Diplomarbeit

Komplexitätsabschätzung von hardwareakzelerierten Attacken auf ECC-Kryptoverfahren

Markus Böttger

m-boettger@m-boettger.de

Universität Hamburg - Fachbereich Informatik

Arbeitsbereich Technische Grundlagen der Informatik

Januar 2002

Betreut von:

Dr. habil. Reinhard Rauscher

Prof. Dr.-Ing. Karl Kaiser

Inhaltsverzeichnis

1 Kryptographie **1**

2 Motivation / State Of The Art **4**

3 Mathematische Grundlagen **6**

3.1 Gruppe und Körper 6

3.2 Endliche Körper 7

3.3 Elliptische Kurven 10

3.3.1 Arithmetik mit elliptischen Kurven 10

3.3.2 Elliptische Kurven über $\mathcal{GF}(2^m)$ 12

3.3.3 Affine und Projektive Koordinaten 17

4 Kryptosysteme mittels elliptischer Kurven **20**

4.1 Falltürfunktionen 20

4.2 Die Diffie-Hellmann-Schlüsselvereinbarung 21

4.3 Das ElGamal-Verschlüsselungsverfahren 22

4.4 Praktische Anwendungen 22

5 Angriff auf ein ECC-basiertes Kryptosystem **25**

5.1 Angriffsmethoden und Schwachstellen 25

5.2 Kryptosysteme mit skalierbarer Sicherheit 27

5.3 Brute-Force-Methode mittels Punktaddition 30

5.4 Brute-Force-Methode mittels Punktverdoppelung 33

5.5 Die Methoden von Pollard und Shanks 36

5.6 Das gewählte Angriffsszenario 39

5.6.1 Beschreibung der Strategie 39

5.6.2 Beispiel eines Angriffs 40

5.6.3 Komplexität und Erweiterung des Verfahrens 43

6 Implementierung **46**

6.1 Das C++ Modell 46

6.2 Das VHDL-Modell 49

6.2.1 Der Datenfluß 49

6.2.2 Der VHDL-Code 57

6.2.3 Simulation eines Beispiels 60

6.2.4 Suchen unter Berücksichtigung mehrerer mitgehörter Punkte . . . 63

6.3 Der FPGA-Baustein 64

7 Ergebnisse **66**
7.1 Der Synthesevorgang 66
7.2 Ergebnis der Synthese eines Multiplizierers 67
7.3 Ergebnis der Synthese des zentralen Entwurfes 67
7.4 Analyse der Syntheseresultate 70

8 Ausblick und nicht behandelte Aspekte **78**
8.1 Generelle Fragestellungen 78
8.2 Konzeptionelle Angriffsalternativen 79
8.3 Implementierungsalternativen 80
8.4 Anwendungsmöglichkeiten dieser Arbeit 81

A Literatur **83**

B Algorithmen **85**
B.1 C++ - Algorithmus zur Brute-Force Attacke 85
B.2 VHDL-Modell des Punktverdoppelungs-FPGA 89
B.2.1 Der zentrale Entwurf 89
B.2.2 Die Testumgebung 93

C Inhalt der CD **95**

D Eidesstattliche Erklärung **97**

E Danksagung **98**

F Index **99**

Abkürzungsverzeichnis

Folgende Standardabkürzungen werden in dieser Arbeit verwendet:

ALU	-	Arithmetic Logic Unit
ASIC	-	Application Specific IC
BSGS	-	Baby Step / Giant Step - Methode
BSI	-	Bundesamt für Sicherheit in der Informationstechnik
CAE	-	Computer Aided Engineering
DLP	-	Discrete Logarithm Problem
DSA	-	Digital Signature Algorithm
DRC	-	Design Rule Check
EC	-	Elliptic Curve
ECC	-	Elliptic Curve Cryptosystem
EDIF	-	Electronic Design Interchange Format
FPGA	-	Field Programmable Gate Array
FSM	-	Finite State Machine
GSM	-	Global System for Mobile Communication
IC	-	Integrated Circuit
IFP	-	Integer Factorization Problem
LE	-	Logic Element
LSB	-	Least Significant Bit
LUT	-	Look Up Table
MUX	-	Multiplexer
MSB	-	Most Significant Bit
PDA	-	Personal Digital Assistant
VHDL	-	VHSIC Hardware Description Language
VHSIC	-	Very High Speed IC
VPN	-	Virtual Private Networks
WAP	-	Wireless Application Protocol

Abbildungsverzeichnis

1 Die ENIGMA-Chiffriermaschine 2
2 Elliptische Kurve $y^2 = x^3 - 5x + 2$ 11
3 Addition zweier Punkte auf der Elliptischen Kurve 12
4 $P + (-P) = \mathcal{O}$ 12
5 Verdoppelung eines Punktes auf einer Elliptischen Kurve 13
6 Graph einer elliptischen Kurve über $\mathcal{GF}(2^m)$ 16
7 Die Diffie-Hellmann-Schlüsselvereinbarung 21
8 Das ElGamal-Verschlüsselungsverfahren 23
9 Informationen eines Angreifers 26
10 Schlüsselbitbreite von RSA und ECC von 1982 bis 2050 29
11 Indexierte Entwicklung der Schlüsselbitbreiten von 2000 bis 2010 30
12 Das Pollard - Rho 37
13 Das Pollard - Lambda 38
14 Ablaufschema der Verdoppelungs-Strategie 41
15 Erfolgswahrscheinlichkeiten des Suchens mittels mehrerer Punkte 45
16 verdopple() - Routine des C++-Modells 50
17 Prinzip einer Testumgebung 51
18 Datenflussgraph der projektiven Punktverdoppelung 52
19 Ablaufschema des VHDL-Modells 54
20 Datenflussgraph des FPGA 56
21 Schaltbild des Modulo-2 Multiplizierers 58
22 Signalverlauf einer Simulation mit 3 Bit 61
23 Ablaufschema des Suchens mit mehreren Punkten 64
24 Bedarf eines Multiplizierers an Standardgattern 68
25 Benötigte Logikelemente für den zentralen Entwurf 69
26 Maximale Taktrate des FPGA 70
27 Nötige Verdoppelungen und Takte für $P(A_{2^{20},s,r}) = 0{,}5$ 73
28 Vergleich der Zeitaufwendungen für $P(A_{\#E,s,r}) = 0{,}5$ 76
29 FPGA-Kosten in Abhängigkeit von der Bitbreite 77

Tabellenverzeichnis

1 Wertetabelle der elliptischen Beispielkurve 15
2 Sukzessive Addition eines Basispunktes 33
3 Verschiedene Gruppen der Punktordnung 33
4 Sukzessive Verdoppelung eines Basispunktes 34
5 Fortschrittvergleich der Verfahren 35
6 Erreichen der 50% und 80% Erfolgswahrscheinlichkeiten 45
7 Allgemeiner Gatterbedarf eines Multiplizierers (vgl. Abb. 24) 67
8 Bedarf des zentralen Entwurfs an Logikelementen (vgl. Abb. 25) 68
9 Synthesezeitaufwand und Anzahl der Flipflops des zentralen Entwurfs . . 68
10 Maximale Taktrate des FPGA (vgl. Abb. 26) 69
11 Verdoppelungen pro Sekunde 70
12 Zeitaufwand zum Erreichen von $P(A_{2^{20},s,r}) = 0{,}5$ 72
13 Zeitaufwand zum Erreichen von $P(A_{2^{20},s,r}) = 0{,}8$ 72
14 Zeit zum Erreichen von $P(A_{2^{30},s,r}) = 0{,}5$ und $P(A_{2^{30},s,r}) = 0{,}8$ 74
15 Zeit zum Erreichen von $P(A_{2^{40},s,r}) = 0{,}5$ und $P(A_{2^{40},s,r}) = 0{,}8$ 75
16 Zeit zum Erreichen von $P(A_{2^{50},s,r}) = 0{,}5$ und $P(A_{2^{50},s,r}) = 0{,}8$ 75
17 Rahmendaten und Preise einiger FPGA der Altera Flex10K Familie . . . 76

Algorithmenverzeichnis

1 Punktaddition durch Skalarmultiplikation 31
2 Punktaddition durch sukzessive Addition 32
3 Vektormultiplikation Modulo 2 in C++ 48
4 Vektormultiplikation Modulo 2 in VHDL 58

1 Kryptographie

Das Schreiben eines Briefes, den niemals jemand anderes als der eine, gewünschte Leser lesen kann, ist ein sehr alter und häufiger Wunsch. Zwei sich sehr nahe stehende Menschen sind vielleicht in der Lage, eine Ausdrucksform ihrer Kommunikation zu finden die für Außenstehende unverständlich ist, aber auch das ist nicht uninterpretierbar.

Die älteste, uns überlieferte Form eine Nachricht zu verschlüsseln, ist wohl die der Spartaner, mit der sie schon im fünften Jahrhundert vor Christus Nachrichten chiffriert haben. Sie wickelten einen Papierstreifen spiralförmig um einen Stab, der „Skytala“ genannt wurde. Die parallel zum Stab auf dem Papierstreifen geschriebene Nachricht erschien sinnlos, wenn der Papierstreifen abgenommen wurde. Wurde der Papierstreifen jedoch an seinem Bestimmungsort auf einen Stab des gleichen Durchmessers gewickelt, so konnte die verschlüsselte Information leicht entziffert werden [Hor85].

Eine Weiterentwicklung stellte die sog. Steganographie dar, die Informationen durch Unsichtbarmachen der Daten mit Hilfe chemischer Mittel geheim machte. Dabei wurden die geheimen Nachrichten zwischen den Zeilen einer völlig unverfänglichen Nachricht geschrieben, wobei man „unsichtbare“ Tinte benutzte.

„Kryptographie“ entstammt dem griechischem Wortstamm $\kappa\rho\upsilon\pi\tau\sigma\varsigma =$ verborgen, $\gamma\rho\alpha\phi\epsilon\iota\nu$ = schreiben und wurde vor allem beim Nachrichtenaustausch zwischen Militärs, Staaten im diplomatischen Bereich, Terroristen etc. benutzt. Alle Verfahren dienen der Geheimhaltung von Daten oder Nachrichten. Es ist nicht weiter verwunderlich, daß gleichzeitig mit der Einführung der Kryptographie die Suche nach Verfahren zum Brechen der verwendeten Geheimschriften, der sog. Kryptoanalyse, begonnen wurde.

Nach [Hor85] ist der Italiener Leon Battista Alberti (1404-1472) als Vater der Kryptologie anzusehen, dessen Buch von 1466 das erste erhaltene Werk über die Kryptographie ist. Im 16. Jahrhundert nahmen die Bemühungen, sichere Verfahren zur Chiffrierung zu finden, als auch die Suche nach Verfahren diese zu brechen, stark zu. Die Kunst der Geheimschriften entwickelte sich zur Wissenschaft der Kryptologie. John Wallis erhielt 1649, in Anerkennung seiner Dienste als Dechiffrierer, eine Professur in Oxford.

Erst im neunzehnten Jahrhundert entwickelte sich die Kryptologie dank neuer Impulse rasch weiter. 1854 wurde von Charles Wheatstone die sog. Bigrammverschlüsselung eingeführt, die so einfach sein sollte, „so daß sie sogar Diplomaten zugemutet werden konnte“. Fälschlicherweise wird das Wheatston‘sche Verfahren oft Lord Lyon Playfair, dessen Namen es trägt, zugeschrieben. Hauptsächlich wurde daran gearbeitet Chiffren zu brechen. 1863 zeigte der preußische Infanteriemajor Friedrich Kasiski, wie man spezielle Chiffren mit sich periodisch wiederholendem Schlüsselwort brechen kann. Dazu benutzte er eine Häufigkeitsanalyse der verwendeten Klartextsprache. Auguste Kerkhoff dechiffrierte 1883 mehrere Nachrichten, die mit demselben Schlüssel chiffriert worden waren.

Demnach stellt [Hor85] drei Wendepunkte in der Kryptologie fest:
Der erste ist die Entwicklung mechanischer Verschlüsselungsgeräte Anfang des zwanzigsten Jahrhunderts. Meist benutzten diese Maschinen „polyalphabetische Substitutionschiffren" mit langen, periodischen Schlüsseln. Einige dieser Systeme wurden während des zweiten Weltkrieges gebrochen, so z.B. die in Abbildung 1 gezeigte deutsche ENIGMA, die hauptsächlich von Alan Turing entschlüsselt wurde.

Abbildung 1: Die ENIGMA-Chiffriermaschine

Gilbert Vernam entwickelte 1917 das „einzig sichere Kryptosystem" [Hor85]. Vernam benutzte einen Schlüssel, der völlig zufällig und ohne Wiederholung ist. Der Nachweis für die Sicherheit wurde von Claude Elwood Shannon, dem Begründer der Informationstheorie, erbracht. Shannon entwickelte während des zweiten Weltkrieges ein mathematisches Verfahren zur Bestimmung der Sicherheit von Verschlüsselungsverfahren.
Den zweiten Wendepunkt stellt das Aufkommen der ersten modernen Rechenalagen dar. Durch deren ständig wachsende Leistungsfähigkeit erweiterten sich die Anwendungsgebiete und vergrößerten die Wichtigkeit der Kryptographie und der Kryptoanalyse. Um unbefugten Zugriff zu vertraulichen und geheimen Informationen und den sich daraus ergebenen Schäden zu vermeiden, ist die Kryptographie notwendig.
Den dritten, wichtigen Wendepunkt stellt 1976 die Einführung des ersten Public-Key-Kryptosystems durch Diffie und Hellmann dar. Dieses Verfahren macht es überflüssig, daß die beiden Kommunikationsparter vor der ersten Nachrichtenübermittlung untereinander ihre speziellen Schlüssel austauschen. Dieses Verfahren wird im Kapitel 4.2 ab Seite 21 ausführlich dargelegt werden.
Bedenkt man, daß heutzutage neben Zahlungsverkehr und Kommunikation auch die ge-

samten Entwicklungs- und Fertigungsdaten sowohl von kommerziellen als auch militärischen Produkten auf Datenträgern gespeichert werden, so ist der Datenschutz durch kryptographische Verfahren unbedingt erforderlich, um einem unbefugten Zugriff vorzubeugen.

2 Motivation / State Of The Art

Die elektronische Datenverarbeitung hat in fast jedem Lebensbereich Einzug gehalten. In einigen Bereichen ist über die Begeisterung des rasanten Fortschritts leider der Sicherheitsaspekt vernachlässigt worden. Dies wird nun immer stärker zum Problem, da persönlichste Daten eines Jeden in verschiedensten Datenbanken gespeichert werden. In der überwiegenden Zahl der Anwendungen des täglichen Lebens wird den Benutzern nicht bewußt, daß sie mit sicherheitskritischen Datenbanken arbeiten, die teilweise in offenen Netzen operieren. Die Anwendungen, die mit diesen Datenbanken interagieren sind so benutzertransparent, daß Sicherheitsproblematiken für den Benutzer nicht ersichtlich werden. Sind diese Datenbanken etc. nur unzulänglich gegen unbefugten Zugriff geschützt, so lassen sich aus der Fülle solcher Datenbanken intimste Profile von Personen erstellen, die zu jedwedem Zweck ausgenutzt werden können.

Eine sehr große Rolle spielt in diesem Kontext das Problem der persönlichen Authentifizierung. Jeder Mensch muß sich gegenüber seinem gegen die Öffentlichkeit abgeschottetem Raum als Berechtigter zu erkennen geben. In den fünfziger Jahren war dafür der Schlüssel im herkömmlichen Sinne ein zuverlässiges Instrument. Man brauchte ihn, bzw. jeweils einen verschiedenen, z.B. zum Öffnen des Autos, der Wohnung, des Postkastens im Hausflur und des Schließfaches in der Bank. Wollte jemand in einen dieser Bereiche eindringen, so mußte er entweder versuchen das jeweilige Schloß mit einem Dietrich zu öffnen, oder den entsprechenden Originalschlüssel zwecks Duplizierung kurzfristig zu entwenden. Ersteres ist je nach Skalierung der Sicherheit des Schlosses möglich. Wie in Kapitel 5.2 bezogen auf elektronische Informationen noch diskutiert werden wird, ist dies für den Eigentümer ausschließlich eine Kosten/Nutzen-Frage. Den Originalschlüssel zwecks Duplizierung kurzfristig zu entwenden ist - gewaltlos betrachtet - nur durch Unvorsichtigkeit des Eigentümers möglich. Läßt er den Wohnungsschlüssel während der Mittagspause auf seinem Büroschreibtisch im Großraumbüro liegen, benötigt er auch kein extrem teures, schwer überwindbares Wohnungstürschloß. Der konventionelle Metallschlüssel kann nämlich eines nicht: die die Tür aufschließende Person als die Berechtigte einstufen. Hat ein Dieb einen Wohnungsschlüssel erfolgreich entwendet, kann er die Wohnung ausrauben, denn für die Wohnungtür ist die Person die Berechtigte, die den Schlüssel besitzt. Da bereits früh klar wurde, daß das bei höheren Sicherheitsstufen nicht mehr ausreicht, wurden Safes nicht nur mit einem Schlüsselschloß versehen, sondern zusätzlich mit einem Zahlenschloß.

Das Problem im heutigen Informationszeitalter ist, daß der o.g. persönliche „gegen die Öffentlichkeit abgeschottete Raum“ bei weitem nicht mehr geographisch definiert ist. Im Gegensatz zu den fünfziger Jahren ist der heutige persönliche Raum der Zugang zu Kontoauszügen, Bank- und Aktientransaktionen, Besitz- und Einkommensverhältnissen, Telefonverhalten, Online-Auktionen, dem Auto oder Bargeld. Der Schlüssel zu diesem

persönlichen Raum ist meist nicht mehr als eine vorgegebene, vierstellige Nummer. Dies muß hinsichtlich dem immer weiter wachsenden Online-Dienstleistungsverhalten der Gesellschaft grundlegend verbessert werden. Momentan benötigt der Benutzer zu jeder der Fülle der Heute durchführbaren Online-Aktionen einen Schlüssel, respektive ein Zugangskennwort. Da ein Benutzer die Gesamtheit seiner Zugangskennworte unmöglich im Gedächtnis verwalten kann, wie allerdings zumeist von den Systembetreibern gefordert, werden diese zunehmend „an sicherer Stelle“ notiert. Diese „sichere Stelle“ zu finden wird in Zukunft mehr und mehr das Ziel von Angreifern werden.
Ein in diesen Jahren sich verstärkend auswirkender Faktor ist, daß diverse Datenübertragungen durch Benutzung von Funk quasi öffentlich zugreifbar werden. Durch die Etablierung des Bluetooth-Standards werden Datenübertragungen auf kurzen Entfernungen bis zu zehn Metern, für die vorher noch Datenkabel benutzt wurden, per Funk durchgeführt. Zum Einsatz kommen alle Geräte, bei denen eine Datenübertragung überhaupt anfallen kann, z.B. die Verbindung des PDA (Personal Digital Assistant) mit dem Schreibtischrechner, Ohrhörer mit dem Handy, Schreibtischrechner mit der Telefonanlage oder der Drucker mit dem Laptop. Der Angreifer muß nur mit einer geeigneten Empfangseinrichtung in unmittelbarer Nähe stehen, um den Datenverkehr abhören zu können.
Noch einfacher ist der Angriff offensichtlich im Bereich der Funkfernbedienungen für Fahrzeugschließanlagen. Fängt ein Angreifer den Öffnungs- oder Schließimpuls mit einem Universalsender / -empfänger auf, so kann er den Originalsender / -empfänger imitieren. Nach Presseberichten ist ein solches Universalgerät bereits für unter 500 Euro erhältlich.
Durch die o.g. Umstände besteht die Notwendigkeit eines möglichst sicheren Verschlüsselungssystems bei möglichst geringem Verschlüsselungsaufwand. Die Verschlüsselung mittels elliptischer Kurven scheint ein geeignetes Mittel für diese Anforderung zu sein. Durch den bisher niedrigen Forschungsstand dieses Gebietes können jedoch noch keine konkreten Aussagen über die exakte Sicherheit gemacht werden.
Um diesem Mangel entgegenzuwirken, soll in dieser Arbeit untersucht werden, wie effektiv ein Angriff auf ein elliptische Kurven Kryptosystem (ECC) mit Hilfe von spezieller Hardware sein kann.

3 Mathematische Grundlagen

3.1 Gruppe und Körper

Für den Einstieg in die Thematik der elliptischen Kurven werden die Grundbegriffe der Gruppe und des Körpers benötigt.

Definition: Eine Menge G bildet bzgl. einer Verknüpfung $\circ$ eine *Gruppe* $(G, \circ)$, wenn die Axiome (G1) bis (G4) erfüllt sind. Ist zusätzlich (G5) erfüllt, so liegt eine *kommutative* oder *abelsche Gruppe* vor:

(G1)	$\forall\, a,b\; \exists\, c : c = a \circ b$, mit $a,b,c \in G$	G abgeschlossen bzgl. $\circ$
(G2)	$\forall\, a,b,c \in G : (a \circ b) \circ c = a \circ (b \circ c)$	Assoziativgesetz
(G3)	$\exists\, e \in G: \forall\, a \in G: a \circ e = e \circ a = a$	Neutrales Element
(G4)	$\forall\, a \in G: \exists\, a^{-1} \in G: a \circ a^{-1} = e$, mit $a^{-1}, e \in G$	Inverses Element
(G5)	$\forall\, a,b \in G : (a \circ b) = (b \circ a)$	Kommutativgesetz

Die Anzahl der Elemente einer Gruppe G nennt man die *Ordnung* von G, abgekürzt durch $\#G$.

Definition: Ein kommutativer *Körper* $(K, +, \cdot)$ ist eine Menge mit mindestens zwei Elementen und zwei Verknüpfungen $+$ und $\cdot$ mit folgenden Eigenschaften:

(K1) $(G, +)$ ist eine abelsche Gruppe mit neutralem Element 0
(K2) $(G \setminus \{0\}, \cdot)$ ist eine abelsche Gruppe mit neutralem Element 1
(K3) Es gelte das Distributivgesetz: $\forall\, a,b,c \in K : (a + b) \cdot c = a \cdot c + b \cdot c$

Definition: die *Charakteristik* $\chi(K)$ eines Körpers K nennt man das kleinste positive Vielfache l, für das folgendes gilt:

$$\chi: K \mapsto \mathbb{N}$$

$$\chi(K) = \begin{cases} l & \text{falls } l \text{ das kleinste positive Vielfache ist, mit } l \circ a = 0 \;\; \forall a \in K \\ 0 & \text{falls kein solches Vielfaches existiert} \end{cases}$$

Definition [Gor00]**:** Eine Teilmenge U eines Körpers K mit $U \subseteq K$ nennt man einen *Unterkörper*, wenn U eine Teilmenge von K ist und bezüglich der Operationen auch ein Körper ist. Dementsprechend wird K auch Erweiterungskörper von U genannt.
Hat ein Körper außer sich selbst keine weiteren Unterkörper, so nennt man ihn *Primkörper*.

3.2 Endliche Körper

Man unterscheidet endliche und unendliche Körper. Endliche Körper notiert man oftmals mit F_{p^q}, F_p, $\mathcal{GF}(p^q)$ oder $\mathcal{GF}(p)$, wobei $\mathcal{GF}$ für Galois-Feld steht. Unter p versteht man die die prime Charakteristik des Körpers und q bezeichnet den Erweiterungsgrad. Die Bezeichnung Galois-Feld ($\mathcal{GF}$) ist nach dem Mathematiker Evariste Galois benannt [Gor00]. Er war einer der ersten Mathematiker, der sich intensiv mit endlichen Körpern beschäftigte und eine zugehörige Theorie zur Verfügung stellte. Field ist die anglistische Bezeichnung für „mathematische Körper".

Rechnet man in einem endlichen Körper, so ist das Ergebnis der Berechnung wieder ein Element des Körpers. Betrachtet man beispielsweise die Elemente $a = 5$ und $b = 9$ im $\mathcal{GF}(13)$, so ergeben sich für nachstehend angeführte Verknüpfungen folgende Beispielrechnungen:

Addition	$5 + 9 = 14$, $14 \bmod 13 = 1$
Subtraktion	$5 - 9 = -4$, $-4 \bmod 13 = 9$
Multiplikation	$5 \cdot 9 = 45$, $45 \bmod 13 = 6$
Division	$\frac{5}{9} = 5 \cdot 9^{-1} = 5 \cdot 3 = 15$, $15 \bmod 13 = 2$

Zur Durchführung der Division $\frac{x}{y}$ muß man das Inverse y^{-1} des Nenners folgendermaßen ermitteln: finde ein y^{-1}, das folgende Gleichung erfüllt:

$$y \cdot y^{-1} \equiv 1 \bmod p\ .$$

Das bedeutet für o.g. Beispiel: $9 \cdot 3 \equiv 1 \bmod 13 \Rightarrow 9^{-1} = 3$. Das Finden eines Inversen ist außerordentlich zeitaufwendig und sollte deswegen vermieden werden, wie später gezeigt werden wird.

Das o.a. Beispiel bezieht sich auf den Körper $\mathcal{GF}(13)$, also den Fall des sog. Primfeldes. Die andere Möglichkeit der Betrachtung stellt der „binäre Fall" dar. Dabei bezieht man sich auf den Körper $\mathcal{GF}(2^m)$. Anstelle von Zahlenwerten als Elemente des Feldes werden hier Polynome verwendet. Die nach jeder Verknüpfung obligatorische Modulo-Reduktion wird hier ebenfalls mit einem Polynom durchgeführt, welches als Primpolynom bezeichnet wird. Da die Koeffizienten im binären Körper ausschließlich die Werte Null und Eins annehmen können, führt die Addition von (x^n+x^n) nicht zu $2x^n$, sondern zu Null. Das bedeutet, daß man die Koeffizienten modulo 2 nimmt. Nachfolgende Beispielrechnungen beziehen sich auf den Körper $\mathcal{GF}(2^3)$ mit $(x^3 + x + 1)$ als Primpolynom. Dieser Körper enthält folgende Elemente: $\{0, 1, x, x+1, x^2, x^2+1, x^2+x, x^2+x+1)$. Nachstehend sind wieder Beispielverknüpfungen angegeben:

Addition	$(x^3+x^2) + (x^2+1) = (x^3+1)$,
	$(x^3+1) \bmod (x^3+x+1) = x$
Subtraktion	identisch zur Addition
Multiplikation	$(x^3+x^2) \cdot (x^2+1) = (x^5+x^4+x^3+x^2) \bmod (x^3+x+1) = (x^2+x)$
Division	$\left(\frac{x^3+x^2}{x^2+1}\right) = (x^3+x^2) \cdot (x^2+1)^{-1} = (x^3+x^2) \cdot x =$
	$(x^4+x^3) \bmod (x^3+x+1) = (x^2+1)$

Bei der obigen Division ist wieder die Berechnung des Inversen notwendig: es ist das Inverse zu (x^2+1) gesucht, so daß folgende Gleichung erfüllt wird:

$$(x^2+1) \cdot (x^2+1)^{-1} \equiv 1 \bmod (x^3+x+1)$$

Das gesuchte Inverse ist hier x.

Es wird deutlich, daß das Operieren mit Polynomen umständlich ist, insbesondere die immer wieder durchzuführende modulo-Reduktion. Jedes Polynom vom Grad n hat bekanntlich folgende Form:

$$a_n x^n + a_{n-1}x^{n-1} + \ldots a_2x^2 + a_1x + a_0 = \sum_{i=0}^{n} a_i x^i \text{ , mit } a_i \in \mathcal{GF}(2).$$

Somit kann ein Polynom durch Angabe seiner Koeffizienten a_i dargestellt werden. Da momentan das Feld $\mathcal{GF}(2^m)$ betrachtet wird, können die Koeffizienten, wie bereits erwähnt, nur die Werte Null oder Eins annehmen. Die Addition im obigen Beispiel stellt sich dann folgendermaßen dar:

$x^3 \downarrow$	$x^2 \downarrow$	$x^1 \downarrow$	$x^0 \downarrow$		
1	1	0	0	$\leftrightarrow$	(x^3+x^2)
0	1	0	1	$\leftrightarrow$	(x^2+1)
1	0	0	1	$\leftrightarrow$	(x^3+1)

Der Vorteil der Rechnung in dieser binären Darstellungsform liegt darin, daß man zur Durchführung der Addition die einzelnen Koeffizienten nur mittels eines logischen XOR verknüpfen muß. Die Modulo-Reduktion ergibt sich dadurch automatisch und erfordert keinen zusätzlichen Berechnungsschritt.

x	y	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Konvention: in dieser Arbeit steht das MSB (Most-Significant-Bit) immer ganz links.

Die binäre Darstellung der Polynome, sowie die nötige XOR-Verknüpfung sind in Hardware leicht zu realisieren. Das obige Ergebnis (1 0 0 1) hat den Grad des Primpolynoms

(1 0 1 1), deshalb muß noch modulo gerechnet werden. Dies entspricht wieder einer XOR-Verknüpfung:

1	0	0	1	$\leftrightarrow$	(x^3+1)
1	0	1	1	$\leftrightarrow$	(x^3+x+1)
0	0	1	0	$\leftrightarrow$	x

Bei der Multiplikation profitiert man ebenfalls von dieser Methode. Analog zum obigen Multiplikationsbeispiel soll jetzt (1 1 0 0) · (0 1 0 1) berechnet werden:

1	1	0	0			·	0	1	0	1
0	0	0	0							
	1	1	0	0						
		0	0	0	0					
			1	1	0	0				
	1	1	1	1	0	0				

Man multipliziert jede Stelle der rechten Zahl mit der linken und schreibt das jeweilige Ergebnis versetzt um eine Stelle untereinander. Am Ende werden die Einzelprodukte miteinander XOR-Verknüpft und es ergibt sich (1 1 1 1 0 0), was $(x^5+x^4+x^3+x^2)$ entspricht. Jetzt muß noch die obligatorische Reduktion durch das Primpolynom erfolgen. Dazu reduziert man sukzessive vom MSB ausgehend, bis der Grad des Ergebnispolynoms kleiner ist als der des Primpolynoms:

1	1	1	1	0	0		
1	0	1	1			$\leftrightarrow$	Primpolynom
0	1	0	0	0	0		
	1	0	1	1		$\leftrightarrow$	Primpolynom
	0	0	1	1	0	$\leftrightarrow$	(x^2+x)

Das Ergebnis ist somit (x^2+x). Bei dieser Methode besteht der Nachteil, daß zur Multiplikation zweier Binärzahlen der Bitbreite n ein Zwischenergebnisregister der Breite $2n$ vorgehalten werden muß, bis dieses Zwischenergebnis mittels Modulo-Rechnungen reduziert wird. Dies kann man umgehen, indem man nach jeder Teiladdition eine Reduktion vornimmt, wenn der Grad des Zwischenergebnisses größer oder gleich dem Grad des Primpolynoms ist:

		1	1	0	0				· 0 1 0 1
1. Teilmultiplikation		0	0	0	0	*0*	*0*	*0*	
2. Teilmultiplikation	+		1	1	0	0	*0*	*0*	
			1	1	0	0	*0*	*0*	
Primpolynom	+		1	0	1	1	*0*	*0*	
Ergebnis d. 1. Schritts				1	1	1	*0*	*0*	
3. Teilmultiplikation	+			0	0	0	0	*0*	
				1	1	1	0	*0*	
Primpolynom	+			1	0	1	1	*0*	
Ergebnis d. 2. Schritts					1	0	1	*0*	
4. Teilmultiplikation	+				1	1	0	0	
Endergebnis						1	1	0	$\leftrightarrow$ (x^2+x)

Bei diesem Verfahren kann garantiert werden, daß die Bitbreite der Partialresultate nicht größer als der Grad des Primpolynoms wird. Eine zeit- und platzoptimalere Variante eines Modulo-Multiplizierers wird später in den verschiedenen Implementierungen vorgestellt werden.
Es wurden die verschiedenen Rechenverfahren in den jeweiligen endlichen Feldern $\mathcal{GF}(p)$ und $\mathcal{GF}(2^m)$ vorgestellt.
Das Rechnen in binären Feldern hat hier einen deutlichen Vorteil gegenüber Berechnungen in Primfeldern, da sich die Operationen auf XOR-Verknüpfungen und das Schieben von Bitsequenzen reduzieren läßt. Da in dieser Arbeit eine Hardwarestruktur zur Berechnung von elliptischen Kurvenpunkten entwickelt werden soll, werden im weiteren ausschließlich Berechnungen in binären Feldern betrachtet.

3.3 Elliptische Kurven

Eine elliptische Kurve wird allgemein durch die sog. Weierstrass-Gleichung dargestellt

$$y^2 + a_1xy + a_2y = x^3 + a_3x^2 + a_4x + a_5, \text{ mit } x,\ y,\ a_i\ \in\ K\ , \tag{1}$$

in der die Variablen x und y aus einem beliebigen algebraischen Körper K der komplexen, reellen, rationalen oder natürlichen Zahlen sein können. Insbesondere können sie auch Elemente eines wie vorhergehend beschriebenen endlichen Körpers sein.

3.3.1 Arithmetik mit elliptischen Kurven

Die Funktionsweise der Arithmetik mit elliptischen Kurven läßt sich am besten über dem Körper der reellen Zahlen zeigen, da sich hier die Verknüpfungen geometrisch darstellen lassen.

Man verwendet hier folgende Vereinfachung der Weierstrass-Gleichung für elliptische Kurven über $\mathbb{R}$[1]:

$$y^2 = x^3 + a_4x + a_5 \tag{2}$$

Als Beispiel wähle man $a_4 = -5$ und $a_5 = 2$. Der Graph der elliptischen Kurve ergibt sich aus Darstellung der positiven und negativen y-Werte der Funktion $y = \sqrt{x^3 - 5x + 2}$ (s. Abb. 2).

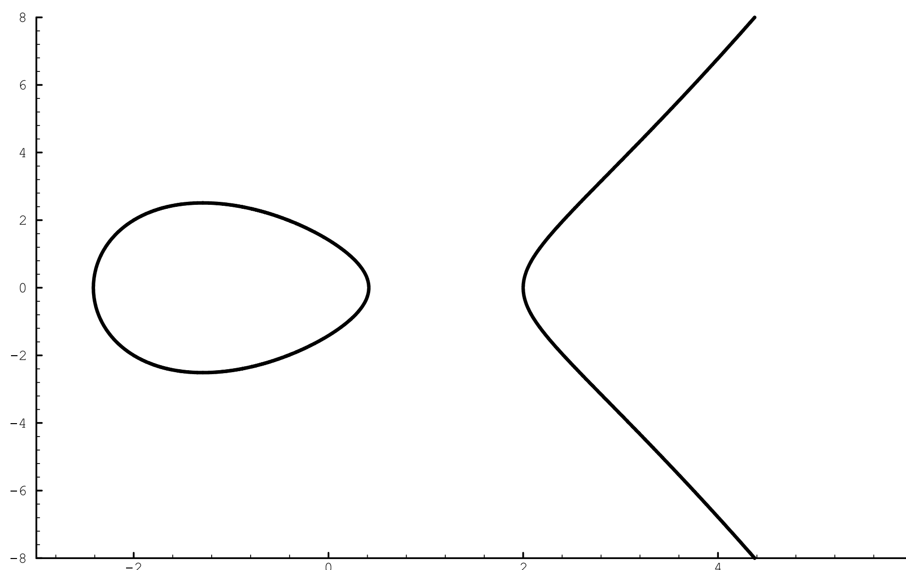

Abbildung 2: Elliptische Kurve $y^2 = x^3 - 5x + 2$

Die Koeffizienten a_4 und a_5 bestimmen die Form der jeweiligen elliptischen Kurve. [Boh97] zeigt, daß man sechs Fallunterscheidungen bzgl. der Kurvenform machen kann. Allen Kurven gemein ist, daß sie symmetrisch zur Ordinate sind.
Wie findet nun das Rechnen mit elliptischen Kurven über endlichen Feldern statt? Beginnt man mit der Addition, so betrachte man zwei Punkte P und Q auf der elliptischen Kurve. Addiert man diese Punkte, so ist der resultierende Punkt $R = P + Q$ ein anderer auf dieser Kurve. Zusätzlich benötigt man noch ein neutrales Element $\mathcal{O}$. Man kann jeden Punkt zu dem neutralen Punkt addieren und erhält immer den Ausgangspunkt zurück: $P + \mathcal{O} = P$, $-P + \mathcal{O} = -P$. Der neutrale Punkt $\mathcal{O}$ wird auch als Unendlichkeitspunkt bezeichnet. Graphisch stellt sich die Addition zweier Punkte einer elliptischen Kurve wie in Abbildung 3 dar.
Man legt eine Gerade durch die beiden zu addierenden Punkte P und Q. Die Gerade schneidet dann die elliptische Kurve im Punkt $-R = -(P + Q)$. Die Vorzeichenumkehr von Punkten ist als Spiegelung an der Ordinate definiert: $P = (x, y) \Rightarrow -P = (x, -y)$. Somit ergibt sich in Abb. 3 der Ergebnispunkt $R = P + Q$. Nach diesem Schema läßt sich auch die Existenz des Unendlichkeitspunktes $P + (-P) = \mathcal{O}$ zeigen (s. Abb. 4). Verläuft die Gerade durch die zu verknüpfenden Punkte senkrecht, so schneidet sie die Kurve kein

[1] Für den Körper $\mathcal{GF}(2^m)$ existieren ähnliche Vereinfachungen, wie später gezeigt werden wird

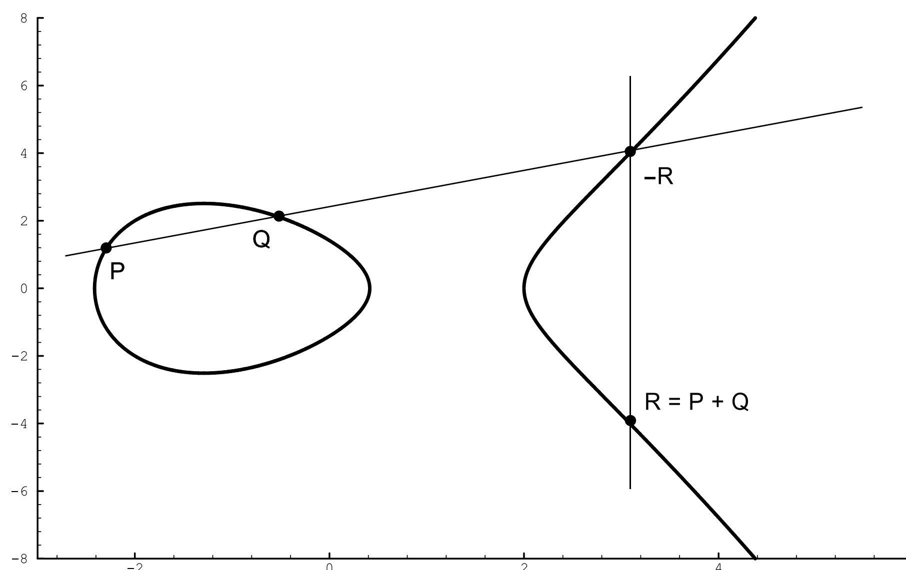

Abbildung 3: Addition zweier Punkte auf der Elliptischen Kurve

drittes Mal mehr. Dann ist die Lösung der Unendlichkeitspunkt.

