DIPLOMARBEIT

Kryptographie mit Elliptischen Kurven
über Körpern gerader Ordnung
für tief eingebettete Systeme

Robert Escherich

– August, 2005 –

Diese Arbeit wurde mit LaTeX am 6. September 2005 um 12:34 Uhr gesetzt.
Alle abgedruckten Diagramme wurden mit GNUPLOT 4.0 erzeugt.

University of Applied Sciences

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Informatiker(FH)

an der

Georg-Simon-Ohm Fachhochschule Nürnberg

Fachbereich: Informatik Studiengang: Informatik

in Zusammenarbeit mit der Firma 3Soft

Thema:

Kryptographie mit Elliptischen Kurven über Körpern gerader Ordnung für tief eingebettete Systeme.

Abgabedatum: 31. August 2005

Diplomand: Robert Escherich
Walkersbrunn 77
91322 Gräfenberg

Betreuer Dr. Steffen Reith
Firma 3Soft

1. Prüfer: Prof. Dr. Delfs
Fachhochschule Nürnberg

2. Prüfer: Prof. Dr. Knebl
Fachhochschule Nürnberg

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der erwähnten Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Robert Escherich

Für meinen Großvater

Gerhard Weigel

⋆ 01.08.1922 † 28.01.2004

Vorwort

Diese Diplomarbeit entstand im Rahmen meines Informatikstudiums an der *Georg-Simon-Ohm Fachhochschule Nürnberg* in Zusammenarbeit mit der Firma 3Soft, die mir die benötigte Hardware zur Verfügung gestellt hat. Sie beschreibt die Implementierung der Arithmetik über Körpern $\mathbb{F}_{2^n}$ mit gerader Charakteristik, sowie darauf aufbauend die Arithmetik auf Elliptischen Kurven (Punktaddition und -multiplikation) über solchen Körpern. Betrachtet werden jedoch nicht Desktop PCs, sondern resourcenarme Mikroprozessoren mit 8 Bit, 16 Bit oder 32 Bit Registerbreite, sogenannte *embedded* Prozessoren. Diese werden unter anderem im Automobilbereich eingesetzt, dem Anwendungsgebiet bei 3Soft.

Bedanken möchte ich mich besonders bei Herrn Prof. Delfs sowie Herrn Dr. Steffen Reith (3Soft), welche die Betreuung dieses interessanten Themas übernahmen und mir stets mit Rat und Tat zur Seite standen. Mein weiterer Dank gilt Alexander Much und Georg Weber von der Firma 3Soft für Ihre weitreichenden Erläuterungen, sowie Herrn Prof. Knebl für seine Vorlesung und das sehr lehrreiche Vorlesungsskript [Kne04] über Elliptische Kurven in der Kryptographie.

Besonderer Dank gilt meiner Familie und meiner Freundin Anja Hofmann für das aufgebrachte Verständnis, die Aufmunterungen und die Unterstützung während dieser Arbeit.

Nürnberg, im August 2005

Robert Escherich

Inhaltsverzeichnis

Inhaltsverzeichnis ii

Abbildungsverzeichnis iv

Tabellenverzeichnis v

1 Einleitung 1
- 1.1 Aufteilung der Arbeit . . . 2
- 1.2 Zusammenfassung der Ergebnisse . . . 3

2 Arithmetik und Darstellung von $\mathbb{F}_{2^n}$ 5
- 2.1 Mathematische Grundlagen . . . 5
  - 2.1.1 Gruppe . . . 5
  - 2.1.2 Ringe . . . 7
  - 2.1.3 Körper . . . 8
  - 2.1.4 Polynome . . . 9
  - 2.1.5 Endliche Körper . . . 12
  - 2.1.6 Quadratische Gleichungen in Charakteristik 2 . . . 13
- 2.2 Darstellung von $\mathbb{F}_{2^n}$ . . . 15
  - 2.2.1 Polynomielle Darstellung . . . 15
  - 2.2.2 Optimale Normalbasis (ONB) . . . 16
  - 2.2.3 Zusammengesetzte Erweiterungskörper . . . 17
  - 2.2.4 Auswahl der Darstellung . . . 18

3 Implementierung von $\mathbb{F}_{2^n}$ 21
- 3.1 Darstellung der Polynome . . . 21
- 3.2 Arithmetik im Rechner . . . 22
  - 3.2.1 Addition von Polynomen . . . 22
  - 3.2.2 Multiplikation von Polynomen . . . 23
  - 3.2.3 Quadrieren von Polynomen . . . 30
  - 3.2.4 Reduktion modulo $f(x)$ . . . 32
  - 3.2.5 Inversenbildung . . . 35
- 3.3 Vergleich der Implementierungen . . . 42
  - 3.3.1 Der C167 . . . 42
  - 3.3.2 Der ST30 . . . 43

3.3.3 Multiplikationsalgorithmen . . . 43
3.3.4 Quadrieren von Polynomen . . . 44
3.3.5 Reduktionsalgorithmen . . . 46
3.3.6 Inversenbildung . . . 48

4 Einführung in Elliptische Kurven **51**
4.1 Grundlagen der Algebraischen Geometrie . . . 51
4.1.1 Definitionen der ebenen Alg. Geometrie . . . 52
4.1.2 Die Projektive Geometrie . . . 54
4.1.3 Singularitäten und Schnittpunkte . . . 57
4.2 Elliptische Kurven . . . 60
4.2.1 Vereinfachte Weierstrass-Gleichungen . . . 61
4.2.2 Elliptische Kurven über $\mathbb{F}_{2^n}$ . . . 63
4.2.3 Das Gruppengesetz und Punktaddition . . . 65

5 Implementierung der Punktmultiplikation auf Elliptischen Kurven **75**
5.1 Verschiedene Koordinatendarstellungen . . . 75
5.1.1 Projektive Koordinaten . . . 76
5.1.2 Vergleich der Darstellungen . . . 79
5.2 Algorithmen zur Punktmultiplikation . . . 80
5.2.1 Möglichkeiten der Punktaddition . . . 80
5.2.2 Affine Punktmultiplikation . . . 82
5.2.3 Punktmultiplikation mit Projektiven Koordinaten . . . 83
5.2.4 Punktmultiplikation mit NAF . . . 85
5.2.5 B-ary Punktmultiplikation . . . 88
5.2.6 Montgomery Punktmultiplikation . . . 91
5.3 Vergleich der Implementierungen . . . 92
5.3.1 Timingvergleiche . . . 93
5.3.2 Recourcenbedarf . . . 97
5.3.3 Vergleich des ROM Bedarf . . . 99
5.3.4 Fazit . . . 100

6 Kryptosysteme mit Elliptischen Kurven **103**
6.1 Diffie-Hellman Schlüsselaustausch . . . 104
6.2 ElGamal Kryptosystem für Elliptische Kurven . . . 105
6.3 Digitaler Signatur Standard für Elliptische Kurven . . . 106

7 Ausblick **109**

A Punktmultiplikation auf PC-Systemen **III**

B Binäre NIST Kurven **IX**

Literaturverzeichnis **XI**

Stichwortverzeichnis **XX**

Abbildungsverzeichnis

3.1 Darstellung eines Polynoms für $n = 163$ und $W = 16Bit$ 22
3.2 Addition zweier Polynome 23
3.3 *Right-to-Left comb* Multiplikation 25
3.4 Visualisierung der Arraynotation 25
3.5 Verschiedene Multiplikationsalgorithmen im Vergleich 30
3.6 Quadrieren eines Polynoms 31
3.7 Wortweise Reduktion 35
3.8 Multiplikationsalgorithmen auf dem ST30 44
3.9 Multiplikationsalgorithmen auf dem C167 45
3.10 Quadrieralgorithmen auf dem C167 45
3.11 Quadrieralgorithmen auf dem ST30 46
3.12 Reduce-Algorithmen auf dem C167 47
3.13 Reduce-Algorithmen auf dem ST30 47
3.14 Inversenbildung auf dem ST30 48
3.15 Inversenbildung auf dem C167 49

4.1 Beispiele für Singularitäten (im Ursprung) einer Kurve 62
4.2 Drei Beispiele für Elliptische Kurven über $\mathbb{R}$ 63
4.3 Punktaddition veranschaulicht über $\mathbb{R}$ 66
4.4 Die Punkte einer Elliptischen Kurve über $\mathbb{F}_{2^4}$ 73

5.1 Ein Iterationsschritt der Montgomery Punktmultiplikation 92
5.2 Punktmultiplikation auf C167 – Spezielles Reduce – (B-163) 95
5.3 Punktmultiplikation auf C167 – Spezielles Reduce – (B-233) 95
5.4 Punktmultiplikation auf C167 – Allgemeines Reduce – (B-233) 96
5.5 Punktmultiplikation auf ST30 – Spezielles Reduce – (B-163) 98
5.6 Punktmultiplikation auf ST30 – Spezielles Reduce – (B-233) 98
5.7 Punktmultiplikation auf ST30 – Allgemeines Reduce – (B-233) 99

6.1 Diffie Hellman Schlüsselaustausch mit Elliptischen Kurven 104
6.2 ElGamal Kryptosystem mit Elliptischen Kurven 105
6.3 Digitaler Signatur Algorithmus mit Elliptischen Kurven 107

A.1 Punktmultiplikation auf P IV – Spezielles Reduce – (B-163) III
A.2 Punktmultiplikation auf P IV – Allgemeines Reduce – (B-163) V

A.3 Punktmultiplikation auf P IV – Allgemeines Reduce – (B-233) V
A.4 Punktmultiplikation auf P IV – Spezielles Reduce – (B-233) V
A.5 Punktmultiplikation auf Xeon – Spezielles Reduce – (B-163) VI
A.6 Punktmultiplikation auf Xeon – Allgemeines Reduce – (B-163) VII
A.7 Punktmultiplikation auf Xeon – Allgemeines Reduce – (B-233) VII
A.8 Punktmultiplikation auf Xeon – Spezielles Reduce – (B-233) VII

Tabellenverzeichnis

2.1 Darstellung von $\mathbb{F}_{2^4}$ 16

3.1 Lookup-Tabelle für $w = 4$ 27
3.2 Quadrieren: Lookup-Tabelle 32
3.3 Irreduzible Polynome 42
3.4 Timing-Vergleich der Multiplikations-Algorithmen 44
3.5 Timing-Vergleich der Reduce-Algorithmen 46
3.6 Timing-Vergleich der Inversenbildung 48

5.1 Gegenüberstellung Koordinatensysteme 79
5.2 Vergleich des Bedarfs an Körperoperationen 79
5.3 Speicherbedarf Punktaddition 16 Bit 82
5.4 Punktmultiplikations-Algorithmen auf dem C167 94
5.5 Punktmultiplikations-Algorithmen auf dem ST30 97
5.6 Speicherbedarf Punktmultiplikation 16 Bit 100
5.7 ROM-Verbrauch der Punktmult-Algorithmen auf dem C167 101
5.8 ROM-Verbrauch der Punktmult-Algorithmen auf dem ST30 102

A.1 Punktmultiplikations-Algorithmen auf einem P IV IV
A.2 Punktmultiplikations-Algorithmen auf einem Xeon Dual VI

B.1 Binäre NIST Kurve B-163 IX
B.2 Binäre NIST Kurve B-233 X
B.3 Binäre NIST Kurve B-283 X

Kapitel 1

Einleitung

Die Arbeit *New Directions in Cryptography* von Whitfield Diffie und Martin E. Hellman [DH76] aus dem Jahr 1976 wird im allgemeinen als der Beginn der modernen Public-Key Kryptographie betrachtet. Die beiden Autoren beschreiben darin die Verwendung sogenannter *trap-door one-way functions* für den Aufbau eines Public-Key-Kryptosystems folgendermaßen:

> Since in a public key cryptosystem the general system in which E and D are used must be public, specifying E specifies a complete algorithm for transforming input messages into output cryptograms. As such a public key system is really a set of *trap-door one-way functions*. These are functions which are not really one-way in that simply computed inverses exist. But given an algorithm for the forward function it is computationally infeasible to find a simply computed inverse. Only through knowledge of certain trapdoor information (e.g., the random bit string which produced the E-D pair) can one easily find the easily computed inverse.

Dieses von ihnen vorgestellte Verfahren ist unter dem Namen *Diffie-Hellman Schlüsselaustausch* bekannt und wird heute noch in gängigen Verfahren verwendet.
Kurze Zeit später veröffentlichten Rivest, Shamir und Adelman einen Artikel mit dem Namen *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems* [RSA78], in dem sie das mittlerweile bekannteste und nach ihnen benannte *RSA-Verfahren* vorstellten.
Im Jahr 1985 veröffentlichte Taher ElGamal den Artikel *A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms* [ElG85], worin er ein weiteres Public-Key Verfahren vorstellte, welches auf der Schwierigkeit beruht, diskrete Logarithmen in zyklischen Gruppen effektiv berechnen zu können. Der Vorteil des *ElGamal-Verfahrens* besteht darin, dass es sich nicht nur in der primen Restklassengruppe modulo einer Primzahl, sondern in allen zyklischen Gruppen verwenden lässt.

Ebenfalls 1985 wurden auch erstmals Elliptische Kurven als Basis von Kryptosystemen vorgeschlagen. Neal Koblitz [Nea87] und Victor S. Miller [Mil86] veröffentlichten unabhängig voneinander zwei Artikel, in denen sie das Diffie-Hellman Verfahren in

der Punktgruppe einer Elliptischen Kurve vorstellten. Zunächst fanden Kryptosysteme basierend auf Elliptischen Kurven (ECC) in der Praxis keine Anwendung, was an der komplizierten und bis dato in der Kryptographie relativ unbekannten Mathematik der Elliptischen Kurven liegen mochte. Mittlerweile hat sich dies geändert und man findet immer häufiger Implementierungen dieser Kryptosysteme. Der Nachteil einer relativ komplizierten Mathematik wird durch wesentlich kürzere Schlüssellängen im Vergleich zum RSA- oder ElGamal-Verfahren mehr als ausgeglichen.
Gerade diese kurzen Schlüssellängen machen eine Verwendung auf Mikroprozessoren in sogenannten *embedded Systemen* sehr interessant, da man hier häufig nur über sehr geringe Speicher- und Prozessorresourcen verfügt.
Für RSA Verfahren werden heute Schlüssellängen von 1024 bzw. 2048 Bits verwendet. Dies entspricht einem Speicherbedarf von 128 bzw. 256 Byte für einen RSA Schlüssel. Für ein ECC System mit vergleichbarer Sicherheit werden nur 170 bzw. 233 Bit (22 bzw 30 Byte) benötigt.
Seit den Vorschlägen von Miller und Koblitz 1985 wurden schon zahlreiche Artikel über ECC veröffentlicht. Viele davon behandeln auch Implementierungsdetails und Optimierungsmöglichkeiten auf den unterschiedlichsten Systemen, wobei als Grundkörper häufig $\mathbb{F}_p$, mit einer grossen Primzahl p verwendet wird. Binäre Grundkörper wurden vor allem für Hardwareimplementierungen untersucht, da sich die binäre Repräsentation der Elemente einfach in Hardware giessen lässt. Mittlerweile findet man auch verschiedene Artikel über Implementierungen von ECC über binären Grundkörpern. Die Vergleichbarkeit der Ergebnisse gestaltet sich jedoch nicht trivial, da es vielfältigste Faktoren gibt die Einfluss auf die Laufzeit der verschiedenen Implementierungen nehmen. Hankerson et. al. vergleichen in zwei Artikeln [HHM01] und [BHLM01] ihre Implementierungen auf einem Pentium II 400 MHz für die NIST-Kurven über binären und primen Grundkörpern. In ihren Timingvergleichen schneiden die Algorithmen für Punktoperationen auf Elliptischen Kurven über Körpern mit primer Ordnung deutlich besser ab als die über binären Körpern.

Die Motivation für diese Arbeit besteht nun darin, zu erfahren wie hoch der Speicherbedarf einer ECC Implementierung auf zwei speziellen embedded Systemen ist und welche Zeit für die Vielfachenbildung eines Punktes benötigt wird.

1.1 Aufteilung der Arbeit

In dieser Arbeit sollen die Grundlagen, die für kryptographische Verfahren auf Elliptischen Kurven benötigt werden, Punktaddition und -multiplikation, betrachtet werden. Vorhandene Algorithmen werden verglichen und auf ihre Brauchbarkeit im embedded Umfeld hin untersucht. Es werden Kurven über dem Grundkörper $\mathbb{F}_{2^n}$ betrachtet, dessen Arithmetik ebenfalls implementiert und die Algorithmen verglichen werden.
Das Hauptaugenmerk liegt auf der Beschreibung der Algorithmen mit Hinweisen für eine effektive Implementierung. Auf die Einzelheiten der erstellten Referenzimplementierung wird nur sehr bedingt eingegangen, während die erzielten Ergebnisse für Timing- und Resourcenvergleiche genauer beleuchtet werden. Die erstellte Implementierung

wurde so portabel gehalten, dass sie sowohl auf 8 Bit, 16 Bit und 32 Bit Systemen lauffähig ist. Während der Kompilierung kann die zu verwendende Wortbreite ausgewählt werden. Für die Beurteilung des Laufzeitverhalten der verschiedenen Algorithmen gilt es diese Eigenschaft zu beachten, da keine prozessorspezifischen Anpassungen, wie etwa Assemblerroutinen codiert wurden.
Alle für diese Arbeit implementierten Algorithmen befinden sich mitsamt einer Dokumentation auf der beigelegten CD-ROM.

Die Gliederung dieser Arbeit gestaltet sich wie folgt. In Kapitel 2 werden zunächst noch einmal die wichtigsten mathematischen Grundlagen für das Rechnen in endlichen Körpern vorgestellt. Weiterhin werden die verschiedenen Möglichkeiten zur Darstellung eines solchen Körpers beschrieben. Dieses Kapitel soll Lesern mit nur geringen Vorkenntnissen in diesem Bereich der Mathematik helfen sich darin zurecht zu finden. Für tiefergehende Grundlagen sei jedoch auf Lehrbücher der Algebra, wie beispielsweise [Mey75a; Mey75b], [Bos04], [Brö04] und [Art93] verwiesen.
Im dritten Kapitel wird die Arithmetik in $\mathbb{F}_{2^n}$ beschrieben und verschiedene Algorithmen dazu vorgestellt. Dabei stehen vor allem Multiplikations- und Reduktionsalgorithmen, sowie Verfahren zur Berechnung des multiplikativen Inversen im Blickpunkt. Anschließend werden die einzelnen Implementierungen auf verschiedenen Mikroprozessoren miteinander verglichen. Timing- und Resourcenvergleiche dienen dabei als Kriterium für die weitere Verwendung.
In Kapitel 4 werden die Elliptischen Kurven eingeführt und die Punktaddition sowie -multiplikation erklärt. Dem Leser werden dazu einige wesentliche Begriffe aus der algebraischen Geometrie vorgestellt. Mit Hilfe dieser Begriffe werden nun Elliptische Kurven erläutert. Verschiedene Koordinatendarstellungen der Punkte einer solchen Kurve werden ebenfalls eingeführt und hinsichtlich ihrer speziellen Eigenschaften für die Punktaddition und -multiplikation verglichen.
Im folgenden Abschnitt (Kapitel 5) werden Algorithmen zur Punktaddition und Vielfachenbildung vorgestellt und verglichen. Verschiedene Koordinatendarstellungen gehen dabei ebenso mit ein, wie verschiedene Darstellungen des Faktors mit dem ein Punkt multipliziert wird. Den Abschluss dieses Kapitels bildet ebenfalls ein Vergleich der implementierten Algorithmen auf den verschiedenen Plattformen. Der Tatsache, dass es sich bei der Vielfachenbildung eines Punktes um eine n-fache Punktaddition handelt, wird in dieser Arbeit dadurch Rechnung getragen, dass diese Vielfachenbildung meist *Punktmultiplikation* genannt wird.
Kapitel 6 stellt kurz drei der bekanntesten Public-Key-Verfahren mit Elliptischen Kurven vor.
Den Abschluss bildet Kapitel 7 mit einem Ausblick auf mögliche zukünftige Entwicklungen im Bereich der Kryptographie.

1.2 Zusammenfassung der Ergebnisse

Für die Darstellung der Elemente des Körpers $GF(2^n)$ wurde die polynomielle Darstellung gewählt, da sich diese besonders einfach implementieren lässt und keine Ein-

schränkungen in der Wahl des Exponenten hervorruft. Die für die Laufzeit entscheidenden Körperoperationen sind vor allem die Multiplikation sowie die Inversenbildung von Körperelementen. Für die Multiplikation erreichte der *Left-to-right comb Window* Algorithmus die beste Performance bei einem Speicherbedarf von 384 Byte (für eine Wortbreite von 16 Bit und einer *Windowgröße* von 4 Bit).
Für die Inversenbildung wurden zwei gängige Algorithmen (*Erweiterter Euklidischer* und *Modified Almost Inverse*) verglichen. Dabei zeigte sich die zweite Variante für spezielle Polynome etwas performanter. Im allgemeinen Fall ist jedoch der *Erweiterte Euklidische* Algorithmus schneller.
Für die Reduktion der Multiplikationsergebnisse wurden ebenfalls allgemeine Varianten mit einer auf ein bestimmtes Polynom angepassten Version verglichen. In der verwendeten Implementierung treten dabei für den C167 Laufzeitvorteile um den Faktor $n = 40$ auf.

Für die weiteren Betrachtungen der Punktoperationen auf Elliptischen Kurven wurden die beiden *NIST*-Kurven B-163 und B-233 ausgewählt. Für diese beiden Kurven ist das irreduzible Polynom zur Körperdarstellung bereits gegeben und der Reduktionsalgorithmus wurde jeweils entsprechend angepasst.
Für die Addition und Verdopplung von Punkten wurden verschiedene Koordinatendarstellungen verglichen und schließlich eine Form von gemischten Koordinaten, aus affinen und *López-Dahab*-Koordinaten, verwendet.
Für die eigentliche Punktmultiplikation wurden einfache binäre *Double-and-Add*-Algorithmen, Algorithmen mit spezieller Darstellung des Exponenten (Non Adjacent Form) und der *Montgomery* Algorithmus verglichen. Zusätzlich wurde noch ein „Sliding Window" Verfahren untersucht, welches eine B-äre Darstellung des Faktors verwendet.
Die Implementierung wurde auf einem C167 (16Bit) und ST30 (32Bit) Mikroprozessor getestet und die Algorithmen darauf verglichen. Auf beiden Prozessoren erreichte jeweils der Montgomery Algorithmus mit die besten Timing-Werte. Etwas langsamer zeigten sich die *Modified B-ary* sowie die *Left-to-right NAF* Variante, jeweils mit gemischten $\mathcal{LD}$-$\mathcal{A}$ Koordinaten.
Im Bezug auf den Speicherbedarf schneidet ebenfalls der Montgomery Algorithmus am Besten ab. In der für diese Arbeit implementierten Variante für die NIST-Kurve B-233 und einer Registerbreite von 16 Bit, werden 692 Byte temporärer Speicher benötigt. Mit 1264 Byte bzw. 1000 Byte ist der RAM-Bedarf für die beiden Algorithmen *Modified B-ary* und *Left-to-right NAF* zwar deutlich höher, jedoch schließt diese eine Verwendung im embedded Bereich nicht gänzlich aus.

Kapitel 2

Arithmetik und Darstellung von $\mathbb{F}_{2^n}$

In diesem Kapitel sollen zunächst die mathematischen Grundlagen für das Rechnen mit Polynomen und endlichen Körpern beschrieben werden. Zur Darstellung eines endlichen Körpers können verschiedene Darstellungen verwendet werden, die von der verwendeten Basis abhängen. Die Eigenschaften dieser Basen werden im Verlauf des Kapitels beschrieben und im Hinblick auf die kryptographische Sicherheit der Elliptischen Kurven wird eine Basis ausgewählt.

2.1 Mathematische Grundlagen

Zunächst werden wichtige mathematische Begriffe nochmals kurz erklärt, um schließlich den Begriff des *endlichen Körpers* einführen zu können.

2.1.1 Gruppe

Definition 2.1.1. *Eine* Gruppe *ist eine Menge G zusammen mit einer Verknüpfung*

$$\star : G \times G \to G, \qquad (a, b) \mapsto a \star b$$

welche die folgenden Axiome erfüllt:

(I) (Assoziativität) Es ist $a \star (b \star c) = (a \star b) \star c$ für alle $a, b, c \in G$.

(II) (neutrales Element) Es existiert ein Element $e \in G$, so dass gilt:
$a \star e = e \star a = a, \quad$ für alle$a \in G$.

(III) (Inverses) Zu jedem $a \in G$ existiert ein inverses Element $b \in G$ so dass gilt: $a \star b = b \star a = e$.

(IV) Gilt für G zusätzlich noch die Bedingung $a \star b = b \star a, \quad$ für alle$a, b \in G$, so nennt man G eine kommutative *oder* abelsche Gruppe.

Eine häufige Schreibweise zur Darstellung einer Gruppe ist die folgende: $(G, \star)$. Dabei wird die Menge G und die auf ihr definierte Operation $\star$ beschrieben. Man bezeichnet

die Gruppe $(G, \cdot)$ (bzw. $(G, +)$) multiplikativ geschrieben (bzw. additiv geschrieben) und verwendet dann die Schreibweise $a \cdot b = ab$ (bzw. $a + b$). Die Symbole „$\cdot$“ und „$+$“ stehen dabei für eine beliebige Gruppenverknüpfung.

Beispiel 2.1.2. *Die Symmetrische Gruppe S_n besteht aus allen Permutationen π der Menge $M = \{1, 2, 3, \ldots, n\}$. Als Gruppenoperation wird die Verknüpfung $\circ$ der Permutationen verwendet. Die Gruppe S_n ist für $n > 2$ nicht abelsch. Für die Elemente $p \in S_n$ gibt es zwei Schreibweisen. Im allgemeinen schreibt man*

$$p = \begin{pmatrix} 1 & 2 & \ldots & n \\ \pi(1) & \pi(2) & \ldots & \pi(n) \end{pmatrix},$$

wobei hier die Anordnung der Spalten beliebig ist. Sind die Spalten in aufsteigender Reihenfolge sortiert, so wird die obere Zeile redundant und man kann einfacher schreiben $p = (\pi(1)\ \pi(2)\ \ldots\ \pi(n))$.
Für eine Menge M mit n Elementen ergeben sich $n!$ viele Permutationen. Die Gruppe S_3 besteht somit aus den folgenden 6 Elementen:

$$S_3 = \big\{(1\ 2\ 3),\ (1\ 3\ 2),\ (2\ 1\ 3),\ (2\ 3\ 1),\ (3\ 2\ 1),\ (3\ 1\ 2)\big\}$$

Eine sehr verbreitete Schreibweise für symmetrische Gruppen ist die Zykelschreibweise, *die genauso aussieht wie die hier verwendete, jedoch zyklisch gelesen wird. Die Darstellung $(1\ 2\ 3)$ bedeutet in Zykelschreibweise $1 \mapsto 2$, $2 \mapsto 3$ und, daher der Name, $3 \mapsto 1$. Mehr zur Zykelschreibweise und symmetrischen Gruppen findet sich u.a. in [Mey75a].*

Definition 2.1.3. *Die Anzahl der Elemente einer endlichen Gruppe G bezeichnet man als die* Ordnung *von G und schreibt dafür $ord(G)$ oder $|G|$.*

Wie das vorherige Beispiel gezeigt hat, gilt $ord(S_3) = 6$ und allgemein $ord(S_n) = n!$ für $n \geq 1$.

Definition 2.1.4. *Eine Teilmenge H von G heißt* Untergruppe *von $(G, \star)$, falls H bezüglich der Operation $\star$ eine Gruppe ist.*

Für jedes $g \in G$ bildet die Menge $g^k, k \in \mathbb{Z}$ eine Untergruppe von G. Man nennt sie die von g erzeugte Untergruppe und schreibt $\langle g \rangle$ für diese Untergruppe.

Definition 2.1.5. *Sei $G = \langle g \rangle$ für ein $g \in G$, dann heißt G* zyklisch *und g heißt* Generator *oder primitive Wurzel von G. Man nennt G dann die von g erzeugte Gruppe.*

Ist G endlich, so ist die Ordnung jeder Untergruppe H_i von G ein Teiler der Ordnung von G. Diese Aussage heißt auch **Satz von Lagrange**. Ein Beweis findet sich in [Buc04].

Lemma 2.1.6. *Sei G eine endliche multiplikative Gruppe $(G, \cdot)$, so ist die* Ordnung eines Elements *$a \in G$ definiert als die kleinste positive Ganzzahl $m > 0$ mit $a^m = 1$.*

Theorem 2.1.7. *Sei G eine endliche Gruppe der Ordnung n und $a \in G$ ein Element der Ordnung m, dann gilt:*

$$a^k = 1 \quad \text{genau dann wenn} \quad m|k$$

Beweis. Für den Beweis müssen beide Richtungen betrachtet werden.

„⇒“ Sei $a^k = 1$ und $a^m = 1$, dann ist die Zerlegung $k = qm + r$ eindeutig mit $0 \leq r < m$.
Annahme: $r \neq 0$, dann ist $1 = a^k = (\underbrace{a^m}_{=1})^q \cdot a^r$. Da $m > 0$ die kleinste Zahl mit $a^m = 1$ ist, ist $a^r \neq 1 \quad \Rightarrow$ Widerspruch.
$\Rightarrow \quad r = 0$ und $m|k$

„⇐“ $m|k \Rightarrow k = qm \Rightarrow a^k = (\underbrace{a^m}_{=1})^q = 1$

□

2.1.2 Ringe

Definition 2.1.8. *Ein* Ring *ist eine Menge $\mathcal{R}$ zusammen mit zwei auf $\mathcal{R}$ definierten binären Verknüpfungen $+$ und $\cdot$, so dass die folgenden Bedingungen erfüllt sind.*

(I) $(R, +)$ ist eine abelsche Gruppe

(II) $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ für alle $a, b, c \in R$ (Assoziativität von $\cdot$)

(III) $a \cdot (b + c) = a \cdot b + a \cdot c$ und $(a + b) \cdot c = a \cdot c + b \cdot c$ für alle $a, b, c \in R$ (Distributivität von $+$ und $\cdot$)

Ist auch die Multiplikation der Ringelemente kommutativ, so spricht man von einem *kommutativen Ring*. Ein Ringelement e für das gilt $e \cdot a = a \cdot e = a$ für alle $a \in R$ bezeichnet man als das *neutrale Element* bezüglich der Multiplikation oder das *Einselement*. Hat $\mathcal{R}$ ein solches Einselement, so nennt man $\mathcal{R}$ einen *kommutativen Ring mit Einselement.*

Ein Beispiel für einen kommutativen Ring mit Einselement ist die Menge der Restklassen $\mathbb{Z}/n\mathbb{Z}$ der ganzen Zahlen, zusammen mit der für Restklassen definierten Operationen der Addition und Multiplikation. Man nennt einen solchen Ring auch den *Restklassenring*.

Definition 2.1.9. *Sei $\mathcal{R}$ ein kommutativer Ring mit Einselement $e \in \mathcal{R}$*

(I) $a \in \mathcal{R}$ heißt Einheit *in $\mathcal{R}$, wenn es ein $b \in \mathcal{R}$ gibt mit $a \cdot b = e$. Die Menge der Einheiten von $\mathcal{R}$ bezeichnet man als $\mathcal{R}^*$.*

(II) $a \in \mathcal{R}$ heißt Nullteiler *in $\mathcal{R}$, wenn es ein $b \in \mathcal{R} \setminus \{0\}$ gibt mit $ab = 0$.*

(III) $\mathcal{R}$ heißt Integritätsbereich *falls es in $\mathcal{R}$ keine Nullteiler außer 0 gibt.*

2.1.3 Körper

Zunächst die formale Definition eines Körpers:

Definition 2.1.10. *Ein* Körper K *ist ein kommutativer Ring* $\mathcal{R}$ *mit Einselement, in dem jedes von Null verschiedene Element ein Inverses bezüglich der Körpermultiplikation besitzt.*

Definition 2.1.11. *Sei* K *ein Körper und* $k \subset K$*.*

(I) k *heißt* Teilkörper *von* K*, wenn* k *mit den für* K *gültigen Verknüpfungen* $+$ *und* $\cdot$ *ein Körper ist.*

(II) Ist k *ein Teilkörper von* K*, dann nennt man* (K, k) *eine* Körpererweiterung *von* k *und* K *einen* Erweiterungskörper *von* k*.*

Jede Körpererweiterung (K, k) lässt sich auch als Vektorraum über k auffassen. Dabei entspricht die Multiplikation von Elementen aus k mit Elementen aus K einer Skalarmultiplikation. Die Körpererweiterung ist endlich, falls der entstandene Vektorraum endlich-dimensional ist. Der *Grad* einer endlichen Körpererweiterung $[K : k]$ ist die Dimension des entsprechenden Vektorraums.
Grundlagen zu Körpererweiterungen und Vektorräumen finden sich in jedem Lehrbuch zur Algebra, z.B. [Bos04, Kapitel 3.2] und [Brö04, Kapitel 1.3].

Lemma 2.1.12. *Sei* $\phi : \mathbb{Z} \to K, n \mapsto n \cdot 1_K$*, dann nennt man die kleinste natürliche Zahl* $n \in \mathbb{N} \setminus \{0\}$ *mit der Eigenschaft*

$$n \cdot 1_K = \sum_{k=1}^{n} 1_K = 0$$

Charakteristik *von* K *und schreibt auch* $char(K)$*.*

Man sagt ein Körper hat *Charakteristik* 0 ($char(K) = 0$), wenn $\sum_{k=1}^{n} 1_K \neq 0$ für alle natürlichen Zahlen $n > 0$.
Es lässt sich leicht zeigen, dass $n = char(K)$ entweder 0 oder eine Primzahl ist.
Aus

$$n = p \cdot q = \sum_{k=1}^{pq} 1_K = \sum_{k=1}^{p} 1_K \cdot \sum_{k=1}^{q} 1_K = 0$$

folgt $\sum_{k=1}^{p} 1_K = 0$ oder $\sum_{k=1}^{q} 1_K = 0$, was einen Widerspruch zu der Annahme darstellt, dass n die kleinste Zahl mit dieser Eigenschaft ist.

Lemma 2.1.13. *Für jede Primzahl* p *ist* $\mathbb{Z}/p\mathbb{Z}$ *ein Körper, den man auch mit* $\mathbb{F}_p$ *bezeichnet.*

$\mathbb{F}_p$ hat genau p viele Elemente, nämlich die Restklassen modulo p, dargestellt z.B. durch $0, 1, \ldots, p-1$. Dies nennt man auch ein vollständiges Restesystem. Die Charakteristik von $\mathbb{F}_p$ ist $char(\mathbb{F}_p) = p$.
Neben den Körpern $\mathbb{F}_p$ der Charakteristik p mit p Elementen, gibt es noch weitere endliche Körper für deren Verständnis im folgenden Polynome erklärt werden.

Eine weitere wichtige Eigenschaft von Körpern steckt in folgendem Lemma:

Lemma 2.1.14 (Frobenius). *Sei K ein Körper mit $char(K) = p > 0$ und $q = p^m$, dann gilt für alle $a, b \in K$*

$$(a + b)^q = a^q + b^q$$

Beweis. Mit vollständiger Induktion lässt sich die Aussage leicht nachrechnen. Für den Induktionsanfang ($m = 1$), lässt sich zunächst zeigen

$$(a + b)^p = \sum_{i=0}^{p} \binom{p}{i} a^i b^{p-i} = a^p + b^p,$$

denn der Binomialkoeffizient schreibt sich

$$\binom{p}{i} = \frac{p!}{i!(p-i)!} = \frac{p \cdot (p-1) \cdots 2 \cdot 1}{i!(p-i)!}$$

wodurch bei $char(K) = p$ alle mittleren Binomialkoeffizienten wegfallen. Der Schritt von m zu $m + 1$ ist nun trivial und die Behauptung ist gezeigt. □

Eine weitere elementare Definition bestimmt den Begriff des algebraisch abgeschlossenen Körpers.

Definition 2.1.15. *Ein Körper K heißt* algebraisch abgeschlossen*, wenn sich jedes Polynom $f(x) \in K[x]$ von positivem Grad als Produkt von Polynomen vom Grad 1 schreiben läßt, d.h. wenn*

$$f(x) = d(x - c_1) \dots (x - c_m)$$

für Elemente c_i, d aus K gilt. In diesem Fall ist $m = \deg(f)$ und d der Koeffizient vor x^m.

2.1.4 Polynome

Polynome werden für die Darstellung von endlichen Körpern benötigt und sollen deshalb im folgenden vorgestellt werden.
Es sei $\mathcal{R}$ ein kommutativer Ring mit Einselement $e \neq 0$. Ein *Polynom* in einer Variablen über $\mathcal{R}$ ist ein Ausdruck der Form

$$f(x) = \sum_{k=0}^{n} a_k x^k = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$$

wobei x die Variable ist und für die Koeffizienten gilt $a_0, a_1, ..., a_n \in \mathcal{R}$. Die Menge aller Polynome über $\mathcal{R}$ in der Variablen x wird mit $\mathcal{R}[x]$ bezeichnet.
Sei $n \geq 0$ die größte natürliche Zahl mit $a_n \neq 0$, dann heißt $n = \deg(f)$ der *Grad* des Polynoms.
Weiterhin bezeichnet man a_n als *Leitkoeffizient* oder *führender Koeffizient* von f. Ist $a_n \neq 0$ und alle weiteren Koeffizienten $a_k = 0$ mit $0 \leq k \leq n-1$, dann heißt f *Monom*. Ist $r \in \mathcal{R}$ so bezeichnet $f(r) = a_n r^n + a_{n-1} r^{n-1} + ... + a_1 r + a_0$ den *Wert* von f an der Stelle r. Gilt $f(r) = 0$, so heißt r *Nullstelle* von f.

Um mit Polynomen zu rechnen, betrachtet man die Koeffizienten a_i über dem Ring $\mathcal{R}$ oder einem Körper K.

Die *Addition* zweier Polynome wird dabei koeffizientenweise durchgeführt.
Sei $g(x) = b_m x^m + b_{m-1}x^{m-1} + ... + b_1 x + b_0$ mit $b_i \in \mathcal{R}$ ein zweites Polynom. O.B.d.A. sei $\deg(f) = n \geq \deg(g) = m$, dann werden die fehlenden Koeffizienten von g mit Nullen aufgefüllt und man definiert die Summe der beiden Polynome folgendermaßen:

$$(f+g)(x) =_{def} \sum_{k=0}^{n} (a_k + b_k)x^k = (a_n + b_n)x^n + (a_{n-1} + b_{n-1})x^{n-1} + ... + (a_0 + b_0)$$

$(f+g)(x)$ ist dabei offensichtlich wieder einem Polynom über $\mathcal{R}$.
Die Multiplikation zweier Polynome ist nicht ganz so trivial wie die Addition und ist folgendermaßen erklärt. Seien f und g zwei Polynome über $\mathcal{R}$, dann ist

$$(f * g)(x) =_{def} c_{n+m}x^{n+m} + ... + c_0$$
$$\text{mit} \qquad c_k =_{def} \sum_{i=0}^{k} a_i b_{k-i}, \quad 0 \leq k \leq n+m$$

das *Produkt* dieser beiden Polynome. Die nicht definierten Koeffizienten a_i, b_i werden dabei wieder auf Null gesetzt.
Durch die Möglichkeit der Addition und Multiplikation von Polynomen über $\mathcal{R}$ sieht man leicht ein, dass $(\mathcal{R}[x], +, \cdot)$ ein *kommutativer Ring mit Einselement* ist. Man schreibt dafür $\mathcal{R}[x]$ und sagt $\mathcal{R}[x]$ ist ein *Polynomring* über $\mathcal{R}$.

Betrachten wir fortan den Polynomring $K[x]$ über dem Körper K. Ein solcher Polynomring ist immer Nullteilerfrei und es lässt sich eine Division mit Rest in diesem Polynomring erklären.

Theorem 2.1.16. *Seien $f(x), g(x) \in K[x]$ mit $g \neq 0$. Dann gibt es eindeutig bestimmte Polynome $q(x), r(x) \in K[x]$ mit*

$$f(x) = q(x)g(x) + r(x)$$

und $r(x) = 0$ oder $\deg(r(x)) < \deg(g(x))$.

Beweis. [LN83, Seite 20] □

Ist $r(x)$ das Nullpolynom so teilt $g(x)$ das Polynom $f(x)$ und man bezeichnet $g(x)$ als *Divisor*.
Existieren für ein Polynom $f(x) \in K[x]$ keine Divisorpolynome $g(x) \in K[x]$ mit $\deg(g(x)) \geq 1)$, so nennt man $f(x)$ *irreduzibel* über K.

Theorem 2.1.17. *Sei $f(x) \in K[x]$ ein vom Nullpolynom verschiedenes Polynom und ist $a \in K$ eine Nullstelle von $f(x)$, dann ist $f(x)$ teilbar durch das Polynom $x - a$ mit $f(x) = (x-a)q(x)$ mit $q(x) \in K[x]$*

Beweis. Nach Theorem 2.1.16 gibt es Polynome $q(x), r(x) \in K[x]$, so dass gilt $f(x) = (x-a)q(x) + r(x)$ mit $r(x) = 0$ oder $\deg(r(x)) < 1 = \deg(x-a)$.
Es folgt $r(x) = c \in K$ und es gilt $f(a) = 0 = c$ weshalb $f(x) = (x-a)q(x)$ gilt. □

Theorem 2.1.18. *Sei $f \in K[x]$ mit $\deg(f) = n \geq 0$. Die Elemente $b_1, \ldots, b_m \in K$ seien unterschiedliche Nullstellen von f mit $k_1, \ldots, k_m \in \mathbb{N}\backslash\{0\}$ und $k_1 + \ldots + k_m \leq n$, dann wird $f(x)$ geteilt von $(x - b_1)^{k_1} \cdots (x - b_m)^{k_m}$. Die Zahlen k_i nennt man auch die* Ordnung der Nullstelle*. f hat höchstens n unterschiedliche Nullstellen in K.*

Beweis. Jedes Polynom $x - b_j, \quad 1 \leq j \leq m$ ist irreduzibel über dem Körper K. Damit erscheint $(x - b_j)^{k_j}$ als ein Faktor in der kanonischen Faktorisierung von f und $(x - b_1)^{k_1} \cdots (x - b_m)^{k_m}$ teilt somit f. Ein Vergleich der Grade ($m \leq k_1 + \ldots + k_m \leq n$) zeigt die Behauptung. □

Genau wie bei gewöhlichen Funktionen, so ist auch für Polynome aus $K[x]$ die Ableitung erklärt.

Definition 2.1.19. *Sei $f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \in K[x]$, dann ist die* Ableitung *f' von f definiert durch $f'(x) = a_1 + 2a_2 x + \ldots + n a_n x^{n-1} \in K[x]$.*

Theorem 2.1.20. *Ein Element $b \in K$ ist genau dann eine mehrfache Nullstelle von $f \in K[x]$ wenn $f(b) = f'(b) = 0$.*

Beweis. Ist r die Vielfachheit der Nullstelle $b \in K$, so gibt es eine Zerlegung von f des Typs $f = (x - b)^r g$ mit $g \in K[x], g(b) \neq 0$. Wegen

$$f' = (x - b)^r g' + r(x - b)^{r-1} g$$

ist $f'(b) = 0$ äquivalent zu $r \geq 2$. □

Es besteht ein Zusammenhang zwischen der Irreduzibilitätseigenschaft und der Nichtexistenz von Nullstellen für Polynome. Sei f ein irreduzibles Polynom in $K[x]$ mit deg(f) ≥ 2, dann hat f nach Theorem 2.1.17 keine Nullstellen in K.
Umgekehrt gilt dies nur für Polynome vom Grad 2 und 3, aber nicht notwendigerweise für Polynome mit einem Grad > 3.

Beispiel 2.1.21. *Das Polynom $x^2 + 1$ ist irreduzibel über $\mathbb{R}$, da es keine reellen Nullstellen ($x^2 = -1$) hat. Das Polynom $(x^2 + 1)^2 = x^4 + 2x^2 + 1$ hat über $\mathbb{R}$ ebenfalls keine Nullstellen, ist aber nicht irreduzibel wie die Konstruktion zeigt.*

Die irreduziblen Polynome über einem Körper K entsprechen genau den primen Elementen des Polynomrings K[x].

Genau wie in $\mathbb{Z}$ lassen sich auch für Polynome $f \in K[x]$ Restklassenringe $K[x]/(f)$ erzeugen.
Die Elemente des Restklassenrings sind alle Restklassen $g + fK[x]$ mit $g \in K[x]$. Dabei ist $fK[x]$ die Menge $\{fh \mid h \in K[x]\}$ aller Vielfachen von f. Zwei Polynome $g, h \in K[x]$ sind genau dann Repräsentanten der gleichen Restklasse, wenn f ein Teiler von $g - h$ im Polynomring $K[x]$ ist. Diese Definition ist somit völlig analog zu der des *mod*-Operators für $\mathbb{Z}$.
Der kleinste Repräsentant $r(x) \in K[x]$ einer Restklasse $g + fK[x]$ mit $\deg(r) < \deg(f)$ entspricht dem Rest der Polynomdivision von g/f und ist nach Theorem 2.1.16 eindeutig bestimmt.

Satz 2.1.22. *Sei K ein Körper und $f \in K[x]$ ein irreduzibles Polynom über K, dann ist $K[x]/(f)$ ein Körper.*

Der Beweis dazu findet sich in [Brö04, Seite 311].

2.1.5 Endliche Körper

Zunächst betrachten wir allgemein Körper der Charakteristik p und ihre Eigenschaften. Im Anschluss wird dann im Speziellen auf endliche Körper der Charakteristik 2 eingegangen und ihre Besonderheiten vorgestellt.
Für endliche Körper mit q Elementen schreibt man allgemein $\mathbb{F}_q$. Die Charakteristik eines endlichen Körpers kann nicht 0 sein und muss somit, wie bereits gezeigt, eine Primzahl p sein. Ist $char(\mathbb{F}_q) = p$ so enthält $\mathbb{F}_q$ also den Grundkörper $\mathbb{F}_p$ und ist somit selbst ein Vektorraum über $\mathbb{F}_p$ mit endlicher Dimension vom Grad der Körpererweiterung $[\mathbb{F}_q : \mathbb{F}_p] = m$. Damit folgt, das in $\mathbb{F}_q$ genau $q = p^m$ viele Elemente enthalten sind.

Zur Verdeutlichung betrachten wir im folgenden irreduzible Polynome f vom Grad n über dem Körper $\mathbb{F}_p$.
Der Restklassenring $\mathbb{F}_p[x]/(f)$ ist ein Körper dessen Elemente genau die Restklassen modulo f aller Polynome aus $\mathbb{F}_p[x]$ sind. Stellt man diese Restklassen durch ihre kleinsten Repräsentanten dar, so sind dies genau alle Polynome r aus $\mathbb{F}_p[x]$ deren Grad kleiner als $\deg(f) = n$ ist. Man erhält genau p^n viele Elemente.

Lemma 2.1.23. *Sei $f \in \mathbb{F}_p[x]$ irreduzibel über $\mathbb{F}_p$ und $\deg(f) = n$. Man kann zeigen, dass der Körper $\mathbb{F}_p[x]/(f) = \mathbb{F}_{p^n}$, bis auf Isomorphie, der einzige Körper mit p^n Elementen ist und bezeichnet ihn als $\mathbb{F}_{p^n}$.*

Beweis. Der Beweis dazu findet sich in vielen Büchern über Algebra, explizit in [Art93, Kapitel 13.6]. □

Satz 2.1.24. *Die multiplikative Gruppe $\mathbb{F}_{p^n}^* = \mathbb{F}_{p^n} \setminus \{0\}$ eines jeden endlichen Körpers ist zyklisch.*

Der Körper $\mathbb{F}_{2^n}$

Für diese Arbeit ist der Körper $\mathbb{F}_2$ mit den zwei Elementen $0, 1$ und dessen Erweiterungskörper $\mathbb{F}_{2^n}$ sehr interessant, da sich die Körperarithmetik für solche Körper einfach und effizient implementieren lässt.
Die Addition zweier Elemente aus $\mathbb{F}_{2^n}$ entspricht einer Rechnung modulo 2 und damit genau der bitweisen *XOR*-Operation (geschrieben $\oplus$). Auch die Subtraktion zweier Körperelemente entspricht dem bitweisen XOR und kann somit durch die Addition substituiert werden.
Diese Körper $\mathbb{F}_{2^n}$ stellen in dieser Arbeit auch die Grundlage für die späteren Berechnungen mit Elliptischen Kurven dar.
Für die Darstellung eines Körpers $\mathbb{F}_{2^n}$ gibt es verschiedene Möglichkeiten. Folgende drei werden im Abschnitt 2.2 genauer erläutert:

- Polynomielle Darstellung (für alle n)

- Optimale Normalbasis (ONB) (nur für bestimmte n)
- Zusammengesetzte Erweiterungskörper von $\mathbb{F}_2$ ($(\mathbb{F}_{2^n}, \mathbb{F}_{2^m})$, nur möglich wenn $m|n$)

2.1.6 Quadratische Gleichungen in Charakteristik 2

Für die Erzeugung und Verifizierung von zufälligen Punkten auf einer Elliptischen Kurve, ist es notwendig Quadratische Gleichungen lösen zu können. Für das Lösen solcher Gleichungen in endlichen Körpern der Charakteristik 2 sind noch zwei mathematische Hilfsmittel von Nöten.

