Zürcher
Hochschule
Winterthur

Hochschule
für Architektur, Technik,
Wirtschaft und Sprache

Efficient Elliptic Curve Cryptoalgorithm for Embedded Systems

Diplomarbeit: DA2 Sna 02/3

Projekt-Gruppe: Vitalii Guidoulianov
Dominik Wild

Dozent: Prof. Dr. A. Steffen

Ausgabedatum: 05.09.2002

Abgabedatum: 28.10.2002

1 Inhaltsverzeichnis

1 Inhaltsverzeichnis....2
2 Einführung....5
2.1 Zusammenfassung....5
2.2 Abstract....6
3 Aufgabenstellung....7
4 Einführung Kryptographie....8
4.1 Public-Key-Konzept....8
4.2 Diffie-Hellman....8
4.2.1 Sicherheit....9
4.3 ElGamal....9
4.4 RSA....10
4.4.1 Verschlüsselung....10
4.4.2 Authentifizierung....10
4.4.3 Sicherheit....11
5 Elliptische Kurven....12
5.1 Elliptische Kurven in realen Zahlen....12
5.1.1 Punktaddition....13
5.1.2 Neutrales Element....14
5.1.3 Punktverdopplung....15
5.1.4 Skalarmultiplikation....16
5.2 Elliptische Kurven in Fp....16
5.2.1 Beispiel einer elliptischen Kurve in Fp....17
5.2.2 Arithmetik elliptischer Kurven in Fp....18
5.3 Elliptische Kurven in F2m....18
5.3.1 Polynom-Repräsentation....19
5.3.2 Beispiel einer elliptischen Kurve in F2m....21
5.3.3 Arithmetik elliptischer Kurven in F2m....22
5.4 Sicherheit....23
6 EC-Anwendungen in der Kryptographie....26
6.1 Schlüsselaustausch....26
6.2 Public-Key-Verfahren....27
6.3 Unterschriften....28
7 Grundlagen von Finite Fields....29
7.1 Einführung....29
7.2 Feld-Definition....32
7.3 Feldbeispiel mit Begriff-Erklärung....33
7.3.1 Das neutrale Element....34
7.3.2 Das inverse Element....34
7.3.3 Das Null-Element....36
7.3.4 Nullteiler....37
7.3.5 Kommutativgesetz: 37
7.3.6 Assoziativgesetz:....37
7.3.7 Distributivgesetz:....37
7.4 Funktionen in den Finite Fields....38
7.5 Einsatzzweck der Finite Fields....39
8 Secure Sockets Layer - Grundlagen 40
8.1 SSL-Platz im OSI-Layer-Model....40

8.2 Aufbau einer SSL-Cipher-Suite....41
8.3 Interner Aufbau des SSL-Layers....42
8.4 SSL-Handshake....45
8.5 Erklärung der Meldungen....47
8.5.1 Client Hello....47
8.5.2 Server Hello....47
8.5.3 Certificate Message (vom Server)....47
8.5.4 Server Key Exchange Message....47
8.5.5 Certificate Request....47
8.5.6 Certificate Message (vom Client)....48
8.5.7 Client Key Exchange....48
8.5.8 Certificate Verify....48
8.5.9 Change Cipher Spec....48
8.5.10 Finished....48
8.6 SSL-Handshake (resumed Session)....49
9 Problemanalyse und Lösungskonzept....50
9.1 Ausgangslage (Problem)....50
9.2 Lösungsansatz....50
9.3 Vorgehen....51
9.4 Ergebnisse....53
9.5 Infrastruktur....54
9.5.1 Hardware....54
9.5.2 Software....54
9.6 Projektplan....55
10 Betrachtung EC in Embedded Systems....56
10.1 Primfelder Fp....57
10.2 Binäre Felder F2m....57
10.3 Weitere Felder....58
10.4 Zusammenfassung....58
11 Effiziente Implementation....59
11.1 Algorithmen....59
11.1.1 Feldoperationen in Fp....59
11.1.2 Punktaddition/-verdopplung....59
11.1.3 Skalarmultiplikation....63
11.2 C versus Assembler....69
11.3 Schätzung der Berechnungszeit in Assembler....71
12 Softwarebeschreibung....74
12.1 Modul-Zusammenhänge....74
12.2 Zustandsmaschine für das SSL-Handshake....78
12.3 EC - Daten-Strukturen....79
12.4 EC- Funktionen....82
12.5 Algorithmus-Auswahl über Funktionen-Pointer....83
12.6 Bignum-Strukturen....84
12.7 Bignum-Funktionen....84
12.8 Software-Tests....87
13 Software-Bedienung....88
13.1 EC-Zertifikat-Generierung....88
13.2 Zertifikat-Formate....90
13.3 Aufbau einer SSL-Session mit dem OpenSSL-Client....91
14 Schlusswort....92

15 Anhang.....93
15.1 Referenzen.....93
15.2 Source-Code.....96
15.2.1 EC-Grundfunktionen.....96
15.2.2 Testfunktion.....102
15.3 Software-Testresultate.....105
15.3.1 SSL_CLIENT: Anonymous ECDH mit RC4, SHA.....105
15.3.2 mSSL_SERVER: Anonymous ECDH mit RC4, SHA.....106
15.3.3 SSL_CLIENT: ECDH,RSA mit RC4, SHA.....107
15.3.4 mSSL_SERVER: ECDH,RSA mit RC4, SHA.....110
15.3.5 SSL_CLIENT: RSA,RSA mit RC4, MD5.....111
15.3.6 mSSL_SERVER: RSA,RSA mit RC4, MD5.....114

2 Einführung

2.1 Zusammenfassung

In der Diplomarbeit „Mini-Webserver mit SSL“ (Sna 00/3) wurde die OpenSSL-Bibliothek auf den IPC@CHIP, ein 80186-Microcomputer (20 MHz, 256kByte Flash-ROM, 512kByte RAM) mit integrierter Ethernet-Schnittstelle und frei ansteuerbaren 24-Volt Ausgängen, portiert. Dadurch stand auf dem Embedded System eine SSL-Implementation zur Verfügung, um Daten mit dem Public-Key-Verfahren RSA verschlüsselt über unsichere Kanäle zu transportieren. Für diesen Zweck ist der RSA-Algorithmus seit längerem der gewohnte Standard und wird deshalb auch von zahlreichen Web-Browsern unterstützt. Die Dechiffrierung dauerte mit einem 1024-Bit Schlüssel auf dem Embedded System 48 Sekunden, was im Notfall gerade noch akzeptabel ist. Die steigenden Sicherheitsanforderungen verlangen aber, dass schon bald stärkere Schlüssel eingesetzt werden.

In den letzten Jahren wurde aufgrund von elliptischen Kurven ein alternatives Public-Key-Verfahren geschaffen, das als ECC bekannt ist (Elliptic Curve Cryptography). Besonderes Merkmal dieser Klasse von Verfahren ist, dass die notwendigen Berechnungen in diesem Fall nicht direkt mit Zahlen, sondern mit anderen Objekten, den Punkten auf den elliptischen Kurven durchgeführt werden. Dadurch, dass in der damit zugrunde liegenden mathematischen Struktur einem potenziellen Angreifer bestimmte Angriffsmethoden nicht zur Verfügung stehen, lassen sich die verwendeten Schlüssel- und Parameterlängen ohne Sicherheitseinbusse deutlich reduzieren. Dies macht die Algorithmen insbesondere interessant für den Einsatz innerhalb von Umgebungen mit beschränkten Rechen- oder Speicherkapazitäten.

In dieser Arbeit wurde primär eine ECC-Implementation realisiert und viel Zeit für Grundlagenarbeit investiert. Denn um aus einer Fülle von Verfahren und Algorithmen jeweils die richtigen auszuwählen, ist ein mathematisches Verständnis unumgänglich. Zudem veröffentlichte Sun Microsystems am 18. September 2002 eine Pressemitteilung, dass das Unternehmen eine ECC-Implementation an die OpenSSL-Gemeinde vermacht habe. Diese ist wohl eine der umfangreichsten und schnellsten Implementationen eines EC-Kryptosystems, weshalb wir uns entschlossen haben, diese auf das Embedded System zu portieren.

Als Resultat kann eine schlanke und effiziente sowie eine umfangreiche und interoperable ECC-Implementation vorgewiesen werden. Dank den effizienten ECC-Implementationen und einigen Assembler-Optimierungen für das Zielsystem werden nun für den Schlüsselaustausch Zeiten von unter 7 Sekunden erreicht. Dies ohne Einbussen bei der Sicherheit in Kauf nehmen zu müssen.

Jetzt liegt es nur noch an den Browser-Entwicklern, um der Kryptographie mit elliptischen Kurven zum Durchbruch zu verhelfen.

Winterthur, 28. Oktober 2002

Vitalii Guidoulianov Dominik Wild

2.2 Abstract

As part of the diploma thesis „Micro-Webserver with SSL“ (Sna 00/3) the OpenSSL library was ported to the IPC@CHIP, a 80186-microcomputer (20 MHz, 256kByte Flash-ROM, 512kByte RAM) featuring an integrated Ethernet interface and 24-volt IO ports. In that way an SSL implementation for the embedded system was available to transport data, encrypted with the RSA public key algorithm over unsecure channels. For this purpose the RSA algorithm has been established as a well accepted standard. It is featuered by a wide variety of web browsers. On the embedded system the deciphering of a 1024-Bit RSA key took about 48 seconds, what in normal cases is barely sufficent. Moreover the growing request for security demands the use of stronger keys as soon as possible.

In the bygone years an alternative public key algorithm on the base of elliptic curves has been developed, which is known as ECC (Elliptic Curve Cryptography). A special characteristic of these algorithms is the fact, that in this case the necessary calculations are not directly performed with numbers, but with other objects, called points on the elliptic curves. Because of this structure some special vulnerabilities are not existing. Hence the used key- and parameterlengths can be reduced without loss of security. These algorithms are especially attractive for low performance systems equiped with meager memory.

The primary target of this thesis was an efficient ECC implementation, which was the most time consuming task. Also there was spent a lot of time for the mathematical basics. Without this basics it is quite impossible to chose the right one out of an abundance of algorithms and techniques. On September 18, 2002 Sun Microsystems announced that they donated the source code of a ECC implementation to the OpenSSL community. This implementation is one of the most extensive and efficient available. Hence we decided to port this code to the embedded system to provide a solution which is compatible with a lot of EC systems available.

The result is a tiny and efficient as well as an extensive and interoperable ECC implementation. Because of the efficient ECC implementations and some assembler language optimizations specially adapted for the target platform, the key exchange lasts now about 7 seconds without any loss of security.

Now it is up to the browser developers to help the elliptic curve cryptography to make its breakthrough.

Winterthur, October 28, 2002

Vitalii Guidoulianov Dominik Wild

3 Aufgabenstellung

Praktische Diplomarbeiten 2002 - Sna02/3

Efficient Ellliptic Curve Cryptoalgorithm for Embedded Systems

Studierende:
- Vitalii Guidoulianov, IT3a
- Dominik Wild, IT3a

Termine:
- Ausgabe: Freitag, 6.09.2002 8:00 im E509
- Abgabe: Montag, 28.10.2002 12:00

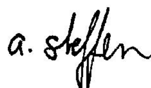
Beschreibung:
Aus Kosten- und Platzgründen sind die in Embedded Systems eingesetzten Prozessoren nur mit begrenzten Rechner- und Speicherressourcen ausgestattet. Typisch sind Taktfrequenzen von 20 MHz und ein RAM-Speicher von 512 kBytes. Die im Rahmen des SSL-Protokolls benötigte RSA Dechiffrierung des Premaster-Secrets durch den SSL-Server mit Hilfe seines 1024 Bit langen Private Keys kann deshalb bis zu einer Minute betragen.

In den letzten Jahren wurde mit den neuartigen Elliptic Curve Cryptoalgorithmen (ECC) ein alternatives Public Key Verfahren geschaffen, dass mit wesentlich kürzeren Schlüsseln bis zu zehnmal schneller ist. In dieser Diplomarbeit soll dieser Algorithmus in der Programmiersprache C auf einem 80186 Prozessor implementiert und ausgetestet werden.

Ziele:
- Erarbeitung einer Übersicht über effiziente ECC Implementationen, die sich für den Einsatz in Embedded System Plattformen eignen.
- Wahl von geeigneten elliptischen Kurven.
- Exemplarische Implementation eines ECC Verfahrens auf der 16 bit-IPC@CHIP Plattform unter Zuhilfenahme von Crypto-Funktionen der OpenSSL Library.

Infrastruktur / Tools:
- Raum: **E523**
- Rechner: 2 PCs unter Windows 2000 / SuSE Linux
- Hardware: IPC@CHIP 80186 Embedded Platform
- SW-Tools: Borland 16bit C/C++ Compiler, OpenSSL Crypto Library

Winterthur, 5. September 2002



Prof. Dr. Andreas Steffen

4 Einführung Kryptographie

Herkömmliche Kryptographie basiert darauf, dass Sender und Empfänger einer Nachricht den gleichen geheimen Schlüssel kennen und benutzen: Der Sender benutzt den geheimen Schlüssel zum Chiffrieren und der Empfänger benutzt ihn zum Entschlüsseln. Die Methode wird symmetrische Verschlüsselung genannt. Das Hauptproblem hierbei ist, dass sich Sender und Empfänger auf den gleichen geheimen Schlüssel einigen, ohne dass ihn jemand anders zu Gesicht bekommt. Wenn sie sich an verschiedenen geographischen Orten befinden, müssen sie einem Kurier, einer Telefonverbindung oder einem anderen Kommunikationsmedium trauen, um die Offenlegung des geheimen Schlüssels zu verhindern. Jeder, der den Schlüssel während der Übertragung abfangen oder mithören kann, kann danach alle verschlüsselten Nachrichten lesen, ändern, fälschen oder unterschreiben, indem er diesen Schlüssel verwendet. Die Schlüsselerzeugung, -übertragung und -speicherung wird Schlüsselmanagement genannt, alle Kryptosysteme müssen sich damit befassen. Da alle Schlüssel eines symmetrischen Kryptosystems geheim bleiben müssen, haben diese Systeme oft Schwierigkeiten mit einem sicheren Schlüsselmanagement, speziell in offenen Umgebungen mit vielen Nutzern.

4.1 Public-Key-Konzept

Das Konzept der asymmetrischen Kryptographie wurde 1976 von Whitfield Diffie und Martin Hellman vorgeschlagen, um das Schlüsselmanagementproblem zu lösen. In ihrem Konzept hat jeder Beteiligte zwei Schlüssel, einen Öffentlichen und einen Privaten. Jeder öffentliche Schlüssel wird veröffentlicht und der private Schlüssel bleibt geheim. Die Notwendigkeit eines gemeinsamen Geheimnisses zwischen Sender und Empfänger ist damit verschwunden: Jede Kommunikation umfasst nur öffentliche Schlüssel, private Schlüssel werden nie übertragen oder geteilt. Es ist nicht länger notwendig, einem Kommunikationskanal hinsichtlich Abhörsicherheit zu trauen. Die einzige Anforderung ist, dass der Zuordnung zwischen öffentlichem Schlüssel und Nutzer getraut werden kann (z.B. durch ein vertrauenswürdiges Verzeichnis). Jeder, der eine wichtige Information versenden will, chiffriert mit dem öffentlichen Schlüssel, das Chiffrat kann jedoch nur mit dem privaten Schlüssel dechiffriert werden, der sich ausschliesslich in der Hand des Empfängers befindet. Darüberhinaus kann asymmetrische Kryptographie nicht nur zur Geheimhaltung (Verschlüsselung) sondern auch zur Authentifikation (digitale Unterschriften) verwendet werden.

4.2 Diffie-Hellman

Das Diffie-Hellman-Schlüsselaustausch-Protokoll wurde 1976 durch Diffie und Hellman entwickelt. Das Protokoll erlaubt es, einen geheimen Schlüssel über einen unsicheren Kanal abzumachen.

Das Protokoll hat zwei Parameter p und g, wobei beide öffentlich sein dürfen. Der Parameter p ist eine Primzahl und g ist eine Zahl, die kleiner als p ist und mit der alle anderen Zahlen von 1 bis p-1 berechnet werden können, wenn g mit eine gewisse Anzahl mal mit sich selber multipliziert und das Resultat modulo p reduziert wird.

Wollen nun zwei Parteien A und B einen geheimen Schlüssel abmachen, ermitteln sie jeweils eine

Zufallszahl *a* resp. *b* und potenzieren damit die gemeinsame Basis *g*. Die erhaltenen Resultate werden dann ausgetauscht.

$$A = g^a \, mod \, p$$
$$B = g^b \, mod \, p$$

Schlussendlich berechnen die beiden Parteien mit den ausgetauschten Werten und dem eigenen Zufallswert *a* resp. *b* den geheimen Schlüssel *s*:

$$s = B^a \, mod \, p = (g^b)^a \, mod \, p$$
$$s = A^b \, mod \, p = (g^a)^b \, mod \, p$$

Angenommen *p*=31 und *g*=3:

A: $a = 8 \rightarrow A = 3^8 \, mod \, 31 = 20$
B: $b = 6 \rightarrow B = 3^6 \, mod \, 31 = 16$

A: $s = 16^8 \, mod \, 31 = 4$
B: $b = 20^6 \, mod \, 31 = 4$

4.2.1 Sicherheit

Das Diffie-Hellman-Protokoll ist anfällig auf „middleperson"-Attacken, da die Authentizität der Kommunikationspartner nicht überprüft wird. Wenn sich jemand während dem Schlüsselaustausch zwischen die zwei Parteien schaltet, ist die Sicherheit unterwandert.

Das System basiert auf dem diskreten Logarithmus-Problem (DLP):

$$y = g^x (mod \, p)$$

Das DLP modulo *p* ist es, zu einem gegebenen Paar *g* und *y* den Exponenten *x* zu finden.

4.3 ElGamal

Das erste Public-Key-Kryptosystem, das auf dem DLP basierte, war das ElGamal-Schema, das nach seinem Entwickler Taher ElGamal benannt ist. Das ElGamal-Schema enthält sowohl Verschlüsselungs- als auch Unterschriftsalgorithmen.

Das System wird bestimmt durch zwei Paramter, eine Primzahl *p* und eine Ganzzahl *g*, deren Potenzen modulo *p* eine grosse Anzahl von Elementen umfasst.

Der private Schlüssel von Alice sei *a* und ihr öffentlicher Schlüssel sei $y = g^a \; (mod \; p)$. Will nun Bob Alice eine Nachricht m schicken, so wählt er eine Zufallszahl *k* kleiner als *p* und berechnet dann:

$$y_1 = g^k (mod \, p)$$
$$y_2 = m \, XOR (y^k \, mod \, p)$$

Übermittelt wird nun (y_1,y_2). Nachdem Alice dies empfangen hat, berechnet sie:

$$m = (y_1^a \, mod \, p) \, XOR \, y_2 .$$

Ein Nachteil von ElGamal ist, dass das Chiffrat doppelt so gross ist, wie die Nachricht selbst. Dies ist jedoch vernachlässigbar, wenn ElGamal nur zum Schlüsselaustausch verwendet wird und die Daten danach symmetrisch chiffriert werden (Hybridsystem).

Auf dem Unterschriftenalgorithmus von ElGamal basiert unter anderem auch DSA/DSS (Digital Signature Algorithm/Digital Signature Standard).

4.4 RSA

RSA war das erste asymmetrisches Kryptosystem, dass sich zum Verschlüsseln und zur Authentifikation eignete. Es wurde 1977 von Ron Rivest, Adi Shamir und Leonard Adleman erfunden. Ein Schlüsselpaar wird folgendermassen generiert:

1. *Wählen von zwei grossen Primzahlen p und q*
2. *Berechnen von $n = pq$*
3. *Wählen einer Zahl $e < (p-1)(q-1)$*
   (e muss teilerfremd zu $(p-1)(q-1)$ sein)
4. *Berechnen von $d = e - 1 \, mod \, (p-1)(q-1)$*
5. *$Public-Key(n, e), Private-Key(n, d)$*
6. *Vernichten von p und q*

4.4.1 Verschlüsselung

Alice möchte eine Nachricht *m* an Bob schicken. Alice erstellt den Chiffretext *c* durch modulare Potenzierung mit Bob's Public-Key (*n,e*):

$$c = m^e \, mod \, n$$

Um dies zu dechiffrieren potenziert Bob ebenfalls, diesmal mit seinem Private-Key (*d,n*):

$$c^d \, mod \, n = m^{ed} \, mod \, n = m^{(p-1)(q-1)-1} \, mod \, n = m$$

Dies nützt die Eulersche Verallgemeinerung des kleinen Satzes von Fermat (ohne Beweis).

4.4.2 Authentifizierung

Alice möchte eine Nachricht *m* an Bob unterschreiben, so dass Bob sicher sein kann, dass diese Nachricht unverändert und von Alice ist. Alice erstellt eine elektronische Unterschrift durch Potenzieren mit dem Private-Key von Alice (*n,d*):

$$s = m^d \, mod \, n$$

Sie sendet *m* und *s* an Bob. Dieser prüft die Unterschrift durch eine Potenzierung mit dem Public-Key von Alice (*n,e*):

$s^e \bmod n = m$, wenn die Authentifikation erfolgreich war.

4.4.3 Sicherheit

RSA ist das bekannteste System einer Familie, deren Sicherheit auf dem IFP (Integer Factorization Problem) basiert. Das IFP ist folgendermassen defniert. Erinnern wir uns, dass eine Primzahl *p* nur eine Primzahl ist, wenn sie nur durch 1 und durch sich selber teilbar ist. Gegeben ist eine Zahl *n*, die das Produkt zweier grosser Primzahlen ist:

$$n = pq$$

Zu finden seien nun die Primzahlen *p* und *q*. Ein RSA-Public-Key besteht aus einem Paar (*n,e*), wobei *e* eine Zahl im Bereich [1,*n*-1] ist. Es wird angenommen, bis heute wurde jedenfalls nichts Gegenteiliges bewiesen, dass, um RSA zu brechen, das IFP für *n* gelöst werden muss. Da kein Algorithmus bekannt ist, der das IFP effizient lösen kann, wird ein RSA-System als sicher betrachtet, wenn *n* gross genug gewählt wird.

Die Analyse auf Basis der besten verfügbaren Faktorisierungsalgorithmen zeigt, dass RSA und ElGamal bei gleicher Schlüssellänge ähnliche Sicherheit bieten.

5 Elliptische Kurven

Wer sich mit elliptischen Kurven beschäftigt, wird schnell feststellen, dass sie keine Ellipsen darstellen. Elliptische Kurven stammen eigentlich aus der Funktionentheorie, genauer gesagt aus der Theorie elliptischer Funktionen. Diese wiederum sind entstanden durch Umkehrung elliptischer Integrale, und diese verdanken ihren Namen dem Versuch, den Umfang von Ellipsen zu berechnen.

1985 schlugen Neil Koblitz von der University of Washington und Victor Miller, der zu dieser Zeit für IBM arbeitet, unabhängig voneinander die Verwendung von elliptischen Kurven in der Kryptographie vor, deren Sicherheit auf einem diskreten Logarithmus-Problem (DLP) basieren. ECC (Elliptic Curve Cryptosystems) können für Verschlüsselung und Unterschriften verwendet werden. In diesem Sinne stellen ECC eine Alternative für bekannte Public-Key-Verfahren dar.

In diesem Kapitel stellen wir die grundlegenden Konzepte der Theorie der elliptischen Kurven vor, jedoch ohne mathematische Beweisführung, stattdessen werden Beispiele angeführt. Mathematisches Grundlagenwissen über Gruppen und Felder wäre für das Verständnis dieses Kapitels von Vorteil, diese Grundlagen werden in Kapitel 7 vermittelt.