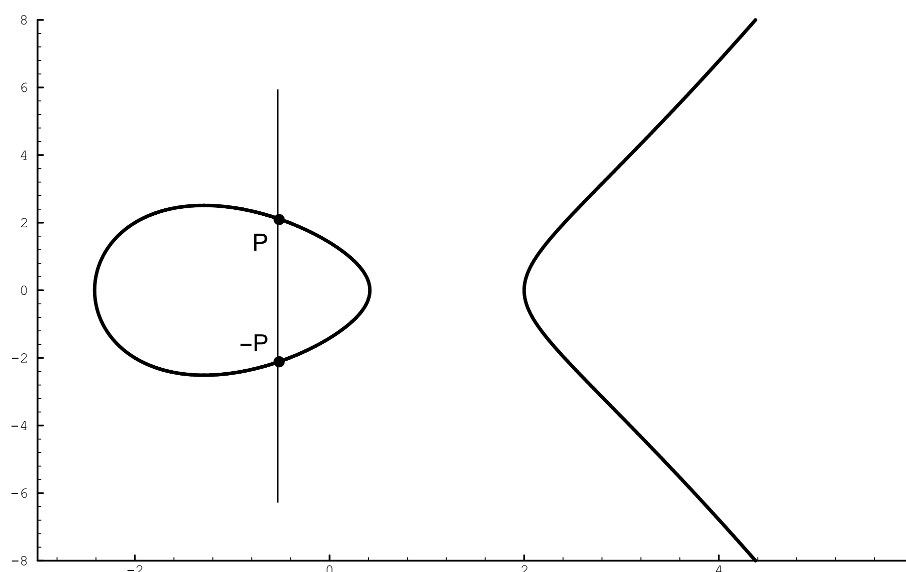

Abbildung 4: $P + (-P) = \mathcal{O}$

Sind bei der Addition die zu addierenden Punkte P und Q identisch, also $P = Q$, so ergibt sich der resultierende Punkt $R = P + P$ wie in Abbildung 5 gezeigt. Man legt eine Tangente an die elliptische Kurve im Punkt P. Wo diese Tangente die elliptische Kurve schneidet, liegt der Punkt $-R$. Die Spiegelung an der x-Achse ergibt den Lösungspunkt R.

3.3.2 Elliptische Kurven über $\mathcal{GF}(2^m)$

Ähnlich der Betrachtung über dem Körper der reellen Zahlen läßt sich die Weierstrass-Gleichung auch bei Beschränkung auf den Körper $\mathcal{GF}(2^m)$ vereinfachen. Da es hier eine

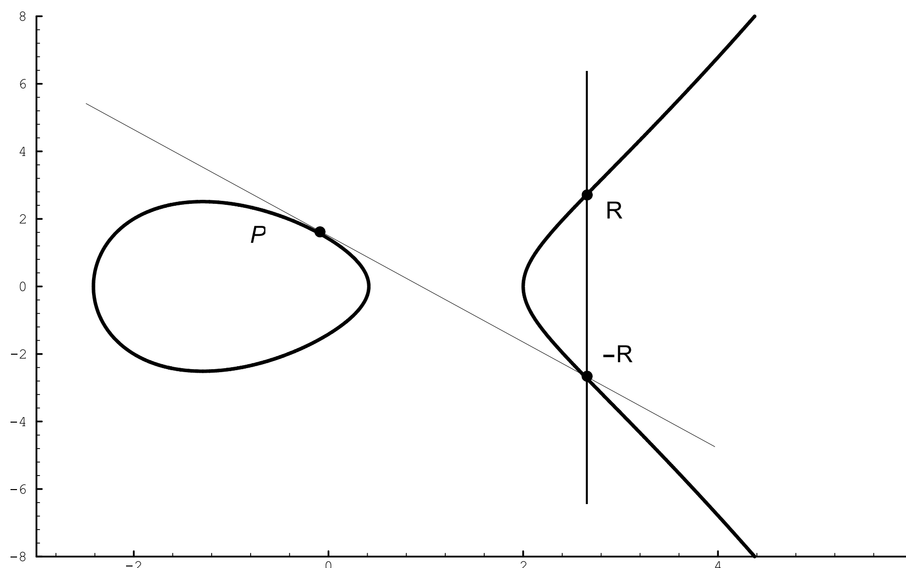

Abbildung 5: Verdoppelung eines Punktes auf einer Elliptischen Kurve

Vereinfachung mit besonderen Eigenschaften gibt, seien zwei Formen angegeben:

$$y^2 + xy = x^3 + ax^2 + b\,,\ \text{mit}\, b\, \neq\, 0 \tag{3}$$

sowie

$$y^2 + y = x^3 + ax + b\,,\ \text{mit}\, b\, \neq\, 0 \tag{4}$$

Die Kurven der letzteren Form (4) nennt man „supersinguläre elliptische Kurven“. Eine wichtige Eigenschaft dieser supersingulären Kurven ist, daß ihre Punkte sehr schnell ermittelt werden können [Sil94]. Das macht diese Familie von elliptischen Kurven für Kryptographieanwendungen unbrauchbar. Die Kurven der Form (3) nennt man „nicht-supersinguläre Kurven“. Diese Kurven sind für Verschlüsselungsanwendungen gut geeignet.
Die richtige Wahl der Koeffizienten ist sehr wichtig, da diese indirekt die Kurvenordnung $\#E$ festgelegen. Wie später ausführlich dargelegt werden wird, sollte die Ordnung einer elliptischen Kurve über möglichst wenige Primteiler verfügen.

Definition [IEEE99, S. 119]**:** Die Anzahl der Punkte einer elliptischen Kurve E inklusive des Unendlichkeitspunktes $\mathcal{O}$ wird die Ordnung der Kurve genannt und mit $\#E(\mathcal{GF}(2^m))$ bezeichnet.

Werden elliptische Kurven über $\mathcal{GF}(2^m)$ betrachtet, so kann man arithmetische Operationen mit Punkten nicht graphisch darstellen. Daher wurde diese Anschauung im Körper über den reellen Zahlen durchgeführt.
Die Addition zweier (affiner)[2] Punkte P und Q ist über $\mathcal{GF}(2^m)$ folgendermaßen definiert

[2]Der Begriff wird in Kapitel 3.3.3 definiert werden

[IEEE99, S. 125]:

Seien $P = (x_1, y_1)$, $Q = (x_2, y_2)$, $R = P + Q = (x_3, y_3)$

Wenn $P \neq Q$ ist, dann gilt:

$$\theta = \frac{y_2 - y_1}{x_2 - x_1}\,, \tag{5}$$

$$x_3 = \theta^2 + \theta + x_1 + x_2 + a\,, \tag{6}$$

$$y_3 = \theta\,(x_1 + x_3) - y_1\,. \tag{7}$$

Wenn $P = Q$ ist, dann sei $P = (x_1, y_1)$ und es gilt:

$$\theta = x_1 + \frac{y_1}{x_1}\,, \tag{8}$$

$$x_3 = \theta^2 + \theta + a\,, \tag{9}$$

$$y_3 = x_1^2 + (\theta + 1)\,x_3\,. \tag{10}$$

Die Vorzeichenumkehr des Punktes P hat die folgende Regel:

$P = (x, y)$, $-P = (x, y + x)$.

Beispiel:

Man betrachte die elliptische Kurve $E : y^2 + xy = x^3 + ax^2 + b$ mit $a = (t + 1)$ und $b = 1$ über dem Feld $\mathcal{GF}(2^3)$ mit Primpolynom $t^3 + t + 1 = 0$.
Man suche zuerst alle Punkte der Kurve $E : y^2 + xy = x^3 + (t + 1)x^2 + 1$:

- Man substituiere t durch x:
  $y^2 + xy = x^3 + (x + 1)x^2 + 1 \Leftrightarrow$
  $y^2 + xy = x^3 + x^3 + x^2 + 1 \Leftrightarrow$
  $y^2 + xy = x^2 + 1$
  Für $x = (000)_2$ ($\leftrightarrow$ polynomiell: 0) :
  $y^2 = 1$.
  Diese Gleichung hat eine Lösung für $y = 1$
  $\Rightarrow$ Es wurde ein erster Punkt mit den Koordinaten (000, 001) gefunden.

- Für $x = (001)_2$ ($\leftrightarrow$ polynomiell: 1) :
  $y^2 + y = 1 + (t + 1) + 1 \Leftrightarrow y^2 + y = t + 3$.
  Diese Gleichung hat keine Lösung.

- Für $x = (010)_2$ ($\leftrightarrow$ polynomiell: x) :
  $y^2 + xy = x^3 + (x+1)x^2 + 1 \Leftrightarrow$
  $y^2 + xy = x^3 + x^3 + x^2 + 1 \Leftrightarrow$
  $y^2 + xy = x^2 + 1$.

  1. Diese Gleichung hat eine Lösung für $y = x^2$,
     denn $(x^2)^2 + x \cdot x^2 = x^4 + x^3$.
     Dies muß noch durch das Primpolynom reduziert werden:

1	1	0	0	0	$\leftrightarrow$	$(x^4 + x^3)$
1	0	1	1		$\leftrightarrow$	Primpolynom
	1	1	1	0	$\leftrightarrow$	$(x^3 + x^2 + x)$
	1	0	1	1	$\leftrightarrow$	Primpolynom
		1	0	1	$\leftrightarrow$	$(x^2 + 1)$

     Wie bereits im Kapitel 3.2 erwähnt, kann man die Reduktion auch in die Multiplikation integrieren, um die Breite eines Partialergebnisregisters zu verringern.

     $\Rightarrow$ Es wurde ein zweiter Punkt mit den Koordinaten (010, 100) gefunden.

  2. Diese Gleichung hat eine Lösung für $y = x^2 + x$, denn
     $((x^2 + x)\,(x^2 + x)) + x\,(x^2 + x) =$
     $(x^4 + x^2) + (x^3 + x^2) =$
     $x^4 + x^3$.
     Analog zu obiger Reduktion gilt:
     $(x^4 + x^3) \bmod (x^3 + x + 1) = (x^2 + 1)$.

     $\Rightarrow$ Es wurde ein dritter Punkt mit den Koordinaten (010, 110) gefunden.

Auf diese Weise ermittelt man für alle $2^3 = 8$ möglichen x-Werte des $\mathcal{GF}(2^3)$ die Werte der Tabelle 1. Insgesamt liegen auf der Beispielkurve 13 Punkte.

binär			dezimal		
x	y_1	y_2	x	y_1	y_2
000	001	—	0	1	—
001	—	—	1	—	—
010	100	110	2	4	6
011	100	111	3	4	7
100	001	101	4	1	5
101	010	111	5	2	7
110	000	110	6	0	6
111	001	110	7	1	6

Tabelle 1: Wertetabelle der elliptischen Beispielkurve

Die dezimalen Werte sind hier nur zur Verdeutlichung der Motivation des Kapitels 3.3.1 angegeben. Wie schon in [Gor00, S. 34] gezeigt, kann man die Arithmetik elliptischer

Kurven über $\mathcal{GF}(2^m)$ nicht mehr graphisch veranschaulichen. Die graphische Darstellung der Tabelle 1 zeigt Abbildung 6. Man sieht deutlich, daß die Form einer elliptischen Kurve

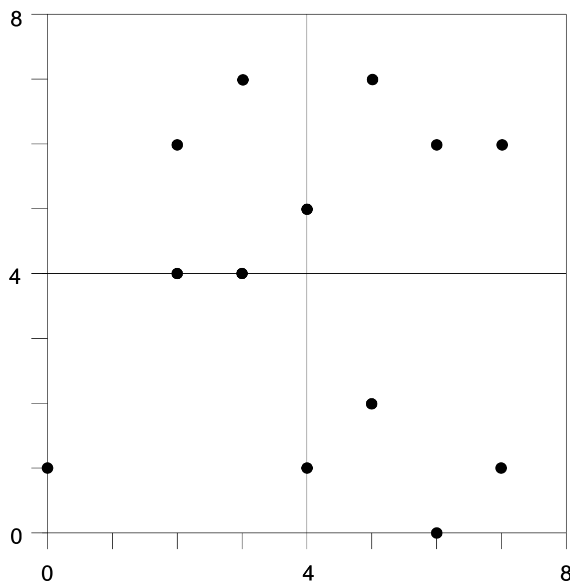

Abbildung 6: Graph einer elliptischen Kurve über $\mathcal{GF}(2^m)$

nicht erkennbar ist, auch die im Kapitel 3.3.1 vorgeführten arithmetischen Operationen sind im $\mathcal{GF}(2^m)$ nicht darstellbar. Erkennbar sind bis auf Punkt (0, 1) die inversen Punkte, die auf jeweils demselben x-Wert liegen.

Da nun alle Punkte der Beispielkurve E bekannt sind, läßt sich auch die Ordnung $\#E$ bestimmen. Wie schon definiert, ist die Ordnung einer Kurve die Anzahl derer Punkte zuzüglich dem Unendlichkeitspunkt $\mathcal{O}$. Hier gilt also $\#E\ (\mathcal{GF}(2^3)) = 14$.

Als nächster Schritt soll hier mit Hilfe der Formeln (8) - (10) ein Punkt der Beispielkurve verdoppelt werden. Wählt man z.B. den Punkt $P = (3, 7) = (011, 111)$ zur Verdoppelung aus, so ergibt sich:

$\theta = x + \frac{y}{x} \Rightarrow (011) + \frac{(111)}{(011)}$.

Hier muß das Inverse zu (011) bestimmt werden, so daß gilt:

$(011) \cdot (011)^{-1} \equiv (001) \bmod (1011)$.

Nach [IEEE99, S. 97 ff.] läßt sich zu einem Element β im Körper $\mathcal{GF}(2^m)$ das inverse Element β^{-1} berechnen durch: $\beta^{-1} = \beta^{2^m-2}$.

Im Falle dieses Beispiels ist $\beta = (011) \Rightarrow \beta^{-1} = (011)^{2^3-2} = (011)^6 = (110)$.

Damit folgt:

$\theta = (011) + \frac{(111)}{(011)} = (011) + (111) \cdot (110) = (10001)$. $(10001) \bmod (1011) = (111)$,

$x_3 = \theta^2 + \theta + a \Rightarrow (111)^2 + (111) + (011) = (10001)$. $(10001) \bmod (1011) = (111)$,

$y_3 = x^2 + (\theta + 1) \cdot x_3 \Rightarrow (011)^2 + (110) \cdot (111) = (10111)$.

$(10111) \bmod (1011) = (001)$.

Das Ergebnis der Verdoppelung des Punktes P = (011, 111) ist R = (111, 001). Eine notwendige Bedingung für die Korrektheit der Rechnung ist, daß der Ergebnispunkt R wieder auf der elliptischen Kurve liegt. Das kann in diesem Fall aus Abbildung 6 abgelesen werden.

3.3.3 Affine und Projektive Koordinaten

Bisher wurden die Punkte einer elliptischen Kurve über einem endlichen Körper K als Tupel (x, y) in der sog. *affinen* Form angegeben. Diese Form beschreibt einen Punkt eindeutig, und die Formeln zur Addition eines solchen Punktes sind schon dargestellt worden. Für die Laufzeit eines Algorithmus zur Addition zweier Punkte ist die in den Formeln vorkommende Berechnung des multiplikativen Inversen ein inakzeptabler Faktor. Mit einer anderen Darstellungsweise der Punkte läßt sich dieser Umstand umgehen.
Dabei vergleiche man die Form des affinen Punktes mit einer ganzen Zahl. Die Addition zweier Punkte ist so sehr einfach. Eine ganze Zahl läßt sich aber auch als Bruch darstellen. Wichtig ist hier, daß eine ganze Zahl durch unendlich viele Brüche dargestellt werden kann, z.B. $3 = \frac{3}{1} = \frac{6}{2} = \frac{9}{3} \dots$. Die Addition zweier Brüche ist nicht unbedingt einfach, denn man muß u.U. erst für Nennergleichheit sorgen. Wenn in einer beliebigen Rechnung in der Menge $\mathbb{Q}$ einmal ein Bruch auftritt, den man zu einer ganzen Zahl kürzen könnte, ist es dann sinnvoll, ihn zu kürzen, oder rechnet man nicht besser gleich mit den zwei Werten Zähler und Nenner bis zum Schluß weiter, und kürzt dann erst bei Bedarf? Man muß den Aufwand eines zweiten Registers mit einem zusätzlichen Wert mit dem Aufwand einer Division vergleichen. Je nach Ablauf der weiteren Berechnungen kann das Ergebnis dieses Vergleiches unterschiedlich ausfallen.
In der Darstellungsweise der *projektiven* Form werden die Punkte einer elliptischen Kurve durch Tripel (X, Y, Z) beschrieben. Ein projektives Tripel ist, ähnlich zu den Brüchen des o.g. Vergleichs, für einen affinen Punkt nicht eindeutig, sondern es existieren Äquivalenzklassen, in denen die Tripel jeweils skalare Vielfache voneinander sind:

$$(X,\ Y,\ Z)\ =\ (\lambda^2 X,\ \lambda^3 Y,\ \lambda Z),\ \forall\, \lambda \neq 0,\ \lambda \in K \tag{11}$$

Eine solche Äquivalenzklasse nennt man einen *projektiven Punkt* [Kob98]. Hat ein solcher projektiver Punkt ein Tripel mit $Z \neq 0$, so existiert in der gesamten Äquivalenzklasse nur ein Element mit $Z = 1$ und aus diesem kann man den affinen Wert folgendermaßen ableiten:

$$(x,\ y,\ 1),\ \text{mit}\ x = \frac{X}{Z^2},\ y = \frac{Y}{Z^3} \tag{12}$$

Während also die Umrechnung von projektiven zu affinen Koordinaten einen größeren Rechenaufwand bedeutet, ist der umgekehrte Weg deutlich einfacher:

$$X \leftarrow x,\ Y \leftarrow y,\ Z \leftarrow 1 \tag{13}$$

Die projektiven Koordinaten des Unendlichkeitspunktes $\mathcal{O}$ haben die Form $(\lambda^2, \lambda^3, 0)$, mit $\lambda \neq 0, \lambda \in K$.
Da dieses Prinzip einen entscheidenden Vorteil hat, wird jetzt die Addition zweier Punkte einer elliptischen Kurve über $\mathcal{GF}(2^m)$ in *projektiver* Form dargestellt:
Seien $P = (X_0, Y_0, Z_0)$, $Q = (X_1, Y_1, Z_1)$ Punkte einer elliptischen Kurve der Form (3) über $\mathcal{GF}(2^m)$. Dann gilt nach [IEEE99, S. 130]:
$R = P + Q = (X_2, Y_2, Z_2)$, mit

$$U_0 = X_0 Z_1^2 \tag{14}$$

$$S_0 = Y_0 Z_1^3 \tag{15}$$

$$U_1 = X_1 Z_0^2 \tag{16}$$

$$W = U_0 + U_1 \tag{17}$$

$$S_1 = Y_1 Z_0^3 \tag{18}$$

$$R = S_0 + S_1 \tag{19}$$

$$L = Z_0 W \tag{20}$$

$$V = R X_1 + L Y_1 \tag{21}$$

$$Z_2 = L Z_1 \tag{22}$$

$$T = R + Z_2 \tag{23}$$

$$X_2 = a Z_2^2 + T R + W^3 \tag{24}$$

$$Y_2 = T X_2 + V L^2 \tag{25}$$

In [IEEE99, S. 130] findet man einen Algorithmus in Pseudocode für die Addition von Punkten in projektiver Darstellung. In [Gor00, S. 54] wird zusätzlich noch ein Datenfluß-graph zu diesem Algorithmus vorgestellt.

Der Vorteil dieser Form der Addition liegt darin, daß keine Inversion mehr nötig ist. In Abhängigkeit von weiteren Berechnungsschritten ist es vielleicht nicht sinnvoll, das „projektive Ergebnis“ in die affine Form umzurechnen. In der projektiven Darstellung lassen sich auch weitere Berechnungen durchführen, wie später gezeigt werden wird.

4 Kryptosysteme mittels elliptischer Kurven

4.1 Falltürfunktionen

Für ein Public-Key Kryptosystem benötigt man öffentliche und geheime Schlüssel zur Ver- und Entschlüsselung von Nachrichten. Jeder Empfänger besitzt einen individuellen öffentlichen Schlüssel, der dem Absender bekannt gegeben werden muß. Der Absender verschlüsselt seine Nachricht mit Hilfe des öffentlichen Schlüssels des Empfängers und verschickt dann die Nachricht. Der Empfänger besitzt zu seinem öffentlichen Schlüssel einen geheimen Schlüssel, mit dessen Hilfe er ohne erheblichen Aufwand die Nachricht entschlüsseln kann. Sollte sich ein Unbefugter der verschlüsselten Nachricht habhaft gemacht haben, so kann dieser die Nachricht nur unter sehr großem Zeit- und Ressourcenaufwand entschlüsseln.

Mathematische Funktionen, die einfach auszuführen aber schwer zu invertieren sind, heißen Falltürfunktionen (Trapdoor-Functions). Ein alltägliches Beispiel für eine Falltürfunktion ist das Telefonbuch; die auszuführende Funktion ist die, einem Namen die entsprechende Telefonnummer zuzuordnen. Da die Namen alphabetisch geordnet sind, ist diese Zuordnung einfach auszuführen. Aber ihre Invertierung, also die Zuordnung eines Namens zu einer *gegebenen* Nummer, ist offensichtlich viel schwieriger, wenn man nur ein Telefonbuch zur Verfügung hat.

Ein weiteres Beispiel einer für die Kryptographie wichtigen Falltür- oder Einwegfunktion ist die Multiplikation natürlicher Zahlen. Diese Funktion läßt sich einfach ausführen, aber die Umkehrfunktion ist die Faktorisierung, und die ist für große Zahlen praktisch unmöglich durchzuführen. Als einfaches Beispiel diene hier das Quadrieren (x^2), welches einfach zu berechnen ist; die Umkehrfunktion ist die Quadratwurzel ($\sqrt{x}$), welche sehr viel schwieriger zu berechnen ist.

Eine wichtige Klasse von Falltürfunktionen sind die sog. diskreten Exponentialfunktionen:

Sei p eine Primzahl, und g eine natürliche Zahl mit $g \leq p - 1$. Dann ist die diskrete Exponentialfunktion zur Basis g definiert durch

$$k \mapsto g^k \bmod p \ (1 \leq k \leq p-1)\,. \tag{26}$$

Die Umkehrfunktion wird diskrete Logarithmusfunktion dl_g genannt und es gilt:

$$dl_g\,(g^k) = k\,. \tag{27}$$

Unter dem Problem des diskreten Logarithmus (DLP) versteht man das Folgende:

Gegeben p, g und y, bestimme k so, daß $y = g^k \bmod p$ gilt [BeSW95].

4.2 Die Diffie-Hellmann-Schlüsselvereinbarung

Dieses sehr bekannte Protokoll ist schon 1976 in der ersten Arbeit zur Public-Key-Kryptographie von Whit Diffie und Martin Hellmann enthalten [DiHe76]. Die meisten klassischen Verschlüsselungsstrategien sind nur dann durchführbar, wenn die Teilnehmer einen gemeinsamen Schlüssel ausgetauscht haben. Bislang mußte dieser über einen sicheren Kanal übermittelt werden. Mit Hilfe der Diffie-Hellmann-Schlüsselvereinbarung kann man geheime Schlüssel über einen öffentlichen Kanal übertragen.
Das Verfahren funktioniert, wie in Abbildung 7 dargestellt: Sei E eine öffentlich bekannte elliptische Kurve über F_q und B ein zufällig gewählter, öffentlich bekannter Basispunkt auf dieser Kurve. Der Sender sucht sich eine ganzzahlige Zufallszahl k_a, rechnet den Punkt $k_a \cdot B$ aus und übermittelt ihn an den Empfänger. Umgekehrt sucht sich der Empfänger eine ganzzahlige Zufallszahl k_b, rechnet den Punkt $k_b \cdot B$ aus und übermittelt ihn an den Sender. Jetzt sind beide in der Lage, den Punkt $P = k_a \cdot k_b \cdot B$ zu berechnen, der den geheimen Verschlüsselungsschlüssel darstellt und dessen Entstehungsfaktoren nur die beiden Partner kennen. Dabei hat zusätzlich der Sender keine Kenntnis von k_b, der Empfänger keine Kenntnis von k_a.

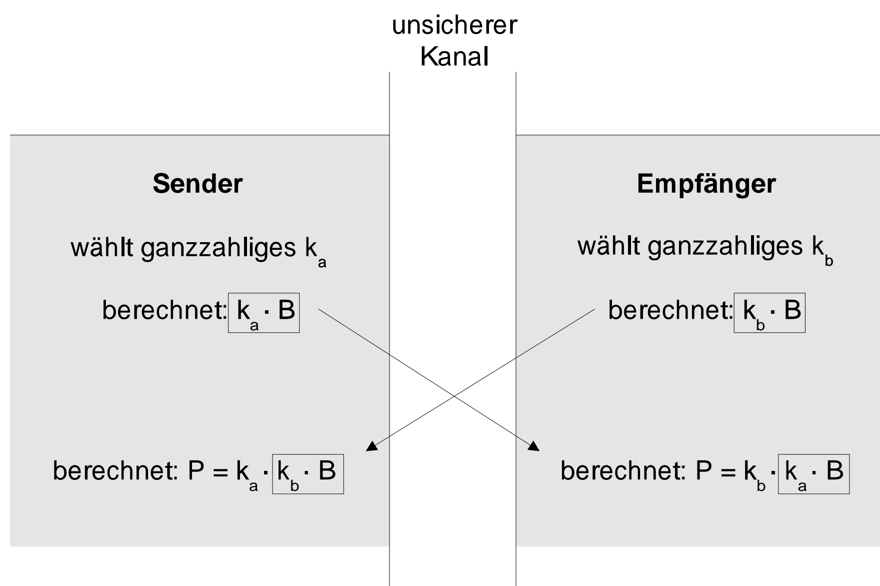

Abbildung 7: Die Diffie-Hellmann-Schlüsselvereinbarung

Die Schlüsselvereinbarung ist gelungen, da der Sender und der Empfänger den gemeinsamen Punkt P erhalten. Der Punkt P ist geheim, da ihn niemand außer dem Sender und Empfänger berechnen kann. Ein potentieller Angreifer, der ja die Kurve E und den Basispunkt B kennt, kann durch Abhören des Übertragungskanals zwar die Punkte $k_a \cdot B$ und $k_b \cdot B$ erlangen, aber nicht die Zahlen k_a oder k_b. Dieses Problem wird das „Diffie-Hellmann-Problem für elliptische Kurven" genannt.
Das o.a. Problem des diskreten Logarithmus läßt sich bzgl. der Diffie-Hellmann Schlüsselvereinbarung folgendermaßen umformulieren: E ist eine elliptische Kurve über F_q und

B ist ein Basispunkt dieser Kurve. Für eine Gruppe G zur Basis $g \in G$ soll bei gegebenem $y \in G$ eine Zahl gefunden werden, so daß $g^x = y$ gilt. Schreibt man die Gruppenoperation in G additiv, erhält man $x \cdot g = y$. Wenn solch ein Lösungselement existiert, muß y in der von g erzeugten Untergruppe vorhanden sein. Für den Fall $G = E$, ist das Problem des diskreten Logarithmus für ein gegebenes $P \in E$ ein x zu finden, für das gilt: $P = x \cdot B$, wenn ein solches existiert.
Nun ist offensichtlich, daß das Diffie-Hellmann-Problem für elliptische Kurven gelöst ist, wenn man das des diskreten Logarithmus lösen kann. Wenn der Angreifer, der durch Abhören des öffentlichen Kanals die Werte $k_a \cdot B$ und B kennen kann, das k_a findet, ist das Verschlüsselungssystem gebrochen.
Der Beweis, daß das Diffie-Hellmann-Problem für elliptische Kurven komplexitätstheoretisch äquivalent zum Problem des diskreten Logarithmus ist, ist noch nicht erfolgreich durchgeführt worden [Kob98].

4.3 Das ElGamal-Verschlüsselungsverfahren

Wenn man das Diffie-Hellmann Schlüsselvereinbarungsverfahren leicht verändert, erhält man einen asymmetrischen Verschüsselungsalgorithmus, der auf ein Konzept von Taher ElGamal [ElG85] zurückgeht:
Sei E wieder eine öffentlich bekannte Kurve über F_q und B ein zufällig gewählter, öffentlich bekannter Basispunkt auf dieser Kurve. Wie in Abbildung 8 dargestellt, sucht sich der Empfänger eine ganzzahlige Zufallszahl d_{s_e} und berechnet damit seinen öffentlichen Schlüssel $d_{p_e} = d_{s_e} \cdot B$. Der Sender sucht sich eine ganzzahlige Zufallszahl k und berechnet mit dieser die zwei Punkte $(x_1, y_1) = k \cdot B$ und $(x_2, y_2) = k \cdot d_{p_e}$. Die zu verschlüsselnde Nachricht M wird in Feldelemente m_1 und m_2 aufgeteilt und in einer vorgeschriebenen, komplexen Weise mit x_2 verknüpft. Daraus ergibt sich dann der Punkt (c_1, c_2).[3] Dann übermittelt der Sender die verschlüsselte Nachricht in Form von (x_1, y_1, c_1, c_2) an den Empfänger. Der Empfänger ist in der Lage, sich den Punkt (x_2, y_2) auszurechnen, da $(x_2, y_2) = d_{s_e} \cdot (x_1, y_1)$. Da der Empfänger somit x_2 kennt, kann er die Nachricht aus (c_1, c_2) ermitteln.

4.4 Praktische Anwendungen

Obwohl die Kryptographie mittels elliptischer Kurven noch eine relativ junge Disziplin ist, haben sich schon eine Reihe von Unternehmen diesem Thema angenommen und kommerzielle Anwendungen entwickelt.
Im September 2000 schlossen sich beispielsweise die Firmen Motorola und Entrust zu einer strategischen Allianz zusammen, um Lösungen für den sicheren, mobilen Handel zu Entwickeln. Dabei steht vor allem die Sicherheit bei drahtloser Kommunikation und

[3] Details zur Einbindung der Nachrichten findet man in Bohnsack, [Boh97, S. 36 ff.]

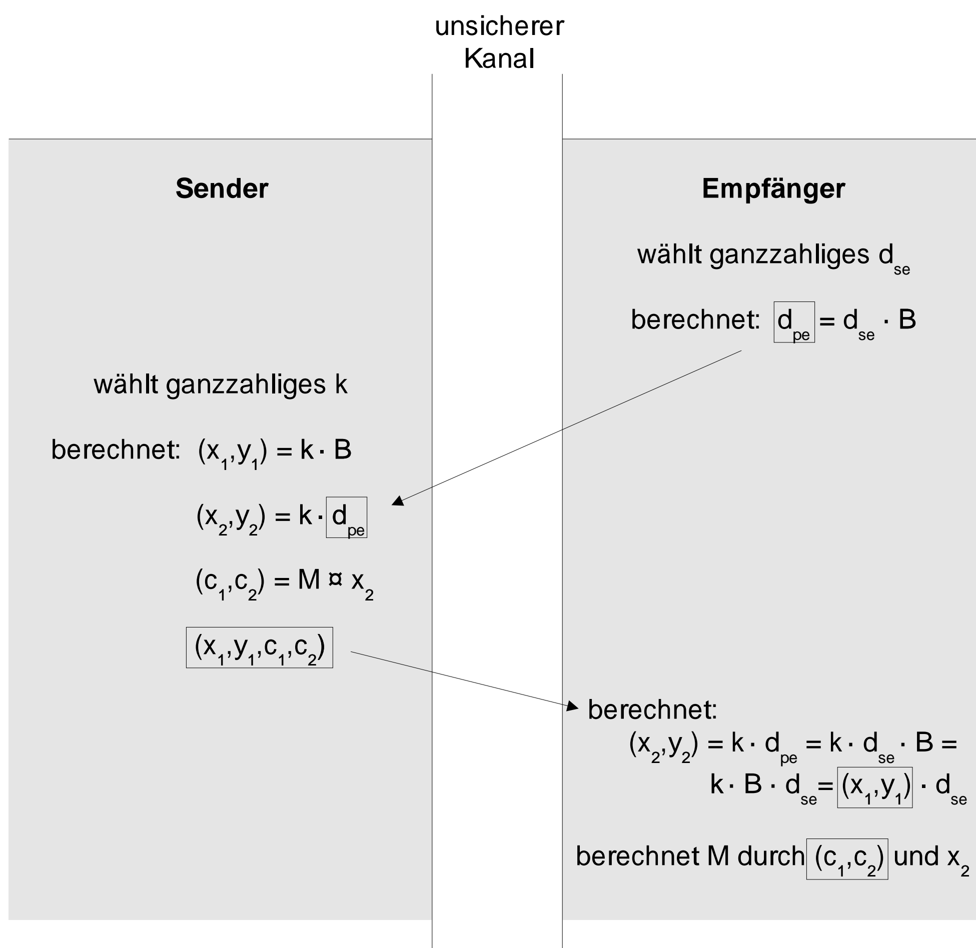

Abbildung 8: Das ElGamal-Verschlüsselungsverfahren

Transaktion im Vordergrund. Entrust entwickelte zu diesem Zweck den Entrust.net WAP (Wireless Application Protocol) Service, mit dessen Hilfe ein PDA (Personal Digital Assistant) mit drahtloser Bluetooth-Verbindung über ein Motorola Mobiltelefon als elektronisches Handelsterminal für mobile Anwender fungiert. Der Entrust.net Service benutzt zur Authentifizierung und Autorisation des PDA Kryptographie mittels elliptischer Kurven. Diese wird zur Datenverschlüsselung verwendet, sowie zur Erzeugung einer digitalen Signatur auf Basis des ECDSA (Elliptic Curve Digital Signature Algorithm) nach dem Standard ANSI 9.62.
In diesem Berech ist die Firma Cryptovision ebenfalls tätig. Sie entwickelt Lösungen zur Verschlüsselung von Informationen, für digitale Signaturen und zur Authentifizierung mittels ECC als Hard- und Softwareimplementation. Vorrangig wird dabei die Implementierung in Smart-Cards und GSM-Karten (Global System for Mobile Communication) behandelt. Referenzkunde der Cryptovision GmbH ist das Bundesamt für Sicherheit in der Informationstechnik (BSI). Ein Softwareaufsatz für die weit verbreiteten E-Mail-Programme Microsoft Outlook, Novell Groupwise, sowie Lotus Notes zur verschlüsselten Übertragung von E-Mails und Anhängen ist seit längerem erhältlich.
Eine große Palette von Anwendungen, die auf ECC basieren, bietet die Firma Certicom. Sie bietet Produkte zur verschlüsselten Kommunikation innerhalb von virtuellen, privaten Netzwerken (VPN), zur sicheren Übertragung von Faxnachrichten, zur Datenverschlüsselung auf PDA´s, etc. Certicom ist im Besitz diverser Patente bzgl. Elliptischer Kurven über $\mathcal{GF}(2^m)$, weshalb viele Unternehmen ihre kommerziellen Anwendungen über dem Körper $\mathcal{GF}(p)$ realisieren, um Lizenzabgaben zu umgehen.

