

Signaturverfahren mit Elliptischen Kurven über  
Körpern der Charakteristik 2 und deren  
Arithmetik

Diplomarbeit von  
Oliver Seiler

nach einem Thema von Prof. Dr. J. Buchmann  
betreut durch Birgit Henhagl  
am Fachbereich Informatik,  
Fachgebiet Theoretische Informatik,  
der Technischen Universität Darmstadt



Darmstadt, den 7. Mai 2001

### **Danksagung**

Ich danke Professor Dr. J. Buchmann für die Vergabe dieser Diplomarbeit. Sie hat mir viel Spass gemacht und mich für das Themengebiet begeistern können.

Ich bedanke mich auch bei Birgit Henhagl für die Betreuung dieser Diplomarbeit. Sie hat mich stets bei allen Fragen und Problemen unterstützt und sich immer Zeit für meine Anliegen genommen.

Ich möchte mich auch bei allen Mitarbeitern am Institut für Theoretische Informatik bedanken, die mich bei der Erstellung dieser Arbeit unterstützt haben.

Besonderer Dank gilt meinen Freunden Marc Freidhof, Thilo Frotscher, Oliver Hirschberg und Christian Weyer mit denen ich zahlreiche mittägliche Diskussionen über die verwendeten Technologien führte und die immer einen guten Rat für mich hatten.

Ich möchte mich auch bei meinen Eltern bedanken, die mich in den letzten Jahren finanziell und privat unterstützt haben und so zum Gelingen meines Studiums beigetragen haben.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>5</b>
1.1	Ziele dieser Arbeit . . . . .	6
1.2	Übersicht . . . . .	6
<b>2</b>	<b>Mathematische Grundlagen und Algorithmen</b>	<b>7</b>
2.1	Die Arithmetik von Elementen endlicher Körper der Charakteristik 2 ( $GF(2^n)$ ) . . . . .	7
2.1.1	Repräsentation durch Polynombasen . . . . .	7
2.1.1.1	Addition . . . . .	8
2.1.1.2	Multiplikation . . . . .	8
2.1.1.3	Reduktion . . . . .	9
2.1.1.4	Invertierung . . . . .	11
2.1.1.5	Quadrierung . . . . .	12
2.1.1.6	Quadratwurzel . . . . .	13
2.1.1.7	Exponentiation . . . . .	14
2.1.1.8	Wurzel . . . . .	14
2.1.2	Optimale Normalbasen . . . . .	14
2.1.2.1	Berechnung des Körperpolynoms einer Normalbasis . . . . .	15
2.1.3	Basiskonvertierung . . . . .	15
2.1.3.1	Berechnung der Konvertierungsmatrix . . . . .	16
2.2	Elliptische Kurven über $GF(2^n)$ . . . . .	18
2.3	EC Domain Parameter . . . . .	19
2.4	Schlüsselerzeugung . . . . .	20
2.5	ECDSA . . . . .	20
2.6	ECNR . . . . .	21
<b>3</b>	<b>Java und JCA</b>	<b>23</b>
3.1	Die Java Cryptography Architecture . . . . .	23
3.1.1	Benutzung der JCA . . . . .	24
3.1.1.1	Erzeugen eines Schlüsselpaares und einer ECNR-Signatur . . . . .	25
3.1.1.2	Verifizieren einer ECNR-Signatur . . . . .	27
3.2	Optimierungen . . . . .	28
3.2.1	Just-In-Time Compiler . . . . .	29
3.2.2	HotSpot . . . . .	29
3.3	Verwendete Werkzeuge . . . . .	30
3.3.1	JBuilder4 . . . . .	30

3.3.2	JDK 1.3 & HotSpot Virtual Machine . . . . .	30
3.3.3	OptimizeIt . . . . .	30
3.3.4	Rational Rose . . . . .	31
<b>4</b>	<b>Entwurf</b>	<b>33</b>
4.1	cdc.ec.arithmetic.gf . . . . .	34
4.1.1	Basiskonvertierung . . . . .	36
4.2	cdc.ec.arithmetic.curves . . . . .	38
4.3	cdc.ec.ecparameters . . . . .	39
4.4	cdc.ec.ecnr . . . . .	40
4.5	cdc.ec.ecdsa . . . . .	41
4.6	cdc.ec.test . . . . .	42
<b>5</b>	<b>Laufzeiten</b>	<b>45</b>
<b>6</b>	<b>Zusammenfassung &amp; Ausblick</b>	<b>49</b>
<b>A</b>	<b>Implementierung</b>	<b>51</b>
A.1	cdc.ec.arithmetic.gf . . . . .	51
A.1.1	GF2nField . . . . .	51
A.1.2	GF2nONBField . . . . .	53
A.1.3	GF2nPolynomialField . . . . .	54
A.1.4	GF2nElement . . . . .	57
A.1.5	GF2nONBElement . . . . .	60
A.1.6	GF2nPolynomialElement . . . . .	62
A.1.7	Bitstring . . . . .	66
A.1.8	PolynomialGF2n . . . . .	71
A.2	cdc.ec.ecnr . . . . .	73
A.2.1	ECNRSignature . . . . .	73
A.2.2	ECNRKeyPairGenerator . . . . .	75
A.2.3	ECNRPrivateKey . . . . .	75
A.2.4	ECNRPublicKey . . . . .	77
A.3	cdc.ec.ecdsa . . . . .	78
A.3.1	ECDSASignature . . . . .	78
A.3.2	ECDSAKeyPairGenerator . . . . .	79
A.3.3	ECDSAPrivateKey . . . . .	80
A.3.4	ECDSAPublicKey . . . . .	81
A.4	cdc.ec.test . . . . .	82
A.4.1	Test . . . . .	82
A.4.2	TestGF2nField . . . . .	83
<b>B</b>	<b>Glossar</b>	<b>85</b>

# Kapitel 1

## Einleitung

Public Key Kryptographie gewinnt durch die Verbreitung des Internets, leistungsfähige Personal Digital Assistents (PDAs) und Mobiltelefone und ein gesteigertes Sicherheitsbewusstsein mehr und mehr an Bedeutung. Das Signaturgesetz stellt digitale Signaturen mit herkömmlichen Unterschriften gleich, und Zertifikate sind allgemein erhältlich. Die zugrundeliegenden Algorithmen sind jedoch sehr rechenintensiv, und mit steigender Schlüssellänge steigt der Rechenbedarf und erreicht für mobile Geräte oder Server, die viele Clients bedienen müssen (Webserver, Zertifizierungsserver in Public Key Infrastrukturen), nicht akzeptierbare Bereiche.

Kryptographische Verfahren basierend auf Elliptischen Kurven bieten schon bei deutlich kürzeren Schlüssellängen die gleiche Sicherheit wie herkömmliche Verfahren. So entspricht ein vermutlich schon bald nicht mehr ausreichenden Schutz bietender 1024-Bit RSA Schlüssel zur Zeit etwa einem 160-Bit EC Schlüssel.<sup>1</sup>

Digitale Unterschriften garantieren die Authentizität (Echtheit) und Integrität (Unveränderbarkeit) eines Dokumentes, sowie die Identität des Unterzeichners und das “auf einen Blick” und bei Bedarf über große Entfernungen, zum Beispiel in Netzwerken.

Das deutsche Signaturgesetz (SigG) stellt digitale Unterschriften normalen, herkömmlichen Unterschriften gleich. Es schafft somit rechtliche Klarheit, ermöglicht den weitverbreiteten Einsatz digitaler Signaturen und bietet so die Grundlage für e-Commerce Anwendungen, elektronisches Geld (e-Cash, e-Payment), digitale Ämter und elektronische Wahlen (e-Voting).

Herkömmliche Public Key Verfahren, wie zum Beispiel DSA, die auf der Schwierigkeit des Lösens diskreter Algorithmen über endlichen Gruppen basieren, können auf Punktgruppen elliptischer Kurven über endlichen Körpern übertragen werden und stehen somit mit kleinen Änderungen als Elliptische Kurven-Version zur Verfügung (ECDSA).

---

<sup>1</sup>Genau wie beim Faktorisieren bzw. Lösen diskreter Logarithmen wird es vermutlich auch im Bereich der Elliptischen Kurven in Zukunft bessere Algorithmen geben, die die Sicherheit beeinträchtigen.

## 1.1 Ziele dieser Arbeit

Diese Arbeit befasst sich mit der Implementierung einer schnellen Basisarithmetik für Körper der Charakteristik 2 ( $GF(2^n)$ ), insbesondere in Polynomdarstellung, einer Basiskonvertierung zwischen unterschiedlichen Polynombasen und Optimalen Normalbasen und dem digitalen Signaturverfahren von Nyberg und Rueppel (ECNR).

Besonderer Wert wurde auf Portabilität und Kompatibilität zu anderen Systemen, sowie auf möglichst gute Performance gelegt. Die Implementierung erfolgte in Java, um maximale Portabilität zu garantieren, und konform zum zukünftigen IEEE Standard P1363, um Kompatibilität mit anderen Implementierungen zu gewährleisten. Die zu dieser Arbeit gehörende Implementierung ist Teil des CDCECProviders des Instituts für Theoretische Informatik der Technischen Universität Darmstadt, der weitreichende kryptographische Verfahren für die Java Cryptography Architecture bereitstellt. Sie kann dadurch sehr flexibel in bestehende und neue Anwendungen integriert werden. Anregungen zur Basisarithmetik für  $GF(2^n)$  in Polynomdarstellung wurden der am Institut entwickelten C++ Bibliothek LiDIA entnommen.

## 1.2 Übersicht

Kapitel 2 *Mathematische Grundlagen und Algorithmen* gibt eine Einleitung in die Arithmetik endlicher Körper der Charakteristik 2 ( $GF(2^n)$ ) und beschreibt die verschiedenen Algorithmen zur Basisarithmetik und Basiskonvertierung, die EC Domain Parameter, sowie die Algorithmen zum Signieren und Verifizieren mit den Verfahren ECDSA und dem Verfahren von Nyberg und Rueppel.

Kapitel 3 *Java und JCA* beschreibt die Architektur und Benutzung der Java Cryptography Architecture, in die die Implementierung dieser Arbeit integriert ist. Die Benutzung des Signaturverfahrens von Nyberg und Rueppel mit der JCA wird anhand von Sequenzdiagrammen erläutert. Außerdem wird auf Java-spezifische Aspekte und Laufzeitoptimierungen unter Java sowie die zur Implementierung verwendeten Werkzeuge eingegangen.

In Kapitel 4 *Entwurf* wird das Design der Implementierung dargestellt und auf die einzelnen Packages und Klassen eingegangen. Eine genauere Beschreibung der einzelnen Klassen und Methoden findet sich im Anhang A.

Kapitel 5 *Laufzeiten* befasst sich mit der Performance dieser Implementierung und vergleicht die gemessenen Laufzeiten mit denen, der in C++ geschriebenen Implementierung von LiDIA.

Kapitel 6 *Zusammenfassung & Ausblick* fasst die Ergebnisse dieser Arbeit noch einmal zusammen und gibt einen Ausblick auf zukünftige, nicht in dieser Arbeit berücksichtigte, Aspekte.

Anhang A *Implementierung* gibt eine detaillierte Beschreibung der einzelnen Klassen und zugehörigen Methoden.

Darmstadt, 7.5.2001

Oliver Seiler

## Kapitel 2

# Mathematische Grundlagen und Algorithmen

### 2.1 Die Arithmetik von Elementen endlicher Körper der Charakteristik 2 ( $GF(2^n)$ )

Die Elemente von endlichen Körpern der Charakteristik 2,  $GF(2^n)$ , lassen sich auf zwei unterschiedliche Arten darstellen: als Polynome vom Grad  $n - 1$  oder über Optimalen Normalbasen (ONB). Der Schwerpunkt dieser Arbeit liegt auf der Polynomdarstellung, deren Arithmetik in diesem Kapitel dargestellt wird, und der Basiskonvertierung in unterschiedliche Polynom- oder Optimale Normalbasen.

#### 2.1.1 Repräsentation durch Polynombasen

Jedes Element aus  $GF(2^n)$  lässt sich durch ein binäres Polynom vom Grade kleiner gleich  $(n - 1)$  mit dazugehörigem irreduziblem Körperpolynom  $p_n(x)$  vom Grad  $n$  darstellen:

$$a_{n-1}x^{n-1} + \dots + a_2x^2 + a_1x + a_0$$

Das Körperpolynom  $p(x)$  dient zur Reduzierung der Elemente, zum Beispiel nach einer Multiplikation, und muss irreduzibel sein. Irreduzibel heißt ein Polynom, wenn es nicht das Produkt zweier Polynome kleineren Grades sein kann. Für jeden Körpergrad existiert ein irreduzibles Polynom und ungefähr jedes  $n$ -te Polynom vom Grad  $n$  ist irreduzibel [P1363, A.3.4, S91]. Die Reduktion modulo  $p(x)$  ist besonders effizient, wenn  $p(x)$  möglichst wenige Koeffizienten gleich 1 hat.

Die irreduziblen Polynome mit den wenigsten Koeffizienten sind Trinome  $x^n + x^r + 1$ ,  $n > r > 0$ , die aber nicht für alle Körpergrade existieren. Wenn kein Trinom existiert, sollte man ein Pentanom  $x^n + x^r + x^s + x^t + 1$ ,  $n > r > s > t > 0$  verwenden. Zumindest für jeden Körpergrad bis 3000 existiert ein irreduzibles Tri- oder Pentanom.

Die Arithmetik ergibt sich aus den normalen Regeln zum Rechnen mit Polynomen modulo dem Körperpolynom  $p_n(x)$ , wobei die Koeffizienten modulo 2 gerechnet werden.

Zur Eindeutigen Identifikation des Polynoms genügt die Speicherung der Koeffizienten :

$$(a_{n-1}, a_{n-2}, \dots, a_2, a_1, a_0)$$

Da die Koeffizienten modulo 2 gerechnet werden, können sie einfach in einem Bitstring der Länge  $n$  gespeichert werden. Ein Element aus  $GF(2^{160})$  benötigt also 5 (32-Bit) Integers oder 20 Bytes Speicherplatz.

### 2.1.1.1 Addition

Da in den Koeffizienten der Polynome modulo 2 gerechnet wird, ergibt sich die Addition von zwei Polynomen durch ein bitweises XOR der Koeffizienten:

$$(a_{n-1}, \dots, a_1, a_0) + (b_{n-1}, \dots, b_1, b_0) = (a_{n-1} \oplus b_{n-1}, \dots, a_1 \oplus b_1, a_0 \oplus b_0)$$

Der Grad des Ergebnisses ist dabei kleiner gleich dem größten Grad der Summanden, es muss also nicht reduziert werden.

**Beispiel:**

$$\begin{array}{r} x^3 + \phantom{x^2} + \phantom{x} + \phantom{1} \\ + \phantom{x^3} + x^2 + x + 1 \\ \hline x^3 + x^2 + \phantom{x} + \phantom{1} \end{array}$$

Beim Speichern der Koeffizienten als Bitstrings, also Arrays aus Integern, besteht die Addition aus einem bitweisen XOR der einzelnen Arrayelemente und ist dadurch sehr schnell. Die Laufzeit beträgt  $O(n)$ .

### 2.1.1.2 Multiplikation

Zur Multiplikation lässt sich einfach die Schulmethode zur Multiplikation zweier Polynome anwenden, wobei bei der Addition wieder modulo 2 zu rechnen ist und das Ergebnis modulo des Körperpolynoms  $p_n(x)$  reduziert werden muss. Das Restpolynom der Reduktion ist vom Grad wieder kleiner gleich  $(n - 1)$ :

$$(a_{n-1}, \dots, a_1, a_0) \cdot (b_{n-1}, \dots, b_1, b_0) = (c_{n-1}, \dots, c_1, c_0)$$

wobei

$$\sum_{k=0}^{n-1} c_k x^k \equiv \left( \sum_{i=0}^{n-1} a_i x^i \right) \cdot \left( \sum_{j=0}^{n-1} b_j x^j \right) \text{ mod } p_n(x)$$

Dies kostet  $O(n^2)$  Bitoperationen plus die Kosten der anschließenden Reduktion.

**Beispiel:**  $(x^3 + x) \cdot (x^2 + x + 1) = x^5 + x^4 + x^2 + x$

$$\begin{array}{r} x^3 \cdot (x^2 + x + 1) = x^5 + x^4 + x^3 \\ + x \cdot (x^2 + x + 1) = \phantom{x^5} + \phantom{x^4} + x^3 + x^2 + x \\ \hline x^5 + x^4 + \phantom{x^3} + x^2 + x \end{array}$$

Auf dem Rechner entspricht die Multiplikation einem Linksshift des ersten Bitstrings um  $i$  für jedes im zweiten Bitstring gesetzte Bit an der Stelle  $i$ . Das Ergebnis ist das bitweise XOR aller geschifteten Bitstrings reduziert modulo des

Körperpolynoms  $p_n(x)$ . Da das Shiften auf einem Rechner relativ lange dauert, und sich nicht nur auf ein Register beschränkt, ist diese Methode langsam.

Deutlich schneller ist die sogenannte Karatsuba- (oder Karatzuba-) Multiplikation (siehe [Knu81, S279ff], [Han]). Sie multipliziert zwei Zahlen  $u, v$  der Länge  $2n$ .  $u$  und  $v$  werden jeweils in zwei Zahlen der Länge  $n$  aufgeteilt.  $u_1, v_1$  enthalten die höherwertigen  $n$  Bits von  $u, v$  und  $u_0, v_0$  die niederwertigen  $n$  Bits:

$$u = 2^n u_1 + u_0, v = 2^n v_1 + v_0$$

Die Rekursion beruht auf der folgenden Gleichung:

$$uv = (2^{2n} + 2^n)u_1v_1 + 2^n(u_1 - u_0)(v_1 - v_0) + (2^n + 1)u_0v_0$$

oder

$$uv = 2^{2n}u_1v_1 + 2^n[u_1v_1 + (u_1 - u_0)(v_1 - v_0) + u_0v_0] + u_0v_0$$

Um zwei Zahlen der Länge  $2n$  Bit zu multiplizieren, führt man also 3 Multiplikationen der Länge  $n$ , 4 Additionen, 2 Subtraktionen und zwei Linksshifts durch.

Diese Methode lässt sich auch zum Multiplizieren von Polynomen verwenden. Der Rechenaufwand für die Rekursion, das Shiften und die Additionen ist günstiger als eine Multiplikation der Länge  $2n$  nach der Schulmethode.

$$uv = u_1v_1 \lll 2n \oplus [u_1v_1 \oplus u_0v_0 \oplus (u_1 \oplus u_0)(v_1 \oplus v_0)] \lll n \oplus u_0v_0$$

Sobald in der Rekursion  $n$  einen bestimmten Wert unterschreitet, (beispielsweise  $n \leq 16$  auf 32-Bit Maschinen) und so das Ergebnis zweier Teilmultiplikationen in ein Register passt, wird letztendlich eine normale Multiplikation nach der Schulmethode durchgeführt.

### Beispiel

Sei  $n = 2$ ,  $u = x^3 + x^2 + 1 = (1101)$ ,  $v = x^3 + 1 = 1001$

Dann ist  $u_0 = (01)$ ,  $u_1 = (11)$  und  $v_0 = (01)$ ,  $v_1 = (10)$ .

$$\begin{aligned} uv &= (11)(10) \lll 4 \oplus [(11)(10) \oplus (01)(01) \oplus ((11) \oplus (01))((10) \oplus (01))] \lll 2 \oplus (01)(01) \\ &= (0110) \lll 4 \oplus [(0110) \oplus (0001) \oplus (10)(11)] \lll 2 \oplus (01) \\ &= (01100000) \oplus [(0111) \oplus (0110)] \lll 2 \oplus (01) \\ &= (01100000) \oplus [0001] \lll 2 \oplus (01) \\ &= (01100000) \oplus (000100) \oplus (01) \\ &= (01100101) \end{aligned}$$

#### 2.1.1.3 Reduktion

Nach Multiplikationen (und Quadrierungen) muss das Ergebnis gegebenenfalls reduziert werden, weil der Grad des Polynoms sonst den Körpergrad überschreiten würde. Die Reduktion ist das Berechnen des Restes der Division durch das Körperpolynom. Die Division mit Rest erfolgt wie auch die Multiplikation nach der Schulmethode zum Dividieren von Polynomen:

$$\begin{array}{r} x^3 + \phantom{x} \phantom{+} \phantom{1} \phantom{:} \phantom{(x^2 + 1)} = x \text{ Rest } (x + 1) \\ x^3 + x \phantom{+} \phantom{1} \phantom{:} \phantom{(x^2 + 1)} \\ \hline x + 1 \end{array}$$

Diese Methode funktioniert für alle Körperpolynome.

Deutlich schneller ist die Modulare Reduktion für Polynome vom Grad kleiner gleich  $(2n - 2)$  und einem Tri- bzw. Pentanom als Körperpolynom [Sch95, 4.3][Han, Algorithm 5, 6].

Wird das Trinom  $x^n + x^r + 1$  zur Reduktion benutzt beruht sie auf der Kongruenz:

$$x^n \equiv x^r + 1 \pmod{p(x)}$$

bzw. für Koeffizienten  $i > n$ :

$$x^i \equiv x^{i-(n-r)} + x^{i-n} \pmod{p(x)}$$

Bei Benutzung eines Pentanoms  $x^n + x^r + x^s + x^t + 1$  gilt:

$$x^n \equiv x^r + x^s + x^t + 1 \pmod{p(x)}$$

bzw. für Koeffizienten  $i > n$ :

$$x^i \equiv x^{i-(n-r)} + x^{i-(n-s)} + x^{i-(n-t)} + x^{i-n} \pmod{p}$$

**Beispiel:**

Sei  $n = 4$ ,  $p(x) = x^4 + x + 1$

Das nichtreduzierte Ergebnis der Multiplikation  $(x^3 + x) * (x^3 + x^2 + 1)$  ist gleich  $x^6 + x^5 + x^4 + x$ , als Bitstring dargestellt 1110010. Das Ergebnis kann jetzt bitweise reduziert werden. Zur Elimination des führenden Bits dient die Kongruenz  $x^6 \equiv x^3 + x^2$  (1001100):

$$\begin{array}{r} 1110010 \\ + 1001100 \\ \hline 0111110 \end{array}$$

Nun kann das nächste Bit durch die Kongruenz  $x^5 \equiv x^2 + x$  (100110) eliminiert werden:

$$\begin{array}{r} 111110 \\ + 100110 \\ \hline 011000 \end{array}$$

Und schließlich  $x^4 \equiv x + 1$ (10011):

$$\begin{array}{r} 11000 \\ + 10011 \\ \hline 01011 \end{array}$$

Das Ergebnis der Reduktion ist also das Polynom  $x^3 + x + 1$  (1011). Das Verfahren funktioniert natürlich prinzipiell genauso wie die normale Resteberechnung bei der Division, man kann die Reduktion aber auch gleich wortweise durchführen. Das ist bei eng zusammenliegenden Koeffizienten im Körperpolynom besonders effizient. Bei einem Trinom  $p(x) = x^n + x^r + 1$  als Körperpolynom nimmt man dazu den Bitstring mit allen Koeffizienten  $i..n$ , verschiebt diesen sowohl um  $i - (n - r)$  als auch um  $n$  Stellen nach rechts und addiert ihn wieder zum zu reduzierenden Bitstring, solange bis alle Koeffizienten  $\geq n$  0 enthalten. Bei Pentanomen verfährt man entsprechend.



