TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Integrierte Schaltungen und Systeme
Prof. Dr.-Ing. Sorin Huss

Studienarbeit

Integration des Kryptoprozessors ECP in den Java-basierten FlexiProvider

Nima Barraci (Matr.-Nr.: 861236) und Sven Becker (Matr.-Nr.: 838036)

Betreuer: Dipl.-Inform. Markus Ernst und Dipl.-Math. Birgit Henhapl

31. August 2003

Zusicherung

Zur Erstellung der vorliegenden Studienarbeit wurden nur die in der Arbeit angegebenen Hilfsmittel verwendet.

Nima Barraci

Sven Becker

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

In der heutigen Zeit gehören E-Commerce-Anwendungen, Online-Banking und Austausch sensibler Daten über öffentliche, ungesicherte Kommunikationswege zum Alltag. Im Zuge dieser Entwicklung gewannen Public-Key Kryptoverfahren einen immer höheren Stellenwert.

Im Bereich der Public-Key-Kryptographie hat sich das RSA-Verfahren [1] als Standard etabliert. Als Alternative dazu haben Elliptische Kurven (EC) in jüngster Zeit viel Aufmerksamkeit erregt. Ihr Einsatz wurde von N. Koblitz [2] und V. Miller [3] erstmals vorgeschlagen und sie sind die bisher einzig standardisierte Alternative zu RSA.

Die Sicherheit von RSA beruht auf dem Faktorisierungsproblem, also der Schwierigkeit, große ganze Zahlen in ihre Primfaktoren zu zerlegen. Da aber sowohl die Rechenleistung von Computern, als auch die verwendeten Berechnungsverfahren immer leistungsfähiger werden, muss die Länge der verwendeten RSA-Schlüssel stetig steigen, um eine ausreichende Sicherheit zu gewährleisten. EC-Kryptographie dagegen basiert darauf, dass es bisher keinen bekannten subexponentiellen Algorithmus gibt, der das zugrunde liegende Diskreter-Logarithmus-Problem über elliptischen Kurven löst.

Da die gesamte EC-Arithmetik sehr komplex und aufwendig zu berechnen ist, kann mit EC-Kryptographie - trotz kurzer Schlüssellänge von 160 Bit - das gleiche Sicherheitsniveau erreicht wie bei RSA mit 1024 Bit [4]. Gerade in mobilen Anwendungen, kann mit EC-Kryptographie eine hohe Sicherheit auf Geräten mit sehr begrenzten Ressourcen realisiert werden.

Sowohl RSA als auch EC sind keine beweisbar sicheren Kryptographie-Verfahren und beruhen allein auf der Tatsache, dass bisher keine Lösungen für die zugrundeliegenden mathematischen Probleme gefunden wurden. Daher ist es besonders wichtig, verschiede Alternativen zu haben, denn es ist jederzeit möglich, dass eines der Verfahren gebrochen wird. Da das Faktorisierungsproblem und das Diskreter-Logarithmus-Problem völlig verschieden sind, bleibt eines der beiden Kryptographie-Verfahren sicher, auch wenn das andere gebrochen wird.

1.1 Grundlagen der Arbeit

Nachfolgend werden die Vorarbeiten, auf denen diese Arbeit basiert, vorgestellt und kurz beschrieben.

1.1.1 FlexiProvider

Am Lehrstuhl für Theoretische Informatik der TU Darmstadt wurde der FlexiProvider [19] entwickelt. Hierbei handelt es sich um einen Java-basierten Krypto-Provider, der unter anderem Digitale Signaturen auf Basis elliptischer Kurven (ECDSA) anbietet (Details siehe [5]).

Der FlexiProvider entspricht der von der JCA[1] vorgegebenen Schnittstelle und kann daher sehr einfach von beliebigen Dritt-Anwendungen zur Schlüsselpaargenerierung, Signaturerzeugung und Verifikation verwendet werden. Der FlexiProvider wird ab Kapitel 6 detailliert beschrieben.

1.1.2 Kryptoprozessor ECP

Zur hocheffizienten Berechnung von Operationen der EC-Arithmetik (Details siehe Kapitel 2 und 3) wurde am Fachgebiet Integrierte Schaltungen und Systeme der TU Darmstadt der Kryptoprozesser ECP entwickelt. Der ECP ist auf einer PCI-Karte realisiert und kann als Coprozessor in einem PC eingesetzt werden um den Prozessor des Hostsystems zu entlasten. Er basiert auf einer flexiblen FPGA-Plattform, die zur Laufzeit rekonfiguriert werden kann.

Diese Hardware-Implementierung der EC-Arithmetik arbeitet um ein Vielfaches schneller als die hochoptimierten Software-Varianten im FlexiProvider, ist jedoch deutlich unhandlicher in der Anwendung. Um die Funktionalität des ECP zu nutzen, müssen u.a. die Operanden in eine Hardware-verständliche Binärform transformiert und über den PCI-Bus übertragen werden. Danach kann die Berechnung gestartet und auf einen Interrupt gewartet werden, der die Fertigstellung der Operation signalisiert. Nach der Berechnung müssen die Ergebnisdaten wieder über den PCI-Bus gelesen und in eine Software-kompatible Darstellung übersetzt werden.

1.2 Ziele der Arbeit

Ziel dieser Arbeit war zum einen, die Funktionalität des ECP über eine einfach benutzbare Schnittstelle für beliebige Anwendungen zur Verfügung zu stellen. Über diese Schnittstelle sollte dann im zweiten Schritt der FlexiProvider angebunden werden, so dass arithmetische Operationen auf EC-Ebene auf den ECP ausgelagert und dort berechnet werden können. Durch die Berechnung der aufwendigen EC-Operationen in Hardware soll schließlich eine deutliche Performanzsteigerung für auf dem FlexiProvider basierende EC-Kryptosysteme ermöglicht werden. Basierend auf den in Kapitel 3 angegebenen Leistungsdaten des ECP sollten Performanzsteigerungen von deutlich über 1000% gegenüber der Java-basierten Softwareimplementierung realisierbar sein.

[1] JCA: Java Cryptography Architecture [8]. Die standardisierte Schnittstelle, über die in Java kryptographische Anwendungen über einheitliche Protokolle kommunizieren können.

Kapitel 2

Elliptische Kurven Arithmetik

Für kryptographische Anwendungen sind elliptische Kurven (EC) über endlichen Körpern grundsätzlich geeignet. In der Praxis werden die endlichen Körper *GF(p)* und *GF(2^n)* als Basis für EC-basierte Kryptoverfahren verwendet. Mit *GF(p)* werden *Primkörper* bezeichnet: Für die Primzahl p bilden die ganzen Zahlen $\{0, \ldots, p-1\}$ zusammen mit der Addition und Multiplikation modulo p einen Körper. Der Körper *GF(2^n)* besteht aus 2^n Elementen und repräsentiert den n-dimensionalen Vektorraum über der Menge $\{0, 1\}$. Im Rahmen dieser Arbeit werden ausschließlich elliptische Kurven über *GF(p)* betrachtet, d.h. die arithmetischen Operationen auf Körperebene sind Addition, Subtraktion, Multiplikation und Inversion modulo p. Diese werden auch als *FF-Add*, *FF-Sub*, *FF-Mult* und *FF-Invert* bezeichnet.

Eine elliptische Kurve E über *GF(p)* ist definiert als die kubische Gleichung

$$E: \quad y^2 = x^3 + ax + b \tag{2.1}$$

mit $x, y, a, b \in$ *GF(p)* und $4a^3 + 27b^2 \not\equiv 0 \mod p$. Die Punkte auf der elliptischen Kurve E werden durch die Menge aller Lösungen $\{(x, y) \mid y^2 = x^3 + ax + b\}$ repräsentiert.

Nimmt man zu der Menge der Punkte noch einen Punkt hinzu, den mit $\mathcal{O}$ bezeichneten Punkt im Unendlichen, erhält man eine additive Gruppe mit $\mathcal{O}$ als neutralem Element. Eine Gruppe ist eine Menge, die sich durch die folgenden drei Eigenschaften - hier veranschaulicht an elliptischen Kurven - auszeichnet:

1. Das Ergebnis einer Operation in der Gruppe ist wieder ein Element der Gruppe: liegen P und Q auf der Kurve E, dann liegt auch $R = P + Q$ auf E.
2. Es existiert ein neutrales Element, der Punkt im Unendlichen $\mathcal{O}$ mit $P + \mathcal{O} = \mathcal{O} + P = P$.
3. Jedes Element besitzt ein dazu inverses Element mit $P + (-P) = \mathcal{O}$, mit $P = (x, y)$ und $-P = (x, -y)$.

In Abbildung 2.1 ist eine elliptische Kurve über dem Körper der reellen Zahlen exemplarisch dargestellt. Hier lässt sich die Addition zweier Punkte graphisch veranschaulichen. Um die

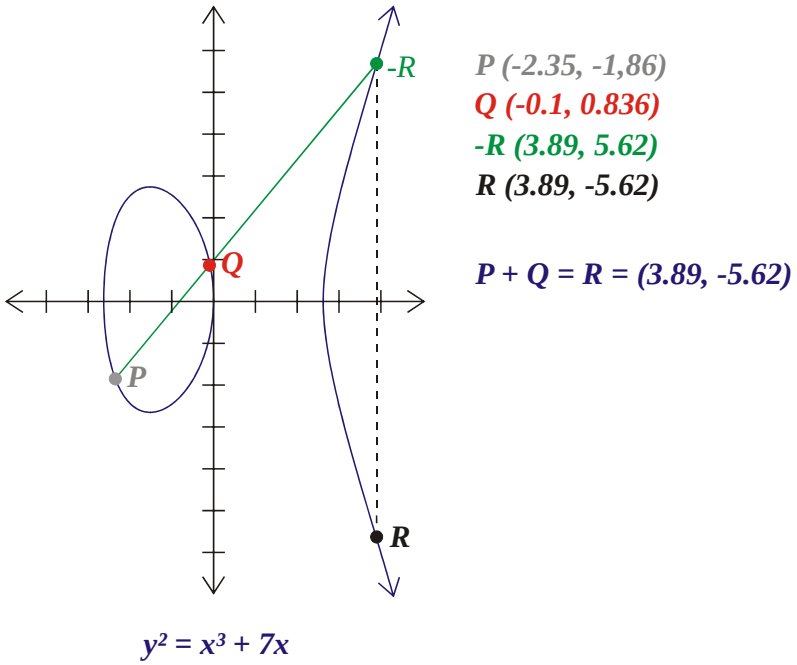

Abbildung 2.1: Beispiel einer elliptischen Kurve zur Visualisierung der Punktaddition.

Punkte P und Q zu addieren, legt man eine Gerade durch sie hindurch. Diese schneidet die Kurve an einem weiteren Punkt $-R$. Diesen spiegelt man an der X-Achse und erhält den Ergebnispunkt $R = P + Q$.

2.1 Affine Koordinatendarstellung

Für eine elliptische Kurve E über *GF(p)* und die Punkte $P = (x_P,\ y_P)$ und $Q = (x_Q,\ y_Q)$ auf E berechnet sich die Addition $R = (x_R,\ y_R) = P + Q$ wie folgt:

$$x_R = \lambda^2 - x_1 - x_2$$
$$y_R = \lambda(x_R - x_Q) - y_Q$$

mit

$x_P \neq x_Q$:	$\lambda =$	$\frac{y_Q - y_P}{x_Q - x_P}$
$x_P = x_Q$ und $y_P = y_Q$:	$\lambda =$	$\frac{3x_P^2 + a}{2y_P}$
$x_P = x_Q$ und $y_P = -y_Q$:	$P + Q = \mathcal{O}$	

Bei der Berechnung der Addition muss grundsätzlich zwischen den Fällen $P \neq Q$ und $P = Q$ unterschieden werden. Im Falle $P \neq Q$ wird die Operation als Punktaddition bzw. *EC-Add*, für $P = Q$ als Punktverdopplung oder *EC-Double* bezeichnet.

Sowohl *EC-Add* als auch *EC-Double* sind also keine trivialen Operationen, sie benötigen jeweils mehrere Berechnungen im zugrundeliegenden Körper *GF(p)*. Nachfolgend sind die Aufwands-

abschätzungen angegeben, wobei A für den Aufwand von *FF-Add* oder *FF-Sub*, M für den Aufwand von *FF-Mult* und I für den Aufwand von *FF-Invert* steht.

Aufwand für *EC-Add*: 6A + 3M + 1I
Aufwand für *EC-Double*: 8A + 4M + 1I

Die affine Koordinatendarstellung hat den Nachteil, dass bei jeder Operation auf EC-Ebene eine aufwändige Inversion in *GF(p)* erforderlich ist. Dies kann durch die Verwendung von projektiven Koordinaten umgangen werden.

2.2 Projektive Koordinatendarstellung

Die in diesem Abschnitt beschriebene *Jacobische Projektive Koordinatendarstellung* entspricht der im Kryptoprozessor ECP (siehe Kapitel 3) verwendeten Darstellung, so dass bei der Implementierung der ECP-Schnittstelle (siehe Kapitel 4) auf zusätzliche Koordinatentransformationen verzichtet werden kann.

Die Ersetzung von $x = \frac{X}{Z^2}$ und $y = \frac{Y}{Z^3}$ in Gleichung 2.1 führt zu der folgenden projektiven Kurvengleichung

$$E: \quad Y^2 = X^3 + aXZ^4 + bZ^6 \,. \tag{2.2}$$

Sei $P = (X_P,\, Y_P,\, Z_P)$ und $Q = (X_Q,\, Y_Q,\, Z_Q)$ auf E.

ADDITION: Für $Q = (-P)$ ist $R = P + Q = \mathcal{O}$.
Für $P \neq Q$ und $Q \neq (-P)$ ist $R = (X_R,\, Y_R,\, Z_R) = P + Q$ mit

$$\begin{aligned} X_R &= -H^3 - 2U_1H^2 + r^2 \\ Y_R &= -S1H^3 + r(U_1H^2 - X_R) \\ Z_R &= Z_PZ_QH \end{aligned}$$

und

$$\begin{aligned} U_1 &= X_PZ_Q^2 \\ S_1 &= Y_PZ_Q^3 \\ U_2 &= X_QZ_P^2 \\ S_2 &= Y_QZ_P^3 \\ H &= U_2 - U_1 \\ r &= S_2 - S_1 \end{aligned}$$

Aufwand für *EC-Add*: $7A + 16M$ wenn P und Q projektiv.
$7A + 11M$ wenn Q affin.

VERDOPPLUNG: Für $P = \mathcal{O}$ ist $R = 2 \cdot P = \mathcal{O}$
Für $P \neq \mathcal{O}$ ist $R = (X_R,\ Y_R,\ Z_R) = 2 \cdot P$ mit

$$\begin{aligned} X_R &= T \\ Y_R &= -8Y_P^4 + M(S - T) \\ Z_R &= 2Y_P Z_P \end{aligned}$$

und

$$\begin{aligned} S &= 4X_P Y_P^2 \\ M &= 3X_P^2 + aZ_P^4 \\ T &= -2S + M^2 \end{aligned}$$

Aufwand für *EC-Double*: $13A + 10M$ wenn P projektiv.
$13A + 6M$ wenn P affin.

Durch die Verwendung projektiver Koordinaten erhöht sich zwar die Anzahl an Additionen und Multiplikationen zur Berechnung von *EC-Add* und *EC-Double*. Auf Inversionen kann jedoch komplett verzichtet werden, so dass sich insgesamt ein deutlicher Performanzgewinn einstellt.

2.3 Punktmultiplikation

Wie bereits ausgeführt, bildet die Menge aller Punkte auf einer elliptischen Kurve eine additive Gruppe. Basierend auf der Punktaddition ist die Punktmultiplikation (*EC-Mult*) definiert als die wiederholte Anwendung der Punktaddition ausgehend von einem Basispunkt P. Es gilt

$$R = k \cdot P = \underbrace{P + P + \ \ldots\ + P + P}_{k\text{-mal}},$$

wenn der Basispunkt P k-mal zu sich selbst addiert wird. Man nennt k den diskreten Logarithmus von R zur Basis P. Sind P und R gegeben, so ist es schwierig den diskreten Logarithmus k zu berechnen. Das Diskreter-Logarithmus-Problem (DLP) über der Punktegruppe elliptischer Kurven bildet die Grundlage für die Sicherheit aller EC Kryptoverfahren.

Zur Berechnung von *EC-Mult* können Verfahren zur modularen Exponentiation in abgewandelter Form[1] eingesetzt werden. Die Laufzeit der unterschiedlichen Algorithmen hängt im Wesentlichen von der Art und der Anzahl vorberechneter Punkte ab. Grundsätzlich gilt: Je mehr Punkte vorberechnet werden, umso schneller ist die Berechnung der eigentlichen Punktmultiplikation. Verfahren mit aufwändiger Vorberechnung sind allerdings nur für Anwendungen sinnvoll, in denen viele *EC-Mult* Operation mit dem selben Basispunkt P zu berechnen sind.

Das nachfolgend dargestellte *Sliding-Window* Exponentiationsverfahren stellt einen guten Kompromiss zwischen der Anzahl vorberechneter Punkte und der Laufzeit des Algorithmus dar und ist für effiziente Software- und Hardware-Implementierungen bestens geeignet.

[1] Anstelle von Multiplikationen werden *EC-Add* und *EC-Double* Operationen durchgeführt.

Sei $k = \sum_{i=0,\ldots n-1} 2^i b_i$ mit $b_i \in \{0,\ 1\}$ der Exponent und P ein Punkt auf der Kurve E über *GF(p)*. Zu berechnen ist $R = k \cdot P$.

VORBERECHNUNG:

$P_1 \leftarrow P$
$P_2 \leftarrow 2 \cdot P$
`for` $i = 1$ `to` $2^{w-1} - 1$
  $P_{2i+1} \leftarrow P_{2i-1} + P_2$

HAUPTRECHNUNG:

$R \leftarrow \mathcal{O}$
$i \leftarrow n - 1$
`while` $i \geq 0$
  `if` $k_i = 0$ `then`
    $R \leftarrow 2 \cdot R$
    $i \leftarrow i - 1$
  `else`
    finde den längsten Bitstring $k_i k_{i-1} \ldots k_l$ mit $i - l + 1 \leq w$ und $k_l = 1$
    $R \leftarrow 2^{i-l+1} \cdot R + P_{(k_i k_{i-1} \ldots k_l)_b}$
    $i \leftarrow l - 1$
`return` R

Aufwand der Vorberechnung: 1 *EC-Double* und $2^{w-1} - 1$ *EC-Add*.
Aufwand der Hauptrechnung: $n - 1$ *EC-Double* und $\frac{n}{w+1}$ *EC-Add*.

Kapitel 3

Kryptoprozessor (ECP)

Hardware-seitig basiert diese Arbeit auf dem am ISS entwickelten Kryptoprozessor ECP der auf einer rekonfigurierbaren HW-Plattform implementiert wurde. Dem ECP liegt ein generisches HW-Modell zugrunde, das es erlaubt, abhängig von der geforderten Bitbreite der Operanden spezifische Ausprägungen des ECP zu nutzen. Für die in Kapitel 5 und Kapitel 8 detaillierten Laufzeitmessungen standen jeweils bitbreitenoptimale Ausprägungen des ECP zur Verfügung. Diese sind in Tabelle 3.2 explizit aufgeführt und Charakterisiert. Nachfolgend wird die dem ECP zugrundeliegende HW-Plattform, die Architektur des ECP und dessen Schnittstelle inkl. Befehlssatz vorgestellt.

3.1 Alpha Data ADM-XPL Plattform

Als Hardware-Plattform für den ECP dient eine ADM-XPL PMC-Karte [13] der Firma Alpha Data Parallel Systems Ltd. (siehe Abbildung 3.1). Bei dieser ADM-XPL Karte (im Folgenden

Abbildung 3.1: ADM-XPL FPGA-Karte

auch als FPGA-Karte bezeichnet) handelt es sich um eine universell einsetzbare, auf der *Virtex-II Pro* FPGA-Technologie [18] der Firma Xilinx basierende, rekonfigurierbare HW-Plattform. Neben dem zentralen *Field Programmable Gate Array* (FPGA), verfügt die Karte über einen PCI-Controller, statisches RAM und einen programmierbaren Taktgenerator, so dass komplette HW Subsysteme auf der Karte implementiert werden können.

Die ADM-XPL Karte kann mit unterschiedlichen *Virtex-II Pro* FPGA-Bausteinen bestückt werden. Die im Rahmen dieser Arbeit verwendete Karte ist mit einem XC2VP20 FPGA-Baustein bestückt, was einer Komplexität von ungefähr 2 Mio. Gatteräquivalenten entspricht. Zusätzlich verfügt dieser Baustein über zwei eingebettete Power-PC Kerne, die von der aktuellen ECP-Architektur jedoch nicht verwendet werden.

Die physikalische Systemintegration erfolgt über eine ADC-PMC-64 PCI-Adapterkarte [14]. Diese Adapterkarte dient als Trägerplatine für die eigentliche FPGA-Karte und wird in einem PCI-Steckplatz des Hostsystems montiert. Zur Kommunikation mit der FPGA-Karte steht Treibersoftware (u.a. für Microsoft Windows 2000) und ein *Software Development Kit (SDK)* (u.a. für Microsoft Visual C++) mit Konfigurations- und Kommunikationsfunktionen zur Verfügung [15].

3.2 Architektur des ECP

In Abbildung 3.2 ist die generische Architektur des ECP dargestellt. Diese soll dem Leser einen groben Überblick der HW-Architektur vermitteln, deshalb wurde auf die explizite Darstellung von Takt- und Kontrollsignalen bewusst verzichtet.

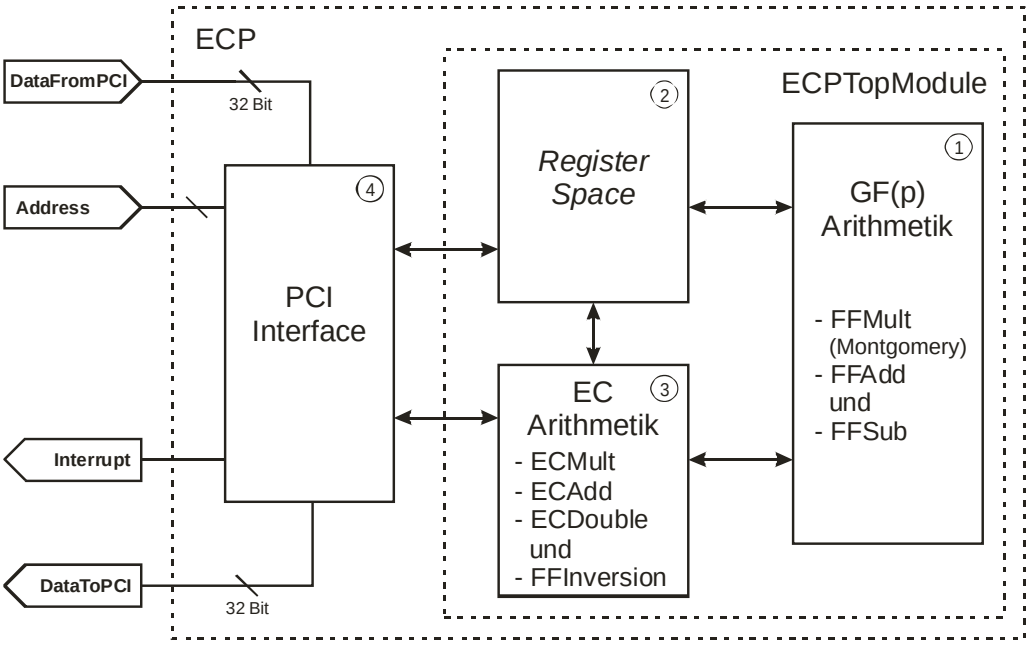

Abbildung 3.2: Architektur des ECP

Die Architektur des ECP besteht im Wesentlichen aus folgenden funktionalen Blöcken:

1. ***GF(p)* Arithmetik**
   Hierin sind drei der arithmetischen Basisoperationen auf Körperebene implementiert (*FF-Add*, *FF-Sub* und *FF-Mult*).

2. **Register Space**
   8192 Byte SRAM-Speicher für Kurvenparameter, Punktkoordinaten und temporäre Variablen. Auf den *Register Space* kann Software-seitig in Worten zu je 32 Bit lesend und schreibend zugegriffen werden.