Definition 2.1.25. *Spurfunktion Sei $q = 2^n$ und $x \in \mathbb{F}_q$. Die* Spur *$Tr(x)$ von x ist in diesem Fall definiert durch*

$$Tr(x) = \sum_{j=0}^{n-1} x^{2^j} = x + x^2 + x^{2^2} + x^{2^3} + x^{2^4} + \ldots + x^{2^{n-1}}$$

Die Funktion der *Spur* erfüllt nun mehrere Eigenschaften, die im folgenden Lemma allgemein zusammengefasst sind. (Für die Beweise siehe [LN83, Kapitel 2.3])

Lemma 2.1.26. *Es sei $\mathbb{F}_2 \subset \mathbb{F}_{2^n}$ eine Körpererweiterung vom Grad n. Für alle $\alpha, \beta \in \mathbb{F}_{q^n}$ und $\lambda \in \mathbb{F}_2$ gilt:*

(I) Sei $Tr(x) \in \mathbb{F}_2$. Die Spurfunktion liefert eine Abbildung

$$Tr: \quad \mathbb{F}_{2^n} \longrightarrow \mathbb{F}_2, \qquad x \longmapsto Tr(x)$$

(II) $Tr(\alpha + \beta) = Tr(\alpha) + Tr(\beta)$

(III) $Tr(\lambda \cdot \alpha) = \lambda \cdot Tr(\alpha)$

(IV) $Tr(\alpha^2) = Tr(\alpha)$

(V) Tr ist surjektiv auf $\mathbb{F}_2$

(VI) $Tr(0) = 0$

Die zweite benötigte Funktion ist die der *Halbspur*. Diese ist definiert für den Fall $\mathbb{F}_{2^n}$ mit ungeradem n durch

$$\tau(x) = \sum_{j=0}^{(n-1)/2} x^{2^{2j}} = x + x^{2^2} + x^{2^4} + x^{2^6} + \ldots + x^{2^{n-1}}$$

Die beiden Funktionen hängen dann folgendermaßen zusammen

$$\tau(x)^2 + \tau(x) = x + Tr(x) \tag{2.1}$$

was sich einfach nachrechnen lässt.

Mit Hilfe der Spurabbildung und der Halbspurfunktion lassen sich nun quadratische Gleichungen über $\mathbb{F}_q$, $q = 2^n$ lösen.
Diese Gleichungen schreiben sich in einer allgemeinen Normalform als

$$X^2 + \alpha X + \beta = 0 \tag{2.2}$$

Falls $\alpha = 0$, so lässt sich die Gleichung trivial durch Berechnung von $\sqrt{\beta} = \beta^{2^{n-1}} = \beta^{q/2}$ lösen.
Für den Fall $\alpha \neq 0$ lässt sich die Gleichung durch eine Substitution von $X := \alpha X$ in die Normalform bringen:

$$X^2 + X + \delta = 0 \quad \text{mit } \delta = \frac{\beta}{\alpha^2} \tag{2.3}$$

Sei nun x eine Lösung, dann stellt $x+1$ die zweite Lösung der Gleichung dar, wie sich leicht nachrechnen lässt.

$$(X + x)(X + (x + 1)) = X^2 + X(x + 1) + xX + x(x + 1) = X^2 + X + \overbrace{x(x + 1)}^{=\delta}$$

Eine Lösung von 2.3 existiert nur, wenn $Tr(\delta) = 0$, wie hier gezeigt ist:

$$\begin{aligned} 0 &= Tr(\overbrace{x^2 + x + \delta}^{=0}) \\ &= Tr(x^2) + Tr(x) + Tr(\delta) \\ &= Tr(x) + Tr(x) + Tr(\delta) \\ &= Tr(\delta) \end{aligned}$$

Für die Lösung der quadratischen Gleichung in $\mathbb{F}_{2^n}$ unterscheidet man nun zwei Fälle, die auf einen Satz von HILBERT 90 ([Bos04, Theorem 1, Seite 200]) zurückzuführen sind.

(I) n ungerade:
Für die Halbspur gilt die Gleichung 2.1, welche für $Tr(\delta) = 0$ zu $\tau(\delta)^2 + \tau(\delta) = \delta$ wird. Eine Lösung von 2.3 ist somit gegeben durch die Halbspur $\tau(\delta)$.

(II) n gerade.
Zunächst benötigt man ein Element $d \in \mathbb{F}_{2^n}$ mit $Tr(d) = 1$. Dabei kann man solange zufällig Elemente wählen und im Anschluss die Spur ausrechnen bis man ein entsprechendes gefunden hat. Da $Tr(x)$ surjektiv ist, muss ein solches $d \in \mathbb{F}_{2^n}$ mit $Tr(d) = 1$ existieren. Eine zweite Möglichkeit ist die Berechnung der Spuren aller Basisvektoren $v \in \{1, x, x^2, x^3, \ldots, x^{n-1}\}$, von denen mindestens für einen gilt $Tr(v) = 1$.
Für die Praxis ist eine solche Vorberechnung der Spuren für die Basisvektoren eine lohnende Investition, da sich damit Spuren anderer Elemente effektiv berechnen lassen.

Verfügt man nun über ein geeignetes d so lässt sich eine Lösung x von 2.3 folgendermaßen schreiben

$$x = \sum_{i=0}^{n-2} \left(\sum_{j=i+1}^{n-1} d^{2^j} \right) \delta^{2^i}.$$

Um zu zeigen, dass x tatsächlich eine Lösung von 2.3 ist, rechnet man nach

$$\begin{aligned}
x^2 + x &= \sum_{i=0}^{n-2} \left(\sum_{j=i+1}^{n-1} d^{2^{j+1}} \right) \delta^{2^{i+1}} + \sum_{i=0}^{n-2} \left(\sum_{j=i+1}^{n-1} d^{2^j} \right) \delta^{2^i} \\
&= \sum_{i=1}^{n-1} \left(\sum_{j=i+1}^{n} d^{2^j} \right) \delta^{2^i} + \sum_{i=0}^{n-2} \left(\sum_{j=i+1}^{n-1} d^{2^j} \right) \delta^{2^i} \\
&= d(\delta^{2^{n-1}} + \delta^{2^{n-2}} + \ldots + \delta^2) + \delta(d^{2^{n-1}} + d^{2^{n-2}} + \ldots + d^2) \\
&= d(Tr(\delta) + \delta) + \delta(Tr(d) + d) \\
&= dTr(\delta) + \delta
\end{aligned}$$

Da $Tr(\delta) = 0$, gilt also die Behauptung $x^2 + x = \delta$ und somit ist x eine Lösung von 2.3

2.2 Darstellung von $\mathbb{F}_{2^n}$

Im Hinblick auf eine effiziente Implementierung haben sich, für die Körperdarstellung von $\mathbb{F}_{2^n}$, drei verschiedene Möglichkeiten herauskristallisiert. Diese werden im folgenden genauer erklärt und die Wahl der Darstellung für diese Arbeit wird begründet.

2.2.1 Polynomielle Darstellung

Die *Polynomielle Darstellung* eines Körpers $\mathbb{F}_{2^n}$ ist für jedweden Grad der Körpererweiterung $[\mathbb{F}_{2^n} : \mathbb{F}_2]$ möglich. Die Elemente $\alpha_i \in \mathbb{F}_{2^n}$ werden dabei als Polynome vom Grad kleiner n dargestellt, mit Koeffizienten a_k aus $\mathbb{F}_2$.

$$\mathbb{F}_{2^n} = \left\{ a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_2x^2 + a_1x + a_0 \mid a_k \in \{0,1\} \right\}$$

Es ist leicht einzusehen, dass $\mathbb{F}_{2^n}$ ein Vektorraum über $\mathbb{F}_2$ ist. Eine Basis für diesen Vektorraum stellen die *Vektoren* $\{1, x, x^2, \ldots, x^{n-2}, x^{n-1}\}$ dar. Damit können wir die skalaren Koeffizienten $a_k \in \mathbb{F}_2$ auch in Vektorform schreiben als $(a_{n-1}a_{n-2} \ldots a_1a_0)$. Diese Vektoren stellen dann eindeutig ein Polynom aus $\mathbb{F}_{2^n}$ dar. Für die Körperoperationen wird nun auf die Vektordarstellung der Elemente zurückgegriffen, wobei man das Ergebnis modulo dem irredzuiblen Polynom f reduziert.

Beispiel 2.2.1. *Betrachten wir den endlichen Körper $\mathbb{F}'_{2^4}$ der durch das irreduzible Polynom $f(x) = x^4 + x + 1 \in \mathbb{F}_2[x]$ vollständig bestimmt ist.*
Es ist bekannt, dass zwei endliche Körper mit gleicher Elementanzahl isomorph sind. (Siehe dazu Lemma 2.1.23.)

Einen solchen, zu $\mathbb{F}'_{2^4}$ isomorphen, zweiten Körper $\mathbb{F}''_{2^4}$ legt das irreduzibles Polynom $g(x) = x^4 + x^3 + 1, g \in \mathbb{F}_2[x]$ fest. In Tabelle 2.1 sind zur Verdeutlichung beide Repräsentanten des Körpers $\mathbb{F}_{2^4}$, die durch die irreduziblen Polynome f und g festgelegt sind, dargestellt. Als Generator wurde dabei das Element $x \in \mathbb{F}_{2^4}$ verwendet.
Ein möglicher Isomorphismus der $\mathbb{F}'_{2^4}$ auf $\mathbb{F}''_{2^4}$ abbildet ist beispielsweise

$$\Theta : \mathbb{F}'_{2^4} \to \mathbb{F}''_{2^4}, \quad \alpha \mapsto \beta^3 + \beta^2,$$

wobei $\alpha \bmod f$ die Restklasse von x modulo f und $\beta \bmod g$ die Restklasse von x modulo g bezeichnet.

Potenz von x	**Polynomdarstellung**		**Vektordarstellung mit** $(x^3, x^2, x, 1)$	
	mod f	mod g	mod f	mod g
0	0	0	$(0,0,0,0)$	$(0,0,0,0)$
$x^0 = x^{15} = 1$	1	1	$(0,0,0,1)$	$(0,0,0,1)$
x^1	x	x	$(0,0,1,0)$	$(0,0,1,0)$
x^2	x^2	x^2	$(0,1,0,0)$	$(0,1,0,0)$
x^3	x^3	x^3	$(1,0,0,0)$	$(1,0,0,0)$
x^4	$x+1$	x^3+1	$(0,0,1,1)$	$(1,0,0,1)$
x^5	x^2+x	x^3+x+1	$(0,1,1,0)$	$(1,0,1,1)$
x^6	x^3+x^2	x^3+x^2+x+1	$(1,1,0,0)$	$(1,1,1,1)$
x^7	x^3+x+1	x^2+x+1	$(1,0,1,1)$	$(0,1,1,1)$
x^8	x^2+1	x^3+x^2+x	$(0,1,0,1)$	$(1,1,0,1)$
x^9	x^3+x	x^2+1	$(1,0,1,0)$	$(0,1,0,1)$
x^{10}	x^2+x+1	x^3+x	$(0,1,1,1)$	$(1,0,1,0)$
x^{11}	x^3+x^2+x	x^3+x^2+1	$(1,1,1,0)$	$(1,1,0,1)$
x^{12}	x^3+x^2+x+1	$x+1$	$(1,1,1,1)$	$(0,0,1,1)$
x^{13}	x^3+x^2+1	x^2+x	$(1,1,0,1)$	$(0,1,0,1)$
x^{14}	x^3+1	x^3+x^2	$(1,0,0,1)$	$(1,1,0,0)$

Tabelle 2.1: *Zwei verschiedene Darstellungen des Körpers $\mathbb{F}_{2^4}$*

2.2.2 Optimale Normalbasis (ONB)

Die Darstellung eines endlichen Körpers, speziell $\mathbb{F}_{2^n}$, durch eine Normalbasis (siehe Def. 2.2.2), hat den Vorteil, dass die Quadrierung von Körperelementen ohne Multiplikation möglich ist. Weiterhin ist bei der Multiplikation von Körperelementen die Anzahl von Operationen über $\mathbb{F}_2$ minimal. Diese Art der Darstellung lässt sich vor allem in Hardware effizient umsetzen. Für eine Implementierung in Software zeigt sich die Wahl der polynomiellen Darstellung als performanter. Siehe dazu auch [DWMPM98].
Ein weiterer Punkt, der die Wahl einer *Optimalen Normalbasis* (ONB) zur Darstellung von $\mathbb{F}_{2^n}$ für diese Arbeit ausschließt, ist die Tatsache, dass gewisse Teile davon patentrechtlich geschützt sind [OM86][MOV96, Kapitel 15].
Dennoch soll hier kurz erläutert werden wie eine ONB-Darstellung aussieht.
Zunächst eine ganz allgemeine Definition der Normalbasis für endliche Körper $\mathbb{F}$.

Definition 2.2.2. *(Normalbasis) Eine Basis* $\{\beta_0, \beta_1, \ldots, \beta_{n-1}\}$*, von* $\mathbb{F}_{p^n}$ *über* $\mathbb{F}_p$*, wobei* $\beta_i \in \mathbb{F}_{p^n}$*, heißt* Normalbasis*, wenn* $\beta_i = \beta^{p^i}$ *für ein* $\beta \in \mathbb{F}_{p^n}$ *und* $1 \leq i \leq n-1$*. Die Potenzen* $\beta^p, \beta^{p^2}, \ldots, \beta^{p^{n-1}}$ *sind die sog.* Konjugierten *von* β *über* $\mathbb{F}_p$*. Man sagt,* β erzeugt *die Normalbasis.*

Weiterhin gilt das *Normal-Basis-Theorem*, dessen Beweis [LN83, Theorem 2.35] entnommen werden kann.

Theorem 2.2.3. *(Normal-Basis-Theorem) Zu jedem endlichen Körper* $\mathbb{F}_{p^n}$ *existiert eine Normalbasis über* $\mathbb{F}_p$*.*

Für $\mathbb{F}_{2^n}$ lautet die Definition von ONB laut [BRS98] folgendermaßen:

Definition 2.2.4. *Sei* $\mathcal{B} = \{\beta_0, \beta_1, \ldots, \beta_{n-1}\}$*,* $\beta_i \in \mathbb{F}_{2^n}$ *eine Normalbasis von* $\mathbb{F}_{2^n}$ *über* $\mathbb{F}_2$ *und für alle* $i_1 \neq i_2$ *mit* $0 \leq i_1, i_2 \leq n-1$ *gibt es* j_1, j_2 *so das gilt* $\beta^{2^{i_1}+2^{i_2}} = \beta^{2^{j_1}} + \beta^{2^{j_2}}$*, dann nennt man* $\mathcal{B}$ optimale Normalbasis*.*

Nach [MBG+93] ist bekannt, dass es ONB in $\mathbb{F}_{2^n}$ nur in den folgenden beiden Fällen gibt:

- **ONB Typ I** Bedingung: $p = n+1$ ist eine Primzahl und $\langle 2 \rangle = \mathbb{Z}_p^*$
- **ONB Typ II** Bedingung: $p = 2n+1$ ist eine Primzahl und es gilt zusätzlich
  - (i) $\langle 2 \rangle = \mathbb{Z}_p^*$ oder
  - (ii) $p \equiv 3 \pmod 4$ und $\langle 2 \rangle = QR_p$ (d.h. 2 erzeugt alle quadratischen Reste modulo p was dazu führt, dass die multiplikative Ordnung von $2 \mod p$ gleich n ist).

Die Bedingungen für die Charakterisierung von ONB Typ I/II macht eine allgemeine Verwendung für endliche Körper der Charakteristik 2 nicht möglich. Insbesondere lässt sich keine ONB Typ I für die NIST-Polynome [Nat00] erzeugen, da es sich dort beim Grad n der Körpererweiterung $[\mathbb{F}_{2^n} : \mathbb{F}_2]$ bereits um eine Primzahl $(163, 233, 283, 409$ und $571)$ handelt.
Mehr zu ONB findet sich in [GL92] und [BRS98].

2.2.3 Zusammengesetzte Erweiterungskörper

Für diese Art der Darstellung stellt man den Körper $\mathbb{F}_{2^n}$ nicht als Vektorraum direkt über $\mathbb{F}_2$, sondern über einer Körpererweiterung $[\mathbb{F}_{2^m} : \mathbb{F}_2]$ vom Grad m dar. Wie man an der Konstruktion schon erkennt, können damit nur endliche Körper mit $2^{m \cdot n}$ Elementen dargestellt werden, was die Anwendung für die NIST-Polynome ausschließt.

Dennoch soll diese Methode kurz am aus [HJS93] entnommenen Beispiel des Körpers $\mathbb{F}_{2^{104}}$ als Vektorraum über $\mathbb{F}_{2^8}$ erläutert werden.
Das Polynom

$$g(x) = x^8 + x^7 + x^3 + x^2 + 1$$

ist irreduzibel über $\mathbb{F}_2$. Damit lässt sich $\mathbb{F}_{2^8}$ als $\mathbb{F}_2[x]/g(x)$ erzeugen.
Ein zweites Polynom

$$f(x) = x^{13} + x^7 + x^6 + x + 1$$

ist ebenfalls irreduzibel über $\mathbb{F}_2$ und weiterhin auch irreduzibel über $\mathbb{F}_{2^8}$, da $gcd(8,13) = 1$. Die Elemente aus $\mathbb{F}_{2^{104}}$ schreiben wir nun als Polynome aus $\mathbb{F}_{2^8}[x]$ vom Grad kleiner 13. Somit schreiben sich Polynome die ein Element aus $\mathbb{F}_{2^{104}}$ repräsentieren folgendermaßen:

$$\sum_{i=0}^{12} a_i x^i = a_{12}x^{12} + \ldots + a_1 x + a_0 \quad a_i \in \mathbb{F}_{2^8}$$

Durch die Zerlegung des großen Körpers $\mathbb{F}_{2^{104}}$ ist es nun möglich die Implementierung mit Hilfe von sogenannten *Lookup-Tables* zu beschleunigen. Die Elemente des kleinen Körpers $\mathbb{F}_{2^8}$ werden dabei vorberechnet und in zwei Tabellen gespeichert. Diese nennt man beispielsweise *log* und *antilog* und belegt sie mit folgenden Werten:

$$\begin{aligned} log[\alpha] &= i, \qquad \text{mit } \alpha = \beta^i \text{ und } 0 \leq i \leq 254 \\ antilog[i] &= \alpha, \qquad \text{mit } \alpha = \beta^i \text{ und } 0 \leq i \leq 254 \end{aligned}$$

Dabei sei β ein erzeugendes Element der multiplikativen Gruppe (siehe Satz 2.1.24). Die Tabelleneinträge speichert man als Binärvektoren der polynomiellen Darstellung. Die Multiplikation von Körperelementen (Koeffizienten) aus $\mathbb{F}_{2^8}$ reduziert sich dann auf einfache Lookups in den entsprechenden Tabellen. Für $a, b \in \mathbb{F}_{2^8}$ gilt dann:

$$a \cdot b = antilog[(log[a] + log[b]) \mod 255]$$

Bemerkung 1. *Dieser Trick wird auch bei der Implementierung des* AES *verwendet und wird in [RD02] genauer beschrieben.*

Die Teilkörper deren Elemente man in einer Lookup-Table speichert, kann man dabei an die Größe des verwendeten Prozessoregisters anpassen, so dass nicht nur der Körper $\mathbb{F}_{2^8}$ [HJS93] interessant ist, sondern auch $\mathbb{F}_{2^{16}}$ noch von Interesse ist (siehe [DWBV96] [Gua97]). Die Verwendung von Lookup-Tables für $\mathbb{F}_{2^{32}}$ ist aufgrund beschränkter Hardware-Resourcen nur eingeschränkt denkbar. Immerhin benötigt man hier nur für die beiden Lookup-Tables bereits $2 \cdot 2^{32} \cdot 4\text{Byte} = 32$ GB Speicher.

Die Verwendung von zusammengesetzten Körpern für die Kryptographie auf Elliptischen Kurven gilt jedoch als nicht sicher, da in [SHG00] eine mögliche Attacke beschrieben wird.

2.2.4 Auswahl der Darstellung

Für diese Arbeit wird die polynomielle Darstellung (siehe 2.2.1) gewählt. Dies hat vor allem den Vorteil, dass damit eine allgemeine Bibliothek für polynomielle Arithmetik für endliche Körper $\mathbb{F}_{2^n}$ beliebiger 2er-Potenz implementiert werden kann.

Wichtig für die endlichen Körper sind die irreduziblen Polynome $f(x) \in \mathbb{F}_2[x]$, welche eine Darstellung des entsprechenden Körpers festlegen. Um speziell die Reduktion

möglichst effizient implementieren zu können, ist es wichtig irreduzible Polynome zu besitzen, deren Hamming-Gewicht klein ist.
Das heißt, man sucht Polynome mit einer minimalen Anzahl an Koeffizienten $a_i = 1$. Das Minimum für dieses Hamming-Gewicht ist dabei 3. Diese Polynome nennt man allgemein *Trinome*. Polynome mit fünf 1-Koeffizienten nennt man *Pentanome*. Nach einem Theorem von Swan [Swa62] existieren Trinome nicht für einen Grad n, wenn $n \equiv 0 \bmod 8$ und sie sind selten für $n \equiv 3 \bmod 8$ oder $n \equiv 5 \bmod 8$. Die Frage, ob es für alle Grade n Pentanome gibt, ist noch nicht geklärt.

Mehr zur Existenz von Trinomen und Pentanomen, sowie eine Tabelle irreduzibler Tri- und Pentanome bis zum Grad $n = 10000$ findet sich in [Ser98],[BGL96] und [CQS99] sowie in der darin aufgeführten Literatur.

Kapitel 3

Implementierung von $\mathbb{F}_{2^n}$

In diesem Kapitel werden die für diese Arbeit spezifischen Implementierungsdetails näher erläutert. Grundlage für die interne Darstellung der Elemente von $\mathbb{F}_{2^n}$ ist dabei die polynomielle Darstellung (siehe 2.2.1 auf Seite 15). Im weiteren Verlauf werden existierende Algorithmen für die Körperarithmetik vorgestellt, sowie auf eine Tauglichkeit im embedded Umfeld hin untersucht.
Ein Timing- und Resourcen Vergleich der implementierten Algorithmen rundet dabei die jeweilige Betrachtung ab.
Implementierungsvorschläge für 32-Bit Systeme finden sich in [Ros99] sowie [HMV04].

3.1 Darstellung der Polynome

Als erstes gilt es zu überlegen, wie die Körperelemente im Rechner dargestellt werden sollen. Wie man in Kapitel 2.2.1 gesehen hat, lassen sich die Elemente aus $\mathbb{F}_{2^n}$ als Polynome $g = \sum_{i=0}^{n-1} a_i x^i = a_{n-1}x^{n-1} + \ldots + a_2x^2 + a_1x + a_0 \in \mathbb{F}_2[x]$ auffassen und als Binärvektoren ihrer Koeffizienten $a = (a_{n-1}, \ldots, a_1, a_0)$ der Länge n schreiben. Somit können pro Byte acht Koeffizienten gespeichert werden. Die Anzahl benötigter Datenfelder(Wörter) zur Speicherung eines Polynoms errechnet sich mit $t = \lceil n/W \rceil$. Innerhalb des höchstwertigen Wortes gibt es $s = W \cdot t - n$ *freie* Bits, die nicht mehr zur Darstellung des Binärvektors verwendet werden. Diese *freien* Bits werden standardmäßig mit Nullen belegt und sind so für eine weitere Betrachtung nicht relevant. Die Größe W steht für die Anzahl an Bits, die in einem Wort gespeichert werden können (Wortbreite) und ist, abhängig vom verwendeten Datentyp, entweder 8 Bit, 16 Bit oder 32 Bit.
Um die Implementierung effizient zu gestalten, richtet man sich bei der Wahl des Datentyps nach der Registerbreite des Prozessors. In der erstellten Referenzimplementierung kann die Wahl des zu verwendenden Datentyps mit Hilfe einiger Compiler-Schalter erfolgen.
Das gesamte Polynom wird in einem Array $A = (A[t-1], \ldots, A[2], A[1], A[0])$ mit t Elementen gespeichert. In Abb. 3.1 ist dies an einem Beispiel für $n = 163$ und eine

Registerbreite $W = 16$ illustriert. Zur Speicherung der Koeffizienten beginnt man

175	159		31	15	0
A[t-1]		...	A[1]	A[0]	
0...0	$a_{162}, a_{161},$	...	, a_2, a_1, a_0		

Abbildung 3.1: *Darstellung eines Polynoms für $n = 163$ und $W = 16Bit$*

im niederwertigsten Bit des Arrayelements $A[0]$ und speichert dort den Koeffizienten a_0. Die weiteren Koeffizenten $a_i, 1 \leq i \leq n-1$ speichert man nun in aufsteigender Reihenfolge in den Bits der jeweiligen Arrayelemente $A[j], 0 \leq j < t$.

3.2 Arithmetik im Rechner

Durch die zuvor beschriebene Darstellung der Polynome im Rechner, ist es möglich die einzelnen arithmetischen Operatoren plattformunabhängig zu implementieren. Um die in der Folge betrachteten Algorithmen und Operationen möglichst modular zu realisieren, sind zunächst einige Primitive für den Umgang mit Polynomen notwendig. Diese werden hier nicht näher beleuchtet, sondern sollen nur kurz erwähnt, sowie deren Verwendung geklärt werden. Näheres findet sich in den entsprechenden Headerfiles.
Häufig benötigt wird in den nächsten Algorithmen die Möglichkeit ein Polynom mit x^i zu multiplizieren oder durch x^i zu dividieren. Diese beiden Operationen lassen sich durch einfache Links- und Rechtsshifts des gegebenen Polynoms realisieren. Die beiden Funktionen können zwischen einer und n Stellen schieben, wobei die Wortgröße in der Implementierung berücksichtigt wird. Jeweils leere Stellen werden mit 0 aufgefüllt.
Um den Wert einzelner Koeffizienten abzufragen (zu setzten), gibt es Funktionen die diese Aufgaben erfüllen. Eine Funktion zur Gradbestimmung von Polynomen ($n = \deg(f)$) rundet die Palette der Primitive ab.

Mit Hilfe dieser einfachen Primitive können nun die verschiedenen Algorithmen und Operationen effizient implementiert werden. Die einfachste Operation stellt dabei die Addition dar.

3.2.1 Addition von Polynomen

Seien $f, g, h \in \mathbb{F}_2[x]$ vom Grad kleiner n und damit Repräsentanten von Elementen aus $\mathbb{F}_{2^n}$. Die Addition zweier Polynome $h = g + f$, kann durch eine XOR-Operation ($\oplus$) der entsprechenden Koeffizienten berechnet werden. Da bei der Addition der Grad des Ergebnispolynoms nicht größer werden kann als der größte Grad der Summanden, ist keine Reduktion modulo dem irreduziblen Polynom notwendig.
Für die Implementierung bedeutet dies, dass man die jeweiligen Arrayelemente ($A[i], B[i], 0 \leq i < t$) direkt XOR verknüpfen darf. Abbildung 3.2 zeigt dieses für den allgemeinen Fall $h = g + f$ in Arrayschreibweise $C[i] = A[i] \oplus B[i], \quad 0 \leq i < t$.
Die Anzahl der XOR-Verknüpfungen für die Addition hängt damit nur von der verwendeten Registerbreite und dem Grad des irreduziblen Polynoms (welches $\mathbb{F}_{2^n}$ erzeugt)

A =	A[t-1]	...	A[1]	A[0]
B =	B[t-1]	...	B[1]	B[0]
C=A ⊕ B	A[t-1] ⊕ B[t-1]	...	A[1] ⊕ B[1]	A[0] ⊕ B[0]

Abbildung 3.2: *Addition zweier Polynome*

ab. Eine Implementierung dieser Funktion kommt demzufolge ohne temporären Speicher aus.
Die Subtraktion ist für Koeffizienten aus $\mathbb{F}_2$ nicht weiter interessant, da sie genau der Addition entspricht.

3.2.2 Multiplikation von Polynomen

Für die Multiplikation von Polynomen gibt es verschiedene Ansätze und Algorithmen. Die wesentlichen Algorithmen sollen dabei in diesem Kapitel beschrieben werden, sowie auf eine Verwendung auf embedded Systemen hin untersucht werden. Referenzen für die hier aufgeführten Algorithmen finden sich in [HHM01], [HMV04], [GG03], [HPCTZ99] sowie [SOO95].
Durch die Multiplikation zweier Polynome $a, b \in \mathbb{F}_2[x]$ vom Grad kleiner n, ergibt sich für das Ergebnispolynom $c(x) = a(x) \cdot b(x)$ ein maximaler Grad $\deg(c) = 2n - 2$. Da man nur mit den kleinsten Repräsentanten der Elemente von $\mathbb{F}_{2^n}$ rechnen möchte, muss c noch modulo dem irreduziblen Polynom f reduziert werden. Wie man in Kapitel 3.2.4 sehen wird, lässt sich die Reduktion modulo $f(x)$ als einfache Addition ($\oplus$) mit einem bestimmten Restpolynom $r(x)$ implementieren.

Shift-and-Add Algorithmus Den Anfang macht ein Algorithmus (3.2.1) der die *Schiebe und Addiere* (*shift-and-add*) Methode verwendet. Grundlage dazu ist die Tatsache, dass man für die Polynommultiplikation $a \cdot b$ mit $a(x) = \sum_{i=0}^{n-1} a_i x^i$ und $b(x) = \sum_{i=0}^{n-1} b_i x^i$ und $a, b \in \mathbb{F}_2[x]$ auch schreiben kann:

$$a(x) \cdot b(x) = a_{n-1}x^{n-1}b(x) + \ldots + a_2x^2b(x) + a_1xb(x) + a_0b(x)$$

Die Multiplikation des Polynoms $b(x)$ mit x entspricht einem Linksshift des Koeffizientenvektors was leicht einzusehen ist.

$$\begin{aligned} b(x) \cdot x &= (b_{n-1}x^{n-1} + \ldots + b_2x^2 + b_1x + b_0) \cdot x \\ &= b_{n-1}x^n + \ldots + b_2x^3 + b_1x^2 + b_0x \\ &\equiv b_{n-1}r(x) + b_{n-2}x^{n-1} + \ldots + b_2x^3 + b_1x^2 + b_0x \mod f(x) \end{aligned}$$

Algorithmus 3.2.1 Shift-and-Add (Right-to-left) Körpermultiplikation

INPUT: Binärvektoren der Polynome $a(x)$, $b(x)$ vom Grad $\leq n-1$.
OUTPUT: $c(x) = a(x) \cdot b(x) \mod f(x)$

1: **if** $a_0 = 1$ **then** $c \leftarrow b$ **else** $c \leftarrow 0$ ▷ Initialisierung
2: **for** $i \leftarrow 1$ **to** $m-1$ **do**
3: $\quad b \leftarrow b \cdot x \mod f(x)$ ▷ $b \cdot x$ entspricht linksshift
4: $\quad$ **if** $a_i = 1$ **then** $c \leftarrow c \oplus b$
5: **end for**
6: **return** c

Dieser Algorithmus ist vor allem für Hardware-Systeme geeignet, auf denen Shiftoperationen für Binärvektoren innerhalb eines Taktzyklus abgearbeitet werden können. Die hohe Anzahl n an Shiftoperationen des gesamten Vektors macht diese Methode für Softwareimplementierungen wenig empfehlenswert (siehe [HHM01], [HMV04]), weshalb dieser Algorithmus auch nicht implementiert wurde.

Right-to-left comb Algorithmus Dieser Algorithmus funktioniert nach der *right-to-left comb* Methode . Die beiden zu multiplizierenden Polynome $a(x) = \sum_{i=0}^{n-1} a_i x^i$ und $b(x) = \sum_{i=0}^{n-1} b_i x^i$ mit $a, b \in \mathbb{F}_2[x]$ sind in den Arrays A und B gespeichert. Dabei werden die Koeffizienten der einzelnen Arrayelemente $A[i], B[i]$ von rechts nach links (also vom niederwertigsten zum höchstwertigen Bit) betrachtet. Innerhalb der Arrayelemente beginnt man ebenfalls mit dem niederwertigsten Bit.
Grundlage für diese Art der Algorithmen ist die Tatsache, dass man aus der Berechnung von $a(x) \cdot x^k, 0 \leq k < W$ durch Multiplikation mit x^{Wj}, was einem Links-Shift um j Arrayelementen entspricht, das Polynom $a(x) \cdot x^{Wj+k} = (a(x) \cdot x^k) \cdot x^{Wj}$ erhält.

$$
\begin{aligned}
a(x) \cdot b(x) = \quad & a_0 b(x) + a_W b(x) x^W + \ldots + a_{(t-1)W} x^{(t-1)W} \\
& + a_1 b(x) x + a_{W+1} b(x) x^{W+1} + \ldots + a_{(t-1)W+1} x^{(t-1)W} \\
& \quad \vdots \\
& + a_{W-1} b(x) x^{W-1} + a_{2W-1} b(x) x^{2W-1} + \ldots + a_{tW-1} x^{tW-1}
\end{aligned}
$$

Ein Links-Shift von der Größe eines Arrayelements wird in Algorithmus 3.2.2 durch eine nach links verschobene Addition der Array-Elemente realisiert.
In Abbildung 3.3 ist dies exemplarisch für eine Registerbreite $W = 16$ und $n = 80$ dargestellt. Um den Algorithmus formal beschreiben zu können, wird die folgende Notation verwendet: $C = (C[v], \ldots, C[1], C[0])$ stellt ein Array (mit $v+1$ Elementen) für den Binärvektor eines Polynoms dar. Das Arrayelement $C[v]$ bezeichnet dabei das Element mit dem höchsten Index. Das Teilarray $C\{j\}$ von C besteht aus den Elementen $C\{j\} = (C[v], C[v-1], \ldots, C[j+1], C[j])$ mit $0 \leq j \leq v$. Anschaulich gesprochen bedeutet dies, dass $C\{j\}$ aus den $v-j+1$ höchstwertigen Elementen von C besteht. Abbildung 3.4 verdeutlicht diesen Zusammenhang.
In dem folgenden Algorithmus steht in Array C das noch nicht reduzierte Ergebnis der Polynommultiplikation $a \cdot b$, womit $v = 2t - 2$.

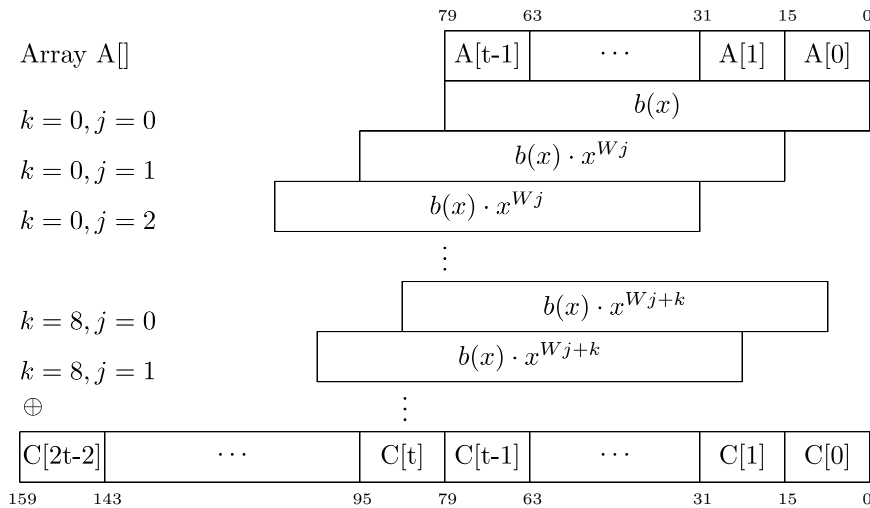

Abbildung 3.3: *Multiplikation mit dem Algorithmus 3.2.2 (*right-to-reft comb*) für die Parameter* $W = 16$ *und* $n = 80$

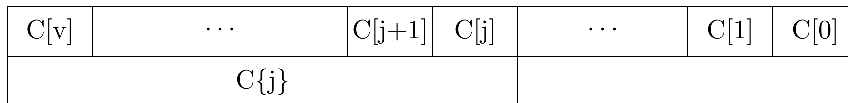

Abbildung 3.4: *Darstellung der Notation für die Verwendung von Teilarrays*

Algorithmus 3.2.2 Right-to-left comb Körpermultiplikation

INPUT: Binärvektoren der Polynome $a(x)$, $b(x)$ vom Grad $\leq n - 1$.
OUTPUT: $c(x) = a(x) \cdot b(x)$

1: $C \leftarrow 0$
2: **for** $k \leftarrow 0$ **to** $W - 1$ **do**
3:     **for** $j \leftarrow 0$ **to** $t - 1$ **do**
4:         **if** $a_{jW+k} = 1$ **then** $C\{j\} \leftarrow C\{j\} \oplus B$ ▷ a_{jW+k} entspricht $k$tem Bit in A[j]
5:     **end for**
6:     **if** $k \neq (W - 1)$ **then** $B \leftarrow B \cdot x$ ▷ Linksshift von B
7: **end for**
8: **return** C ▷ $\deg(c(x)) \leq 2n - 2$

Entscheidend für die Laufzeit dieses Algorithmus ist die Anzahl der Linksshifts. Diese ist nur abhängig von der Registerbreite W und beträgt genau $W - 1$
Im Gegensatz zum vorher betrachteten *shift-and-add* Algorithmus (Alg. 3.2.1) ist das Ergebnis in diesem Fall noch nicht reduziert. Dies bedeutet, der Grad des Ergebnispolynoms $\deg(c(x)) = m$ liegt im Bereich $0 \leq m \leq 2n - 2$. Dieser Umstand muss bei der Implementierung beachtet werden, denn man benötigt zur Darstellung von $c(x)$ ein Array C doppelter Länge und damit temporären Speicher.

Left-to-Right comb Algorithmus Die Multiplikation von Polynomen lässt sich, in Anlehnung an das Hornerschema, auch etwas anders schreiben:

$$\begin{aligned} a(x)\cdot b(x) = & \Big(a_{tW-1}b(x)x^{(t-1)W} + \ldots + a_{2W-1}b(x)x^W + a_{W-1}b(x)\Big)\cdot x^{W-1} \\ + & \Big(a_{tW-2}b(x)x^{(t-1)W} + \ldots + a_{2W-2}b(x)x^W + a_{W-2}b(x)\Big)\cdot x^{W-2} \\ & \vdots \\ + & \Big(a_{(t-1)W+1}b(x)x^{(t-1)W} + \ldots + a_{W+1}b(x)x^W + a_1 b(x)\Big)\cdot x \\ + & a_{(t-1)W}b(x)x^{(t-1)W} + \ldots + a_W b(x)x^W + a_0 b(x) \end{aligned}$$

Dabei entsprechen die Ausdrücke $b(x)x^{jW}$ einem Linksshift des Polynoms $b(x)$ um jW Bits oder einfacher, einer Addition ($\oplus$) von B nach $C\{j\}$. Der jeweils äußerst rechts stehende Faktor x^i entspricht einem Linksshift des Ergebnisarrays C und wird durch die Iteration genau $W-1$ mal benötigt.

Algorithmus 3.2.3 *Left-to-right comb* Körpermultiplikation

INPUT: Binärvektoren der Polynome $a(x)$, $b(x)$ vom Grad $\leq n-1$.
OUTPUT: $c(x) = a(x)\cdot b(x)$

1: $C \leftarrow 0$
2: **for** $k \leftarrow W-1$ **to** 0 **do**
3:     **for** $j \leftarrow 0$ **to** $t-1$ **do**
4:         **if** $a_{jW+k} = 1$ **then** $C\{j\} = C\{j\} \oplus B$ ▷ a_{jW+k} entspricht $k$tem Bit in A[j]
5:     **end for**
6:     **if** $k \neq 0$ **then** $C \leftarrow C\cdot x$ ▷ Linksshift von C
7: **end for**
8: **return** C ▷ $\deg(c(x)) \leq 2n-2$

Genau wie bei Algorithmus 3.2.2 ist auch bei der *Left-to-right comb* Methode die Anzahl der Linksshifts entscheidend für die Laufzeit des Algorithmus. Ebenso wie in der vorherigen Methode, ist diese Anzahl auch hier nur abhängig von der Registerbreite und beträgt damit $W-1$. Der Unterschied zwischen den beiden Algorithmen liegt jedoch in dem zu *schiebenden* Array. Während bei Alg. 3.2.2 das Array B geschoben wird, muss beim *Left-to-Right*-Algorithmus das doppelt so lange Ergebnisarray C geschoben werden. Wie sich dieser Unterschied im Laufzeitverhalten äußert, wird in Abschnitt 3.3 gezeigt.
López und Dahab beschreiben in [LD00] die Verwendung einer Lookup-Tabelle zur Beschleunigung des *Left-to-Right*-Algorithmus.

Left-to-right comb Window Algorithmus Bei diesem Algorithmus handelt es sich um eine Erweiterung von Algorithmus 3.2.3 wie sie in [LD00] beschrieben wird.
Man macht sich für die Multiplikation von $c(x) = a(x)\cdot b(x)$ mit $a, b, c \in \mathbb{F}_2[x]$ eine *Lookup-Tabelle* zunutze, die man zu Beginn des Algorithmus einmalig berechnet. Dadurch lässt sich die Multiplikation einer w-Bits langen Bitsequenz (Koeffizientenvektor $(a_w, a_{w+1}, \ldots a_{2w-1})$) durch ein einfaches *Table-Lookup* erledigen.

Die Tabelle wird folgendermaßen erzeugt. Man multipliziert alle Polynome $u(x)$ vom Grad $\deg(u(x)) < w$ mit dem Polynom $b(x)$. Der Index für die Tabelle entspricht dem Binärvektor der Koeffizienten von $u(x)$. Tabelle 3.1 zeigt dies exemplarisch für $w = 4$. Bei der Konstruktion der Tabelle kann man sich wieder die Eigenschaft der Polynom-

Index	**Binärvektor von $u(x)$**	**Lookup-Wert**
0	$(0,0,0,0)$	Nullpolynom
1	$(0,0,0,1)$	$b(x)$
2	$(0,0,1,0)$	$b(x)x$
4	$(0,1,0,0)$	$b(x)x^2$
8	$(1,0,0,0)$	$b(x)x^3$
3	$(0,0,1,1)$	$LU[1] \oplus LU[2]$
5	$(0,1,0,1)$	$LU[4] \oplus LU[1]$
6	$(0,1,1,0)$	$LU[4] \oplus LU[2]$
	$\vdots$	
15	$(1,1,1,1)$	$LU[8] \oplus LU[7]$

Tabelle 3.1: *Lookup-Tabelle für $w = 4$. $LU[i]$ bezeichnet den Tabelleneintrag mit Index i.*

multiplikation (Multiplikation von $b(x)$ mit x^n entspricht einem Linksshift von $b(x)$ um n-Stellen) zunutze machen und diese mit $w-1$ Linksshifts sowie $(2^{w-1}-1) + (2^{w-2}-1) + \ldots + (2^1-1) = \sum_{i=1}^{w-1}(2^i-1) = 2^w - w - 1$ XOR-Operationen erzeugen. Die einzelnen Summanden entsprechen jeweils der Anzahl an Möglichkeiten, um aus den bereits vorhandenen 2^{i-1} Binärvektoren und dem durch Linksshift erzeugten Binärvektor für 2^i, alle Kombinationen zu erzeugen. Ein Beispiel soll dies noch einmal verdeutlichen.

Beispiel 3.2.1. *Sei $b(x) \in \mathbb{F}_2[x]$ und $w = 4$. Die Lookup-Tabelle besteht aus $2^w = 2^4 = 16$ Einträgen. Man benötigt $w - 1 = 4 - 1 = 3$ Linksshifts um die Polynome $b(x)x, b(x)x^2$ und $b(x)x^3$ zu erzeugen. Zusammen mit dem Nullpolynom und $b(x)$ sind nun bereits fünf Einträge der Tabelle bekannt, es fehlen also noch $2^w - w - 1 = 11$ Einträge, die durch Addition (XOR) der vorhanden Einträge errechnet werden können. In Tabelle 3.1 sind die elf Additionen mit $LU[i_1] \oplus LU[i_2]$ beschrieben, wobei $LU[i]$ einen Tabellen-Lookup mit Index i bezeichnet.*

Diese Konstruktion entspricht im wesentlichen dem *Shift-and-Add* Algorithmus, wird aber für kleine w fest einprogrammiert. Für w sollte zusätzlich $w|W$ gelten, da damit die Wortgrößen innerhalb des Rechners berücksichtigt werden und es zu keinen Bereichsüberschreitungen (Verschnitt) innerhalb der Arrayelemente kommt.
Die Größe der Tabelle im Speicher hängt nur von w und dem maximalen Grad n von $b(x)$ ab. Der höchste Exponent der bei einer Multiplikation von $b(x)$ mit $u(x)$ auftreten kann, hat den Grad $n - 1 + w - 1 = n + w - 2$. Für die einzelnen Arrays der Lookup-Table bedeutet dies, dass sie $w - 1$ Bits mehr aufnehmen können müssen als das Array B.
Ist $w < W$ so muss dass Array für $u(x) \cdot b(x)$ ein Element größer sein als das Array B,

wobei B die Arrayrepräsentation des Polynoms $b(x) \in \mathbb{F}_2[x]$ ist, denn $u(x) \cdot b(x)$ hat den Grad $n + w - 2 < (t + 1)W$.

Beispiel 3.2.2. *Sei $n = 163, W = 16, w = 4$ und damit $t = \lceil n/W \rceil = 11$. Für die Lookup-Table werden $2^w = 16$ Arrays der Länge $t + 1 = 12$ benötigt. Damit ergibt sich für die Tabelle eine Größe von $2^w(t + 1) = 16 \cdot 12 = 192$ Arrayelementen. Für $W = 16$ sind das entsprechend 384Byte.*
Für $w = 8$ ergibt sich bereits ein Speicherbedarf von 6KByte was eine Verwendung im embedded Umfeld nahezu ausschließt.

Algorithmus 3.2.4 *Left-to-right comb with Window* Körpermultiplikation

INPUT: Binärvektoren der Polynome $a(x)$, $b(x)$ vom Grad $\leq n - 1$.
OUTPUT: $c(x) = a(x) \cdot b(x)$

```
 1: Berechne Lookup-Tabelle B_u = u(x)·b(x) für alle u(x) mit deg(u) ≤ w − 1
 2: C ← 0
 3: for k ← ⌈W/w⌉ − 1 to 0 do
 4:     for j ← 0 to t − 1 do
 5:         u ← (u_{w−1}, ..., u_1, u_0)        ▷ mit u_i entspricht Bit (wk + i) von A[j]
 6:         C{j} ← B_u ⊕ C{j}
 7:     end for
 8:     if k ≠ 0 then C ← C · x^w              ▷ Linksshift von C um w Bits
 9: end for
10: return C                                   ▷ deg(c(x)) ≤ 2n − 2
```

Karatsuba-Ofman Multiplikation Die *Karatsuba-Ofman*-Methode funktioniert nach dem *divide-and-conquer* Prinzip und wurde zunächst für die Multiplikation von Ganzzahlen beschrieben [Knu81]. Divide-and-conquer lässt sich sinngemäß mit teile-und-herrsche[1] übersetzen und bedeutet, ein Problem solange aufzuteilen, bis man es beherrschen kann.
Übertragen auf die Multiplikation von Polynomen heißt das, dass man zwei Polynome $a(x)$, $b(x)$ von hohem Grad $< m$ in jeweils zwei Polynome a_1, a_0, b_1, b_0 mit

$$a(x) = a_1 x^{m/2} + a_0, \qquad b(x) = b_1 x^{m/2} + b_0$$

zerlegt und dann versucht diese zu multiplizieren (zu beherrschen). Falls dies noch nicht möglich ist, zerlegt man die entstandenen Polynome rekursiv weiter, bis man bei einer beherrschbaren Länge angekommen ist. Polynome dieser Länge werden dann mit den klassischen Methoden multipliziert und das Ergebnis nach oben durchgereicht.

Beispiel 3.2.3. *Sei* $\deg(a), \deg(b) < m$ *und* $l = \lceil m/2 \rceil$.

$$\begin{aligned} a(x) \cdot b(x) &= (a_1 x^l + a_0)(b_1 x^l + b_0) \\ &= a_1 b_1 x^{2l} + (a_1 b_0 + a_0 b_1) x^l + a_0 b_0 \\ &= a_1 b_1 x^{2l} + ((a_1 + a_0)(b_1 + b_0) + a_1 b_1 + a_0 b_0)\, x^l + a_0 b_0 \end{aligned}$$

[1] siehe http://www.nist.gov/dads/HTML/divideconqr.html

Durch die Umformung von der vorletzten zur letzten Zeile wird die Anzahl der weiter zu betrachtenden Produkte von 4 auf 3 reduziert. Die Faktoren dieser Produkte $(a_1b_1, (a_1 + a_0)(b_1 + b_0), a_0b_0)$ *sind alle vom Grad* $\leq l$. *Ist die Polynommultiplikation mit Polynomen vom Grad* $\leq l$ *effizient mit Standardalgorithmen beherrschbar, so werden diese drei Produkte nun direkt berechnet. Andernfalls wiederholt man diese Prozedur für alle 3 Produkte bis man zu beherrschbaren Graden kommt. Mehr zur Karatsuba-Methode für Polynome findet sich auch in [WP02]. In [WSCS03] findet man einen möglichen Algorithmus für die Karatsuba Methode, der hier, als Algorithmus 3.2.5, in etwas vereinfachter Form dargestellt ist.*

Algorithmus 3.2.5 Karatsuba Multiplikation von Polynomen

INPUT: Binärvektoren der Polynome $a(x)$, $b(x)$ die aus s Wörtern bestehen,
OUTPUT: $c(x) = a(x) \cdot a(x)$

```
1:  function KARATSUBA(a, b, s)                              ▷ Karatsuba Funktion
2:      if s ≤ 4 then                  ▷ Ende der Rekursion (beherrschbarer Grad)
3:          Multipliziere klassisch (z.B. Alg. 3.2.3)
4:      else
5:          Zerlege a(x) und b(x) in a0, a1, b0, b1
6:          mit den Längen a1, b1 = ⌊s/2⌋ und a0, b0 = ⌈s/2⌉.
7:          ab1 ← KARATSUBA(a1, b1, ⌊s/2⌋)                  ▷ Rekursive Aufrufe
8:          ab0 ← KARATSUBA(a0, b0, ⌈s/2⌉)
9:          ab ← KARATSUBA((a1 ⊕ a0), (b1 ⊕ b0), ⌈s/2⌉)
10:         c ← ab1 x^(⌈s/2⌉·2) ⊕ (ab ⊕ ab1 ⊕ ab0) x^⌈s/2⌉ ⊕ ab0   ▷ Ergebnis einer Berechnung
11:     end if
12:     return C
13: end function
```

Für die Zerlegung der Polynome gibt es zwei Möglichkeiten. Man kann entweder immer genau in der Mitte der Grade teilen $(m/2)$, oder man orientiert sich bei der Zerlegung an der Wortgröße des verwendeten Systems. Die in Algorithmus 3.2.5 verwendete Methode ist eine Kombination dieser beiden Arten. Die zu multiplizierenden Polynome a, b werden in der Mitte ihrer Wortdarstellung geteilt. Ist die Anzahl s der Wörter zur Darstellung der Polynome ungerade, so teilt man in $\lfloor s/2 \rfloor$ und $\lceil s/2 \rceil$. Die Rekursionsendebedingung sorgt bei beherrschbaren Graden für eine Multiplikation mit einem beliebigen Multiplikationsalgorithmus (z.B. Alg. 3.2.3). In Zeile 2 wurde dafür beispielhaft $s \leq 4$ gewählt. Diese Wahl hängt jedoch von der Performance der klassischen Multipliktationsalgorithmen ab und kann durchaus auch größer sein.
Andernfalls werden die zerlegten Polynome rekursiv multipliziert (Zeilen 7-9) bis die Endebedingung erfüllt ist.