5.1 Elliptische Kurven in realen Zahlen

Eine elliptische Kurve in den realen Zahlen kann definiert werden als Menge von Punkten (x,y), welche eine Gleichung der folgenden Form erfüllen.

$$y^2 = x^3 + ax + b$$

Gleichung 5.1: Weierstrass Normalform einer elliptischen Kurve

Dabei bestimmen die Parameter a und b die Form der Kurve. Zum Beispiel ergibt die Wahl $a=-4$ und $b=0.67$ die in gezeigte Kurve. Besitzt $x^3 + ax + b$ keine mehrfachen Nullstellen oder gleichbedeutend, wenn $4a^3 + 27b^2 \neq 0$, dann kann die elliptische Kurve nach Gleichung 5.1 benutzt werden, um eine Gruppe zu bilden. Eine Gruppe über eine elliptische Kurve in den realen Zahlen besteht aus den Punkten auf der entsprechenden elliptischen Kurve und einem speziellen Punkt O, der auch als Punkt in der Unendlichkeit bezeichnet wird.

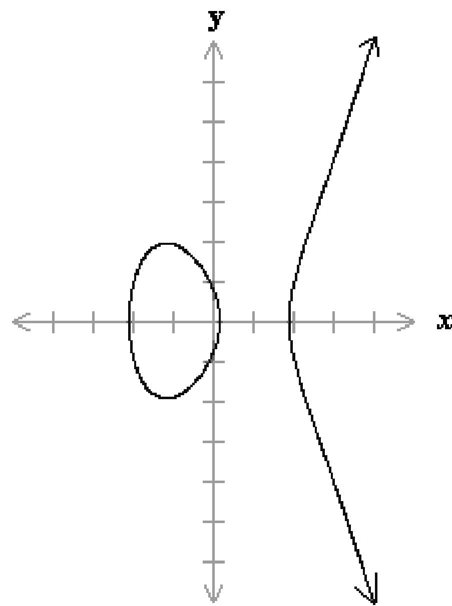


Abbildung 5.1: Elliptische Kurve $y^2=x^3-4x+0.67$ (Quelle: [ECCT])

5.1.1 Punktaddition

Gruppen über elliptische Kurven sind additive Gruppen, was bedeutet, dass ihre Basisfunktion die Addition ist. Die Addition von zwei Punkten auf einer elliptischen Kurve ist geometrisch definiert. Das Negative eines Punktes $P=(x_P,y_P)$ ist definiert als seine Spiegelung an der x-Achse, der Punkt $-P$ ist also $(x_P,-y_P)$. Dabei fällt für jeden Punkt P auf einer elliptischen Kurve sein Pendant $-P$ ebenfalls auf die Kurve.

Nehmen wir an, dass P und Q zwei unterschiedliche Punkte auf einer elliptischen Kurve sind und P nicht $-Q$ ist. Um nun die beiden Punkte zu addieren, wird eine Gerade durch die Punkte gelegt. Diese Gerade wird die elliptische Kurve in genau einem Punkt schneiden, der als $-R$ bezeichnet wird. Der Punkt $-R$ wird gespiegelt an der x-Achse und ergibt R. Somit lautet das Gesetz zur Berechnung einer Punktaddition $P+Q=R$, was als Beispiel in dargestellt ist.

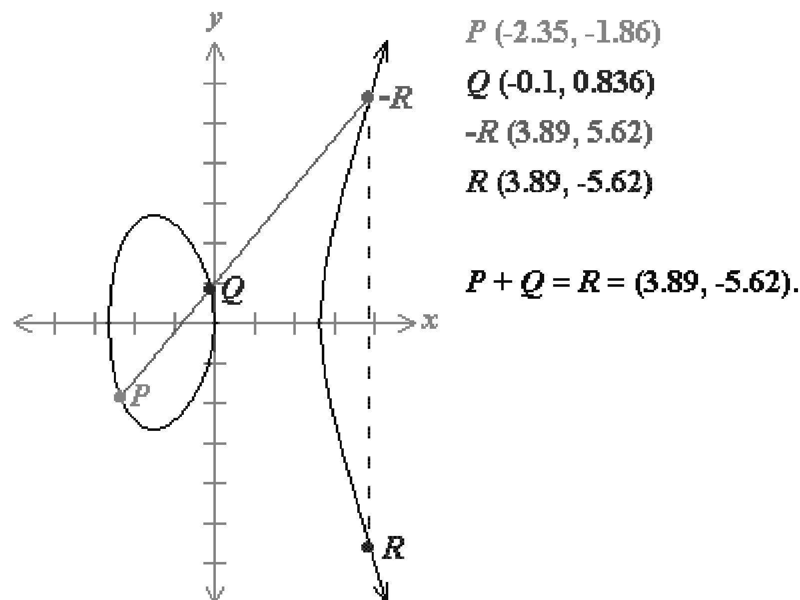


Abbildung 5.2: Punktaddition auf $y^2=x^3-7x$ (Quelle [ECCT])

Wenn die Punkte P und Q nicht ihr gegenseitiges Inverses sind, dann lässt sich *R=P+Q* algebraisch berechnen:

$$s = \frac{y_P - y_Q}{x_P - x_Q}$$

$$x_R = s^2 - x_P - x_Q$$

$$y_R = -y_P + s(x_P - x_R)$$

Gleichung 5.2: Algebraische Punktaddition

Dabei ist *s* die Steigung der Geraden durch die Punkte *P* und *Q*.

5.1.2 Neutrales Element

Eine Ausnahme der Additionsdefinition, die besagt, dass eine Gerade durch zwei Punkte die Kurve in einem Punkt schneidet, ist die Addition eines Punktes *P* mit seinem Inversen *-P*. Denn in diesem Fall wird die Gerade die elliptische Kurve in keinem Punkt schneiden, weshalb die Punkte *P* und *-P* nicht nach der vorher definierten Regel addiert werden können. Aus diesem Grund beinhaltet die Gruppe über die elliptische Kurve den Punkt *O* in der Unendlichkeit. Per Definition gilt *P+(-P)=O*. Als Resultat dieser Gleichung und der mathematischen Definition des neutralen Elements in einer Gruppe gilt deshalb ebenfalls *P+O=P*.

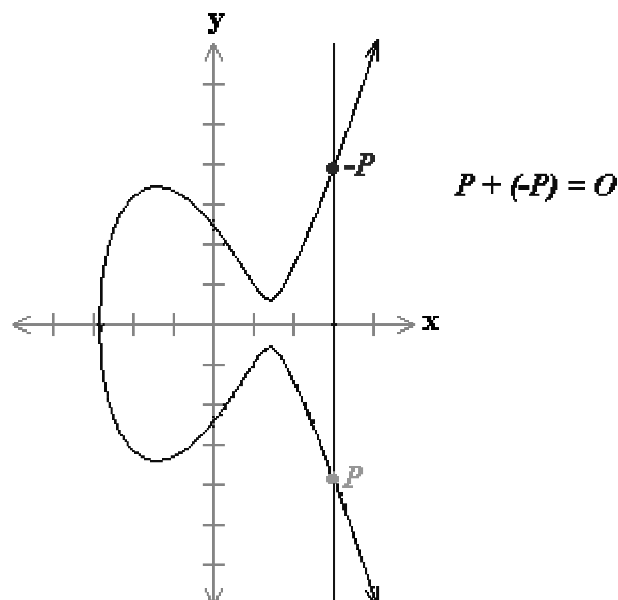


Abbildung 5.3: Addition P+(-P)=O auf $y^2=x^3-6x+6$ (Quelle [ECCT])

5.1.3 Punktverdopplung

Um einen Punkt *P* mit sich selber zu addieren, wird eine Tangente im Punkt *P* an die elliptische Kurve gelegt. Wenn y_P nicht 0 ist, schneidet die Tangente die elliptische Kurve in genau einem Punkt. Wie bei der Addition wird dieser Punkt *-R* durch Spiegelung an der x-Achse zu *R*. Diese Operation wird Punktverdopplung genannt, das Gesetz dazu ist definiert als *P+P=2P=R*.

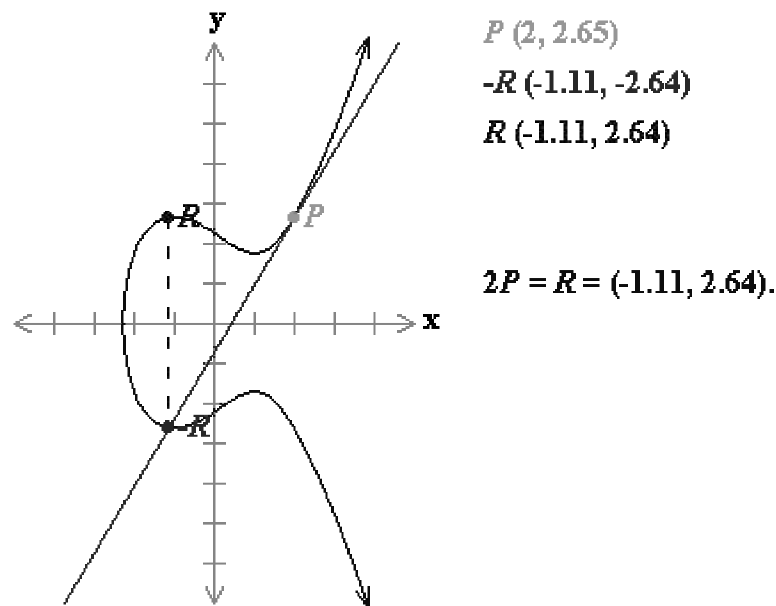


Abbildung 5.4: Punktverdopplung auf $y^2=x^3-3x+5$ (Quelle [ECCT])

Liegt ein Punkt auf der x-Achse ($y_P=0$), dann ist die Tangente vertikal und schneidet die Kurve in keinem Punkt. Per Definition gilt deshalb für einen solchen Punkt $2P=O$. Wenn mit solch einem Punkt der Punkt 3P berechnet werden soll, kann dies durch $2P+P$ ausgedrückt werden. Dies wird zu $O+P=P$, weshalb $3P=P$.

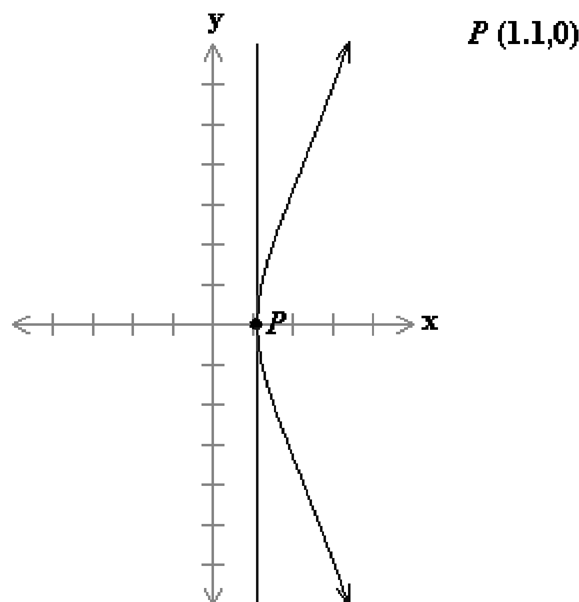


Abbildung 5.5: Punktverdopplung wenn $y_P=0$ auf y2=x3+5x-7 (Quelle [ECCT])

Wenn y_P nicht 0 ist, dann kann eine Punktverdopplung $2P=R$ algebraisch berechnet werden:

$$s = \frac{dy}{dx} = \frac{3x_P^2 + a}{2y_P}$$

$$x_R = s^2 - 2x_P$$

$$y_R = -y_P + s(x_P - x_R)$$

Gleichung 5.3: Algebraische Punktverdopplung

Dabei ist s die Steigung der Tangente an die elliptische Kurve im Punkt P und a ist ein Parameter der Weierstrass-Gleichung (siehe Gleichung 5.1).

5.1.4 Skalarmultiplikation

Mit Skalarmultiplikation oder auch einfach Multiplikation eines Punktes P auf einer elliptischen Kurve mit einer Zahl e, wird die Berechnung von eP bezeichnet. Dazu wird P einfach e mal zu sich selber addiert:

$$\underbrace{P + P + ... + P}_{e} = eP = Q$$

Natürlich existieren Algorithmen, die dies effizienter erledigen (siehe Kapitel 11). Neben der in der Literatur häufig verwendeten additiven Schreibweise, existiert auch eine multiplikative Notation, d.h. dass Punktaddition als Punktmultiplikation und Punktverdopplung als Punktquadrierung bezeichnet wird. Als Konsequenz daraus wird dann die Skalarmultiplikation als Punktexponention betitelt. Wir benutzen in dieser Arbeit weiterhin die additive Notation.

5.2 Elliptische Kurven in F_p

Berechnungen in den realen Zahlen sind langsam und wegen Rundungsfehlern ungenau. Kryptographische Anwendungen benötigen schnelle und genaue Arithmetik, weshalb in der Praxis Gruppen über elliptische Kurven in endlichen Feldern F_p (prime finite field) und F_{2^m} (binary finite field) verwendet werden. In diesem Abschnitt werden die Felder F_p mit zugrundeliegender Primzahl betrachtet.

Das Feld F_p umfasst die Zahlen 0 bis p-1, jede Berechnung in diesem Feld endet mit der Restbildung der Division durch p (modulo p). Zum Beispiel enthält das Feld F_{23} die Zahlen von 0 bis 22 und jede Operation endet wieder in einer dieser Zahlen.

Eine elliptische Kurve mit dem darunterliegenden Feld F_p kann gebildet werden, indem die Paramter a und b aus dem Feld F_p gewählt werden. Die elliptische Kurve umfasst alle Punkte, welche die Gleichung der elliptischen Kurve modulo p erfüllen (x und y sind Zahlen aus F_p). Zum Beispiel hat $y^2 \bmod p = x^3 + ax + b \bmod p$ ein darunterliegendes Feld F_p, wenn a und b in F_p sind.

Besitzt $x^3 + ax + b$ keine mehrfachen Nullstellen oder gleichbedeutend, wenn $4a^3 + 27b^2 \bmod p \neq 0$, dann kann die elliptische Kurve benutzt werden, um eine Gruppe zu bilden. Eine elliptische Kurve im Feld F_p besteht aus den Punkten der entsprechenden Kurve und dem Punkt in der Unendlichkeit. Auf einer solchen Kurve liegen demnach nur endlich viele Punkte, die Anzahl dieser Punkte wird Ordnung genannt.

5.2.1 Beispiel einer elliptischen Kurve in F_p

Betrachten wir ein Beispiel mit F_{23} und den Kurvenparametern $a=1$ und $b=0$. Die entsprechende elliptische Kurve ist demnach $y^2 = x^3 + x$. Der Punkt (9,5) erfüllt die Gleichung, da:

$$
\begin{aligned}
&y^2 \, mod \, p = x^3 + x \, mod \, p \\
&25 \, mod \, 23 = 729 + 9 \, mod \, 23 \\
&25 \, mod \, 23 = 738 \, mod \, 23 \\
&2 = 2
\end{aligned}
$$

Die 23 weiteren Punkte, die die Gleichung erfüllen sind:

(0,0), (1,5), (1,18), (9,5), (9,18), (11,10), (11,13), (13,5), (13,18), (15,3), (15,20), (16,8), (16,15), (17,10), (17,13), (18,10), (18,13), (19,1), (19,22), (20,4), (20,19), (21,6), (21,17).

Diese Punkte sind in graphisch dargestellt.

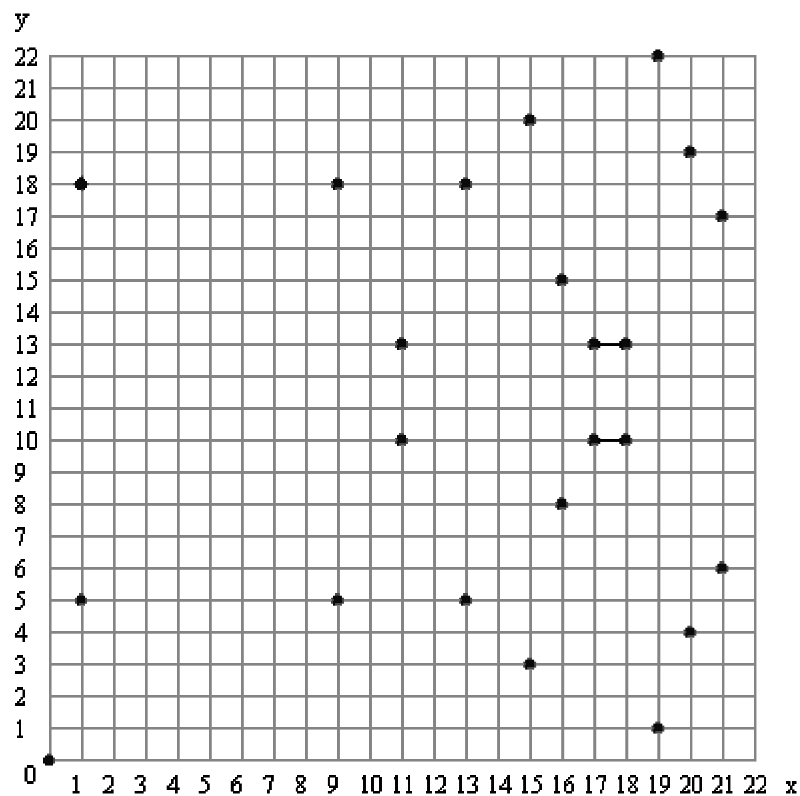


Abbildung 5.6: Punkte der elliptischen Kurve $y^2=x^3+x$ in F_{23} (Quelle: [ECCT])

Wie bei den realen Zahlen existieren auch hier für jeden x-Wert zwei Punkte. Obwohl die Verteilung der Punkte zufällig erscheint, existiert eine Symmetrie bei *y=11,5*. Auch hier existiert also für jeden Punkt ein negativer Punkt, welcher an der x-Achse gespiegelt ist. In einem Feld werden negative Zahlen als Differenz vom Modulus p dargestellt. Also z.B. wird $P(11,10)$ zu $-P(11,13)$, da $-10 \, mod \, p = p - 10$, d.h. $23 - 10 = 13$.

5.2.2 Arithmetik elliptischer Kurven in F_p

Die Regeln der Arithmetik in F_p sind eigentlich genau gleich, wie in den realen Zahlen, mit der Ausnahme, dass Berechungen immer modulo p durchgeführt werden. Zudem besteht eine elliptische Kurve in F_p aus einer endlichen Anzahl Punkte, weshalb nicht klar ist, wie die Punkte verbunden werden sollen, damit der Graph wie eine Kurve aussieht. Deshalb kann die Geometrie, die in den realen Zahlen verwendet wurde, nicht in F_p verwendet werden. Trotzdem gelten aber die algebraischen Zusammenhänge auch in F_p.

Grundsätzlich werden effiziente Feldoperationen benötigt, um in F_p zu rechnen. Dazu wurden spezielle Algorithmen entwickelt. Die Arithmetik nach Montgomery ist bei weitem der erfolgreichste Ansatz, um dies zu implementieren.

5.3 Elliptische Kurven in F_2^m

Wie schon erwähnt, werden als unterliegende Felder für elliptische Kruven in der Praxis momentan fast ausschliesslich Primfelder und binäre Felder eingesetzt. Dieser Abschnitt widmet sich den binären Felder und ihrer nicht offensichtlichen Arithmetik. Die Elemente in einem F_2^m-Feld sind Bitfolgen der Länge m. Weil die Operationen in solch einem Feld nicht intuitiv einleuchten, wird die Arithmetik genauer beleuchet als in F_p. Die Regeln der Arithmetik in einem binären Feld (F_2^m) sind völlig neu definiert worden, sie können durch die Polynom-Repräsentation definiert werden. Eine elliptische Kurve in einem F_2^m-Feld wird durch die Wahl von a und b aus F_2^m gebildet, wobei b nicht 0 sein darf. Die Gleichung der elliptischen Kurve wird, weil sie neu in einem F_2^m-Feld liegen soll, leicht angepasst:

$$y^2 + xy = x^3 + ax^2 + b$$

Gleichung 5.4: Elliptische Kurve in F_2^m

Die elliptische Kurve beinhaltet alle Punkte, die diese Gleichung über F_2^m erfüllen, d.h. die x- und y-Koordinaten Elemente in F_2^m sind.

5.3.1 Polynom-Repräsentation

Die Elemente in einem Feld der Charakteristik F_2^m sind Bit-Strings der Länge m. Die Elemente können auch als Polynome mit einem Grad kleiner m und Koeffizienten aus F_2 (Subfield) betrachtet werden. Die Elemente sind also in der Form:

$$\{a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \ldots + a_2x^2 + a_1x + a_0 \mid a_i \in (0,1)\}$$

Die Hauptoperationen in F_2^m sind Addition und Subtraktion. Einige Berechnungen benötigen ein Polynom $f(x) = x_m + f_{m-1}x^{m-1} + f_{m-2}x^{m-2} + \ldots + f_{2x}^2 + f_{1x} + f_0$, wobei f_i in F_2 und das Polynom irreduzibel sein muss.

Zum Beispiel sind die Elemente in F_2^4:

(0000), (0001), (0010), (0011), (0100), (0101), (0110), (0111), (1000), (1001), (1010), (1011), (1100), (1101), (1110), (1111).

Addition und Subtraktion

$$(a_{m-1} \ldots a_1 a_0) + (b_{m-1} \ldots b_1 b_0) = (c_{m-1} \ldots c_1 c_0)$$

d.h.

$$(a_{m-1} \ldots a_1 a_0) + (a_{m-1} \ldots a_1 a_0) = (0 \ldots 00)$$

Die Regeln der Addition sind analog der Polynomaddition, was bedeutet, dass keine Carry-Propagation existiert. Denn wenn zwei Polynome addiert werden, hat z.B. der Koeffizient von x^2 keinen Einfluss auf den Koeffizienten von x^3. Und da die Koeffizienten so definiert sind, dass sie in F_2 liegen müssen, ist die komponentenweise Addition modulo 2 ($c_i = (a_i+b_i) \bmod 2$) durchzuführen, was auch als exklusives ODER (XOR) betrachtet werden kann. Deshalb ist in F_{2^m} jedes Element auch sein eigenes additives Inverses, was bedeutet, dass in F_{2^m}-Feldern Addition und Subtraktion äquivalente Operationen sind.

Multiplikation

$$(a_{m-1} \ldots a_1 a_0)(b_{m-1} \ldots b_1 b_0) = (r_{m-1} \ldots r_1 r_0)$$

Dabei ist das Polynom $r_{m-1}x^{m-1} + \ldots + r_1x + r_0$ der Rest, der entsteht, wenn das Resultat der Polynommultiplikation durch das irreduzible Reduktionspolynom $f(x)$ geteilt wird. Die Polynommultiplikation kann durch schieben und addieren relativ einfach durchgeführt werden, da durch die Definition der Addition keine Überträge existieren. Als Beispiel eine Multiplikation im Feld F_{2^4}:

```
1101 * 1001
-----------
       1101
      0000
     0000
    1101
    -------
    1100101
```

Der Multiplikand wird dabei jeweils mit einem Bit des Multiplikators multipliziert und das Resultat mit dem vorhergehenden Resultat addiert (XOR). Die Multiplikation algebraisch mit Polynomen ausgedrückt:

$$\begin{aligned}
&(1101)(1001) \\
&= (x^3 + x^2 + 1)(x^3 + 1) \bmod f(x) \\
&= x^6 + x^5 + 2x^3 + x^2 + 1 \bmod f(x) \\
&= x^6 + x^5 + x^2 + 1 \bmod f(x) \quad \text{(Koeffizienten werden modulo 2 reduziert)} \\
&= (1100101).
\end{aligned}$$

Die Modulo-Reduktion erfolgt daraufhin, wie erwähnt, durch Division mit dem Reduktionspolynom, in diesem Beispiel ist $f(x) = x^4 + x + 1$:

$$
\begin{array}{l}
1100101 \div 10011 = 11 \\
\underline{10011} \\
\;\;10100 \\
\;\;\underline{10011} \\
\;\;\;\;1111
\end{array}
$$

Das Gesuchte bei einer Modulo-Operation ist ja der Rest, das eigentliche Resultat der Division interessiert nicht. Die Modulo-Reduktion algebraisch:

$$
\begin{aligned}
&(1100101)\,mod\,(10011) \\
&= x^6 + x^5 + x^2 + 1 \; mod\, f(x) \\
&= (x^4 + x + 1)(x^2 + x) + (x^3 + x^2 + x + 1)\, mod\, f(x) \\
&= x3 + x2 + x + 1 \\
&= (1111).
\end{aligned}
$$

Diese Division lässt sich schlussendlich ebenfalls wieder durch Additionen ausdrücken, geeignete Algorithmen für Multiplikation und Modulo sind zu finden in [Blake99] und [Hank00].