3. **EC-Arithmetik**
   Hierin sind die Operationen auf EC-Ebene (*EC-Add*, *EC-Double* und *EC-Mult*) sowie die Inversion auf Körperebene (*FF-Invert*) implementiert.

4. **PCI-Interface**
   Hierin ist die *ECP-Command* Schnittstelle implementiert und es erfolgt die Anbindung des ECP an das vorgegebene physikalische *Interface* der FPGA-Karte.

3.3 Schnittstelle und Befehlssatz des ECP

Über die *ECP-Command* Schnittstelle wird der ECP mit Instruktionen (*OpCodes*) beschickt. In Abhängigkeit dieser *OpCodes* werden die entsprechenden Operationen im ECP ausgeführt. Das (Software-seitige) Setzten von *OpCodes* geschieht über das an der *Offset-Adresse* 10000_h in den Speicherbereich der Software eingeblendete 32 Bit *ECP-Command* Register. Der Aufbau des Registers ist in Abbildung 3.3 illustriert.

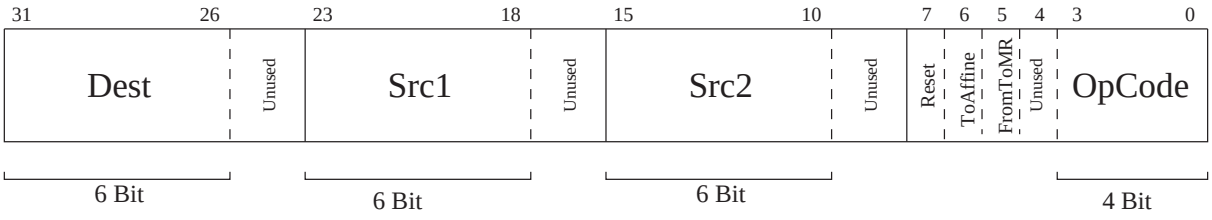

Abbildung 3.3: Aufbau des *ECP-Command* Registers

In dieser Darstellung bezeichnet `Dest` die Zieladresse der jeweiligen Operation im *Register Space*. In den Feldern `Src1` und `Src2` werden analog dazu die Adressen der Eingangs-Operanden angegeben. Bei den Bitpositionen 7 - 5 im *ECP-Command* Register handelt es sich um *One-Hot* codierte Steuerinformationen, die zeitgleich mit dem *OpCode* und den Adressen anzulegen sind:

- Bit 7: `Reset`
  Unabhängig vom sonstigen Inhalt des *ECP-Command* Registers wird für `Reset=1` das *Chip-globale* Reset-Signal gesetzt.

- Bit 6: `ToAffine`
  Über das *Flag* `ToAffine` wird im Falle der Punktmultiplikation *EC-Mult* die Konvertierung des Ergebnispunkts in die affine Koordinatendarstellung an- (`ToAffine=1`) oder abgeschaltet (`ToAffine=0`).

- Bit 5: `FromToMR`
  Über das *Flag* `FromToMR` wird die Konvertierung der Operanden in die/aus der Montgomery-Darstellung gesteuert. Dies ist notwendig, da die Implementierung von *FF-Mult* im ECP auf der Montgomery-Multiplikation beruht und die Operanden in der zugehörigen Darstellung vorliegen müssen (siehe [6, S. 600f]).

  Im Falle von *EC-InitSLW* werden für `FromToMR=1` die Punkt-Koordinaten zusätzlich in die Montgomery-Darstellung transferiert. Im Falle von *EC-Mult* werden für `FromToMR=1` die Koordinaten des Ergebnispunktes zusätzlich in die Standard-Darstellung umgerechnet.

Der eigentliche *OpCode* ist in den niederwertigsten 4 Bit des *ECP-Command* Registers codiert. Der Befehlssatz des ECP, inkl. der Binär-Codierung der *OpCodes*, ist in Tabelle 3.1 zusammengefasst. Aus Tabelle 3.1 geht ebenfalls hervor, bei welchen ECP Befehlen, welche Address-Argumente und welche der obigen *Flags* berücksichtigt werden.

Tabelle 3.1: Befehlssatz (*OpCodes*) des ECP

OpCode	Codierung	Dest	Src1	Src2	ToAffine	FromToMR	Reset
Reset							√
Nop	0000_b						
FF-Init	0001_b						
FF-Mult	0010_b	√	√	√			
FF-Add	0011_b	√	√	√			
FF-Sub	0100_b	√	√	√			
EC-Double	0101_b	√	√				
EC-Add	0110_b	√	√	√			
EC-Mult	1000_b	√	√		√	√	
EC-InitCurve	1001_b						
EC-InitPoint	1010_b	√	√				
EC-InitSLW	1011_b		√			√	
FF-Invert	1100_b	√	√				
EC-CopyPoint	1101_b	√	√				

Nach der Beendigung einer Operation wird vom ECP ein Interrupt ausgelöst der Software-seitig abgefangen werden kann.

Nachfolgend wird die Funktionalität der einzelnen *OpCodes* aus Tabelle 3.1 kurz erläutert:

1. ***Reset***
   Setzen des *Chip-globalen* Reset-Signals.

2. ***Nop***
   No Operation.

3. ***FF-Init***
   Durchführung der Initialisierungssequenz für die *GF(p)*-Arithmetik des ECP.

4. ***FF-Mult***
   Berechnung der Multiplikation $(x * y) \mod p$ auf dem ECP. Für die Speicherung der Operanden werden die Adressen `dest`, `src1` und `src2` verwendet.

5. ***FF-Add***
   Berechnung der Addition $(x + y) \mod p$ auf dem ECP. Für die Speicherung der Operanden werden die Adressen `dest`, `src1` und `src2` verwendet.

6. ***FF-Sub***
   Berechnung der Subtraktion $(x - y) \mod p$ auf dem ECP. Für die Speicherung der Operanden werden die Adressen `dest`, `src1` und `src2` verwendet.

7. ***EC-Double***
   Berechnung der Operation *EC-Double* im ECP. Für die Speicherung der Punktkoordinaten werden die Adressen `dest` und `src1` verwendet.

8. ***EC-Add***
   Berechnung der Operation *EC-Add* im ECP. Für die Speicherung der Punktkoordinaten werden die Adressen `dest`, `src1` und `src2` verwendet.

9. ***EC-Mult***
   Berechnung der Punktmultiplikation $k \cdot P$ auf dem ECP. Der Exponent k wird von Adresse `src1` gelesen. Der Ergebnispunkt wird an Adresse `dest` geschrieben.

10. ***EC-InitCurve***
    Konvertierung der Kurvenparameter a und b in die Montgomery-Darstellung.

11. ***EC-InitPoint***
    Konvertierung der Koordinaten des Basispunktes P in die Montgomery-Darstellung.

12. ***EC-InitSLW***
    Durchführung der Vorberechnungen zur Initialisierung des *Sliding-Window* Exponentiationsverfahrens.

13. ***FF-Invert***
    Berechnung der Inversion $a^{-1} \mod p$ auf dem ECP. Für die Speicherung der Operanden werden die Adressen `dest` und `src1` verwendet.

14. ***FF-CopyPoint***
    Kopieren der Punktkoordinaten von der Adresse `src1` nach `dest` im *Register Space* des ECP.

3.4 Ausprägungen des ECP

Im Rahmen der Arbeit wurden unterschiedliche Ausprägungen des ECP verwendet, d.h. Implementierungen des ECP mit unterschiedlichen maximalen Bitbreiten der Operanden. Diese sind in Tabelle 3.2 zusammengefasst und mit einigen Eckdaten charakterisiert.

Tabelle 3.2: Unterschiedliche Ausprägungen des ECP

Bitbreite des ECP	Auslastung des FPGA	# Taktzyklen pro *EC-Mult*	Laufzeit bei 66 MHz
144	38 %	89.128	1,35 ms
160	40 %	103.135	1,56 ms
192	45 %	134.388	2,04 ms
208	48 %	169.373	2,57 ms
272	56 %	274.586	4,16 ms
304	59 %	322.490	4,89 ms
400	77 %	558.427	8,46 ms
512	90 %	856.010	12,97 ms

Alle in Tabelle 3.2 aufgeführten ECP Implementierungen arbeiten mit einer maximalen Taktfrequenz von 66 MHz. Für jedes ECP Hardware-Design sind in Tabelle 3.2 folgende Informationen aufgeführt:

1. Die maximale Bitbreite der Operanden.
2. Die Auslastung der Logikressourcen des verwendeten XC2VP20 FPGA-Bausteins, also eine Angabe über die Größe bzw. Komplexität der jeweiligen ECP Implementierung.
3. Die benötigte Anzahl an Taktzyklen zur Berechnung einer *EC-Mult* Operation. Hier liegt die Annahme zugrunde, dass die Bitbreite n der Operanden voll ausgeschöpft wird und das das *Hamming-Gewicht* des Exponenten k genau $\frac{n}{2}$ beträgt.
4. Die reale Laufzeit einer *EC-Mult* Operation auf der Grundlage der angegebenen Taktzyklen und bei Taktung des ECP mit exakt 66 MHz.

Die Interpretation von Tabelle 3.2 führt zu folgender Schlussfolgerung: Je größer die Bitbreite des ECP ist, umso komplexer ist das jeweilige Hardware-Design und umso länger dauert eine einzelne *EC-Mult* Operation. Da die *EC-Mult* Operation die Kernoperation im Bereich EC-basierter Kryptosysteme darstellt, wurde auf die Charakterisierung weiterer ECP-*OpCodes* an dieser Stelle bewusst verzichtet.

Kapitel 4

ECP-Schnittstelle

Die Software-Schnittstelle zum ECP wurde in Form einer DLL[1] implementiert. Diese Schnittstelle bietet dem Benutzer eine umfangreiche Funktionsbibliothek zur Durchführung von Körperoperationen (*GF(p)*-Arithmetik) und Operationen auf EC-Ebene (*EC-Mult, EC-Add* und *EC-Double*).

Die DLL stellt zum einen eine Software-Implementierung dieser Funktionen zur Verfügung und bietet weiterhin die Möglichkeit, die Berechnungen in Hardware im ECP durchzuführen. Die Hardware-Beschleunigung ist dabei völlig transparent gekapselt. Wenn keine ECP Hardware-Unterstützung im System vorhanden ist, werden die Berechnungen automatisch in Software durchgeführt. Ist eine FPGA-Karte mit ECP Funktionalität im System installiert, dann werden sämtliche Berechnung auf die ECP Hardware ausgelagert.

4.1 Details zur ECP Integration

Die für diese Arbeit zur Verfügung stehenden ECP Designs wurden bereits in Abschnitt 3.4 für unterschiedliche Bitbreiten aufgeführt. Die Software-Implementierung der EC-Arithmetik ist bezüglich der Bitbreite der Operanden nicht grundsätzlich beschränkt, jedoch konnte durch die Verwendung dedizierter ECP Ausprägungen mit unterschiedlichen Bitbreiten eine enorme Steigerung der Performanz erzielt werden (siehe Laufzeitmessungen in Kapitel 5).

Da alle ECP Designs abwärtskompatibel sind, können auf einem für maximal 512 Bit Operanden ausgelegten HW-Design auch alle Berechnungen mit kleineren Operanden durchgeführt werden. Um jedoch die maximale Leistung des ECP auszuschöpfen, wird bei der Initialisierung die maximale Operandenlänge festgelegt. Von dieser Bitbreite ausgehend wird dann automatisch ein ECP-Design ausgewählt, dessen Bitbreite mindestens so groß ist wie die angegebene, jedoch nicht größer als unbedingt erforderlich.

[1] DLL: Dynamic Link Library. Ein Feature des Windows-Betriebssystems, mit dem ausführbare Routinen (die im Allgemeinen einer bestimmten Funktion oder einer Gruppe von Funktionen dienen) einzeln als Dateien mit der Dateinamenerweiterung .dll gespeichert werden können. Diese Routinen werden nur geladen, wenn sie vom Programm benötigt werden, das sie aufruft.

4.1.1 Initialisierung des ECP

Die Initialisierung der ECP-Schnittstelle erfolgt über die Funktion `EC_Bitwidth`. Beim Aufruf dieser Funktion wird versucht, den ECP zu initialisieren. Wenn ECP-Hardware im System grundsätzlich zur Verfügung steht, dann werden zuerst die benötigten Ressourcen reserviert. Danach wird nach einer passenden ECP-Konfiguration für die entsprechende Bitbreite gesucht. Eine ECP-Konfiguration ist immer an eine Bitbreite gebunden und umfasst die Datei mit dem Konfigurations-Bitstream für das FPGA und die Taktfrequenz mit der dieses ECP-Design betrieben werden kann. Sollte keine solche Konfiguration für die aktuelle Bitbreite existieren, dann wird die Konfiguration der nächstgrößeren Bitbreite verwendet. Der zugehörige Konfigurations-Bitstream wird dann mit Hilfe der Funktionen des Alpha-Data-SDK [15] auf das FPGA übertragen und die Frequenz des Taktgenerators auf der FPGA-Karte wird entsprechend eingestellt. Danach wird ein ECP-Reset-Befehl abgesendet, um den ECP in seinen Startzustand zu versetzen.

Wenn diese Initialisierungsprozedur fehlerfrei durchgeführt werden konnte, wird der Hardware-Status auf den Wert `HWSTAT_READY` gesetzt. Das bedeutet, dass nachfolgende EC-Berechnungen hardwarebeschleunigt im ECP durchgeführt werden können. Tritt ein Fehler in diesem Prozess auf, wird dieser über den Hardware-Status signalisiert. In diesem Fall steht die Hardwarebeschleunigung für nachfolgende Berechnungen nicht zur Verfügung. Stattdessen werden die Berechnungen mit Hilfe der Software-Implementierung durchgeführt.

Durch dieses Konzept konnte eine hohe Flexibilität der ECP-Schnittstelle erreicht werden. EC-Operationen bis zu der vom ECP unterstützten Bitbreite werden in Hardware berechnet und für größere Bitbreiten werden die Berechnungen in Software durchgeführt. Diese HW/SW Aufteilung wird komplett von der ECP-Schnittstelle realisiert und erfolgt aus Sicht des Benutzers vollkommen transparent. Diese Transparenz wurde durch die funktional äquivalente Implementierung der gesamten ECP-Funktionalität innerhalb der ECP-Schnittstelle realisiert.

4.1.2 Berechnungen im ECP

Bei Berechnungen mit Hardware-Unterstützung müssen zuerst die Operanden (auch als *GF(p)*-Elemente bezeichnet) zum ECP übertragen werden. Dazu werden sie nacheinander in 32 Bit große Blöcke unterteilt und über den PCI-Bus zum ECP übertragen. Dort werden sie in speziell dafür vorgesehenen Registern gespeichert. Beim Schreiben oder Lesen von *GF(p)*-Elementen in bzw. aus dem ECP sind nur minimale Umformatierungen nötig, da die Software-interne Zahlendarstellung dem im ECP verwendeten Format sehr ähnlich ist.

Danach wird je nach durchzuführender Berechnung der entsprechende *OpCode* kombiniert mit den Adressen der Operanden- und Ergebnisregister an eine vorgegebene Adresse im ECP geschrieben. Die Berechnung wird gestartet, indem im Anschluss an die selbe Adresse eine `0` geschrieben wird.

Während der ECP die Berechnung durchführt, geht der Thread, der die Berechnung ausgelöst hat, in den Wartezustand über. Er erzeugt somit für die CPU des Hostsystems keinerlei Rechenlast mehr, so dass die CPU des Hostsystems während der Berechnung im ECP für andere Threads und Prozesse zur Verfügung steht.

Das Ende einer Berechnung im ECP wird durch einen Hardware-Interrupt angezeigt. Daraufhin weckt das Betriebssystem den pausierten Thread auf, so dass dieser die Ergebnisdaten aus dem ECP lesen kann. Dabei werden die Ergebnisdaten wiederum in Blöcken zu je 32 Bit gelesen und zu *GF(p)*-Elementen zusammengesetzt.

4.2 Software-Implementierung der Arithmetik

Die Langzahl-Berechnungen in der Software-Implementierung basieren auf NTL [17]. Dabei handelt es sich um eine Bibliothek, die eine Vielzahl mathematischer Funktionen für C-Programme zur Verfügung stellt. Der verwendete Datenentyp `ZZ` ist vergleichbar mit der Klasse `BigInteger` aus Java. Damit können Berechnungen mit beliebig großen ganzen Zahlen durchgeführt werden, ohne dass typbedingte Begrenzungen beachtet werden müssen.

Zur Berechnung der Punktmultiplikation *EC-Mult* wurde das *Sliding-Window*-Verfahren implementiert, wobei die *Window-Size* über einen Parameter im Quellcode eingestellt werden kann (siehe Laufzeitmessungen in Kapitel 5, Tabelle 5.4).

4.3 Kurzbeschreibung der Funktionen

Im Folgenden sind die Funktionen der ECP-Schnittstelle separat aufgeführt. Für die detaillierte Beschreibung der einzelnen Funktionen mit den zugehörigen Signaturen und Rückgabewerten wird auf Anhang A verwiesen.

4.3.1 Initialisierung

Diese Funktionen werden zur Initialisierung der DLL benötigt. Hierin werden alle zur Berechnung benötigten Daten definiert.

EC_Bitwidth
: Setzt die maximale Bitbreite der Operanden.

EC_LoadPrime
: Setzt die Primzahl, über der der endliche Körper *GF(p)* definiert wird.

EC_LoadPoint, EC_LoadPointAffine
: Setzt den Basispunkt, mit dem die nachfolgenden EC-Mult Berechnungen durchgeführt werden.

EC_LoadCurve
: Setzt die Kurvenparameter, die die elliptische Kurve über *GF(p)* definieren.

EC_Init
: Kombiniert die Initialisierungsroutinen EC_LoadPrime, EC_LoadCurve und EC_LoadPoint.

EC_SetHardwareSupport
: Schaltet die Hardware-Unterstützung explizit ein oder aus.

EC_SupportBitwidth
: Setzt die maximale Bitbreite, mit der die Supportfunktionen arbeiten.

4.3.2 *GF(p)* -Arithmetik

In diesem Abschnitt sind alle Funktionen zusammengefaßt, die sich auf ein einzelnes *GF(p)*-Element beziehen.

EC_Compare
: Vergleicht zwei *GF(p)*-Elemente und gibt an, welches von ihnen größer ist.

EC_GetHamming
: Ermittelt das Hamminggewicht eines *GF(p)*-Elements, das heißt, die Anzahl der Bits mit Wert 1 in der Binärdarstellung.

EC_Random
: Erzeugt ein zufälliges *GF(p)*-Element.

EC_RandomHamming
: Erzeugt ein zufälliges *GF(p)*-Element mit einem bestimmten Hamminggewicht.

EC_CopyValue
: Kopiert den Wert eines *GF(p)*-Element in ein anderes.

EC_CompBinEqual
: Vergleicht zwei *GF(p)*-Elemente auf binäre Gleichheit.

4.3.3 EC-Arithmetik

Hier sind alle Funktionen zur Arithmetik auf EC-Ebene zusammengefaßt. Die Funktionen erwarten Punkte als Parameter und geben zumeist auch solche als Ergebnis zurück.

EC_ConvToAffine
: Konvertiert einen Punkt von projektiver in affine Darstellung.

EC_Double, EC_DoubleAffine
: Addiert einen Punkt auf sich selbst.

EC_Add, EC_AddAffine
: Addiert zwei unterschiedliche Punkte aufeinander.

EC_Negate, EC_NegateAffine
: Negiert einen Punkt.

EC_Mult, EC_MultAffine
: Multipliziert den Basispunkt mit einem skalaren Wert.

EC_MultPoint, EC_MultPointAffine
: Multipliziert einen beliebigen Punkt mit einem skalaren Wert.

EC_IsOnCurve, EC_IsOnCurveAffine
: Überprüft, ob ein Punkt auf der elliptischen Kurve liegt.

EC_ComparePoints, EC_ComparePointsAffine
: Vergleicht zwei Punkte.

EC_CopyPoint, EC_CopyPointAffine
: Kopiert die Werte eines Punktes in einen anderen.

4.3.4 Konvertierung

Die Konvertierungsroutinen dienen der Umwandlung der Zahlendarstellungen.

EC_ConvBinToHexstr
: Binär → Hexadezimal

EC_ConvBinToDecstr
: Binär → Dezimal

EC_ConvBinToBinstr
: Binär → Binäre Stringdarstellung (1 Byte pro Bit, 0 oder 1)

EC_ConvHexstrToBin
: Hexadezimal → Binär

EC_ConvDecstrToBin
: Dezimal → Binär

4.3.5 Statusabfragen

In diesem Abschnitt sind alle Funktionen zusammengefaßt, die Informationen über die DLL selbst, oder über Berechnungsdaten liefern.

EC_GetBuildInfo
: Gibt einen String zurück, der Compilerinformationen zur DLL enthält.

EC_GetHardwareConfiguration
: Ermittelt aktuelle Hardware-Konfiguration, also der Name des aktuell geladenen Bitstreams und die eingestellte Taktfrequenz

EC_GetHardwareSupport
: Gibt den aktuellen Status der Hardware-Unterstützung zurück.

EC_GetBytewidth
: Gibt an, wieviele Bytes zur Darstellung eines *GF(p)*-Elements benötigt werden.

EC_GetBitwidth
: Gibt an, wieviele Bits zur Darstellung eines *GF(p)*-Elements benötigt werden.

EC_GetBitwidthHW
: Gibt an, mit welcher internen Bitbreite die Hardware initialisiert wurde. Aufgrund der Abwärtskompatibilität kann diese größer sein als die extern eingestellte Bitbreite.

EC_GetSupportBytewidth
: Gibt an, wieviele Bytes zur Darstellung eines *GF(p)*-Elements benötigt werden.

EC_GetSupportBitwidth
: Gibt an, wieviele Bits zur Darstellung eines *GF(p)*-Elements benötigt werden.

4.3.6 Protokollierung

Die DLL kann alle Operationen in einer Log-Datei speichern. Dadurch wird allerdings die Performance deutlich beeinträchtigt, da mitunter eine erhebliche Menge Protokollausgaben auf die Festplatte geschrieben werden. Die Protokollierung erleichtert allerdings die Fehlersuche bei der Entwicklung neuer Anwendungen.

EC_LogStart
: Startet die Protokollierung.

EC_LogStop
: Beendet die Protokollierung.

EC_LogGetAutologging
: Gibt den Status des automatischen Loggings zurück.

EC_LogGetAutologgingFile
: Gibt den Namen der Datei, in die automatisch protokolliert wird, zurück.

EC_LogSetAutologging
: Schaltet automatisches Logging ein oder aus.

EC_LogSetAutologgingFile
: Gibt den Namen der Datei an, in die automatisch protokolliert wird.

4.3.7 Konfiguration

Mit diesen Funktionen wird die DLL konfiguriert. Hiermit wird definiert, bei welcher Bitbreite welches Hardware-Design auf die Karte geladen werden soll und wie hoch die Taktfrequenz ist.

Außerdem werden diese Funktionen verwendet, um bei der Initialisierung mit einer bestimmten Bitbreite passende Konfigurationsdaten zu ermitteln.

EC_GetNextStoredBitwidth
: Gibt die nächstgrößere Bitbreite zurück, für die eine Konfiguration existiert.

EC_GetStoredBitwidth
: Gibt die Konfigurationsdaten für eine bestimmte Bitbreite zurück.

EC_GetCompatibleBitwidth
: Ermittelt eine Bitbreite, für die eine Konfiguration existiert und die kompatibel zu der von außen gesetzten Bitbreite ist.

EC_SetStoredBitwidth
: Speichert Konfigurationsdaten für eine Bitbreite.

EC_DeleteStoredBitwidth
: Löscht alle Konfigurationsdaten zu einer Bitbreite.

EC_GetFeature
: Ermittelt, ob ein bestimmtes Feature in der Konfiguration verfügbar ist.

EC_SetFeature
: Speichert, ob ein bestimmtes Feature in der Konfiguration verfügbar ist.

Kapitel 5

Laufzeiten - ECP-Schnittstelle

Die Laufzeitmessungen auf Basis der ECP-Schnittstelle wurden mit Hilfe einer in Visual-Basic erstellten Test-Applikation durchgeführt. Diese Anwendung verwendet die ECP-Schnittstelle und führt darüber verschiedene arithmetische Operationen durch. Außerdem können alle Konfigurationen der Schnittstelle mit dieser Anwendung durchgeführt werden.

