Überblick liefert [Gol99]. Im folgenden Abschnitt werden wir die entsprechenden Konzepte der Komplexitätstheorie präsentieren und darauf basierend Pseudozufallsbit-Generatoren definieren. Außerdem geben wir eine Konstruktion für solche Generatoren an, welche die kryptographischen Primitive Oneway-Funktion (bzw. Oneway-Permutation) und Hard-Core-Prädikat nutzt. Die folgenden Definitionen und Sätze sind im wesentlichen den beiden oben genannten Quellen entnommen.

Bemerkung

Goldreich verwendet den Begriff Pseudozufallsbit-Generator ausschließlich für Generatoren der im folgenden beschriebenen Bauart. In der Praxis werden aber aus historischen Gründen auch Generatoren so bezeichnet, die nur bzgl. einiger statistischer Test die gleichen Eigenschaften wie echte Zufallsbit-Generatoren aufweisen.

2.3 Komplexitätstheorie

Die von uns betrachteten Objekte sind Bitstrings bzw. genauer die Verteilung von Bitstrings, welche Pseudozufallsbit-Generatoren erzeugen. Wir werden, wie in der Komplexitätstheorie üblich, asymptotische Aussagen treffen. Wir modellieren dazu die betrachteten Verteilungen als Ensembles. Die für unsere Anwendung benötigten Ensembles sind wie folgt definiert.

Definition 2.3.1 (Ensemble)

Gegeben eine natürliche Zahl $n \in \mathbb{N}$ und ein positives Polynom $p(n)$. Ein Ensemble X ist eine abzählbar unendliche Folge $X \stackrel{\text{def}}{=} (X_n)_{n \in \mathbb{N}}$ von Wahrscheinlichkeitsverteilungen X_n auf der Menge $\{0, 1\}^{p(n)}$. Das Polynom $p(n)$ heißt Längentyp des Ensembles.

Als Maß für die Güte der Pseudozufallsbitstrings gelten, wie eingangs erwähnt, die echten Zufallsbitstrings. Diese sind in der folgenden Definition erfaßt:

Definition 2.3.2 (Gleichverteilungs-Ensemble)

Beschreibe U_n die Gleichverteilung auf der Menge $\{0, 1\}^n$ der Bitstrings der Länge n . Das Gleichverteilungs-Ensemble U ist definiert als

$$U \stackrel{\text{def}}{=} (U_n)_{n \in \mathbb{N}}. \quad (2.2)$$

Der Längentyp von U ist gegeben als $p(n) = n$.

In der folgenden Definition ist eines der wesentlichen Konzepte angegeben. Hier geht die oben erwähnten Berechnungsmöglichkeiten eines Beobachters ein. Wir verwenden ein sogenanntes uniformes Angreifer-Modell, das heißt, wir beschreiben einen Angreifer durch einen probabilistischen Polynomzeit-Algorithmus. Das bedeutet zum einen, daß die Anzahl der elementaren Berechnungsschritte, die der Algorithmus zum Lösen eines Problems benötigt, durch ein Polynom in der Eingabelänge beschränkt ist (Polynomzeit). Zum anderen darf der Algorithmus sogenannte "Rate-Schritte" einlegen (probabilistisch): Zu bestimmten Zuständen gibt es mehrere unterschiedliche Folgezustände, die jeweils mit einer gewissen Wahrscheinlichkeit eingenommen werden. In diesem Zusammenhang findet man oft den Begriff der Turing-Maschine, der es erlaubt die obigen Begriffe exakter zu fassen. Wir werden diesen Formalismus jedoch nicht verwenden und uns mit den obigen, intuitiven Begriffserklärungen begnügen.

Wir sagen, ein Angreifer bricht ein Verfahren, wenn der Algorithmus für unendliche viele Instanzen des Problems in polynomieller Zeit in der Eingabelänge eine Lösung errechnet.

Wir wollen zwei Ensembles als gleich betrachten, wenn kein Angreifer sie unterscheiden kann.

Definition 2.3.3 (Algorithmische Ununterscheidbarkeit)

Zwei Verteilungs-Ensembles $X = (X_n)_{n \in \mathbb{N}}$ und $Y = (Y_n)_{n \in \mathbb{N}}$ heißen algorithmisch ununterscheidbar, falls für jeden probabilistischen Polynomzeit-Algorithmus A , für jedes Polynom $p(n) > 0$ und genügend große n gilt:

$$|Pr_{x \sim X_n}[A(x) = 1] - Pr_{y \sim Y_n}[A(y) = 1]| < \frac{1}{p(n)} \quad (2.3)$$

Die Wahrscheinlichkeiten werden sowohl die Verteilungen X_n bzw. Y_n als auch über die internen Münzwürfe des Algorithmus A genommen.

Existiert ein solcher Algorithmus A , nennen wir ihn Unterscheider der Ensembles X_n und Y_n .

Wir lassen bei der Abschätzung der Wahrscheinlichkeit einen Fehler zu, der allerdings kleiner sein muß als der Kehrwert jedes positiven Polynoms in n . Diese Forderung deckt sich mit der Modellierung des Angreifers als probabilistischen Polynomzeit-Algorithmus. Eine Möglichkeit, diesen sehr geringen Vorteil beim Raten zu nutzen, bestünde zum Beispiel darin, den Algorithmus A hinreichend oft und unabhängig zu wiederholen. Ein solches Verfahren ist aber nicht praktikabel, da die Anzahl der Wiederholungen schon nicht mehr polynomiell in der Länge der Eingabe wäre. Diese Vorgehensweise würde also keinen Polynomzeit-Algorithmus liefern. Solche Terme, die kleiner sind als der Kehrwert jedes positiven Polynoms in n , nennen wir vernachlässigbar.

Es folgt die Definition des pseudozufälligen Ensembles:

Definition 2.3.4 (Pseudozufälliges Ensemble)

Ein Ensemble $X = (X_n)_{n \in \mathbb{N}}$ heißt pseudozufällig, wenn es vom Gleichverteilungs-Ensemble U algorithmisch ununterscheidbar ist, falls also für jeden probabilistischen Polynomzeit-Algorithmus A , für jedes Polynom $p(n) > 0$ und genügend große n gilt:

$$|Pr_{x \sim X_n}[A(x) = 1] - Pr_{y \sim U_n}[A(y) = 1]| < \frac{1}{p(n)}. \quad (2.4)$$

Wir können nun (kryptographisch) sichere Pseudozufallsbit-Generatoren definieren.

2.4 Pseudozufallsbit-Generator**Definition 2.4.1 (Pseudozufallsbit-Generator)**

Ein deterministischer Algorithmus $G : \{0, 1\}^* \rightarrow \{0, 1\}^*$ heißt (kryptographisch sicherer) Pseudozufallsbit-Generator, falls G folgende Eigenschaften besitzt:

1. (Effizienz) G ist ein Polynomzeit-Algorithmus.
2. (Expansion) Es existiert ein positives Polynom l mit $l(n) \geq n + 1$ für $n \in \mathbb{N}$, so daß für alle $x \in \{0, 1\}^*$ gilt $|G(x)| = l(|x|)$. Das Polynom $l(n)$ heißt Expansionsfunktion von G .
3. (Pseudozufälligkeit) Das durch G induzierte Ensemble $(G(U_n))_{n \in \mathbb{N}}$ ist pseudozufällig.

In der Definition fordern wir lediglich, daß die Ausgabe um mindestens ein Bit länger ist als die Eingabe. In der Praxis ist man jedoch daran interessiert, daß die Ausgabe wesentlich (polynomiell) länger als die Eingabe ist. Um dies zu erreichen, ist der folgende Satz nützlich.

Satz 2.4.1 (Verbesserung der Expansions-Eigenschaft)

Sei G ein Pseudozufallsbit-Generator mit Expansionsfunktion $l(n) = n + 1$, $l'(n)$ eine beliebige (polynomielle) Expansionsfunktion, die in Polynom-Zeit in n berechenbar ist. Bezeichne $G_0(x)$ das $|x|$ -Bit lange Präfix und $G_1(x)$ das ein Bit lange Postfix von $G(x)$, d.h. $G(x) = G_0(x)G_1(x)$. Dann gilt

$$G'(s) \stackrel{\text{def}}{=} \sigma_1 \sigma_2 \dots \sigma_{l'(|s|)}, \quad (2.5)$$

wobei $x_0 = s$, $\sigma_i = G_1(x_{i-1})$ und $x_i = G_0(x_{i-1})$, für $i = 1, \dots, l'(|s|)$

2.5 Konstruktion

Wir wenden uns nun einer Konstruktion von Pseudozufallsbit-Generatoren zu. Sie basiert auf dem kryptographischen Primitiv "Oneway-Funktion". Vereinfacht gesagt haben Funktionen die Oneway-Eigenschaft, wenn sie effizient zu berechnen, aber nicht effizient zu invertieren sind. So erhalten wir zum einen die Effizienzeigenschaft des Generators, zum anderen können wir aus der Schwierigkeit, eine Oneway-Funktion zu invertieren, die Pseudozufälligkeit gewinnen. Zunächst die Definition:

Definition 2.5.1 (Oneway-Funktion)

Eine Funktion $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ heißt Oneway-Funktion, falls sie folgenden Bedingungen genügt:

1. Die Funktion f ist berechenbar in polynomieller Zeit in der Eingabelänge.
2. Für jeden probabilistischen Polynomzeit-Algorithmus A , jedes Polynom $p(n) > 0$ gilt

$$\Pr_{x \sim U_n} [A(f(x)) \in f^{-1}(f(x))] < \frac{1}{p(n)}, \quad (2.6)$$

wobei U_n die Gleichverteilung auf Binärstrings der Länge n bezeichnet.

Gilt für alle $x \in \{0, 1\}^*$ zudem $|f(x)| = |x|$ und ist f bijektiv, so heißt f Oneway-Permutation.

Die erste Eigenschaft ist notwendig, da sonst der Generator nicht effizient ist. Die zweite Eigenschaft garantiert, daß wir zu einem gegebenen Bildelement y ein zugehöriges Urbildelement $f^{-1}(y)$ nicht effizient bestimmen können. Dies bedeutet aber nicht, daß alle Bits des Urbildelements gleichermaßen schwer zu berechnen sind. Goldreich gibt in [Gol95, Abschnitt 2.5] als Beispiel hierfür eine Oneway-Funktion, welche die Hälfte der Bits der Eingabe konstant läßt. D.h. die Hälfte der Bits des Urbilds kann man direkt am Bild ablesen.

Wir werden nun eine Konstruktion sehen, die einen Teil der Informationen, sprich bestimmte Bits, sicher vor einem Beobachter verhüllt. Diese in polynomieller Zeit aus dem Urbildelement x berechenbaren Bits modellieren sozusagen den "harten Kern", an dem die Invertierung einer Oneway-Funktion tatsächlich scheitert. Formalisiert wird dieser Sachverhalt mit dem Begriff des Hard-Core-Prädikats.

Definition 2.5.2 (Hard-Core-Prädikat)

Ein in der Eingabelänge in polynomieller Zeit berechenbares Prädikat $b : \{0, 1\}^n \rightarrow \{0, 1\}$ heißt Hard-Core-Prädikat bezüglich einer Funktion f , falls für jeden probabilistischen Polynomzeit-Algorithmus A , jedes Polynom $p(n) > 0$ und alle genügend großen n gilt

$$\Pr_{x \sim U_n} [A(f(U_n)) = b(U_n)] < \frac{1}{2} + \frac{1}{p(n)} \quad (2.7)$$

Da $b(x)$ aus x nach Definition leicht berechnet werden kann, gibt es zwei Gründe, warum $b(x)$ aus der Kenntnis von $f(x)$ schwer zu bestimmen ist. Zum einen kann dies daran liegen, daß unter der Abbildung f Informationen verloren gehen. So ist zum Beispiel das Prädikat $b(\sigma x) = \sigma$ ein Hard-Core-Prädikat zur Funktion $f(\sigma x) = 0x$, mit $\sigma \in \{0, 1\}$ und $x \in \{0, 1\}^*$ (vgl. [Gol95]). Ein solches Hard-Core-Prädikat ist *informationstheoretisch* sicher.

Im anderen Fall, wenn unter f keine Informationen verloren gehen (f also injektiv ist), folgt, daß f eine Oneway-Funktion sein muß. Wäre f keine Oneway-Funktion, wäre es nämlich möglich, x aus Kenntnis von $f(x)$ und anschließend $b(x)$ aus Kenntnis von x in polynomieller Zeit zu berechnen. Ein solches Hard-Core-Prädikat heißt *algorithmisch* sicher.

Zur Definition ist noch zu bemerken, daß wir o.B.d.A. nur solche Algorithmen A betrachten, die das Hard-Core-Prädikat mit einer Wahrscheinlichkeit von *mindestens* $\frac{1}{2}$ raten. Jeder Algorithmus mit schlechterer Ratewahrscheinlichkeit als $\frac{1}{2}$ kann durch Invertieren der Ausgabe leicht in einen solchen überführt werden.

Wir konstruieren nun einen Pseudozufallsbit-Generator:

Satz 2.5.1 (Konstruktion mit Oneway-Permutation)

Sei $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ eine Oneway-Permutation und $b : \{0, 1\}^* \rightarrow \{0, 1\}$ ein Hard-Core-Prädikat bezüglich f . Dann ist der Algorithmus $G(s) \stackrel{\text{def}}{=} f(s)b(s)$ ein kryptographisch sicherer Pseudozufallsbit-Generator.

Die Beweisidee ist die Folgende: Wir nehmen an, das Ensemble $(f(U_n)b(U_n))_{n \in \mathbb{N}}$ sei nicht pseudozufällig. Dann gibt es einen probabilistischen Polynomzeit-Algorithmus A' entsprechend Definition 2.3.3. Diesen könnten wir benutzen, um $b(U_n)$ mit nicht vernachlässigbarer Wahrscheinlichkeit von U_1 zu unterscheiden. Der Beweis ist in [Gol95, Abschnitt 3.4.1] angegeben.

Bei dieser Konstruktion haben wir die gleichen Voraussetzungen wie in der Vorbemerkung zu Satz 2.4.1. Die Ausgabe ist lediglich um ein Bit länger als die Eingabe. Wir wenden also auch hier die iterative Konstruktion zur Verbesserung der Expansions-Eigenschaften an. Dies führt zu folgender Darstellung:

Satz 2.5.2

Seien f, b wie in Satz 2.5.1. Sei $l(n) \geq n+1$ ein Polynom in der Eingabelänge n . Sei der Seed s gemäß U_n zufällig gewählt in der Menge der Bitstrings $\{0, 1\}^n$. Wir definieren den Generator $G : \{0, 1\}^n \rightarrow \{0, 1\}^{l(n)}$ wie folgt.

$$G(s) \stackrel{\text{def}}{=} b(s_0) \dots b(s_i) \dots b(s_{l(n)-1}), \quad (2.8)$$

wobei

$$s_0 = s \quad (2.9)$$

$$s_i = f(s_{i-1}) \text{ für } 0 < i < l(n). \quad (2.10)$$

Wir geben also eine Folge von Prädikaten auf der Funktion f aus, die nach den Sätzen 2.4.1 und 2.5.1 pseudozufällig ist.

Die Oneway-Eigenschaft von f besagt, daß $f(U_n)$ von U_n ununterscheidbar ist. Damit ist aber auch $b(f(U_n))$ von $b(U_n)$ und damit von U_1 ununterscheidbar. Entsprechendes gilt für $b(f(\dots f(U_n) \dots))$. Daraus folgt, daß der $l(n)$ -Bit lange String der aneinandergehängten Prädikate von $U_{l(n)}$ ununterscheidbar ist.

2.6 Schaaren von Oneway-Funktionen

Wir haben in der Definition 2.5.1 Oneway-Funktionen eingeführt, die Bitstrings beliebiger Länge abbilden. Wenn wir uns später Kandidaten für solche Funktionen zuwenden, wird sich dies als unhandlich erweisen. Wir werden deshalb noch eine Konstruktion einführen, bei der wir Funktionen verwenden, die auf Bitstring fester Länge n operieren (vgl. [Gol95, §§ 2.4.2]).

Die Idee ist, eine unendliche Schar von Funktionen f_i anzugeben, wobei i Element einer unendlichen Indexmenge \bar{I} ist. Die f_i sind hierbei jeweils auf einem endlichen Wertebereich D_i definiert. Wenn wir zu jeder Eingabelänge n (mindestens) einen Index $i \in \bar{I}$ angeben können, sodaß für den Wertebereich $D_i \subseteq \{0, 1\}^n$ die Funktion f_i definiert ist als $f_i : D_i \rightarrow \{0, 1\}^*$, haben wir eine ähnliche Situation wie in Definition 2.5.1 erreicht. Sicherlich müssen wir noch für einige Aspekte der Berechenbarkeit Sorge tragen, was wir in der folgenden Definition tun wollen.

Definition 2.6.1 (Scharen von Oneway-Funktionen)

Sei \bar{I} eine unendliche Indexmenge. Eine Schar von Oneway-Funktionen ist ein Tripel (I, D, F) mit folgenden Eigenschaften:

1. I ist ein probabilistischer Polynomzeit-Algorithmus, der an Eingabe von n (in unärer Darstellung¹) einen Index i berechnet, sodaß die Wahrscheinlichkeit, daß $i \notin \bar{I}$ gilt, vernachlässigbar klein ist.
2. D ist ein probabilistischer Polynomzeit-Algorithmus, der an Eingabe eines Index i ein Element $x \in D_i$ berechnet, wobei $D_i \subseteq \{0, 1\}^n$ und x gleichverteilt in D_i gewählt wird.
3. F ist ein Polynomzeit-Algorithmus, der an Eingabe (x, i) die Funktion $f_i(x)$ auswertet.
4. Die Funktionen $f_i, i \in \bar{I}$ sind Oneway-Funktionen, daß heißt für jeden probabilistischen Polynomzeit-Algorithmus A , jedes positive Polynom p und genügenden große n gilt:

$$\Pr [A(f_{I_n}(X_n), I_n) \in f_{I_n}^{-1}f_{I_n}(X_n)] < \frac{1}{p(n)}, \quad (2.11)$$

wobei I_n die Verteilung der Ausgabe von Algorithmus I an Eingabe n (unär) und X_n die Verteilung der Ausgabe von Algorithmus D an Eingabe I_n bezeichnet.

2.7 Steigerung der Effizienz

Wir haben uns bisher darauf beschränkt, pro Iteration ein einzelnes Bit auszugeben, das Hard-Core-Prädikat. Es stellt sich nun die Frage, ob es möglich ist mehrere unterschiedliche Hard-Core-Prädikate gleichzeitig zu verwenden, um eine größere Menge Bits pro Iteration zu erzeugen. Dies ist möglich, wenn die Prädikate *simultan sicher* sind. Das heißt, wenn jedes Prädikat auch unter der Bedingung ein Hard-Core-Prädikat ist, daß der Wert der anderen Prädikate bekannt ist. Wir nennen die Konkatenierung simultan sicherer Hard-Core-Prädikate schlicht Hard-Core-Funktion. Die Hard-Core-Funktionen können in den obigen Konstruktionen anstelle der Hard-Core-Prädikate verwendet werden. Die simultane Sicherheit garantiert, daß die Beweise gültig bleiben.

¹Mit diesem kleinen Trick verhindern wir, daß der Algorithmus I von vorneherein exponentielle Laufzeit hat. In der späteren Realisierung werden wir ebenfalls gezwungen sein, zu vorgegebener Länge n (in Binärcodierung), einen n Bit langen String zu wählen. Alleine diesen String hinzuschreiben, bedeutet schon exponentielle Laufzeit in der Eingabelänge, jedoch fällt dieser Aspekt, im Gesamtkontext betrachtet, nicht ins Gewicht.

Wie wir diese Konstruktion verwenden, werden wir im nächsten Kapitel sehen. Dort werden wir als Kandidat für Oneway-Funktion die diskrete Exponentiation in endlichen Gruppen angeben. Wir werden zeigen, daß die Oneway-Eigenschaft dann gegeben ist, wenn das Diskrete-Logarithmen-Problem in der jeweiligen Gruppe schwer ist.

Kapitel 3

Elliptische Kurven und diskrete Logarithmen

Ziel der Diplomarbeit ist die Erstellung eines Pseudozufallsbit-Generators basierend auf elliptischen Kurven. Wir werden in diesem Kapitel elliptische Kurven über endlichen Körpern großer Charakteristik einführen. Die Menge der Punkte auf einer solchen Kurve läßt sich zusammen mit einer entsprechenden Verknüpfung als endliche Abelsche Gruppe auffassen. In diesem Kontext geben wir die diskrete Exponentiation als einen Kandidaten für eine Funktion an, welche die Oneway-Eigenschaft besitzt, wenn die Umkehrfunktion, der diskrete Logarithmus, in der entsprechenden Gruppe schwer zu berechnen ist (DL-Problem). Die Oneway-Eigenschaft ist essentiell für die kryptographische Sicherheit des Generators. Genau an dieser Stelle stoßen wir aber an die Grenzen des Beweisbaren, da es lediglich eine Annahme ist, daß das DL-Problem bezüglich elliptischer Kurven schwer ist.

3.1 Grundlagen

3.1.1 Gruppen

In diesem Abschnitt präsentieren wir einige Begrifflichkeiten und Eigenschaften bezüglich Gruppen. Wir werden dies in einem etwas allgemeineren Rahmen tun, als es für Elliptische-Kurven-Gruppen, die für uns von Interesse sind, eigentlich nötig wäre. Wir werden im nächsten Kapitel jedoch einige Ergebnisse der Dissertation von B.S. Kaliski ([Kal88]) verwenden, welche in diesem allgemeineren Kontext angegeben sind.

Definition 3.1.1 ((additive) Gruppen)

Eine Gruppe ist eine Menge G mit einer zweistelligen Operation, in unserem Fall die Addition. In einer additiven Gruppe $(G, +)$ gelten für alle $a, b, c \in G$ die folgenden Gruppenaxiome:

1. (Abgeschlossenheit) $a + b \in G$
2. (Neutrales Element) Es existiert ein neutrales Element $0 \in G$, sodaß $0 + a = a + 0 = a$.

3. (Inverses Element) Es existiert ein inverses Element $(-a) \in G$, sodaß $a + (-a) = 0$.

4. (Assoziativität) $(a + b) + c = a + (b + c)$.

Einen Gruppe heißt kommutativ oder Abelsch, falls gilt

5. (Kommutativität) $a + b = b + a$

Wenn die Gruppenoperation aus dem Zusammenhang ersichtlich ist, werden wir der Einfachheit halber anstelle von $(G, +)$ schlicht G schreiben.