Weiter muss erwähnt werden, dass Quadrieren im Vergleich zur allgemeinen Multiplikation in F_{2^m} viel einfacher ist. Dazu muss lediglich zwischen jede binäre Ziffer des Ausgangselements eine 0 eingeschoben werden.

Multiplikative Inversion

In jedem Feld F_{2^m} existiert mindestens ein Element *g*, durch das alle anderen Nicht-Null-Elemente als Potenz von *g* ausgedrückt werden können. Solch ein Element *g* wird Generator genannt. Im Feld F_{2^4} ist zum Beispiel *g*=(0010) ein Generator:

$$
\begin{array}{llll}
g^0 = (0001) & g^1 = (0010) & g^2 = (0100) & g^3 = (1000) \\
g^4 = (0011) & g^5 = (0110) & g^6 = (1100) & g^7 = (1011) \\
g^8 = (0101) & g^9 = (1010) & g^{10} = (0111) & g^{11} = (1110) \\
g^{12} = (1111) & g^{13} = (1101) & g^{14} = (1001) & g^{15} = (0001)
\end{array}
$$

Wenn $a = g^i$, ist das multiplikative Inverse nun definiert als $a^{-1} = g^{(-i)\,mod(2^m-1)}$. Das Inverse von $g^7 = (1011)$ im Feld F_{2^4} ist $g^{-7\,mod\,15} = g^8 = (0101)$. Wenn das stimmen würde, müsste $g^7 g^8 = 1$ sein:

$$
\begin{aligned}
&(1011)(0101) \\
&= (x3 + x + 1)(x2 + 1)\, mod\, f(x) \\
&= x5 + x2 + x + 1 \; mod\, f(x) \\
&= (x4 + x + 1)(x) + (1)\, mod\, f(x) \\
&= 1 \\
&= (0001)
\end{aligned}
$$

5.3.2 Beispiel einer elliptischen Kurve in F_2^m

Es wird weiterhin das Beispiel-Feld F_2^4 mit dem irreduziblen Polynom $f(x) = x^4 + x + 1$ und dem Generator $g = (0010)$ betrachtet. Natürlich muss in einer kryptographischen Anwendung m so gross gewählt werden, damit das Ermitteln aller Potenzen von g verunmöglicht wird, andernfalls könnte das Kryptosystem gebrochen werden. In der Praxis ist momentan m=160 ein akzeptabler Wert. Über F_2^4 legen wir nun die Kurve $y^2 + xy = x^3 + g^4x^2 + 1$. Hier sind also $a = g^4$ *and* $b = g^0 = 1$. Der Punkt (g^5, g^3) erfüllt diese Gleichung über F_2^m:

$$y^2 + xy = x^3 + g^4 x^2 + 1$$
$$(g^3)^2 + g^5 g^3 = (g^5)^3 + g^4 (g^5)^2 + 1$$
$$g^6 + g^8 = g^{15} + g^{14} + 1$$
$$(1100) + (0101) = (0001) + (1001) + (0001)$$
$$(1001) = (1001)$$

Die 15 Punkte, die die Gleichung erfüllen sind also:

$(1, g^{13})$, (g^3, g^{13}), (g^5, g^{11}), (g^6, g^{14}), (g^9, g^{13}), (g^{10}, g^8), (g^{12}, g^{12}), $(1, g^6)$, (g^3, g^8), (g^5, g^3), (g^6, g^8), (g^9, g^{10}), (g^{10}, g), $(g^{12}, 0)$, $(0, 1)$.

Diese Punkte sind in graphisch dargestellt.

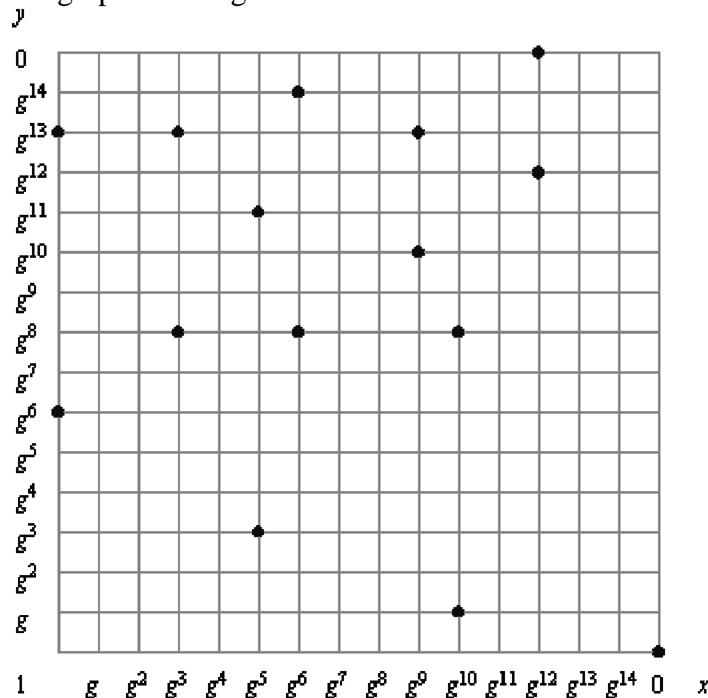


Abbildung 5.7: Punkte der elliptischen Kurve $y^2 + xy = x^3 + g^4x^2 + 1$ in F_2^4 (Quelle: [ECCT])

5.3.3 Arithmetik elliptischer Kurven in F_2^m

Punktaddition

Das Negative des Punktes $P = (x_P, y_P)$ ist der Punkt $-P = (x_P, x_P + y_P)$. Wenn P und Q zwei

verschiedene Punkte sind, die nicht ihr gegenseitiges additives Inverses sind, dann lässt sich $P + Q = R$ berechnen durch:

$$s = \frac{y_P - y_Q}{x_P + x_Q}$$
$$x_R = s^2 + s + x_P + x_Q + a$$
$$y_R = s(x_P + x_R) + x_R + y_P$$

Gleichung 5.5: Punktaddition in F_2^m

Wie bei elliptischen Kurven in den realen Zahlen gilt auch hier $P + (-P) = O$, der Punkt in der Unendlichkeit. Und weiter gilt $P + O = P$ für alle Punkte P auf der elliptischen Kurve.

Punktverdopplung

Wenn die x-Koordinate des zu verdoppelnden Punktes 0 ist, gilt $2P = O$. Andernfalls lässt sich $2P = R$ berechnen nach:

$$s = \frac{x_P + y_P}{x_P}$$

$$x_R = s^2 + s + a$$

$$y_R = x_P + (s+1)x_R$$

Gleichung 5.6: Punktverdopplung in F_{2^m}

Dabei ist *a* wieder ein Parameter der elliptischen Kurve und *s* die Steigung der Tangente.

5.4 Sicherheit

Die Auswahl einer Kurve ist ein langsamer, komplexer, aber wichtiger Prozess, um sicherzustellen, dass eine elliptische Kurve die an sie gestellten Sicherheitsanforderungen erfüllt. Mittlerweile sind einige Kurven als unsicher klassifiziert worden, welche aber glücklicherweise im Vergleich zur Anzahl möglichen Kurven nur einen kleinen Bruchteil ausmachen und anhand von wenigen Merkmalen erst noch ziemlich einfach zu erkennen sind. In ein F_p-Feld lassen sich ca. $2p$ elliptische Kurven legen, wobei auf jeder an die *p* Punkte liegen. Aktuell ist es am effizientesten, elliptische Kurven nach einem Zufallsverfahren zu generieren. Dabei werden die Parameter solange zufällig gewählt, bis eine Kurve gefunden ist, die nicht als unsicher erkenntlich ist. Das Mass der Sicherheit eines auf elliptischen Kurven basierenden Kryptosytems ist gegeben durch den grössten Primteiler der Anzahl Punkte (Ordnung) auf der Kurve. Folglich ist die Sicherheit nicht (!) direkt mit der Grösse des Feldes gekoppelt, eine Kurve in F_{167} könnte unter Umständen gleich sicher sein wie eine Kurve in F_{247}. Unklar ist zudem, wie sich die Schlüssellängen im Vergleich von F_p- und F_{2^m}-Implementationen verhalten, letzte Erkenntnisse liessen im Vergleich mit RSA-1024 darauf schliessen, dass die Sicherheit einer elliptischen Kurve in einem 162-Bit-F_p-Feld etwa die gleiche Sicherheit bietet, wie in einem 171..180-Bit-F_{2^m}-Feld. Es existieren aber auch Behauptungen, dass die Sicherheit in beiden Feldtypen identisch sein soll.

Die Grundlage jedes kryptographischen Systems ist ein komplexes mathematisches Problem, für das es keine einfachere Lösung gibt. Das heisst, um das System zu brechen, ist es nötig, das mathematische Problem zu lösen. Das DLP (discrete logarithm problem) ist die Grundlage für die Sicherheit vieler Kryptosysteme einschliesslich den elliptischen Kurven. Genauer basiert ein Kryptosystem, das auf elliptischen Kurven aufbaut, auf der Komplexität des ECDLP (elliptic curve discrete logarithm problem). Das ECDLP basiert auf der Unvorhersagbarkeit der Skalarmultiplikation.

Gegeben sind dabei die Punkte *P* und *Q*. Nun ist eine Zahl k zu suchen, dass $kP = Q$, wobei *k* als diskreter Logarithmus von *Q* bezeichnet wird.

Als Beispiel wird die Kurve $y^2 = x^3 + 9x + 17$ im Feld F_{23} betrachtet:

Was ist der diskrete Logarithmus k von $Q = (4,5)$ zur Basis $P = (16,5)$?

Um k zu bestimmen wäre es ein Ansatz, alle Vielfachen von P zu berechnen, bis Q gefunden ist. Die ersten Vielfachen wären in diesem Beispiel:

$P = (16,5)$, $2P = (20,20)$, $3P = (14,14)$, $4P = (19,20)$, $5P = (13,10)$, $6P = (7,3)$, $7P = (8,7)$, $8P = (12,17)$, $9P = (4,5)$.

Da nun $9P = (4,5) = Q$ ist, ist k=9 der diskrete Logarithmus von Q zur Basis P. In einem realen Einsatz in der Praxis wäre natürlich k so gross, damit es praktisch nicht möglich wäre, k in dieser Weise zu ermitteln.

Das ECDLP wird als komplexer eingeschätzt, als die bekannten DLP und IFP (integer factorization problem). Oder anders formuliert wurden bis heute für die anderen beiden Probleme Algorithmen entwickelt, die das jeweilige Problem schneller lösen, als jeder bekannte Algorithmus das ECDLP löst. Aus diesem Grund benötigt ein EC-Kryptosystem weniger lange Schlüssel als RSA. Der Aufwand, um ein Kryptosystem zu brechen, wird in der Literatur meist als MIPS-Jahre bezeichnet, was bedeutet, dass dazu ein Rechner mit einer Leistung von einer MIPS (million instructions per second) die angegeben Anahl Jahre beschäftigt ist.

Time to break in MIPS years	***RSA/DSA key size (bits)***	***ECC key size (bits)***	***RSA/ECC key size ratio***
10^4	512	106	5:1
10^8	768	132	6:1
10^{11}	1'024	160	7:1
10^{20}	2'048	210	10:1
10^{78}	21'000	600	35:1

Tabelle 5.1: RSA/ECC-Schlüssellängen für äquivalente Sicherheit

Aus und Tabelle 5.1 ist klar ersichtlich, dass die benötigten Schlüssellängen mit steigenden Sicherheitsanforderungen bei EC-Kryptosystemen deutlich besser skalieren als bei RSA. Da die Berechnungen bei ECC also in viel kleineren Feldern durchgeführt werden können, wird auch ein grosser Geschwindigkeitsvorteil erreicht.

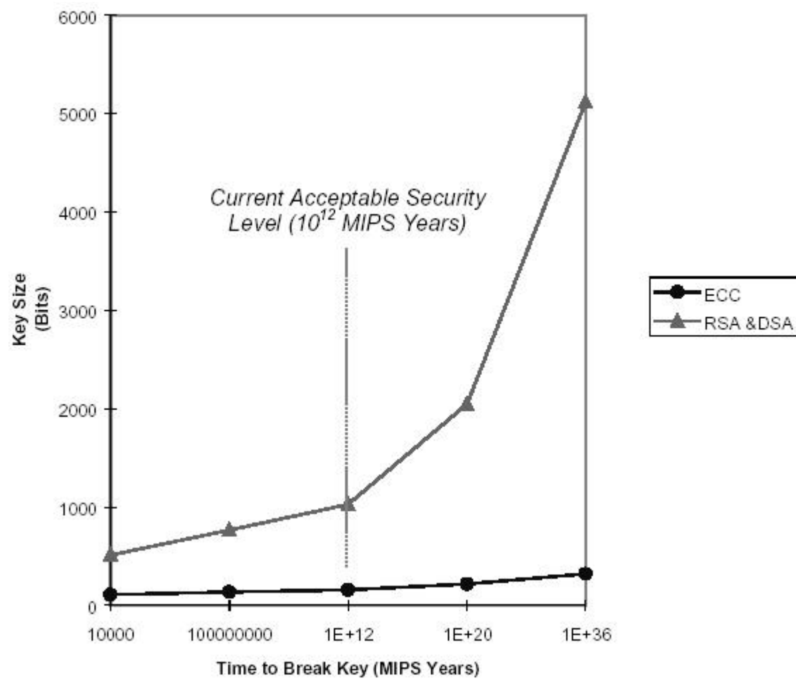


Abbildung 5.8: Vergleich des Aufwandes um RSA/ECC zu brechen (Quelle: [CeSec])

Bis heute sind keine grossen Schwachstellen von EC-Kryptosystemen entdeckt worden. Im Juli 2000 wurde aber eine 108-Bit-EC-Verschlüsselung geknackt. Dafür liefen 9'500 Parallelrechner vier Monate lang.

6 EC-Anwendungen in der Kryptographie

Die Anwendungen der elliptischen Kurven in der Kryptographie sind nicht weit zu suchen, denn wie schon gezeigt basieren EC-Kryptosysteme auf dem ECDLP, das eine direkte Analogie zum DLP aufweist. Darum war es eigentlich offensichtlich, ein auf EC adaptiertes DH-Schlüsselaustausch-Schema zu definieren.

6.1 Schlüsselaustausch

Um über eine unsichere Strecke einen gemeinsamen Schlüssel für eine symmetrische Verschlüsselung zu erzeugen, wurde auch für EC-Kryptosysteme ein Diffie-Hellman-Verfahren entworfen, das allgemein als ECDH bekannt ist. ECDH funktioniert wie der „normale" Schlüsselaustausch nach Diffie-Hellman, der ja folgendermassen definiert ist:

$$A = g^a \, mod \, p$$
$$B = g^b \, mod \, p$$

Jede Seite ermittelt jeweils einen zufälligen Wert *a* und *b* und exponiert damit eine bekannte, gemeinsame Basis *g*. Die Resultate A und B werden in dieser Form gegenseitig übermittelt. Den ausgetauschten Wert exponentieren die beiden Parteien nun nochmals mit ihrem eigenen Wert:

$$s = B^a \, mod \, p$$
$$s = A^b \, mod \, p$$
$$(s = g^{ab} \, mod \, p)$$

Beide Seiten erhalten durch diesen Austausch den identischen Wert *s*, der dazu verwendet werden kann, einen Schlüssel für eine symmetrische Verschlüsselung zu generieren. Die Analogie dazu mit elliptischen Kurven ist nun als Elliptic Curve Diffie-Hellman (ECDH) definiert:

$$Q_A = aP$$
$$Q_B = bP$$

Beide Seiten generieren wieder jeweils einen zufälligen Wert *a* und *b* und führen eine Skalarmultiplikation mit einem bekannten, gemeinsamen Basispunkt *P* durch. Die so berechneten Punkte können nun ausgetauscht werden, durch das ECDLP wird verhindert, dass *a* und *b* während der Übermittlung abgefangen und ermittelt werden können. Die beiden Parteien werden nach dem gegenseitigen Empfang die erhaltenen Punkte nochmals mit den eigenen Skalaren *a* und *b* multiplizieren und erhalten so einen Punkt *S*:

$$S = aQ_B$$
$$S = bQ_A$$
$$(S = abP)$$

Durch den Austausch entsteht ein Punkt *S*, dessen x-Koordinate (nach [X9-63]) zur Schlüsselgenerierung für eine symmetrische Verschlüsselung verwendet werden kann.

6.2 Public-Key-Verfahren

Obwohl es theoretisch möglich wäre, ein RSA-ähnliches Verfahren auch mit elliptischen Kurven zu definieren, hat man darauf verzichtet. Denn diese würden ja wie RSA auf dem IFP aufbauen und würden so keine erweiterte Sicherheit bieten, weil das Problem weiterhin die Faktorisierung wäre. Auch eine Adaption des ElGamal-Schemas wäre theoretisch machbar gewesen, bis heute aber ist ECIES das einzige Public-Key-Verfahren, das auf elliptischen Kurven aufbaut. In [X9-63] wurde ECIES noch als „Elliptic Curve Augmented Encryption Scheme“ und andersweitig auch einfach als „Elliptic Curve Encryption Scheme“ bezeichnet. ECIES wird in [SEC1] unter dem Namen „Elliptic Curve Integrated Encryption Scheme“ als rein hybrides System eingeführt. Hinter diesem Standard steht die secg (Standards for Efficient Cryptography Group), welche auf Initiative von Certicom gegründet wurde.

Eigentlich baut ECIES direkt auf ECDH auf. Der einzige Unterschied ist, dass eine Seite als Public-Key einen fixen Punkt hat. Zum Public-Key gehört auch noch die verwendete Kurve und der Generator-Punkt. Der entsprechende Private-Key wäre dann der diskrete Logarithmus dieses fixen Punktes.

Wenn nun eine Partei A ein solches Schlüsselpaar generiert hat, kann sie den Public-Key-Teil (den Punkt) publizieren:

1. *Wahl oder Generierung einer Kurve E, deren Anzahl Punkte durch eine möglichst grosse Primzahl n teilbar sein sollte*
2. *Wahl eines Punktes P auf E*
3. *Wahl einer zufälligen Zahl d_A*
4. *Berechnen von $Q_A = d_A P$*
5. $\textit{Public-Key} = (E, n, P, Q_A),\quad \textit{Private-Key} = (d_A)$

Will eine Partei B nun eine sichere Botschaft an Partei A übermitteln, nimmt sie den veröffentlichten Punkt Q auf und generiert ihrerseits ein temporäres Schlüsselpaar. Mit dem zufälligen Wert wird sie den Punkt von Partei A multiplizieren und so einen Schlüssel für eine symmetrische Verschlüsselung erhalten. Partei B hat also auf ihrer Seite den ECDH-Key-Exchange vollzogen. Nun kann die Botschaft symmetrisch chiffriert und übermittelt werden:

1. *Wahl einer zufälligen Zahl d_B*
2. *Berechnen von $Q_B = d_B P$*
3. *Berechnen von $d_B Q = d_B d_A P = (x, y)$*
4. *Berechnen eines symmetrischen Schlüssels S aus x*
5. *Verschlüsseln der Mitteilung M mit dem vereinbarten Cipher*
6. *Übermitteln von Q_B und $\textit{Cipher}(M, S)$*

Mitgesendet wird auch der temporäre Public-Key von Partei B, mit diesem Teil und dem eigenen Private-Key kann auch Partei A den Schlüsselaustausch vollziehen und so die symmetrisch verschlüsselte Botschaft dechiffrieren:

1. *Berechnen von* $d_A Q_B = d_A d_B P = (x, y)$
2. *Berechnen eines symmetrischen Schlüssels S aus x*
3. *Entschlüsseln der Mitteilung mit dem vereinbarten Cipher*

6.3 Unterschriften

Momentan ist das einzige Signatur-Schema, das auf Kryptographie mit elliptischen Kurven basiert, ist ECDSA (Elliptic Curve Digital Signature Algorithm). ECDSA ist spezifiziert in [X9-62]. Wie für ECIES wird auch bei ECDSA ein Schlüsselpaar vorausgesetzt, dies kann generiert werden durch:

1. *Wahl oder Generierung einer Kurve* E*, deren Anzahl Punkte durch eine möglichst grosse Primzahl* n *teilbar sein sollte*
2. *Wahl eines Punktes* P *auf* E
3. *Wahl einer zufälligen Zahl* d
4. *Berechnen von* $Q = dP$
5. $\textit{Public-Key} = (E, n, P, Q)$, $\textit{Private-Key} = (d)$

Mit diesem Schlüsselpaar wird dann eine Signatur erzeugt:

1. *Wahl einer zufälligen Zahl* k
2. *Berechnen von* $kP = (x_1, y_1)$ *und* $r = x_1 \bmod n$

   (x_1 sollte nicht 0 sein, sonst → 1.)
3. *Berechnen von* $s = k^{-1}(\text{SHA-1}(M) + dr) \bmod n$

   (wenn $s = 0$ ist, existiert $s^{-1} \bmod n$ nicht → 1.)
4. *Die Signatur für M ist* (r, s)

Die erzeugte Signatur kann nun von einer zweiten Partei überprüft werden:

1. *Erhalt des Public-Keys*
2. *Überprüfen, dass* r *und* s *in* $[1, n-1]$ *sind*
3. *Berechnen von* $w = s^{-1} \bmod n$ *und* $\text{SHA-1}(M)$
4. *Berechnen von* $u_1 = \text{SHA-1}(M)\, w \bmod n$ *und* $u_2 = rw \bmod n$
5. *Berechnen von* $u_1 P + u_2 Q = (x_0, y_0)$ *und* $v = x_0 \bmod n$
6. *Signatur wird akzeptiert, wenn* $v = r$

7 Grundlagen von Finite Fields

7.1 Einführung

Ein „Feld“ ist eine Menge von Zahlen, für die zwei arithmetische Operationen definiert sind.

In unserem alltäglichen Leben rechnen wir in der Menge der reellen Zahlen „R“ mit den Operationen Plus „+“ und Mal „·“ („-“ und „/“ können aus diesen abgeleitet werden). Wir rechnen also in einem Feld (R,+,·) ohne es zu wissen. Uns sind die Vorteile dieses Systems gar nicht bewusst, weil wir normalerweise nicht wissen, dass man Systeme kreieren könnte, in denen das Rechnen viel schwieriger sein würde. Solche Systeme hat man früher aber kreiert und benutzt, weil man es nicht besser wusste. Die Mathematiker sind nicht von heute auf morgen auf dieses praktische System mit den beiden Operationen gekommen. Was es auch noch praktisch macht, ist die Wahl des Zahlensystems (dezimal).

Die Eigenschaften der Felder hängen von ihrer Grösse ab und davon, wie die beiden Operationen definiert sind. Die mathematischen Definitionen dafür werden später erklärt. Das Ergebnis der Operationen muss übrigens immer im Feld bleiben.