- 3.4.1  $m = m^2 a$
- 3.4.2  $k = k + 1$
- 4. Ausgabe  $m^2$

**Erweiterter Euklidischer Algorithmus:** Mit dem leicht modifizierten Erweiterten Euklidischen Algorithmus lässt sich direkt und am schnellsten das Inverse finden [P1363, A.4.4, S98f][Han, Algorithm 8]:

Invertieren durch Erweiterten Euklidischen Algorithmus  
 EINGABE:  $a \in GF(2^n)$ ,  $a \neq 0$ , Körperpolynom  $p(x)$   
 AUSGABE:  $a^{-1}$

1.  $b = 1$ ,  $c = 0$ ,  $u = a$ ,  $v = p(x)$
2. while  $deg(u) \neq 0$  do
  - 2.1  $j = deg(u) - deg(v)$
  - 2.2 if  $j < 0$  then
    - 2.2.1  $u \leftrightarrow v$ ,  $b \leftrightarrow c$ ,  $j = -j$
  - 2.3  $u = u + (v \ll j)$
  - 2.4  $b = b + (c \ll j)$
3. Ausgabe  $b$

**Modified Almost Inverse Algorithm:** Dieser Algorithmus beruht auf dem Almost Inverse Algorithm [Sch95, 4.4][Han, Algorithm 9], der für ein gegebenes Element  $a$  ein Element  $b$  und  $k$  mit  $ab \equiv x^k \pmod{p(x)}$  berechnet. Um  $a^{-1}$  zu erhalten muss man  $b$  noch durch  $x^k$  teilen.

Der Modified Almost Inverse Algorithm [Han, Algorithm 10] berechnet direkt das Inverse  $a^{-1}$  und ist etwas langsamer als der Erweiterte Euklidische Algorithmus:

Modified Almost Inverse Algorithm  
 EINGABE:  $a \in GF(2^n)$ ,  $a \neq 0$ , Körperpolynom  $p(x)$   
 AUSGABE:  $a^{-1}$

1.  $b = 1$ ,  $c = 0$ ,  $u = a$ ,  $v = p(x)$
2. while  $x \mid u$  do
  - 2.1  $u = u/x$  ( $u = u \gg 1$ )
  - 2.2 if  $x \mid b$  then
    - 2.2.1  $b = b/x$  ( $b = b \gg 1$ )
  - 2.3 else
    - 2.3.1  $b = (b + p(x)) \gg 1$
3. if  $u = 1$  then
  - 3.1 Ausgabe  $b$
4. if  $deg(u) < deg(v)$  then
  - 4.1  $u \leftrightarrow v$ ,  $b \leftrightarrow c$
5.  $u = u + v$ ,  $b = b + c$
6. Gehe zu 2.

### 2.1.1.5 Quadrierung

Beim Quadrieren eines Polynoms werden die Koeffizienten vom Index  $i$  nach  $2i$  verschoben, alle anderen Koeffizienten bleiben 0:

$$(a_{n-1}x^{n-1} + \dots + a_2x^2 + a_1x + a_0)^2 = a_{n-1}x^{2(n-1)} + \dots + a_2x^4 + a_1x^2 + a_0 \pmod{p(x)}$$

In der Repräsentation als Bitstring werden die Bits aufgespreizt und die Lücken mit Nullen aufgefüllt:

$$(1011)^2 = 1000101$$

Das Ergebnis muss natürlich noch modulo des Körperpolynoms reduziert werden.

Werden Tri- oder Pentanome zur Reduktion benutzt, und ist die Reduktion relativ schnell, benutzt man eine vorberechnete Tabelle, die für jedes mögliche Byte (oder den entsprechenden Datentypen, der zur Speicherung der Bitstrings benutzt wird) die zwei resultierenden Bytes enthält ( $1011 \rightarrow 0100\ 0101$ ) und reduziert anschließend. Dies ist die einfachste und zugleich schnellste Methode.

Falls die Reduktion nur langsam implementiert ist, oder keine Tri- oder Pentanome zur Reduktion benutzt werden, kann eine dem Körperpolynom entsprechende, vorberechnete Squaring-Matrix  $S$  zur Quadrierung benutzt werden. Die Matrix hat die Dimension  $n \times n$  und enthält Elemente aus  $\{0, 1\}$ . Das Ergebnis der Quadrierung erhält man aus dem Produkt:

$$(a_{n-1} \dots a_1 a_0)^2 = (a_{n-1} \dots a_1 a_0)S$$

Der so erhaltene Bitstring muss nicht mehr reduziert werden. Die Squaring-Matrix berechnet sich wie folgt [P1363, A.4.2, S97f]:

**Berechnung der Squaring Matrix**

**EINGABE:** Körperpolynom  $p(x) = x^n + c_{n-1}x^{n-1} \dots + c_1x + c_0$

**AUSGABE:** S

Sei:  $d_{0,j} \leftarrow c_j, 0 \leq j \leq n$

**Berechne:**

$$x^n = d_{0,n-1}x^{n-1} + \dots + d_{0,1}x + d_{0,0}$$

$$x^{n+1} = d_{1,n-1}x^{n-1} + \dots + d_{1,1}x + d_{1,0}$$

$\vdots$

$$x^{2n-2} = d_{n-2,n-1}x^{n-1} + \dots + d_{n-2,1}x + d_{n-2,0}$$

durch wiederholte Multiplikation mit  $p$ .

Definiere die Matrix S

$$S := \begin{pmatrix} S_{1,1} & \dots & S_{1,m} \\ \vdots & \ddots & \vdots \\ S_{m,1} & \dots & S_{m,m} \end{pmatrix}$$

mit

$$S_{i,j} := \begin{cases} d_{n-2i} & \text{wenn } i \leq \lfloor m/2 \rfloor \\ 1 & \text{wenn } i > \lfloor m/2 \rfloor \text{ und } 2i = j + m \\ 0 & \text{sonst} \end{cases}$$

### 2.1.1.6 Quadratwurzel

Zur Berechnung der Quadratwurzel quadriert man das entsprechende Element  $(n-1)$ -mal:

$$\sqrt{a} = a^{(2^{n-1})}$$

### 2.1.1.7 Exponentiation

Zur Exponentiation kann man die Schnelle Exponentiation nutzen, andere Algorithmen finden sich in [Gor98].

```
Schnelle Exponentiation für Polynome in  $GF(2^n)$ 
EINGABE: Element  $a$ ,
           $k = k_r k_{r-1} \dots k_1 k_0$  die Binärdarstellung von  $k$ 
AUSGABE:  $a^k$ 
1.  $x = a$ 
2. for  $i = r - 1$  downto 0 do
   2.1  $x = x^2$ 
   2.2 if  $k_i = 1$  then  $x = ax$ 
3. Ausgabe  $x$ 
```

### 2.1.1.8 Wurzel

Ein irreduzibles Polynom in  $GF(2^n)$  vom Grad  $d$ ,  $d \mid n$ , hat  $d$  verschiedene Wurzeln im Körper  $GF(2^n)$ . Eine Wurzel wird zur Basistransformation zwischen Polynom- und ONB-Darstellungen benötigt. Eine zufällige Wurzel lässt sich mit dem folgenden Algorithmus berechnen:

```
Zufällige Wurzel
EINGABE: ein irreduzibles Polynom
           $f(x) = a_d x^d + \dots + a_1 x + a_0$  vom Grad  $d$ ,  $d \mid m$ 
AUSGABE: eine zufällige Wurzel von  $f(x)$  in  $GF(2^n)$ 
Sei  $g(t) = b_d t^d + \dots + b_1 t + b_0$ ,
      $c(t)$  Polynome über Elemente aus  $GF(2^n)$ 
1. Setze  $g(t) = f(x)$ :
    $b_i = \begin{cases} 1 - \text{Element der Darstellung} & \text{falls } a_i = 1 \\ 0 - \text{Element der Darstellung} & \text{falls } a_i = 0 \end{cases}$ 
2. while  $\deg(g) > 1$  do
  2.1 do
   2.1.1 wähle ein zufälliges  $u \in GF(2^n)$ 
   2.1.2  $c(t) = ut$ 
   2.1.3 for  $i = 1$  to  $m - 1$  do
    2.1.3.1  $c(t) = c(t)^2 + ut \bmod g(t)$ 
   2.1.4  $h(t) = \gcd(c(t), g(t))$ 
   while  $\deg(h(t)) = 0$  or  $\deg(g) = \deg(h)$ 
  2.2 if  $2\deg(h) > \deg(g)$ 
   2.2.1 then  $g(t) = g(t)/h(t)$ 
   2.2.2 else  $g(t) = h(t)$ 
3. Ausgabe  $g(0)$ 
```

## 2.1.2 Optimale Normalbasen

Elemente aus  $GF(2^n)$  lassen sich auch als Linearkombination der Elemente einer Normalbasis darstellen:

$$B = \{\theta, \theta^2, \theta^{2^2}, \dots, \theta^{2^{n-1}}\}$$

Die Elemente der Basis müssen linear unabhängig sein. Der Bitstring  $(a_0 a_1 \dots a_{m-1})$  wird als das Element

$$a_0\theta + a_1\theta^2 + a_2\theta^{2^2} + \dots + a_{n-1}\theta^{2^{n-1}}$$

dargestellt. Alle Elemente einer Normalbasis erfüllen das gleiche irreduzible, binäre Körperpolynom  $p(t)$ . Multiplikationen lassen sich in Optimaler Normalbasendarstellung in Hardware sehr schnell realisieren, Die entsprechende Arithmetik findet sich in [P1363, Annex A], [Fer00].

### 2.1.2.1 Berechnung des Körperpolynoms einer Normalbasis

Zur Basiskonvertierung wird das Körperpolynom der zu konvertierenden Optimalen Normalbasis benötigt. Der folgende Algorithmus berechnet das Körperpolynom für Normalbasen vom Typ I oder II, die Algorithmen für andere Typen finden sich in [P1363, A.7.2, S110f].

Körperpolynom einer Normalbasis vom Typ I

EINGABE: Körpergrad  $n$

AUSGABE: Körperpolynom  $p(x)$

1.  $p(x) = t^n + t^{n-1} + \dots + t + 1$
2. Ausgabe  $p(x)$

Körperpolynom einer Normalbasis vom Typ II

EINGABE: Körpergrad  $n$

AUSGABE: Körperpolynom  $p(x)$

1.  $q(x) = 1, p(x) = t + 1$
2. for  $i = 1$  to  $n - 1$  do
  - 2.1  $r(x) = q(x), q(x) = p(x), p(x) = x \cdot q(x) + r(x)$
3. Ausgabe  $p(x)$

### 2.1.3 Basiskonvertierung

Elemente in Polynomdarstellung mit unterschiedlichen Körperpolynomen, sowie Elemente in Optimaler Normalbasendarstellung, lassen sich mit Hilfe der Basiskonvertierung ineinander überführen. Das ist besonders dann hilfreich oder notwendig, wenn externe Daten (Schlüssel, Zertifikate, ...) benutzt werden sollen, die in anderen Basen vorliegen, oder wenn zum Beispiel zur schnellen hardwarebasierten Multiplikation Optimale Normalbasen genutzt werden sollen.

Zur Basiskonvertierung berechnet man die sogenannte Konvertierungsmatrix  $\Gamma$ , eine  $(n \times n)$  Matrix mit Elementen aus  $\{0, 1\}$ .

Für den Bitstring  $u$ , der das Element  $\alpha$  in der Basis  $B_0$  repräsentiert erhält man den Bitstring  $v$ , der  $\alpha$  in der Basis  $B_1$  repräsentiert, als Ergebnis der Vektormultiplikation:

$$v = u\Gamma$$

Die Konvertierung von  $B_1$  nach  $B_0$  erfolgt mit Hilfe der Inversen von  $\Gamma$ :

$$u = v\Gamma^{-1}$$

### 2.1.3.1 Berechnung der Konvertierungsmatrix

Der folgende Algorithmus berechnet die Konvertierungsmatrix von  $B_0$  nach  $B_1$ :

Berechnung der Konvertierungsmatrix

EINGABE: Körpergrad  $n$ , Basis  $B_0$  mit Körperpolynom  $p_0(x)$ , Basis  $B_1$  mit Körperpolynom  $p_1(x)$

AUSGABE: Konvertierungsmatrix  $\Gamma$

1. Berechne  $u$  als eine Wurzel von  $p_0(u)$  in Darstellung von  $B_1$
2. Definiere Elemente  $e_0, \dots, e_{n-1}$  wie folgt:

$$e_j = \begin{cases} t^{n-1-j} & \text{wenn } B_1 = \{t^{n-1}, \dots, t^2, t, 1\} \text{ (Polynombasis)} \\ \theta^{2^j} & \text{wenn } B_1 = \{\theta, \theta^2, \theta^{2^2}, \dots, \theta^{2^{n-1}}\} \text{ (ONB)} \end{cases}$$

3. Berechne die Elemente  $\gamma_{i,j}$ ,  $0 \leq i < n$ ,  $0 \leq j < n$ :
  - falls  $B_0$  eine Polynombasis ist:

$$1 = \sum_{j=0}^{n-1} \gamma_{n-1,j} e_j$$

$$u = \sum_{j=0}^{n-1} \gamma_{n-2,j} e_j$$

⋮

$$u^{n-2} = \sum_{j=0}^{n-1} \gamma_{1,j} e_j$$

$$u^{n-1} = \sum_{j=0}^{n-1} \gamma_{0,j} e_j$$

durch wiederholte Multiplikation mit  $u$ .

- falls  $B_0$  eine Optimale Normalbasis ist:

$$u = \sum_{j=0}^{n-1} \gamma_{0,j} e_j$$

$$u^2 = \sum_{j=0}^{n-1} \gamma_{1,j} e_j$$

$$u^{2^2} = \sum_{j=0}^{n-1} \gamma_{2,j} e_j$$

⋮

$$u^{2^{n-1}} = \sum_{j=0}^{n-1} \gamma_{n-1,j} e_j$$

durch wiederholtes Quadrieren.

#### 4. Ausgabe

$$\Gamma \leftarrow \begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,n-1} \\ \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{n-1,0} & \gamma_{n-1,1} & \cdots & \gamma_{n-1,n-1} \end{pmatrix}$$

#### Beispiel: ONB $\rightarrow$ Polynomdarstellung

Sei  $B_0$  die Optimale Normalbasis über  $GF(2^4)$  und  $B_1$  die Polynombasis modulo  $p_1(x) = x^4 + x^3 + x^2 + x + 1$ . Dann ist  $p_0(x) = x^4 + x + 1$  und eine Wurzel

$u = x^3 + x^2 + x + 1$ . Durch wiederholtes Quadrieren erhält man:

$$u = x^3 + x^2 + x + 1$$

$$u^2 = x^3 + x$$

$$u^4 = x^3$$

$$u^8 = x^3 + x^2$$

und die Konvertierungsmatrix

$$\Gamma = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

Das Element  $a = 0010_{B_0}$  wird konvertiert zu  $a' = 1100_{B_1} = x^3 + x^2$ .

#### Beispiel: Polynomdarstellung $\rightarrow$ Polynomdarstellung

Sei  $B_0$  die Polynombasis über  $GF(2^5)$  modulo  $p_0(x) = x^5 + x^3 + 1$  und  $B_1$  die Polynombasis modulo  $p_1(x) = x^5 + x^3 + x^2 + x + 1$ . Dann ist  $u = x^4 + x^3 + x + 1$  eine Wurzel von  $p_0(x)$  und durch Multiplizieren mit  $u$  erhält man:

$$1 = 1$$

$$u = x^4 + x^3 + x + 1$$

$$u^2 = x^4 + x$$

$$u^3 = x^4 + x^3 + x^2$$

$$u^4 = x^3 + x^2 + 1$$

und die Konvertierungsmatrix

$$\Gamma = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Das Element  $a = 00100_{B_0} = x^2$  wird konvertiert zu  $a' = 10010_{B_1} = x^4 + x$ .

### Beispiel: Polynomdarstellung $\rightarrow$ ONB

Sei  $B_0$  die Polynombasis modulo  $p_0(x) = x^5 + x^3 + 1$  und  $B_1$  die Optimale Normalbasis über  $GF(2^5)$ . Dann ist  $u = \theta^2 + \theta^4 + \theta^8 + \theta^{16}$  eine Wurzel von  $p_0(x)$  und man erhält:

$$1 = \theta + \theta^2 + \theta^4 + \theta^8 + \theta^{16}$$

$$u = \theta^2 + \theta^4 + \theta^8 + \theta^{16}$$

$$u^2 = \theta + \theta^4 + \theta^8 + \theta^{16}$$

$$u^3 = \theta + \theta^8 + \theta^{16}$$

$$u^4 = \theta + \theta^2 + \theta^8 + \theta^{16}$$

und die Konvertierungsmatrix

$$\Gamma = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Das Element  $a = 11100_{B_0} = x^4 + x^3 + x^2$  wird konvertiert zu  $a' = 10011_{B_1}$ .

## 2.2 Elliptische Kurven über $GF(2^n)$

Die Implementierung der Elliptischen Kurven wurde aus dem CDCECProvider [CDC] übernommen und an die Polynom- bzw. Optimale Normalbasen-Basisarithmetik angepasst. Ich verweise deswegen an dieser Stelle auf das sehr gute Online-Tutorial von Certicom (<http://www.certicom.com/research.html>) [Cer], Arbeiten aus dem Fachbereich [Hen99], [Ham00], [Fer00] und den IEEE-Standard [P1363]. Grundlagen finden sich auch in [Bar97], [Joh99], [Kob94], [Ros98] und [Men93].

## 2.3 EC Domain Parameter

Zu jedem kryptographischen Verfahren in der Public Key Kryptographie (Verschlüsselung, Schlüsselaustausch, Signatur) gehört ein Satz Parameter, die *Domain Parameter*. Zu den Verfahren, die mit Elliptischen Kurven arbeiten, gehört immer ein Satz von EC Domain Parametern, mit denen die benutzten Schlüssel assoziiert sind. Ein gültiger Schlüssel kann nicht zusammen mit den "falschen" EC Domain Parametern benutzt werden.

Die EC Domain Parameter bestehen aus:

- einem Körper  $GF(q)$ , wobei  $q$  eine Primzahl oder  $2^m$  ist ( $m$  ist eine positive ganze Zahl)
- zwei Elementen aus  $GF(q)$ :  $a, b$ , die die Elliptische Kurve  $E$  definieren
- einer Primzahl  $r$ , die die Anzahl der Punkte auf der Kurve  $E$  teilt
- einem Punkt  $G$ , der Erzeuger einer Untergruppe der Ordnung  $r$  ist.

Implizit werden auch diese Parameter definiert:

- $\#E$ , die Anzahl der Punkte auf  $E$
- der Kofaktor  $k = \#E/r$ .

Ein Satz EC Domain Parameter besteht also aus  $q, a, b, r, G, k$ .

Die folgende Liste zeigt die aus dem Standard P1363 [P1363] übernommene und in dieser Arbeit gebrauchte Notation:

$q$	die Anzahl der Elemente des benutzten Körpers
$a, b$	die Koeffizienten der Elliptischen Kurve $E$ , Elemente aus $GF(q)$ : $y^2 = x^3 + ax + b, a, b \in GF(q)$
$E$	die Elliptische Kurve über dem Körper $GF(q)$ , definiert durch $a$ und $b$
$\#E$	die Anzahl der Punkte auf der Kurve $E$
$r$	ein Primfaktor von $\#E$ und Ordnung von $G$
$G$	ein Punkt auf $E$ , Erzeuger einer Untergruppe der Ordnung $r$
$k$	$\#E/r$ , der Kofaktor
$(u, V)$	ein Schlüsselpaar: $u$ ist eine ganze Zahl und der private Schlüssel, $V$ ist ein Punkt auf $E$ und der öffentliche Schlüssel
$M$	die zu signierende Nachricht
$f$	der Hashwert der Nachricht $M$ , eine ganze Zahl

## 2.4 Schlüsselerzeugung

Ein Schlüsselpaar besteht aus dem privaten Schlüssel  $u$  und dem öffentlichen Schlüssel  $V$ , wobei  $u$  eine ganze Zahl und  $V$  ein Punkt auf der Elliptischen Kurve ist. Ein Schlüsselpaar hat nur Gültigkeit in Verbindung mit den assoziierten EC Domain Parametern.

Um ein neues Schlüsselpaar zu erzeugen, wählt man eine zufällige ganze Zahl  $u$  mit  $1 \leq u < r$  und berechnet den Punkt  $V$  auf der Elliptischen Kurve  $E$  mit  $V = uG$ ,  $V \neq \mathcal{O}$ , ( $\mathcal{O}$  beschreibt den Punkt im Unendlichen)

## 2.5 ECDSA

Der Elliptic Curve Digital Signature Algorithm ist die Elliptische Kurven Version des DSA (Digital Signature Algorithm) und basiert auf [Mil86], [Kob87] und [Kra93]. Er berechnet eine Signatur des Nachrichten-Hashwertes mit Hilfe des privaten Schlüssels  $s$  des Unterzeichners. Der Hashwert kann bei der Verifikation nicht wiederhergestellt werden, das heißt der Hashwert der zu überprüfenden Nachricht geht als Eingabe in den Algorithmus hinein und die Ausgabe ist lediglich wahr oder falsch. Im Gegensatz dazu berechnet der ECNR-Algorithmus bei der Verifikation den Hashwert, der dann mit dem der zu überprüfenden Nachricht verglichen werden muss.

### Signieren mit ECDSA

#### EINGABE:

- EC Domain Parameter:  $q, a, b, r$  und  $G$  assoziiert mit dem privaten Schlüssel  $s$
- privater Schlüssel  $s$
- Nachrichten-Hashwert  $f \geq 0$

#### AUSGABE:

- Signatur  $(c, d)$ ,  $1 \leq c < r$ ,  $1 \leq d < r$
- 1. Generiere Einmalschlüsselpaar  $(u, V)$  mit den gleichen Domain-Parametern wie  $s$ ,  $V = (x_V, y_V)$  ( $V \neq \mathcal{O}$ , weil  $V$  ein öffentlicher Schlüssel ist)
- 2. Konvertiere  $x_V$  in einen Integer-Wert  $i$  (entsprechend FE2IP aus P1363)
- 3. Berechne  $c = i \bmod r$ . Falls  $c = 0$  gehe zu 1.
- 4. Berechne  $d = u^{-1}(f + sc) \bmod r$ . Falls  $d = 0$  gehe zu 1.
- 5. Ausgabe der Signatur  $(c, d)$

### Verifizieren mit ECDSA

#### EINGABE:

- EC Domain Parameter:  $q, a, b, r$  und  $G$  assoziiert mit dem öffentlichen Schlüssel  $W$
- öffentlicher Schlüssel des Unterzeichners  $W$
- Nachrichten-Hashwert  $f \geq 0$
- zu verifizierende Signatur  $(c, d)$

#### AUSGABE:

- "gültig", falls  $(c, d)$  eine gültige Signatur von  $f$  mit den

dazugehörigen EC Domain-Parametern und Schlüssel ist,  
 “ungültig” sonst.

1. if  $\text{not}(1 \leq c < r)$  or  $\text{not}(1 \leq d < r)$  then  
 Ausgabe “ungültig” und beenden
2. Berechne  $h = d^{-1} \text{mod } r$ ,  $h_1 = fh \text{mod } r$ ,  $h_2 = ch \text{mod } r$
3. Berechne Punkt  $P = h_1G + h_2W$ ,  $P = (x_P, y_P)$
4. if  $P = \mathcal{O}$  Ausgabe “ungültig” und beenden
5. Konvertiere  $x_p$  in einen Integer-Wert  $i$  (entsprechend FE2IP aus P1363)
6. Berechne  $c' = i \text{mod } r$
7. if  $c = c'$  then
  - 7.1 Ausgabe “gültig” und beenden
 else
  - 7.2 Ausgabe “ungültig” und beenden

## 2.6 ECNR

ECNR ist das Elliptische Kurven Signaturverfahren von Nyberg und Rueppel. Es basiert auf [Mil86], [Kob87] und [NR93] und berechnet die Signatur eines Nachrichten-Hashwertes mit Hilfe des privaten Schlüssels  $s$  des Unterzeichners. Der Nachrichten-Hashwert wird bei der Verifikation wiederhergestellt und muss mit dem der zu überprüfenden Nachricht verglichen werden.