Der rekursive Teil des Karatsuba-Algorithmus macht eine Verwendung in einem embedded System nicht empfehlenswert, da es dadurch zu einem großen Speicherbedarf kommt. Weiterhin verbietet der *MISRA-C* Standard [The04] die Verwendung von rekursiven Algorithmen. Eine iterative Implementierung würde nur für kleine Polynomgrade Sinn machen, da sonst die Codegröße für den embedded Bereich zu groß würde und auch der Speicherbedarf die vorhandenen Resourcen übersteigt. Umgeht

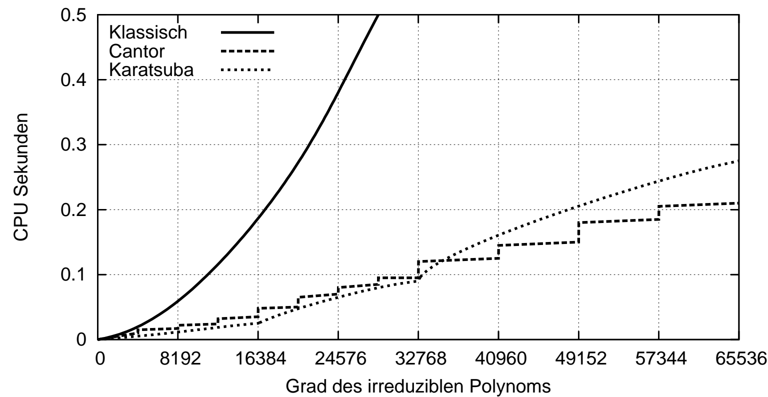

Abbildung 3.5: *Vergleich von Multiplikationsalgorithmen für Polynome aus $\mathbb{F}_2[x]$ mit hohem Grad. (übernommen aus [GG03])*

man die Rekursion durch eine Verwendung von Schleifen und Stacks, so wird auch in dieser Variante der RAM-Bedarf sehr hoch, was auch diese Methode unpraktikabel macht.

Man kann zeigen, dass die Karatsuba-Ofman Methode einen Aufwand von $O(n^{\log_2 3})$ besitzt, während die klassischen Multiplikationsalgorithmen allesamt einen Aufwand von $O(n^2)$ haben. Trotzdem ist die Karatsuba-Ofman Methode für Polynome von kleinem Grad nicht zwangsläufig schneller als die klassischen Varianten mit Lookup-Table. Ein Timingvergleich in [HHM01, Kapitel 3.5, Tabelle 3] zeigt dies für Polynome vom Grad $m = 163, 233$ und 283. Auch in [WSCS03] wird die Karatsuba-Methode vor allem für Server und resourcenreiche Systeme empfohlen. Aus diesen Gründen wurde in dieser Arbeit von der Implementierung des Karatsuba-Algorithmus für ein embedded System abgesehen. Eine Referenzimplementierung für den 32 Bit PC Sektor, die sich an dem oben beschriebenen Algorithmus orientiert, wurde zu Testzwecken erstellt und bestätigt die vorherigen Aussagen zur Laufzeit.

Die aus [GG03, Kapitel 9.7] entnommene Abbildung 3.5 zeigt einen Vergleich des klassischen Multiplikationsalgorithmus mit der Karatsuba-Methode und einem Algorithmus von Cantor [Can89]. Wie aus der Abbildung ersichtlich ist, hat der Cantor-Algorithmus Performancevorteile vor allem für sehr große Polynomgrade, weshalb er in dieser Arbeit auch nicht weiter behandelt wird.

3.2.3 Quadrieren von Polynomen

Das Quadrieren von Polynomen über $\mathbb{F}_2$ lässt sich im Rechner besonders gut implementieren, da man in Charakteristik 2 die Eigenschaft (Frobenius, Lemma 2.1.14) ausnutzen kann. Durch diese Eigenschaft verschwindet beim quadrieren der mittlere Term von $(a+b)^2 = a^2 + 2ab + b^2$. Quadriert man das Polynom $a(x) \in \mathbb{F}_2[x]$ so ergibt

sich folgendes Ergebnis:

$$\begin{aligned} a(x)^2 &= \big(a_{n-1}x^{n-1} + (a_{n-2}x^{n-2} + \ldots + a_1x + a_0)\big)^2 \\ &= a_{n-1}x^{(n-1)2} + \big(a_{n-2}x^{n-2} + (\ldots + a_1x + a_0)\big)^2 \\ &= a_{n-1}x^{2n-2} + \ldots + a_2x^4 + a_1x^2 + a_0 \end{aligned}$$

Übertragen auf den Binärvektoren von $a(x)$ bedeutet das, dass zwischen allen Elementen jeweils ein 0-Koffizient eingefügt werden muss. In Abbildung 3.6 ist dies veranschaulicht. Daran lässt sich erkennen, dass das Quadrieren von Elementen in $\mathbb{F}_{2^n}$ ein linearer Algorithmus ist. Für die Implementierung bieten sich zwei Varianten an.

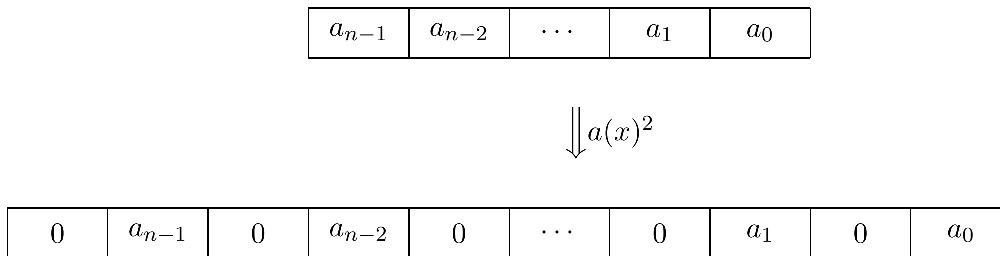

Abbildung 3.6: *Quadrieren eines Polynoms* $a(x)^2 = a_{n-1}x^{2n-2} + \ldots + a_2x^4 + a_1x^2 + a_0$

Die erste berechnet das Ergebnis unter zu Hilfe nahme einer Lookup-Tabelle, während in der zweiten Variante die einzelnen Bits verschoben werden.
Algorithmus 3.2.6 beschreibt die Variante mit Lookup-Tabelle für eine Wortbreite $W = 32$ Bit. Die Version für 16 Bit oder 8 Bit sieht entsprechend aus. Für die 16 Bit Version ist auch eine Variante möglich, welche die gleiche Tabelle verwendet wie ihr Pendant mit 32 Bit Wortbreite.

Algorithmus 3.2.6 Quadrieren von Polynomen (mit Wordlänge $W = 32$)

INPUT: Binärvektor des Polynoms $a(x)$ vom Grad $\leq n-1$.
OUTPUT: $c(x) = a(x)^2$
1: **for** $i \leftarrow 0$ **to** $i < 2^8$ **do** ▷ Für alle Bytes $D_i = (d_7, d_6, \ldots, d_1, d_0)$
2: $T(i) \leftarrow D_i^2 = (0, d_7, 0, d_6, 0, \ldots, 0, d_1, 0, d_0)$ ▷ Quadrat von D_i in Lookup-Table
3: **end for**
4: **for** $j \leftarrow 0$ **to** $t-1$ **do**
5: Sei $A[j] = (u_3, u_2, u_1, u_0)$ für u_i ist genau ein Byte D
6: $C[2j] \leftarrow (T(u_1), T(u_0))$ ▷ Expansion durch Table-Lookup
7: $C[2j+1] \leftarrow (T(u_3), T(u_2))$
8: **end for**
9: **return** C

Wie für alle Algorithmen mit Lookup-Table ist es auch hier wichtig, deren Größe sowie Beschaffenheit zu betrachten. Der Speicherbedarf ist nach dem in Algorithmus 3.2.6 beschriebenen Verfahren $2^8 \cdot 2 = 512$ Byte. Für den gleichen Algorithmus bei einer Wortbreite von $W = 16$ Bit ergibt sich für die Lookup-Tabelle eine Größe von $2^4 \cdot 1 = 16$ Byte.

Im Gegensatz zum Multiplikationsalgorithmus mit Lookup-Tabelle, ist die Struktur

der Tabelle in diesem Fall unabhängig von den beteiligten Polynomen, da sie, wie in Tabelle 3.2 zu sehen, nur eine Abbildung von Bitvektoren ist. Damit wäre es auch

Index	Vektor d	$\mapsto$	Quadrat
0	$(0,0,0,0)$	$\mapsto$	$(0,0,0,0,0,0,0,0)$
1	$(0,0,0,1)$	$\mapsto$	$(0,0,0,0,0,0,0,1)$
2	$(0,0,1,0)$	$\mapsto$	$(0,0,0,0,0,1,0,0)$
3	$(0,0,1,1)$	$\mapsto$	$(0,0,0,0,0,1,0,1)$
4	$(0,1,0,0)$	$\mapsto$	$(0,0,0,1,0,0,0,0)$
5	$(0,1,0,1)$	$\mapsto$	$(0,0,0,1,0,0,0,1)$
	$\vdots$		
15	$(1,1,1,1)$	$\mapsto$	$(0,1,0,1,0,1,0,1)$

Tabelle 3.2: *Lookup-Tabelle zum Quadrieren von Polynomen am Beispiel für $W = 16$ Bit*

möglich, die Tabelle vorzuberechnen, permanent im Speicher (ROM Bereich) abzulegen und dann an die *Square*-Funktion zu übergeben. Will man ROM Speicher sparen, ist auch eine temporäre Speicherung im RAM denkbar. Dies würde vor allem dann Sinn machen, wenn innerhalb eines Anweisungsblocks mehrere Quadrierungen zu erledigen sind. Somit spart man die Vorberechnung für jeden einzelnen Funktionsaufruf. Im Anschluss daran kann der RAM Speicher wieder von anderen Funktionen benutzt werden. Im embedded Umfeld gilt es hierbei zwischen Performance und RAM-Bedarf abzuwiegen, weshalb auch noch größere Tabellen im Allgemeinen nicht verwendet werden.

Eine triviale Methode zur Berechnung von $a(x)^2$ ohne Lookup-Tabelle stellt Algorithmus 3.2.7 dar.
Der Nachteil dieser Methode ergibt sich aus der Tatsache, dass alle Koeffizienten von $a(x)$ einzeln betrachtet werden müssen, um das Quadrat zu berechnen. Andererseits wird kein zusätzlicher Speicher benötigt um die Berechnung durchzuführen, was wiederrum für einen Einsatz spricht.

Algorithmus 3.2.7 Quadrieren von Polynomen ohne Lookup-Table

INPUT: Binärvektor des Polynoms $a(x)$ vom Grad $\leq n-1$.
OUTPUT: $c(x) = a(x)^2$
1: $C \leftarrow 0$ ▷ Intitialisierung
2: **for** $j \leftarrow 0$ **to** $\deg(a(x)) - 1$ **do** ▷ Alle Koeffizienten von $a(x)$ betrachten
3: $c_{2j} \leftarrow a_j$
4: **end for**
5: **return** C

3.2.4 Reduktion modulo $f(x)$

Die Multiplikation zweier Polynome $a, b \in \mathbb{F}_2[x]$ jeweils vom Grad $\leq n$, sowie das Quadrieren eines Polynoms vom Grad $\leq n$, liefern als Ergebnis Polynome vom Grad $\leq 2n-2$. Diese müssen modulo dem irreduziblen Polynom $f \in \mathbb{F}_2[x]$ reduziert werden. Die Reduktion modulo einem irreduziblen Polynom $f(x)$ ist über $\mathbb{F}_2$ besonders effektiv

durchführbar, da sie durch einfache Additionen ($\oplus$), sowie Linksshifts realisiert werden kann.
Sei $f(x) = x^n + r(x), \deg(r(x)) < n$ das irreduzible Polynom vom Grad n, dann gelten folgende einfache Kongruenzen:

$$
\begin{aligned}
x^n &\equiv r(x) \bmod f(x) \\
x^{n+1} = x^n \cdot x &\equiv r(x) \cdot x \bmod f(x) \\
x^{n+2} = x^n \cdot x^2 &\equiv r(x) \cdot x^2 \bmod f(x) \\
&\vdots \\
x^{n+i} = x^n \cdot x^i &\equiv r(x) \cdot x^i \bmod f(x)
\end{aligned}
$$

Für ein Polynom $c(x)$ vom Grad $\leq 2n-2$ bedeutet dies, dass man für alle Potenzen x^m mit $m \geq n$ auch $r(x) \cdot x^{m-n}$ schreiben kann. Somit erhält man folgende Kongruenz:

$$
\begin{aligned}
c(x) &= c_{2n-2}x^{2n-2} + \ldots + c_n x^n + c_{n-1}x^{n-1} + \ldots + c_1 x + c_0 \\
&\equiv c_{2n-2}r(x)x^{n-2} + \ldots + c_n r(x) + c_{n-1}x^{n-1} + \ldots + c_1 x + c_0 \bmod f(x)
\end{aligned}
$$

Die Terme $c_i r(x) x^{i-n}$ entsprechen einem Polynom vom Grad $\deg(r(x)) + (i-n)$ und werden zu $c(x)$ addiert.

In einer Implementierung kann der Ausdruck $r(x) \cdot x^{m-n}$ wieder durch $m-n$ Linksshifts dargestellt werden. Diese Tatsache macht auch hier die Verwendung einer Lookup-Table möglich, was in Algorithmus 3.2.8 dargestellt wird. Da man einen Linksshift um i Bits mit $i \geq W$ auch in $j = \lfloor i/W \rfloor$ Wortshifts und $k = i - jW$ Bitshifts darstellen kann, empfehlen sich für die Lookup-Table die Bitshifts.
Die Größe der Tabelle ergibt sich aus der Wortbreite W und der Länge des Polynoms $r(x)$. Da $\deg(r(x)) < n$ und maximal um W Bits geschoben wird, braucht man als obere Grenze für ein Polynom $r(x)x^k$ höchstens $t = \lceil n/W \rceil + 1$ Arrayelemente. Für eine Wortbreite $W = 16$ Bit und ein irreduzibles Polynom vom Grad $n = 163$ ergibt sich für die Lookup-Tabelle eine Größe von $W \cdot t = 16 \cdot 12 = 192$ Wörtern, was 384 Bytes entspricht.
Da das irreduzible Polynom $f(x)$ beim Programmstart festgelegt wird (d.h. die Darstellung des endlichen Körpers ist fest gewählt) und damit während des Programmlaufs fest bleibt, ändert sich auch $r(x)$ nicht. Damit wäre auch hier eine Vorberechnung der Tabelle, sowie eine Ablage im RAM, zu Beginn des Programms möglich. Auch eine Ablage im ROM ist denkbar.

Algorithmus 3.2.8 Reduktion modulo $f(x)$

INPUT: Binärvektor des Polynoms $c(x)$ vom Grad $\leq 2n-2$.
OUTPUT: $c(x) \mod f(x)$

```
 1: r(x) = f(x) − x^n                      ▷ Berechne Restpolynom
 2: for k ← 0 to W − 1 do                  ▷ Lookup-Table berechnen
 3:     u_k(x) = r(x)x^k
 4: end for
 5: for i ← 2n − 2 to n do                 ▷ Wenn Grad ≥ n reduzieren
 6:     if c_i = 1 then
 7:     j ← ⌊(i − n)/W⌋                    ▷ Welches Element in C
 8:     k ← (i − n) − Wj                   ▷ Welches Bit in C[j]
 9:     C{j} ← C{j} ⊕ u_k(x)
10: end for
11: return (C[t − 1], . . . , C[1], C[0])  ▷ deg(c(x)) < n
```

Neben der Verwendung von Lookup-Tables, gibt es noch eine zweite Möglichkeit um die Performance der Reduktionsalgorithmen zu steigern. Dabei geht es um die Wahl des irreduziblen Polynoms. Wählt man ein *Trinom* oder ein *Pentanom*, dessen mittlere Koeffizienten möglichst dicht beieinander liegen, so lässt sich die Reduktion Wortweise durchführen.

Beispiel 3.2.4. *Sei* $W = 16$ *und* $f(x) = x^{233}+x^{74}+1 = x^{233}+r(x)$ *das irreduzible Polynom und damit* $t = \lceil 233/16 \rceil = 15$. *Der höchste Grad der nach einer Polynommultiplikation* $c = a \cdot b, \deg(a) < n, \deg(b) < n$ *auftreten kann beträgt* $\deg(c) = 2n-2 = 464$. *Es wird nun das Arrayelement* $C[23]$ *mit den Koeffizienten* $c_{383}, \ldots, c_{368}$ *betrachtet. Die Reduktion schreibt sich zunächst folgendermaßen:*

$$c_{383}x^{383} + \ldots + c_{369}x^{369} + c_{368}x^{368} \bmod f(x)$$

Für die Potenzen x^i *lassen sich nun folgende Kongruenzen aufstellen:*

$$\begin{aligned} x^{383} &\equiv r(x)x^{383-233} \equiv r(x)x^{150} \bmod f(x) \\ x^{382} &\equiv r(x)x^{382-233} \equiv r(x)x^{149} \bmod f(x) \\ &\vdots \\ x^{368} &\equiv r(x)x^{368-233} \equiv r(x)x^{135} \bmod f(x) \end{aligned}$$

Setzt man diese Kongruenzen nun ein, so erhält man:

$$\begin{aligned} c_{383}r(x)x^{150} &+ c_{382}r(x)x^{149} + \ldots + c_{368}r(x)x^{135} \\ &\equiv r(x)(c_{383}x^{150} + \ldots + c_{368}x^{135}) \\ &\equiv (x^{74}+1)(c_{383}x^{150} + \ldots + c_{368}x^{135}) \\ &\equiv x^{74}(c_{383}x^{150} + \ldots + c_{368}x^{135}) + (c_{383}x^{150} + \ldots + c_{368}x^{135}) \\ &\equiv (c_{383}x^{224} + \ldots + c_{368}x^{209}) + (c_{383}x^{150} + \ldots + c_{368}x^{135}) \end{aligned}$$

An der letzten Zeile kann man nun gut den Vorteil eines Trinoms für die Implementierung der Reduktion erkennen. Wie man sieht, kann man auf diese Weise ein

komplettes Arrayelement (hier $C[23]$) in einem Schritt reduzieren, indem man das Element beginnend an den Bitstellen 209 und 135 zu C addiert ($\oplus$). Bei Trinomen muss das Element $C[i]$ zweimal, bei Pentanomen viermal addiert werden. In Abbildung 3.7 ist dies noch einmal graphisch dargestellt.

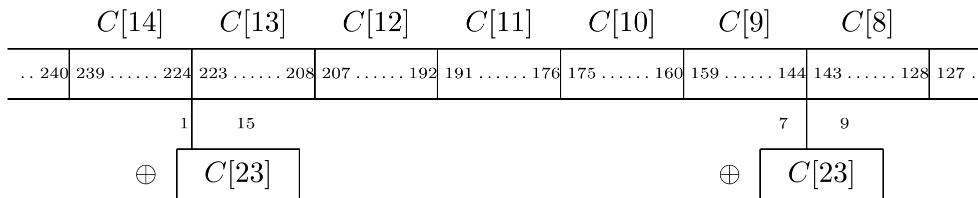

Abbildung 3.7: *Reduktion modulo $f(x) = x^{233} + x^{74} + 1$ für $W = 16(Bit)$*

Algorithmus 3.2.9 beschreibt eine wortweise Reduktion modulo $f(x) = x^{233}+x^{74}+1$ für eine Wortbreite von $W = 16$. Dieser Algorithmus ist dabei für das verwendete Polynom $f(x)$ optimiert, lässt sich aber auf beliebige andere Trinome und Pentanome anwenden. Im Vergleich, zu dem vorher betrachteten allgemeinen Reduktionsalgorithmus, ist diese spezielle Variante deutlich performanter und sie benötigt keinen zusätzlichen Speicher in der Form einer Lookup-Tabelle.
Von der NIST[2] werden im FIPS 186-2 Standard [Nat00] fünf Tri- und Pentanome vorgeschlagen, die sich auf die oben beschriebene Art und Weise effektiv in Reduktionsalgorithmen umsetzen lassen. In [HMV04] sind diese Algorithmen für $W = 32$ aufgeführt.

Algorithmus 3.2.9 Reduktion modulo $f(x) = x^{233} + x^{74} + 1$

INPUT: Binärvektor des Polynoms $c(x)$ vom Grad $\leq 2n - 2$.
OUTPUT: $c(x) \mod x^{233} + x^{74} + 1$

1: **for** $i \leftarrow 30$ **to** 15 **do** ▷ Nur die zu reduzierenden Elemente
2: $T \leftarrow C[i]$ ▷ Betrachtetes Element
3: $C[i-15] \leftarrow C[i-15] \oplus (T << 7)$ ▷ Entsprechenden Teil addieren
4: $C[i-14] \leftarrow C[i-14] \oplus (T >> 9)$
5: $C[i-10] \leftarrow C[i-10] \oplus (T << 1)$
6: $C[i-9] \leftarrow C[i-9] \oplus (T >> 15)$
7: **end for**
8: $T \leftarrow C[14]$ & 0xFE00 ▷ Betrachte nur noch die Bits $233, \ldots, 239$
9: $C[0] \leftarrow C[0] \oplus (T >> 9)$
10: $C[4] \leftarrow C[4] \oplus (T << 1)$
11: $C[5] \leftarrow C[5] \oplus (T >> 15)$
12: $C[14] \leftarrow C[14]$ & 0x01FF ▷ Lösche die reduzierten Bits $233, \ldots, 239$
13: **return** $(C[t-1], \ldots, C[1], C[0])$ ▷ Reduzierten Teil zurückgeben

3.2.5 Inversenbildung

Um mit Punkten auf Elliptischen Kurven rechnen zu können, benötigt man das *Inverse* oder Reziproke eines Elements aus $\mathbb{F}_{2^n}$. Für die Darstellung des Inversen eines Elements

[2] National Institute of Standards and Technology

$a \in \mathbb{F}_{2^n}$ verwendet man die Schreibweise $a^{-1} \bmod f(x)$, oder auch nur a^{-1}. Es gilt: $a \cdot a^{-1} \equiv 1 \bmod f$.
Für die Berechnung des Inversen werden im folgenden zwei verschiedene Methoden vorgestellt und verglichen.

Erweiterter Euklidischer Algorithmus für Polynome Bevor die Funktionsweise des *Erweiterten Euklidischen Algorithmus* (EEA) erklärt wird, soll zunächst einmal kurz der *normale* euklidische Algorithmus erläutert werden.
Der euklidische Algorithmus berechnet den größten gemeinsamen Teiler (kurz: $\gcd(a, b)$) zweier Zahlen a und b, lässt sich aber auch analog zur Berechnung von Polynomen $a(x), b(x) \in \mathbb{F}_2[x]$ verwenden. Dies liegt an der Tatsache, dass der euklidische Algorithmus in jedem euklidischen Ring durchführbar ist und es sich beim Polynomring $\mathbb{F}_2[x]$ um einen solchen handelt. Mehr zum euklidischen Algorithmus, sowie euklidischen Ringen findet sich in [LN83; Mey75a].
Sei im folgenden $a(x), b(x) \in \mathbb{F}_2[x]$ und $\deg(a(x)) > \deg(b(x)) > 0$. Der größte gemeinsame Teiler $c(x) = \gcd(a(x), b(x))$ berechnet sich durch fortgesetzte Division mit Rest (Polynomdivision). Die dazu nötige Zerlegung ist eindeutig, da $\mathbb{F}_2[x]$ ein euklidischer Ring ist. (Siehe dazu auch [LN83, Seite 20ff.].)

$$r_0 = a(x), \qquad r_1 = b(x)$$
$$\begin{aligned} r_0 &= q_1 r_1 + r_2, && \deg(r_2) < \deg(r_1) \\ r_1 &= q_2 r_2 + r_3, && \deg(r_3) < \deg(r_2) \\ &\vdots \\ r_{n-1} &= q_n r_n + r_{n+1}, && r_{n+1} = 0 \text{ (Nullpolynom)} \end{aligned}$$

Aus der letzten Zeile ergibt sich der gesuchte Zusammenhang für die Polynome a, b: $\gcd(a(x), b(x)) = r_n(x) = c(x)$. Dies ist auch leicht einzusehen, da ein rückwärtiges Lesen der obigen Gleichungen folgendes ergibt: $r_n | r_{n-1}, r_n | r_{n-2}, \ldots, r_n | r_1, r_n | r_0 \Rightarrow r_n | r_1 = b, r_n | r_0 = a$.
Der $\gcd(a, b)$ lässt sich auch noch folgendermaßen schreiben: $\gcd(a, b) = da + eb$ (Siehe auch [Buc04, Kapitel 2.9]). Die auftretenden Koeffizienten b, e sind ebenfalls Polynome in $\mathbb{F}_2[x]$ und lassen sich mit dem erweiterten Euklidischen Algorithmus berechnen.
Für den Fall der Inversenberechnung in $\mathbb{F}_{2^n}$ gilt zusätzlich: $f(x)$ irreduzibel über $\mathbb{F}_2$ und $\deg(f(x)) = n$. Daraus folgt, dass ein Polynom $(a(x), \deg(a(x)) < n)$ immer teilerfremd ist zu $f(x)$ und somit für den größten gemeinsamen Teiler gilt: $\gcd(a, f) = 1 = da + ef$
Rechnet man nun modulo $f(x)$, so ergibt sich die Kongruenz $da \equiv 1 \bmod f$, worin man d als a^{-1} identifizieren kann. Wie man sieht, ist eine Berechnung des zweiten Koeffizienten für die Inversenbildung nicht explizit nötig.
Für das Verständnis von Algorithmus 3.2.10 ist noch ein weiteres einfaches Theorem von Nöten.

Theorem 3.2.5. *Seien $a(x)$ und $b(x)$ Polynome über $\mathbb{F}_2[x]$.*
Es gilt $\gcd(a, b) = \gcd(b - ca, a)$ für alle binären Polynome $c(x)$.

Beweis. Sei $\gcd(a,b) = d$. Etwas umgeschrieben erhält man $\gcd(v_1 d, v_2 d) = d$ mit $a = v_1 d,\ b = v_2 d$ und $\gcd(v_1, v_2) = 1$. Für den rechten Teil der Gleichung gilt damit:

$$\gcd(b - ca, a) = \gcd(v_2 d - cv_1 d, v_1 d) = \gcd((v_2 - cv_1)d, v_1 d) = d = \gcd(a, b)$$

□

Unter Ausnutzung von Theorem 3.2.5 lässt sich zunächst der Euklidische Algorithmus vereinfachen. Wie bekannt ist, berechnet man für Polynome $a, b \in \mathbb{F}_2[x]$ mit $\deg(a) \leq \deg(b)$ die Division mit Rest r so, dass gilt

$$b = qa + r \quad \text{mit} \deg(r) < \deg(a).$$

Es lässt sich nun auch schreiben $\gcd(a,b) = \gcd(r,a)$. Damit reduziert sich das Problem des $\gcd(a,b)$ auf die Berechnung von $\gcd(r,a)$, wobei für den Grad von r gilt: $\deg(r) < \deg(a) \leq \deg(b)$. Der Algorithmus terminiert, da der Grad von r sukzessive kleiner wird, bis schließlich $\gcd(0,d) = d$ erreicht ist. Diese Methode benötigt im schlechtesten Fall $k = \deg(a)$ Polynomdivisionen mit Rest. Der in dieser Arbeit betrachtete Algorithmus 3.2.10 aus [HMV04] verwendet eine kleine Variante des Euklidischen Algorithmus. Dabei wird nicht eine komplette Polynomdivision durchgeführt, sondern man führt nur den ersten Divisionsschritt aus. Dieser berechnet für $\deg(b) \geq \deg(a)$ ein Monom $q(x) = x^j$ mit $j = \deg(q) = \deg(b) - \deg(a)$ für das gilt:

$$b(x) = q(x)a(x) + r(x) \Rightarrow r(x) = b(x) + x^j a(x).$$

Unter Zuhilfenahme von Theorem 3.2.5 $\gcd(a,b) = \gcd(r,a)$ kann man nun dieses Problem ebenfalls solange reduzieren, bis ein Rest $r_i = 0$ entsteht und man erhält $\gcd(0,d) = d$.
Durch diese besondere Konstruktion von r gilt für dessen Grad nur noch $\deg(r) < \deg(b)$. Es kann also durchaus dazu kommen, dass $\deg(r) > \deg(a)$. In einem solchen Fall macht man sich die Kommutativität des gcd zunutze und berechnet $\gcd(r,a) = \gcd(a,r) = \gcd(r - qa, a), \deg(r) > \deg(a)$. Damit ergibt sich für die maximale Anzahl an Divisionsschritten $2k$ mit $k = max(\deg(a), \deg(b))$.

Der untersuchte Algorithmus 3.2.10 verwendet zwei Invarianten

$$\begin{aligned} &\text{(I)} \qquad ag_1 + fh_1 = u \\ &\text{(II)} \qquad ag_2 + fh_2 = v, \end{aligned}$$

wobei die Koeffizientenpolynome h_1 und h_2 nur für die Berechnung einer allgemeinen Linearkombination des $\gcd(a,f) = da + ef$ benötigt werden und deshalb in diesem Fall nicht explizit berechnet werden müssen. Dies liegt an der Tatsache, dass für das irreduzible Polynom $f \in \mathbb{F}_2[x]$ und $a \in \mathbb{F}_2[x]$ gilt $\gcd(a,f) = 1$. Reduziert man nun mit f, so erhält man $\gcd(a,f) = 1 = da$, d.h. bei d handelt es sich um das Inverse von a, während der Faktor vor f durch die Reduktion verschwindet.
Beide Invarianten werden zu Beginn initialisiert (Zeile 1,2), so dass man

$$\begin{aligned} &\text{(I*)} \qquad a + f = a \\ &\text{(II*)} \qquad f = f \end{aligned}$$

erhält. Die nun folgenden Berechnungen werden solange durchgeführt, bis man in Gleichung I $u = 1$ erhält (Zeile 3). Die Berechnungen innerhalb der Schleife setzen sich aus den folgenden Schritten zusammen. Zunächst berechnet man die Differenz der Grade von u und v. Dies entspricht dem ersten Schritt einer Polynomdivision. Falls man einen negativen Grad erhält, so vertauscht man die beiden Invarianten, was der Kommutativität des gcd entspricht. Die beiden Zuweisungen in den Zeilen 6 und 7 entsprechen der Addition von Gleichungen (I* = I* + II*). Die Zuweisung in Zeile 6 sorgt im Speziellen für eine Verringerung des Grades von u um mindestens eins.

Algorithmus 3.2.10 Inversenberechnung in $\mathbb{F}_{2^n}$

INPUT: Binärvektor des Polynoms $a(x) \neq 0$ vom Grad $\leq n-1$.
OUTPUT: $a^{-1} \mod f$

```
1: u ← a, v ← f                                    ▷ Initialisierungsteil
2: g1 ← 1, g2 ← 0
3: while u ≠ 1 do
4:     j ← deg(u) − deg(v)
5:     if j < 0 then u ↔ v, g1 ↔ g2, j ← −j         ▷ Vertauschen der Zeilen
6:     u ← u + x^j v          ▷ Addition der Invarianten; Verringern des Grads
7:     g1 ← g1 + x^j g2            ▷ Addition der Invarianten; Koeffizienten
8: end while
9: return (g1)                                      ▷ g1 = a^{-1}
```

Das nun folgende Beispiel soll die Funktionsweise von Algorithmus 3.2.10 noch einmal verdeutlichen. Dabei gilt es zu beachten, dass man, um die Korrektheit der Gleichungen einzusehen, jeweils beide Seiten modulo f reduzieren muss.

Beispiel 3.2.6. *Sei $f(x) = x^7 + x + 1$ irreduzibel über $\mathbb{F}_2$, $a(x) = x^2 + 1$ und $a, f \in \mathbb{F}_2[x]$. Gesucht: a^{-1} mit $a^{-1}a \equiv 1 \mod f(x)$. Zum besseren Verständnis sind die beiden Invarianten mit I und II bezeichnet.*
Die Initialisierung (Zeilen 1 und 2) $u = a$, $v = f$, $g_1 = 1$ und $g_2 = 0$ führt zunächst zu den beiden Gleichungen

$$\begin{aligned} &(I) && \overbrace{x^2+1}^{a}+f &&= \overbrace{x^2+1}^{a=u} \\ &(II) && f &&= f. \end{aligned}$$

Da $\deg(u) < \deg(v)$ werden die beiden Gleichungen I und II vertauscht (Zeile 5) und man erhält

$$\begin{aligned} &(I) && f &&= f \\ &(II) && a+f &&= a, \end{aligned}$$

wobei nun $u = f$, $v = a$, $g_1 = 0$ und $g_2 = 1$ gilt. Für die Differenz $j = \deg(u) - \deg(v) = 5$ addiert man nun auf die erste Gleichung ein Vielfaches (x^j) der zweiten (Zeilen 6 und 7) und es ergibt sich

$$\begin{aligned} &(I) && ax^5+f &&= \overbrace{x^7+x+1}^{f}+\overbrace{(x^2+1)}^{a}x^5 = x^5+x+1 \\ &(II) && a+f &&= a, \end{aligned}$$

mit $u = ax^5 + f$, $v = a$, $g_1 = x^5$ *und* $g_2 = 1$.
Im nächsten Durchlauf berechnet sich $j = \deg(u) - \deg(v) = 3$ *und man verwendet den Faktor* x^3 *bei der Addition der beiden Gleichungen.*

$$\begin{array}{lll} \text{(I)} & a(x^5+x^3)+f & = \overbrace{x^5+x+1}^{u} + ax^3 = x^3+x+1 \\ \text{(II)} & a+f & = a \end{array}$$

Demzufolge ergibt sich für $u = a(x^5 + x^3) + f$, $v = a$, $g_1 = x^5 + x^3$ *und* $g_2 = 1$.
Der folgende Durchlauf ergibt $j = \deg(u) - \deg(v) = 1$ *woraus sich ein Faktor* x *ableitet, der folgendes Ergebnis für die Addition der beiden Gleichungen liefert*

$$\text{(I)} \qquad a(x^5+x^3+x)+f = x^3+x+1+\overbrace{(x^2+1)}^{a}x = 1.$$

Für die Variablen gilt $u = 1$, $v = a$, $g_1 = x^5 + x^3 + x$ *und* $g_2 = 1$.
Jetzt bricht die Schleife ab, da die Bedingung $u \neq 1$ *verletzt ist.*
Bei genauerem betrachten der entstandenen Gleichung sieht man bereits das Ergebnis für a^{-1}, *denn* $a \cdot (x^5 + x^3 + x) \equiv 1 \bmod f$ *mit* $x^5 + x^3 + x = a^{-1}$. *Dies lässt sich einfach verifizieren:*

$$(x^2+1)(x^5+x^3+x) \equiv x^7 + x \equiv 1 \bmod x^7 + x + 1$$

Almost Inverse Algorithmus Der *Almost Inverse Algorithmus* ist eine Methode, die das inverse Element nicht direkt berechnet, sondern zunächst ein Vielfaches davon. In einem weiteren Schritt muss dann noch ein entsprechender Faktor abdividiert werden um das richtige Ergebnis zu erhalten. In [SOO95] wird dieser Algorithmus als Alternative zum Erweiterten Euklidischen Algorithmus vorgestellt.
Bevor der Algorithmus im Speziellen erläutert wird, sollen zunächst zwei Konstrukte eingeführt werden, die für das Verständnis der Funktionsweise von Vorteil sind.
Sei $a(x) \in \mathbb{F}_{2^n}$ ein gegebenes Polynom und a^{-1} das gesuchte Ergebnis. Der Almost Inverse Algorithmus berechnet nun zunächst ein Polynom $g(x)$ und eine positive Ganzahl k, so dass gilt:

$$a(x) \cdot g(x) \equiv x^k \bmod f(x), \quad \deg(g) < n, k < 2n$$

Für die Berechnung dieser Kongruenz werden zwei Invarianten benötigt, die man folgendermaßen schreibt:

$$\begin{array}{ll} \text{(I)} & ag_1 + fh_1 = x^k u \\ \text{(II)} & ag_2 + fh_2 = x^k v \end{array}$$

Die Faktoren h_1 und h_2 werden in diesem Algorithmus nicht explizit berechnet und dienen nur dem Verständnis.
In [HMV04] findet sich eine Variante des Almost Inverse Algorithmus, die pro Durchlauf beide Invarianten berechnet, während im Orginal von Schroeppel et al. [SOO95]

immer nur eine explizit berechnet und dann mit Vertauschungen gearbeitet wird. Für das Verständnis der Funktionsweise ist erstere Variante besser geeignet, weshalb diese im folgenden (Algorithmus 3.2.11) näher erläutert wird.

Algorithmus 3.2.11 Almost Inverse Algorithmus

INPUT: Binärvektor des Polynoms $a(x) \neq 0$ vom Grad $\leq n-1$.
OUTPUT: $a^{-1} \bmod f$

```
 1: u ← a, v ← f                                          ▷ Initialisierungsteil
 2: g1 ← 1, g2 ← 0, k ← 0
 3: while u ≠ 1 und v ≠ 1 do
 4:     while u teilbar durch x do
 5:         u ← u · x^{-1}, g2 ← g2 · x, k ← k + 1   (l mal)
 6:     end while
 7:     while v teilbar durch x do
 8:         v ← v · x^{-1}, g1 ← g1 · x
 9:         k ← k + 1   (r mal)          ▷ Annahme zum Verständnis: r Durchläufe
10:     end while
11:     if deg(u) > deg(v) then
12:         u ← u + v, g1 ← g1 + g2                       ▷ Rechnet I=I+II
13:     else
14:         v ← v + u, g2 ← g2 + g1                       ▷ Rechnet II=I+II
15:     end if
16: end while
17: if u = 1 then g ← g1 else g ← g2
18: return (g x^{-k} mod f)                  ▷ Reduziertes Ergebnis zurückgeben
```

Die in den Zeilen 4 und 8 verwendeten Bedingungen der Teilbarkeit, entsprechen einer Überprüfung des Koeffizenten $u_0 = 0$ bzw. $v_0 = 0$. Ist diese Bedingung erfüllt, so lässt sich x abdividieren was in der Implementierung einem Rechtsshift des Binärvektors entspricht.

$$u_{n-1}x^{n-1} + \ldots + u_2x^2 + u_1x = \left(u_{n-1}x^{n-2} + \ldots + u_2x^1 + u_1\right)x$$

Beide *while*-Schleifen dividieren u, bzw. v so lange durch x bis dies nicht mehr ohne Rest möglich ist. Sei nun l die Anzahl der Divisionschritte für die erste Schleife und r die Anzahl für die zweite Schleife. Für die beiden Invarianten ergibt sich nach dem Durchlauf der beiden Schleifen folgendes:

Erste while-Schleife (Zeile 4):
$$\begin{aligned} &(\mathrm{I}) & a \cdot g_1 + fh_1 &= x^l \overbrace{x^{-l} \cdot u}^{u'} \\ &(\mathrm{II}) & a \cdot \underbrace{g_2 \cdot x^l}_{g_2'} + fh_2 &= x^l v \end{aligned}$$

Erste while-Schleife (Zeile 8):
$$\begin{aligned} &(\mathrm{I}) & a \cdot \overbrace{g_1 \cdot x^r}^{g_1'} + fh_1 &= x^{l+r} u' \\ &(\mathrm{II}) & a \cdot g_2' + fh_2 &= x^{l+r} \underbrace{x^{-r} \cdot v}_{v'} \end{aligned}$$

In diesem Zusammenhang bezeichnen die Variablen v', u', g_1' und g_2' die neuen Variablen nach dem Ablauf der beiden while-Schleifen.
Da die Koeffizienten h_1, h_2 nicht weiter benötigt werden (verschwinden bei Reduktion modulo f), entsprechen die Anweisungen in den Zeilen 12 und 14 der Addition der beiden Invarianten. Abhängig vom Grad von u und v wird dabei entweder $I = I + II$ oder $II = II + I$ berechnet. Betrachten wir nun den Fall $\deg(u) > \deg(v)$:

$$\begin{aligned} &(\mathrm{I} = \mathrm{I} + \mathrm{II}) \qquad a \cdot \overbrace{(g_1 + g_2)}^{g_1'} + fh_1' &&= x^{l+r} \overbrace{(u+v)}^{u'} \\ &(\mathrm{II}) \qquad a \cdot g_2 + fh_2 &&= x^{l+r} v \end{aligned}$$

Durch die Addition der beiden Invarianten, hat das neu entstandene u' nun wieder die Eigenschaft, dass es durch x teilbar ist. Dies ist einfach einzusehen, da sowohl für den Koeffizient $v_0 = 1$ als auch für $u_0 = 1$ gegolten hat. Das bitweise XOR setzt nun wenigstens den neuen Koeffizienten $u_0' = 0$.
Solange keines der beiden Polynome u, v das Nullpolynom ist, dividiert man in den beiden inneren while-Schleifen wieder x ab. Es lässt sich erkennen, dass dieser Algorithmus terminieren muss, da die Grade der beiden Polynome u, v sukzessive kleiner werden, bis schließlich eines der beiden dem konstanten Polynom entspricht.
In diesem Fall lässt sich dann aus dem entsprechenden Faktor $g_{1/2}$ und k das Inverse a^{-1} berechnen.
Sei $u = 1$ und damit $ag + fh = x^k$. Rechnet man modulo f gilt:

$$ag \equiv x^k \bmod f \quad \overset{\cdot x^{-k}}{\Longrightarrow} \quad agx^{-k} \equiv 1 \bmod f \quad \overset{\cdot a^{-1}}{\Longrightarrow} \quad gx^{-k} \equiv a^{-1} \bmod f$$

Wie man sieht, entspricht gx^{-k} bis auf Reduktion (beinahe) dem Inversen a^{-1}. Diesen letzten Schritt der Reduktion kann man nun durchführen. Sei dazu $l = min\{i \geq 1 | f_i = 1\}$ der erste Koeffizient f_i des irreduziblen Polynoms der den Wert $f_i = 1$ hat. Die Einschränkung $i \geq 1$ rührt daher, dass bei irreduziblen Polynomen über $\mathbb{F}_2$ für den Koeffizienten f_0 immer gilt $f_0 = 1$. Zum besseren Verständnis schreiben wir $f(x)$ nun etwas anders:

$$f(x) = \overbrace{x^n + \ldots + x^l}^{t(x)} + 0x^{l-1} + \ldots + 0x^1 + 1 = t(x) + 1$$

Das Polynom s besteht aus den niederwertigsten l Bits von g und schreibt sich demzufolge

$$s(x) = g_{l-1}x^{l-1} + \ldots + g_1x + g_0.$$

Jetzt kann man das durch x^l teilbare Polynom $sf + g$ vom Grad kleiner n berechnen. Die Teilbarkeit lässt sich wie folgt zeigen.

$$\begin{aligned} sf + g &= \overbrace{g_{l-1}x^{l-1}f(x) + \ldots + g_0f(x)}^{s \cdot f} + \overbrace{g_mx^m + \ldots + g_lx^l}^{\tilde{g}(x)} + g_{l-1}x^{l-1} + \ldots + g_0 \\ &= g_{l-1}x^{l-1}t(x) + g_{l-1}x^{l-1} + \ldots + g_0t(x) + g_0 + \\ &\qquad \tilde{g}(x) + g_{l-1}x^{l-1} + \ldots g_1x + g_0 \\ &= g_{l-1}x^{l-1}t(x) + \ldots + g_0t(x) + \tilde{g}(x) \end{aligned}$$

Da die kleinste Potenz von x, die in $t(x)$ und $g_o(x)$ auftritt genau x^l ist, lässt sich diese ausklammern und eine Teilbarkeit durch x^l ist nachgewiesen.
Da nun für $T = (sf + g)/x^l$ gilt $\deg(T) < n$, kann man auch zur Restklasse modulo $f(x)$ übergehen und schreiben $T = g'x^{-l} \bmod f(x)$. Diese Schritte werden nun $\lfloor \frac{k}{l} \rfloor$ Mal wiederholt, ehe man im letzten Schritt noch durch x^r mit $r = k \bmod l$ dividieren muss. Nach Abschluss dieser gezeigten Schritte erhält man schließlich $gx^{-k} \equiv a^{-1} \bmod f(x)$.

Wie schon bei den Reduktionsalgorithmen, so kann man auch beim Almost Inverse Algorithmus durch die Wahl eines *geeigneten* irreduziblen Polynoms den Reduktionsteil erheblich beschleunigen, da man für $l > W$ beispielsweise wortweise reduzieren kann.

3.3 Vergleich der Implementierungen

Die im vorigen Kapitel beschriebenen Algorithmen für die Arithmetik in $\mathbb{F}_{2^n}$ wurden implementiert und auf verschiedenen Targetboards getestet. Die Laufzeiten der einzelnen Algorithmen wurden für verschiedene Grade n eines irreduziblen Polynoms $f(x)$ verglichen. Die dazugehörigen Polynome sind aus [Ser98] entnommen und in Tabelle 3.3 nochmals aufgeführt. Die beiden Polynome vom Grad $n = 163$ und $n = 233$ sind zusätzlich auch von der NIST vorgeschlagene Polynome. Für diese wurde ein spezieller Reduce-Algorithmus verwendet, wie er in Kapitel 3.2.9 beschrieben ist. Für den Almost Inverse Algorithmus wurde auch ein Vergleich zwischen zwei Polynomen mit nahezu gleichem Grad ($n = 333, 332$), die sich jedoch speziell durch ihre Gestalt unterscheiden, durchgeführt.
Alle Messungen wurden jeweils drei Mal wiederholt. Um ein möglichst aussagekräftiges Ergebnis zu erhalten wurden pro Messung jeweils d Durchläufe der zu messenden Routine ausgeführt. Aus der benötigten Ausführungszeit sowie der Anzahl der Durchläufe lässt sich nun die Laufzeit der einzelnen Algorithmen auf den verschiedenen Prozessoren bestimmen.

Grad $n =$	**irred. Polynom** $f(x) =$
15	$x^{15} + x + 1$
31	$x^{31} + x^3 + 1$
62	$x^{62} + x^{29} + 1$
163	$x^{163} + x^7 + x^6 + x^3 + 1$
233	$x^{233} + x^{74} + 1$
250	$x^{250} + x^{103} + 1$
332	$x^{332} + x^{89} + 1$
333	$x^{333} + x^2 + 1$
401	$x^{401} + x^{152} + 1$

Tabelle 3.3: *Die für die Timing-Vergleiche verwendeten irreduziblen Polynome*

3.3.1 Der C167

Der C167CR ist ein 16Bit Mikrocontroller von Infineon, der mit einer Taktrate von 25 MHz oder 33 MHz betrieben werden kann. Die Zeit für einen Befehlszyklus wird mit

80 bzw. 60 ns angegeben. Für die Multiplikation von zwei 16 Bit Werten ergibt sich demzufolge, laut Datenblatt, eine Zeit von 400 bzw. 303 ns, während die Division eines 32 Bit Wertes durch einen 16 Bit Wert genau doppelt so lange dauert. Der Prozessor verfügt über insgesamt 4 KByte RAM, sowie 128/32 KByte ROM. Weitere spezifische Daten findet man im Datenblatt des *C167CR LM* auf der Infineon Webseite [4].

Als Entwicklungsumgebung für den C167 wurde das Programm *TASKING CrossView Pro C166/ST10* in der Version 8.0r1 von der Firma Altium verwendet. Es handelt sich hierbei um eine vollständige Toolchain, die vom Cross C Compiler über Assembler und Linker bis zum hin zum Debugger alles abdeckt. Das Debbugging kann dabei wahlweise in C-Code oder Assembler-Anweisungen erfolgen. Mehr zu dieser Entwicklungsumgebung findet sich auf den dazugehörigen Internetseiten der Firma *Altium* [1].

3.3.2 Der ST30

Beim ST30 handelt es sich um einen 32 Bit Mikroprozessor, der, auf Basis des ARM7, von der Firma STMicrocontroller vertrieben wird. Die Erweiterungen des zugrunde liegenden ARM-Kerns sind vor allem eine erweiterte Peripherie sowie vielfältige I/O Möglichkeiten, wie beispielsweise zwei I^2C Bus Schnittstellen. Das für diese Arbeit verwendete Derivat mit der Bezeichnung ST30F772Z verfügt über 256 kByte Flash, sowie einen RAM-Bereich von 16 kByte. Die Taktfrequenz des Prozessors ist variabel zwischen 0-36 MHz und lässt sich außerdem über eine externe Taktreferenz einstellen. Mehr zum ST30 entnimmt man dem Datenblatt zur ST30 ZEPHYRUS Family auf der Webseite von STMicroelectronics [6].

Für den ST30 wurden zwei Tools verwendet. Der C Compiler stammt aus der *ARM Developer Suite v1.2* für den ARM7 Prozessor. Diese enthält auch weitere Tools wie Assembler und Linker. Weitere Informationen finden sich unter [2].
Für das Debugging auf auf dem Prozessor wurde die Software *TRACE32* sowie ein *LAUTERBACH Debug Interface* verwendet. Mit Hilfe der Software und des Interfaces ist es möglich, einzelne Anweisungen Schritt für Schritt auszuführen und diverse Prozessor- und Registerzustände auszulesen oder zu setzen. Weitere Informationen finden sich auf der Homepage der *Lauterbach Datentechnik GmbH* unter [5].