Da unser gebräuchliches Feld (R,+,·) unendlich gross ist, kann man damit die Theorie, die hinter Feldern steckt, nicht anschaulich erklären. Es wird direkt auf die endlichen Felder (Finite Fields) umgestiegen. Das Wort „endlich“ bezieht sich übrigens auf die Anzahl der Elemente und nicht auf den möglichen Zahlenbereich. Wenn wir aus der Menge der reellen Zahlen ein endlichen Abschnitt {0..1} nehmen, so hat es immer noch unendlich viele Zahlen in dieser Menge. Somit ist das kein endliches Feld.

Aus der Menge der Ganzen Zahlen Z={-   ,...-2,-1,0,1,2,...,   } verwenden wir für das Feld nur eine Teilmenge Z_5={0,1,2,3,4}.

Wenn wir die gebräuchliche Arithmetik für diese Zahlenmenge verwenden, kann man dies wie folgt darstellen:

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	5
2	2	3	4	5	6
3	3	4	5	6	7
4	4	5	6	7	8

·	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	6	8
3	0	3	6	9	12
4	0	4	8	12	16

Hier wurde also für jede Zahlenkombination einzeln definiert, was die Operationen „+“ und „·“ bedeuten. Das ist die explizite Art, eine Arithmetik zu definieren.

Einige Ergebnisse würden jedoch die Menge Z_5 verlassen (z.B. 4+3 = 7). Die „Welt“ besteht in diesem Fall aber nur aus den Elementen 0,1,2,3,4. In allen anderen Fällen gäbe es kein Ergebnis. Ein billiger Taschenrechner, welcher mit der normalen Arithmetik rechnet und dessen Elemente 0...99'999'999 sind, kann auch nichts mit 99'999'999 + 1 anfangen. Wenn man es genau nimmt, rechnet dieser Taschenrechner nicht in einem „Feld“.

Die normale Addition und Multiplikation sind also nicht brauchbar für ein finites Feld.

Eine für endliche Felder geeignete Arithmetik ist die „Modulare Arithmetik“.

Wir rechnen alle Ergebnisse der Addition und Multiplikation Modulo 5 (für die Menge Z_5).

Statt e=a·b verwenden wir e=a·b *modulo* n.

z.B.: 4 * 4 modulo 5 = 16 modulo 5 = 1

(16 dividiert mit 5 gibt 3 mit 2 als Rest, interessant ist für uns nur der Rest)

Für die Modulare Addition und Multiplikation verwenden wir weiterhin die Notation „+“ und „·“.

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

·	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Bei der Definition der expliziten Arithmetik weiter oben haben wir die Zahlen mit Hilfe von Regeln gesetzt (angelehnt an die gebräuchliche Arithmetik). Man könnte die Zahlen aber auch zufällig setzen. In den meisten Fällen bekommt man aber eine völlig unbrauchbare Arithmetik, die gar kein Sinn macht. Damit diese in Kryptographie-Praxis brauchbar ist, muss ausserdem die Anzahl der Elemente der Menge im Bereich von 10^{50} bis 10^{500} liegen (je nach Krypto-Algorithmus). Bei dieser Grösse kann eine LookUp-Tabelle für die beiden Operationen gar nicht mehr gespeichert werden. Die Arithmetik wäre auch nicht mehr prüfbar. Deswegen braucht es eine Arithmetik, bei der das Ergebnis für jedes Wertepaar berechnet werden kann.

e = a·b *modulo* n.

Dabei kann das Ergebnis e mit dieser Berechnungs-Regel für jedes a und b berechnet werden. Der Aufwand dafür hält sich auch in einem Feld der Grösse 10^{500} in Grenzen.

Das Ziel ist es, eine Arithmetik zu kreieren, mit der die Hardware effizient rechnen kann. Das muss nicht unbedingt die gleiche sein, die für Menschen geeignet ist. Es ist schwierig sich vorzustellen, dass es überhaupt andere brauchbare Arithmetiken geben kann, aber in der Hardware wird die gebräuchliche Arithmetik schon lange mit Hilfe der boolschen Arithmetik implementiert. Das war aber nicht von Anfang an so. Zuerst wurde übrigens das Zahlensystem vom dezimalen auf das binäre gewechselt.

Beispiele von brauchbaren Arithmetiken:

- Gebräuchliche Arithmetik mit +,· (-,/).
- Boolsche Arithmetik mit and, or, (not, xor, ...).
- Modulo-Arithmetik mit der normalen Addition und Multiplikation (nur modulo gerechnet).
- Polynom-Modulo-Arithmetik. Dies ist auch eine Modulare Arithmetik. Sie besitzt zwei Operationen deren Eigenschaften teilweise der gebräuchlichen Arithmetik und teilweise der boolschen Arithmetik ähneln. Auch wenn diese Arithmetik für Menschen ungeeignet ist, ist sie in Hardware sehr effizient realisierbar und besonders in der Kryptographie und der Fehlerkorrektur verwendbar.

7.2 Feld-Definition

Wie am Anfang diese Kapitels gesagt wurde, sind „Felder“ Mengen von Zahlen, für die zwei Operationen definiert sind. Hier soll nun erklärt werden, welche Kriterien für diese Feldgrösse (Grösse der Zahlenmenge) und die Operationen gelten müssen, damit eine brauchbare Arithmetik entsteht. Etwas weiter unten wird anhand eines Beispiels gezeigt, was die einzelnen Bedingungen bedeuten und die Konsequenzen der Nichteinhaltung. Die weiter oben aufgeführten Arithmetiken erfüllen alle Voraussetzungen für ein Feld.

Mathematische Feld-Definition:

- abgeschlossen (die Ergebnisse der Operationen bleiben in der Menge)

    - ... gilt auch als abgeschlossen

    wenn die Menge der Elemente endlich ist, heisst das Feld „Finite Field“.

- eine Operation „like +“
    - *assoziativ* $(a+b)+c = a+(b+c)$
    - *kommutativ* $a+b = b+a$
    - additives inveses z.B.: $inv(a) = -a$

        zu jedem Element muss es ein inverses Element geben

    - additives neutrales (Null-Element) z.B: $a+0 = a$
- eine zweite Operation „like ·“
    - *assoziativ* $(a\cdot b)\cdot c = a\cdot(b\cdot c)$
    - *kommutativ* $a\cdot b = b\cdot a$
    - *distributiv* $a\cdot(b+c)=(a\cdot b)+(a\cdot c)$ *(ist die Kopplungsstelle der beiden Operation)*
    - multiplikatives inverses: z.B.: $inv(a)=a^{-1}$

        zu jedem Element muss es ein inveses Element geben (ausser null-element)

    - multiplikatives neutrales (Eins-Element) z.B.: $a\cdot 1=a$
    - multiplikatives null-element: z.B. $a\cdot 0=0$
    - keine Nullteiler: wenn $a\neq 0$ und $b\neq 0$ muss das Ergebnis $a\cdot b \neq 0$ sein.

Es wird extra auf ein Namenskonflikt hingewiesen: Das „additive neutrale Element“ wird oft als Null-Element bezeichnet, welches jedoch bei der Multiplikation eine andere Bedeutung hat.

An dieser Stelle soll nicht unerwähnt bleiben, dass es Vorstufen zu einem Feld gibt. Diese heissen Gruppe und Ring:

- Eine Gruppe besteht aus einer Menge und nur einer Operation. Von der Operation wird verlangt, dass sie assoziativ ist. Für jedes Element muss es ein inverses geben und es muss ein neutrales Element in der Menge geben.

Ein Ring besitzt bereits zwei Operation, die Regeln sind jedoch nicht so restriktiv wie beim Feld.

Je nach dem, wie die Arithmetischen Operationen definiert sind, sind nur gewisse Feldgrössen erlaubt. Bei der Modulo-Arithmetik mit gebräuchlicher Addition und Multiplikation muss die Feldgrösse (also die Anzahl Elemente) prim sein.

Einige Beispiele:

$Z_2 = \{0,1\}$

$Z_7 = \{0,1,2,3,4,5,6\}$

$Z_{11} = \{0,1,2,3,4,5,6,7,8,9,10\}$

Bei der Polynom-Arithmetik muss die Feldgrösse nicht mehr prim sein. Ihre Primfaktoren müssen jedoch alle gleich sein: 2^m, 3^m, 5^m.

Einige Beispiele:

$Z_2 = Z_2^1 = \{0,1\}$

$Z_4 = Z_2^2 = \{0,1,2,3\}$

$Z_8 = Z_2^3 = \{0,1,2,3,4,5,6,7\}$

$Z_9 = Z_3^2 = \{0,1,2,3,4,5,6,7,8\}$

7.3 Feldbeispiel mit Begriff-Erklärung

Um zu zeigen, wie wichtig die Feldgrösse ist, machen wir ein Beispiel mit einer Feldgrösse, die für die normale Modulo-Arithmetik ungeeignet ist. Es soll gleichzeitig die Bedeutung der einzelnen Elemente erklärt werden

Die Menge ist $Z_8=\{0,1,2,3,4,5,6,7\}$.

Die Feldgrösse 8 ist nicht mehr prim, der Primfaktor 2 ist in dritter Potenz. Man spricht in diesem Fall von einem „Feld der Charakteristik 2^m“. Um die Arithmetik zu bilden, versuchen wir hier die Modulo-Arithmetik mit normaler Addition und Multiplikation (modulo 6 in diesem Fall) einzusetzen. Es wird nun geprüft, ob die entstandenen Operationen für das Feld brauchbar sind.

7.3.1 Das neutrale Element

Das additive neutrale Element ist die „0“. Wenn man zu einer beliebigen Zahl die „0“ addiert, bleibt das Ergebnis immer noch gleich.

Das multiplikative neutrale Element ist die „1“. Wenn man eine beliebige Zahl mit „1“ multipliziert bleibt das Ergebnis gleich.

Das Ganze ist auch kommutativ. Das heisst $5+0 = 0+5$ und $5 \cdot 1 = 1 \cdot 5$

additives neutrales Element

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6

multiplikatives neutrales Element

?	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	4	7	2	5
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

7.3.2 Das inverse Element

Nun prüfen wir, ob es für jedes Element bei der Addition ein Inverses gibt.

Beispiele von inversen Elementen bei der Addition a   -a :

Im Feld(R,+,·)   6   -6,   -3   3

Im endlichen Feld(Z_8,+,·)   6   2 3   5

Wenn zu einem Element sein Inverses hinzuaddiert wird, gibt es als Ergebnis das additive neutrale Element $a + -a = 0$:

Im Feld(R,+,·) $6 + -6 = 0$, $-3 + 3 = 0$

Im endlichen Feld(Z_8,+,·) $6 + 2 = 0$, $3 + 5 = 0$

Die Subtraktion kann somit auf die Addition mit dem Inversen des Subtrahenden zurückgefürhrt werden:

Im Feld(R,+,·) $6 - 2 = 6 + -2 = 4$, $3 - 4 = 3 + -4 = -1$

Im endlichen Feld(Z_8,+,·) $6 - 2 = 6 + 6 = 4$, $3 - 4 = 3 + 4 = 7$

In der Additionstabelle taucht in jeder Spalte und Zeile das additive neutrale Element auf, somit gibt es für jedes Element ein Inverses.

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	4	7	2	5
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

Nun prüfen wir, ob es für jedes Element bei der Multiplikation ein Inverses gibt. Das erkennt man daran, dass es in jeder Spalte und Zeile das multiplikative neutrale Element gibt, die „1“. Dies ist aber im Feld(Z_8,+,·) nicht der Fall. Für die Zahlen 2, 4, 6 sollte es ein Inverses geben, es gibt aber keines. Die Zahl 0 ist etwas Besonderes, für sie darf es kein Inverses geben.

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	4	7	2	5
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

Das endliche Feld (Z_5,+,·) hat eine gültige Multiplikation, weil für jedes Element ausser dem multiplikativen Null-Element ein Inverses Existiert:

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

·	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Beispiele von inversen Elementen in der Multiplikation a a^{-1}:

Im Feld(R,+,·) 5 0.2, 4 ¼

Im endlichen Feld(Z_8,+,·) 5 5, 4 ? (für die 4 gibt es kein Inverses im diesem Feld)

Im endlichen Feld(Z_5,+,·) 3 2 (siehe Daten des Feldes früher im Text)

Wenn ein Element mit seinem Inversen multipliziert wird, gibt es als Ergebnis das multiplikative neutrale Element $a + a^{-1} = 1$:

Im Feld(R,+,·) $5 \cdot 0.2 = 1$, $4 \cdot ¼ = 1$

Im endlichen Feld(Z_8,+,·) $5 \cdot 5 = 1$

Im endlichen Feld(Z_5,+,·) $3 \cdot 2 = 1$

Die Division kann auf die Multiplikation mit dem inversen Element des Divisors zurückgeführt werden:

Im Feld(R,+,·) $4 / 5 = 4 \cdot 0.2 = 0.8$

Im endlichen Feld(Z_8,+,·) $6 / 2 = 6 \cdot 6 = 4$, (bei 6 / 4 gibt es aber keine Lösung)

Im endliche Feld(Z_5,+,·) $4 / 3 = 4 \cdot 2 = 1$

7.3.3 Das Null-Element

Wenn ein Element mit dem multiplikativen Null-Element multipliziert wird, ist das Ergebnis in einem Gültigen Feld immer noch das Null-Element. z.B.: $4 \cdot 0 = 0 \cdot 4 = 0$

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	4	7	2	5
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

7.3.4 Nullteiler

Nullteiler darf es in einem gültigen Feld nicht geben, im Feld (Z_8,+,·) gibt es aber einige.

z.B.: 6·4 = 0 obwohl keiner der beiden Elemente gleich Null war.

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	4	7	2	5
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

Wir haben also festgestellt, dass die entstandene Arithmetik unbrauchbar ist, weil bei der Multiplikation nicht zu jedem Element ein Inverses existiert und weil die Multiplikation Nullteiler hat.

Wenn alles bis jetzt zufriedenstellend gelaufen wäre, müssten jetzt noch für jede Kombination die folgenden Gesetze geprüft werden:

7.3.5 Kommutativgesetz:

4+3 = 3+4 = 7
5·7 = 7·5 = 3

7.3.6 Assoziativgesetz:

(4·5)·3 = 4·3 = 4
4·(5·3) = 4·7=4

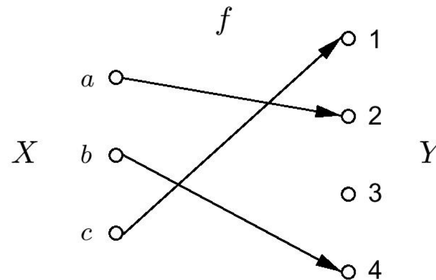
7.3.7 Distributivgesetz:

6·4 + 6·3 = 6·(4+3)
0 + 2 = 6·7
2 = 2

Obwohl diese drei Gesetze erfüllt werden, ist das entstandene Feld(Z_8,+,·) trotzdem unbrauchbar, da die Anforderungen an die Elemente nicht erfüllt werden. Eigentlich ist es somit gar kein Feld. Die Zahlen in der Additions- und Multiplikations-Tabelle könnte man jedoch auch anders setzen. Die Zahlenmenge Z_8 ist für die normale Modulo-Arithmetik ungeeignet. Wenn man für diese Zahlenmenge aber die Polynom-Modulo-Arithmetik verwendet, kann man damit ein gültiges Feld kreieren. Dies wird im Kapitel über die Grundlagen der EC-Kurven anhand eines Beispiels demonstriert.

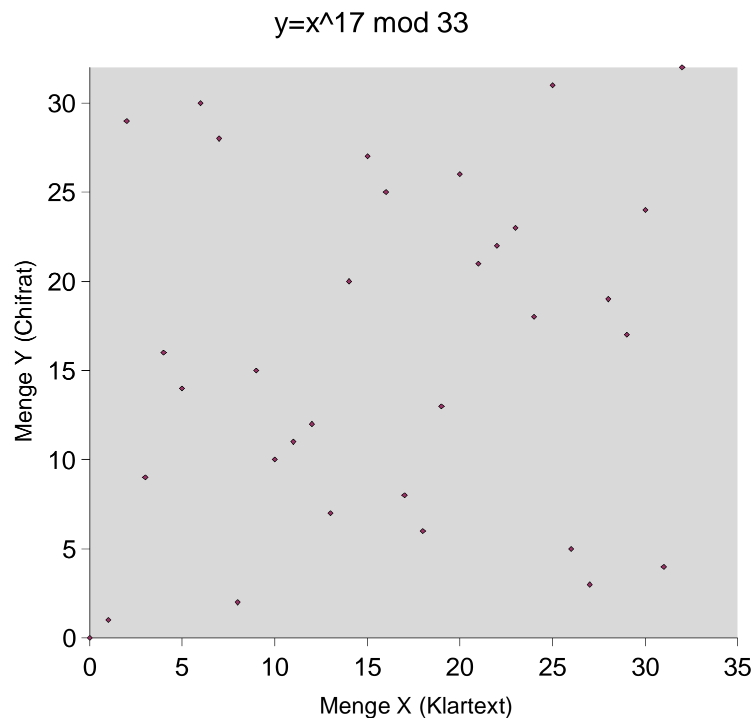
7.4 Funktionen in den Finite Fields

Der Sinn jeder Funktion ist es, die Elemente einer Menge auf eine andere abzubilden. Die nachfolgende Funktion bildet die Elemente aus der Menge X auf die Elemente der Menge Y ab.



Wir wollen die Funktion $y=x^{17}$ in dem endlichen Feld$(Z_{33},+,\cdot)$ mit der normalen modularen Arithmetik in einem Graphen darstellen. Die Potenz 17 von x bedeutet, dass die Operation „Multiplikation“, die für das Feld definiert ist, 16 mal angewendet wird: $y=x\cdot x\cdot x\cdot x\cdot x\cdot x\cdot x\cdot x\cdot x\cdot x\cdot x\cdot x\cdot x\cdot x\cdot x\cdot x\cdot x$.

Auf der X-Achse ist die Menge X (Menge Z_{33}) dargestellt und auf der Y-Achse ist die Menge Y dargestellt (gleich grosse Menge Z_{33}). *Es wird noch explizit darauf hingewiesen, dass das Feld eine Menge ist, und nicht der grau dargestellte Bereich im Funktionen-Graph oder der Inhalt der Operations-Tabellen.*



Die Werte der Funktion sind pseudo-zufällig verteilt. Somit sind auch die Werte der Menge X pseudo-zufällig auf die Menge Y abgebildet worden. Beispielsweise wurde 5 auf 14 abgebildet. Die wichtigste Frage ist nun, wie man vom Element 14 der Menge Y auf das richtige Element in der Menge X kommt. Das wird sehr aufwendig, wenn die Anzahl Zahlen in der Menge grösser wird.

7.5 Einsatzzweck der Finite Fields

Als die Eigenschaften der Finite Fields analysiert wurden, stellte man fest, dass es möglich war, Funktionen in diesen Feldern zu bilden, die keine inverse Funktion besassen oder diese äusserst schwierig zu finden waren.

Die Public-Key-Kryptographie baut auf solchen Einwegfunktionen auf. Bei einem bestimmten Typ von Einwegfunktionen gibt es eine Inverse, die für einen Angreifer praktisch unauffindbar ist. Nur der rechtmässige Empfänger kennt den geheimen Wert, mit dem er eine Umkehrfunktion bilden kann. Auf diese Art ist das RSA-Public-Key-Verfahren realisiert worden. Es gibt auch noch etwas andere Ansätze. Bei Dieffie-Hellman wird eine „echte" Einwegunktion verwendet, für die keine Inverse existiert oder zumindest keine bekannt ist.

Zum Beispiel ist $y=x^{10}$ mit der normalen Arithmetik keine Einwegfunktion. Wenn man die Potenz kennt (hier 10), kann man einfach eine Umkehrfunktion bilden: $x=y^{0.1}$. Ausserdem ist die Grösse von y als Chiffrat sehr unpraktisch für die Datenübertragung.

Hingegen ist $y=x^{17} \bmod 33$ eine einfache Einwegfunktion. Die Zahl x (Klartext) und y (Chiffrat) sind durch die Modulo-Operation auf den Bereich 0 bis 32 begrenzt.

Obwohl der Angreifer den Exponent 17 kennt (dieser stellt den Public-Key dar), ist es für ihn unmöglich zu einem Ciphertext y den Cleartext x zu finden, weil er den Private-Key nicht kennt. Der rechtmässige Empfänger besitzt aber den Private-Key (hier 13), mit dem er die inverse Funktion bilden kann und mit dieser den Klartext dechiffrieren kann:

$x=y^{13} \bmod 33$

Weil diese Funktion mit dem RSA-Algorithmus generiert wurde, gibt es hier eine Umkehrfunktion. Bei einem so kleinem Modulo (33) kann der Angreifer noch alle möglichen Exponenten durchrechnen.

$y=x^{e} \bmod 33$ für e={0, 1, .. ,31 ,32}

Falls keine Umkehrfunktion existiert, könnte auch eine Lookup-Tabelle für alle Werte gebildet werden. Dafür muss man die Ergebnisse für alle x-Werte (Klartext) berechnen und in einer Tabelle nach y (Chiffrat) ordnen lassen.

Als Exponent und als Modulo werden heutzutage 2000-stellige Binärzahlen verwendet, womit der Versuch, den Privat-Key zu ermitteln, fast unmöglich wird. Eine Lookup-Tabelle ist auch nicht mehr möglich, weil es im Universum weniger Atome gibt, als Einträge in dieser Lookup-Tabelle nötig wären.

8 Secure Sockets Layer - Grundlagen

Im vorhergehenden Kapitel sind die Grundlagen der verwendeten Verschlüsselungsverfahren erklärt worden. An dieser Stelle soll nun ein kurzer Einblick in die OpenSSL-Funkionalität gewährt werden, der für das Verständnis dieser Arbeit notwendig ist. Für dieses Kapitel wird vorausgesetzt, dass der Leser den Unterschied zwischen den Public-Key-Kryptoalgorithmen und den symmetrischen Kryptoalgorithmen kennt und weiss, wie das OSI-Layer-Model aufgebaut ist.

8.1 SSL-Platz im OSI-Layer-Model

Die Verschlüsselung und Signierung der Daten kann prinzipiell auf jedem OSI-Layer erfolgen. Ein sinnvoller Einsatz ist jedoch auf Layer 3 (IPsec), Layer 4 (SSL), Layer 7 (PGP,...)

SSL wird im OSI-Layer-Model zwischen der Transportschicht (Layer 4) und der Verarbeitungsschicht (Layer 5) eingefügt.

Verarbeitungsschicht (HTTP,FTP,...)
Secure Socket Layer
Transportschicht (TCP)
Vermittlungsschicht (IP)
Vermittlungschicht
Bitübertragungsschicht

SSL baut auf dem darunterliegenden TCP/IP-Protokoll auf.

Auf SSL können Applikationen aufsetzen, die vorher direkt mit dem TCP-Layer kommunizierten. Voraussetzung ist, dass sich diese Applikationen der Existenz von SSL bewusst sind.

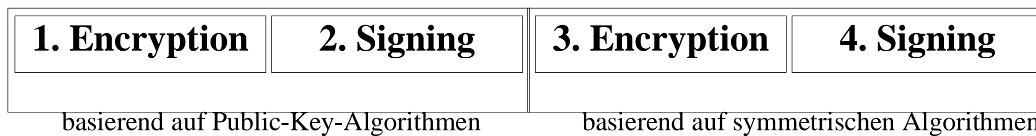
SSL nimmt die Daten von Applikationen entgegen, verschlüsselt sie und übergibt sie der Transportschicht. Vor der Kommunikation müssen jedoch die Parameter für die Verschlüsselung mit Hilfe des SSL-Handshake-Protokolls ausgetauscht werden.

8.2 Aufbau einer SSL-Cipher-Suite

Die Cipher Suite enthält 4 Angaben.