Sie erlaubt es, umfangreiche Testreihen mit verschiedenen Bitbreiten automatisiert ablaufen zu lassen. Die Test-Applikation ist in Abbildung 5.1, die dazugehörige Konfigurationsoberfläche in Abbildung 5.2 dargestellt.

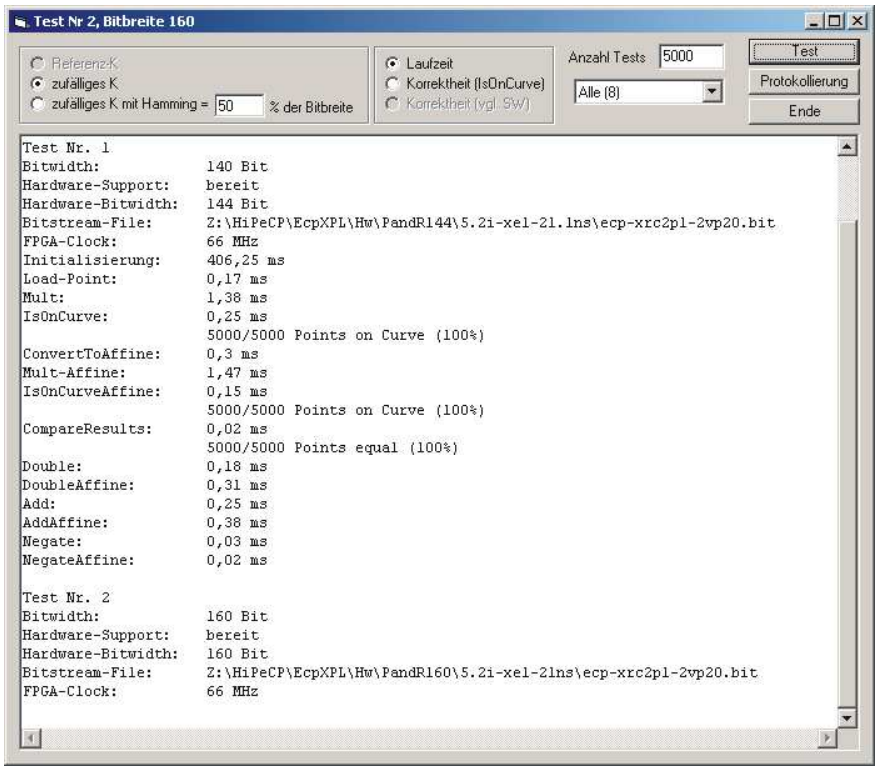

Abbildung 5.1: Laufzeitmessungen der ECP-Schnittstelle

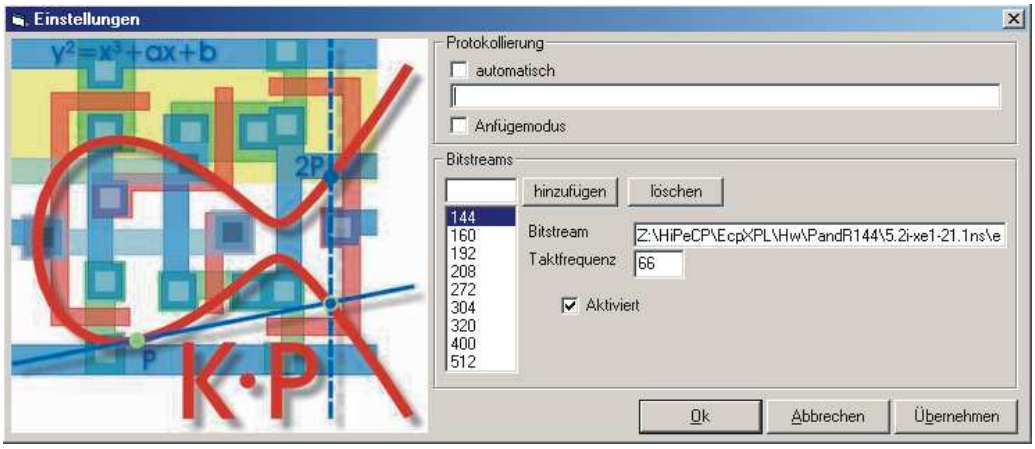

Abbildung 5.2: Konfiguration der ECP-Schnittstelle

Alle Testreihen wurden auf folgendem Testsystem durchgeführt:

- Intel Xeon 1,8 GHz
- 1 GByte RAM
- Betriebssystem: Microsoft Windows 2000 Professional

Sämtliche, in den nachfolgenden Tabellen angegebenen Laufzeiten wurden in Echtzeit auf diesem Testsystem gemessen. Bei allen Testläufen wurden für jede Bitbreite n jeweils 1000 Berechnungen mit unterschiedlichen Operanden durchgeführt. Das durchschnittliche Hamming-Gewicht lag dabei jeweils bei $\frac{n}{2}$.

5.1 Arithmetik in Software

Zuerst wurden alle Berechnungen mit Hilfe der Software-Implementierung in der ECP-Schnittstelle durchgeführt. Die dabei ermittelten Zeiten sind in Tabelle 5.1 zusammengefaßt.

Tabelle 5.1: Laufzeiten in Software

Bitbreite	*EC-Mult*	*EC-MultAffine*	*ConvToAffine*	*EC-Double*	*EC-Add*
140	14,38 ms	16,31 ms	1,62 ms	0,12 ms	0,25 ms
160	19,81 ms	22,50 ms	2,19 ms	0,12 ms	0,25 ms
192	28,88 ms	32,88 ms	3,19 ms	0,25 ms	0,31 ms
197	29,75 ms	33,94 ms	3,25 ms	0,19 ms	0,31 ms
272	64,00 ms	73,00 ms	6,56 ms	0,31 ms	0,50 ms
300	76,06 ms	86,81 ms	7,81 ms	0,31 ms	0,50 ms
400	177,10 ms	201,31 ms	16,94 ms	0,56 ms	0,94 ms
512	325,56 ms	353,75 ms	36,12 ms	0,69 ms	1,25 ms

Die angegebenen Zeiten beziehen sich jeweils auf eine einzige Operation des entsprechenden Typs. *EC-Mult* bezeichnet eine Punktmultiplikation, *EC-MultAffine* eine Punktmultiplikation mit anschließender Umrechnung des Ergebnispunktes in die affine Darstellung, *ConvToAffine* eine einzelne Umrechnung eines projektiven Punktes in die affine Darstellung und *EC-Double* und *EC-Add* eine Punkt-Verdopplung bzw. Punkt-Addition.

An diesen Zeiten wird deutlich, dass der Berechnungsaufwand exponentiell mit steigender Bitbreite wächst. Die aufwendigste Operation ist die Punktmultiplikation. Aus diesem Grund werden Performance-Vergleiche verschiedener Implementierungen nur anhand der Laufzeiten der *EC-Mult*-Operation gezogen.

5.2 Arithmetik in Hardware

Bei der Verwendung des ECP für die Arithmetischen Operationen kann bei Betrachtung der theoretischen Leistungsdaten aus Tabelle 3.2 von deutlich besseren Performance-Werten im Vergleich zur Software-Implementierung ausgegangen werden. Es ist allerdings nicht zu erwarten, tatsächlich die Zeiten aus Tabelle 3.2 zu erreichen, da dort noch keinerlei Kommunikations- und Verwaltungsaufwand berücksichtigt wurde.

5.2.1 Alle Berechnungen auf einem 512 Bit ECP-Design

Aufgrund der Abwärtskompatibilität der Hardware-Designs können auf einem einzigen 512 Bit ECP-Design alle hier betrachteten Berechnungen durchgeführt werden.

Die dabei ermittelten Zeiten sind in Tabelle 5.2 zusammengefaßt. Hier wird ein deutlicher Leistungsvorteil des ECP gegenüber der Software-Implementierung deutlich. Bei 140 Bit Operanden ist die ECP-Variante mit 3,61 ms etwa vier mal schneller als die Software mit 14,38 ms. Bei höheren Bitbreiten wird die Leistungsfähigkeit des ECP immer deutlicher. Bei 512 Bit wird eine *EC-Mult* Operation mit Hardware-Beschleunigung 24,8 mal schneller als in Software berechnet.

Tabelle 5.2: Laufzeiten mit 512 Bit ECP-Design

Bitbreite	*EC-Mult*	*EC-MultAffine*	*ConvToAffine*	*EC-Double*	*EC-Add*
140	3,61 ms	3,88 ms	0,49 ms	0,20 ms	0,29 ms
160	4,12 ms	4,43 ms	0,53 ms	0,20 ms	0,29 ms
192	4,93 ms	5,29 ms	0,59 ms	0,20 ms	0,29 ms
197	5,08 ms	5,43 ms	0,60 ms	0,21 ms	0,30 ms
272	6,98 ms	7,46 ms	0,74 ms	0,22 ms	0,31 ms
300	7,72 ms	8,20 ms	0,79 ms	0,22 ms	0,32 ms
400	10,25 ms	10,94 ms	0,97 ms	0,24 ms	0,33 ms
512	13,12 ms	13,96 ms	1,24 ms	0,26 ms	0,35 ms

Die tatsächliche Laufzeit für eine 512 Bit Punktmultiplikation ist nur geringfügig langsamer, als die in Tabelle 3.2 angegebene theoretische Laufzeit. Daran wird deutlich, dass der in Tabelle 3.2 nicht berücksichtigte Kommunikationsaufwand nur sehr gering ausfällt.

Die Berechnungen mit kleineren Bitbreiten laufen schneller ab, da dabei das durchschnittliche Hamming-Gewicht deutlich niedriger ist. Führende Nullen im Exponenten k werden im ECP innerhalb weniger Taktzyklen übersprungen und haben keine aufwendigen Berechnungen zur Folge.

5.2.2 Berechnung auf dedizierten ECP-Designs

Die im vorigen Abschnitt dargestellten, im Vergleich zur Software sehr guten Leistungsdaten können nochmals deutlich verbessert werden. Dazu werden anstatt einem einzigen 512 Bit ECP-Design, mehrere auf die verwendeten Bitbreiten abgestimmte Hardware-Designs verwendet. Die Ergebnisse dieser Testreihe sind in Tabelle 5.3 zusammengefasst.

Tabelle 5.3: Laufzeiten mit dedizierten HW-Designs

Bitbreite extern	*Bitbreite des ECP*	*EC-Mult*	*EC-MultAffine*	*ConvToAffine*	*EC-Double*	*EC-Add*
140	144	1,37 ms	1,45 ms	0,29 ms	0,17 ms	0,24 ms
160	160	1,62 ms	1,72 ms	0,31 ms	0,17 ms	0,24 ms
192	192	2,09 ms	2,23 ms	0,35 ms	0,17 ms	0,25 ms
197	208	2,49 ms	2,66 ms	0,38 ms	0,19 ms	0,26 ms
272	272	4,23 ms	4,53 ms	0,52 ms	0,20 ms	0,28 ms
300	304	4,91 ms	5,25 ms	0,57 ms	0,21 ms	0,29 ms
400	400	8,59 ms	9,12 ms	0,84 ms	0,23 ms	0,32 ms
512	512	13,12 ms	13,96 ms	1,24 ms	0,26 ms	0,35 ms

Es zeigt sich, dass auf diese Weise die Geschwindigkeit der Punktmultiplikation für 140 Bit-Operanden nochmals um den Faktor 2,6 gesteigert werden kann. Die in Tabelle 5.3 angegebenen Zeiten sind jeweils nur geringfügig langsamer, als die theoretischen Laufzeiten aus Tabelle 3.2. Die Berechnungen auf dem 208 Bit ECP-Design sind sogar geringfügig schneller. Das liegt in der Operandengröße begründet, die bei diesem Datensatz nur 197 Bit beträgt. Schon durch das daraus resultierende niedrigere Hamming-Gewicht bzw. durch die führenden Nullen im Exponenten sind die tatsächlich gemessenen Zeiten schneller, als die theoretisch errechneten.

5.3 Variationen des *Sliding-Window*-Verfahrens

Die Software-Implementierung des *Sliding-Window*-Verfahrens (SLW) lässt sich leicht an verschiedene Fenstergrößen anpassen. Je nach Fenstergröße ergibt sich ein stark unterschiedliches Laufzeitverhalten der Punkt-Multiplikation. Je größer das Fenster gewählt wird, umso schneller

Tabelle 5.4: Auswirkung der Fenstergröße bei SLW auf die Software-Laufzeit bei 192 Bit

Fenster-größe	*Vorberechnete Punkte*	*InitSLW*	*EC-Mult*	*opt*
D&A	-	0,05 ms	40,30 ms	-
1	0	0,17 ms	38,81 ms	-
2	1	0,38 ms	34,89 ms	-
3	3	0,73 ms	30,45 ms	-
4	7	1,45 ms	28,72 ms	1
5	15	2,92 ms	27,78 ms	2
6	31	5,83 ms	26,98 ms	3
7	63	11,69 ms	26,36 ms	5
8	127	23,38 ms	25,92 ms	9
9	255	46,78 ms	25,59 ms	16
10	511	93,47 ms	25,28 ms	29

wird die Multiplikation durchgeführt. Allerdings steigt der Aufwand der Vorberechnung erheblich an.

Die Laufzeiten in der ersten Zeile von Tabelle 5.4 sind mit dem *Double-and-Add* Algorithmus bestimmt worden, für die übrigen Zeiten wurde das SLW Verfahren verwendet. SLW ist bei einer Fenstergröße von 1 identisch mit *Double-and-Add*, doch aufgrund einer leicht verbesserten Implementierung sind die Zeiten bei Fenstergröße = 1 geringfügig besser als bei *Double-and-Add*.

Anhand von Tabelle 5.4 wird der Aufwand für größere Fenstergrößen ws deutlich. Die Anzahl vorberechneter Punkte ($2^{ws-1} - 1$) hängt exponentiell von der Fenstergröße ab. Proportional dazu steigt natürlich auch die Zeit, die die Initialisierung des SLW-Verfahrens beansprucht.

In der *opt*-Spalte ist die Anzahl der Multiplikationen angegeben, ab der diese Fenstergröße optimal wird. Das heißt, bei *opt* und mehr Multiplikationen ist die benötigte Zeit für die Initialisierung und die durchgeführten Berechnungen minimal.

Es zeigt sich, dass schon bei einer einzigen Multiplikation der Aufwand für das SLW-Verfahren mit einer Fenstergröße von 4 gerechtfertigt ist. Wenn mit einem einzelnen Basispunkt viele Multiplikationen durchgeführt werden sollen, dann lohnt sich auch der Aufwand für weit größere Fenster. Es sei an dieser Stelle jedoch erwähnt, dass es deutlich effizientere Methoden zur Punkt-Multiplikation gibt, die mit einer andersartig gestalteten Vorberechnung von Punkten arbeiten. Daher ist es in der Praxis nicht üblich, SLW-Verfahren mit Fenstergrößen signifikant größer als 4 zu verwenden.

Kapitel 6

Der FlexiProvider

Der Java-basierte FlexiProvider [19] nutzt die modulare Struktur der Java Cryptography Architecture (JCA) [8] um kryptographische Funktionen über elliptischen Kurven anderen Anwendungen zu Verfügung zu stellen. Der Plattformunabhängigkeit von Java steht gleichzeitig jedoch die - verglichen zu C - geringere Geschwindigkeit entgegen. Der FlexiProvider wurde am Lehrstuhl für Theoretische Informatik der TU Darmstadt entwickelt und bietet sowohl Verschlüsselung also auch digitale Signaturen über elliptische Kurven an. Eine Auflistung aller angebotenen Verfahren findet sich unter [10].

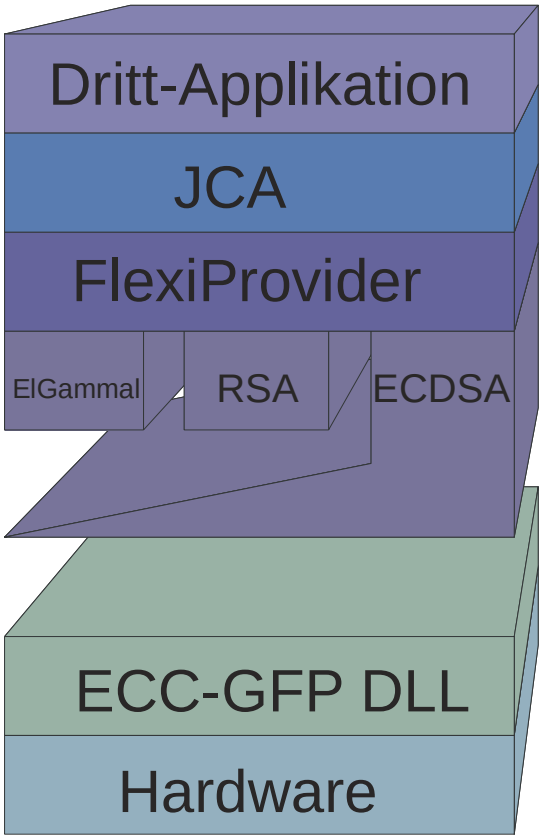

Abbildung 6.1: Eingliederung des FlexiProviders in die Gesamtstruktur

In diesem Kapitel wird auf das Konzept der JCA, welche die Basis des Providers bildet, und den *Elliptic Curve Digital Signature Algorithm* (ECDSA) eingegangen. Die arithmetischen Operationen über elliptischen Kurven, welche durch den ECP beschleunigt werden sollen, sowie eine Gegenüberstellung der Laufzeiten verschiedener Java-Implementierungen dieser Operationen im FlexiProvider werden ebenfalls behandelt.

6.1 Java Cryptography Architecture (JCA)

Der FlexiProvider gliedert sich in die so genannte Java Cryptography Architecture (JCA) ein. Sinn und Zweck der JCA ist es, Dritt-Anwendungen, die auf Java Security Funktionalitäten zugreifen, eine einheitliche und standardisierte Schnittstelle zur Verfügung zu stellen. Der FlexiProvider stellt in diesem Zusammenhang dabei das Back-End dar, d.h. die konkrete Implementierung der einzelnen kryptographischen Funktionen und Protokolle. Dritt-Anwendungen können auf verschiedene dieser *Cryptographic Service Provider* zur Laufzeit zugreifen. Dabei bleibt es dem Entwickler überlassen, welches Protokoll er über welchen Provider benutzen möchte.

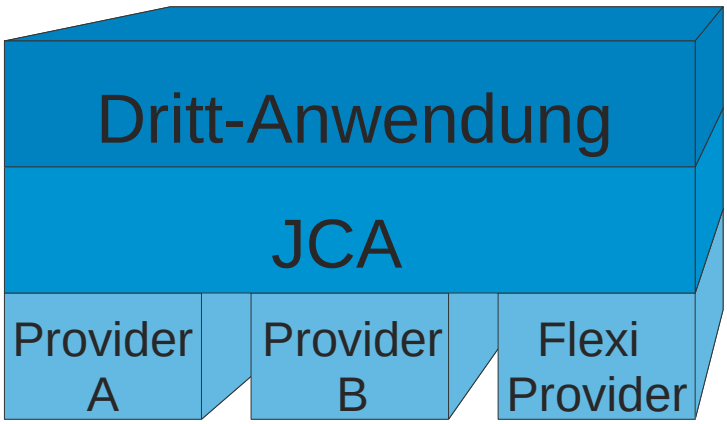

Abbildung 6.2: JCA: Einordnung der Applikationsebenen

Aufgrund der Austauschbarkeit der hinter der JCA liegenden Provider, ist die Schnittstelle zur JCA strikt definiert. Es ist so selbst zur Laufzeit möglich, zwischen einzelnen Security-Providern, die verschiedene Algorithmen auf Basis unterschiedlicher Körper anbieten können, zu wechseln. Die Funktionalität des Providers und die konkrete Implementierung der einzelnen Algorithmen im Provider ist für Dritt-Anwendungen transparent. Entwickler, die einfach nur kryptographische Operationen wie *Schlüsselpaargenerierung*, *Signaturerzeugung* und *Verifikation* benötigen, können ohne weiteres einfach über die JCA diese Funktionalitäten nutzen, ohne sich um die Implementierung der Algorithmen oder gar die dahinter liegenden mathematischen Theorien kümmern zu müssen. Diese Vielfalt und Austauschbarkeit macht die JCA zum *de-facto* Standard für die plattformunabhängige und flexible Implementierung kryptographischer Algorithmen. Abbildung 6.2 verdeutlicht die Struktur, die durch das JCA angeboten und forciert wird.

6.2 Elliptic Curve Digital Signature Algorithm (ECDSA)

Wie schon im vorangegangenen Abschnitt erwähnt, implementiert der FlexiProvider eine Vielzahl von kryptographischen Verfahren und Algorithmen. Hier wird auf ECDSA zur Berechnung digitaler Signaturen auf Basis elliptischer Kurven eingegangen. In einfachen Worten ausgedrückt, stellt ECDSA das über elliptischen Kurven arbeitende Gegenstück zu DSA dar. Es handelt sich hierbei sowohl um einen ANSI X9.62 [11] als auch um einen IEEE [12] definierten Standard.

In Kapitel 1 wurde bereits ausgeführt, dass EC-basierte Kryptoverfahren, also auch ECDSA, mit deutlich kürzeren Schlüsseln das Sicherheitsniveau von RSA erreichen. Andererseits sind die Basisoperationen der EC-Arithmetik, vor allem die Punktmultiplikation *EC-Mult* sehr Aufwendig, was sich in hohen Rechenzeiten niederschlägt. Für Details zu den Laufzeiten der einzelnen Operationen des FlexiProviders wird auf Abschnitt 6.5 verwiesen.

6.3 EC-Arithmetik

Die in Kapitel 2 bereits eingeführten arithmetischen Operationen, werden im FlexiProvider für die Berechnungen über elliptischen Kurven eingesetzt. In der Software stehen sich dabei verschiedene Implementierungsalternativen gegenüber, die in diesem Abschnitt dargestellt werden.

6.3.1 Punkt-Addition (*EC-Add*)

Bei der Punkt-Addition handelt es sich um eine relativ einfache Operation mit einem sehr guten Zeitverhalten. Wie Tabelle 6.1 zu entnehmen ist, fallen diese Zeiten - ebenso wie die Zeiten für die Negation - gegenüber der Punkt-Multiplikation kaum ins Gewicht. Für die Addition ist ein Verfahren basierend auf dem Algorithmus von *Chudnovsky und Chudnovsky* im FlexiProvider implementiert.

Dazu müssen alle Punkte im *Chudnovsky Jacobischen Koordinatensystem* vorliegen. Ein Punkt der Form $P = (X, Y, Z)$ wird dann in der Form $P = (X, Y, Z, (Z^2), (Z^3))$ bei der Erzeugung gespeichert. Dies schlägt sich dann bei dem verwendeten Verfahren dahingehend positiv nieder, dass bei der Addition zweier projektiver Punkte eine Multiplikation und eine Quadrierung im Vergleich zur klassischen Addition im *Jacobischen Koordinatensystem* eingespart werden. Demgegenüber steht der einmalig höhere Aufwand bei der Punkterstellung. In Software ist dieses Verfahren von Vorteil, da die Kosten durch den höheren Speicherverbrauch - im Gegensatz zu einer Hardware Implementierung - nicht ins Gewicht fallen. Weitere Details zu diesem Verfahren finden sich in [6].

Die realen, in Echtzeit gemessenen Laufzeiten von *EC-Add* bei unterschiedlichen Bitbreiten sind in Tabelle 6.1 zusammengefasst.

6.3.2 Punkt-Negation (*EC-Negate*)

Bei der Punkt-Negation handelt es sich um eine der einfachsten Operationen, die in nahezu konstanter Zeit rechnet. Zur Negation eines Punktes P braucht nur die Y-Koordinate negiert zu werden, sprich $P = (x, y, z) \rightarrow -P = (x, -y, z)$. Die Negation der Koordinate wird gemäß $y_{neu} = -y_{alt} \mod p$ durchgeführt.

Die für unterschiedliche Bitbreiten nur leicht variierenden, in Echtzeit gemessenen Laufzeiten von *EC-Negate* sind in Tabelle 6.1 zusammengefasst.

6.3.3 Punkt-Multiplikation (*EC-Mult*)

Die Punkt-Multiplikation, als komplexeste und teuerste Operation der EC-Arithmetik, ist im FlexiProvider auf verschiedene Weisen implementiert. Dabei ist grunsätzlich zwischen Verfahren *mit* und *ohne* Vorberechnung zur Initialisierung, zu unterschieden. Diese beiden Gruppen haben je nach Anwendungsfall gewisse Vor- bzw. Nachteile. In Abschnitt 6.4 wird auf die einzelnen Anwendungsfälle genauer eingegangen.