3.3.3 Multiplikationsalgorithmen

Die Messergebnisse für die verschiedenen Multiplikationsalgorithmen sind in Tabelle 3.4 zusammengetragen und in den Abbildungen 3.8 sowie 3.9 für die jeweiligen Prozessoren visualisiert. Für den *Left-to-Right Window* Algorithmus wurde eine Windowgröße von $w = 4$ verwendet.

Wie man sieht, sind sich die beiden Algorithmen ohne Lookup-Tabelle in ihrem Timingverhalten sehr ähnlich, was sich, wie in Kapitel 3.2.2 betrachtet, an ihrem vergleichbaren Aufbau festmachen lässt. Dennoch stellt man für den *Right-to-Left* Algorithmus geringe Performancevorteile fest. Diese lassen sich damit erklären, dass innerhalb des Algorithmus das Polynom $b(x)$ geschoben werden muss, während es beim

CPU	Algorithmus	Grad von $f(x)$								
		401	333	332	250	233	163	62	31	15
C167	Right-to-Left	27,6	20,8	20,6	12,3	10,9	7,3	2,0	1,0	0,53
	Left-to-Right	27,9	21,3	21,3	12,9	11,4	7,9	2,2	1,1	0,62
	Left-to-Right Window	16,7	13,5	13,4	8,1	7,3	5,5	1,8	1,0	0,56
ST30	Right-to-Left	3,1	2,3	2,3	1,5	1,5	0,91	0,30	0,25	0,16
	Left-to-Right	3,1	2,4	2,4	1,6	1,5	0,99	0,33	0,28	0,19
	Left-to-Right Window	1,9	1,5	1,5	0,92	0,84	0,61	0,21	0,18	0,13

Tabelle 3.4: *Timing-Vergleich der Multiplikations-Algorithmen für verschiedene Polynomlängen. (Zeiten in ms)*

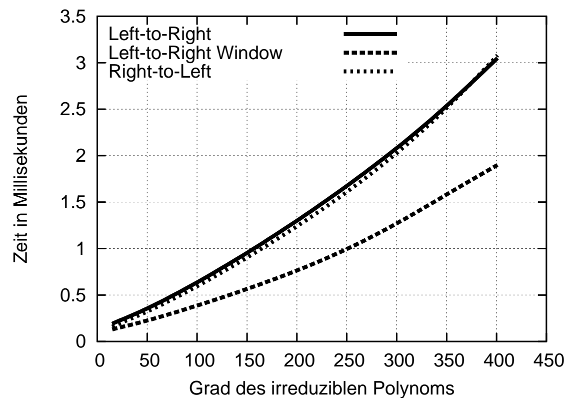

Abbildung 3.8: *Vergleich der verschiedenen Multiplikationsalgorithmen auf einem ST30 Mikroprozessor*

Left-to-Right Algorithmus das doppelt so lange Ergebnispolynom ist.
Der Geschwindigkeitsvorteil des Multiplikationsalgorithmus mit Lookup-Table macht sich vor allem bei Polynomen mit höherem Grad bemerkbar. Bei niedrigeren Graden ist dieser Vorteil nicht mehr so gravierend, da die Vorberechnung der Lookup-Tabelle jetzt einen größeren Anteil an der Gesamtlaufzeit besitzt. Das heißt, der Laufzeitvorteil der durch die Benutzung der Tabelle entsteht, wird durch deren Berechnung aufgehoben. Für eine Windowgröße von $w = 8$ verschiebt sich diese Grenze deutlich nach oben, was eine Benutzung auf den genannten Prozessoren, für die aktuell in der Kryptographie verwendeten Polynomgrößen, uninteressant macht. Ebenso steigt der RAM Bedarf für die Lookup-Table exponentiell, wie in Beispiel 3.2.2 bereits für den Fall $W = 16$ gezeigt ist.

3.3.4 Quadrieren von Polynomen

Für das Quadrieren von Polynomen wurden die beiden, in Kapitel 3.2.3 vorgestellten Algorithmen 3.2.6 und 3.2.7 implementiert. Ein dritter implementierter Algorithmus entspricht nahezu Alg. 3.2.6 nur mit dem Unterschied, dass die verwendete Lookup-Tabelle nicht bei jedem Funktionsaufruf berechnet werden muss, sondern bereits fest im ROM abgelegt ist. Die Messergebnisse für die verwendeten Prozessoren sind in den Abbildungen 3.10 und 3.11 dargestellt.
Im Fall des C167 (Abb. 3.10) stellt man zwischen den beiden Algorithmen, die sich die

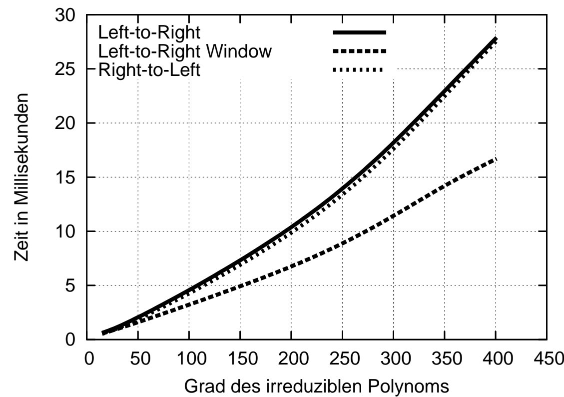

Abbildung 3.9: *Vergleich der verschiedenen Multiplikationsalgorithmen auf einem C167 Mikroprozessor*

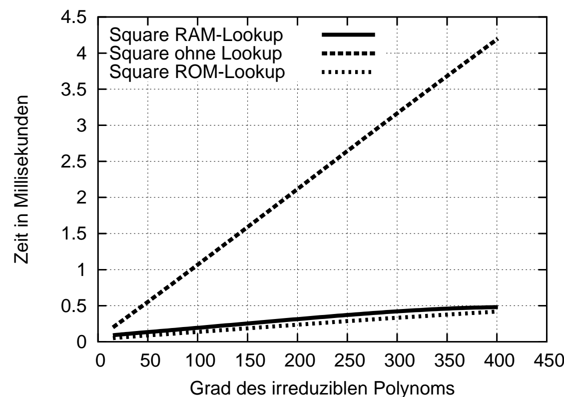

Abbildung 3.10: *Vergleich von drei Quadrieralgorithmen auf einem C167 Mikroprozessor*

Lookup-Tabelle zunutze machen, keine markanten Unterschiede fest. Der Zugriff auf eine im ROM liegende Tabelle bringt in diesem Fall nur minimale Vorteile gegenüber der RAM Variante. Dies lässt sich durch die Größe der Tabelle erklären. In ihr stehen die vorberechneten Werte für alle 4 Bit Muster. Sie besteht aus 16 Byte, die entweder im ROM abgelegt sind, oder bei jedem Funktionsaufruf neu berechnet werden und dann im RAM liegen. Die Berechnung dieser 16 Werte wird so performant durchgeführt, dass sich keine markanten Laufzeitnachteile im Vergleich zur ROM-Variante ergeben. Verzichtet man auf Vorberechnungen, so vergrößert sich die Laufzeit um einen Faktor ≈ 8.

In Fall des ST30 wird eine Lookup-Tabelle für alle 8 Bit Muster verwendet, wodurch man auf eine Größe von $2^8 \cdot 2 = 512$ Byte kommt. Nun kann man deutliche Laufzeitunterschiede zwischen den beiden Algorithmen mit Lookup-Tabelle feststellen. Wie aus Abbildung 3.11 zu erkennen, ist die Verwendung der ROM Methode um den Faktor ≈ 5 mal schneller, als eine Vorberechnung mit anschließender Ablage im RAM. Verwendet man Algorithmus 3.2.7 so ist dieser, für kleine Grade des irreduziblen Polynoms, zunächst schneller als Algorithmus 3.2.6. Erst für Grade $n > 230$ lohnt die Verwendung RAM Variante.

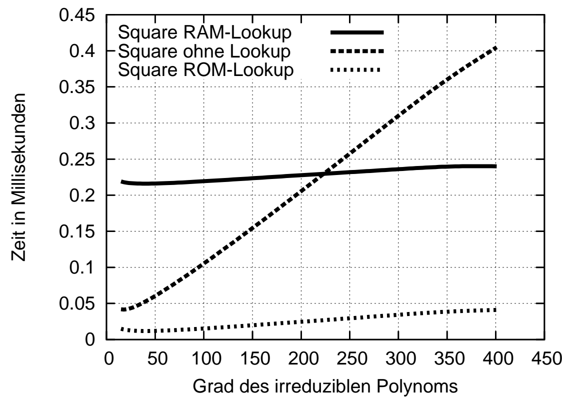

Abbildung 3.11: *Vergleich von drei Quadrieralgorithmen auf einem ST30 Mikroprozessor*

Zusammenfassend kann man sagen, dass die Nutzung von vorberechneten Werten in Lookup-Tabellen sehr große Auswirkungen auf die Laufzeit der betrachteten Algorithmen hat. Insbesondere durch eine dauerhafte Ablage im ROM kann das Quadrieren in allen Fällen beschleunigt werden. Zu Bedenken ist jedoch der für das embedded Umfeld große ROM Bedarf von 512 Byte, für eine Tabelle mit 256 Werten. Lookup-Tabellen mit einer geringeren Anzahl an Einträgen führen, wie in Abbildung 3.10 dargestellt, zu keinen nennenswerten Laufzeitvorteilen.

In den beiden Abbildungen (3.11 und 3.10) ist die Linearität des zugrunde liegenden Quadrieralgorithmus sehr gut zu erkennen.

3.3.5 Reduktionsalgorithmen

Eine Aufstellung der Laufzeiten der in Kapitel 3.2.4 beschriebenen Algorithmen befindet sich in Tabelle 3.5. Die dazugehörigen Abbildungen 3.12 und 3.13 verdeutlichen die gemessenen Werte.
Der in Zeile 3 aufgeführte Algorithmus *Reduce für NIST* ist eine Adaption von Algorithmus 3.2.9 an die NIST Polynome vom Grad 163 bzw. 233 und kann deshalb nur für diese Messungen verwendet werden. Man beobachtet deutliche Laufzeitvorteile für

		Grad von $f(x)$								
CPU	Algorithmus	401	333	332	250	233	163	62	31	15
C167	Reduce	40,0	29,2	30,7	18,4	16,0	12,0	3,0	1,7	0,99
	Reduce ohne Lookup	89,0	61,1	62,7	38,6	32,2	20,2	3,7	1,5	0,42
	Reduce für NIST					0,40	0,31			
ST30	Reduce	4,3	3,2	3,2	2,1	2,0	1,4	0,63	0,57	0,48
	Reduce ohne Lookup	9,6	7,2	7,2	4,2	4,0	2,3	0,54	0,25	0,07
	Reduce für NIST					0,031	0,009			

Tabelle 3.5: *Timing-Vergleich der Reduce-Algorithmen für verschiedene Polynomlängen. (Zeiten in ms)*

den *Reduce*-Algorithmus im Bezug auf den *Reduce ohne Lookup*-Algorithmus, die jedoch für kleiner werdende Gradzahlen immer weiter zurückgehen. Genau wie bei den

Multiplikationsalgorithmen lässt sich dies mit der Zeit zur Vorberechnung der Lookup-Tabelle erklären.

Die beiden, speziell für die NIST Polynome, optimierten Reducealgorithmem weisen

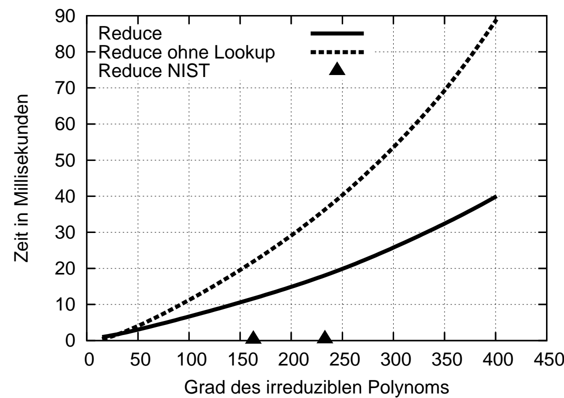

Abbildung 3.12: *Vergleich der Algorithmen zur Reduzierung modulo dem irreduziblen Polynom auf einem C167 Mikroprozessor*

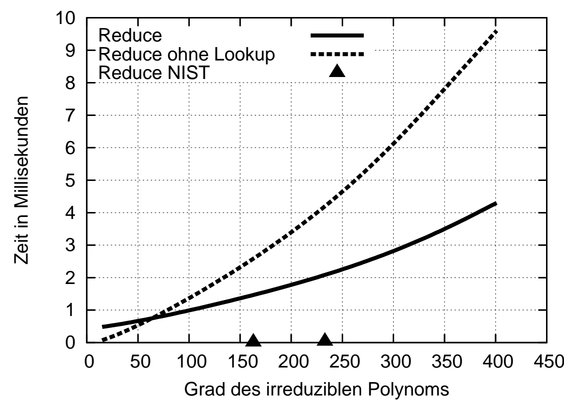

Abbildung 3.13: *Vergleich der Algorithmen zur Reduzierung modulo dem irreduziblen Polynom auf einem ST30 Mikroprozessor*

im Vergleich zur Reducevariante mit Lookup-Tabelle für den C167 eine Verbesserung um den Faktor ≈ 38 auf. Für den ST30 findet sich ein deutlich größerer Faktor, der für die beiden *Reduce NIST* Algorithmen nochmals variiert. Für den das Polynom vom Grad 233 wurde ein Faktor ≈ 65 gemessen, der sich für das Polynom vom Grad 163 mit nahezu 155 sogar mehr als verdoppelt. Dies bestätigt die theoretischen Betrachtungen in Kapitel 3.2.4.

Eine Verwendung dieser *angepassen* Methode empfiehlt sich vor allem dann, wenn sich das irreduzible Polynom nicht mehr ändert. Dies ist bei Kryptosystemen mit Elliptischen Kurven der Fall, weswegen hier eine Implementierung dieser speziellen Reduce Variante lohnt.

Für eine allgemeinere Verwendung, beispielsweise in einer Bibliothek für das Rechnen mit Polynomen in endlichen Körpern $\mathbb{F}_{2^n}$, müsste dieser Algorithmus für alle in Frage kommenden Polynome implementiert werden, was einen sehr großen Aufwand

bedeutet.

3.3.6 Inversenbildung

Die zwei in Kapitel 3.2.5 betrachteten Algorithmen zur Inversenberechnung sind in Tabelle 3.6 bezüglich ihres Laufzeitverhaltens miteinander verglichen. Die Abbildungen 3.14 und 3.15 verdeutlichen die gemessenen Werte nochmals graphisch. Die Laufzeit

CPU	Algorithmus	Grad von $f(x)$								
		401	333	332	250	233	163	62	31	15
C167	Erweiterter Euklid	365	261	261	152	127	85	15,8	5,9	1,8
	Almost Inverse	275	1021	217	124	112	237	16,9	14,2	7,7
ST30	Erweiterter Euklid	40	30	30	19	17	10	2,5	1,3	0,6
	Almost Inverse	33	191	27	16	16	45	2,8	4,2	2,5

Tabelle 3.6: *Timing-Vergleich der Algorithmen zur Inversenberechnung für verschiedene Polynomlängen. (Zeiten in ms)*

des Erweiterten Euklidischen Algorithmus hängt im wesentlichen vom Grad des verwendeten irreduziblen Polynoms ab.

Wie bereits in Kapitel 3.2.5 betrachtet, hängt die Laufzeit des *Almost Inverse* Algorithmus neben dem Grad des irreduziblen Polynoms vor allem von dessen Gestalt ab. Ist die Anzahl der 0-Koeffizienten zwischen den niederwertigsten beiden 1-Koeffizienten sehr gering (z.B. bei $x^{163} + x^7 + x^6 + x^3 + 1, x^{333} + x^2 + 1$), so muss man auf Grund der Funktionsweise des Algorithmus deutliche Performanceeinbußen hinnehmen. Teilweise ist der *Almost Inverse* Algorithmus dann um den Faktor ≈ 5 mal langsamer als der *Erweiterte Euklidische* Algorithmus. Wählt man ein spezielles, für den *Almost Inverse* Algorithmus passendes Polynom, so erfolgt die Berechnung teilweise bis zu 17% schneller als mit dem *Erweiterten Euklidischen* Algorithmus. Beispiele für solche speziellen Polynome $f(x)$ sind die vom Grad $401, 332, 250$ und 233 aus Tabelle 3.6. Die Tatsache, dass der *Almost Inverse* Algorithmus nur für ausgewählte Polynome

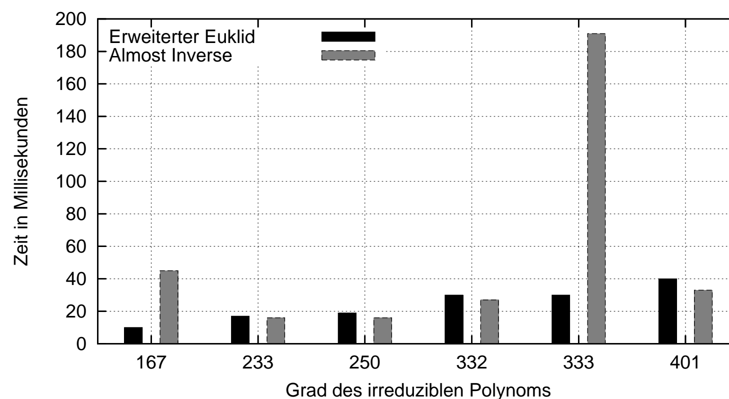

Abbildung 3.14: *Vergleich der beiden Algorithmen zur Inversenbildung auf einem ST30 Mikroprozessor*

eine bessere Performance als der *erweiterte Euklidische* Algorithmus besitzt, macht

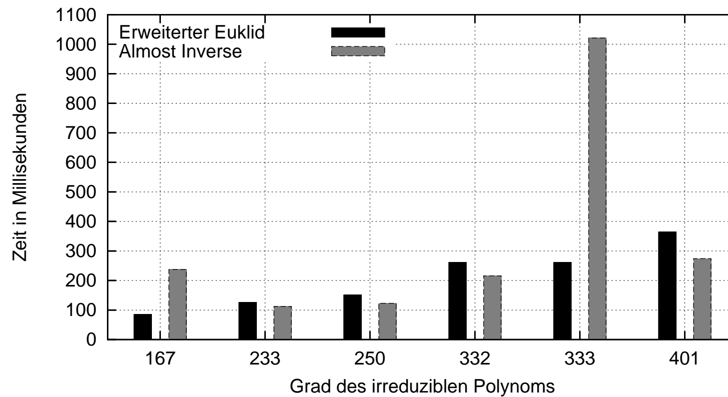

Abbildung 3.15: *Vergleich der beiden Algorithmen zur Inversenbildung auf einem C167 Mikroprozessor*

eine allgemeine Verwendung des *Almost Inverse* Algorithmus nicht empfehlenswert. Für spezielle Polynome ist ein Einsatz jedoch durchaus denkbar.
Für den Speicherbedarf der beiden Algorithmen ergibt sich folgendes Bild. Der Almost Inverse Algorithmus benötigt, in der implementierten Variante, $9 \cdot Len$ Speicherwörter. Der Erweiterte Euklidische Algorithmus ist, was den Speicherbedarf angeht, deutlich sparsamer und verwendet nur $5 \cdot Len$ Speicherwörter. Der Ausdruck Len steht dabei für die Anzahl an Speicherworten, die für die Darstellung des irreduziblen Polynoms $f(x) \in \mathbb{F}_2[x]$ benötigt werden. Für den Fall $W = 32$ und $\deg(f) = m = 233$ ergibt sich für den Almost Inverse Algorithmus ein Speicherbedarf von $9 \cdot \lceil \frac{233}{32} \rceil \cdot 4 = 288$ Bytes. Für den Erweiterten Euklidischen Algorithmus werden nur $5 \cdot 8 \cdot 4 = 160$ Bytes benötigt.

Kapitel 4

Einführung in Elliptische Kurven

Kryptosysteme auf Basis Elliptischer Kurven nutzen die Gruppeneigenschaft der Punkte auf diesen Kurven. Die zur Gruppeneigenschaft gehörende Gruppenoperation wird in diesem Abschnitt definiert und die entsprechenden Formeln für die Punktaddition sollen hergeleitet werden.
Um sich jedoch überhaupt mit Elliptischen Kurven beschäftigen zu können, sind zunächst noch einige weitere mathematische Grundlagen nötig. Diese sind für das allgemeine Verständnis der Elliptischen Kurven sehr wichtig, weshalb sie auch in diesem Kapitel behandelt werden sollen.

Da es sich bei Elliptischen Kurven um einen speziellen Typ von algebraischen Kurven handelt, werden diese vorzugsweise in der Algebraischen Geometrie behandelt. Über diesen Teilaspekt der Mathematik und ebene algebraische Kurven soll auch in diesem Kapitel der Zugang zu den Elliptischen Kurven erfolgen. Dazu werden zunächst einige Grundlagen der Algebraischen Geometrie erläutert, ehe dann Elliptische Kurven im allgemeinen und im speziellen über endlichen Körpern der Charakteristik 2 ($\mathbb{F}_{2^n}$) betrachtet werden.

4.1 Grundlagen der Algebraischen Geometrie

Bewegt sich ein Gegenstand im Laufe der Zeit durch den Raum, so beschreibt er dabei eine Kurve. Solche Vorgänge werden in der Kurventheorie näher untersucht und betrachtet. Für die in dieser Arbeit vorkommenden Betrachtungen reicht es jedoch völlig aus, sich auf *ebene algebraische Kurven* zu konzentrieren.
In diesem Zusammenhang bedeutet die erste Einschränkung *eben*, dass es sich bei dem Raum, in dem die Bewegung stattfindet, um einen Raum der Dimension zwei handelt. Das zweite Merkmal *algebraisch* sagt aus, dass die der Kurve zugrunde liegende Funktion ein Polynom ist. Andernfalls spricht man auch von einer *transzendenten Kurve*. Diese beiden Einschränkungen vereinfachen viele Betrachtungen. Dennoch sind viele der folgenden Definitionen allgemein formuliert, da sie innerhalb der algebraischen Geometrie eine allgemeinere Bedeutung haben.

4.1.1 Ausgewählte Definitionen der Algebraischen Geometrie in der Ebene

Für die nun folgenden allgemeineren Aussagen sei K stehts ein algebraisch abgeschlossener Körper.

Definition 4.1.1. *Sei K ein algebraisch abgeschlossener Körper und $f_1, \dots, f_s \in K[x_1, \dots, x_n]$. Dann heißt*

$$V(f_1, \dots, f_s) = \{(x_1, \dots, x_n) \in K^n | f_i(x_1, \dots, x_n) = 0 \text{ für } 1 \leq i \leq s\}$$

die affine algebraische Varietät, *die durch $f_1, \dots, f_s$ definiert ist.*

Bemerkung 2. *Betrachtet man die Varietät $V(f_1, \dots, f_s)$ nicht über dem algebraischen Abschluss, sondern über einem Körper $k \subseteq K$ so schreibt man dafür auch $V_k(f_1, \dots, f_s)$. Für solche Teilkörper k gilt weiterhin $V_k(f_1, \dots, f_s) = V(f_1, \dots, f_s) \cap k^n$.*

Definition 4.1.2. *Ist in Definition 4.1.1 $s = 1$, $n = 2$ und $f_1 = f \in K[x_1, x_2]$ nicht konstant, so heißt*

$$C = V(f) = \{(x_1, x_2) \in K^2 | f(x_1, x_2) = 0\}$$

ebene algebraische Kurve *über K^2.*

Ein solches Polynom f ist durch die gegebene Varietät $C = V(f)$ nicht eindeutig bestimmt, denn für $\lambda \in K^*$ und $i \in \mathbb{N} \setminus \{0\}$ gilt $V(f) = V(\lambda f) = V(f^i)$.

Beispiel 4.1.3. *Sei im folgenden $n = 2$. Zunächst zwei einfache Beispiele für konstante Funktionen $f_1 = 0, f_2 = 1$. Damit ergibt sich für die Varietät $V(f_1) = V(0) = K^2$, $V(f_2) = V(1) = \emptyset$*
Ein weiteres Beispiel ergibt sich für die Funktion $f_3(x, y) = x^2 + y^2 + 1$. Vergleicht man die Varietäten $V_{\mathbb{R}}(f_3)$ und $V(f_3) = V_{\mathbb{C}}(f_3)$, so ergibt sich $V_{\mathbb{R}}(f_3) = \emptyset \neq V(f_3)$. Dies ist leicht einzusehen, da f_3 keine reellen Lösungen hat, während $(0, i) \in \mathbb{C}$ eine Lösung im Komplexen darstellt. Zudem gilt $\mathbb{R} \subset \mathbb{C} = K$.

Lemma 4.1.4. *(Study) Seien $f, g \in \mathbb{C}[x, y]$ und f irreduzibel und nicht konstant. Ist $V_{\mathbb{C}}(f) \subseteq V_{\mathbb{C}}(g)$, dann ist f Teiler von g, d.h. $f|g$.*

Beweis. Zum Beweis des *Studyschen Lemmas* werden klassische geometrische Methoden eingesetzt. Eine Variante findet sich in [Fis94, Seite 13f]. □

Korollar 4.1.5. *Ist $f \in \mathbb{C}[x, y]$ eine Nicht-Einheit, so ist $V(f) \neq \emptyset$.*

Beweis. Angenommen $V(f) = \emptyset$; ist $h \in \mathbb{C}[x, y]$ ein nicht konstanter irreduzibler Faktor von f, so ist $V(h) = \emptyset$.
Nach Lemma 4.1.4 teilt dann h jedes $g \in \mathbb{C}[x, y]$, ein Widerspruch. □

Da Polynomringe über Körpern faktoriell sind, lässt sich $f \in \mathbb{C}[x, y]$ in einer, bis auf Einheiten und Reihenfolge eindeutigen Zerlegung $f = f_1^{k_1}, \dots, f_r^{k_r}$ darstellen. Die Polynome $f_i \in \mathbb{C}[x, y]$ sind irreduzibel und paarweise nicht assoziiert.
Damit lässt sich die durch f definierte Kurve in Komponenten $V(f_i)$ zerlegen mit $V(f) = V(f_1) \cup \dots \cup V(f_r)$. Dies lässt sich durch die folgende Definition präzisieren.

Definition 4.1.6. *Eine ebene algebraische Kurve $C \subset \mathbb{C}^n$ heißt* reduzibel *, wenn es algebraische Kurven C_1, C_2 gibt mit $C_1 \neq C_2$ und $C = C_1 \cup C_2$ (nach Korollar 4.1.5 ist $C_i \neq \emptyset$).*
Irreduzibel *heißt nicht reduzibel, d.h. für jede Zerlegung $C = C_1 \cup C_2$ folgt $C_1 = C_2$.*

Lemma 4.1.7. *Eine algebraische Kurve $C = V(f) \subset \mathbb{C}^2$ ist genau dann irreduzibel, wenn es ein irreduzibles $g \in \mathbb{C}[x, y]$ und $k \in \mathbb{N} \setminus \{0\}$ gibt mit $f = g^k$.*

Beweis. Ein Beweis findet sich in [Fis94, Seite 15]. □

Jede algebraische Kurve $C \subset \mathbb{C}^2$ gestattet eine bis auf die Reihenfolge eindeutige Darstellung $C = C_1 \cup \cdots \cup C_r$ mit irreduziblen algebraischen Kurven $C_1, \ldots, C_r$. Diese nennt man *irreduzible Komponenten* .

Korollar 4.1.8. *Sei $C = V(f) \subset \mathbb{C}^2$ eine algebraische Kurve und $f = f_1^{k_1} \cdot \ldots \cdot f_r^{k_r}$ eine Zerlegung in irreduzible Faktoren. Ist g ein weiteres Polynom mit $C = V(g)$, so ist nach Lemma 4.1.4 $g = \lambda \cdot g_1^{l_1} \cdot \ldots \cdot g_r^{l_r}$ mit $\lambda \in \mathbb{C}^*$ und $l_i \in \mathbb{N}^*$.*

Daraus ergibt sich eine vollständige Übersicht über die möglichen Gleichungen von C. Analog zu Polynomen mit einer Variablen nennt man

$$\tilde{f} = f_1 \cdot \ldots \cdot f_r$$

das, bis auf eine Einheit eindeutig bestimmte, *Minimalpolynom* der Kurve C.

Definition 4.1.9. *Ist $C = V(f) \subset \mathbb{C}^2$ eine algebraische Kurve und f Minimalpolynom von C, so heißt* $\deg(C) = \deg(f)$ *der* Grad *von C.*

Kurven vom Grad 1 nennt man im allgemeinen *Geraden* und Kurven vom Grad 3 heißen *kubische Kurven* .

Beispiele ebener algebraischer Kurven Für die folgenden Beispiele gelte wenn nicht anders erwähnt $K^2 = \{(x, y) | x, y \in K\}$.

- *Kegelschnitte* Sei $K = \mathbb{R}$ mit $a, b, c, d, e, f \in \mathbb{R}$. Die zugehörige Gleichung lautet $ax^2 + bxy + cy^2 + dx + ey + f = 0$.
  Zu ihrer Familie gehören Ellipsen, Kreise, Parabeln und Hyperbeln.
- *Konchoiden* Sei $C \subseteq \mathbb{R}^2$ eine reelle Kurve, $P \in \mathbb{R}^2$ ein fester Punkt und $a > 0$ eine Konstante. Dann nennt man $\{Q \in \mathbb{R}^2 | \overline{PQ} \cap C$ ist ein Punkt mit Abstand a von $Q\}$ die Konchoide von C bezüglich P mit Parameter a.
- *kubische Kurven* (Kurven vom Grad 3)
  - Neilsche Parabel: $x^3 = y^2$
  - Newtonscher Knoten: $x^2(x + 1) = y^2$
  - Kartesisches Blatt: $x^3 + y^3 - 3xy = 0$
  - Kissoide von Diokles: $y^2(1 - y) = x^3$
  - Elliptische Kurven: $y^2 = x^3 + ax^2 + b, \quad 4a^3 + 27b^2 \neq 0$

4.1.2 Die Projektive Geometrie

Damit die Punkte einer Elliptischen Kurve eine Gruppe bilden, benötigt man ein neutrales Element. Dieses wird in der Regel als der *unendlich ferne Punkt* $\mathcal{O}$ bezeichnet. Betrachtet man elliptische Kurven im zweidimensionalen Raum $\mathbb{R}^2$, so stellt man sich diesen Punkt unendlich weit in positiver y-Richtung, sowie unendlich weit in negativer y-Richtung gelegen vor. Man könnte also etwas lax sagen, im unendlich fernen Punkt sind die beiden Äste der y-Achse hinter der xy-Ebene verklebt. Im affinen Raum $\mathbb{R}^2$ ist dieser Punkt nicht greifbar, woraus auch der Name *unendlich ferner* Punkt resultiert. Weiterhin stellt man sich vor, dass sich zwei parallele Geraden ebenfalls in genau einem Punkt schneiden. Solche Schnittpunkte paralleler Geraden sind ebenfalls Punkte im unendlich fernen.

Um diesen Sachverhalt nun auch mathematisch fassen zu können, erweitert man den affinen Raum K^n über einem beliebigen Körper K gerne zum projektiven Raum $\mathbb{P}^n(K)$ über diesem Körper. Die geometrische Vorstellung von höher dimensionalen projektiven Räumen ist im allgemeinen recht problematisch und schwierig.
Für die Betrachtung Elliptischer Kurven reicht es aus, sich auf den ebenen Fall der Dimension 2 ($\mathbb{P}^2(K)$) zu beschränken.
In diesem Fall ($n = 2$) bezeichnet man $\mathbb{P}^2(K)$ als die *projektive Ebene* über K, d.h. $\mathbb{P}^2(K)$ ist die Menge aller Geraden durch den Ursprung des K^3.
Im Fall $(0,0,0) \neq (x,y,z) = Q \in K^3$ bezeichnet $[x : y : z] = (x,y,z)K$ die Gerade durch den Punkt $Q \in \mathbb{P}^2(K)$. Die Koordinaten $[x : y : z]$ bezeichnet man als die *homogenen Koordinaten* von Q. Um mit diesen Koordinaten rechnen zu können, gilt die folgende Regel:

$$[x_1 : y_1 : z_1] \equiv [x_2 : y_2 : z_2] \quad \Leftrightarrow \quad \begin{aligned} x_1 &= \lambda x_2 \\ y_1 &= \lambda y_2 \quad \text{mit } \lambda \in K^* \\ z_1 &= \lambda z_2 \end{aligned}$$

Etwas anders formuliert lässt sich auch sagen, die Gerade durch die Punkte

$$(\lambda x, \lambda y, \lambda z), \lambda \in K^*$$

stellt eine Äquivalenzklasse dar, die durch den Repräsentanten $[x : y : z]$ bezeichnet wird.
Die affine Ebene K^2 lässt sich durch verschiedene Abbildungen in die projektive Ebene $\mathbb{P}^2(K)$ einbetten. Eine solche Abbildung ϕ, die zusätzlich noch kanonisch ist, wird für diese Arbeit gewählt mit:

$$\phi : K^2 \to \mathbb{P}^2(K), \quad (x,y) \mapsto (x,y,1)$$

Die Menge der unendlich fernen Punkte von K^2 ist in diesem Fall

$$\mathbb{P}^2(K) \setminus \phi(K^2) = \{(x,y,z) \in \mathbb{P}^2(K) : z = 0\}$$

eine *projektive Gerade* $\mathbb{P}^1(K)$, die man auch als die Gerade im unendlichen bezeichnet. Jeder einzelne Punkt $(x,y,0)$ auf dieser Geraden definiert eine Richtung (Parallelklasse) $[x : y]$ in K^2. Für den Fall zweier paralleler Geraden in K^2 bedeutet dies, dass sich deren Schnittpunkt als ein Punkt auf der projektiven Gerade beschreiben lässt.

Homogene Polynome

Um *homogene Polynome* einführen zu können, müssen zunächst noch einige Begriffe für Polynome $f(x_1, \ldots, x_n)$ mit mehreren Variablen definiert werden.

Definition 4.1.10. *Sei $f \in K[x_1, \ldots, x_n]$ gegeben durch*

$$f(x_1, \ldots, x_n) = \sum a_{i_1 \ldots i_n} x_1^{i_1} \cdots x_n^{i_n}.$$

Wenn $a_{i_1 \ldots i_n} \neq 0$, dann heißt $a_{i_1 \ldots i_n} x_1^{i_1} \cdots x_n^{i_n}$ Term *von f und $i_1 + \ldots + i_n$ bezeichnet den* Grad *des Terms.*
Sei weiterhin $f \neq 0$, dann bezeichnet $\deg(f)$ *das Maximum der Grade aller Terme von f und heißt* Grad von f.

Die für das Verständnis dieser Arbeit benötigten homogenen Polynome sind solche mit drei Variablen und werden demzufolge entsprechend definiert.

Definition 4.1.11. *Sei $f(x, y, z) \in K[x, y, z]$ ein Polynom. $f(x, y, z)$ heißt* homogenes Polynom *vom Grad m wenn gilt:*

$$f(x, y, z) = \sum_{i+j+k=m} a_{i,j,k} x^i y^j z^k,$$

d.h. Koeffizienten $a_{i,j,k} \neq 0$ treten nur dann auf, falls gilt $i + j + k = m$.

Beispiel 4.1.12. *Die folgenden Polynome $g(x, y, z), f(x, y, z)$ haben beide Grad 3, wobei f zusätzlich homogen ist.*

$$\begin{aligned} g(x, y, z) &= y^2 z + a_1 x + a_3 y^2 \\ f(x, y, z) &= y^2 z + a_1 xyz + a_3 yz^2 + x^3 + a_2 x^2 z + a_4 xz^2 + a_6 z^3 \end{aligned}$$

Drei wichtige Eigenschaften von homogenen Polynomen sind in der nun folgenden Bemerkung zusammengefasst.

Bemerkung 3. *Für homogene Polynome gilt:*

(I) Das Nullpolynom ist homogen von jedem Grad

(II) Jedes Polynom $f(x, y, z)$ lässt sich eindeutig als Summe von homogenen Polynomen darstellen mit:

$$f(x, y, z) = f_l(x, y, z) + \ldots + f_k(x, y, z),$$

Dabei ist $l \leq k$ und $f_i(x, y, z)$ homogen vom Grad i.
Man nennt f_l die Leitform *von f und f_k die* Gradform *von f.*

(III) Sei $f \in K[x, y, z]$ ein homogenes Polynom vom Grad m, dann gilt für alle $k \in K$:

$$f(kx, ky, kz) = k^m \cdot f(x, y, z),$$

was sich durch einfaches Nachrechnen leicht zeigen lässt.

Jedem Polynom $f(x,y)$ vom Grad d kann ein homogenes Polynom f_h, die *Homogenisierung* von f,

$$f_h(x,y,z) =_{def} z^d \cdot f\left(\frac{x}{z}, \frac{y}{z}\right)$$

zugeordnet werden. Ist $f = f(x,y,z)$ ein homogenes Polynom, dann bezeichnet $f_d =_{def} f(x,y,1)$ das f zugeordnete *dehomogenisierte* Polynom.

Beispiel 4.1.13. *Sei $f(x,y) = x^2 + 2xy + x + y + 3$ ein Polynom vom Grad zwei. Das entsprechende zugeordnete homogene Polynom $f_h(x,y,z) = x^2 + 2xy + xz + yz + 3z^2$ ist dann ebenfalls vom Grad zwei.*

Projektive Kurven

Mit Hilfe homogener Polynome lassen sich nun projektive Kurven definieren.

Definition 4.1.14. *Sei $C \subset \mathbb{P}^2(K)$, dann heißt C* projektive algebraische (ebene) Kurve*, wenn es ein nichtkonstantes homogenes Polynom $f(x,y,z) \in K[x,y,z]$ gibt mit*

$$C = \{[x:y:z] \in \mathbb{P}^2(K) | f(x,y,z) = 0\}.$$

Ist $f(x,y,z) \in K[x,y,z]$ ein nichtkonstantes homogenes Polynom, so bezeichnet

$$V(f) =_{def} C = \{[x:y:z] \in \mathbb{P}^2(K) | f(x,y,z) = 0\}.$$

die projektive Varietät *von f. Man bezeichnet dies manchmal auch als die* Nullstellenmenge *von f.*

Man bezeichnet einen mit Hilfe eines Koordinatensystems definierten Begriff als *koordinatenunabhängig*, wenn der Begriff invariant bei Koordinatentransformation ist.

Lemma 4.1.15. *Der Begriff* projektive ebene Kurve *ist koordinatenunabhängig.*

Beweis. Sei $(x,y,z) \mapsto (\tilde{x},\tilde{y},\tilde{z}) := (x,y,z) \cdot A$ eine Koordinatentransformation mit Transformationsmatrix A und sei weiter $f(x,y,z)$ eine Gleichung für die Kurve C. Dann ist $\tilde{f}(\tilde{x},\tilde{y},\tilde{z}) := f((\tilde{x},\tilde{y},\tilde{z}) \cdot A^{-1})$ eine Gleichung für die transformierte Kurve $\tilde{C}(\tilde{f}(\tilde{x},\tilde{y},\tilde{z})) = \tilde{C}(f((\tilde{x},\tilde{y},\tilde{z}) \cdot A^{-1})) = \tilde{C}(f((x,y,z) \cdot AA^{-1})) = \tilde{C}(f(x,y,z))$. □

Beispiel 4.1.16. *Sei $f(x,y,z) := y^2z + xyz = x^3 + ax^2z + bz^3 \in \mathbb{F}_2[x,y,z]$ ein homogenes Polynom über einem Körper der Charakteristik zwei. Die Transformationsmatrix A und die dazugehörige inverse Matrix A^{-1} mit*

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & z & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \textit{und} \quad A^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{z} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

transformieren das Polynom f mit $(x,y,z) \mapsto (\tilde{x},\tilde{y},\tilde{z}) := (x,y,z) \cdot A$ nach

$$\tilde{f}(\tilde{x},\tilde{y},\tilde{z}) := \tilde{y}^2 + \tilde{x}\tilde{y}\tilde{z} = \tilde{x}^3\tilde{z} + a\tilde{x}^2\tilde{z}^2 + b\tilde{z}^4$$

und mit $(\tilde{x}, \tilde{y}, \tilde{z}) \mapsto (x, y, z) := (\tilde{x}, \tilde{y}, \tilde{z}) \cdot A^{-1}$ *entsprechend wieder zurück.*
Das Polynom f *entspricht der projektiven Form der Weierstrasschen Gleichung einer Elliptischen Kurve in Charakteristik zwei.* $\tilde{f}$ *bezeichnet die projektive Gleichung für* LÓPEZ-DAHAB-*Koordinaten. Diese spielen für eine effiziente Implementierung eine wichtige Rolle, wie man in Kapitel 5 noch sehen wird.*

Bemerkung 4. *Drei wichtige Eigenschaften von projektiven algebraischen Kurven sind hier zusammengefasst:*

(I) Sei m *der Grad von* f*. Da* f *homogen ist gilt* $f(\lambda x, \lambda y, \lambda z) = \lambda^m \cdot f(x, y, z)$.
$\Rightarrow$ $f(x, y, z) = 0$ *ist daher unabhängig von der Wahl des Repräsentanten von* $[x : y : z]$.

(II) Ist f *eine Gleichung für die Kurve* C*, dann ist auch* λf, $\lambda \in K^*$ *und* f^k, $k \geq 1$*, eine Gleichung für* C*.*

(III) Der Grad einer Kurve C *ist koordinatenunabhängig, denn der er ist über das Minimalpolynom von* C *definiert.*

Ist $C = V(f) \subset K^2$ eine affine-algebraische Kurve und f_h die zugeordnete Homogenisierung von f, dann wird $\overline{C} = V(f_h) \subset \mathbb{P}^2(K)$ der *projektive Abschluss* von C genannt.

In der Mathematik es es üblich, dass man sich nicht nur für bestimmte Objekte interessiert, sondern auch Abbildungen zwischen solchen Objekten betrachtet. In dieser Arbeit sind diese Objekte ebene algebraische Kurven oder später, der Spezialfall, Elliptische Kurven. Besonders interessant sind *Isomorphismen* zwischen solchen Objekten, in unserem Fall zwischen den einzelnen Kurven. In der algebraischen Geometrie definiert man einen Isomorphismus zwischen Kurven mit Hilfe von *rationalen Abbildungen* und *Morphismen*. Für eine genauere Definition des Isomorphismus zwischen Elliptischen Kurven sei auf [Sil86, Kapitel II.2] sowie [Hus04, Kapitel 3] verwiesen.

4.1.3 Singularitäten und Schnittpunkte

Wie noch gezeigt wird, lassen sich die Punkte einer Elliptischen Kurve kanonisch mit einer Gruppenstruktur versehen. Für die Definition der Gruppenoperation ist es notwendig, das Schnittverhalten zwischen einer Elliptischen Kurve und einer Geraden näher zu betrachten.
In diesem Abschnitt soll dazu der Schnitt einer projektiven Gerade mit einer projektiven Kurve vom Grad d erläutert werden.
Diese Betrachtung entspricht einem Spezialfall des Satzes von BÉZOUT, der allgemein aussagt, dass sich zwei projektive Kurven C_1 und C_2 vom Grad d_1 und d_2 immer in $n = d_1 d_2$ Punkten, mit Vielfachheiten gezählt, schneiden (für den allgemeinen Fall siehe [Fis94, 2.7]).
Bevor dieser Spezialfall in Satz 4.1.20 näher erläutert wird, müssen noch verschiedene Begrifflichkeiten eingeführt werden.

Definition 4.1.17. *Sei* $g \in K[x, y, z]$ *ein homogenes Polynom vom Grad 1 in folgender Form*

$$g(x, y, z) = \alpha x + \beta y + \gamma z, \quad \alpha, \beta, \gamma \in K$$

wobei α, β und γ nicht gleichzeitig Null werden, so nennt man die Kurve $C_g = V(g)$ projektive Gerade. *Man schreibt auch häufig $L(\alpha, \beta, \gamma)$.*

Für projektive Geraden gilt nun das folgende Lemma

Lemma 4.1.18. *(I) Durch je zwei verschiedene Punkte des $\mathbb{P}^2(K)$ führt genau eine projektive Gerade $L(\alpha, \beta, \gamma)$*

(II) Zwei verschiedene projektive Geraden L_1, L_2 schneiden sich in genau einem Punkt $Q \in \mathbb{P}^2(K)$.

Beweis. (I) Seien $P_1 = [a_1 : b_1 : c_1]$ und $P_2 = [a_2 : b_2 : c_2]$ zwei verschiedene Punkte und $P_1, P_2 \in \mathbb{P}^2(K)$. Gesucht sind nun Koeffizienten $(\alpha, \beta, \gamma) \neq (0, 0, 0)$ so dass

$$\begin{aligned} \alpha a_1 + \beta b_1 + \gamma c_1 &= 0 \quad \text{und} \\ \alpha a_2 + \beta b_2 + \gamma c_2 &= 0 \end{aligned}$$

gilt. Das entspricht einem linearen Gleichungssystem mit der Koeffizientenmatrix

$$\begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \end{pmatrix}.$$

Laut Definition sind die beiden Punkte P_1 und P_2 verschieden, woraus folgt, dass die beiden Zeilen der Matrix linear unabhängig sind, die Matrix also den Rang zwei besitzt. Die Dimensionsformel für lineare Abbildungen sagt, dass der Lösungsraum im K^3 in diesem Fall eindimensional ist.
Anders ausgedrückt, es gibt ein Tripel $(\alpha, \beta, \gamma) \neq (0, 0, 0)$, so daß $P_1, P_2 \in L(\alpha, \beta, \gamma)$. Jedes weitere Tripel $(\alpha', \beta', \gamma')$ das diese Bedingung erfüllt ist ein Vielfaches von (α, β, γ), weswegen es nur genau eine projektive Gerade gibt, die P_1 und P_2 enthält.

(II) Ein Beweis findet sich in [Wer02, Seite 34].

□

Die Schnittmultiplizität oder Vielfachheit eines Schnittpunkts zwischen einer projektiven Geraden und einer projektiven Kurve ist folgendermaßen definiert.

Definition 4.1.19. *Sei $P = [a_1 : b_1 : c_1] \in \mathbb{P}^2(K)$ ein Punkt, $L = L(g(x, y, z))$ mit $g(x, y, z) = \alpha x + \beta y + \gamma z = 0$ eine projektive Gerade über K und $C = C(f(x, y, z))$ mit $f(x, y, z) = 0$ eine projektive Kurve über K. Vorausgesetzt das $g(x, y, z)$ kein Teiler von $f(x, y, z)$ ist, definiert man $i(P, L, C)$, die* Vielfachheit *(*Schnittmultiplizität*) des Schnittpunktes P von C und L wie folgt.*
Sei $P \notin V(f) \cap V(g)$, dann ist $i(P, L, C) = 0$. Andernfalls löst man die Gleichung $g(x, y, z)$ für L nach einer beliebigen Variablen auf, z.B. $z = -\frac{\alpha}{\gamma}x - \frac{\beta}{\gamma}y$ (für $\gamma \neq 0$) und setzt diesen Ausdruck in $f(x, y, z)$ ein. Anschaulich entspricht dies dem Schnitt der Gerade mit der Kurve. Man erhält somit ein homogenes Polynom $H(x, y) = f(x, y, -\frac{\alpha}{\gamma}x - \frac{\beta}{\gamma}y)$ in zwei Variablen, das durch $(a_1 y - b_1 x)$ teilbar ist. (Wurde x oder y eliminiert dann $(b_1 z - c_1 y)$ bzw. $(a_1 z - c_1 x)$.) Die Vielfachheit dieses Faktors in $H(x, y)$ ist dann $i(P, L, C)$

Aus der Definition lässt sich der folgende Satz ableiten.

Satz 4.1.20. *Sei K algebraisch abgeschlossen. Sei $C : f(x,y,z) = 0$ eine projektive Kurve vom Grad d über K und $L : g(x,y,z) = \alpha x + \beta y + \gamma z = 0$ eine projektive Gerade über K mit $L \not\subset C$. Dann gilt*

$$\sum_{P \in C \cap L} i(P, L, C) = d$$

Beweis. Sei o.B.d.A. $\gamma \neq 0$ und $\alpha' = -\alpha/\gamma, \beta' = -\beta/\gamma$. Die Geradengleichung schreibt sich dann $z = \alpha' x + \beta' y$. Man setzt z in $f(x,y,z)$ ein und bekommt ein homogenes Polynom $H(x,y) = f(x,y,\alpha' x + \beta' z)$ vom Grad d in $K[x,y]$. Da K algebraisch abgeschlossen ist, zerfällt $H(x,y)$ vollständig in Linearfaktoren:

$$H(x,y) = \xi(a_1 x - b_1 y)^{d_1} \cdots (a_k x - b_k y)^{d_k}$$

Für jeden Schnittpunkt $P = [a : b : c] \in C \cap L$ gilt $H(a,b) = 0$ und $c = \alpha' a + \beta' b$ und umgekehrt. Die Schnittpunkte sind also gerade $[a_1 : b_1 : \alpha' a_1 + \beta' b_1], \ldots, [a_k : b_k : \alpha' a_k + \beta' b_k]$ mit den Vielfachheiten $d_1, \ldots, d_k$ mit $\sum_{i=1}^{k} d_i = d$. □

Beispiel 4.1.21. *Sei K ein algebraisch abgeschlossener Körper. Sei C_1 eine Kurve vom Grad 1 (projektive Gerade) mit $C_1 = V(f_1) \subset K^2$ und C_2 eine Kurve vom Grad 3 mit $C_2 = V(f_2) \subset K^2$. Dabei ist $f_1(x,y,z) = ax + by + cz$ und $f_2(x,y,z) = y^2 z + a_1 xyz + a_3 yz^2 - x^3 - a_2 x^2 z - a_4 xz^2 - a_6 z^3$. Das Polynom f_2 definiert dabei eine Elliptische Kurve.*
Nach obigen Überlegungen schneidet eine Gerade eine Elliptische Kurve in genau drei Punkten (Vielfachheiten mitgezählt), d.h. $|C_1 \cap C_2| = 1 \cdot 3 = 3$.