5 Angriff auf ein ECC-basiertes Kryptosystem

5.1 Angriffsmethoden und Schwachstellen

Die Sicherheit eines Public-Key-Kryptoverfahrens liegt darin, daß ein Angreifer Schwierigkeiten hat, dessen Basisproblem oder Falltürfunktion effizient lösen zu können. Daran sind zwei Faktoren maßgeblich beteiligt: erstens das Fehlen eines deterministisch möglichst polynomialzeitberechenbaren Verfahrens zur Lösung des Basisproblems, also eine Aufgabe des Mathematikers. Zweitens, solange Ersteres noch nicht vorliegt, das „Erraten durch hinreichende Versuche" der richtigen Entschlüsselung dadurch unwahrscheinlich zu machen, daß die Anzahl der in Frage kommenden Schlüssel so hoch wie möglich ist. Die Anzahl der in Frage kommenden Schlüssel steigt exponentiell zur Schlüsselbitbreite. Durch welche Art von Angriffsszenario kann ein Angreifer in möglichst kurzer Zeit mit möglichst geringen Kosten und Entwicklungsaufwand einen effektiven und erfolgreichen Angriff auf ein ECC-basiertes Kryptosystem durchführen?

Es sind viele verschiedene Angriffsstrategien auf ein Elliptische Kurven Kryptosystem (ECC) denkbar. Einige davon versuchen, aus Fehlern der Benutzer Vorteile zu ziehen. So läßt sich beispielsweise das beste Kryptosystem brechen, wenn der Empfänger seinen geheimen Schlüssel aus Unachtsamkeit preisgibt. Der Schlüssel könnte bei der Eingabe auf dem Bildschirm ausspioniert werden, oder er wurde auf einer zugehörigen Chipkarte notiert - und diese dann verloren oder gestohlen. Auch sollte der Benutzer auf telefonische Anfrage hin den Schlüssel nicht verraten. Während solch grob fahrlässiges Handeln eines Benutzers kaum unterbunden werden kann, so kann wenigstens von Seiten des Trust-Centers versucht werden, Schwachpunkte von Elliptischen-Kurven-Kryptosystemen zu umgehen. Dazu zählt z.B. daß die Ordnung der dem Kryptosystem zugrundeliegende elliptische Kurve prim sein sollte. Wenn dies nicht der Fall ist und die Ordnung der Kurve einen oder mehrere Primteiler besitzt, so kann man einen ungeeigneten Punkt zum Basispunkt wählen, der eine geringe Punktordnung besitzt. Dadurch wird einem eventuellen Angreifer der mögliche Schlüsselraum unnötig verkleinert. Details zu diesen Umständen werden in den Abschnitten 5.3 und 5.4 ausführlich dargelegt werden.

Ein vermeintlicher Angreifer verfügt, wie auch die legitimen Benutzer des Systems, über alle Informationen, die vom Trust-Center für ein bestimmtes Kryptosystem als öffentliche Parameter herausgegeben werden (s. Abbildung 9). Den öffentlichen Schlüssel des Empfängers kann er u.U. rechtmäßig erhalten, da dieser zumeist auch öffentlich gemacht wird. Wenn nicht, kann er diesen durch Abhören des Kommunikationskanals zwischen dem Sender und dem Empfänger mitschreiben. Die wertvollsten Informationen erhält der Angreifer dann, wenn der Sender seine verschlüsselte Nachricht an den Empfänger übermittelt. Dabei ist das Mitschreiben möglichst vieler Nachrichtenpakete der Form (x_1, y_1, c_1, c_2) hilfreich, wie später gezeigt werden wird.

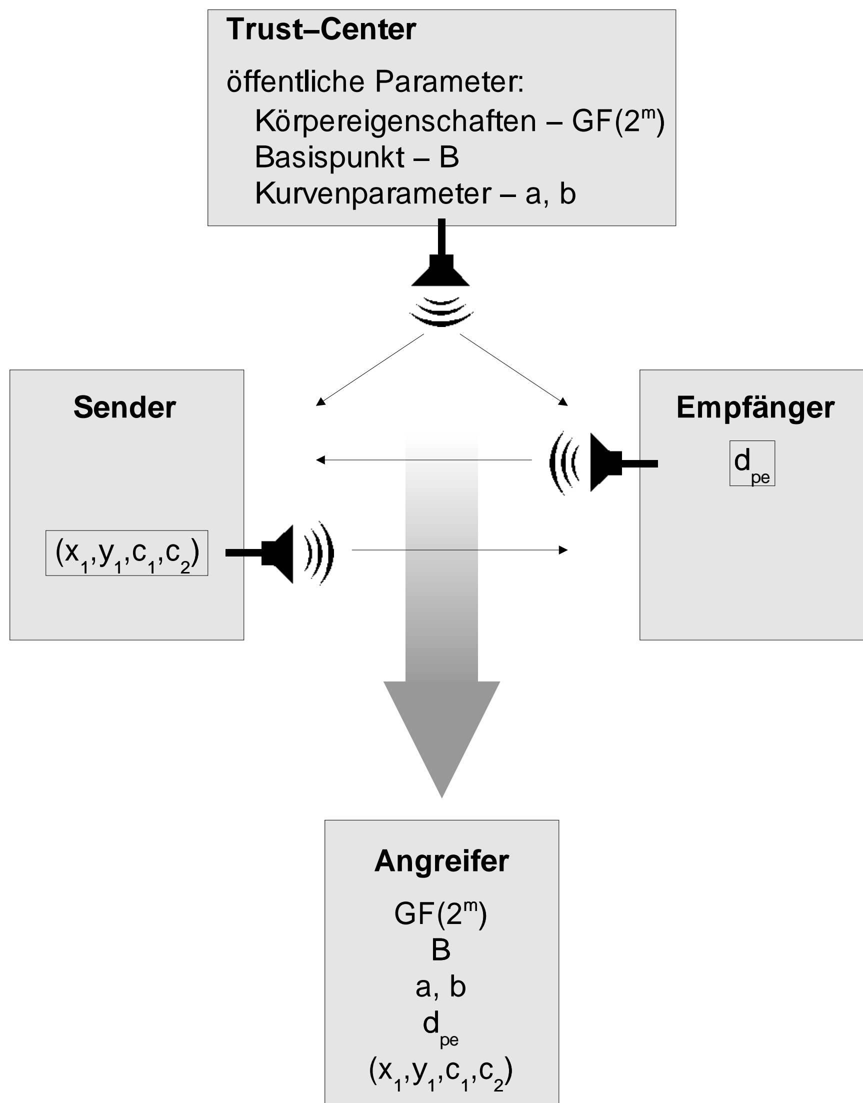

Abbildung 9: Informationen eines Angreifers

5.2 Kryptosysteme mit skalierbarer Sicherheit

Die Motivation, ein Kryptosystem brechen zu wollen, steht immer in Verbindung mit der dadurch zu erlangenden Information. Dem durch das Überwinden des Systems zu erwartenden Gewinn steht immer der Aufwand gegenüber, der sich folgendermaßen zusammensetzt:

$$\text{Aufwand} = \text{Kosten} \cdot \text{Zeit}^2 \tag{28}$$

Je nach Wert oder Wichtigkeit der Information ist die Stufe der Sicherheit, d.h. die Länge bzw. Bitbreite der Schlüssel ausgelegt. Je breiter die jeweiligen Schlüssel sind, umso größer ist der Aufwand zum Ver- oder Entschlüsseln einer Information. Dies ist insbesondere für Low-Security-Systeme wichtig. Low-Security-Systeme sollen ein geringwertiges Gut mit geringem Aufwand möglichst gut schützen. Als Anwendungen bieten sich z.B. Pkw-Mautkarten, Pkw-Parkkarten, Prepaid-Handykarten, Pkw-Funkschlüssel, Pkw-Wegfahrsperren, digitale Preisschilder und Zugangskontrollen an. Dabei authentifiziert sich der Benutzer gegenüber dem „Anbieter“ durch eine Chipkarte, einem digitalen Autoschlüssel o.ä. Es wird dabei sofort klar, daß sich in Chipkarten, den sog. Smart-Cards oder Autoschlüsseln nicht beliebig umfangreiche Kryptotechnologie implementieren läßt. Deshalb existieren in vielen denkbaren Anwendungen Obergrenzen für manche Kryptosystemparameter:

Zeit: Führt man beispielsweise einen Autoschlüssel eines Fahrzeugs mit Wegfahrsperre in das Zündschloß ein, so sollten in der Zeit, die der Fahrer zum Einstecken und Umdrehen des Schlüssels benötigt, die Datenübertragungen zwischen dem Schlüssel und der Wegfahrsperre erfolgt sein, um den Schlüssel zu authentifizieren. Die maximal zulässige Zeit für den Authentifizierungsvorgang liegt bei ca. 0.3 Sekunden. Dieser Zeitraum ist die Obergrenze für die Empfindung, daß ein Vorgang in „Realtime“ abläuft. Da ein Fahrzeugschlüssel das Fahrzeug inklusive Inhalt vor Diebstahl schützen soll, muß das System trotz der zeitlichen Maßgabe hinreichend sicher gestaltet werden. Wählt man deshalb die Schlüsselbitbreite zu groß, so könnte die maximale Reaktionszeit zur Deaktivierung der Wegfahrsperre überschritten werden.

Kosten: Wichtig bei der Konzeption eines Kryptographieprozessors oder einer Chipkarte ist die Kostenobergrenze. Je höher der Integrationsgrad der gewählten Fertigungsmethode, desto höher die Produktionskosten. Einen großen Einfluß auf diesen Faktor hat die Produktionsmenge. Soll das Produkt in Massenfertigung produziert werden, verringert dies die Produktionskosten. Vorteile von höherer Integration des Bauteils kann die Einsparung von Platz und ein geringerer Energieverbrauch sein.

Energie: Bei vielen denkbaren Anwendungen wirkt die Menge der zur Verfügung stehenden elektrischen Energie leistungsbegrenzend. In dem o.g. Autoschlüssel befindet sich entweder eine Batterie in Form einer Knopfzelle, oder die Energie wird durch Induktion in den Schlüssel übertragen. In beiden Fällen ist nur wenig Energie vorhanden, so

daß die Leistungsaufnahme der Kryptographiehardware entsprechend gering sein muß.

Größe: Die geringen Abmessungen eines Autoschlüssels lassen den Einsatz eines Prozessors mit großen Ausmaßen vielleicht noch zu, Energie verbraucht in Form von Batterien aber deutlich mehr Raum und stellt hier den kritischsten Faktor dar.

Offensichtlich beeinflußt die Wahl der Schlüsselbitbreite am deutlichsten alle Systemparameter. Eine geringere Bitbreite verursacht weniger Chipfläche, weniger Zeitaufwand, weniger Energieeinsatz, weniger Entwicklungs- und Produktionskosten etc. Aus diesen Gründen ist die Auswahl eines geeigneten Kryptographieverfahrens nötig, das das beste Verhältnis von Schlüssellänge zu Sicherheit aufweist, um möglichst kurze Schlüssel bei maximaler Sicherheit einsetzen zu können.

Das derzeit am weitesten verbreitete Kryptosystem ist RSA, das 1978 von Rivest, Shamir und Adleman vorgestellt wurde. Der öffentliche Schlüssel enthält den sog. RSA-Modul. Er ist das Produkt zweier großer Primzahlen. Sollte es einem Angreifer gelingen, diese Primzahlen zu finden, so kann er auch die geheimen Schlüssel berechnen und das System ist gebrochen. Die Falltürfunktion des RSA-Systems ist somit komplexitätstheoretisch äquivalent zum Integer-Faktorisierungs-Problem (Integer-Factorization-Problem, IFP). Da das Problem um so schwerer zu lösen ist, je größer die Integerzahl wird, ist die Schlüssellänge der sicherheitsrelevante Faktor.

Lenstra und Verheul vergleichen in [LeVe99] die Schlüssellängen von RSA und ECC bei äquivalenter Sicherheit. Ausgehend von der als sicherheitstechnisch hinreichend anzunehmenden Anzahl von Mips-Jahren[4] zum Berechnen der jeweiligen Falltürfunktionen werden die dazu notwendigen Hardwarekosten ermittelt. Daraus läßt sich eine Abschätzung der unteren Schlüssellänge des jeweiligen Kryptosystems vornehmen. Lenstra und Verheul beginnen die tabellarische Ausführung mit dem Jahr 1982. Zu diesem Zeitpunkt waren schon zahlreiche Veröffentlichungen zur Abschätzung der RSA-Systemsicherheit existent. Mit zunehmender Publizität eines Kryptosystems steigt auch der Umfang der Forschung an diesem System. Da sich ECC verglichen mit RSA in der Anfangsphase der Untersuchungen befindet, geben Lenstra und Verheul nicht nur die aus heutiger Sicht zu RSA äquivalente Schlüssellänge an. Sie lassen zusätzlich einen Index c einfließen, der die Fortentwicklung der Forschung über ECC beschreibt und sich mit auf die minimale, sichere Schlüssellänge auswirkt. Der Index c ist die Anzahl von Monaten, in denen sich die Effizienz von Algorithmen und Methoden, ECC zu brechen, verdoppelt. Sie stellen in [LeVe99] als Standardwert $c = 18$ Monate ein, da dies, verglichen mit dem Forschungsfortschritt bei RSA, vergleichbar sei.

Erstellt man aus der in [LeVe99] angegebenen Datentabelle der Schlüsselbreiten eine graphische Darstellung, so ergibt sich Abbildung 10. Diese zeigt, daß das RSA-Verfahren

[4][LeVe99, S. 3]: Def.: 1 Mips Jahr = Berechnungen, die in einem Jahr auf einer DEC VAX 11/780 durchgeführt werden können. Die Einheit ist nicht ohne Kritik, da Vergleiche mit anderen Prozessoren durch unterschiedliche Instruktionssätze schwierig sind. Siehe: R.D. Silverman, *Exposing the Mystical MIPS Year*, IEEE Computer, August 1999, pp. 22-26

bei vergleichbarer Sicherheit eine deutlich höhere Schlüsselbreite als ECC verlangt.

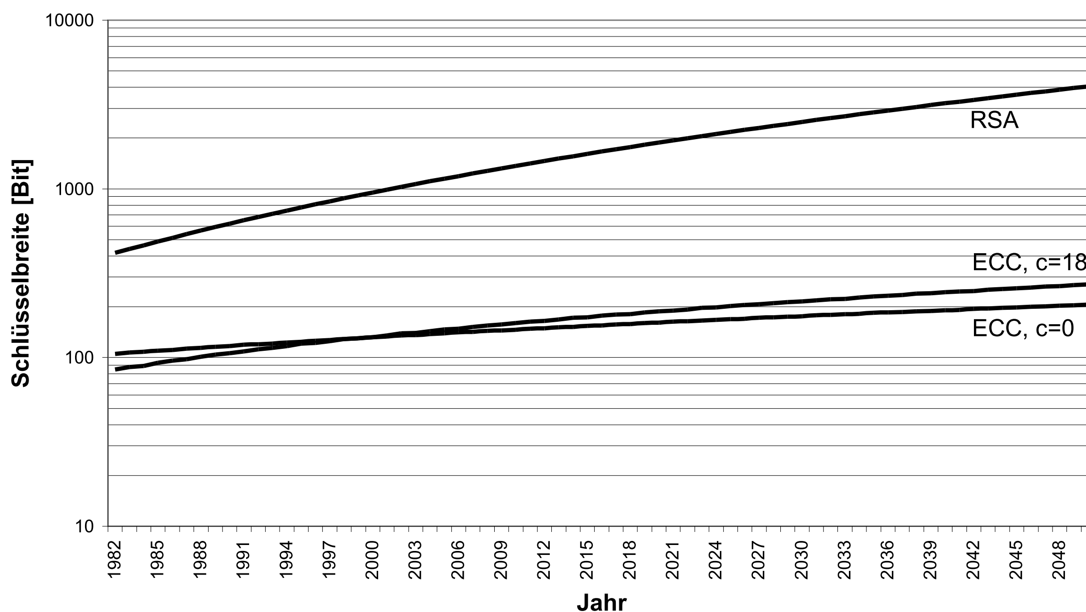

Abbildung 10: Schlüsselbitbreite von RSA und ECC von 1982 bis 2050

Nach o.a. Formel sind die Aufwandfaktoren zum Brechen eines Kryptoverfahrens Zeit und Kosten. Lenstra und Verheul zeigen, daß 1982 der als sicher einzuschätzende Berechnungsaufwand bei $5 \cdot 10^5$ Mips Jahren lag. Die untere Kostenschranke für Hardware mit solcher Leistungsfähigkeit habe 1982 bei $3.98 \cdot 10^7$ US-$ pro Tag gelegen. Im Jahr 2000 gilt ein Aufwand von $7.13 \cdot 10^9$ Mips Jahren als hinreichend sicher; die Kosten werden dann bei $1.39 \cdot 10^8$ US-$ pro Angriffstag liegen. Die Prognose für 2010 sagt einen Rechenaufwand von $1.45 \cdot 10^{12}$ Mips Jahren bei Hardwarekosten von $2.77 \cdot 10^8$ US-$ pro Tag voraus.

In der Abbildung 10 ist die Datenreihe für ECC ($c = 18$) von 1982 an eingezeichnet. Da vor 1999 allerdings noch keine nennenswerten Fortschritte zur Brechung des ECC-Kryptosystems gemacht wurden, haben Lenstra und Verheul die Werte nur zum Vergleich angegeben. Dasselbe gelte für die Prognose der Schlüsselbitbreiten nach dem Jahr 2040. Der steigenden Leistungsfähigkeit der Hardware entsprechend, müssen zur Erhaltung der Sicherheit des Kryptosystems die Schlüsselbitbreiten steigen. 1982 genügte für RSA eine Schlüsselbreite von 417 Bit, ECC ($c = 0$) benötigte 105. Im Jahre 2000 waren bei RSA schon 952 Bit nötig, während ECC ($c = 0$) noch mit 132 Bit auskam. Lenstra und Verheul beginnen mit dem Einfluß der Forschung auf die ECC-Schlüssellängen im Jahr 2000, so daß auch ECC ($c = 18$) 132 Bit benötigt. Diesem Prinzip folgend, steigt der Bitbedarf von ECC ($c = 18$) im Jahre 2005 auf 147, während ECC ($c = 0$) nur 139 Bit benötigt. RSA braucht dann 1.149 Bit.

Zum besseren Vergleich der Entwicklung der Schlüsselbitbreiten und deren Einflußfaktoren ist in Abbildung 11 der für diese Arbeit interessante Bereich der Jahre 2000 bis 2010

indexiert dargestellt. Hierbei fällt auf, daß die von Lenstra und Verheul als sicher angesehene minimale Berechnungsschranke der Falltürfunktionen (in Abb. 11: Mips-Jahre) sehr stark ansteigt. Sie erreicht im Jahr 2010 den Wert 20.300 % (in Worten: zwanzigtausenddreihundert). Die aus dem nötigen Berechnungsaufwand zum Brechen eines

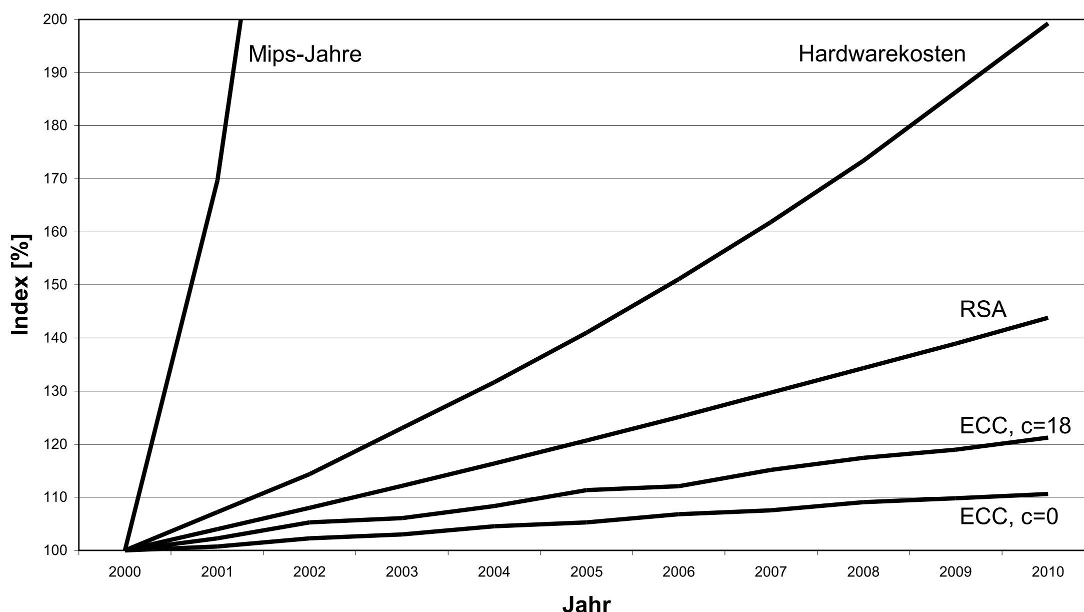

Abbildung 11: Indexierte Entwicklung der Schlüsselbitbreiten von 2000 bis 2010

Systems resultierenden Kosten für eine hinreichende Hardware werden auf der Grundlage des Moore´schen Gesetzes angegeben. Das Moore´sche Gesetz besagt, daß sich die Dichte der Komponenten eines integrierten Schaltkreises etwa alle 18 Monate verdoppelt. Daraus ergibt sich die allgemein akzeptierte Interpretation, daß sich auch die Rechenleistung eines Chips alle 18 Monate verdoppelt. Es ist fraglich, ob dies in den nächsten Jahren aufrecht erhalten werden kann, da der heutigen Standardtechnologie der Chipfertigung durch die Minimierung Grenzen gesetzt sind, die nahezu erreicht sind. Um eine Verdoppelung der Komponentendichte auch in Zukunft in dieser zeitlichen Geschwindigkeit zu erreichen, werden bald neue Technologien entwickelt werden müssen.

5.3 Brute-Force-Methode mittels Punktaddition

Zum gegenwärtigen Zeitpunkt ist kein strukturiertes Angriffsszenario auf ein ECC-Kryptosystem bekannt, das in einer akzeptablen Laufzeit arbeitet. Diese Aussage bezieht sich natürlich nur auf Implementationen von ECC-Kryptosystemen, bei denen das zuständige Trust-Center alle bisher bekannten Schwachstellen ausgeschlossen hat. Durch Abwesenheit eines strukturierten Angriffsszenarios mit kurzer Laufzeit kann nur in Form eines „Brute-Force-Angriffs“, d.h. mittels „Durchprobieren“ aller möglicher Schlüssel zu einem Erfolg

führen. Durch die hohe Anzahl der in Frage kommenden Schlüssel, ist die Geschwindigkeit des „Durchprobierens“ der interessanteste Faktor.
Man nehme an, es liege ein Kryptosystem in Form des ElGamal-Verfahrens, wie in Abschnitt 4.3 beschrieben, vor. Nachfolgende Werte sind dann öffentlich bekannt (vgl. Abbildung 9): die elliptische Kurve mit ihren Parametern a und b, die Felddimension $\mathcal{GF}(2^m)$, das Primpolynom, der Basispunkt B, der öffentliche Schlüssel des Empfängers d_{p_e} und der öffentliche Schlüssel des Senders d_{p_s}. Hat man ein oder mehrere Pseudotextquadrupel der Form (x_1, y_1, c_1, c_2) mithören können, so benötigt man in erster Linie den Wert x_2 zur Dechiffrierung des Klartextes. Es gilt: $(x_2, y_2) = k \cdot d_{p_e} \wedge (x_1, y_1) = k \cdot B$.
Von letzterer Gleichung sind der Basispunkt B, sowie der Punkt $P = (x_1, y_1)$ bekannt, da er Teil des mitgehörten Quadrupels ist. Sollte es gelingen, daraus das k zu ermitteln, so ließe sich mit diesem durch Einsetzen in erstere Gleichung der Punkt (x_2, y_2) bestimmen. Damit wäre eine Dechiffrierung der Nachricht geglückt.
Leider ist die Punktdivision in der Arithmetik mit elliptischen Kurven nicht definiert, so daß die Ermittlung des k in der Gleichung $(x_1, y_1) = k \cdot B$ bei bekannten (x_1, y_1) und B mittels probieren erfolgen muß.
Die naheliegende Methode ist, das k sukzessive von Eins ausgehend zu Inkrementieren und mit dem Basispunkt zu Multiplizieren. Danach vergleicht man das Ergebnis mit dem Punkt $P = (x_1, y_1)$. Sind diese beiden Punkte identisch, so ist die bisherige Anzahl der Multiplikationen das gesuchte k. In Algorithmus 1 ist dies beispielhaft dargestellt.

```
1  FOR k = 1 TO (#E - 1) DO
2  {
3    testpunkt = k * B;
4    IF testpunkt = P
5      THEN PRINT "k gefunden, k =",k ; END;
6    ELSE NEXT k;
7  }
```

Algorithmus 1: Punktaddition durch Skalarmultiplikation

Die Multiplikation eines Punktes einer elliptischen Kurve mit einem Skalar wird auf die sog. Additions-Subtraktions-Methode zurückgeführt.[5] Da hier der Skalar sukzessive um den Faktor eins erhöht wird, läßt sich der Ablauf am einfachsten wie in Algorithmus 2 formulieren.
Das Finden des jeweils nächsten Punktes ist jetzt eine einfache affine Punktaddition, die bereits mit den Gleichungen (5) - (7) auf Seite 14 definiert wurde. Dabei ist relevant, daß die erste Addition eine Punkt*verdoppelung* darstellt. Deshalb müssen zur Berechnung die Gleichungen (8) - (10) benutzt werden.
Um einen neuen Punkt zu errechnen, benötigt man bei der ersten Operation, der affinen Verdoppelung, 5 Additionen, 3 Multiplikationen und eine Berechnung des multiplikativen

[5]Eine Beschreibung findet man in [IEEE99, S. 126]

```
1  testpunkt = (0,0);   /* Setze testpunkt auf Unend-
lichkeitspunkt */
2  FOR k = 1 TO (#E - 1) DO
3  {
4    testpunkt = testpunkt + B;
5    IF testpunkt = P
6      THEN PRINT "k gefunden, k =",k ; END;
7    ELSE NEXT k;
8  }
```

Algorithmus 2: Punktaddition durch sukzessive Addition

Inversen. Für alle weiteren Berechnungen, die dann die affine Punktaddition darstellen, benötigt man 8 Additionen, 2 Multiplikationen und einmal die Errechnung des multiplikativen Inversen.

Hat man einmal das gesuchte k ermittelt, so ist das gesamte Kryptosystem für die von den beiden Kommunikationspartnern ausgewählten geheimen Schlüssel gebrochen. Da man über den Kommunikationskanal sehr viele verschiedene Nachrichtenpakete (x_1, y_1, c_1, c_2) mitschreiben kann, besteht die Möglichkeit nicht nur nach einem P, sondern nach vielen verschiedenen zu suchen. Dieser Aspekt wird später noch einmal aufgegriffen werden.

Bei der sukzessiven Addition des Basispunktes kann ein Schwachpunkt der elliptischen Kurve zu Tage treten, der schon in Kapitel 5.1 erwähnt wurde: durch Wahl einer Kurve mit nicht primer Kurvenordnung $\#E$ können durch die sukzessive Addition des Basispunktes nicht alle Punkte der elliptischen Kurve aufgezählt werden. Welche und wieviele Punkte gefunden werden, hängt von der Ordnung des verwendeten Basispunktes und der Ordnung der Kurve ab.

Definition [IEEE99, S. 121]**:** Die Ordnung eines Punktes P einer elliptischen Kurve ist die kleinste positive Zahl r, so daß $r \cdot P = \mathcal{O}$. Die Ordnung eines Punktes existiert immer und ist ein Teiler der Kurvenordnung $\#E(\mathcal{GF}(2^m))$. Zusätzlich gilt: $k, l \in \mathbb{N}$, wenn $k \cdot P = l \cdot P$ gdw. $k \equiv l \pmod r$.

Eine elliptische Kurve E über $\mathcal{GF}(2^m)$ verfügt über eine endliche Anzahl von Punkten, die zuzüglich des Unendlichkeitspunktes die Kurvenordnung $\#E$ darstellen. Nach dem Theorem von Hasse [BlSS99, S. 103] gilt:

$$2^m - 2\sqrt{2^m} + 1 \;\leq\; \#E \;\leq\; 2^m + 2\sqrt{2^m} + 1. \tag{29}$$

D.h., daß die Kurvenordnung $\#E$ näherungsweise 2^m beträgt. Die Beispielkurve aus Abschnitt 3.3.2 bezieht sich auf den Körper $\mathcal{GF}(2^3)$ und hat eine Ordnung von 14. Nach obiger Definition können also die Punkte der Kurve ihrer Ordnung nach in Gruppen aufgeteilt werden. Hier sind die Gruppen 2, 7 und 14 möglich, da diese Punktordnungen die Kurvenordnung mit ganzzahligem Ergebnis teilen. Wählt man einen beliebigen Basispunkt B, z.B. $B = (2, 4)$, so errechnet man durch die sukzessive Addition des Basispunktes mit

sich selbst so viele Kurvenpunkte, wie die Ordnung des Basispunktes - 2. Tabelle 2 zeigt die Errechnung der Punktordnung anhand dreier Punkte verschiedener Ordnungsgruppen. Auf der hier gezeigten Beispielkurve existieren Punkte jeder möglichen Punkteordnung, wie in Tabelle 3 gezeigt. Mit der Methode der Punktaddition erreicht man also nur dann

$B_a = (0,1)$	$B_b = (5,7)$	$B_c = (2,4)$
$2 \cdot B_a = \mathcal{O}$	$2 \cdot B_b = (3,4)$	$2 \cdot B_c = (3,7)$
	$3 \cdot B_b = (7,1)$	$3 \cdot B_c = (4,1)$
	$4 \cdot B_b = (7,6)$	$4 \cdot B_c = (7,1)$
	$5 \cdot B_b = (3,7)$	$5 \cdot B_c = (6,6)$
	$6 \cdot B_b = (5,2)$	$6 \cdot B_c = (5,7)$
	$7 \cdot B_b = \mathcal{O}$	$7 \cdot B_c = (0,1)$
		$8 \cdot B_c = (5,2)$
		$9 \cdot B_c = (6,0)$
		$10 \cdot B_c = (7,6)$
		$11 \cdot B_c = (4,5)$
		$12 \cdot B_c = (3,4)$
		$13 \cdot B_c = (2,6)$
		$14 \cdot B_c = \mathcal{O}$

Tabelle 2: Sukzessive Addition eines Basispunktes

Ordnung	Punkte
2	(0, 1)
7	(3, 4), (3, 7), (5, 2), (5, 7), (7, 6)
14	(2, 4), (2, 6), (4, 1), (4, 5), (6, 0), (6, 6), (7, 1)

Tabelle 3: Verschiedene Gruppen der Punktordnung

alle Punkte der Kurve, wenn die Ordnung des Basispunktes gleich der Ordnung der elliptischen Kurve ist.
Wie bereits in Kapitel 3.2 auf Seite 7 bemerkt, ist vor allem das Finden eines multiplikativen Inversen i.A. eine extrem langwierige und rechenintensive Operation. Da bei jeder Punktverdoppelung u.a. eine Inversion notwendig ist, ist diese Methode als sehr zeitaufwendig einzustufen.

5.4 Brute-Force-Methode mittels Punktverdoppelung

Eine einfachere und weniger teure Möglichkeit als die affine Punktaddition ist die Verdoppelung von Punkten in projektiver Form, die folgendermaßen definiert ist:
Sei $P = (X_1, Y_1, Z_1)$ der zu verdoppelnde Punkt, dann gilt $2 \cdot (X_1, Y_1, Z_1) = (X_2, Y_2, Z_2)$, mit

$$Z_2 = X_1 Z_1^2 \tag{30}$$

$$X_2 = (X_1 + cZ_1^2)^4 \tag{31}$$

$$U = Z_2 + X_1^2 + Y_1 Z_1 \tag{32}$$

$$Y_2 = X_1^4 Z_2 + U X_2 \tag{33}$$

Die Konstante c in (31) errechnet sich folgendermaßen:

$$c = b^{2^{m-2}} \tag{34}$$

Für die Durchführung einer Punktverdoppelung dieser Form sind vier Additionen, fünf Multiplikationen und fünf Quadrierungen notwendig. Gleichung (34) muß nur einmal berechnet werden. Das Entscheidende ist, daß hier keine Division benötigt wird, und somit kein Zeitaufwand für eine Inversion anfällt. Für das Gesamtkonzept bedeutet die Benutzung der projektiven Koordinatendarstellung, sowie die Benutzung der Verdoppelungs- anstelle einer sukzessiven Additionsstrategie allerdings einige Veränderungen.
Der Vorteil der projektiven Punktverdoppelung über $\mathcal{GF}(2^m)$ ist u.a. die einfache und schnelle Realisierung in Hardware. Aus diesem Grund scheint es die erfolgversprechendste Brute-Force-Angriffsstrategie für Low-Security ECC-Kryptosysteme zu sein. Allerdings gilt auch hier, daß das Trust-Center eine Kurve mit primer Ordnung $\#E$ verwenden sollte.
Betrachte man wieder die Beispielkurve aus Abschnitt 3.3.2. Sie bezieht sich auf den Körper $\mathcal{GF}(2^3)$ und hat eine Ordnung von 14. Wählt man wieder den Basispunkt B = (2, 4, 1) und verdoppelt ihn sukzessive, so erhält man die Punkte der Tabelle 4. Es sind hier die Resultate der Verdoppelungen mittels der Gleichungen (30) - (34) angegeben. Mit Hilfe von Gleichung (12) auf Seite 17 kann aus den projektiven Koordinaten das affine Äquivalent ermittelt werden. Bereits nach der ersten Verdoppelung gerät die Strategie in

B					=	(2, 4, 1)
$2 \cdot B$	=	2·(2, 4, 1)	=	(7, 2, 2)	=	(3, 7, 1)
$4 \cdot B$	=	2·(2·(2, 4, 1))	=	(6, 4, 3)	=	(7, 1, 1)
$8 \cdot B$	=	2·(2·(2·(2, 4, 1)))	=	(4, 4, 7)	=	(5, 2, 1)
$16 \cdot B$	=	2·(2·(2·(2·(2, 4, 1))))	=	(2, 4, 5)	=	(3, 7, 1)
$32 \cdot B$	=	2·(2·(2·(2·(2·(2, 4, 1)))))	=	(6, 4, 3)	=	(7, 1, 1)

Tabelle 4: Sukzessive Verdoppelung eines Basispunktes

eine Schleife von drei Punkten (3, 7), (7, 1) und (5, 2), aus der die Berechnung nicht mehr herausführt.
Vergleicht man die Methoden der Addition mit der der Verdoppelung, bezogen auf denselben Basispunkt, z.B. (2, 4, 1), so ergibt sich Tabelle 5. Es zeigt sich, daß die Verdoppelungsstrategie genau dann in eine kurze Schleife kommt, wenn der Multiplikationsfaktor k größer als die Kurvenordnung wird. Alle Punkte einer elliptischen Kurve erreicht man demnach nur, wenn die Kurvenordnung prim ist. Dann sind alle Punktordnungen gleich

Addition			Verdoppelung		
B	=	$(2,4)$	B	=	$(2,4)$
$2 \cdot B$	=	$(3,7)$	$2 \cdot B$	=	$(3,7)$
$3 \cdot B$	=	$(4,1)$			
$4 \cdot B$	=	$(7,1)$	$4 \cdot B$	=	$(7,1)$
$5 \cdot B$	=	$(6,6)$			
$6 \cdot B$	=	$(5,7)$			
$7 \cdot B$	=	$(0,1)$			
$8 \cdot B$	=	$(5,2)$	$8 \cdot B$	=	$(5,2)$
$9 \cdot B$	=	$(6,0)$			
$10 \cdot B$	=	$(7,6)$			
$11 \cdot B$	=	$(4,5)$			
$12 \cdot B$	=	$(3,4)$			
$13 \cdot B$	=	$(2,6)$			
$14 \cdot B$	=	$\mathcal{O}$			
$15 \cdot B$	=	$(2,4)$			
$16 \cdot B$	=	$(3,7)$	$16 \cdot B$	=	$(3,7)$
$17 \cdot B$	=	$(4,1)$			
$18 \cdot B$	=	$(7,1)$			
	$\vdots$			$\vdots$	

Tabelle 5: Fortschrittvergleich der Verfahren

der Kurvenordnung, da es keine Punktordnung gibt, die die prime Kurvenordnung ganzzahlig teilt.

Leider reicht die Bedingung, daß die Ordnung der Kurve prim sein muß, nicht aus um alle Punkte mittels Verdoppelung zu errechnen. Man wird nicht alle Punkte erreichen, wenn die Kurvenordnung $\#E$ eine Fermat'sche Primzahl P_f

$$P_f = 2^{2^n} + 1$$

oder eine Mersenn'sche Primzahl P_m

$$P_m = 2^n - 1$$

ist.

Man erreicht genau dann mittels sukzessiver Verdoppelung des Basispunktes alle Kurvenpunkte, wenn die kleinste Primitivwurzel der Kurvenordnung gleich 2 ist. Nach der Artin'schen Vermutung über Primitivwurzeln haben ca. 37 % der ungeraden Primzahlen die Primitivwurzel 2. Eine Tabelle der Primitivwurzeln der Primzahlen < 1000 findet man in [Apo76, S. 213].