Definition 3.1.2 (Diskrete Exponentiation)

Gegeben eine Abelsche Gruppe G und ein Element $a \in G$. Für $\nu \in \mathbb{Z}$ definieren wir die diskrete Exponentiation:

$$\nu \cdot a \stackrel{\text{def}}{=} \underbrace{a + \cdots + a}_{\nu} \quad (3.1)$$

Insbesondere gilt $0_{\mathbb{Z}} \cdot a = 0_G$. In einer multiplikativ geschriebenen Gruppe entspricht die skalare Multiplikation der Exponentiation mit ganzzahligen Exponenten, geschrieben a^ν . Es sei angemerkt, daß $-1 \cdot a$ der Invertierung von a entspricht.

Definition 3.1.3 (Erzeuger)

Gegeben eine endliche Abelsche Gruppe G und eine Teilmenge $X \subseteq G$. Die von X erzeugte Untergruppe H ist definiert als

$$H = \langle X \rangle \stackrel{\text{def}}{=} \{ \epsilon_1 x_1 + \cdots + \epsilon_n x_n \mid x_1 \dots x_n \in X, \epsilon_1 \dots \epsilon_n \in \{1, -1\}, n \in \mathbb{N} \} \quad (3.2)$$

Existiert eine einelementige Teilmenge $X = \{x\} \subseteq G$ mit $\langle X \rangle = G$, so heißt x Erzeuger von G . Eine Gruppe, die einen Erzeuger hat, heißt zyklische Gruppe. Wir schreiben vereinfacht $\langle x \rangle$ anstelle von $\langle \{x\} \rangle$.

Definition 3.1.4 (Elementordnung)

Sei G eine Abelsche Gruppe. Für ein Element $a \in G$ ist die Elementordnung wie folgt definiert

$$\text{order}(a) = \begin{cases} \min\{\nu \in \mathbb{N}/\{0\} \mid \nu a = 0\} & , \text{ falls so ein } \nu \text{ existiert,} \\ \infty & , \text{ sonst.} \end{cases} \quad (3.3)$$

Gewissermaßen die Umkehrfunktion der oben definierten diskreten Exponentiation ist der diskrete Logarithmus.

Definition 3.1.5 (Diskreter Logarithmus)

Gegeben eine Abelsche Gruppe $(G, +)$ und Elemente $a, g \in G$. Das Element g erzeugt eine zyklische Untergruppe $\langle g \rangle \subseteq G$ mit $|\langle g \rangle| = \text{order}(g)$ vielen Elementen. Falls $a \in \langle g \rangle$ ist, existiert eine ganze Zahl ν mit $0 \leq \nu < |\langle g \rangle|$, sodaß

$$a = \nu g. \quad (3.4)$$

Die Zahl ν heißt diskreter Logarithmus von a zu Basis g . Wir schreiben

$$\nu = \log_g(a) \iff \nu \cdot g = a. \quad (3.5)$$

Es gelten folgende Rechengesetze: Seien $x, y \in \langle g \rangle$ und $\mu \in \mathbb{Z}$

$$\log_g(x + y) = \log_g(x) + \log_g(y) \quad (3.6)$$

$$\log_g(\mu x) = \mu \log_g(x) \quad (3.7)$$

Satz 3.1.1 (Gruppenstruktur)

Sei G eine endliche Abelsche Gruppe, dann existiert ein $r \in \mathbb{N}$, sodaß folgende eindeutige Isomorphie gilt

$$G \cong (\mathbb{Z}/\nu_1\mathbb{Z}) \times \cdots \times (\mathbb{Z}/\nu_r\mathbb{Z}), \quad (3.8)$$

wobei ν_i von ν_{i+1} geteilt wird für $1 \leq i < r$. Die Gruppen $(\mathbb{Z}/\nu_i\mathbb{Z})$ sind zyklische Gruppen mit ν_i Elementen. Wir nennen (ν_1, \dots, ν_r) Gruppenstruktur von G . Die ganze Zahl r heißt Rang von G .

Lemma 3.1.1

Gegeben eine endliche Abelsche Gruppe G . Aus Gleichung (3.8) folgt:

1. Für alle $a \in G$ gilt $\text{order}(a) | \nu_1$
2. $\nu_1 = \max\{\text{order}(a) | a \in G\}$ heißt Maximalordnung.
3. Es existieren Tupel $(g_1, \dots, g_r) \in G^r$, sodaß jedes $a \in G$ eindeutig geschrieben werden kann als

$$a = \alpha_1 g_1 + \cdots + \alpha_r g_r, \quad 0 \leq \alpha_i < \nu_i. \quad (3.9)$$

Ein solches Tupel heißt allgemein Erzeuger-Tupel, für Rang $r = 2$ Erzeuger-Paar und für $r = 1$, wie oben, Erzeuger.

4. Das Tupel (ν_1, \dots, ν_r) heißt Gruppenstruktur.

Wenn wir später in solchen Gruppen rechnen, wird es nötig sein, ihre Erzeuger zu bestimmen. Der folgende Satz garantiert uns, daß es davon viele gibt und wir uns darauf beschränken können, zufällig Gruppenelemente zu wählen und zu überprüfen, ob sie Erzeuger sind. Herleitung und Beweis finden sich in [Kal88, Lemma 4.2].

Satz 3.1.2 (Anzahl der Erzeuger-Tupel)

Sei G eine endliche Abelsche Gruppe mit ν Elementen und Gruppenstruktur (ν_1, \dots, ν_r) . Für den Anteil der Erzeuger-Tupel $(x_1, \dots, x_r) \in G^r$ erhält man folgende Abschätzung

$$\prod_{1 \leq i \leq r} \frac{\varphi(\nu_i)}{\nu_i} \geq \prod_{1 \leq i \leq r} \frac{1}{6 \ln(1 + \ln \nu_i)} \geq \frac{1}{6^r (\ln(1 + \ln \nu))^r}, \quad (3.10)$$

wobei $\varphi(\nu)$ die Eulersche Phi-Funktion bezeichnet.

3.1.2 Folgen regulärer Gruppen

Wir haben in Satz 2.5.1 des letzten Kapitels den Pseudozufallsbit-Generator zunächst für Oneway-Permutationen definiert, die Eingaben beliebiger Länge akzeptiert. Das läßt sich aber nicht in einer endlichen Gruppe bewerkstelligen. Wir haben deshalb die Definition 2.6.1 eingeführt, die es uns erlaubt, unendliche Folgen endlicher Mengen zu verwenden. Entsprechendes für endliche Gruppen leistet folgende Definition:

Definition 3.1.6 (Folgen von Gruppen)

Sei \bar{I} eine Indextmenge. Eine Klasse von Gruppen S ist eine Folge von Mengen S_i von Gruppen

Es ist natürlich sinnvoll, in einer Menge S_i nur Gruppen zusammenzufassen, für die der Index i eine Bedeutung hat. Wir werden den Index i als Problemgröße interpretieren, in dem Sinne, daß der Zeitaufwand für Berechnungen in einer Gruppe $G \in S_i$ mit der Größe von i skaliert (d.h. eine Funktion in i ist). Wir fassen deshalb Gruppen zusammen, deren Ordnung in Binärdarstellung (ungefähr) i Bits hat. In die folgenden Definition geht sehr stark unser Bestreben ein, die Konstruktionen auf Rechnersysteme abzubilden. Wir müssen sicherstellen, daß alle verwendeten Objekte (Gruppen, Gruppenelemente) und alle Berechnungen bezüglich dieser Objekte in effizienter Weise in Rechnersystemen abgebildet werden können. Gruppen für die das gemäß folgender Definition möglich ist, nennt Kaliski *regulär* (vgl. [Kal88, §5.4.1]).

Definition 3.1.7 (Folgen regulärer Gruppen)

Sei $S = (S_i)_{i \in \mathbb{N}}$ eine Klasse kommutativer Gruppen. Die Klasse S heißt *regulär*, falls

1. Konstanten c_1 und c_2 existieren, sodaß für genügend große i für alle Gruppen $G \in S_i$ gilt:

$$c_1 2^i \leq |G| \leq c_2^i, \quad (3.11)$$

2. ein Polynomzeit-Algorithmus A existiert, der an Eingabe zweier Elemente $a, b \in G, G \in S_i$ die Gruppenoperation $a + b$ berechnet. Selbiges gilt auch für die Berechnung von $-a$.
3. ein probabilistischer Polynomzeit-Algorithmus D existiert, der an Eingabe von i und einer Gruppe $G \in S_i$ gleichverteilt zufällig ein Element $x \in G$ wählt, wobei der Algorithmus fehlerhaft sein kann, solange die Wahrscheinlichkeit für eine falsche Wahl vernachlässigbar ist.
4. ein probabilistischer Polynomzeit-Algorithmus I existiert, der an Eingabe von i , gleichverteilt zufällig eine Gruppe G in S_i auswählt (gemeint ist hier die Beschreibung von G als Bitstring!).

Zu Beachten ist, daß dies die (teilweise) Umsetzung von Definition 2.6.1 auf Klassen von Gruppen darstellt. Die Algorithmen D und I entsprechen sich jeweils. Der in Definition 2.6.1 geforderte Algorithmus F hat hier zunächst keine Entsprechung, aber sicherlich ist der hier angegebene Algorithmus A , d.h. die effiziente Berechenbarkeit der Gruppenoperation, eine notwendige Voraussetzung für F .

3.1.3 Körper

Wir betrachten elliptische Kurven über endlichen Körpern. Es folgen nun einige grundlegenden Eigenschaften solcher Körper.

Definition 3.1.8 (Körper)

Gegeben eine Menge K , einen Additions-Operator $+$: $K \times K \rightarrow K$ und einen Multiplikations-Operator \cdot : $K \times K \rightarrow K$. $\mathbb{K} \stackrel{\text{def}}{=} (K, +, \cdot)$ heißt Körper, falls

1. $\mathbb{K}^+ \stackrel{\text{def}}{=} (K, +)$ eine additive Abelsche Gruppe bildet mit neutralem Element $0_{\mathbb{K}^+}$,
2. $\mathbb{K}^* \stackrel{\text{def}}{=} (K/\{0_{\mathbb{K}^+}\}, \cdot)$ eine multiplikative Abelsche Gruppe bildet mit neutralem Element $1_{\mathbb{K}^*}$ und
3. das Distributiv-Gesetz gilt, nämlich $a(b + c) = ab + ac$ für alle $a, b, c \in K$.

Definition 3.1.9 (Charakteristik)

Für einen Körper \mathbb{K} definiert man die Charakteristik von \mathbb{K} wie folgt:

$$\text{char}(\mathbb{K}) = \begin{cases} 0 & , \text{ falls } \text{order}_{\mathbb{K}^+}(1) = \infty \\ \text{order}_{\mathbb{K}^+}(1_{\mathbb{K}^*}) & , \text{ sonst.} \end{cases} \quad (3.12)$$

Satz 3.1.3

Sei \mathbb{K} ein Körper. Den kleinsten Unterkörper \mathbb{K}_0 von \mathbb{K} nennt man Primkörper von \mathbb{K} . Falls $\text{char}(\mathbb{K}) = 0$ gilt, ist $\mathbb{K}_0 = \mathbb{Q}$. Falls $\text{char}(\mathbb{K})$ eine ganze Zahl $p \in \mathbb{N}$ ist, gilt für den Primkörper:

$$K_0 = \{\lambda \cdot 1_{\mathbb{K}^*} \mid \lambda \in \mathbb{N}\} \cong \mathbb{Z}_p. \quad (3.13)$$

Daraus folgt, daß p eine Primzahl ist. Man kann \mathbb{K} als \mathbb{K} -Vektorraum über \mathbb{K}_0 auffassen, woraus sich ergibt, dass $|K| = p^r$ für p prim und $r \in \mathbb{N}$. Insbesondere hat jeder endliche Körper Primzahlcharakteristik. Wir werden im wesentlichen endliche Körper mit p Elementen verwenden (geschrieben \mathbb{F}_p), wobei p eine Primzahl ist.

3.2 Elliptische Kurven

Wir wollen uns nun den elliptischen Kurven zuwenden. Da dieses Gebiet ausgesprochen umfangreich ist, beschränken wir uns auf die für uns relevanten Teilbereiche. In kryptographischen Anwendungen kommen in der Regel nur elliptische Kurven über endlichen Körpern \mathbb{F}_{q^m} zum Einsatz, wobei q eine Primzahl und m eine positive ganze Zahl ist. Für unsere Belange können wir uns noch weiter einschränken auf Körper \mathbb{F}_p mit Primzahl $p > 3$ ¹. Wir werden im folgenden, auch wenn wir nur einen Spezialfall elliptischer Kurven betrachten, allgemein von elliptischen Kurven reden.

¹In den späteren Anwendungen ist p sehr viel größer als drei, nämlich mindestens eine 160-Bit-Zahl

Definition 3.2.1 (Elliptische Kurve)

Sei $p > 3$ eine Primzahl und $a, b \in \mathbb{F}_p$. Wir definieren die elliptische Kurve $E_{a,b}(\mathbb{F}_p)$ als die Menge der Lösungen $(x, y) \in (\mathbb{F}_p)^2$ der kubischen Gleichung

$$E_{a,b}(\mathbb{F}_p) : y^2 = x^3 + ax + b, \quad (3.14)$$

falls

$$4a^3 - 27b^2 \neq 0 \quad (3.15)$$

gilt, zusammen mit einem Punkt \mathcal{O}^2 , dem Punkt im Unendlichen. Wenn die Parameter a, b aus dem Zusammenhang klar oder nicht weiter von Bedeutung sind, schreiben wir vereinfachend $E(\mathbb{F}_p)$.

Die Gleichung 3.14 eine sogenannte *reduzierte Weierstrass-Normalform*. Diese kann man unter der Bedingung, daß die Kurve über einem endlichen Körper \mathbb{F}_p definiert ist und daß $p > 3$ gilt, mittels zulässiger Transformationen aus der *allgemeinen Weierstrass-Normalform* herleiten. In unseren Anwendungen erfüllen wir diese Voraussetzung immer, wir werden deshalb vereinfachend annehmen, daß wir elliptische Kurven durch die Gleichung 3.14 beschreiben können. Die Bedingung 3.15 gewährleistet, daß die Kurve nicht singulär ist. Normalerweise unterscheidet man zwischen singulären und nichtsingulären elliptischen Kurven. Da sich die singulären Kurven für kryptographische Zwecke nicht eignen, wollen wir im folgenden unter elliptischen Kurven ausschließlich nichtsinguläre elliptische Kurven verstehen. Für die Verwendung elliptischer Kurven in kryptographischen Anwendungen ist der folgende Satz von großer Bedeutung:

Satz 3.2.1 (Elliptische-Kurve-Gruppe)

Eine elliptische Kurve $E(\mathbb{F}_p)$ bildet mit der Operation "Tangenten und Sekanten" $+ : E(\mathbb{F}_p) \times E(\mathbb{F}_p) \rightarrow E(\mathbb{F}_p)$ eine endliche Abelsche Gruppe $(E(\mathbb{F}_p), +)$ mit \mathcal{O} als neutralem Element. Wenn aus dem Zusammenhang ersichtlich ist, was gemeint ist, schreiben wir $E(\mathbb{F}_p)$ für $(E(\mathbb{F}_p), +)$. Die Gruppenoperationen (Addition und Invertierung) lassen sich effizient in Rechensystem durchführen vgl. [Ham98, §2.5].

Satz 3.2.1 beinhaltet sehr weitreichende Erkenntnisse über elliptische Kurven, die wir für unsere Konstruktionen einfach verwenden wollen. Eine umfassende Einführung in diese Materie findet sich unter anderem in [Ham98]. Über die Anzahl der Punkte gibt das Hasse'sche Theorem Aufschluß.

3.2.1 Anzahl der Punkte**Satz 3.2.2 (Theorem von Hasse)**

Sei p eine Primzahl und $m \in \mathbb{N}$. Sei $E(\mathbb{F}_q)$ eine elliptische Kurve mit $q = p^m$ und p Primzahl. Dann existiert eine ganze Zahl t , sodaß

$$|E(\mathbb{F}_q)| = q + 1 - t \quad \text{mit} \quad |t| \leq 2\sqrt{q}. \quad (3.16)$$

Die ganze Zahl t heißt *Spur der elliptischen Kurve*. Für uns ist hierbei nur der Fall $m = 1$ relevant.

²Diesen Punkt gewinnt man, wenn man elliptische Kurven in der sogenannten projektiven Darstellung betrachtet. In der von uns verwendeten affinen Darstellung läßt sich der Punkt nicht durch ein Koordinatenpaar beschreiben.

Wir wollen uns an dieser Stelle genauer ansehen, was diese Abschätzung bedeutet. Wir betrachten zu einer elliptischen Kurve $E_{a,b}(\mathbb{F}_p)$ die Funktion

$$\psi(x) \stackrel{\text{def}}{=} x^3 + ax + b \quad (3.17)$$

Wir finden zu einem Element $x \in \mathbb{F}_p$ einen Kurvenpunkt, wenn es ein $y \in \mathbb{F}_p$ mit $y^2 = \psi(x)$ gibt. In der Literatur findet man solche Problemstellungen in der Regel bezogen auf die multiplikative Gruppe $(\mathbb{Z}/p\mathbb{Z})^*$, wo man die Quadratwurzeln modulo p betrachtet. Da $\mathbb{F}_p^* \cong (\mathbb{Z}/p\mathbb{Z})^*$ ist, werden wir die Darstellung modulo p synonym verwenden. Zunächst eine Definition:

Definition 3.2.2 (Quadratischer Rest)

Gegeben sei eine Gruppe $(\mathbb{Z}/p\mathbb{Z})^*$, p eine Primzahl, und ein Element $a \in (\mathbb{Z}/p\mathbb{Z})^*$. Falls ein Element $w \in (\mathbb{Z}/p\mathbb{Z})^*$ existiert mit

$$w^2 \equiv a \pmod{p}, \quad (3.18)$$

dann heißt w *Quadratwurzel von a modulo p* und a heißt *quadratischer Rest modulo p* (geschrieben $a \in \text{QR}_p$). Insbesondere ist $-w$ ebenfalls *Quadratwurzel von a modulo p* und es gilt $(-w) \not\equiv w \pmod{p}$. Existiert keine solche Wurzel w , so heißt a *quadratischer Nichtrest modulo p* (geschrieben $a \in \text{QNR}_p$).

Eine Aussage, ob ein Element quadratischer Rest oder quadratischer Nichtrest ist modulo p , liefert uns das Jacobi-Symbol:

Definition 3.2.3

Gegeben sei eine Gruppe $(\mathbb{Z}/p\mathbb{Z})^*$, p eine Primzahl, und ein Element $a \in (\mathbb{Z}/p\mathbb{Z})^*$. Das Jacobi-Symbol ist definiert als:

$$\left(\frac{a}{p}\right) \stackrel{\text{def}}{=} \begin{cases} 1 & , \text{ falls } a \in \text{QR}_p \\ 0 & , \text{ falls } a \in \text{QNR}_p \end{cases} \quad (3.19)$$

Das Jacobi-Symbol läßt sich in polynomieller Zeit in der Eingabelänge berechnen.

Wichtig ist für uns auch folgender Satz:

Satz 3.2.3

Gegeben sei die Gruppe $(\mathbb{Z}/p\mathbb{Z})^*$, p Primzahl. Die Hälfte der $\varphi(p) = p - 1$ vielen Elemente hat Jacobi-Symbol 1.

Sei $E_{a,b}(\mathbb{F}_p)$ wie oben gegeben, dann stellen für ein $x \in \mathbb{F}_p$ bezüglich der Funktion ψ in Gleichung 3.17 fest:

1. Falls $\psi(x) \in \text{QR}_p$, liegen die zwei Punkte $(x, \pm\sqrt{\psi(x)})$ auf der Kurve.
2. Falls $\psi(x) \in \text{QNR}_p$, liegt kein Punkt mit dieser x -Koordinate auf der Kurve
3. Falls $\psi(x) = 0$, liegt der Punkt $(x, 0)$ auf der Kurve.

Wenn wir $\psi(x)$ als zufällige Abbildung annehmen, ist im Durchschnitt für die Hälfte der $x \in \mathbb{F}_p$ der Funktionswert $\psi(x) \in \text{QR}$. Nach 1 können wir also durchschnittlich p viele Punkte (x, y) zuzüglich des Punktes im Unendlichen \mathcal{O} erwarten, was sich im Theorem von Hasse widerspiegelt.

3.2.2 Struktur der Elliptische-Kurven-Gruppen

Satz 3.2.4 (Struktur der Gruppe)

Sei $E_{a,b}(\mathbb{F}_p)$ eine beliebige elliptische Kurve. Die Gruppenstruktur von $E(\mathbb{F}_q)$ ist (ν_1, ν_2) , es gilt demnach folgende Isomorphie:

$$E(\mathbb{F}_q) \cong \mathbb{Z}/\nu_1\mathbb{Z} \times \mathbb{Z}/\nu_2\mathbb{Z}, \quad (3.20)$$

wobei ν_1 von ν_2 geteilt wird. Der Rang von $E(\mathbb{F}_q)$ ist also höchstens zwei. Gilt $\nu_2 = 1$, so ist der Rang eins, die Gruppe also zyklisch.

3.2.3 Binärkodierung der Kurvenpunkte

Gegeben eine elliptische Kurve $E(\mathbb{F}_p)$ mit $y^2 = x^3 + ax + b$. Einen Punkt P auf der Kurve (mit Ausnahme des Punkts \mathcal{O}) können wir durch seine Koordinaten $(x, y) \in \mathbb{F}_p^2$ beschreiben. Diese Elemente in \mathbb{F}_p wiederum können wir durch ganze Zahlen $\gamma \in \{0, \dots, p-1\}$ repräsentieren und als solche binär kodieren. Folgende Beobachtung macht die Kodierung noch effizienter: Für die y -Koordinate gilt nämlich

$$y = \pm \sqrt{x^3 + ax + b} \quad (3.21)$$

Das heißt, die y -Koordinate läßt sich bis auf das Vorzeichen aus der x -Koordinate berechnen. Es ist folglich ausreichend, zusätzlich zur x -Koordinate das Vorzeichen von y zu speichern, wobei wir dieses Vorzeichen wie folgt definieren:

$$\text{sgn}(y) = \begin{cases} 0 & , \text{ falls } y < \lfloor \frac{p}{2} \rfloor \\ 1 & , \text{ falls } y \geq \lfloor \frac{p}{2} \rfloor \end{cases} \quad (3.22)$$

Für die Speicherung von x benötigen wir $\lfloor \lg(p) \rfloor + 1$ viele Bits, folglich ist die Kodierung des Punktes P $|\text{code}(P)| = \lfloor \lg(p) \rfloor + 2$ Bits lang. Wie wir in Abschnitt 3.2.1 gesehen haben, ist diese Abbildung in die Menge der Binärstrings $\{0, 1\}^{\lfloor \lg(p) \rfloor + 2}$ nicht surjektiv.