Singularitäten Ein weiterer wichtiger Punkt bei der Betrachtung von ebenen algebraischen Kurven sind die *Singularitäten*. Dies kommt ursprünglich aus der Analysis, wo man häufig Wert darauf legt, dass betrachtete Objekte keine Ecken und Kanten haben, also *glatt* sind. Dazu werden Differenzierbarkeitseigenschaften verwendet. Überträgt man dies auf algebraische Kurven, so kann man zwar nicht mehr Funktionen im Sinne eines Grenzwertes von Differenzenquotienten ableiten, aber man kann die verwendeten Polynome einfach formal ableiten. Dies erfolgt durch die üblichen Rechenregeln.
Somit lässt sich *glatt* folgendermaßen definieren, wobei man anstatt glatt den Begriff *regulär* verwendet.

Definition 4.1.22. *Sei C eine projektive Kurve und $f(x,y,z)$ das Minimalpolynom für C und $P = [a : b : c] \in C$. C heißt* singulär *in P wenn gilt*

$$\frac{\partial f}{\partial x}(P) = \frac{\partial f}{\partial y}(P) = \frac{\partial f}{\partial z}(P) = 0$$

Ist C in P nicht singulär, so heißt C in P regulär.

Man nennt eine projektive Kurve C regulär, wenn sie in allen Punkten $P \in C$ regulär ist.

Bemerkung 5. *Sei $C : f(x, y, z) = 0$ eine projektive Kurve über K, $P = [a : b : c] \in C$ und sei K algebraisch abgeschlossen. Folgende beiden Eigenschaften lassen sich dann zeigen (für eine ausführliche Darstellung siehe [Hus04, Kapitel 2])*

(I) C ist genau dann regulär in P, wenn

$$i(P, C) = \min\{i(P, L, C) | L \text{ eine Gerade durch } P\} = 1.$$

Ansonsten ist $i(P, C) \geq 2$. Die Zahl $i(P, C)$ heißt auch die Ordnung *(oder* Vielfachheit*) von P auf C.*

(II) Sei C regulär in P. Es gibt genau eine Gerade L durch P, so dass $i(P, L, C) \geq 2$. Diese Gerade ist die Tangente *an C in P und hat die Gleichung*

$$\frac{\partial f}{\partial x}(P)x + \frac{\partial f}{\partial y}(P)y + \frac{\partial f}{\partial z}(P)z = 0.$$

Ist $i(P, L, C) \geq 3$ so heißt P ein Wendepunkt *von C und die Tangente an P bezeichnet man als* Wendetangente *in P.*

4.2 Elliptische Kurven

Im allgemeinen definiert man Elliptische Kurven mit Hilfe der sogenannten langen *Weierstrass-Gleichung*.

Definition 4.2.1. *Eine* Elliptische Kurve *über einem Körper K ist eine ebene reguläre projektive Kurve $\mathcal{E}$ vom Grad 3 über K, die durch die homogene Weierstrass-Gleichung der Form*

$$F : Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \tag{4.1}$$

mit Koeffizienten $a_1, a_2, a_3, a_4, a_6 \in K$ gegeben ist.

Wie später gezeigt wird, lässt sich die Regularitätsbedingung an den Koeffizienten von F ablesen.
Die homogene Weierstrass-Gleichung lässt sich auch in affiner Form schreiben ($x = \frac{X}{Z}, y = \frac{Y}{Z}, Z = 1$) und sieht dann folgendermaßen aus:

$$f : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{4.2}$$

Die etwas ungewöhnliche Nummerierung der Koeffizienten hängt u.a. mit den Isomorphismen von Elliptischen Kurven zusammen, soll aber hier nicht weiter erläutert werden.

Definition 4.2.2. *Sei K ein Körper und $\overline{K}$ der algebraische Abschluss von K. Für den Körper $\tilde{K}$ gelte $K \subseteq \tilde{K} \subseteq \overline{K}$. Ein Punkt $P = (X, Y, Z) \in \mathcal{E}$ heißt $\tilde{K}$-*rational *wenn*

$$(X, Y, Z) = \alpha(\tilde{X}, \tilde{Y}, \tilde{Z}), \quad \alpha \in \overline{K}, (\tilde{X}, \tilde{Y}, \tilde{Z}) \in \tilde{K}^3 \setminus \{(0, 0, 0)\}.$$

Die Menge der $\tilde{K}$-rationalen Punkte von $\mathcal{E}$ bezeichnet man als $\mathcal{E}(\tilde{K})$. Ist offensichtlich über welchem Körper K die Elliptische Kurve $\mathcal{E}$ definiert wurde, so bezeichnet man die K-rationalen Punkte auch einfach als *rationale Punkte*.
Der Punkt $\mathcal{O} = [0 : 1 : 0] \in \mathcal{E}$ ist der einzige rationale Punkt mit der Koordinate Z=0. Man bezeichnet ihn auch als den *unendlich fernen Punkt*. Es lässt sich leicht zeigen, dass $\mathcal{O}$ immer ein regulärer Punkt auf einer durch 4.1 gegebenen Elliptischen Kurve $\mathcal{E}$ ist, unabhängig von den Koeffizienten a_i. Nach Definition 4.1.22 ist ein Punkt einer projektiven Kurve dann regulär, wenn nicht gleichzeitig alle drei partiellen Ableitungen verschwinden. Im Fall von F ergibt sich für die partiellen Ableitungen:

$$\begin{aligned}
\frac{\partial F}{\partial X}(X,Y,Z) &= a_1YZ - 3X^2 - 2a_2XZ - a_4Z^2 \\
\frac{\partial F}{\partial Y}(X,Y,Z) &= 2YZ + a_1XZ + a_3Z^2 \\
\frac{\partial F}{\partial Z}(X,Y,Z) &= Y^2 + a_1XY + 2a_3YZ - a_2X^2 - 2a_4XZ - 3a_6Z^2
\end{aligned}$$

Für den Punkt $\mathcal{O} = [0 : 1 : 0]$ verschwinden die beiden ersten Ableitungen, für die dritte erhält man jedoch

$$\frac{\partial F}{\partial Z}(0,1,0) = 1,$$

woraus folgt, dass $\mathcal{O}$ ein regulärer Punkt auf $\mathcal{E}$ ist.

4.2.1 Vereinfachte Weierstrass-Gleichungen

Sei im folgenden $\mathcal{E}$ eine Elliptische Kurve, die durch die lange Weierstrass-Gleichung (4.2) gegeben ist.
Abhängig von der Charakteristik des zu Grunde liegenden Körpers K, lässt sich diese Gleichung nun vereinfachen. Da in dieser Arbeit endliche Körper der Charakteristik 2 – also $\mathbb{F}_{2^n}$– die Hauptrolle spielen, werden die entsprechenden Vereinfachungen für andere Charakteristika nur kurz dargestellt. In Abschnitt 4.2.2 wird dann im Speziellen auf Elliptische Kurven über $\mathbb{F}_2$ eingegangen.
Für die Visualisierung einiger exemplarischer Kurven wurde jedoch eine Darstellung über $\mathbb{R}$ gewählt.

Zur Vereinfachung der langen Weierstrass-Gleichung wurden verschiedene, von den Koeffizienten abhängige Größen definiert, deren Bezeichnungen allgemein gebräuchlich sind.

$$\begin{aligned}
b_2 &= a_1^2 + 4a_2 \\
b_4 &= 2a_4 + a_1a_3 \\
b_6 &= a_3^2 + 4a_6 \\
b_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2, \\
\triangle &= -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6, \\
j &= c_4^3/\triangle
\end{aligned}$$

Sei $char(K) \neq 2$, dann lässt sich die Gleichung 4.2 durch quadratische Ergänzung vereinfachen. Man setzt für $y \mapsto \frac{1}{2}(y - a_1x - a_3)$ und erhält schließlich eine Gleichung der Form

$$\mathcal{E} : y^2 = 4x^3 + b_2x^2 + 2b_4x + b_6$$

mit den vorher definierten Koeffizienten.

Gilt weiter $char(K) \neq 2, 3$, dann lässt sich durch eine Substitution von $x \mapsto \frac{x-3b_2}{36}$ und $y \mapsto \frac{y}{108}$ der Term bei x^2 eleminieren und man erhält schließlich die einfachere Gleichung für $\mathcal{E}$ der Form

$$\mathcal{E} : y^2 = x^3 - 27c_4x - 54c_6$$

mit $c_4 = b_2^2 - 24b_4$ und $c_6 = -b_2^3 + 36b_2b_4 - 216b_6$.

Die Größe $\triangle$ heißt die *Diskriminante* der Weierstrass-Gleichung. Die zweite wichtige Größe j bezeichnet die *j-Invariante* der Kurve $\mathcal{E}$.
Mit Hilfe der Diskriminante ergibt sich folgendes Lemma.

Lemma 4.2.3. *Die Weierstrass-Gleichung in der Form 4.2 definiert genau dann eine Elliptische Kurve, wenn die Diskriminante $\triangle \neq 0$ für diese Gleichung ist.*

Der Beweis zu diesem Lemma soll hier nur kurz skizziert werden. Im Fall von Charakteristik 2 wird er dann genauer erläutert.

Beweis. Um zu zeigen, dass eine gegebene Kurve regulär ist, muss gezeigt werden dass sie in jedem Punkt regulär ist. Für den Punkt $\mathcal{O}$ haben wir dies schon gezeigt, so dass sich der weitere Beweis nur noch auf den affinen Teil der Kurve beschränkt. Man muss nun zeigen, dass für jeden beliebigen Punkt $P = (x, y) \in \mathcal{E}$ nicht beide partiellen Ableitungen der affinen Weierstrass-Gleichung gleichzeitig verschwinden. Rechnet man dies durch, so kommt man schließlich auf die oben definierte Diskriminante $\triangle$ und stellt fest, dass für $\triangle \neq 0$ gilt, $\mathcal{E}$ ist regulär. Einen genauen Beweis findet man in [Sil86, Kapitel 3.1] oder [Wer02, Kapitel 2]. □

In Abbildung 4.2 sind Beispiele für die reellen Varietäten von Elliptischen Kurven gezeigt. Abbildung 4.1 zeigt zwei Kurven die Singularitäten enthalten, also nicht regulär und damit keine Elliptischen Kurven im Sinn von Definition 4.2.1 sind. Die Singularitäten liegen in diesen Fällen im Punkt $(0, 0)$.
Die j-Invariante ist ein Merkmal der Isomorphieklassen von Elliptischen Kurven.

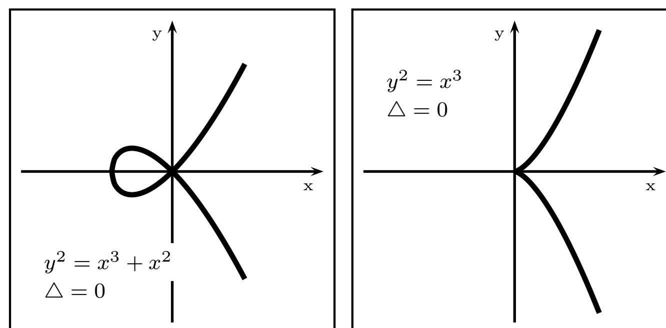

Abbildung 4.1: *Beispiele für Singularitäten (im Ursprung) einer Kurve*

Es lässt sich zeigen, dass für zwei über K isomorphe Elliptische Kurven $\mathcal{E}_1(K)$ und

$\mathcal{E}_2(K)$ gilt, $j(\mathcal{E}_1) = j(\mathcal{E}_2)$.
Der umgekehrte Fall gilt zumindest immer dann, wenn K ein algebraisch abgeschlossener Körper ist(siehe dazu [Sil86, Kapitel 3, Prop. 1.4(b)]

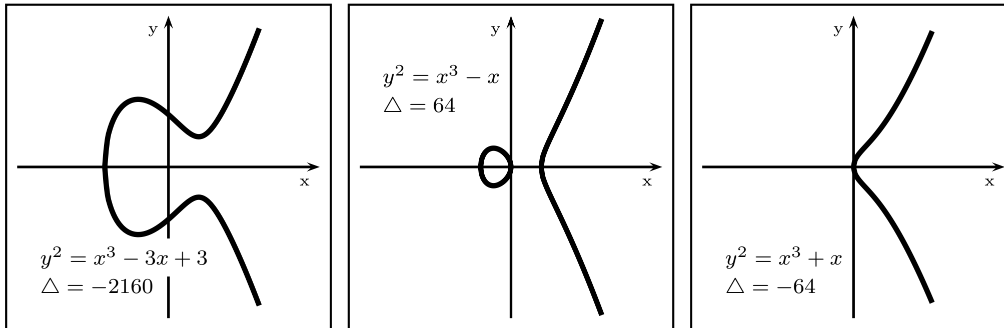

Abbildung 4.2: *Drei Beispiele für Elliptische Kurven über* $\mathbb{R}$

4.2.2 Elliptische Kurven über $\mathbb{F}_{2^n}$

In dieser Arbeit soll besonders der Fall für Charakteristik 2 behandelt werden. Auch in diesem Fall lässt sich die Weierstrass-Gleichung (4.2) durch Koordinatentransformation in eine speziellere Form bringen. Dabei werden die zwei Fälle für $a_1 \neq 0$ und $a_1 = 0$ unterschieden.
Im zweiten Fall ($a_1 = 0$) führt eine Transformation mit $x \mapsto x' + a_2, y \mapsto y'$ zu folgender Gleichung

$$y^2 + a_3 y = x^3 + a_4 x + a_6.$$

Man kann zeigen, dass diese Form der Elliptischen Kurve zu einer besonderen Art von Elliptischen Kurven gehört. Diese besonderen Kurven nennt man allgemein *supersingulär*. Durch diese Eigenschaft werden sie für eine Verwendung im kryptographischen Umfeld uninteressant. Für supersinguläre Kurven existiert die sogenannte *MOV*-Attacke, welche das diskrete Logarithmusproblem in der Punktegruppe einer solchen Kurve auf das diskrete Logarithmusproblem in einem endlichen Körper reduziert. Aus diesem Grund ist eine Verwendung solcher spezieller Kurven für krypographische Anwendungen nicht empfehlenswert.
Mehr zu supersingulären Kurven und zur *MOV*-Attacke findet sich unter anderem in [MOV93],[Wer02] und [BSS99].

Der kryptographisch wesentlich wichtigere Fall $a_1 \neq 0$ soll nun etwas genauer beleuchtet werden.
Ausgehend von der langen affinen Weierstrass-Gleichung

$$y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6$$

führt die Koordinatentransformation

$$x \mapsto a_1^2 x' + \frac{a_3}{a_1}, \quad y \mapsto a_1^3 y' + \frac{a_1^2 a_4 + a_3^2}{a_1^3}$$

zu folgender Gleichung: (Für die bessere Lesbarkeit wurden die Striche weggelassen.)

$$\left(a_1^3y + \frac{a_1^2a_4 + a_3^2}{a_1^3}\right)^2 + a_1\left(a_1^2x + \frac{a_3}{a_1}\right)\left(a_1^3y + \frac{a_1^2a_4 + a_3^2}{a_1^3}\right) + a_3\left(a_1^3y + \frac{a_1^2a_4 + a_3^2}{a_1^3}\right)$$
$$= \left(a_1^2x + \frac{a_3}{a_1}\right)^3 + a_2\left(a_1^2x + \frac{a_3}{a_1}\right)^2 + a_2\left(a_1^2x + \frac{a_3}{a_1}\right) + a_6$$

Ausrechnen der quadratischen und kubischen Terme unter Beachtung der Rechenregeln für Charakteristik 2 ergibt

$$a_1^6y^2 + a_1^6xy = a_1^6x^3 + (a_1^3a_3 + a_1^4a_2)x^2 + \frac{a_1^6a_6 + a_1^5a_3a_4 + a_1^4a_3^2a_2 + a_1^4a_4^2 + a_1^3a_3^3 + a_3^4}{a_1^6}$$

Da nach Voraussetzung $a_1 \neq 0$ gilt, kann durch a_1^6 dividiert werden und man erhält letztlich die Gleichung

$$y^2 + xy = x^3 + ax^2 + b \tag{4.3}$$

mit den Koeffizienten

$$a = \frac{a_3 + a_1a_2}{a_1^3}, \quad b = \frac{a_1^6a_6 + a_1^5a_3a_4 + a_1^4a_3^2a_2 + a_1^4a_4^2 + a_1^3a_3^3 + a_3^4}{a_1^{12}}.$$

Die für die allgemeine Weierstrass-Gleichung (4.2) definierte Diskriminante $\triangle$ lässt sich durch Einsetzen der Koeffizienten ebenfalls in eine Form für Charakteristik 2 bringen. Die Diskriminante ist allgemein definiert als $\triangle = -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6$. In Charakteristik 2 wird daraus

$$\begin{aligned} \triangle &= b_2^2b_8 + b_6^2 + b_2b_4b_6 \\ &= (a_1^2)^2(a_1^2a_6 + a_1a_3a_4 + a_2a_3^2 + a_4^2) + (a_3^2)^2 + (a_1^2)(a_1a_3)(a_3^2) \\ &= a_1^6a_6 + a_1^5a_3a_4 + a_1^4a_3^2a_2 + a_1^4a_4^2 + a_1^3a_3^3 + a_3^4. \end{aligned}$$

Die j-Invariante wird in Charakteristik 2 zu

$$j = \frac{c_4^3}{\triangle} = \frac{(b_2^2)^3}{\triangle} = \frac{((a_1^2)^2)^3}{\triangle} = \frac{a_1^{12}}{\triangle}.$$

Sieht man sich die Größe $\triangle$ etwas genauer an, so fällt auf, dass sie bereits in dem vorher berechneten Koeffizienten b steckt. Man könnte b auch schreiben als $b = \frac{\triangle}{a_1^{12}}$. Damit lässt sich nun auch Lemma 4.2.3 für den Fall der Charakteristik 2 beweisen.

Beweis. Sei $\mathcal{E}$ eine durch $f(x,y) = y^2 + a_1xy + a_3y + x^3 + a_2x^2 + a_4x + a_6$ gegebene Kurve über $\mathbb{F}_2$. Man sucht einen singulären Punkt $P = (p_1, p_2) \in \mathcal{E}$. Für einen solchen Punkt gilt:

$$\begin{aligned} f(p_1,p_2) &= 0 \\ \frac{\partial f}{\partial x}(p_1,p_2) &= a_1p_2 + p_1^2 + a_4 = 0 \\ \frac{\partial f}{\partial y}(p_1,p_2) &= a_1p_1 + a_3 = 0. \end{aligned}$$

Da $a_1 \neq 0$ können wir dividieren und erhalten

$$p_1 = \frac{a_3}{a_1} \quad \text{und} \quad p_2 = \frac{1}{a_1}\left(a_4 + \frac{a_3^2}{a_1^2}\right) = \frac{a_3^2 + a_1^2a_4}{a_1^3}.$$

Setzt man dies in $f(p_1, p_2)$ ein, so erhält man

$$f(p_1, p_2) = \frac{\overbrace{a_1^6 a_6 + a_1^5 a_3 a_4 + a_1^4 a_3^2 a_2 + a_1^4 a_4^2 + a_1^3 a_3^3 + a_3^4}^{\triangle}}{a_1^6},$$

woraus für $f(p_1, p_2) = 0$ folgen muss $\triangle = 0$.
Im umgekehrten Fall ($\triangle = 0$) kann man sich einen singulären Punkt folgendermaßen konstruieren. Setze p_1, p_2 so wie vorher, dann berechnet sich $f(p_1, p_2) = \frac{\triangle}{a_1^6}$, woraus $f(p_1, p_2) = 0$ folgt.
Damit ist gezeigt, dass singuläre Punkte nur dann auftreten können, wenn die Diskriminate $\triangle = 0$. Für $\triangle \neq 0$ ist $\mathcal{E}$ somit eine reguläre Kurve und damit eine Elliptische Kurve. □

Korollar 4.2.4. *Sei $\mathcal{E}$ gegeben durch die transformierte Weierstrass-Gleichung 4.3 dann ist die Diskriminante für diese Kurve gegeben durch $\triangle = b$.*
Die j-Invariante berechnet sich mit $j = \frac{1}{b}$.

4.2.3 Das Gruppengesetz und Punktaddition

Für die Punkte einer Elliptischen Kurve $\mathcal{E}$ kann mit Hilfe der *Sekanten-Tangenten-Methode* eine Addition von Punkten definiert werden. Dazu verwendet man nun die Vorüberlegungen zum Schnittverhalten von projektiven Kurven mit Geraden aus Kapitel 4.1.3. Wie dort gezeigt wurde, schneidet eine projektive Gerade eine projektive Kurve vom Grad 3 immer in genau drei Punkten, wenn man die Schnittmultiplizitäten mitzählt. Damit definiert man die Punktaddition nun folgendermaßen.

Definition 4.2.5. *Sei $\mathcal{E}$ eine durch eine Weierstrass-Gleichung gegebene Elliptische Kurve und $\mathcal{O} \in \mathcal{E}$ ein ausgezeichneter Punkt. Seien weiter $P, Q \in \mathcal{E}$ und L eine Gerade durch P und Q (wenn $P = Q$ ist L die Tangente an $\mathcal{E}$ in P). Der dritte Schnittpunkt von $L \cap \mathcal{E}$ sei $R \in \mathcal{E}$ und sei weiter L' eine Gerade durch R und $\mathcal{O}$. Man nennt den dritten Schnittpunkt von $L' \cap \mathcal{E}$ dann $P \oplus Q$.*

In Abbildung 4.3 ist diese Definition noch einmal graphisch dargestellt. Für die Punktaddition gilt das folgende Lemma, welches man sich auch anhand der Abbildung klarmachen kann.

Lemma 4.2.6. *Seien P, Q, R die drei Schnittpunkte beim Schnitt einer Geraden L mit einer Elliptischen Kurve $\mathcal{E}$, die nicht notwendigerweise verschieden sein müssen. Es gilt*

$$(P \oplus Q) \oplus R = \mathcal{O}.$$

Beweis. Der Punkt $T = P \oplus Q$ ist der dritte Schnittpunkt der Geraden L' durch R und $\mathcal{O}$ mit der Kurve $\mathcal{E}$. Addiert man nun T und R, so erhält man zunächst $\mathcal{O}$ als dritten Schnittpunkt und muss nun eine Gerade durch diesen und $\mathcal{O}$ legen, was einer Tangente an $\mathcal{E}$ in $\mathcal{O}$ entspricht. Diese Tangente hat im Punkt $\mathcal{O}$ die Schnittmultiplizität 3. □

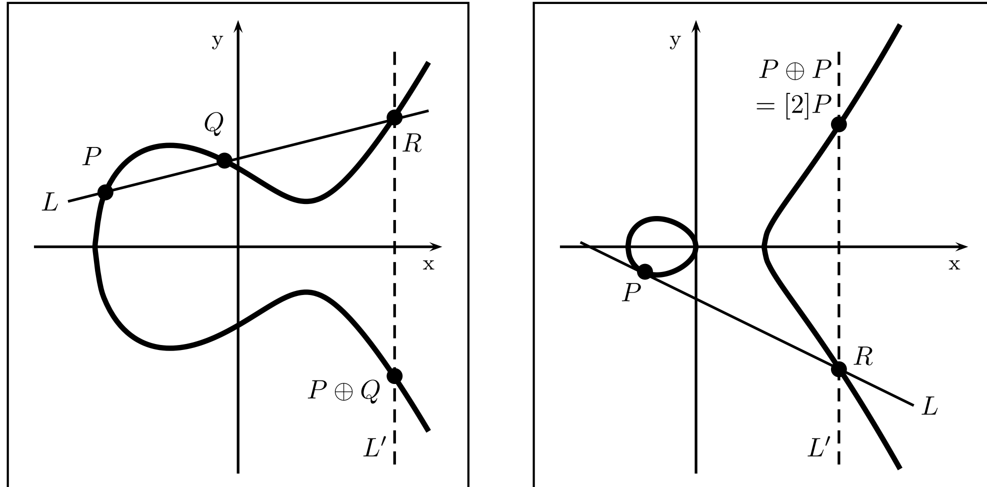

Abbildung 4.3: *Punktaddition veranschaulicht über* $\mathbb{R}$

Durch die soeben definierte Punktaddition werden die Punkte auf einer Elliptischen Kurve zuammen mit der Addition zu einer abelschen Gruppe. Dies ist in dem folgenden Satz von POINCARÉ zusammengefasst.

Satz 4.2.7. *Es sei $\mathcal{E}$ eine Elliptische Kurve, $\mathcal{O} \in E$ ein ausgezeichneter Punkt und $\oplus$ die in Definition 4.2.5 definierte Punktaddition. Es gilt $(\mathcal{E}, \oplus)$ ist eine abelsche Gruppe mit neutralem Element $\mathcal{O}$. Folgende Eigenschaften gelten:*

(I) $P \oplus \mathcal{O} = P$ *für alle* $P \in \mathcal{E}$ (Neutrales Element)

(II) $P \oplus Q = Q \oplus P$ *für alle* $P, Q \in \mathcal{E}$ (Kommutativität)

(III) Für alle $P \in \mathcal{E}$ *gibt es einen Punkt* $\ominus P \in \mathcal{E}$ *so dass gilt:* (Inverses Element)

$$P \oplus (\ominus P) = \mathcal{O}$$

(IV) Sei $P, Q, R \in \mathcal{E}$*. Es gilt:* (Assoziativität)

$$(P \oplus Q) \oplus R = P \oplus (Q \oplus R)$$

Beweis. (I) Sei L eine Gerade durch die Punkte P und $\mathcal{O}$. Der dritte Schnittpunkt von $L \cap \mathcal{E}$ sei R. Die Gerade L' die durch R und $\mathcal{O}$ gelegt wird, hat als dritten Schnittpunkt mit $\mathcal{E}$ wieder P, denn $L = L'$.

(II) Dies folgt sofort aus der Konstruktion der Punktaddition. Die Gerade L hängt nicht von der Reihenfolge der Punkte P und Q ab und damit auch nicht das Ergebnis $P \oplus Q$.

(III) Man definiert das Inverse $\ominus P$ als den dritten Schnittpunkt einer Geraden L durch $\mathcal{O}$ und P mit $\mathcal{E}$. Nach Lemma 4.2.6 gilt für die drei Schnittpunkte einer Geraden mit einer Elliptischen Kurve

$$(P \oplus \mathcal{O}) \oplus (\ominus P) = \mathcal{O}.$$

Ein Spezialfall ist der Punkt $\mathcal{O}$, für diesen gilt $\mathcal{O} = \ominus\mathcal{O}$.

(IV) Der allgemeine Beweis für die Assoziativität erfordert tiefergehendes algebraisches Wissen, weshalb hier auf [Sil86, Kapitel III.3] oder [Was03, Kapitel 2.4] sowie darin verwiesene Literatur hingewiesen werden soll.
Eine zweite Möglichkeit besteht darin, mit Hilfe der im nächsten Abschnitt entwickelten Additionsformeln alle möglichen Fälle nachzurechnen.

□

Explizite Formeln zur Punktaddition

In diesem Abschnitt sollen Formeln für die Punktaddition gezeigt und hergeleitet werden. Ausgehend von der langen Weierstrass-Gleichung werden zunächst die Formeln ganz allgemein berechnet und dann für den Spezialfall $char(K) \neq 2, 3$ angegeben. Für den Fall $char(K) = 2$ werden die Formeln für die vereinfachte Weierstrass-Gleichung bewiesen.
Vorher jedoch noch eine Bemerkung zur Schreibweise.

Bemerkung 6. *Für die speziellen Symbole zur Punktaddition $\oplus$ und für das Inverse $\ominus$ eines Punktes werden im folgenden wieder die gewöhnlichen Additionssymbole $+, -$ verwendet. Weiterhin schreibt man für die m-fache Addition eines Punktes P mit $m \in \mathbb{Z}$ und $P \in \mathcal{E}$*

$$\begin{aligned} [m]P &=_{def} \overbrace{P + \ldots + P}^{m \ mal} \quad \text{für } m > 0 \\ [-m]P &=_{def} -[m]P \quad \text{für } m < 0 \text{ und} \\ [0]P &=_{def} \mathcal{O}. \end{aligned}$$

Zunächst betrachtet man eine Elliptische Kurve $\mathcal{E}$, gegeben durch die allgemeine affine Weierstrass-Gleichung $f(x, y)$. Sei $P = (x_0, y_0) \in \mathcal{E}$ ein Punkt auf der Kurve, dann berechnet sich $-P \in \mathcal{E}$ als der dritte Schnittpunkt einer Geraden L durch P und $\mathcal{O}$. Zur Verdeutlichung hier nochmals die affine Weierstrass-Gleichung sowie die affine Gleichung einer Geraden durch P und $\mathcal{O}$

$$\begin{aligned} f(x, y) &= y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6 = 0 \\ L: &\quad x - x_0 = 0 \end{aligned}$$

Ersetzt man nun $x = x_0$ in f, so erhält man ein quadratisches Polynom in y folgender Form: $y^2 + (a_1x_0 + a_3)y - x_0^3 - a_2x_0^2 - a_4x_0 - a_6 = 0$. Dies entspricht einer quadratischen Gleichung, die man allgemein schreiben kann als

$$y^2 + Ay + B = 0$$

mit $A = a_1x_0 + a_3$ und $-B = x_0^3 + a_2x_0^2 + a_4x_0 + a_6$. Diese Gleichung hat allgemein zwei Lösungen y_0 und y_1, von denen die erste bereits bekannt ist. Da sich die Nullstellen immer abdividieren lassen, können wir für $f(x_0, y)$ auch schreiben

$$f(x_0, y) = c(y - y_0)(y - y_1).$$

Durch Ausmultiplizieren und anschließenden Koeffizientenvergleich bei y^2 erhalten wir $c = 1$, bei y erhält man $A = -y_0 - y_1$. Stellt man um und setzt obiges A ein, so berechnet sich mit $y_1 = -y_0 - a_1x_0 - a_3$ die y-Koordinate des Punktes

$$-P = (x_0, -y_0 - a_1x_0 - a_3).$$

Betrachtet man nun die Addition zweier verschiedener Punkte $P_1(x_1, y_1), P_2(x_2, y_2) \in \mathcal{E}$, so unterscheidet man folgende drei Fälle.

(I) Sei zunächst $P_1 = -P_2$, d.h. $x_1 = x_2$ und $y_2 = -y_1 - a_1x_1 - a_3$ so folgt aus obiger Formel sofort $P_1 + P_2 = \mathcal{O}$.

(II) Es sei nun $P_1 \neq P_2$ und weiterhin $x_1 \neq x_2$. Gilt die zweite Bedingung nicht, so befindet man sich wieder in Fall *I*. Legt man nun eine Gerade L mit der Gleichung

$$L : y = \lambda x + \nu$$

durch die Punkte P_1 und P_2, so schneidet diese Gerade die Kurve $\mathcal{E}$ in genau einem weiteren Punkt. Der Koeffizient λ stellt anschaulich die Steigung dieser Geraden dar und lässt sich demzufolge einfach mit

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$$

berechnen. Damit kann nun ν allgemein geschrieben werden als

$$\nu = y_1 - \lambda x_1 = y_1 - \frac{y_2 - y_1}{x_2 - x_1}x_1 = \frac{y_1(x_2 - x_1) - x_1(y_2 - y_1)}{x_2 - x_1} = \frac{y_1x_2 - y_2x_1}{x_2 - x_1}.$$

Schneidet man nun die Gerade L mit der Elliptischen Kurve $\mathcal{E}$, d.h. man setzt in die affine Weierstrass-Gleichung $f(x, y) = 0$ ein, so müssen alle affinen Punkte $P = (x, y)$ der Gleichung

$$\begin{aligned}(\lambda x + \nu)^2 + a_1x(\lambda x + \nu) + a_3(\lambda x + \nu) - x^3 - a_2x^2 - a_4x - a_6 = \\ -x^3 + (\lambda^2 + a_1\lambda - a_2)x^2 + (2\lambda\nu + a_1\nu + a_3\lambda - a_4x)x + (\nu^2 + a_3\nu - a_6) = 0\end{aligned} \tag{4.4}$$

entsprechen. Diese Gleichung entspricht einer Polynom-Gleichung dritten Grades in x, für die zwei Lösungen (x_1 und x_2) bereits bekannt sind. Über dem algebraischen Abschluss zerfällt das Polynom in drei Faktoren. Diese lassen sich schreiben als

$$c(x - x_1)(x - x_2)(x - x_3') = 0.$$

Ausmultipliziert ergibt sich

$$cx^3 - c(x_3' + x_2 + x_1)x^2 + \ldots = 0.$$

Ein Koeffizientenvergleich dieser beiden Polynom-Gleichungen ergibt $c = -1$ und $x_3' + x_2 + x_1 = \lambda^2 + a_1\lambda - a_2$ woraus folgt

$$x_3' = \lambda^2 + a_1\lambda - a_2 - x_2 - x_1 = (\lambda + a_1)\lambda - a_2 - x_2 - x_1.$$

Damit sind die Koordinaten des dritten Schnittpunktes P_3' von $L \cap \mathcal{E}$ berechnet und lauten $P_3'(x_3', \lambda x_3' + \nu)$.
Aus der Definition der Punktaddition folgt

$$P_3' = -(P_1 + P_2) \Rightarrow P_3 = -P_3' = P_1 + P_2$$

und man erhält durch Negation von P_3' das Ergebnis der Punktaddition von $P_3 = P_1 + P_2$ mit

$$\begin{aligned} x_3 = x_3' &= \lambda^2 + a_1\lambda - a_2 - x_2 - x_1 \\ y_3 &= -y_3' - a_1 x_3' - a_3 = -(\lambda x_3' + \nu) - a_1 x_3' - a_3 \\ &= -(\lambda + a_1)x_3' - \nu - a_3 = -(\lambda + a_1)x_3 - \nu - a_3 \end{aligned}$$

(III) Sei $P_1 = P_2 = P = [x_1 : y_1 : 1]$, dann legt man im Punkt P eine Tangente an $\mathcal{E}$. Im projektiven schreibt sich diese Tangente mit $L(\alpha, \beta, \gamma) = \frac{\partial f}{\partial x}(P)x + \frac{\partial f}{\partial y}(P)y + \frac{\partial f}{\partial z}(P)z$ und die drei Koeffizienten berechnen sich mit

$$\begin{aligned} \alpha = \frac{\partial F}{\partial X}(x_1, y_1, 1) &= a_1 y_1 - 3x_1^2 - 2a_2 x_1 - a_3 \\ \beta = \frac{\partial F}{\partial Y}(x_1, y_1, 1) &= 2y_1 + a_1 x_1 + a_3 \quad \text{und} \\ \gamma = \frac{\partial F}{\partial Z}(x_1, y_1, 1) &= y_1^2 + a_1 x_1 y_1 + 2a_3 y_1 - a_2 x_1^2 - 2a_4 x_1 - 3a_6. \end{aligned}$$

Es muss $\beta \neq 0$ gelten, da sonst der Punkt $\mathcal{O} = [0 : 1 : 0]$ auf der Tangente L liegt und es sich damit um den ersten Fall $P_1 + P_1 = \mathcal{O}$ handelt, woraus folgt $P_1 = -P_1$. Geht man nun zum affinen über, so liegt der Punkt $P = (x_1, y_1)$ auf der affinen Gerade

$$y_1 = \lambda x_1 + \nu,$$

wobei sich λ und ν aus der nach y_1 aufgelösten projektiven Geradengleichung für L folgendermaßen ergibt:

$$\begin{aligned} \lambda = -\frac{\alpha}{\beta} &= \frac{a_1 y_1 - 3x_1^2 - 2a_2 x_1 - a_3}{2y_1 + a_1 x_1 + a_3} \quad \text{und} \\ \nu = -\frac{\gamma}{\beta} &= \frac{y_1^2 + a_1 x_1 y_1 + 2a_3 y_1 - a_2 x_1^2 - 2a_4 x_1 - 3a_6}{2y_1 + a_1 x_1 + a_3} \\ &= \frac{-x_1^3 + a_4 x_1 + 2a_6 - a_3 y_1}{2y_1 + a_1 x_1 + a_3} \end{aligned}$$

Schneidet man nun, wie in Fall *II*, die affine Gerade mit der affinen Weierstrass-Gleichung, so erhält man als Ergebnis Gleichung 4.4, die über dem algebraischen Abschluss wieder in drei Linearfaktoren zerfällt. Auch hier lässt sich also schreiben

$$c(x - x_1)(x - x_2')(x - x_3') = 0 \quad c \in K, x_2', x_3' \in \overline{K}. \tag{4.5}$$

Ausmultiplizieren und vergleichen der Koeffizienten bei x^3 und x^2 führt zu

$$c = -1 \text{ und } y^2 + a_1\lambda - a_2 = x_1 + x_2' + x_3'.$$

Da L Tangente in P ist, muss die Vielfachheit von P in $\mathcal{E} \cap L$ größer oder gleich zwei sein. Dies ist gleich der Ordnung der Nullstelle von x_1 in 4.5, weshalb man o.B.d.A. $x_1 = x_2'$ setzen kann, woraus folgt

$$x_3' = \lambda^2 + a_1\lambda - a_2 - 2x_1, \quad x_3' \in K.$$

Wie im zweiten Fall schneidet L die Kurve $\mathcal{E}$ noch im Punkt $P_3' = (x_3', y_3')$ mit $y_3' = \lambda x_3' + \nu$.

Verfährt man nun weiter wie in Fall *II*, so erhält man für den Punkt $P_3 = (x_3, y_3) = P_1 + P_2 = [2]P$ die folgenden Koordinaten

$$\begin{aligned} x_3 &= \lambda^2 + a_1\lambda - a_2 - 2x_1 \\ y_3 &= -(\lambda + a_1)x_3 - \nu - a_3 \end{aligned}$$

wobei λ und ν den für Fall *III* berechneten Formeln entsprechen.

Für die allgemeine Weierstrass-Gleichung über einem Körper K mit $char(K) \neq 2, 3$ vereinfachen sich die allgemeinen Additionsformeln. Die nachfolgende Bemerkung fasst dies kurz zusammen. Für einen Beweis wird auf [Wer02, Satz 2.3.14], [Sil86] und [Was03, Kapitel 2.2] verwiesen.

Bemerkung 7. *Eine Elliptische Kurve $\mathcal{E}$ sei definiert durch $y^2 = x^3 + Ax + B$ und $char(K) \neq 2, 3$.*

(I) Für $P = (x, y) \in \mathcal{E}$ ist $-P = (x, -y)$.

(II) Sei $P_1 = -P_2$, dann ist $P_1 + P_2 = \mathcal{O}$.

(III) Sei $P_1 \neq P_2$, dann berechnet sich $P_3 = P_1 + P_2$ durch

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \quad \textit{mit} \\ \lambda &= \begin{cases} \frac{y_2 - y_1}{x_2 - x_1}, & \textit{falls } P_1 \neq P_2 \\ \frac{3x_1^2 + A}{2y_1}, & \textit{falls } P_1 = P_2. \end{cases} \end{aligned}$$

In dem für diese Arbeit wichtigeren Fall $char(K) = 2$ ergeben sich die Additionsformeln entsprechend.

Satz 4.2.8. *Es sei $char(K) = 2$ und $\mathcal{E}$ eine Elliptische Kurve definiert durch die vereinfachte Weierstrass-Gleichung*

$$y^2 + xy = x^3 + Ax^2 + B.$$

(I) Sei $P = (x, y) \in \mathcal{E}$, dann ist $-P = (x, x + y)$

(II) Seien $P_1 = (x_1, y_1)$ und $P_2 = (x_2, y_2) \in \mathcal{E}$ mit $P_1 \neq -P_2$. Der Punkt $P_3 = (x_3, y_3) = P_1 + P_2$ berechnet sich mit

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + A + x_1 + x_2, \\ y_3 &= (x_1 + x_3)\lambda + x_3 + y_1 \quad \textit{wobei} \\ \lambda &= \begin{cases} \frac{y_2 + y_1}{x_2 + x_1}, & \textit{falls } x_1 \neq x_2 \\ x_1 + \frac{y_1}{x_1}, & \textit{falls } x_1 = x_2 \neq 0 \end{cases} \end{aligned}$$

Beweis. Der Beweis für Charakteristik zwei erfolgt äquivalent zum allgemeinen Fall und wird eben deshalb nur kurz skizziert.

(I) Der dritte Schnittpunkt einer Geraden L durch P und $\mathcal{O}$ mit $\mathcal{E}$ ist definiert als $-P$. Die Gleichung für die Gerade $L : x = x_0$ schneidet man mit der affinen Weierstrass-Gleichung und erhält $y^2+x_0y = x_0^3+Ax_0^2+B$, wofür y_0 eine Nullstelle ist. Die zweite Nullstelle ist dann $y_1 = x_0 + y_0$ da $(x_0 + y_0)^2 + x_0(x_0 + y_0) = x_0^2 + y_0^2 + x_0^2 + x_0y_0 = y_0^2 + x_0y_0$. Somit folgt $-P = (x_0, x_0 + y_0)$.

(II) Es sei L eine Gerade durch $P_1 = (x_1, y_1)$ und $P_2 = (x_2, y_2)$ die durch $y = \lambda x + \nu$ definiert ist.
Wie für die allgemeine Weierstrass-Gleichung, müssen auch hier zwei Fälle unterschieden werden.

  (i) Wenn $x_1 \neq x_2$, dann berechnet sich $\lambda = \frac{y_2+y_1}{x_2+x_1}$ anschaulich als Steigung von L.

  (ii) Gilt $x_1 = x_2 = x$, so reicht es den Fall $y_1 = y_2$ zu betrachten, da im zweiten Fall $y_1 \neq y_2$ gilt $y_1 = y_2 + x$. Dies bedeutet $P_1 = -P_2$ und es gilt damit $P_1 + P_2 = \mathcal{O}$. Man betrachtet für $y_1 = y_2$ nun eine Tangente an $\mathcal{E}$ im Punkt P. Für die Steigung dieser Tangente gilt für $x \neq 0$: $\lambda = \frac{x^2+y}{x} = x + \frac{y}{x}$. Ist $x = 0$, so gilt $P = -P$ und damit $[2]P = \mathcal{O}$.

Schneidet man nun L mit $\mathcal{E}$, d.h. einsetzen der Geradengleichung für y in die Gleichung von $\mathcal{E}$, so erhält man wie im allgemeinen Fall ein Polynom dritten Grades, für das bereits zwei Nullstellen bekannt sind. Durch Koeffizientenvergleich berechnet sich der dritte Schnittpunkt $P_3 = (x_3, y_3')$ mit

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + A + x_1 + x_2 \\ y_3' &= \lambda x_3 + \nu = \lambda x_3 + y_1 + \lambda x_1 = (x_1 + x_3)\lambda + y_1. \end{aligned}$$

Weiterhin gilt $P_3 = -P_3'$ woraus folgt

$$(x_3, y_3) = -(x_3, y_3') = (\lambda^2 + \lambda + A + x_1 + x_2, (x_1 + x_3)\lambda + y_1 + x_3).$$

□

Elliptische Kurven über $\mathbb{R}$ lassen sich durch Zeichnungen sehr einfach veranschaulichen. Über endlichen Körpern ist dies nicht mehr so einfach möglich, da man hier keine „Kurven“ im eigentlichen Sinn zeichnen kann, sondern sich die einzelnen Punkte auf einer Art Gitter vorstellt. Das folgende Beispiel, welches von der Webseite der Firma Certicom[3] stammt, soll diese Zusammenhänge etwas veranschaulichen.

Beispiel 4.2.9. *Betrachtet man den endlichen Körper $\mathbb{F}_{2^m}$ $(m = 4)$ mit dem irreduziblen Polynom $f(x) = x^4 + x + 1$ und dem Generatorelement $g = x \mod f$ (siehe dazu auch Beispiel 2.2.1), dann ist eine mögliche Elliptische Kurve $\mathcal{E}$ gegeben durch*

$$\mathcal{E} : y^2 + xy = x^3 + g^4x^2 + 1,$$

mit Koeffizienten $A = g^4$ und $B = g^0 = 1$. Sowohl für die Diskriminante $\triangle$, als auch die j-Invariante gilt $\triangle = j = 1$, womit $\mathcal{E}$ tatsächlich eine Elliptische Kurve darstellt. Für heutige kryptographische Anwendungen sollte der Exponent m mindestens $m > 160$ sein, um damit einer Berechnung und vor allem Speicherung aller $2^m - 1$ Potenzen von g entgegen zu treten.
In Abbildung 4.4 sind alle affinen Punkte von $\mathcal{E}$ dargestellt. Alle diese Punkte erfüllen die vereinfachte Weierstrass-Gleichung für $\mathcal{E}$. Zur Verdeutlichung sei der Punkt $P = (g^5, g^3) \in \mathcal{E}$ gewählt. Die Darstellung der Art (1010) entpricht der in Kapitel 2.2.1 vorgestellten Vektordarstellung eines Polynoms.

$$\begin{aligned}(g^3)^2 + g^5 g^3 &= (g^5)^3 + g^4 (g^5)^2 + 1 \\ g^6 + g^8 &= g^{15} + g^{14} + g^0 \\ (1100) + (0101) &= (0001) + (1001) + (0001) \\ (1001) &= (1001)\end{aligned}$$

Alle affinen Punkte zusammen mit $\mathcal{O}$ bilden die Punktegruppe $\mathcal{E}(\mathbb{F}_{2^4})$ der Elliptischen Kurve.

$$\begin{aligned}\mathcal{E}(\mathbb{F}_{2^4}) = \{&\mathcal{O}, (1, g^{13}), (1, g^6), (g^3, g^{13}), (g^5, g^{11}), (g^6, g^{14}), (g^9, g^{13}), (g^{10}, g^8), \\ &(g^3, g^8), (g^5, g^3), (g^6, g^8), (g^9, g^{10}), (g^{10}, g), (g^{12}, 0), (g^{12}, g^{12}), (0, 1)\}\end{aligned}$$

Im folgenden vier Beispiele für die Anwendung der Additionsregeln:

(I) *Seien $P_1 = (g^{10}, g), P_2 = (g^{10}, g^8) \in \mathcal{E}$, dann ist $P_1 + P_2 = \mathcal{O}$, denn $P_1 = -P_2 = (g^{10}, g^{10} + g^8) = (g^{10}, (0111) + (0101)) = (g^{10}, (0010)) = (g^{10}, g)$.*

(II) *Seien $P_1 = (g^9, g^{10}), P_2 = (g^{10}, g) \in \mathcal{E}$, dann ergibt sich für $P_1 + P_2 = P_3 \in \mathcal{E}$ ein $\lambda = \frac{g^8}{g^{13}} = \frac{g^8}{g^{-2}} = g^8 g^2 = g^{10}$, woraus für $x_3 = (g^{10})^2 + g^{10} + g^4 + g^9 + g^{10} = g^5 + g^{10} + g^4 + g^9 + g^{10} = (0110) + (0011) + (1010) = (1111) = g^{12}$ und $y_3 = (g^9 + g^{12}) g^{10} + g^{12} + g^{10} = g^4 + g^7 + g^{12} + g^{10} = (0011) + (1011) + (1111) + (0111) = (0000) = 0$ der Punkt $P_3 = (g^{12}, 0)$ berechnet wird.*

(III) *Für den Punkt $Q = (g^6, g^{14})$ berechnet sich $[2]Q = P_3$ mit $\lambda = g^6 + \frac{g^{14}}{g^6} = g^6 + g^{14} g^9 = g^6 + g^8 = (1100) + (0101) = (1001) = g^{14}$ zu $x_3 = (g^{14})^2 + g^{14} + g^4 = g^{13} + g^{14} + g^4 = (0111) = g^{10}$ und $y_3 = (g^6 + g^{10}) g^{14} + g^{10} + g^{14} = g^5 + g^9 + g^{10} + g^{14} = (0010) = g$.*

(IV) *Für $P = (0, 1)$ ist $[2]P = \mathcal{O}$, denn $P = -P(0, 1 + 0)$ und es folgt $P + (-P) = \mathcal{O}$.*

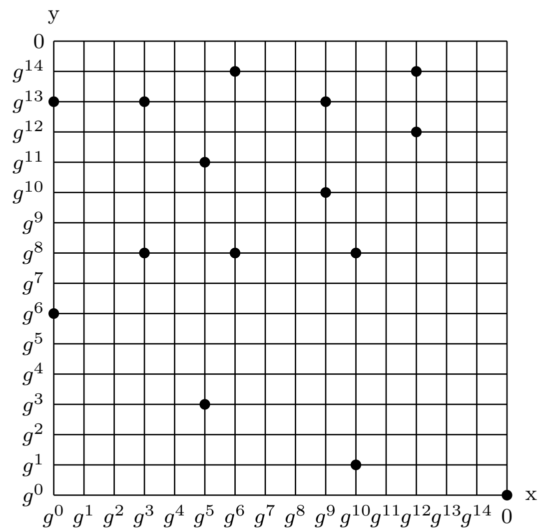

Abbildung 4.4: *Die Punkte einer Elliptischen Kurve ($\mathcal{E} : y^2 + xy = x^3 + g^4x^2 + 1$) über $\mathbb{F}_{2^4}$*

Kapitel 5

Implementierung der Punktmultiplikation auf Elliptischen Kurven

Kryptosysteme die auf Elliptischen Kurven basieren, nutzen die Gruppenstruktur der Punkte auf einer solchen Kurve. Dabei ist vor allem die Berechnung von Vielfachen eines Punktes $[m]P = Q$ wichtig. Diese Berechnung kann mit Hilfe verschiedener Algorithmen, die in diesem Kapitel vorgestellt und untersucht werden sollen, sehr effizient durchgeführt werden. Die Schreibweise der Vielfachenbildung $[m]P = Q$ erinnert stark an die Schreibweise der Multiplikation. Aus diesem Grund wird die Vielfachenbildung auch häufig als Punktmultiplikation bezeichnet. Diese Arbeit bildet dabei keine Ausnahme.
Die Umkehrung einer solchen Punktmultiplikation, also das Berechnen des Faktors m, ist im allgemeinen sehr schwierig und wird auch das *Diskrete Logarithmusproblem einer Elliptischen Kurve (ECDLP)*[1] genannt.
Für spezielle Elliptische Kurven, sogenannte *supersinguläre Kurven*, gibt es Methoden dieses Problem zu vereinfachen, indem man es auf das Diskrete Logarithmus Problem in einem endlichen Körper abbildet (siehe dazu [MOV93], [Wer02] und [BSS99]). Solche Kurven werden aus diesem Grund in kryptographischen Anwendungen nicht (mehr) verwendet.