Der erste und der zweite Parameter definieren die Algorithmen, die für die Abwicklung des Handshakes notwendig sind. Das Ergebnis ist ein „Premaster-Secret“, dessen Länge je nach Public-Key-Algorithmus variieren kann. Das „Premaster-Secret“ stellt die verteilte geheime Information dar. Daraus wird ein „Master-Secret“ von 48 Byte Länge gebildet. Aus diesem wird das Schlüsselmaterial für die symmetrische Verschlüsselung und Signierung generiert.

Der dritte und der vierte Parameter definieren die Algorithmen, die vom Record Layer für die eigentliche Datenübertragung verwendet werden.



Die vier Parameter im Detail:

1. Public-Key-Verschlüsselungs-Algorithmus

   Um die geheimen Schlüssel für die symetrische Verschlüsselung und Signierung auszutauschen wird ein Public-Key-Verfahren benötigt. Die Public-Key-Verfahren sind sehr rechenintensiv, weswegen diese ausschliesslich für die Übertragung der symmetrischen Schlüssel verwendet werden und nicht für die eigentliche Datenübertragung.

2. Signatur-Verfahren basierend auf Public-Key-Algorithmen.

   Von einer Certificate Authority wird ein Zertifikat ausgestellt, in welchem der Public-Key des Servers mit einer digitalen Signatur signiert ist. Der Client kann nun durch die Prüfung der Signatur die Authentizität des Public-Keys sicherstellen.

3. Symetrischer Verschlüsselungs-Algorithmus

   Die eigentliche Datenübertragung ist mit symmetrischem Verschlüsselungs-Algorithmus gesichert. Die symmetrische Verschlüsselung weist einen sehr hohen Datendurchsatz auf. Das Problem hier ist die sichere Verteilung der symmetrischen Schlüssel. Dieses wird durch die Einbindung des Public-Key-Verfahrens gelöst.

4. Symetrisches Unterschriftverfahren

   Mit diesem wird die Authentizität der übertragenen Daten sichergestellt. Diese Verfahren unterscheiden sich grundlegend von denen, die auf Public-Key-Verfahren basieren. Eigentlich sind es Message-Digest-Funktionen. Der Initialisierungsvektor dieser Funktion ist der Secret-Key.

Beispiele für verwendete Cipher-Suites:

„ECDH_RSA_with_RC4_MD5“

1. Für die Bildung der geheimen Schlüssel wird ECDH verwendet.
2. Der ECDH-Public-Wert wird durch ein Zertifikat mit einer RSA-Unterschrift bestätigt.
3. Das symmetrische Verschlüsselungsverfahren ist RC4
4. Der verwendete Message-Digest-Algorithmus ist MD5

„DH_anonym_with_RC4_MD5“ (anonym heisst „ohne Zertifikat“)

„RSA_with_RC4_MD5“ (eigentlich bedeutet das „RSA_RSA_with_RC4_MD5“)

„RSA_with_DES_MD5“

8.3 Interner Aufbau des SSL-Layers

SSL ist in zwei Sub-Layer unterteilt. Der obere Sub-Layer besteht aus 4 Protokollen, diese bauen auf dem Record Layer auf.

Das Handshake-Protokoll und das Change-Cipher-Protokoll werden für den Aufbau einer Verbindung benötigt.

Wenn die Verbindung steht, wird das Application-Protokoll für den Transport der von den oberen Protokoll-Schichten kommenden Daten verwendet.

Für die Meldung von Fehlern gibt es schliesslich noch das Alert-Protokoll.

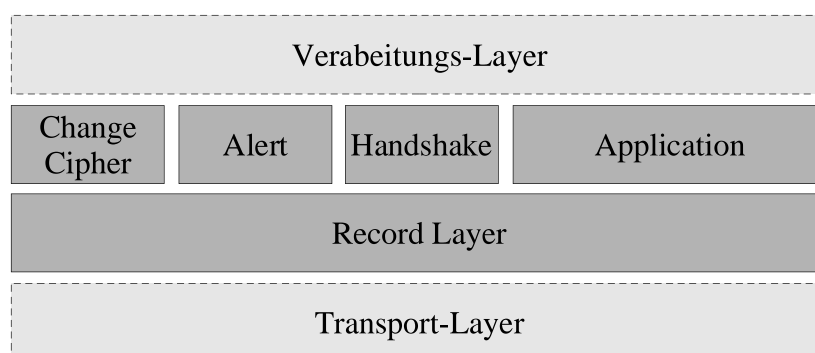


Bild 8.1: SSL-intern

Record Protokoll

Der Record Layer ist für die Fragmentierung, Verschlüsselung und Verpackung der Daten verantwortlich. Es können durchaus mehrere Meldungen in einen Record gepackt und als ein TCP/IP-Paket versendet werden.

Der Header des Record-Protokolls besteht aus:
- Protokoll-Bezeichnung
- Versionsnummer
- Länge des Records
- Message Body (beliebig viele Meldungen aneinander gereiht)
- Message Authentication Code (MAC).

Handshake-Protokoll

Dies ist das umfangreichste Protokoll des SSL-Layers. Es besteht aus 10 Message-Typen:

- HelloRequest
- ClientHello
- ServerHello
- Certificate
- ServerKeyExchange
- CertificateRequest
- ServerHelloDone
- CertificateVerify
- ClientKeyExchange
- Finished

Im nebenstehenden Bild ist z.B. die ClientHello-Message dargestellt.

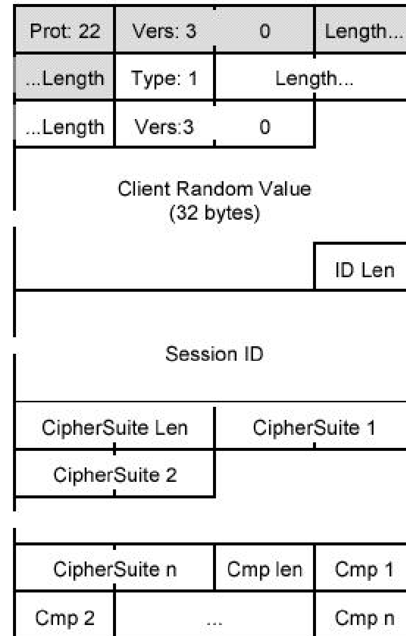


Bild 8.2: Hello Message
(Quelle: DA_Sna_00/3)

Application-Protokoll

Protokolle, die vorher direkt auf dem TCP/IP-Stack aufgesetzt waren, müssen nun über das Application-Protokoll gehen. Das kann HTTP, FTP, Telnet oder jedes andere Protokoll sein, welches aber den SSL-Stack bedienen können muss.

Change Cipher

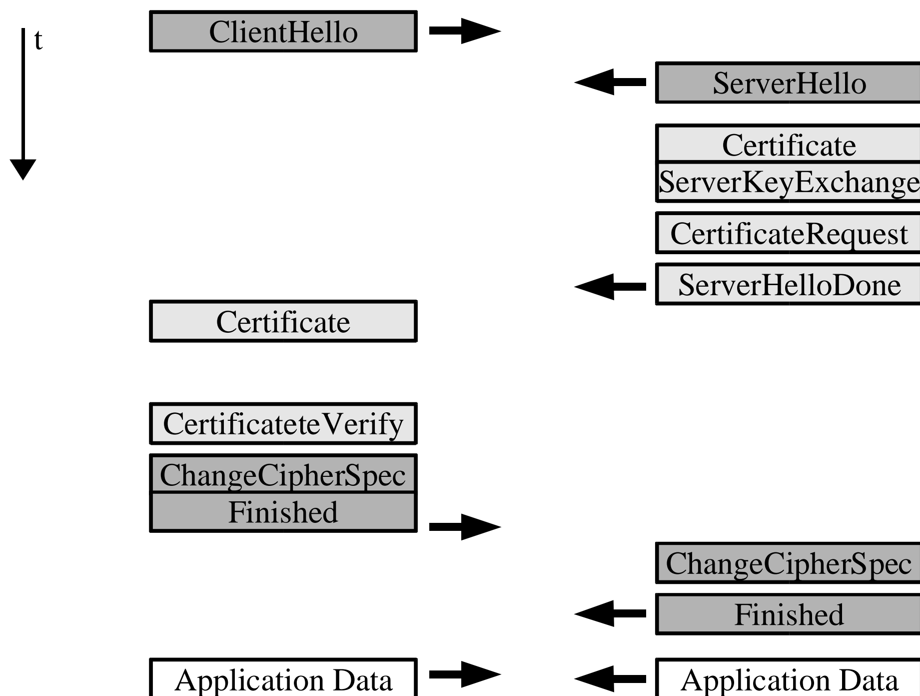
Dieses Protokoll besteht aus einer einzigen Meldung: „Change Cipher Spec“. Mit dieser Meldung wird der Umstieg auf die neue Cipher-Suite signalisiert.

8.4 SSL-Handshake

Das Ziel des Handshakes ist, die Parameter für den Aufbau eines verschlüsselten Kanals auszutauschen. Das sind:

- Kryptoverfahren für die eigentliche Datenübertragung
- Message Digest Algorithmus (welcher hier als ein digitaler Signatur-Algorithmus dient)

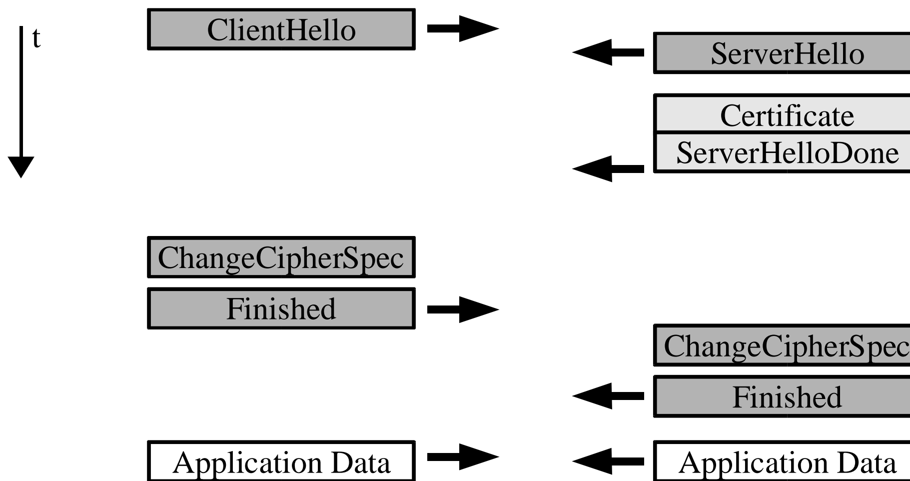
Im Folgenden soll ein Handshake mit allen möglichen Optionen dargestellt werden. Im realen Einsatz können je nach dem einige Meldungen in diesem Ablauf fehlen.



Aus dieser Zeichnung kann jedoch nicht abgeleitet werden, welche Meldungen in einem bestimmten Handshake-Fall übrig bleiben. Deswegen werden folgend die spezifischen Handshakes, die wir benötigt haben, aufgezeigt.

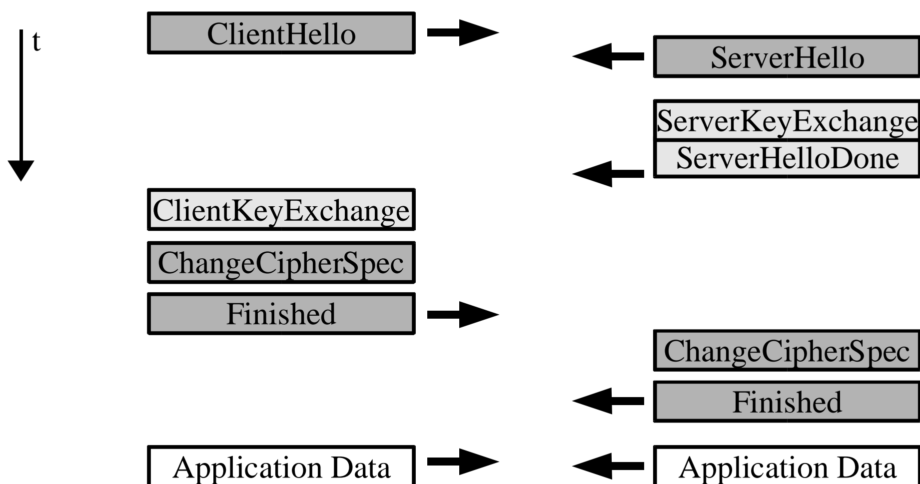
Handshake mit Zertifizierung des Servers

Wird verwendet bei:
- RSA (mit RSA-Signatur)
- ECDH mit RSA-Signatur
- ECDH mit DSA-Signatur:



Handshake ohne Zertifizierung des Servers.

Wird verwendet für anonymous ECDH oder anonymous DH.



8.5 Erklärung der Meldungen

8.5.1 Client Hello

Der Client sendet diese Meldung an den Server, um eine neue Verbindung aufzubauen oder eine inaktive Verbindung zu aktivieren. Diese Meldung enthält folgende Angaben:

- Protocol Version
- Session ID
- Cipher Suites - Liste
- Compressions Methoden - Liste
- Client Random Value (für jede Verbindung neu generiert)

Die Liste der Cipher Suites enthält die vom Client unterstützten Suites. Der Server wird eine passende auswählen.

8.5.2 Server Hello

Direkt nach dem Client Hello muss der Server mit einer Server Hello-Meldung antworten, sonst wird der Verbindungsaufbau abgebrochen.

Diese Meldung enthält:

- Bestätigung der Protocol Version
- Session ID
- ausgewählte Cipher Suite
- ausgewählte Kompressionsmethoden
- Server Random Value

8.5.3 Certificate Message (vom Server)

In dieser Meldung wird ein Zertifikat übermittelt. Damit die Verschlüsselung einen Sinn hat, muss ein Zertifikat gesendet und vom Client überprüft werden. Der anonyme Modus ist meist gar nicht sinnvoll und wird nicht unterstützt. Es gibt viele Attacken, die die Anonymität ausnützen, wie die „Man In The Middle“-Attacke.

8.5.4 Server Key Exchange Message

Diese Meldung wird gesendet, wenn kein Zertifikat gesendet wurde (anonymer Modus) oder die Zertifikat-Meldung nicht genügend Information zur Bildung der Premaster Secrets enthält. Die Bildung des Premaster Secrets und des Master Secrets wird im nächsten Kapitel behandelt.

8.5.5 Certificate Request

Mit dieser Meldung fordert der Server einen Client Zertifikat an.

8.5.6 Certificate Message (vom Client)

Falls der Server einen Zertifikat angefordert hat, muss der Client nun eines schicken.

Diese Art der Client-Authentifizierung ist jedoch von Nachteil, wenn der Client ein beliebiger Rechner sein soll. In dem Fall ist eine Nutzer-Identifizierung per Passwort die bessere bzw. die einzig mögliche Lösung.

8.5.7 Client Key Exchange

Nachdem der Server dem Client in der Server-Hello-Meldung mitgeteilt hat, welche Cipher Suite verwendet werden soll, sendet der Client die Parameter für die der Server benötigt. Das kann sein:

- Der Premaster Secret mit dem RSA-Public-Key des Servers verschlüsselt.
- Der Public-DH-Value des Clients ($g^{ClientSecret}$), den der Server mit seinem DH-Private-Key exponentieren muss ($g^{ClientSecret*ServerSecret}$).
- Der Public-ECDH-Wert des Clients (Ein Punkt bestehend aus zwei Werten: x,y).

8.5.8 Certificate Verify

Der Client kann damit eine explizite Überprüfung des Zertifikates anfordern. Ob das sicherheitstechnisch in irgendeiner Weise sinnvoll ist, ist fraglich.

8.5.9 Change Cipher Spec

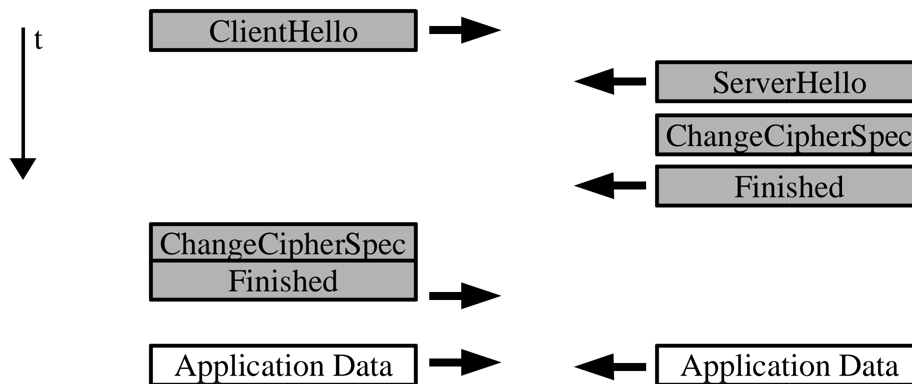
Die Change Cipher Spec-Meldung signalisiert den Umstieg auf die Cipher Suite und könnte sinngemäss ein weiterer Message-Typ im Handshake-Protokoll sein. Diese Meldung ist aber so wichtig, dass ihr eine eigene Protokollnummer zugewiesen wurde.

8.5.10 Finished

Diese Meldung signalisiert der Gegenpartei, dass der Handshake nun beendet ist. Für den Ersten, der diese Meldung gesendet hat, heisst es jedoch, dass er noch weitere Handshake-Meldungen vom Client empfangen können muss. Nach dieser Meldung steht die Leitung mit symmetrischer Verschlüsselung.

8.6 SSL-Handshake (resumed Session)

Am Ende des Datenaustausches gehen die Session-Daten nicht verloren (meist solange das Programm nicht beendet wurde). Das heisst man könnte die Session wiederherstellen ohne die Parameter neu aushandeln zu müssen. Hier also das Handshake nach einem Session-Resume:



Es müssen keine Parameter mehr ausgehandelt werden. Nach den beiden Hello-Meldungen folgt somit der Umstieg auf die neuen Cipher_Specs.

9 Problemanalyse und Lösungskonzept

9.1 Ausgangslage (Problem)

Über dem TCP/IP-Stack des Embedded Systems DK40@CHIP wurde ein SSL-Layer eingefügt. Dieser verwendet das RSA Public-Key-Verfahren, um die symmetrischen Schlüssel sicher vom Client zum Server zu übertragen. Die RSA-Verschlüsselung und Entschlüsselung sind aber sehr rechenintensiv. Auf heutigen leistungsfähigen PCs beträgt die Berechnungsdauer weniger als eine Sekunde. In Embedded Systems werden jedoch relativ leistungsschwache Prozessoren verwendet, weswegen die Ver-/Entschlüsselungszeit massiv ansteigt.

Im DK40@CHIP wurde ein 80186er verwendet. Dieser läuft mit einer Taktrate von 20MHz und besitzt einen CISC-Befehlssatz. Die gemessenen Zeiten für die RSA-Entschlüsselung auf dem Embedded-Server sind:

Schlüssellänge[Bit]	*Zeit [s]*
1024	47
2048	355

Im Moment wird ein 1024-Bit Schlüssel noch als sicher angesehen. In wenigen Jahren wird das nicht mehr der Fall sein, weswegen bereits heute der Umstieg auf die 2048-Bit-Schlüssel empfohlen wird. Solch lange Wartezeiten können im Normalfall keinem Benutzer zugemutet werden.

9.2 Lösungsansatz

Das Problem mit den langen Wartezeiten bei RSA hat einen Aufschwung bei einem anderen Krypto-Algorithmus bewirkt, welcher zwar komplizierter als RSA ist, dafür aber eine viel kürzere Berechnungszeit bei gleicher Sicherheit benötigt. Dieses Verfahren ist der Elliptic Curve Cryptoalgorithm. Die Sicherheit eines EC-Schlüssels mit einer Länge von 160 Bit ist vergleichbar mit der eines RSA-Schlüssels von 1024 Bit Länge. Der Grund dafür ist, dass in einer Zahl, die als RSA-Schlüssel dienen soll, einige Primfaktoren nicht vorkommen dürfen. Dadurch reduziert sich die Menge gültiger RSA-Schlüssel massiv. Bei einem 160-Bit EC-Schlüssel sind hingegen alle 2^{160} Kombinationen gültige Schlüssel.

Der zweite Vorteil der EC-Kryptoalgorithmen ist, dass bei einer Erhöhung der Schlüssellänge die Anzahl möglicher Kombinationen exponentiell ansteigt. Hingegen gehen bei RSA immer mehr Werte verloren, weil sie die verbotenen Primfaktoren enthalten.

Wenn man die Schlüsselmenge 2^{32} mal erhöhen will, reicht es bei ECC, den Schlüssel von 160 Bit auf 210 Bit zu erhöhen. Bei RSA wird eine Verdopplung der Schlüssellänge fällig:

RSA[Bit]	*EC [Bit]*
1024	160
2048	210

9.3 Vorgehen

Die EC-Implementation musste in mSSL, eine einfache Implementation eines SSL-Stacks, integriert werden. Diese lief auf einem Embedded-System mit einem 80186-Prozessor.

Der erste Vorschlag des Dozenten war es, die ECC-Grundfunktionen zu implementieren, diese in OpenSSL zu integrieren und dann auf das Embedded System wechseln.

Als erstes war es nötig, dass wir uns über einige Gebiete einen Überblick verschafften:

- Einarbeitung in ECC-Theorie und publizierte Implementationen
- Übersicht der OpenSSL-Implementation
  - Abschätzung des Integrationsaufwands der ECC-Funktionen und Strukturen
  - Übersicht der Bignum-Bibliothek (diese war als Fundament für die ECC-Implementation gedacht)
- Übersicht des mSSL-Stacks
- Konsequenzen des Wechsels von 32Bit =>16Bit-Rechner für das Rechnen mit Bignumbers.

Der OpenSSL-Code ist nicht gerade „human readable“, d.h. der Umfang dieses Stacks ist enorm und die Zusammhänge und Abhängigkeiten zwischen den einzelenen Bibliotheken ist komplex und unüberschaubar. Deswegen dauerte auch die Einarbeitungsphase relativ lange. Es zeichnete sich ab, dass der Integrationsaufwand in OpenSSL sehr hoch sein würde. Demgegenüber war mSSL viel einfacher und übersichtlicher, weswegen wir entschieden haben, die EC-Funktionen nicht in OpenSSL zu integrieren, sondern von Anfang an mSSL auf der Embedded-Umgebung zu verwenden.

Der Aufwand für die Einarbeitung in die ECC-Theorie war enorm. Es gibt viele unterschiedliche Algorithmen und noch viel mehr Implementationen, die oft auf bestimmte Feldtypen und elliptische Kurven spezialisiert sind. Zusätzlich muss erwähnt werden, dass in der Kryptologie die Tragweite von aufgestellten Gesetzen nicht auf einen Blick erkennbar sind. Es bieten sich ständig neue Einsichten in „alte“ Ideen. Wie z.B.: Wenn man die Idee von ECDH etwas modifiziert, kann man es als ein Public-Key-Verfahren mit zertifizierbaren Public-Keys betrachten.

Bei der Fülle der ECC-Implementationen war zum Teil auch Widersprüchliches zu lesen, was die Einarbeitung erschwert hatte. Als bei uns die Einarbeitung in ECC auf Hochtouren lief, haben wir festgestellt, dass in der OpenSSL-Community auch an einer ECC-Implementation gearbeitet wurde. Genauer gesagt wurde an deren Integration gearbeitet, die ECC-Implementation wurde von Sun Microsystems zur Verfügung gestellt. Erst am 18. September 2002, als die Integration in OpenSSL einen brauchbaren Zustand angenommen hatte, veröffentlichte Sun eine Pressemitteilung über diese Code-Schenkung. Die Sun-Implementation umfasst alle ECC-Cipher-Suites, die in [IETF-ECC02] zur Verwednung mit TLS spezifiziert sind, den ECDH-Schlüsselaustausch (in [X9-63] spezifiziert) und die Arithmetik in binären Feldern. Etwa zur Hälfte der Diplomarbeit erreichte die Integration einen Zustand, der es erlaubte, mit den Kommandozeilenbefehlen EC-Zertifikate und -Schlüssel zu generieren.