Man muss bei der Punkt-Multiplikation zwischen einer einfachen Multiplikation $k \cdot P$ und der Linearkombination zweier mit Skalaren zu multiplizierenden Punkten $k_1 \cdot P_1 + k_2 \cdot P_2$ unterscheiden. Im Folgenden werden für die beiden Fälle jeweils 2 der verwendeten Algorithmen (mit und ohne Vorberechnung) exemplarisch vorgestellt. Weitere Einzelheiten zur konkreten Implementierung dieser Algorithmen sowie Alternativen können unter anderem in [6] nachgeschlagen werden.

Einfache Punkt-Multiplikation: $k \cdot P$

Als Algorithmus *ohne* Vorberechnung wird im FlexiProvider die *Sliding-Window-Exponentiation* mit einer *Window-Size* von 4 eingesetzt. Die Punkte, die dieser Algorithmus benötigt, werden vor der eigentlichen Durchführung der Multiplikation berechnet. Da die Berechnung dieser Punkte innerhalb der Multplikationsmethode für den zu multiplizierenden Punkt jedesmal neu durchgeführt wird (und damit auch in die Zeitmessung fallen), handelt es sich in diesem Kontext um ein Verfahren ohne Verwendung vorberechneter Punkte.

Dabei benötigt die *Hauptrechnung* $n - 1$ *EC-Double* und $\frac{n}{w+1}$ *EC-Add* Operationen, wobei n die Länge des Schlüssels und w die Window-Size, in unserem Fall 4, bezeichnen. Die *Vorberechnung* benötigt eine *EC-Double* und $2^{w-1} - 1$ *EC-Add* Operationen.

Die realen, in Echtzeit gemessenen Laufzeiten von *EC-Mult* auf Basis der *Sliding-Window-Exponentiation* mit *Window-Size = 4*, für unterschiedliche Bitbreiten sind ebenfalls in Tabelle 6.1 zusammengefasst.

Das implementierte Verfahren *mit* Vorberechnungen arbeitet basierend auf ± 1-Ketten (siehe [6]). Hierbei benötigt die Vorberechnung $n-1$ *EC-Double* und die *Hauptrechnung* $\frac{n}{2}$ *EC-Add* Operationen. Zu beachten ist hierbei, dass deutlich mehr Punkte vorzuberechnen und abzuspeichern sind, jedoch die Hauptrechnung völlig ohne *EC-Double* Operationen auskommt. Auch hier bezeichnet n die Länge des Schlüssels.

Linearkombination: $k_1 \cdot P_1 + k_2 \cdot P_2$

Ohne Vorberechnungen arbeitet die *Mehrfache-Simultane-Exponentiation* (siehe [6]). Dieses Verfahren benötigt in der *Hauptrechnung* $\frac{3}{4}n$ *EC-Add* und $n-1$ *EC-Double* Operationen. Auch in diesem Verfahren wird in der Multiplikationsroutine eine *Vorberechnung* durchgeführt, diese besteht jedoch lediglich aus einer *EC-Add* Operation.

Das Äquivalent zur mehrfachen simultanen Exponentiation *mit* Vorberechnungen ist die w*-NAF-based Interleaving Exponentiation*. Dieses Verfahren basiert im wesentlichen auf der einfachen Exponentiation mit Non-adjacent-forms und benötigt in der *Hauptrechnung* $\frac{n\cdot(w_1+w_2+4)}{(w_1+2)\cdot(w_2+2)}$ *EC-Add* und $n-1$ *EC-Double* Operationen. Dabei bezeichnen w_1 und w_2 die *Window-Size*. Die Vorberechnung schlägt mit zwei *EC-Double* und $2^{w_1-1}+2^{w_2-1}-2$ *EC-Add* Operationen zu Buche. Für Einzelheiten sei auch hier auf [6] verwiesen.

6.4 Provider Operationen (ECDSA)

Auf Provider Ebene werden die in Abschnitt 6.3 dargestellten EC-Operationen verwendet, um die Basisfunktionalitäten Schlüsselpaargenerierung, Signaturerstellung und Signaturverifikation zu Verfügung zu stellen. Die Performanz dieser Funktionen hängt direkt von der Implementierung der zugrundeliegenden Operationen auf EC-Ebene ab.

6.4.1 Schlüsselpaargenerierung

Die Generierung von Schlüsselpaaren greift einmal auf die Multiplikation der EC-Ebene zu. Im Provider wird standardmäßig die Punktmultiplikation mit vorberechneten Punkten gewählt, um eine Beschleunigung der Schlüsselpaargenerierung zu erreichen.

Zur Erstellung eines *sicheren* Private Key/Public Key Paares wird für eine elliptische Kurve ein statistisch einmaliger und nicht vorher bestimmbarer `BigInteger` b im Wertebereich 1 bis $ord(G)-1$ ermittelt, dabei ist $ord(G)$ die Ordnung[1] des Basispunktes G. Auf Basis von b wird der *Private Key* errechnet. Zur Berechnung des *Public Key* wird eine einfache Punkt-Multiplikation $b \cdot G$ durchgeführt.

Die realen, in Echtzeit gemessenen Laufzeiten der Schlüsselpaargenerierung für unterschiedliche Bitbreiten sind in Tabelle 6.2 zusammengefasst. Zum Vergleich werden in Tabelle 6.2 auch noch die Zeiten aufgeführt, die für die Schlüsselpaargenerierung ohne Vorberechnungen benötigt werden.

6.4.2 Signaturerstellung

Bei der Signaturerstellung wird ebenfalls eine *EC-Mult* Operation durchgeführt. Somit ist die Laufzeit für die Signaturerstellung im Wesentlichen von der Laufzeit der Multiplikation ab-

[1] Die Ordnung eines Punktes ist die kleinste ganze Zahl q, für die $q \cdot P = \mathcal{O}$ gilt.

hängig. Auch hier stehen beide Verfahren, mit und ohne Vorberechnungen, zu Verfügung. Der FlexiProvider rechnet jedoch standardmäßig mit vorberechneten Punkten (siehe Tabelle 6.3), da bei der Signaturerstellung davon ausgegangen wird, dass eine Vielzahl von Signaturen mit dem gleichen Basispunkt erstellt werden.

Der Ablauf der Signaturerstellung für eine gegebene Nachricht m mit ihrem Hashwert f und einem Paar ganzer Zahlen $(c, d), 0 < c, d < r$ mit r = Primteiler der Ordnung der elliptischen Kurve, ist:

1. Erzeugung eines Einmal-Schlüsselpaares (u, V) mit $1 < u < r$ und $V = u \cdot G = (x_V, y_V)$.
2. Berechnung: $c = x_V \mod r$.
3. Berechnung: $d = u^{-1}(f + sc) \mod r$. Ist d=0: Neustart bei 1.

G ist hierbei ein Punkt auf der elliptischen Kurve, für den gilt $r \cdot G = \mathcal{O}$

Die realen, in Echtzeit gemessenen Laufzeiten der Signaturerstellung für unterschiedliche Bitbreiten sind in Tabelle 6.3 zusammengefasst. Zum Vergleich werden in Tabelle 6.3 auch noch die Zeiten aufgeführt, die für die Signaturerstellung ohne Vorberechnungen benötigt werden.

6.4.3 Signaturverifikation

Bei der Signaturverifikation wird die Linearkombination $k_1 \cdot P_1 + k_2 \cdot P_2$ auf EC-Ebene durchgeführt. Somit bestimmt sich die Laufzeit dieser Operation aus der Laufzeit von zwei *EC-Mult* Operationen und einer *EC-Add* Operation. Da jedoch, wie schon unter 6.3.3 erläutert, effizientere Verfahren zur Addition zweier mit Skalaren zu multiplizierenden Punkte existieren, lässt sich die Gesamtlaufzeit im Vergleich zur Summation der Einzellaufzeiten noch deutlich verbessern. Wie in Tabelle 6.4 dargestellt, ist selbst die Laufzeit bei der Berechnung ohne Vorberechnungen nicht einfach die Summe der Laufzeiten zweier Multiplikationen und einer Addition.

Eine Signaturverifikation läuft wie folgt ab: Für ein Paar ganzer Zahlen $(c, d), 0 < c, d < r$ mit $r =$ Primteiler der Ordnung der elliptischen Kurve

1. Liegen c und d nicht im Intervall $[1, r-1]$ bricht die Verifikation ab und gibt $false$ zurück
2. Berechnung: $h = d^{-1} \mod r, h_1 = fh \mod r$ und $h_2 = ch \mod r$,
3. Berechnung: $P = h_1 \cdot G + h_2 \cdot W = (x_P, y_P)$. Ist $P = \mathcal{O}$, gib $false$ zurück. Ist $c = x_P$, gib $true$ zurück.

Hier ist G der schon im Verfahren der *Signaturerstellung* definierte Punkt, W der *Public Key* und $\mathcal{O}$ der Punkt im Unendlichen.

6.5 Laufzeiten auf Providerebene

Die im Folgenden dargestellten Messeregebnisse wurden auf einem Intel Xeon Prozessor mit 1,8 GHz Systemtakt und 1 GB RAM gemessen. Laufzeitumgebung für den FlexiProvider, aus dem alle Messungen heraus vorgenommen wurden, ist das Java Runtime Enviornment (JRE) 1.3.1 auf einem Windows 2000 System. Alle Zeiten wurden in Echtzeit gemessen und spiegeln die reale Dauer auf einem ansonsten unbelasteten System wieder. Alle Berechnungen wurden in Software im FlexiProvider *ohne* Hardware-Beschleunigung durchgeführt.

6.5.1 EC-Operationen

Tabelle 6.1 zeigt die Laufzeiten der einzelnen Basisoperationen. Wie bereits ausgeführt, sind die Addition und Negation hoch-performante Operationen und schlagen sich in den auf Provider-Ebene gemessenen Zeiten kaum nieder.

Tabelle 6.1: Laufzeiten der EC-Basisoperationen - Java

Bitbreite	*EC-Mult*	*EC-Add*	*EC-Negate*
140	27,61 ms	0,14 ms	0,047 ms
160	33,37 ms	0,17 ms	0,032 ms
192	47,75 ms	0,19 ms	0,047 ms
197	61,00 ms	0,23 ms	0,063 ms
272	124,30 ms	0,36 ms	0,078 ms
300	162,36 ms	0,44 ms	0,078 ms
400	332,45 ms	0,66 ms	0,140 ms

In Abbildung 6.3 ist aufgetragen, wie sich die Laufzeit bei steigender Bitbreite verhält.

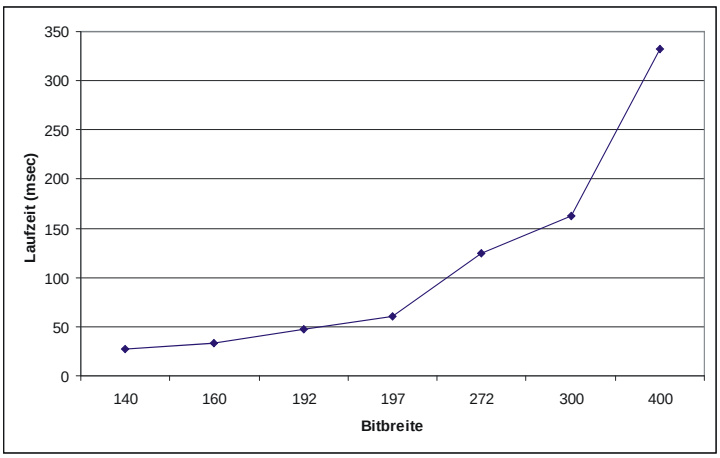

Abbildung 6.3: *EC-Mult* in Java: Laufzeit bei steigender Bitbreite

6.5.2 Provider-Operationen

Auf Ebene der Provider Operationen werden Verfahren mit Vorberechnungen, wie in Abschnitt 6.3 erläutert, eingesetzt. Um einen unmittelbaren Laufzeit-Vergleich zur ECP Hardware-Lösung zu ermöglichen, die ohne Vorberechnungen arbeitet, wurden auch Laufzeittests ohne Vorberechnungen durchgeführt und in die jeweiligen Tabellen aufgenommen.

Schlüsselpaargenerierung

Als Verfahren *mit* Vorberechnung wird der $\pm$1Ketten Algorithmus eingesetzt. Dadurch ergibt sich, wie in Tabelle 6.2 dargestellt, eine Zeitersparnis von ca. 65% gegenüber dem einfachen Verfahren *ohne* Vorberechnungen.

Tabelle 6.2: Laufzeiten der Schlüsselpaargenerierung - Java

Bitbreite	*mit Vorberechnung*	*ohne Vorberechnung*
140	10,95 ms	33,90 ms
160	11,87 ms	33,59 ms
192	18,20 ms	50,00 ms
197	23,45 ms	62,50 ms
272	48,07 ms	126,88 ms
300	62,89 ms	165,31 ms
400	130,55 ms	339,37 ms

Signaturerstellung

Auch bei der Signaturerstellung wird zur Beschleunigung das Verfahren der $\pm$1-Ketten eingesetzt.

Tabelle 6.3: Laufzeiten der Signaturerstellung - Java

Bitbreite	*mit Vorberechnung*	*ohne Vorberechnung*
140	11,42 ms	29,22 ms
160	12,21 ms	37,19 ms
192	18,54 ms	51,09 ms
197	23,87 ms	62,50 ms
272	48,67 ms	126,87 ms
300	63,89 ms	165,16 ms
400	132,15 ms	338,13 ms

Signaturverifikation

Die Signaturverifikation stellt, wie bereits ausgeführt, die komplexeste Operation auf Providerebene dar. Zur Beschleunigung wird hierbei die w-*NAF-based Interleaving Exponentiation* eingesetzt.

Tabelle 6.4: Laufzeiten der Signaturverifikation - Java

Bitbreite	*mit Vorberechnung*	*ohne Vorberechnung*
140	36,12 ms	60,79 ms
160	37,56 ms	68,91 ms
192	56,40 ms	99,22 ms
197	73,53 ms	126,57 ms
272	146,43 ms	255,00 ms
300	188,39 ms	331,56 ms
400	386,09 ms	681,56 ms

Kapitel 7

Integration des ECP in den FlexiProvider

Im Zuge der Integration des ECP in den FlexiProvider war vor allem das Problem zu lösen, wie ein sauberer Übergang aus der Java Virtual Machine (JVM) nach C zu bewerkstelligen ist. Darauf wird in Abschnitt 7.1 eingegangen. Auf die für den Wechsel zwischen der JVM und C benötigte Klasse (*ECPInterface.java*) und die korrespondierende DLL (*ECPInterface.dll*) wird in den Abschnitten 7.2 und 7.3 eingegangen. Für die detaillierte Beschreibung der einzelnen Klassen und Methoden wird auf Anhang B.2 und Anhang C.2 verwiesen.

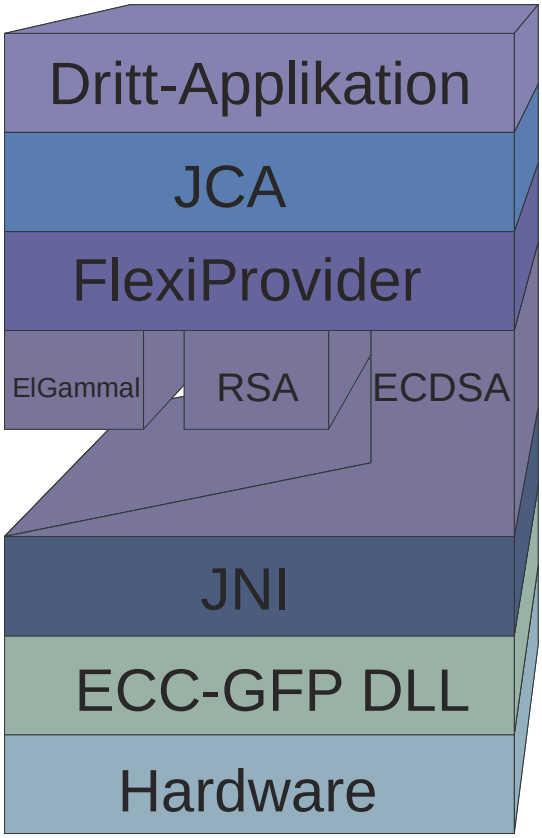

Abbildung 7.1: Eingliederung des JNI in die Gesamtstruktur

7.1 JNI - Brücke zwischen Java und C

Eigentliche Enstehungsnotwendigkeit des Java Native Interface (JNI) war es, dem Entwickler die Möglichkeit zu geben auf eine Java Architektur zu bauen, ohne ggf. bereits vorhandenen Code in Sprachen wie C oder C++ neu schreiben zu müssen. Über JNI kann native Code ausserhalb der Java Virtual Machine (JVM) ausgeführt werden. Zum JNI gehören auch einige plattformabhängige C-Header Dateien, die u.a. Zugriff auf die verschiedenen Konvertierungsfunktionen von Java Objekten bieten. Der Vorteil der vollkommenen Plattformunabhängigkeit ist somit bei einer JNI-basierten Lösung nicht mehr gegeben. In Abschnitt 10.2 wird auf dieses Thema noch einmal eingegangen. Einzelheiten zu den Konvertierungsfunktionen des JNI können in [7] nachgeschlagen werden.

7.1.1 Umsetzung

Zu Beginn der Arbeit an der Java - C Kommunikation stellte sich die Frage, auf welcher Ebene die Konvertierung der Parameter aus der Java-Welt nach C und die Java-konformen Funktionsaufrufe implementiert werden sollten. Es bot sich an, eine weitere DLL zu entwickeln um mit dieser dann über die *ecc_gfp.dll* mit dem ECP zu kommunizieren. Alternativ dazu bestand noch die Möglichkeit, alle Java Konvertierungen ebenfalls direkt in der *ecc_gfp.dll* zu implementieren und so dem FlexiProvider einen direkten Zugriff ohne den Umwege über die *ECPInterface.dll* zu geben. Der Nachteil dieser Lösung besteht jedoch darin, dass zum einen das bewährte Schichtenmodell hätte aufgeweicht werden müssen und zum anderen die Flexibilität der *ecc_gfp.dll* eingeschränkt gewesen wäre. Die aktuelle Lösung ermöglicht jedweder Anwendungen über die *ecc_gfp.dll* auf den ECP zuzugreifen, während Java Anwendungen - da diese noch Konvertierungsoperationen brauchen - noch eine Zwischenschicht (Wrapper-DLL) benötigen.

Eine detaillierte Beschreibung der *ECPInterface.dll* und der dazu korrespondierenden Java Klasse *ECPInterface.java* befinden sich in Anhang B.2 und Anhang C.2.

7.1.2 Zeitverhalten

Die aus Java ermittelten Zeiten für eine Punktmultiplikation unter Verwendung des ECP sind geringfügig schlechter, als die mit der Visual-Basic Anwendung gemessenen Zeiten in Tabelle 5.3. Diese Verschlechterung begründet sich im zusätzlichen Aufwand, den der Übergang von Java nach C über das JNI erzeugt. Allerdings wird durch diesen zusätzlichen Aufwand deutlich weniger Leistung eingebüßt, als erwartet.

7.2 JNI Kommunikation - Java

Auf Java Seite wird zur Kommunikation über JNI die *ECPInterface.java* benötigt. Diese Klasse stellt die einzige Schnittstelle des FlexiProviders zur *ECPInterface.dll* dar, welche die C-

Gegenseite bildet. Eine Besonderheit der *ECPInterface.java* ist, dass von ihr in einer JVM nur *genau eine* Instanz existieren kann. Damit wird vermieden, das mehrere Instanzen dieses Objekts zur gleichen Zeit versuchen, über die *ECPInterface.dll* auf die *ecc_gfp.dll* zuzugreifen.

Die *ECPInterface.java* Klasse unterteilt sich in *Intitialisierungs-*, *Arithmetische-*, *Support-* und *Konvertierungs-* Methoden. Dabei nehmen die arithemtischen Operationen den größten Teil des Umfangs ein. Diese Klasse kann ihre Aufgabe, die Kommunikation mit der *ecc_gfp.dll* nur Wahrnehmen, wenn auch die *ECPInterface.dll* als Systembibliothek geladen wurde. Sollte dies nicht der Fall sein, meldet sie an das aufrufende Objekt zurück, dass keine Unterstützung durch native Code vorhanden ist. In diesem Fall rechnet der FlexiProvider auf klassische Art, in Java Software, weiter und greift nicht mehr auf die *ECPInterface.java* zu.

7.2.1 Initialisierungs-Methoden

Diese Methoden dienen nur der erstmaligen Initialisierung und werden, soweit kein Wechsel in der Bitbreite oder der Kurve durchgeführt wird, nur genau einmal zum Beginn der Operationen ausgeführt. Diese Klasse speichert die in der darunter liegenden DLL benutzte Kurve. Hierdurch wird das erneute Laden bei einer Neuinitialisierung mit identischen Parametern verhindert.

loadPrime
: Überträgt die dem Körper zu Grunde liegende Primzahl p an die darunter liegende DLL

loadCurve
: Lädt die Kurve in der Form von zwei Kurvenparametern

setBitwidth
: Setzen der Bitbreite für die folgenden Berechnungen

7.2.2 Arithmetische Methoden

Alle arithmetischen Methoden arbeiten mit Punkten über *GF(p)* . Unabhängig davon, ob eine Instanz des Objekts *PointGFP* oder *PointGFPHardware* übergeben wird, ist der Ergebnispunkt immer eine Instanz des Objekts *PointGFP*. Eine Erweiterung der Hardware Unterstützung um Operationen über *GF(*2^n*)* erfordert an dieser Klasse geringfügige Veränderungen, auf die in Kapitel 10 näher eingegangen wird.

loadPoint
: Überträgt einen Punkt an die darunter liegende DLL

add
: Gibt das Ergebnis der Addition zweier Punkte in der DLL zurück

mult
: Gibt das Ergebnis einer Multiplikation eines Punktes mit einem Skalar in der DLL zurück

negate
Gibt das Ergebnis einer Negation eines Punktes in der DLL zurück

onCurve
Läßt in der DLL einen on Curve Check durchführen für den übertragenen Punkt

7.2.3 Support-Methoden

Die Support Methoden dienen der komfortablen Verwendung des ECP bzw. der zu Grunde liegenden DLL.

dllSupportAvailable
Prüft, ob DLL-Unterstützung vorhanden ist

status
Gibt den momentanen Status der DLL Unterstützung zurück

logStart
Startet das Protokollieren in eine Textdatei

logStop
Beendet das Protokollieren

7.2.4 Konvertierungs-Methoden

Es existiert nur eine Konvertierungsmethode in der *ECPInterface.java* Klasse. Die Aufgabe von *getBI* ist die Umformung der verschiedenen Zahldarstellungen. Um einfach die von C zurückgegeben Bit-Arrays für die einzelnen Punktkoordinaten in BigInteger umzuwandeln, müssen die Bit-Arrays in *Big-Endian Darstellung* und in *2er Komplement* Darstellung konvertiert werden. Sollte sich an der Zahldarstellung des native Codes etwas ändern, so genügt es diese Methode anzupassen. Weitere Änderungen am Provider sind nicht notwendig.

7.3 JNI Kommunikation - C

Die *ECPInterface.dll* bildet das C-Seitige Pendant zur *ECPInterface.java*. Ihre Funktionen entsprechen allen in der *ECPInterface.java* aufgeführten native Code Funktionen. Sie ist, aufgrund von Vorgaben durch das JNI an die Provider Struktur angepasst. Anpassungen an andere *Java Cryptography Provider* sind mit geringen Aufwand möglich, da hier nur die Funktionsnamen ersetzt werden müssen. Einzelheiten hierzu können in [7] nachgeschlagen werden.

Die Gliederung der Funktionen ist dementsprechend identisch zur *ECPInterface.java*. Zur Erhöhung der Lesbarkeit dieses Abschnittes wird der Header (Ja-

va_de_flexiprovider_ECPInterface_EC_1) jeder exportierten Funktion nicht explizit aufgeführt. Diese Funktionen werden durch einen Stern (*) gekennzeichnet. Interne Funktionen der DLL tragen diesen Header nicht.