Die untersuchten Algorithmen für die Punktmultiplikation sind den Methoden für schnelles Potenzieren einer Zahl, sogenannten *square and multiply* Methoden, sehr ähnlich und werden demzufolge oft als *double and add* Verfahren bezeichnet. Es existieren verschieden Optimierungen und spezielle Anpassungen, welche in diesem Kapitel beschrieben werden sollen.

5.1 Verschiedene Koordinatendarstellungen

Ein wichtiges Merkmal sind die in einem Verfahren verwendeten Koordinaten. Wie man im vorherigen Kapitel gesehen hat, gibt es affine und projektive Koordinaten um die Punkte darzustellen.

[1] Englisch: Elliptic Curve Discrete Logarithm Problem

Der Vorteil der projektiven Koordinaten liegt darin, dass auf die kostspielige Operation der Invertierung eines Elements in $\mathbb{F}_{2^n}$ verzichtet werden kann. Jedoch sind im projektiven wesentlich mehr Multiplikationen von Körperelementen notwendig, weshalb es vom Verhältnis zwischen Invertierung und Multiplikation abhängt, mit welchen Koordinaten die Berechnung schneller erfolgt.
Für die projektiven Koordinaten gibt es noch zwei weitere Darstellungen, welche durch Koordinatentransformation erzeugt werden können. Die *López-Dahab* und *Jakobischen* projektiven Koordinaten benötigen im Vergleich zu den einfachen projektiven Koordinaten noch weniger Multiplikationen.

Additionsformeln für die Punktaddition und Verdopplung mit affinen Koordinaten wurden bereits in Kapitel 4.2.3 eingeführt und bewiesen, weshalb an dieser Stelle darauf verwiesen wird. Für weitere Vergleiche in dieser Arbeit werden affine Koordinaten mit $\mathcal{A}$ bezeichnet.
Die einzelnen Algorithmen, sowie die verschiedenen Koordinatendarstellungen werden durch die Anzahl der Körperoperationen in $\mathbb{F}_{2^n}$ verglichen. Dabei wird eine Invertierung durch I, eine Multiplikation mit M und eine Quadrierung mit S abgekürzt.

5.1.1 Projektive Koordinaten

In diesem Kapitel werden drei mögliche projektive Koordinatendarstellungen vorgestellt und verglichen, um letztlich eine Darstellung für die Verwendung in dieser Arbeit auszuwählen. Für alle diese Darstellungen werden sowohl die Kurvengleichung, als auch die entsprechenden Additionsformeln angegeben, jedoch wird auf die Herleitung, sowie Beweise nahezu vollständig verzichtet. Dazu sei verwiesen auf [Lan04], [LD99b], [HMV04, Kapitel 3], [CF05] und [BSS99].
Für alle folgenden Formeln gilt die Voraussetzung das mit Körpern $\mathbb{F}_{2^n}$ gearbeitet wird.

Standard projektive Koordinaten $\mathcal{P}$

Darunter versteht man die projektiven Koordinaten im allgemeinsten Fall, so wie sie in Kapitel 4.1.2 eingeführt wurden. Die Umrechnung von einem affinen Punkt in einen projektiven und umgekehrt ist folgendermaßen erklärt:

$$\begin{array}{rcl} \mathcal{A} & & \mathcal{P} \\ (x,y) & \rightarrow & [X:Y:1] \\ \left(\dfrac{X}{Z},\dfrac{Y}{Z}\right) & \leftarrow & [X:Y:Z], \quad Z \neq 0. \end{array}$$

Der unendlich ferne Punkt $\mathcal{O}$ hat in dieser Darstellung die Koordinaten $\mathcal{O} = [0:1:0]$. Für das Negative eines Punktes $P = [X:Y:Z]$ ergibt sich $-P = [X, Y+X, Z]$. Die Gleichung einer Elliptischen Kurve mit standard projektiven Koordinaten ist bereits aus dem vorhergehenden Kapitel bekannt und schreibt sich

$$\mathcal{E} : Y^2Z + XYZ = X^3 + aX^2Z + bZ^3.$$

Die Additionsformeln mit diesen Koordinaten ergeben sich für zwei Punkte $P_1 = [X_1 : Y_1 : Z_1], P_2 = [X_2 : Y_2 : Z_2]$ für die gilt $P_1 \neq \pm P_2$ und $P_1 + P_2 = Q = [X_3 : Y_3 : Z_3]$ zu

$$\begin{aligned} A &= Y_1Z_2 + Z_1Y_2, & B &= X_1Z_2 + Z_1X_2, & C &= B^2, \\ D &= Z_1Z_2, & E &= (A^2 + AB + aC)D + BC, & & \\ X_3 &= BE, & Y_3 &= C(AX_1 + Y_1B)Z_2 + (A+B)E, & Z_3 &= B^3D. \end{aligned}$$

Sei $P = [X_1 : Y_1 : Z_1]$ dann ist $[2]P = Q = [X_3 : Y_3 : Z_3]$ gegeben durch

$$\begin{aligned} A &= X_1^2, & B &= A + Y_1Z_1, & C &= X_1Z_1, \\ D &= C^2, & E &= B^2 + BC + aD, & & \\ X_3 &= CE, & Y_3 &= (B+C)E + A^2C, & Z_3 &= CD. \end{aligned}$$

Somit ergibt sich für eine allgemeine Addition ein Aufwand von $16M + 2S$, während die Verdopplung eines Punktes nur $8M + 4S$ benötigt. Verwendet man für die Addition gemischte Koordinaten, d.h. ein Punkt ist in affinen Koordinaten gegeben (z.B. $P_1 = [X : Y : 1]$), so werden nur noch $12M + 2S$ Operationen benötigt. Wählt man $a \in \{0, 1\}$, so verringern sich die benötigten Multiplikationen für die allgemeine Punktaddition sowie die Punktverdopplung um jeweils eine Operation.

Jakobische projektive Koordinaten $\mathcal{J}$

Die Jakobischen Koordinaten werden in manchen Büchern (z.B. [BSS99]) auch als gewichtete projektive Koordinaten bezeichnet. Der unendlich ferne Punkt $\mathcal{O}$ hat in dieser Form die Koordinaten $\mathcal{O} = [1 : 1 : 0]$ und für das Negative eines Punktes $P = [X : Y : Z]$ gilt $-P = [X : ZX + Y : Z]$. Für die Umrechnung von affinen in jakobische Koordinaten gilt das folgende

$$\begin{array}{ccc} \mathcal{A} & & \mathcal{J} \\ (x, y) & \rightarrow & [X : Y : 1] \\ \left(\frac{X}{Z^2}, \frac{Y}{Z^3}\right) & \leftarrow & [X : Y : Z]. \end{array}$$

Die Kurvengleichung für $\mathcal{E}$ in jakobischen Koordinaten lässt sich schreiben als

$$\mathcal{E} : Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^6.$$

Seien $P_1 = [X_1 : Y_1 : Z_1]$ und $P_2 = [X_2 : Y_2 : Z_2]$ für die gilt $P_1 \neq \pm P_2$, dann berechnet sich $P_1 + P_2 = Q = [X_3 : Y_3 : Z_3]$ folgendermaßen

$$\begin{aligned} A &= X_1Z_2^2, & B &= X_2Z_1^2, & C &= Y_1Z_2^3, \\ D &= Y_2Z_1^3, & E &= A + B, & F &= C + D \\ G &= EZ_1, & H &= FX_2 + GY_2, & Z_3 &= GZ_2, \\ I &= F + Z_3 & X_3 &= aZ_3^2 + FI + E^3, & Y_3 &= IX_3 + G^2H. \end{aligned}$$

Sei $P = [X_1 : Y_1 : Z_1]$ dann ist $[2]P = Q = [X_3 : Y_3 : Z_3]$ gegeben durch

$$\begin{aligned} A &= X_1^2, & B &= A^2, & C &= Z_1^2, \\ X_3 &= B + bC^4, & Z_3 &= X_1C, & Y_3 &= BZ_3 + (A + Y_1Z_1 + Z_3)X_3. \end{aligned}$$

Die Addition zweier Punkte in jakobischen Koordinaten benötigt im allgemeinen Fall $15M + 5S$ Operationen. Wenn einer der beiden Punkte in affinen Koordinaten dargestellt ist (z.B. $P_1 = [X : Y : 1]$), so reduziert es sich auf $11M + 4S$. Wenn zusätzlich $a \in \{0, 1\}$ wird noch eine Multiplikation eingespart. Die Verdopplung $[2]P$ benötigt $5M + 5S$ Körperoperationen.

López-Dahab projektive Koordinaten $\mathcal{LD}$

Diese Form der projektiven Koordinaten wurde in einer Arbeit von López und Dahab [LD99b] vorgestellt und auch nach ihnen benannt. In [Lan04] wird noch eine weitere Verbesserung der Additionsformeln für diese Koordinaten vorgeschlagen.
Der Punkt $\mathcal{O}$ hat in dieser Darstellung die Koordinaten $\mathcal{O} = [1 : 0 : 0]$ und das Inverse von $P = [X : Y : Z]$ berechnet sich mit $-P = [X : ZX + Y : Z]$.
Das Umrechnen zwischen affinen und López-Dahab Koordinaten erfolgt nach folgenden Vorschriften

$$\begin{array}{rcl} \mathcal{A} & & \mathcal{LD} \\ (x, y) & \rightarrow & [X : Y : 1] \\ \left(\frac{X}{Z}, \frac{Y}{Z^2}\right) & \leftarrow & [X : Y : Z]. \end{array}$$

Die Kurvengleichung schreibt sich in diesen Koordinaten mit

$$\mathcal{E} : Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4.$$

Seien $P_1 = [X_1 : Y_1 : Z_1]$ und $P_2 = [X_2 : Y_2 : Z_2]$ für die gilt $P_1 \neq \pm P_2$, dann berechnet sich $P_1 + P_2 = Q = [X_3 : Y_3 : Z_3]$ folgendermaßen

$$\begin{array}{lll} A_0 = Y_2 Z_1^2, & A_1 = Y_1 Z_2^2, & B_0 = X_2 Z_1, \\ B_1 = X_1 Z_2, & C = A_0 + A_1, & D = B_0 + B_1, \\ E = Z_1 Z_2, & F = DE, & Z_3 = F^2, \\ G = D^2(F + aE^2) & H = CF, & X_3 = C^2 + H + G, \\ I = D^2 B_0 E + X_3, & J = D^2 A_0 + X_3, & Y_3 = HI + Z_2 J. \end{array}$$

Sei $P = [X_1 : Y_1 : Z_1]$ dann ist $[2]P = Q = [X_3 : Y_3 : Z_3]$ gegeben durch

$$Z_3 = X_1^2 Z_1^2, \qquad X_3 = X_1^4 + bZ_1^4, \qquad Y_3 = bZ_1^4 Z_3 + X_3(aZ_3 + Y_1^2 + bZ_1^4)$$

Es ergibt sich für die Addition ein Bedarf von $14M + 6S$ Operationen. In [CF05, Kapitel 13.3] ist eine von Higuchi und Takagi bewiesene allgemeine Addition beschrieben, die nur $13M + 4S$ Operationen benötigt. Die Punktverdopplung verwendet im allgemeinen $5M + 5S$ Körperoperationen. Für $a \in \{0, 1\}$, wird noch eine Multiplikation eingespart.
Die Verwendung von gemischten Koordinaten, für die allgemeine Punktaddition, ist in diesem Falle sehr effektiv und benötigt nur $9M + 5S$ Operationen. Gilt weiterhin für $a \in \{0, 1\}$, so werden nur noch $8M + 5S$ benötigt. Dies stellt eine von Verbesserung des in [LD99b] publizierten Verfahrens dar, welches $10M + 3S$ benötigt und wurde von Al-Daoud et al. [ADMRK02] gezeigt.

Sei $P_2 = [X_2 : Y_2 : 1]$ der affine Punkt, dann ergeben sich für die Addition zweier Punkte in gemischten Koordinaten folgende Formeln

$$A = Y_1 + Y_2 Z_1^2, \quad B = X_1 + X_2 Z_1, \quad C = BZ_1,$$
$$Z_3 = C^2, \quad D = X_2 Z_3, \quad X_3 = A^2 + C(A + B^2 + aC),$$
$$Y_3 = (D + X_3)(AC + Z_3) + (Y_2 + X_2) Z_3^2.$$

5.1.2 Vergleich der Darstellungen

Die im vorherigen beschriebenen Koordinatendarstellungen sollen hier noch einmal kurz zusammengestellt und verglichen werden. Tabelle 5.1 listet dazu nochmals alle vier betrachteten Darstellungen mit den entprechenden Kurvengleichungen sowie der Koordinaten für den unendlich fernen Punkt $\mathcal{O}$ auf.

In Tabelle 5.2 sind die für die Punktaddition und -verdopplung notwendigen Körper-

Koordinaten-system	Kurvengleichung für $\mathcal{E}$	$\mathcal{O}$
$\mathcal{A}$	$y^2 + xy = x^3 + ax^2 + b$	–
$\mathcal{P}$	$Y^2Z + XYZ = X^3 + aX^2ZbZ^3$	$[0:1:0]$
$\mathcal{J}$	$Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^6$	$[1:1:0]$
$\mathcal{LD}$	$Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4$	$[1:0:0]$

Tabelle 5.1: *Gegenüberstellung der verschiedenen Koordinatensysteme*

operationen in $\mathbb{F}_{2^n}$ zusammengefasst. Dabei sind jeweils die Fälle für $a \in \{0, 1\}$ aufgeführt. Die Abkürzung I steht für die Operation einer Körperinvertierung. Die Operation S für die Quadrierung eines Körperelements ist nicht mit in die Tabelle aufgenommen, da sie wesentlich schneller als eine Multiplikation ausgeführt wird und daher nur sehr geringen Einfluss auf die Laufzeit hat.

Wie man aus Tabelle 5.2 erkennen kann, benötigt man für die affine Berechnung des

Koordinaten-system	Addition	Addition (gemischte Koordinaten)	Verdopplung
$\mathcal{A}$	$I + 2M$	–	$I + 2M$
$\mathcal{P}$	$15M$	$12M$	$7M$
$\mathcal{J}$	$14M$	$10M$	$5M$
$\mathcal{LD}$	$13M$	$8M$	$4M$

Tabelle 5.2: *Vergleich des Bedarf von Körperoperationen für Addition und Verdopplung von Punkten in verschiedenen Koordinatensystemen ($a \in \{0, 1\}$)*

neuen Punktes deutlich weniger Körperoperationen als für das projektive Äquivalent. Jedoch ist die Berechnung des Inversen eines Elements eine relativ „teuere“ Operation, wie in Kapitel 3.2.5 gezeigt wurde. Es lässt sich nun sehr einfach feststellen, ab welchem Verhältnis zwischen Invertierung und Multiplikation $\frac{I}{M}$ der Übergang in die projektive Darstellung Effizienzvorteile bringt. Multiplikation bedeutet in diesem Zusammenhang neben der reinen Multiplikation von Elementen aus $\mathbb{F}_{2^n}$, auch die anschließende Reduktion des Ergebnis.

Im schlechtesten projektiven Fall, einer Darstellung in $\mathcal{P}$, muss gelten $15M < I + 2M$ damit der Übergang Vorteile bringt. Somit folgt, dass bereits ab einem Verhältnis von $\frac{I}{M} \geq 13$ eine Berechnung im projektiven Vorteile hat.
Rechnet man mit gemischten Koordinaten ($\mathcal{LD} + \mathcal{A}$), so reduziert sich dieses Verhältnis $\frac{I}{M} \geq 6$ sogar auf auf die Hälfte.
Für die Verdopplung eines Punktes in $\mathcal{LD}$-Koordinaten bestimmt sich das Verhältnis mit $\frac{I}{M} \geq 2$.

5.2 Algorithmen zur Punktmultiplikation

In diesem Abschnitt werden verschiedene Möglichkeiten zur Punktmultiplikation vorgestellt. Dabei wird im Besonderen auf die Unterschiede zwischen affinen und projektiven Verfahren, sowie Algorithmen mit gemischten Koordinaten eingegangen. Die vorgestellten und untersuchten Algorithmen sind aus [HMV04], [BSS99], [Sol97], [Gor98], [JT01], [MO90], [LD99b] und [HHM01] entnommen, wo sich noch tiefergehende Erklärungen dazu finden lassen.
Für diese Arbeit wurden nur Algorithmen implementiert, die für beliebige Punkte $P \in \mathcal{E}$ verwendet werden können. Spezielle Algorithmen, die von einem festen Basispunkt ausgehen, wurden nicht betrachtet.

5.2.1 Möglichkeiten der Punktaddition

Ein Großteil der im nächsten Kapitel vorgestellten Verfahren zur Punktmultiplikation funktioniert nach dem *double and add* Prinzip. Das Vielfache eines Punktes $[k]P$ wird dabei, abhängig vom betrachteten Koeffizienten des Faktors k, entweder nur Verdoppelt oder Verdoppelt und zusätzlich zum ursprünglichen Punkt addiert.
Aus diesem Grund ist es nötig zuerst Algorithmen für die Punktaddition sowie Punktverdopplung zu implementieren.

Die theoretischen Ergebnisse aus dem vorherigen Abschnitt führen dazu, dass für diese Algorithmen die López-Dahab Koordinatendarstellung ($\mathcal{LD}$) bzw. eine gemischte $\mathcal{LD}$-$\mathcal{A}$ Darstellung verwendet wird. Zum Vergleich wurde auch eine Variante mit rein affinen Koordinaten implementiert.
Die expliziten Algorithmen zur Punktaddition und -verdopplung in affinen ($\mathcal{A}$) und projektiven López-Dahab ($\mathcal{LD}$) Koordinaten werden hier nicht näher beschrieben, da sie genau den Formeln zur Addition und Verdopplung von Punkten in dem jeweiligen Koordinatensystem entsprechen. In den vorangegangenen Abschnitten wurden diese Formeln bereits detailliert vorgestellt, weshalb hier darauf verwiesen wird.

Der aus [HMV04] entnommene Algorithmus 5.2.1 zur Addition zweier Punkte $P \oplus Q$, mit $P = [X_1 : Y_1 : Z_1]$ in $\mathcal{LD}$ Koordinaten und $Q = (x_2, y_2)$ in affinen Koordinaten, ist im folgenden dargestellt. Für die Kurvengleichung $y^2 + xy = x^3 + ax^2 + b$ gilt dabei die Einschränkung $a \in \{0, 1\}$.
Für die Timing- und Resourcenvergleiche in dieser Arbeit wurden nur NIST-Kurven herangezogen. Diese haben allesamt die Eigenschaft $a = 1$, weshalb obige Einschränkung

für diese Arbeit keine Folgen hat.
Auch in kryptographischen Anwendungen die NIST-Kurven verwenden, bleibt die Einschränkung ohne Auswirkungen.

Algorithmus 5.2.1 Punktaddition in gemischten $\mathcal{LD}$-$\mathcal{A}$ Koordinaten ($y^2 + xy = x^3 + ax^2 + b, a \in \{0, 1\}$)

INPUT: $P = [X_1 : Y_1 : Z_1]$ in $\mathcal{LD}$ Koordinaten, $Q = (x_2, y_2)$ in affinen Koordinaten.
OUTPUT: $P \oplus Q = [X_3 : Y_3 : Z_3]$ in $\mathcal{LD}$ Koordinaten.

1: **if** $Q = \mathcal{O}$ **then return** P
2: **if** $P = \mathcal{O}$ **then return** $[x_2 : y_2 : 1]$ ▷ Rückgabe in $\mathcal{LD}$
3: $T_1 \leftarrow Z_1 \cdot x_2;\quad T_2 \leftarrow Z_1^2;$
4: $X_3 \leftarrow X_1 + T_1;\quad T_1 \leftarrow Z_1 \cdot X_3;$
5: $T_3 \leftarrow T_2 \cdot y_2;\quad Y_3 = Y_1 + T_3;$
6: **if** $X_3 = 0$ **then**
7:     **if** $Y_3 = 0$ **then**
8:         berechne $[X_3 : Y_3 : Z_3] = [2][x_2 : y_2 : 1]$
9:         **return** $[X_3 : Y_3 : Z_3]$
10:     **else**
11:         **return** $\mathcal{O}$
12:     **end if**
13: **end if**
14: $Z_3 \leftarrow T_1^2;\quad T_3 \leftarrow T_1 \cdot Y_3;$
15: **if** a=1 **then** $T_1 \leftarrow T_1 + T_2;$
16: $T_2 \leftarrow X_3^2;\quad X_3 \leftarrow T_2 \cdot T_1;$
17: $T_2 \leftarrow Y_3^2;\quad X_3 \leftarrow X_3 + T_2;$
18: $X_3 \leftarrow X_3 + T_3;\quad T_2 \leftarrow x_2 \cdot Z_3;$
19: $T_2 \leftarrow T_2 + X_3;\quad T_1 \leftarrow Z_3^2;$
20: $T_3 \leftarrow T_3 + Z_3;\quad Y_3 \leftarrow T_3 \cdot T_2;$
21: $T_2 \leftarrow x_2 + y_2;\quad T_3 \leftarrow T_1 \cdot T_2;$
22: $Y_3 \leftarrow Y_3 + T_3;$
23: **return** $[X_3 : Y_3 : Z_3]$

Um den Speicherbedarf der im kommenden Abschnitt betrachteten Algorithmen zur Punktmultiplikation bestimmen zu können, ist es nötig, auch den RAM-Bedarf für die Punktaddition und -verdopplung zu kennen. Alle diese Algorithmen verwenden die Funktionen zur Körperarithmetik in $\mathbb{F}_{2^n}$, weshalb deren Speicherbedarf natürlich auch mit in die Betrachtungen für Punktaddition und -verdopplung mit eingeht. Abhängig vom Grad des irreduziblen Polynoms f variiert der RAM-Bedarf, weshalb alle folgenden Vergleiche für das irreduzible Polynom $f = x^{233} + x^{74} + 1$ vom Grad 233 aufgeführt sind. In Tabelle 5.3 ist der Bedarf an temporärem Speicher für die verschiedenen implementierten Algorithmen zur Punktaddition angegeben. Dabei wurde eine Registerbreite von 16 Bit angenommen.

Verwendet man eine 32 Bit Implementierung, so erhöht sich der RAM-Bedarf des Quadrier-Algorithmus auf 520 Byte. Außerdem wurde als Reduce-Algorithmus die angepasste Variante (Algorithmus 3.2.9) gewählt. Diese hat neben der Laufzeitvorteile auch noch den Vorteil, dass sie keinen temporären Speicher benötigt. Soll eine

Algorithmus	**RAM Bedarf lokal**	Quadrieren	Multiplizieren	Inversenbildung	Punktverdopplung $\mathcal{LD}$	Punktverdopplung $\mathcal{LD}$-$\mathcal{A}$	**Gesamtbedarf**
RAM-Bedarf des Primitiv		40	512	150			
Addition $\mathcal{A}$	210	–	–	–	$\oplus$	–	872
Verdopplung $\mathcal{A}$	150	$\oplus$	$\oplus$	$\oplus$	–	–	662
Addition $\mathcal{LD}$	360	–	–	–	–	$\oplus$	984
Verdopplung $\mathcal{LD}$	120	$\oplus$	$\oplus$	$\oplus$	–	–	624
Addition $\mathcal{LD}$-$\mathcal{A}$	240	–	–	–	–	$\oplus$	864

Tabelle 5.3: *Bedarf an temporärem Speicher in Bytes, für die Punktaddition und -verdopplung bei 16 Bit Registerbreite und einem speziellen Reduce-Algorithmus für ein Reducepolynom vom Grad 233. Der RAM-Bedarf setzt sich aus den temporären Variablen (lokal) und dem benötigten Speicher für die Algorithmen der Körperarithmetik zusammen. $\oplus$ zeigt welche Funktionen jeweils benötigt werden, wobei nur der maximale RAM-Bedarf dieser Funktionen in den Gesamtbedarf mit eingeht.*

allgemeine Reduce-Routine verwendet werden, so schlägt diese, abhängig von der Registerbreite W, mit $60 \cdot W$ Bytes zu buche.
Als Multiplikations-Routine wurde der *Left-to-Right comb Window* Algorithmus aus Kapitel 3.2.2 gewählt. Dieser bietet die beste Performance der untersuchten Multiplikations-Algorithmen, benötigt jedoch in der implementierten Variante 512 Bytes RAM für eine Windowgröße $w = 4$. Um für die Punktmultiplikationen die Laufzeiten so gering als möglich zu halten, wurde er trotz dieses Nachteils in allen folgenden Algorithmen eingesetzt.

5.2.2 Affine Punktmultiplikation

Die Grundlage für die Mehrzahl der folgenden Verfahren zur Vielfachenbestimmung $[k]P$ eines Punktes $P \in \mathcal{E}$, sind die soeben beschriebenen Funktionen zur Punktaddition und -verdopplung. Das erste Beispiel für einen solchen Algorithmus ist der *Right-to-left binary* Algorithmus für die Punktmultiplikation. Dieser berechnet das Vielfache des Punktes $Q = [k]P$ nach dem *double and add* Prinzip. Man kann dies am einfachsten mit der *quadriere und multipliziere* Methode zur Berechnung großer Potenzen $x^n \bmod p$ vergleichen. Abhängig vom jeweiligen Koeffizienten der Binärdarstellung von $n = (n_i n_{i-1} \dots n_1 n_0)_2$ wird x quadriert und evtl. noch mit dem Zwischenergebnis multipliziert. Dabei kann nach jedem Zwischenschritt reduziert werden. Durch diese Art der Multiplikation werden die Zwischenergebnisse nicht sehr groß und lassen sich einfach handhaben. Folgendes Beispiel soll dies noch einmal vergegenwärtigen.

Beispiel 5.2.1. *Gesucht sei* $5^{37} \bmod 13$. *Binärdarstellung des Exponenten ergibt* $37 =$

100101_2. *Es lässt sich nun schreiben*

$$5^{100101} \bmod 13.$$

Für jede Eins des Exponenten muss die Basis sowohl quadriert als auch multipliziert werden. Im Falle der Null wird nur quadriert. Damit ergibt sich

$$(((((5)^2)^2)^2 5)^2)^2 5 \bmod 13.$$

Dies lässt sich leicht ausrechnen, da eine Reduktion modulo 13 *in jedem Zwischenschritt erfolgen kann und ergibt letztlich* 5 mod 13*.*

Analog funktioniert Algorithmus 5.2.2 für die Punktmultiplikation. Durch diese Methode ist es möglich, die eigentliche Punktmultiplikation auf einfachere Punktadditionen und Punktverdopplungen zu reduzieren. Der Name *Right-to-left binary* rührt daher, dass der Faktor k von $[k]P$ binär dargestellt wird und von rechts nach links, also vom niederwertigsten zum höchstwertigen Bit, durchlaufen wird. Abhängig vom Koeffizienten k_i aus der Binärdarstellung von k wird der Punkt P entweder nur verdoppelt oder zusätzlich noch zum aktuellen Zwischenergebnis addiert (Zeile 3 und 4).

Algorithmus 5.2.2 Right-to-left binary Methode zur Punktmultiplikation

INPUT: $k = (k_{m-1} \dots k_1 k_0)_2, \quad P \in \mathcal{E}$
OUTPUT: $Q = [k]P$
1: $Q \leftarrow \mathcal{O}$ ▷ Initialisierung
2: **for** $i \leftarrow 0$ **to** $m-1$ **do**
3: **if** $k_i = 1$ **then** $Q \leftarrow Q \oplus P$
4: $P \leftarrow [2]P$ ▷ Verdopple P
5: **end for**
6: **return** Q

Der *Right-to-left binary* Algorithmus benötigt in der implementierten Variante zwei temporäre affine Punkte zur Speicherung der Zwischenergebnisse.
Eine Variante dieses Algorithmus ist die Möglichkeit, den Exponenten von links nach rechts, somit vom höchstwertigen zum niederwertigsten Bit, zu durchlaufen. Dies ist in Algorithmus 5.2.3 dargestellt, wobei der Punkt $Q_{\mathcal{LD}}$ sowohl affin als auch projektiv dargestellt werden kann. Im nächsten Abschnitt wird darauf noch näher eingegangen.

5.2.3 Punktmultiplikation mit Projektiven Koordinaten

Wie in Kapitel 5.1.2 bereits theoretisch gezeigt, macht die Verwendung von projektiven Koordinaten nur dann Sinn, wenn die Körperoperation der Inversenberechnung wesentlich langsamer erfolgt als die der Multiplikation.
Der *Left-to-right binary* Algorithmus macht sich genau diese Eigenschaft zunutze. Primär ist dieser zunächst sehr ähnlich dem Algorithmus 5.2.2, mit dem Unterschied, dass der Exponent von links nach rechts durchlaufen wird. Zusätzlich bietet sich noch eine zweite Änderung an, nämlich die Nutzung von $\mathcal{LD}$ Koordinaten. Dies liegt an der Tatsache, dass der Punkt P, dessen Vielfaches $Q = [k]P$ berechnet wird, während

des gesamten Algorithmus nicht verändert wird. Somit lässt sich die Punktaddition in Zeile 3 mit gemischten Koordinaten durchführen. Dazu verwendet man nun den vorher beschriebenen Algorithmus 5.2.1. Der Punkt Q liegt während der Abarbeitung in $\mathcal{LD}$ Koordinaten vor und wird deshalb mit $Q_{\mathcal{LD}}$ bezeichnet.

Algorithmus 5.2.3 Left-to-right binary Methode zur Punktmultiplikation

INPUT: $k = (k_{m-1} \dots k_1 k_0)_2, \quad P \in \mathcal{E}$
OUTPUT: $Q_{\mathcal{LD}} = [k]P$
1: $Q_{\mathcal{LD}} \leftarrow \mathcal{O}$ ▷ Initialisierung
2: **for** $i \leftarrow m-1$ **to** 0 **do**
3: $Q_{\mathcal{LD}} \leftarrow [2]Q_{\mathcal{LD}}$ ▷ Verdopple Q
4: **if** $k_i = 1$ **then** $Q_{\mathcal{LD}} \leftarrow Q_{\mathcal{LD}} \oplus P$
5: **end for**
6: **return** $Q_{\mathcal{LD}}$

Der Bedarf an temporärem lokalem Speicher liegt für diesen Algorithmus mit gemischten $\mathcal{LD}$-$\mathcal{A}$ Koordinaten bei zwei projektiven Punkten. Es spielt in diesem Zusammenhang keine Rolle, ob es sich bei den verwendeten projektiven Koordinaten um $\mathcal{LD}$ Koordinaten handelt, da der benötigte Speicher für alle drei projektiven Typen gleich ist.

In Algorithmus 5.2.3 wird die Punktaddition in Zeile 4 abhängig von k_i durchgeführt. Für die Binärdarstellung der Länge t von k ergibt sich die Anzahl der Koeffizienten $k_i = 1$ im Mittel zu $t/2$. Die Länge des Faktors k von $[k]P$ lässt sich abschätzen zu $m < \deg(f)$, dem Grad des für die Körperdarstellung benötigten irreduziblen Polynoms f. Für die Laufzeit des *Left-to-Right binary* Algorithmus lässt sich nun die folgende Abschätzung angeben

$$\frac{m}{2}A + mD,$$

wobei A für die Punktadditionen und D für die Verdopplungen steht. Verwendet man rein affine Koordinaten, so ergibt sich die Laufzeit, ausgedrückt in Körpermultiplikationen (M) und Invertierungen (I), mit

$$3mM + 1.5mI.$$

Rechnet man wie in Algorithmus 5.2.3 mit gemischten Koordinaten, so ergibt sich für die gemischte Punktaddition eine Laufzeit von $A = 8M$ und für die Punktverdopplung mit $\mathcal{LD}$ Koordinaten ergibt sich $D = 4M$. Somit ergibt sich insgesamt eine Laufzeit von

$$8mM + (2M + I) \tag{5.1}$$

Körperoperationen. Der Term $2M + I$ bezeichnet den Aufwand um den resultierenden $\mathcal{LD}$ Punkt in affine Koordinaten zu konvertieren. Verwendet man anstatt gemischter $\mathcal{A}$-$\mathcal{LD}$ Koordinaten für die Punktaddition ebenfalls reine $\mathcal{LD}$ Koordinaten, so werden

$$10.5mM + (2M + I) \tag{5.2}$$

Körperoperationen benötigt. Die Verwendung von gemischten $\mathcal{A}$-$\mathcal{LD}$ Koordinaten führt damit theoretisch zu einem Laufzeitvorteil von ca. 24%.

5.2.4 Verwendung der Non-Adjacent Form (NAF) zur Punktmultiplikation

Die Laufzeit der beiden Algorithmen 5.2.3 und 5.2.2 hängt, trotz Verbesserungen durch die Einführung gemischter Koordinaten, unter anderem von der Anzahl der Punktadditionen ab. Diese ist wiederum abhängig von der Anzahl der Koeffizienten $k_i = 1$ in der Binärdarstellung des Faktors k von $[k]P$.
Durch eine speziellere Repräsentation von k verringern die folgenden Algorithmen die Anzahl an Punktadditionen.

Die Subtraktion eines Punktes $P \in \mathcal{E}$ entspricht der Addition mit dem Negativen $\ominus P$ von P. Das Negative eines Punktes $P = [X_1 : Y_1 : Z_1]$ in $\mathcal{LD}$ Koordinaten lässt sich mit $\ominus P = [X_1 : Z_1 X_1 + Y_1 : Z_1]$ sehr einfach bestimmen, wodurch die Subtraktion nahezu die gleichen Laufzeiteigenschaften aufweist wie die Addition.
Durch die Möglichkeit Punkte auch subtrahieren zu können, kann die Punktmultiplikation $[k]P$ auch noch anders gelöst werden. Der Faktor k lässt sich mit Hilfe von sogenannten *addition-subtraction chains* darstellen. Diese Darstellung besteht aus den drei Elementen $\{1, 0, -1\}$, wobei -1 im folgenden als $\bar{1}$ bezeichnet wird, um die Lesbarkeit zu erhöhen.
Formal geschrieben ergibt sich für eine beliebige Ganzzahl $k \in \mathbb{N}$ nun folgende Darstellung:

$$k = \sum_{j=0}^{l} s_j 2^j, s_j \in \{1, 0, \bar{1}\}.$$

Dies bezeichnet man im allgemeinen als (binäre) *signed digit* Darstellung. Wie leicht einzusehen ist, beinhaltet diese Art der Darstellung auch die reine Binärdarstellung einer Zahl k. Insgesamt können alle Zahlen $k, 0 \leq k \leq 2^{l+1} - 1$ sowie deren Negative dargestellt werden. Folgendes Beispiel soll die verschiedenen Möglichkeiten der *signed Digit* Darstellung verdeutlichen.

Beispiel 5.2.2. *Gegeben sei die Zahl* $k = 18$ *und deren Binärdarstellung* $k_2 = 10010$. *Mögliche signed Digit Darstellungen sind z.B.* $18 = 101\bar{1}0 = 16 + 4 - 2$, $18 = 11\bar{1}\bar{1}0 = 16 + 8 - 4 - 2$ *oder auch* $18 = 10\bar{1}\bar{1}\bar{1}0 = 32 - 8 - 4 - 2$. *Wie man sieht, ergeben sich viele Redundanzen für diese Art der Darstellung.*

Die Anzahl der Punktadditionen in den bisher untersuchten Punktmultiplikations-Algorithmen hängt davon ab, wie viele Koeffizienten $k_i = 1$ in der Binärdarstellung von k auftreten. Um nun die Anzahl an Punktadditionen zu veringern, sucht man nach einer Darstellung für k die möglichst wenige Koeffizienten $k_i \neq 0$ enthält. Eine solche ist die sogenannte *Non-Adjacent Form*, die sich folgendermaßen definiert.

Definition 5.2.3. *Eine Non-Adjacent Form (NAF) einer positiven Ganzzahl* $k \in \mathbb{N}$ *ist ein Ausdruck der Form* $k = \sum_{j=0}^{l-1} k_j 2^j, k \in \{1, 0, \bar{1}\}$ *und* $k_{l-1} \neq 0$, *wobei für zwei aufeinanderfolgende Bits gilt* $k_j \cdot k_{j+1} = 0$, *d.h. es dürfen nicht beide ungleich Null sein. Die Länge der NAF ist* l.

Die wichtigsten Eigenschaften der NAF sind in folgendem Theorem zusammengetragen. Die dazugehörigen Beweise finden sich u.a. in [Rei60] und [MO90].

Theorem 5.2.4. *Sei $k \in \mathbb{N}$ dann gilt folgendes:*

(I) k besitzt eine eindeutige NAF, bezeichnet als NAF(k).

(II) NAF(k) besitzt die wenigsten von 0 verschiedenen Bits aller signed digit *Darstellungen von k.*

(III) Die Länge von NAF(k) ist höchstens um ein Bit länger als die Binärdarstellung von k.

(IV) Die durchschnittliche Anzahl von Bits $k_i \neq 0$ einer NAF der Länge l beträgt $l/3$.

Eine Möglichkeit die NAF einer beliebigen Zahl $k \in \mathbb{N}$ zu erhalten, stellt ein Algorithmus von Reitwiesner [Rei60] dar. Dieses Verfahren berechnet die NAF(k) von rechts nach links aus der Binärentwicklung von k. Im folgenden ist dieser Algorithmus 5.2.4 abgedruckt. In [JY00] findet sich ein Beweis für die NAF-Eigenschaft des Ergebnisses.

Algorithmus 5.2.4 NAF Erzeugung nach Reitwiesner

INPUT: $k = (k_{m-1} \dots k_1 k_0)_2$ ▷ Binärdarstellung von k
OUTPUT: NAF(k)= $k' = (k'_m k'_{m-1} \dots k'_1 k'_0)_{NAF}$
1: $k' \leftarrow 0$
2: $c_0 \leftarrow 0; k_{m+1} \leftarrow 0; k_m \leftarrow 0$ ▷ Initialisierung
3: **for** $i \leftarrow 0$ **to** m **do**
4: $c_{i+1} \leftarrow \lfloor (c_i + k_i + k_{i+1})/2 \rfloor$
5: $k'_i \leftarrow c_i + k_i - 2c_{i+1}$
6: **end for**
7: **return** k'

Die NAF von $k \in \mathbb{N}$ lässt sich auch bestimmen, indem die Binärdarstellung von k von links nach rechts abgearbeitet wird. Entsprechende Algorithmen und tiefergehende Erläuterungen finden sich in [HMV04], [JY00], [Sol97], [JT01] sowie [Gor98].
Wie bereits angemerkt, lassen sich mit Hilfe der NAF Darstellung die Algorithmen 5.2.3 und 5.2.2 dahingehend verbessern, dass weniger Punktadditionen benötigt werden. In Algorithmus 5.2.5 ist dies für eine Abarbeitung von links nach rechts gezeigt.

Algorithmus 5.2.5 Left-to-right NAF Punktmultiplikation

INPUT: $k = (k_{m-1} \dots k_1 k_0)_2, \quad P \in \mathcal{E}$
OUTPUT: $Q = [k]P$
1: $Q \leftarrow \mathcal{O}$
2: **for** $i \leftarrow l-1$ **to** 0 **do**
3: Berechne Koeffizient k_i der NAF(k)
4: $Q \leftarrow [2]Q$
5: **if** $k_i = 1$ **then** $Q \leftarrow Q \oplus P$
6: **if** $k_i = \bar{1}$ **then** $Q \leftarrow Q \ominus P$
7: **end for**
8: **return** Q

Um den Speicherbedarf gering zu halten, muss die NAF Darstellung des Faktors k von $[k]P$ möglichst effizient gespeichert werden. Aus diesem Grund muss für die Darstellung

von $\bar{1}$ innerhalb einer NAF eine geeignete Codierung gewählt werden, damit diese durch einen Binärstring dargestellt werden kann.
Verwendet man die intuitive Variante und codiert alle Koeffizienten k_i von NAF(k) mit zwei Bits, so resultiert eine NAF(k), die doppelt so lang ist wie die Binärdarstellung von k. Aus diesem Grund berechnet man die jeweiligen Bits einer NAF oftmals „on the fly" innerhalb der Multiplikationsalgorithmus, d.h. man berechnet aus der Binärdarstellung von k pro Schleifendurchlauf einen Koeffizienten der NAF und führt, abhängig davon, Punktaddition und Punktverdopplung durch.
In Algorithmus 5.2.5 ist eine solche „on the fly" Berechnung integriert (Zeile 3).
Soll die NAF zuerst komplett berechnet und gespeichert werden, so muss diese effizient codiert werden. Eine Möglichkeit der effizienten Codierung einer NAF wird in [JT01] beschrieben. Man macht sich die Eigenschaft der NAF zunutze, dass $k_i \cdot k_{i+1} = 0$ gilt, d.h. auf eine 1 oder $\bar{1}$ immer eine 0 folgen muss. Es wird die folgende *right-to-left* Codierung vorgeschlagen:

$$\mathcal{R} : \begin{cases} 01 & \mapsto & 01 \\ 0\bar{1} & \mapsto & 11 \\ 0 & \mapsto & 0 \end{cases} , \qquad \mathcal{R}^{-1} : \begin{cases} 01 & \mapsto & 01 \\ 11 & \mapsto & 0\bar{1} \\ 0 & \mapsto & 0 \end{cases}$$

Ein Beispiel soll dies noch einmal verdeutlichen.

Beispiel 5.2.5. *Die Zahl $k = 29$ ergibt sich als NAF(k)= $(100\bar{1}01)$ und wird durch die Verwendung von $\mathcal{R}$ codiert zu (101101). In diesem Fall erfolgt die Codierung von recht nach links.*

$$(\underline{1}\ \underline{0}\ \underline{0\bar{1}}\ \underline{01}) \mapsto (\underline{1}\ \underline{0}\ \underline{11}\ \underline{01})$$

Wendet man $\mathcal{R}^{-1}$ darauf an, so erhält man die NAF Darstellung zurück.

In [JT01] ist weiterhin auch noch eine *Left-to-right* Codierung der NAF vorgeschlagen. Der *Right-to-left NAF* Algorithmus (5.2.6) berechnet das Vielfache $[k]P$ eines Punktes, wobei er die zuvor (Zeile 2) erzeugte NAF(k) von rechts nach links abarbeitet. Dabei kann die NAF wie zuvor beschrieben codiert sein.

Algorithmus 5.2.6 Right-to-left NAF Punktmultiplikation

INPUT: $k = (k_{m-1} \dots k_1 k_0)_2, \quad P \in \mathcal{E}$
OUTPUT: $Q = [k]P$
1: $Q \leftarrow \mathcal{O}$
2: Berechne NAF(k)= $\sum_{j=0}^{l-1} k_j 2^j$ ▷ Benutze z.B. Alg. 5.2.4
3: **for** $i \leftarrow l-1$ **to** 0 **do**
4: **if** $k_i = 1$ **then** $Q \leftarrow Q \oplus P$
5: **if** $k_i = \bar{1}$ **then** $Q \leftarrow Q \ominus P$
6: $P \leftarrow [2]P$
7: **end for**
8: **return** Q

Die Benutzung einer NAF Repräsentation des Faktors k in den soeben betrachteten Algorithmen, führt zur folgenden Laufzeitbetrachtung. Aus den Punkten III und IV

von Theorem 5.2.4 folgt

$$\frac{m}{3}A + mD.$$

Werden nur projektive $\mathcal{LD}$ Koordinaten verwendet, so werden

$$8.33mM + (2M + I) \tag{5.3}$$

Körperoperationen benötigt. Dieses gilt für den *Right-to-left NAF* Algorithmus, da hier die Möglichkeit einer Punktaddition mit gemischten Koordinaten keine Vorteile bringt. Anders ist dies bei Algorithmus 5.2.5. Verwendet man dort für die Punktaddition gemischte $\mathcal{A}$-$\mathcal{LD}$ Koordinaten, so werden im Mittel nur

$$6.66mM + (2M + I) \tag{5.4}$$

Körperoperationen benötigt. Es ergibt sich in diesem Fall ein Laufzeitunterschied von ca. 20%.

Der RAM-Bedarf der beiden implementierten Algorithmen unterscheidet sich durch die Tatsache, dass in Algorithmus 5.2.5 die Addition und Subtraktion des Punktes P (Zeile 5, 6) mit gemischten Koordinaten erfolgen kann. Somit kann der Punkt $\ominus P$ als affiner Punkt berechnet und zwischengespeichert werden. Es ergibt sich letztlich ein Bedarf von einem affinen Punkt, zwei projektiven Punkten und einem Polynom für die NAF Darstellung des Faktors.
Im Fall von Algorithmus 5.2.6 bringt die Verwendung von affinen Punkten keine Performance-Vorteile, da der affine Punkt P pro Iterationsschritt verdoppelt wird (Zeile 6), was für affine Punkte eine Inversenbildung zur Folge hat. Demnach werden alle Zwischenergebnisse in projektiven Punkten gespeichert, was einen Bedarf von vier $\mathcal{LD}$ Punkten bedeutet. Für die Negation des $\mathcal{LD}$ Punktes P in Zeile 5 und die Speicherung der NAF(k) werden zusätzlich noch vier Polynome benötigt.

5.2.5 B-ary Punktmultiplikation

Der in diesem Abschnitt untersuchte Algorithmus verwendet eine weitere mögliche Darstellung des Faktors k von $Q = [k]P$. Diese wird im folgenden die B*-ary* Darstellung genannt. Der Faktor k schreibt sich dann

$$k = \sum_{j=0}^{d-1} k_j B^j, \quad k_i \in \{0, 1, \ldots, B-1\}, B = 2^r, r \in \mathbb{N} \setminus \{0\}.$$

Der Fall $r = 1$ entspricht der Binärdarstellung von k. Für $r = 4$ erhält man die Hexadezimaldarstellung mit $k_i \in \{0, \ldots, 9, A, B, C, D, E, F\}$. Der Vorteil dieser Darstellungsarten liegt darin, dass sie sich sehr einfach ineinander umrechnen lassen. Rechnet man beispielsweise von der Binärdarstellung ($r = 1$) in die Hexadezimaldarstellung ($r = 4$) um, so werden, vom niederwertigsten Bit beginnend, immer 4 Bits zusammengefasst. Diese bilden dann einen Koeffizienten k_i der Hexadezimaldarstellung.
Verwendet man eine B-ary Darstellung des Faktors k, so lassen sich die Algorithmen aus Abschnitt 5.2.3 entsprechend anpassen. Zusätzlich verwendet man nun eine

Lookup-Tabelle, in der vorberechnete Punkte P_i abgelegt werden. Für den Fall $r = 4$ benötigt man im allgemeinen eine Tabelle mit $2^r - 1 = 15$ Punkten. Diese Punkte entsprechen den Vielfachen $[1]P \dots [15]P$ und werden mit $P_i, \dots, P_{15}$ bezeichnet. Im Vergleich mit der allgemeinen B-ary Darstellung des Faktors k entsprechen diese vorberechneten Punkte nun genau den Koeffizienten k_j. Algorithmus 5.2.7 ist eine Variante des *Left-to-right binary* Algorithmus, welche die B-ary Darstellung des Faktors k nutzt. Den wichtigsten Teil dieses Algorithmus stellen die Zeilen 7 und 8 dar. Dort wird zunächst das [B]-fache des Punktes Q berechnet. Dies entspricht der Wertigkeit des Koeffizienten k_j in der B-ary Darstellung. Anschließend addiert man einen vorberechneten Punkt P_{k_j} zum Ergebnis, was dem Koeffizient k_j entspricht. Mit Hilfe des Horner-Schemas lässt sich die Korrektheit dieses Verfahrens verifizieren.

$$[B]\Big(\dots\big([B]([k_{d-1}]P) + [k_{d-2}]P\big) + \dots\Big) + [k_0]P$$

Algorithmus 5.2.7 benötigt für die Berechnung des Vielfachen zunächst $2^r - 2$ Additionen um die Lookup-Tabelle zu erzeugen. Im weiteren Verlauf werden $d = \lceil \frac{m}{r} \rceil$ Punktadditionen und $d \cdot r \approx m$ Punktverdopplungen ausgeführt. Damit ergeben sich

$$2^r - 2A + \lceil \frac{m}{r} \rceil A + \lceil \frac{m}{r} \rceil rD$$

Punktoperationen. Vergleicht man dies mit den binären Algorithmen 5.2.3 und 5.2.2, so ergibt sich eine Einsparung nur bei den Punktadditionen.

Algorithmus 5.2.7 Left-to-right B-ary Algorithmus

INPUT: Punkt $P \in \mathcal{E}$, Faktor $k = \sum_{j=0}^{d-1} k_j B^j, \quad k_j \in \{0, 1, \dots, B-1\}$
OUTPUT: $Q = [k]P$

```
 1: P_1 ← P                                  ▷ Vorberechnungen
 2: for i ← 2 to B − 1 do
 3:     P_i ← P_{i−1} + P                    ▷ P_i bezeichnet [i]P
 4: end for
 5: Q ← O
 6: for j ← d − 1 to 0 do                    ▷ Hauptteil
 7:     Q ← [B]Q
 8:     Q ← Q ⊕ P_{k_j}
 9: end for
10: return Q
```

Eine Variante des *Left-to-right B-ary* Algorithmus stellt Algorithmus 5.2.8 dar. Der Vorteil ergibt sich vor allem in einer kleineren Lookup-Tabelle. In dieser müssen nicht mehr alle Vielfache $P_i, \dots, P_{B-1}$ gespeichert werden, sondern nur die ungeraden Vielfachen $P_{2i+1}, 1 \leq i \leq (B-2)/2$, sowie die Punkte P_1 und P_2. Für $r = 4$ ergibt sich damit eine Speicherersparnis von 40%, für $r = 6$ bereits 48%.