Signieren mit ECNR

EINGABE:

- EC Domain Parameter:  $q$ ,  $a$ ,  $b$ ,  $r$  und  $G$  assoziiert mit dem privaten Schlüssel  $s$
- privater Schlüssel  $s$
- Nachrichten-Hashwert  $f \geq 0$

AUSGABE:

- Signatur  $(c, d)$ ,  $1 \leq c < r$ ,  $1 \leq d < r$
1. Generiere Einmalschlüsselpaar  $(u, V)$  mit den gleichen Domain-Parametern wie  $s$ ,  $V = (x_V, y_V)$   
 ( $V \neq \mathcal{O}$ , weil  $V$  ein öffentlicher Schlüssel ist)
  2. Konvertiere  $x_V$  in einen Integer-Wert  $i$  (entsprechend FE2IP aus P1363)
  3. Berechne  $c = i + f \text{mod } r$ . Falls  $c = 0$  gehe zu 1.
  4. Berechne  $d = u - sc \text{mod } r$
  5. Ausgabe der Signatur  $(c, d)$

Verifizieren mit ECNR

EINGABE:

- EC Domain Parameter:  $q$ ,  $a$ ,  $b$ ,  $r$  und  $G$  assoziiert mit dem öffentlichen Schlüssel  $W$
- öffentlicher Schlüssel des Unterzeichners  $W$
- zu verifizierende Signatur  $(c, d)$

AUSGABE:

- Nachrichten-Hashwert  $f$ , ( $0 \leq f < r$ ) oder “ungültig”

1. if  $\text{not}(1 \leq c < r)$  or  $\text{not}(1 \leq d < r)$  then  
Ausgabe "ungültig" und beenden
2. Berechne Punkt  $P = dG + cW$ ,  $P = (x_P, y_P)$
3. if  $P = \mathcal{O}$  Ausgabe "ungültig" und beenden
4. Konvertiere  $x_p$  in einen Integer-Wert  $i$  (entsprechend FE2IP aus P1363)
5. Berechne  $f = c - i \bmod r$
6. Ausgabe  $f$  als Nachrichten-Hashwert

# Kapitel 3

## Java und JCA

Dieses Kapitel beschreibt die Architektur und Benutzung der Java Cryptography Architecture, in die die Implementierung dieser Arbeit integriert ist. Die Benutzung des Signaturverfahrens von Nyberg und Rueppel mit der JCA wird anhand von Sequenzdiagrammen erläutert. Außerdem wird auf Java-spezifische Aspekte und Laufzeitoptimierungen unter Java, sowie die zur Implementierung verwendeten Werkzeuge eingegangen.

### 3.1 Die Java Cryptography Architecture

Die Java Cryptography Architecture (JCA) ist seit Version 1.2 fester Teil des Java Development Kit (JDK) von Sun [Java]. Sie bietet weitreichende kryptographische Funktionen für Java-Programme und ist für den Anwender recht einfach und transparent zu benutzen. An dieser Stelle soll nur ein kleiner Einblick in die Architektur und Funktionalität gegeben werden, weitreichende Informationen finden sich auf der JCA-Seite von SUN [JCA] und in [Knu98].

Ein grundlegendes Konzept der JCA ist die Trennung in Interface (API) und Provider. Kryptographische Konzepte (Verschlüsselung, Message-Digest, Signatur, Schlüsselverwaltung, ...) werden in der API festgelegt, während die eigentliche Implementierung in einem Cryptographic Service Provider, im folgenden Provider genannt, steckt. Die API ist fester Teil der JCA<sup>1</sup> und findet sich im package *java.security*. Inzwischen existieren verschiedene Provider, die ein mehr oder weniger umfassendes Subset der API implementieren. Einer davon wird von Sun mitgeliefert, der Standard-Provider "SUN", der aber aufgrund der US-Amerikanischen Exportrestriktionen nur eine stark limitierte Funktionalität bereitstellt. Am Institut wurde der sehr umfangreiche CDCProvider entwickelt [CDC], der neben den Standard-Verfahren auch Elliptische Kurven-Kryptographie anbietet und in den die zu dieser Arbeit gehörende Implementierung integriert ist.

Die Trennung der Konzepte von den Implementierungen in der JCA ermöglicht es die folgenden Ziele zu erreichen:

- Implementierungs- und Algorithmusunabhängigkeit

---

<sup>1</sup>Ein Teil der API befindet sich in der sog. JCE (Java Cryptography Extensions), die aufgrund von Exportrestriktionen nicht aus den USA exportiert werden darf.

- Interoperabilität
- Erweiterbarkeit

**Implementierungs- und Algorithmusunabhängigkeit** lassen den Anwender kryptographische Konzepte nutzen, ohne sich um die jeweilige Implementierung kümmern zu müssen. Er verlangt lediglich nach einer Instanz von `Signature`, z.B. vom Typ `DSA`, erhält diese und kann dann ohne weiteres damit arbeiten. Er kann aber auch eine bestimmte Implementierung eines Verfahrens verlangen (z.B. `DSA` vom `CDCProvider`) und somit auf Implementierungsunabhängigkeit verzichten. Algorithmusunabhängigkeit wird durch die Definition sogenannter *engines* (services) erreicht. *engine classes* stellen die Funktionalität der *engines* (z.B. `MessageDigest`, `Signature`, `KeyFactory`) bereit. Implementierungs- und Algorithmusunabhängigkeit wird durch die Provider-basierte Architektur ermöglicht.

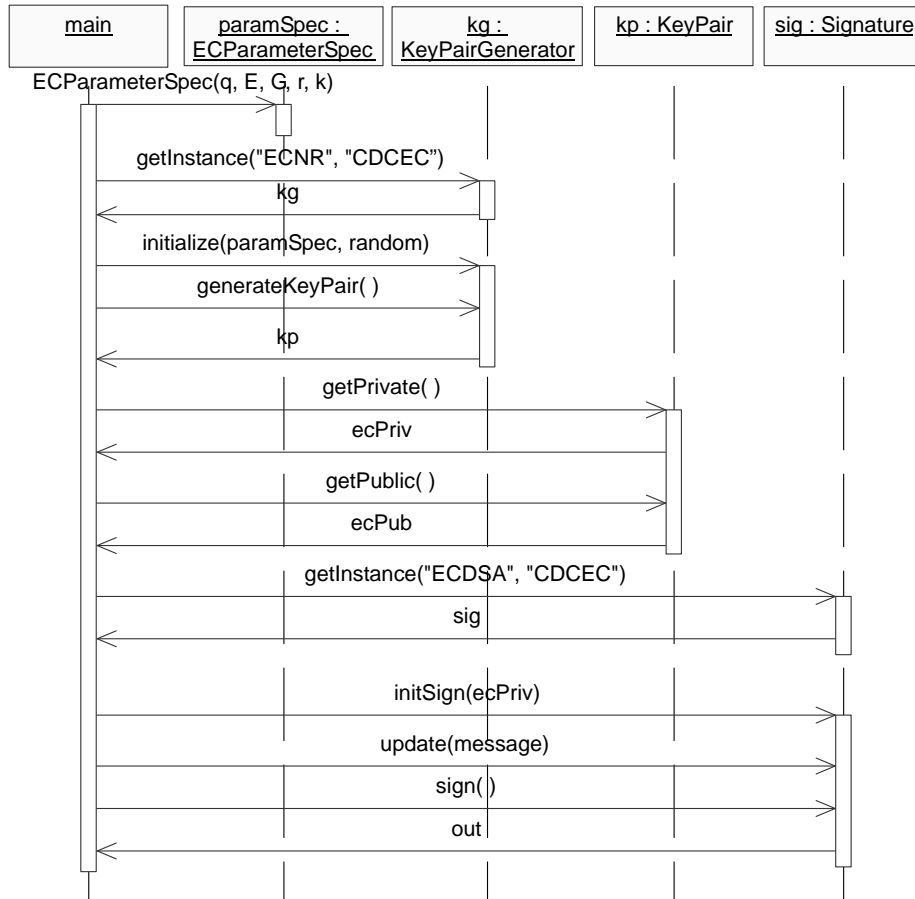
**Interoperabilität** sorgt dafür, dass verschiedene Implementierungen zusammenarbeiten können, so dass ein von einer Implementierung erzeugter Schlüssel von einer anderen benutzt werden kann. Dadurch ist es möglich Schlüssel, Signaturen, verschlüsselte Daten, usw. auszutauschen und in unterschiedlichen Implementierungen zu nutzen.

**Erweiterbarkeit** bedeutet, dass neue Algorithmen einfach durch Austausch des Providers hinzugefügt oder bestehende durch schnellere, sicherere ersetzt werden können. Genauso kann eine unsichere Implementierung durch ein sicheres Verfahren, einen aktuellen oder einen anderen Provider ersetzt werden.

### 3.1.1 Benutzung der JCA

Im Folgenden soll anhand eines Beispiels kurz auf die Benutzung der Java Cryptography Architecture eingegangen werden. Dazu wird der Ablauf beim Erzeugen und Verifizieren einer Signatur dargestellt.

### 3.1.1.1 Erzeugen eines Schlüsselpaares und einer ECNR-Signatur



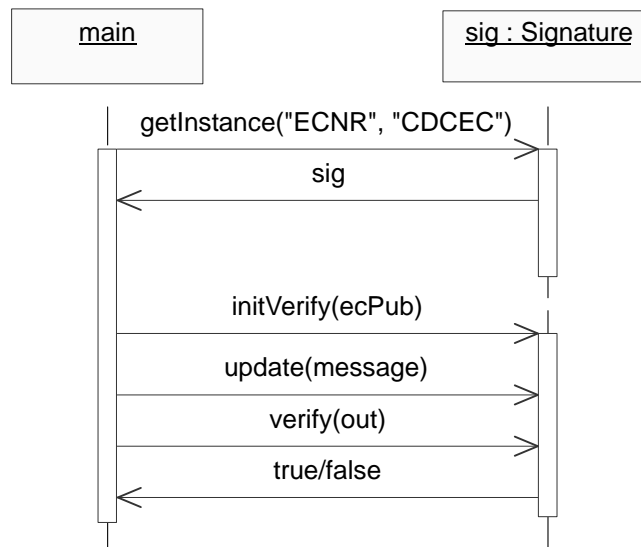
Zum Erzeugen eines Schlüsselpaares und einer Signatur geht der Anwender, wie im Sequenzdiagramm gezeigt, folgendermaßen vor:

1. Als erstes muss der Anwender sich einen Satz von EC Domainparametern erzeugen. Dazu generiert er eine Instanz `paramSpec` von `ECPParameterSpec` mit den gewünschten Werten für  $(q, E, G, r, k)$ .
2. Als nächstes wird ein Schlüsselpaar benötigt. Um ein passendes Schlüsselpaar erzeugen zu können, besorgt sich der Anwender einen zum verwendeten Algorithmus passenden `KeyPairGenerator` durch Aufrufen der *factory method* `KeyPairGenerator.getInstance("ECNR", "CDCEC")`. Der Name des Providers ("CDCEC") muss dabei nicht übergeben werden, wenn es unwichtig ist, von welchem Provider die angeforderte Implementierung von ECNR stammt.
3. Der erhaltene `KeyPairGenerator` `kg` wird dann mit den schon erzeugten EC Domainparametern `paramSpec` und einem `SecureRandom` initialisiert

`kg.initialize(paramSpec, Random)` und mit `kg.generateKeyPair()` wird schließlich das gewünschte `KeyPair kp` erzeugt.

4. Nun können der private (`ecPriv`) und der öffentliche (`ecPub`) Schlüssel mit den Methoden `getPrivate()` und `getPublic()` aus dem `KeyPair kp` ausgelesen werden.
5. Wie bei Punkt 2 erhält der Anwender eine Instanz `sig` der Klasse `ECNR-Signature` durch Aufrufen der *factory method* `Signature.getInstance("ECNR", "CDCEC")`
6. Die Instanz `sig` wird nun mit dem privaten Schlüssel `ecPriv` zum Signieren initialisiert: `sig.initSign(ecPriv)`. Danach wird mit der Methode `update(message)` die zu signierende Nachricht übergeben und durch Aufruf von `sign()` die eigentliche Signatur `out` erzeugt.

### 3.1.1.2 Verifizieren einer ECNR-Signatur



Die Schritte zum Verifizieren einer Signatur sind denen beim Erzeugen ähnlich:

1. Der Anwender erzeugt sich eine Instanz `sig` der Klasse `ECNRSignature` durch Aufrufen der *factory method* `Signature.getInstance("ECNR", "CDCEC")`
2. Die Instanz `sig` wird durch den Aufruf der Methode `sig.initVerify(ecPub)` und Übergabe des öffentlichen Schlüssels `ecPub` zum Verifizieren initialisiert. Die zu verifizierende Nachricht wird durch die Methode `sig.update(message)` übergeben und schließlich mit `sig.verify(out)` überprüft. Der Anwender erhält bei einer gültigen Signatur `true` zurück, ansonsten `false`.

## 3.2 Optimierungen

Eines der Hauptziele dieser Diplomarbeit war es, die Implementierung möglichst leistungsfähig zu machen, aber plattformunabhängig zu bleiben. Im Gegensatz zu den meisten Programmiersprachen, werden Java-Programme nicht in direkt ausführbaren Maschinencode, sondern einen Bytecode übersetzt, der von einer *Virtual Machine*, einem imaginären Rechner, ausgeführt wird. Es gibt also für jede Hardware Plattform einen Interpreter für Java Bytecode, der diesen in ausführbaren Maschinencode dieser Plattform umsetzt und so das Programm interpretiert.

Durch die Übersetzung während der Laufzeit, oder genauer genommen der Simulation eines Virtuellen Rechners, geht natürlich viel Zeit verloren, die teilweise durch Technologien wie Just-In-Time Compiler (JIT) oder HotSpot ausgeglichen werden kann.

Trotzdem können nicht alle Optimierungen, wie zum Beispiel bei einem C++ Programm, angewendet werden. Die folgenden Möglichkeiten wurden in der Implementierung benutzt:

**Schnelle Algorithmen** sind sicherlich der wichtigste Schritt, um eine gute Performance zu erreichen. Die besten Compiler, Optimierungen und schnellsten Rechner, nutzen nichts, wenn der verwendete Algorithmus eine schlechte Laufzeit hat. Es wurde versucht, für alle verwendeten Verfahren den besten bekannten Algorithmus zu finden und anzuwenden. Die entsprechenden Algorithmen sind in Kapitel 2 beschrieben.

**Möglichst wenige Objekte erzeugen.** Jedes erzeugte Objekt benötigt Speicherplatz und muss initialisiert und nach Gebrauch wieder gelöscht werden, was in Java aufgrund der Architektur des Garbage Collectors besonders lange dauert. Das liegt unter anderem an der Speicherarchitektur des Stacks der *Virtual Machine* und an der Tatsache, dass die Zugriffsrechte auf Klassen und Methoden zur Laufzeit geprüft werden müssen.

**Nicht mehr benötigten Objekten NULL zuweisen,** damit der Garbage Collector diese schneller finden, wegwerfen und den benötigten Speicherplatz wieder freigeben kann. Unnötig bei lokalen Variablen in kleinen Methoden, sinnvoller bei Instanzvariablen mit längerer Gültigkeit.

**Nutzen maschinennaher Operationen** zum Beispiel Linksshiften um  $i$  Stellen anstelle von Multiplizieren mit  $2^i$ . Auch wenn Java Bytecode interpretiert wird läuft der Interpreter letztendlich auf einem herkömmlichen Rechner, der bestimmte Operationen viel schneller durchführt als andere und das Nutzen maschinennaher Operationen wirkt sich, wie bei jeder anderen Sprache auch, positiv aus.

**Vorberechnete Tabellen nutzen** zum Beispiel vorberechnete Bitmasken `a&bitMask[i]` anstelle von Operationen wie `a&=0x00000001 << i`. Das Auflösen des Verweises in eine Tabelle geht im Allgemeinen deutlich schneller als das Berechnen des entsprechenden Wertes.

**Nutzen von schnellen Datentypen.** Die Klasse Bitstring, die die Koeffizienten der GF2nPolynome speichert, wurde einmal mit einem Integer-Array (32 Bit) und zum Vergleich mit anderen Datentypen implementiert.

Die Implementierung mit Arrays vom Typ Long (64 Bit) war um bis zu 30% langsamer, was daran liegen dürfte, dass der Interpreter für Integers direkt die Maschinenregister in Hardware benutzen kann, Longs aber – zumindest auf 32-Bit Maschinen – emulieren muss.

**Auflösen von Rekursionen** Man sollte versuchen, rekursive Methodenaufrufe in iterative zu verwandeln, da diese vom Compiler besser optimiert werden können und somit schneller sind. Dies wurde zum Beispiel in der Methode `karaMult`, die die Karatsuba-Multiplikation implementiert, in der Klasse `Bitstring` gemacht.

### 3.2.1 Just-In-Time Compiler

Java hat sich in der vergangenen Zeit stark weiterentwickelt und es gab mehrere Versuche, die anfänglich schlechte Performance zu verbessern. Mit dem JDK 1.2 kam der sogenannte Just-In-Time Compiler, der Java Bytecode vor der Ausführung in nativen Maschinencode übersetzt und ihn dann direkt ausführt. Das Ergebnis kann aber aufgrund der Architektur von Java niemals so gut werden wie das eines "richtigen" Compilers einer nicht interpretierten Sprache. Das liegt unter anderem daran, dass bei Java während der Laufzeit Klassen nachgeladen werden können und Zugriffsrechte auf Klassen und Methoden noch während der Laufzeit verändert werden können. Ein Nachteil des Just-In-Time Compilers ist die anfängliche Wartezeit, in der der Bytecode compiliert wird, bevor das Java-Programm startet.

### 3.2.2 HotSpot

HotSpot war der letzte Schritt in Richtung eines schnelleren Javas und findet sich in Form einer speziellen *HotSpot Virtual Machine* [HotSpot] wieder. Der Bytecode wird nicht wie beim JIT-Compiler am Anfang komplett compiliert, sondern ganz normal in einer Interpretation gestartet. Gleichzeitig wird die Laufzeit von einem Profiler überwacht und es werden Teile des Programms identifiziert, in denen viel Rechenzeit verbraucht wird, sogenannte *Hot Spots*. Laut Suns HotSpot Technologie Dokumenten verbraucht ein typisches Java-Programm ca. 90% seiner Rechenzeit in ca. 10% des Codes, eben in den Hot Spots. Die Virtual Machine compiliert und optimiert nun diese Stellen, zum Beispiel durch das Einfügen von Methoden in den Code, anstelle eines Methodenaufrufs (*inlining*). Diese Optimierungen finden auf dem Stack statt, also im laufenden Programm. Nur so kann zum Beispiel bei Bedarf eine Methode von einer nachgeladenen Klasse überschrieben und ersetzt werden. Um das realisieren zu können, nutzt die HotSpot Virtual Machine zwei Aufrufstacks, einen für compilierten, optimierten Code und einen nur bei Bedarf aktualisierten mit Referenzen im Bytecode. Werden Klassen ersetzt kann mit Hilfe des zweiten Stacks der ursprüngliche, nichtoptimierte Zustand in diesen Klassen wiederhergestellt und ersetzt werden und die Optimierung beginnt in diesen Teilen von neuem.

Vorteile der HotSpot Virtual Machine sind sehr gute Performance bei sehr kurzen Latenzzeiten beim Starten der Programme und trotzdem unverändert hohe Flexibilität. Je länger ein Programm läuft, umso besser und umfangreicher wird es optimiert und die Ergebnisse im Kapitel 5 zeigen, dass das gut

funktioniert. Ein Nachteil von HotSpot ist die etwas schlechtere Performance bei Anwendungen mit komplexen (GUI-lastigen) Benutzerschnittstellen (langsamere Reaktion auf Benutzerinteraktion).

Die HotSpot Virtual Machine ist im JDK für Solaris und Linux bereits integriert, muss aber in der Windows-Version extra installiert werden. Sie wird durch den Aufruf `java -server` aktiviert. Standardmäßig wird der JIT-Compiler benutzt, der auch durch den Aufruf `java -classic` erzwungen werden kann.

### 3.3 Verwendete Werkzeuge

Dieser Abschnitt gibt einen kurzen Überblick über die zum Entwurf, Design und zur Implementierung benutzten Entwicklungsumgebungen und Werkzeuge.

#### 3.3.1 JBuilder4

Zur Programmierung wurde die Entwicklungsumgebung JBuilder in Version 4 von Inprise (ehemals Borland) benutzt, die in der Foundation-Version frei von Inprise (<http://www.inprise.com>) zur Verfügung gestellt wird. Da JBuilder selber in Java entwickelt wurde, gibt es Versionen für Solaris, Linux und Windows Plattformen. Mitgeliefert wird das JDK 1.3 von Sun, es können aber auch andere Virtual Machines, zum Beispiel die HotSpot VM, benutzt werden. Beim Testen der Laufzeiten wurde kein signifikanter Unterschied zwischen JBuilder und dem reinen JDK von Sun festgestellt, dafür bietet JBuilder aber unter anderem einen sehr komfortablen Debugger.

#### 3.3.2 JDK 1.3 & HotSpot Virtual Machine

Zur Performancemessung wurde direkt das installierte, frei verfügbare Java Development Kit von Sun [Java] (<http://java.sun.com>) benutzt. Um optimale Performance zu erreichen, muss die HotSpot Virtual Machine benutzt werden, was durch die Kommandozeilenoption `-server` erreicht wird. Im JDK für Solaris, Unix und Linux ist die HotSpot VM bereits integriert, für die Windows Umgebung muss sie separat installiert werden.

#### 3.3.3 OptimizeIt

Zum Optimieren wurde auch der Profiler OptimizeIt von Intuisys ([www.intuisys.com](http://www.intuisys.com)) in einer 15 Tage Testversion benutzt. Er verbindet sich über ein Socket mit der Virtual Machine und bereitet die gemessenen Zeiten optisch auf. Alle durchlaufenen Methoden lassen sich mit absoluter Zeit, prozentualer Laufzeit und Gesamtlaufzeit darstellen und nach verschiedenen Kriterien sortieren. Leider wurden bei jedem Programmdurchlauf teilweise stark unterschiedliche Zeiten gemessen. Um kritische Stellen zu identifizieren ist das unerheblich, da die Relationen der Laufzeiten der Methoden untereinander im großen und ganzen gleich blieben. Vermutlich gab es hier Konflikte mit der HotSpot Server Virtual Machine.

### 3.3.4 Rational Rose

Zum Designentwurf und zum Zeichnen der Diagramme wurde teilweise das Werkzeug Rational Rose 2001 der Firma Rational, ebenfalls in einer 15 Tage Testversion, benutzt. Leider ist die Benutzerführung in großen Teilen eher schlecht, und die Reverse Engineering Funktionen erkennen selbst einfache Datentypen wie `String` oder `BigInteger` nicht, geschweige denn die Klasse `AlgorithmParametersSpi`, obwohl angeblich das komplette JDK 1.3 geparkt wurde. Die Diagramme lassen sich nicht exportieren, sondern nur über Umwege in Postscript Dateien drucken. Dabei werden teilweise Rahmen zu groß oder zu klein gezeichnet und Texte verschwinden, so dass alle Diagramme von Hand nachbearbeitet werden müssen. Es empfiehlt sich hier auf andere Produkte wie zum Beispiel Visio ([www.microsoft.com/office/visio](http://www.microsoft.com/office/visio)) oder ArgoUML ([argouml.tigris.org](http://argouml.tigris.org)) auszuweichen.