7.3.1 Initialisierungs-Methoden

*LoadCurve**
: Lädt die Kurve in die *ecc_gfp.dll*

*Init**
: Lädt Primzahl, Kurvenparameter und Basispunkt in die *ecc_gfp.dll*

*Bitwidth**
: Setzt die Bitbreite

*LoadPrime**
: Lädt die dem Körper zu Grunde liegende Primzahl

*LoadPoint**
: Lädt einen Punkt

*LoadPointAffine**
: wie *LoadPoint*, nur in affiner Darstellung

7.3.2 Arithmetische Methoden

Diese Funktionen unterscheiden sich etwas von den in der *ECPInterface.java* aufgeführten Methoden, da hier zwischen Punkt-Koordinaten in *projektiver* und *affiner* Darstellung unterschieden wird.

*Mult**
: Multipliziert den auf der Karte befindlichen Basispunkt mit einem Skalar

*MultPoint**
: Multipliziert einen Punkt mit einem Skalar

*Compare**
: Vergleicht zwei Punkte

*IsOnCurve**
: Führt einen on-Curve Check durch

*Add**
Additert zwei Punkte

*Negate**
Negiert den Punkt

*MultPointAffine**
wie *MultPoint*, nur in affiner Darstellung

*MultAffine**
wie *Mult*, nur in affiner Darstellung

*NegateAffine**
wie *Negate*, nur in affiner Darstellung

*AddAffine**
wie *Add*, nur in affiner Darstellung

*IsOnCurveAffine**
wie *IsOnCurve*, nur in affiner Darstellung

7.3.3 Support-Methoden

*LogStart**
Starten des Log Vorgangs

*LogStop**
Beenden des Log Vorgangs

*Support**
Überprüft, ob Support durch die *ecc_gpf.dll* vorhanden ist

*CompBinEqual**
Vergleicht zwei Binär-Darstellungen auf Gleichheit

*ConvToAffine**
Konvertiert projektive Koordinaten nach Affin

*ConvHexstrToBin**
Konvertiert einen String in Hex- in eine Binär-Darstellung

*ConvDecstrToBin**
Konvertiert einen String in Dezimal- in eine Hex-Darstellung

*ConvBinToHexstr**
Konvertiert eine Binär- in einen String in Binär-Darstellung

*ConvBinToBinstr**
Konvertiert eine Binär- in einen String in Hex-Darstellung

*ConvBinToDecstr**
Konvertiert eine Binär- in einen String in Dezimal-Darstellung

*GetBytewidth**
Abfrage der Byte-Array Länge

*GetBitwidth**
Abfrage der Bitbreit

*GetBitwidthHW**
Abfrage der in Hardware gerade verwendeten Bitbreite

7.3.4 Interne Methoden

Unter diese Gruppe fallen alle Funktionen zur Konvertierung von Java- nach C-Datentypen (bzw. umgekehrt) und zur Allokation und Freigabe von Speicherbereichen, die zur internen Speicherung der Parameter notwendig sind.

getByteWidth
Abfrage der Byte-Array Länge

jStringToLPSTR
Konvertierung eines Java String in die C-äquivalente Struktur

jByteToLongArray
Konvertierung eines Java Byte in eine Long-Array Struktur

LongTojByteArray
Konvertierung eines Long-Arrays in ein Java Byte

resizeGlobals
Allokiert die Größe der globalen Variablen neu

emptyGlobals
: Löscht den Inhalt der globalen Variablen

clearGlobals
: Gibt die globalen Variablen wieder frei

7.4 Modifikation am FlexiProvider

Um eine Anbindung des ECP an den FlexiProvider zu realisieren, mussten einige Klassen verändert bzw. hinzugefügt werden. Änderungen wurden an *EllipticCurveGFP.java,Point.java* und *PointGFP.java* durchgeführt, während für die Hardware Unterstützung die dedizierten Klassen *EllipticCurveGFPHW.java* und *PointGFPHW.java* hinzugekommen sind. Die Schnittstelle zu C bildet die Klasse *ECPInterface.java*.

Im Folgenden wird auf die einzelnen Klassen näher eingegangen. Die API Spezifikation der hier aufgeführten Klassen sind unter [10] zu finden.

EllipticCurveGFP.java

Package: `de.flexiprovider.ec.arithmetic.curves`
Diese Klasse repräsentiert elliptische Kurven über Primkörpern. Die Veränderungen an dieser Klasse beschränken sich auf das Einfügen einer neuen Funktion, die ermittelt, ob ein Hardware Interface vorhanden ist. Im Falle, dass Hardware Unterstützung vorhanden ist, werden bei Erzeugung einer neuen Objektinstanz die Kurvenparameter über die *ECPInterface.java* nach C übergeben.

Point.java

Package: `de.flexiprovider.ec.arithmetic.curves`
In dieser Klasse werden, falls Hardware Unterstützung vorhanden ist, alle Methoden zur Multiplikation auf die statische Hardware Multiplikations Methode in *PointGFPHardware.java* umgeleitet. Sollte keine Hardware Unterstützung vorhanden sein, arbeitet diese Klasse unverändert.

PointGFP.java

Package: `de.flexiprovider.ec.arithmetic.curves`
Auch in dieser Klasse werden die arithmetischen Operationen (add,onCurve) auf die statischen Hardware Methoden in *PointGFPHardware.java* umgeleitet. Wie bereits in *Point.java* arbeitet diese Klasse auch unverändert, falls keine Hardware Unterstützung vorhanden sein.

EllipticCurveGFPHW.java

Package: `de.flexiprovider.ec.arithmetic.curves`
Bei dieser Klasse handelt es sich um eine dedizierte Hardware Klasse. In den statischen Methoden stellt sie anderen Klassen die Hardware Unterstützung zu Verfügung.

Das Benutzen dieser Klasse setzt weitere Veränderungen am Quellcode voraus. Der Vorteil dabei ist, dass nicht abgefragt wird, ob Hardware Unterstützung vorhanden ist. Daraus ergibt sich ein minimaler zeitlicher Vorteil gegenüber der Standard Klasse. Aufgrund der Inflexibiliät dieser Lösung sind jedoch die Standard Klassen vorzuziehen.

PointGFPHardware.java

Package: `de.flexiprovider.ec.arithmetic.curves`
Diese Klasse ist die zu *PointGFP.java* korrespondierende dedizierte Hardware Klasse. Wie auch bei *EllipticCurveGFPHW.java* stellt sie einige statisch Hardware Zugriffsmethoden zur Verfügung, ist aber ebenfalls den Standard Klassen nicht vorzuziehen.

ECPInterface.java

Package: `de.flexiprovider.ec.arithmetic.curves`
Der Zugriff auf den ECP von Java aus geschieht über das so genannte ECPInterface. Das Interface besteht aus den beiden, in den vorangegangenen Abschnitten beschriebenen, Komponenten *ECPInterface.java* und *ECPInterface.dll*.

Kapitel 8

Laufzeiten - Providerebene

Aufgrund der modularen Struktur des FlexiProviders konnte die Integration des ECP für Anwendungen, die den FlexiProvider nutzen, vollkommen transparent durchgeführt werden. Es ist der eigentlichen Anwendung nicht möglich festzustellen, ob die Berechnungen auf dem ECP ablaufen oder ob die Software Implementierung hier die Arbeit erledigt.

Im diesem Kapitel wird auf das Laufzeitverhalten des Providers mit und ohne Hardware-Unterstützung eingegangen. Zu Beginn des Projektes stand es außer Frage, dass die Punkt-Multiplikation in Hardware in einem deutlichen Geschwindigkeitsvorteil resultiert. Es war jedoch auch interessant zu überprüfen, ob auch einzelne Punkt-Additionen und Negationen in Hardware schneller ablaufen. Ziel der Integration des ECP ist letztendlich, die Laufzeit der Provider Operationen Schlüsselpaargenerierung (KPG), Signaturerstellung (Sign) und Signaturverifikation (Verify) zu steigern und damit auch die Gesamtperformanz.

Laufzeitmessungen der reinen Software- bzw. Hardware-Implementierung sind in Abschnitt 8.1 und Abschnitt 8.2 dokumentiert. Die Realisierung mit der besten Gesamtperformanz ist in Abschnitt 8.3 behandelt. Bei den angegebenen HW-SW Vergleichen wurde grundsätzlich die schnellste Java Implementierung zugrunde gelegt, d.h. alle Berechnungen in Java werden mit vorberechneten Punkten durchgeführt. Der höhere Aufwand bei der Initialisierung fällt nicht ins Gewicht, da diese bei der Verwendung des FlexiProviders nur einmal zu Beginn durchgeführt wird.

8.1 FlexiProvider ohne ECP

In Analogie zu den Zeitmessungen in Kapitel 5 wurde auf der selben Testplattform die Performanz des FlexiProviders ermittelt. Dabei wurden zuerst Referenzzeiten mit der Java-Implementierung ohne ECP-Unterstützung in Echtzeit gemessen. Diese Zeiten sind in Tabelle 8.1 zusammengefasst.

Die geringeren Laufzeiten der Provider Operationen *Schlüsselpaarerstellung*, *Signaturerstellung* und *Signaturverifikation* im Vergleich zur Basisoperation *EC-Mult* ergeben sich aus der Verwendung vorberechneter Punkte.

Tabelle 8.1: Laufzeiten aller optimierten EC-Basisoperationen (Java)

Bitbreite	*EC-Mult*	*EC-Add*	*EC-Negate*	*KPG*[1]	*Sign*[2]	*Verify*[3]
140	27,61 ms	0,14 ms	0,05 ms	10,95 ms	11,42 ms	36,13 ms
160	33,38 ms	0,17 ms	0,03 ms	11,88 ms	12,22 ms	37,56 ms
192	47,75 ms	0,19 ms	0,05 ms	18,20 ms	18,55 ms	56,41 ms
197	61,00 ms	0,23 ms	0,06 ms	23,45 ms	23,88 ms	73,53 ms
272	124,30 ms	0,36 ms	0,08 ms	48,08 ms	48,67 ms	146,44 ms
300	162,36 ms	0,44 ms	0,08 ms	62,89 ms	63,89 ms	188,39 ms
400	332,45 ms	0,66 ms	0,14 ms	130,55 ms	132,16 ms	386,09 ms

8.2 FlexiProvider mit ECP (alle Operationen)

Im nächsten Schritt wurden *alle* Operationen mit Hilfe des ECP in Hardware ausgeführt. Die ermittelten Zeiten sind in Tabelle 8.2 zusammengefaßt. Dabei fällt auf, dass die Laufzeiten für *EC-Mult*, Schlüsselpaargenerierung, Signaturerstellung und Signaturverifikation wesentlich schneller sind, als in der Java-Implementierung. Die Punktmultiplikation bei 400 Bit ist in der ECP-Realisierung 37 mal schneller, als in der reinen Java-Variante. Die sehr einfachen Operationen *EC-Add* und *EC-Negate* arbeiten allerdings in Java schneller. Bei diesen Funktionen fällt der höhere Kommunikationsaufwand der ECP-Variante im Verhältnis zu den einfachen arithmetischen Operationen deutlich stärker ins Gewicht.

Tabelle 8.2: Laufzeiten aller EC-Basisoperationen (mit ECP)

Bitbreite	*EC-Mult*	*EC-Add*	*EC-Negate*	*KPG*	*Sign*	*Verify*
140	1,61 ms	0,47 ms	0,25 ms	1,75 ms	2,33 ms	4,52 ms
160	1,84 ms	0,44 ms	0,27 ms	1,84 ms	2,13 ms	4,67 ms
192	2,34 ms	0,45 ms	0,25 ms	2,34 ms	2,66 ms	5,69 ms
197	2,73 ms	0,49 ms	0,28 ms	2,73 ms	3,11 ms	6,64 ms
272	4,50 ms	0,50 ms	0,31 ms	4,52 ms	5,08 ms	10,52 ms
300	5,19 ms	0,53 ms	0,27 ms	5,17 ms	5,78 ms	11,94 ms
400	8,88 ms	0,63 ms	0,31 ms	8,89 ms	9,78 ms	19,89 ms

[1] Schlüsselpaargenerierung - KeyPair Generation
[2] Signaturerstellung
[3] Signaturverifikation

8.3 FlexiProvider mit ECP (nur EC-Mult)

Um die Performanzvorteile der Software bei *EC-Add* und *EC-Negate* und die des ECP bei *EC-Mult* zu nutzen, wurde eine kombinierte Lösung implementiert. Dabei laufen die *EC-Add* und *EC-Negate* Operationen auf klassische, in Java implementierte Weise und die *EC-Mult* Operation wird auf dem ECP durchgeführt.

Tabelle 8.3: Laufzeiten der EC-Basisoperationen der kombinierten Implementierung

Bitbreite	*EC-Mult*	*EC-Add*	*EC-Negate*	*KPG*	*Sign*	*Verify*
140	1,45 ms	0,16 ms	0,05 ms	1,61 ms	2,17 ms	4,08 ms
160	1,69 ms	0,17 ms	0,05 ms	1,67 ms	1,97 ms	4,20 ms
192	2,17 ms	0,20 ms	0,05 ms	2,19 ms	2,52 ms	5,22 ms
197	2,61 ms	0,23 ms	0,05 ms	2,64 ms	2,97 ms	6,27 ms
272	4,41 ms	0,36 ms	0,09 ms	4,41 ms	4,95 ms	10,24 ms
300	5,13 ms	0,41 ms	0,13 ms	5,13 ms	5,72 ms	11,81 ms
400	8,89 ms	0,64 ms	0,13 ms	8,89 ms	9,83 ms	19,91 ms

Die Laufzeiten dieser kombinierten Lösung sind in Tabelle 8.3 dargestellt. Am deutlichsten ist der Performanzgewinn bei der Verifikation. Der Vergleich der Laufzeiten aus Tabelle 8.1 mit Tabelle 8.3 zeigt, dass für die ECP-basierte Verifikation ein Beschleunigungsfaktor von bis zu 19,4 (bei 400 Bit) realisiert werden konnte. Tabelle 8.4 zeigt für alle Provider-Operationen, um welchen Faktor die kombinierte Implementierung schneller ist als die reine Java Realisierung.

Tabelle 8.4: Beschleunigungsfaktoren der kombinierten Lösung gegenüber der Java Software Implementierung

Bitbreite	*KPG*	*Sign*	*Verify*
140	6,81	5,26	8,86
160	7,10	6,21	8,94
192	8,32	7,37	10,81
197	8,88	8,04	11,74
272	10,91	9,83	14,31
300	12,27	11,17	15,95
400	14,68	13,45	19,40

8.4 FlexiProvider mit Berechnungen in C-Software

Besonders interessant für Anwendungsszenarien, in denen die benötigte Bitbreite die Kapazitäten der Hardware übersteigt, sind die Laufzeiten in Tabelle 8.5. Erwartungsgemäß sind hierbei

Tabelle 8.5: Laufzeiten der EC-Basisoperationen in der kombinierten Implementierung mit C-SW Berechnungen

Bitbreite	*EC-Mult*	*EC-Add*	*EC-Negate*	*KPG*	*Sign*	*Verify*
140	14,78 ms	0,34 ms	0,12 ms	15,00 ms	15,63 ms	33,72 ms
160	20,84 ms	0,44 ms	0,16 ms	21,37 ms	22,03 ms	45,94 ms
192	30,34 ms	0,47 ms	0,19 ms	30,37 ms	31,59 ms	63,41 ms
197	30,88 ms	0,47 ms	0,19 ms	30,97 ms	32,28 ms	66,31 ms
272	65,53 ms	0,72 ms	0,25 ms	65,62 ms	68,56 ms	140,69 ms
300	76,63 ms	0,78 ms	0,28 ms	76,81 ms	82,50 ms	168,63 ms
400	179,75 ms	1,34 ms	0,44 ms	180,37 ms	190,41 ms	394,75 ms

die Laufzeiten größer als mit ECP-Unterstützung. Von Vorteil ist jedoch die Möglichkeit der Berechnung in C, wenn in einer Dritt-Anwendung keine eigene EC-Arithmetik implementiert wurde. Solche Anwendungen können dennoch, unabhängig von der Bitbreite, alle Berechnungen über die *ecc_gfp.dll* laufen lassen.

Kapitel 9

Zusammenfassung

Durch die Integration des ECP in den Java-basierten FlexiProvider konnte eine Leistungssteigerung der teuersten Operation, der Multiplikation, bis um Faktor 37 erzielt werden. Die aufwendigen arithemtischen Operationen über Elliptischen Kurven können mit Hilfe des ECP sehr effizient berechnet werden. Das eingesetzte Schichtenmodell, durch das die Funktionalität auch anderen Anwendungen zur Verfügung steht, stellt eine hohe Flexibilität des Gesamtsystems sicher. Der Aufwand, der durch die mehrfachen Konvertierungen zwischen den einzelnen Schichten entsteht, fällt im Vergleich zur enormen Leistungssteigerung nur sehr gering aus und beeinträchtigt die Performanz des Systems nicht wesentlich.

9.1 Ergebnisse

Diese Arbeit hat gezeigt, wie sich die Vorteile des FlexiProviders und damit der Java Cryptography Architecture mit dem Kryptoprozessor ECP verbinden lassen. Die JCA stellt den *de-facto* Standard für Plattform-unabhängige und flexible Implementierung kryptographischer Algorithmen dar. Dies ist einer der entscheidenden Vorteile des FlexiProviders, da dieser in Übereinstimmung mit der JCA arbeitet. Ein grosser Nachteil ist jedoch die Komplexität der Algorithmen über elliptischen Kurven, die in einer sehr hohen Laufzeit der Software Berechnungen resultieren. An diesem Punkt springt der ECP ein. Er implementiert hoch-effizient in Hardware die benötigten Algorithmen und bietet der Java-Welt somit eine Alternative, zu den langsamen Operationen in Software. Damit wird eine Beschleunigung der Provider Operationen um das bis zu *20-fache* erreicht. Dadurch, dass Hardware Designs zur Laufzeit ausgewechselt werden können, ist die integrierte Lösung dennoch flexibel. Grund für diese Flexibilität ist der Einsatz eines FPGA, der keine feste und unveränderliche Hardware-Plattform darstellt, sondern dynamisch rekonfiguriert werden kann. So ist es möglich, für spezielle Bitbreiten optimierte Hardware-Designs zur Laufzeit auszuwechseln.

Der FlexiProvider bzw. das DLL-Modell vergrößern das Anwendungsspektrum für den ECP, nicht zuletzt und vor allem durch die in Kapitel 10 erläuterten Erweiterungen. Der ECP wurde durch die verschiedenen Zugriffsebenen (*ecc_gfp.dll*, *ECPInterface.dll*) für eine Vielzahl neuer, auf verschiedenen Plattformen implementierten Anwendungen erschlossen.

Die Integration des ECP in den FlexiProvider hat auch ein weiteres Einsatzgebiet für den FlexiProvider erschlossen: den Einsatz im Krypto Server Bereich. Hierbei eignet sich diese Lösung besonders, da dadurch Last vom Hauptprozessor genommen wird und dieser sich um andere Aufgaben wie der Kommunikation mit verschiedenen Clients und der Bearbeitung von Client-Anfragen kümmern kann. Dies ergibt ein besseres Anwortverhalten und eine höhere Effizienz des Gesamtsystems.

Als weiterer wichtiger Punkt ist herauszustellen, dass die gesamte Funktionalität des ECP für den FlexiProvider komplett transparent ist. Eine Änderung der zu Grunde liegenden Hardware-Plattform, eine Anpassung der untersten Schnittstellen voraussgesetzt, hat keine Auswirkung auf die Funktionalität des Providers, unter Umständen aber auf seine Performanz.

Der FlexiProvider muss sich dementsprechend in keiner Weise um die eingesetzte Hardware kümmern, vor allem nicht um die Unterschiede verschiedener Hardware Implementierung. Dies liegt allein im Verantwortungsbereich der *ecc_gfp.dll*.

9.1.1 ECP-Schnittstelle

Um den ECP für Anwendungen transparent nutzbar zu machen, wurde eine Software-Schnittstelle entwickelt, die alle Aufgaben hinsichtlich der ECP-Initialisierung der Ressourcen, Datenkonvertierung, Transfers, Berechnungen und des Interrupt-Handlings, übernimmt. Diese Schnittstelle bietet für den Anwender eine Fülle von Funktionen zur Arithmetik über elliptischen Kurven. Um die Schnittstelle selbst dann nutzen zu können, wenn keine entsprechende Hardware-Plattform im System integriert ist, wurden alle EC-Operationen auch in Software implementiert und in die Schnittstelle eingefügt. Die Software-Schnittstelle wurde in Form einer DLL (*ecc_gfp.dll*) für das Betriebssystem Microsoft Windows zur Verfügung gestellt.

9.1.2 FlexiProvider

Bei der Integration des ECP in den FlexiProvider standen Flexibilität und Transparenz im Vordergrund. Aus diesem Grund wurde das schon erwähnte, mehrstufige DLL Verfahren gewählt. Dabei werden in Java EC-Objekte in ihre atomaren Daten zerlegt und diese dann in eine schnittstellenkonforme Darstellung konvertiert um sie zur Hardware zu übertragen. Nach den Berechnungen erfolgt ein Rücktransformationsprozeß, bis der FlexiProvider wieder mit den Java Objekten weiterarbeiten kann.

9.2 Schichtenmodell

Das Schichtenmodell hat sich als einfach zu erweiterndes und performantes System erwiesen, welches ermöglicht, verschiedenen Anwendungen bis zum FlexiProvider hin, die Vorteile des ECP zu Verfügung zu stellen. Dabei sind, wie schon gezeigt, die Nachteile durch den erhöhten Kommunikationsaufwand, vernachlässigbar.

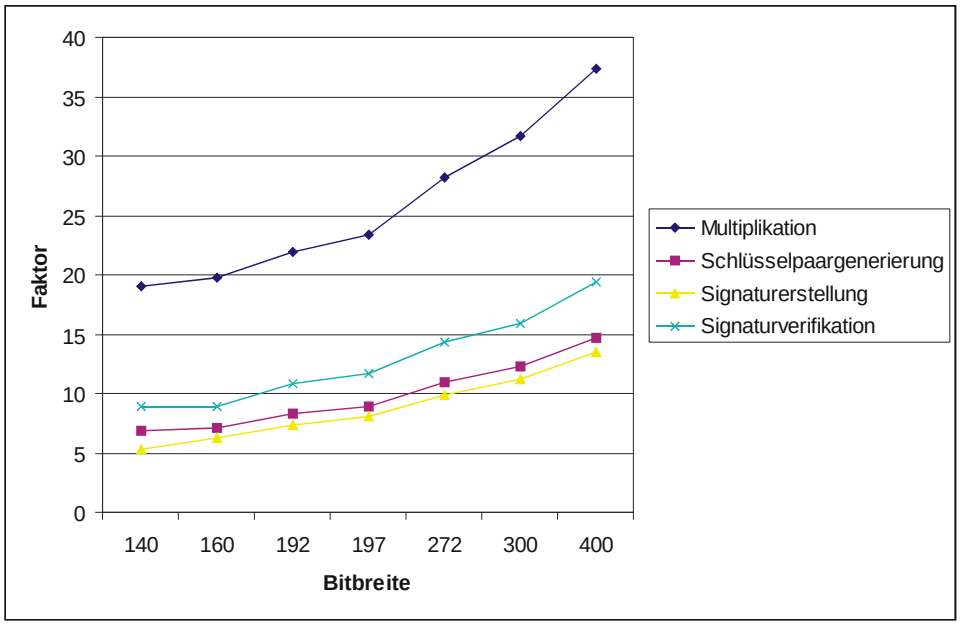

Abbildung 9.1: Beschleunigungsfaktoren der Operationen bei optimaler ECP Integration

Dieses Modell gewährleistet auch höchstmögliche Flexibilität beim Einsatz verschiedener FPGAs, Hardware Implementierungen und End-Anwendungen. Solange jede Schicht sich an die vorgegeben Konventionen hält, ist ein Austausch der jeweils zu Grunde liegenden Hardware oder aufrufenden Anwendung ohne weiteres möglich. Dieses Modell kann auch beliebig erweitert werden. So wird in Kapitel 10 ein durchaus realistisches Anwendungsszenario vorgestellt, welches eine Erweiterung um einige Schichten benötigt.

9.3 Laufzeiten - Vergleich

Die Tabellen 9.1 und 9.2 stellen einen abschliessenden Vergleich zwischen der klassischen, in Software berechneten Java Implementierung und der optimalen, hardwarebeschleunigten Lösung dar. Bei den hier vorgestellten Zeiten handelt es sich um Echtzeit-Messergebnisse. Ermittelt wurden diese auf einem Intel Xeon 1,8 GHz System mit 1 GByte RAM unter JRE 1.3.1 und Windows 2000.