Algorithmus 5.2.8 Modified B-ary Algorithmus

INPUT: Punkt $P \in \mathcal{E}$, Faktor $k = \sum_{j=0}^{d-1} k_j B^j, \quad k_j \in \{0, 1, \dots, B-1\}$
OUTPUT: $Q = [k]P$

```
 1: P_1 ← P;   P_2 ← [2]P                                 ▷ Vorberechnungen
 2: for i ← 1 to (B − 2)/2 do
 3:     P_{2i+1} ← P_{2i−1} + P_2
 4: end for
 5: Q ← O
 6: for j ← d − 1 to 0 do                                 ▷ Hauptteil
 7:     if k_j ≠ 0 then                                   ▷ k_j besteht aus r Bits
 8:         Berechne s_j, h_j so, dass gilt: k_j = 2^{s_j} · h_j,   h_j ≡ 1 mod 2
 9:         Q ← [2^{r−s_j}]Q
10:         Q ← Q ⊕ P_{h_j}
11:     else
12:         s_j ← r
13:     end if
14:     Q ← [2^{s_j}]Q
15: end for
16: return Q
```

Werden die Punkte der Lookup-Tabelle in affinen Koordinaten vorberechnet, so kann die Punktaddition in Zeile 10 mit gemischten Koordinaten durchgeführt werden. Der Punkt Q wird dabei in $\mathcal{LD}$ Koordinaten dargestellt. Für diese Konstellation und ein $r = 4$ ergibt sich

$$2^{r-1}(2M + I) + \frac{m}{4} 8M + 4mM = 2^{r-1}(2M + I) + 6mM.$$

Die Vorberechnung der Lookup-Tabelle benötigt $(2^{r-1} - 1)A + 1D$ Operationen, was dem ersten Summanden entspricht.

Der Speicherbedarf dieser Algorithmen hängt vor allem von der Größe der Lookup-Tabelle und den verwendeten Koordinaten ab. Die untersuchten Algorithmen wurden alle für ein $r = 4$ implementiert. Der höhere Speicherbedarf der Lookup-Tabelle macht die Wahl größerer r für den ebedded Bereich uninteressant. Im ersten Fall wurden zur Speicherung der vorberechneten Punkte $\mathcal{LD}$ Koordinaten verwendet, woraus sich, unter Hinzunahme von temporären Punkten, ein Bedarf von elf projektiven Punkten ergibt.
Verwendet man für die Lookup-Tabelle affine Punkte, so benötigt man zehn Punkte, inklusive einem für affine Zwischenergebnisse. Hinzu kommen noch zwei projektive Punkte, die für die Berechnung des Endergebnisses benötigt werden.

Man kann die B-ary Varianten zur Punktmultiplikation auch als *Sliding Window* Methoden verstehen. Für den Fall einer B-ary Darstellung mit $r = 4$ wäre die Window-Größe mit $w = r = 4$ gegeben.

5.2.6 Montgomery Punktmultiplikation

Diese Methode zur Punktmultiplikation geht auf López und Dahab zurück [LD99a] und verwendet eine Idee von Montgomery. Ihr Vorteil gegenüber den bisher betrachteten Verfahren besteht vor allem darin, dass nur sehr wenig temporärer Speicher benötigt wird. Weiterhin wird in den Zwischenschritten dieser speziellen Methode nur jeweils die x-Koordinate der benötigten Punkte berechnet. Erst im letzten Schritt berechnet man dann die zugehörige y-Koordinate. Die benötigten Formeln für das Verständnis dieses Algorithmus sind im folgenden nur für den affinen Fall angegeben. Die dazugehörenden Beweise sowie Formeln für den projektiven Fall sind in [LD99a] zu finden. Der implementierte Algorithmus 5.2.9 für projektive Koordinaten stammt aus [HMV04].

Gegeben sei ein fester Punkt $P = (x, y)$, sowie die beiden Punkte $P_1 = (x_1, y_1)$ und $P_2 = (x_2, y_2)$, für die allesamt gilt $P, P_1, P_2 \in \mathcal{E}$. Weiterhin sei $P_2 = P_1 \oplus P$ oder anders geschrieben $P = P_2 \ominus P_1$. Die x-Koordinate des Punktes $P_1 \oplus P_2$ sei mit x_3 bezeichnet und lässt sich aus den x-Koordinaten der Punkte P, P_1 und P_2 folgendermaßen berechnen

$$x_3 = \begin{cases} x + \frac{x_1}{x_1+x_2} + \left(\frac{x_1}{x_1+x_2}\right)^2, & P_1 \neq P_2 \\ x_1^2 + \frac{b}{x_1^2}, & P_1 = P_2 \end{cases} \tag{5.5}$$

Die y-Koordinate des Punktes $P_1 = (x_1, y_1)$ lässt sich mit Hilfe der x-Koordinaten von P_1 und P_2 sowie der Koordinaten von $P = P_2 \ominus P_1 = (x, y)$ wie folgt ausdrücken

$$y_1 = (x_1 + x)\big((x_1 + x)(x_2 + x) + x^2 + y\big)x^{-1} + y. \tag{5.6}$$

Der *Montgomery* Algorithmus zur Punktmultiplikation macht sich nun Gleichung 5.5 zunutze und berechnet pro Iterationsschritt i die x-Koordinaten der beiden Punkte $T_i = ([l]P, [l + 1]P)$. Der Faktor l entspricht den j höchstwertigen Bits der Binärdarstellung von k. Im nächsten Iterationschritt $i + 1$ wird nun abhängig von $k_{m-(i+1)}$ entweder $T_{i+1} = ([2l]P, [2l + 1]P)$ für $k_{m-(i+1)} = 0$ oder $T_{i+1} = ([2l + 1]P, [2l + 2]P)$ für $k_{m-(i+1)} = 1$ berechnet. Dazu wird genau eine Punktverdopplung und eine Punktaddition benötigt.
Nach dem letzten Iterationsschritt erhält man mit $T_k = ([k]P, [k + 1]P)$ die x-Koordinaten der beiden Punkte. Unter Zuhilfenahme von 5.6 wird nun noch die y-Koordinate des Punktes $[k]P$ bestimmt.
In Abbildung 5.1 sind die verschiedenen Iterationsschritte noch einmal illustriert. Die für diese Arbeit implementierte Variante des *Montgomery* Algorithmus verwendet standard projektive Koordinaten ($\mathcal{P}$) und berechnet während der einzelnen Iterationsschritte nur die X und Z Koordinaten der Zwischenergebnisse. Für die Anzahl an Körperoperationen ergibt sich folgende Abschätzung

$$6mM + (1I + 10M).$$

Der zweite Term entspricht der Berechnung des affinen Ergebnisses $[k]P$.
Der *Montgomery* Algorithmus in dieser Form beinhaltet jedoch noch eine Einschränkung, die in der Praxis der Kryptographie keine Rolle spielt. Für den Faktor k muss

$$[k]P = [\underbrace{(k_{m-1}k_{m-2}\dots k_{m-j}}_{\downarrow}\underbrace{k_{m-(j+1)}}_{\downarrow}k_{m-(j+2)}\dots k_1k_0)_2]P$$

$$\Rightarrow \quad ([l]P,[l+1]P) \rightarrow \boxed{\begin{array}{ll} ([2l]P,[l]P \oplus [l+1]P), & \text{wenn } k_{m-(j+1)} = 0 \\ ([l]P \oplus [l+1]P,[2(l+1)]P), & \text{wenn } k_{m-(j+1)} = 1 \end{array}}$$

Abbildung 5.1: *Ein Interationsschritt der Montgomery Punktmultiplikation. Nach i Schritten sind die x-Koordinaten von $[l]P$ und $[l+1]P$ für $l = (k_{m-1}\dots k_{m-j})_2$ bekannt. Der nächste Iterationsschritt $i+1$ benötigt eine Punktverdopplung und eine Punktaddition um die Punkte $[l']P$ und $[l'+1]P$ mit $l' = (k_{m-1}\dots k_{m-(j+1)})_2$ zu berechnen.*

gelten $k < n-1$, wobei $n = \text{Ord}(P)$ ist. Würde diese Bedingung nicht erfüllt, so ergäbe sich für den Faktor $k = n-1$ ein Punktepaar $T_{n-1} = (P_{n-1}, P_n = \mathcal{O})$.

Algorithmus 5.2.9 Montgomery Punktmultiplikations-Algorithmus

INPUT: Faktor $k = (k_{m-1}\dots k_1k_0)_2$ mit $k_{m-1} = 1, k < n-1$, Punkt $P = (x,y) \in \mathcal{E}$
OUTPUT: $Q = [k]P$

1: $X_1 \leftarrow x, Z_1 \leftarrow 1$ ▷ Berechne $(P,[2]P)$
2: $X_2 \leftarrow x^4; Z_2 \leftarrow x^2$
3: **for** $i \leftarrow m-2$ **to** 0 **do**
4:     **if** $k_i = 1$ **then**
5:         $T \leftarrow Z_1; Z_1 \leftarrow (X_1Z_2 + X_2Z_1)^2; X_1 \leftarrow xZ_1 + X_1X_2TZ_2$ ▷ $P_1 = P_1 \oplus P_2$
6:         $T \leftarrow X_2; X_2 \leftarrow X_2^4 + bZ_2^4; Z_2 \leftarrow T^2Z_2^2$ ▷ $P_2 = [2]P_2$
7:     **else** ▷ $k_i = 0$
8:         $T \leftarrow Z_2; Z_2 \leftarrow (X_1Z_2 + X_2Z_1)^2; X_2 \leftarrow xZ_2 + X_1X_2TZ_1$ ▷ $P_2 = P_1 \oplus P_2$
9:         $T \leftarrow X_1; X_1 \leftarrow X_1^4 + bZ_1^4; Z_1 \leftarrow T^2Z_1^2$ ▷ $P_2 = [2]P_1$
10:     **end if**
11: **end for**
12: $x_3 \leftarrow X_1Z_1^{-1}$ ▷ Berechne affines Ergebnis $Q = [k]P$
13: $y_3 \leftarrow (x + x_3)\big((X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1Z_2)\big)(xZ_1Z_2)^{-1} + y$
14: **return** (x_3, y_3)

Der Montgomery Algorithmus benötigt zur Zwischenspeicherung zwei projektive Punkte sowie sechs Polynome zur Berechnung der einzelnen Komponenten. Ein großer Vorteil gegenüber allen anderen betrachteten Algorithmen liegt jedoch darin, dass keine speziellen Funktionen zur Punktaddition oder -verdopplung aufgerufen werden. Dies hat zur Folge, dass diese Funktion keinen temporären Speicher für explizite Punktoperationen zur Verfügung stellen muss, sondern nur die Primitive der Körperoperationen verwendet.

5.3 Vergleich der Implementierungen

In diesem Abschnitt werden die Ergebnisse der Timing- und Resourcenmessungen für die einzelnen implementierten Algorithmen verglichen und ausgewertet. Wie für

die Algorithmen der Körperoperationen in $\mathbb{F}_{2^n}$ (siehe Kapitel 3) wurden auch für die Punktmultiplikation verschiedene Prozessortypen verwendet. Zunächst die beiden bereits bekannten, d.h. *C167* und *ST30* . Ebenso sind auch Vergleichsmessungen auf einem Intel Pentium IV 1,8 GHz und einem Intel Xeon Dual 2,8 GHz durchgeführt worden, deren Ergebnisse im Anhang zu finden sind.

5.3.1 Timingvergleiche

Die abgedruckten Tabellen enthalten die Messwerte der betrachteten Punktmultiplikations-Algorithmen die für eine Punktmultiplikation $[k]G$ mit einem zufällig gewählten Faktor $k, 1 \leq k \leq \mathrm{ord}(G)$ aufgenommen wurden. Für alle Messungen wurden die von der NIST als B-163 und B-233 beschriebenen Elliptischen Kurven verwendet. Die Ordnung des dazugehörigen Basispunkt G ist ebenso gegeben, wie die Koordinaten von G. Alle diese Werte sind in [Nat00] zu finden und zusätzlich im Anhang B abgedruckt. Das jeweilige irreduzible Polynom $f(x)$, welches für die Repräsentation des Körpers $\mathbb{F}_{2^n}$ benötigt wird, ist ebenfalls gegeben und lautet $f = x^{163} + x^7 + x^6 + x^3 + 1$ für die Kurve B-163 und $f = x^{233} + x^{74} + 1$ für die Kurve B-233 .
Alle Messungen wurden jeweils dreimal durchgeführt, um im Anschluss daran die hier abgedruckten Mittelwerte zu errechnen. Abhängig von der Performance des Prozessors wurden pro Messung und Algorithmus zwischen 5 und 100 Vielfachenbildungen $[k]G$ berechnet. Alle Algorithmen wurden jeweils in einer 8 Bit, 16 Bit und 32 Bit Version kompiliert und deren Laufzeit auf den einzelnen Prozessoren gemessen. Wie bereits im Theorieabschnitt beschrieben, hängt es vom Verhältnis der Körperoperationen Inversenbildung und Multiplikation ab, ob sich die Verwendung von projektiven Koordinaten anbietet. Die für die Multiplikation und ebenfalls für die Quadrierung von Körperelementen benötigte Reduzierung modulo $f(x)$ ist dabei der entscheidende Faktor. Aus diesem Grund wurden alle Messreihen zweimal ausgeführt. Die erste Messreihe beschreibt jeweils die Verwendung einer speziellen Reducevariante, die genau auf obiges $f(x)$ abgestimmt ist. Diese wurde im Abschnitt über Reduktionsalgorithmen auch als *Reduce NIST* Variante bezeichnet.
Für den zweiten Testfall wurde eine allgemeine Reducevariante verwendet, welche beliebige Polynome als Reducepolynom anwenden kann. Die beiden Algorithmen sind in Kapitel 3.2.4 näher beschrieben und wurden dort auch einzeln miteinander verglichen.

Punktmultiplikation auf dem C167 In Tabelle 5.4 sind alle gemessenen Werte für den C167 aufgeführt. Die grau hinterlegten Zeilen (16 Bit) sollen die native Registerbreite dieses Prozessors hervorheben. In den Diagrammen Abb. 5.2 und Abb. 5.3 sind zunächst die Messwerte für die beiden NIST-Kurven unter Verwendung eines speziellen Reduce-Algorithmus aufgeführt. Als Vergleich dazu sind in Abbildung 5.4 die Messwerte für die Kurve B-233 unter Verwendung eines allgemeinen Reduce-Algorithmus visualisiert. In allen drei Diagrammen wird jeweils zwischen den verschiedenen Wortbreiten unterschieden.
Wie man aus der Tabelle und den dazugehörigen Diagrammen erkennen kann, ist die 16 Bit Variante der Implementierung in allen Fällen ihren Pendants mit 8 Bit und 32

C167	RTL $\mathcal{A}$	RTL Bin $\mathcal{LD}$	LTR Bin $\mathcal{LD}$-$\mathcal{A}$	RTL NAF $\mathcal{LD}$	LTR NAF $\mathcal{LD}$-$\mathcal{A}$	Modi B-ary $\mathcal{LD}$	Modi B-ary $\mathcal{LD}$-$\mathcal{A}$	Montgomery
NIST Polynom B-163 – Spezielle Reduce Variante								
8 Bit	26,81	11,90	9,21	9,68	7,37	8,16	6,94	6,47
16 Bit	18,50	7,32	5,80	6,33	5,01	5,52	5,11	4,64
32 Bit	26,81	11,90	9,21	9,68	7,37	8,16	6,94	6,47
NIST Polynom B-233 – Spezielle Reduce Variante								
8 Bit	75,11	24,11	18,77	20,32	16,16	17,63	15,75	14,56
16 Bit	47,61	16,15	12,71	13,56	10,87	11,81	10,61	9,75
32 Bit	60,56	21,04	16,53	17,99	14,58	15,57	14,05	13,34
NIST Polynom B-163 – Allgemeine Reduce Variante								
8 Bit	36,21	37,47	31,77	32,17	27,52	29,40	25,70	25,87
16 Bit	25,85	29,87	25,36	25,51	22,20	23,38	20,82	21,19
32 Bit	38,85	50,18	42,41	40,49	34,44	36,44	31,46	32,37
NIST Polynom B-233 – Allgemeine Reduce Variante								
8 Bit	99,87	104,16	87,51	89,60	76,55	78,50	69,70	71,47
16 Bit	65,24	72,62	61,44	62,68	53,82	55,06	48,76	51,10
32 Bit	83,86	94,16	81,62	82,14	71,67	73,87	66,51	68,14

Tabelle 5.4: *Timing-Vergleich der verschiedenen Punktmultiplikations-Algorithmen auf dem C167 für die beiden NIST Kurven B-163 und B-233 und unterschiedliche Reduktionsalgorithmen. Die ermittelten Werte entsprechen einer Punktmultiplikation $[k]P$ mit $0 \leq k < ord(G) - 1$. (Alle Zeiten in Sekunden)*

Bit überlegen. Dies zeigt, dass bereits eine solche, leicht zu realisierende Anpassung an die spezifischen Eigenschaften eines Prozessors sehr große Einflüsse auf die letztliche Performance der implementierten Algorithmen nehmen kann. Weitere, tiefergehende Anpassungen, beispielsweise die Verwendung spezieller Prozessorregister, wurden in dieser Arbeit nicht durchgeführt, um die Portierbarkeit nicht einzuschränken.
In Abbildung 5.3 sind die Messwerte für die Implementierungen der Kurve B-233 aufgezeichnet, welche den speziellen NIST-Reduce Algorithmus verwenden. Deutlich ist der Vorteil der Algorithmen mit projektiven oder gemischten Koordinaten gegenüber der affinen Variante zu sehen. Im besten Fall der 16 Bit Version sind Laufzeitvorteile von bis zu 80% erreicht worden.
Für die Messungen zur Kurve B-163 zeigt sich ein ähnliches Bild (Diagramm Abb. 5.2). In diesem Fall beziffern sich die Laufzeitunterschiede für die einzelnen Algorithmen der 16 Bit Version auf bis zu 76%.

Wird kein spezieller Reduce-Algorithmus verwendet, so ergibt sich ein völlig anderes Bild. Die Laufzeiten aller untersuchten Algorithmen verlängern sich deutlich. Vor allem diejenigen Algorithmen die projektive oder gemischte Koordinaten verwenden werden deutlich langsamer. Dies liegt an der hohen Anzahl an Körpermultiplikationen und -quadrierungen und der damit einhergehenden Anzahl an Reduktionen, welche bei der Verwendung projektiver Koordinaten auftreten. Abbildung 5.4 zeigt dies für die Kurve

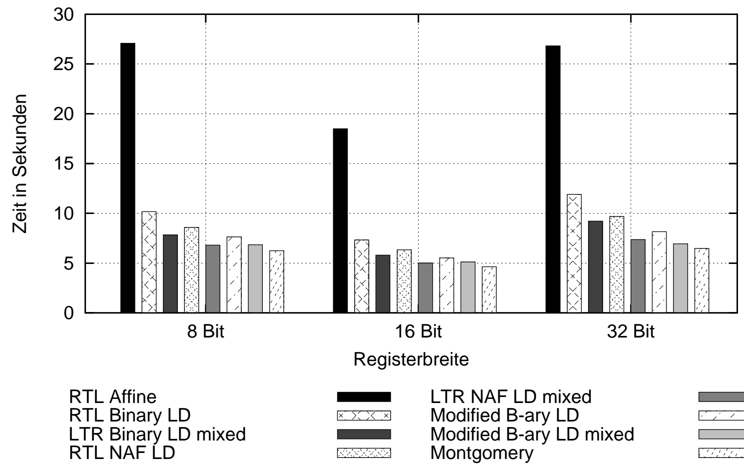

Abbildung 5.2: *Punktmultiplikationsalgorithmen auf einem C167 für die Kurve B-163 unter der Verwendung eines speziell angepassten Reduce-Algorithmus*

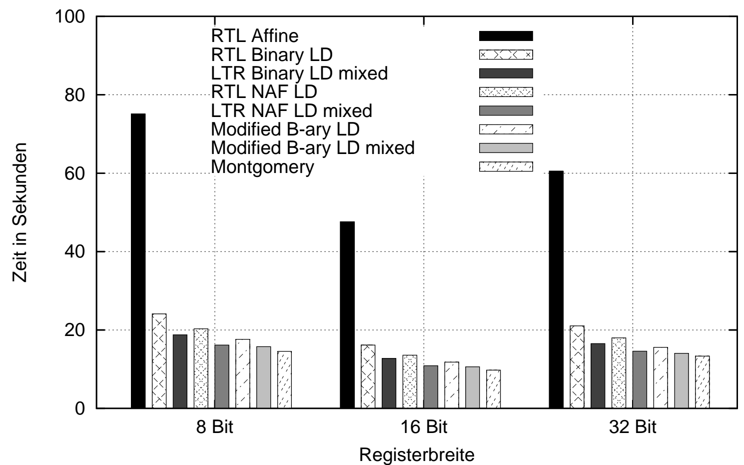

Abbildung 5.3: *Punktmultiplikationsalgorithmen auf einem C167 für die Kurve B-233 unter der Verwendung eines speziell angepassten Reduce-Algorithmus*

B-233. Wie man sieht, ergeben sich nur noch geringe Vorteile einzelner Algorithmen gegenüber der rein affinen Variante. Manche bewegen sich auf etwa dem gleichen Level wie die rein affine Variante, während der *Right-to-left binary* $\mathcal{LD}$ Algorithmus sogar etwas langsamer ist.

Die Festlegung auf ein bestimmtes Reducepolynom $f(x)$ und die damit einhergehende Möglichkeit einen speziellen Reduce-Algorithmus zu implementieren, führt insgesamt zu sehr großen Performanceunterschieden. Dies zeigen auch die folgenden Vergleiche für die Kurve B-233. Für den *Right-to-left* Algorithmus mit rein affinen Koordinaten ergibt sich für die 16 Bit Variante ein Unterschied von ca. 27%. Für den *Montgomery* Algorithmus ergibt sich sogar ein Vorteil von ca. 80%, wenn man anstatt einer allgemeinen Reduce-Routine die speziell angepasste verwendet.

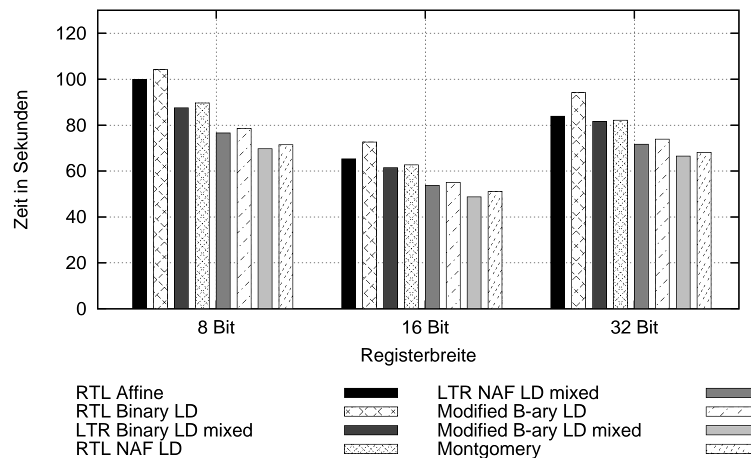

Abbildung 5.4: *Punktmultiplikationsalgorithmen auf einem C167 für die Kurve B-233 unter der Verwendung eines allgemeinen Reduce-Algorithmus*

Punktmultiplikation auf dem ST30 Beim ST30 handelt es sich um einen 32 Bit Prozessor, weshalb in Tabelle 5.5 die entprechenden Zeilen auch grau hinterlegt ist. Ebenso wie beim C167 sind auch hier alle Implementierungen, welche die native Prozessorbreite nutzen, die günstigsten. Tabelle 5.5 listet alle Messwerte zu den jeweiligen Algorithmen in 8 Bit, 16 Bit und 32 Bit Varianten auf. Zusätzlich wurde auch hier zwischen den beiden Möglichkeiten des Reduce-Algorithmus unterschieden. Alle Messungen wurden ebenfalls für die beiden NIST Kurven B-163 und B-233 durchgeführt.

Diagramm Abb.5.6 zeigt die Messwerte unter Verwendung des speziellen Reduce-Algorithmus für die Kurve B-233. Insgesamt ergibt sich hier ein ähnliches Bild wie im entsprechenden Diagramm des C167, nur mit dem Unterschied, dass hier die 32 Bit Implementierungen die performantesten sind. Für den affinen *Right-to-Left* Algorithmus ergibt sich beim Übergang von 8 Bit nach 32 Bit ein Unterschied von ca. 63%. Der Vorteil des speziellen Reduce-Algorithmus ist auch hier unverkennbar und äußert sich unabhängig von der verwendeten Wortbreite.
Im Fall von 32 Bit beziffert sich der Performancegewinn durch die Verwendung von projektiven Koordinaten auf ca. 81%.
Genau wie auf dem C167 Prozessor zeigt sich auch auf dem ST30 der *Montgomery* Algorithmus als am performantesten. Nur unwesentlich langsamer ist hier die *Modified B-ary* $\mathcal{LD}$-$\mathcal{A}$ Methode, welche jedoch den Nachteil eines höheren Speicherbedarfs hat. (Siehe dazu 5.3.2.)

Wird auf einen speziellen Reduce-Algorithmus verzichtet, ergibt sich abermals ein ähnliches Bild (Abb. 5.7) wie für den C167. Die 32 Bit Varianten zeigen die beste Performance, da sie der nativen Registerbreite entsprechen. Insgesamt liegen alle implementierten Algorithmen auf einem ähnlichen Level. Die Laufzeitunterschiede zwischen affinen und projektiven Algorithmen sind von einer sehr geringen Größenordung und nicht jeder projektive Algorithmus ist schneller als der rein affine. Dies entpricht den bereits beim C167 gemachten Beobachtungen für die Verwendung einer allgemeinen Reduce-Funktion.

ST 30	RTL $\mathcal{A}$	RTL Bin $\mathcal{LD}$	LTR Bin $\mathcal{LD}$-$\mathcal{A}$	RTL NAF $\mathcal{LD}$	LTR NAF $\mathcal{LD}$-$\mathcal{A}$	Modi B-ary $\mathcal{LD}$	Modi B-ary $\mathcal{LD}$-$\mathcal{A}$	Montgomery
NIST Polynom B-163 – Spezielle Reduce Variante								
8 Bit	5,53	1,88	1,47	1,62	1,26	1,43	1,30	1,14
16 Bit	3,69	1,41	1,09	1,20	0,97	1,05	0,99	0,88
32 Bit	2,63	1,09	0,78	0,87	0,67	0,71	0,66	0,57
NIST Polynom B-233 – Spezielle Reduce Variante								
8 Bit	15,35	4,49	3,47	3,82	3,03	3,26	2,97	2,70
16 Bit	9,76	3,03	2,44	2,58	2,08	2,27	1,98	1,87
32 Bit	5,81	1,82	1,39	1,58	1,27	1,34	1,22	1,15
NIST Polynom B-163 – Allgemeine Reduce Variante								
8 Bit	7,10	6,88	5,76	5,88	4,99	5,22	4,67	4,76
16 Bit	5,09	5,48	4,66	4,60	4,04	4,31	3,76	3,86
32 Bit	3,73	5,05	4,25	4,08	3,46	3,60	3,15	3,28
NIST Polynom B-233 – Allgemeine Reduce Variante								
8 Bit	19,96	18,93	15,78	16,25	13,93	14,21	12,64	13,07
16 Bit	12,94	13,08	11,07	11,38	9,73	9,85	8,82	9,27
32 Bit	8,16	9,18	7,91	7,99	6,99	7,18	6,56	6,63

Tabelle 5.5: *Timing-Vergleich der verschiedenen Punktmultiplikations-Algorithmen auf einem ST30 Prozessor für die beiden NIST Kurven B-163 und B-233 und unterschiedliche Reduktionsalgorithmen. Die ermittelten Werte entsprechen einer Punktmultiplikation $[k]P$ mit $0 \leq k < ord(G) - 1$. (Alle Zeiten in Sekunden)*

Die Diagramme für den ST30 sind charakteristisch für 32 Bit Systeme. Dies zeigt auch ein Vergleich mit den, im Angang A, abgedruckten Diagrammen für Intel Pentium IV und Intel Xeon Dual Prozessoren. Die dort erreichten Laufzeiten sind zwar utopisch für einen embedded Prozessor wie sie heute eingesetzt werden, jedoch sind die genannten Diagramme topographisch einander sehr ähnlich.

5.3.2 Recourcenbedarf

In diesem Abschnitt wird der Speicherbedarf aller implementierten Algorithmen zur Punktmultiplikation verglichen. Die dargestellten Werte sind für eine Registerbreite $W = 16$ Bit und ein irreduzibles Polynom f vom Grad $m = 233$ angegeben. Durch diese Festlegung ergibt sich ein Speicherbedarf für ein Polynom mit $t = \lceil \frac{m}{W} \rceil = 15$ Bytes. Um einen Punkt $P_{\mathcal{A}}$ in affinen Koordinaten speichern zu können, werden $2 \cdot t + 1 = 31$ Bytes benötigt. Für projektive Punkte im allgemeinen und Punkte $P_{\mathcal{LD}}$ in López-Dahab Koordinaten im Speziellen, sind $3 \cdot t = 45$ Bytes von nöten. Eine Möglichkeit die Darstellung eines affinen Punktes zu komprimieren ist in [BSS99] beschrieben, wurde aber für diese Arbeit nicht berücksichtigt da zusätzliche Körperoperationen von Nöten wären. Eine solche Kompression ist vor allem für die Datenübertragung interessant.

Die folgende Tabelle 5.6 fasst alle Werte der untersuchten Algorithmen zusammen.

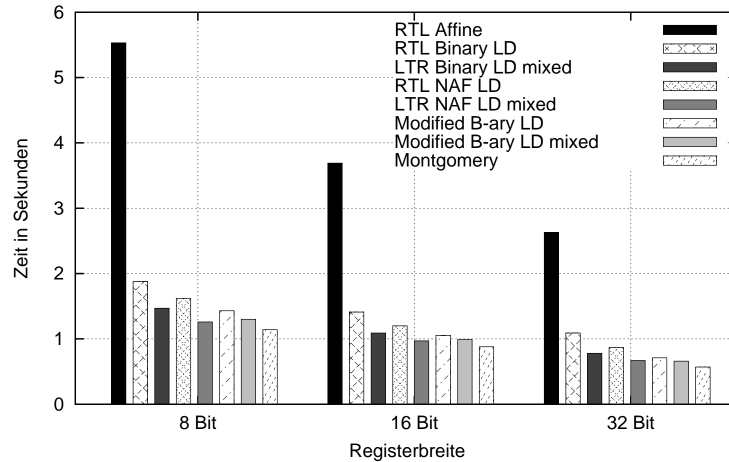

Abbildung 5.5: *Punktmultiplikationsalgorithmen auf einem ST30 Prozessor für die Kurve B-163 unter Verwendung eines speziellen Reduce-Algorithmus.*

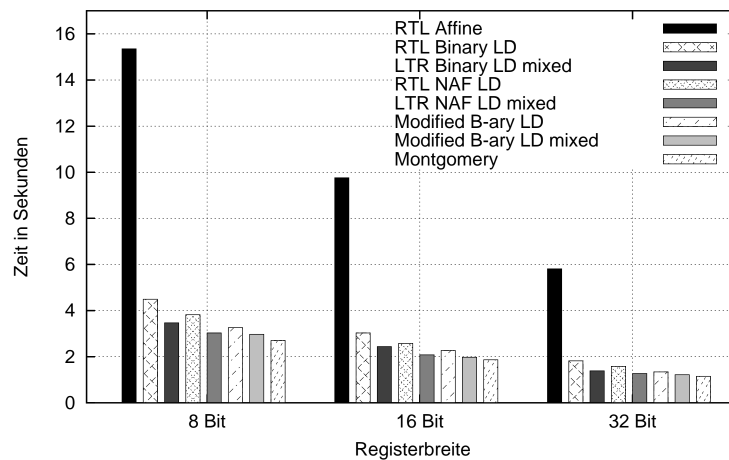

Abbildung 5.6: *Punktmultiplikationsalgorithmen auf einem ST30 Prozessor für die Kurve B-233 unter Verwendung eines speziellen Reduce-Algorithmus.*

Für die beiden Varianten des *Modified B-ary* Algorithmus wurde eine Tabellengröße für $r = 4$ vorausgesetzt.

Wie man sieht, wird der größte Teil des benötigten temporären Speichers für die Funktionen zur Punktaddition und -verdopplung benötigt. Ausnahme ist der *Montgomery* Algorithmus, der nur mit Hilfe der Körperoperationen die Vielfachenberechnung durchführt. Zwar benötigt dieser Algorithmus etwas mehr lokalen Speicher als verschiedene *Right-to-left* und *Left-to-right* Varianten, jedoch werden insgesamt bis zu 38% weniger belegt.

Im Vergleich zu den beiden *Modified B-ary* Algorithmen mit Lookup-Tabelle schneidet der *Montgomery* Algorithmus mit bis zu 54% Einsparung noch deutlich besser ab.

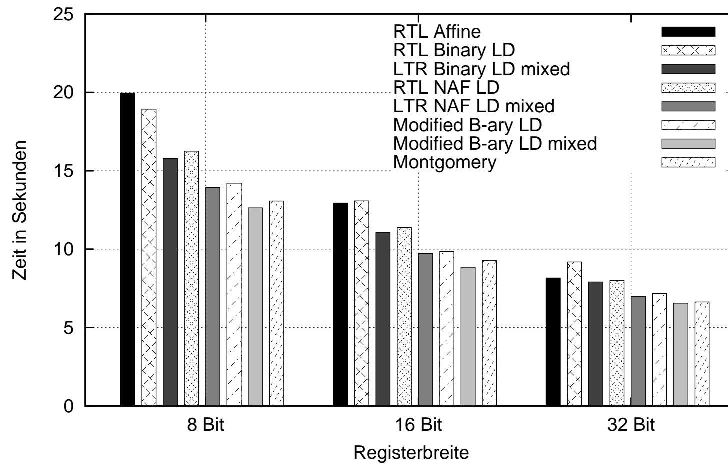

Abbildung 5.7: *Punktmultiplikationsalgorithmen auf einem ST30 Prozessor für die Kurve B-233 unter Verwendung eines allgemeinen Reduce-Algorithmus.*

5.3.3 Vergleich des ROM Bedarf

In diesem Abschnitt werden die implementierten Algorithmen zur Punktmultiplikation im Hinblick auf ihren ROM Bedarf untersucht und verglichen. Ebenso wie in den vorherigen Abschnitten wird auch dabei zwischen den beiden Mikroprozessoren C167 und ST30 unterschieden. Für jeden der beiden Prozessoren wurden 8 Bit, 16 Bit und 32 Bit Versionen kompiliert und betrachtet. Die aus den Linker-Files entnommenen Werte sind in den beiden Tabellen 5.7 und 5.8 aufgelistet. Auch in diesem Fall wurden die bereits bekannten Kurven B-163 und B-233 mit ihren irreduziblen Polynomen verwendet. Für diese beiden Kurven wurde der speziell auf die dazugehörigen Polynome angepasste Reduce-Algorithmus verwendet. Im dritten Fall wurde für die Kurve B-233 und ihr Polynom die allgemeine Reduce-Variante einkompiliert.

Insgesamt lässt sich feststellen, dass der ROM Speicherbedarf zunächst vom Prozesser und dem damit verbundenen Compiler abhängt. Jedoch bleiben die Verhältnisse der verschiedenen Algorithmen auf dem ST30 und C167, nahezu gleich. Weiterhin ergibt sich kaum ein Unterschied im ROM Bedarf im Bezug auf die verwendete Kurve und den dazugehörigen Reduce-Algorithmus. Eine Ausnahme macht hier der Montgomery Algorithmus, welcher für den Fall des allgemeinen Reduce-Verfahrens ca. 16% mehr Speicher benötigt als für die Spezielle Reduce Variante. Der insgesamt sehr hohe ROM-Bedarf des Montgomery Algorithmus lässt sich durch seinen Aufbau leicht erklären. Während in den weiteren untersuchten Algorithmen hauptsächlich Funktionsaufrufe für Punktadditon und Verdopplung vorkommen, sind es im Montgomery Algorithmus direkt die Körperoperationen über $\mathbb{F}_{2^n}$. Da die Reduktion für Multiplikation und Quadrierung von Körperelementen als eine seperate Funktion gehalten wurde, die immer nach diesen Operationen aufgerufen werden muss, führt dies zu einer grossen Zahl an Codezeilen. Jeder Multiplikations- oder Quadrierungsaufruf führt immer auch zu einem Folgeaufruf einer Reduktion. Abhilfe könnte durch die Integration des Reduktionsaufrufes am Ende der Multiplikations- und Quadrierfunktion geschehen. Abhängig

Algorithmus	RAM Bedarf lokal	Primitive: Quadrieren	Multiplizieren	Inversenbildung	Punktaddition $\mathcal{A}$	Punktverdopplung $\mathcal{A}$	Punktaddition $\mathcal{LD}$	Punktverdopplung $\mathcal{LD}$	Punktverdopplung $\mathcal{LD}$-$\mathcal{A}$	Gesamtbedarf
RAM-Bedarf		40	512	150	872	662	984	624	864	
Right-to-left Binary $\mathcal{A}$	62	–	–	–	⊕	⊕	–	–	–	934
Right-to-left Binary $\mathcal{LD}$	135	–	–	–	–	–	⊕	⊕	–	1119
Left-to-Right Binary $\mathcal{LD}$-$\mathcal{A}$	90	–	–	–	–	–	–	⊕	⊕	954
Right-to-left NAF $\mathcal{LD}$	241	–	⊕	–	–	–	⊕	⊕	–	1205
Left-to-Right NAF $\mathcal{LD}$-$\mathcal{A}$	136	–	–	–	–	–	–	⊕	⊕	1000
Modified B-ary $\mathcal{LD}$	495	–	–	–	–	–	⊕	⊕	–	1479
Modified B-ary $\mathcal{LD}$-$\mathcal{A}$	400	–	–	–	–	–	–	⊕	⊕	1264
Montgomery $\mathcal{P}$	180	⊕	⊕	⊕	–	–	–	–	–	692

Tabelle 5.6: *Bedarf an temporärem Speicher in Bytes, für die verschiedenen Punktmultiplikations-Algorithmen bei 16 Bit Registerbreite und einem speziellen Reduce-Algorithmus für ein Reducepolynom vom Grad 233.*

von der verwendeten Reduce Funktion würde sich dann allerdings in diesen beiden Funktionen der ROM- und evtl. auch der RAM-Bedarf erhöhen, weshalb für die Referenzimplementierung davon abgesehen wurde.
Der ebenfalls sehr hohe ROM-Verbrauch der beiden B-ary Algorithmen erklärt sich durch die Vorberechnungen der verschiedenen Lookup-Tabellen.
Der für alle untersuchten Punktmultiplikationsalgorithmen verwedete Algorithmus zur Körpermultiplikation, der Left-to-Right comb Window Algorithmus, hat in der implementierten Variante auf dem C167 für 16 Bit Registerbreite und die Kurve B-233 einen ROM Bedarf von 606 Bytes während sein Pendant ohne Lookup-Tabelle mit 238 Bytes auskommt. Für die beiden Reduce-Algorithmen gibt es ebenfalls grosse Unterschiede. Während der allgemeine Reduce Algorithmus einen ROM Bedarf von 406 Bytes aufweist, kommt die speziell an das NIST 233 Polynom angepasste Variante mit 256 Bytes zurecht. Somit ist diese Variante nicht nur im Bezug auf Laufzeit und RAM-Bedarf deutlich günstiger, sondern auch was den ROM Verbrauch angeht. Auch die beiden bereits vorher untersuchten Algorithmen zur Inversenbildung weisen hinsichtlich ihrers ROM Verbrauch deutliche Unterschiede auf. Der Almost Inverse Algorithmus benötigt 960 Bytes ROM, während es beim Erweiterten Euklidischen Algorithmus in der erstellten Variante nur 388 Bytes sind.

5.3.4 Fazit

Auf Grund des niedrigen RAM Speicherbedarfs und trotzdem sehr guter Timingwerte ist der Montgomery Algorithmus für den Einsatz in embedded Systemen gut geeignet. Der durch die spezielle Art der Implementierung entstandene relative hohe ROM Be-

C167	RTL $\mathcal{A}$	RTL Bin $\mathcal{LD}$	LTR Bin $\mathcal{LD}$-$\mathcal{A}$	RTL NAF $\mathcal{LD}$	LTR NAF $\mathcal{LD}$-$\mathcal{A}$	Modi B-ary $\mathcal{LD}$	Modi B-ary $\mathcal{LD}$-$\mathcal{A}$	Montgomery
NIST Polynom B-163 – Spezielle Reduce Variante								
8 Bit	370	456	414	940	760	1282	1153	1836
16 Bit	374	464	418	950	774	1296	1157	1846
32 Bit	416	508	462	1168	922	1500	1416	1890
NIST Polynom B-233 – Spezielle Reduce Variante								
8 Bit	370	456	414	928	760	1282	1153	1836
16 Bit	374	464	418	938	774	1296	1157	1846
32 Bit	416	508	462	1162	922	1500	1416	1890
NIST Polynom B-233 – Allgemeine Reduce Variante								
8 Bit	370	456	414	940	760	1282	1153	2184
16 Bit	374	464	418	950	774	1296	1157	2194
32 Bit	416	508	462	1168	922	1500	1416	2238

Tabelle 5.7: *ROM-Bedarf der verschiedenen Punktmultiplikationsalgorithmen auf einem C167. Die ermittelten Werte entsprechen einer Punktmultiplikation $[k]P$ mit $0 \leq k < ord(G) - 1$. (Alle Werte in Bytes)*

darf des Montgomery Algorithmus muss zunächst als Nachteil dieser Variante gesehen werden. Jedoch lässt sich dies durch verschiedene Optimierungen in der Implementierung auf ein wohl akzeptables Maß reduzieren. Allerdings gilt für diesem Algorithmus die Einschränkung $k < n-1$ für $[k]G$, die folglich für alle kryptographischen Verfahren beachtet werden muss. Werden, wie beim Diffie-Hellman Verfahren (siehe 6.1), zwei Faktoren a, b gewählt, die dann zur Vielfachenbildung $[a][b]P$ herangezogen werden, so muss obige Einschränkung auch für das Produkt $a \cdot b$ gelten.

Für alle anderen untersuchten Algorithmen zur Punktmultiplikation gelten keine Einschränkungen in obiger Form. Die Wahl eines Algorithmus hängt sehr von der verwendeten Hardware und der verwendbaren Speichergröße ab. Beide *Modified B-ary* Algorithmen verwenden eine Lookup-Tabelle und benötigen deshalb deutlich mehr Speicher (sowohl RAM als auch ROM) als beispielsweise der *Left-to-right Binary* Algorithmus, der gemischte $\mathcal{LD}$-$\mathcal{A}$ Koordinaten verwendet. Vorteil ist jedoch eine bessere Laufzeit der *Modified B-ary* Varianten.

Nur geringfügig mehr RAM Speicher als der *Left-to-right Binary* Algorithmus benötigt die *Left-to-right NAF* Variante mit gemischten $\mathcal{LD}$-$\mathcal{A}$ Koordinaten. Jedoch ist dieser Algorithmus teilweise deutlich schneller als der *Left-to-right* mit binär dargestelltem Faktor. Zusätzlich wird der *Modified B-ary* $\mathcal{LD}$-$\mathcal{A}$ Algorithmus ebenfalls teilweise deutlich unterboten.

Obwohl für den *Left-to-right NAF* $\mathcal{LD}$-$\mathcal{A}$ zusätzlich die NAF des Faktors k berechnet werden muss, empfiehlt sich dieser als der wohl beste Kompromiss zwischen Speicherbedarf und Laufzeit.

Die Firma 3Soft verfügt über eine Testimplementierung zur Punktmultiplikation, wel-

ST 30	RTL $\mathcal{A}$	RTL Bin $\mathcal{LD}$	LTR Bin $\mathcal{LD}$-$\mathcal{A}$	RTL NAF $\mathcal{LD}$	LTR NAF $\mathcal{LD}$-$\mathcal{A}$	Modi B-ary $\mathcal{LD}$	Modi B-ary $\mathcal{LD}$-$\mathcal{A}$	Montgomery
NIST Polynom B-163 – Spezielle Reduce Variante								
8 Bit	488	604	548	1156	904	1576	1440	2240
16 Bit	500	616	560	1216	940	1632	1488	2256
32 Bit	484	600	544	1136	892	1572	1428	2240
NIST Polynom B-233 – Spezielle Reduce Variante								
8 Bit	488	604	548	1144	904	1576	1440	2240
16 Bit	500	616	560	1204	940	1632	1488	2256
32 Bit	484	600	544	1124	892	1572	1428	2240
NIST Polynom B-233 – Allgemeine Reduce Variante								
8 Bit	488	604	548	1156	904	1576	1440	2684
16 Bit	500	616	560	1216	940	1632	1488	2700
32 Bit	484	600	544	1136	892	1572	1428	2684

Tabelle 5.8: *ROM-Bedarf der verschiedenen Punktmultiplikationsalgorithmen auf einem ST30. Die ermittelten Werte entsprechen einer Punktmultiplikation $[k]P$ mit $0 \leq k < ord(G) - 1$. (Alle Werte in Bytes)*

che mit einem primen Grundkörper $\mathbb{F}_p$ als Basis für die Elliptische Kurve rechnet. Die dort ermittelten Performancewerte sind auf einem C167 für die NIST-Kurve P-192 mit einer Primzahl $p = 2^{192} - 2^{64} - 1$ für $\mathbb{F}_p$ entstanden. Dies erschwert einen Vergleich mit den für diese Arbeit gemessenen Werten für binäre Grundkörper $\mathbb{F}_{2^n}$. Es lässt sich jedoch schließen, dass die Algorithmen über $\mathbb{F}_{2^n}$, bei polynomieller Darstellung der Elemente, etwas langsamer sind als ihre Pendants über $\mathbb{F}_p$. Dies bestätigt auch die eingangs erwähnten Ergebnisse von Hankerson et al. [HHM01; BHLM01].

Kapitel 6

Kryptosysteme mit Elliptischen Kurven

Im Jahr 1985 wurden von Neal Koblitz [Nea87] und Victor S. Miller [Mil86] unabhängig voneinander zwei Artikel veröffentlicht, in denen sie das Diffie-Hellman Verfahren für eine Elliptische Kurve vorstellten. Seit dieser Zeit wurden mehrere klassische *Public-Key*-Verfahren auf die Elliptischen Kurven adaptiert. Die wichtigsten und gebräuchlichsten werden in diesem Abschnitt kurz vorgestellt.
Alle hier aufgeführten Verfahren basieren auf dem Diskreten-Logarithmus-Problem für Elliptische Kurven, für das die bisher besten bekannten Lösungsalgorithmen allesamt exponentielle Laufzeit aufweisen. Beispiele für solche Algorithmen sind der *Baby-Step Giant-Step* Algorithmus, das *Silver-Pohling-Hellman*-Verfahren oder die *Pollard* ρ-Methode. Die genauen Funktionsweisen dieser Methoden werden in dieser Arbeit jedoch nicht betrachtet und es sei dazu auf [BSS99], [Was03], [Wer02], [HMV04] und [CF05] verwiesen.
Auf zwei besondere Typen von Elliptischen Kurven, sogenannte *anomale* Kurven und *supersinguläre* Kurven, für die es spezielle Algorithmen zur Lösung des ECDLP gibt, soll ebenso nicht weiter eingegangen werden. Sie sind auf Grund ihrer speziellen Eigenschaften kryptographisch nicht geeignet und werden praktisch nicht eingesetzt. Die Gründe dafür findet man u.a. in [Wer02], [Was03], [BSS99], [MOV93], [Sem98a] und [Sem98b].

Die folgenden drei Kryptosysteme, bzw. deren Varianten für Elliptische Kurven, sollen in diesem Abschnitt vorgestellt werden.

- Diffie-Hellman Schlüsselaustausch mit Elliptischen Kurven
- ElGamal Verschlüsselung
- Der Digitale-Signatur-Algorithmus für Elliptische Kurven (ECDSA)

Dabei setzt man voraus, dass die gewählte Elliptische Kurve $\mathcal{E}$ *kryptographisch sicher* ist, d.h., der Diskrete Logarithmus ist schwer zu berechnen. Weiterhin sei die Kenntnis dieser drei Verfahren im allgemeinen Fall vorausgesetzt, weshalb hier nur die zur Adaption notwendigen Schritte beleuchtet werden.

Mehr zu diesen drei Methoden findet man in [ElG85],[DH76], [DK01], [Buc04] und [MOV96].

6.1 Diffie-Hellman Schlüsselaustausch

Die von Diffie und Hellman vorgestellte Methode zur Vereinbarung eines gemeinsamen geheimen Schlüssels über einen unsicheren Kanal ([DH76]), lässt sich ohne großen Aufwand auf Elliptische Kurven übertragen.
Wollen sich die beiden Kommunikationspartner Alice und Bob auf einen gemeinsamen Schlüssel verständigen, aus dem dann beispielsweise ein AES Schlüssel abgeleitet werden kann, so verwenden sie das in Abbildung 6.1 dargestellte Protokoll.
Einem Angreifer Eve stehen durch Abhören der Leitung folgende Informationen zur

Diffie Hellman Schlüsselaustausch mit Elliptischen Kurven

① Alice und Bob einigen sich auf eine Elliptische Kurve $\mathcal{E}$ über einem endlichen Körper $\mathbb{F}_{q^n}$ mit $q = p^n, p = Primzahl, n \in \mathbb{N} \setminus 0$ und vereinbaren einen gemeinsamen Punkt $P \in \mathcal{E}(\mathbb{F}_{q^n})$. Die von P erzeugte Untergruppe $\langle P \rangle \subseteq \mathcal{E}(\mathbb{F}_{q^n})$ muss eine große Ordnung m haben. Gewöhnlich wählt man Kurve $\mathcal{E}$ und den Punkt P so, dass die Ordnung m von P eine große Primzahl ist.

② Alice wählt eine Zahl a, mit $1 < a < m$, berechnet $P_a = [a]P$ und sendet P_a zu Bob.

③ Bob wählt ebenso eine Zahl b, mit $1 < b < m$, berechnet $P_b = [b]P$ und sendet P_b zu Alice.

④ Alice berechnet $[a]P_b = [ab]P$.