3.2.4 Folgen von Elliptische-Kurven-Gruppen

Wir bündeln die Elliptische-Kurven-Gruppen als Folgen von Gruppen gemäß Definition 3.1.6.

Definition 3.2.4

Die Menge EK_i der Elliptischen-Kurven-Gruppen zum Parameter i definieren wir wie folgt:

$$EK_i \stackrel{\text{def}}{=} \{E_{a,b}(\mathbb{F}_p) \mid p \text{ Primzahl mit } (\lfloor \lg p \rfloor + 1) = i \text{ und } a, b \in \mathbb{F}_p\} \quad (3.23)$$

Die Folge der Elliptischen-Kurven-Gruppen ist dann

$$EK \stackrel{\text{def}}{=} (EK_i)_{i \in \mathbb{N}} \quad (3.24)$$

Satz 3.2.5

Die Gruppen in EK sind regulär, ihr Rang ist höchstens zwei.

Beweis

Wir weisen die Bedingungen in Definition 3.1.7 nach:

1. Die Konstanten c_1 und c_2 garantiert uns das Theorem von Hasse (3.2.2 zusammen mit der Tatsache, daß die Primzahl p nach Voraussetzung in der Größenordnung 2^i liegt. Wähle z.B. $c_1 = \frac{1}{3}$ und $c_2 = 3$.
2. Die effiziente Berechenbarkeit der Gruppenelemente garantiert uns Satz 3.2.1.
3. Die Möglichkeit, Kurvenpunkte zufällig gleichverteilt wählen zu können, läßt sich wie folgt sicherstellen. Wir setzen hierbei voraus, daß wir die Gruppenstruktur (ν_1, ν_2) und ein Erzeuger-Tupel (G_1, G_2) von $E_{a,b}(\mathbb{F}_p)$ kennen. Wir werden später zeigen, daß diese Annahme gerechtfertigt ist. Dann läßt sich das zufällige Wählen eines Punktes wie folgt realisieren:
 - (a) Wähle zufällig, gleichverteilt eine ganze Zahl $0 \leq \alpha < \nu = \nu_1 \cdot \nu_2$.
 - (b) Berechne den Punkt $P = (\alpha \bmod \nu_1)G_1 + \left\lfloor \frac{\alpha}{\nu_1} \right\rfloor G_2$.
4. Dies läßt sich wie folgt realisieren:
 - (a) Wähle zufällig, gleichverteilt eine Primzahl p . Diese beschreibt den Körper \mathbb{F}_p .
 - (b) Wähle zufällig, gleichverteilt zwei Elemente $a, b \in \mathbb{F}_p$.

Satz 3.2.4 garantiert, daß die Elliptische-Kurven-Gruppen höchstens Rang zwei haben.

Definition 3.2.5 (Elliptische-Kurven-Instanz)

Sei EK die Folge Elliptischer-Kurven-Gruppen. Eine Instanz zum Index i besteht aus einer elliptischen Kurve $E_{a,b}(\mathbb{F}_p)$ zum Parameter i , einem Erzeuger-Tupel (G_1, G_2) und einem Punkt P auf der Kurve. Wir schreiben: $\langle E_{a,b}(\mathbb{F}_p), (G_1, G_2), P \rangle$.

3.3 Erzeugung Elliptischer Kurven zum Index i

Wir wollen nun das zufällige Erzeugen elliptischer Kurven zu gegebener Problemgröße i genauer betrachten. Insbesondere müssen wir in der Lage sein, zusätzlich zur eigentlichen Kurve ihre Gruppenstruktur und ein Generator-Paar zu bestimmen. Das Vorgehen ist nachfolgend skizziert.

3.3.1 Primkörper

Der erste Schritt besteht darin, den zugrundeliegenden Primkörper \mathbb{F}_p zu bestimmen. Wir wählen dazu zufällig, gleichverteilt eine Primzahl p , für die gilt $\lfloor \lg p \rfloor + 1 = i$. Das Theorem von Hasse (3.2.2) garantiert, daß die Größe des Primkörpers \mathbb{F}_p tatsächlich der Problemgröße entspricht. Insbesondere lassen sich alle Berechnungen mit ganzen Zahlen $\{0, \dots, p-1\}$ als Repräsentant des Körpers $(\mathbb{Z}/p\mathbb{Z}, +, \cdot) \cong \mathbb{F}_p$ durchführen.

Eine Möglichkeit zur zufälligen Wahl einer Primzahl besteht darin, zufällig, gleichverteilt ganze Zahlen in der entsprechenden Größenordnung (i Bits) zu wählen, bis ein Primzahltest (z.B. Miller-Rabin Test [Coh93]) die gewählte Zahl als Primzahl bestätigt.

3.3.2 Elliptische Kurve

Dieser Schritt beschränkt sich zunächst auf das zufällige Wählen von $a, b \in \{0, \dots, p-1\}$. Wir werden später sehen, daß nicht jede so erzeugte elliptische Kurve für die Verwendung in kryptographischen Verfahren geeignet ist, was es gegebenenfalls sinnvoll macht, an dieser Stelle einen Test für die Eignung der Kurve anzuschließen.

3.3.3 Ordnung der Gruppe

Die Ordnung der Elliptische-Kurven-Gruppe zu bestimmen, resultiert darin, die Punkte auf der Kurve zu zählen. Ein Algorithmus, der diese Aufgabe in polynomieller Zeit leistet, wurde von Schoof (vgl. z.B. [Men93]) entwickelt. Elkies und Atkins haben eine Verbesserung angegeben, mit welcher der Algorithmus eine in der Praxis vertretbare Laufzeit erzielt. Eine Implementierung ist in [Mül95] zu finden. Dennoch bleibt dieser Schritt sehr aufwendig.

3.3.4 Elementordnung

Wir setzen beim Bestimmen der Elementordnung zunächst voraus, daß wir die Maximalordnung ν_1 von $E(\mathbb{F}_p)$ (vgl. 3.1.1) und ihre Primfaktorzerlegung $\nu_1 = p_1^{a_{11}} \dots p_k^{a_{k1}}$ kennen. Modifiziert für Elliptische-Kurven-Gruppen gilt der Satz (vgl. [Kal88, Lemma 4.5 und Beweis]):

Lemma 3.3.1

Gegeben eine elliptische Kurve $E(\mathbb{F}_p)$. Sei ν_1 ihre Maximalordnung und $\nu_1 = p_1^{a_{11}} \dots p_k^{a_{k1}}$ ihre Primfaktorzerlegung. Seien weiter b_i die kleinsten natürlichen Zahlen, sodaß

$$\left(\frac{\nu_1}{p_i^{a_{i1}}} p_i^{b_i} \right) X = 0, \quad (3.25)$$

für $0 < i \leq k$. Dann ist $p_1^{b_1} \dots p_k^{b_k}$ die Ordnung von X .

Kaliski gibt einen Polynomzeit-Algorithmus an, der die Ordnung eines Elements berechnet (vgl. [Kal88, Abbildung 4.3]).

Es reicht an dieser Stelle aus, wenn wir die Gruppenordnung ν und ihre Faktorisierung kennen. Da $\nu_2 | \nu_1 | \nu$ und $\nu = \nu_1 \nu_2$ gilt, sind die Primfaktoren p_1, \dots, p_k die gleichen wie oben und für die Koeffizienten c_1, \dots, c_k mit $\nu = p_1^{c_1} \dots p_k^{c_k}$ gilt $c_i \geq a_{i1}$ für $i \in \{1, \dots, k\}$, woraus folgt:

$$\left(\frac{\nu_1}{p_i^{a_{i1}}} \right) \mid \left(\frac{\nu}{p_i^{c_i}} \right). \quad (3.26)$$

Wir können folglich in Satz 3.3.1 anstelle von ν_1 auch die Gruppenordnung ν und deren Primfaktorzerlegung verwenden.

3.3.5 Faktorisierung der Gruppenordnung

Wir haben im letzten Abschnitt die Kenntnis der Primfaktorzerlegung der Maximalordnung bzw. der Gruppenordnung vorausgesetzt. Tatsächlich ist aber das Faktorisierungsproblem schwer, d.h. es existiert kein Polynomzeit-Algorithmus, der beliebige ganze Zahlen in ihre Primfaktoren zerlegt.

Kaliski verwendet das Verfahren von Lenstra (vgl. [Kal88, §3.3.1] und [Len87]) und erstellt einen probabilistischen Polynomzeit-Algorithmus 'Partial_Factorization', der zumindestens alle "kleinen" Primfaktoren, also eine partielle Zerlegung in Faktoren, bestimmt. In diesem Zusammenhang bedeutet "klein" kleiner als eine vorgegebene Schranke B , wobei diese Schranke so gewählt wird, daß der Algorithmus polynomielle Laufzeit in der Proböemgröße i hat. Kaliski zeigt, daß es für das Bestimmen von Erzeuger-Tupeln ausreicht, die partielle Faktorisierung zu kennen.

3.3.6 Maximalordnung

Wir haben in Abschnitt 3.3.4 die Kenntnis der Maximalordnung ν_1 von $E(\mathbb{F}_p)$ vorausgesetzt. Kaliski gibt einen probabilistischen Polynomzeit-Algorithmus an, der feststellt, ob ein Element Maximalordnung hat (vgl. [Kal88, Lemma 4.9]). Dieser verwendet dazu die zuvor angegebene partielle Faktorisierung. Im wesentlichen beruht der Algorithmus darauf, daß es viele Elemente maximaler Ordnung gibt. Man kann ihre Anzahl mit der Eulerschen φ -Funktion abschätzen und erhält als untere Schranke:

$$\frac{\varphi(\nu_1)}{\nu_1} \geq \frac{1}{6 \ln(1 + \ln \nu_1)} \geq \frac{1}{6 \ln(1 + \ln \nu)}, \quad (3.27)$$

wobei ν die Ordnung von $E(\mathbb{F}_p)$ ist. Dies garantiert, daß man mit guter Wahrscheinlichkeit wenigstens ein Element maximaler Ordnung findet, wenn man genügend (aber nur polynomiell in i viele) Elemente zufällig wählt.

3.3.7 Erzeuger-Tupel

Wir werden nun ein Erzeuger-Tupel der Elliptische-Kurven-Gruppe bestimmen. Wir haben in den vorangegangenen Abschnitten sichergestellt, daß wir die Ordnung von Kurvenpunkten berechnen und einen Punkt G_1 maximaler Ordnung ν_1 bestimmen können. Wir müssen uns nun damit befassen, wie wir einen Punkt G_2 der Ordnung ν_2 finden, sodaß $E(\mathbb{F}_p) = \langle (G_1, G_2) \rangle$ gilt. Die Schwierigkeit besteht darin, zu einem zufällig gewählten Punkt G_2 zu entscheiden, ob $G_2 \in \langle G_1 \rangle$ gilt, also ob diese bereits von G_1 erzeugt wird. Die offensichtliche Methode besteht darin, den diskreten Logarithmus $\log_{G_1}(G_2)$ zu bestimmen. Genau das ist aber nach Voraussetzung schwer, d.h. nicht in polynomieller Zeit zu bewerkstelligen (siehe 3.4). Kaliski schlägt in [Kal88, §6.2] die Verwendung der Weil-Paarung (vgl. [Wei40]) vor, die wir nun einführen. Die von uns verwendeten Ergebnisse sind auch in [Ham98, §7.3.1] zu finden. Wir benötigen zunächst den Begriff des m -Torsionspunktes.

Definition 3.3.1 (m -Torsionspunkte)

Gegeben ein Körper \mathbb{K} , eine elliptische Kurve $E(\mathbb{K})$ und eine natürliche Zahl m . Ein Punkt $P \in E(\mathbb{K})$ heißt m -Torsionspunkt, falls $m \cdot P = \mathcal{O}$ gilt, d.h. seine Ordnung die Zahl m teilt. Die Untergruppe $E(\mathbb{K})[m]$ ist die Menge aller m -Torsionspunkte in $E(\mathbb{K})$.

Wir definieren nun die Weil-Paarung bezüglich der von uns verwendeten elliptischen Kurven. Sie ist definiert über den m -Torsionspunkten einer Gruppe $E(\overline{\mathbb{F}}_p)$, wobei $\overline{\mathbb{F}}_p$ den algebraischen Abschluß von \mathbb{F}_p bezeichnet (vgl. [Bos99a]). Für uns ist hierbei relevant, daß $\mathbb{F}_p \subset \overline{\mathbb{F}}_p$ gilt.

Definition 3.3.2 (Weil-Paarung)

Die Weilpaarung zum Parameter m ist definiert als

$$e_m : E(\overline{\mathbb{F}}_p) \times E(\overline{\mathbb{F}}_p) \longrightarrow \overline{\mathbb{F}}_p \quad (3.28)$$

Seien $P, P_1, P_2, P_3 \in E(\overline{\mathbb{F}}_p)$. Die Weil-Paarung hat folgende Eigenschaften:

1. (Identität) $e_m(P, P) = 1$;
2. (Alternierung) $e_m(P_1, P_2) = e_m^{-1}(P_2, P_1)$;
3. (Bilinearität) $e_m(P_1 + P_2, P_3) = e_m(P_1, P_3) \cdot e_m(P_2, P_3)$ und $e_m(P_1, P_2 + P_3) = e_m(P_1, P_2) \cdot e_m(P_1, P_3)$;
4. (Nicht-Degeneriertheit) $e_m(P_1, \mathcal{O}) = 1$, wenn $e_m(P_1, P_2) = 1$ für alle $P_2 \in E(\overline{\mathbb{F}}_p)$, dann ist $P_1 = \mathcal{O}$;
5. Es existiert ein Algorithmus, der die Weil-Paarung zweier Punkte P_1, P_2 in Polynomzeit berechnet.

Die Weil-Paarung ist in zweierlei Hinsicht für uns von Interesse. Zum einen bietet sie eine Möglichkeit, daß Diskrete-Logarithmen-Problem von $E(\mathbb{F}_p)$ in einen Körper \mathbb{F}_{p^k} zu transportieren, was es ermöglicht, das später erwähnte Index-Berechnungsverfahren in der Multiplikativen Gruppe $\mathbb{F}_{p^k}^*$ zu verwenden. Ein entsprechender Algorithmus wurde von Menezes, Okamoto und Vanstone [MOV93] angegeben. Wir werden im nächsten Kapitel noch einmal kurz darauf eingehen.

Zum anderen läßt sich mit der Weil-Paarung das obige Problem lösen, nämlich zu entscheiden, ob $G_2 \in \langle G_1 \rangle$. Diese Situation beschreibt folgender Satz (vgl. [Ham98, 7.12]):

Lemma 3.3.2

Sei $E(\mathbb{F}_p)$ eine elliptische Kurve mit Gruppenstruktur (ν_1, ν_2) und P ein Punkt maximaler Ordnung ν_1 . Für alle $P_1, P_2 \in E(\mathbb{F}_p)$ gilt, P_1 und P_2 sind in der selben Nebenklasse von $\langle P \rangle$, wenn $e_{\nu_1}(P, P_1) = e_{\nu_1}(P, P_2)$ ist.

Wir haben nun die Möglichkeit, zu einem Punkt $G_1 \in E(\mathbb{F}_p)$ maximaler Ordnung ν_1 einen Punkt G_2 der Ordnung ν_2 zu finden, sodaß (G_1, G_2) ein Erzeuger-Paar der Kurve ist. Zu gegebener Kurve $E(\mathbb{F}_p)$ der Ordnung ν gehen wir wie folgt vor:

1. Wähle zufällig, gleichverteilt einen Punkt $P \in E(\mathbb{F}_p)$.
2. Berechne die Weil-Paarung von G_1 und P . Falls $e_{\nu_1}(G_1, P) = 1$ wiederhole ab Schritt 1. Nach Satz 3.3.2 folgt aus $1 = e_{\nu_1}(G_1, G_1) = e_{\nu_1}(G_1, P)$, daß $P \in \langle G_1 \rangle$.
3. Berechne die Ordnung von P und Prüfe, ob $\nu_2 \mid \text{order}(P)$. Falls nicht, wiederhole ab 1.
4. Setze zweiten Erzeuger $G_2 = \frac{\text{order}(P)}{\nu_2} P$.

3.4 DL-Problem

Wir beschäftigen uns nun mit dem Problem der Berechnung diskreter Logarithmen in Elliptische-Kurven-Gruppen. Wie in der Einleitung zu diesem Kapitel bereits erwähnt, ist es nicht mehr als eine Annahme, daß dieses DL-Problem schwer ist und bildet den "wunden Punkt" unserer Konstruktion, da wir deren Sicherheit lediglich auf diese Annahme reduzieren können. Die Annahme wird jedoch von der Tatsache gestützt, daß trotz intensivem Forschungsaufwand noch kein effizienter Algorithmus zur Berechnung diskreter Logarithmen gefunden wurde. Wir werden nachfolgend zunächst das DL-Problem definieren und dann die drei gebräuchlichsten Algorithmen und ihre Eigenschaften kurz vorstellen.

Am häufigsten findet man in der Literatur das DL-Problem für multiplikative Gruppen (meist \mathbb{F}_p^*) entsprechend folgender Definition:

Definition 3.4.1 (DL-Problem in zyklischen Gruppen)

Sei (G, \cdot) eine zyklische Gruppe und $g \in G$ ein Erzeuger. Das DL-Problem besteht darin, für ein beliebiges Element $a \in G$ die kleinste nichtnegative ganze Zahl λ zu finden, sodaß

$$a = g^\lambda. \quad (3.29)$$

gilt.

Die Definition überträgt sich entsprechend auf den nichtzyklischen Fall von Elliptische-Kurven-Gruppen. Man betrachtet hierzu den diskreten Logarithmus bezüglich eines Punktes G_1 maximaler Ordnung und der von ihm erzeugten zyklischen Gruppe $\langle G_1 \rangle$.

Definition 3.4.2

Sei $E(\mathbb{F}_p)$ eine elliptische Kurve mit Gruppenstruktur (ν_1, ν_2) und Erzeuger-Paar (G_1, G_2) . Das DL-Problem besteht darin, zu einem beliebigen Punkt $P \in \langle G_1 \rangle$ die kleinste nichtnegative ganze Zahl λ zu berechnen, sodaß

$$P = \lambda \cdot G_1 \quad (3.30)$$

gilt.

Wir können nun die Annahme formulieren:

Annahme 3.4.1

Das DL-Problem ist schwer in der Folge der Elliptische-Kurven-Gruppen EK. Das bedeutet, für alle probabilistischen Polynomzeit-Algorithmen A und alle Konstanten c gilt für genügend große i

$$\Pr [A(P) = \log_{G_1}(P)] < \frac{1}{i^c}, \quad (3.31)$$

wobei $\langle E_{a,b}(\mathbb{F}_p), (G_1, G_2), P \rangle$ eine zufällig, gleichverteilt gewählte Instanz bezüglich i darstellt.

3.4.1 Verfahren zur Berechnung diskreter Logarithmen

Wir werden nun kurz auf Verfahren zur Berechnung diskreter Logarithmen eingehen. Eine umfassendere Übersicht ist zum Beispiel in [Odl00] zu finden. Sei im folgenden G eine Gruppe der Ordnung ν .

Die bekanntesten Algorithmen sind der deterministische "Baby-Step-Giant-Step"-Algorithmus von Shanks und die probabilistische ρ -Methode von Pollard. Diese beiden sind sogenannte generische Algorithmen. Sie kommen mit relativ wenig Wissen über die Gruppe aus, in dem Sinne, daß sie in allen Gruppen (auch in den Elliptische-Kurve-Gruppen) funktionieren, die in geeigneter Weise in Rechnersystemen dargestellt werden können. Die Algorithmen haben beide eine Laufzeit von $O(\sqrt{\nu})$. Der "Baby-Step-Giant-Step"-Algorithmus hat einen Speicherplatzbedarf von $O(\sqrt{\nu})$, während die ρ -Methode mit linearem Speicherplatz auskommt. Zudem ist es möglich, mit einem Verfahren von Pohlig und Hellman das DL-Problem von der Gruppe der Ordnung ν in Gruppen mit Primzahlordnung λ zu transportieren, wobei $\lambda|\nu$ gilt. Die Laufzeit der Algorithmen (und der Speicherplatzbedarf beim "Baby-Step-Giant-Step"-Verfahren) reduziert sich dann auf $O(\sqrt{\lambda})$, wobei λ der größte Primteiler von ν ist. Diese beiden Algorithmen sind als generische Verfahren in gewisser Weise optimal. V. Shoup zeigt in [Sho97], daß $\Omega(\sqrt{\lambda})$ eine untere Schranke der Laufzeit aller generischer Algorithmen ist, wobei λ der größte Primteiler der Gruppenordnung ν ist.

Als nicht generisches Verfahren ist vor allem der Index-Berechnungs-Algorithmus von Bedeutung, der zuerst für die multiplikativen Gruppen $\mathbb{F}_q = \mathbb{F}_{p^k}$, p eine Primzahl, eingeführt wurde. Er erreicht hier sogar subexponentielle Laufzeit. Detailliertere Informationen finden sich unter anderem in [Odl00]. Dieses Verfahren bietet in den eben genannten Gruppen eine wesentliche Verbesserung gegenüber den oben genannten generischen, und es stellt sich die Frage, inwieweit sich dieses Verfahren auch für Elliptische-Kurven-Gruppen einsetzen läßt. J.H. Silvermann und J. Suzuki untersuchen dies in [SS98] und kommen zu dem Schluß, daß das Index-Berechnungsverfahren in Elliptische-Kurven-Gruppen zu keinem effizienten Algorithmus führt.

Abschließend können wir festhalten, daß die besten Algorithmen zur Berechnung diskreter Logarithmen in Elliptische-Kurven-Gruppen exponentielle Laufzeit haben.

3.4.2 Kryptographisch starke elliptische Kurven

Wenn wir später den Pseudozufallsbit-Generator realisieren, werden wir den Rahmen der asymptotischen Analysen verlassen und uns auf eine konkrete Problemgröße (oder zumindestens auf eine Unterschranke) festlegen müssen. Zu einer Schranke für i gelangt man, indem man die Rechnerleistung abschätzt, die einem potentiellen Angreifer zur Verfügung steht, und die Laufzeitschranken der in 3.4.1 eingeführten Verfahren berücksichtigt.