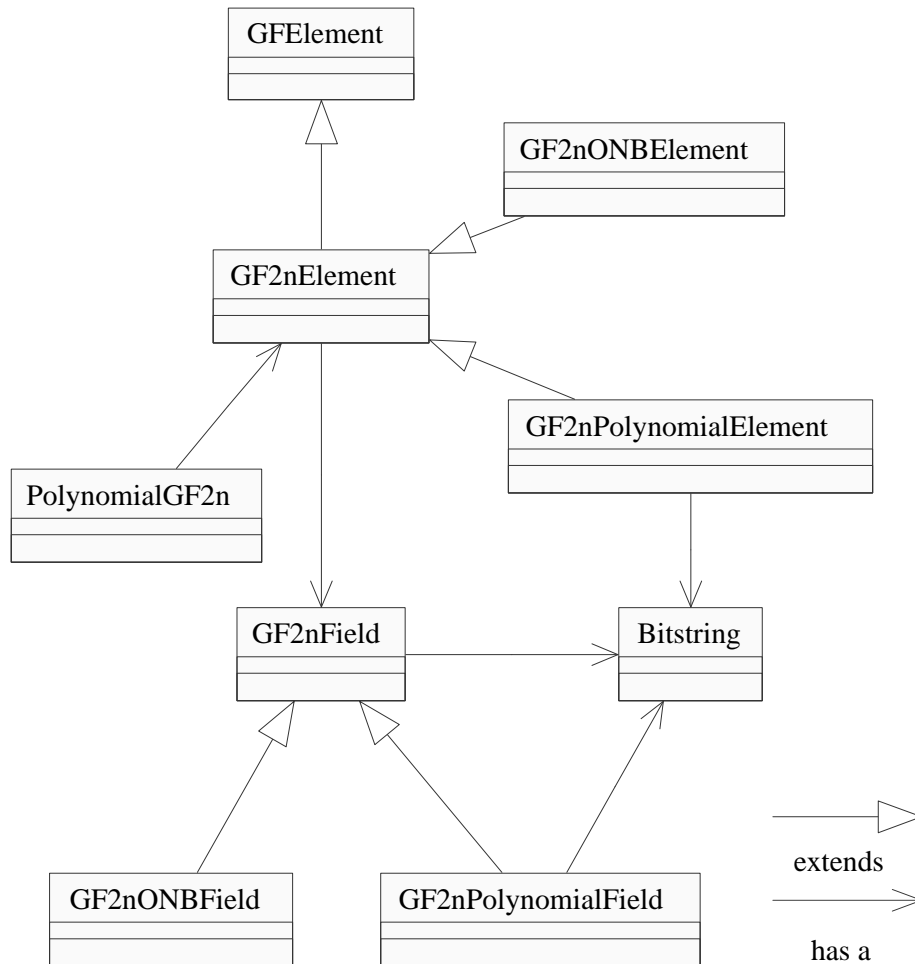


# Kapitel 4

## Entwurf

Dieses Kapitel beschäftigt sich mit dem Entwurf und Design der Implementierung. Es wird eine Übersicht über jedes package gegeben und kurz auf die wichtigsten Klassen eingegangen, sowie als Beispiel die Basiskonvertierung in einem Ablaufdiagramm erläutert. Eine detaillierte Beschreibung der in den einzelnen Klassen benutzten Methoden findet sich im Anhang A.

## 4.1 cdc.ec.arithmetic.gf



Im Package `cdc.ec.arithmetic.gf` sind alle Klassen der  $GF(2^n)$  Basisarithmetik enthalten, also die  $GF(2^n)$ -Elemente in Polynom- und ONB-Darstellung, die Körper über  $GF(2^n)$ , Polynome über  $GF(2^n)$ -Elemente und als elementare Klasse zur Speicherung der Koeffizienten der Elemente die Klasse `Bitstring`.

Die abstrakte Klasse `GF2nElement` beschreibt Elemente aus  $GF(2^n)$  und ist Erbe der Klasse `GFElement`, die Elemente von endlichen Körpern verschiedener Charakteristiken darstellt. Sie definiert die wichtigsten Methoden, speichert den Körpergrad und einen Verweis auf eine Instanz des benutzten Körpers `GF2nField`.

`GF2nONBElement` und `GF2nPolynomialElement` erben von `GF2nElement` und modellieren Elemente aus  $GF(2^n)$  in ONB- bzw. Polynomdarstellung. Sie speichern die Elemente, stellen die eigentliche Basisarithmetik sowie Methoden zur Konvertierung gemäß den Primitiven aus P1363 und zur Basiskonvertierung zur Verfügung und speichern einen Verweis auf den benutzten Körper `GF2nONBField`.

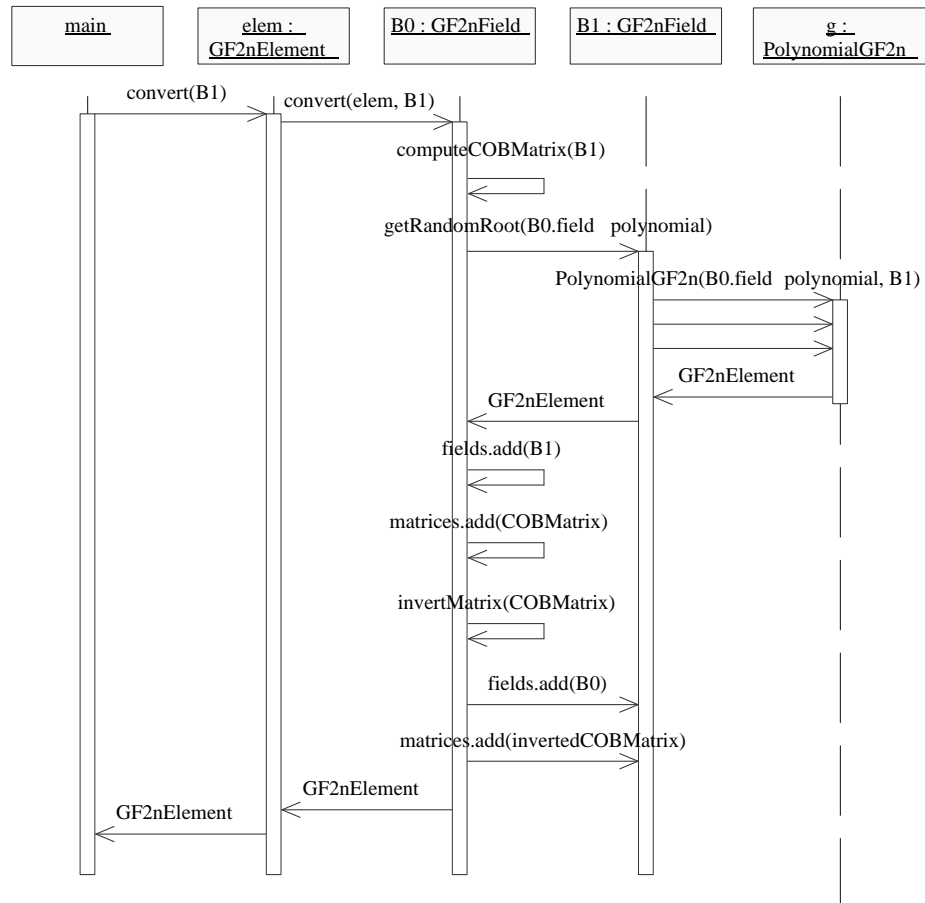
oder `GF2nPolynomialField`.

Die abstrakte Klasse `GF2nField` modelliert die benutzten Körper über  $GF(2^n)$  und definiert Methoden zur Basiskonvertierung zwischen den verschiedenen Darstellungen. Sie speichert den Körpergrad, das irreduzible Körperpolynom und Konvertierungsmatrizen zu schon bekannten anderen Körpern. Die Klassen `GF2nPolynomialField` und `GF2nONBField` realisieren `GF2nField` und stellen diverse darstellungsspezifische Methoden zur Verfügung, beispielsweise eine Quadrierungsmatrix für Elemente in Polynomdarstellung.

Die Klasse `Bitstring` speichert lange, binäre Strings und stellt diverse arithmetische Funktionen und Methoden zur Konvertierung gemäß P1363 zur Verfügung. Sie wird zur Speicherung der Koeffizienten eines Polynoms in den Klassen `GF2nPolynomialElement`, `GF2nField` und `GF2nPolynomialField` benutzt.

Letztendlich stellt die Klasse `PolynomialGF2n` Polynome über `GF2nElemente` und deren Arithmetik dar. Sie wird zur Basistransformation benötigt.

### 4.1.1 Basiskonvertierung



Dieses Ablaufdiagramm zeigt die Methodenaufrufe bei der Basiskonvertierung eines Elementes, für den Fall, dass die entsprechende Basiskonvertierungsmatrix noch nicht berechnet wurde.

Das `GF2nElement` `elem` soll von der Basisdarstellung `B0` nach `B1` konvertiert werden. Dazu ruft das Programm die Methode `convert()` in `GF2nElement` auf, die den Aufruf direkt an das `GF2nField` `B0` weiterreicht. Hier wird zunächst geprüft, ob bereits eine Basiskonvertierungsmatrix von `B0` nach `B1` existiert und falls das zutrifft, wird das Element mittels Vektormultiplikation konvertiert.

Existiert keine Basiskonvertierungsmatrix, muss sie durch die Methode `computeCOBMatrix()` berechnet werden. Da die Polynombasisarithmetik deutlich schneller als die ONB-Arithmetik ist, wird `computeCOBMatrix()` immer in der betei-

ligten Polynombasis berechnet<sup>1</sup>, im Diagramm der Einfachheit halber in  $B_0$ .

Die Methode `computeCOBMatrix()` benötigt eine Wurzel des Körperpolynoms von  $B_0$  in Darstellung von  $B_1$  und ruft dazu `getRandomRoot(fieldpolynomial)` auf, die wiederum mehrere Operationen auf Polynomen über  $GF(2^n)$ -Elementen durchführt und dazu die Klasse `PolynomialGF2n` nutzt. Wenn eine Wurzel gefunden wurde kann die Basiskonvertierungsmatrix berechnet und zusammen mit dem Körper  $B_1$  in den Listen `matrices` und `fields` für weitere Konvertierungen abgelegt werden.

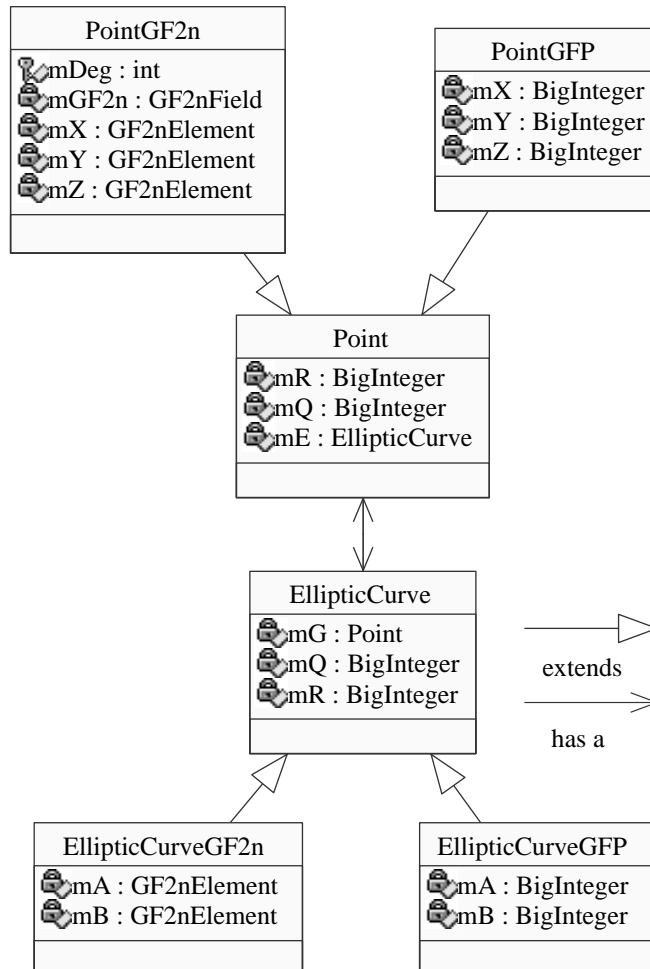
Um auch Konvertierungen in die andere Richtung von  $B_1$  nach  $B_0$  zu ermöglichen, und weil aus Performancegründen immer in Polynomdarstellung gerechnet wird, wird die Matrix nochmal invertiert und mit dem Körper  $B_0$  in  $B_1$  gespeichert. Letztendlich kann das Element konvertiert und eine neue Kopie in der Darstellung von  $B_1$  zurückgegeben werden.

Grundlagen und Beispiele der Basiskonvertierung sind im Kapitel 2.1.3 beschrieben.

---

<sup>1</sup>Eine der beiden Basen muß eine Polynombasis sein, da die ONB-Darstellung für den Körpergrad  $n$  eindeutig ist und eine Konvertierung keinen Sinn macht (Identität), während in Polynomdarstellung verschiedene Körperpolynome zur Reduzierung möglich sind und somit auch verschiedene Basen für den Grad  $n$  benutzt werden können. Es bleiben also drei Szenarien: Polynombasis  $\rightarrow$  ONB, ONB  $\rightarrow$  Polynombasis und Polynombasis  $\rightarrow$  Polynombasis.

## 4.2 cdc.ec.arithmetic.curves



Dieses package beinhaltet Elliptische Kurven über unterschiedlichen Körpern und Punkte auf diesen Kurven, sowie deren Arithmetik.

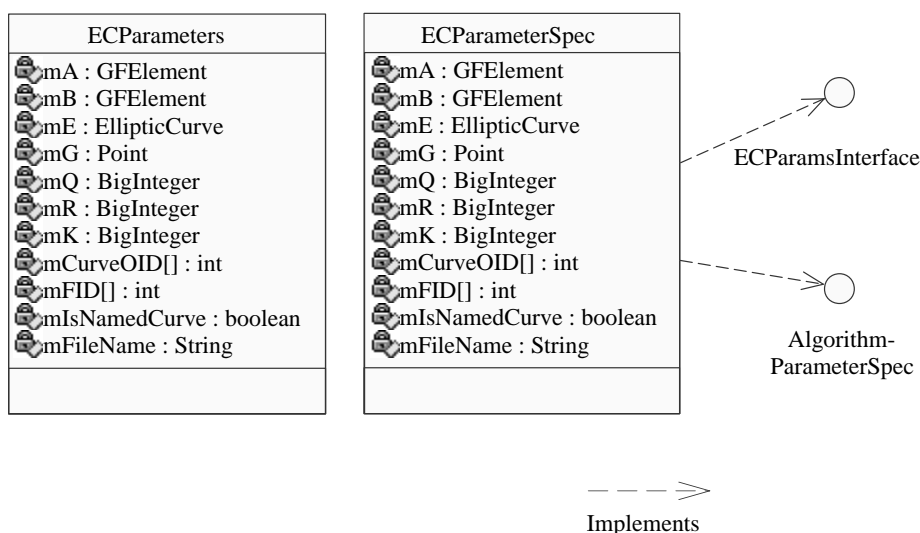
Die Klasse `EllipticCurve` speichert Elliptische Kurven über endlichen Körpern und deren Parameter. Dazu gehören die Koeffizienten `mA` und `mB`, die die Elliptische Kurve definieren, die Größe `mQ` des unterliegenden Körpers, der Erzeuger `mG`, sowie die Ordnung des Erzeugers, `mR`. Der Nachkomme `EllipticCurveGF2n` modelliert Elliptische Kurven über endlichen Körpern der Charakteristik 2 und speichert `mA` und `mB` deswegen als `GF2nElement`, während `EllipticCurveGFP` Elliptische Kurven über Primkörpern darstellt und deswegen `mA` und `mB` als `BigInteger` speichert.

Die Klasse `Point` definiert Punkte auf Elliptischen Kurven und kennt dazu die Elliptische Kurve `mE`, die Größe `mQ` des unterliegenden Körpers, `mR`, die Ordnung des Erzeugers und `mX`, `mY` und `mZ`, die Koordinaten des Punktes. Wie bei `EllipticCurve` auch enthält `PointGF2n` die Koordinaten `mX`, `mY` und `mZ` als `GF2nElement` und `PointGFP` als `BigInteger`.

Die einzelnen Klassen wurden aus dem CDCECProvider [CDC] übernommen

und an die Polynom- bzw. Optimale Normalbasen Basisarithmetik angepasst und werden deswegen im Anhang nicht näher beschrieben. Eine ausführliche Dokumentation in HTML (JavaDoc) wird mit dem Provider zur Verfügung gestellt und steht unter <http://www.informatik.tu-darmstadt.de/TI/Forschung/cdcProvider/overview.html> zur Verfügung.

### 4.3 cdc.ec.ecparameters



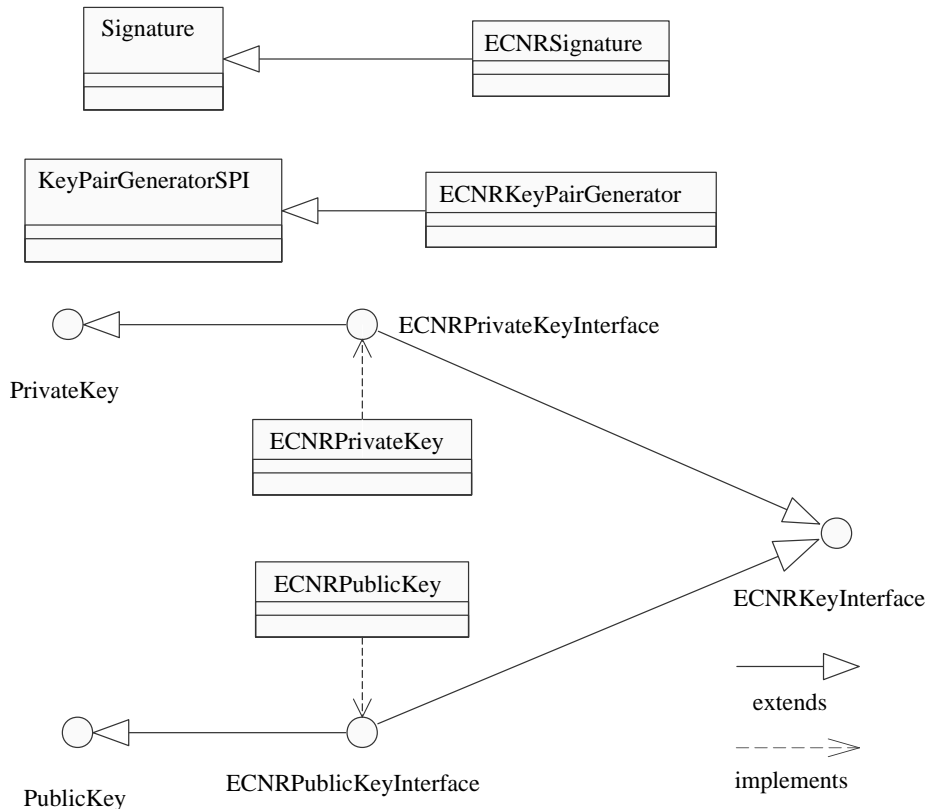
Das Package `cdc.ec.ecparameters` enthält Klassen, die die EC Domain Parameter verwalten und anderen Klassen zur Verfügung stellen.

`EParameters` und `EParameterSpec` speichern jeweils einen Satz von EC Domain Parametern und gewähren anderen Klassen Zugriff auf die einzelnen Parameter. Die Klasse `EParameterSpec` kann als Nachkomme von `AlgorithmParameterSpec` als Parameter an kryptographische Verfahren (Signatur, Verschlüsselung, ...) übergeben werden.

Die einzelnen Klassen wurden aus dem `CDCECProvider [CDC]` übernommen und werden deswegen im Anhang nicht näher beschrieben. Eine ausführliche Dokumentation in HTML (JavaDoc) wird mit dem Provider zur Verfügung gestellt und steht unter

<http://www.informatik.tu-darmstadt.de/TI/Forschung/cdcProvider/overview.html> zur Verfügung.

## 4.4 cdc.ec.ecnr



Im Package `cdc.ec.ecnr` sind alle Klassen enthalten, die die Signatur von Nyberg und Rueppel, sowie die zugehörigen Schlüssel und den `KeyPairGenerator` zur Erzeugung von Schlüsselpaaren implementieren.

Die Klasse `ECNRSignature` erbt von der abstrakten Klasse `Signature`, die Signaturen mit den zugehörigen Methoden zur Initialisierung, zum Updaten der Daten, zum Signieren und zum Verifizieren definiert. Alle Signaturen in der JCA müssen von dieser Klasse erben und lassen sich dadurch gleich benutzen.

Der `ECNRKeyPairGenerator` erbt von der abstrakten Klasse `KeyPairGeneratorSPI` und dient zum Erstellen von neuen Schlüsselpaaren für ECNR-Signaturen.

Das Interface `ECNRKeyInterface` stellt die Private und Public Keys gemeinsamen Funktionen zur Verfügung, in diesem Fall eine Methode, um die EC Domain Parameter auszulesen.

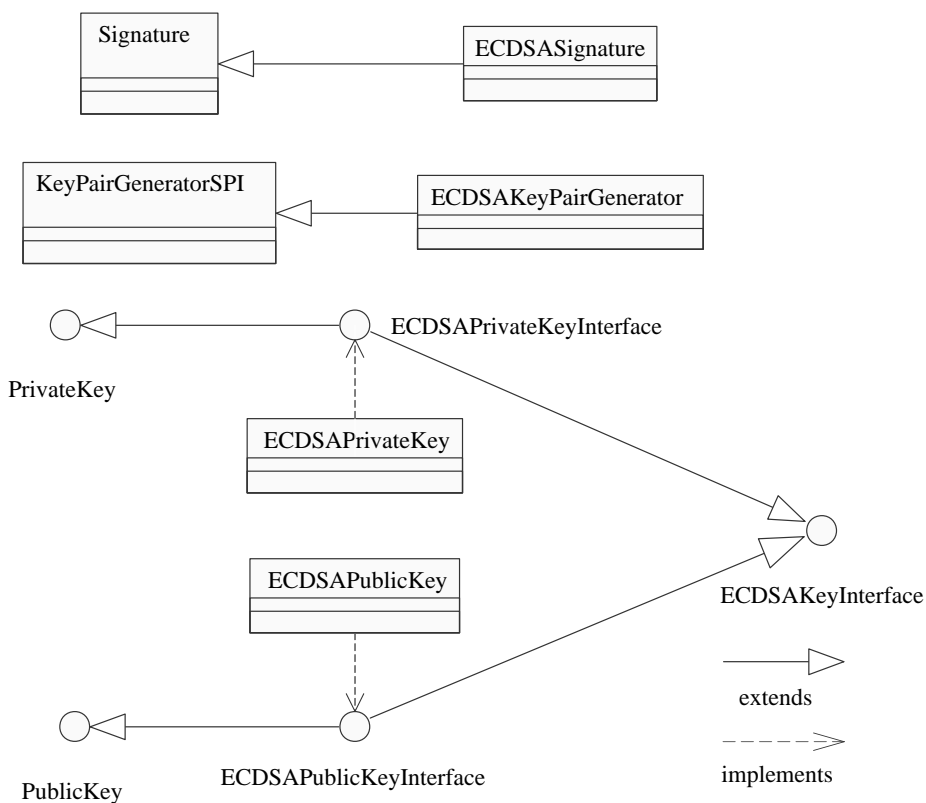
Das Interface `ECNRPrivateKeyInterface` erbt von den Interfaces `PrivateKey` und `ECNRKeyInterface` und stellt eine Methode zum Auslesen des privaten Schlüssels  $s$  zur Verfügung.

Die Klasse `ECNRPrivateKey` implementiert schließlich das Interface `ECNRPrivateKeyInterface` und speichert private ECNR Schlüssel.

Das Interface `ECNRPublicKeyInterface` erbt von den Interfaces `PublicKey` und `ECNRKeyInterface` und stellt eine Methode zum Auslesen des öffentlichen Schlüssels  $W$  zur Verfügung.

Die Klasse `ECNRPublicKey` implementiert schließlich das Interface `ECNR-PublicKeyInterface` und speichert öffentliche ECNR Schlüssel.

## 4.5 cdc.ec.ecdsa



Im Package `cdc.ec.ecdsa` sind alle Klassen enthalten, die die ECDSA-Signatur, sowie die zugehörigen Schlüssel und den `KeyPairGenerator` zur Erzeugung von Schlüsselpaaren implementieren.

Die Klasse `ECDSASignature` erbt von der abstrakten Klasse `Signature`, die Signaturen mit den zugehörigen Methoden zur Initialisierung, zum Updaten der Daten, zum Signieren und zum Verifizieren definiert. Alle Signaturen in der JCA müssen von dieser Klasse erben und lassen sich dadurch gleich benutzen.

Der `ECDSAKeyPairGenerator` erbt von der abstrakten Klasse `KeyPairGeneratorSPI` und dient zum Erstellen von neuen Schlüsselpaaren für ECDSA-Signaturen.

Das Interface `ECDSAKeyInterface` stellt die Private und Public Keys gemeinsamen Funktionen zur Verfügung, in diesem Fall eine Methode, um die EC Domain Parameter auszulesen.