⑤ Bob berechnet $[b]P_a = [ab]P$.

⑥ Mit einer durchaus öffentlich vereinbarten Methode können nun Bob und Alice den gemeinsamen Schlüssel aus $[ab]P$ ableiten. Beispielsweise mit Hilfe einer Hash-Funktion, angewendet auf die x-Koordinate des Punktes.

Abbildung 6.1: *Diffie Hellman Schlüsselaustausch mit Elliptischen Kurven*

Verfügung. Er kennt alle öffentlichen Parameter wie $\mathcal{E}$, $\mathbb{F}_{q^n}$, P und m, sowie die gesendeten Punkte P_a und P_b. Um an das gemeinsame Geheimnis von Alice und Bob zu kommen, muss Eve den Punkt $[ab]P$ berechnen können. Dieses Problem nennt man das *Diffie-Hellman-Problem für Elliptische Kurven*. Es ist lösbar, wenn man den Diskreten Logarithmus für Elliptische Kurven lösen kann. Ob man auch ohne die Lösung des *Diskreten Logarithmus Problems* den Punkt $[ab]P$ berechnen kann, ist im Augenblick noch nicht bekannt.

6.2 ElGamal Kryptosystem für Elliptische Kurven

Die im Augenblick bekannteste und verbreiteste Methode der Public-Key-Kryptographie mit Elliptischen Kurven stellt das *ElGamal*-Kryptosystem dar. Genau wie im Fall des Diffie-Hellman-Protokolls handelt es sich auch bei ElGamal um eine Anpassung auf die Punktegruppe der Elliptischen Kurven. Ursprünglich von Taher ElGamal für multiplikative Gruppen vorgestellt ([ElG85]), soll in diesem Abschnitt die Variante für Elliptische Kurven betrachtet werden.
Möchte Alice eine Nachricht an Bob schicken, so benötigt Bob zunächst einen öffentlichen Schlüssel. Dazu wählt Bob als erstes eine kryptographisch sichere Kurve über $\mathbb{F}_{2^n}$, d.h. dass Diskrete Logarithmus Problem ist in $\mathcal{E}(\mathbb{F}_{2^n})$ schwer zu lösen. Er wählt nun einen Punkt $P \in \mathcal{E}$ mit großer, gewöhnlicherweise primer Ordnung m und berechnet dann $B = [s]P$, mit geheimem $s \in \mathbb{N}$. Der öffentliche Schlüssel von Bob besteht aus der Elliptischen Kurve $\mathcal{E}$, dem endlichen Körper $\mathbb{F}_{2^n}$ und den beiden Punkten P und B. Man schreibt dies auch als Quadrupel $(\mathcal{E}, \mathbb{F}_{2^n}, P, B)$
Die Zahl s ist der geheime Schlüssel von Bob.
In Abbildung 6.2 sind die für eine Nachrichtenübermittlung von Alice zu Bob nötigen Schritte aufgeführt.
Dass die im letzten Schritt durchgeführte Entschlüsselung von M tatsächlich korrekt

ElGamal Kryptosystem mit Elliptischen Kurven

① Alice besorgt sich den öffentlichen Schlüssel von Bob $(\mathcal{E}, \mathbb{F}_{2^n}, P, B)$, beispielsweise durch herunterladen von einem Schlüsselserver.

② Alice bildet ihre Nachricht auf einen Punkt $M \in \mathcal{E}(\mathbb{F}_{2^n})$ ab.

③ Alice wählt eine Zufallszahl k und berechnet $M_1 = [k]M$.

④ Alice berechnet weiterhin $M_2 = M_1 + [k]B$. Im Fall $[k]B = \mathcal{O}$ kehrt Alice zurück zu Schritt 3 und berechnet ein neues k.

⑤ Alice sendet (M_1, M_2) zu Bob.

⑥ Bob entschlüsselt die Nachricht durch Berechnung von $M = M_2 - [s]M_1$.

Abbildung 6.2: *ElGamal Kryptosystem mit Elliptischen Kurven*

ist, lässt sich durch einfaches Nachrechnen zeigen

$$M_2 - [s]M_1 = (M + [k]B) - [s][k]P = M + [k][s]P - [sk]P = M.$$

Ein Angreifer Eve kennt den öffentlichen Schlüssel von Bob und sie kennt die beiden abgehörten Punkte M_1 und M_2. Könnte sie nun das ECDLP lösen, so wäre sie in der Lage die Nachricht $M = M_2 - [s]M_1$ zu berechnen. Genauso könnte sie aus der Kenntnis von M_1 und P den Wert k und damit $M = M_2 - [k]P$ errechnen.
Alice muss bei der Wahl von k beachten, dass sie für jede Verschlüsselung einen neuen

Faktor wählt. Angenommen Alice schickt zwei Nachrichten M und M' and Bob, die sie beide mit dem gleichen k verschlüsselt. Ein Angreifer Eve stellt diesen Fall sehr einfach fest, da dann $M_1 = M'_1$ gilt. Kennt Eve aus irgendeinem Grund eine der beiden Nachrichten, beispielsweise M, so kann sie die zweite Nachricht $M' = M - M_2 + M'_2$ berechnen.

Das ElGamal Kryptosystem lässt sich nicht nur zum Verschlüsseln und Entschlüsseln von Nachrichten verwenden, sondern man kann daraus auch ein Signaturschema konstruieren. Eine dem ElGamal Signaturschema sehr ähnliche Variante für digitale Signaturen wird im folgenden Abschnitt vorgestellt.

6.3 Digitaler Signatur Standard für Elliptische Kurven

Der Digitale Signatur Standard (DSS) basiert auf dem Digitalen Signatur Algorithmus (DSA) [Nat00] und verwendet im Original die multiplikative Gruppe eines endlichen Körpers. Die in diesem Abschnitt betrachtete Version des DSA verwendet Elliptische Kurven als Basis und wird demzufolge auch als *ECDSA* bezeichnet. Der Algorithmus selbst ist eine spezielle Variante des ElGamal Signaturschemas.
Wenn Alice ein Dokument signieren möchte, so generiert sie zunächst den Hashwert m dieses Dokuments. Nun wählt Alice eine Elliptische Kurve $\mathcal{E}$ über einem endlichen Körper $\mathbb{F}_{2^n}$, so dass gilt

$$\#\mathcal{E}(\mathbb{F}_{2^n}) = hr,$$

wobei r eine große Primzahl ist. Der zweite Faktor h (co-Faktor genannt) muss möglichst klein sein (gewöhnlich 1,2 oder 4), um den Algorithmus effektiv durchführen zu können. Alice wählt nun einen Punkt $P \in \mathcal{E}$ der Ordnung r und berechnet mit einem geheimen a den Punkt $Q = [a]P$. Die folgenden Werte macht Alice nun öffentlich

$$\mathbb{F}_{2^n}, \mathcal{E}, r, P, Q.$$

Die einzelnen Schritte zur Signaturerzeugung und -verifikation sind in Abbildung 6.3 aufgezeigt.

Wenn die Signatur akzeptiert wird, dann muss die folgende Gleichung gelten:

$$\begin{aligned} V &= [u_1]P + [u_2]Q \\ &= [s^{-1}m]P + [s^{-1}x]Q \\ &= s^{-1}([m]P + [xa]P) \\ &= k(m+ax)^{-1}[m+ax]P \\ &= [k]P \\ &= R \end{aligned}$$

Der Vorteil des ECDSA gegenüber dem Signaturschema von ElGamal liegt in der Verifikation der Signatur. Während im ECDSA nur zwei Punktmultiplikationen durchgeführt werden müssen, sind es beim ElGamal Verfahren deren drei. Dies macht sich vor allem dann bemerkbar, wenn viele Verifikationen durchgeführt werden müssen.

Digitaler Signatur Algorithmus mit Elliptischen Kurven

Signieren:

① Alice wählt eine zufällige Zahl $k, 1 \leq k < r$ und berechnet den Punkt $R = [k]P = (x, y)$.

② Sie berechnet $s = k^{-1}(m + ax) \pmod r$.

Verifizieren:

① Bob berechnet $u_1 = s^{-1}m \pmod r$ und $u_2 = s^{-1}x \pmod r$.

② Bob berechnet den Punkt $V = [u_1]P + [u_2]Q$.

③ Bob aktzeptiert die Signatur wenn gilt: $V = R$

Abbildung 6.3: *Digitaler Signatur Algorithmus mit Elliptischen Kurven*

Kapitel 7

Ausblick

Die immer leistungsfähigeren Computer führen dazu, dass bei den klassischen *Public-Key-Systemen* die Schlüssellängen immer länger werden müssen, um eine vergleichbare Sicherheit garantieren zu können. Galten für RSA vor Jahren noch 512 Bit Schlüssellänge als ausreichend, so werden mittlerweile bereits 2048 Bit empfohlen.
Nicht zuletzt wegen ihrer, im Vergleich dazu relativ kurzen Schlüssellängen, rücken Verfahren mit Elliptischen Kurven immer mehr in den Blickpunkt. Mittlerweile werden diese Techniken immer häufiger eingesetzt und finden, auf Grund dieser kurzen Schlüssellängen, auch zunehmend Verwendung im embedded Umfeld.

Was wird jedoch passieren, wenn in ein paar Jahren auch diese Schlüssellängen wieder zu lang geworden sind?

Es gibt Ansätze von Public-Key-Systemen die auf einer weiteren Art von *Kurve* basieren. Systeme die solche *hyperelliptischen Kurven* nutzen, werden bereits vielfach diskutiert. Für tiefergehende Erklärungen und mögliche Algorithmen sei auf [Kob98], [BSS99] sowie [CF05] verwiesen.
Eine weitere häufig diskutierte Möglichkeit ist die Verwendung von *Quantenkryptographie*. Hierbei macht man sich verschiedene quantenmechanische Eigenschaften von Photonen zu nutze, die es einem Angreifer unmöglich machen die Kommunikation abzuhören. Die Problematik an dieser Variante sind jedoch die technischen Voraussetzungen. Diese sind sehr aufwendig und damit auch sehr teuer, weshalb man noch nicht über einen gewissen „Laborstatus“ hinausgekommen ist. Einen ersten Einblick in Quantenkryptograpie findet man in [Sin03].
Ein weiteres Public-Key-System ist das von Lenstra und Verheul [LV00] vorgestellte *XTR* Public-Key-System. Der Name XTR steht für die gleichlautende Abkürzung *ECSTR*, welche *Efficient and Compact Subgroup Trace Representation* bedeutet.
Im Unterschied zu den meisten, auf dem diskreten Logarithmus basierenden kryptographischen Verfahren, wird beim XTR keine zyklische Untergruppe von $\mathbb{Z}_p^*$ oder die Punktgruppe einer Elliptischen Kurve verwendet, sondern eine Untergruppe $\langle g \rangle$ von $\mathbb{F}_{p^6}^*$. Diese Untergruppe ist in keinen der Unterkörper von $\mathbb{F}_{p^6}$ einzubetten. Die Elemente $g^n \in \mathbb{F}_{p^6}$ werden durch die Spur $Tr(g^n) \in \mathbb{F}_{p^2}$ als Elemente in $\mathbb{F}_{p^2}$ darge-

stellt. Die Darstellung eines Elements $g^n \in \langle g \rangle$ wird durch diesen Trick um ein Drittel kompakter. Da man anstatt in $\mathbb{F}_{p^6}$ in $\mathbb{F}_{p^2}$ rechnen kann, erhält man weiterhin einen Effizienzvorteil um den Faktor 3.
Die Schlüsselänge von XTR ist mit ECC vergleichbar. Um die Sicherheit eines RSA-Verfahrens mit 1024 Bit Schlüssellänge zu erreichen, werden für das XTR Verfahrenman laut Lenstra und Verheul nur 170 Bit benötigt. In [LV00] wird ein *XTR Diffie-Hellman* Verfahren, sowie ein *XTR ElGamal* Kryptosystem vorgestellt. Auch eine XTR Variante der Nyberg-Rueppel Signatur wird beschrieben.
Die mathematischen Konstrukte und Objekte die man für kryptographische Anwendungen verwendet, werden immer komplizierter und komplexer. Während es für RSA oder das klassische ElGamal Verfahren noch ausreicht, Restklassenarithmetik zu beherrschen, werden bereits bei Elliptischen Kurven anspruchsvollere Gebiete der Algebra verwendet. Für Hyperelliptische Kurven kommen weitere Kapitel der algebraischen Geometrie hinzu, während man für Quantenkryptographie zunächst tiefergehende physikalische Kenntnisse benötigt.
Weitere mathematische Objekte, die für kryptographische Zwecke interessant sein könnten, werden augenblicklich untersucht. Eines ist der *algebraische Torus*, dessen Verwendung in einer aktuellen Arbeit mit dem Titel „Practical Cryptography in High Dimensional Tori“ von van Dijk et.al.[DGP+05] vorgestellt wird.

Insgesamt bleibt festzustellen, dass sich auf dem Gebiet der Kryptographie einiges tut, man sich, will man die Hintergründe verstehen, jedoch sehr ausführlich damit beschäftigen muss.

Anhang

Anhang A

Punktmultiplikation auf PC-Systemen

In diesem Abschnitt finden sich Tabellen und dazugehörige Diagramme für die implementierten Punktmultiplikationsalgorithmen, welche bei verschiedenen Tests auf PC-Systemen entstanden. Für diese Tests wurden jedoch keine spezifischen Anpassungen oder Verbesserungen vorgenommen, so dass die ermittelten Werte nur als Orientierung zu sehen sind.
In Tabelle A.1 und den Diagrammen A.1, A.2, A.3 und A.4 sind die Werte eines Intel Pentium IV Prozessors mit 1,8 GHz und 512 MB RAM aufgeführt. Alle Programmteile wurden in einer *Cygwin* Befehlszeile unter *MS Windows XP* mit dem GNU C-Compiler *gcc 3.3.3* kompiliert.

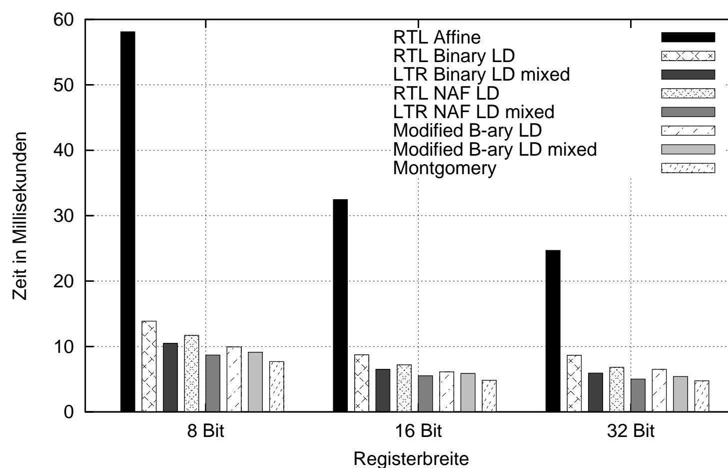

Abbildung A.1: *Punktmultiplikationsalgorithmen auf einem Intel Pentium IV 1,8 GHz für die Kurve B-163 unter der Verwendung eines speziellen Reduce-Algorithmus*

Tabelle A.2 und die Diagramme A.5, A.6, A.7 und A.8 zeigen die Messwerte auf einem Intel Xeon Dual Prozessor mit 2.8 GHz und 3,8 GB RAM. Auf dem System läuft ein *SUSE Linux* mit Kernel 2.6.8. Ebenso wie auf dem Pentium IV System, wurden auch hier alle Programmmodule mit dem GNU C-Compiler *gcc*, allerdings Version 3.3.4 kompiliert. Insgesamt lässt sich aus allen Abbildungen ein für 32 Bit

Pentium IV	RTL $\mathcal{A}$	RTL Bin $\mathcal{LD}$	LTR Bin $\mathcal{LD}$-$\mathcal{A}$	RTL NAF $\mathcal{LD}$	LTR NAF $\mathcal{LD}$-$\mathcal{A}$	Modi B-ary $\mathcal{LD}$	Modi B-ary $\mathcal{LD}$-$\mathcal{A}$	Montgomery
NIST Polynom B-163 – Spezielle Reduce Variante								
8 Bit	58,13	13,87	10,47	11,70	8,70	9,93	9,13	7,70
16 Bit	32,47	8,73	6,50	7,20	5,53	6,13	5,87	4,83
32 Bit	24,69	8,66	5,94	6,82	5,03	6,51	5,40	4,75
NIST Polynom B-233 – Spezielle Reduce Variante								
8 Bit	140,13	32,70	24,90	25,80	20,83	21,97	20,43	17,87
16 Bit	78,08	22,23	17,40	18,33	14,37	15,93	14,63	12,83
32 Bit	57,43	14,17	10,80	11,27	8,90	10,27	9,33	8,57
NIST Polynom B-163 – Allgemeine Reduce Variante								
8 Bit	69,77	54,00	45,03	52,03	40,17	41,13	35,73	34,80
16 Bit	45,33	53,43	47,77	45,43	38,10	41,00	35,80	34,90
32 Bit	40,03	54,50	47,47	45,33	40,23	43,77	37,73	39,97
NIST Polynom B-233 – Allgemeine Reduce Variante								
8 Bit	169,40	128,30	105,83	105,43	89,90	91,80	82,27	82,53
16 Bit	108,00	122,70	104,50	100,80	85,87	88,43	79,60	82,80
32 Bit	86,47	97,30	83,53	80,47	71,00	74,30	66,33	70,27

Tabelle A.1: *Timing-Vergleich der verschiedenen Punktmultiplikations-Algorithmen auf einem Intel Pentium IV 1.8 GHz (Zeiten in ms)*

Systeme charakteristisches Bild ablesen, welches ähnlich auch schon für den ST30 gesehen wurde.

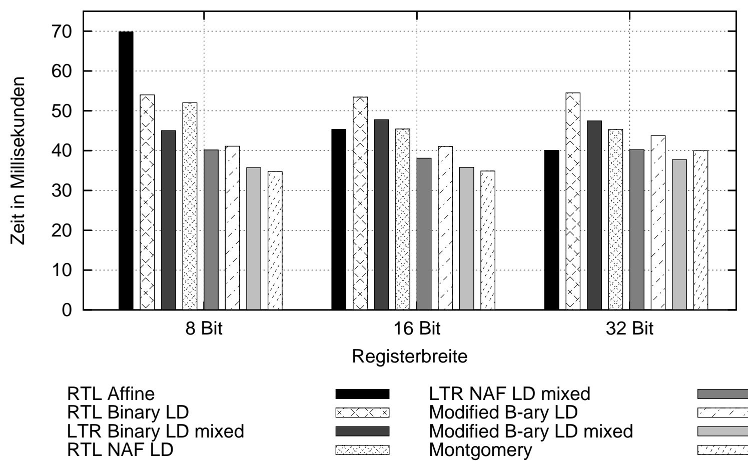

Abbildung A.2: *Punktmultiplikationsalgorithmen auf einem Intel Pentium IV 1,8 GHz für die Kurve B-163 unter der Verwendung eines allgemeinen Reduce-Algorithmus*

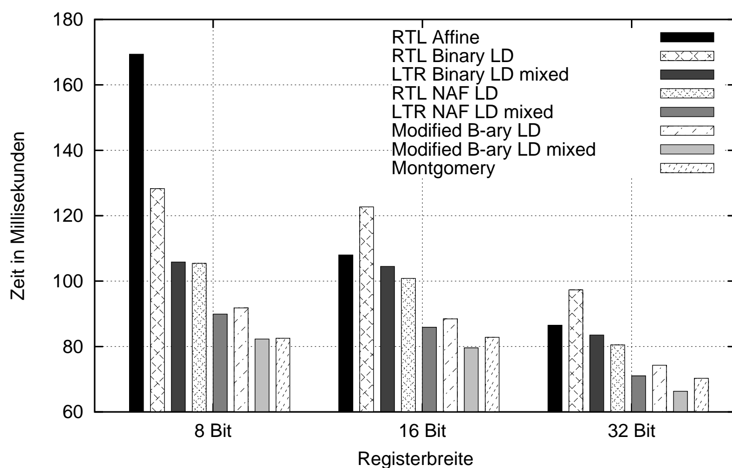

Abbildung A.3: *Punktmultiplikationsalgorithmen auf einem Intel Pentium IV 1,8 GHz für die Kurve B-233 unter der Verwendung eines allgemeinen Reduce-Algorithmus*

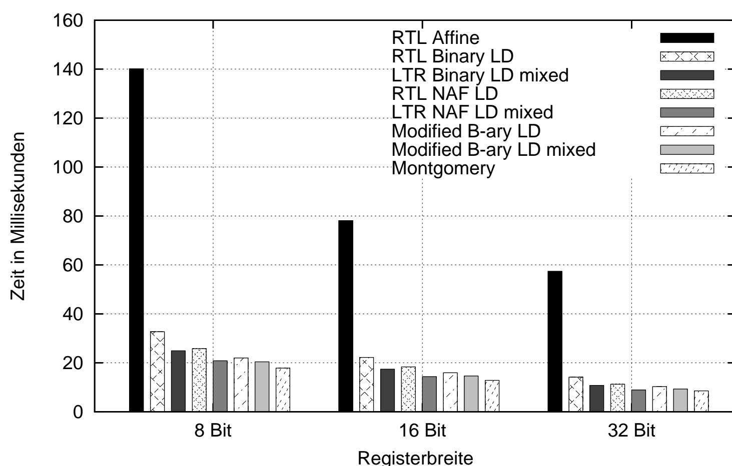

Abbildung A.4: *Punktmultiplikationsalgorithmen auf einem Intel Pentium IV 1,8 GHz für die Kurve B-233 unter der Verwendung eines speziellen Reduce-Algorithmus*

Xeon	RTL $\mathcal{A}$	RTL Bin $\mathcal{LD}$	LTR Bin $\mathcal{LD}$-$\mathcal{A}$	RTL NAF $\mathcal{LD}$	LTR NAF $\mathcal{LD}$-$\mathcal{A}$	Modi B-ary $\mathcal{LD}$	Modi B-ary $\mathcal{LD}$-$\mathcal{A}$	Montgomery
NIST Polynom B-163 – Spezielle Reduce Variante								
8 Bit	56,46	12,12	9,41	10,27	7,73	8,72	8,47	6,85
16 Bit	35,14	9,74	7,60	8,19	6,21	7,08	6,65	5,77
32 Bit	16,64	6,29	4,62	5,43	4,03	4,72	4,02	3,75
NIST Polynom B-233 – Spezielle Reduce Variante								
8 Bit	142,93	25,00	19,27	20,50	16,27	17,67	17,00	14,47
16 Bit	95,43	22,83	16,87	17,40	13,77	15,57	14,47	12,33
32 Bit	38,53	13,10	9,87	10,23	8,23	9,33	8,20	7,77
NIST Polynom B-163 – Allgemeine Reduce Variante								
8 Bit	64,39	45,25	37,69	38,14	33,49	35,40	30,65	29,16
16 Bit	45,27	41,03	34,50	33,93	29,07	28,07	27,33	27,63
32 Bit	29,57	42,40	36,43	35,43	31,03	33,33	29,07	31,33
NIST Polynom B-233 – Allgemeine Reduce Variante								
8 Bit	167,17	106,30	89,37	87,77	74,77	76,73	69,53	70,23
16 Bit	115,77	88,30	74,67	72,07	62,27	65,07	58,37	58,83
32 Bit	60,40	79,87	68,40	64,77	57,27	60,67	53,93	56,97

Tabelle A.2: *Timing-Vergleich der verschiedenen Punktmultiplikations-Algorithmen auf einem Intel Xeon Dual 2.8 GHz (Zeiten in ms)*

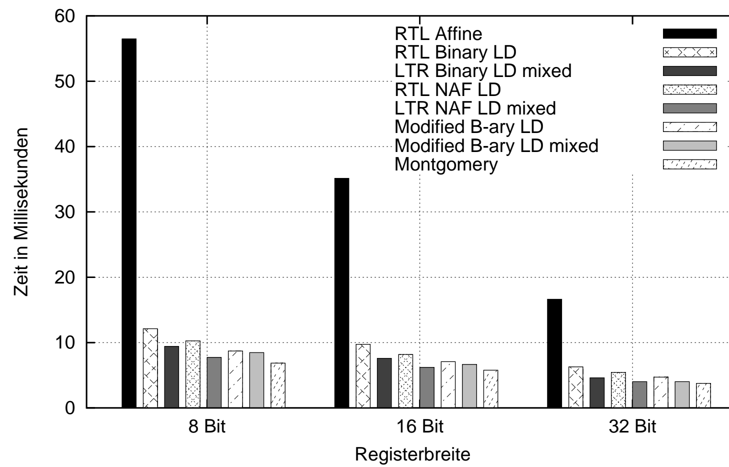

Abbildung A.5: *Punktmultiplikationsalgorithmen auf einem Intel Xeon 2,8 GHz für die Kurve B-163 unter der Verwendung eines speziellen Reduce-Algorithmus*

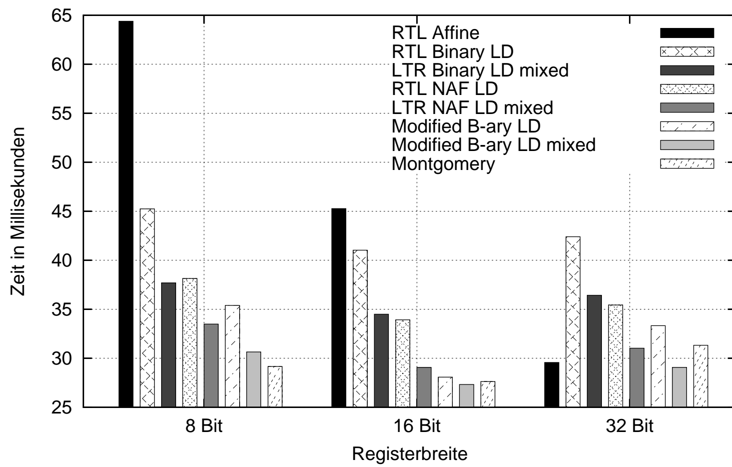

Abbildung A.6: *Punktmultiplikationsalgorithmen auf einem Intel Xeon 2,8 GHz für die Kurve B-163 unter der Verwendung eines allgemeinen Reduce-Algorithmus*

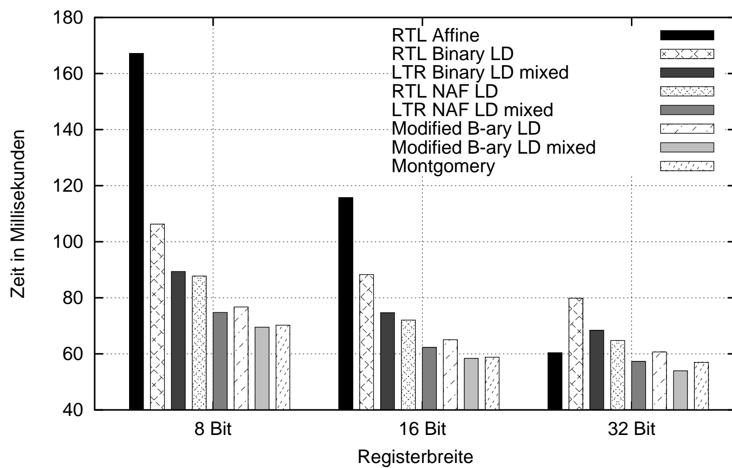

Abbildung A.7: *Punktmultiplikationsalgorithmen auf einem Intel Xeon 2,8 GHz für die Kurve B-233 unter der Verwendung eines allgemeinen Reduce-Algorithmus*

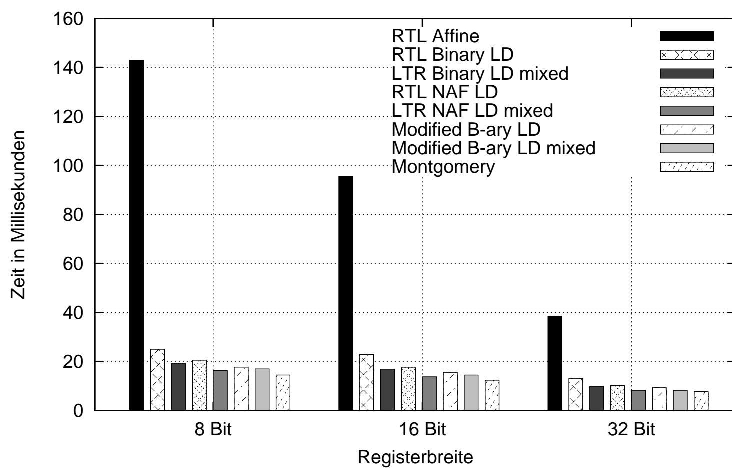

Abbildung A.8: *Punktmultiplikationsalgorithmen auf einem Intel Xeon 2,8 GHz für die Kurve B-233 unter der Verwendung eines speziellen Reduce-Algorithmus*

Anhang B

Binäre NIST Kurven

In diesem Abschnitt sind die Parameter der drei am häufigsten verwendeten binären NIST Kurven abgedruckt. Diese und weitere empfohlene Kurven finden sich im Digital Signature Standard [Nat00].
Die folgenden Parameter sind für jede der Kurven angegeben:

n	Grad der Körpererweiterung von $\mathbb{F}_{2^n}$.
$f(x)$	Reduktionspolynom vom Grad n.
a, b	Koeffizienten der Elliptischen Kurve $\mathcal{E} : y^2 + xy = x^3 + ax^2 + b$.
p	Primfaktor der Ordnung des Punktes $G \in \mathcal{E}$.
h	Der Kofaktor.
G_x, G_y	Die x- und y-Koordinate des Punktes G.

NIST B-163			
	n	=	163
	$f(x)$	=	$x^{163} + x^7 + x^6 + x^3 + 1$
	a	=	0x1
	b	=	0x00000002 0A601907 B8C953CA 1481EB10 512F7874 4A3205FD
	p	=	0x00000004 00000000 00000000 000292FE 77E70C12 A4234C33
	h	=	2
	G_x	=	0x00000003 F0EBA162 86A2D57E A0991168 D4994637 E8343E36
	G_y	=	0x00000000 D51FBC6C 71A0094F A2CDD545 B11C5C0C 797324F1

Tabelle B.1: *Binäre NIST Kurve B-163*

NIST B-233			
	n	=	233
	$f(x)$	=	$x^{233} + x^{74} + 1$
	a	=	0x1
	b	=	0x00000066
			647EDE6C 332C7F8C 0923BB58 213B333B 20E9CE42 81FE115F 7D8F90AD
	p	=	0x00000100
			00000000 00000000 00000000 0013E974 E72F8A69 22031D26 03CFE0D7
	h	=	2
	G_x	=	0x000000FA
			C9DFCBAC 8313BB21 39F1BB75 5FEF65BC 391F8B36 F8F8EB73 71FD558B
	G_y	=	0x00000100
			6A08A419 03350678 E58528BE BF8A0BEF F867A7CA 36716F7E 01F81052

Tabelle B.2: *Binäre NIST Kurve B-233*

NIST B-283			
	n	=	233
	$f(x)$	=	$x^{283} + x^{12} + x^7 + x^5 + 1$
	a	=	0x1
	b	=	0x27B680A C8B8596D
			A5A4AF8A 19A0303F CA97FD76 45309FA2 A581485A F6263E31 3B79A2F5
	p	=	0x03FFFFFF FFFFFFFF
			FFFFFFFF FFFFFFFF FFFFEF90 399660FC 938A9016 5B042A7C EFADB307
	h	=	2
	G_x	=	0x5F93925 8DB7DD90
			E1934F8C 70B0DFEC 2EED25B8 557EAC9C 80E2E198 F8CDBECD 86B12053
	G_y	=	0x3676854 FE24141C
			B98FE6D4 B20D02B4 516FF702 350EDDB0 826779C8 13F0DF45 BE8112F4

Tabelle B.3: *Binäre NIST Kurve B-283*

Literaturverzeichnis

Bücher, Artikel, Tagungsberichte

[ADMRK02] Al-Daoud, Essame ; Mahmod, Ramlan ; Rushdan, Mohammad ; Kilicman, Adem: A New Addition Formula for Elliptic Curves over GF(2^n). In: *IEEE Transactions on Computers* 51 (2002), Nr. 8, S. 972–975

[Art93] Artin, Michael: *Algebra*. Basel : Birkhäuser Verlag, 1993

[BGL96] Blake, Ian F. ; Gao, Shuhong ; Lambert, Robert J.: Construction and Distribution Problems for Irreducible Trinomials over Finite Fields. In: *Applications of Finite Fields*. Oxford-Clarendon Press, 1996, S. 19–32

[BHLM01] Brown, Michael ; Hankerson, Darrel ; López, Julio ; Menezes, Alfred: Software Implementation of the NIST Elliptic Curves Over Prime Fields. In: *Lecture Notes in Computer Science* 2020 (2001), S. 250–

[Bos04] Bosch, Siegfried: *Algebra*. 5. Heidelberg : Springer, 2004

[BRS98] Blake, Ian F. ; Roth, Ron M. ; Seroussi, Gadiel: Efficient Arithmetic in GF(2^n) through Palindromic Representation / HP Laboratories. 1998 (HPL-98-134). – Forschungsbericht

[Brö04] Bröcker, Theodor: *Lineare Algebra und Analytische Geometrie*. 2. Basel : Birkhäuser, 2004

[BSS99] Blake, Ian ; Seroussi, Gadiel ; Smart, Nigel: *Elliptic Curves in Cryptography*. Cambridge : Cambridge University Press, 1999

[Buc04] Buchmann, Johannes: *Einführung in die Kryptographie*. 2. Berlin : Springer-Verlag, 2004

[Can89] Cantor, David G.: On arithmetical algorithms over finite fields. In: *J. Comb. Theory Ser. A* 50 (1989), Nr. 2, S. 285–300

[CF05] Cohen, Henri ; Frey, Gerhard: *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. London : CRC-Press, 2005

[CQS99] Ciet, Mathieu ; Quisquater, Jean-Jacques ; Sica, Francesco: *A Short Note on Irreducible Trinomials in Binary Fields*. 1999. – Universite catholique de Louvain, Crypto Group

[DGP+05] DIJK, Marten van ; GRANGER, Robert ; PAGE, Dan ; RUBIN, Karl ; SILVERBERG, Alice ; STAM, Martijn ; WOODRUFF, David: Practical Cryptography in High Dimensional Tori. In: *Advances in Cryptology - EUROCRYPT 2005, LNCS **3494***. Berlin : Springer-Verlag, 2005, S. 234–250

[DH76] DIFFIE, Whitfield ; HELLMAN, Martin E.: New Directions in Cryptography. In: *IEEE Transactions on Information Theory* IT-22 (1976), November, S. 644–654

[DK01] DELFS, Hans ; KNEBL, Helmut: *Introduction to Cryptography*. Berlin : Springer-Verlag, 2001

[DWBV96] DE WIN, Erik ; BOSSELAERS, Antoon ; VANDENBERGHE, Servaas: A Fast Software Implementation for Arithmetic Operations in GF(2^n). In: *Advances in Cryptology - Asiacrypt '96, LNCS **1163***. Springer-Verlag, 1996, S. 65–76

[DWMPM98] DE WIN, Erik ; MISTER, Serge ; PRENEEL, Bart ; MICHAEL, Wiener: On the Performance of Signature Schemes based on Elliptic Curves. In: *Algorithmic Number Theory*. Springer-Verlag, 1998 (Proceedings 3rd International Symposium, LNCS **1423**), S. 252–266

[ElG85] ELGAMAL, Taher: A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In: *IEEE Transactions on Information Theory* IT-31 (1985), Nr. 4, S. 469–472

[Fis94] FISCHER, Gerd: *Ebene algebraische Kurven*. Wiesbaden : Vieweg, 1994

[GG03] GATHEN, Joachim von z. ; GERHARD, Jürgen: *Modern Computer Algebra*. 2. Cambridge : Cambridge University Press, 2003

[GL92] GAO, Shuhong ; LENSTRA, Hendrik W.: Optimal normal bases. In: *Designs, Codes and Cryptography* 2 (1992), S. 315–323

[Gor98] GORDON, Daniel M.: A Survey of Fast Exponentiation Methods. In: *J. Algorithms* 27 (1998), Nr. 1, S. 129–146

[Gua97] GUAJARDO, Jorge: *Efficient Algorithms for Elliptic Curve Cryptosystems*. 1997

[HHM01] HANKERSON, Darrel ; HERNANDEZ, Julio L. ; MENEZES, Alfred: Software Implementation of Elliptic Curve Cryptography over Binary Fields. In: *Lecture Notes in Computer Science* 1965 (2001), S. 1–24

[HJS93] HARPER, Greg ; J., Menezes A. ; SCOTT, Vanstone: Public-Key Cryptography with Very Small Key Length. In: *Advances in Cryptology - EUROCRYPT '92, LNCS **658***. Berlin : Springer-Verlag, 1993, S. 163–173

[HMV04] HANKERSON, Darrel ; MENEZES, Alfred ; VANSTONE, Scott: *Guide to Elliptic Curve Cryptography*. New York : Springer-Verlag, 2004

[HPCTZ99] HAN, Yongfei ; PENG-CHOR, Leong ; TAN, Peng-Chong ; ZHANG, Jiang: Fast Algorithms for Elliptic Curve Cryptosystems over Binary Finite Field. In: *Advances in Cryptology - Asiacrypt '96, LNCS **1716***. Springer-Verlag, 1999, S. 75–85

[Hus04] HUSEMÖLLER, Dale: *Elliptic Curves*. 2. New York : Springer-Verlag, 2004 (Graduate Texts in Mathematics 111)

[JT01] Joye, Marc ; Tymen, Christophe: Compact Encoding of Non-Adjacent Forms with Applications to Elliptic Curve Cryptography. In: *Public Key Cryptography, LNCS **1992***. Springer-Verlag, 2001, S. 353–364

[JY00] Joye, Marc ; Yen, Sung-Ming: Optimal Left-to-right Binary Signed-Digit Recording. In: *IEEE Transactions on Computers, **49***. Springer-Verlag, 2000, S. 740–748

[Kne04] Knebl, Helmut: *Kryptographie mit Elliptischen Kurven - Vorlesungsskript*. Nürnberg, July 2004. – Georg-Simon-Ohm Fachhochschule Nürnberg

[Knu81] Knuth, Donald D.: *The Art of Computer Programming - Seminumerical Algorithms*. 2. Massachusetts : Addison-Wesley, 1981

[Kob98] Koblitz, Neal: *Algebraic aspects of cryptography*. Berlin : Springer-Verlag, 1998

[Lan04] Lange, Tanja: *A Note on López Dahab coordinates*. Ruhr-University of Bochum, 2004. – Information-Security and Cryptography

[LD99a] López, Julio ; Dahab, Ricardo: Fast multiplication on elliptic curves over GF(2^n) without precomputation. In: *Cryptographic Hardware and Embedded Systems*, Springer-Verlag, 1999, S. 316–327

[LD99b] López, Julio ; Dahab, Ricardo: Improved Algorithms for Elliptic Curve Arithmetic in $\mathbb{F}_{2^n}$. In: *Selected Areas in Cryptography - SAC'98, LNCS **1556***, Springer-Verlag, 1999, S. 201–212

[LD00] López, Julio ; Dahab, Ricardo: *High-Speed Software Multiplication in* $\mathbb{F}_{2^n}$. Sao Paulo, May 2000. – Institute of Computing, State University of Campinas

[LN83] Lidl, Rudolf ; Niederreiter, Harald: *Encyclopedia of Mathematics and Its Applications - Finite Fields*. Bd. 20. London : Addison-Wesley, 1983

[LV00] Lenstra, Arjen K. ; Verheul, Eric R.: The XTR public key system. In: *Advances in Cryptology - CRYPTO 2000, LNCS **1880***. Berlin : Springer-Verlag, 2000, S. 1+

[MBG+93] Menezes, Alfred J. ; Blake, Ian F. ; Gao, XuHong ; Mullin, Ronald C. ; Vanstone, Scott A. ; Yaghoobian, Tomik: *Applications of Finite Fields*. Boston : Kluwer, 1993

[Mey75a] Meyberg, Kurt: *Algebra I*. München : Carl Hanser Verlag, 1975

[Mey75b] Meyberg, Kurt: *Algebra II*. München : Carl Hanser Verlag, 1975

[Mil86] Miller, Victor S.: Use of Elliptic Curves in Cryptography. In: *Advances in Cryptology - CRYPTO '85, LNCS **218***. Berlin : Springer-Verlag, 1986, S. 417–426

[MO90] Morain, François ; Olivos, Jorge: Speeding Up the Computations on an Elliptic Curve Using Addition-Subtraction Chains. In: *RAIRO: Informatique Theorique et Applications/Theoretical Informatics and Applications* 24 (1990)

[MOV93] Menezes, Alfred J. ; Okamato, T. ; Vanstone, S.A.: Reducing elliptic curve logarithms to logarithms in a finite field. In: *IEEE Transactions on Information theory* 39 (1993), S. 1639–1646

[MOV96] Menezes, Alfred J. ; Oorschot, Paul C. ; Vanstone, Scott A.: *Handbook of applied Cryptography*. London : CRC-Press, 1996

[Nat00] National Institute of Standards and Technology: *Digital Signature Standard (DSS)*. January 2000. – FIPS PUB 186-2

[Nea87] Neal, Koblitz: Elliptic Curve Cryptosystems. In: *Mathematics of Computation* 48 (1987), S. 203–209

[OM86] Omura, Jimmy K. ; Massey, James L.: *Computational method and apparatus for finite field arithmetic*. May 1986. – U.S. Patent Nr. 4.587.627

[RD02] Rijmen, Vincent ; Daemen, Joan: *The Design of Rijndeal, AES - The Advanced Encryption Standard*. Berlin : Springer-Verlag, 2002

[Rei60] Reitwiesner, George W.: Binary Arithmetic. In: *Advances in Computers* 1 (1960), S. 231–308

[Ros99] Rosing, Michael: *Implementing Elliptic Curve Cryptography*. Greenwich : Manning, 1999

[RSA78] Rivest, R. L. ; Shamir, A. ; Adelman, L. M.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. In: *Communications of the ACM* 21 (1978), February, S. 120–126

[Sem98a] Semaev, Igor A.: An Algorithm for evaluation of discrete logarithms in some nonprime finite fields. In: *Mathematics of Computation* 67 (1998), October, S. 1679–1689

[Sem98b] Semaev, Igor A.: Evaluation of discrete logarithms in a group of p–torsion points of an elliptic curve in characteristic p. In: *Mathematics of Computation* 67 (1998), January, S. 353–356

[Ser98] Seroussi, Gadiel: Table of Low-Weight Binary Irreducible Polynomials / Computer Systems Laboratory. 1998 (HPL-98-135). – Forschungsbericht

[SHG00] Smart, Nigel ; Hess, Florian ; Gaudry, Pierrick: Constructive and Destructive Facets of Weil Descent on Elliptic Curves / HP Laboratories Bristol. 2000 (HPL-2000-10). – Forschungsbericht

[Sil86] Silverman, Joseph H.: *The Arithmetic of Elliptic Curves*. New York : Springer-Verlag, 1986

[Sin03] Singh, Simon: *Geheime Botschaften*. 4. München : Deutscher Taschenbuch Verlag, 2003

[Sol97] Solinas, Jerome A.: An Improved Algorithm for Arithmetic on a Family of Elliptic Curves. In: *Advances in Cryptology - CRYPTO '97, LNCS **1294***. Springer-Verlag, 1997, S. 357–371

[SOO95] Schroeppel, Richard ; Orman, Hilarie ; O'Malley, Sean: Fast Key Exchange with Elliptic Curve Systems. In: *Lecture Notes in Computer Science* 963 (1995), S. 43–

[Swa62] Swan, Richard G.: Factorization of Polynomials over Finite Fields. In: *Pacific J. Math.* (1962), S. 1099–1106

[The04] The Motor Industry Software Reliability Association: *MISRA-C:2004 - Guidelines for the use of the C language in critical systems*. Nuneaton : MISRA Limited, 2004. – Siehe auch: http://www.misra-c2.com. – ISBN 0952415623

[Was03] Washington, Lawrence C.: *Elliptic Curves - Number Theory and Cryptography*. 1. Boca Raton : Chapman & Hall/CRC, 2003

[Wer02] Werner, Anette: *Elliptische Kurven in der Kryptographie*. Berlin : Springer-Verlag, 2002

[WP02] Weimerskirch, André ; Paar, Christoph: *Generalizations of the Karatsuba Algorithm for Polynomial Multiplication*. Ruhr-University Bochum, 2002. – Communication Security Group - Department of Electrical Engineering & Information Sciences

[WSCS03] Weimerskirch, André ; Stebila, Douglas ; Chang Shantz, Sheueling: *Generic GF(2^m) Arithmetic in Software and its Application to ECC*. Wollongong, Australia : http://www.douglas.stebila.ca/research/papers/WSC03.pdf, July 2003. – The Eight Australasian Conference on Information Security and Privacy (ACISP 2003)

Internetquellen

[1] Altium Limited. TASKING - embedded software development tools from altium. http://www.altium.com/tasking/. Stand: (16.08.2005).

[2] ARM. ARM developer suite. http://www.arm.com/products/DevTools/ADS.html. Stand: (30.07.2005).

[3] Certicom Corp. Certicom - strong efficient cryptography for device manufacturers and software vendors. http://www.certicom.com. Stand: (21.07.2005).

[4] Infineon Technologies. Infineon technologies. http://www.infineon.com/de. Stand: (01.07.2005).

[5] Lauterbach GmbH. Trace32 microprocessor development tool, emulator, debuggers, simulators. http://www.lauterbach.com/frames.html. Stand: (30.07.2005).

[6] STMircroelectronics. Firmenhomepage. http://www.st.com. Stand: (01.07.2005).

Stichwortverzeichnis

A

Advanced Enryption Standard 18
AES . . . . . *siehe* Advanced Enryption Standard
affine algebraische Varietät 52
algebraische Kurve . . . . .
- ebene 51, 52
- Gerade 53
- Grad 53
- irreduzibel 53
- irreduzible Komponente 53
- kubisch 53
- Minimalpolynom 53
- reduzibel 53

B

Baby-Step Giant-Step Algorithmus 103

C

C167 . . . . . *siehe* Prozessor

D

Diffie-Hellman Problem 104
Digitaler Signatur Standard 106
Diskretes Logarithmus Problem 103
Diskriminante 62
divide-and-conquer 28
double and add 75

E

ECC . . . . . *siehe* Elliptic Curve Cryptography
ECDLP 75
ECDSA 103, 106
ECSTR . . . . . *siehe* XTR
Elliptic Curve Cryptography 2

G

Gradform 55
Gruppe 5
- abelsch 5
- Generator 6
- kommutativ 5
- Ordnung 6
- symmetrisch 6
- zyklisch 6

H

Halbspur 13
Hamming-Gewicht 19
Hornerschema 26
hyperelliptische Kurven 109

I

Inversenbildung . . . . .
- Almost Inverse Algorithmus 39
- Erweiterter Euklidischer Algorithmus . 36
- Euklidischer Algorithmus 36

Inverses Element 35
Isomorphismus 57

J

j-Invariante 62

K

Körper 8
- algebraisch abgeschlossen 9
- Charakteristik 8
- Erweiterungskörper 8
- Grad 8
- Körpererweiterung 8
- Teilkörper 8

Konjugiertes 17
Koordinaten . . . . .
- affin 75
- homogen 54
- jakobisch 77
- López-Dahab 57, 78
- projektiv 76

koordinatenunabhängig 56

L

Leitform 55

M

MISRA Regeln 29
Morphismus 57
MOV-Attacke 63
Multiplikation ....
  Karatsuba-Ofman 28
  Left-to-right comb 25
  Left-to-right comb Window 26
  Right-to-left comb 24
  shift-and-add 23

N

NAF .... *siehe* Zahlendarstellung
Normalbasis 17

O

ONB .... *siehe* Optimale Normalbasis
Optimale Normalbasis 13

P

Pentanom .... *siehe* Polynom
Pollard ρ Methode 103
Polynom 9
  Ableitung 11
  Addition 10
  führender Koeffizient 9
  Grad 9
  homogen 55
  irreduzibel 10
  Leitkoeffizient 9
  Monom 9
  Nullstelle 9
  Pentanom 34
  Produkt 10
  Trinom 34
Polynomielle Darstellung 12, 15
Polynomring 10
projektive Ebene 54
projektive Gerade 54
projektiver Abschluss 57
Prozessor ....
  C167 .... 42, 93
  ST30 .... 43, 93, 95
Public-Key-Verfahren ....
  Diffie-Hellman 1, 103
  ElGamal 1, 105
  RSA 1, 110
  XTR 109
Punktmultiplikation ....
  Left-to-right B-ary 89
  Left-to-right NAF 86
  Modified B-ary 90
  Montgomery 91, 92
  Right-to-left NAF 87

Q

Quantenkryptographie 109

R

rationale Punkte 61
regulär 59
Restklassenring 7
Reziprokes .... *siehe* Inverses Element
Ring 7
  Einheit 7
  Einselement 7
  euklidisch 36
  Integritätsbereich 7
  kommutativ 7
  Neutrales Element 7
  Nullteiler 7
RSA .... *siehe* Public-Key-Verfahren

S

Schnittmultiplizität 58
Sekanten-Tangenten-Methode 65
Signed Digit .... *siehe* Zahlendarstellung
Silver-Pohling-Hellman Verfahren 103
singulär 59
Singuläritāt 59
Singularität 59
Sliding Window 90
Spurfunktion 13
ST30 .... *siehe* Prozessor
supersingulär 103
Symmetrische Gruppe .... *siehe* Gruppe

T

transzendente Kurve 51
trap-door one-way functions 1
Trinom .... *siehe* Polynom

U

Unendlich ferner Punkt 54, 61

V

Vektordarstellung 15, 72
Vektorraum 8
Vielfachenbildung .. *siehe* Punktmultiplikation
Vielfachheit .... *siehe* Schnittmultiplizität

W

Weierstrass-Gleichung 60
Wendepunkt 60
Wendetangente 60

X

XTR .... *siehe* Public-Key-Verfahren

Z

Zahlendarstellung ....
  B-ary Darstellung 88
  Non-Adjacent Form 85
  Signed Digit 85
Zykelschreibweise 6

Eigene Notizen