Wir entschieden uns deshalb, parallel an einer eigenen ECC-Implementation weiter zu arbeiten und gleichzeitig die ECC-Implementation von einem OpenSSL-Snapshot auf das Embedded-System zu portieren. Dadurch war die Erweiterung des SSL-Handshakes auf dem Embedded-System von der eigenen ECC-Implementation entkoppelt.

Wir mussten feststellen, dass der Handshake der mSSL-Implementation sehr unflexibel war und die Tragweite einer Änderung nur sehr begrenzt abschätzbar war. Das letztere vor allem deshalb, weil die Zustandsmaschine des Handshakes etwas unsauber implementiert wurde. Die Zustandsmaschine für den Protokoll-Handshake musste also neu erstellt werden. Der Handshake wurde nun schrittweise erweitert. Zuerst haben wir die Unterstützung für den DH-Algorithmus (Diffie-Hellman) hinzugefügt. Danach den anonymen ECDH-Schlüsselaustausch und zum Schluss ECDH mit RSA- und ECDSA-Unterschrift.

Zu diesem Zeitpunkt waren wir dann auch mit der EC-Theorie so weit, dass eine Beurteilung der ECC-Implementation von OpenSSL möglich war. Wir stellten fest, dass jeweils Verfahren verwendet wurden, die für generelle Einsatzgebiete allgemein als am effizientesten betrachtet werden. Zudem unterstützt diese Implementation eine Fülle von Standard-Kurven in Primfeldern und binären Feldern, was eine hervorragende Interoperabilität garantiert.

Parallel dazu wurde eine Eigenimplementation erstellt. Der Aufwand für die Implementierung der Grundfunktionen ist nicht sehr hoch. Diese wurde mit zwei verschiedenen Bignum-Libraries getestet. Wir mussten feststellen, dass die Performance massiv durch die Bignumber-Library mitbestimmt wird.

Zu diesem Zeitpunkt wurde uns klar, dass es keine effiziente EC-Implementation ohne eine effiziente Bignum-Library geben kann.

Als Nächstes haben wir also geprüft, was an der Bignumber-Library effizienter gestaltet werden könnte. Wir mussten feststellen, dass die Bignumber-Library in dieser Form noch viel Potenzial für Optimierungen bot. Das grösste Problem ist die ineffiziente Übersetzung des C-Codes zu Maschinensprache. Wir haben einige Teile der Bignumber-Library in Assembler geschrieben. Mit einem relativ geringem Einsatz war eine Verdopplung der Berechnungsgeschwindigkeit erreicht worden. Die Details dazu sowie weitere Probleme der Bignumber-Library werden im Kapitel 11 genauer behandelt.

9.4 Ergebnisse

Die mit ECC erreichten Zeiten lassen sich durchaus sehen. Vor dieser Diplomarbeit musste sich der Benutzer mit einer Wartezeit von 45 Sekunden abfinden (RSA mit einem 1024-Bit-Schlüssel).

Mit der effizienten ECC-Implementation und der optimierten Bignumber-Library beträgt die Wartezeit nur noch 7.0s für eine Sicherheit, die mit der von 1024-Bit-RSA vergleichbar ist (160-Bit EC).

Die Optimierung der Bignumber-Library ist aber auch der Bilanz von RSA zugute gekommen. Die Dechiffrierungszeit mit einem 1024Bit-RSA-Schlüssel dauert nun nur noch 21s.

Die heutigen Browser unterstützen viele symmetrische Kryptoalgorithmen. Für den sicheren Austausch der symmetrischen Schlüssel steht aber nur ein Verfahren zur Verfügung: RSA.

Die Vorteile von ECC können somit von keinem der heutigen Browser ausgenutzt werden. Mit dem S_CLIENT von OpenSSL (im Moment noch kein offizielles Release) ist es aber möglich eine SSL-Verbindung mit Hilfe von ECC zu öffnen. Es wäre aber zu vermuten, dass Mozilla als erster Browser EC-Kryptographie unterstützt.

Im Verlauf der Eigenimplementation wurde auch ein Blick in den Mozilla-Source-Code geworfen, um den Integrationsaufwand abzuschätzen. Eine Integration hätte aber den Umfang dieser Arbeit gesprengt, weshalb davon abzusehen war. Die eigene EC-Implementation wurde aber zumindest mit der Mozilla-Multi-Precision-Integer-Library, dem Pendant zur OpenSSL-Bignum-Library, getestet. Mit dem ernüchternden Resultat, dass diese Library für leistungsarme Prozessoren absolut ungeeignet ist. Im Vergleich zur PGP-Bignum-Library, die aus der Zeit stammt, als der PGP-Source-Code noch als Buch aus den USA exportiert werden musste, war die in C geschriebene Mozilla-MPI-Library um einen Faktor 10 langsamer. Für den Test mit der PGP-Library musste diese relativ mühsam aus dem PGP-Code extrahiert werden, dazu mussten auch die Memory-Management-Funktionen neu geschrieben werden. Dieser Aufwand wurde deshalb bewältigt, weil die PGP-Bignum-Library einige Assembler-Optimierungen für den verwendeten 80186-Prozessor beinhaltet.

Die Codegrösse von mSSL inkl. Diffie-Hellman und Elliptic Curve Kryptoalgorithmus beträgt jetzt 100 kByte (komprimiert). Bei Bedarf kann die Grösse (auf Kosten der Flexibilität) jedoch leicht reduziert werden.

Zum einen kann das DH-Verfahren weggelassen werden, zum anderen können Teile der ECC-Implementation weggelassen werden. Die ECC-Implementation ist sehr umfangreich. Es sind z.B. die Parameter von über 50 EC-Kurven gespeichert. Für dieses Embedded System wäre aber eine Gruppe von etwa 3 Kurven sinnvoll (alle über einem Primfeld): 160, 192 und 233-Bit. Falls die Felder der Charakteristik 2 definitiv nicht mehr benötigt werden, könnte man die entsprechenden Funktionen mit geringem Aufwand entfernen. Zudem würden in diesem Fall auch die zugehörigen Arithmetikfunktionen in der Bignum-Library überflüssig.

9.5 Infrastruktur

9.5.1 Hardware

- IPC@CHIP DK40

  verwendete Schnittstellen: Ethernet

  verwendete Protokolle: Telnet, FTP, HTTP

  wird mit 24V DC gespiesen

  Eth0-Einstellungen:

```
DHCP=0
ADDRESS=160.85.162.77
NETMASK=255.255.240.0
GATEWAY=160.85.160.1
```

  Sever-Hostname auf Client einstellen (wichtig für Zertifikat-Prüfung):

```
pygmy.strongsec.com
```

9.5.2 Software

- Borland C++ Compiler v. 3.1
- PKLite 2.0.1
- Windows98b-FTP-Client
- Windows98b-Telnet-Client
- SourceNavigator v.5.1.1
- OpenSSL snapshot vom 12.09.2002

  (inoffizielle unvollständige v. 0.9.8)
- Mozilla 1.0
- Netscape Navigator 4.7
- Opera 6.0

9.6 Projektplan

Aufgrund der komplexen Thematik war eine Aufwandabschätzung zu Beginn der Arbeit nur erschwert möglich. Deshalb mussten wir den Plan situativ an neue Erkenntnisse anpassen. Die jeweiligen Planänderungen sind im Kapitel Vorgehen begründet worden.

Soll-Plan

Aktivität / Woche	37	38	39	40	41	42	43
PC & Embedded-Systeme vorbereiten							
ECC-Theorie							
Einarbeiten in OpenSSL							
Einarbeiten in MSSL							
Realisation auf dem PC (Apache+OpenSSL)							
Portierung auf das Embedded-System							
Codeoptimierung (ggf. auf bestimmte EC)							
Test							
Dokumentieren							

Ist-Plan

Aktivität / Woche	37	38	39	40	41	42	43
PC & Embedded-Systeme vorbereiten							
ECC-Theorie							
Einarbeiten in OpenSSL							
Einarbeiten in MSSL							
StateMachine optimieren & Handshake erw.							
DH portieren							
ECC portieren (v. OpenSSL prerelease)							
Optimierte ECC-Verfahren implementieren							
Bignumber-Library optimieren							
Test							
Dokumentieren							

10 Betrachtung EC in Embedded Systems

Der technologische Fortschritt hat es möglich gemacht, dass schnelle, günstige und energiesparende Mikroprozessoren entwickelt wurden. Die Anzahl dieser Mikroprozessoren, auf die wir in unserem alltäglichen Leben angewiesen sind, ist in den letzten Jahren enorm gestiegen. Weiter bestehen Bestrebungen, dass die Vernetzung dieser Prozessoren stark vorangetrieben wird, wodurch eine sichere Kommunikation unumgänglich wird. In Einsatzfeldern wie Embedded Systems, bei denen nicht die Rechenleistung zur Verfügung steht, um die etablierten Verschlüsselungsverfahren einzusetzen, wird es problematisch, die verlangte Sicherheit zu gewährleisten.

Ohne spezielle Krypto-Co-Prozessoren ist es praktisch nicht möglich, unter Verwendung der etablierten Kryptoalgorithmen wie u.a. RSA, ElGamal und DH, die heutigen und zukünftigen Anforderungen an die Sicherheit zu erfüllen. Die Berechnungen für diese Systeme würden einen vertretbaren Zeitrahmen bei weitem überschreiten.

Unter Verwendung der EC-Kryptographie können zumindest die aktuellen Sicherheitsanforderungen erfüllt werden. Dies ist möglich als reine Software-Implementation, auch wenn dies voraussetzt, dass praktisch alle möglichen Optimierungen ausgeschöpft werden müssen. Natürlich würde eine spezielle Krypto-Hardware auch hier enorme Erleichterungen bieten.

Wir werden in diesem Abschnitt den Blickwinkel auf Embedded Systems richten und evaluieren, welche Techniken eingesetzt werden können, um die gewünschte Sicherheit mit EC-Kryptographie zu erreichen.

EC-Kryptosysteme benötigen zwei Arten von Arithmetik:

- Punktarithmetik auf elliptischen Kurven
- Arithmetik in endlichen Feldern

Der wichtigste Teil beim Entwurf eines EC-Kryptosystems ist die Wahl eines geeigneten endlichen Feldes für das jeweilige Zielsystem. Denn das unterliegende Feld bestimmt praktisch die Leistung eines EC-Kryptosystems im Alleingang. Eine Optimierungen der Punktarithmetik auf elliptischen Kurven kann zwar die Berechnung beschleunigen, was aber unbedeutend sein wird, wenn das falsche Feld gewählt wurde. Die folgenden Ausführungen beziehen sich also auf die Arithmetik in endlichen Feldern, die Optimierung der Punktarithmetik wird in Kapitel 11 beleuchtet. Auch die Wahl der elliptischen Kurve hat auf die Leistung des Kryptosystems einen vernachlässigbaren Einfluss. Einzig durch die Wahl des Kurvenparameters $a = p - 3$ können in der Punktverdopplung wenige Feldmultiplikationen durch Additionen ersetzt werden.

In der Praxis werden wie schon erwähnt zwei Felder am meisten eingesetzt:

- F_p, auch bekannt als Primfelder, p ist dabei eine ungerade Primzahl
- F_{2^m}, auch bekannt als binäre Felder oder Felder der Charakteristik 2

10.1 Primfelder F_p

Primfelder sind wahrscheinlich der offensichtlichste Ansatz für eine Implementation. Heute werden mindestens Felder der Grösse F_{160} verwendet, weshalb die Elemente ein Mehrfaches des Prozessor-Word-Size sind und aus diesem Grund schrittweise verarbeitet werden müssen. Das Problem dabei ist, dass während Berechnungen die Überträge zwischen zwei Prozessor-Words propagiert werden müssen. Zudem muss modulo p über mehrere Prozessor-Words berechnet werden, was ein besonderes Problem darstellt. Bis heute gab es in diesem Bereich aber grosse Anstrengungen und als Resultat daraus existieren verschiedene Ansätze, die sogenannte Multi-Precision-Arithmetik effizient zu implementieren. Die bekannteste Methode basiert dabei wie schon erwähnt auf der Montgomery-Reduktion. Dagegen gibt es zur Inversion keine effiziente Lösung, was voraussetzt, dass zur Punktarithmetik auf elliptischen Kurven projektive Koordinaten verwendet werden, um Inversionen möglichst zu vermeiden (siehe Kapitel 11).

Interessant an Primfeldern ist, dass sie die gleiche Arithmetik voraussetzen, die z.B. auch von RSA benötigt wird. Dies bringt gewisse Vorteile mit sich. Einer davon ist, dass ein schon existierender Krypto-Co-Prozessor für RSA auch für EC-Primfelder benutzt werden kann. Ist ein solcher Co-Prozessor nicht vorhanden, wird aber vorausgesetzt, dass das Zielsystem eine Arithmetikeinheit aufweist, die multiplizieren und dividieren kann. Bestes Beispiel dafür sind CISC-Prozessoren wie die 80x86 der Intel-Familie. Im Gegensatz dazu ist eine solche Arithmetik-Einheit z.B. auf Smart-Cards eher selten zu finden.

10.2 Binäre Felder F_{2^m}

Binäre Felder wurden seit 1960 extrem stark untersucht, ihre Verwendung erlebte mit der Codierungstheorie einen grossen Aufschwung. Die Arithmetik in binären Feldern hat mit derjenigen in Primfeldern nicht mehr viel gemeinsam, weshalb sich auch andere Anforderungen an die Zielsysteme ergeben. Binäre Felder sind sogenannte Extension Fields mit einem Subfield F_2, sie benötigen zwei Arten der modularen Reduktion. Einerseits wird mit dem irreduziblen Polynom das Exension Field reduziert (siehe Kapitel 5), andererseits wird das Subfield (die Koeffizienten in F_2) modulo 2 reduziert. Wir haben ebenfalls in Kapitel 5 gezeigt, dass sich eine Polynommultiplikation aus Additionen (XOR) und Schiebeoperationen zusammensetzen lässt.

Dadurch lässt sich verallgemeinern, dass sich binäre Felder hauptsächlich für Prozessoren ohne Multiplikations- und Divisionseinheit eignen. Bestes Beispiel hierfür sind RISC-Prozessoren oder Smart-Cards. In solchen Prozessoren sind F_{2^m}-Felder der einzig gangbare Weg zu einer effizienten Software-Implementation. Neben einer Software-Implementation ist natürlich auch eine Hardware-Implementation denkbar. Und gerade in diesem Bereich offenbaren binäre Felder ihre grossen Vorteile.

Durch die Art der Operationen, die keine Carry-Propagation voraussetzt, ist es möglich, die Arithmetik relativ einfach in Hardware zu realisieren. Eine Multiplikation könnte dadurch im Extremfall (mit einer grossen Anzahl logischer Verknüpfungen) in einem Zyklus erledigt werden, denn mangels Carry-Propagation können die Unterberechnungen parallel ablaufen. Dies wäre auch eine sehr gute Möglichkeit für Smart-Cards, denn ein so realisierter diskreter Co-Prozessor würde auf der Karte immer noch viel weniger Platz beanspruchen, als ein Krypto-Co-Prozessor für Primfelder.

10.3 Weitere Felder

Neben den bekannten und oft verwendeten Primfeldern und binären Feldern existiert noch eine Anzahl an weiteren Feldern, diese sind aber meist nur theoretisch interessant. Eine Ausnahme sind dabei aber Optimal Extension Fields (OEF). Diese wurden von Bailey und Paar an der Crypto '98 erstmals zur Verwendung vorgeschlagen. Sie sind also noch sehr jung und noch nicht Gegenstand von sicherheitsbezogenen Analysen. Trotzdem ist ihr Einsatz gerade auch für Embedded Systems sehr interessant. Die Optimal Extension Fields werden momentan ausschliesslich in EC-Kryptosystemen verwendet, da sie auch aus diesem Umfeld stammen. Die Idee dahinter stammt von den binären Feldern, dieser Ansatz wird aber verallgemeinert.

Ein OEF hat die Form $F_p{}^m$, wobei $p = 2^n \pm c$. Dabei gilt zwischen n und c die Beziehung: $\log_2(c) \leq \frac{n}{2}$

Die Idee ist es, die Multiplikationseinheit eines CISC-Prozessors voll auszunützen, deshalb wird die Primzahl p so gewählt, dass sie in die Prozessor-Word-Size passt. Zusätzlich wird m so gewählt, dass ein irreduzibles binomiales Polynom $P(x) = x^m - w$ existiert, wobei w im Subfield F_p sein muss. Durch diese Wahlen wird erreicht, dass sowohl eine schnelle Subfield-Reduktion als auch eine schnelle Extension-Field-Reduktion möglich ist.

Mit einem CISC der Intel-80x86-Familie kann die Subfield-Reduktion z.B. direkt durch die Divisionseinheit durchgeführt werden. Andernfalls, wenn die Divisionseinheit keine Rest-Bildung unterstützt, kann die Beziehung $2^n \equiv c \; mod \; 2^n - c$ ausgenützt werden. Dabei wird dann eine $2n$-Bit Zahl $x = x_1 2^n + x_0$ reduziert:

$$x \equiv x_1 c + x_0 \; mod \; 2^n - c$$

Dies benötigt eine Multiplikation mit c und eine Addition, aber keine Division oder gar Inversion.

Der Nachteil eines OEF ist aber, dass für jede Prozessor-Word-Size ein spezielles Feld definiert wird, wodurch die Interoperabilität zwischen verschiedenen Systemen erschwert bis verunmöglicht wird. In einem homogenen Umfeld, wie z.B. Smart-Cards, wird dieser Umstand aber mehr als wettgemacht.

10.4 Zusammenfassung

Wir können nun sagen, dass F_p-Felder vorzuziehen sind, wenn das Zielsystem eine Arithmetikeinheit hat, die eine effiziente Multiplikation und Division unterstützt. F_{2^m}-Felder eignen sich hervorragend für Hardware-Implementationen und sind eine passable Lösung für Prozessoren ohne Multiplikationseinheit. OEF schliessen Zielsysteme ohne Multiplikationseinheit sowieso aus, eignen sich wegen ihrer spezialisierten Ausrichtung auf die Prozessor-Word-Size nur in homogenen Umgebungen, bieten in solch einem Einsatzfeld aber grosse Leistungsvorteile.

11 Effiziente Implementation

11.1 Algorithmen

11.1.1 Feldoperationen in F_p

Damit in F_p mit sehr grossen Zahlen, die kryptographischen Anforderungen gerecht werden, gerechnet werden kann, wird eine so genannte Multiprecision-Integer-Arithmetik benötigt. Eine sehr gute Einführung in dieses Gebiet ist zu finden in [HAC96] (Chap. 14). Die Problematik liegt darin, dass die grossen Zahlen, die Word-Size des Prozessors bei weitem überschreiten, die Operation müssen also aus kleineren Schritten zusammengesetzt werden. Addition und Subtraktion sind die am einfachsten zu realisierenden Funktionen, dabei muss jeweils einfach das Carry- resp. Borrow-Bit zur nächsten Stufe propagiert werden. Die Modulo-Reduktion dazu ist ebenfalls relativ einfach zu bewerkstelligen, denn wenn das Resultat einer Addition grösser als der Modulo ist, ist einfach vom Resultat der Modulo zu subtrahieren.

Komplexer ist die Multiplikation, denn wenn zwei n-Bit lange Zahlen multipliziert werden, entsteht eine $2n$-Bit lange Zahl, es ist wohl offensichtlich, dass eine Modulo-Operation auf solch eine grosse Zahl extrem aufwändig ist. Dank dem Algorithmus nach Montgomery, der darauf aufbaut, dass eine Modulo-Operation über eine Potenz von 2 viel einfacher zu berechnen ist als über den eigentlichen Modulo, reduziert sich der Aufwand aber drastisch. Eine ausführliche Beschreibung des Algorithmus und der mathematischen Theorie ist in [HAC96] (Chap. 14) zu finden.

11.1.2 Punktaddition/-verdopplung

Die Algorithmen zur Punktaddition und -verdopplung lassen sich relativ direkt aus den mathematischen Zusammenhängen (siehe Kapitel 5) herleiten. Die publizierten Algorithmen unterscheiden sich nur marginal im Bereich der Anzahl der Feldoperationen und der benötigten temporären Variablen. Der Hauptunterschied der vorgeschlagenen Algorithmen liegt in den verwendeten Koordinaten. Wird eine affine Repräsentation der Koordinaten verwendet, müssen multiplikative Inverse eines Elements berechnet werden (für die Division in F_p), um eine Punktaddition durchzuführen. Diese Inversion kann im Vergleich zu den anderen Feldoperationen (Multiplikation, Addition, Subtraktion) sehr zeitintensiv sein. Deshalb sind Inversionen möglichst zu vermeiden, wozu ein Punkt $P = (x, y)$ in eine gewichtete, projektive Koordinatenrepräsentation $P = (X, Y, Z)$, die in der Literatur auch Jacobi-Repräsentation (nach Karl Jacobi, deutscher Mathematiker, 1804-1851) genannt wird, übergeführt werden kann:

$$X = x\,;\quad Y = y\,;\quad Z = 1$$

Werden nun die Punktoperationen in dieser projektiven Repräsentation durchgeführt, wird keine Inversion benötigt. Einzig wieder beim Übertritt in affine Koordinaten wird eine Inversion eingesetzt:

$$x = \frac{X}{Z^2}\,;\quad y = \frac{Y}{Z^3}$$

Ebenfalls kann die Weierstrass-Gleichung einer elliptischen Kurve in Jacobi-Repräsentation notiert werden:

$$Y^2 = X^3 + aXZ^4 + bZ^6$$

Gleichung 11.1: Weierstrass-Gleichung einer elliptischen Kurve in Jacobi-Repräsentation

In dieser Form kann auch die Punktaddition und -verdopplung für Primfelder definiert werden:

Punktverdopplung:

$$P_2 = 2P_1 = \left\{ \begin{array}{l} X_2 = M^2 - 2S \\ Y_2 = M(S - X_2) - T \\ Z_2 = 2Y_1 Z_1 \end{array} \right\}$$

$$\textit{wobei} \quad M = 3X_1^2 + aZ_1^2, \quad S = 4X_1Y_1^2, \quad T = 8Y_1^4.$$

Punktaddition:

$$P_2 = P_0 + P_1 = \left\{ \begin{array}{l} X_2 = R^2 - TW \\ 2Y_2 = VR - MW^3 \\ Z_2 = Z_0 Z_1 W \end{array} \right\}$$

$$\textit{wobei} \quad W = X_0Z_1^2 - X_1Z_0^2, \quad R = Y_0Z_1^3 - Y_1Z_0^3,$$

$$T = X_0Z_1^2 + X_1Z_0^2, \quad M = Y_0Z_1^3 + Y_1Z_0^3, \quad V = TW^2 - 2X_2.$$

Algorithmus 11.1: Punktaddition und -verdopplung in Jacobi-Repräsentation

Basierend auf Algorithmus 11.1 kann eine Verdopplung mit fünf temporären Variablen und eine Addition mit sieben temporären Variablen realisiert werden. Diese Zahl kann natürlich noch reduziert werden, z.B. in [P1363] wird eine Implementation mit 4 temporären Variablen zur Addition und drei zur Verdopplung beschrieben. Unter Verwendung der Implementation aus [Hase99] können die temporären Variablen in beiden Fällen gar auf zwei reduziert werden. Eine Inversion wird nur noch zur Division durch 2 benötigt, dies kann in F_p aber relativ einfach berechnet werden:

if k even

$$\frac{k}{2} mod\, p = \frac{k}{2}$$

else

$$2^{-1} = \frac{p+1}{2}$$

$$\frac{k}{2} mod\, p = k * 2^{-1} mod\, p$$

Algorithmus 11.2: Division durch 2 in F_p

Die normale Division durch 2 kann effizient durch eine Schiebeoperation implementiert werden. Dies basiert auf der Tatsache, dass bei einer Division durch 2 in einem Feld die geraden Zahlen in die untere Hälfte (von 0 bis *(p-1)/2*) fallen und die ungeraden Zahlen in den oberen Bereich (von *(p+1)/2* bis *p*) geschoben werden. Natürlich kann 2^{-1} für das Feld auch noch vorberechnet werden. Diese Regel gilt aber natürlich nur in Primfeldern.