Die Ergebnisse zeigen, dass durch die Integration der Hardware bei den Provider Operationen eine Beschleunigung von Faktor 5 (Signatur bei 140 Bit) bis 19 (Verifikation bei 400 Bit) erzielt wurde. Die Beschleunigung bei der Punkt-Multiplikation (EC-Mult) reicht von Faktor 19 bei 140 Bit bis Faktor 37 bei 400 Bit. Zur besseren Übersicht stellt Abbildung 9.1 die Beschleunigungsfaktoren grafisch dar.

Tabelle 9.1: Laufzeiten der EC-Basisoperationen im Vergleich (Java Software zu ECP)

Bitbreite	*EC-Mult*		*EC-Add*		*EC-Negate*	
	SW - Java	SW + ECP	SW - Java	SW + ECP	SW - Java	SW + ECP
140	27,61 ms	1,45 ms	0,14 ms	0,16 ms	0,05 ms	0,05 ms
160	33,38 ms	1,69 ms	0,17 ms	0,17 ms	0,03 ms	0,05 ms
192	47,75 ms	2,17 ms	0,19 ms	0,20 ms	0,05 ms	0,05 ms
197	61,00 ms	2,61 ms	0,23 ms	0,23 ms	0,06 ms	0,05 ms
272	124,30 ms	4,41 ms	0,36 ms	0,36 ms	0,08 ms	0,09 ms
300	162,36 ms	5,13 ms	0,44 ms	0,41 ms	0,08 ms	0,13 ms
400	332,45 ms	8,89 ms	0,66 ms	0,64 ms	0,14 ms	0,13 ms

Tabelle 9.2: Laufzeiten der Provider-Operationen im Vergleich (Java Software zu ECP)

Bitbreite	*Schlüsselpaargenerierung*		*Signaturerstellung*		*Signaturverifikation*	
	SW - Java	SW + ECP	SW - Java	SW + ECP	SW - Java	SW + ECP
140	10,95 ms	1,61 ms	11,42 ms	2,17 ms	36,13 ms	4,08
160	11,88 ms	1,67 ms	12,22 ms	1,97 ms	37,56 ms	4,20
192	18,20 ms	2,19 ms	18,55 ms	2,52 ms	56,41 ms	5,22
197	23,45 ms	2,64 ms	23,88 ms	2,97 ms	73,53 ms	6,27
272	48,08 ms	4,41 ms	48,67 ms	4,95 ms	146,44 ms	10,24
300	62,89 ms	5,13 ms	63,89 ms	5,72 ms	188,39 ms	11,81
400	130,55 ms	8,89 ms	132,16 ms	9,83 ms	386,09 ms	19,91

Kapitel 10

Ausblick

In den vorangegangen Abschnitten wurde klar herausgestellt, dass das Ergebnis dieser Arbeit nicht nur für theoretische Belange interessant ist. Deswegen ist auch von Interesse zu sehen, inwiefern die hier angebrachten Theorien und Implementierungen in Zukunft weiter genutzt und ausgebaut werden können.

In diesem Kapitel wird auf eine mögliche Erweiterung auf die Körper *GF(*2^n*)* eingegangen, sowie ein Anwendungsszenario für die kombinierte Lösung aus FlexiProvider und ECP vorgestellt.

10.1 Erweiterung auf *GF(*2^n*)*

Alle bis jetzt dargestellten Änderungen hatten zum Ziel, die Berechnungen in *GF(p)* zu Beschleunigung. Aufgrund des modularen Aufbaus des FlexiProviders ist eine Erweiterung auf *GF(*2^n*)* leicht durchzuführen. Im Falle, dass Hardware-Designs *GF(*2^n*)* unterstützen, wären nur einige Änderungen an der *ECPInterface.java* und die Erweiterung des Providers um *GF(*2^n*)* - Klassen notwendig, um volle Hardware-Beschleunigung auch über diesem Körper anzubieten. Die Veränderungen in der *ECPInterface.java* beschränken sich dabei auf einfache Unterscheidungen und veränderten Rückgabewerten für die beiden Fälle *GF(*2^n*)* und *GF(p)*. Analogien zu solchen Unterscheidung finden sich überall im FlexiProvider.

10.2 KryptoServer

Der FlexiProvider in seiner jetzigen Form, mit Hardware Unterstützung, ist auf Windows-Systeme beschränkt. Die *ECPInterface.dll* und *ecc_gfp.dll* verzichten zwar auf die Verwendung von Plattform-spezifischen Funktionen, die verwendeten Bibliotheken des Hardware Anbieters laufen jedoch ausschließlich unter Windows. Der Gedanke liegt natürlich nahe, dass dadurch der Vorteil der Plattformunabhängigkeit Javas nicht mehr gegeben ist und deshalb auch auf eine vermeintlich langsamere Plattform nicht gebaut werden sollte.

Das Schichtenmodell der DLLs bietet jedoch auch hier die Möglichkeit, mit einem vergleichsweise geringen Aufwand die Flexibilität des Providers zu erhöhen. Dadurch, dass die *ECPInterface.dll* in C++ geschrieben ist, kann sie auch ohne größere Veränderungen auf verschiedenen Plattform zum Laufen gebracht werden.

So könnte das Schichtenmodell um eine weitere, über das Netzwerk kommunizierende Schicht erweitert werden. Dadurch wäre es möglich unter Verwendung von *Remote Procedure Calls* (RPC[1]), eine Vielzahl von FlexiProvidern auf verschiedenen Plattformen über das Netz mit einem zentralen KryptoServer kommunizieren zu lassen. Es sei an dieser Stelle noch einmal darauf hingewiesen, dass jede Schicht, sei es der FlexiProvider, die *ECPInterface.dll* oder *ecc_gfp.dll*, ihre Funktionalitäten der jeweils höheren Schicht transparent zur Verfügung stellt. Eine Erweiterung der Schichten ist also ohne grossen Aufwand möglich. Der Austausch der Schlüssel über ein Netzwerk darf natürlich nur in einer gesichertern Umgebung stattfinden, denn die Sicherheit von vertraulichen Daten ist immer von der Sicherheit des Schlüssels abhängig.

[1] RPC sind eine Einrichtung zur Nachrichtenweitergabe, durch die eine verteilte Anwendung verfügbare Dienste auf mehreren Computern im Netzwerk aufrufen kann.

Literaturverzeichnis

[1] R. L. Rivest, A. Shamir and L. M. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,” *Communications of the ACM*, Feb 1978.

[2] N. Koblitz, “Elliptic Curve Cryptosystems,” *Mathematics of Computation*, vol. 48, pp. 203–209, 1987.

[3] V. Miller, “Use of elliptic curves in cryptography,” *Advances in Cryptology, Proc. CRYPTO’85*, LNCS 218, H. C. Williams, Ed., Springer-Verlag, pp. 417–426, 1986.

[4] A. Lenstra and E. Verheul, “Selecting Cryptographic Key Sizes,” *Proc. Workshop on Practice and Theory in Public Key Cryptography*, Springer-Verlag, ISBN 3540669671, pp. 446–465, 2000.

[5] D. Johnson, A. Menezes and S. Vanstone, “The Elliptic Curve Digitial Signature Algorithm (ECDSA),” *International Journal on Information Security*, pp. 36–63, 2001.

[6] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, “Handbook of Applied Cryptography”, CRC Press, 1997.

[7] Sheng Liang, Java(TM) Native Interface: Programmer’s Guide and Specification.

[8] SUN, “Java Cryptography Architecture API Specification & Reference,” http://java.sun.com/products/jdk/1.1/docs/guide/security/CryptoSpec.html.

[9] SUN, “Java Native Interface,” http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html.

[10] http://www.flexiprovider.de.

[11] ANSI X9.62, “Public key cryptography for the financial services industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)”, (zu beziehen über den ANSI X9 Katalog),1999.

[12] IEEE P1363, “Standard Specifications For Public Key Cryptography”, http://grouper.ieee.org/groups/1363/,2000.

[13] Alpha Data Parallel Systems Ltd., “ADM-XPL PCI Mezzanine Card User Guide”, Ver. 1.4, 2002.

[14] Alpha Data Parallel Systems Ltd., “ADC-PMC-64 User Manual”, Ver. 1.1, 2002.

[15] Alpha Data Parallel Systems Ltd., “ADC-XRC SDK 4.1.1 User Guide”, Ver. 1.1, 2001.

[16] Microsoft, Inc., “MSDN Library Visual Studio 6.0”, 1998.

[17] NTL, “A Library for doing Number Theory”, http://www.shoup.net/ntl/, 2002.

[18] Xilinx, Inc., “VirtexII Pro Platform FPGA Handbook”, 2002.

[19] FlexiProvider-Group, “*FlexiProvider*, a powerful toolkit for cryptography”, in Entwicklung, http://www.flexiprovider.de, 2003.

Anhang A

Ecc_gfp.dll

Vorbemerkungen

Speicherbereiche

Die DLL verwendet zur Übertragung der *GF(p)*-Daten untypisierte Pointer als Parameter. Diese Pointer müssen auf bereits existierende und ausreichend dimensionierte Speicherbereiche verweisen, in denen die Daten enthalten sind. Dabei wird immer die Little-Endian Darstellung verwendet. Das bedeutet, dass das niederwertigste Byte des Zahlenwertes im ersten Byte des Speicherbereichs abgelegt ist. Dies betrifft allerdings nur die Reihenfolge der Bytes innerhalb des Speicherbereichs. Die Bits innerhalb der Bytes sind jeweils in Big-Endian Darstellung.

Eingangsparameter sind dabei immer als *const void* * während Parameter vom Typ *void* * als Rückgabeparameter für Ergebnisdaten zu betrachten sind. Auch die Namensgebung der Parameter erleichtert die Unterscheidung: Rückgabeparameter sind mit *pDest.*. benannt und erscheinen immer vor den Eingangsparametern, welch mit *pSrc*.. benannt sind.

Die erforderliche Größe dieser Speicherbereiche ist abhängig von der verwendeten Bitbreite und kann über die Funktion $EC\_GetBytewidth$ (S. 76) nach dem Setzen der Bitbreite abgefragt werden. Es ist wichtig, dass alle Speicherbereiche mindestens diese Größe haben.

Projektiv vs. Affin

Bei allen EC-Arithmetik-Funktionen gibt es eine projektive und eine affine Variante. Diese erwarten Eingabepunkte immer in affiner Darstellung und rechnen Ergebnispunkte ebenfalls in die affine Darstellung um.

Dabei greift eine affine Funktion jeweils auf die zugehörige projektive Funktion zu und übergibt dabei für alle Z-Parameter den Wert $NULL$. Die affinen Funktionen sind lediglich eine kleine Erleichterung für die weiteren Entwicklungen, es kann aber auch nur mit den projektiven Funktionen gearbeitet werden.

A.1 Initialisierung

Diese Funktionen werden zur Initialisierung der DLL benötigt. Hierin werden alle zur Berechnung benötigten Daten definiert.

EC_Bitwidth

Diese Funktion setzt die Bitbreite der Operanden. Alle *GF(p)*-Element, mit denen gerechnet werden soll, müssen mit dieser Bitanzahl darstellbar sein. Dies ist die wichtigste Initialisierungsfunktion. Sie muß aufgerufen werden, noch bevor irgendein *GF(p)*-Element in einer anderen Funktion verwendet werden kann.

Signatur

```
int EC_Bitwidth (const int iBitwidth)
```

Parameter

- `iBitwidth`: Die Bitbreite der *GF(p)*-Operanden

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_INVALID_BITWIDTH`: Die Bitbreite ist nicht zulässig. Sie muß im Bereich [1..512] liegen.

EC_LoadPrime

Die *GF(p)* zugrundeliegende Primzahl p wird mit dieser Funktion definiert.

Signatur

```
int EC_LoadPrime (const void *pSrcPrime)
```

Parameter

- `pSrcPrime`: Primzahl, auf der *GF(p)* basiert

Rückgabewert

- `ERR_NO_BITWIDTH`: Die Bitbreite wurde noch nicht gesetzt. (→ $EC\_Bitwidth$, S. 57)
- `ERR_OK`: Kein Fehler aufgetreten

EC_LoadPoint

Mit dieser Funktion wird der Basispunkt für die *EC_Mult*-Operation gesetzt. Der Punkt kann in projektiver Darstellung vorliegen. Innerhalb dieser Funktion werden automatisch die für das Sliding-Window Verfahren nötigen Vorberechnungen durchgeführt.

Signatur

```
int EC_LoadPoint
    (const void *pSrcX0, const void *pSrcY0, const void *pSrcZ0)
```

Parameter

- `pSrcX0`: X-Koordinate des Basispunktes
- `pSrcY0`: Y-Koordinate des Basispunktes
- `pSrcZ0`: Z-Koordinate des Basispunktes

Rückgabewert

- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)
- `ERR_OK`: Kein Fehler aufgetreten

EC_LoadPointAffine

Mit dieser Funktion wird der Basispunkt für die *EC_Mult*-Operation gesetzt. Der Punkt muß in affiner Darstellung vorliegen. Innerhalb dieser Funktion werden automatisch die für das Sliding-Window Verfahren nötigen Vorberechnungen durchgeführt.

Signatur

```
int EC_LoadPointAffine (const void *pSrcX0, const void *pSrcY0)
```

Parameter

- `pSrcX0`: X-Koordinate des Basispunktes
- `pSrcY0`: Y-Koordinate des Basispunktes

Rückgabewert

- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)
- `ERR_OK`: Kein Fehler aufgetreten

EC_LoadCurve

Die Parameter der A und B der elliptischen Kurve werden mit *EC_LoadCurve* gesetzt.

Signatur

```
int EC_LoadCurve (const void *pSrcA, const void *pSrcB)
```

Parameter

- `pSrcA`: Parameter A der elliptischen Kurve
- `pSrcB`: Parameter B der elliptischen Kurve

Rückgabewert

- `ERR_NO_PRIME`: Die Primzahl wurde noch nicht gesetzt. (→ $EC\_LoadPrime$, S. 57)
- `ERR_OK`: Kein Fehler aufgetreten

EC_Init

Die *EC_Init*-Funktion vereint die Initialisierungroutinen *EC_LoadPrime*, *EC_LoadCurve* und *EC_LoadPoint* in einer einzigen Routine. Dadurch erhält man das selbe Ergebnis, als wenn man die Funktionen in dieser Reihenfolge einzeln aufruft.

Signatur

```
int EC_Init
    (const void *pSrcPrime, const void *pSrcA, const void *pSrcB,
    const void *pSrcX0, const void *pSrcY0, const void *pSrcZ0)
```

Parameter

- `pSrcPrime`: Primzahl, auf der *GF(p)* basiert
- `pSrcA`: Parameter A der elliptischen Kurve
- `pSrcB`: Parameter B der elliptischen Kurve
- `pSrcX0`: X-Koordinate des Basispunktes
- `pSrcY0`: Y-Koordinate des Basispunktes
- `pSrcZ0`: Z-Koordinate des Basispunktes

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_BITWIDTH`: Die Bitbreite wurde noch nicht gesetzt. (→ $EC\_Bitwidth$, S. 57)
- `ERR_NO_PRIME`: Die Primzahl wurde noch nicht gesetzt. (→ $EC\_LoadPrime$, S. 57)
- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)

EC_SetHardwareSupport

Hiermit kann die Hardware-Unterstützung explizit aktiviert oder deaktiviert werden. Normalerweise ist dies nicht nötig, da die DLL diese Unterstützung völlig transparent verwalten kann.

Signatur

```
int EC_SetHardwareSupport(int iEnabled)
```

Parameter

- `iEnabled`: 0: HW-Unterstützung deaktiviert, 1: HW-Unterstützung aktiviert

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_HARDWARE`: Die Hardwareunterstützung ist nicht verfügbar.
- `ERR_UNKNOWN`: Ein unbekannter Fehler ist aufgetreten.

EC_SupportBitwidth

Mit dieser Funktion kann die Bitbreite der Support-Funktionen gesetzt werden. So können Zahlkonvertierungen mit anderer Bitbreite durchgeführt werden, ohne die Hardware neu initialisieren zu müssen.

Signatur

```
int EC_SupportBitwidth (const int iBitwidth)
```

Parameter

- `iBitwidth`: Die Support-Bitbreite

Rückgabewert

- `ERR_INVALID_BITWIDTH`: Die Bitbreite ist nicht zulässig. Sie muß im Bereich [1..512] liegen.
- `ERR_OK`: Kein Fehler aufgetreten

A.2 *GF(p)* -Arithmetik

In diesem Abschnitt sind alle Funktionen zusammengefaßt, die sich auf ein einzelnes *GF(p)*-Element beziehen.

EC_Compare

Es wird überprüft, welche von zwei Binärzahlen größer bzw. kleiner ist.

Signatur

```
int EC_Compare (const void *pSrc1, const void *pSrc2)
```

Parameter

- `pSrc1`: erste Binärzahl
- `pSrc2`: zweite Binärzahl

Rückgabewert

- `-1`: $pSrc1 < pSrc2$
- `0`: $pSrc1 == pSrc2$
- `1`: $pSrc1 > pSrc2$
- `ERR_NO_CBITWIDTH`: Die Support-Bitbreite wurde noch nicht gesetzt. (→ $EC\_SupportBitwidth$, S. 60)

EC_GetHamming

Diese Funktion ermittelt das Hamminggewicht einer Binärzahl. Unter Hamminggewicht versteht man die Anzahl der Bits mit Wert 1 in der Zahl.

Signatur

```
int EC_GetHamming (const void *pSrc)
```

Parameter

- `pSrc`: Die Zahl, deren Hamminggewicht ermittelt werden soll

Rückgabewert

- `[0..512]`: Hamminggewicht der Zahl
- `ERR_NO_BITWIDTH`: Die Bitbreite wurde noch nicht gesetzt. (→ $EC\_Bitwidth$, S. 57)

EC_Random

Mit dieser Funktion kann eine zufällige Binärzahl generieren werden. Diese Zahlen sind aufgrund des relativ einfachen Zufallszahlengenerators nicht für kryptografische Anwendungen geeignet.

Signatur

```
int EC_Random (void *pDest)
```

Parameter

- `#pDest`: Speicherbereich, in dem die Zufallszahl zurückgegeben wird.

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_BITWIDTH`: Die Bitbreite wurde noch nicht gesetzt. (→ $EC\_Bitwidth$, S. 57)

EC_RandomHamming

Mit dieser Funktion kann eine zufällige Binärzahl mit festgelegtem Hamminggewicht generieren werden. Diese Zahlen sind aufgrund des relativ einfachen Zufallszahlengenerators nicht für kryptografische Anwendungen geeignet.

Signatur

```
int EC_RandomHamming (void *pDest, const int iHamming)
```

Parameter

- `#pDest`: Speicherbereich, in dem die Zufallszahl zurückgegeben wird.
- `iHamming`: Hamminggewicht, mit dem die Zufallszahl generiert werden soll.

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_BITWIDTH`: Die Bitbreite wurde noch nicht gesetzt. (→ $EC\_Bitwidth$, S. 57)

EC_CopyValue

Ein *GF(p)*-Element kann mit dieser Funktion in ein anderes kopiert werden.

Signatur

```
int EC_CopyValue (void *pDest, const void *pSrc)
```

Parameter

- `#pDest`: Speicherbereich, in den das Element kopiert wird.
- `pSrc`: zu kopierendes *GF(p)*-Element

Rückgabewert

- `1`: Der Wert konnte nicht kopiert werden
- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_BITWIDTH`: Die Bitbreite wurde noch nicht gesetzt. (→ $EC\_Bitwidth$, S. 57)

EC_CompBinEqual

Zwei Binärzahlen werden auf Gleichheit überprüft.

Signatur

```
int EC_CompBinEqual (const void *pSrc1, const void *pSrc2)
```

Parameter

- `pSrc1`: erste Binärzahl
- `pSrc2`: zweite Binärzahl

Rückgabewert

- `0`: Die Zahlen sind ungleich
- `1`: Die Zahlen sind gleich
- `ERR_NO_CBITWIDTH`: Die Support-Bitbreite wurde noch nicht gesetzt. (→ $EC\_SupportBitwidth$, S. 60)

A.3 EC-Arithmetik

Hier sind alle Funktionen zur Arithmetik auf EC-Ebene zusammengefaßt. Die Funktionen erwarten alle Punkte als Parameter und geben zumeist auch solche als Ergebnis zurück.

EC_ConvToAffine

Ein projektiver Punkt wird in seine affine Darstellung konvertiert.

Signatur

```
int EC_ConvToAffine
    (void *pDestX,void *pDestY,
    const void *pSrcX, const void *pSrcY, const void *pSrcZ)
```

Parameter

- `#pDestX`: X-Koordinate des affinen Punktes
- `#pDestY`: Y-Koordinate des affinen Punktes
- `pSrcX`: X-Koordinate des projektiven Punktes
- `pSrcY`: Y-Koordinate des projektiven Punktes
- `pSrcZ`: Z-Koordinate des projektiven Punktes

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)

EC_Double

Ein Punkt wird verdoppelt.

Signatur

```
int EC_Double
    (void *pDestX, void *pDestY, void *pDestZ,
    const void *pSrcX, const void *pSrcY, const void *pSrcZ)
```

Parameter

- `#pDestX`: X-Koordinate des Ergebnisses
- `#pDestY`: Y-Koordinate des Ergebnisses
- `#pDestZ`: Z-Koordinate des Ergebnisses
- `pSrcX`: X-Koordinate des zu verdoppelnden Punktes
- `pSrcY`: Y-Koordinate des zu verdoppelnden Punktes
- `pSrcZ`: Z-Koordinate des zu verdoppelnden Punktes

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)

EC_DoubleAffine

Ein affiner Punkt wird verdoppelt. Das Ergebnis wird in affiner Darstellung zurückgegeben.

Signatur

```
int EC_DoubleAffine
    (void *pDestX, void *pDestY, const void *pSrcX, const void *pSrcY)
```

Parameter

- `#pDestX`: X-Koordinate des Ergebnisses
- `#pDestY`: Y-Koordinate des Ergebnisses
- `pSrcX`: X-Koordinate des zu verdoppelnden Punktes
- `pSrcY`: Y-Koordinate des zu verdoppelnden Punktes

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)

EC_Mult

Der Basispunkt wird mit einem skalaren Wert multipliziert. Das Ergebnis wird in projektiver Darstellung zurückgegeben.

Signatur

```
int EC_Mult (void *pDestX, void *pDestY, void *pDestZ, const void *pSrcK)
```

Parameter

- `#pDestX`: X-Koordinate des Ergebnisses
- `#pDestY`: Y-Koordinate des Ergebnisses
- `#pDestZ`: Z-Koordinate des Ergebnisses
- `pSrcK`: Skalarer Wert, mit dem der Basispunkt multipliziert wird.

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)

EC_MultAffine

Der Basispunkt wird mit einem skalaren Wert multipliziert. Das Ergebnis wird in affiner Darstellung zurückgegeben.

Signatur

```
int EC_MultAffine (void *pDestX, void *pDestY, const void *pSrcK)
```

Parameter

- `#pDestX`: X-Koordinate des Ergebnisses
- `#pDestY`: Y-Koordinate des Ergebnisses
- `pSrcK`: Skalarer Wert, mit dem der Basispunkt multipliziert wird.

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)

EC_MultPoint

Ein beliebiger Punkt wird mit einem skalaren Wert multipliziert. Das Ergebnis wird in projektiver Darstellung zurückgegeben. Der Basispunkt wird von dieser Operation nicht beeinflußt.

Signatur

```
int EC_MultPoint
    (void *pDestX, void *pDestY, void *pDestZ, const void *pSrcX,
    const void *pSrcY, const void *pSrcZ, const void *pSrcK)
```

Parameter

- `#pDestX`: X-Koordinate des Ergebnisses
- `#pDestY`: Y-Koordinate des Ergebnisses
- `#pDestZ`: Z-Koordinate des Ergebnisses
- `pSrcX`: X-Koordinate des zu multiplizierenden Punktes
- `pSrcY`: Y-Koordinate des zu multiplizierenden Punktes
- `pSrcZ`: Z-Koordinate des zu multiplizierenden Punktes
- `pSrcK`: Skalarer Wert, mit dem der Punkt multipliziert wird.