Eine weitere Gefahrenquelle ist darin begründet, daß für bestimmte Typen elliptischer Kurven das DL-Problem nicht schwer ist. Es ist jedoch ohne weitere Probleme machbar, diese Kurven zu vermeiden.

Insgesamt führt dies zu einer Liste von Bedingungen, denen eine elliptische Kurve genügen muß, um für kryptographische Verfahren geeignet zu sein. Wir orientieren uns an den Vorgaben, die das Bundesamt für Sicherheit in der Informationstechnik (BSI) empfiehlt ([Bud]). Sei $E(\mathbb{F}_p)$ eine elliptische Kurve der Ordnung $\nu = k \cdot l$, wobei l der

größte Primteiler von ν ist. Wir nennen k den Kofaktor. Dann muß gelten:

1. Die Primzahl l hat eine Bitlänge von mindestens 160 Bits.
2. $l \neq p$
3. $r_0 \stackrel{\text{def}}{=} \{r | l \text{ teilt } p^r - 1\}$ ist größer als 10^4
4. Die Klassenzahl der Hauptordnung, die zum Endomorphismenring von E gehört, ist mindestens 200.

In der Bedingung 1 wird der Tatsache Rechnung getragen, daß die Laufzeitschranken der oben genannten generischen Verfahren zur Berechnung diskreter Logarithmen Funktionen in der Bitlänge des größten Primteilers l sind. Insbesondere geht hier die Annahme ein, daß 2^{80} Gruppenoperationen von keinem Angreifer in heutigen Rechnersystemen effizient berechnet werden können.

Die Bedingung 2 verhindert, daß $E(\mathbb{F}_p)$ eine Kurve der Spur 1 (gemäß Definition 3.2.2) ist. Man kann bei solchen Kurven das DL-Problem effizient auf das in \mathbb{F}_p^+ reduzieren (vgl. [Ham98, §7.4.3]).

Die Bedingung 3 verhindert die sogenannte MOV-Attacke (vgl. [MOV93]). Dabei wird das DL-Problem vermöge der Weil-Paarung in eine multiplikative Gruppe $\mathbb{F}_{p^{r_0}}$ transportiert. Ist r_0 klein genug, führt dies mit dem Index-Berechnungs-Verfahren zu einem Angriff mit subexponentieller Laufzeit. Der Parameter r_0 entspricht der Ordnung von $p \bmod l$ in \mathbb{F}_l^* . In der Literatur findet man des öfteren zwei Fälle besonders erwähnt, nämlich den der elliptischen Kurven mit Spur 0 (supersinguläre Kurven) und Spur 1. Ausführlicher wird dieses Thema zum Beispiel in [Ham98] behandelt.

Zur letzten Bedingung ist zu sagen, daß hierzu noch kein konkreter Angriff bekannt ist, es wird lediglich vermutet, daß eine kleine Klassenzahl zu einer effizienten Anwendbarkeit des Index-Berechnungs-Algorithmus führen könnte. Wenn man eine Kurve zufällig wählt, ist diese Bedingung praktisch immer erfüllt. Wir werden diese Bedingung nicht weiter überprüfen, müssen dann aber darauf gefaßt sein, daß unser Pseudozufallsbit-Generator gegebenenfalls unsicher wird, wenn sich diese Kurven als schwach herausstellen (siehe dazu auch [Bai00]).

Wir haben uns in 3.3 darauf beschränkt, eine elliptische Kurve $E_{a,b}(\mathbb{F}_p)$ zu einem Körper \mathbb{F}_p zufällig zu bestimmen, indem wir die Parameter a, b zufällig wählen. Wenn wir kryptographisch starke Kurven erzeugen wollen, müssen wir bei dieser Vorgehensweise im Nachhinein die oben angegebenen Kriterien überprüfen und die erzeugte Kurve gegebenenfalls wieder verwerfen.

Für die spätere Realisierung wird es zudem sinnvoll sein, den Kofaktor k möglichst klein zu halten, weil der Aufwand für die Berechnung der Gruppenoperationen bzgl. $E_{a,b}(\mathbb{F}_p)$ von der Größe der Gruppe und damit auch von k abhängt. Auch dieses Kriterium können wir nur im Anschluß an die Kurvenerzeugung testen und führt gegebenenfalls dazu, daß wir eine neue Kurve wählen müssen.

Insgesamt ist diese Vorgehensweise nicht sehr effizient, da wir für das Prüfen der Kriterien stets die Ordnung der Kurve bestimmen müssen, was sehr kostspielig ist. Wir werden deshalb in der Implementierung ein anderes Verfahren zur Erzeugung elliptischer Kurven verwenden, daß komplexe Multiplikation verwendet. Es erlaubt uns, die Größenordnung des Primteiler l und des Kofaktors k festzulegen und dazu eine passende Kurve zu bestimmen. Der wesentliche Nachteil dieses Verfahrens ist, daß nicht alle

möglichen kryptographisch starken Kurven zu den Parametern l und k auf diese Weise erzeugen werden können, wir also nicht zufällig, gleichverteilt die Kurven wählen. Wir werden in Abschnitt 6.1.5 noch einmal darauf eingehen.

Kapitel 4

Hard-Core-Prädikat

Im vorangegangenen Kapitel haben wir das DL-Problem in Elliptische-Kurven-Gruppen kennengelernt. Wir haben gesehen, daß wir mit der diskreten Exponentiation eine Oneway-Funktion bauen können unter der Annahme, daß die Berechnung diskreter Logarithmen in dieser Gruppe schwer ist. In diesem Kapitel werden wir die i -ten höchstwertigen Bits als Hard-Core-Prädikate bezüglich diskreter Exponentiation angeben, die wir für den kryptographisch sicheren Pseudozufallsbit-Generator verwenden werden.

Diese Bits sind aus Kenntnis des Logarithmus (in polynomieller Zeit in der Gruppengröße) leicht zu berechnen.

Der Schwerpunkt dieses Kapitels ist, die Hard-Core-Eigenschaft nachzuweisen und damit die Sicherheit unserer Konstruktion zu garantieren. Wir werden hierzu ein Orakel annehmen, das die höchstwertigen Bits mit nicht vernachlässigbarer Wahrscheinlichkeit vorhersagt. Wir werden darauf basierend einen Algorithmus angeben, der unter Verwendung dieses Orakels diskrete Logarithmen in polynomieller Zeit berechnet. Wir werden zeigen, daß wir an Stelle dieses Orakels einen probabilistischen Polynomzeit-Algorithmus einsetzen können, der ein Unterscheider der höchstwertigen Bits darstellt. Wenn es einen solchen Unterscheider gibt, also die Ausgabe unseres Generators mit signifikantem Vorteil geraten werden kann, ist es möglich, mit ebenfalls guter Wahrscheinlichkeit diskrete Logarithmen in Polynomzeit zu berechnen. Dies ist ein Widerspruch zur Annahme, das DL-Problem sei schwer. Woraus wir schließen, daß es keinen solchen Unterscheider geben kann.

Die folgenden Definitionen und Sätze sind im wesentlichen der Dissertation von B.S.Kaliski [Kal88] entnommen. Entsprechend sind dort auch die Beweise zu finden - wir werden die genauen Referenzen jedoch nach den jeweiligen Sätzen angeben. Vorab sei angemerkt, daß Kaliski die Hauptaussagen allgemeiner für reguläre Klassen von Gruppen (siehe Definition 3.1.7) angibt. Hier zählt sich die Vorarbeit aus Abschnitt 3.2.4 aus.

4.1 Hauptsatz

Die Kernaussage dieses Kapitels beinhaltet der folgende, der Dissertation von Kaliski ([Kal88]) entnommene, Satz:

Satz 4.1.1

Sei S eine Folge von Gruppen, deren Rang durch eine Konstante r beschränkt ist. Wenn das DL-Problem schwer ist, dann ist das höchstwertige Bit des Logarithmus für die Klasse S nicht unterscheidbar.

Dieser Satz beinhaltet die Reduktion des DLP auf das Nähern i -ter höchstwertiger Bits. Wir gliedern den Beweis in zwei Schritte.

Der erste Schritt liefert einen Algorithmus, der unter Verwendung eines Orakels diskrete Logarithmen in einer bestimmten Gruppe berechnet. Die Definitionen, Sätze und Beweise entsprechen denen in [Kal88, §5]. Wir beschränken uns deshalb darauf, die wesentlichen Konzepte lediglich zu skizzieren.

Der zweite Schritt erweitert den ersten auf die Folge EK und beweist Satz 4.1.1 in einer eingeschränkten Version für die Folge EK unter Verwendung des uniformen Angreifermodells.

4.2 Erster Beweisschritt

4.2.1 höchstwertige Bits

In diesem Abschnitt werden wir zunächst die Definition der Begriffe "höchstwertiges Bit" des diskreten Logarithmus eines Gruppenelements und "Kreuzkorrelation" einführen. Zudem werden wir hier auf die in den Beweisen der folgenden Abschnitten verwendete Orakel eingehen. Wir gehen dabei genauso wie Kaliski vor und betrachten zunächst zyklische Gruppen und das (erste) höchstwertige Bit. Danach erweitern wir diese Vorgehensweise auf mehrere Bits und nichtzyklische Gruppen.

Definition 4.2.1 (i -tes höchstwertiges Bit)

Bezüglich einer ganzen Zahl c und eines Moduls ν ist das höchstwertige Bit $b_1(c, \nu)$ definiert als

$$b_1(c, \nu) \stackrel{\text{def}}{=} \begin{cases} 1, & \text{wenn } (c \bmod \nu) \geq \frac{\nu}{2} \\ -1, & \text{wenn } (c \bmod \nu) < \frac{\nu}{2} \end{cases}. \quad (4.1)$$

Für $i > 1$ definieren wir das i -te höchstwertige Bit $b_i(c, \nu)$ rekursiv durch die Vorschrift

$$b_i(x, \nu) \stackrel{\text{def}}{=} b_{i-1}(2x, \nu) = b_i(x, \nu) = b_1(2^{(i-1)}x, \nu) \quad (4.2)$$

Kaliski verwendet hier bewußt die Werte $+1$ und -1 anstelle der sonst gebräuchlichen Binärdarstellung mit $1, 0$. Letztere Darstellung ist handlicher in den Abschätzungen, für die er Methoden aus der Signalverarbeitung verwendet, wie wir in Abschnitt 4.2.3 sehen werden.

Wir führen an dieser Stelle die eingangs erwähnten Orakel O_i ein, die das i -te höchstwertige Bit mit einem gewissen Vorteil vorhersagen können. Wir betrachten nur solche Orakel, die ϵ -Approximatoren des i -ten höchstwertigen Bits darstellen. Wir nehmen zunächst an, daß die Orakel pro Aufruf für beliebige Eingabelängen lediglich konstante Zeit benötigen. Wir werden später die in den folgenden Algorithmen verwendeten Orakel durch Polynomzeit-Algorithmen ersetzen. Um dann die Änderung im Laufzeitverhalten der Algorithmen abschätzen zu können, werden wir getrennt von den Rechenschritten auch Orakelaufrufe zählen. Für die Orakel gilt folgende Definition:

Definition 4.2.2

Gegeben sei eine endliche, zyklische Abelsche Gruppe G , ein Generator $g \in G$ der Gruppe und ein beliebiges Element $x \in G$. Ein Orakel O_i ist ein ϵ -Approximator für das i -te höchstwertige Bit des diskreten Logarithmus $\log_g x$, wenn gilt

$$\Pr [O_i(x) = b_i(\log_g x, \nu)] \geq \frac{1}{2} + \epsilon \quad (4.3)$$

Wir sagen, das Orakel hat einen ϵ -Vorteil.

4.2.2 Übersicht

Mit der obigen Definition der höchstwertigen Bits können wir nun den ersten Schritt der Reduktion skizzieren und mit Hilfe obiger Orakel den diskreten Logarithmus eines Elements $x \in G$ berechnen, wobei G eine zyklische Gruppe mit ν Elementen und g ein Erzeuger sei (siehe auch [Kal88, Abschnitt 5.1.2]).

Die Idee ist die folgende: Wir raten als erste Näherung den Logarithmus des Elements $2^{\lceil \lg \nu \rceil} x$. Wir berechnen dann sukzessive bessere Näherungen für die Logarithmen der Elemente $2^{\lceil \lg \nu - 1 \rceil} x, \dots, 2x, x$. Das Verfahren führt zum Erfolg, wenn der erste geratene Logarithmus in einem bestimmten Intervall um den tatsächlichen Logarithmus liegt. Die Größe dieses Intervalls hängt von der Güte ϵ des Orakels ab. Da die Gruppe endlich und zyklisch ist, sind die Logarithmen alle nichtnegativ und kleiner ν . Wir betrachten deshalb die folgenden Ungleichungen modulo ν . Das Vorgehen beinhaltet drei Schritte:

1. Wir raten den ersten Logarithmus $c_{\lceil \lg \nu \rceil}$ und nehmen an, daß gilt

$$c_{\lceil \lg \nu \rceil} - \frac{\nu \epsilon}{2} \leq \log_g(2^{\lceil \lg \nu \rceil} x) \leq c_{\lceil \lg \nu \rceil} + \frac{\nu \epsilon}{2} \quad (4.4)$$

2. Für alle $i \in \{\lceil \lg \nu \rceil - 1, \dots, 0\}$ benutzen wir das Orakel, um zu entscheiden, welcher der beiden Werte $\frac{c_{i+1}}{2}$ oder $\frac{c_{i+1}}{2} + \frac{\nu}{2}$ sich für c_i besser eignet bezüglich der Ungleichungen:

$$\frac{c_i}{2} - \frac{\nu \epsilon}{2^{\lceil \lg \nu \rceil - i + 1}} \leq \log_g(2^i x) \leq \frac{c_i}{2} + \frac{\nu \epsilon}{2^{\lceil \lg \nu \rceil - i + 1}} \quad (4.5)$$

3. Wir prüfen, ob $[c_0] \cdot g = x$ gilt. Falls nicht, wiederholen wir ab Schritt 1.

Falls wir den ersten Logarithmus $c_{\lceil \lg \nu \rceil}$ richtig geraten haben und somit die Annahme aus Schritt 1 zutrifft, gilt für c_0 :

$$c_0 - \frac{1}{2} < c_0 - \frac{\nu \epsilon}{2^{\lceil \lg \nu \rceil}} \leq \log_g(x) \leq c_0 + \frac{\nu \epsilon}{2^{\lceil \lg \nu \rceil}} < c_0 + \frac{1}{2} \quad (4.6)$$

Folglich ist c_0 der gesuchte Logarithmus.

Am wichtigsten ist Schritt 2. Auf diesen werden wir nun näher eingehen.

4.2.3 Kreuzkorrelation

Um die Entscheidung in Schritt 2 des eben skizzierten Algorithmus zu realisieren nutzt Kaliski Methoden der Signalverarbeitung. Wir betten zu diesem Zweck die (natürlich diskreten) Bitfolgen, die durch Auswertung des Prädikats bzw. der Orakel erzeugt werden, in kontinuierliche Signale ein, stellen sie also als Funktionen der Zeit dar. Die Modellvorstellung ist folgende: Die Prädikate bzw. die Orakel erzeugen ein kontinuierliches Rechtecksignal¹, das wir an diskreten Abtaststellen (Samples) auswerten. Der Wert des Signals an einer Abtaststelle entspricht dabei einem ausgegebenen Bit.

In der Definition 4.2.5 führen wir die sogenannten Kreuzkorrelation ein, die es uns erlaubt, zwei Signale über ein Zeitintervall hinweg zu vergleichen, wobei wir das zweite Signal gegenüber dem ersten um eine bestimmte Zeitdifferenz (Lag) verschieben können.

Doch zunächst folgt die Einbettung der höchstwertigen Bits und des Orakels in kontinuierliche Signale (vgl. [Kal88, Abschnitt 5.1.3]).

Definition 4.2.3 (Das höchstwertige Bit als Signal)

Zu gegebenem Modul ν definieren wir das Signal

$$\hat{s} : \mathbb{R} \longrightarrow \{-1, +1\}, \text{ mit} \quad (4.7)$$

$$\hat{s}(t) \stackrel{\text{def}}{=} b_1(t, \nu) \quad (4.8)$$

Es gelten folgende Eigenschaften:

1. $\hat{s}(t) = \hat{s}(t + k\nu)$, für alle $k \in \mathbb{Z}$
2. $\hat{s}(t) = -\hat{s}(t + \frac{\nu}{2})$

Ähnlich definieren wir für die Orakel:

Definition 4.2.4 (Die Antworten des Orakels als Signal)

Gegeben eine endliche, zyklische Abelsche Gruppe G , ein Generator $g \in G$ der Gruppe und ein beliebiges Element $x \in G$. Für dieses feste Element x definieren wir das Signal

$$s_x(t) \stackrel{\text{def}}{=} O_1(x + tg) \quad (4.9)$$

Wir können nun die Kreuzkorrelation bzgl. der Signale $\hat{s}(t)$ und $s_x(t)$ definieren. Da die betrachteten Signale periodisch (mit Periode ν) und die ausgewerteten Abtaststellen diskret sind, kann dies wie folgt als Summe über alle Abtaststellen geschehen:

Definition 4.2.5 (Kreuzkorrelation)

Die Signale $\hat{s}(t)$, $s_x(t)$ seien wie oben definiert. Die Kreuzkorrelation ist eine Abbildung für ein Lag l

$$R_{\hat{s}s_x} : [0, \nu) \longrightarrow \mathbb{R} \quad (4.10)$$

¹Die Pegel dieses Signals liegen bei ± 1 entsprechend Definition 4.2.1 und die Signalflanken sind unendlich steil.

mit

$$R_{s\hat{s}}(l) \stackrel{\text{def}}{=} \frac{1}{\nu} \sum_{t=0}^{\nu-1} s_x(t) \hat{s}(t+l) \quad (4.11)$$

$$\frac{1}{\nu} \sum_{t=0}^{\nu-1} O_1(x+tg) b_1(t+1, \nu) \quad (4.12)$$

Die Kreuzkorrelation ist also der Erwartungswert des Produkts der beiden Signale $s_x(t) \hat{s}(t+l)$. Aufgrund der Eigenschaften der Signale ergeben sich folgende Beziehungen für die Kreuzkorrelation für beliebige Lags $l \in [0, n)$ und $k \in \mathbb{Z}$:

Lemma 4.2.1

1. $|R_{s\hat{s}}(l)| \leq 1$
2. $R_{s\hat{s}}(l) = R_{s\hat{s}}(l+k\nu)$
3. $R_{s\hat{s}}(l) = -R_{s\hat{s}}(l+\frac{\nu}{2})$

Weitere Eigenschaften der Kreuzkorrelation liefern die folgenden drei Hilfssätze:

Lemma 4.2.2

Gegeben eine endliche, zyklische Abelsche Gruppe G der Ordnung ν . Für alle Generatoren g und alle $x \in G$ erfüllt die Kreuzkorrelation für die Signale $s_x(t) = O_1(x+tg)$ und $\hat{s}(t) = b_1(t, \nu)$ die folgenden Eigenschaften:

$$R_{s\hat{s}}(\log_g x) \geq 2\epsilon \quad (4.13)$$

$$R_{s\hat{s}}(\log_g x) + \frac{\nu}{2} \leq -2\epsilon \quad (4.14)$$

Lemma 4.2.3

Gegeben eine endliche, zyklische Abelsche Gruppe G der Ordnung ν . Für alle Generatoren g , alle $x \in G$ und beliebige Lags l_1, l_2 erfüllt die Kreuzkorrelation, definiert auf den Signale $s_x(t) = O_1(x+tg)$ und $\hat{s}(t) = b_1(t, \nu)$, folgende Ungleichung:

$$|R_{s\hat{s}}(l_2) - R_{s\hat{s}}(l_1)| \leq \frac{4}{n} (|l_2 - l_1| + 1) \quad (4.15)$$

Lemma 4.2.4

Gegeben eine endliche, zyklische Abelsche Gruppe G der Ordnung ν . Für alle Generatoren g , alle $x \in G$ und beliebige Lags l_1, l_2 hat die Kreuzkorrelation, definiert auf den Signale $s_x(t) = O_1(x+tg)$ und $\hat{s}(t) = b_1(t, \nu)$, für ein $\epsilon \geq \frac{8}{\nu}$ die folgende Eigenschaft:

Wenn

$$l - \frac{\nu\epsilon}{4} \leq \log_g x \leq l + \frac{\nu\epsilon}{4}, \quad (4.16)$$

dann folgt

$$R_{s\hat{s}}(l) \geq \frac{\epsilon}{2} \quad \text{und} \quad R_{s\hat{s}}\left(l + \frac{\nu}{2}\right) \leq -\frac{\epsilon}{2} \quad (4.17)$$

Die Sätze entsprechen den Lemmata 5.1, 5.2 und 5.3 in [Kal88, Seite 101 ff.].

Lemma 4.2.3 stellt die für die Abschätzung wichtige Eigenschaft zur Verfügung, daß die Kreuzkorrelation sich nur langsam mit ihrem Lag l ändert. D.h. die Differenz zweier Korrelationen bzgl. der gleichen Signale läßt sich durch die Differenz der beiden Lags nach oben abschätzen.

4.2.4 Schätzen der Kreuzkorrelation

Da es zu aufwendig wäre, die Kreuzkorrelation direkt auszurechnen - sie ist immerhin eine Summe über alle ν Gruppenelemente - nutzen wir aus, daß wir mit wesentlich weniger Aufwand einen guten Schätzwert angeben können. Hierzu wählen wir $\eta < \nu$ viele Werte t_i zufällig, gleichverteilt und verwenden folgenden Schätzwert:

Definition 4.2.6 (Schätzer für die Kreuzkorrelation)

$$R_{ss}^\eta(l) \stackrel{\text{def}}{=} \frac{1}{\eta} \sum_{i=1}^{\eta} s_x(t_i) \hat{s}(t_i + l) \quad (4.18)$$

Lemma 4.2.5 (Güte des Schätzers)

Der Schätzer in Definition 4.2.6 hat den Erwartungswert $\mu = R_{ss}(l)$

Der Beweis ist in [Kal88, Beweis zu Lemma 5.4, Seite 105] zu finden.

Der folgende Satz rechtfertigt die Näherung der Kreuzkorrelation und führt zu dem darauf folgende Algorithmus.