11.1.3 Skalarmultiplikation

Bei der Skalarmultiplikation wird angestrebt, mit möglichst wenig Punktadditionen zum Ziel zu kommen. Gegeben ist eine Zahl e und ein Punkt P, gesucht wird das Resultat von eP. Da die Skalarmultiplikation auf elliptischen Kurven ein Spezialfall des Exponentionsproblems in abelschen Gruppen ist, können Techniken für das allgemeine Problem auch auf die Skalarmultiplikation übertragen werden. In diesem Fall ist das Problem verbunden mit Additionsketten. Eine Additionskette ist eine Sequenz von Zahlen beginnend bei 1 und endend bei e, wobei jeder Zwischenschritt die Summe von zwei vorangehenden Zahlen in der Sequenz ist.

Beispiel: e = 55

1	2	3	6	12	13	26	27	54	55
1	2	3	6	12	13	26	52	55	
1	2	4	5	10	20	40	50	55	
1	2	3	5	10	11	22	44	55	

Tabelle 11.1: Mögliche Wege zur Berechung von eP

Eine Additionskette stellt einen Algorithmus zum Berechnen von eP dar. Die Länge der Kette ist die Anzahl Operationen, die für diese Berechnung benötigt werden. Das Beispiel mit $e=55$ könnte also in 9 Schritten berechnet werden. Das Problem ist nun, die kürzest mögliche Additionskette zu finden. Das Finden der optimalen Additionskette ist aber in der Praxis untauglich, mit den folgend vorgestellten Verfahren können aber mit vertretbarem Aufwand schon relativ kurze Additionsketten gefunden werden.

Binäre Methode

Die älteste Methode zur Bildung von Additionsketten ist die binäre Methode, sie ist schon seit über 2000 Jahren bekannt. Die Idee ist es, die binäre Repräsentation von e bitweise abzutasten und dabei jeweils eine Punktverdopplung und abhängig vom Bit $e[i]$ eine Punktaddition durchzuführen.

$$
\begin{array}{l}
INPUT: \quad Ein\ Punkt\ P\ und\ eine\ l\text{-}Bit\ lange\ Zahl\ e \\
OUTPUT: \quad Q = eP \\
Q \leftarrow O \\
for\ j = l-1\ to\ 0 \\
\quad Q \leftarrow Q + Q \\
\quad if\ e[i] = 1 \\
\quad\quad Q \leftarrow Q + P \\
return\ Q
\end{array}
$$

Algorithmus 11.3: Binäre Methode

Am Beispiel von $e=55=110111_2$ kann die entstehende Additionskette gut verfolgt werden:

i	*e[i]*	*Verdopplung*	*Addition*
5	1	O + O = O	O + P = P
4	1	P + P = 2P	2P + P = 3P
3	0	3P + 3P = 6P	
2	1	6P + 6P = 12P	12P + P = 13P
1	1	13P + 13P = 26P	26P + P = 27P
0	1	27P + 27P = 54P	54P + P = 55P

Die binäre Methode benötigt also $l-1$ Punktverdopplungen und $W-1$ Punktadditionen, wobei l die Länge und W das Gewicht (Anzahl der Einsen) der binären Repräsentation von e ist. Für W kann $l/2$ angenommen werden, da e eine Zufallszahl nach Laplace ist. Die hier vorgestellte binäre Methode ist schon relativ effizient, die Anzahl Punktverdopplungen sind auf dem Minimum, aber die Punktadditionen können durch die folgenden heuristischen Methoden reduziert werden.

M-äre Methode

Diese Methode benutzt die m-äre Repräsentation von e, wobei $m=2^r$. Die binäre Methode ist eigentlich nur ein Spezialfall mit $r=1$. Aus Algorithmus 11.4 ist ersichtlich, dass in einem ersten Schritt eine Vorberechnung von m Punkten getätigt wird, diese werden später in der Hauptschleife verwendet.

$$
\begin{aligned}
&INPUT: \quad Ein\ Punkt\ P\ und\ eine\ l\text{-}Bit\ lange\ Zahl\ e\\
&OUTPUT: \ Q = eP\\
&Vorberechnung:\\
&P[1] \leftarrow P\\
&for\ i = 2\ to\ m-1\\
&\quad P[i] \leftarrow P[i-1] + P \qquad \text{(d.h. P[i] = iP)}\\
&Q \leftarrow O\\
&Hauptschleife:\\
&for\ j = d-1\ to\ 0\\
&\quad Q \leftarrow mQ \qquad \text{(benötigt r Verdopplungen)}\\
&\quad Q \leftarrow Q + P[e[j]]\\
&return\ Q
\end{aligned}
$$

Algorithmus 11.4: m-äre Methode

Statt wie bei der binären Methode die Zahl e bitweise abzutasten, wird jetzt ein Fenster $e[j]$ der Breite r verwendet. Dadurch kann e in $d=l/r$ Schritten durchlaufen werden. Die beim Schritt j im Fenster $e[j]$ enthaltene m-äre Zahl kann nun als Index benutzt werden, um die vorberechneten Punkte zu erhalten.

Am Beispiel von $e=55=110111_2$ und einer Fensterbreite von $r=2$ kann diese Methode gut mitverfolgt werden.

j	*e[j]*	*Verdopplung*	*vorberechneter Punkt*	*Addition*
2	11	4O = O	$P[11_2] = 3P$	O + 3P = 3P
1	01	4(3P) = 12 P	$P[01_2] = P$	12P + P = 13P
0	11	4(13P) = 52P	$P[11_2] = 3P$	52P + 3P = 55P

Die Anzahl der Punktadditionen kann ausgehend von l-1 um maximal r-1 reduziert werden. Intuitiv ist klar, dass bei einem zu grossem Fenster in der Vorberechnungsphase zu viele Punkte berechnet und bei einem zu kleinem Fenster in der Hauptschleife zu viele Additionen durchgeführt werden.

Durch ein Modifizieren der Hauptschleife kann in der Vorberechnungsphase auf die Berechnung der geraden Vielfachen verzichtet werden, was die Anzahl der Additionen weiter reduziert.

INPUT : *Ein Punkt P und eine l - Bit lange Zahl e*
OUTPUT : $Q = eP$
Vorberechnung :
$P[1] \leftarrow P, P[2] \leftarrow 2P$
for $i = 1$ *to* $(m-2)/2$
  $P[2i+1] \leftarrow P[2i-1] + P[2]$
$Q \leftarrow O$
Hauptschleife :
for $j = d-1$ *to* 0
  if $e[j] \neq 0$
    Wähle s, h so, dass $e[j] = 2^s h$, h ungerade
    $Q \leftarrow 2^{r-s} Q$
    $Q \leftarrow Q + P[h]$
  else $s \leftarrow r$
  $Q = 2^s Q$
return Q

Algorithmus 11.5: Modifizierte m-äre Methode

Sliding Window Methode

Die m-äre Methode kann als Spezialfall der Window-Methode betrachtet werden, bei dem die Bits des Multiplikators in Blöcken der Grösse *r* verarbeitet werden. Dies führt nun verallgemeinert zur Sliding Window Methode, bei der die Fenstergrösse dynamisch angepasst wird und die Fenster nicht nahtlos aneinandergereiht sein müssen.

$$
\begin{array}{l}
INPUT: \quad \text{Ein Punkt } P \text{ und eine } l\text{-Bit lange Zahl } e \\
OUTPUT: \quad Q = eP \\
Vorberechnung: \\
P[1] \leftarrow P,\ P[2] \leftarrow 2P \\
for\ i = 1\ to\ 2^{r-1} - 1 \\
\quad P[2i+1] \leftarrow P[2i-1] + P[2] \\
j \leftarrow l - 1 \\
Q \leftarrow O \\
Hauptschleife: \\
while\ j \geq 0 \\
\quad if\ e[j] = 0 \\
\qquad Q \leftarrow 2Q \\
\qquad j \leftarrow j - 1 \\
\quad else \\
\qquad \text{Wähle } t \text{ möglichst klein, dass } e[t] = 1 \text{ und } j - 1 + 1 \leq r \\
\qquad h \leftarrow (e[j], e[j-1], \ldots, e[t])_2 \\
\qquad Q \leftarrow 2^{j-t+1} Q + P[h] \\
\qquad j \leftarrow t - 1 \\
return\ Q
\end{array}
$$

Algorithmus 11.6: Sliding Window Methode

Folgen von Nullen fallen also nicht in ein Fenster, bei jedem 0-Bit wird der Punkt verdoppelt. Ein Fenster ist maximal *r*-Bit breit, es muss durch ein 1-Bit abgeschlossen werden, damit *h* ungerade wird. Denn dadurch werden ja in der Vorberechnungsphase die geraden Vielfachen gespart. Das Verwenden der Sliding Window Methode hat den gleichen Effekt wie die Verwendung einer Methode mit fixen Fenstergrösse, die um ein Bit länger ist, aber ohne die Kosten der Vorberechnung zu erhöhen.

Als Beispiel soll die Zahl *e=262359474289536631831191* betrachtet werden. Die binäre Repräsentation von *e* ist:

1011000111001000000111010010100111010100000010111110000011111001100101010111

Mit der Wahl von *r=4* ergäben sich folgende Fenster:

$$
\underset{11}{\underline{1011}}000\underset{7}{\underline{111}}00\underset{1}{\underline{1}}000000\underset{7}{\underline{111}}0\underset{9}{\underline{1001}}0\underset{9}{\underline{1001}}\ \underset{13}{\underline{1101}}0\underset{1}{\underline{1}}000000\underset{11}{\underline{1011}}\ \underset{3}{\underline{11}}00000\underset{15}{\underline{1111}}\ \underset{9}{\underline{1001}}\ \underset{9}{\underline{1001}}0\underset{5}{\underline{101}}0\underset{7}{\underline{111}}
$$

Vorzeichenbehaftete Methoden

Wie schon in Kapitel 5 erwähnt, ist bei elliptischen Kurven der Aufwand für eine Subtraktion praktisch identisch wie für eine Addition. Das führt dazu, dass in Skalarmultiplikationen Additions-Subtraktions-Ketten verwendet werden können, was die Anzahl der Kurvenoperationen reduziert. Um die vorhergehenden Methoden anpassen zu können, wird eine neue Repräsentation benötigt, diese wird Signed-Digit-Repräsentation genannt und umfasst im Vergleich zur binären Form nun auch -1. Zum Beispiel kann die Zahl 3 durch 011_2 oder $10\bar{1}_2$ dargestellt werden, wobei $\bar{1}$ für -1 steht. Es gibt dabei jeweils 3^{l+1} verschiedene Darstellungen für eine Zahl. Diese Redundanz kann jetzt dazu genutzt werden, die Skalarmultiplikation effizienter zu gestalten.

Eine spezielle Form der SD-Repräsentation ist die Non-Adjacent-Form (NAF), diese sagt aus, dass $e[i]e[i+1]=0$ an jeder Stelle $i\geq 0$ gilt. Also dass keine angrenzenden Nicht-Null-Ziffern in der SD-Repräsentation auftreten. Der Vorteil der NAF ist darin zu finden, dass sie weniger Nicht-Null-Ziffern als die binäre Darstellung aufweist, was die Anzahl Additionen und Subtraktionen reduziert.

$$
\begin{aligned}
&INPUT: \quad Eine\ l\text{-}Bit\ lange\ Zahl\ e\\
&OUTPUT: \quad SD-Repräsentation\ von\ e\ in\ NAF\\
&c[0] \leftarrow 0\\
&for\ j = 0\ to\ l\\
&\quad c[j+1] \leftarrow (k[j] + k[j+1] + c[j])/2\\
&\quad s[j] \leftarrow k[j] + c[j] - 2c[j+1]\\
&return\,(s[l], s[l-1], \ldots, s[0])_{SD}
\end{aligned}
$$

Algorithmus 11.7: Wandlung in NAF

Die Adaption der binären Methode zur Skalarmultiplikation ist schon fast selbstverständlich. Statt einer Addition wird einfach eine Subtraktion ausgeführt, wenn eine negative Ziffer angetroffen wird.

$$
\begin{aligned}
&INPUT: \quad Ein\ Punkt\ P,\ -P\ und\ eine\ l\text{-}Bit\ lange\ Zahl\ e\\
&OUTPUT: \quad Q = eP\\
&Generieren\ einer\ SD-Repräsentation\ f\ von\ e\\
&for\ i = l-1\ to\ 0\\
&\quad Q \leftarrow Q + Q\\
&\quad if\ f[i] = 1\\
&\quad\quad Q \leftarrow Q + P\\
&\quad if\ f[i] = -1\\
&\quad\quad Q \leftarrow Q + (-P)\\
&return\ Q
\end{aligned}
$$

Algorithmus 11.8: Binäre Methode mit SD-Repräsentation

Natürlich können auch die m-äre und Sliding-Window-Methoden für die NAF angepasst werden, hier wird aber nur die binäre Methode gezeigt, da sie auch im Standard IEEE P1363 [P1363] und ANSI X9.63 [X9-63] vorgeschlagen werden. Für die anderen Methoden wird auf die Literatur [Blake99] verwiesen.

Multiplizieren von fixen Punkten

In einigen Anwendungen (z.B. im Diffie-Hellman Key-Exchange) werden verschiedene Vielfache eines gleichbleibenden Punktes benötigt. In solch einem Fall kann es sinnvoll sein, verschiedene Vielfache vorzuberechnen und in einer Tabelle abzulegen. Bei den m-ären und Sliding-Window-Methoden könnte dadurch die Fenstergrösse sehr gross werden, da die Kosten für die Vorberechnungsstufe nicht mehr bei jeder Skalarmultiplikation anfallen. Das Verfahren nach Brickell, Gordon, McCurly und Wilson (BGMW) erlaubt es zudem, die Punktverdopplungen völlig zu vermeiden. Das einfachste Beispiel dazu ist die Verwendung der binären Methode und der Vorberechnung der Punkte 2^kP für *k=1, 2, ..., l-1*. Das Beispiel mit *e=55* könnte dadurch mit fünf Additionen berechnet werden (ohne Vorberechnung werden bei der binären Methode für dieses Beispiel zusätzlich noch fünf Verdopplungen benötigt):

i	*e[i]*	*Addition*
5	1	O + 32P = 32P
4	1	32P + 16P = 48P
3	0	
2	1	48P + 4P = 52P
1	1	52P + 2P = 54P
0	1	54P + P = 55P

Weitere geeignete Verfahren zur Vorberechnung für die anderen Methoden sind beschrieben in [Gor98] und [Möll01].

11.2 C versus Assembler

Wie aus der letzten Diplomarbeit (DA Sna 00/3) bekannt ist, dauerte die Berechnung mit einem 1024-Bit RSA-Schlüssel fast eine Minute, während bei einem 2048-Bit RSA-Schlüssel sechs Minuten notwendig waren.

Die Berechnungszeit mit einer 192-Bit EC-Curve über einem Prim-Feld (äquivalent zu 2028-Bit RSA) dauerte immerhin 16.6s. Diese Zeit war trotzdtem enttäuschend, wenn man Berichte über Implementationen liest, wo 1024-Bit-RSA in 10s und 160-Bit-EC in 0.5s auf einem 20MHz-Rechner gelöst werden soll.

Da für die EC-Operationen die emfohlenen effizienten Verfahren verwendet wurden, fingen wir an nach anderen Ursachen zu suchen.

Mit einem Decompiler wurden Teile des generierten Codes analysiert, was zu einem vernichtenden Ergebnis über die Fähigkeiten des Compilers führte. Es war schon fast skandalös wie ineffizient der generierte Code war.

Als ein Teil der OpenSSL-Big-Number-Library in Assembler umgeschrieben und noch einige wenige C-Code-Optimierungen vorgenommen wurden, reduzierte sich die Berechnungszeit stark:

1024-Bit RSA dauerte 21.3s und 192-Bit EC dauerte 9.3s

Es folgt eine Übersicht der Berechnungszeiten mit verschiedenen Libraries:

Algorithmus	*OpenSSL Bignum-Library ohne Optimierungen*	*Bignum-Library mit Assembler-Optimierungen*	*PGP-Bignum-Library (Assembler-Optimierungen)*
RSA1024	47s	21s	
RSA2048	355s	140s	
EC160 (prime)	13.4	7.0s	6.8s
EC192 (prime)	16.6s	9.3s	8.7s
EC193 (2m)	19s	14.4s	

Tabelle 11.2Vergleich der Berechnungszeiten

Die Zeit für die EC-Berechnung ist jedoch immer noch weit von in verschiedenen Publikationen veröffentlichten Zahlen entfernt.

Hauptursachen dafür sind:

- Ein wichtiger Grund für diese Verzögerung wird von uns weiterhin in der ineffizienten Übersetzung des C-Codes vermutet (nur die Bignum-Library in Assembler schreiben würde reichen, EC kann in C gelassen werden).
- Der Hauptgrund liegt jedoch in der universellen Auslegung der Bignum-Library. Auf bestimmte Felder spezialisierte Lösungen sind schätzungsweise um Faktor 5 leistungsfähiger.
  - Es werden zu viele temporäre Variablen erstellt. Diese könnte man teilweise einsparen.
  - Die Bignums müssen manchmal expandiert werden. Dies ist sehr zeitaufwändig, weil die Bignum dabei kopiert wird. Man könnte für die Bignums eine feste Grösse allozieren, die an eine bestimmte Feldgrösse angepasst ist.
  - Fehlerprüfungen werden oft mehrfach gemacht (ob genügend Platz reserviert, ob die Variable wirklich initialisiert ist). Bei Änderungen ist es sehr nützlich, das jedes Modul prüft, ob die ihm übergebenen Werte korrekt sind. Es verbraucht jedoch unnötig Rechenzeit.
- Overhead durch Calls (mit der ganzen Stack-Benutzung) und „decision code“. Dieser fällt bei EC 192 anteilmässig viel stärker ins Gewicht als dies bei RSA 1024 oder gar 2048 der Fall ist, weswegen die Assembler-Kodierung der Bignum-Grundfunktionen nicht im gleichen Masse die EC-Zeiten verbesserte.
- Obwohl die EC-Punkt-Addition komplexer ist, ist das kein Argument bei den Effizienzüberlegungen. Der Overhead ist ganz klar wegen der kürzeren Zahlen stärker spürbar.

Es gibt zwar viele Whitepapers über EC-Implementationen. Alle schreiben bereitwillig über die optimierten Berechnungs-Algorithmen auf der EC-Ebene. Keiner will jedoch die Details über seine Implementation auf der Prozessorebene preisgeben. Das einzige was man noch erfährt, ist, dass diese Implementationen in Assembler geschrieben wurden. Effiziente Lösungsansätze kann man daraus nicht ableiten.

11.3 Schätzung der Berechnungszeit in Assembler

Bei der Berechnung fallen folgende Operationen ins Gewicht (der „Verwaltungsoverhead“ wird hier nicht berücksichtigt):

- Multiplikationen
- Modulo-Operationen
- Additionen
- Bignums kopieren

Die Multiplikation eines 192-Bit-Wertes ist auf einem 80186er etwa 10x aufwendiger als die Addition. Somit reicht die Betrachtung der Multiplikation, um einen guten Richtwert für die minimale Berechnungszeit abzuschätzen. Der Rechenaufwand für eine Multiplikation steigt quadratisch mit der Schlüssellänge an:

$$t_{\text{Multiplikation}} = \left(\frac{\text{Anzahl Bits}}{\text{Multiplikationsbreite}}\right)^2 * t_{\text{Word-Mult}}$$

Weil auch die Anzahl Multiplikationen entsprechend der Schlüssellänge ansteigt, verhält sich die Gesamtausführungszeit in dritter Potenz zur Schlüssellänge:

$$t_{\text{Ges}} = \frac{\text{Anzahl Bits}^3}{\text{Multiplikationsbreite}^2} * t_{\text{Word-Mult}}$$

Ein Vorteil des 80186er ist, dass er word-weise und nicht nur byte-weise multiplizieren kann. Dabei wird der entstehende 32-Bit-Wert in zwei Register aufgeteilt gespeichert. Das Low-Word im AX, das High Word im DX. Eine Word-Multiplikation kann wie folgt dargestellt werden:

c:r[1..n] = w * a[1..n] + c + r[1..n]

- c carry-Byte
- a die erste Zahl
- w ein Word aus der zweiten Zahl
- r ist das Ergebnis
- c:r Aufteilung in das High Word (c) und Low Word (r)

In Assember geschrieben:

mov ax,es:[si+bx]	9
mul w ;// a * w	42
add ax,c ;// + c altes Carry	10
adc dx,0	4
add ax,ds:[di+bx] ;// + r	10
adc dx,0	4
mov ds:[di+bx],ax ;//r <= Ergebnis	12
mov c,dx	12
	0
inc bx	3
inc bx	3
dec cx ;//if (--n <= 0) break;	3
jg ErsteZeile ;// jump on greater	13
Gesamt	125

Für die Berechnung mit einem 1024-Bit RSA-Schlüssel sind 1024 Bignum-Verdopplungen nötig:

$$\frac{1024^3}{16^2} * 125 * \left(\frac{1}{20000000}\right) = 26s$$

(bei 20MHz Taktrate, 125 Taktzyklen pro Word-Multiplikation)

Die Stackbedienung beim Funktionsaufruf sowie die Initialisierung wirken sich nur zu weniger als 2% auf die obere Bilanz aus und werden deswegen vernachlässigt. Die Aufaddition der Werte sowie die Addition wurden genauso vernachlässigt.

Das Fetchen ist ebenfalls unberücksichtigt: Z.B. können unter bestimmten Umständen bei der Operation „adc dx,0“ mit der Ausführungszeit von 4 Taktzyklen zusätzliche Verzögerungen durch das Fetchen der 4 Bytes, was 8 Taktzyklen entspricht, entstehen.

Die Berechnung über eine EC-Kurve ist etwas komplizierter. Pro Punktverdopplung werden ca. 11 Bignum-Multiplikationen benötigt. Die Berechnung mit einem 192-Bit EC-Schlüssel:

$$\frac{192^3}{16^2} * 125 * 11 * (\frac{1}{20000000}) = 2s$$

Initialisierung wirkt sich hier bereits zu 5% auf die Bilanz aus. Die Zahl der Punktverdopplungen kann durch spezielle Verfahren leicht reduziert werden.

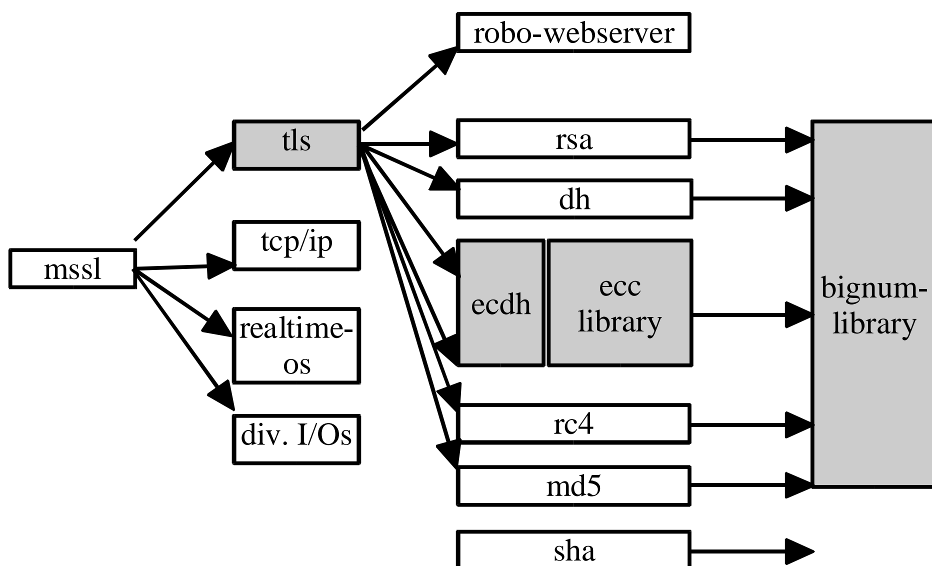
Die Dauer der Word-Multiplikation ist ausschlaggebend für die Berechnungsdauer.