Ähnlich dem im folgenden Kapitel 5.5 beschriebenen Pollard-Lambda Algorithmus führt man das Verfahren der Punktverdoppelung immer zum Ziel, wenn man nach einer gewissen fruchtlosen Zeit den Ablauf als „Sackgasse" betrachtet und mit einem veränderten

Startwert neu beginnt. Dieser neue Startpunkt muß natürlich von $k^i \cdot B$ verschieden sein, um nicht in die gleiche Schleife zu geraten.
Welche Eigenschaften die zu brechende elliptische Kurve hat, ist von der Institution abhängig, die diese Kurve für ein Kryptoverfahren errechnet und freigibt, dem sog. Trust-Center. Das Trust-Center erzeugt Kurven, prüft sie auf Schwachpunkte, gibt deren Ordnung und zu verwendende Basispunkte an. Schwachpunkte sind die schon erwähnte Supersingularität oder eben auch die Teilbarkeit der Kurvenordnung. Je mehr Teiler die Ordnung einer Kurve besitzt, umso größer ist die Wahrscheinlichkeit, das k zu finden; der Punkt $k \cdot B$ kann ja eben nur in der selben Punktordnungsgruppe wie B sein. Hat die Kurvenordnung einen Primteiler wie beispielsweise die 2, so kommen nur noch die Hälfte aller Kurvenpunkte als zu findender Punkt $k \cdot B$ in Frage.

5.5 Die Methoden von Pollard und Shanks

Im Juli 1978 stellte J. M. Pollard in [Pol78] ein Verfahren vor, mit dem er das diskrete Logarithmus Problem (DLP) in kurzer Laufzeit lösen konnte. Die sog. Pollard Rho Methode gilt bislang als die schnellste Angriffsstrategie auf das allgemeine EC-DLP.
Gegeben sei ein Punkt P einer elliptischen Kurve, der sich in einer Punktuntergruppe mit Punktordnung n befindet, die von einem Basispunkt B erzeugt wird. Gesucht sei k, für das gilt: $P = k \cdot B$. Nach der Pollard ρ-Methode geht man dann folgendermaßen vor [WiZu98]: Man teilt die Punkte der elliptischen Kurve in drei möglichst gleich große Teilmengen S_1, S_2 und S_3. Dann definiert man folgende Iterationsfunktion für einen Punkt Z:

$$f(z) = \begin{cases} 2\,Z & \text{wenn} \quad Z \in S_1 \\ Z + P & \text{wenn} \quad Z \in S_2 \\ Z + Q & \text{wenn} \quad Z \in S_3 \end{cases}$$

Nun wählt man zufällig ein A_0, $B_0 \in [1, n - 1]$ und berechnet einen Startpunkt $Z_0 = A_0 \cdot P + B_0 \cdot Q$. Da in der Iteration A_i und B_i weiter verändert werden sollen, benutzt man für die weiteren Iterationsschritte folgende Funktion:

$$(Z_{i+1}\,,\, A_{i+1}\,,\, B_{i+1}) = \begin{cases} (2\,Z_i\,,\, 2\,A_i\,,\, 2\,B_i) & \text{wenn} \quad Z \in S_1 \\ (Z_i + P,\, A_i + 1,\, B_i) & \text{wenn} \quad Z \in S_2 \\ (Z_i + Q,\, A_i\,,\, B_i + 1) & \text{wenn} \quad Z \in S_3 \end{cases}$$

Während der Iteration können A_i und B_i modulo n genommen werden, um die Entstehung zu großer Werte zu verhindern. Die Iterationsergebnisse (Z_i, A_i, B_i) eines jeden Schrittes werden zum späteren Vergleich mitprotokolliert. Da die Anzahl der Punkte auf einer elliptischen Kurve endlich sind, befinden sich die Werte von Z_i ab einem gewissen Punkt in einem Zyklus. Die grafische Darstellung des Verfahrensablaufs erklärt die Namensgebung „Rho-Methode", wie Abbildung 12 zeigt. Nach jedem Iterationsschritt müssen die bisher errechneten Z_i mit dem aktuell errechneten Z_j verglichen werden. Sollte sich in

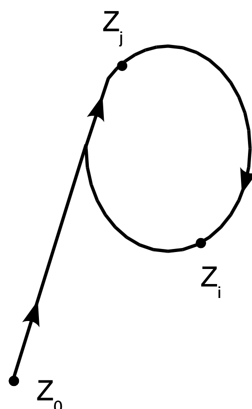

Abbildung 12: Das Pollard - Rho

einem Iterationsschritt $Z_i = Z_j$ ergeben, so gilt $A_i \cdot P + B_i \cdot Q = A_j \cdot P + B_j \cdot Q$, und somit $l = \frac{A_i - A_j}{B_j - B_i} \bmod n$, oder man hat unglücklicherweise den Fall erwischt, daß $B_i \equiv B_j \pmod{n}$. Die durchschnittliche Anzahl der zum Finden des k benötigten Iterationen ist $\sqrt{\frac{\pi n}{2}}$. Es gibt sehr viele Arbeiten zur Verbesserung der Laufzeit dieser Strategie. Möglichkeiten zur Beschleunigung ergeben sich besonders durch Ausnutzung gewisser Eigenschaften spezieller Kurvenarten, die eine Einschränkung der in Frage kommenden Schlüssel erlauben. Details sind z.B. in [WiZu98] zu finden.

Eine ebenfalls von Pollard in [Pol78] vorgestellte Strategie ist die Pollard-Lambda-Methode, die i.A. auch als Kangaroo-Fangmethode (method for catching kangaroos) bekannt ist. Sie gilt als eine der effizientesten Methoden zur Lösung des DLP und funktioniert folgendermaßen [StTe99]: sei G eine endliche, zyklische Gruppe die von einem Gruppenelement g erzeugt wird, und $h \in G$. Dann existiert ein positives x, für das gilt: $h = g^x$. Analog zu Abschnitt 4.1 auf Seite 20 wird x als der *diskrete Logarithmus* von h zur Basis g bezeichnet, dessen Wert der Angreifer berechnen möchte. Man nehme an, daß ganze Zahlen a und b bekannt sind, für die gilt: $a \leq x < b$. Nun stelle man sich zwei Kangaroos vor, ein zahmes Kangaroo Z mit Ausgangspunkt $z_0 = g^b$ und ein wildes Kangaroo W mit Ausgangspunkt $w_0 = h$. Das zahme Kangaroo Z startet nun am oberen Ende des Intervalls $[a, b[$, während W von einem unbekannten Ort x aus startet. Sei $\delta_0(Z) = b$ die Distanz von Z zum Ursprung und $\delta_0(W) = 0$ die Distanz von W zu h. Sei $S = \{g^{s_1}, \ldots, g^{s_r}\}$ eine Menge von Kangaroo-Sprüngen und v: $G \to \{1, \ldots, r\}$ eine Hash-Funktion. Man betrachte die s_i als zurückgelegte Wegstrecken, die im Vergleich zur Intervallänge $b - a$ klein sind. Jetzt bewegt sich das zahme Kangaroo auf dem Weg $z_{j+1} = z_j \cdot g^{s_{v(z_j)}}, j \in \mathbb{N}_0$ durch die Gruppe. Zusätzlich zum Weg (z_j) berechnet man die zurückgelegten Streckenlängen $\delta_j(Z)$: $\delta_{j+1}(Z) = \delta_j(Z) + s_{v(z_j)}, j \in \mathbb{N}_0$. Somit gilt: $z_j = g^{\delta_j(Z)}\ \forall\, j \in \mathbb{N}_0$. Nach einigen Sprüngen hält das zahme Kangaroo an und installiert an seinem erreichten Endpunkt z_M eine Falle, bestehend aus einem tiefen Loch, daß gut durch Zweige abgedeckt ist [Pol78]. Dann springt das wilde Kangaroo los und bewegt sich auf dem Weg w_{j+1}

$= w_j \cdot g^{s_{v(w_j)}}$, $j \in \mathbb{N}_0$ mit den Streckenlängen $\delta_j(W)$: $\delta_{j+1}(W) = \delta_j(W) + s_{v(w_j)}, j \in \mathbb{N}_0$. Nun gilt: $w_j = h \cdot g^{\delta_j(W)}, j \in \mathbb{N}_0$. Bei der Bewegung des zahmen Kangaroos war dessen Position nach jedem Sprung eindeutig, da die Ausgangsposition bekannt war. Im Fall des wilden Kangaroos kennt man die Ausgangsposition nicht, somit kann man seine Position nicht bestimmen. Deshalb ist W wild [StTe99]. Nach jeden Sprung von W wird geprüft, ob es in die Falle geraten ist. Wenn dies der Fall ist, z.B. an der Stelle w_N, führt die Gleichung $t_M = w_N$ zu einer Lösung von $g^x = h$, da $x = \delta_M(Z) - \delta_N(W)$. Wenn W nach einer gewissen Anzahl von Sprüngen noch nicht in die Falle geraten ist, wird es angehalten und ein neues, wildes Kangaroo wird gestartet. Diesmal startet es vom Punkt $w_0 = h \cdot g^z$ mit einer Startdistanz $\delta_0(W) = z$ mit kleinem z. Sollte dieses Kangaroo W in die Falle von Z fallen, so impliziert dies, daß sich die Wege der beiden an einem Punkt getroffen haben müssen, von dem aus sie dann identisch waren. Wieder erklärt die grafische Darstellung des Verfahrens in Abbildung 13 die Namensgebung „Lambda-Methode". Die

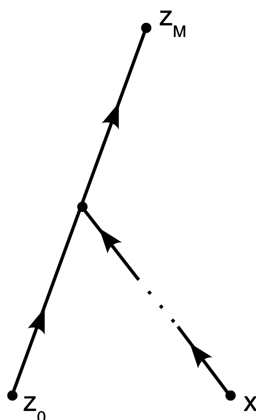

Abbildung 13: Das Pollard - Lambda

Laufzeit des Verfahrens ist minimal, wenn der Durchschnittswert von S ungefähr $\frac{\sqrt{b-a}}{2}$ beträgt und das zahme Kangaroo ungefähr $0{,}7 \cdot \sqrt{b-a}$ Sprünge macht, bevor es die Falle aushebt. Das wilde Kangaroo muß nach spätestens $2{,}7 \cdot \sqrt{b-a}$ Sprüngen in der Falle liegen, was mit einer Wahrscheinlichkeit von 75% geschieht, oder es wird angehalten und durch ein Neues ersetzt. Damit liegt die gesamte Laufzeit bei $3{,}28 \cdot \sqrt{b-a}$ Gruppenoperationen. Speicherplatz ist nötig für die Menge S und die Positionen der beiden Kangaroos. Das ergibt einen Platzbedarf für $|S| = O(\log(b-a))$, wenn die Exponenten s_i Zweierpotenzen sind.

Ein ebenfalls sehr bekanntes und effizientes Verfahren zur Lösung des DLP ist die sogenannte „Baby Step / Giant Step" (BSGS) Methode von D. Shanks. Das Verfahren funktioniert folgendermaßen [BlSS99]: seien P und B wieder Punkte einer elliptischen Kurve E und k der gesuchte Skalar. Es gelte $P = k \cdot B$.

Mit Hilfe der euklid'schen Division ist bekannt, daß k dargestellt werden kann als $k = \lceil \sqrt{n} \rceil \cdot a + b$, mit $0 \leq a, b < \lceil \sqrt{n} \rceil$. Einsetzen in die ursprüngliche Gleichung ergibt: (B

$- b \cdot P) = a \cdot (\lceil\sqrt{n}\,\rceil \cdot P)$. Zuerst werden nun die „Baby Steps“ R_b berechnet: $R_b = B - b \cdot P$, mit $0 < b < \lceil\sqrt{n}\,\rceil - 1$. Die Ergebnistabelle sollte nach R_b geordnet im Speicher abgelegt werden, damit eine schnelle Suche nach dem Vorhandensein eines Ergebnisses möglich ist. Jetzt werden die „Giant Steps“ S_a berechnet: $S_a = a \cdot (\lceil\sqrt{n}\,\rceil \cdot P)$. Nach jeder Berechnung eines „Giant Steps“ wird in der Ergebnistabelle der R_b nachgesehen, ob S_a vorhanden ist. Wenn ja, sind die Werte für a und b gefunden, ansonsten berechnet man den nächsten Schritt des Riesen. Der Algorithmus muß terminieren, bevor a den Wert von $\lceil\sqrt{n}\,\rceil$ erreicht.
Nach [BlSS99] hat der BSGS-Algorithmus eine Komplexität von $O(\sqrt{n})$. Dabei ist der Aufwand des Suchens in der Ergebnistabelle nicht enthalten. Es müssen dabei $O(\sqrt{n})$ Werte im Speicher abgelegt werden. Ein Beispiel zur BSGS-Methode findet man in [BlSS99, S. 92ff.].

5.6 Das gewählte Angriffsszenario

5.6.1 Beschreibung der Strategie

In dieser Arbeit sollen vordergründig Betrachtungen von Low-Security-Systemen durchgeführt werden. Dabei kann momentan von Schlüsselbitbreiten unter 60 Bit ausgegangen werden. Um das beste Verhältnis der maximalen Sicherheit zu einer festgelegten Bitbreite zu erzielen, wird eine elliptische Kurve benötigt, die möglichst wenige Schwachstellen besitzt, wie schon in Kapitel 5.4 angeführt. Das Finden einer solchen Kurve mit wenigen Schwachstellen ist Aufgabe des Trust-Centers und muß hier als gegeben vorausgesetzt werden. Zu den notwendigen Eigenschaften einer solchen elliptischen Kurve E gehört, daß sie eine prime Kurvenordnung $\#E$ besitzt. Wie schon am Ende von Abschnitt 5.4 angeführt, verringert eine nicht-prime Kurvenordnung u.a. die Anzahl der in Betracht kommenden Schlüssel und somit den Suchraum den Angreifers, erheblich.
Liegt eine Kurve mit den hier geforderten Eigenschaften vor, so ist eine hardwaregestützte Brute-Force-Attacke ein geeignetes Angriffsszenario. Die Laufzeit der gesamten Attacke hängt u.a. davon ab, wie schnell man jeden möglichen Schlüssel auf Richtigkeit prüfen kann. Dazu benötigt man aber zuerst nacheinander alle gültigen Schlüssel, also die Punkte der betreffenden elliptischen Kurve. Die Forderung, den nächsten zulässigen Schlüssel schnell zu finden, erfüllt am Besten die Methode der projektiven Punktverdoppelung, da im Unterschied zur Punktaddition kein multiplikatives Inverses berechnet werden muß. Nachteil der projektiven Punktverdoppelung ist, daß der Ergebnispunkt nicht affin ausgegeben wird. Mit der Punktverdoppelung erzeugt man sukzessiv neue Punkte Q_r der Form $k \cdot B$ und vergleicht diese dann jeweils mit dem in der Nachricht mitgehörten Punkt $P = (x_P, y_P)$. Dieser Punkt P liegt aber leider affin, also in der Form $P = (x_{P_1}, y_{P_1}, \mathbf{1})$ vor.

Konvention: Die Indizes in diesem Kapitel 5.6 haben folgende Bedeutung: $y_{U_{vw}}$ reprä-

sentiert die y-Komponente des Punktes U_v, wobei die z-Komponente des Punktes U_v den Wert w hat. Für den Punkt U_v gilt damit: $U_v = (x_{U_{vw}}, y_{U_{vw}}, w)$. Da die einzelnen Komponenten eines Punktes in projektiver Darstellung voneinander in der Form von Gleichung (11) auf Seite 17 abhängig sind, gilt mit dieser Konvention bzgl. der Indizes beispielsweise:
$U_v = (x_{U_{vw}}, y_{U_{vw}}, w) = (x_{U_{vt}}, y_{U_{vt}}, t) = (x_{U_{vs}}, y_{U_{vs}}, s),\ \forall\, w,\, t,\, s \neq 0,\, w,\, t,\, s \in K.$

Mit Hilfe der Gleichungen (12) von Seite 17 kann man einen Punkt von einer projektiver Darstellung in die affine Darstellung transformieren. Diese Gleichungen benötigen neben den Multiplikationen von Z zu Z^2 und Z^3 zusätzlich noch die Berechnung der multiplikativen Inversen von Z^2 und Z^3. Da dieses Vorgehen extrem rechenintensiv ist, wäre die affine Punktaddition in diesem Fall die schnellere Strategie, da zum Errechnen des nächsten Punktes, respektive Schlüssels, zwar eine Inversion vonnöten ist, das Ergebnis danach aber in affiner Form vorliegt. Es kann dann ohne weitere Rechnung mit dem Punkt $P = (x_{P_1}, y_{P_1}, 1)$ auf Richtigkeit verglichen werden und man spart sich so die Berechnung einer Inversion gegenüber dem Vorgehen mit der projektiven Verdoppelung.
Ein anderes, besseres Verfahren ist, die jeweils neuen Punkte $Q_r = k \cdot B$ mittels der projektiven Verdoppelung zu errechnen, und dann den Punkt $P = (x_{P_1}, y_{P_1}, 1)$ von seiner affinen Darstellung in eine mit $Q_r = (x_{Q_{r_\beta}}, y_{Q_{r_\beta}}, \beta)$ vergleichbare projektive Form $(x_{P_\beta}, y_{P_\beta}, \beta)$ zu bringen. Dazu bedient man sich der Gleichung (11) von Seite 17.
Abbildung 14 zeigt anhand eines VENN-Diagramms den Ablauf der Strategie: nachdem dem System alle Systemparameter bekannt gemacht wurden, kreiert man eine Arbeitsvariable für den Punkt Q_r. Nachdem die Zählvariable r auf Null gesetzt wurde, wird Q_r mit dem Wert des Basispunktes B initialisiert ($Q_0 = B$) und es startet die zentrale Iterationsschleife: der Zähler r wird inkrementiert; der Arbeitspunkt Q_r wird mit Hilfe der projektiven Punktverdoppelung verdoppelt und danach wird sequentiell auf Gleichheit der x- bzw. y-Komponenten von P und Q_r geprüft, wobei die x- und y-Komponenten von P auf die z-Komponente von Q_r adaptiert werden müssen.

5.6.2 Beispiel eines Angriffs

Man betrachte wieder die Kurve des Beispiels in Kapitel 3.3.2 auf Seite 12. Der Basispunkt B sei (2, 4, 1) und ein mit einer fremden Nachricht mitgehörter Punkt $P = (x_{P_1}, y_{P_1}, 1)$ sei (7, 1, 1). Diese Werte sind so gewählt, daß die Tabelle 4 auf Seite 34 zur Anschauung dienen kann.
Wendet man den Algorithmus zur projektiven Punktverdoppelung auf den Punkt $B = (2, 4, 1)$ zum ersten Mal ($r = 1$) an, so ergibt sich entsprechend der Tabelle 4 der Punkt $Q_1 = k \cdot B = 2^r \cdot B = 2^1 \cdot B = 2 \cdot B = (7, 2, 2)$. Um herauszufinden, ob $k = 2$ die Lösung ist, muß man bestimmen, ob $P = Q_1$ ist. Das bedeutet in diesem Fall: Ist (7, 1, 1) gleich (7, 2, 2)?

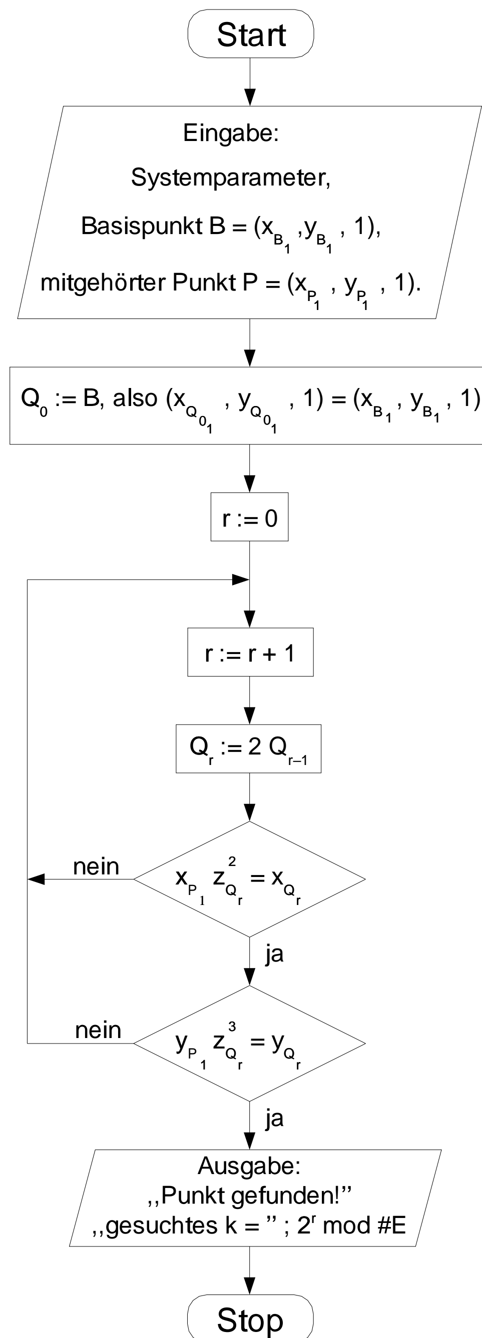

Abbildung 14: Ablaufschema der Verdoppelungs-Strategie

Da die Koordinatenkomponenten x, y und z eines projektiven Punktes nach Gleichung (11) auf Seite 17 voneinander abhängig sind, ergibt sich:

$(7, 1, 1) \Leftrightarrow (7 \cdot \lambda^2, 1 \cdot \lambda^3, 1 \cdot \lambda)$.

Da ein Äquivalent zu (7, 2, **2**) gesucht ist, setzt man $\lambda = 2$ und erhält:

$(7 \cdot 2^2, 1 \cdot 2^3, 1 \cdot 2)$.

Analog zu Kapitel 3.3.2 sind diese Werte als Polynome zu betrachten:

$((x^2 + x + 1) \cdot x^2, 1 \cdot x^3, 1 \cdot x) \Leftrightarrow (x^4 + x^3 + x^2, x^3, x) \bmod (x^3 + x + 1) \Leftrightarrow$
$(x, x + 1, x)$.

Die polynomielle Darstellung wird dann wieder in eine dezimale umgewandelt werden: $(x, x + 1, x) \Rightarrow (2, 3, 2)$. Damit ist dies nicht der gesuchte Punkt P.

Es wurden hier der Übersichtlichkeit halber die x- und y-Komponenten zusammen berechnet. Dies ist, wie schon in Abbildung 14 gezeigt, nicht sinnvoll. Stimmt x_{Q_i} nicht mit x_{P_i} überein, kann sofort der nächste Punkt Q_r berechnet werden. Es existieren für jede elliptische Kurve nur jeweils maximal zwei Punkte mit derselben x_{J_i}-Koordinate. Sollte der Fall eintreten, daß $x_{Q_i} = x_{P_i}$, so muß noch y_{Q_i} auf Gleichheit mit y_{P_i} geprüft werden. Die erste Iteration in diesem Beispiel führte noch nicht zur Lösung. Führt man die zweite Iteration durch, ergibt sich:

$r := r + 1 \Rightarrow r = 2$.

$Q_2 = k \cdot B = 2^r \cdot B = 2^2 \cdot B = 4 \cdot B = (6, 4, 3)$, (vgl. Tab. 4, S. 34)

Nun bringt man $P = (7, 1, 1)$ in die gewünschte Form $(x_{P_3}, y_{P_3}, 3)$:

Es gilt: $(7, 1, 1) \Leftrightarrow (7 \cdot \lambda^2, 1 \cdot \lambda^3, 1 \cdot \lambda)$.

Mit $\lambda = 3$ folgt: $(7, 1, 1) \Leftrightarrow (7 \cdot 3^2, 1 \cdot 3^3, 1 \cdot 3)$.

Für die x_{P_3}-Komponente ergibt sich folgende Rechnung:

$x_{P_3} = 7 \cdot 3^2 \Rightarrow ((x^2 + x + 1) \cdot (x + 1)^2) \Leftrightarrow ((x^2 + x + 1) \cdot (x^2 + 1)) \Leftrightarrow$
$(x^4 + x^3 + x + 1) \bmod (x^3 + x + 1) \Leftrightarrow (x^2 + x) \Rightarrow 6$.

Da $x_{Q_{2_3}} = 6 = x_{P_3}$, ist der Punkt Q_2 mit einer Wahrscheinlichkeit von 50% der Gesuchte. Die Berechnung von y_{P_3} gibt endgültige Klärung:

$y_{P_3} = 1 \cdot 3^3 \Rightarrow (x + 1)^3 \Leftrightarrow ((x^2 + 1) \cdot (x + 1)) \Leftrightarrow$
$(x^3 + x^2 + x + 1) \bmod (x^3 + x + 1) \Leftrightarrow x^2 \Rightarrow 4$.

Daraus folgt, daß $y_{Q_{2_3}} = 4 = y_{P_3}$.

Damit ist gezeigt worden, daß folgendes gilt:

$P = (x_{P_1}, y_{P_1}, 1) = (7, 1, 1) = (x_{P_3}, y_{P_3}, 3) = (6, 4, 3) =$
$(x_{Q_{2_3}}, y_{Q_{2_3}}, 3) = Q_2 = 4 \cdot B$

Damit ist der Angriff erfolgreich durchgeführt worden, mit dem Ergebnis, daß der geheime Schlüssel des Senders $k = 2^r = 4$ ist.

5.6.3 Komplexität und Erweiterung des Verfahrens

In diesem Abschnitt wird die Komplexität des Verfahrens aus stochastischer Sicht analysiert werden. Dazu betrachte man die Menge M aller Punkte $P_i = (x_i, y_i)$ einer elliptischen Kurve E mit $|M| = \#E$. Unter den bisher beschriebenen Umständen hat man eine Nachricht abgehört und besitzt somit genau einen Punkt S, nach dem durch sukzessive Verdoppelung des Basispunktes B gesucht werden soll, um die Anzahl der Verdoppelungen als Lösung angeben zu können. Sei $P(B_{\#E,1})$ die Wahrscheinlichkeit, den gesuchten Punkt S durch die erste Verdoppelung zu finden, so gilt:

$P(B_{\#E,1}) = \frac{1}{\#E}.$

Nach [Hüb96] ergibt sich für das einfache stochastische Modell des „zufälligen Ziehens ohne Zurücklegen“ die Wahrscheinlichkeit $P(B_{\#E,r})$, nach r Verdoppelungen den gesuchten Punkt gefunden zu haben, zu

$$P(B_{\#E,r}) \; = \; \frac{r}{\#E}, \; \text{mit}\, r \leq \#E. \tag{35}$$

Daraus folgt, daß die Komplexität des Verfahrens bei $O(r)$ liegt.

In vielen als Angriffsziel denkbaren Anwendungen besteht die Möglichkeit, durch Abhören der Kommunikation, an mehrere Nachrichtenpakete zu gelangen. Das bedeutet, daß diverse Punkte der elliptischen Kurve ermittelt werden können, nach denen durch Verdoppelung gesucht werden kann. Für das Verfahren folgt daraus, daß sich die Wahrscheinlichkeit $P(A_{\#E,s,r})$, einen Punkt S_i aus einer Menge $S = \{S_1, S_2, \ldots, S_s\}$ von unterschiedlichen Punkten mit $s = |S|$ nach r Verdoppelungen zu finden, erhöht.

Im Folgenden betrachte man eine elliptische Kurve mit der Ordnung $\#E = 6$. Sei $s = 1$, d.h. es ist nur ein einziger Punkt dieser Kurve bekannt, welcher durch Verdoppeln des Basispunktes gesucht wird. Nach obiger Gleichung (35) ergibt sich für die Wahrscheinlichkeit, diesen Punkt nach beispielsweise drei Verdoppelungen gefunden zu haben:

$\frac{r}{\#E} \Rightarrow \frac{3}{6} = \frac{1}{2} = 0{,}5.$

Man habe nun Kenntnis über $s = 2$ Punkte dieser Kurve, die man errechnen möchte. Nach $r = 3$ Verdoppelungen können drei mögliche Lösungen eingetreten sein (die bereits hier angegebenen *u* werden in späteren Formeln relevant):

1. Kein errechneter Punkt ($u = 2$) entspricht den Gesuchten;

2. Ein errechneter Punkt ($u = 1$) ist ein Gesuchter, die beiden anderen nicht;

3. Zwei errechnete Punkte ($u = 0$) entsprechen den Gesuchten, einer nicht.

Die Wahrscheinlichkeiten für die Ereignisse ergeben sich nach folgender Formel:

$$\frac{\begin{pmatrix} s \\ s-u \end{pmatrix}\begin{pmatrix} \#E-s \\ r-(s-u) \end{pmatrix}}{\begin{pmatrix} \#E \\ r \end{pmatrix}} .$$

Die Wahrscheinlichkeit für die o.a. Lösungsform 1 lautet:

$$\frac{\begin{pmatrix} 2 \\ 2-2 \end{pmatrix}\begin{pmatrix} 6-2 \\ 3-(2-2) \end{pmatrix}}{\begin{pmatrix} 6 \\ 3 \end{pmatrix}} = \frac{\begin{pmatrix} 2 \\ 0 \end{pmatrix}\begin{pmatrix} 4 \\ 3 \end{pmatrix}}{\begin{pmatrix} 6 \\ 3 \end{pmatrix}} = ...$$

Die Form dieses Ausdruckes kann man informell folgendermaßen beschreiben: „von zwei gesuchten Punkten sind null gefunden; von den restlichen vier auf der Kurve befindlichen sind drei gefunden; insgesamt sind von sechs Punkten drei errechnet worden“. Die weitere Berechnung ergibt:

$$... = \frac{1 \cdot \frac{4!}{3! \cdot (4-3)!}}{\frac{6!}{3! \cdot (6-3)!}} = \frac{4}{20} = 0,2 .$$

Der Fall 2 hat eine Wahrscheinlichkeit von $\frac{12}{20} = 0{,}6$ und die Berechnung des letzten Falles ergibt $\frac{4}{20} = 0{,}2$. Da diese drei Fälle alle möglichen Lösungen repräsentieren, muß die Summe der Einzelwahrscheinlichkeiten Eins ergeben.

Da das Angriffsverfahren bereits erfolgreich ist, wenn ein einziger gesuchter Punkt gefunden wurde, summiert man alle Einzelwahrscheinlichkeiten, die die Bedingung „mindestens ein gesuchter Punkt wurde gefunden“ erfüllen. Hier erfüllen die Fälle zwei und drei diese Bedingung und daraus folgt 0,6 + 0,2 = 0,8. Die Wahrscheinlichkeit, mindestens einen von insgesamt zwei gesuchten Punkten auf einer elliptischen Kurve mit Ordnung $\#E = 6$ nach drei Verdoppelungen zu finden, ist somit **0,8**.

Zusammengefaßt läßt sich die Berechnung der Wahrscheinlichkeiten folgendermaßen darstellen:

$$P(A_{\#E,s,r}) = \sum_{u=0}^{s-1} \frac{\begin{pmatrix} s \\ s-u \end{pmatrix}\begin{pmatrix} \#E-s \\ r-(s-u) \end{pmatrix}}{\begin{pmatrix} \#E \\ r \end{pmatrix}} . \quad (36)$$

Abbildung 15 verdeutlicht die Vorteile des Suchens mittels mehrerer Punkte: als Basis sei hier eine elliptische Kurve mit Ordnung $\#E = 20.000$ angenommen. Der Graph mit der Bezeichnung $s = 1$ stellt den Verlauf der Wahrscheinlichkeiten dar, wenn nach nur einem Punkt auf dieser Kurve gesucht wird. Analog zu Gleichung 35 verläuft die Wahrscheinlichkeit proportional zur Anzahl der Verdoppelungen. Die Wahrscheinlichkeit, daß

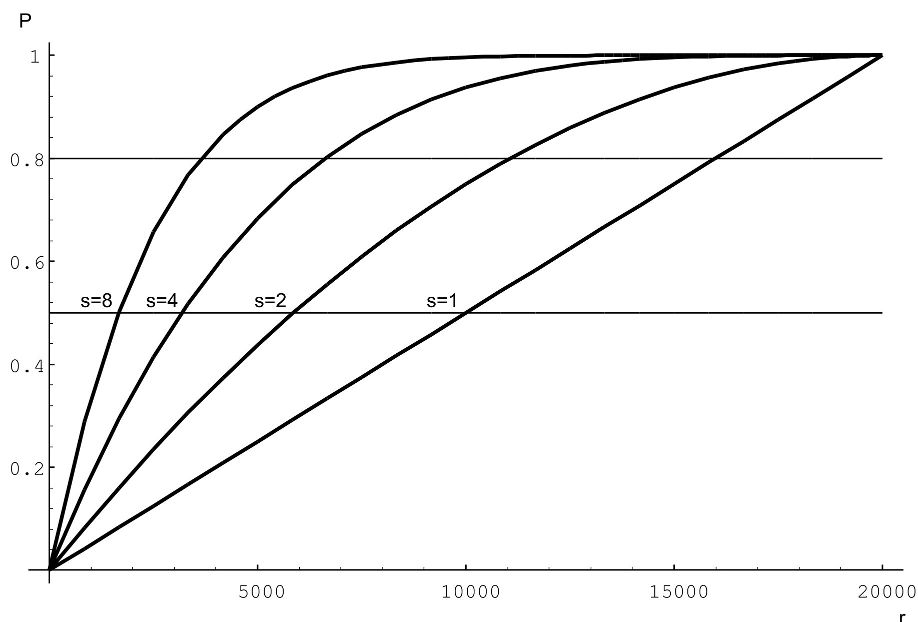

Abbildung 15: Erfolgswahrscheinlichkeiten des Suchens mittels mehrerer Punkte

der gesuchte Punkt zu 50% gefunden wird, tritt entsprechend o.a. Linearität erwartungsgemäß nach 10.000 errechneten Punkten ein. Eine Wahrscheinlichkeit von 80% tritt nach 16.000 Verdoppelungen ein.
Die folgenden Kurven zeigen die Wahrscheinlichkeitsverläufe in Abhängigkeit von der Menge der zur Verfügung stehenden Suchpunkte. Zum exakten Zahlenvergleich ist in folgender Datentabelle angegeben, nach welcher Verdoppelungsanzahl die jeweiligen Graphen die 50%, sowie die 80%-Schwelle erreichen:

	$s = 1$	$s = 2$	$s = 4$	$s = 8$
50%	10.000	5.858	3.182	1.660
80%	16.000	11.055	6.625	3.644

Tabelle 6: Erreichen der 50% und 80% Erfolgswahrscheinlichkeiten

Es wird deutlich, wie sehr das Verfahren beschleunigt werden kann, wenn mehrere Suchpunkte vorhanden sind. Das Suchen mit acht Punkten führt sechsmal schneller zur 50% Schwelle, als wenn nur mit einem einzelnen Punkt gesucht würde.

6 Implementierung

6.1 Das C++ Modell

Die erste Implementierung des in Kapitel 5.6 beschriebenen Angriffsszenarios findet in einem C++-Modell statt. Als Entwicklungsplattform wird eine Linux-Arbeitsstation mit einem Gnu-C++-Compiler der Version 2.95 verwendet. Das komplette C++-Programm ist im Anhang B.1 zu finden.

Zielsetzung des Softwaremodells ist es, den Ablauf des gewählten Angriffsschemas schrittweise zu realisieren, um daraus etwaige Probleme bzgl. der Geschwindigkeit erkennen zu können. Dazu wird der Gesamtablauf nach Top-Down-Methode in funktionale Unterprogramme aufgeteilt, implementiert und dann analysiert. Man kann so Verhaltenssimulationen des Angriffsablaufs durchführen und Optimierungen bzgl. der Kontroll- und Datenstrukturen vornehmen.

Die Repräsentation der Punkte auf der elliptischen Kurve erfolgt in der in Kapitel 3.2 eingeführten Vektordarstellung, d.h. ein projektiver Punkt besteht aus drei statischen Arrays, stellvertretend für die x-, y- und z-Komponenten. Jedes Array hat die Länge `bit`, die als Integer definiert wird. Da das Primpolynom noch ein Bit mehr in seiner Darstellung benötigt, wird dafür eine Integervariable `abit` eingeführt. Da in C++ Arrays statische Datenstrukturen sind, muß die Dimension bereits bei der Kompilation im Code festgelegt sein. Daher ist im Code fest einzutragen, über welchem endlichen Körper $\mathcal{GF}(2^m)$ die elliptische Kurve betrachtet werden soll. Der Bereich der individuellen Parameter, die an die jeweilige Aufgabenstellung angepaßt werden müssen, steht als erster im Abschnitt der Variablendeklarationen. Dazu gehören die Vektordarstellungen des Primpolynoms `ppoly[abit]`, der Koeffiziente b der elliptischen Kurvengleichung `cur-veb[abit]`, der Felddimension m aus $\mathcal{GF}(2^m)$ `fieldm[abit]`, des Wertes $c = b^{2^{m-2}}$ (vgl. Gleichung 34) `c_reg[abit]` und der drei Komponenten des projektiven Startpunktes `eins_x[abit]`, `eins_y[abit]` und `eins_z[abit]`. Zum Schluß muß noch der mitgehörte Punkt, nach dem gesucht werden soll, angegeben werden. Dieser liegt immer in affiner Form vor, so daß von ihm nur die x- und die y-Komponenten als `punkt1_x[abit]` und `punkt1_y[abit]` im Code eingetragen werden müssen.

Die anderen in der Variablendeklaration folgenden Variablen sind Arbeitsvariablen und werden z.T. während der weiteren Ausführungen erklärt. Das C++ Modell hat keinesfalls den Anspruch, eine optimale Lösung hinsichtlich der Laufzeit oder des Ressourcenverbrauchs zu sein. Diese Aspekte werden bei der Entwicklung des FPGA ausführlich behandelt werden.