Satz 4.2.1

Gegeben eine endliche, zyklische Abelsche Gruppe G . Für alle Generatoren g , alle Elemente $x \in G$ und beliebige Lags l kann man die Wahrscheinlichkeit, daß der Schätzer $R_{ss}^\eta(l)$ um mehr als $\frac{\epsilon}{2}$ vom Erwartungswert abweicht, nach oben durch den Ausdruck $\frac{4}{\eta^2 \epsilon^2}$ abgeschätzt werden. Es gilt nämlich nach der Ungleichung von Tschebyscheff.

$$\Pr \left[|R_{ss}^\eta(l) - R_{ss}(l)| \geq \frac{\epsilon}{2} \right] \leq \frac{\text{Var}(R_{ss}^\eta(l))}{\left(\frac{\epsilon}{2}\right)^2} \quad (4.19)$$

$$\leq \frac{4}{\eta^2 \epsilon^2} \quad (4.20)$$

Wenn wir den Parameter η beim Schätzen der Kreuzkorrelation groß genug wählen, können wir die Wahrscheinlichkeit, daß der Fehler des Schätzwerts über der Schranke $\frac{2}{\epsilon}$ liegt, beliebig verkleinern. Wollen wir für diese Fehlerwahrscheinlichkeit eine obere Schranke δ garantieren, müssen wir

$$\eta = \left\lceil \frac{2}{\epsilon \sqrt{\delta}} \right\rceil \quad (4.21)$$

wählen.

Es folgt der Algorithmus, der die Kreuzkorrelation bezüglich eines Gruppenelements x und eines Lags l im Sinne eines ϵ -Approximators nähert. Der folgende Algorithmus nimmt zu einer Gruppe G einen Generator g , ein Element x , die Ordnung ν der Gruppe und ein Lag l als Eingabe und gibt einen Schätzwert für die Kreuzkorrelation gemäß Satz 4.2.1 aus.

ALGORITHMUS KORREL(g, x, η, l, δ)

- 1 // Berechne die Anzahl der Rateversuche
- 2 $m \leftarrow \left\lceil \frac{2}{\epsilon\sqrt{\delta}} \right\rceil$
- 3 // Berechnung des Schätzwerts
- 4 $s \leftarrow 0$
- 5 **for** $i \leftarrow 1$ **to** m
- 6 **do** $t \leftarrow$ zufällig, gleichverteilte ganze Zahl in $\{1, \dots, \nu\}$
- 7 $s \leftarrow s + O_1(x + rg)b_1(t + 1, \nu)$
- 8 **return** $\frac{s}{m}$

Abbildung 4.1: Schätzen der Korrelation

Wie in der Übersicht (4.2.2) angegeben, sind wir an einer Möglichkeit interessiert, den dortigen Schritt 2 zu realisieren. Dies ermöglicht uns die folgende Abschätzung:

Lemma 4.2.6

Gegeben eine endliche, zyklische Abelsche Gruppe G der Ordnung ν Sei eine Schranke $\delta \in \mathbb{R}$ gegeben mit $0 < \delta \leq 1$ und sei $\eta = \left\lceil \frac{2}{\epsilon\sqrt{\delta}} \right\rceil$. Für alle Erzeuger g , Elemente x und Zahlen c_{i+1} gilt:

1. Falls $\frac{c_{i+1}}{2} - \frac{\nu\epsilon}{4} \leq \log_g(x) \leq \frac{c_{i+1}}{2} + \frac{\nu\epsilon}{4}$, dann ist

$$\Pr \left[R_{ss}^\eta \left(\frac{c_{i+1}}{2} \right) > R_{ss}^\eta \left(\frac{c_{i+1}}{2} + \frac{\nu}{2} \right) \right] \geq (1 - \delta)^2 \quad (4.22)$$

2. Falls $\frac{c_{i+1}}{2} + \frac{\nu}{2} - \frac{\nu\epsilon}{4} \leq \log_g(x) \leq \frac{c_{i+1}}{2} + \frac{\nu}{2} + \frac{\nu\epsilon}{4}$, dann ist

$$\Pr \left[R_{ss}^\eta \left(\frac{c_{i+1}}{2} + \frac{\nu}{2} \right) > R_{ss}^\eta \left(\frac{c_{i+1}}{2} \right) \right] \geq (1 - \delta)^2 \quad (4.23)$$

Wir werten also in jedem Schritt i die Differenz der Korrelationen $\Delta = R_{ss}^\eta \left(\frac{c_{i+1}}{2} + \frac{\nu}{2} \right) - R_{ss}^\eta \left(\frac{c_{i+1}}{2} \right)$ aus und entscheiden je nach Vorzeichen von Δ ob wir $c_i = \frac{c_{i+1}}{2}$ oder $c_i = \frac{c_{i+1}}{2} + \frac{\nu}{2}$ wählen. Bleibt die Frage nach dem ersten zu ratenden Logarithmus $c_{\lceil \lg \nu \rceil}$, der ja die Ungleichungen

$$c_{\lceil \lg n \rceil} - \frac{n\epsilon}{2} \leq \log_g(2^{\lceil \lg n \rceil} x) \leq c_{\lceil \lg n \rceil} + \frac{n\epsilon}{2} \quad (4.24)$$

erfüllen soll. Hierzu wenden wir das Verfahren kurzerhand auf eine Folge von möglichen Werten $c_{\lceil \lg \nu \rceil}$ an, wobei diese zueinander einen Abstand von $\nu\epsilon$ haben. Einer dieser Werte muß dann der Ungleichung 4.24 genügen. Dieses Vorgehen spiegelt sich in folgendem Satz und dem darauffolgenden Algorithmus wider.

Satz 4.2.2

Sei G eine zyklische Abelsche Gruppe mit ν Elementen und g ein Erzeuger von G . Sei O_1 ein Orakel, welches für ein Element $x \in G$ das höchstwertige Bit $b_1(\log_g(x), \nu)$ ϵ -approximiert, dann gilt: Es existiert ein probabilistischer Polynomzeit-Algorithmus, der unter Benutzung des Orakels O_1 für alle $x \in G$ den Logarithmus $\log_g(x)$ korrekt berechnet. Er benötigt dafür erwartete $O\left(\lg^{\frac{5}{2}}\left(\frac{\nu}{\epsilon}\right)\right)$ viele Gruppenoperationen und erwartete $O\left(\lg^{\frac{3}{2}}\left(\frac{\nu}{\epsilon}\right)\right)$ viele Orakel-Aufrufe.

Der Satz entspricht [Kal88, Theorem 5.6], der Beweis ist ebenfalls dort angegeben.

```

ALGORITHMUS LOGARITHMUS( $g, x, \nu$ )
1 // Berechne Schrittweite, Anzahl der Iterationen und Fehlerschranke
2  $k \leftarrow \nu \epsilon$ 
3  $l \leftarrow \lceil \lg n \rceil$ 
4  $\delta \leftarrow 0.25l$ 
5 // Wiederhole, bis Logarithmus gefunden
6 while true
7 do // Rate ersten Logarithmus
8   for  $c \leftarrow 0$  to  $\nu - 1$  next  $step$ 
9   do for  $i \leftarrow l - 1$  to  $0$  next  $-1$ 
10    do  $X_i \leftarrow 2^i x$ 
11    if  $Korrel(G, x_i, n, \frac{c}{2}, \delta) > Korrel(G, x_i, n, \frac{c}{2} + \frac{\nu}{2}, \delta)$ 
12    then  $c \leftarrow c/2$ 
13    else  $c \leftarrow c/2 + \nu/2$ 
14    if  $[c]g = x$ 
15    then return  $[c]$ 

```

Abbildung 4.2: Berechnung von Logarithmen

4.2.5 Erweiterungen

Kaliski gibt in [Kal88] drei Erweiterungen des obigen Konzepts an. Die erste Anpassung ermöglicht das Verwenden i -ter höchstwertiger Bits. Die zweite Erweiterung ändert das Verfahren dahingehend ab, daß es die Kenntnis einiger höchstwertiger Bits mit einbezieht, woraus wir die simultane Sicherheit mehrerer Bits ableiten können. Die dritte und letzte Erweiterung beinhaltet die Erweiterung auf nicht-zyklische Gruppen endlichen Rangs. Diese Erweiterungen entsprechen [Kal88, Abschnitte 5.2.1, 5.2.2 und 5.2.30] und sind dort nachzulesen. Lediglich die Idee soll hier verdeutlicht werden.

Wichtig für die Erweiterungen ist die folgende Beobachtung bei der Reduktion des diskreten Logarithmus Problem auf das Vorhersagen der höchstwertigen Bits: Der Algorithmus 4.2.4 zur Berechnung der Logarithmen hängt nicht direkt von den verwendeten Orakeln, sondern vielmehr von den Eigenschaften der Kreuzkorrelation gemäß der Lemmata 4.2.2 und 4.2.3 ab. Die Vorgehensweise ist demnach, das Orakelsignal der jeweiligen Erweiterung anzupassen und daraufhin die Definition der Kreuzkorrelation so zu modifizieren, daß sie die oben genannten Eigenschaften erfüllt, selbst dann, wenn sie dadurch keine Kreuzkorrelation im eigentlichen Sinne mehr ist.

Die Erweiterungen führen zu den folgendem Satz:

Satz 4.2.3

Sei G eine Abelsche Gruppe mit ν Elementen und Gruppenstruktur (ν_1, \dots, ν_r) . Sei (g_1, \dots, g_r) ein Erzeuger-Tupel und b, d positive ganze Zahlen. Gegeben ein Orakel O_b , welches für ein Element $x \in \langle g_1 \rangle$ das b -te höchstwertige $b_b(a_1, \nu_1)$ mit $(a_1, \dots, a_r) = \log_{(g_1, \dots, g_r)}(x)$ unter Kenntnis der korrekten Bits $b_{b+1}(a_1, \nu_1), \dots, b_{b+d}(a_1, \nu_1)$ ϵ -approximiert. Es gilt: Falls $b \leq \lg \nu - \lg(\frac{1}{\epsilon}) - 1$

ist, existiert ein probabilistischer Polynomzeit-Algorithmus, der unter Benutzung von O_b und Kenntnis der Gruppenstruktur den Logarithmus $\log_{g_1}(x)$ für alle $x \in \langle g_1 \rangle$ berechnet. Dieser Algorithmus benötigt erwartete $O\left(2^d \lg^{\frac{5}{2}}\left(\frac{\nu}{\epsilon}\right)\right)$ Gruppenoperationen sowie erwartete $O\left(2^d \lg^{\frac{3}{2}}\left(\frac{\nu}{\epsilon}\right)\right)$ Orakel-Aufrufe.

Der Sätze 4.2.3 folgt aus den Sätzen 5.12 bzw. 5.14 und der einleitenden Bemerkung zu §5.2 in [Kal88].

4.3 Zweiter Beweisschritt

Hier kommt der größte Unterschied zwischen dem Ansatz von Kaliski und dem dieser Diplomarbeit zum tragen. Er verwendet ein nicht uniformes Angreifermodell. Das bedeutet, er modelliert den Angreifer durch eine Familie $C = C_k$ von Schaltkreisen polynomieller Größe. Betrachten wir Probleminstanzen I_k , so wird *jeder* dieser Instanzen ein *eigener* Schaltkreis zugeordnet, der das Problem löst. Diese Schaltkreise sind parametrisiert durch k , was aber nicht bedeutet, daß sie in irgendeiner Form ähnlich sein müssen. Die Vorstellung ist, daß man für jede Probleminstanz einen eigenen Algorithmus entwickeln darf.

Im Gegensatz dazu haben wir der Definition des Pseudozufallsbit-Generators ein uniformes Angreifermodell zugrunde gelegt. Wir fordern *einen* probabilistischen Polynomzeit-Algorithmus, der für *alle* Instanzen definiert ist und unendlich viele davon löst.

Wir präsentieren nun den am Kapitelanfang eingeführten Hauptsatz in einer auf Elliptische-Kurven-Gruppen eingeschränkten Version:

Satz 4.3.1

Sei EK die Folge Elliptischer-Kurven-Gruppen. Wenn das DL-Problem schwer ist, dann ist das höchstwertige Bit des Logarithmus für die Folge EK ununterscheidbar.

Beweis

Wir beweisen die Umkehrung und nehmen an, daß das höchstwertige Bit der Logarithmen für die Folge EK unterscheidbar ist. Dann existiert ein probabilistischer Algorithmus A , der für genügend große i , eine zufällig bezüglich i gewählte Instanz $\langle E_{a,b}(\mathbb{F}_p), (G_1, G_2), P \rangle$ mit $P \in \langle G_1 \rangle$ und eine Konstante c das höchstwertige Bit mit folgender Wahrscheinlichkeit vorhersagt:

$$\Pr[A(P) = b_1(P)] > \frac{1}{i^c}, \quad (4.25)$$

wobei $b_1(P)$ das höchstwertige Bit von $\log_{G_1}(P)$ ist und ein Polynom p existiert, sodaß für die Laufzeit des Algorithmus $T_A(i) \leq p(i)$ gilt.

Wir verwenden den Algorithmus Logarithm (4.2.4) in der modifizierten Version gemäß Satz 4.2.3 (mit $b=1$ und $d=0$) und ersetzen das Orakel durch den Algorithmus A . Der resultierende probabilistische Algorithmus L löst das DL-Problem in polynomieller Zeit in i . Dies resultiert aus den folgenden beiden Beobachtungen:

1. Nach 4.25 entspricht A bezüglich des Vorteils beim Schätzen einem Orakel O_1 mit $\epsilon = \frac{1}{i^c}$.

2. Die Laufzeit $T_L(i)$ des gewonnenen Algorithmus L beträgt gemäß 4.2.3 und Abschätzung von $T_A(i)$:

$$T_L(i) = O(\lg^{\frac{3}{2}}(\nu i^{2c}) + O(\lg^{\frac{5}{2}}(\nu i^{2c}) \cdot T_A(i)) \quad (4.26)$$

$$\leq O(\lg^{\frac{3}{2}}(\nu i^{2c}) + O(\lg^{\frac{5}{2}}(\nu i^{2c}) \cdot p(i)) \quad (4.27)$$

für obiges Polynom p . Die Laufzeit ist folglich durch ein Polynom in i beschränkt.

Damit ist der Widerspruch zur Annahme hergestellt, daß das DL-Problem schwer ist.

Kapitel 5

Konstruktionen

Wir sind nun an der Stelle angelangt, wo wir die Ergebnisse der Kapitel 2, 3 und 4 zusammenführen können.

Wir haben die diskrete Exponentiation als Oneway-Funktion eingeführt, die Konstruktion 2.5.1 fordert jedoch eine Oneway-Permutation. Ziel ist es nun zunächst, aus dem Kandidaten für die Oneway-Funktion eine Oneway-Permutation zu erzeugen.

5.1 Diskretes Exponentieren als Oneway-Permutation

Wir betrachten als erstes den einfachen Fall einer zyklischen Gruppe, daß heißt, wir beschränken uns zunächst, wie das auch in anderen kryptographischen Verfahren der Fall ist, auf eine zyklische Untergruppe großer Ordnung.

Sei also $E(F_p)$ eine elliptische Kurve der Ordnung ν und mit Gruppenstruktur (ν_1, ν_2) und $P \in E(\mathbb{F}_p)$ ein Punkt.

Die Abbildung

$$\begin{aligned} f : \{0, \dots, \text{order}(P)\} &\longrightarrow E(F_p), \text{ mit} \\ f(\alpha) &= \alpha \cdot P \end{aligned} \tag{5.1}$$

ist eine Oneway-Funktion unter der Annahme, daß das DL-Problem in dieser Gruppe schwer.

Um eine Oneway-Permutation bezüglich Bitstrings zu erhalten, wie wir dies in Definition 2.5.1 fordern, müssen wir sowohl die Exponenten α als auch die Punkte αP auf effiziente Weise binär kodieren. Für die Exponenten, die ja ganze Zahlen im Intervall zwischen 0 und $p - 1$ sind, ist dies die gewöhnliche Binärekodierung. Probleme bereitet die Binärekodierung der Punkte. Wir haben in 3.2.3 gesehen, wie wir allgemein elliptische Kurvenpunkte binär darstellen können. Diese Darstellung ist allerdings nicht surjektiv und wir können sie somit nicht verwenden, um die Oneway-Funktion f zu einer Permutation auf Bitstrings fortzusetzen.

Man kann das Problem auch umformulieren: Gesucht ist eine bijektive Funktion

$$\chi : \langle P \rangle \longrightarrow \{0, \dots, p - 1\}. \tag{5.2}$$

Die Funktion $\chi \circ f$ wäre somit eine Permutation auf einer Teilmenge der natürlichen Zahlen. Die binäre Kodierung ist hier nicht mehr kritisch. Zu bemerken ist allerdings,

daß die Funktion χ nicht der diskrete Logarithmus bzgl. P sein darf, da zum einen $\chi \circ f$ die Identität darstellt und zum anderen die Berechnung des diskreten Logarithmus nach Voraussetzung schwer ist.

Kaliski gibt in seiner Dissertation [Kal88] zwei verschiedene Konstruktionen an, die dieses Problem umgehen.

5.1.1 Verwendung spezieller Kurven

Die eine Möglichkeit ist die Verwendung spezieller Kurven, für die man eine Funktion χ (gemäß 5.2) leicht findet. Dies sind zum Beispiel Kurven $E_{0,b}(F_p) : y^2 = x^3 + b$, wobei zudem $p \equiv 2 \pmod{3}$ gilt. Kaliski führt sie in [Kal88, §6.1.1] mit dem Namen *einfache elliptische Kurven* ein. Er zeigt, daß die folgende Funktion die gesuchten Eigenschaften hat:

$$\chi_{E_{0,b}(F_p)}(P) = \begin{cases} y & , \text{ falls } P = (x, y), \\ p & , \text{ falls } P = \mathcal{O}. \end{cases} \quad (5.3)$$

Die einfachen elliptischen Kurven lassen sich jedoch aus einem anderen Grund nicht sinnvoll verwenden. Kaliski zeigt von diesen Kurven $E(F_p)$, daß sie $p+1$ Punkte haben. Gemäß Satz 3.2.2 sind dies also Kurven der Spur 1. In solchen supersingulären Kurven ist das Diskrete-Logarithmen-Problem jedoch nicht schwer.

5.1.2 Twist

Wir haben oben gesehen, daß das Problem bei der Verwendung der Kodierung gemäß Abschnitt 3.2.3 darin begründet ist, daß wir nicht jeder möglichen x -Koordinate einen Punkt auf der Kurve zuordnen können. Das Problem löst Kaliski, indem er dem Generator nicht mehr eine Kurve sondern ein Paar von Kurven, ein sogenanntes Twisted-Pair, zugrundelegt. Er geht von einer beliebigen Kurve $E(F_p)$ aus und bestimmt dazu eine weitere Kurve, den Twist $E^{\text{TW}}(F_p)$. Es läßt sich zeigen, daß, wenn einer x -Koordinate kein Punkt auf der einen Kurve zugeordnet werden kann, dies auf der anderen möglich ist. Kaliski beschreibt dies in [Kal88, §§6.1.2], wir werden diese Überlegungen hier ebenfalls nachzeichnen.

Definition 5.1.1 (Twist)

Sei $E(F_p) : y^2 = x^3 + ax + b$ eine elliptische Kurve und $\gamma \in \mathbb{F}_p$ quadratischer Nichtrest modulo p ¹. Wir konstruieren eine zweite Kurve $E_{\gamma^2 a, \gamma^3 b}^{\text{TW}}(F_p)$ wie folgt:

$$y^2 = x^3 + \gamma^2 ax + \gamma^3 b \quad (5.4)$$

Man kann folgende Eigenschaften zeigen:

¹Wenn wir an dieser Stelle für γ einen quadratischen Rest wählen, erhalten wir eine zu $E_{a,b}(F_p)$ isomorphe Kurve. Vergleiche dazu auch [Ham98, Definition 2.21]. Insbesondere folgt daraus, daß die Twists für verschiedene quadratische Nichtreste ebenfalls isomorph zueinander sind. Es ist in sofern beliebig, welchen quadratischen Rest wir verwenden.

Satz 5.1.1

Sei γ ein quadratischer Nichtrest in \mathbb{F}_p^* . Sei $E_{a,b}(\mathbb{F}_p)$ eine elliptische Kurve und $E_{\gamma^2 a, \gamma^3 b}^{\text{TW}}(\mathbb{F}_p)$ ihr Twist. Sei für beliebige Elemente $x \in \mathbb{F}_p$ die Funktion $f(x) = x^3 + ax + b$ gegeben. Dann gilt:

1. Falls $f(x) \neq 0$ quadratischer Rest ist, dann ist der Punkt $(x, \pm\sqrt{f(x)})$ auf der Kurve $E_{a,b}(\mathbb{F}_p)$.
2. Falls $f(x) \neq 0$ quadratischer Nichtrest ist, dann liegt der Punkt $(\gamma x, \pm\sqrt{\gamma^3 f(x)})$ auf der Kurve $E_{\gamma^2 a, \gamma^3 b}^{\text{TW}}(\mathbb{F}_p)$.
3. Falls $f(x) = 0$, dann ist der Punkt $(x, 0)$ auf der Kurve $E_{a,b}(\mathbb{F}_p)$ und der Punkt $(\gamma x, 0)$ auf der Kurve $E_{\gamma^2 a, \gamma^3 b}^{\text{TW}}(\mathbb{F}_p)$.

Beweis

Zu 1: Das haben wir uns weiter oben schon klar gemacht.

Zu 2: Man berechnet:

$$f(\gamma x) = (\gamma x)^3 + \gamma^2 a(\gamma x) + \gamma^3 b \quad (5.5)$$

$$= \gamma^3(x^3 + ax + b) \quad (5.6)$$

$$= \gamma^3 f(x) \quad (5.7)$$

Bleibt zu zeigen, daß $\gamma^3 f(x)$ ein quadratischer Rest ist in \mathbb{F}_p^* . Das ist aber klar, da das Produkt zweier quadratischer Nichtreste in \mathbb{F}_p^* wiederum quadratischer Rest ist.

Zu 3: Aus $y = 0$ folgt zunächst $f(x) = 0$, was $(x, 0) \in E_{a,b}(\mathbb{F}_p)$ impliziert. Wir haben oben gesehen, daß $f(\gamma x) = \gamma^3 f(x)$ gilt, woraus $(\gamma x, 0) \in E_{\gamma^2 a, \gamma^3 b}^{\text{TW}}(\mathbb{F}_p)$ folgt.