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)

EC_MultPointAffine

Ein beliebiger Punkt wird mit einem skalaren Wert multipliziert. Das Ergebnis wird in affiner Darstellung zurückgegeben. Der Basispunkt wird von dieser Operation nicht beeinflußt.

Signatur

```
int EC_MultPointAffine
    (void *pDestX, void *pDestY, const void *pSrcX, const void *pSrcY,
    const void *pSrcK)
```

Parameter

- `#pDestX`: X-Koordinate des Ergebnisses
- `#pDestY`: Y-Koordinate des Ergebnisses
- `pSrcX`: X-Koordinate des zu multiplizierenden Punktes
- `pSrcY`: Y-Koordinate des zu multiplizierenden Punktes
- `pSrcK`: Skalarer Wert, mit dem der Punkt multipliziert wird.

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)

EC_Negate

Ein projektiver Punkt wird negiert. Dabei wird lediglich das Vorzeichen der Y-Koordinate geändert.

Signatur

```
int EC_Negate
    (void *pDestX, void *pDestY, void *pDestZ,
    const void *pSrcX, const void *pSrcY, const void *pSrcZ)
```

Parameter

- `#pDestX`: X-Koordinate des Ergebnisses
- `#pDestY`: Y-Koordinate des Ergebnisses
- `#pDestZ`: Z-Koordinate des Ergebnisses
- `pSrcX`: X-Koordinate des zu negierenden Punktes
- `pSrcY`: Y-Koordinate des zu negierenden Punktes
- `pSrcZ`: Z-Koordinate des zu negierenden Punktes

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)

EC_NegateAffine

Ein affiner Punkt wird negiert. Dabei wird lediglich das Vorzeichen der Y-Koordinate geändert.

Signatur

```
int EC_NegateAffine
    (void *pDestX, void *pDestY, const void *pSrcX, const void *pSrcY)
```

Parameter

- `#pDestX`: X-Koordinate des Ergebnisses
- `#pDestY`: Y-Koordinate des Ergebnisses
- `pSrcX`: X-Koordinate des zu negierenden Punktes
- `pSrcY`: Y-Koordinate des zu negierenden Punktes

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)

EC_Add

Zwei projektive Punkte werden addiert. Das Ergebnis wird in projektiver Darstellung zurückgegeben.

Signatur

```
int EC_Add
    (void *pDestX, void *pDestY, void *pDestZ,
    const void *pSrcX1, const void *pSrcY1, const void *pSrcZ1,
    const void *pSrcX2, const void *pSrcY2, const void *pSrcZ2)
```

Parameter

- `#pDestX`: X-Koordinate des Ergebnisses
- `#pDestY`: Y-Koordinate des Ergebnisses
- `#pDestZ`: Z-Koordinate des Ergebnisses
- `pSrcX1`: X-Koordinate des 1. Punktes
- `pSrcY1`: Y-Koordinate des 1. Punktes
- `pSrcZ1`: Z-Koordinate des 1. Punktes
- `pSrcX2`: X-Koordinate des 2. Punktes
- `pSrcY2`: Y-Koordinate des 2. Punktes
- `pSrcZ2`: Z-Koordinate des 2. Punktes

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)

EC_AddAffine

Zwei affine Punkte werden addiert. Das Ergebnis wird in affiner Darstellung zurückgegeben.

Signatur

```
int EC_AddAffine
    (void *pDestX, void *pDestY,
    const void *pSrcX1, const void *pSrcY1,
    const void *pSrcX2, const void *pSrcY2)
```

Parameter

- `#pDestX`: X-Koordinate des Ergebnisses
- `#pDestY`: Y-Koordinate des Ergebnisses
- `pSrcX1`: X-Koordinate des 1. Punktes
- `pSrcY1`: Y-Koordinate des 1. Punktes
- `pSrcX2`: X-Koordinate des 2. Punktes
- `pSrcY2`: Y-Koordinate des 2. Punktes

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)

EC_IsOnCurve

Diese Funktion überprüft, ob ein Punkt auf der Kurve liegt. Der Punkt wird dabei in projektiver Darstellung erwartet.

Signatur

```
int EC_IsOnCurve (const void *pSrcX, const void *pSrcY, const void *pSrcZ)
```

Parameter

- `pSrcX`: X-Koordinate des zu überprüfenden Punktes
- `pSrcY`: Y-Koordinate des zu überprüfenden Punktes
- `pSrcZ`: Z-Koordinate des zu überprüfenden Punktes

Rückgabewert

- `0`: Der Punkt liegt nicht auf der Kurve
- `1`: Der Punkt liegt auf der Kurve
- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)

EC_IsOnCurveAffine

Diese Funktion überprüft, ob ein Punkt auf der Kurve liegt. Der Punkt wird dabei in affiner Darstellung erwartet.

Signatur

```
int EC_IsOnCurveAffine (const void *pSrcX, const void *pSrcY)
```

Parameter

- `pSrcX`: X-Koordinate des zu überprüfenden Punktes
- `pSrcY`: Y-Koordinate des zu überprüfenden Punktes

Rückgabewert

- `0`: Der Punkt liegt nicht auf der Kurve
- `1`: Der Punkt liegt auf der Kurve
- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)

EC_ComparePoints

Es werden zwei projektive Punkte auf Gleichheit geprüft. Dabei werden die X, Y und Z-Koordinaten auf binäre Gleichheit überprüft. Da aber mehrere verschiedene projektive Darstellungen auf den selben affinen Punkt projeziert werden, kann hiermit nicht die Äquivalenz zweier Punkte nachgewiesen werden. Dazu müssen die Punkte in die eindeutige affine Darstellung konvertiert werden (→ $EC\_ConvToAffine$, S. 64). Bei den affinen Punkten kann dann die Äquivalenz geprüft werden (→ $EC\_ComparePointsAffine$, S. 71).

Signatur

```
int EC_ComparePoints
    (const void *pSrcX0, const void *pSrcY0, const void *pSrcZ0,
    const void *pSrcX1, const void *pSrcY1, const void *pSrcZ1)
```

Parameter

- `pSrcX0`: X-Koordinate des ersten Punktes
- `pSrcY0`: Y-Koordinate des ersten Punktes
- `pSrcZ0`: Z-Koordinate des ersten Punktes
- `pSrcX1`: X-Koordinate des zweiten Punktes
- `pSrcY1`: Y-Koordinate des zweiten Punktes
- `pSrcZ1`: Z-Koordinate des zweiten Punktes

Rückgabewert

- `0`: Die Punkte sind ungleich
- `1`: Die Punkte sind gleich
- `ERR_NO_CBITWIDTH`: Die Support-Bitbreite wurde noch nicht gesetzt. (→ $EC\_SupportBitwidth$, S. 60)

EC_ComparePointsAffine

Es werden zwei affine Punkte auf Gleichheit geprüft. Dabei werden die X und Y-Koordinaten auf binäre Gleichheit überprüft. Da die affine Darstellung eindeutig ist, kann so die Äquivalenz zweier Punkte gezeigt werden.

Signatur

```
int EC_ComparePointsAffine
    (const void *pSrcX0, const void *pSrcY0,
    const void *pSrcX1, const void *pSrcY1)
```

Parameter

- `pSrcX0`: X-Koordinate des ersten Punktes
- `pSrcY0`: Y-Koordinate des ersten Punktes
- `pSrcX1`: X-Koordinate des zweiten Punktes
- `pSrcY1`: Y-Koordinate des zweiten Punktes

Rückgabewert

- `0`: Die Punkte sind ungleich
- `1`: Die Punkte sind gleich
- `ERR_NO_CBITWIDTH`: Die Support-Bitbreite wurde noch nicht gesetzt. (→ $EC\_SupportBitwidth$, S. 60)

EC_CopyPoint

Ein Punkt wird kopiert. Dabei werden die einzelnen Koordinaten in neue Speicherbereiche kopiert. Der Punkt kann in projektiver Darstellung vorliegen.

Signatur

```
int EC_CopyPoint
    (void *pDestX, void *pDestY, void *pDestZ,
    const void *pSrcX, const void *pSrcY, const void *pSrcZ)
```

Parameter

- `#pDestX`: Speicherbereich, in den die X-Koordinate kopiert wird.
- `#pDestY`: Speicherbereich, in den die Y-Koordinate kopiert wird.
- `#pDestZ`: Speicherbereich, in den die Z-Koordinate kopiert wird.
- `pSrcX`: X-Koordinate des Punktes
- `pSrcY`: Y-Koordinate des Punktes
- `pSrcZ`: Z-Koordinate des Punktes

Rückgabewert

- `[1..3]`: Anzahl Werte, die nicht kopiert werden konnten.
- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_BITWIDTH`: Die Bitbreite wurde noch nicht gesetzt. (→ $EC\_Bitwidth$, S. 57)

EC_CopyPointAffine

Ein Punkt wird kopiert. Dabei werden die einzelnen Koordinaten in neue Speicherbereiche kopiert. Der Punkt muß in affiner Darstellung vorliegen.

Signatur

```
int EC_CopyPointAffine
    (void *pDestX, void *pDestY, const void *pSrcX, const void *pSrcY)
```

Parameter

- `#pDestX`: Speicherbereich, in den die X-Koordinate kopiert wird.
- `#pDestY`: Speicherbereich, in den die Y-Koordinate kopiert wird.
- `pSrcX`: X-Koordinate des Punktes
- `pSrcY`: Y-Koordinate des Punktes

Rückgabewert

- `interval12`: Anzahl Werte, die nicht kopiert werden konnten.
- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_BITWIDTH`: Die Bitbreite wurde noch nicht gesetzt. (→ $EC\_Bitwidth$, S. 57)

A.4 Konvertierungen

Die Konvertierungsroutinen dienen der Umwandlung der Zahldarstellungen von der von allen Funktionen zur Kommunikation verwendeten Binärdarstellung in eine Menschenlesbare Form, sowie der Gegenrichtung.

EC_ConvBinToHexstr

Konvertiert eine Binärzahl in lesbare Hexadezimaldarstellung.

Signatur

```
int EC_ConvBinToHexstr (char *pcDest, const void *pSrc)
```

Parameter

- `#pcDest`: In diesem Parameter wird das Ergebnis gespeichert. Die Größe dieses Speicherbereichs muß mindestens $Bitwidth/4 + 3$ Bytes betragen.
- `pSrc`: In diesem Parameter wird die umzurechnende Binärzahl übergeben.

Rückgabewert

- `i`: Länge der Hexadezimaldarstellung in Bytes
- `ERR_NO_CBITWIDTH`: Die Support-Bitbreite wurde noch nicht gesetzt. (→ $EC\_SupportBitwidth$, S. 60)

EC_ConvBinToDecstr

Konvertiert eine Binärzahl in lesbare Dezimaldarstellung.

Signatur

```
int EC_ConvBinToDecstr (char *pcDest, const void *pSrc)
```

Parameter

- `#pcDest`: In diesem Parameter wird das Ergebnis gespeichert.
- `pSrc`: In diesem Parameter wird die umzurechnende Binärzahl übergeben.

Rückgabewert

- `i`: Länge der Dezimaldarstellung in Bytes
- `ERR_NO_CBITWIDTH`: Die Support-Bitbreite wurde noch nicht gesetzt. (→ $EC\_SupportBitwidth$, S. 60)

EC_ConvBinToBinstr

Konvertiert eine Binärzahl in eine lesbare, binäre Stringdarstellung.

Signatur

```
int EC_ConvBinToBinstr (char *pcDest, const void *pSrc)
```

Parameter

- `#pcDest`: In diesem Parameter wird das Ergebnis gespeichert. Die Größe dieses Speicherbereichs muß mindestens $Bitwidth + 1$ Bytes betragen.
- `pSrc`: In diesem Parameter wird die umzurechnende Binärzahl übergeben.

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_CBITWIDTH`: Die Support-Bitbreite wurde noch nicht gesetzt. (→ $EC\_SupportBitwidth$, S. 60)

EC_ConvHexstrToBin

Eine lesbare Hexadezimalzahl wird in Binärdarstellung umgewandelt.

Signatur

```
int EC_ConvHexstrToBin (void *pDest, const char *pcSrc)
```

Parameter

- `#pcDest`: In diesem Parameter wird das Ergebnis gespeichert.
- `pSrc`: Dieser Parameter enthält die umzurechnende Zahl

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_CBITWIDTH`: Die Support-Bitbreite wurde noch nicht gesetzt. (→ $EC\_SupportBitwidth$, S. 60)

EC_ConvDecstrToBin

Eine lesbare Dezimalzahl wird in Binärdarstellung umgewandelt.

Signatur

```
int EC_ConvDecstrToBin (void *pDest, const char *pcSrc)
```

Parameter

- `#pcDest`: In diesem Parameter wird das Ergebnis gespeichert.
- `pSrc`: Dieser Parameter enthält die umzurechnende Zahl

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_CBITWIDTH`: Die Support-Bitbreite wurde noch nicht gesetzt. (→ $EC\_SupportBitwidth$, S. 60)

A.5 Information

In diesem Abschnitt sind alle Funktionen zusammengefaßt, die Informationen über die DLL selbst, oder über Berechnungsdaten liefern.

EC_GetBuildInfo

Gibt einen String zurück, der Compilerinformationen zur DLL enthält.

Signatur

```
int EC_GetBuildInfo (char *pcDest, int iDestLength)
```

Parameter

- `#pcDest`: Speicherbereich, in den der String geschrieben wird.
- `iDestLength`: Größe von *pcDest*

Rückgabewert

- `i`: Länge des zurückgegebenen Strings

EC_GetHardwareConfiguration

Gibt die aktuelle Konfiguration der Hardware zurück.

Signatur

```
int EC_GetHardwareConfiguration
    (int *piHardwareBitwidth, char *pcFilename, int iFilenameLength,
    int *piFpgaClk)
```

Parameter

- `#piHardwareBitwidth`: Hierin wird die Hardware-Bitbreite zurückgegeben.
- `#pcFilename`: Speicherbereich, in den der Dateiname des Hardware-Designs gespeichert wird.
- `iFilenameLength`: Größe von *pcFilename*
- `#piFpgaClk`: Hierin wird die aktuelle Taktfrequenz zurückgegeben.

Rückgabewert

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_HARDWARE`: Die Hardwareunterstützung ist nicht verfügbar.

EC_GetHardwareSupport

Gibt den aktuellen Status der Hardware-Unterstützung zurück.

Signatur

```
int EC_GetHardwareSupport ( )
```

Rückgabewert

- `HWSTAT_NULL`: Es ist nicht bekannt, ob Hardware-Unterstützung vorhanden ist.
- `HWSTAT_NOHW`: Es ist keine Hardware-Unterstützung vorhanden.
- `HWSTAT_READY`: Der ECP ist fertig initialisiert und kann verwendet werden.
- `HWSTAT_NOSUPPORT`: Die DLL unterstützt in dieser Version keine Hardware-Berechnungen.
- `HWSTAT_DISABLED`: Die Hardware-Unterstützung wurde manuell deaktiviert.
- `ERR_NO_HARDWARE`: Die Hardwareunterstützung ist nicht verfügbar.

EC_GetBitwidth

Die Funktion ermittelt die eingestellte Bitbreite. Alle FF-Elemente müssen mit dieser Anzahl Bits darstellbar sein.

Signatur

```
int EC_GetBitwidth ( )
```

Rückgabewert

- `[0..512]`: verwendete Bitbreite
- `ERR_NO_BITWIDTH`: Die Bitbreite wurde noch nicht gesetzt. (→ $EC\_Bitwidth$, S. 57)

EC_GetBytewidth

Die Funktion ermittelt die Anzahl Bytes, die für ein FF-Element benötigt werden.

Signatur

```
int EC_GetBytewidth ( )
```

Rückgabewert

- `[0..64]`: Anzahl Bytes für ein FF-Element
- `ERR_NO_BITWIDTH`: Die Bitbreite wurde noch nicht gesetzt. (→ $EC\_Bitwidth$, S. 57)

EC_GetBitwidthHW

Die Hardware kann unter Umständen mit einer höheren Bitbreite als angefordert initialisiert werden. Dies geschieht vollkommen Automatisch und kann nicht beeinflußt werden. Diese Bitbreite kann mit dieser Funktion abgefragt werden.

Signatur

```
int EC_GetBitwidthHW ( )
```

Rückgabewert

- `[0..512]`: in Hardware verwendete Bitbreite
- `ERR_NO_BITWIDTH`: Die Bitbreite wurde noch nicht gesetzt. (→ $EC\_Bitwidth$, S. 57)

EC_GetSupportBitwidth

Diese Funktion gibt die Support-Bitbreite zurück (→ $EC\_SupportBitwidth$, S. 60).

Signatur

```
int EC_GetSupportBitwidth ( )
```

Rückgabewert

- `[0..512]`: in Support-Bitbreite
- `ERR_NO_CBITWIDTH`: Die Support-Bitbreite wurde noch nicht gesetzt. (→ $EC\_SupportBitwidth$, S. 60)

EC_GetSupportBytewidth

Diese Funktion gibt die Support-Bytegröße zurück (→ $EC\_SupportBitwidth$, S. 60).

Signatur

```
int EC_GetSupportBytewidth ( )
```

Rückgabewert

- `[0..64]`: Anzahl Bytes
- `ERR_NO_BITWIDTH`: Die Bitbreite wurde noch nicht gesetzt. (→ $EC\_Bitwidth$, S. 57)

A.6 Protokollierung

Die DLL kann alle Operationen in einer Log-Datei speichern. Dadurch wird allerdings die Performance deutlich beeinträchtigt, da mitunter eine erhebliche Menge Protokollausgaben auf die Festplatte geschrieben werden. Die Protokollierung erleichtert allerdings die Fehlersuche bei der Entwicklung neuer Anwendungen.

EC_LogStart

Startet die Protokollierung.

Signatur

```
int EC_LogStart (const char *pcSrcFilename, const int iAppend)
```

Parameter

- `pcSrcFilename`: Name der Datei, in die die Protokollausgaben geschrieben werden.
- `iAppend`: 0: neue Datei anlegen, 1: an bestehende Datei anfügen.

Rückgabewert

- `0`: Kein Fehler aufgetreten
- `1`: Die Datei konnte nicht für Schreibzugriffe geöffnet werden.

EC_LogStop

Beendet die Protokollierung.

Signatur

```
int EC_LogStop ( )
```

Rückgabewert

- `0`: Kein Fehler aufgetreten
- `1`: Die Datei konnte nicht geschlossen werden.

EC_LogGetAutologging

Gibt den Status des automatischen Loggings zurück.

Signatur

```
int EC_LogGetAutologging ( )
```

Rückgabewert

- `0`: Autologging aktiviert.
- `1`: Autologging deaktiviert.

EC_LogGetAutologgingFile

Gibt den Namen der Datei, in die automatisch protokolliert wird, zurück.

Signatur

```
int EC_LogGetAutologgingFile
    (char *pcDestFilename, const int iDestFilenameLength)
```

Parameter

- `#pcDestFilename`: Speicherbereich, in dem der Dateiname zurückgegeben wird
- `iDestFilenameLength`: Größe von *pcDestFilename*

Rückgabewert

- `0`: Eine neue Datei wird angelegt.
- `1`: Protokollausgaben werden an eine bestehende Datei angefügt.

EC_LogSetAutologging

Schaltet automatisches Logging ein oder aus.

Signatur

```
int EC_LogSetAutologging (int iEnabled)
```

Parameter

- `iEnabled`: 0: Autlogging deaktiv, 1: Autlogging aktiv

Rückgabewert

- `0`: Kein Fehler aufgetreten.
- `1`: Die Einstellung konnte nicht gespeichert werden.

EC_LogSetAutologgingFile

Gibt den Namen der Datei an, in die automatisch protokolliert wird.

Signatur

```
int EC_LogSetAutologgingFile (const char *pcSrcFilename, const int iAppend)
```

Parameter

- `pcSrcFilename`: In diese Datei wird protokolliert.
- `iAppend`: 0: neue Datei anlegen, 1: an bestehende Datei anfügen.

Rückgabewert

- `0`: Kein Fehler aufgetreten.
- `1,2`: Die Einstellungen konnten nicht gespeichert werden.

A.7 Einstellungen

Mit diesen Funktionen wird die DLL konfiguriert. Hiermit wird definiert, bei welcher Bitbreite welches Hardware-Design auf die Karte geladen werden soll und wie hoch die Taktfrequenz ist. Außerdem werden diese Funktionen verwendet, um bei der Initialisierung mit einer bestimmten Bitbreite passende Konfigurationsdaten zu ermitteln.

EC_GetNextStoredBitwidth

Die Funktion gibt die nächstgrößere Bitbreite zurück, für die eine Konfiguration existiert.

Signatur

```
int EC_GetNextStoredBitwidth (const int iCurrentBitwidth)
```

Parameter

- `iCurrentBitwidth`: Bitbreite, zu der die nächstgrößere gesucht wird.

Rückgabewert

- `-1`: Keine größere Bitbreite gefunden.
- `[1..512]`: Wert der größeren Bitbreite

EC_GetStoredBitwidth

Gibt die Konfigurationsdaten für eine bestimmte Bitbreite zurück.

Signatur

```
int EC_GetStoredBitwidth
    (const int iBitwidth, char *pcDestFilename,
    const int iDestFilenameLength, int *piFPGAClock)
```

Parameter

- `iBitwidth`: Bitbreite, für die die Konfigurationsdaten abgefragt werden sollen
- `#pcDestFilename`: Speicherbereich, in den der Name der Bitstream-Datei gespeichert wird
- `iDestFilenameLength`: Größe von *pcDestFilename*
- `#piFPGAClock`: Hierin wird die Taktfrequenz zurückgegeben

Rückgabewert

- `0`: Kein Fehler aufgetreten
- `>0`: Es ist ein Fehler aufgetreten

EC_GetCompatibleBitwidth

Ermittelt eine Bitbreite, für die eine Konfiguration existiert und die kompatibel zu der von außen gesetzten Bitbreite ist.

Signatur

```
int EC_GetCompatibleBitwidth (const int iBitwidth)
```

Parameter

- `iBitwidth`: Zu dieser Bitbreite wird eine kompatible Bitbreite gesucht.

Rückgabewert

- `-1`: Keine passende Konfiguration gefunden
- `>=iBitwidth`: kompatible Bitbreite mit aktiver Konfiguration

EC_SetStoredBitwidth

Speichert Konfigurationsdaten für eine Bitbreite.

Signatur

```
int EC_SetStoredBitwidth
    (const int iBitwidth, const char *pcSrcFilename, const int iFPGAClock)
```

Parameter

- `iBitwidth`: Bitbreite, für die die Konfiguration gespeichert werden soll
- `pcSrcFilename`: Dateiname des Hardware-Designs
- `iFPGAClock`: Taktfrequenz, mit der der FPGA-Chip getaktet werden soll

Rückgabewert

- `0`: Kein Fehler aufgetreten
- `>0`: Es ist ein Fehler aufgetreten

EC_DeleteStoredBitwidth

Löscht alle Konfigurationsdaten zu einer Bitbreite.

Signatur

```
int EC_DeleteStoredBitwidth (const int iBitwidth)
```

Parameter

- `iBitwidth`: Zu dieser Bitbreite sollen die Konfigurationsdaten gelöscht werden

Rückgabewert

- `0`: Kein Fehler aufgetreten
- `>0`: Es ist ein Fehler aufgetreten

EC_GetFeature

Ermittelt, ob ein bestimmtes Feature in der Konfiguration verfügbar ist.

Signatur

```
int EC_GetFeature (const int iBitwidth, const int iFeature)
```

Parameter

- `iBitwidth`: Zu dieser Bitbreite wird ein Feature abgefragt
- `iFeature`: Dieses Feature wird abgefragt

Rückgabewert

- `-1`: Es ist ein Fehler aufgetreten
- `0`: Das Feature ist nicht aktiv
- `1`: Das Feature ist aktiv

EC_SetFeature

Speichert, ob ein bestimmtes Feature in der Konfiguration verfügbar ist.