Die zentralen Bestandteile des Programmes sind die Addition, Multiplikation und Modulo-Reduktion von Polynomen, bzw. Vektoren im endlichen Körper $\mathcal{GF}(2^m)$. Deshalb hat die Effizienz dieser Routinen höchste Priorität.

Die Addition zweier Vektoren `add_a[bit]` und `add_b[bit]` wird im Unterprogramm-

teil `add()` berechnet. Wie schon in Kapitel 3.2 ausführlich dargelegt wurde, ist die Vektoraddition im Körper $\mathcal{GF}(2^m)$ eine einfache XOR-Verknüpfung der Komponenten `add_a[i]` und `add_b[i]`. Man kann davon ausgehen, daß beide Vektoren vor der Verknüpfung bereits in einer durch das Primpolynom `ppoly` reduzierten Form vorliegen. Die einzelnen Komponenten der Vektoren werden mittels einer inkrementierenden `for`-Schleife mit Laufintervall [0, `bit`] XOR-verknüpft und in den Ergebnisvektor `add_c[bit]` geschrieben. Da bei der Vektoraddition im Körper $\mathcal{GF}(2^m)$ kein Überlauf eintreten kann, ergibt sich unter der Voraussetzung, daß die beiden Eingangsvektoren `add_a[bit]` und `add_b[bit]` reduziert sind, kein Ergebnisvektor `add_c[bit]`, der reduziert werden muß. Die Laufzeit einer solchen Vektoraddition beträgt dann `bit` Schritte, da die `for`-Schleife genau einmal komplett durchlaufen werden muß.

Ein Polynom ist, nebenbei bemerkt, genau dann mittels des Primpolynoms zu reduzieren, wenn sein Grad größer oder gleich dem des Primpolynoms ist. Dies kann durch Vergleich der Positionen der führenden Einsen in den beiden Vektoren bestimmt werden.

Die Unterroutine für die Vektormultiplikation ist etwas aufwendiger. G. Orlando und C. Paar zeigen in [OrPa99] eine Architektur für einen Multiplizierer, der für eine Multiplikation zweier Vektoren in $\mathcal{GF}(2^m)$ inklusive Modulo-Reduktion unter Verwendung von m Prozessoren m Schritte benötigt. In einem PC sind natürlich i.A. keine m Prozessoren verfügbar. Da dieser Multiplizierer aber auch in die Realisierung des FPGA integriert werden soll, und in diesem Fall mit Prozessoren nur ein Teil einer ALU gemeint ist, stellt dieser Multiplizierer ein geeignetes Instrument dar.

Man betrachte den endlichen Körper $\mathcal{GF}(2^m)$ mit $P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i$ als Primpolynom. Dann sei eine Multiplikation $W(x) = U(x) \cdot V(x) \bmod P(x)$ definiert, mit $U(x) = \sum_{i=0}^{m-1} u_i x^i$, $V(x) = \sum_{i=0}^{m-1} v_i x^i$ und $W(x) = \sum_{i=0}^{m-1} w_i x^i$.

Eine Multiplikation von $U(x)$ und $V(x)$ im Körper $\mathcal{GF}(2^m)$ kann auf verschiedene Arten durchgeführt werden: entweder man multipliziert $U(x)$ und $V(x)$ und reduziert danach mittels $P(x)$, oder man führt die Reduktion während der Multiplikation durch. Der erste Fall wurde bisher in allen Beispielen mit Punktverdoppelung etc. benutzt. Für die Reduktion während der Multiplikation benutzt man folgende Rekursionsgleichung [OrPa99]:

$$\begin{gathered} W'^{(i)} = xW'^{(i-1)} \bmod P(x) + u_{m-1}V(x) \\ \text{mit}\, i = 1, 2, ..., m\,;\; W'^{(0)} = 0\,;\; W(x) = W'^{(m)} \end{gathered} \tag{37}$$

Die Produkte $xW'^{(i-1)}$ in Gleichung (37) sind jeweils Polynome des Grades m, die noch mittels $P(x)$ modulo reduziert werden müssen. Diese Reduktion führt man nach [OrPa99] folgendermaßen durch:

$$x^m \equiv p_{m-1}x^{m-1} + ... + p_1 x + p_0 \bmod P(x) \tag{38}$$

Demnach sind für eine Multiplikation m^2 Bitoperationen und m-1 Polynomreduktionen notwendig. Der Algorithmus 3 ist die Multiplikationsunterroutine `mult()` aus dem Ge-

samtalgorithmus im Anhang B.1 ab Seite 85. In der Routine sind die oben angeführten Gleichungen (37) und (38) implementiert worden.

```
 1  void mult()
 2  {
 3    load (mult_t1, nullv);          // Setze Vek-
tor = 0
 4    load (mult_c , nullv);          // Setze Vek-
tor = 0
 5    for (int i = bit-1; i >= 0; --i)
 6    {
 7      for (int j = 0; j <= bit-1; ++j)
 8      {
 9       if (j == 0)
10        {
11           mult_t1[j] = 0 ^ (mult_a[i]     & mult_b[j])
                            ^ (mult_c[bit-
1] & ppoly[j] );
12        }
13        else
14        {
15          mult_t1[j] = mult_c[j-
1] ^ (mult_a[i]     & mult_b[j])
                                     ^ (mult_c[bit-
1] & ppoly[j] );
16        }
17      }
18    load (mult_c, mult_t1);
19    }
20  }
```

Algorithmus 3: Vektormultiplikation Modulo 2 in C++

Die Eingangswerte $U(x)$ und $V(x)$ des Multiplikationsalgorithmus sind die Variablen `mult_a[abit]` und `mult_b[abit]`. Bei Aufruf der Routine müssen diese Variablen die zu verknüpfenden Polynome bzw. Vektoren enthalten. Das gilt natürlich auch für alle anderen Basisparameter, wie dem Primpolynom `ppoly[abit]` und der Bitbreite `bit`.
In Zeile drei und vier wird die Arbeitsvariable `mult_t1` und die Ergebnisvariable `mult_c` auf den Nullvektor `nullv` gesetzt. Der Nullvektor `nullv` besteht, wie im Gesamtalgorithmus ersichtlich ist, aus einem eindimensionalen Array der Länge `abit` und wird zu Beginn mit Nullen gefüllt. Die Zeilen 5 bis 19 arbeitet eine äußere Schleife mit der Zählvariablen `i`, die im Intervall [`bit-1`, `0`] läuft, ab. Darin läuft eine innere Schleife mit Zählvariable `j` im Intervall [`0`, `bit-1`]. Die zentrale Verknüpfung ist die Zuweisung in Zeile 15. Da für `j` = 0 der Zugriff auf `mult_c[j-1]` eine Bereichsverletzung darstellt, wird dieser Fall mit Hilfe der Fallunterscheidung in Zeile 9 abgefangen. Es findet dann die Zuweisung in Zeile 11 statt.
Um Vektoren von einem Array in ein anderes umzukopieren, wurde die Unterroutine

`load(int arr_a[], int arr_b[])` konstruiert. Sie speichert, ähnlich der Befehle LDA oder LDY für Register in Maschinensprache, den Wert des Arrays `arr_b[]` in dem Array `arr_a[]`.
Letztlich ist noch eine Routine zum Vektorenvergleich notwendig. Dies führt die Routine `compare()` aus. Sie benutzt als Eingabe die Arrays `comp_a[]` und `comp_b[]` und setzt bei Gleichheit der Arrays das Flag `comp_f` auf den Wert 1.
Für die Berechnung der Gleichungen (30) - (33) der projektiven Punktverdoppelung sind die Unterroutinen `berechne_z()`, `berechne_x()`, `berechne_z2_quadrat()`, `berechne_test_x()`, `berechne_u()`, `berechne_y()` und `berechne_test_y()` zuständig.
Die Routine `verdopple()` koordiniert dann den Gesamtablauf, der aus Abbildung 16 ersichtlich wird. Die `verdopple()`-Routine wird aus der `main()`-Routine gestartet.
Der Algorithmus im Anhang B.1 ab Seite 85 führt eine Beispielrechnung über $\mathcal{GF}(2^{11})$ durch. Das Beispiel rechnet mit einer elliptischen Kurve mit dem Koeffizienten $b = 1$ und Primpolynom $(x^{11} + x^9 + 1)$. Im Deklarationsteil ist der Basispunkt als (6, 1313, 1) angegeben und der gesuchte Punkt ist (445, 1315, 1). Mit dem Algorithmus wird der gesuchte Punkt nach 294 Verdoppelungen gefunden.

6.2 Das VHDL-Modell

Das konkrete Ziel dieses VHDL-Modells liegt in der Erstellung der Performance-Analyse eines FPGA mit der Aufgabenstellung der Durchführung einer Brute-Force-Attacke mittels Punktverdoppelung. Da, wie im vorhergehenden Kapitel angesprochen, die Vektoradditionen und -multiplikationen bei dieser Art des Angriffs die zeitintensivsten Operationen darstellen, erfahren diese auch hier die höchste Geschwindigkeitsoptimierung. Das bedeutet, daß zu Gunsten der Geschwindigkeit die Parameter Leistungsaufnahme und Chipfläche, bzw. FPGA-Zellenanzahl und damit auch die Kosten, eine untergeordnete Rolle spielen.

6.2.1 Der Datenfluß

Zunächst soll die Struktur des VHDL-Modells, das in Anhang B.2.1 ab Seite 89 zu finden ist, erläutert werden.
Das VHDL-Gesamtmodell ist in zwei Elemente aufgeteilt. Es besteht aus einem funktionalen Kern und einer zugehörigen Testumgebung (s. Abbildung 17). Den funktionalen Kern stellt das FPGA dar, das die Berechnungen durchführt. Die äußeren Randbedingungen wie Kurvenparameter, Basispunkt oder gesuchten Punkt bekommt das FPGA in der Simulation von der Testumgebung, in der Realität über seine PINs. Die FPGA-Komponente wird durch den Algorithmus `verdoppler_xx`, die Testumgebung durch Algorithmus `tb_verdoppler_xx` beschrieben, welche in VHDL geschrieben sind.

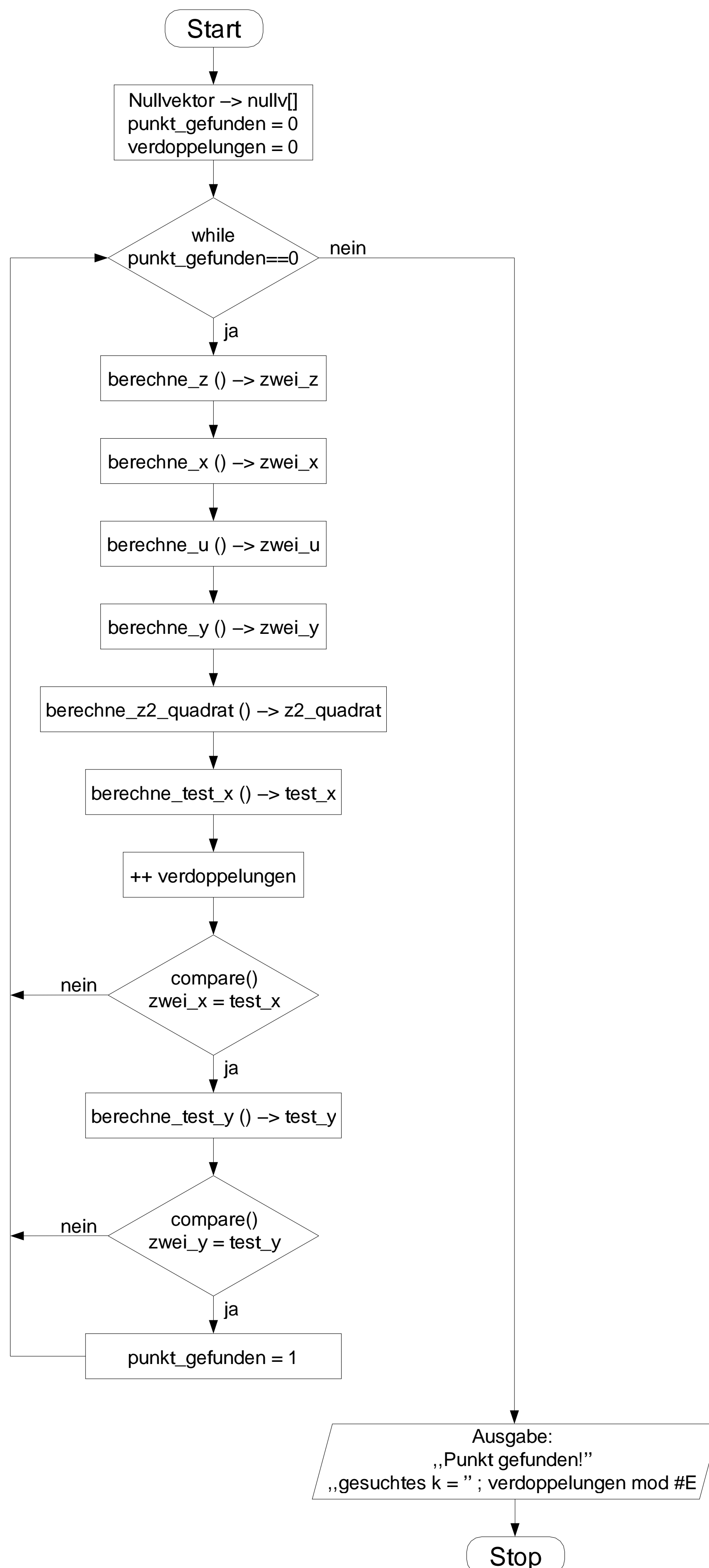

Abbildung 16: verdopple() - Routine des C++-Modells

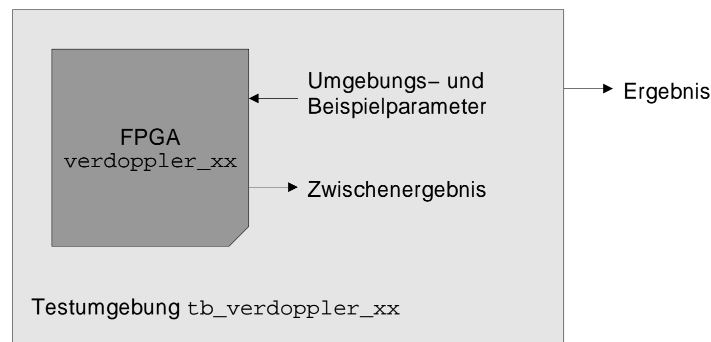

Abbildung 17: Prinzip einer Testumgebung

Dabei ist xx durch die aktuelle Versionsnummer zu substituieren.
Zur Durchführung der projektiven Punktverdoppelung hat das FPGA die folgenden Gleichungen zu berechnen, die schon aus Kapitel 5.4 bekannt sind:
Sei $P = (X_1, Y_1, Z_1)$ der zu verdoppelnde Punkt, dann gilt
$2 \cdot (X_1, Y_1, Z_1) = (X_2, Y_2, Z_2)$, mit

$$Z_2 = X_1 Z_1^2$$

$$X_2 = (X_1 + cZ_1^2)^4$$

$$U = Z_2 + X_1^2 + Y_1 Z_1$$

$$Y_2 = X_1^4 Z_2 + U X_2$$

Die Konstante c in der Gleichung zur Berechnung von X_2 errechnet sich folgendermaßen:

$$c = b^{2^{m-2}}$$

Um die Ressourcen auf dem FPGA optimal zu nutzen, erstellt man für den Ablauf der Berechnung der o.a. Gleichungen einen Datenflußgraphen. Einen Ansatz zeigt [Gor00, S. 54 ff.]. Der Datenflußgraph liefert zunächst Informationen über die Anzahl der gleichzeitig auf dem FPGA benötigten Register. Diese ist allerdings abhängig von der Zahl der Verknüpfungseinheiten, sowie dem Minimum der gewünschten Ausführungsschritte.
Abbildung 18 zeigt den Datenflußgraphen, der sich ergibt, wenn man zwei Multiplizierer und einen Addierer benutzt. Die Anzahl der Verknüpfungseinheiten ist der kritische Faktor, da diese bei möglichst hoher Geschwindigkeit einen enormen Platzbedarf auf dem FPGA haben. Die Verwendung von zwei Multiplizierern und einem Addierer benötigt, wie später noch gezeigt werden wird, extrem viel Fläche, errechnet einen nächsten projektiven Kurvenpunkt allerdings auch schon nach nur sieben Schritten.
Der Graph in Abbildung 18 ist demnach folgendermaßen zu interpretieren: die Knoten stellen, bis auf die erste und letzte Zeile, Verknüpfungsglieder dar. Die Art der Verknüpfung, Multiplikation oder Addition ist als Symbol dargestellt. Die Ziffer links neben dem

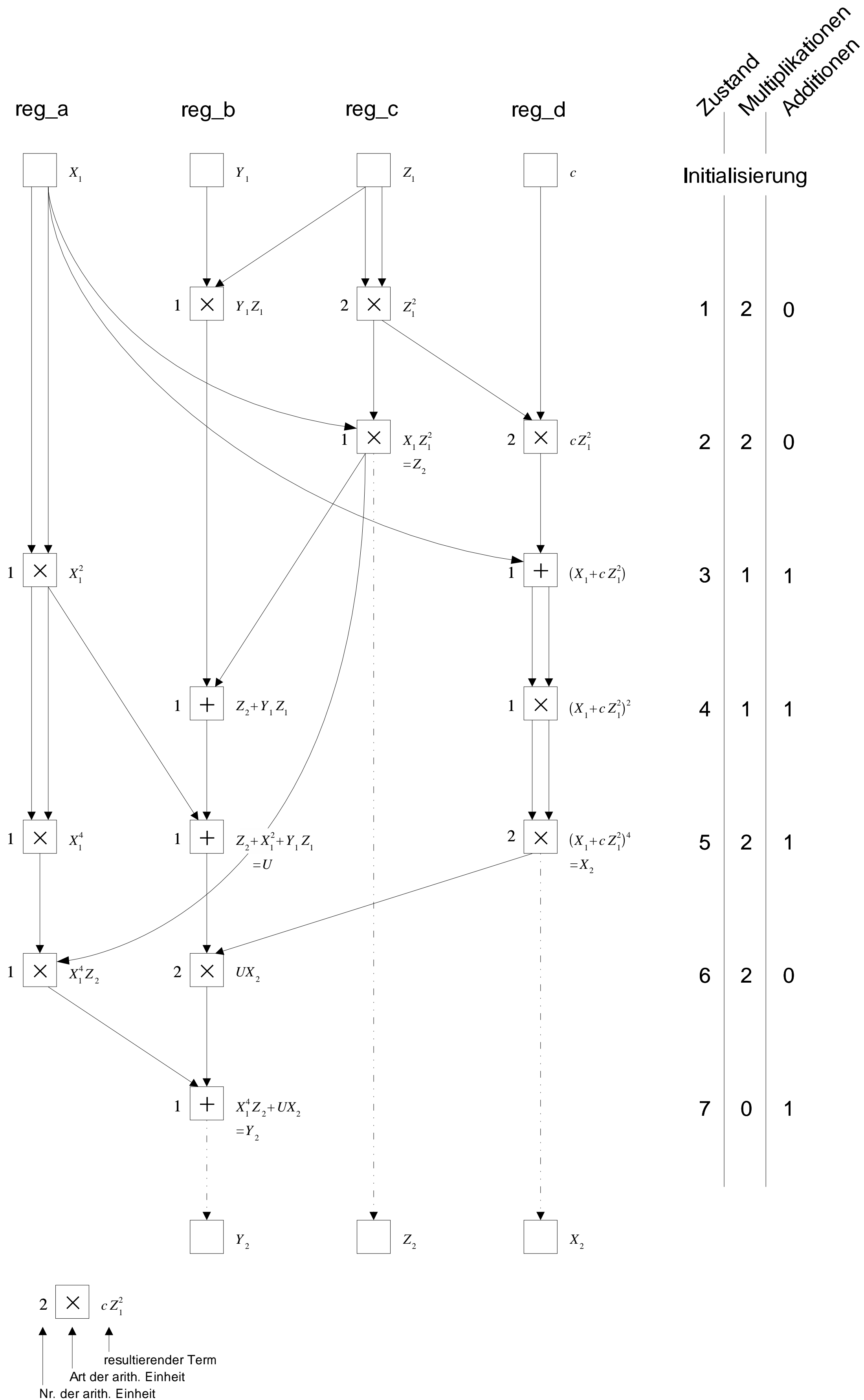

Abbildung 18: Datenflussgraph der projektiven Punktverdoppelung

Knoten bezeichnet die fortlaufende Nummer der Einheiten einer Verknüpfungsart. Diese sind hier schon konform mit den später folgenden Variablendeklarationen des VHDL-Algorithmus. Rechts neben den Knoten steht der aus der jeweiligen Verknüpfung resultierende Term.

Auf der rechten Seite findet man neben der Zustandsnummerierung die Anzahl der in jedem Schritt verwendeten Verknüpfungseinheiten. Man erkennt, daß der Addierer in drei von sieben Zuständen ungenutzt bleibt, und in der übrigen Zeit keine Einheiten auf noch unerledigte Additionsverknüpfungen warten müssen. Der Einsatz eines einzelnen Addierers ist somit ausreichend. Hingegen ist die Auslastung der Multiplizierer deutlich höher. Man kann aber leicht erkennen, daß sich auch durch den Einsatz eines dritten Multiplizierers kein Zustand einsparen ließe. Somit ist die Wahl von zwei Multiplizierern sinnvoll.

Der Datenflußgraph macht zudem deutlich, daß eine fortlaufende projektive Punktverdoppelung auf iterative Art und Weise leicht realisiert werden kann. Bezogen auf Abbildung 18 müßte nur in Zustand Sieben das Register `reg_d` nach `reg_a` kopiert, und der Wert der Konstante c in Register `reg_d` kopiert werden. Dann könnte das Verfahren erneut starten und die zweite Punktverdoppelung vornehmen.

Die fortlaufende Punktverdoppelung ist das gewünschte Resultat des FPGA. Es muß mit einem Startpunkt, i.A. dem Basispunkt, beginnen und nach jeder Verdoppelung auf Erreichen des gesuchten Punktes prüfen. Damit sind Funktion und Datenfluß des VHDL-Modells bereits skizziert: wie in Abbildung 19 erkennbar, benötigt man als Datenspeicher die vier Register, sowie als Funktionseinheiten die drei Verknüpfungsglieder. Der Datenverkehr zwischen Speicher, Funktionseinheiten und umgekehrt wird über zwei Multiplexer (MUX) abgewickelt, die in Abhängigkeit vom jeweiligen Zustand agieren. Die Zustandsänderungen werden von einem zentralen, taktgesteuerten endlichen Automaten (Finite State Machine, FSM) in Mealy-Form durchgeführt.

Während das Anlegen des Startpunktes im Initialisierungsschritt keinen zusätzlichen Zeit- oder Berechnungsaufwand bedeutet, ist die Prüfung auf Erreichen des Lösungpunktes aufwendiger. Analog zu den in Abbildung 14 auf Seite 41 dargestellten bedingungsabhängigen Operationen wird diese Prüfung durchgeführt. Der gesuchte affine Punkt wird dabei in eine äquivalente Darstellung umgerechnet, die mit dem durch die Punktverdoppelung errechneten Punkt vergleichbar ist: das während der Punktverdoppelung in Zustand Zwei errechnete Z_2 wird quadriert und mit der x-Komponente des gesuchten Punktes multipliziert. Danach wird Z_2^3 errechnet und mit der y-Komponente des gesuchten Punktes multipliziert. Nun kann man die Punkte auf Gleichheit prüfen.

Wie aus Abbildung 18 deutlich wird, kann die Punktverdoppelung in sieben Schritten durchgeführt werden. Für die Durchführung einer Attacke wäre es nachteilig, wenn die Prüfungsoperationen darüber hinaus zusätzliche Zeit erfordern.

Abbildung 20 zeigt, wie die Prüfoperationen optimal in den bestehenden Verdoppelungs-

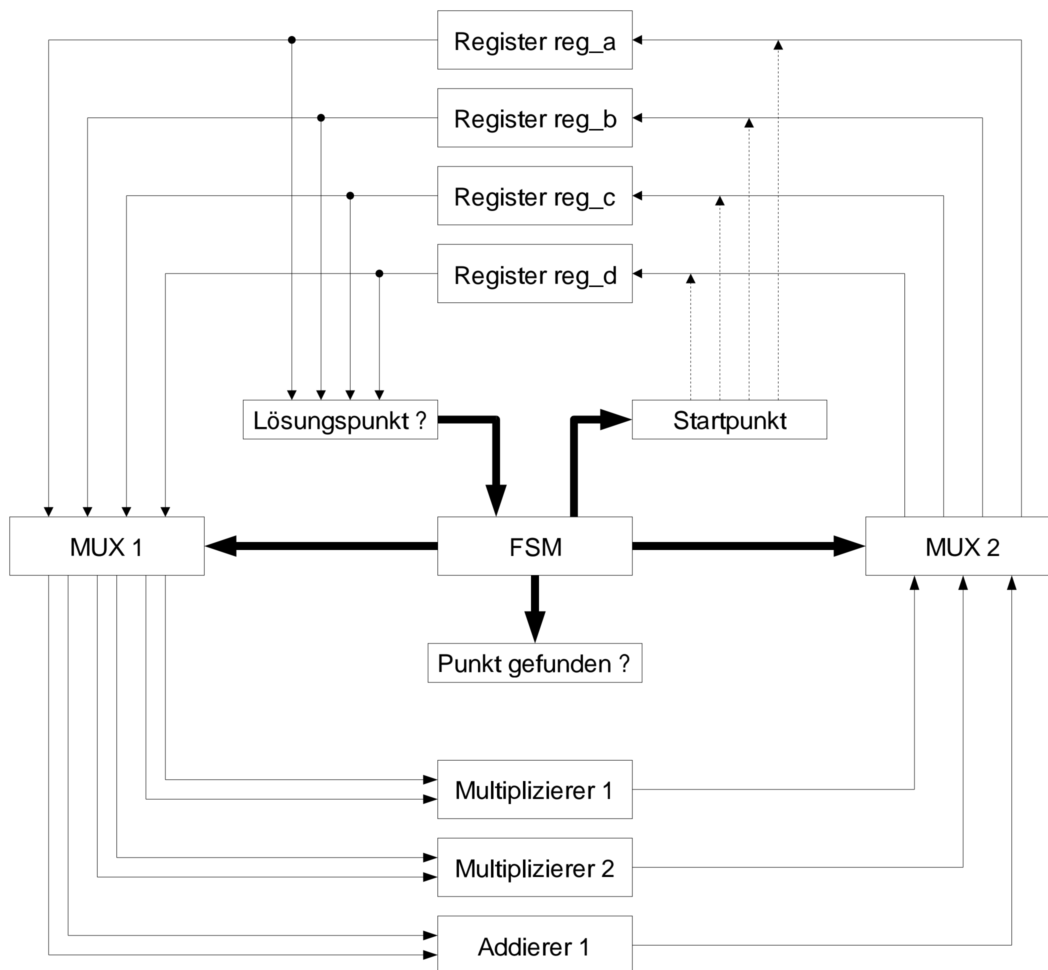

Abbildung 19: Ablaufschema des VHDL-Modells

ablauf integriert werden können, ohne die Gesamtlaufzeit zu erhöhen. Es werden dazu zwei weitere Register `z2_hoch_x` und `gespkt_proj` benötigt. Da im dritten Zustand der Punktverdoppelung der Multiplizierer 2 unbenutzt ist, kann zu diesem Zeitpunkt die Berechnung von Z_2^2 stattfinden. Im darauffolgenden Schritt wird dann `gespkt_proj` = $Z_2^2 \cdot$ `gespkt_1_x_aff` berechnet, da der Multiplizierer 2 wieder unbenutzt ist. Damit sind die x-Komponenten der beiden Punkte vergleichbar.

In Zustand Sieben werden die x-Komponenten auf Gleichheit geprüft, was in Abbildung 20 durch zwei graue Quadrate gekennzeichnet ist. Der Vergleich erfolgt erst in Zustand Sieben, da er zu einem früheren Zeitpunkt keinen Zeitgewinn für den Gesamtablauf des Verfahrens bedeuten würde. Der Ablauf der projektiven Punktverdoppelung benötigt in jedem Fall sieben Schritte, da erst dann das Ergebnis von Y_2 vorliegt.

Stellt sich durch den Vergleich der x-Komponenten in Zustand Sieben heraus, daß diese nicht identisch sind, so ist der aktuell errechnete Punkt nicht der Gesuchte und das Verfahren kann durch Übergang in den Zustand Eins mit der nächsten Verdoppelung fortfahren. Sind die beiden x-Komponenten allerdings identisch, so ist der errechnete Punkt mit einer Wahrscheinlichkeit von mindestens 50 % der Gesuchte, da maximal zwei Punkte auf einer elliptischen Kurve existieren, die den selben x-Wert haben. Ein eindeutiges Ergebnis erhält man nur durch den zusätzlichen Vergleich der y-Komponenten. Da in Zustand Sieben beide Multiplizierer unbenutzt sind, kann an dieser Stelle Multiplizierer 1 die Berechnung von $Z_2^3 = Z_2^2 \cdot Z_2$ durchführen. In Zustand Acht berechnet Multiplizierer 1 dann `gespkt_proj` $= Z_2^3 \cdot$ `gespkt_1_y_aff`. In Zustand Neun folgt dann der Vergleich der y-Komponenten der beiden Punkte, was in Abbildung 20 wieder durch zwei graue Quadrate dargestellt ist. Sind die y-Komponenten identisch, so ist der errechnete Punkt der Gesuchte und das Verfahren mit positivem Ergebnis beendet. Sind die Werte nicht identisch, so ist der errechnete Punkt, wie bereits erwähnt, der einzige auf dieser elliptischen Kurve mit derselben x-Komponente wie der Gesuchte. Das Verfahren fährt dann mit der nächsten Verdoppelung durch Übergang in Zustand Eins fort.

Wichtig ist an dieser Stelle, daß das Verfahren, bis auf mindestens ein und maximal zwei Vorkommen pro elliptischer Kurve, nur sieben Zustände zur Berechnung durch projektive Punktverdoppelung und anschließender Prüfung eines Punktes benötigt. In den mindestens einem und maximal zwei anderen Fällen benötigt man neun Zustände.

Abbildung 20 zeigt die optimale Auslastung der arithmetischen Einheiten in den Zuständen Eins bis Sieben, die immer durchlaufen werden müssen. Beide Multiplizierer werden, bis auf Schritt Sieben, in jedem Zustand benutzt. Die Auslastung des Addierers beträgt im Bereich der Zustände Eins bis Sieben zwar nur $\frac{3}{7}$, mehr Additionen sind aber pro Verdoppelung nicht notwendig. Wie später noch ausführlich dargelegt werden wird, hat diese Ablaufstrategie den erheblichen Vorteil, daß der Rechen- und damit auch der Zeitaufwand in jedem Zustand gleich groß ist. Da der Zeitaufwand einer Multiplikation größer ist, als der einer Addition, läßt sich hier feststellen, daß jede Punktverdoppelung i.A. sieben mal

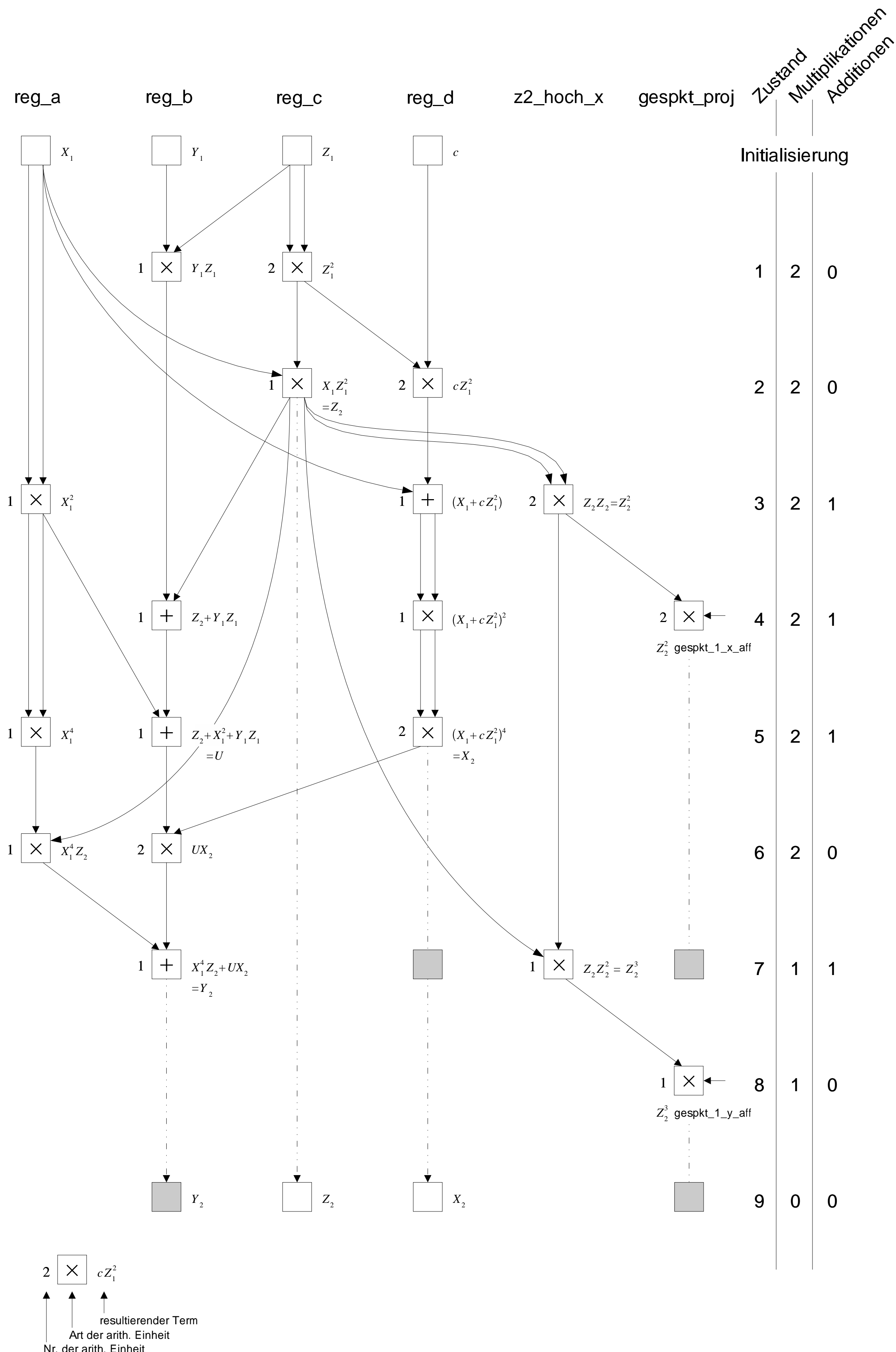

Abbildung 20: Datenflussgraph des FPGA

die Dauer einer Multiplikation benötigt.

6.2.2 Der VHDL-Code

Nachdem die strategischen Vorüberlegungen bzgl. Einsatz und Verteilung von Funktionseinheiten dargelegt wurden, soll im Folgenden der VHDL-Code näher betrachtet werden. Dabei steht zunächst der funktionale Kern, also das FPGA, im Vordergrund. Die Funktion der Testumgebung, die in der Realität die Platine darstellt, auf die das FPGA eingesteckt ist, wird im Anschluß daran ebenfalls eingehend beschrieben und diskutiert werden.
Der VHDL-Algorithmus des funktionalen Kerns, der im Anhang B.2.1 zu finden ist, gliedert sich in drei VHDL-typische Abschnitte:

`library`: die Bibliotheken, deren Funktionalität im Code verwendet werden. Diese müssen zum Zeitpunkt der Kompilierung eingebunden werden. Hier sind dies `IEEE.std_logic_1164.all`, sowie `IEEE.numeric_std.all`.

`entity`: beschreibt die Sichtweise der Einheit von außen. Hier wird der Name der Einheit und die Ein- und Ausgänge definiert. Diese Angaben werden benötigt, wenn man diese Einheit als Teil einer anderen Einheit benutzt. Das ist hier der Fall, da der funktionale Kern von der Testumgebung verwendet wird.

`architecture`: stellt die interne Realisierungs- bzw. Verhaltensbeschreibung der `entity` dar.