Wir werden die Oneway-Funktion nicht mehr auf einzelnen elliptischen Kurven sondern auf Twisted-Pairs definieren (vgl. [Kal88, Definition 6.1]):

Definition 5.1.2 (Twisted-Pairs)

Sei $E_{a,b}$ eine elliptische Kurve zur Problemgröße i , $\gamma \in \mathbb{F}_p$ ein quadratischer Nichtrest modulo p und $E_{\gamma^2 a, \gamma^3 b}^{\text{TW}}(\mathbb{F}_p)$ der Twist. Ein Twisted-Pair $T_{a,b,\gamma}(\mathbb{F}_p)$ ist definiert als die disjunkte Vereinigung von $E_{a,b}(\mathbb{F}_p)$ und $E_{\gamma^2 a, \gamma^3 b}^{\text{TW}}(\mathbb{F}_p)$. Für die Menge aller Twisted-Pairs zur Problemgröße i schreiben wir TP_i .

Da die Kurven $E(\mathbb{F}_p)$ und $E^{\text{TW}}(\mathbb{F}_p)$ sich in manchen Punkten schneiden können, werden wir bei den Punkten markieren, zu welcher Kurve sie gehören². Wir werden die Punkte, die zur Kurve E^{TW} gehören, mit dem Symbol ' kennzeichnen.

Definition 5.1.3 (Instanz von Twisted-Pairs)

Zu gegebener Problemgröße i besteht eine Twisted-Pair-Instanz aus einem Twisted-Pair $T_{a,b,\gamma}(\mathbb{F}_p)$, Erzeugerpaaren (G_1, G_2) bzw. (G'_1, G'_2) der elliptischen Kurven $E_{a,b}(\mathbb{F}_p)$ bzw. $E_{\gamma^2 a, \gamma^3 b}^{\text{TW}}(\mathbb{F}_p)$, und einem Punkt $P \in T_{a,b,\gamma}(\mathbb{F}_p)$, geschrieben $\langle T_{a,b,\gamma}(\mathbb{F}_p), (G_1, G_2), (G'_1, G'_2), P \rangle$.

²Dies wird in der Definition durch *disjunkte* Vereinigung ausgedrückt.

Wichtig ist das folgende Lemma:

Lemma 5.1.1

Sei $T_{a,b,\gamma}$ ein Twisted-Pair. Dann definiert die Funktion

$$\chi_{T_{a,b,\gamma}(\mathbb{F}_p)}(P) = \begin{cases} 2x + \operatorname{sgn}(y) & , \text{ falls } P = (x, y), y \neq 0, \\ 2\frac{x}{\gamma} + \operatorname{sgn}(y) & , \text{ falls } P = (x, y)', y \neq 0, \\ 2x + 0 & , \text{ falls } P = (x, 0), \\ 2x + 1 & , \text{ falls } P = (x, 0)', \\ 2p + 0 & , \text{ falls } P = \mathcal{O}, \\ 2p + 1 & , \text{ falls } P = \mathcal{O}', \end{cases} \quad (5.8)$$

eine bijektive Abbildung zwischen dem Twisted-Pair und der Menge $\{0, \dots, 2p + 1\}$, die in Polynom-Zeit berechnet werden kann.

Das Lemma entspricht [Kal88, Lemma 6.6], der Beweis ist ebenfalls dort zu finden. An dieser Stelle ist auch die Umkehrfunktion χ_T^{-1} angegeben. Wir haben nun die Möglichkeit, eine Permutation auf Bitstrings anzugeben, die im Kern das diskrete Exponentieren verwendet. Allerdings geschieht dies bezüglich zweier elliptischer Kurven, d.h. einer beliebigen Kurve und ihrem Twist. Wir werden bei der späteren Realisierung dafür sorgen müssen, daß das DL-Problem bezüglich beider Kurven schwer ist.

5.2 Definition des Pseudozufallsbit-Generators

Wir sind nun in der Lage, den Elliptische-Kurven-Pseudozufallsbit-Generator (EK-PZBG) zu definieren (vgl. [Kal88, §6.3.2]).

Definition 5.2.1 (EK-PZBG)

Zu einer zum Parameter i zufällig, gleichverteilt gewählten Twisted-Pair-Instanz $\langle T_{a,b,\gamma}(\mathbb{F}_p), (G_1, G_2), (G'_1, G'_2), P \rangle$ definieren wir den Elliptische-Kurven-Pseudozufallsbit-Generator G für ein Polynom $l(i)$ wie folgt:

$$G(s) \stackrel{\text{def}}{=} v(s_1) \dots v(s_{l(i)}) \quad (5.9)$$

für folgende Übergangsfunktionen:

1. Der initiale Zustand ist $s_0 = s$.
2. Der Punkt P_i berechnet sich aus dem aktuellen Zustand s_i vermöge

$$P_i = f(s_i) \quad (5.10)$$

$$= \begin{cases} (s_i \bmod \nu_1)G_1 + \left\lfloor \frac{s_i}{\nu_1} \right\rfloor G_2, & \text{falls } 0 \leq s_i < \nu; \\ ((s_i - \nu) \bmod \nu'_1)G'_1 + \left\lfloor \frac{s_i - \nu}{\nu'_1} \right\rfloor G'_2, & \text{falls } \nu \leq s_i < 2p + 2 \end{cases} \quad (5.11)$$

für $0 < i < l(i)$.

3. Der Folgezustand s_{i+1} berechnet sich aus dem Punkt P_i vermöge

$$s_{i+1} = \chi_T(P_i), \quad (5.12)$$

für $0 < i < l(i)$.

4. Die Funktion v ist definiert als

$$v(s_i) = \begin{cases} b_1(s_i, \nu_1), & \text{falls } 0 \leq s_i < \nu; \\ b_1(s_i - \nu, \nu'_1), & \text{falls } \nu \leq s_i < 2p + 2 \end{cases} \quad (5.13)$$

Der Satz 5.1.1 und das Lemma 5.1.1 garantieren, daß die Abbildung $\chi \circ f$ eine Permutation auf der Menge $\{0, \dots, 2p - 1\}$ ist. Gemäß Abschnitt 3.4 können wir zudem annehmen, daß sie eine Oneway-Permutation ist, wenn die elliptische Kurve und ihr Twist kryptographisch stark sind.

Es bleibt zu zeigen, daß die Ausgabe eine Folge von Hard-Core-Prädikaten ist. Dazu betrachten wir zu einem Index i den Punkt P_i und den Zustand s_i . Es gilt zunächst $v(s_i) = v(f^{-1}(P_i))$. Wir unterscheiden zwei Fälle:

- Falls $0 \leq s_i < \nu$ gilt, ist nach Konstruktion der Funktion f die ganze Zahl s_i der diskrete Logarithmus von P_i bezüglich des Erzeuger-Paars (G_1, G_2) .
- Falls $\nu \leq s_i < 2p + 2$ gilt, ist nach Konstruktion der Funktion f die ganze Zahl $s_i - \nu$ der diskrete Logarithmus von P_i bezüglich des Erzeuger-Paars (G'_1, G'_2) .

In beiden Fällen folgt die Hard-Core-Eigenschaft aus Satz 4.3.1. Satz 4.2.3 bietet hier die Möglichkeit die Funktion v dahingehend zu ändern, daß sie mehrere simultan sichere höchstwertige Bits pro Iteration ausgibt. Vergleich dazu auch 2.7.

Kapitel 6

Realisierung

Dieses letzte Kapitel der Diplomarbeit beschreibt die Implementierung des Elliptische-Kurven-Pseudozufallsbit-Generators. Als Erstes werden die Rahmenbedingungen dargestellt, die sich teils direkt aus der Aufgabenstellung der Diplomarbeit, teils aus der Systemumgebung ergeben.

6.1 Vorgaben

In diesem Abschnitt sind die grundlegenden Überlegungen hinsichtlich Vorgaben der bestehenden Infrastruktur, Wahl der Programmiersprache und Design des Generators wiedergegeben.

6.1.1 Wahl der Programmiersprache

Die Verwendung der Programmiersprache Java von Sun Microsystems stand für diese Diplomarbeit von vornherein fest, da der zu implementierende Pseudozufallsbit-Generator konform zur Java Cryptography Architecture (JCA) sein sollte. Die JCA stellt ein Framework dar, welches ermöglicht, kryptographische Dienste, unter anderem Pseudozufallsbit-Generatoren, in einer standardisierten Weise in Java-Umgebungen bereitzustellen. Wir werden im Abschnitt 6.1.2 genauer auf die JCA und ihr Provider-Konzept sowie die Anforderungen, die sich für dazu konforme Generatoren ergeben, eingehen.

Als weiteres Argument für die Verwendung von Java wird häufig die Plattformunabhängigkeit der erstellten Programme angeführt. Wir werden jedoch bei der Implementierung auf diese Eigenschaft verzichten, um einen wesentlichen Nachteil von Java-Programmen zu kompensieren: Die Geschwindigkeit ausführbarer Programme ist immer noch um ein vielfaches langsamer als zum Beispiel die eines vergleichbaren in C++ geschriebenen Programms.

In unserem Fall bietet sich eine hybride Lösung an, die sowohl die Konformität zur JCA wahrt, als auch die Geschwindigkeitsvorteile nativer (in unserem Falle C++) Implementierungen nutzt. Die rechenintensiven Programmteile, also Kurven- und Parametererzeugung und der eigentliche Pseudozufallsbit-Generator, sind in der Sprache C++ geschrieben. Die in Java entwickelten Programmteile bilden lediglich eine Schnittstelle, über die man den Generator auch aus Java heraus via Java Native Inter-

face nutzen kann. Eine Ausnahme bildet der Seed-Generator, der vollständig in Java implementiert ist. Auf die Gründe hierfür gehen wir in Abschnitt 6.3.2 ein.

Um die Schnittstelle zwischen Java- und C++ -Programmteilen zu implementieren, verwenden wir ein weitere Framework von Java, das Java Native Interface. Es ermöglicht Teile des Programms, genauer gesagt Methoden-Rumpfe, in einer anderen Programmiersprache zu implementieren.

Zum Einsatz kamen in dieser Diplomarbeit die Programmierumgebungen "Java 2 SDK, Standard Edition, v 1.3" der Firma Sun Microsystems und "Microsoft Visual Studio 6.0 Enterprise Edition" der Firma Microsoft.

6.1.2 Java Cryptography Architecture (JCA)

Die Java Cryptography Architecture bildet ein Grundgerüst zur Entwicklung und Bereitstellung kryptographischer Komponenten unter Java und ist als solches Bestandteil des oben genannten Java-Packets.

Die Grundbausteine im Framework sind die Dienste. Sie werden aus Sicht eines Programmierers, der einen Dienst bereitstellen will, durch abstrakte Basisklassen repräsentiert, wobei für jede Art Dienst eine Klasse existiert. Die Namen dieser Basisklassen enden stets mit der Endung "Spi", was für "Service Provider Interface" steht. Die in diesen Interfaces vorgesehenen Methoden beginnen mit dem Präfix "engine". Für Pseudozufallsbit-Generatoren heißt diese Klasse "SecureRandomSpi". In "SecureRandomSpi" sind die Funktionalitäten festgelegt, die das Framework von einer Implementierung eines Pseudozufallsbit-Generators erwartet. Dies geschieht im Sinne einer abstrakten Basisklasse, welche die Signaturen¹ der benötigten Funktionen vorgibt. Um den Dienst Pseudozufallsbit-Generator zu implementieren müssen wir also eine Klasse erstellen, die von `SecureRandomSpi` erbt. Sie beinhaltet folgende Methoden:

Rückgabewert	Funktionsname / Parameter
protected abstract byte[]	engineGenerateSeed(int numBytes)
protected abstract void	engineNextByte(byte[] bytes)
protected abstract void	engineSetSeed(byte[] seed)

Tabelle 6.1: Klasse `SecureRandomSpi`

Zunächst ist zu bemerken, daß die JCA vorschreibt, daß jeder Pseudozufallsbit-Generator einen eingebauten Seed-Generator hat. Dieser wird verwendet, um seinen initialen Seed zu erzeugen, falls der Generator nicht von der Anwendung mit einem Seed versorgt wird. Der Seed-Generator ist in Abschnitt 6.3.2 beschrieben.

Die erste Methode bietet einer Anwendung die Möglichkeit, den Seed-Generator direkt anzusprechen. Über die zweite Methode wird der eigentliche Pseudozufallsbit-Generator aufgerufen und die dritte Methode ermöglicht es schließlich, den Seed des Generators explizit zu setzen.

¹Signatur bedeutet in diesem Zusammenhang, die Beschreibung einer Funktion durch ihren Namen, ihre Parameterliste und ihren Rückgabewert, es sind nicht die Signaturen im Sinne einer digitalen Unterschrift gemeint!

Zur Anwendungsseite hin repräsentiert das Framework einen Dienst zunächst als Klassen (Proxy). Diese Proxy-Klassen haben jeweils die selben Namen wie die korrespondierenden Basisklassen, jedoch ohne die Endung 'Spi'. In unserem Fall heißt diese also `SecureRandom`. Sie besitzt statische Methoden `getInstance(...)`, mit denen man ein Objekt zu diesem Dienst erzeugen kann, wobei alle Implementierungen dieses Dienstes aller beim System bekannten Provider zur Verfügung stehen. Das Resultat ist ein Proxy-Objekt vom Typ `SecureRandom`. Die Anwendung kann nun den Dienst nutzen, indem es Methoden beim Proxy-Objekt aufruft. Dieser Aufruf wird dann an das entsprechende Objekt einer `Spi`-Klasse durchgereicht und dort bearbeitet. Das Zusammenspiel beschreibt folgendes Schaubild:

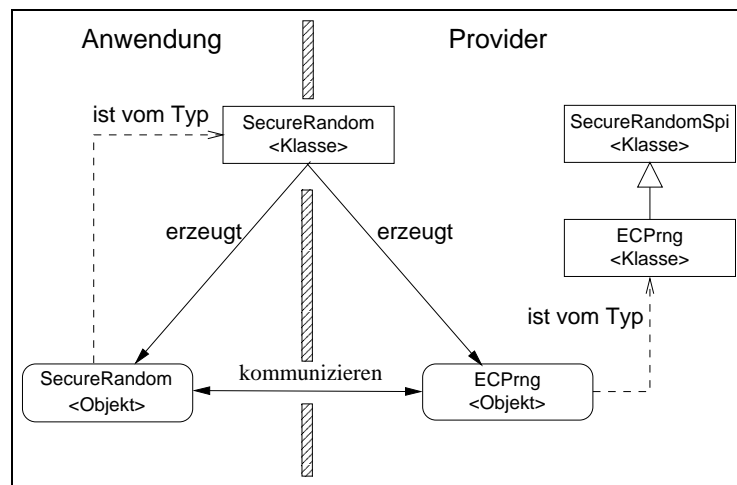


Abbildung 6.1: Zusammenspiel von `SecureRandom` und `SecureRandomSpi`

6.1.3 JNI

Das Java Native Interface bietet, wie oben erwähnt die Möglichkeit, den Rumpf von Java-Methoden in einer anderen Programmiersprache zu implementieren. Zu diesem Zweck werden solche Methoden im Java-Quelltext mit dem Schlüsselwort `native` gekennzeichnet. Die Implementierung der Methode erfolgt dann in der gewählten Zielsprache und wird als dynamische Bibliothek zur Laufzeit eingebunden. Zu diesem Zweck erzeugt der Java-Übersetzer an den entsprechenden Stellen Code-Fragmente, die nach festgelegten Namenskonventionen die entsprechenden nativen Funktionen in der dynamischen Bibliothek aufrufen. Diese Funktionen sind auf C++-Seite jedoch nicht in eine Klasse eingebettet und kennen nicht den Zustand des Objekts, in dem sie aufgerufen werden. Dies kann zu Problemen führen, wenn auf der Java-Seite mehrere Objekte gleichen Typs existieren, da hier die Berechnungen in den nativen Funktionen davon abhängen, für welches Objekt diese durchzuführen sind.

Eine Möglichkeit wäre, den relevanten Teil des Objektzustands bei jedem Aufruf einer nativen Methode als Parameter zu übergeben. Diese Vorgehensweise stößt auf zwei Probleme. Zum einen können dies durchaus sehr umfangreiche Datenmengen sein, die jeweils zwischen der Java- und der C++-Seite hin- und herbewegt werden müssen. Zum anderen müssen auf beiden Seiten die entsprechenden Struk-

turen vorhanden sein, um die Daten auf der jeweiligen Seite repräsentieren zu können. Der Elliptische-Kurven-Pseudozufallsbit-Generator verwendet zum Beispiel elliptische Kurven. Es müssten also sowohl in Java als auch in C++ Datentypen für elliptische Kurven vorhanden sein.

Um diese Probleme zu umgehen, erzeugen wir zu jedem Java-Objekt auch ein Objekt auf der C++-Seite. Diese beiden Objekte werden eng miteinander verwoben. Wird das Java-Objekt erzeugt, dann sorgt der Konstruktor dafür, daß auch das C++-Objekt erzeugt wird. Wird ein Java-Objekt vom Garbage-Collector aus dem Speicher entfernt, sorgt es in seiner `finalize()`-Methode dafür, daß das korrespondierende C++-Objekt ebenfalls gelöscht wird. Um dies realisieren zu können, müssen wir uns auf der Java-Seite einen Zeiger auf das C++-Objekt merken. Da Java keinen Datentyp für Zeiger kennt, verwenden wir den Datentyp `long`, in der Annahme, beide haben die gleiche Bitlänge. Dieses Vorgehen wurde bereits in einem anderen Projekt im Zusammenhang mit der LiDIA-Bibliothek (siehe Abschnitt 6.1.4) verwendet (vgl. [Hah00]).

6.1.4 Die Bibliothek LiDIA

LiDIA stellt eine C++-Bibliothek für algorithmische Zahlentheorie dar. Insbesondere stellt sie die von uns benötigten Komponenten zum Rechnen mit langen Ganzzahlen, Primkörpern und elliptischen Kurven bereit. Die Bibliothek wird von der LiDIA-Gruppe am Fachbereich Informatik der TU-Darmstadt entwickelt (vgl. [LiD]). Wir verwenden in unserer Implementierung die in folgender Tabelle aufgelisteten Klassen:

Klassenname	Funktion
<code>bigint</code>	Ganzzahlarithmetik
<code>galois_field</code>	Repräsentation endlicher Körper
<code>gf_element</code>	Repräsentation von Körperelementen
<code>elliptic_curve <gf_element></code>	Elliptische Kurven über endlichen Körpern
<code>point <gf_element></code>	Kurvenpunkte

Tabelle 6.2: Verwendete Klassen aus LiDIA

6.1.5 Die Klasse `gac_complex_multiplication`

Diese Klasse repräsentiert einen Algorithmus zur Erzeugung kryptographisch starker elliptischer Kurven $E(\mathbb{F}_p)$, der einen anderen Ansatz verfolgt, als daß zufällige Wahlen der Parameter a, b (vgl. Abschnitt 3.4.2). Der Algorithmus nutzt die Tatsache, daß elliptische Kurven eine komplexe Multiplikation besitzen. Da dieses Thema den Rahmen der Diplomarbeit sprengen würde, sei der Leser zum Beispiel auf [Ham98, §4] verwiesen. Für uns relevant ist die Tatsache, daß wir bei diesem Verfahren die Größenordnung des großen Primteilers l und des Kofaktors k der Gruppenordnung im Voraus angeben können. Die Klasse `gac_complex_multiplication` gehört zu einer Sammlung von Klassen, welche LiDIA ergänzen. Die von uns benötigten Komponenten wurden von H. Baier implementiert (vgl. [Bai00]), und wir werden diese als statische Bibliothek in unser Programm einbinden. Eine Anpassung war nötig, um diese Komponenten für unsere Zwecke nutzbar zu machen. Ursprünglich war das Paket

darauf ausgelegt, nur *eine* kryptographisch starke Kurve zu erzeugen. Dazu wurde zu entsprechend gewähltem Primfaktor l und Kofaktor k die Primzahl p des Körpers \mathbb{F}_p so gewählt, daß eine Kurve *oder* oder ihr Twist kryptographisch stark ist. Wir benötigen aber, daß *beide* kryptographisch stark sind, was durch eine Änderung der letzten Fallunterscheidung in der Funktion `find_good_prime()` (vgl. [Bai00, Algorithmus `findPrime(...)`]) erreicht wird. Entsprechend wurden die Methoden eingefügt, um die Parameter des Twists abfragen zu können.

Aus der Klasse `gec_complex_multiplication` verwenden wir folgende Methoden:

Methodenname	Beschreibung
<code>set_lower_bound_bitlength_r</code>	Untere Schranke der Bitlänge des großen Primteilers
<code>set_upper_bound_k</code>	Obere Schranke des Kofaktors k
<code>generate</code>	startet die Kurven-Erzeugung

Tabelle 6.3: Verwendete Methoden von `gec_complex_multiplication`

6.2 Design und Implementierung

Wir können nun die einzelnen Komponenten identifizieren. Folgendes Diagramm gibt wieder, wie sich die einzelnen Komponenten im System verteilen und wie sie untereinander und mit den Systemgrenzen in Beziehung stehen.