Signatur

```
int EC_SetFeature
    (const int iBitwidth, const int iFeature, const int iEnabled)
```

Parameter

- `iBitwidth`: Zu dieser Bitbreite wird ein Feature gespeichert
- `iFeature`: Dieses Feature wird gespeichert
- `iEnabled`: 0: das Feature wird deaktiviert, 1: das Feature wird aktiviert

Rückgabewert

- `0`: Kein Fehler aufgetreten
- `>0`: Es ist ein Fehler aufgetreten

Anhang B

Rückgabekonstanten von Ecc_gfp.dll

B.1 Fehlercodes

- `ERR_OK`: Kein Fehler aufgetreten
- `ERR_NO_BITWIDTH`: Die Bitbreite wurde noch nicht gesetzt. (→ $EC\_Bitwidth$, S. 57)
- `ERR_NO_CBITWIDTH`: Die Support-Bitbreite wurde noch nicht gesetzt. (→ $EC\_SupportBitwidth$, S. 60)
- `ERR_NO_PRIME`: Die Primzahl wurde noch nicht gesetzt. (→ $EC\_LoadPrime$, S. 57)
- `ERR_NO_CURVE`: Die Kurve wurde noch nicht gesetzt. (→ $EC\_LoadCurve$, S. 58)
- `ERR_NO_POINT`: Der Basispunkt wurde noch nicht gesetzt. (→ $EC\_LoadPoint$, S. 57)
- `ERR_INVALID_BITWIDTH`: Die Bitbreite ist nicht zulässig. Sie muß im Bereich [1..512] liegen.
- `ERR_INVALID_HAMMING`: Das Hamminggewicht ist nicht zulässig. Es muß im Bereich [1..512] liegen.
- `ERR_NO_HARDWARE`: Die Hardwareunterstützung ist nicht verfügbar.
- `ERR_UNKNOWN`: Ein unbekannter Fehler ist aufgetreten.

B.2 Hardwarestatus

- `HWSTAT_NULL`: Es ist nicht bekannt, ob Hardware-Unterstützung vorhanden ist.
- `HWSTAT_NOHW`: Es ist keine Hardware-Unterstützung vorhanden.
- `HWSTAT_READY`: Der ECP ist fertig initialisiert und kann verwendet werden.
- `HWSTAT_NOSUPPORT`: Die DLL unterstützt in dieser Version keine Hardware-Berechnungen.
- `HWSTAT_DISABLED`: Die Hardware-Unterstützung wurde manuell deaktiviert.

Tabelle B.1: Werte der Fehlerkonstanten

Fehler	*Wert*
ERR_OK	0
ERR_NO_BITWIDTH	1030
ERR_NO_PRIME	1031
ERR_NO_CURVE	1032
ERR_NO_POINT	1033
ERR_NO_CBITWIDTH	1034
ERR_INVALID_BITWIDTH	1050
ERR_INVALID_HAMMING	1051
ERR_NO_HARDWARE	1060
ERR_UNKNOWN	2000

Tabelle B.2: Statuscodes der Hardware

Status	*Wert*
HWSTAT_NULL	0
HWSTAT_NOHW	1
HWSTAT_READY	2
HWSTAT_NOSUPPORT	3
HWSTAT_DISABLED	4

Anhang C

ECPInterface.java

Vorbemerkungen

Aufgabe und Beschreibung der Klasse

Die Klasse *ECPInterface.java* bildet die JAVA Seite der Schnittstelle zum native Code in der *ECPInterface.dll*. Sie stellt alle auf der Hardware berechenbaren Operationen der JAVA Umgebung zur Verfügung. Da es sich bei den Rückgabewerten der arithmetischen Methoden in der *ECPInterface.java* ausschließlich um Punkte handelt, benötigt die konkrete Implementierung bzw. etwaige notwendige Änderungen in der Darstellung (big-endian, little-endian) keine Beachtung von Seite des JAVA Providers. Alle Änderungen diesbezüglich sind alleinig in der *ECPInterace.java* vorzunehmen.
Eine weitere Besonderheit dieser Klasse ist es, dass sich nur höchstens eine Instanz von ihr in einer JAVA Virtual Machine befinden kann. Dies ist wichtig, da sonst versucht werden könnte, über mehrere Interfaces auf eine Karte zuzugreifen.
Jede hier aufgeführte Methode (ausser der internen Konvertierungsmethoden natürlich) entsprechen ihrer korrespondierenden Funktion in der *ECPInterface.dll*.

C.1 Support Methoden

dllSupportAvailable

Überprüft, falls noch nicht geschehen, ob die *ECPInterface.dll* geladen werden konnte. Sollte dies der Fall sein, wird gleichzeitig überprüft ob DLL Unterstützung seitens der *ecc_gfp.dll* vorhanden ist.

Signatur

```
public boolean dllSupportAvailable()
```

getBI

Führt die notwendigen Konvertierung der ByteArrays zu BigInteger durch. Die JAVA Darstellung eines BigIntegers erfolgt in Big-Endian Darstellung und im 2er Komplement.

Signatur

```
private BigInteger getBI(byte[] byteArray)
```

Parameter

- `byteArray:` Ein ByteArray beliebiger Länge, dass zu einem BigInteger konvertiert werden soll.

Rückgabewert

- Das dem ByteArray entsprechende BigInteger

logStart

Startet den Logging Vorgang der *ecc_gfp.dll*

Signatur

```
public int logStart(String filename, int append)
```

Parameter

- `filename:` Name der zu schreibenden Datei
- `append:`
  - 0 = wenn eine eventuell vorhandene Datei überschrieben werden soll
  - 1 = wenn das Log an eine eventuell vorhandene Datei gehängt werden soll

Rückgabewert

- Statusmeldung der DLL

logStop

Stoppt den Logging Vorgang der *ecc_gfp.dll*

Signatur

```
public void logStop()
```

Rückgabewert

- Statusmeldung der DLL

pointIsLoaded

Überprüft, ob die BigInteger x1,y1 und z1 identisch sind mit den BigIntegern x2,y2 und z2. Diese Methode wird benötigt, da die *ECPInterface.java* Klasse um unnötigen Kommunikationsaufwand zu vermeiden, sich intern merkt, welcher Punkt sich auf der Karte bzw. in der darunter liegenden DLL befindet.

Signatur

```
private boolean
        pointIsLoaded(BigInteger x1, BigInteger y1, BigInteger z1,
                      BigInteger x2, BigInteger y2, BigInteger z2)
```

Parameter

- `x1:` X-Koordinate des ersten Punkts
- `y1:` Y-Koordinate des ersten Punkts
- `z1:` Z-Koordinate des ersten Punkts
- `x2:` X-Koordinate des zweiten Punkts
- `y2:` Y-Koordinate des zweiten Punkts
- `z2:` Z-Koordinate des zweiten Punkts

Rückgabewert

- `true:` die Koordinaten und somit die Punkte sind identisch
- `false:` die Koordinaten sind nicht identisch

status

Gibt einen Status Code zurück, der Rückschlüße auf den momentanen Interface bzw. DLL Status zuläßt.

Signatur

```
public int status()
```

Rückgabewert

- `0:` falls das Interface und die darunterliegende DLL benutzt werden können

setBitwidth

Setzt die Bitbreite mit der gerechnet werden soll

Signatur

```
public int setBitwidth(int bitwidth)
```

Parameter

- `bitwidth`: zu setzende Bitbreite

Rückgabewert

- Statusmeldung der DLL

C.2 Arithmetische Methoden

add

Führt eine Addition der übergebenen Punkten durch.

Signatur

```
public Point add(Point a, Point b)
```

Parameter

- `a:` Instanz des Objektes Point
- `b:` Instanz des Objektes Point

Rückgabewert

- Ergebnispunkt als Instanz des Objekts Point

loadCurve

Übergibt die Kurvenparameter a und b und somit eine Kurve an die darunter liegenden DLLs.

Signatur

```
public int loadCurve(BigInteger a, BigInteger b)
```

Parameter

- `a:` Kurvenparameter a
- `b:` Kurvenparameter b

Rückgabewert

- Statusmeldung der DLL

loadPoint

Übergibt einen Punkt in projektiver Darstellung an die darunter liegenden DLLs.

Signatur

```
public int loadPoint(BigInteger mX, BigInteger mY, BigInteger mZ)
```

Parameter

- `mX:` X-Koordinate der projektive Darstellung des Punktes
- `mY:` Y-Koordinate der projektive Darstellung des Punktes
- `mZ:` Z-Koordinate der projektive Darstellung des Punktes

Rückgabewert

- Statusmeldung der DLL

loadPoint

Übergibt einen Punkt in affiner Darstellung an die darunter liegenden DLLs.

Signatur

```
public int loadPoint(BigInteger mX, BigInteger mY)
```

Parameter

- `mX`: X-Koordinate der affinen Darstellung des Punktes
- `mY`: Y-Koordinate der affinen Darstellung des Punktes

Rückgabewert

- Statusmeldung der DLL

loadPrime

Übergibt die zur Kurve gehörende Primzahl an die darunter liegenden DLLs.

Signatur

```
public int loadPrime(BigInteger q)
```

Parameter

- `q`: Primzahl

Rückgabewert

- Statusmeldung der DLL

mult

Führt eine Multiplikation des übergebenen Punktes mit k durch.

Signatur

```
public Point mult(Point a, BigInteger k)
```

Parameter

- `a:` Punkt, der multipliziert werden soll
- `k:` Skalar, der mit dem Punkt multipliziert werden soll

Rückgabewert

- Ergebnispunkt als Instanz des Objekts Point

mult

Führt eine Multiplikation des in der DLL vorgehaltenen Basispunktes mit k durch.

Signatur

```
public Point mult(BigInteger k)
```

Parameter

- `k:` Skalar, der mit dem Punkt multipliziert werden soll

Rückgabewert

- Ergebnispunkt als Instanz des Objekts Point

negate

Führt eine Negation des übergebenen Punktes durch.

Signatur

```
public Point negate(Point a)
```

Parameter

- `a:` Punkt, der negiert werden sol

Rückgabewert

- Ergebnispunkt als Instanz des Objekts Point

onCurve

Führt eine Check durch, ob der übergenen Punkt auf der Kurve liegt.

Signatur

```
public boolean onCurve(Point a)
```

Parameter

- `a:` Punkt, der negiert werden sol

Rückgabewert

- `true:` der Punkt liegt auf der Kurve
- `false:` der Punkt liegt nicht auf der Kurve

Anhang D

ECPInterface.dll

Vorbemerkungen

Aufgabe der DLL

Die *ECPInterface.dll* dient als Schnittstelle zwischen dem native Code in der *ecc_gfp.dll* und dem in JAVA geschriebenen *FlexiProvider*. Unter den Aufgaben der DLL fällt unter anderem die Konvertierung von C-Datentypen in ihre JAVA Äquivalente.

Speicherbereiche

Die DLL verwendet zur Übertragung der Daten zwischen JAVA und der ecc_gfp.dll *unsigned long* Pointer. Die Größe der Speicherbereiche wird über die Bitbreite gesteuert. Bei erstmaliger Initialisierung der DLL werden alle Speicherbereich neu allokiert und angelegt, während der Nutzung ggf. neu allokiert und bei Beendigung frei gegegeben.

Erläuterung der Speicherbereiche

Die globalen Variablen für das Zwischenspeichern von Daten folgen der unten stehenden Namenskonvention.

_gl<Transferart> <Parametertyp> <Bezeichner>

Hierbei steht _gl für *globale Variable*.

<Transferart>

- *Load* Transfer von Daten aus JAVA nach C
- *Return* Transfer von Daten aus C nach JAVA

<Parametertyp>

- *Point* Beinhaltet einen Parameter eines Punktes
- *Param* Beinhaltet einen der Kurvenparameter
- *Mult* Multiplikativer Wert

<Bezeichner>

- *X0* X-Koordinate Punkt 0
- *Y0* Y-Koordinate Punkt 0
- *Z0* Z-Koordinate Punkt 0
- *X1* X-Koordinate Punkt 1
- *Y1* Y-Koordinate Punkt 1
- *Z1* Z-Koordinate Punkt 1
- *A* Kurvenparameter A
- *B* Kurvenparameter B
- *Q* Primzahl Q
- *Array* nicht näher spezifiziertes Array
- *Compa* Array, gegen das verglichen werden soll
- *Result* wird ausschließlich zur internen Konvertierung gebraucht

D.1 Konvertierungen

jStringToLPSTR

Konvertiert eine JAVA String Struktur in die äquivalente Struktur in C

Signatur

```
LPSTR jStringToLPSTR
      (JNIEnv *env, jstring convString)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `convString`: Zu konvertierende JAVA jstring Struktur

Rückgabewert

- `LPSTR`: Ergebnis der Konvertierung

jByteToLongArray

Konvertiert eine JAVA jByte Struktur und legt sie unter dem übergebenen resultArray ab

Signatur

```
void jbyteToLongArray
      (JNIEnv *env, unsigned long* resultArray, jbyteArray javaSourceArray)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `resultArray`: Speicherbereich in welche das jbyteArray konvertiert abgelegt wird
- `javaSourceArray`: zu konvertierendes jbyteArray

LongTojByteArray

Konvertiert den Speicherinhalt eines Long Array Pointers in eine JAVA jbyteArray Struktur

Signatur

```
void LongTojByteArray
      (JNIEnv *env, unsigned long* LongArray, jbyteArray JavaResult)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `LongArray`: Pointer auf die zu konvertierenden Daten
- `JavaResult`: Ergebnis der Konvertierung

D.2 Interne Funktionen

DllMain

Einzelheiten dieser Funktion können der Dokumentation zu MS Visual Studio 6.0 entnommen werden

Signatur

```
BOOL WINAPI DllMain
      (HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
```

getByteWidth

Fragt die benötigte Bitbreite für die Parameter bei der ecc_gfp.dll ab

Signatur

```
int getByteWidth()
```

Rückgabewert

- `int`: Momentane Byte Width

resizeGlobals

Allokiert die Speicherbereiche für die globalen Variablen neu

Signatur

```
void resizeGlobals()
```

emptyGlobals

Setzt alle Werte in den Speicherbereichen für die globalen Variablen auf 0

Signatur

```
void emptyGlobals
    (int emptyLevel)
```

Parameter

- `emptyLevel`: Bestimmt welche Speicherbereiche zurückgesetzt werden
  - 0 = alle Bereiche werden zurückgesetzt
  - 1 = alles bis auf der Speicherbereich für die Primzahl (_glLoadParamQ) wird zurückgesetzt
  - 2 = alles bis auf -1- und die Kurvenparameter (_glLoadParamA,_glLoadParamB) wird zurückgesetzt
  - 3 = nur _glResult wird zurückgesetzt

clearGlobals

Gibt den Speicher für alle globalen Variablen frei. Diese Funktion wird bei Beendigung der DLL aufgerufen

Signatur

```
void clearGlobals()
```

D.3 Nach JAVA exportierte Funktionen

Java_de_flexiprovider_ECPInterface_EC_1Support

Allokiert die Speicherbereiche für die globalen Variablen neu

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1Support
```

Rückgabewert

- `0`: Hardware ist vorhanden
- `1`: Hardware ist nicht vorhanden

Java_de_flexiprovider_ECPInterface_EC_1Bitwidth

Allokiert die Speicherbereiche für die globalen Variablen neu

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1Bitwidth
          (JNIEnv *env, jobject obj, jint loadBitwidth)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt
- `loadBitwidth`: Zu landende Bitbreite

Java_de_flexiprovider_ECPInterface_EC_1LoadCurve

Reicht die Kurvenparameter weiter an die *ecc_gfp.dll*

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1LoadCurve
          (JNIEnv *env, jobject obj, jbyteArray loadA, jbyteArray loadB)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt
- `loadA`: Kurvenparameter A
- `loadB`: Kurvenparameter B

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1Init

Diese Funktion reicht Primzahl, Kurvenparameter und den Basispunkt weiter an die *ecc_gfp.dll*.

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1Init
          (JNIEnv *env, jobject obj, jbyteArray loadPrime,
          jbyteArray loadA, jbyteArray loadB,jbyteArray loadX0,
          jbyteArray loadY0, jbyteArray loadZ0)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt
- `loadA`: Kurvenparameter A
- `loadB`: Kurvenparameter B
- `loadX0`: Parameter X des Basispunkts
- `loadY0`: Parameter Y des Basispunkts
- `loadZ0`: Parameter Z des Basispunkts

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1LoadPrime

Diese Funktion reicht die Primzahl weiter an die *ecc_gfp.dll*.

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1LoadPrime
          (JNIEnv *env, jobject obj, jbyteArray loadPrime)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt
- `loadPrime`: Primzahl

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1GetBytewidth

Fragt die aktuelle Bytebreite von der *ecc_gfp.dll* ab und reicht diese weiter

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1GetBytewidth
          (JNIEnv *env, jobject obj)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1GetBitwidth

Fragt die aktuelle Bitbreite von der *ecc_gfp.dll* ab und reicht diese weiter

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1GetBitwidth
          (JNIEnv *env, jobject obj)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1GetBitwidthHW

Fragt die aktuelle von der Hardware - wenn eine vorhanden ist - gesetzte Bitbreit von der *ecc_gfp.dll* ab und reicht diese weiter

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1GetBitwidthHW
          (JNIEnv *env, jobject obj)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1LoadPointAffine

Lädt einen affinen Punkt in die *ecc_gfp.dll*

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1LoadPointAffine
          (JNIEnv *env, jobject obj, jbyteArray loadX0, jbyteArray loadY0)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt
- `loadX0`: X-Koordinate des Punktes
- `loadY0`: Y-Koordinate des Punktes

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1MultAffine

Führt eine Multiplikation mit dem in der *ecc_gfp.dll* DLL gespeicherten Punkt und dem Skalar K durch. Das Ergebnis wird affin zurück gegeben.

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1MultAffine
          (JNIEnv *env, jobject obj, jbyteArray returnX0,
           jbyteArray returnY0, jbyteArray loadK)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt
- `#returnX0`: X-Koordinate des Ergbnis-Punktes
- `#returnY0`: Y-Koordinate des Ergbnis-Punktes

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1MultPointAffine

Führt eine Multiplikation mit dem übergebenen Punkt und dem Skalar durch und gibt das Ergebnis in affinen Koordinaten zurück.

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1MultPointAffine
          (JNIEnv *env, jobject obj, jbyteArray returnX0, jbyteArray returnY0,
          jbyteArray loadX0, jbyteArray loadY0, jbyteArray loadK)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt
- `loadX0`: X-Koordinate des Punktes
- `loadY0`: Y-Koordinate des Punktes
- `#returnX0`: X-Koordinate des Ergbnis-Punktes
- `#returnY0`: Y-Koordinate des Ergbnis-Punktes

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1NegateAffine

Java_de_flexiprovider_ECPInterface_EC_1NegateAffine

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1MultPointAffine
          (JNIEnv *env, jobject obj, jbyteArray returnX0, jbyteArray returnY0,
           jbyteArray loadX0, jbyteArray loadY0)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt
- `loadX0`: X-Koordinate des Punktes
- `loadY0`: Y-Koordinate des Punktes
- `#returnX0`: X-Koordinate des Ergbnis-Punktes
- `#returnY0`: Y-Koordinate des Ergbnis-Punktes

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1AddAffine

Addiert die beiden übergebenen affinen Punkte und gibt die affine Koordinaten des Ergebnispunktes zurück.

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1AddAffine
          (JNIEnv *env, jobject obj, jbyteArray returnX0, jbyteArray returnY0,
          jbyteArray loadX0, jbyteArray loadY0, jbyteArray loadX1,
          jbyteArray loadY1)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt
- `loadX0`: X-Koordinate des ersten Punktes
- `loadY0`: Y-Koordinate des ersten Punktes
- `loadX1`: X-Koordinate des zweiten Punktes
- `loadY1`: Y-Koordinate des zweiten Punktes
- `#returnX0`: X-Koordinate des Ergbnis-Punktes
- `#returnY0`: Y-Koordinate des Ergbnis-Punktes

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1IsOnCurveAffine

Überprüft ob der übergebene Punkt auf der in der *ecc_gfp.dll* gespeicherten Kurve liegt oder nicht.

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1IsOnCurveAffine
          (JNIEnv *env, jobject obj, jbyteArray loadX0, jbyteArray loadY0)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt
- `loadX0`: X-Koordinate des Punktes
- `loadY0`: Y-Koordinate des Punktes

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1LoadPoint

Lädt einen Punkt in die *ecc_gfp.dll*

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1LoadPoint
          (JNIEnv *env, jobject obj, jbyteArray loadX0, jbyteArray loadY0,
          jbyteArray loadZ0)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt
- `loadX0`: X-Koordinate des Punktes
- `loadY0`: Y-Koordinate des Punktes
- `loadZ0`: Y-Koordinate des Punktes

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1Mult

Führt eine Multiplikation mit dem in der *ecc_gfp.dll* DLL gespeicherten Punkt und dem Skalar K durch. Das Ergebnis wird affin zurück gegeben.

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1Mult
          (JNIEnv *env, jobject obj, jbyteArray returnX0, jbyteArray returnY0,
          jbyteArray returnZ0, jbyteArray loadK)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt
- `#returnX0`: X-Koordinate des Ergbnis-Punktes
- `#returnY0`: Y-Koordinate des Ergbnis-Punktes
- `#returnZ0`: Z-Koordinate des Ergbnis-Punktes

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1MultPoint

Führt eine Multiplikation mit dem übergebenen Punkt und dem Skalar durch und gibt das Ergebnis in n Koordinaten zurück.

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1MultPoint
          (JNIEnv *env, jobject obj, jbyteArray returnX0, jbyteArray returnY0,
          jbyteArray returnZ0, jbyteArray loadX0, jbyteArray loadY0,
          jbyteArray loadZ0, jbyteArray loadK)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt
- `loadX0`: X-Koordinate des Punktes
- `loadY0`: Y-Koordinate des Punktes
- `loadZ0`: Z-Koordinate des Punktes
- `#returnX0`: X-Koordinate des Ergbnis-Punktes
- `#returnY0`: Y-Koordinate des Ergbnis-Punktes
- `#returnZ0`: Z-Koordinate des Ergbnis-Punktes

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1Negate

Negiert den übergebenen Punkt und gibt das Ergebnis wiederrum affin zurück.

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1MultPoint
          (JNIEnv *env, jobject obj, jbyteArray returnX0, jbyteArray returnY0,
          jbyteArray returnZ0, jbyteArray loadX0, jbyteArray loadY0,
          jbyteArray loadZ0)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt
- `loadX0`: X-Koordinate des Punktes
- `loadY0`: Y-Koordinate des Punktes
- `loadZ0`: Z-Koordinate des Punktes
- `#returnX0`: X-Koordinate des Ergbnis-Punktes
- `#returnY0`: Y-Koordinate des Ergbnis-Punktes
- `#returnZ0`: Z-Koordinate des Ergbnis-Punktes

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1Add

Addiert die beiden übergebenen n Punkte und gibt die Koordinaten des Ergebnispunktes zurück.

Signatur

```
JNIEXPORT jint JNICALL
           Java_de_flexiprovider_ECPInterface_EC_1Add
          (JNIEnv *env, jobject obj, jbyteArray returnX0,
          jbyteArray returnY0, jbyteArray returnZ0, jbyteArray loadX0,
          jbyteArray loadY0, jbyteArray loadZ0, jbyteArray loadX1,
          jbyteArray loadY1, jbyteArray loadZ1)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt
- `loadX0`: X-Koordinate des ersten Punktes
- `loadY0`: Y-Koordinate des ersten Punktes
- `loadZ0`: Y-Koordinate des ersten Punktes
- `loadX1`: X-Koordinate des zweiten Punktes
- `loadY1`: Y-Koordinate des zweiten Punktes
- `loadZ1`: Y-Koordinate des zweiten Punktes
- `#returnX0`: X-Koordinate des Ergbnis-Punktes
- `#returnY0`: Y-Koordinate des Ergbnis-Punktes
- `#returnZ0`: Y-Koordinate des Ergbnis-Punktes

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*

Java_de_flexiprovider_ECPInterface_EC_1IsOnCurve

Überprüft ob der übergebene Punkt auf der in der *ecc_gfp.dll* gespeicherten Kurve liegt oder nicht.

Signatur

```
JNIEXPORT jint JNICALL
          Java_de_flexiprovider_ECPInterface_EC_1IsOnCurve
          (JNIEnv *env, jobject obj, jbyteArray loadX0,
          jbyteArray loadY0, jbyteArray loadZ0)
```

Parameter

- `*env`: Pointer auf die JAVA Umgebung
- `obj`: JAVA Referenz auf das Objekt
- `loadX0`: X-Koordinate des Punktes
- `loadY0`: Y-Koordinate des Punktes
- `loadZ0`: Z-Koordinate des Punktes

Rückgabewert

- Der Rückgabewert dieser Funktion ist der Rückgabewert der korrespondierenden Funktion der *ecc_gfp.dll*