Im ersten Teil des `architecture`-Bereiches befinden sich die Definitionen der internen Signale, der zwölf Zustände des endlichen Automaten und die Beschreibung der Funktionen `Add` und `Mult`. Alle Vektoren werden abhängig von der `generic`-Größe `len` definiert, was eine schnellere Anpassung des Codes bzgl. der Bitbreite erlaubt. Die Variable `len` muß im Code in Abhängigkeit von der Feldgröße 2^m angegeben werden: `len` $= m + 1$.
Die Funktion `Add` beschreibt einen Vektoraddierer, der zwei Vektoren der Länge `(len-1 downto 0)` modulo zwei addiert, indem er eine stellenweise XOR-Verknüpfung vornimmt.
Danach wird der Multiplizierer `Mult` in Form einer Funktion beschrieben. Algorithmus 4 ist ein Auszug des VHDL-Programms in Anhang B.2.1. Die Funktion ist identisch mit der des C++-Modells, welche in Kapitel 6.1 bereits eingehend dargestellt wurde. In der VHDL-Beschreibung kann man allerdings das Problem mit dem Zugriff auf ein negatives Array-Element im Gegensatz zu dem C++ Algorithmus 3 von Seite 48 einfacher lösen: man rechnet mit allen Vektoren in üblicher Weise und exkludiert bei der Übergabe des Ergebnisvektors das LSB; in Algorithmus 4 wird in Zeile 3 der Vektor `c` mit Länge `(len downto 0)` initialisiert, der Ergebnisvektor in Zeile 14 wird aber nur mit der Länge `(len downto 1)` übergeben.

```
 1  function Mult (a, b, pp: unsigned (len-1 downto 0))
 2    return unsigned is
 3    variable c : unsigned (len   downto 0);
 4    variable t1: unsigned (len-1 downto 0);
 5  begin
 6    c  := (others => '0');
 7    t1 := (others => '0');
 8    for i in len-2 downto 0 loop
 9      for j in 0 to len-2 loop
10        t1(j) := c(j) xor (a(i) and b(j)) xor (c(len-
1) and pp(j));
11      end loop;
12      c(len downto 1) := t1;
13    end loop;
14    return c(len downto 1);
15  end function Mult;
```

Algorithmus 4: Vektormultiplikation Modulo 2 in VHDL

Die beiden identischen Multiplizierer benötigen auf dem FPGA die meisten Logikzellen (Logic Elements, LE). Nach [OrPa99] ergibt sich für jeweils einen dieser Multiplizierer das in Abbildung 21 gezeigte Schaltbild. Für jede Stelle des Ergebnisvektors wird eine

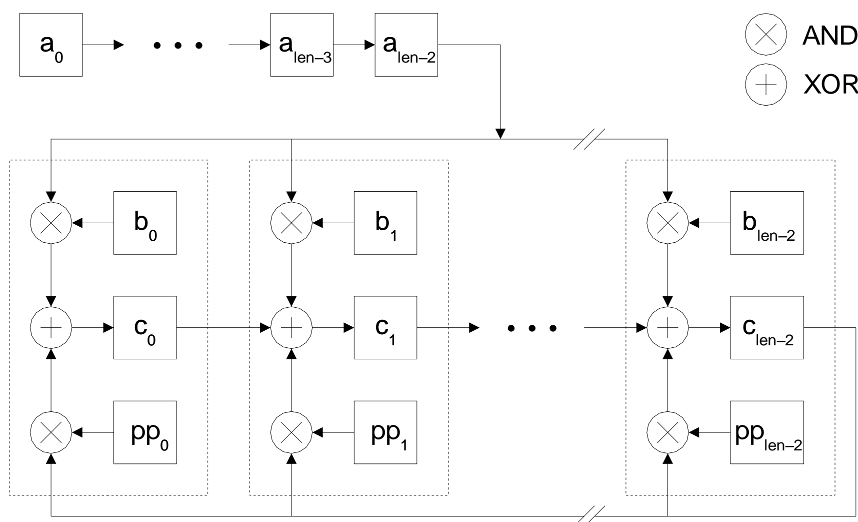

Abbildung 21: Schaltbild des Modulo-2 Multiplizierers

Rechenzelle benötigt, die je aus zwei AND (⊗) und einem XOR (⊕) besteht. Diese Rechenzellen sind in Abbildung 21 mit einer gestrichelten Linie umrahmt. Zeile 10 des Algorithmus 4 beschreibt eine solche Rechenzelle. Der Vorteil dieser VHDL-Beschreibung liegt darin, daß die innere Schleife der Zeilen 9 bis 11 parallel durchgeführt werden kann. Zahlenmäßig werden somit für jedes der in Abbildung 21 dargestellten a_i mit $0 \leq i \leq$ `(len-2)` die Anzahl von `(len-1)` Rechenzellen auf dem FPGA realisiert werden. Je-

der der beiden Multiplizierer benötigt somit `(len-1)`^{2} Rechenzellen. Dadurch erhält man eine hohe Geschwindigkeit auf Kosten des Platzbedarfs. Eine weiterführende Analyse des Platzbedarfs der Multiplizierer in Abhängigkeit von der jeweiligen Bitbreite wird im Ergebnisteil dieser Arbeit angeführt werden.

Nach den einleitenden Deklarationen der Signale und Funktionen folgt nun die eigentliche Funktionsbeschreibung dieser `entity`. Dabei wird im ersten Teil der endliche Automat beschrieben. Er ist vorderflankengesteuert ausgelegt und verfügt über insgesamt zwölf Zustände, die mit `state` bezeichnet sind. Zu jeder Zustandsbeschreibung gehört eine Zuweisung der Ausgänge der arithmetischen Einheiten zu den Eingängen der Register. Das bedeutet, daß in der Beschreibung der endlichen Maschine die des Multiplexers 2 aus Abbildung 19 enthalten ist. Dabei sind die Ein- und Ausgänge der arithmetischen Einheiten folgendermaßen benannt:

`Multein_1_1` und `Multein_1_2` bezeichnen die Eingänge des ersten Multiplizierers; `Multaus_1` bezeichnet den Ausgang des ersten Multiplizierers. Dies gilt analog für den zweiten Multiplizierer. Des weiteren seien `Addein_1_1` und `Addein_1_2` die zwei Eingänge des ersten und einzigen Addierers. Der Ausgang des ersten Addierers ist mit `Addaus_1` bezeichnet.

Der erste Zustand ist der sog. `start_state`. In diesem werden diverse Variablen initialisiert, u.a. die Anzahl der während des nachfolgenden `input_state` zu erwartenden Bits.

Während des `input_state` wird dafür gesorgt, daß die für die Berechnungen notwendigen Startwerte in die Register des FPGA kopiert werden. Dies geschieht in `len` Takten über die sechs Pins `input(0)` bis `input(5)`.

Die folgenden neun Zustände `eins_state`, `zwei_state`, ... , `neun_state` beschreiben den Datenfluß zwischen den arithmetischen Einheiten und den Registern analog zu Abbildung 20. Dabei verhält sich die Maschine bzgl. der errechneten Teilergebnisse so wie im Vorfeld beschrieben: in Zustand Sieben wird bei Gleichheit zweier Werte der Zustand Acht, bei Ungleichheit Zustand Eins eingenommen. Ist analog dazu die in Zustand Neun abgefragte Kondition positiv, so wird das Signal `pkt_gefunden` auf high gesetzt und der Zustand `fertig_state` eingenommen, der dafür sorgt, daß das System ab sofort unveränderlich bleibt. Ist die Kondition negativ, so wird mit Zustand Eins fortgefahren.

Nach der Beschreibung der endlichen Maschine findet durch drei Codezeilen die Instanziierung der arithmetischen Einheiten mit den o.g. Anschlußbezeichnungen statt. Diese drei Zeilen erzeugen den Hauptanteil der LE´s.

Im letzen Abschnitt der `architecture` wird der Multiplexer 1 aus Abbildung 19 beschrieben. Dabei werden die Eingänge der arithmetischen Einheiten, in Abhängigkeit vom jeweiligen Zustand der endlichen Maschine, einem Registerausgang zugewiesen. Die Kommentierung des Codes ist hier der Übersichtlichkeit halber jeweils nur für einen

Eingang einer jeden Einheit angegeben, obwohl sie natürlich für beide Eingänge einer jeden Einheit gültig ist.
Nach der VHDL-Beschreibung des funktionalen Kerns soll im Folgenden das Verhalten durch eine Simulation mit konkreten Werten analysiert werden. Da, wie gezeigt, der funktionale Kern ausschließlich die Berechnungen vornimmt, werden die Startparameter dieser Berechnungen von außen durch die sog. Testumgebung zugeführt. Man kann sich die Testumgebung auch als eine Platine vorstellen, auf die das FPGA eingesteckt ist.
Der VHDL-Algorithmus der Testumgebung, der in Anhang B.2.2 zu finden ist, enthält neben den aus der Beschreibung des funktionalen Kerns bekannten drei Teilen `library`, `entity` und `architecture` noch den in diesem Fall leeren Teil `configuration`. Der Inhalt dieses `library`-Abschnittes ist mit dem des FPGA identisch. Die Beschreibung der `entity` enthält, da von außen kein Signal sichtbar sein muß, nichts.
Die Funktionsbeschreibung der `architecture` der Testumgebung enthält zuerst den Eintrag des `component verdoppler_xx`, welches den FPGA mit dessen Anschlüssen darstellt. Dann folgen die internen Signale der Testumgebung. Anschließend wird im Hauptteil die `port map` zwischen der Testumgebung und dem FPGA festgelegt. In den folgenden zwei Prozessen wird zuerst ein 16 MHz Taktsignal `clk` erzeugt. Danach wird ein Active-Low-Reset-Signal mittels der Variablen `nrst` generiert, welches von initialem Low nach 207 ns auf High verändert wird.
Der nachfolgende dritte Prozeß übergibt gleichzeitig die nötigen Berechnungsparameter `start_x`, `start_y`, `ppoly`, `c_wert`, `ges_1_x` und `ges_1_y` in `len` Schritten an die FPGA-Komponente. Dazu werden die sechs Pins `input(5)` bis `input(0)` benutzt. Die Werte werden in der o.a. Reihenfolge übergeben, d.h. `start_x` über Pin `input(5)`, `start_y` über Pin `input(4)`, usw. Das MSB wird jeweils zuerst übertragen. Das FPGA nimmt diese Werte entgegen und füllt seine Register mit ihnen entsprechend. Die Variable `start_x` bezeichnet beispielsweise die x-Komponente des Startwertes der Punktverdoppelung, also des Basispunktes. Diese wird von der Testumgebung via Pin `input(5)` an das FPGA übergeben. Das legt den Wert, wie aus der Beschreibung des `input_state` ersichtlich wird, in seinem Register `reg_a` ab. Dies ist durch das Datenflußschema (s. Abb. 20) festgelegt.

6.2.3 Simulation eines Beispiels

Für die Simulation wurden die VHDL-Beschreibungen `verdoppler_26_3Bit.vhd` und `tb_verdoppler_26_3Bit.vhd` mit dem SYNOPSYS VHDL-Compiler auf einer SUN-Workstation unter SOLARIS kompiliert. Die durch Simulation mit dem SYNOPSYS VHDL-Debugger erzeugten Signalverläufe sind in Abbildung 22 dargestellt. Die o.g. und im Anhang befindlichen VHDL-Beschreibungen rechnen wieder mit der Kurve des in Kapitel 3.3.2 auf Seite 12 angegebenen Beispiels. Wichtig sind die Angabe der Bitbreite $m = 3$ des der Kurve zugrundeliegenden Körpers $\mathcal{GF}(2^m)$, des verwendeten

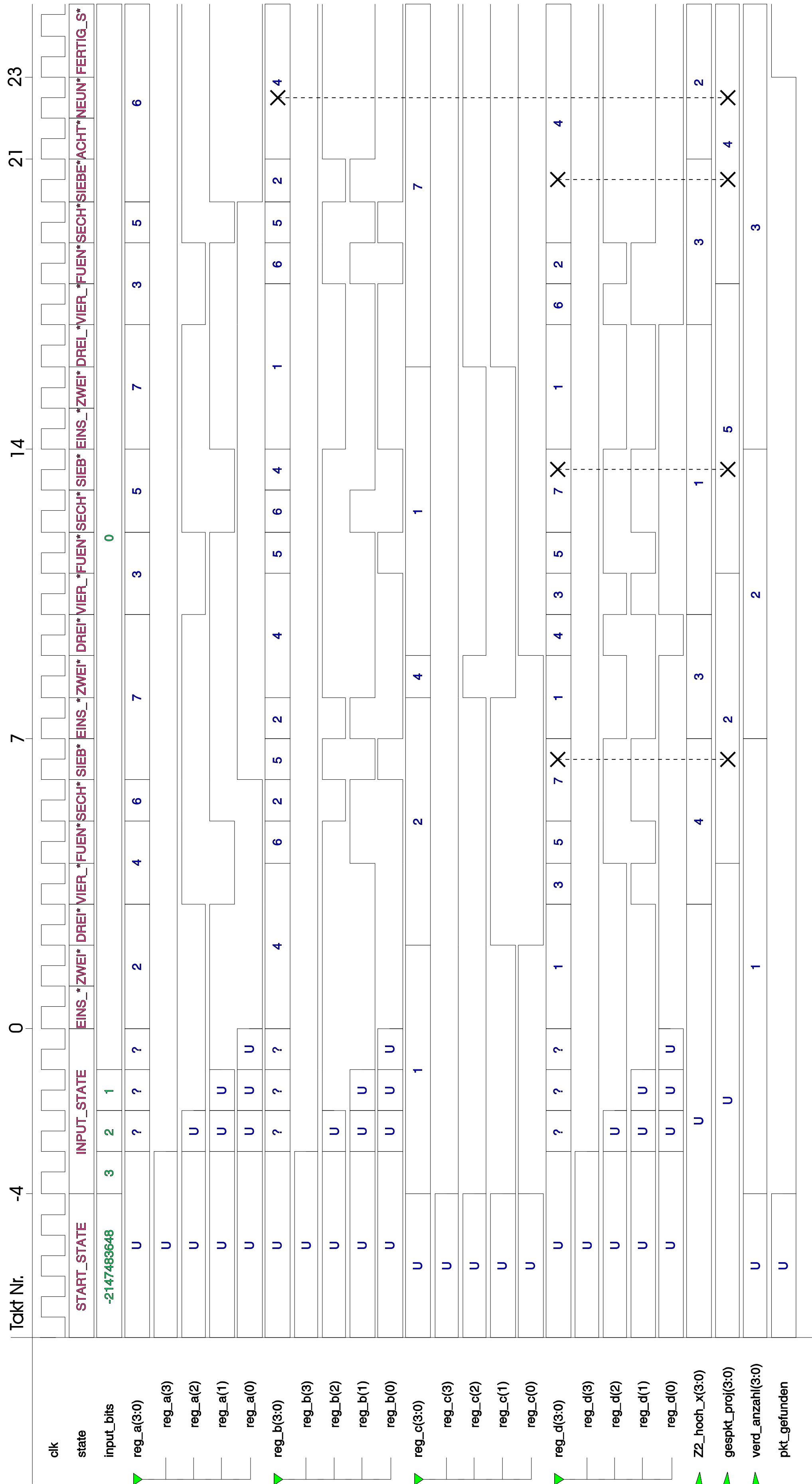

Abbildung 22: Signalverlauf einer Simulation mit 3 Bit

Primpolynoms $(x^3 + x + 1) \Rightarrow (1011)$, sowie der Konstante $c = 1 \Rightarrow (0001)$. Ziel der Beispielberechnung ist, ausgehend von einem Basispunkt (2, 4) den Punkt (5, 2) durch Angriff mittels Punktverdoppelung zu finden, und die Anzahl der dazu benötigten Verdoppelungen anzugeben.

In Abbildung 22 sind die Verläufe der wichtigsten Signale für die vollständige Berechnung dargestellt. Die Signale der vier Register `reg_a` bis `reg_d` sind nicht nur dezimal als `reg_a(3:0)` bis `reg_d(3:0)` zusammengefaßt dargestellt, sondern auch in ihre einzelnen Bits aufgeteilt. Dies soll die Werteübergabe von der Testumgebung im `input_state` illustrieren.

Man erkennt im Zustand `start_state` zu Beginn der Simulation, daß alle Werte bis auf `input_bits` den Status Unsigned (`U`) tragen. Der Wert, der für `input_bits` angegeben ist, ist ein durch den Einschaltvorgang zufällig eingenommener. Zu einem Zeitpunkt zwischen Takt Nr. -5 und -4, nach 207 ns wie aus der Testumgebung ersichtlich, wird das Reset-Signal `nrst` auf High gesetzt. Damit geht die FSM in Takt -4 in den `input_state` über und das Signal `input_bits` beschreibt die zu erwartende Bitlänge der Startparameter. Von Beginn des ersten Taktes des `input_state` an, werden die Startwerte in das FPGA eingelesen. Diese Werte sind im dritten Prozeß der `architecture` der VHDL-Beschreibung der Testumgebung `tb_verdoppler_26_3Bit.vhd` in Anhang B.2.2 angegeben. Da diese Beispielkurve über eine Bitbreite von $m = 3$ Bit verfügt, beansprucht der `input_state` Zustand zur Werteübertragung die Dauer von `len` $= (m + 1) = 4$ Takten. Das Kriterium zum Wechseln des Zustandes ist das Erreichen des Wertes Null des Signals `input_bits`.

In Abbildung 22 wird dieser Zeitpunkt als Takt Nr. 0 bezeichnet, da ab diesem Zeitpunkt der eigentliche Berechnungsvorgang beginnt. Analog zum Datenflußgraph in Abbildung 20 werden nun in jedem Zustand Gleichungen der projektiven Punktverdoppelung berechnet. Jeweils im `sieben_state` wird auf Gleichheit der Werte in den Registern `reg_d` und `gepkt_proj` geprüft. Diese Vergleiche sind in der Abbildung 22 mit zwei durch eine gestrichelte Linie verbundenen Kreuzen gekennzeichnet. Man kann leicht erkennen, daß die Werte in den Takten 7 und 14 nicht identisch sind und das FPGA dann sofort durch Übergang in den `eins_state` mit der nächsten Verdoppelung beginnt. In Takt 21 sind die Werte jedoch gleich, und die Simulation fährt mit Zustand `acht_state` fort. Im Register `z2_hoch_x` erkennt man, wie in Takt 22 der Wert von Z_2^3 ermittelt wird. Im darauffolgenden `neun_state` wird `gespkt_proj` neu ermittelt, was hier aber zufällig wieder den Wert 4 ergibt. Dieser wird, wie durch die zwei Kreuze dargestellt, mit `reg_b` verglichen. Da diese Werte auch identisch sind, ist der gesuchte Punkt gefunden worden. Das Flag `pkt_gefunden` wird auf High gesetzt und das Ergebnis liegt an den Pins `verd_anzahl(0)` bis `verd_anzahl(3)` an. In diesem Fall ist die Lösung 3.

6.2.4 Suchen unter Berücksichtigung mehrerer mitgehörter Punkte

Wie bereits in Kapitel 5.6.3 auf Seite 43 ausführlich dargelegt wurde, stellt das Suchen mit mehreren Punkten eine Möglichkeit der Geschwindigkeitssteigerung dar. Je mehr Suchpunkte zur Verfügung stehen, um so schneller erreicht man höhere Wahrscheinlichkeiten, einen dieser Punkte auf der Kurve zu finden. Im Rahmen dieser Arbeit ist kein Code zur Realisierung dieser Strategie entwickelt worden. Es soll im Folgenden aber gezeigt werden, daß es dazu verschiedene Möglichkeiten mit verschieden großem Realisierungsaufwand gibt.

Eine Realisierung dieser Strategie sollte in der Lage sein, nach erfolgter Punktverdoppelung den neu errechneten Punkt mit einer maximalen Menge an Suchpunkten möglichst ohne Zeitverlust zu vergleichen. Damit wäre sichergestellt, daß weiterhin nach jedem siebten Takt ein neuer Punkt durch Verdoppelung errechnet wird. Verzögert der Mehrfachvergleich die Punktverdoppelung, so reduziert das den angestrebten Geschwindigkeitsgewinn.

Wie anhand von Abbildung 20 auf Seite 56 gezeigt wurde, kann der Punktvergleich genau dann beginnen, wenn im dritten Schritt der Verdoppelung das Register `z2_hoch_x` den Wert Z_2^2 angenommen hat. Die x-Komponente des, bzw. der Suchpunkte muß dann zum Punktvergleich mit Z_2^2 multipliziert werden. In Abbildung 20 geschieht dies im vierten Schritt. Danach kann ab dem siebten Schritt durch Registervergleich eine Entscheidung getroffen werden, ob der errechnete Punkt einer der Gesuchten ist. Man benötigt also für jeden zusätzlichen Suchpunkt ein Register, sowie eine Multiplikation. Da in der vorliegenden VHDL-Beschreibung zwei Multiplizierer vorgesehen sind, kann man in jedem Schritt gleichzeitig die x-Komponenten von zwei Suchpunkten mit Z_2^2 Multiplizieren. Im darauf folgenden Schritt werden dann die resultierenden x-Komponenten mit dem X_2-Wert in Register `reg_d` verglichen. Abbildung 23 zeigt den Ablauf der Strategie. In Zustand n für $n \geq 8$ werden mit diesem Verfahren $s = 2 \cdot (n - 7)$ Suchpunkte auf Gleichheit mit der x-Komponente des durch die Verdoppelung errechneten Punktes verglichen. Die Laufzeit des Verfahrens mit nur einem Suchpunkt S_i beträgt sieben Takte je Punktverdoppelung und Vergleich der x-Komponenten. Bei Einsatz von $s = 2 \cdot (n - 7)$ Suchpunkten beträgt die Laufzeit n Takte.

Mit Hilfe der stochastischen Betrachtungen des Kapitels 5.6.3 läßt sich abschätzen, wie groß der Vorteil dieses Verfahrens ist. Man betrachte dazu eine Beispielkurve E mit der Ordnung $\#E = 20.000$. Nach Tabelle 6 auf Seite 45 benötigt man 16.000 Verdoppelungen zum Erreichen einer Erfolgswahrscheinlichkeit von 80%, wenn man $s = 1$ Suchpunkt benutzt. Das bedeutet einen Zeitaufwand von ca. $16.000 \cdot 7 = 112.000$ Takten. Bei Benutzung von $s = 8$ Suchpunkten erreicht man die 80%-Schwelle bereits nach 3.644 Verdoppelungen. Für jede Verdoppelung mit Vergleich der x-Komponenten sind jetzt allerdings, wie auch aus Abbildung 23 ersichtlich wird, 11 Takte nötig. Das ergibt eine Gesamtlaufzeit von ca. $3.644 \cdot 11 = 40.084$ Takten. Damit ist das Verfahren durch

Zustand		gesuchte Punkte 1	2	3	4	5	6	7	8	...		s
1	herkömmliche Strategie											
2												
3												
4		×										
5												
6												
7		=	×									1
8			=	×	×							2
9				=	=	×	×					4
10						=	=	×	×			6
11								=	=			8
⋮										⋱		⋮
n											=	$2 \cdot (n-7)$
n+1			u.U. Multiplikation mit Z_2^3									
n+2			u.U. Vergleich mit Y_2									

× Multiplikation mit Z_2^2
= Vergleich mit X_2

Abbildung 23: Ablaufschema des Suchens mit mehreren Punkten

Einsatz von sieben zusätzlichen Suchpunkten um den Faktor **2.9** beschleunigt worden.
Ohne weitere Darlegungen ist offensichtlich, daß die Beschleunigung durch den Einsatz zusätzlicher Suchpunkte deutlich größer ist, als die daraus resultierende Verzögerung durch zusätzliche Takte.
Ein Schwachpunkt des o.a. Verfahrens könnte der Chipflächenbedarf sein. Je nach betrachteter Bitbreite und Anzahl der Suchpunkte kann der Bedarf an zusätzlichen Registern sehr hoch ausfallen. Aus diesem Grund kann es zweckmäßig sein, einen Teil der Logik auf ein zweites FPGA auszulagern. Dieser könnte zur Speicherung der Suchpunkte dienen, oder, wenn er auch über einen Multiplizierer verfügt, die x-Komponenten der Suchpunkte mit Z_2^2 multiplizieren.
Denkbar ist auch ein Assoziativspeicher, der, wenn an seinem Eingang beispielsweise X_2 angelegt wird, die dazu äquivalenten Suchpunkte ausgibt.

6.3 Der FPGA-Baustein

Die Realisierung des VHDL Modells erfolgt hier in Form eines FPGA (Field Programmable Gate Array). Dies ist die für diese Anwendung kostengünstigste Methode der Realisierung. FPGA´s sind Standardprodukte, die in Massenfertigung hergestellt werden, aber trotzdem eine anwendungsspezifische Funktion ausüben können. Dazu wird der FPGA vor Ort programmiert und ist damit viel schneller verfügbar als die Alternative, der speziell zur Ausführung dieser Funktion entworfene ASIC (Application Specific Intergrated Circuit). Zusätzlich ist das FPGA durch Reprogrammierbarkeit sehr flexibel. Dies ist in

diesem Fall besonders wichtig, da je nach Bitbreite des Körpers der elliptischen Kurve eine andere Schaltung zur Anwendung kommt. Nach [Wan98] sind FPGAs aus vielen kleinen Logikzellen, sog. LE´s (Logic Elements) in einem Array angeordnet. Diese können über ein Netzwerk von Verbindungsleitungen in Form einer Schaltmatrix-Verdrahtung miteinander verbunden werden. Die Verbindungsinformationen für die Schaltmatrix werden in sog. LUT´s (Look Up Tables) programmiert. Die Anzahl und Art der Segmente in einem Signalpfad bestimmen die Signallaufzeit einer Verbindung. Deshalb kann erst nach der Aufteilung der Funktionen in LE´s und deren Verdrahtung das genaue zeitliche Verhalten des FPGA festgestellt werden.
Je nach Hersteller des FPGA sind die LE´s unterschiedlich aufgebaut. Eine wichtige Eigenschaft ist nach [Wan98] die Granularität des Bausteins. Haben die LE´s nur wenige Ein- und Ausgänge und kann nur wenig anwendungsspezifische Logik in einem Block implementiert werden, so spricht man von einer feinen, im gegensätzlichen Fall von einer groben Granularität. Eine feine Granularität ist synthesefreundlich, d.h. für die automatischen Synthesewerkzeuge ist es einfach, die gewünschte Logik auf die vorhandenen LE-Ressourcen abzubilden. Je feiner die Granularität eines Bausteins jedoch ist, um so mehr Verbindungsleitungen werden benötigt, was wiederum Platz kostet.
Die Realisierung der Anwendung dieser Arbeit soll auf einem FPGA der Bausteinfamilie FLEX10K der Firma Altera stattfinden. Die FLEX10K-Reihe wurde 1995 in der ersten Version FLEX10K/-A vorgestellt. Diese Bausteine werden in 0.5 μm Technologie mit drei Metallisierungsebenen gefertigt. Es gibt acht verschiedene Bausteine mit einer Komplexität von 10.000 bis zu 130.000 Gattern, was 576 bis 6.656 LE´s entspricht.
Seit 1998 existiert die FLEX10KB-Reihe. Diese Bausteine werden in 0.25 μm Technologie mit fünf Metallisierungsebenen hergestellt. Die Komplexität liegt zwischen 30.000 und 250.000 Gattern, bzw. 1.728 bis 12.160 LE´s. Bei Bausteinen dieser Art besteht jede LE im Wesentlichen aus einer Funktionsgeneratoreinheit mit vier Eingängen und einem programmierbaren Flipflop. Mit Hilfe des Funktionsgenerators ist eine LE in der Lage, jede gewünschte Verknüpfung von vier Variablen durchzuführen [Alt01, S. 14].

7 Ergebnisse

Nach der rein funktionalen Analyse und Beschreibung in Kapitel 6.2 werden nunmehr die tatsächliche Größe des FPGA, seine Geschwindigkeit, sowie die Kosten behandelt.

7.1 Der Synthesevorgang

Um das VHDL-Modell in eine Form zur Programmierung in ein FPGA zu bringen, findet die sog. Synthese statt. Da hier die FLEX10K-Bausteine von Altera zum Einsatz kommen, liegt es nahe dazu die Altera-Software Max+Plus-II zur verwenden. Diese vollführt die Synthese der VHDL-Beschreibung u.a. durch den Einsatz folgender Module [Alt00, S. 133 ff.]:

Compiler Netlist Extractor: Dieses Modul konvertiert jede VHDL-Projektdatei in eine oder mehrere hierarchisch geordnete Netzlistendateien.

Logic Synthesizer: Der Logik Synthesizer benutzt diverse Algorithmen zur Eliminierung redundanter Logik und sorgt dafür, daß die in der Zielbausteinfamilie enthaltenen Logikelemente möglichst effizient genutzt werden.

Fitter: Der Fitter ordnet den Anforderungen des Projektes die passenden Ressourcen des Zielbausteins zu. Er weist jeder Logikfunktion die bestmöglichen Logikzellen zu und bestimmt die passenden Verbindungspfade und Pins.

Functional SNF Generator: Dieses Modul erzeugt ein funktionales *Simulator Netzlist File*, welches alle Netze des Projektes beinhaltet.

Timing SNF Generator: Mit diesem Modul wird ein *Timing Simulator Netzlist File* erzeugt, das die Timing-Daten zu jedem Netz enthält. Damit sind Aussagen über das Zeitverhalten des FPGA, die sog. Timing Analyse, möglich. Diese Datei wird benötigt, wenn die Ausgabe der Synthese in Form einer EDIF-Datei (.edf) (EDIF: Electronic Design Interchange Format) stattfinden soll. Das EDIF-Format in den Versionen 2 und 3 wird zum Austausch von Projektdaten zwischen Simulatoren, CAE- (Computer Aided Engineering) Systemen etc. benutzt.

Assembler: Der Assembler konvertiert die Ergebnisse des Fitter-Moduls bzgl. Logikzellen, Pin-Belegung und Netze in ein *Programmer Object File* (.pof). Mit Hilfe der .pof-Datei und eines FPGA-Programmiergerätes, wie z.B. dem Altera Byte-Blaster, kann das FPGA programmiert werden.

Da am Arbeitsbereich TECH des Fachbereiches Informatik der Universität Hamburg neben Altera Max+Plus-II in der Version 10.0 auch der Synopsys FPGA-Compiler-II in der

Version 3.6.1 zur Verfügung steht, wird hier auch letztere Software zur Synthese eingesetzt. Der FPGA-Compiler-II wird dabei zur Erzeugung einer EDIF-Datei aus der VHDL-Beschreibung benutzt. Mit anderen Worten, es werden alle Module bis auf den Assembler vom Synopsys FPGA-Compiler-II ausgeführt. Es zeigt sich, daß die Synopsys-Software aus der VHDL-Beschreibung eine EDIF-Datei generiert, die mit weniger Logikzellen auskommt, als die EDIF-Datei der Max+Plus-II-Software. Aus diesem Grund basieren alle folgenden Syntheseresultate auf der vom Synopsys FPGA-Compiler-II generierten EDIF-Datei. Diese wird dann zur Synthesevervollständigung in den Altera Max+Plus-II-Compiler importiert, der zum Schluß eine .pof-Datei erzeugt. Mit dieser ist dann das FPGA z.B. dem Altera Byte-Blaster programmierbar.
Maßgeblich für das Ergebnis der Synthese ist die Auswahl der Bitbreite der den Berechnungen zugrundeliegende elliptischen Kurve. Die Bitbreite, die in der VHDL-Beschreibung in Form der `generic`-Varialbe `len` beschrieben ist, bestimmt hauptsächlich die Anzahl der Logikzellen (Logic Elements, LE) des FPGA.

7.2 Ergebnis der Synthese eines Multiplizierers

Im Folgenden soll zunächst die VHDL-Beschreibung der Funktion, die einen einzelnen Multiplizierer beschreibt, isoliert betrachtet werden. Bildet man diese Funktion mit Hilfe des Synopsys VHDL-Compilers auf eine allgemeine Gatterbibliothek ab, so ergibt sich der in Tabelle 7 dargestellte Gatterbedarf, in Abhängigkeit von der jeweiligen Bitbreite. Die Angabe der Gatteranzahl ist hier wenig aussagekräftig, da der Inhalt der Gatterbibliothek mit Absicht nicht genauer dargestellt ist. Von Interesse ist der Faktor des Aufwandes an zusätzlichen Elementen bei Erhöhung der Bitbreite. Aus diesem Grund ist in Tabelle 7 ein Index angegeben, der den Bedarf von 306 Gattern bei 10 Bit als 100% normiert. Man

Bit	10	20	30	40	50	60
Gatterbedarf	306	1.430	3.341	6.051	9.559	13.728
Index	100	467	1.092	1.977	3.124	4.486

Tabelle 7: Allgemeiner Gatterbedarf eines Multiplizierers (vgl. Abb. 24)

beachte, daß die hier angeführte allgemeine Gatterbibliothek völlig unabhängig von den Logikzellen eines FPGA ist. Abbildung 24 stellt den Verlauf des in Tabelle 7 angegebenen Gatterbedarfs graphisch dar. Man erkennt, daß mit zunehmender Bitbreite die Menge der Logikelemente steigt. Die Verdoppelung der Bitbreite hat ungefähr eine Vervierfachung der Logikelemente zur Folge.

7.3 Ergebnis der Synthese des zentralen Entwurfes

Zur Durchführung der Synthese des zentralen Entwurfes mit verschiedenen Bitbreiten wird in der `entity`-Definition des VHDL-Codes die `generic`-Variable `len` auf den

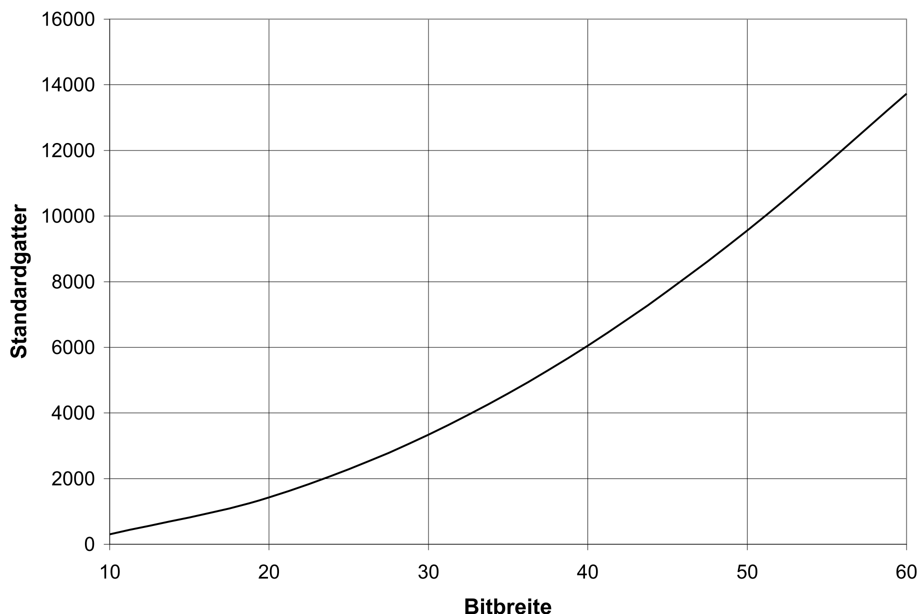

Abbildung 24: Bedarf eines Multiplizierers an Standardgattern

jeweils gewünschten Wert eingestellt. Dieser Code wird dann nach dem in Kapitel 7.1 angegebenen Schema mit Hilfe des Synoposys FPGA-Compiler-II bis zum Stadium einer EDIF-Datei synthetisiert. Aus dieser sind dann alle relevanten Parameter des resultierenden FPGA, wie Anzahl der Logikzellen und Gesamtgeschwindigkeit, ersichtlich.
Die Synthese wurde auf einer SunSparc Ultra-2 mit zwei 200 MHz Prozessoren durchgeführt. Tabelle 8 zeigt als ein Resultat der Synthese die Anzahl der benötigten Logikelemente in Abhängigkeit von der jeweiligen Bitbreite. Die Zunahme der Logikelemente

Bit	10	20	30	40	50	60
Logikelemente	864	2.286	4.317	7.210	10.635	14.700
Index	100	265	500	835	1.231	1.701

Tabelle 8: Bedarf des zentralen Entwurfs an Logikelementen (vgl. Abb. 25)

zeigt einen ähnlichen Verlauf wie bei der Synthese eines einzelnen Multiplizierers. Dies läßt den Schluß zu, daß die Multiplizierer der Hauptfaktor für die Größenentwicklung des FPGA sind. Abbildung 25 stellt die Daten der Tabelle 8 graphisch dar.
Die Syntheseanalyse des zentralen Entwurfes kann noch weiter verfeinert angegeben werden, wie in Tabelle 9 gezeigt. Es ist hier exemplarisch die Synthesezeitdauer, sowie die Anzahl der benötigten Flipflops, in Abhängigkeit von der Bitbreite dargestellt.