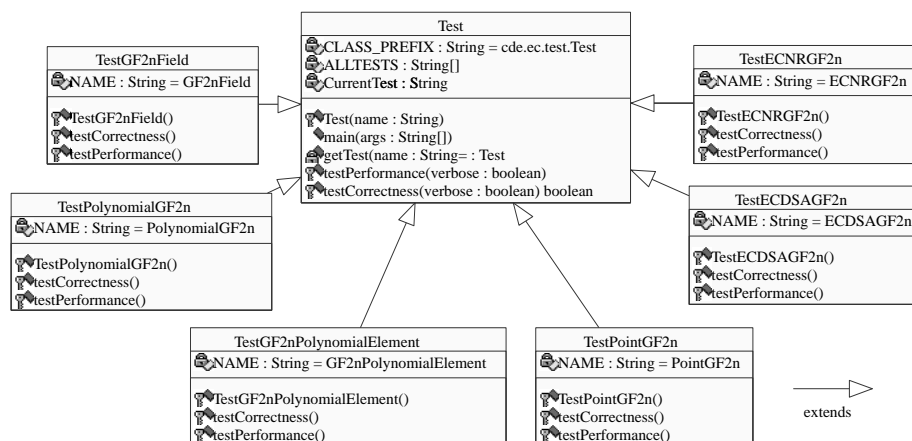
Das Interface `ECDSAPrivateKeyInterface` erbt von den Interfaces `PrivateKey` und `ECDSAKeyInterface` und stellt eine Methode zum Auslesen des privaten Schlüssels  $s$  zur Verfügung

Die Klasse `ECDSAPrivateKey` implementiert schließlich das Interface `ECDSAPrivateKeyInterface` und speichert private ECDSA Schlüssel.

Das Interface `ECDSAPublicKeyInterface` erbt von den Interfaces `PublicKey` und `ECDSAKeyInterface` und stellt eine Methode zum Auslesen des öffentlichen Schlüssels  $W$  zur Verfügung

Die Klasse `ECDSAPublicKey` implementiert schließlich das Interface `ECDSAPublicKeyInterface` und speichert öffentliche ECDSA Schlüssel.

## 4.6 cdc.ec.test



Das package `cdc.ec.test` enthält die Klassen zum Testen der Klassen `GF2nField`, `GF2nPolynomialElement`, `PolynomialGF2n`, `PointGF2n`, `ECNRGF2n` und `ECDSAGF2n`.

Alle Testklassen erben von der ausführbaren abstrakten Klasse `Test`, die den vom Benutzer gewünschten Test lädt und ausführt. Sie definiert dazu die Methoden `testCorrectness()` und `testPerformance()`, die von den Erben überschrieben werden müssen, und die die Korrektheit bzw. die Laufzeit der entsprechenden Testklasse testen.

Die Klasse `Test` wird vom Benutzer mit dem Namen des gewünschten Tests als Kommandozeilenparameter aufgerufen, beispielsweise `Test PointGF2n`, oder `ALL` für alle bekannten Tests. Zusätzlich können die Optionen `-c` und `-p` übergeben werden, die bewirken, dass jeweils nur die Korrektheit oder die Laufzeit getestet wird. Die Option `-v` (`verbose`) sorgt für ausführliche Meldungen, was gerade getestet wird. Ohne diese Option erhält der Benutzer lediglich kurze Informationen und die Aussage, dass eine Klasse korrekt bzw. mit einer bestimmten Geschwindigkeit getestet wurde. Wichtig für die Laufzeitmessung ist das Ausführen der Testklasse auf der HotSpot Virtual Machine.

Die einzelnen Testklassen versuchen die zu testenden Klassen so ausführlich wie möglich "auf Herz und Nieren" zu testen. Dazu werden beispielsweise diverse arithmetische Gesetze (Assoziativität, Distributivität, Kommutativität, etc), Testvektoren und andere Kriterien in vielen Durchläufen und unterschiedlichen Körpergraden überprüft.

Soll eine weitere Testklasse hinzugefügt werden, muss diese lediglich von `Test` erben, den Namen `cdc.ec.test.Test<Name der Tests>` haben, die Methoden `testCorrectness()` und `testPerformance()` implementieren und einen String `NAME` mit dem Namen der Testklasse enthalten. Die Klasse `Test` versucht

bei Ausführung dann, die angegebene Klasse zu laden und auszuführen. Falls sie nicht gefunden werden kann, wird eine Exception geworfen.



# Kapitel 5

## Laufzeiten

Dieses Kapitel befasst sich mit den Laufzeiten einzelner Komponenten der zu dieser Arbeit gehörenden Implementierung. Als Vergleich zur vorliegenden Implementierung dient die C++ Bibliothek LiDIA [LiDIA], die am Institut für Theoretische Informatik der Technischen Universität Darmstadt entwickelt wurde, und die zahlreiche Optimierungen enthält. Viele dieser Optimierungen sind in Java nicht möglich, da entsprechende Anweisungen (Schlüsselwort *inline*, Präprozessor, ...) nicht vorhanden sind. Einen Teil gleicht die Java HotSpot Server Virtual Machine [HotSpot] nach ausreichend vielen Durchläufen wieder aus, indem der Java Bytecode in Maschinencode übersetzt wird und bestimmte Optimierungsstufen durchläuft, zum Beispiel das Ersetzen von Methodenaufrufen durch eine Kopie der entsprechenden Methode (entspricht *inline* in C). Die Funktionsweise von Hotspot ist in Abschnitt 3.2.2 erläutert.

Alle Messungen fanden auf einem Athlon Thunderbird System mit 1 GHz Prozessortakt unter dem Betriebssystem Linux statt. Verwendet wurde das Java Development Kit 1.3 von Sun [Java] mit der Kommandozeilenoption *-server* (HotSpot Server Virtual Machine), sowie der C++ Compiler g++ in Version 2.95.2 (*-O2 -fomit-frame-pointer*) für die LiDIA Implementierung.

Bei Messungen an der Java-Implementierung wurde vor der Messung mehrmals iteriert, um die Optimierungen durch HotSpot anzuwenden. In einer Server-Umgebung entspricht das auch dem normalen Profil, denn die Implementierung läuft eine Zeit lang und wird bis zu einem bestimmten Grad immer weiter optimiert. Auf einem Client-System, das die Implementierung nur sporadisch startet und somit am Anfang durch die fehlende HotSpot Optimierung etwas schlechtere Laufzeiten hat, fällt das nicht so stark ins Gewicht, denn sobald ein höherer Durchsatz benötigt wird, greifen die Optimierungen auch wieder.

Die folgenden Tabellen zeigen die Basisoperationen für Elemente aus  $GF(2^n)$  in Polynomdarstellung und unterschiedlichen Körpergraden. Alle Werte sind in Mikrosekunden ( $10^{-6}$  Sekunden) angegeben.

160 Bit	Java Implementierung	LiDIA C++ Implementierung	Faktor
Multiplikation	19,4 $\mu$ s	5 $\mu$ s	3,88
Addition	0,862 $\mu$ s	1 $\mu$ s	0,862
Invertierung	80,12 $\mu$ s	32,4 $\mu$ s	2,47
Quadrierung	0,837 $\mu$ s	0,5 $\mu$ s	1,67

320Bit	Java Implementierung	LiDIA C++ Implementierung	Faktor
Multiplikation	65,462 $\mu$ s	13,2 $\mu$ s	4,96
Addition	1,32 $\mu$ s	1 $\mu$ s	1,32
Invertierung	161,04 $\mu$ s	88,2 $\mu$ s	1,83
Quadrierung	1,382 $\mu$ s	0,9 $\mu$ s	1,54

1000 Bit	Java Implementierung	LiDIA C++ Implementierung	Faktor
Multiplikation	335,5 $\mu$ s	57,5 $\mu$ s	5,83
Addition	1,88 $\mu$ s	1,5 $\mu$ s	1,25
Invertierung	890,3 $\mu$ s	567,7 $\mu$ s	1,57
Quadrierung	3,627 $\mu$ s	2,6 $\mu$ s	1,4

Insgesamt zeigen diese Werte, dass Java-Code dank HotSpot sehr schnell werden kann, lediglich die Multiplikation ist nennenswert langsamer als in der LiDIA Implementierung. Der Grund dafür liegt vermutlich an der Rekursion, die nur teilweise aufgelöst wurde und der Nutzung von vorberechneten Tabellen durch LiDIA. Beide Implementierungen nutzen eine Karatsuba-Multiplikation, bei LiDIA sind die Methoden für die jeweils halbe Bitlänge jedoch *inline* in die anderen Methoden integriert. Insgesamt existiert also bis zu einer bestimmten Größe nur eine Methode, die die gesamte Multiplikation in einem Anlauf durchführt, während bei Java viele Methodenaufrufe dazwischenliegen, die die Multiplikation ausbremsen.

Die folgende Tabelle zeigt die Gesamtlaufzeiten der Signaturverfahren ECDSA und ECNR. Alle Werte sind in Millisekunden ( $10^{-3}$  Sekunden) angegeben.

191 Bit	Schlüssel- generierung	Einmalschlüssel- erzeugung	Signieren	Verifizieren
ECDSA	24 ms	24 ms	0,391 ms	105 ms
ECNR	24 ms	24 ms	0,25 ms	104 ms

Die Schlüsselgenerierung ist das Erzeugen eines gültigen Schlüsselpaares, bestehend aus einem privaten Schlüssel zum Signieren und einem öffentlichen Schlüssel zum Verifizieren der Signatur. Dieser Vorgang muss nur einmal für jede Instanz, die signieren soll, durchgeführt werden. Die Einmalschlüssel müssen für jede Signatur gesondert erzeugt werden und dürfen auch nur einmal benutzt werden. Sie können jedoch schon vor dem Signiervorgang vorliegen, bevor das zu unterzeichnende Dokument bekannt ist. Das eigentliche Signieren geht dann recht schnell.

ECNR ist beim Verifizieren schneller, da hier nur zwei Punktmultiplikationen und eine Addition durchgeführt werden, beim ECDSA wird zusätzlich vorher noch ein Integer modulo  $r$  invertiert. Das erklärt die unterschiedlichen Zeiten beim Verifizieren.



## Kapitel 6

# Zusammenfassung & Ausblick

Zwei wichtige Anforderungen an diese Arbeit waren, einerseits eine möglichst gute Performance zu erreichen, aber andererseits eine vollständige Implementierung in Java, um möglichst gute Portabilität zu gewährleisten.

Portabilität wurde durch die Nutzung von reinem Java und der Java Cryptography Architecture ermöglicht. Die Integration in den mächtigen CDCECProvider [CDC] erlaubt die einfache Benutzung durch den Anwender. Durch die Benutzung der standardisierten Schnittstellen der JCA steht die Implementierung vielen Anwendungen offen. Der Austausch von Daten, Schlüsseln und Zertifikaten mit anderen Programmen wird durch die konsequente Einhaltung des potenziellen IEEE Standards P1363 garantiert.

Dank der neuen HotSpot Server Virtual Machine konnten gute Performan-cewerte erreicht werden. Ich denke die Zeiten (siehe Kapitel 5) im Vergleich zur C++ Implementierung LiDIA sprechen für sich, besonders die Tatsache, dass LiDIA in einem (zwar unbedeutenden) Punkt unterboten werden konnte.

Will man die Performance noch weiter erhöhen sind folgenden Möglichkeiten denkbar:

- Man kann auf Plattformunabhängigkeit verzichten und mittels JNI eine native Sprache für die eigentliche Arithmetik verwenden, wie in [Hen99] geschehen. Dabei sollte man möglichst die Schnittstellen zur JCA weiterhin verwenden, aber so viel Code wie möglich in der nativen Sprache benutzen, denn jeder Aufruf durch das JNI kostet auf Grund des Over-heads Zeit.
- Eine Möglichkeit die Multiplikation zu beschleunigen, ist sicherlich das Verwenden einer Multithreading Version der Karatsuba Multiplikation und das Benutzen von Mehrprozessorsystemen als Plattform. Die Rekursion der Karatsuba Multiplikation sollte sich recht einfach entsprechend anpassen lassen.
- Eine letzte Möglichkeit, die zur Zeit am Institut auch getestet wird, ist das Einbinden von spezieller Hardware zum Rechnen in der Basisarithmetik.  $GF(2^n)$ -Elemente in ONB-Darstellung lassen sich besonders gut in Hardware umsetzen und beschleunigen. Eine solche Implementierung kann, falls die entsprechende Hardware vorhanden ist, gegebenenfalls die

Elemente in ONB-Darstellung konvertieren, an die Hardware übergeben und das Ergebnis, falls benötigt, wieder zurück konvertieren. Falls zum Ansprechen der Hardware das JNI benutzt werden soll, gilt auch hier, dass man mit möglichst wenigen Aufrufen auskommen sollte, da jeder einzelne wieder durch den Overhead Zeit kostet. Falls keine spezielle Hardware vorhanden ist, kann die vorliegende Softwareimplementierung benutzt werden.

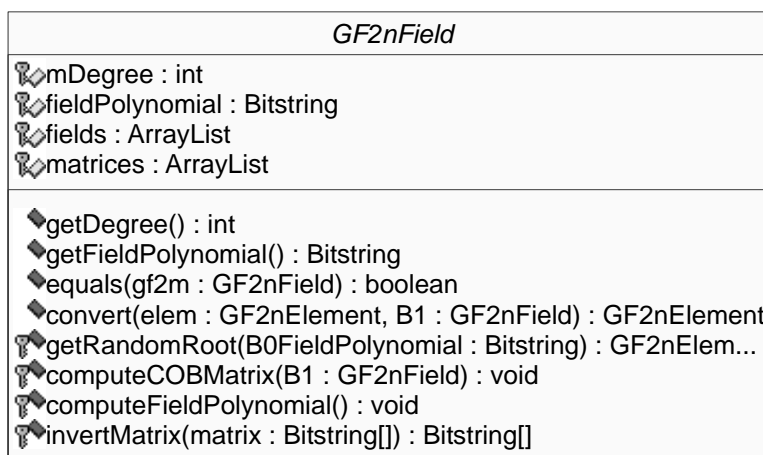
Der in dieser Arbeit benutzte zukünftige IEEE Standard P1363 lag bis zur Fertigstellung der Arbeit leider nur als Draft (Version D13) vor. Es kann nicht ausgeschlossen werden, dass am Standard noch Änderungen vorgenommen werden. Bei Fertigstellung des endgültigen Standards sollte die Implementierung noch einmal auf Konformität geprüft und gegebenenfalls an die endgültige Version angepasst werden, um Austauschbarkeit von Daten mit anderen Anwendungen zu garantieren.

# Anhang A

## Implementierung

### A.1 cdc.ec.arithmetic.gf

#### A.1.1 GF2nField



GF2nField ist eine abstrakte Klasse zur Darstellung von  $GF(2^n)$ -Körpern und speichert dazu den Körpergrad mDegree und das Körperpolynom fieldPolynomial. Außerdem werden Methoden zur Basiskonvertierung von  $GF(2^n)$ -Elementen zur Verfügung gestellt. Erben sind die Klassen GF2nPolynomialElement und GF2nONBElement.

Die Methoden convert, getRandomRoot, computeCOBMatrix und invertMatrix werden zur Basiskonvertierung benutzt. Die Konvertierungsmatrizen werden mit den dazugehörigen GF2nFields in den Attributen fields und matrices gespeichert, so dass die Konvertierungsmatrizen nur einmal für jedes Paar aus GF2nFields berechnet werden muss. Das Inverse der Matrix wird in dem jeweils anderen GF2nField gespeichert. Da Berechnungen in Polynomdarstellung deutlich schneller als in ONB-Darstellung sind, wird die Konvertierungsmatrix immer im beteiligten GF2nPolynomialField berechnet und als Inverse im anderen GF2nField gespeichert.

### Attribute:

`protected int mDegree` Körpergrad  $n$

`protected Bitstring fieldPolynomial` Körperpolynom zur Reduktion

`protected ArrayList fields` eine Liste der `GF2nFields`, in die schon konvertiert wurde und zu denen Konvertierungsmatrizen existieren.

`protected ArrayList matrices` speichert eine Liste der Konvertierungsmatrizen zu bekannten `GF2nFields`.

### Methoden:

`int getDegree()` gibt den Körpergrad  $n$  zurück.

`Bitstring getFieldPolynomial()` gibt eine Kopie des Körperpolynoms als Instanz der Klasse `Bitstring` zurück.

`boolean equals(GF2nField other)` gibt `true` zurück, falls dieser Körper mit dem übergebenen Körper `other` übereinstimmt.

`GF2nElement convert(GF2nElement elem, GF2nField B1)` konvertiert das Element `elem` in die Darstellung des Körpers `B1`.

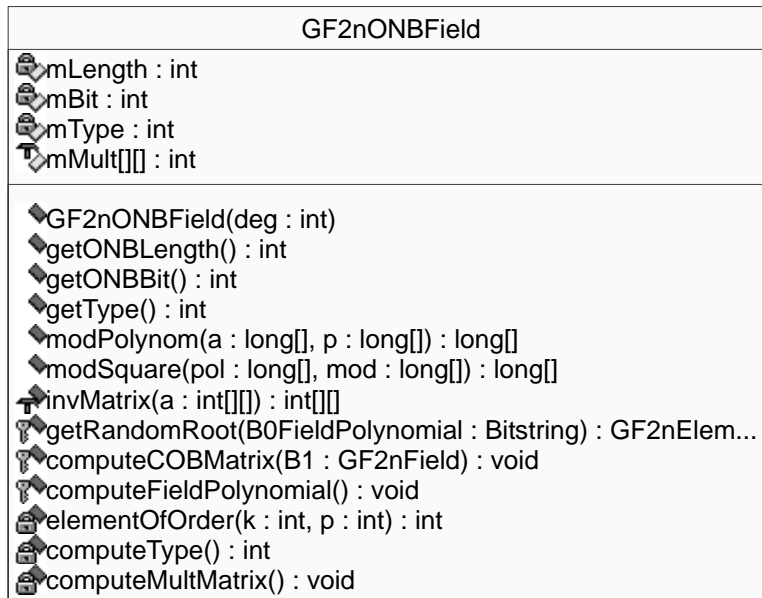
`protected GF2nElement getRandomRoot(Bitstring B0FieldPolynomial)` berechnet eine zufällige Wurzel des Körperpolynoms von `B0` in Darstellung dieses `GF2nFields`.

`protected computeCOBMatrix(GF2nField B1)` berechnet die Matrix zur Konvertierung von Elementen dieses `GF2nFields` nach `B1`.

`protected computeFieldPolynomial()` berechnet das Körperpolynom.

`protected Bitstring[] invertMatrix(Bitstring[] matrix)` invertiert die übergebene Matrix `matrix` und gibt das Ergebnis als `Bitstring-Array` zurück.

### A.1.2 GF2nONBField



GF2nONBField implementiert die abstrakte Klasse GF2nField und definiert einen Körper für  $GF(2^n)$ -Elemente in ONB-Darstellung und stellt Teile der ONB-Basisarithmetik zur Verfügung. Alle Instanzen der Klasse GF2nONBElement speichern einen Verweis zum benutzten GF2nONBField.

Ein großer Teil der Implementierung dieser Klasse wurde aus Birgit Henhapls Arbeit übernommen.

#### Attribute:

`private mLength` gibt die Größe des Arrays aus Basistypen (`long`) an, die benötigt werden, um ein  $GF(2^n)$ -Element in ONB-Darstellung zu speichern.

`private mBit` speichert die Anzahl der Bits im höchstwertigsten Wert des Arrays `mONBPol[mLength-1]`.

`private mType` speichert den Typ der Optimalen Normalbasis.

`package mMult` speichert die Multiplikationsmatrix.

#### Methoden:

`GF2nONBField(int deg)` Konstruktor für eine neue Instanz von `GF2nONBField` vom Grad `deg`

`int getONBLength()` gibt den Wert von `mLength` zurück.

`int getONBBit()` gibt den Wert von `mBit` zurück.

`int getType()` gibt den Typ der Optimalen Normalbasis (`mType`) zurück.

`long[] modPolynom(long[] a, long[] p)` gibt den Rest der Division des Polynoms `a` geteilt durch das Polynom `b` zurück.

`long[] modSquare(long[] pol, long[] mod)` quadriert `pol` modulo `mod` und gibt das Ergebnis zurück.

`protected GF2nElement getRandomRoot(Bitstring B0FieldPolynomial)` berechnet eine zufällige Wurzel des Körperpolynoms von `B0` in Darstellung dieses `GF2nONBFields`.

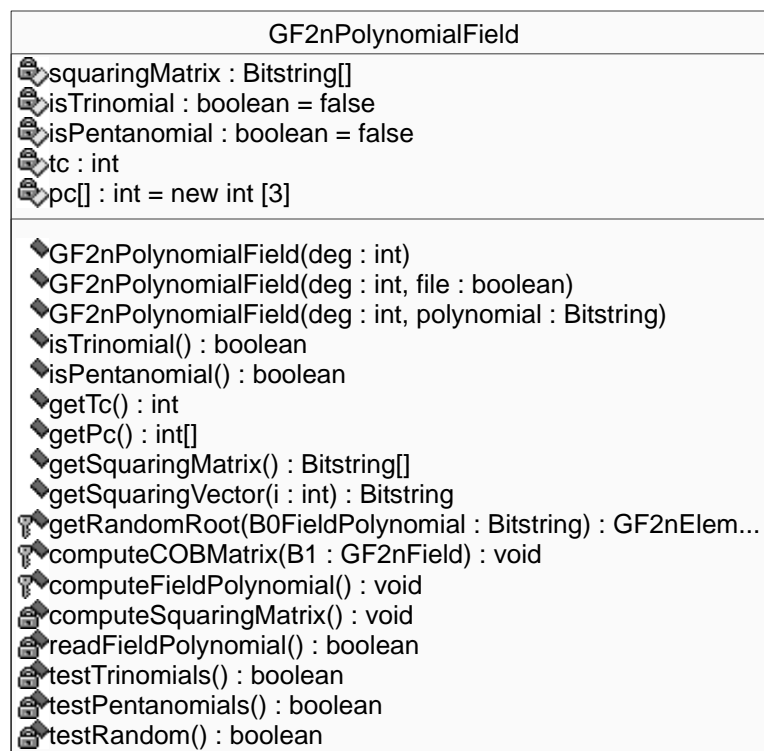
`protected computeCOBMatrix(GF2nField B1)` berechnet die Matrix zur Konvertierung von Elementen dieses `GF2nONBFields` nach `B1`.

`protected computeFieldPolynomial()` berechnet das Körperpolynom.

`private int computeType()` berechnet den Typ der Optimalen Normalbasis.

`private computeMultMatrix()` berechnet die Multiplikationsmatrix für die Multiplikation in ONB-Darstellung.

### A.1.3 GF2nPolynomialField



`GF2nPolynomialField` implementiert die abstrakte Klasse `GF2nField` und stellt einen Körper für  $GF(2^n)$ -Elemente in Polynomdarstellung dar. Alle Instanzen der Klasse `GF2nPolynomialElement` speichern einen Verweis zum benutzten `GF2nPolynomialField`.

### Attribute:

`private Bitstring[] squaringMatrix` speichert die Matrix zum schnellen Quadrieren von `GF2nPolynomialElementen`.

`private boolean isTrinomial` ist `true`, falls das Körperpolynom ein Trinom ist.

`private boolean isPentanomial` ist `true`, falls das Körperpolynom ein Pentanom ist.

`private int tc` speichert den Index des mittleren 1-Koeffizienten des Körperpolynoms (die anderen beiden sind 0 und  $n$ ) bei einem Trinom als Körperpolynom.

`private int[3] pc` speichert die Indices der mittleren 3 1-Koeffizienten des Körperpolynoms (die anderen beiden sind 0 und  $n$ ) bei einem Pentanom als Körperpolynom.

### Methoden:

`GF2nPolynomialField(int deg)` Konstruktor für eine neue Instanz von `GF2nPolynomialField` vom Grad  $n$

`GF2nPolynomialField(int deg, boolean file)` Konstruktor für eine neue Instanz von `GF2nPolynomialField` vom Grad  $n$ . `file` gibt an, ob das Körperpolynom aus der Datei `gf2n.database` gelesen (`true`) oder berechnet werden soll (`false`).