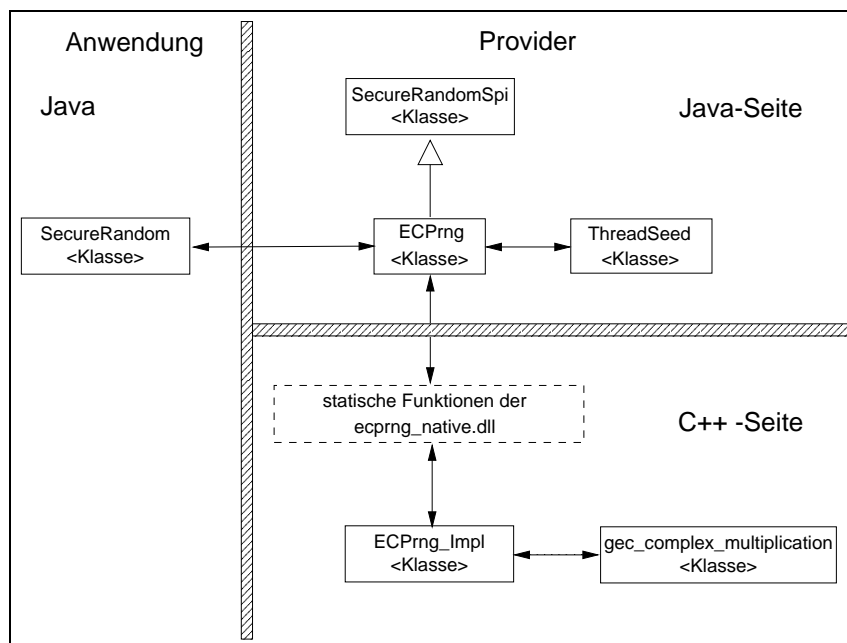


Abbildung 6.2: Verteilung der Komponenten

Die nachfolgende Tabelle listet alle Programm-Komponenten zusammen mit der Form ihrer Realisierung und eine kurze Beschreibung auf:

Komponente	Realisierung/ Datei	Kurzbeschreibung
ECPrng	Java-Klasse ECPrng.java	Die Repräsentation des Generators auf der Java-Seite
ThreadSeed	Java-Klasse ECPrng.java	Erzeugung von Seeds, falls nicht vom Anwender bereitgestellt (Vorgabe der JCA)
Native-Interface, C++-Seite	dynamische C++-Bibliothek ecprng_native.cpp	Spiegelung der Funktionalität von der C++- auf die Java-Seite; Eindeutige Zuordnung Java-Objekt zu C++-Objekt
ECPrng_impl	C++-Klasse ecprng_impl.cpp	Der eigentliche Generator
Kurven-Generator	Teil von ECPrng_impl generate.cpp	Erzeugen einer elliptischen Kurve und ihres Twists mittels CM-Methode

Tabelle 6.4: Liste der Komponenten

6.3 Beschreibung der Komponenten

6.3.1 ECPrng

Diese Klasse repräsentiert den Elliptische-Kurven-Pseudozufallsbit-Generator auf der Java Seite. Sie ist, wie in Abschnitt 6.1.2 gefordert, von der Klasse `SecureRandomSpi` abgeleitet und implementiert als solche die in Tabelle 10 aufgeführten Methoden. Die Methoden `engineNextByte()` und `engineSetSeed()` reichen die Parameter im wesentlichen an die Implementierung des Generators auf der C++-Seite weiter. In erster Methode wird zudem überprüft, ob der Generator bereits mit einem Seed versorgt wurde. Ist dies nicht der Fall, wird vor dem Wechsel auf die C++-Seite, die Seed-Erzeugung angestoßen. Die Methode `engineGenerateSeed()` ruft den als innere Klasse implementierten Seed-Generator auf. Dieser ist im Detail im Abschnitt 6.3.2 beschrieben.

6.3.2 ThreadSeed

Den Seed-Generator sieht die Java Cryptography Architecture für den Fall vor, daß eine Anwendung eine Pseudozufallsbit-Folge anfordert, aber der Generator noch nicht mit einem Seed initialisiert wurde. In diesem Fall ist es sinnvoll, daß der Generator seinen Seed selber erzeugen kann. Der Seed-Generator ist als inneren Klasse `ThreadSeed` innerhalb der Klasse `ECPrng` implementiert.

Die Schnittstelle zur Anwendung bildet die Methode `public byte[] generateSeed(int bitLength)`.

Die Seed-Erzeugung arbeitet wie folgt:

Der Thread, in dem der Aufruf erfolgt, wir nennen ihn A, erzeugt einen neuen Thread B und startet ihn. Dann "schläft" er für eine bestimmte Zeit (50 Millisekunden). Das Java-Laufzeitsystem "weckt" ihn dann nach der vorgegebenen Zeit wieder auf. Während dieser Zeit zählt Thread B eine Variable hoch, angefangen von Null. Wenn A wieder aufwacht, stoppt er B und läßt die Zählervariable aus. Ist das Ergeb-

nis ungerade, gibt er das Bit '1' aus, sonst '0'. Dieser Vorgang wird so oft wiederholt, bis alle Bits erzeugt werden.

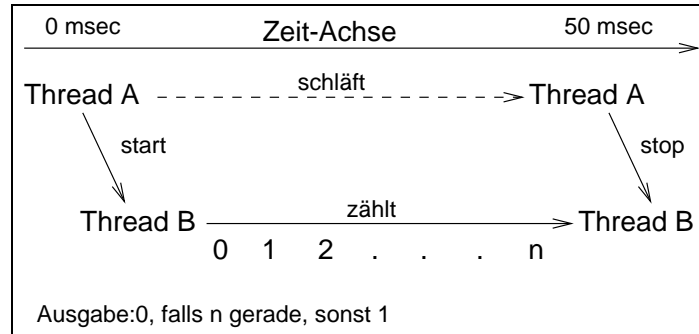


Abbildung 6.3: Arbeitsweise von ThreadSeed

6.3.3 ECPrng_impl

Die Klasse `ECPrng_impl` beinhaltet den eigentlichen Pseudozufallsbit-Generator nebst dem Programmteil für die Erzeugung der Kurven. Sie ist primär für die Integration in die JCA mittels der Klasse `ECPrng` und den Bibliotheksfunktionen der Datei `ecprng_native.cpp` vorgesehen, lässt sich aber ohne weiteres auch auf C++-Seite direkt verwenden. In diesem Abschnitt wird die Schnittstelle der Klasse nach außen, sowie die Funktionsweise des Generators beschrieben. Der Kurvenerzeugung ist ein eigener Abschnitt (6.3.4) gewidmet.

Folgende Tabelle listet die Schnittstelle der Klasse:

Rückgabewert	Funktionsname / Parameter
void	<code>generate_bytes (unsigned char * bytes, unsigned int count)</code>
int	<code>get_seed_estimate()</code>
void	<code>set_seed(const signed char * seed_bytes, const unsigned int len)</code>

Tabelle 6.5: Schnittstelle der Klasse `ECPrng_impl`

Die erste Methode beinhaltet den eigentlichen Generator. Er funktioniert exakt, wie in Kapitel 5 beschrieben, wobei die dortigen Funktionen χ und sgn durch die privaten Methoden `chi()` und `sgn()` repräsentiert werden.

Die rechenintensivste Operation in dieser Methode ist die diskrete Exponentiation, die auf Basis der Erzeuger-Paare der elliptischen Kurve bzw. ihres Twists statt findet. Bei einer ersten Implementierung, bei der für diese Operation die in LiDIA implementierte Exponentiation verwendet wurde, zeigte sich, daß 99,5% der Laufzeit von diesem Rechenschritt verbraucht wurde². Es ist also Sinnvoll, hier bei einer Optimierung anzusetzen.

²Dieser und die folgenden Werte wurden mit Hilfe des Profilers aus der Microsoft Visual Studio 6.0 Enterprise Edition gemessen

Wir können uns an dieser Stelle zu Nutze machen, daß wir immer bezüglich der gleichen Basis exponentieren, nämlich der Generator-Paare beider Kurven. Dies legt nahe, für die schnelle Exponentiation die Punkte $2^i G$ für alle Erzeuger G der beiden Erzeuger-Paare und die ganzen Zahlen i für $0 < i < \lg(G) + 1$ vorzuberechnen und in Tabellen zu speichern. Dieser Schritt geschieht einmal, direkt nach dem Erzeugen der Kurven. Bei dem obigen Rechenschritt, sind dann lediglich gemäß der Binärdarstellung des Exponenten die entsprechenden Punkte aus den Tabellen zu wählen und zu addieren. Dies bewirkt eine Steigerung der Geschwindigkeit um Faktor 10, 3.

6.3.4 Erzeugung der Kurven

Wir werden uns nun mit der Komponente beschäftigen, die während der Initialisierungsphase des Generators zwei elliptischen Kurve, eine primäre und ihren Twist, auswählt und die später nötigen Parameter zu diesen Kurven berechnet. Die Kurvenerzeugung ist innerhalb der C++-Methode `generate_curve()` implementiert. Sie ist eine Methode der Klasse `ecprng_impl` und in der Datei `generate.cpp` getrennt von den anderen Methoden dieser Klasse angesiedelt. Dies ermöglicht, die Kurvenerzeugung einfach durch eine andere Implementierung zu ersetzen.

Wir werden von dem in Abschnitt 3.3 angestellten Überlegungen aus Effizienzgründen teilweise abweichen.

Der in 3.4.2 eingeführte Begriff der starken elliptischen Kurve ermöglicht uns, die Kurvenerzeugung auf einen konkreten Sicherheitsparameter hin auszulegen. Zum einen müssen wir garantieren, daß die Ordnung ν der elliptischen Kurve einen großen Primteiler l (mindestens 160 Bit lang) besitzt, auf der anderen Seite hängt die Laufzeit des Generators wesentlich davon ab, welche Ordnung die elliptische Kurve hat und, damit verbunden, wie groß (in Binärdarstellung) die Elemente des zugrundeliegenden Primkörpers sind. Wir sind also daran interessiert, den Kofaktor k möglichst klein zu halten. Der von uns verwendete Algorithmus zum Erzeugen der Kurven (siehe Abschnitt 6.1.5) benötigt als Eingabe eine untere Schranke der Bitlänge des Primteilers l und eine obere Schranke der Bitlänge des Kofaktors k . Erstere folgt aus den Kriterien für kryptographisch starke Kurven (160 Bit). Für letztere hat sich bewährt, einen Kofaktor $k < 2^{16}$ zu wählen. Die Gruppenordnung hat dann etwa 176 Bit.

Insgesamt funktioniert die Kurvenerzeugung nun wie folgt:

1. Wir bestimmen eine Kurve $E_{a,b}(\mathbb{F}_p)$ und ihren Twist $E_{\gamma^2 a, \gamma^3 b}^{\text{TW}}(\mathbb{F}_p)$ mittels komplexer Multiplikation. Das Verfahren liefert uns zusätzlich zu den Kurven noch die Primteiler l_1, l_2 und die Kofaktoren k_1, k_2 der beiden Kurven. Die weiteren Schritte erfolgen jeweils für beide Kurven. Wir schreiben für die jeweilige Kurve kurz E .
2. Wir faktorisieren die Ordnung der Kurve E . Das ist schnell zu machen, da wir den Primteiler l von $\text{order}(E)$ kennen. Wir müssen also lediglich den Kofaktor faktorisieren und der ist kleiner als 2^{16} .
3. Wir bestimmen ein Element maximaler Ordnung ν_1 mit dem Verfahren, daß wir in 3.3 angegeben haben. Wir machen uns bei der Bestimmung der Ordnung zu Nutze, daß wir die Faktorisierung der Gruppenordnung und insbesondere den großen Primteiler l kennen. Dieser teilt auch die Maximalordnung und es reicht

folglich aus, wenn wir Punkte P zufällig auf der Kurve wählen, jeweils die Punkte lP berechnen, und den Punkt P als Punkt maximaler Ordnung annehmen, für den $\text{order}(lP)$ am größten war. Der Vorteil ist, daß die Berechnung der Ordnung von lP schneller vonstatten geht. Wir setzen $G_1 = P$.

4. Wir müssen nun einen Punkt G_2 finden, der zusammen mit G_1 die Kurve erzeugt und die Ordnung $\nu_2 = \frac{\nu}{\nu_1}$ hat. Da LiDIA keine Weil-Paarung zur Verfügung stellt und ihre Implementierung den Rahmen dieser Arbeit sprengen würde, verwenden wir entgegen der Bemerkung zu 3.3 die Berechnung des diskreten Logarithmus mittels dem in LiDIA implementierten 'Baby-Step-Giant-Step'-Verfahren.

- Wählen einen Punkt P zufällig
- Berechne $\text{order}(P)$. Falls $n_2 \nmid \text{order}(P)$ wiederhole ab Schritt 1.
- Setze $Q = \frac{\text{order}P}{\nu_2} P$. Dieser Punkt hat jetzt Ordnung ν_2 .
- Prüfe, ob $Q \in \langle \frac{\nu}{\nu_2} G_1 \rangle$, der Untergruppe von $\langle G \rangle$ der Ordnung ν_2 ist. Falls Ja, sind wir fertig, falls Nein, wiederhole ab Schritt 1.

6.4 Laufzeiten

Für die Analyse der Laufzeiten des Generators haben wir die drei zeitaufwendigsten Komponenten untersucht: Seed-Generator, Kurven-Erzeugung und der eigentliche Pseudozufallsbit-Generator. Die Zeiten wurden jeweils aus Java heraus mittels der Funktion `System.currentTimeMillis()` gemessen. Als Testsystem diente ein Pentium-III-500MHz mit 128 MB Speicher. Die Werte sind jeweils der Durchschnitt über 10 Meßreihen. Der Wert bezüglich der Seed-Erzeugung bezieht sich auf das Bereitstellen von 52 Bytes, berechnet gemäß der Methode `get_seed_estimate()`, der Wert bezüglich der Kurven-Erzeugung mißt die Erzeugung der Kurve und des Twists zusammen mit den nötigen Parametern, der letzte Meßwert bezieht sich auf das Erzeugen von 5000 pseudozufälligen Bytes.

Komponente	Laufzeit [sec]
Seed-Generator	19.6
Kurven-Erzeugung	223.2
Pseudozufallsbit-Erzeugung	177.0

Tabelle 6.6: Timings

Dazu ist zu bemerken, daß die Zeiten für die Kurvenerzeugung beträchtlich schwanken können (in den Messungen um bis zu 92%). Die beiden anderen Meßreihen liegen sehr dicht an ihrem Durchschnitt.

6.5 Ausblick

Der in dieser Diplomarbeit erstellte Pseudozufallsbit-Generator läuft stabil und die Integration in die Java Cryptography Architecture gelang problemlos. Dennoch ist genug

Spielraum für Erweiterungen und Verbesserungen. Dieser Abschnitt soll eine kurze Übersicht darüber geben.

Integration mit LiDIA

Im Zusammenhang mit der Verwendung der Klasse `gec_complex_multiplication` zur Erzeugung der elliptischen Kurven, war es nötig eine angepaßte LiDIA-Bibliothek zu verwenden, da für die Implementierung der Klasse Änderungen in LiDIA nötig waren. Das hat zur Folge, daß der in dieser Diplomarbeit bereitgestellte Programm-Code nicht mit den allgemein verfügbaren LiDIA-Paketten kompiliert werden kann.

Verbesserung der Geschwindigkeit

Hier bieten sich eine Reihe Möglichkeiten an. Prädestiniert dafür sind die beiden Komponenten Kurven-Erzeugung und Pseudozufallsbit-Generator, da sie die meiste Rechenzeit verbrauchen.

Bei der Kurven-Erzeugung sind die Möglichkeiten eingeschränkt, da das eigentliche Erzeugen der Kurven mittels komplexer Multiplikation schon zu den schnellen Verfahren zählt. Lediglich das anschließende Berechnen der nötigen Parameter (Gruppenstruktur, Generator-Paar) bietet da noch Spielraum. Ein leistungsfähigerer Algorithmus ist zum Beispiel in dem Algebra-Paket "Magma" (vgl. [mag]) implementiert.

Auch der eigentliche Generator bietet noch Möglichkeiten zur Effizienzsteigerung. Auch nach der Verbesserung aus Abschnitt 6.3.3 ist der Schritt der diskreten Exponentiation bezüglich des Generator-Paars die aufwendigste Operation. Eine Verbesserung der implementierten schnellen Exponentiation durch die Verwendung vorzeichenbehafteter Bits schlägt W.Bosma in [Bos99b] vor.

Die Bibliothek LiDIA läßt sich auch noch beschleunigen. Das derzeit in LiDIA standardmäßig verwendete Paket zur Ganzzahlarithmetik "libl" läßt sich durch eine schnellere Version ersetzen.

Anhang A

Java Quellcode

A.1 cdc.ECPrng.java

```
package cdc;

/**
 * This class makes the Elliptic Curve Pseudorandom Number
 * Generator available to the Java Cryptography Architecture
 * (JCA). It supplies additionally functionality, to make the
 * generator JCA compliant.
 *
 * @author Marcus Lippert
 * @version 1.0
 * @since 21.12.2000
 */
public class ECPrng extends java.security.SecureRandomSpi {

    private boolean isSeeded = false;

    private byte[] theSeed = null;
    /**
     * A reference to the attached native object. Since there
     * are no pointers in Java, the pointer to the native
     * object is converted into a long, which can be stored
     * on the java side.
     *
     * @see #attacheNativeObject()
     * @see #releaseNativeObject()
     */
    private long NativePointer = 0;

    /**
     * Reference to the internally used seed generator.
     */
    private ThreadSeed myThreadSeed;

    /**
     * The (default-) constructor of the Pseudorandom Number
     * Generator. It usually gets invoked by the JCA framework.
     * The constructor attaches a native object and installs a
     * shutdown hook, which will ensure that it's
     * <T>finalize()</T>-Method gets called, when the

```

```

    * application terminates. Secondly an instance of the
    * internal seed generator is set up.
    */

public ECPrng() {
    super(); // Perhaps SecureRandomSpi has some startup code

    NativePointer = attacheNativeObject(); // connect the
                                           // native object

    // Installing a shutdown hook, to assure that
    // the native side gets cleaned up, when the
    // system goes down.
    Runtime.getRuntime().
        addShutdownHook (new Thread() { // adding a shutdown hook
public void run() {
    endApp(); // which will call endApp()
    }
});

    myThreadSeed = new ThreadSeed(); // Instanciating the
                                     // seed generator
}

/**
 * During finalization the attached objekt gets deleted
 */
protected void finalize() throws java.lang.Throwable {
    if (NativePointer != 0)
        releaseNativeObject(NativePointer);

    super.finalize();
}

////////////////////////////////////
// Mehods needed in a JCA complient PRNG.
////////////////////////////////////

/**
 * This method will seed the Pseudorandom Number Generator.
 * Since the user chooses how many seed bytes he will
 * provide, the generator expands the seed if needed.
 * This is done by using the LiDIA builtin Pseudorandom
 * Number Generator.
 *
 * @param seed byte array containing the seed.
 */
protected void engineSetSeed(byte[] seed) {

    int needed = EstimateSeed(NativePointer);

    if (seed != null) {
        byte[] tmp = new byte[needed];

        System.arraycopy( seed, 0, tmp, 0,

```

```

Math.min( needed, seed.length );
    if (seed.length < needed) {
byte[] tmp2 =
    myThreadSeed.generateSeed( 8 * ( needed - seed.length ));

System.arraycopy( tmp2, 0, tmp, seed.length,
    needed - seed.length );
    }

    SetSeed(NativePointer, tmp);
    isSeeded = true;
}
}

/**
 * Generates the psudorandom numbers or actually a byte
 * array containig a pseudorandom sequence of bytes. To
 * do so, the corresponding native method is invoked. If
 * an error ocurred during generation the resulting array
 * is set to <T>null</T>.
 *
 * @param bytes  array which the result bytes will be
 *                stored in. The array
 *                length determs the amount of bytes to be
 *                generated.
 *
 * @see #LiDIA_NextBytes()
 */
protected void engineNextBytes(byte[] bytes) {

    byte[] b;

    if (NativePointer!=0) {
        if (!isSeeded)
selfSeed();

        b = LiDIA_NextBytes(NativePointer, bytes.length);

        System.arraycopy(b,0,bytes,0,bytes.length);
    }
}

/**
 * Generates seed bytes using the builtin seed generator.
 *
 * @param numBytes The number of bytes to generate
 *
 * @return array containing the random bytes.
 */
protected byte[] engineGenerateSeed(int numBytes) {
    return myThreadSeed.generateSeed(8*numBytes);
}

```

```

////////////////////////////////////

```

```

// private methods
////////////////////////////////////

private void selfSeed() {
    int needed = EstimateSeed(NativePointer);

    byte[] tmp = myThreadSeed.generateSeed( 8 * ( needed ));

    SetSeed(NativePointer, tmp);
    isSeeded = true;
}

/**
 * Attaches a native object to the java object.
 * This method is native and is implemented in c++.
 *
 * @return the pointer to the native object converted to
 *         a long.
 *
 * @see #releaseNativeObject()
 */
private native long attacheNativeObject();

/**
 * Releases an attached native object, i.e. deletes it.
 * This method is native and is implemented in c++.
 *
 * @param ptr the pointer to the object as a long
 *
 * @see #attacheNativeObject()
 */
private native void releaseNativeObject(long ptr);

/**
 * The native version of the byte generation.
 *
 * @param ptr the pointer to the object as a long
 * @param count the number of bytes which will be generated
 *
 * @return the generated array
 */
private native byte[] LiDIA_NextBytes(long ptr, int count);

/**
 * Ask the native object, which amount of seed-bytes is
 * needed.
 *
 * @param ptr the pointer to the object as long
 * @return the number of bytes
 */
private native int EstimateSeed(long ptr);

/**
 * Supports the native object with the seed.
 *
 * @param ptr the pointer to the object as long

```

```

    * @param SeedBytes array containing the seed
    */
private native void SetSeed(long ptr, byte []SeedBytes);

////////////////////////////////////
// class-intitilisation code
////////////////////////////////////

static {
    System.loadLibrary("cdc/dll/ecprng_native");
}

private void endApp() {
    if (NativePointer != 0)
        releaseNativeObject(NativePointer);
}

////////////////////////////////////
// inner classes
////////////////////////////////////

/**
 * An object of this inner class is used, to generate an
 * initial seed, if requested. It uses multithreading.
 * Two threads (A and B) are generated. A starts counting.
 * B sleeps for an amount of time and then stops A.
 *
 * @author Marcus Lippert
 * @version %I% %V%
 */
protected final class ThreadSeed
implements Runnable {

    private final int
        SLEEP_TIME = 50;        // time for B to sleep
    private volatile Thread
        MyThread,CountThread; // References to the threads.
    private volatile int
        Counter;                // Variable counted up by A
    private volatile boolean
        stopCountThread;        // Used for handshaking
                                // between A and B
    private Exception
        CountThreadException; // Exception may be thrown by A

    /**
     * The default constructor which will actually do nothing.
     */
    public ThreadSeed() {
    }

    /**
     * This method is implemented according to the runnable
     * interface and is called by the runtime system to
     * start the thread.
     */
}

```



```

        // hasn't occurred yet.
    }
    ResultArray[i] |=
    (bitmask * (Counter & 1)); // set bit
    bitmask <<=1;           // shift bitmask
    }
}

if (fewMoreBits!=0) { // create leftover bits in
    // the same way as above
    bitmask=1;
    ResultArray[fullBytes] = 0;

    for (j=0;j<fewMoreBits;j++) {
Counter =0;
stopCountThread = false;
CountThread = new Thread(this);
CountThread.start();

try {
    MyThread.sleep(SLEEP_TIME);
    stopCountThread = true;
    CountThread.join();
}
catch (InterruptedException ie) {
    ie.printStackTrace();
    }
    ResultArray[fullBytes] |= (bitmask * (Counter & 1));
    bitmask <<= 2;
    }
}
return ResultArray;
}
} // Ende der Klasse ECPrng