Bit	10	20	30	40	50	60
Synthesezeit [min]	5	15	35	82	145	264
Flipflops	162	272	382	492	602	712

Tabelle 9: Synthesezeitaufwand und Anzahl der Flipflops des zentralen Entwurfs

Der für die Durchführung der Attacke wichtigste Aspekt ist die maximal erreichbare

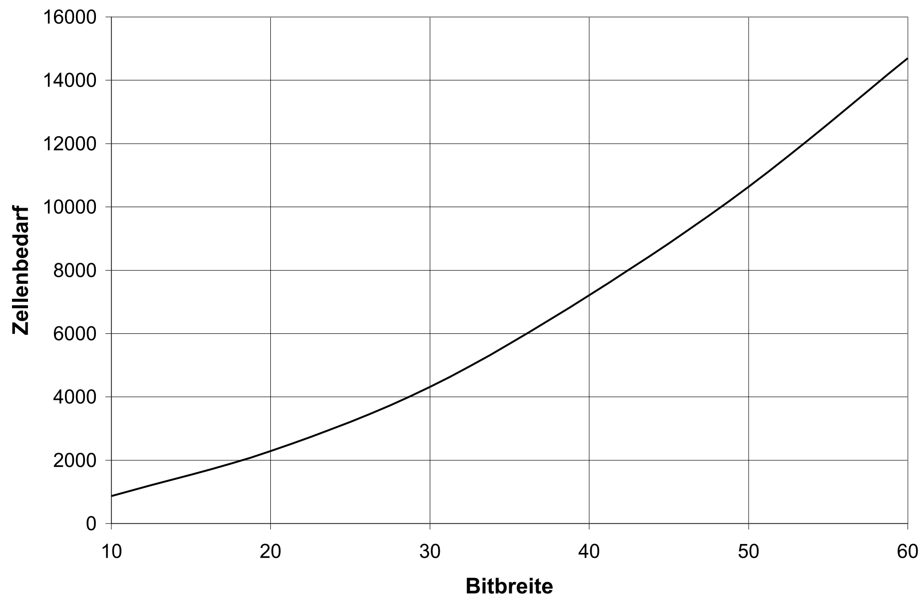

Abbildung 25: Benötigte Logikelemente für den zentralen Entwurf

Taktfrequenz des FPGA. In Abhängigkeit von der Bitbreite der zugrundeliegenden elliptischen Kurve wird, wie vorstehend dargestellt, die Anzahl der benötigten Logikelemente des FPGA deutlich umfangreicher. Damit nimmt die durchschnittliche Netzlänge, sowie die Anzahl der zu durchlaufenden Gatter je Signal, zu. Da die Gesamtgeschwindigkeit des FPGA durch den langsamsten Signalpfad bestimmt wird, sinkt die maximale Taktgeschwindigkeit des FPGA mit zunehmender Bitbreite. Ausschlaggebend dabei ist die externe Taktfrequenz des FPGA. Diese wird von den verwendeten Synthesesoftwares standardmäßig auf einen Wert von 50 MHz eingestellt. Es sei hier erwähnt, daß bereits FPGA am Markt verfügbar sind, die externe Taktfrequenzen von 300 MHz unterstützen.

Aus der Gesamtheit der Signalverzögerungen und dem kritischsten Signalpfad läßt sich somit die maximale Taktrate des FPGA bestimmen. Diese ist, in Abhängigkeit von der jeweiligen Bitbreite, in Tabelle 10 angegeben. Auch hier ist zur besseren Vergleichbarkeit der Ergebnisse ein Index mit angegeben, der mit einer Bitbreite von 60 Bit bei 2,37 MHz zu 100% normiert wurde.

Bit	10	20	30	40	50	60
MHz	10,83	6,32	4,46	3,44	2,81	2,37
Index	457	267	188	145	119	100

Tabelle 10: Maximale Taktrate des FPGA (vgl. Abb. 26)

Abbildung 26 stellt die oberen Grenzen der Taktfähigkeit des FPGA dar, je nach Bitbreite der elliptischen Kurve. Es wird durch Vergleich der Abbildungen 25 und 26 deutlich, daß die maximale Taktrate im gleichen Maß bei zunehmender Bitbreite abfällt, wie auch der Bedarf an Logikzellen zunimmt.

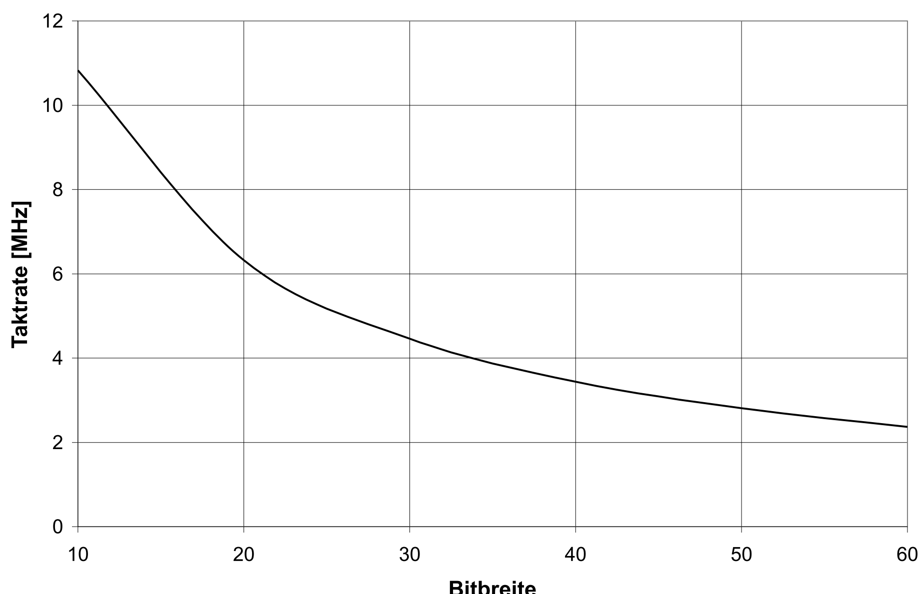

Abbildung 26: Maximale Taktrate des FPGA

7.4 Analyse der Syntheseresultate

Durch den in Kapitel 6.2.4 auf Seite 63 dargestellten Verlauf der Angriffsstrategie, benötigt man i.A. sieben Takte für eine Punktverdoppelung unter Berücksichtigung eines einzelnen Suchpunktes. Durch die maximale Taktrate des FPGA bei gegebener Bitbreite steht fest, wieviele Verdoppelungen pro Sekunde durchgeführt werden können. Es ergibt sich für die Anzahl der Verdoppelungen in jeder Sekunde:

$$\frac{\text{Verdoppelungen}}{\text{Sekunde}} = \frac{f_{max} \cdot 10^6}{7}, \text{ mit } f_{max} = \text{max. Taktrate [MHz]} \tag{39}$$

Berechnet man mit den aus der Synthese resultierenden maximalen Taktraten in Abhängigkeit von der Bitbreite die Verdoppelungen pro Sekunde, so ergibt sich Tabelle 11. Da

Bit	10	20	30	40	50	60
Verdopp./Sekunde	1.547.143	902.857	637.134	491.429	401.429	338.571
Index	457	267	188	145	119	100

Tabelle 11: Verdoppelungen pro Sekunde

die Verdoppelungen pro Sekunde direkt von der Taktfrequenz abhängen, ist der graphische Verlauf von Tabelle 11 identisch mit dem in Abbildung 26 und deshalb hier nicht zusätzlich gezeigt.

Wie in Kapitel 5.6.3 von Seite 43 sowie in Kapitel 6.2.4 auf Seite 63 ausführlich dargelegt wurde, ist vor allem der Zeitaufwand zum Erreichen einer gewissen Wahrscheinlichkeitsschwelle zum Finden eines bestimmten Punktes $k \cdot B$ mittels einem oder mehrerer Suchpunkte S_i interessant.

Bei gegebener Bitbreite von 20 Bit kann leicht ermittelt werden, in welcher Zeit man mit

einer Wahrscheinlichkeit von 50% einen gesuchten Punkt mittels eines einzelnen Suchpunktes S_1 findet:
Die Bitbreite einer elliptischen Kurve von 20 Bit impliziert nach dem Hasse-Theorem (vgl. Formel (29) von Seite 32) näherungsweise eine Kurvenordnung $\#E$ von 2^{20} = 1.048.576. Um einen Punkt mit einer Wahrscheinlichkeit von 50% zu finden, muß die Hälfte der Kurvenordnung geprüft werden, d.h. es sind (1.048.576 / 2) = 524.288 Verdoppelungen durchzuführen. Da jede Verdoppelung i.d.R. sieben Takte benötigt, ist dazu ein Aufwand von (524.288 · 7) = 3.670.016 Takten notwendig. Die Synthese ergab, daß das FPGA bei einer Bitbreite von 20 Bit mit einer maximalen Taktfrequenz von 6,32 MHz betrieben werden kann. Damit benötigt man für die Durchführung (3.670.016 / 6.320.000) = 0,581 Sekunden.
In dem o.a. Beispiel einer elliptischen Kurve mit Ordnung $\#E = 2^{20}$ = 1.048.576 sucht man mittels eines einzelnen Suchpunktes. Die Anzahl der Verdoppelungen zum Erreichen der Wahrscheinlichkeit von 50% errechnet sich durch die Formel (36) von Seite 44, wonach gilt:

$$P(A_{\#E,s,r}) = \sum_{u=0}^{s-1} \frac{\binom{s}{s-u}\binom{\#E-s}{r-(s-u)}}{\binom{\#E}{r}} .$$

Gesucht ist dabei die Anzahl der Verdoppelungen r, so daß gilt: $P(A_{2^{20},1,r}) = 0,5$. Das ist für r = 524.288 erfüllt.
Steht eine gewisse Anzahl von Suchpunkten zur Verfügung, so ist zur Berechnung des zeitlichen Aufwands zum Erreichen einer Wahrscheinlichkeit w folgendes zu berechnen:

$$\text{Zeit [s]} = \frac{[\text{Anz. Verdoppelungen fuer } P(A_{\#E,s,r}) = w] \cdot [\text{Takte pro Verdoppelung}]}{\text{maximale Taktfrequenz}} \Rightarrow$$

$$T_{(\#E,s,w,f_{max})}\,[\text{Sekunden}] = \frac{[P(A_{\#E,s,r}) = w] \cdot \left\lfloor \frac{s}{2} + 7 \right\rfloor}{f_{max} \cdot 10^6}, \text{ mit} \quad (40)$$

s: Anzahl der Suchpunkte (vgl. Kap. 6.2.4, Seite 63),
f_{max}: maximale Taktrate in MHz (vgl. Tab. 10, Seite 69),
$P(A_{\#E,s,r}) = w$: Anzahl der notwendigen Verdoppelungen zum Finden eines Punktes mittels s Suchpunkten mit Wahrscheinlichkeit w bei einer Kurvenordnung von $\#E$.

In den Tabellen 12 und 13 sind die notwendigen Rechenzeiten zum Suchen auf einer elliptische Kurve mit 20 Bit in Abhängigkeit von 50%- und 80%-Wahrscheinlichkeiten und der Menge der Suchpunkte angegeben.
In der ersten Spalte der beiden Tabellen ist die Kardinalität s der verwendeten Suchpunktmenge angegeben. Wie bereits in Abbildung 23 auf Seite 64 dargestellt, ist die in

Such-punkte s	nötige Takte n	Rate $\frac{1}{z}$ z	Verdoppelungen $P(A_{2^{20},s,r}) = 0{,}5$ r	Takte $n \cdot r$	Zeitaufwand $T_{(2^{20},s,(0,5),(6,32))}$ Sekunden
		1,00	1.048.576	7.340.032	1,161
1	7	2,00	524.288	3.670.016	0,581
6	10	9,17	114.400	1.144.000	0,181
10	12	14,93	70.220	842.640	0,133
16	15	23,59	44.456	666.840	0,106
20	17	29,36	35.718	607.206	0,096
26	20	38,01	27.585	551.700	0,087
30	22	43,78	23.949	526.878	0,083
36	25	52,44	19.996	499.900	0,079
40	27	58,21	18.014	486.378	0,077
50	32	72,64	14.436	461.952	0,073
60	37	86,92	12.064	446.357	0,071
70	42	101,45	10.336	434.107	0,069
80	47	115,79	9.056	425.625	0,067
90	52	130,03	8.064	419.334	0,066
100	57	144,35	7.264	414.055	0,066
200	107	289,10	3.627	388.089	0,061
500	257	722,16	1.452	373.164	0,059

Tabelle 12: Zeitaufwand zum Erreichen von $P(A_{2^{20},s,r}) = 0{,}5$

Such-punkte s	nötige Takte n	Rate $\frac{1}{z}$ z	Verdoppelungen $P(A_{2^{20},s,r}) = 0{,}8$ r	Takte $n \cdot r$	Zeitaufwand $T_{(2^{20},s,(0,8),(6,32))}$ Sekunden
		1,00	1.048.576	7.340.032	1,161
1	7	1,25	838.861	5.872.027	0,929
6	10	4,25	246.704	2.467.040	0,390
10	12	6,73	155.881	1.870.572	0,294
16	15	10,45	100.344	1.505.160	0,238
20	17	12,93	81.074	1.378.258	0,218
26	20	16,66	62.939	1.258.780	0,199
30	22	19,15	54.771	1.204.962	0,191
36	25	22,87	45.845	1.146.125	0,181
40	27	25,36	41.352	1.116.504	0,177
50	32	31,57	33.214	1.062.848	0,168
60	37	37,78	27.752	1.026.824	0,163
70	42	44,00	23.833	1.000.986	0,158
80	47	50,21	20.884	981.548	0,155
90	52	56,42	18.584	966.368	0,153
100	57	62,64	16.740	954.180	0,151
200	107	124,79	8.403	899.121	0,142
500	257	311,24	3.369	865.833	0,137

Tabelle 13: Zeitaufwand zum Erreichen von $P(A_{2^{20},s,r}) = 0{,}8$

der zweiten Spalte angegebene Anzahl der benötigten Takte n abhängig von der Anzahl der Suchpunkte: $n = \left\lfloor \frac{s}{2} + 7 \right\rfloor$. In der vierten Spalte ist die Anzahl der Verdoppelungen r angegeben, die nötig sind um $P(A_{2^{20},s,r}) = 0{,}5$ zu erfüllen. Die in der dritten Spalte angeführte Rate z gibt das Verhältnis von Kurvenordnung zu notwendigen Verdoppelungen bis zum Erreichen der Wahrscheinlichkeit an. In Tabelle 12 müssen beispielsweise bei Verwendung von $s = 50$ Suchpunkten nur 14.436 Verdoppelungen zum Erreichen der Wahrscheinlichkeit von 50% durchgeführt werden. Damit müssen nur $(1/z)$ Punkte der Kurvenordnung durchsucht werden mit $z = (1.048.576 \ / \ 14.436) = 72{,}64$. Die fünfte Spalte stellt die insgesamt notwendigen Takte zur Berechnung dar, die sich durch die Multiplikation der Anzahl der nötigen Takte n einer Verdoppelung mit s Suchpunkten und der Anzahl der nötigen Verdoppelungen r ergibt. Aus diesen Werten läßt sich leicht mittels Formel (40) der jeweils resultierende Zeitaufwand errechnen.

Die Tabellen 12 und 13 machen erneut deutlich, daß die Zeitersparnis durch den Einsatz mehrerer Suchpunkte sehr groß ist. Allerdings ist erkennbar, daß die Einsparungen bei der Verwendung sehr vieler Suchpunkte weniger stark zunehmen als bei der Betrachtung weniger Suchpunkte. Dies wird durch die Abbildung 27 verdeutlicht. Da, wie bereits

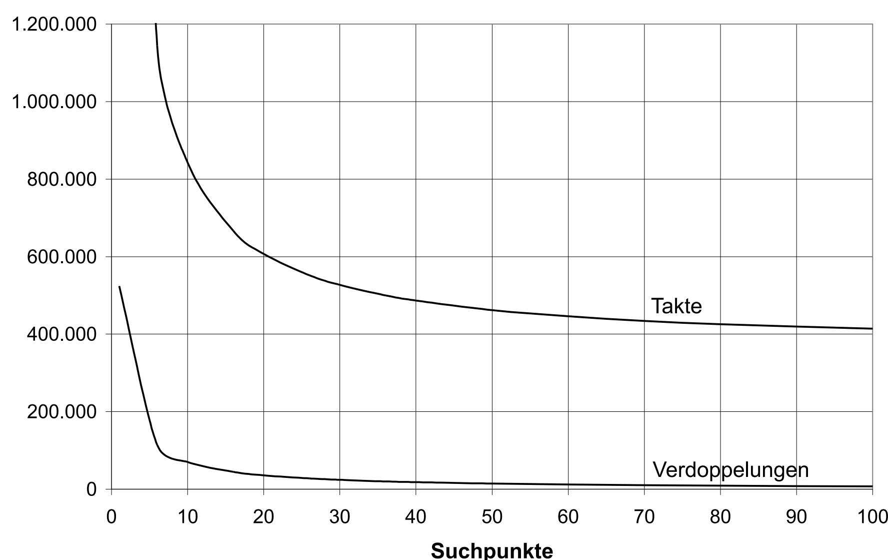

Abbildung 27: Nötige Verdoppelungen und Takte für $P(A_{2^{20},s,r}) = 0{,}5$

angeführt, die Steigerung der Anzahl der Suchpunkte auch die Anzahl der nötigen Takte je Verdoppelung anhebt, ist der Gewinn nicht linear. Abbildung 27 zeigt, daß bereits bei Einsatz von wenigen Suchpunkten die nötigen Verdoppelungen extrem verringert werden. Bei Betrachtung des Verlaufs der nötigen Takte sieht man, daß diese Einsparungen mit zunehmender Suchpunktzahl immer geringer werden. Aus Tabelle 12 wird beispielsweise ersichtlich, daß bei 500 Suchpunkten immer noch 373.164 Takte auszuführen sind. Abbildung 27 macht deutlich, daß bereits die Verwendung von 50 Suchpunkten ausreicht um einen hinreichenden Einsparungseffekt zu erzielen. Es muß darauf hingewiesen wer-

den, daß die Verwendung von sehr vielen Suchpunkten auch einen hohen zusätzlichen Hardwareaufwand bedeutet, der bei 50 Suchpunkten u.U. noch moderat ausfallen dürfte. Durch die Ermittlung der Rate $(1\ /\ z)$ läßt sich der Rechenzeitbedarf für die Wahrscheinlichkeitsschwellen in Abhängigkeit von Bitbreite und Anzahl der Suchpunkte auch für größere Bitbreiten interpolieren. Die Tabellen 14, 15 und 16 zeigen die nötigen Rechenzeiten für Kurven mit einer Bitbreite von 30, 40 und 50 Bit bei 50, bzw. 80% Wahrscheinlichkeit.

Such-punkte s	Verdoppelungen $P(A_{2^{30},s,r}) = 0{,}5$ r	Zeitaufwand $T_{(2^{30},s,(0,5),(4,46))}$ Sekunden	Verdoppelungen $P(A_{2^{30},s,r}) = 0{,}8$ r	Zeitaufwand $T_{(2^{30},s,(0,8),(4,46))}$ Sekunden
	1.073.741.824	1685,25	1.073.741.824	1685,25
1	536.870.912	842,62	858.993.665	1348,20
6	117.145.600	262,66	252.624.901	566,42
10	71.905.283	193,47	159.622.145	429,48
16	45.522.945	153,10	102.752.262	345,58
20	36.575.233	139,41	83.019.782	316,44
26	28.247.040	126,67	64.449.539	289,01
30	24.523.776	120,97	56.085.507	276,66
36	20.475.904	114,78	46.945.282	263,15
40	18.446.336	111,67	42.344.449	256,35
50	14.782.464	106,06	34.011.137	244,03
60	12.353.219	102,48	28.418.048	235,76
70	10.583.951	99,67	24.404.992	229,82
80	9.273.183	97,72	21.385.216	225,36
90	8.257.647	96,28	19.030.016	221,88
100	7.438.461	95,07	17.141.760	219,08
200	3.714.048	89,10	8.604.673	206,44
500	1.486.848	85,68	3.449.856	198,79

Tabelle 14: Zeit zum Erreichen von $P(A_{2^{30},s,r}) = 0{,}5$ und $P(A_{2^{30},s,r}) = 0{,}8$

Die Rechenzeiten sind in allen Tabellen in Sekunden angegeben, damit sie direkt vergleichbar sind. In Abbildung 28 sind alle Zeiten in einem Graph zusammengefaßt dargestellt. Man kann gut erkennen, daß die Verläufe der durchzuführenden Taktzahlen bei den verschiedenen Bitbreiten gleich sind, und sich nur auf verschiedenen Niveaus befinden. Aus der Abbildung 28 geht außerdem hervor, daß die Durchführung einer Attacke auf eine elliptische Kurve mit einer Bitbreite von 50 Bit nicht mehr in akzeptabler Zeit möglich ist. Unter Verwendung von 500 Suchpunkten benötigt ein FPGA zur Berechnung der 50%-Wahrscheinlichkeit ca. 4,5 Jahre. Die selbe Berechnung unter Verwendung einer 40 Bit breiten Kurve benötigt dagegen eine Rechenzeit von ca. 1,3 Tagen.

Somit kann eine Bitbreite von 40 bis 45 Bit als Obergrenze für die Durchführung einer Attacke angesehen werden.

Für die Durchführung einer Attacke sind die entstehenden finanziellen Kosten ein wichtiger Faktor. Gerade bei einem Angriff auf eine Kurve mit hoher Bitbreite ist der Bedarf

Such-punkte s	Verdoppelungen $P(A_{2^{40},s,r}) = 0{,}5$ r	Zeitaufwand $T_{(2^{40},s,(0,5),(3,44))}$ Sekunden	Verdoppelungen $P(A_{2^{40},s,r}) = 0{,}8$ r	Zeitaufwand $T_{(2^{40},s,(0,8),(3,44))}$ Sekunden
	1.099.511.627.776	2.237.378	1.099.511.627.776	2.237.378
1	549.755.813.888	1.118.689	879.609.513.327	1.789.903
6	119.957.094.739	348.712	258.687.898.921	752.000
10	73.631.010.266	256.852	163.453.076.563	570.185
16	46.615.495.496	203.265	105.218.316.633	570.185
20	37.453.038.671	185.088	85.012.256.309	420.119
26	28.924.968.997	168.168	65.996.328.001	383.700
30	25.112.346.649	160.602	57.431.558.722	367.295
36	20.967.326.039	152.379	48.071.968.409	367.295
40	18.889.048.257	148.257	43.360.715.606	340.331
50	15.137.243.201	140.812	34.827.404.014	323.976
60	12.649.696.592	136.058	29.100.081.434	312.995
70	10.837.965.774	132.324	24.990.711.890	305.119
80	9.495.739.077	129.738	21.898.461.360	299.194
90	8.455.830.407	127.821	19.486.736.448	294.567
100	7.616.983.913	126.212	17.553.162.415	294.567
200	3.803.185.281	118.297	8.811.184.820	274.069
500	1.522.532.372	113.747	3.532.652.603	263.922

Tabelle 15: Zeit zum Erreichen von $P(A_{2^{40},s,r}) = 0{,}5$ und $P(A_{2^{40},s,r}) = 0{,}8$

Such-punkte s	Verdoppelungen $P(A_{2^{50},s,r}) = 0{,}5$ r	Zeitaufwand $T_{(2^{50},s,(0,5),(2,81))}$ Sekunden	Verdoppelungen $P(A_{2^{50},s,r}) = 0{,}8$ r	Zeitaufwand $T_{(2^{50},s,(0,8),(2,81))}$ Sekunden
	1.125.899.906.842.620	2.804.732.864	1.125.899.906.842.620	2.804.732.864
1	562.949.953.421.312	1.402.366.432	900.720.141.646.933	2.243.786.830
6	122.836.065.012.351	437.139.021	264.896.408.495.335	942.691.845
10	75.398.154.512.497	321.985.001	167.375.950.400.852	714.772.742
16	47.734.267.387.581	254.809.257	107.743.556.232.650	575.143.539
20	38.351.911.598.636	232.022.241	87.052.550.459.991	526.652.441
26	29.619.168.253.032	210.812.585	67.580.239.872.514	480.998.149
30	25.715.042.968.898	201.327.739	58.809.916.131.099	460.433.507
36	21.470.541.863.825	191.019.056	49.225.695.650.979	437.951.029
40	19.342.385.415.428	185.852.102	44.401.372.780.715	426.632.408
50	15.500.537.037.312	176.518.571	35.663.261.710.512	406.129.671
60	12.953.289.310.201	170.559.325	29.798.483.388.148	392.364.372
70	11.098.076.952.613	165.878.730	25.590.488.975.272	382.491.294
80	9.723.636.815.292	162.637.342	22.424.024.432.223	375.063.754
90	8.658.770.336.404	160.233.472	19.954.418.122.679	369.263.254
100	7.799.791.526.447	158.216.412	17.974.438.313.406	364.606.044
200	3.894.461.727.831	148.294.397	9.022.653.255.237	343.567.101
500	1.559.073.149.204	142.591.286	3.617.436.265.428	330.847.138

Tabelle 16: Zeit zum Erreichen von $P(A_{2^{50},s,r}) = 0{,}5$ und $P(A_{2^{50},s,r}) = 0{,}8$

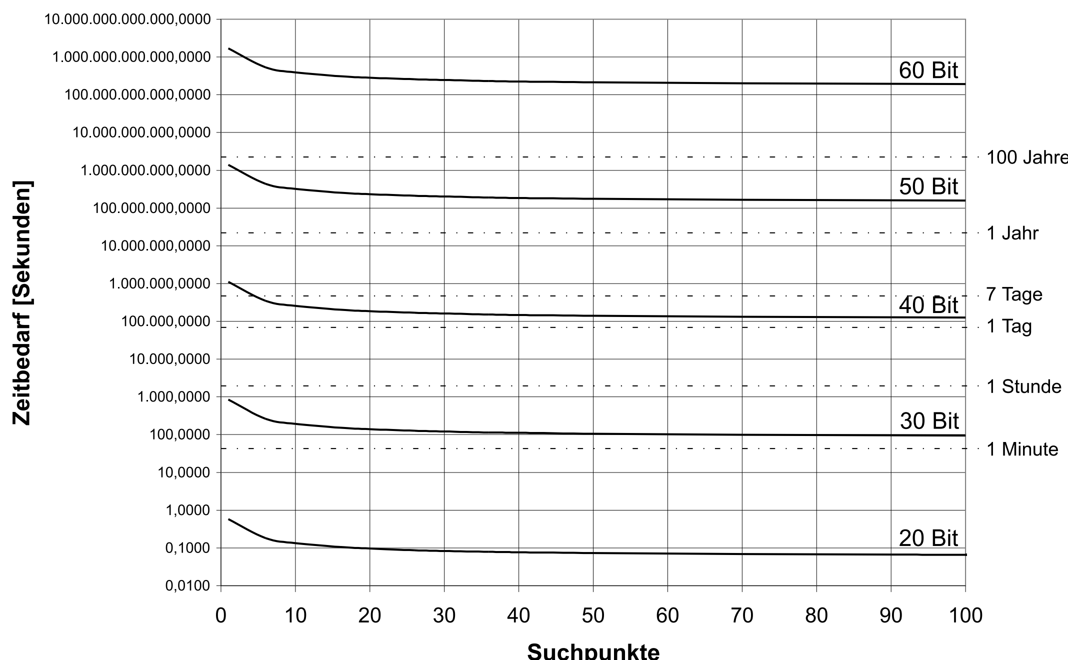

Abbildung 28: Vergleich der Zeitaufwendungen für $P(A_{\#E,s,r}) = 0{,}5$

eines FPGA an Logikzellen sehr groß, wie Abbildung 25 auf Seite 69 gezeigt hat. Deshalb müssen, um so größer die Bitbreite der angegriffenen Kurve ist, entsprechende FPGA verwendet werden, die über eine hinreichende Anzahl an Logikzellen verfügen. Die für die letzten Syntheseschritte verwendete Software Max+Plus-II von Altera schlägt in Abhängigkeit vom Umfang des Design einen dafür geeigneten FPGA-Baustein vor. Die Tabelle 17 zeigt eine kleine Auswahl der sich zum Zweck einer Attacke eignenden Bausteine der Flex10K-Familie der Firma Altera.

Typ	Gates	LE	Preis (EUR)	Typbezeichnung	Pins
10K10	10.000	576	25,00	EPF10K10ATC100-3	100
10K30	30.000	1.728	45,00	EPF10K30ETC144-3	144
10K50	50.000	2.880	89,00	EPF10K50STC144-3	144
10K100	100.000	4.992	157,00	EPF10K100EQC208-3	208
10K130	130.000	6.656	240,00	EPF10K130EQC240-3	240
10K200	200.000	9.984	337,00	EPF10K200SRC240-3	240
10K250	250.000	12.160	1.247,00	EPF10K250ABC600-3	600

Tabelle 17: Rahmendaten und Preise einiger FPGA der Altera Flex10K Familie

Generell geben die Bezeichnungen der verschiedenen Typen der Flex10K-Familie Auskunft über deren Eigenschaften. Er werden dabei u.a. die Gehäuseform, Gehäusematerial, Arbeitstemperaturbereich, Pinzahl und Maximalgeschwindigkeit differenziert. Die meisten dieser Eigenschaften sind für die Kostenabschätzung hier uninteressant. Deshalb wurden der Einfachheit halber die kostengünstigsten Bausteine einer jeden Größenkategorie in Tabelle 17 angegeben.

Die in Tabelle 17 angegebenen Preise beziehen sich auf eine Abnahme von mindestens 25

Stück und sind Stand von Juni 2001. Die Abbildung 29 stellt die Abhängigkeit der Kosten

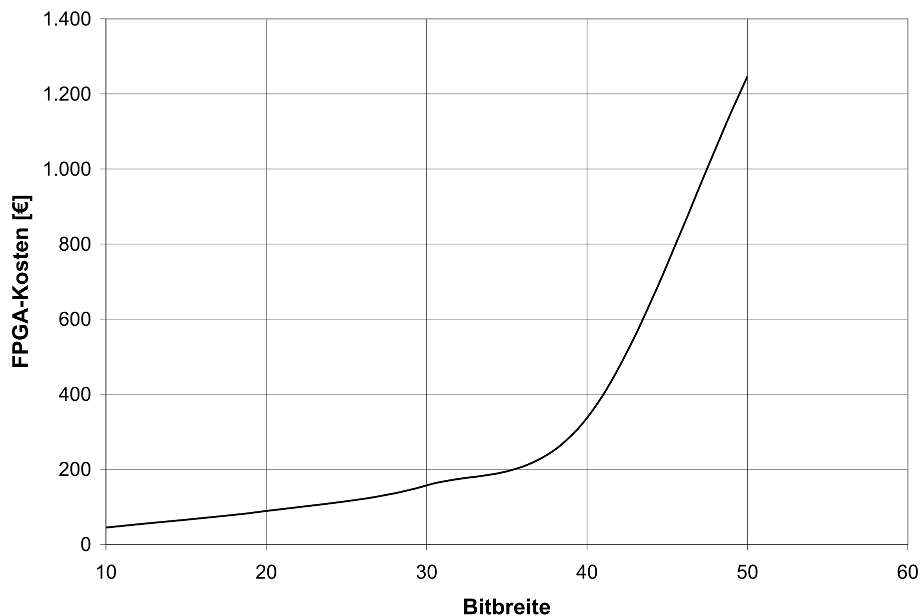

Abbildung 29: FPGA-Kosten in Abhängigkeit von der Bitbreite

von der Bitbreite der angegriffenen elliptischen Kurve dar. Kostenwerte oberhalb von 50 Bit sind in der Abbildung nicht vorhanden, da derzeit kein FPGA der Altera-Flex10K-Familie mehr als 12.160 Logikelemente bietet.
Da ein Angriff auf eine Kurve mit mehr als 40 Bit von der Laufzeit her nicht akzeptabel ist, sind Maximalkosten für ein FPGA mit 7.210 Logikelemente zu erwarten. **Ein solches Altera-Flex10K-FPGA kostet ca. 337 Euro.** Das durch die Verschlüsselung zu sichernde Gut darf dementsprechend nur einen Bruchteil von 337 Euro wert sein, da sich der Angriff ansonsten als lohnend erweisen könnte.
Soll anstelle eines einzelnen FPGA eine Vielzahl von Bausteinen hergestellt werden, so ist die Verwendung eines Standardzellentwurfs sinnvoll. Die Geschwindigkeit dieser Bausteine wird so hoch sein, daß Kurven mit einer Bitbreite von 50 Bit durchaus angreifbar sein werden.

8 Ausblick und nicht behandelte Aspekte

Während der Erstellung dieser Arbeit traten diverse Aspekte auf, deren Behandlung bei dem gegebenen Zeitrahmen leider nicht möglich war. Einige dieser Aspekte sollen im Folgenden thematisch angerissen werden, da sie Ansatzpunkte für weitergehende Arbeiten darstellen.

8.1 Generelle Fragestellungen

Es sind verschiedene Ansätze zum Angriff auf ein ECC bekannt. Jeder Ansatz versucht, gewisse Schwachstellen des Kryptosystems möglichst effektiv auszunutzen. Bei Kryptosystemen basierend auf elliptischen Kurven kann eine solche Schwachstelle z.B. die Beschaffenheit der elliptischen Kurve selbst sein. Die wichtigsten Parameter sind dabei die Art der Kurve, die verwendete Bitbreite, das Primpolynom und die Kurvenordnung. Diese Randbedingungen werden in der Konzeptionsphase des Kryptosystems in Abhängigkeit von den Erfordernissen an die jeweilige Sicherheit von einem Trust-Center festgelegt. In dieser Arbeit wurde davon ausgegangen, daß ein potentieller Angreifer auf eine elliptische Kurve trifft, die für ihn die schwierigsten Voraussetzungen erfüllt. Somit wurde bzgl. des Angriffs eine „worst-case“-Abschätzung aus Sicht des Angreifers durchgeführt. Allerdings ist fraglich, ob ein Trust-Center für jedes zu implementierende Kryptosystem in der Lage oder gewillt ist, die bzgl. diverser Angriffsmethoden „härteste“ elliptische Kurve zu finden. Vielmehr wird es hinsichtlich vorgegebener Rahmenbedingungen elliptische Kurven benutzen, die vielleicht die „gängigsten“ Angriffsstrategien erschweren. In den meisten Fällen werden diese elliptischen Kurven aus heutiger Sicht potentiellen Angriffen mittels Standardstrategien standhalten können.

Alle an diesem Kryptosystem beteiligten Instanzen sind von einer Kosten - Nutzen Relation abhängig. Der Aufwand des Trust-Centers zum Finden einer für diese Anwendung und Sicherheitsstufe geeigneten Kurve muß relativ zum Aufwand zum Brechen des Verfahrens sein. Dieser Aufwand hat adäquat zu dem Wert des zu sichernden Guts zu sein, da sonst die Dimensionen der etwaigen Faktoren Kosten, Hardware-, Entwicklungs- und Wartungsaufwand etc. gesprengt werden könnten. Der Aufwand für den Angreifer wiederum muß so groß sein, daß er sich zur Erlangung des gesicherten Guts nicht lohnt. Sollte der Angreifer allerdings so viel Zeit und evtl. auch Kosten in Kauf nehmen um eine effiziente Strategie zu entwickeln, die zum Erfolg führt, so ist dies u.U. bei kommerziellen Systemen, mit zumeist engen Kostenschranken auf der Seite des Betreibers, nicht zu verhindern.

8.2 Konzeptionelle Angriffsalternativen

Im Zuge der Einarbeitung in das Thema der Angriffe auf Kryptosysteme mittels elliptischer Kurven wurden viele bekannte Strategien recherchiert und analysiert. Es zeigte sich schnell, daß es verschiedenste Möglichkeiten gibt, die bereits genannten Schwachstellen elliptischer Kurven auszunutzen. Jede der Strategien nutzt aber i.A. eine einzelne Gegebenheit aus, um schnellstmöglich zum Erfolg zu kommen. Zusätzlich schränken sie die Menge der elliptischen Kurven ein, auf denen sie überhaupt funktionieren können.
Die Methode der projektiven Punktverdoppelung benötigt z.B. elliptische Kurven, die über eine prime Kurvenordnung verfügen. Zusätzlich muß diese Ordnung auch noch als kleinste Primitivwurzel die 2 haben, andernfalls ist das Verfahren, je nach vorgegebenem Basispunkt, in einer mehr oder weniger großen Schleife gefangen. Wählt man allerdings anstelle der sukzessiven Punktverdoppelung die sukzessive Punktaddition, so braucht man keine Einschränkungen bzgl. der Kurve mehr zu machen. Die sukzessive Punktaddition wird alle Punkte einer elliptischen Kurve errechnen, ohne in eine Schleife zu geraten. Die affine Punktaddition benötigt allerdings die Berechnung einer Inversion. Würde ein effektiver Algorithmus zur schnellen Berechnung der Inversion zur Verfügung stehen, so wäre die sukzessive affine Addition das vorteilhafteste Brute-Force-Angriffsverfahren.
Elliptische Kurven mit Kurvenordnungen ohne Primteiler gelten als besonders sicher. Aus diesem Grund kann das Verfahren der sukzessiven Punktverdoppelung auch die o.g. Voraussetzungen an die Kurvenordnung machen. Dieser Umstand ist dem Trust-Center vor der Konzeption eines Kryptosystems bekannt und es ergibt sich die Frage: werden elliptische Kurven mit nicht-primer Ordnung für ECC-Verfahren zugelassen? Ist es vielleicht erwünscht, daß zu verwendende Kurven diverse Primteiler und damit diverse Untergruppen haben? Das Verfahren der sukzessiven Verdoppelung wäre dann zumindest nur eingeschränkt nutzbar, wie in Kapitel 5.4 ausführlich dargestellt wurde.
Der große Vorteil der Brute-Force-Angriffe ist die hohe Geschwindigkeit der einzelnen Verfahrensschritte. Die kürzeste Zeitspanne benötigt bekanntlich die projektive Punktverdoppelung, wobei die o.g. Einschränkungen bzgl. der Kurven gemacht werden müssen. Es ist u.U. lohnenswert, andere Verknüpfungsarten von Punkten elliptischer Kurven zu untersuchen. Vielleicht existieren Zusammenhänge zwischen Punkten, die sich mit anderen Verknüpfungen als Addition und Verdoppelung, oder anderen Punktdarstellungen als Affin oder Projektiv, sehr schnell berechnen lassen. Durch die Verkürzung der Einzelschritte einer Brute-Force-Attacke läßt sich das Gesamtverfahren erheblich beschleunigen.
Vielleicht lassen sich aber auch durch Verknüpfung bestehender Verfahren neue Strategien entwickeln. Dabei werden bekannte Vorteile der herkömmlichen Verfahren ausgenutzt. Die Verfahren von Pollard und Shanks beispielsweise bestechen durch eine geringe Laufzeit von $O(\sqrt{n})$. Die Zeit, die jeder Schritt dieser Verfahren benötigt, ist allerdings deutlich länger als der Zeitaufwand eines Schrittes der Attacke mittels sukzessiver, projektiver

Punktverdoppelung. Letztere hat aber, auch bei Verwendung mehrerer Suchpunkte, eine im Mittel längere Laufzeit. Es ist durchaus denkbar, daß eine Verknüpfung der beiden Verfahren eine effektive Strategie hervorbringen könnte, bei der die Laufzeit ähnlich kurz wie bei Pollard-Lambda ist, wobei die einzelnen Schritte so schnell durchführbar sind wie bei der projektiven Punktverdoppelung.