`GF2nPolynomialField(int deg, Bitstring polynomial)` Konstruktor für eine neue Instanz von `GF2nPolynomialField` vom Grad  $n$  mit `polynomial` als Körperpolynom. Standardmäßig wird das übergebene Polynom nicht auf Irreduzibilität geprüft, da das für große  $n$  ( $>150$ ) sehr lange dauern kann.

`boolean isTrinomial()` gibt `true` zurück, falls das verwendete Körperpolynom ein Trinom ist.

`boolean isPentanomial()` gibt `true` zurück, falls das verwendete Körperpolynom ein Pentanom ist.

`int getTc()` gibt den Index des mittleren 1-Koeffizienten des Körperpolynoms (die anderen beiden sind 0 und  $n$ ) bei einem Trinom als Körperpolynom zurück.

`int[] getPc()` gibt die Indices der mittleren 3 1-Koeffizienten des Körperpolynoms (die anderen beiden sind 0 und  $n$ ) bei einem Pentanom als Körperpolynom zurück.

`Bitstring[] getSquaringMatrix()` gibt eine neue Kopie der Matrix zum schnellen Quadrieren von `GF2nPolynomialElementen` zurück.

`Bitstring getSquaringVector(int i)` gibt eine Kopie des Vektors  $i$  der Matrix zum schnellen Quadrieren von `GF2nElementen` zurück.

`protected GF2nElement getRandomRoot(Bitstring B0FieldPolynomial)` berechnet eine zufällige Wurzel des Körperpolynoms von `B0` in Darstellung dieses `GF2nPolynomialFields`.

`protected computeC0BMatrix(GF2nField B1)` berechnet die Matrix zur Konvertierung von Elementen dieses `GF2nPolynomialFields` nach `B1`.

`protected computeFieldPolynomial()` berechnet das Körperpolynom.

`private computeSquaringMatrix()` berechnet die Matrix zum schnellen Quadrieren von `GF2nPolynomialElementen` ohne Reduktion.

`private boolean readFieldPolynomial()` liest das Körperpolynom aus der Datei `gf2n.database` und gibt `true` zurück, `false`, falls das Körperpolynom nicht aus der Datei gelesen werden konnte.

`private boolean testTrinomials()` testet sukzessive alle Trinome dieses Körpers bis ein irreduzibles gefunden wurde und nutzt dieses als Körperpolynom. Gibt `false` zurück, falls kein irreduzibles Trinom für diesen Körper existiert.

`private boolean testPentanomials()` testet sukzessive alle Pentanome dieses Körpers bis ein irreduzibles gefunden wurde und nutzt dieses als Körperpolynom. Gibt `false` zurück, falls kein irreduzibles Pentanom für diesen Körper existiert.

`private boolean testRandom()` testet zufällige Polynome dieses Körpers, bis ein irreduzibles Polynom gefunden wurde. Gibt immer `true` zurück.

### A.1.4 GF2nElement

<i>GF2nElement</i>
🔒 mDegree : int 🔒 mField : GF2nField
◆ getDegree() : int ◆ getField() : GF2nField ◆ isZero() : boolean ◆ isOne() : boolean ◆ equals(other : GF2nElement) : boolean ◆ assignZero() : void ◆ assignOne() : void ◆ ZERO(f : GF2nField) : GF2nElement ◆ ONE(f : GF2nField) : GF2nElement ◆ RANDOM(f : GF2nField) : GF2nElement ◆ copy() : GF2nElement ◆ add(other : GF2nElement) : GF2nElement ◆ subtract(other : GF2nElement) : GF2nElem... ◆ multiply(other : GF2nElement) : GF2nElement ◆ square() : GF2nElement ◆ squareRoot() : GF2nElement ◆ invert() : GF2nElement ◆ increase() : GF2nElement ◆ convert(B1 : GF2nField) : GF2nElement ◆ trace() : int ◆ solveQuadraticEquation() : GF2nElement ◆ toString(radix : int) : String ◆ toString(radix : int, s : String) : String ◆ toBigInteger() : BigInteger ◆ toByteArray() : byte[] ◆ testRightmostBitForP1363() : boolean ◆ testBit(index : int) : boolean

GF2nElement ist eine abstrakte Klasse zur Modellierung von Elementen aus  $GF(2^n)$  in verschiedenen Darstellungen. Sie definiert allgemeine arithmetische Methoden, Methoden solche zur Basiskonvertierung und den Verweis auf den benutzten Körpern GF2nField. Erben sind GF2nONBElement und GF2nPolynomialElement.

#### Attribute:

protected GF2nField mField Verweis auf den von diesem Element benutzten Körper

protected private int mDegree benutzter Körpergrad  $n$

#### Methoden:

GF2nElement ZERO(GF2nField f) erzeugt ein Element zum Körper f mit dem Wert 0.

`GF2nElement ONE(GF2nField f)` erzeugt ein Element zum Körper `f` mit dem Wert 1.

`GF2nElement RANDOM(GF2nField f)` erzeugt ein Element zum Körper `f` mit einem zufälligen Wert.

`GF2nElement copy()` erzeugt eine Kopie dieses Elementes.

`int getDegree()` gibt den Körpergrad  $n$  dieses Elementes zurück.

`GF2nField getField()` gibt einen Verweis auf den benutzten Körper zurück.

`boolean isZero()` gibt `true` zurück, falls das Element den Wert 0 hat.

`boolean isOne()` gibt `true` zurück, falls das Element den Wert 1 hat.

`boolean equals(GF2nElement other)` gibt `true` zurück wenn dieses Element und `other` gleich sind, also den gleichen Wert haben und den gleichen Körpern nutzen.

`assignZero()` weist dem Element den Wert 0 zu.

`assignOne()` weist dem Element den Wert 1 zu.

`GF2nElement add(GF2nElement other)` addiert dieses Element und `other` und gibt das Ergebnis in einem neuen Element zurück.

`GF2nElement subtract(GF2nElement other)` subtrahiert `other` von diesem Element und gibt das Ergebnis in einem neuen Element zurück.

`GF2nElement multiply(GF2nElement other)` multipliziert dieses Element und `other` und gibt das Ergebnis in einem neuen Element zurück.

`GF2nElement square()` quadriert dieses Element und gibt das Ergebnis in einem neuen Element zurück.

`GF2nElement squareRoot()` berechnet die Quadratwurzel dieses Elementes und gibt das Ergebnis in einem neuen Element zurück.

`GF2nElement invert()` invertiert dieses Element und gibt das Ergebnis in einem neuen Element zurück.

`GF2nElement increase()` erzeugt ein neues Element mit dem Wert dieses Elementes erhöht um 1.

`GF2nElement convert(GF2nField B1)` gibt dieses Element konvertiert in die Darstellung von `B1` zurück.

`int trace()` berechnet den Trace dieses Elementes.

`GF2nElement solveQuadraticEquation()` löst die quadratische Gleichung  $z^2 + z = this$  und gibt  $z$  zurück.

`String toString(int radix)` gibt einen String, der den Wert dieses Elementes darstellt in der Basis `radix` zurück.

`String toString(int radix, String s)` gibt einen String, der den Wert dieses Elementes darstellt in der Basis `radix` zurück und konkateniert `s`.

`BigInteger toBigInteger()` konvertiert dieses Element entsprechend FE2IP aus P1363 in einen Integer.

`byte[] toByteArray()` konvertiert dieses Element entsprechend FE2OSP aus P1363 in einen Octet-String.

`boolean testRightmostBitForP1363()` gibt `true` zurück, falls das am weitesten rechts gelegene Bit den Wert 1 hat, `false` sonst.

`boolean testBit(int index)` gibt `true` zurück, falls das Bit mit dem Index `index` den Wert 1 hat, `false` sonst.

### A.1.5 GF2nONBElement



GF2nONBElement implementiert GF2nElement und modelliert Elemente aus  $GF(2^n)$  in ONB-Darstellung.

Hier werden nur die zusätzlichen, nicht geerbten Methoden beschrieben.

Ein großer Teil der Implementierung dieser Klasse wurde aus Birgit Henhapls Arbeit übernommen.

#### Attribute:

protected int mLength gibt die Größe des Arrays aus Basistypen (long) an,

die benötigt werden, um ein Element zu speichern.

`protected int mBit` speichert die Anzahl der Bits im höchstwertigsten Wert des Arrays `mNBPol[mLength-1]`.

`protected long[] mPol` speichert den Wert des Elementes in einem Array aus `long`.

### Methoden:

`GF2n0NBElement(GF2n0NBField gf2n, String value)` Konstruktor für ein neues Element mit Körper `gf2n`. Der Wert entspricht dem übergebenen String `value`, zulässige Werte sind `'ZERO'`, `'ONE'` und `'RANDOM'`.

`GF2n0NBElement(GF2n0NBField gf2n, byte[] e)` Konstruktor für ein neues Element mit Körper `gf2n` und dem Wert von `e` entsprechend zu OS2FEP aus P1363 konvertiert.

`GF2n0NBElement(GF2n0NBField gf2n, BigInteger val)` Konstruktor für ein neues Element mit Körper `gf2n` und dem Wert von `val` entsprechend zu I2FEP aus P1363 konvertiert.

`private GF2n0NBElement(GF2n0NBField gf2n, long[] val)` Konstruktor für ein neues Element mit Körper `gf2n` und dem Wert von `val`.

`GF2n0NBElement(GF2nElement gf2n)` Konstruktor für ein neues Element aus einer Kopie von `gf2n`

`assign(BigInteger val)` konvertiert den Wert von `val` entsprechend zu I2FEP aus P1363 und weist ihn diesem Element zu.

`assign(long[] val)` weist diesem Element den Wert von `val` zu.

`assign(byte[] val)` konvertiert den Wert von `val` entsprechend zu OS2FEP aus P1363 und weist ihn diesem Element zu.

`package long[] getElement()` gibt den Wert dieses Elementes in einem `long`-Array zurück.

`private long[] getElementReverseOrder()` gibt den Wert dieses Elementes mit vertauschter Bitordnung in einem `long`-Array zurück.

`package reverseOrder()` vertauscht die Bitordnung dieses Elementes.

`package setBit(int index)` setzt das Bit an Stelle `index`.

## A.1.6 GF2nPolynomialElement

GF2nPolynomialElement
<ul style="list-style-type: none"> <li>bitstring : Bitstring</li> <li>GF2nPolynomialElement(f : GF2nPolynomialField)</li> <li>GF2nPolynomialElement(f : GF2nPolynomialField, value : String)</li> <li>GF2nPolynomialElement(f : GF2nPolynomialField, rand : SecureRand...)</li> <li>GF2nPolynomialElement(f : GF2nPolynomialField, bs : Bitstring)</li> <li>GF2nPolynomialElement(f : GF2nPolynomialField, bi : BigInteger)</li> <li>GF2nPolynomialElement(f : GF2nPolynomialField, os : byte[])</li> <li>GF2nPolynomialElement(f : GF2nPolynomialField, is : int[])</li> <li>GF2nPolynomialElement(b : GF2nElement)</li> <li>ONE(f : GF2nField) : GF2nElement</li> <li>ZERO(f : GF2nField) : GF2nElement</li> <li>RANDOM(f : GF2nField) : GF2nElement</li> <li>copy() : GF2nElement</li> <li>toString(radix : int) : String</li> <li>toByteArray() : byte[]</li> <li>toIntegerArray() : int[]</li> <li>toBigInteger() : BigInteger</li> <li>getBitstring() : Bitstring</li> <li>toBitstring() : Bitstring</li> <li>assignZero() : void</li> <li>assignOne() : void</li> <li>randomize() : void</li> <li>randomize(rand : SecureRandom) : void</li> <li>equals(b : GF2nElement) : boolean</li> <li>isZero() : boolean</li> <li>isOne() : boolean</li> <li>testBit(index : int) : boolean</li> <li>testRightmostBitForP1363() : boolean</li> <li>addToThis(b : GF2nElement) : void</li> <li>add(b : GF2nElement) : GF2nElement</li> <li>increase() : GF2nElement</li> <li>increaseThis() : void</li> <li>multiplyThisBy(b : GF2nElement) : void</li> <li>multiply(b : GF2nElement) : GF2nElement</li> <li>multiplyThisClassicBy(b : GF2nPolynomialElement) : void</li> <li>multiplyClassic(b : GF2nPolynomialElement) : GF2nPolynomialElement</li> <li>invert() : GF2nElement</li> <li>invertEEA() : GF2nPolynomialElement</li> <li>invertSquare() : GF2nPolynomialElement</li> <li>invertMAIA() : GF2nPolynomialElement</li> <li>squareThis() : void</li> <li>square() : GF2nElement</li> <li>squareMatrix() : GF2nPolynomialElement</li> <li>squareThisMatrix() : void</li> <li>squareThisBitwise() : void</li> <li>squareBitwise() : GF2nPolynomialElement</li> <li>squareThisPreCalc() : void</li> <li>squarePreCalc() : GF2nPolynomialElement</li> <li>power(k : int) : GF2nPolynomialElement</li> <li>gcd(b : GF2nPolynomialElement) : GF2nPolynomialElement</li> <li>squareRoot() : GF2nElement</li> <li>solveQuadraticEquation() : GF2nElement</li> <li>trace() : int</li> <li>halfTrace() : GF2nPolynomialElement</li> <li>reduce() : void</li> <li>reduceTrinomialBitwise(tc : int) : void</li> <li>reducePentanomialBitwise(pc : int[]) : void</li> </ul>

`GF2nPolynomialElement` implementiert `GF2nElement` und modelliert Elemente aus  $GF(2^n)$  in Polynomdarstellung. Die Koeffizienten des Polynoms werden in einer Instanz der Klasse `Bitstring` gespeichert.

Hier werden nur die zusätzlichen, nicht geerbten Methoden beschrieben.

#### Attribute:

`private Bitstring bitstring` speichert die Koeffizienten des Polynoms.

#### Methoden:

`GF2nPolynomialElement(GF2nPolynomialField f)` Konstruktor für ein neues Element mit Körper `f` und dem Wert 0

`GF2nPolynomialElement(GF2nPolynomialField f, String value)` Konstruktor für ein neues Element mit Körper `f`. Der Wert entspricht dem übergebenen String `value`, zulässige Werte sind `'ZERO'`, `'ONE'`, `'X'` und `'RANDOM'`.

`GF2nPolynomialElement(GF2nPolynomialField f, SecureRandom rand)` Konstruktor für ein neues Element mit Körper `f` und zufälligem Wert. Als Quelle wird der in `rand` übergebene Zufallsgenerator genutzt.

`GF2nPolynomialElement(GF2nPolynomialField f, Bitstring bs)` Konstruktor für ein neues Element mit Körper `f` und dem übergebenen Bitstring `bs` als Wert

`GF2nPolynomialElement(GF2nPolynomialField f, BigInteger bi)` Konstruktor für ein neues Element mit Körper `f` und dem Wert von `bi` entsprechend zu I2FEP aus P1363 konvertiert

`GF2nPolynomialElement(GF2nPolynomialField f, byte[] os)` Konstruktor für ein neues Element mit Körper `f` und dem Wert von `os` entsprechend zu OS2FEP aus P1363 konvertiert

`GF2nPolynomialElement(GF2nPolynomialField f, int[] is)` Konstruktor für ein neues Element mit Körper `f` und dem Wert des übergebenen `int[] is`

`GF2nONEElement(GF2nElement b)` Konstruktor für ein neues Element aus einer Kopie von `b`

`int[] toIntegerArray()` gibt ein `int[]` entsprechend dem Wert dieses Elementes zurück.

`Bitstring getBitstring()` gibt den Wert dieses Elementes in einem neuen Bitstring zurück.

`Bitstring toBitstring()` gibt den Wert dieses Elementes in einem neuen Bitstring zurück.

`randomize()` weist diesem Element einen zufälligen Wert zu. Wurde dieses Element mithilfe von `GF2nPolynomialElement(GF2nPolynomialField f, SecureRandom rand)` erzeugt, wird der bei dieser Gelegenheit übergebene `SecureRandom rand` benutzt, ansonsten eine neue Instanz von `Random`.

`addToThis(GF2nElement b)` addiert `b` zu diesem Element dazu.

`increaseThis()` erhöht den Wert dieses Elementes um 1.

`multiplyThisBy(GF2nElement b)` multipliziert dieses Element mit `b`.

`GF2nPolynomialElement multiplyClassic(GF2nPolynomialElement b)` multipliziert dieses Element mit `b` und gibt das Ergebnis als neues `GF2nPolynomialElement` zurück. Dazu wird die Schulmethode benutzt und nicht die schnellere Karatsuba-Multiplikation.

`multiplyThisClassic(GF2nPolynomialElement b)` multipliziert dieses Element mit `b`. Dazu wird die Schulmethode benutzt und nicht die schnellere Karatsuba-Multiplikation.

`GF2nPolynomialElement invertEEA()` invertiert dieses Element durch den erweiterten Euklidischen Algorithmus und gibt das Ergebnis in einem neuen `GF2nPolynomialElement` zurück.

`GF2nPolynomialElement invertSquare()` invertiert dieses Element durch Quadrieren und gibt das Ergebnis in einem neuen `GF2nPolynomialElement` zurück.

`GF2nPolynomialElement invertMAIA()` invertiert dieses Element durch den *Modified Almost Inverse Algorithm* und gibt das Ergebnis in einem neuen `GF2nPolynomialElement` zurück.

`squareThis()` quadriert dieses Element mit dem schnellsten Algorithmus, `squareThisPreCalc()`.

`GF2nPolynomialElement squareMatrix()` quadriert dieses Element mit Hilfe der im `GF2nPolynomialField` gespeicherten Matrix und gibt das Ergebnis in einem neuen `GF2nPolynomialElement` zurück.

`GF2nPolynomialElement squareThisMatrix()` quadriert dieses Element mit Hilfe der im `GF2nPolynomialField` gespeicherten Matrix.

`GF2nPolynomialElement squareBitwise()` quadriert dieses Element durch Linksshiften der Koeffizienten und gibt das Ergebnis in einem neuen `GF2nPolynomialElement` zurück.

`GF2nPolynomialElement squareThisBitwise()` quadriert dieses Element durch Linksshiften der Koeffizienten.

`GF2nPolynomialElement squarePreCalc()` quadriert dieses Element durch Ersetzen der einzelnen Integers im Bitstring mit vorberechneten Werten und gibt das Ergebnis in einem neuen `GF2nPolynomialElement` zurück.

`GF2nPolynomialElement squareThisPreCalc()` quadriert dieses Element durch Ersetzen der einzelnen Integers im Bitstring mit vorberechneten Werten und gibt das Ergebnis in einem neuen `GF2nPolynomialElement` zurück.

`GF2nPolynomialElement power(int k)` berechnet die `k`-te Potenz dieses Elementes und gibt das Ergebnis in einem neuen `GF2nPolynomialElement` zurück.

`GF2nPolynomialElement halfTrace` berechnet den Halftrace dieses Elementes und gibt das Ergebnis in einem neuen `GF2nPolynomialElement` zurück.

`reduce()` reduziert dieses Element mit Hilfe des Körperpolynoms aus dem `GF2nPolynomialField`.

`private reduceTrinomialBitwise(int tc)` reduziert dieses Element falls das Körperpolynoms aus dem `GF2nPolynomialField` ein Trinom ist.

`private reducePentanominalBitwise(int tc)` reduziert dieses Element falls das Körperpolynoms aus dem `GF2nPolynomialField` ein Pentanom ist.

## A.1.7 Bitstring

Bitstring
len : int
blocks : int
value[] : int
<ul style="list-style-type: none"> <li>◆ Bitstring(length : int)</li> <li>◆ Bitstring(length : int, rand : SecureRandom)</li> <li>◆ Bitstring(length : int, value : String)</li> <li>◆ Bitstring(length : int, bs : int[])</li> <li>◆ Bitstring(length : int, os : byte[])</li> <li>◆ Bitstring(length : int, bi : BigInteger)</li> <li>◆ Bitstring(b : Bitstring)</li> <li>◆ getLength() : int</li> <li>◆ toIntegerArray() : int[]</li> <li>◆ toString(radix : int) : String</li> <li>◆ toByteArray() : byte[]</li> <li>◆ toBigInteger() : BigInteger</li> <li>◆ assignOne() : void</li> <li>◆ assignX() : void</li> <li>◆ assignAll() : void</li> <li>◆ assignZero() : void</li> <li>◆ randomize() : void</li> <li>◆ randomize(srandom : SecureRandom) : v...</li> <li>◆ equals(b : Bitstring) : boolean</li> <li>◆ isZero() : boolean</li> <li>◆ isOne() : boolean</li> <li>◆ addToThis(b : Bitstring) : void</li> <li>◆ add(b : Bitstring) : Bitstring</li> <li>◆ subtractFromThis(b : Bitstring) : void</li> <li>◆ subtract(b : Bitstring) : Bitstring</li> <li>◆ increaseThis() : void</li> <li>◆ increase() : Bitstring</li> <li>◆ multiplyClassic(b : Bitstring) : Bitstring</li> <li>◆ multiply(b : Bitstring) : Bitstring</li> <li>◆ karaMult(b : Bitstring) : Bitstring</li> <li>◆ mult128(a : int[], b : int[]) : int[]</li> <li>◆ mult64(a : int[], b : int[]) : int[]</li> <li>◆ mult16(a : short, b : short) : int</li> <li>◆ mult32(a : int, b : int) : int[]</li> <li>◆ remainder(g : Bitstring) : Bitstring</li> <li>◆ quotient(g : Bitstring) : Bitstring</li> <li>◆ divide(g : Bitstring) : Bitstring[]</li> <li>◆ gcd(g : Bitstring) : Bitstring</li> <li>◆ isIrreducible(r : int) : boolean</li> <li>◆ reduceN() : void</li> <li>◆ expandN(i : int) : void</li> <li>◆ squareThisBitwise() : void</li> <li>◆ squareThisPreCalc() : void</li> <li>◆ vectorMult(b : Bitstring) : boolean</li> <li>◆ xor(b : Bitstring) : Bitstring</li> <li>◆ xorThisBy(b : Bitstring) : void</li> <li>◆ setBit(i : int)</li> <li>◆ resetBit(i : int) : void</li> <li>◆ xorBit(i : int) : void</li> <li>◆ testBit(i : int) : boolean</li> <li>◆ shiftLeft() : Bitstring</li> <li>◆ shiftLeftThis() : void</li> <li>◆ shiftLeft(k : int) : Bitstring</li> <li>◆ shiftLeftAddThis(b : Bitstring, k : int) : void</li> </ul>

Die Klasse `Bitstring` speichert lange binäre Strings und stellt eine simple Arithmetik zur Verfügung. Sie wird zur Speicherung der Koeffizienten eines Polynoms in den Klassen `GF2nPolynomialElement`, `GF2nField` und `GF2nPolynomialField` benutzt.

#### Attribute:

`private int[] value` speichert den Bitstring, das niedrigwertigste Bit im Bit 0 von `value[0]`. Aus Performancegründen kann das Array größer sein, als die benötigte Anzahl, beispielsweise nach der Reduktion eines Polynoms.

`private int len` die Länge des Bitstrings, also die Anzahl der gespeicherten Bits

`private int blocks` die Anzahl der benutzten Integers in `value[]`

#### Methoden:

`Bitstring(int length)` Konstruktor für einen neuen Bitstring der Länge `length` und Wert 0

`Bitstring(int length, SecureRandom rand)` Konstruktor für einen neuen Bitstring der Länge `length` und zufälligem Wert. Als Quelle wird der in `rand` übergebene Zufallsgenerator genutzt.

`Bitstring(int length, String value)` Konstruktor für einen neuen Bitstring der Länge `length`. Der Wert entspricht dem übergebenen String `value`, zulässige Werte sind `'ZERO'`, `'ONE'`, `'X'`, `'RANDOM'` und `'ALL'`.