```

Anhang B

C++ Quellcode

B.1 impl.h

```
/////////////////////////////////////////////////////////////////
//
// ECPrng_impl.h: interface for the ECPrng_impl class.
//
/////////////////////////////////////////////////////////////////

#ifndef _ECPRNG_IMPL
#define _ECPRNG_IMPL

#include <LiDIA/bigint.h>
#include <LiDIA/galois_field.h>
#include <LiDIA/gf_element.h>
#include <LiDIA/elliptic_curve.h>
#include <LiDIA/point.h>

enum curves {      // aliases for the curve, we're on
    primary = 0,   // this is to make things clear
    twisted = 1
};

/*
 * definition of the class, holding the elliptic curve pseudorandom
 * bit generator.
 */
class ECPrng_impl
{
    ///////////////////////////////////////////////////////////////////
    // Construction/Destruction
    ///////////////////////////////////////////////////////////////////

public:

    ECPrng_impl();
    virtual ~ECPrng_impl();

    ///////////////////////////////////////////////////////////////////
    // public methods
    ///////////////////////////////////////////////////////////////////

public:
    void generate_bytes(unsigned char * bytes,
        unsigned int count);

    int get_seed_estimate();
};
```

```

void set_seed(const signed char *seed_bytes,
const unsigned int len);

/////////////////////////////////////////////////////////////////
// public constants
/////////////////////////////////////////////////////////////////

const int Strength;      // lower bound bitsize of the large prime
const int CofactorBound; // upper bound on bitsize of cofactor

/////////////////////////////////////////////////////////////////
// private methods
/////////////////////////////////////////////////////////////////

private:

void fast_multiplication( point <gf_element> &P,
const bigint &e,
curves which) const;

bigint chi (point <gf_element> &P);

int sgn(const bigint &y) const;

void generate_curve();

void find_maximum_order(curves which);

void compute_second_generator(curves which);

/////////////////////////////////////////////////////////////////
// private member variables
/////////////////////////////////////////////////////////////////

private:

// status

bool curves_are_generated;
bigint state; // generators state
curves act_curve; // curve, we're on

// data of the two curves

galois_field K; // primefield

elliptic_curve <gf_element> Curve[2]; // primary and twisted curve

bigint QuadraticNonresidue;

point <gf_element> Generators[2][2];
bigint n1[2]; // maximum order (G_1)
bigint n2[2]; // order of G_2
bigint PrimeDevisor[2]; // big prime
bigint Cofactor[2]; // cofactor
bigint GroupOrder[2]; // order of curve and
rational_factorization GroupOrder_rf[2]; // its factorization

// other data

unsigned char *SeedBytes;
unsigned int SeedLength;
bigint half_of_K;
point <gf_element> *FastMultiplicationTable[2][2];

```

```
};
#endif
```

B.2 types.h

```
#ifndef _CDC_TYPES_H
#define _CDC_TYPES_H

#include <jni.h>

// Umwandlung des nativen Zeigers auf Java long

union jPointer {
    jlong l;
    void * p;
};

#endif
```

B.3 ecprng_native.cpp

```
////////////////////////////////////
//
// ECPrng_native.cpp
//
// This class constitutes an interface to the java side. In
// here all functions are implemented, according to the
// header 'native.h', generated automatically.
// Furthermore here is the code, initializing a dynamic
// library. For a description of most of the functions, see
// the corresponding java-source 'ECPrng.java'.
//
////////////////////////////////////

#include <jni.h>
#include <ecprng/native.h>
#include <ecprng/types.h>
#include <ecprng/impl.h>

JNIEXPORT jlong JNICALL Java_cdc_ECPrng_attacheNativeObject
( JNIEnv *env, jobject obj ) {

    jPointer myPtr;

    myPtr.p = (void *) new ECPrng_impl();

    return myPtr.l;
}

JNIEXPORT void JNICALL Java_cdc_ECPrng_releaseNativeObject
( JNIEnv *env, jobject obj, jlong ptr ) {

    jPointer myPtr;
```

```

myPtr.l=ptr;

if ( myPtr.p != NULL ) {
    delete (ECPrng_impl *) myPtr.p;
    myPtr.p = NULL;
}
}

JNIEXPORT jbyteArray JNICALL Java_cdc_ECPrng_LiDIA_1NextBytes
( JNIEnv *env, jobject obj, jlong ptr, jint count ) {

    jPointer jptr;

    unsigned char isCopy;
    unsigned char *bytes;

    jptr.l = ptr;

    jbyteArray myjba = env->NewByteArray(count);

    bytes = (unsigned char *) env->
        GetPrimitiveArrayCritical( myjba, &isCopy);
    ((ECPrng_impl *) ( jptr.p ))->
        generate_bytes( bytes, count );

    env->ReleasePrimitiveArrayCritical( myjba, bytes, 0 );

    return myjba;
}

JNIEXPORT jint JNICALL Java_cdc_ECPrng_EstimateSeed
( JNIEnv *env, jobject obj, jlong ptr ) {

    jPointer jptr;
    jptr.l = ptr;

    return ((ECPrng_impl *) ( jptr.p ))->get_seed_estimate();
}

JNIEXPORT void JNICALL Java_cdc_ECPrng_SetSeed
( JNIEnv *env, jobject obj, jlong ptr, jbyteArray seed_array ) {

    jPointer jptr;
    jptr.l = ptr;

    jsize len = env->GetArrayLength( seed_array );

    signed char *tmp = env->GetByteArrayElements( seed_array, 0 );

    ((ECPrng_impl *) (jptr.p))->set_seed( tmp, len );

    env->ReleaseByteArrayElements( seed_array, tmp, 0 );
}

```

```

unsigned char __cdecl DllMain
( void * hModule,
  unsigned int ul_reason_for_call,
  void * lpReserved ) {
    return TRUE;
}

```

B.4 ecprng_impl.cpp

```

////////////////////////////////////////////////////////////////
//
// ECPrng_impl.cpp: implementation of the ECPrng_impl class.
//
// the generator itself.
//
////////////////////////////////////////////////////////////////

#include <ecprng/impl.h>

////////////////////////////////////////////////////////////////
// Construction/Destruction
////////////////////////////////////////////////////////////////

/*
 * default constructor
 *
 * marks certain variables as uninitialized
 */
ECPrng_impl::ECPrng_impl():
    Strength(160), CofactorBound(16)
{
    curves_are_generated = false;
    SeedBytes = NULL;

    n1[primary] = 0;
    n1[twisted] = 0;
    FastMultiplicationTable[0][0] = NULL;
    FastMultiplicationTable[0][1] = NULL;
    FastMultiplicationTable[1][0] = NULL;
    FastMultiplicationTable[1][1] = NULL;
}

/*
 * destructor
 *
 * cleans up the structures used
 */
ECPrng_impl::~ECPrng_impl()
{
    if ( FastMultiplicationTable[0][0] != NULL )
        delete[] FastMultiplicationTable[0][0];
    if ( FastMultiplicationTable[0][1] != NULL )
        delete[] FastMultiplicationTable[0][1];
    if ( FastMultiplicationTable[1][0] != NULL )

```



```

// binary representation
    if ( (tmp_state % n1[primary]) >= // evaluate most
        (n1[primary]/2)) {          // significant bits
bytes[i] |=j;                        // set bit
    }
    tmp_state = tmp_state<<1;
}
}
else {
act_curve = twisted;                // we are on the twisted curve
tmp_state = (state - GroupOrder[primary]);
for ( j=1; j <= 128; j<<=1 ) {     // loop over the bits
// of the byte by
// shifting through
// its binary
// representation

    if ((tmp_state % n1[twisted]) >= // evaluate most
        (n1[twisted]/2 )){          // significant bits
bytes[i] |=j;                        // set bit
    }
    tmp_state = tmp_state<<1;
}
} // end of main loop
}

/*
 * method get_seed_estimate
 *
 * Returns the amount of seed bits needed for setting up
 * LiDIA's RNG and the initial state of the ECPRNG.
 *
 * result: number of seed bytes needed
 */
int ECPrng_impl::get_seed_estimate() {
// Let's see, what is needed:
// - 8 Bytes for seed LiDIA's RNG (actually LiDIA takes a
//   biginit to seed the generator, but it takes only the
//   lower 8 bytes).
// - about  $\lfloor 2 \cdot p + 2 \rfloor / 8$  Bytes are needed to generate a starting
//   point on the primary curve or its twist. p has bitsize
//   less than Strength+CofactorBound.

return ( 8 + (Strength + CofactorBound) / 4);
}

/*
 * method set_seed
 *
 * Stores the seed provided to the generator in an array
 * for later use. If 'seed_bytes' contains fewer bytes
 * than need, the seed is padded using uninitialized bytes.
 *
 * params: seed_bytes: the seed
 *         len: length of the seed
 */

```

```

void ECPrng_impl::set_seed(const signed char *seed_bytes,
    unsigned int len) {

    if ( SeedBytes != NULL )           // if there is already one,
        delete[] SeedBytes;           // discard it

    SeedBytes = new unsigned char[len]; // allocate new array
    SeedLength = len;

    if ( seed_bytes != NULL )         // copy, if there is
        memcpy( SeedBytes, seed_bytes, len ); // something to copy
}

////////////////////////////////////
// private methods
////////////////////////////////////

/*
 * method chi
 *
 * computes the mapping from points P into the set of integers
 * {0,...,2p+1}
 *
 * params: the point P to map
 *
 * result: the integer corresponding to P
 */
bigint ECPrng_impl::chi(point <gf_element> &P) {

    bigint x,y; // coordinates of the point
    bigint result; // temporary bigint

    if (act_curve == primary) { // test, which curve we're on

        if (P.is_zero()) {
            result = (K.characteristic()<<1);
        }
        else {
            x = P.get_x().lift_to_Z();
            if (P.get_y().is_zero()) {
result = x<<1;
            }
            else {
y = P.get_y().lift_to_Z();
result = ( x<<1 ) + sgn( y );
            }
        }
    }
    else { // the same on the twisted curve

        if (P.is_zero()) {
            result = (K.characteristic()<<1) + 1;
        }
        else {
            x = (P.get_x() / QuadraticNonresidue).lift_to_Z();
            if (P.get_y().is_zero()) {
result = (( x<<1 ) + 1 );
            }
        }
    }
}

```

```

        }
        else {
y = P.get_y().lift_to_Z();
result = ( x<<1 ) + sgn(y);
        }
    }
}
return result;
}

/*
 * method sgn
 *
 * evaluates the sign-bit of the y-coordinate, which is needed
 * to calculate y, knowing the x-coordinate
 *
 * params: y the coordinate
 *
 * result: the sign as an int, having 0 or 1
 */
int ECPrng_impl::sgn(const bigint &y ) const
{
    if (y <= half_of_K)
        return 0;
    else
        return 1;
}

/*
 * method fast_multiplication
 *
 * Uses tabled points to calculate discrete exponentiation in
 * a more efficient way.
 *
 * parameter: P:    reference of point, to store the result in
 *               e:    the exponent
 *               which: the curve we're on
 */
void ECPrng_impl::fast_multiplication(point <gf_element> &P,
    const bigint &e,
    curves which) const
{
    bigint e1,e2; //temporary variables
    int j;
    int len;

    if (which==primary) {
        e1 = e % n1[primary]; // splitting the exponent according
        e2 = e / n1[primary]; // to the generation pair
    }
    else {
        e2 = (e - GroupOrder[primary]); // temporary calculation
        e1 = e2 % n1[twisted]; // splitting the exponent according
        e2 = e2 / n1[twisted]; // to the generation pair
    }

    P.assign_zero(Curve[which]); // initialize P

```

```

len = e1.bit_length();          // first part (G_1)
for ( j=0; j<len;j++ )
    if (e1.bit(j))
        P+= FastMultiplicationTable[which][0][j];

len = e2.bit_length();          // second part (G_2)
for ( j=0;j<len;j++ )
    if (e2.bit(j))
        P+= FastMultiplicationTable[which][1][j];
}

```

B.5 generate_curve.cpp

```

/////////////////////////////////////////////////////////////////
//
// generate_curve.cpp: implementation of the ECPrng_impl
// class.
//
// improved multiplication scalar * point, using the fact,
// that the point is always the same (the generator).
//
/////////////////////////////////////////////////////////////////

#include <LiDIA\gec\gec_complex_multiplication.h>
#include <LiDIA\rational_factorization.h>
#include <ecprng\impl.h>

/*
 * method generateCurve
 *
 * This method sets up a cryptographic strong curve together
 * with its twist, which is cryptographic strong, too, using
 * the complex multiplication method.
 * Secondly, it determines the group structure and a
 * generating pair for either curve.
 *
 */
void ECPrng_impl::generate_curve()
{
    // seeding LiDIA:

    union {
        // interpret the first 8 bytes of
        unsigned char as_array[8]; // the generator's seed and ...
        long as_long;
    } tmp_seed;
    memcpy ( tmp_seed.as_array, SeedBytes, 8 );
    seed ( tmp_seed.as_long ); // supply it to the generator

    // finding the two curves, using the gec-package:

    gec_complex_multiplication *my_gec =
        new gec_complex_multiplication(); // instantiate the
        // proper gec
    my_gec->set_upper_bound_k((1<<CofactorBound)-1); // and supply

```

```

my_gec->set_lower_bound_bitlength_r(Strength); // it with
// the bounds

my_gec->generate(); // generate the curve

// reading parameters from the gec

K = galois_field(my_gec->get_q()); // Primefield
half_of_K = K.characteristic()>>1; // used during calculations

QuadraticNonresidue = // quadratic nonresidue,
(my_gec->get_q_nonres()).lift_to_Z(); // which sets up the
// twist

Curve[primary].set_coefficients(
(my_gec->get_a4()), // primary curve
(my_gec->get_a6()));
PrimeDevisor[primary] = my_gec->get_r(); // big prime divisor
Cofactor[primary] = my_gec->get_k(); // times cofactor
GroupOrder[0] = PrimeDevisor[0]*Cofactor[0]; // is group order

Curve[1].set_coefficients(my_gec->get_at4(), // twisted curve
my_gec->get_at6());
PrimeDevisor[1] = my_gec->get_rt(); // big prime divisor
Cofactor[1] = my_gec->get_kt(); // times cofactor
GroupOrder[1] = PrimeDevisor[1]*Cofactor[1]; // is group order

// finding an element of maximum order in the primary curve

find_maximum_order(primary);

// complete generator pair of primary curve (i.e. looking
// for G_2)

compute_second_generator(primary);

// finding an element of maximum order in the twisted curve

find_maximum_order(twisted);

// complete generator pair of twisted curve (i.e. looking
// for G_2)

compute_second_generator(twisted);

// precomputations:
// computing the points 2^i*G, for all four generators
// of the two curves.

int t,i; // temporary variables

// primary curve, first generator

t = n1[primary].bit_length();
FastMultiplicationTable[primary][0] =
new point<gf_element>(t);
FastMultiplicationTable[primary][0][0] =

```

```

        Generators[primary][0];
for (i=1;i<t;i++)
    FastMultiplicationTable[primary][0][i] =
        FastMultiplicationTable[primary][0][i-1].twice();

// primary curve, second generator

t = n2[primary].bit_length();
FastMultiplicationTable[primary][1] =
    new point<gf_element>[t];
FastMultiplicationTable[primary][1][0] =
    Generators[primary][1];
for (i=1;i<t;i++)
    FastMultiplicationTable[primary][1][i] =
        FastMultiplicationTable[primary][1][i-1].twice();

// twisted curve, first generator

t = n1[twisted].bit_length();
FastMultiplicationTable[twisted][0] =
    new point<gf_element>[t];
FastMultiplicationTable[twisted][0][0] =
    Generators[twisted][0];
for (i=1;i<t;i++)
    FastMultiplicationTable[twisted][0][i] =
        FastMultiplicationTable[twisted][0][i-1].twice();

// twisted curve, second generator

t = n2[twisted].bit_length();
FastMultiplicationTable[twisted][1] =
    new point<gf_element>[t];
FastMultiplicationTable[twisted][1][0] =
    Generators[twisted][1];
for (i=1;i<t;i++)
    FastMultiplicationTable[twisted][1][i] =
        FastMultiplicationTable[twisted][1][i-1].twice();

// Initializing state:

bigint tmp;
state = 0;
for (i = 0 ; i < SeedLength; i++) {
    tmp = SeedBytes [i];
    tmp <<= (8*i);
    state += tmp;
}
state %= (2*K.characteristic()+2);

// ready!

curves_are_generated = true;

delete my_gec; // cleaning up
}

/*

```

```

* method find_maximum_order
*
* This method tries to find an element of maximum order,
* in the curve determined by 'which'
*
* params: the curve to work on
*/
void ECPrng_impl::find_maximum_order( curves which ) {

    int trials = 100;      // bound on number of points
                          // selected at random
    bigint max_order = 0; // will contain maximum order
                          // after the main loop
    bigint akt_order;     // temporary bigint

    point <gf_element> P,Q,G;      // temporary points
    rational_factorization tmp_rf; // temporary factorization

    if ( n1[which] != 0 ) // test, if something has to be done
        return;

    // We will now factor the group order of the curve, which is
    // easy, because we already know the large prime divisor.
    // It therefore amounts to factoring the cofactor

    GroupOrder_rf[which].assign(Cofactor[which]); // factoring...
    GroupOrder_rf[which].factor();                // the cofactor

    multiply(GroupOrder_rf[which],                // adding the large
             GroupOrder_rf[which],                // prime divisor to
             PrimeDevisor[which]);                // the factorization

    tmp_rf = (GroupOrder_rf[which]                // we know, that the big
              / PrimeDevisor[which]);            // prime does divide n_1

    // main loop:

    for ( int i = 0; i < trials; i++ ) {

        P = Curve[which].random_point(); // select a random point of
        Q = PrimeDevisor[which] * P;     // order at most n_1/l

        if ( !Q.is_zero() ) {            // test if zero
            akt_order = order_point( Q,tmp_rf ); // calculate its
                                                // order

            if ( akt_order > max_order ) { // if it is bigger,
                max_order = akt_order;    // record the order
                G.assign( P );            // and the point
            }
        }
    }

    // storing results for further use

    n1[which] = max_order * PrimeDevisor[which]; // maximum order
    n2[which] = GroupOrder[which] / n1[which];   // order of G_2

```

```

    Generators[which][0].assign(G);          // G_1
}

/*
 * method compute_second_generator
 *
 * This method tries to compute the second generator of the
 * generator pair of the curve, determined by 'which'. Which
 * amounts to finding a point G_2 of order n_2 which is not
 * in the subgroup <G_1>.
 *
 * params: the curve to work on
 */
void ECPrng_impl::compute_second_generator(curves which) {

    bool found = false; // condition of main loop
    bigint order, m, dl; // temporary bigints

    point <gf_element> P,G; // temporary points

    G = (n1[which]/n2[which]) * // we will have to work in the
        Generators[which][0]; // subgroup <G> of order
                               // n_2 of <G_1>

    if (n2[which].is_one() // test if curve is cyclic
        Generators[which][1]. // in which case the generator
            assign_zero(Curve[which]); // is the point 0 at infinity

    // main loop: repeat until G_2 is found

    do {
        P = Curve[which].random_point(); // randomly select a point
        order = order_point(
            P, GroupOrder_rf[which]); // and calculate its order

        if (n2[which] == // the order has to be
            gcd(order, n2[which])) { // divisible by n_2

            m = order / n2[which]; // calculate a point
            P = m*P; // of order n_2

            dl = bg_algorithm(G,P,0,n2[which]); // calculate the
                                                // discrete logarithm
                                                // using baby-steps-
                                                // giant-steps.

            if (dl == -1) { // if no dl can be found
                Generators[which][1].assign(P); // we have the generator
            }
        }
    }
    while (!found); // end of main loop
}

```

Literaturverzeichnis

- [AMV96] P. van Oorschot A. Menezes and S. Vanstone. Crc press series on discrete mathematics and its applications, 1996.
- [Bai00] Harald Baier. Efficient construction of cryptographically strong elliptic curves, June 2000.
- [BM84] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM Journal on Computing*, 13(4):850–864, November 1984.
- [Bos99a] Siegfried Bosch. *Algebra, 3. Auflage*. Springer-Verlag, 1999.
- [Bos99b] Wieb Bosma. Signed bits and fast exponentiation, 1999.
- [Bud] Bundesamt für Sicherheit in der Informationstechnik. Geeignete kryptoalgorithmen gemäß § 17 (2) sigv.
- [Coh93] Henri Cohen. A course in computational algebraic number theory, 1993.
- [ECS94] D. Eastlake, S. Crocker, and J. Schiller. Randomness recommendations for security, 1994.
- [EEE98] Institute of Electrical and Inc Electronics Engineers. IEEE P1363/D8: Standard specifications for public key cryptography, 1998.
- [GM84] S. Goldwasser and S. Micali. *Journal of computer and system sciences*, 1984.
- [Gol95] O. Goldreich. Oded goldreich. foundations of cryptography (fragments of a book), 1995.
- [Gol99] Oded Goldreich. Pseudorandomnes, 1999.
- [Hah00] Tobias Hahn. The java interface to lidia, July 2000.
- [Ham98] Safuat Hamdy. ANNOTATIONES DE RATIONIBUS CRYPTOLOGIAE: Anwendungen elliptischer kurven in der kryptologie. Master’s thesis, Universität Hamburg, Fachbereich Informatik, November 1998.
- [Kal88] Burton S. Kaliski, Jr. *Elliptic Curves and Cryptography: A Pseudorandom Bit Generator and Other Tools*. PhD thesis, Massachusetts Institute of Technology, March 1988.

- [Len87] H. Lenstra. Factoring integers with elliptic curves, 1987.
- [LiD] LiDIA: A c++ library for computational number theory. <http://www.informatik.tu-darmstadt.de/TI/LiDIA/>.
- [mag] MAGMA version 2.7. <http://www.maths.usyd.edu.au:8000/u/magma/htmlhelp/MAGMA.html>.
- [Men93] A. Menezes. Elliptic curve cryptosystems, 1993.
- [Mül95] V. Müller. Ein algorithmus zur bestimmung der punktzahl von elliptischen kurven über endlichen primkörpern der charackteristik größer drei, 1995.
- [MOV93] A. Menezes, T. Okamoto, and S. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field, 1993.
- [Od100] A. Odlyzko. Codes and cryptography, 2000.
- [Sho97] V. Shoup. Lower bounds for discrete logarithms and related problems, 1997.
- [SS98] J. Silverman and J. Suzuki. Elliptic curve discrete logarithms and the index calculus, 1998.
- [Wei40] A. Weil. Sur les fonctions alg'ebriques 'a corps de constantes finis, 1940.
- [Yao82] A. Yao. Theory and applications of trapdoor functions, 1982.