8.3 Implementierungsalternativen

Ein Ziel dieser Arbeit sollte sein, eine Aussage über die Leistungsfähigkeit einer hardwaregestützten Attacke gegen ein ECC zu treffen. Zur Bewältigung dieser Aufgabenstellung wurde zunächst eine C++-Verhaltensbeschreibung implementiert, mit deren Hilfe eine für Hardware vorteilhafte Daten- und Modulstruktur entwickelt werden konnte. Die vorliegende VHDL-Beschreibung zeigt deutliche Geschwindigkeitsvorteile gegenüber dem C++-Modell. Dies liegt darin begründet, daß die benutzte Bit-Vektor-Datenstruktur für ein C++-Programm auf „General-Purpose“-Maschinen, wie beispielsweise einer Linux-Arbeitsstation, sehr ungeeignet ist. Für eine Realisierung in Hardware ist diese Datenstruktur allerdings sehr günstig. Es können einerseits einige Verknüpfungen von Punkten einer elliptischen Kurve einfach mittels Gatter durchgeführt werden, andererseits lassen sich auch komplexere Operatoren, wie Punktmultiplizierer, gut realisieren. Es scheint deshalb lohnenswert, geeignete Datenstrukturen auf Basis höherer Programmiersprachen wie C++ für Punkte auf elliptischen Kurven zu erarbeiten. Damit ist man dann in der Lage, die Leistungsfähigkeit von Prozessoren in Arbeitsstationen besser auszunutzen.
Die Resultate dieser Arbeit beziehen sich auf die Syntheseergebnisse der VHDL-Beschreibung für ein FPGA des Herstellers Altera. Ein FPGA wurde deshalb gewählt, da es diese Form der Hardware erlaubt, sehr schnell einen lauffähigen Prototypen zu entwickeln. Fällt nachher beim Test dieses Bausteins ein Design-Fehler auf, so kann man diesen schnell durch Änderung des VHDL-Code, Synthetisieren einer neuen .pof-Datei und Neuprogrammierung des FPGA eliminieren. Auf diese Weise ist man auch sehr flexibel bzgl. Veränderungen der anzugreifenden elliptischen Kurve. Die Multiplizierer arbeiten um so langsamer, je größer deren Bitbreite ist, da der kritische Signalpfad länger wird. Aus diesem Grund wird man vermeiden wollen, mit einem Baustein, dessen VHDL-Modell für 40 Bit synthetisiert wurde, ein 20 Bit breites ECC anzugreifen. Dieser arbeitet nämlich dann nur mit einer Taktfrequenz von 3,44 MHz, anstelle von den für 20 Bit möglichen 6,22 MHz.
Eine große Leistungssteigerung läßt sich erzielen, wenn statt eines FPGA ein ASIC eingesetzt wird, der auf einem Standardzellentwurf basiert. Bei dieser Art des Entwurfes wählt das Synthese-Werkzeug die zu verwendenden Logikelemente aus einer Standardzellbibliothek, die von dem Bausteinhersteller geliefert wird. Feste Umgebungsparameter des FPGA, wie z.B. ein äußerer Takt von 50 MHz, sind hier frei dimensionierbar, bis die Technologie der Standardzellen an ihre Grenzen gerät. Taktraten von mehreren hundert

MHz sollten dabei durchaus realisierbar sein. Nach erfolgter Synthese muß der Baustein allerdings vom Hersteller maschinell gefertigt werden. Für kleine Stückzahlen ist dies wegen der hohen Kosten der Maskenherstellung etc. kaum realisierbar. Dieses Entwurfsvorgehen eignet sich nur für die Massenherstellung von Bausteinen.
Verfügt man über hinreichend Spielraum bzgl. des Entwicklungszeitraums und über gute ASIC-Entwerfer, so kann man auch über einen Entwurf im Full-Custom-Design nachdenken. Hierbei sind alle Bestandteile der Schaltung von Hand mittels Entwurfswerkzeugen herzustellen. Dies ist außerordentlich fehleranfällig, da es wenige den Entwurf überprüfende Hilfsmittel gibt. Grundlegend wird während der Erstellung nur die physikalische Struktur auf Herstellbarkeit und Sinnhaftigkeit durch die sog. Design-Regel-Prüfung (Design-Rule-Check, DRC) kontrolliert. Diese Form des Entwurfes wird dann, wie beim Standardzellentwurf, individuell von einem Chiphersteller gefertigt. Somit eignet sich dieses Vorgehen auch nur für die Massenfertigung von ASICs.

8.4 Anwendungsmöglichkeiten dieser Arbeit

In zunehmendem Maße setzen diverse Hard- und Softwarehersteller für ihre Produkte rund um die Kryptographie elliptische Kurven ein. Um ein Gefühl für die richtige Dimensionierung der Schlüsselbitbreiten zu bekommen, sind Komplexitäts- und Laufzeitabschätzungen von unterschiedlichsten Angriffsstrategien basierend auf Hard- und Software notwendig. Der Impuls zur Erstellung einer Komplexitätsabschätzung von hardwareakzelerierten Attacken auf ECC-Kryptoverfahren kam von der Industrie. In Folge dessen wurde bereits während der Konzeption dieser Arbeit über die kryptographischen Probleme realer Anwendungen nachgedacht.
Möchte eine solche Anwendung z.B. Produkte schützen, so muß man abschätzen können, mit welcher Bitbreite dies geschehen muß, um einem potentiellen Angreifer einen adäquaten Aufwand zum Brechen des Verfahren aufzuerlegen. Eine Smart-Card für ein ECC-Zugangssicherungssystem sollte mit Hilfe eines PC nicht innerhalb von 12 Stunden gebrochen werden können. Die Authentifizierung einer solchen Karte an der Tür sollte allerdings auch nicht länger als 2 Sekunden benötigen. Das Problem ist nur die Abschätzung über den Sicherheitsverlust des Systems pro Jahr durch immer leistungsstärkere Angriffshard- und Software.
Von besonderem Interesse ist deshalb die Erarbeitung von bzgl. der Sicherheit skalierbaren Systemen, bei denen durch weitsichtige Konzeption die Schlüsselbitbreite vergrößert werden kann, um zukünftigen Anforderungen gerecht zu werden. Somit könnte, beispielsweise zur Absicherung eines Modem-Einwahlservers, die Bitbreite des zur Authentifikation notwendigen Schlüssels mit der jährlich erfolgenden Accountverlängerung um 3 Bit erhöht und neu ausgehandelt werden. ECC-Kryptosysteme, die in Software implementiert sind und auf Standardhardware laufen, lassen sich auf diese Weise aktuell halten, da die u.U. höhere Rechenleistung in Folge breiterer elliptischer Kurven und

Schlüssel durch Austausch der darunterliegenden Hardware kompensiert wird.

A Literatur

[Alt00] Altera Corp., *MAX+PLUS II Documentation*, San Jose, 2000

[Alt01] Altera Corp., *FLEX10K FPGA Data Sheet*, ver. 4.1, San Jose, 2001

[Apo76] T. M. Apostol, *Introduction to Analytic Number Theory*, Springer-Verlag, New York, 1976

[BeSW95] A. Beutelspacher, J. Schwenk, K. Wolfenstetter, *Moderne Verfahren der Kryptographie*, Vieweg, Braunschweig/Wiesbaden, 1995

[BlSS99] I. Blake, G. Seroussi, N. Smart, *Elliptic Curves In Cryptography*, Cambridge University Press, 1999

[Boh97] F. Bohnsack, *Untersuchung von Elliptischen Kurven für die Tauglichkeit zur Hardwareakzeleration von Kryptoverfahren*, Diplomarbeit, Fachbereich Informatik der Universität Hamburg, 1997

[DiHe76] W. Diffie, M.E. Hellmann, *New Directions In Cryptography*, IEEE IT 22, 1976

[ElG85] T. ElGamal, *A Public-Key Cryptosystem And A Signature Scheme Based On Discrete Logarithms*, IEEE Transactions On Information Theory, volume 31, pp. 469-472, 1985

[Gor00] S. Gorr, *Konzeption, Evaluierung und Implementation eines Akzelerators für Elliptische Kurven*, Diplomarbeit, Fachbereich Informatik der Universität Hamburg, 2000

[Hor85] P. Horster, *Kryptologie*, Bibliographisches Institut, Zürich, 1985

[Hüb96] G. Hübner, *Stochastik*, Vieweg Verlagsges., Braunschweig/Wiesbaden, 1996

[IEEE99] IEEE, *IEEE P1363/D13. Standard Specifications For Public Key Cryptography*, Annex A, New York, 1999

[Kob98] N. Koblitz, *Algebraic Aspects Of Cryptography*, Springer Verlag, New York, 1998

[LeVe99] Arjen K. Lenstra, Eric R. Verheul, *Selecting Cryptographic Key Sizes*, preprint, 1999

[OrPa99] G. Orlando, C. Paar, *A Super-Serial Galois Fields Multiplier For FPGAs And Its Application To Public-Key Algorithms*, proc. FCCM '99, April 21-23 1999, Napa Valley/CA

[Pol78] J. M. Pollard, *Monte Carlo Methods For Index Computation*, Mathematics Of Computation, Vol. 32, No. 143, pp. 918-924, 1978

[Sil94] J. H. Silverman, *Advanced Topics In The Arithmetic Of Elliptic Curves*, Springer Verlag, New York, 1994

[StTe99] A. Stein, E. Teske, *Catching Kangaroos In Function Fields*, Centre For Applied Cryptographic Research, University Of Waterloo / Canada, 1999

[Wan98] M. Wannemacher, *Das FPGA-Kochbuch*, International Thomson Publishing, Bonn, 1998

[WiZu98] M. J. Wiener, R. J. Zuccherato, *Faster Attacks On Elliptic Curves Cryptosystems*, Selected Areas In Cryptography 1998, pp. 190-200, 1998

B Algorithmen

B.1 C++ - Algorithmus zur Brute-Force Attacke

```
/* Markus Boettger  11/2000 - 05/2001 */
/* Version 1.6v */

// ---------------------------------------------------------- Header-Dateien --

#include <iostream.h>

// -------------------------------------------------- Variablendeklarationen --

/* Bitbreite */
const unsigned int abit = 12; /* setzte abit = bit + 1  !!! */
const unsigned int bit = 11;  /* setze bit = m */

// Zu setzende Eingangswerte; ACHTUNG! LSB links!
  // Primpolynom
int ppoly [abit]={1,0,1,0,0,0,0,0,0,0,0,1};
  // Kurven-b-Koeffizient
int curveb[abit]={1};
  // Feldexponent in Binärdarstellung
int fieldm[abit]={1,1,0,1};
  // Konstante c: c=curveb ^ 2 ^ (m-2)
int c_reg [abit]={1};
  // Startpunkt x-Komponente
int eins_x[abit]={0,1,1};  // = 6
  // Startpunkt y-Komponente
int eins_y[abit]={1,0,0,0,0,1,0,0,1,0,1};   // = 1313
  // Startpunkt z-Komponente
int eins_z[abit]={1};  // = 1
  // gesuchter Punkt x-Komponente
int punkt1_x[abit]={1,0,1,1,1,1,0,1,1};  // = 445
  // gesuchter Punkt y-Komponente
int punkt1_y[abit]={1,1,0,0,0,1,0,0,1,0,1};  // = 1315

// Register
int comp_a[abit];        // Komparatorregister_a
int comp_b[abit];        //                   _b, Ergebnis  in comp_f
int verdopplungen;       // Zaehler für die gesamten Verdopplungen
int zwei_x[abit];        // Register für Partialergebnisse x
int zwei_y[abit];        //                                y
int zwei_z[abit];        //                                z
int zwei_u[abit];        //                                u
int test_x[abit];        // Punktregister_test
int test_y[abit];        //     Register zum Prüfen der Punktegleichheit
int mult_a[abit];        // Multiplikationsregister_a  M1
int mult_b[abit];        //                        _b  M2
int mult_c[abit];        //                        _c  ERGEBNIS
int mult_t1[abit];       //                        _t1 Arbeitsregister

int add_a[bit];          // ECC-Modulo-Addierregister_a SUMMAND 1
int add_b[bit];          //                          _b SUMMAND 2
int add_c[bit];          //                          _b ERGEBNIS
int work_a[abit];        // Arbeitsregister fuer temp. Werte
int z1_quadrat[abit];    // Zwischenspeicher fur Z1²
int z2_quadrat[abit];    // Zwischenspeicher für Z2²
int x1_quadrat[abit];    // Zwischenspeicher für X1²

// feste Vektoren
int nullv [abit];        // fester Nullvektor

// flags
int comp_f  = 0;         // Flag für comp_a = comp_b
int punkt_gefunden = 0;  // Flag zum Suchende

// ------------------------------------------------------------ Programmteil --

// Testet die Vektoren in comp_a[] und comp_b[] auf Gleichheit und
// schreibt das Ergebnis in comp_f
void compare ()
```

```
{
  int flagtmp = 1;
  for (int zv=0; zv<=bit; ++zv)
  {
    if (comp_a[zv] != comp_b[zv])
      {
        flagtmp = 0;
      }
  }
  comp_f = flagtmp;
}

// Kopiert den übergebenen Vektor arr_b[] nach arr_a[]
void load (int arr_a[], int arr_b[])
{
  for (int zv = 0; zv < bit; ++zv)
    {
      arr_a[zv] = arr_b[zv];
    }
}

// Verknüpft die Vektoren add_a[] und add_b[] mittels XOR und
// schreibt das Ergebnis in add_c[];
// Das entspricht der Addition im Körper GF(2^m)
void add ()
{
  for (int zv = 0; zv <= bit; ++zv)
    {
      add_c[zv] = add_a[zv] ^ add_b[zv];
    }
}

// Multipliziert die Vektoren mult_a[] und mult_b[] nach den Regeln des
// Körpers GF(2^m) mit gleichzeitiger Durchführung der modulo-Reduktion
// mittels Primpolynom ppoly[]. Das Ergebnis wird in mult_c geschrieben.
void mult ()
{
  mult_c[-1] = 0;
  load (mult_t1, nullv);
  load (mult_c, nullv);

  for (int i = bit-1; i >= 0; --i)
  {
    for (int j = 0; j <= bit-1; ++j)
    {
      if (j == 0)
      {
        mult_t1[j] = 0 ^ (mult_a[i]&mult_b[j]) ^ (mult_c[bit-1]&ppoly[j]);
      }
      else
      {
       mult_t1[j]=mult_c[j-1] ^ (mult_a[i]&mult_b[j]) ^ (mult_c[bit-1]&ppoly[j]);
      }
    }
    load (mult_c, mult_t1);
  }
}

// berechnet Z2 aus eins_z[] und schreibt in zwei_z[]
void berechne_z ()
{  // Z2 = X1 · Z1²
  load (mult_a, eins_z);
  load (mult_b, eins_z);
  mult();
  load (z1_quadrat, mult_c);  // = z1_quadrat
  load (mult_a, z1_quadrat);
  load (mult_b, eins_x);
  mult();
  load (zwei_z, mult_c);
}

// berechnet X2 aus eins_x[] und schreibt in zwei_x[]
void berechne_x ()
{  // X2 = (X1 + (c · Z1²) )^4
```

```
  load (mult_a, z1_quadrat);
  load (mult_b, c_reg);
  mult();
  load (add_a, mult_c);
  load (add_b, eins_x);
  add();  // = x1+(c·Z1²)
  load (mult_a, add_c);
  load (mult_b, add_c);
  mult();  // = (x1+(c·Z1²))²
  load (mult_a,mult_c);
  load (mult_b,mult_c);
  mult();  // = X2 (x1+(c·Z1²))^4
  load (zwei_x, mult_c);
}

// ber. z2² aus zwei_z[] und schreibt in z2_quadrat[] */
void berechne_z2_quadrat ()
{
  load (mult_a, zwei_z);
  load (mult_b, zwei_z);
  mult();  // = Z2²
        load (z2_quadrat, mult_c);
}

// ber. X = z2_quadrat·punkt1_x und schreibt in test_x
void berechne_test_x ()
{
  load (mult_a, z2_quadrat);
  load (mult_b, punkt1_x);
  mult();  // = X
  load (test_x, mult_c);
}

// ber. U aus eins,zwei und schreibt in zwei_u
void berechne_u ()
{
  load (mult_a, eins_x);
  load (mult_b, eins_x);
  mult();  // = x1²
  load (x1_quadrat, mult_c);
  load (mult_a, eins_y);
  load (mult_b, eins_z);
  mult();  // = y1·z1
  load (add_a, mult_c);
  load (add_b, x1_quadrat);
  add();  // = X1² + Y1·Z1
  load (add_a, add_c);
  load (add_b, zwei_z);
  add();  // = U
  load (zwei_u, add_c);
}

// ber. Y2 aus eins,zwei und schreibe in zwei_y
void berechne_y ()
{
  load (mult_a, x1_quadrat);
  load (mult_b, x1_quadrat);
  mult();  // = x^4
  load (mult_a, mult_c);
  load (mult_b, zwei_z);
  mult();  // = x^4·Z2
  load (work_a, mult_c);
  load (mult_a, zwei_u);
  load (mult_b, zwei_x);
  mult();  // = U·X2
  load (add_a, mult_c);
  load (add_b, work_a);
  add ();  // = Y2
  load (zwei_y, add_c);
}

// ber. test_y aus zwei_z³·punkt1_y
void berechne_test_y ()
{
```

```
    load (mult_a, z2_quadrat);
    load (mult_b, zwei_z);
    mult();  // = z3
    load (mult_a, mult_c);
    load (mult_b, punkt1_y);
    mult();  // = y
    load (test_y, mult_c);
  }

  // Algorithmus zum Finden des Punktes
  void verdopple ()
  {
    // kreiere einen Nullvektor in nullv[]
    for (int zv = bit; zv >= 0; --zv)
    {
      nullv[zv] = 0;
    }

    while (punkt_gefunden == 0)
    {
      berechne_z();
      berechne_x();
      berechne_u();
      berechne_y();
      berechne_z2_quadrat();
      berechne_test_x();
      ++ verdopplungen;

      load (comp_a, zwei_x);
      load (comp_b, test_x);
      compare ();
      if (comp_f == 1)  // berechne y
      {
        berechne_test_y ();
        load (comp_a, zwei_y);
        load (comp_b, test_y);
        compare ();
        if (comp_f == 1)  // Punkt gefunden
        {
          punkt_gefunden = 1;
          break;
        }
      }
      cout <<"\rVerdopplung Nr.: "<< verdopplungen;
      load (eins_x, zwei_x);
      load (eins_y, zwei_y);
      load (eins_z, zwei_z);
    }
    cout <<"\n\nDer Punkt wurde nach "<< verdopplungen <<" Verdopplungen gefunden!\n\n";
  }  /*verdopple */


  void main()
  {
    verdopple();
  }
```

B.2 VHDL-Modell des Punktverdoppelungs-FPGA

B.2.1 Der zentrale Entwurf

```
-- zentraler Entwurf zum Angriff auf ein ECC mittels Punktverdoppelung
--
-- filename: verdoppler_26_3Bit.vhd
-- Bitbreite: 3
-- gesuchte Punkte: 1
-- version 2.6
-- Markus Boettger  - 06/2001


Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;


entity verdoppler_26 is
  generic ( len : integer range 4 to 64 := 4); -- m+1
  port    ( nrst, clk   : in  std_logic;
            input       : in     unsigned (5 downto 0);
            verd_anzahl : buffer unsigned (len-1 downto 0);
            pkt_gefunden: out std_logic
          );
end entity verdoppler_26;


architecture behaviour of verdoppler_26 is
  type state_type is (start_state, input_state, eins_state, zwei_state, drei_state, vier_state,
fuenf_state,
        sechs_state, sieben_state, acht_state, neun_state, fertig_state);
  signal state          : state_type;
  signal input_bits     : integer;
  signal gespkt_proj    : unsigned (len-1 downto 0);
  signal Z2_hoch_x      : unsigned (len-1 downto 0);
  signal Multein_1_1    : unsigned (len-1 downto 0);
  signal Multein_1_2    : unsigned (len-1 downto 0);
  signal Multein_2_1    : unsigned (len-1 downto 0);
  signal Multein_2_2    : unsigned (len-1 downto 0);
  signal Multaus_1      : unsigned (len-1 downto 0);
  signal Multaus_2      : unsigned (len-1 downto 0);
  signal Addein_1_1     : unsigned (len-1 downto 0);
  signal Addein_1_2     : unsigned (len-1 downto 0);
  signal Addaus_1       : unsigned (len-1 downto 0);
  signal reg_a          : unsigned (len-1 downto 0);
  signal reg_b          : unsigned (len-1 downto 0);
  signal reg_c          : unsigned (len-1 downto 0);
  signal reg_d          : unsigned (len-1 downto 0);
  signal ppoly          : unsigned (len-1 downto 0);
  signal c_wert         : unsigned (len-1 downto 0);
  signal gespkt_1_x_aff : unsigned (len-1 downto 0);
  signal gespkt_1_y_aff : unsigned (len-1 downto 0);

  -- Funktion zur Beschreibung des Addierers; er addiert zwei Vektoren modulo 2

  function Add (a, b: unsigned (len-1 downto 0))
    return unsigned is
  begin
    return(a xor b);
  end function Add;

  -- Funktion zur Beschreibung eines Multiplizierers;
  --  er multipliziert zwei Vektoren modulo 2

  function Mult (a, b, pp: unsigned (len-1 downto 0))
    return unsigned is
    variable c : unsigned (len   downto 0);
    variable t1: unsigned (len-1 downto 0);
  begin
    c  := (others => '0');
    t1 := (others => '0');
    for i in len-2 downto 0 loop
```

```
      for j in 0 to len-2 loop
        t1(j) := c(j) xor (a(i) and b(j)) xor (c(len-1) and pp(j));
      end loop;
      c(len downto 1) := t1;
    end loop;
    return c(len downto 1);
  end function Mult;


begin

  -- Beschreibung des endlichen Automaten zur Steuerung der Zustandsübergänge
  -- und der Multiplexer an den Ausgängen der Funktionseinheiten

  FSM: process (nrst, clk)
  begin
    if (nrst = '0') then
      state <= start_state;
    elsif (clk'event and clk = '1') then
    case state is

    when start_state =>
      state         <= input_state;
      verd_anzahl   <= to_unsigned(1,len);
      reg_c         <= to_unsigned(1,len);
      input_bits    <= len-1;
      pkt_gefunden  <= '0';

    when input_state =>
      if (input_bits = 0) then
        state       <= eins_state;
      else
        input_bits  <= input_bits-1;
        state       <= input_state;
      end if;
      reg_a(input_bits)  <= input(5);
      reg_b(input_bits)  <= input(4);
      ppoly (input_bits) <= input(3);
      reg_d(input_bits)  <= input(2);
      c_wert(input_bits) <= input(2);
      gespkt_1_x_aff(input_bits) <= input(1);
      gespkt_1_y_aff(input_bits) <= input(0);

    when eins_state =>
      state         <= zwei_state;
      reg_b         <= Multaus_1;
      reg_c         <= Multaus_2;

    when zwei_state =>
      state         <= drei_state;
      reg_c         <= Multaus_1;
      reg_d         <= Multaus_2;

    when drei_state =>
      state         <= vier_state;
      reg_a         <= Multaus_1;
      Z2_hoch_x     <= Multaus_2;
      reg_d         <= Addaus_1;

    when vier_state =>
      state         <= fuenf_state;
      reg_d         <= Multaus_1;
      gespkt_proj   <= Multaus_2;
      reg_b         <= Addaus_1;

    when fuenf_state =>
      state         <= sechs_state;
      reg_a         <= Multaus_1;
      reg_b         <= Addaus_1;
      reg_d         <= Multaus_2;

    when sechs_state =>
      state         <= sieben_state;
      reg_a         <= Multaus_1;
```

```
    reg_b         <= Multaus_2;

  when sieben_state =>
    state         <= eins_state;
    reg_b         <= Addaus_1;
    Z2_hoch_x     <= Multaus_1;
    if (gespkt_proj = reg_d) then
      state       <= acht_state;
    else state    <= eins_state;
      verd_anzahl <= verd_anzahl + 1;
      reg_a       <= reg_d;
      reg_d       <= c_wert;
    end if;

  when acht_state =>
    state         <= neun_state;
    gespkt_proj   <= Multaus_1;

  when neun_state =>
    state         <= eins_state;
    if (gespkt_proj = reg_b) then
      state       <= fertig_state;
      pkt_gefunden <= '1';
    else state    <= eins_state;
      verd_anzahl <= verd_anzahl + 1;
      reg_a       <= reg_d;
      reg_d       <= c_wert;
    end if;

  when fertig_state => null;

  end case;
  end if;
end process FSM;

-- Beschreibung der Ein- und Ausgänge der drei Funktionseinheiten

Multaus_1 <= Mult (Multein_1_1, Multein_1_2, ppoly);

Multaus_2 <= Mult (Multein_2_1, Multein_2_2, ppoly);

Addaus_1  <= Add  (Addein_1_1, Addein_1_2);

-- Beschreibung der Multiplexer an den Eingängen der
      --   Funktionseinheiten

with state select
  Multein_1_1 <= reg_b           when eins_state,   -- Y1 * Z1
                 reg_a           when zwei_state,   -- Z2 = Z1**2 * X1
                 reg_a           when drei_state,   -- X1**2
                 reg_d           when vier_state,   -- ((X1+c*Z1**2)**2)
                 reg_a           when fuenf_state,  -- X1**4
                 reg_c           when sechs_state,  -- X1**4 * Z2
                 reg_c           when sieben_state, -- Z2**3 = Z2**2 * Z2
                 Z2_hoch_x       when acht_state,   -- Z2**2 * gespkt_1_y_aff
                 (others => '-') when others;

with state select
  Multein_1_2 <= reg_c           when eins_state,
                 reg_c           when zwei_state,
                 reg_a           when drei_state,
                 reg_d           when vier_state,
                 reg_a           when fuenf_state,
                 reg_a           when sechs_state,
                 Z2_hoch_x       when sieben_state,
                 gespkt_1_y_aff  when acht_state,
                 (others => '-') when others;

with state select
  Multein_2_1 <= reg_c           when eins_state,   -- Z1**2
                 reg_c           when zwei_state,   -- Z1**2 * c
                 reg_c           when drei_state,   -- Z2**2 = Z2*Z2
                 Z2_hoch_x       when vier_state,   -- Z2**2 * gespkt_1_x_aff
                 reg_d           when fuenf_state,  -- ((X1+c*Z1**2)**4)
```

```
                   reg_b            when sechs_state,  -- U * X2
                   (others => '-') when others;

     with state select
       Multein_2_2 <= reg_c            when eins_state,
                      reg_d            when zwei_state,
                      reg_c            when drei_state,
                      gespkt_1_x_aff   when vier_state,
                      reg_d            when fuenf_state,
                      reg_d            when sechs_state,
                      (others => '-') when others;

     with state select
       Addein_1_1  <= reg_a            when drei_state,   -- (X1 + c*Z1**2)
                      reg_b            when vier_state,   -- Z2 + Y1 * Z1
                      reg_a            when fuenf_state,  -- U = Z2+X1**2+Y1*Z1
                      reg_a            when sieben_state, -- Y2 = X1**4*Z2+U*X2
                      (others => '-') when others;

     with state select
       Addein_1_2  <= reg_d            when drei_state,
                      reg_c            when vier_state,
                      reg_b            when fuenf_state,
                      reg_b            when sieben_state,
                      (others => '-') when others;

   end architecture behaviour;
```

B.2.2 Die Testumgebung

```
-- Testumgebung fuer den zentralen Entwurf zum Angriff auf ein
-- ECC mittels Punktverdoppelung
--
-- filename: tb_verdoppler_26_3Bit.vhd
-- Bitbreite: 3
-- gesuchte Punkte: 1
-- version 2.6
-- Markus Boettger - 06/2001


Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;


entity tb_verdoppler_26 is
end;

architecture tb of tb_verdoppler_26 is

  component verdoppler_26 is
    port( nrst, clk   : in  std_logic;
          input       : in     unsigned (5 downto 0);
          verd_anzahl : buffer unsigned (3 downto 0);
          pkt_gefunden: out std_logic
        );
  end component;

  signal nrst, clk   : std_logic;
  signal input       : unsigned (5 downto 0) := "000000";
  signal verd_anzahl : unsigned (3 downto 0);
  signal pkt_gefunden: std_logic;

begin

  my_verdoppler_26: verdoppler_26
    port map ( nrst         => nrst,
               clk          => clk,
               input        => input,
               verd_anzahl  => verd_anzahl,
               pkt_gefunden => pkt_gefunden
             );

  -- 16 MHz = 2 * 31.25 ns
  clk_pro: process
  begin
    clk <= '0';
    wait for 31 ns;
    clk <= '1';
    wait for 31 ns;
  end process;

  nrst_pro: process
  begin
    nrst <= '0';
    wait for 207 ns;
    nrst <= '1';
    wait;
  end process;

  bsp_pro: process
  constant zeit : time := 62 ns;   -- dem Takt anpassen
  begin
    input <= "000000"; wait for 279 ns;

    -- start_x, start_y, ppoly, c_wert, ges_1_x, ges_1_y

    input <= "0" & "0" & "1" & "0" & "0" & "0" ; wait for zeit; -- 3   MSB
    input <= "0" & "1" & "0" & "0" & "1" & "0" ; wait for zeit;
    input <= "1" & "0" & "1" & "0" & "0" & "1" ; wait for zeit;
    input <= "0" & "0" & "1" & "1" & "1" & "0" ; wait for zeit; -- 0   LSB
```

```
    input <= "000000" ; wait for zeit;

    wait;
  end process;

end tb;

configuration cfg_tb_verdoppler_26 of tb_verdoppler_26 is
  for tb
  end for;
end;
```

C Inhalt der CD

Diplomarbeit:

Komplexitätsabschätzung von hardwareakzelerierten Attacken auf ECC-Kryptoverfahren,
Böttger M., Diplomarbeit Universität Hamburg, Januar 2002, da-boettger.ps, da-boettger.pdf

Seminarvortrag:

Seminarvortrag zur Diplomarbeit
Komplexitätsabschätzung von hardwareakzelerierten Attacken auf ECC-Kryptoverfahren,
Böttger M., November 2001, da-boettger-seminarvortrag.pdf

Algorithmen:

C++ - Modell, q_verdopplepunkte-16-v.c.pdf
VHDL - Modell, verdoppler_26_3Bit.vhd.pdf
Testumgebung zum VHDL - Modell, tb_verdoppler_26_3bit.vhd.pdf

Literatur:

Altera Flex 10K FPGA Data Sheet,
Altera Corp., 2001, dsf10k.pdf

Altera Flex 10KE FPGA Data Sheet,
Altera Corp., 2001, dsf10ke.pdf

ANSI X9.62-1998 ECDSA - Working Draft,
ANSI, 1998, x9-62-09-20-98.pdf

ANSI X9.63-199x Key Agreement and Key Transport usind ECC - Working Draft,
ANSI, 1999, x9-63-01-08-99.pdf

Untersuchung von EC für die Tauglichkeit von Hardwareakzeleration von Kryptoverfahren,
Bohnsack F., Diplomarbeit Universität Hamburg, Juni 1997, da-bohnsack.ps

Improving the Parallelized Pollard Lambda Search On Binary Anomalous Curves,
Galland R., Lambert R., Vanstone S., Certicom Corp. Ontario, April 1998, lambda.pdf

Counting Points on Hyperelliptic Curves over Finite Fields,
Gaudry P., Harley R., Ecole Polytechnique Le Chesnay, antsIV.ps

Konzeption, Evaluierung und Implementation eines Akzelerators für Elliptische Kurven,
Gorr S., Diplomarbeit Universität Hamburg, Februar 2000, da-gorr.pdf

IEEE P1373 (Draft Version 7) Standard Specs for Public Key Crypt. -
Annex A, B, C, D, E, F, M,
IEEE,1998-99, P1363A.pdf,P1363B.pdf,P1363C.pdf,P1363D.pdf,
P1363E.ps,P1363F.ps,P1363M.pdf

The Elliptic Curve Digital Signature Algorithm,
Johnson D., Menezes A., University of Waterloo, August 1999, ecdsa.ps

IP Security Protocols,
Kulkarni U., Indian Institute of Technology, Kanpur, April 1999, ip-security-protocols.ps

Selecting Cryptographic Key Sizes,
Lenstra A.K., Verheul E.R., preprint, November 1999, cryptosizes.pdf

FPGA Implementation of a Microcoded Elliptic Curce Cryptographic Processor,
Leung, Ma, Wong, Leong, University of Hong Kong, 2000, fpga.pdf

An Overview of Elliptic Curve Cryptography,
Lopez J., Dahab R., University of Sao Paulo, Mai 2000, overview.ps

VHDL Kurzbeschreibung,
Mäder A., Universität Hamburg, v_all.ps

Fast Multiplication on Elliptic Curves over small Fields of Characteristic Two,
Müller V., TH Darmstadt, June 1997, vmueller.jc.ps

Catching Kangaroos In Function Fields,
Stein. A, Teske. E, University of Waterloo, March 1999, catching_kangaroos.pdf

Speeding up Pollard´s Rho Method for Computing Discrete Logarithms,
Teske E., TU Darmstadt, 1998?, speeding.ps

Faster Attacks On Elliptic Curve Cryptosystems,
Wiener M.J., Zuccherato R.J., Entrust Technologies, Ottawa, April 1998, attackEC.pdf

Elliptic Curve Cryptography - Support in Entrust,
Zuccherato R.J., Entrust Corp., Mai 2000, entrust.ps

D Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, die vorliegende Arbeit selbständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Hamburg, den 14.01.2002

Markus Böttger

E Danksagung

Ich möchte mich bei Herrn Dr. habil. Reinhard Rauscher (Arbeitsbereich Technische Grundlagen der Informatik der Universität Hamburg) und Herrn Prof. Dr.-Ing. Karl Kaiser (Regionales Rechenzentrum der Universität Hamburg) für die sehr intensive und konstruktive Betreuung meiner Arbeit bedanken.
Besonderer Dank gilt meiner Familie, sowie Frau Astrid Fehnemann für die Unterstützung während der Erstellung dieser Arbeit.
Weiterhin möchte ich allen Mitarbeitern des Arbeitsbereiches Technische Grundlagen der Informatik der Universität Hamburg für die unermüdliche Unterstützung bei der Verwendung der am Arbeitsbereich vorhandenen EDV, sowie der diversen Entwicklungssoftwares danken.
Für diverse Durchsichten und konstruktive Anregungen danke ich meinen Komillitonen Ole Blaurock, Peter Grotrian, Sven Hahlbrock und Danijel Sebalj.
Bedanken möchte ich mich auch bei allen, die mich während der Erstellung der Arbeit mit wichtigen Informationen versorgten, so z.B. Herrn Dipl.-Ing. Ralf Suitner der Firma Sasco, der mich über die aktuellen Preise der Altera-FPGA unterrichtete.

F Index

Addition Punkte
- affin, 13
- projektiv, 18

Alberti, Leon Battista, 1
Artin'sche Vermutung, 35

Baby Step/Giant Step - Methode, 38
Bluetooth-Standard, 5
Brute-Force-Angriff, 30

Charakteristik, Def., 6

Datenflußgraph, 51
diskreter Logarithmus, 20
DLP, 20

Erweiterungskörper, 6

Falltürfunktionen, 20

Galois, Evariste, 7
Galois-Feld, 7
Granularität
- fein, 65
- grob, 65

Gruppe, Def., 6

Hasse-Theorem, 32

Inverser Punkt, 7, 16

Körper, Def., 6
Kangaroo-Fangmethode, 37
Komplexität, 43

Low-Security-Systeme, 27

Mips-Jahr, Def., 28
Multiplexer, 53

Ordnung
- Kurve, Def., 13
- Punkt, Def., 32

Pollard-Lambda-Methode, 37
Pollard-Rho-Methode, 36
Primitivwurzel, 35
Primkörper, Def., 6
Primzahl
- Fermat'sche, 35
- Mersenn'sche, 35

projektiver Punkt, 17

Realtime, 27

supersinguläre Kurve, 13

Testumgebung, 49
Trap-Door-Functions, 20
Trust-Center, 36

Unendlichkeitspunkt, 11
Unterkörper, Def., 6

Verdoppelung Punkte
- affin, 14
- projektiv, 33

Weierstrass-Gleichung, 10

Ziehen ohne Zurücklegen, 43