`Bitstring(int length, int[] bs)` Konstruktor für einen neuen Bitstring der Länge `length` und dem in `bs` übergebenen Wert

`Bitstring(int length, byte[] os)` Konstruktor für einen neuen Bitstring der Länge `length` und dem in `os` übergebenen Wert

`Bitstring(int length, BigInteger bi)` Konstruktor für einen neuen Bitstring der Länge `length` und dem in `bi` übergebenen Wert

`Bitstring(Bitstring b)` erzeugt einen neuen Bitstring als Kopie von `b`.

`int getLength()` gibt die Länge `length` des Bitstrings zurück.

`int[] toIntegerArray()` gibt eine Kopie des `int[] value` zurück.

`String toString(int radix)` gibt einen dem Wert entsprechenden String zur Basis `radix` zurück.

`byte[] toByteArray()` konvertiert diesen Bitstring in ein `byte[]`.

`BigInteger toBigInteger()` konvertiert diesen Bitstring in einen `BigInteger`.

`assignONE()` weist diesem Bitstring den Wert 1 zu, setzt also das niederwertigste Bit mit dem Index 0 und setzt alle anderen Bits zurück.

`assignX()` weist diesem Bitstring (als Polynom betrachtet) den Wert `x` zu, setzt also das Bit mit dem Index 1 und setzt alle anderen Bits zurück.

`assignALL()` setzt alle Bits dieses Bitstrings auf 1.

`assignZERO()` setzt alle Bits dieses Bitstrings auf 0.

`randomize()` weist diesem Bitstring einen zufälligen Wert zu, als Quelle wird `Random` benutzt oder der optional dem Konstruktor übergebene `SecureRandom`.

`randomize(SecureRandom rand)` weist diesem Bitstring einen zufälligen Wert zu, als Quelle wird der übergebene `SecureRandom` benutzt.

`boolean equals(Bitstring b)` gibt `true` zurück, falls dieser Bitstring und `b` den gleichen Wert und die gleiche Länge haben, `false` sonst.

`boolean isZero()` gibt `true` zurück, falls alle Bits dieses Bitstrings den Wert 0 haben, `false` sonst.

`boolean isOne()` gibt `true` zurück, falls das niederwertigste Bit dieses Bitstrings gesetzt ist und alle anderen den Wert 0 haben, `false` sonst.

`Bitstring add(Bitstring b)` addiert diesen Bitstring und `b` durch bitweises xor-Verknüpfen und gibt das Ergebnis in einem neuen Bitstring zurück.

`addToThis(Bitstring b)` addiert diesen Bitstring und `b` durch bitweises xor-Verknüpfen.

`Bitstring subtract(Bitstring b)` subtrahiert `b` von diesem Bitstring durch bitweises xor-Verknüpfen und gibt das Ergebnis in einem neuen Bitstring zurück.

`subtractFromThis(Bitstring b)` subtrahiert `b` von diesem Bitstring durch bitweises xor-Verknüpfen.

`Bitstring increase()` erhöht den Wert dieses Bitstrings um 1 durch Kippen des niederwertigsten Bits und gibt das Ergebnis in einem neuen Bitstring zurück.

`increaseThis()` erhöht den Wert dieses Bitstrings um 1 durch Kippen des niederwertigsten Bits.

`Bitstring multiply(Bitstring b)` multipliziert diesen Bitstring mit `b` durch Karatsuba-Multiplikation und gibt das Ergebnis in einem neuen Bitstring zurück.

`private Bitstring karaMult(Bitstring b)` multipliziert diesen Bitstring mit `b` durch Karatsuba-Multiplikation und gibt das Ergebnis in einem neuen Bitstring zurück.

`private int[] mult512(int[] a, int[] b)` multipliziert die beiden 512 Bit langen Integer-Arrays `a` und `b` miteinander und gibt das Ergebnis in einem neuen `int[32]` zurück. Teil der Rekursion der Karatsuba-Multiplikation.

`private int[] mult256(int[] a, int[] b)` multipliziert die beiden 256 Bit langen Integer-Arrays `a` und `b` miteinander und gibt das Ergebnis in einem neuen `int[16]` zurück. Teil der Rekursion der Karatsuba-Multiplikation.

`private int[] mult128(int[] a, int[] b)` multipliziert die beiden 128 Bit langen Integer-Arrays `a` und `b` miteinander und gibt das Ergebnis in einem neuen `int[8]` zurück. Teil der Rekursion der Karatsuba-Multiplikation.

`private int[] mult64(int[] a, int[] b)` multipliziert die beiden 64 Bit langen Integer-Arrays `a` und `b` miteinander und gibt das Ergebnis in einem neuen `int[4]` zurück. Teil der Rekursion der Karatsuba-Multiplikation.

`private int[] mult32(int a, int b)` multipliziert die beiden 32 Bit langen Integers `a` und `b` miteinander und gibt das Ergebnis in einem neuen `int[2]` zurück. Teil der Rekursion der Karatsuba-Multiplikation.

`private int mult16(short a, short b)` multipliziert die beiden 16 Bit langen Shorts `a` und `b` miteinander und gibt das Ergebnis in einem neuen `int` zurück. Letzter Teil der Karatsuba-Multiplikation, führt die eigentliche Berechnung durch.

`Bitstring multiplyClassic(Bitstring b)` multipliziert diesen Bitstring mit `b` mit der Schulmethode und gibt das Ergebnis in einem neuen Bitstring zurück.

`Bitstring remainder(Bitstring g)` berechnet den Rest der Division dieses Bitstrings durch `b` und gibt das Ergebnis in einem neuen Bitstring zurück.

`Bitstring quotient(Bitstring g)` berechnet den Quotienten der Division dieses Bitstrings durch `b` und gibt das Ergebnis in einem neuen Bitstring zurück.

`Bitstring[] divide(Bitstring g)` dividiert diesen Bitstring durch `b` und gibt das Ergebnis in einem neuen `Bitstring[]` zurück. Der Quotient ist am Index 0, der Rest am Index 1 gespeichert.

`Bitstring gcd(Bitstring g)` berechnet den größten gemeinsamen Teiler (greatest common divisor) dieses Bitstrings und `b` und gibt das Ergebnis in einem neuen Bitstring zurück.

`boolean isIrreducible(int r)` gibt `true` zurück, falls dieser Bitstring in  $GF(2^r)$  irreduzibel ist, `false` sonst.

`reduceN()` setzt `len` auf die tatsächlich benutzte Bitlänge, eliminiert also führende 0-Bits. Diese Methode wird unter anderem aus Performancegründen benutzt. Sie führt keine Reduktion von  $GF(2^n)$ -Elementen durch.

`expandN(int i)` erweitert die Bitlänge auf `i` und füllt die führenden Bits mit 0 auf. Diese Methode wird aus Performancegründen und beim `shift-left` benutzt.

`squareThisBitwise()` quadriert diesen Bitstring durch Verschieben der Bits vom Index `i` nach Index `2i`.

`squareThisPreCalc()` quadriert diesen Bitstring durch Nachschlagen in einer vorberechneten Tabelle.

`boolean vectorMult(Bitstring b)` führt eine Vektormultiplikation von diesem und dem übergebenen Bitstring `b` aus und gibt `true` zurück, falls das Ergebnis 1 ist, `false` falls das Ergebnis 0 ist.

`Bitstring xor(Bitstring b)` verknüpft diesen Bitstring und `b` bitweise xor und gibt das Ergebnis in einem neuen Bitstring zurück.

`xorThisBy(Bitstring b)` verknüpft diesen Bitstring und `b` bitweise xor .

`setBit(int i)` setzt das Bit mit dem Index `i` auf den Wert 1.

`resetBit(int i)` setzt das Bit mit dem Index `i` auf den Wert 0.

`xorBit(int i)` kippt das Bit mit dem Index `i`.

`boolean testBit(int i)` gibt `true` zurück, falls das Bit mit dem Index `i` den Wert 1 hat, `false` sonst.


























`Bitstring shiftLeft()` schiebt alle Bits um eine Stelle nach links, setzt das niederwertigste Bit auf den Wert 0 und gibt das Ergebnis in einem neuen Bitstring zurück.

`Bitstring shiftLeft(int k)` schiebt alle Bits um `k` Stellen nach links, setzt das niederwertigste Bit auf den Wert 0 und gibt das Ergebnis in einem neuen Bitstring zurück.

`shiftLeftthis()` schiebt alle Bits um eine Stelle nach links, setzt das niederwertigste Bit auf den Wert 0.

`shiftLeftAddThis(Bitstring b, int k)` schiebt alle Bits um `k` Stellen nach links, setzt das niederwertigste Bit auf den Wert 0, addiert `b` und gibt das Ergebnis in einem neuen Bitstring zurück. Diese Methode wird aus Performancegründen von der Methode `invertEEA()` aus der Klasse `GF2n-PolynomialElement` benutzt.

### A.1.8 PolynomialGF2n

PolynomialGF2n
 size : int
<ul style="list-style-type: none"><li> PolynomialGF2n(deg : int, elem : GF2nElement)</li><li> PolynomialGF2n(deg : int)</li><li> PolynomialGF2n(a : PolynomialGF2n)</li><li> PolynomialGF2n(polynomial : Bitstring, B1 : GF2nField)</li><li> assignZeroToElements() : void</li><li> size() : int</li><li> getDegree() : int</li><li> enlarge(k : int) : void</li><li> shrink() : void</li><li> set(index : int, elem : GF2nElement) : void</li><li> at(index : int) : GF2nElement</li><li> isZero() : boolean</li><li> equals(b : PolynomialGF2n) : boolean</li><li> add(b : PolynomialGF2n) : PolynomialGF2n</li><li> scalarMultiply(s : GF2nElement) : PolynomialGF2n</li><li> multiply(b : PolynomialGF2n) : PolynomialGF2n</li><li> multiplyAndReduce(b : PolynomialGF2n, g : PolynomialGF2n) : PolynomialGF...</li><li> reduce(g : PolynomialGF2n) : PolynomialGF2n</li><li> shiftThisLeft(amount : int) : void</li><li> shiftLeft(amount : int) : PolynomialGF2n</li><li> divide(b : PolynomialGF2n) : PolynomialGF2n[]</li><li> remainder(b : PolynomialGF2n) : PolynomialGF2n</li><li> quotient(b : PolynomialGF2n) : PolynomialGF2n</li><li> gcd(g : PolynomialGF2n) : PolynomialGF2n</li></ul>

Die Klasse `PolynomialGF2n` modelliert Polynome über `GF2nElementen` und stellt eine entsprechende Arithmetik bereit. Sie wird beim Finden einer Wurzel zur Basiskonvertierung von den Klassen `GF2nField`, `GF2n0NBField` und `GF2nPolynomialField` benötigt.

#### Attribute

`int size` gibt den maximalen Grad des Polynoms an.

`GF2nElement[] coeff` speichert die Koeffizienten des Polynoms.

#### Methoden:

`PolynomialGF2n(int deg, GF2nElement elem)` Konstruktor für ein neues Polynom vom Grad `deg` und `elem` für alle Koeffizienten

`PolynomialGF2n(Bitstring polynomial, GF2nField B1)` Konstruktor für ein neues Polynom vom Grad entsprechend der Länge von `polynomial` und zum Körper `B1` gehörenden Koeffizienten aus Null- bzw. Einselementen, je nach Wert des entsprechenden Bits in `polynomial`.

`private PolynomialGF2n(int deg)` Konstruktor für ein neues Polynom vom Grad `deg` und leeren Koeffizienten

`PolynomialGF2n(PolynomialGF2n a)` erzeugt ein neues Polynom aus einer Kopie von `b`.

`assignZeroToElements()` weist allen Koeffizienten das entsprechende Nullelement zu.

`int size()` gibt die Länge (den maximalen Grad) des Polynoms zurück.

`int getDegree()` gibt den wirklichen Grad des Polynoms zurück (führende Koeffizienten mit dem Wert 0 werden ignoriert).

`enlarge(int k)` vergrößert das Polynom auf die Länge `k`.

`shrink()` verkleinert das Polynom auf die kleinstmögliche Länge, den aktuellen Grad.

`set(int index, GF2nElement elem)` setzt den Koeffizienten mit dem Index `index` auf den Wert von `elem`.

`GF2nElement at(int index)` gibt eine Kopie des Koeffizienten mit dem Index `index` zurück.

`boolean isZero()` gibt `true` zurück, wenn alle Koeffizienten das Nullelement enthalten, `false` sonst.

`boolean equals(PolynomialGF2n b)` gibt `true` zurück, wenn dieses Polynom und `b` die gleichen Koeffizienten und Länge haben, `false` sonst.

`PolynomialGF2n add(PolynomialGF2n b)` addiert `b` zu diesem Polynom und gibt das Ergebnis in einem neuen Polynom zurück.

`PolynomialGF2n scalarMultiply(GF2nElement s)` multipliziert dieses Polynom mit dem Skalar `s` und gibt das Ergebnis in einem neuen Polynom zurück.

`PolynomialGF2n multiply(PolynomialGF2n b)` multipliziert dieses Polynom mit `b` und gibt das Ergebnis in einem neuen Polynom zurück.

`PolynomialGF2n multiplyAndReduce(PolynomialGF2n b, PolynomialGF2n g)` multipliziert dieses Polynom mit `b`, reduziert das Ergebnis modulo `g` und gibt das Ergebnis in einem neuen Polynom zurück.

`PolynomialGF2n reduce(PolynomialGF2n g)` reduziert dieses Polynom modulo `g` und gibt das Ergebnis in einem neuen Polynom zurück.

`PolynomialGF2n shiftLeft(int amount)` schiebt die Koeffizienten dieses Polynoms um `amount` Stellen nach links und gibt das Ergebnis in einem neuen Polynom zurück.

`shiftThisLeft(int amount)` schiebt die Koeffizienten dieses Polynoms um `amount` Stellen nach links.

`PolynomialGF2n[] divide(PolynomialGF2n b)` dividiert dieses Polynom durch `b` und gibt das Ergebnis in einem neuen Array aus 2 Polynomen zurück. Der Quotient findet sich in `PolynomialGF2n[0]` und der Rest in `PolynomialGF2n[1]`.

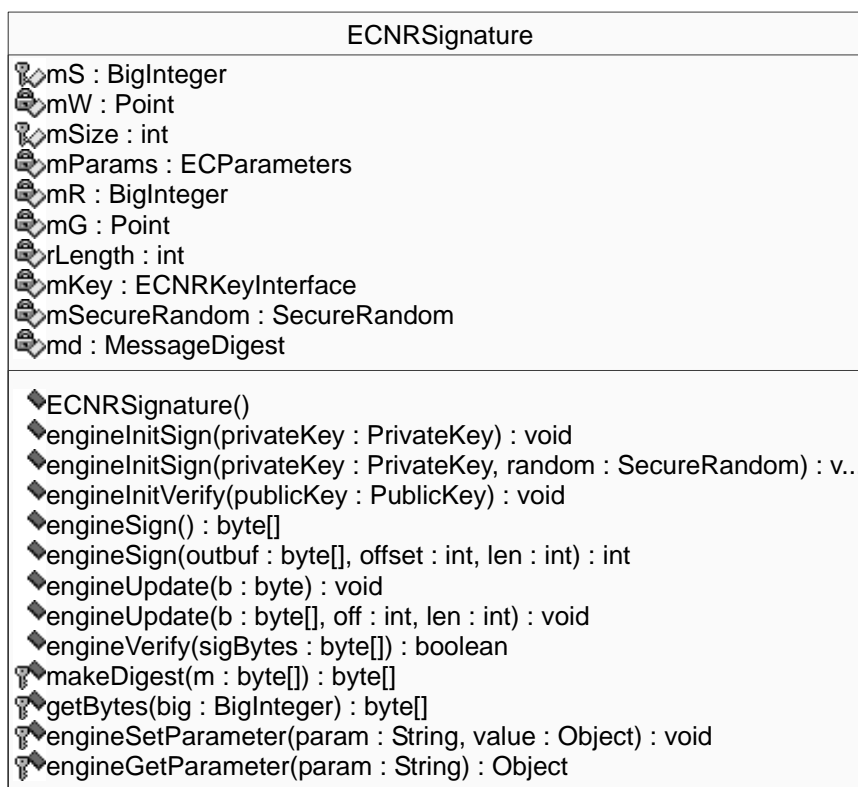
`PolynomialGF2n remainder(PolynomialGF2n b)` dividiert dieses Polynom durch `b` und gibt den Rest in einem neuen Polynomial zurück.

`PolynomialGF2n quotient(PolynomialGF2n b)` dividiert dieses Polynom durch `b` und gibt den Quotienten in einem neuen Polynomial zurück.

`PolynomialGF2n gcd(PolynomialGF2n g)` berechnet den größten gemeinsamen Teiler `gcd` von diesem Polynom und `g` und gibt das Ergebnis in einem neuen Polynomial zurück.

## A.2 cdc.ec.ecnr

### A.2.1 ECNRSignature



Die Klasse `ECNRSignature` erbt von der abstrakten Klasse `Signature` und stellt die Funktionalität der Signatur nach Nyberg und Rueppel zur Verfügung.

#### Attribute:

`BigInteger mS` der private Schlüssel zum Erzeugen einer Signatur

`Point mW` der öffentliche Schlüssel zur Verifikation einer Signatur  
`int mSize` halbe Länge der fertigen Signatur  
`ECParameters mParams` benutzte EC Domainparameter  
`BigInteger mR` EC Domainparameter `r`  
`Point mG` EC Domainparameter `G`  
`int rLength` Bitlänge von `mR`  
`ECNRKeyInterface mKey` benutzter ECNR-Schlüssel  
`SecureRandom mSecureRandom` Zufallsquelle, standardmäßig SHA1PRNG wenn bei der Initialisierung keine andere angegeben wird.  
`MessageDigest md` benutzte Nachrichten Hashfunktion, eine Instanz von SHA

#### Methoden:

`ECNRSignature()` Konstruktor für eine neue Instanz, welche vor Benutzung noch initialisiert werden muss.

`engineInitSign(PrivateKey privateKey)` initialisiert die Instanz zum Signieren und übergibt den zu benutzenden privaten Schlüssel `privateKey`.

`engineInitSign(PrivateKey privateKey, SecureRandom random)` initialisiert die Instanz zum Signieren und übergibt den zu benutzenden privaten Schlüssel `privateKey` und die Zufallsquelle `random`.

`engineInitVerify(PublicKey publicKey)` initialisiert die Instanz zum Verifizieren und übergibt den zu benutzenden öffentlichen Schlüssel `publicKey`.

`byte[] engineSign()` erzeugt die Signatur und gibt das Ergebnis in einem byte-Array zurück.






`int engineSign(byte[] outbuf, int offset, int len)` erzeugt die Signatur, legt sie in dem übergebenen byte-Array `outbuf` an der Stelle `offset` ab und gibt die Anzahl der geschriebenen Bytes zurück. `len` gibt die Anzahl der in `outbuf` für die Signatur reservierten Bytes an. Falls diese nicht ausreichen sollte, wird eine `SignatureException` geworfen.

`engineUpdate(byte b)` fügt das Byte `b` zu den zu signierenden oder verifizierenden Daten hinzu.

`engineUpdate(byte[] b, int off, int len)` fügt `len` Bytes aus dem Byte-Array `b` ab der Stelle `offset` zu den zu signierenden oder verifizierenden Daten hinzu.

`boolean engineVerify(byte[] sigBytes)` überprüft die in `sigBytes` übergebene Signatur auf Gültigkeit und gibt `true` zurück, falls die Signatur gültig ist, `false` sonst.

### A.2.2 ECNRKeyPairGenerator

ECNRKeyPairGenerator	
	mSecureRandom : SecureRandom
	mParams : ECParameters
	initialize(params : AlgorithmParameterSpec, random : SecureRandom) : v...
	initialize(size : int, random : SecureRandom) : void
	generateKeyPair() : KeyPair

Die Klasse ECNRKeyPairGenerator dient zum Erzeugen von neuen ECNR Schlüsselpaaren.

#### Attribute:

SecureRandom mSecureRandom Quelle für Zufallszahlen

ECParameters mParams EC Domainparameter
















Methoden:

initialize(AlgorithmParameterSpec params, SecureRandom random) initialisiert die Instanz mit den übergebenen AlgorithmParameterSpecs, die eine Instanz von ECParameterSpec sein müssen und der Zufallsquelle random.

initialize(int size, SecureRandom random) initialisiert die Instanz mit der übergebenen Bitlänge size und der Zufallsquelle random.

KeyPair generateKeyPair() erzeugt ein neues Schlüsselpaar bestehend aus ECNRPrivateKey und ECNRPublicKey und gibt es zurück.

### A.2.3 ECNRPrivateKey

ECNRPrivateKey	
	mS : BigInteger
	mQ : BigInteger
	mR : BigInteger
	mK : BigInteger
	mE : EllipticCurve
	mG : Point
	mParams : ECParameters
	ECNRPrivateKey(s : BigInteger, params : ECParameters)
	ECNRPrivateKey(s : BigInteger, q : BigInteger, E : EllipticCurve, G : Point, r : BigInteger, k : BigInteger)
	getS() : BigInteger
	getAlgorithm() : String
	getFormat() : String
	getEncoded() : byte[]
	toString() : String
	getParams() : ECParameters

Die Klasse ECNRPrivateKey speichert einen privaten ECNR Schlüssel.

**Attribute:**

`BigInteger mS` der private Schlüssel,  $1 < s < r$

`BigInteger mQ` die Charakteristik des benutzten Körpers,  $2^n$

`BigInteger mR` die Ordnung des Erzeugers  $G$

`BigInteger mK` der Kofaktor

`EllipticCurve mE` benutzte Elliptische Kurve

`Point mG` Erzeuger der Punktegruppe

`ECPParameters mParams` EC Domainparameter

**Methoden:**

`ECNRPrivateKey(BigInteger s, ECPParameters params)` erzeugt einen neuen privaten Schlüssel mit Wert  $s$  und den übergebenen `ECPParameters params`.

`ECNRPrivateKey(BigInteger s, q, EllipticCurve E, Point G, BigInteger r, k)` erzeugt einen neuen privaten Schlüssel mit Wert  $s$  und den übergebenen EC Domainparametern  $q, E, G, r$  und  $k$ .

`BigInteger getS()` gibt den Wert  $s$  des Schlüssels zurück.

`String getAlgorithm()` gibt den Namen des zu diesem Schlüssel gehörenden Algorithmus "ECNR" zurück.

`byte[] getEncoded()` gibt den Schlüssel kodiert nach PKCS#8 zurück.

`String getFormat()` gibt das Format der Kodierung "PKCS#8" zurück .

`String toString()` gibt einen dem Wert des Schlüssels entsprechenden String zurück.

`ECPParameters getParams()` gibt die zu diesem Schlüssel gehörenden `ECPParameters` zurück.

## A.2.4 ECNRPublicKey

ECNRPublicKey
<ul style="list-style-type: none"><li>mQ : BigInteger</li><li>mK : BigInteger</li><li>mR : BigInteger</li><li>mE : EllipticCurve</li><li>mG : Point</li><li>mParams : ECPParameters</li><li>mW : Point</li></ul>
<ul style="list-style-type: none"><li>ECNRPublicKey(w : Point, params : ECPParameters)</li><li>ECNRPublicKey(W : Point, q : BigInteger, E : EllipticCurve, G : Point, r : BigInteger, k : BigInteger)</li><li>getW() : Point</li><li>getAlgorithm() : String</li><li>getFormat() : String</li><li>getEncoded() : byte[]</li><li>getParams() : ECPParameters</li><li>toString() : String</li><li>filterByteArray(barray : byte[]) : byte[]</li></ul>

Die Klasse ECNRPrivateKey speichert einen öffentlichen ECNR Schlüssel.

### Attribute:

Point mW der öffentliche Schlüssel

BigInteger mQ die Charakteristik des benutzten Körpers,  $2^n$

BigInteger mR die Ordnung des Erzeugers  $G$

BigInteger mK der Kofaktor

EllipticCurve mE benutzte Elliptische Kurve

Point mG Erzeuger der Punktgruppe

ECPParameters mParams EC Domainparameter

### Methoden:

ECNRPublicKey(Point W, ECPParameters params) erzeugt einen neuen öffentlichen Schlüssel mit Wert  $s$  und den übergebenen ECPParameters params.

ECNRPublicKey(Point W, BigInteger q, EllipticCurve E, Point G, BigInteger r, k) erzeugt einen neuen öffentlichen Schlüssel mit Wert  $s$  und den übergebenen EC Domainparametern  $q$ ,  $E$ ,  $G$ ,  $r$  und  $k$ .

Point getW() gibt den Wert  $W$  des Schlüssels zurück.

String getAlgorithm() gibt den Namen des zu diesem Schlüssel gehörenden Algorithmus "ECNR" zurück.

byte[] getEncoded() gibt den Schlüssel kodiert nach PKCS#8 zurück.

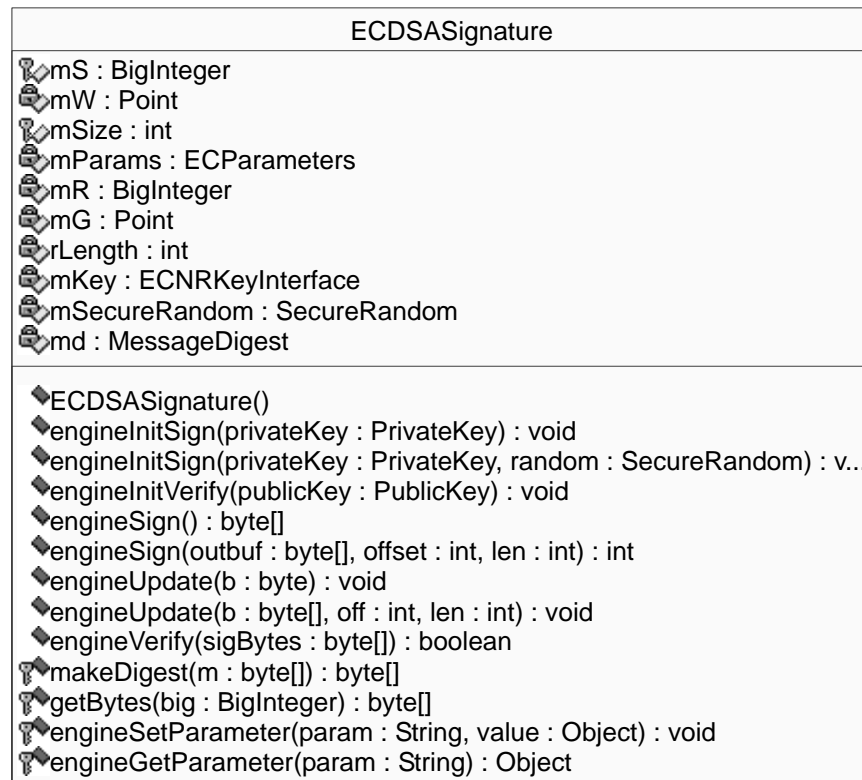
String getFormat() gibt das Format der Kodierung "PKCS#8" zurück.

String toString() gibt einen dem Wert des Schlüssels entsprechenden String zurück.

`ECPParameters getParams()` gibt die zu diesem Schlüssel gehörenden `ECPParameters` zurück.

## A.3 cdc.ec.ecdsa

### A.3.1 ECDSASignature



Die Klasse `ECDSASignature` erbt von der abstrakten Klasse `Signature` und stellt die Funktionalität der Signatur nach Nyberg und Rueppel zur Verfügung.

#### Attribute:

`BigInteger mS` der private Schlüssel zum Erzeugen einer Signatur

`Point mW` der öffentliche Schlüssel zur Verifikation einer Signatur

`int mSize` halbe Länge der fertigen Signatur

`ECPParameters mParams` benutzte EC Domainparameter

`BigInteger mR` EC Domainparameter `r`

`Point mG` EC Domainparameter `G`

`int rLength` Bitlänge von `mR`

`ECDSAKeyInterface mKey` benutzter ECDSA-Schlüssel

`SecureRandom mSecureRandom` Zufallsquelle, standardmäßig `SHA1PRNG` wenn bei der Initialisierung keine andere angegeben wird.

`MessageDigest md` benutzte Nachrichten Hashfunktion, eine Instanz von `SHA`

#### Methoden:

`ECDSASignature()` Konstruktor für eine neue Instanz, welche vor Benutzung noch initialisiert werden muss.

`engineInitSign(PrivateKey privateKey)` initialisiert die Instanz zum Signieren und übergibt den zu benutzenden, privaten Schlüssel `privateKey`.

`engineInitSign(PrivateKey privateKey, SecureRandom random)` initialisiert die Instanz zum Signieren und übergibt den zu benutzenden privaten Schlüssel `privateKey` und die Zufallsquelle `random`.

`engineInitVerify(PublicKey publicKey)` initialisiert die Instanz zum Verifizieren und übergibt den zu benutzenden öffentlichen Schlüssel `publicKey`.

`byte[] engineSign()` erzeugt die Signatur und gibt das Ergebnis in einem `byte`-Array zurück.






`int engineSign(byte[] outbuf, int offset, int len)` erzeugt die Signatur, legt sie in dem übergebenen `byte`-Array `outbuf` an der Stelle `offset` ab und gibt die Anzahl der geschriebenen Bytes zurück. `len` gibt die Anzahl der in `outbuf` für die Signatur reservierten Bytes an. Falls diese nicht ausreichen sollte, wird eine `SignatureException` geworfen.

`engineUpdate(byte b)` fügt das Byte `b` zu den zu signierenden oder verifizierenden Daten hinzu.

`engineUpdate(byte[] b, int off, int len)` fügt `len` Bytes aus dem `byte`-Array `b` ab der Stelle `offset` zu den zu signierenden oder verifizierenden Daten hinzu.

`boolean engineVerify(byte[] sigBytes)` überprüft die in `sigBytes` übergebene Signatur auf Gültigkeit und gibt `true` zurück, falls die Signatur gültig ist, `false` sonst.

### A.3.2 ECDSAKeyPairGenerator

ECDSAKeyPairGenerator	
	<code>mSecureRandom</code> : <code>SecureRandom</code>
	<code>mParams</code> : <code>ECPParameters</code>
	<code>initialize(params : AlgorithmParameterSpec, random : SecureRandom) : void</code>
	<code>initialize(size : int, random : SecureRandom) : void</code>
	<code>generateKeyPair() : KeyPair</code>

Die Klasse `ECDSAKeyPairGenerator` dient zum Erzeugen von neuen ECDSA Schlüsselpaaren.

### Attribute:

SecureRandom mSecureRandom Quelle für Zufallszahlen

ECPParameters mParams EC Domainparameter

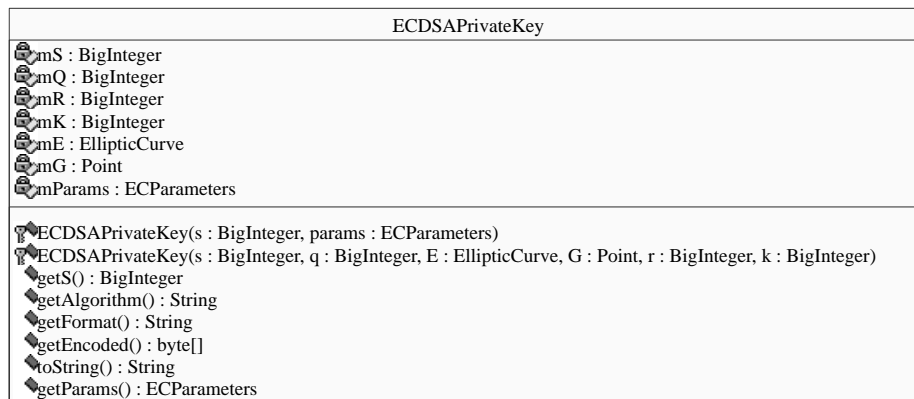
Methoden:

`initialize(AlgorithmParameterSpec params, SecureRandom random)` initialisiert die Instanz mit den übergebenen `AlgorithmParameterSpecs`, die eine Instanz von `ECPParameterSpec` sein müssen und der Zufallsquelle `random`.

`initialize(int size, SecureRandom random)` initialisiert die Instanz mit der übergebenen Bitlänge `size` und der Zufallsquelle `random`.

`KeyPair generateKeyPair()` erzeugt ein neues Schlüsselpaar bestehend aus `ECDSAPrivateKey` und `ECDSAPublicKey` und gibt es zurück.

### A.3.3 ECDSAPrivateKey



Die Klasse `ECDSAPrivateKey` speichert einen privaten ECDSA Schlüssel.

### Attribute:

`BigInteger mS` der private Schlüssel,  $1 < s < r$

`BigInteger mQ` die Charakteristik des benutzten Körpers,  $2^n$

`BigInteger mR` die Ordnung des Erzeugers  $G$

`BigInteger mK` der Kofaktor

`EllipticCurve mE` benutzte Elliptische Kurve

`Point mG` Erzeuger der Punktegruppe

`ECPParameters mParams` EC Domainparameter

## Methoden:

`ECDSAPrivateKey(BigInteger s, ECPParameters params)` erzeugt einen neuen privaten Schlüssel mit Wert `s` und den übergebenen `ECPParameters params`.

`ECDSAPrivateKey(BigInteger s, q, EllipticCurve E, Point G, BigInteger r, k)` erzeugt einen neuen privaten Schlüssel mit Wert `s` und den übergebenen EC Domainparametern `q, E, G, r` und `k`.

`BigInteger getS()` gibt den Wert `s` des Schlüssels zurück.

`String getAlgorithm()` gibt den Namen des zu diesem Schlüssel gehörenden Algorithmus "ECDSA" zurück.

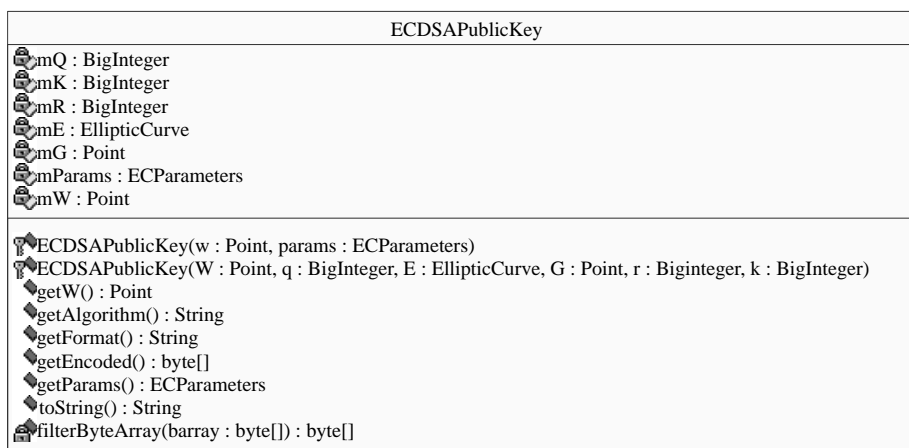
`byte[] getEncoded()` gibt den Schlüssel kodiert nach PKCS#8 zurück.

`String getFormat()` gibt das Format der Kodierung "PKCS#8" zurück.

`String toString()` gibt einen dem Wert des Schlüssels entsprechenden String zurück.

`ECPParameters getParams()` gibt die zu diesem Schlüssel gehörenden `ECPParameters` zurück.

### A.3.4 ECDSAPublicKey



Die Klasse `ECDSAPrivateKey` speichert einen öffentlichen ECDSA Schlüssel.

## Attribute:

`Point mW` der öffentliche Schlüssel

`BigInteger mQ` die Charakteristik des benutzten Körpers,  $2^n$

`BigInteger mR` die Ordnung des Erzeugers `G`

`BigInteger mK` der Kofaktor

`EllipticCurve mE` benutzte Elliptische Kurve

`Point mG` Erzeuger der Punktegruppe

`ECPParameters mParams` EC Domainparameter

#### Methoden:

`ECDSAPublicKey(Point W, ECPParameters params)` erzeugt einen neuen öffentlichen Schlüssel mit Wert `s` und den übergebenen `ECPParameters params`.

`ECDSAPublicKey(Point W, BigInteger q, EllipticCurve E, Point G, BigInteger r, k)` erzeugt einen neuen öffentlichen Schlüssel mit Wert `s` und den übergebenen EC Domainparametern `q, E, G, r` und `k`.

`Point getW()` gibt den Wert `W` des Schlüssels zurück.

`String getAlgorithm()` gibt den Namen des zu diesem Schlüssel gehörenden Algorithmus "ECDSA" zurück.

`byte[] getEncoded()` gibt den Schlüssel kodiert nach PKCS#8 zurück.

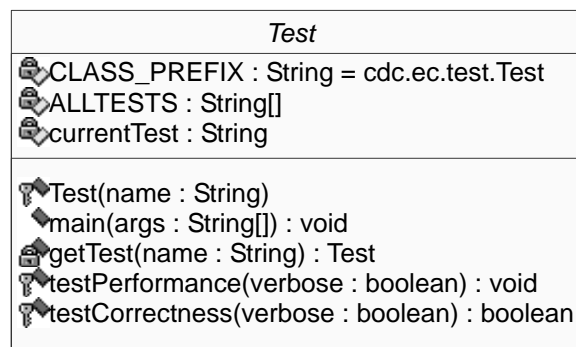
`String getFormat()` gibt das Format der Kodierung "PKCS#8" zurück.

`String toString()` gibt einen dem Wert des Schlüssels entsprechenden String zurück.

`ECPParameters getParams()` gibt die zu diesem Schlüssel gehörenden `ECPParameters` zurück.

## A.4 cdc.ec.test

### A.4.1 Test



Die abstrakte, ausführbare Klasse `Test` erlaubt es dem Benutzer über Kommandozeilenoptionen ausgewählte Klassen zu testen. Die gewählte Klasse muss unter dem Namen `cdc.ec.test.Test <Name>` existieren.

### Attribute:

`private static final String CLASS_PREFIX` enthält das Präfix für den Namen der Testklassen, `"cdc.ec.test.Test"`.

`private static String[] ALLTESTS` enthält eine Liste der Namen aller Testklassen, die bei der Verwendung der Kommandozeilenoption `ALL` getestet werden.

`private static String currentTest` enthält den Namen der gerade ausgewählten Testklasse.

### Methoden:

`protected Test(String name)` Konstruktor für eine neue Instanz von `Test`, muss von einer konkreten Testklasse überschrieben werden. Die Konstrukturen der Nachkommen müssen diesen Konstruktor mit `super(NAME)` aufrufen.

`public static main(String[] args)` Methode, die beim Start der Klasse `Test` vom Benutzer aufgerufen wird. Die Kommandozeilenoptionen werden in `args` übergeben.

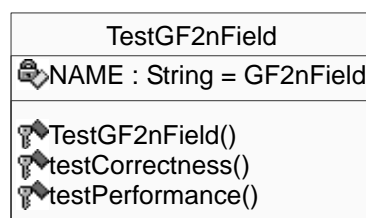
`Test getTest(String name)` versucht die Testklasse mit dem übergebenen Namen zu laden und zu initialisieren und gibt sie bei Erfolg zurück, ansonsten wird eine `ClassNotFoundException` geworfen.

`protected abstract boolean testCorrectness(boolean verbose)` abstrakte Methode, die die Laufzeit der zu testenden Klasse ermittelt. Ist `verbose true` werden ausführliche Informationen ausgegeben, ansonsten nur eine kurze Zusammenfassung am Ende.

`protected abstract testPerformance(boolean verbose)` abstrakte Methode, die die Korrektheit der zu testenden Klasse prüft. Ist `verbose true` werden ausführliche Informationen ausgegeben, ansonsten nur eine kurze Zusammenfassung am Ende.

### A.4.2 TestGF2nField

Als Beispiel aller Testklassen soll hier nur `TestGF2nField` erwähnt werden, alle anderen sind von der Struktur bis auf den Namen gleich.



Attribute:

`private static final String NAME` enthält den Namen der Testklasse ohne das in `Test` definierte Präfix, `"GF2nField"`

**Methoden:**

`protected TestGF2nField(String name)` Konstruktor für eine neue Instanz von `TestGF2nField`, ruft lediglich `super(NAME)` auf.

`protected boolean testCorrectness(boolean verbose)` ermittelt die Laufzeit der Klasse `GF2nField`. Ist `verbose true` werden ausführliche Informationen ausgegeben, ansonsten nur eine kurze Zusammenfassung am Ende.

`protected testPerformance(boolean verbose)` testet die Korrektheit der Klasse `GF2nField`. Ist `verbose true` werden ausführliche Informationen ausgegeben, ansonsten nur eine kurze Zusammenfassung am Ende.

# Anhang B

## Glossar

API	Application Programming Interface
Bytecode	virtueller Maschinencode, zum Beispiel von Java-Quelltexten, der durch einen Interpreter ausgeführt werden kann.
CDCProvider	ein Provider für die JCA, entwickelt am Institut für Theoretische Informatik der Technischen Universität Darmstadt. [CDC]
DSA	Digital Signature Algorithm
EC	Elliptic Curve
ECDSA	Elliptische Kurven Version des Digital Signature Algorithm
ECNR	Elliptische Kurven Version des Signaturverfahrens von Nyberg und Rueppel
FE2IP	Field Element To Integer Primitive, P1363 Konvertierungsmethode von Körperelementen in einen Integer Wert
FE2OSP	Field Element To Octet String Primitive, P1363 Konvertierungsmethode von Körperelementen in ein Array aus Bytes (Octets)
I2FEP	Integer To Field Element Primitive, P1363 Konvertierungsmethode von Integer Werten in ein Körperelement
$GF(2^n)$	Endlicher Körper der Charakteristik 2 mit $2^n$ Elementen
HotSpot	schnelle Java Virtual Machine, die den ausgeführten Bytecode während der Laufzeit optimiert und teilweise in nativen Maschinencode übersetzt.
Java Bytecode	virtueller Maschinencode erzeugt vom Java Übersetzer, der auf einer Virtuellen Maschine ausgeführt werden kann.
JCA	Java Cryptography Architecture
JCE	Java Cryptography Extension
JIT	Just In Time, ein Compiler, der aus Bytecode nativen Maschinencode übersetzt.

JNI	Java Native Interface, die Möglichkeit Programme in nativen Programmiersprachen von Java aus aufzurufen.
LiDIA	[LiDIA]
ONB	Optimale Normalbasen, Repräsentation von Elementen aus $GF(2^n)$
OS2FEP	Octet String To Field Element Primitive, P1363 Konvertierungsmethode von einem Byte(Octet)-Array in ein Körperelement
P1363	IEEE-Standard [P1363] zur Elliptischen Kurven Kryptographie
Polynomdarstellung	auf Polynomen basierende Repräsentation von Elementen aus $GF(2^n)$
RSA	Rivest, Shamir, Adleman, Erfinder des gleichnamigen Public-Key Verschlüsselungsverfahrens
SigG	Signaturgesetz, Gesetz, das in Deutschland die Gültigkeit digitaler Signaturen regelt.
VM	Virtual Machine, ein Programm zur Ausführung von Bytecode auf real existierenden Rechnern.

# Literaturverzeichnis

- [Bar97] G. BARWOOD, “Elliptic Curve Cryptography FAQ”, [http://ds.dial.pipex.com/george.barwood/ec\\_faq.htm](http://ds.dial.pipex.com/george.barwood/ec_faq.htm)
- [Buc99] J.BUCHMANN, “Einführung in die Kryptographie”, Springer Verlag 1999
- [CDC] CDC Standard Provider für die Java Cryptographic Architecture, <http://www.informatik.tu-darmstadt.de/TI/Forschung/cdcProvider/overview.html>
- [Cer] “Elliptic Curve Cryptography”, Certicom Online-Tutorial, <http://www.certicom.com/research.html>
- [Fer00] THOMAS FERTIG, “Digitale Signaturen mit elliptischen Kurven über  $GF(2^n)$ ”, Diplomarbeit, TU Darmstadt, Institut für Theoretische Informatik, <http://www.informatik.tu-darmstadt.de/TI/Forschung/ECC>
- [Fla97] D. FLANAGAN, “Java in a Nutshell”, 2nd edition, O’Reiley, 1997
- [Gor98] D. M. GORDON, “A survey of fast exponentiation methods”, Journal of Algorithms **27** (1998), 129-146, <http://www.idealibrary.com/links/toc/jagm/27/1/0>
- [Ham00] SAFUAT HAMDY, “Anwendungen elliptischer Kurven in der Kryptologie”, TU Darmstadt, Institut für Theoretische Informatik
- [Han] DARREL HANKERSON, JULIO LOPEZ HERNANDEZ, ALFRED MENEZES, “Software Implementation of Elliptic Curve Cryptography Over Binary Fields”, <http://www.cacr.math.uwaterloo.ca>
- [Hen99] BIRGIT HENHAPL, “Digitales Signieren mit Elliptischen Kurven über Primkörpern großer Charakteristik”, Diplomarbeit, TU Darmstadt, Institut für Theoretische Informatik
- [HotSpot] “Java HotSpot Server Virtual Machine”, <http://java.sun.com/products/hotspot/2.0/docs.html>
- [Java] <http://java.sun.com>
- [JCA] “Java Cryptographic Architekture”, <http://java.sun.com/security>

- [Joh99] D.B. JOHNSON, A.J. MENEZES, "The Elliptic Curve Digital Signature Algorithm (ECDSA)", Technical Report CORR 99-31, Dept. of C&O, University of Waterloo, Canada, August 23, 1999, <http://www.cacr.math.uwaterloo.ca>
- [Knu81] D. E. KNUT, "The Art Of Computer Programming. Vol 2: Seminumerical Algorithms", 2nd edition, Addison Wesley, 1981
- [Knu98] J. KNUDSEN, "Java Cryptography", O'Reilley, 1998
- [Kob87] N. KOBLITZ, "Elliptic curve cryptosystems", *Mathematics of Computation*, 48 (1987), 203-209
- [Kob94] N. KOBLITZ, "A First Course in Number Theory and Cryptography", Springer Verlag, 1994
- [Kra93] D.W. KRAVITZ, "Digital signature algorithm", U.S Patent 5,231,668, 27 Jul 1993
- [LiDIA] LiDIA-Group, "LiDIA Manual - A library for computational number theory", TU Darmstadt, Institut für Theoretische Informatik, <http://www.tu-darmstadt.de/TI/LiDIA/Welcome.html>
- [Men93] A.J. MENEZES, "Elliptic Curve Public Key Cryptosystems", Kluwer Academic Publishers, Boston 1993
- [Men97] A.J. MENEZES, P.C VAN OORSCHOT, S.A. VANSTONE, "Handbook of Applied Cryptography", CRC Press, 1997
- [Mil86] V.S. MILLER, "Use of elliptic curves in cryptography", H.C. Williams, editor, *Advances in Cryptology - Crypto '85, Lecture Notes in Computer Science* 218 (1986), Springer Verlag, 417-426
- [NR93] K. NYBERG, R. RUEPPEL "A new signature scheme based on the DSA giving message recovery" *First ACM Conference on Computer and Communication Security* (1993), ACM Press, 58-61
- [P1363] "IEEE P1363 Standard Specifications for Public Key Cryptography (Draft13)", IEEE Specification Draft, <http://grouper.ieee.org/groups/1363/>
- [Ros98] M. ROSING, "Implementing Elliptic Curve Cryptography", Manning Publications Co., 1998
- [Sch95] R. SCHROEPEL, H. ORMAN, S. O'MALLEY, O. SPATSCHECK, "Fast Key Exchange with Elliptic Curve Systems", *Crypto-95*, Springer Verlag,
- [Sma] N.P. SMART, "A Comparison of different Finite Fields for use in Elliptic Curve Cryptosystems"
- [SEC1] "SEC 1: Elliptic Curve Cryptography", Standards For Efficient Cryptography, Certicom Research 2000
- [SEC2] "SEC 2: Recommended Elliptic Curve Domain Parameters", Standards For Efficient Cryptography, Certicom Research 2000

[X9.63] “Public Key Cryptography For The Financial Services Industry”,  
Working Draft, American National Standard X9.63-199x, American  
Bankers Association 1998



**Erklärung zur Diplomarbeit gemäß §19 Abs. 6 DPO/AT**

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

Darmstadt, den 7. Mai 2001

Oliver Seiler