Gemessene Werte für die Dauer der Word-Multiplikation:
- Prime-Feld: 200 Taktzyklen
- 2^m-Feld: 1400 Taktzyklen (konnte in Assembler auf 500 reduziert werden)

12 Softwarebeschreibung

12.1 Modul-Zusammenhänge

Die Umgebung auf dem Embedded-System ist relativ komplex. Im folgenden Bild werden die grundlegenden Zusammenhänge aufgezeigt. Die grau hinterlegten Bereiche zeigen die Module, an denen gearbeitet wurde.



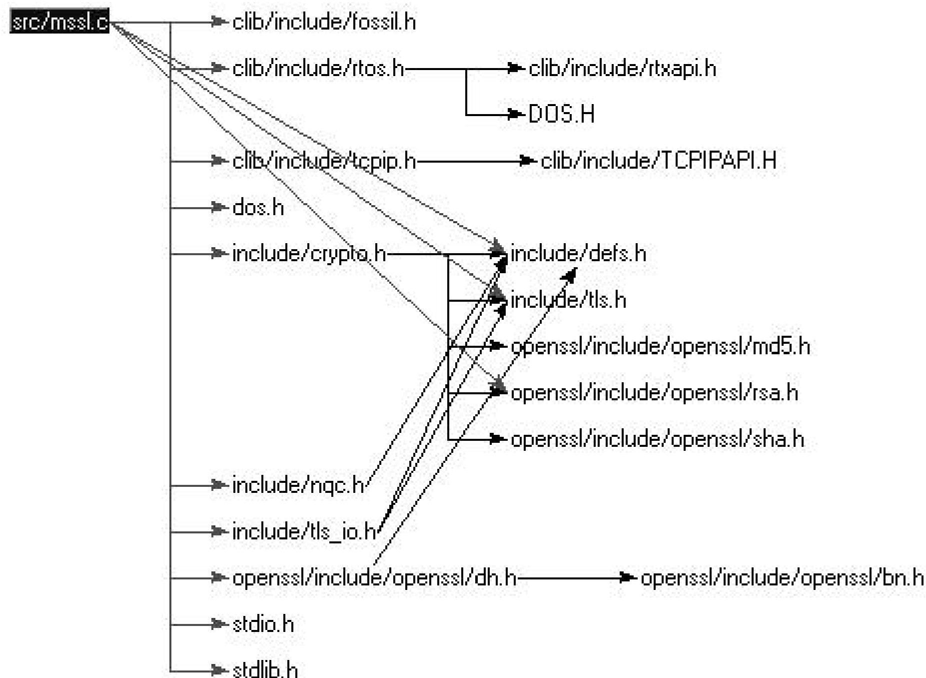
Im Modul TLS wurde die State-Machine für den SSL-Handshake neu geschrieben. Die EC-Funktionalität musste ins TLS integriert werden. Einige Pakete mussten erweitern bzw. neu erstellt werden, um die ECDH-Parameter aufnehmen zu können.

Von der EC-Library gibt es zwei Varianten. Zuerst haben wir die EC-Implementation des OpenSSL portiert. Als wir die verschiedenen EC-Verfahren sowie die OpenSSL-EC-Libary (neuerdings dabei) analysiert haben, stellten wir fest, dass diese Library bereits einen der effizientesten EC-Algorithmen verwendet (gilt gleichermassen für Embedded Systems). Der Source-Code ist zwar etwas aufgeblasen, aber er erfüllt die gewünschten Voraussetzungen. Wir haben ausserdem eine eigene Implementation erstellt, die auf einem anderen Algorithmus basiert. Weil diese nur etwa 5% schneller als die OpenSSL-EC-Library war, orientierten wir uns am OpenSSL-Standard.

In der Bignum-Library von OpenSSL wurden einige Teile, die der Compiler ineffizient übersetzt hat, in Assembler geschrieben. Einige Passagen mussten nur anders formuliert werden, damit der Compiler diese einigermassen effizient übersetzen kann.

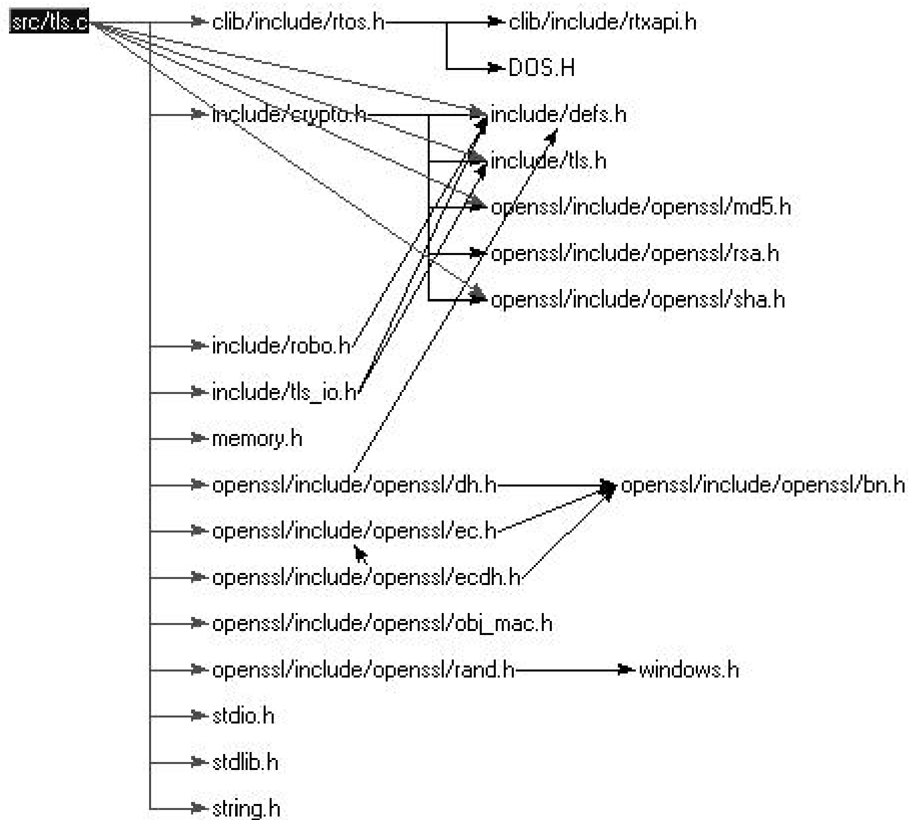
Während bis jetzt nur ein grober Überblick der Softwareumgebung angeboten wurde, werden im Folgenden die Zusammenhänge rund um einige Module etwas detaillierter betrachtet.

mssl.c ist das Hauptprogramm mit der Funktion main(). Sie benutzt hauptsächlich das RTOS-Modul sowie das TLS-Modul. Die Zusammenhänge rund um „mssl.c“ :



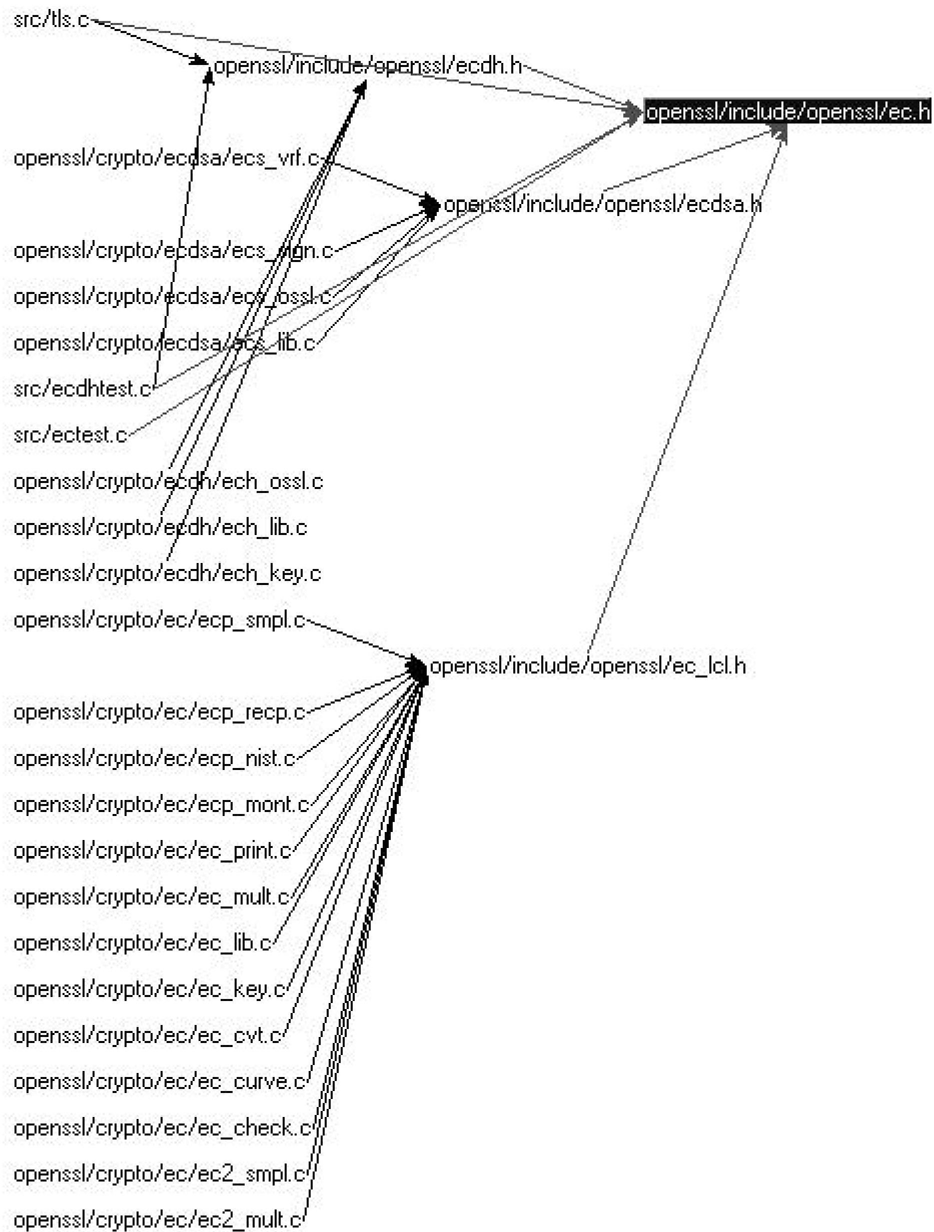
Im TLS laufen alle Fäden zusammen, hier wird der SSL-Handshake abgewickelt und die benötigten Kryptofunktionen aufgerufen.

Die Zusammenhänge rund um „tls.c“ :



Über „ec.h“ werden die EC-Grundfunktionen und Hilfsfunktionen verfügbar gemacht. Die EC-Funktionen werden von ECDH, einem Public-Key-Verschlüsselungs-Verfahren, sowie dem ECDSA, einem Unterschriftenverfahren, benötigt. Das ECDH wird zum Schluss noch von TLS verwendet. In der „ec_local.h“ werden alle internen Funktionen verwaltet. Die öffentlichen EC-Funktionen werden direkt in „ec.h“ deklariert. Es werden öfters Common-Header-Files gebraucht.

Die Zusammenhänge rund um TLS, ECDH sowie die ECC-Library:



12.2 Zustandsmaschine für das SSL-Handshake.

Die State-Machine für den TLS-Handshake in Pseudocode:

```
while (no errors){
   read_next_PACKET,
   read_next_RECORD,
   read_Protocol_Number,
   if proto=TLS  => parse_tls_header (majorminor-Nr,len,read_chunk to RECORD,
                                      decrypt* ,handshake, len, read_chunk to MSG)

   primitive ALERT-Behandlung (für alle Zustände gleich,deswegen vor die Statemachine genommen)

   switch (tls_state)
   {
    TLS_CLIENT_HELLO
         proto=TLS_HANDSHAKE & msg_type=HELLO
                               => client_hello_msg verarbeiten

              proto=SSLV2           => parse_SSLV2_header (abort wenn kein v2_hello),
                                       v2_client_hello_msg verarbeiten

              gemeinsamer Teil:
              proto=TLS ||SSL_V2 & msg_type= HELLO
                                    => server_hello,
                                     session=!resumed => certificate, hello_done,
                                                         cipher=!DH   =>
                                                             GOTO:TLS_CLIENT_KEY_EXCHANGE
                                                         cipher==DH   =>
                                                             server_key_exchange,
                                                             GOTO: TLS_CLIENT_KEY_EXCHANGE
                                     session==resumed => send_change_cipher_spec,
                                                             send_server_finished_rec,
                                                             GOTO: TLS_CLIENT_CHANGE_CIPHER_SPEC

    TLS_CLIENT_CERTIFICATE
         -

    TLS_CLIENT_KEY_EXCHANGE
         proto=TLS_HANDSHAKE & msg_type=client_key_exchange
                                    => client_key_exchange_msg verarbeiten,
                                       compute_master_secret,
                                       compute_secret_keys,
                                                      GOTO: TLS_CLIENT_CHANGE_CIPHER_SPEC
    TLS_CERTIFICATE_VERIFY
         -

    TLS_CLIENT_CHANGE_CIPHER_SPEC
              proto=TLS_CHANGE_CIPHER_SPEC
                                    => Umstieg auf neue Client_Cipher_Suite,
                                                 GOTO: TLS_CLIENT_FINISHED

    TLS_CLIENT_FINISHED
              proto=TLS_HANDSHAKE  && msg_type=FINISHED
                                                     => client_finished_msg verarbeiten
                                                        if =!resumed =>  send_change_cipher_spec,
                                                                         send_server_finished_rec,
                                                                         GOTO: TLS_ACCEPT
                                                        if ==resumed =>  GOTO: TLS_ACCEPT
    TLS_ACCEPT
              proto=TLS_APP_DATA    =>   daten an mssl_robo weiterleiten
              proto=TLS_ALERT       =>   verbindung beenden,
                                         session-parameter speichern
    }
}
```

12.3 EC - Daten-Strukturen

Folgende Datenstrukturen sind in der EC-Implementation von Belang:

```
typedef struct ec_point_st {
            const EC_METHOD *meth;

            /* All members except 'meth' are handled by the method functions,
             * even if they appear generic */

            BIGNUM X;
            BIGNUM Y;
            BIGNUM Z; /* Jacobian projective coordinates:
                       * (X, Y, Z)  represents  (X/Z^2, Y/Z^3)  if  Z != 0 */
            int Z_is_one; /* enable optimized point arithmetics for special case */
} EC_POINT;


typedef struct ec_key_meth_data_st {
            int (*init)(EC_KEY *);
            void (*finish)(EC_KEY *);
            } EC_KEY_METH_DATA;


typedef struct ec_key_st {
            int version;

            EC_GROUP *group;

            EC_POINT *pub_key;
            BIGNUM        *priv_key;

            unsigned int enc_flag;
            point_conversion_form_t conv_form;

            int          references;

            EC_KEY_METH_DATA *meth_data;
            } EC_KEY ;



typedef struct ec_method_st {
            /* used by EC_METHOD_get_field_type: */
            int field_type; /* a NID */

            /* used by EC_GROUP_new, EC_GROUP_free, EC_GROUP_clear_free, EC_GROUP_copy: */
            int (*group_init)(EC_GROUP *);
            void (*group_finish)(EC_GROUP *);
            void (*group_clear_finish)(EC_GROUP *);
            int (*group_copy)(EC_GROUP *, const EC_GROUP *);

            /* used by EC_GROUP_set_curve_GFp, EC_GROUP_get_curve_GFp, */
            /* EC_GROUP_set_curve_GF2m, and EC_GROUP_get_curve_GF2m: */
            int (*group_set_curve)(EC_GROUP *, const BIGNUM *p, const BIGNUM *a, const BIGNUM *b,
                          BN_CTX *);
            int (*group_get_curve)(const EC_GROUP *, BIGNUM *p, BIGNUM *a, BIGNUM *b, BN_CTX *);

            /* used by EC_GROUP_get_degree: */
            int (*group_get_degree)(const EC_GROUP *);

            /* used by EC_GROUP_check: */
            int (*group_check_discriminant)(const EC_GROUP *, BN_CTX *);

            /* used by EC_POINT_new, EC_POINT_free, EC_POINT_clear_free, EC_POINT_copy: */
            int (*point_init)(EC_POINT *);
            void (*point_finish)(EC_POINT *);
            void (*point_clear_finish)(EC_POINT *);
            int (*point_copy)(EC_POINT *, const EC_POINT *);

            /* used by EC_POINT_set_to_infinity,
             * EC_POINT_set_Jprojective_coordinates_GFp,
             * EC_POINT_get_Jprojective_coordinates_GFp,
             * EC_POINT_set_affine_coordinates_GFp,     ..._GF2m,
             * EC_POINT_get_affine_coordinates_GFp,     ..._GF2m,
             * EC_POINT_set_compressed_coordinates_GFp, ..._GF2m:
             */
            int (*point_set_to_infinity)(const EC_GROUP *, EC_POINT *);
            int (*point_set_Jprojective_coordinates_GFp)(const EC_GROUP *, EC_POINT *,
                          const BIGNUM *x, const BIGNUM *y, const BIGNUM *z, BN_CTX *);
            int (*point_get_Jprojective_coordinates_GFp)(const EC_GROUP *, const EC_POINT *,
                          BIGNUM *x, BIGNUM *y, BIGNUM *z, BN_CTX *);
            int (*point_set_affine_coordinates)(const EC_GROUP *, EC_POINT *,
                          const BIGNUM *x, const BIGNUM *y, BN_CTX *);
```

```
            int (*point_get_affine_coordinates)(const EC_GROUP *, const EC_POINT *,
                         BIGNUM *x, BIGNUM *y, BN_CTX *);
            int (*point_set_compressed_coordinates)(const EC_GROUP *, EC_POINT *,
                         const BIGNUM *x, int y_bit, BN_CTX *);

            /* used by EC_POINT_point2oct, EC_POINT_oct2point: */
            size_t (*point2oct)(const EC_GROUP *, const EC_POINT *, point_conversion_form_t form,
                    unsigned char *buf, size_t len, BN_CTX *);
            int (*oct2point)(const EC_GROUP *, EC_POINT *,
                    const unsigned char *buf, size_t len, BN_CTX *);

            /* used by EC_POINT_add, EC_POINT_dbl, ECP_POINT_invert: */
            int (*add)(const EC_GROUP *, EC_POINT *r, const EC_POINT *a, const EC_POINT *b, BN_CTX
                         *);
            int (*dbl)(const EC_GROUP *, EC_POINT *r, const EC_POINT *a, BN_CTX *);
            int (*invert)(const EC_GROUP *, EC_POINT *, BN_CTX *);

            /* used by EC_POINTs_mul, EC_POINT_mul, EC_POINT_precompute_mult: */
            int (*mul)(const EC_GROUP *group, EC_POINT *r, const BIGNUM *scalar,
                         size_t num, const EC_POINT *points[], const BIGNUM *scalars[], BN_CTX *);
            int (*precompute_mult)(EC_GROUP *group, BN_CTX *);

            /* used by EC_POINT_is_at_infinity, EC_POINT_is_on_curve, EC_POINT_cmp: */
            int (*is_at_infinity)(const EC_GROUP *, const EC_POINT *);
            int (*is_on_curve)(const EC_GROUP *, const EC_POINT *, BN_CTX *);
            int (*point_cmp)(const EC_GROUP *, const EC_POINT *a, const EC_POINT *b, BN_CTX *);

            /* used by EC_POINT_make_affine, EC_POINTs_make_affine: */
            int (*make_affine)(const EC_GROUP *, EC_POINT *, BN_CTX *);
            int (*points_make_affine)(const EC_GROUP *, size_t num, EC_POINT *[], BN_CTX *);


            /* internal functions */

            /* 'field_mul', 'field_sqr', and 'field_div' can be used by 'add' and 'dbl' so that
             * the same implementations of point operations can be used with different
             * optimized implementations of expensive field operations: */
            int (*field_mul)(const EC_GROUP *, BIGNUM *r, const BIGNUM *a, const BIGNUM *b, BN_CTX
                         *);
            int (*field_sqr)(const EC_GROUP *, BIGNUM *r, const BIGNUM *a, BN_CTX *);
            int (*field_div)(const EC_GROUP *, BIGNUM *r, const BIGNUM *a, const BIGNUM *b, BN_CTX
                         *);

            int (*field_encode)(const EC_GROUP *, BIGNUM *r, const BIGNUM *a, BN_CTX *); /* e.g. to
                         Montgomery */
            int (*field_decode)(const EC_GROUP *, BIGNUM *r, const BIGNUM *a, BN_CTX *); /* e.g.
                         from Montgomery */
            int (*field_set_to_one)(const EC_GROUP *, BIGNUM *r, BN_CTX *);
}  EC_METHOD ;
```

```
typedef struct ec_group_st {
            const EC_METHOD *meth;

            EC_POINT *generator; /* optional */
            BIGNUM order, cofactor;

            int curve_name;/* optional NID for named curve */
            int asn1_flag; /* flag to control the asn1 encoding */
            point_conversion_form_t asn1_form;

            unsigned char *seed; /* optional seed for parameters (appears in ASN1) */
            size_t seed_len;

            void *extra_data;
            void *(*extra_data_dup_func)(void *);
            void (*extra_data_free_func)(void *);
            void (*extra_data_clear_free_func)(void *);

            /* The following members are handled by the method functions,
             * even if they appear generic */

            BIGNUM field; /* Field specification.
                           * For curves over GF(p), this is the modulus;
                           * for curves over GF(2^m), this is the
                           * irreducible polynomial defining the field.
                           */

            unsigned int poly[5]; /* Field specification for curves over GF(2^m).
                                   * The irreducible f(t) is then of the form:
                                   *     t^poly[0] + t^poly[1] + ... + t^poly[k]
                                   * where m = poly[0] > poly[1] > ... > poly[k] = 0.
                                   */

            BIGNUM a, b; /* Curve coefficients.
                          * (Here the assumption is that BIGNUMs can be used
                          * or abused for all kinds of fields, not just GF(p).)
                          * For characteristic  > 3,  the curve is defined
                          * by a Weierstrass equation of the form
                          *     y^2 = x^3 + a*x + b.
                          * For characteristic  2,  the curve is defined by
                          * an equation of the form
                          *     y^2 + x*y = x^3 + a*x^2 + b.
                          */

            int a_is_minus3; /* enable optimized point arithmetics for special case */

            void *field_data1; /* method-specific (e.g., Montgomery structure) */
            void *field_data2; /* method-specific */
}  EC_GROUP ;
```

```
typedef struct ec_curve_data_st {
            int          field_type;  /* either NID_X9_62_prime_field or
                                       * NID_X9_62_characteristic_two_field */
            const char *p;             /* either a prime number or a polynomial */
            const char *a;
            const char *b;
            const char *x;             /* the x coordinate of the generator */
            const char *y;             /* the y coordinate of the generator */
            const char *order;        /* the order of the group generated by the
                                       * generator */
            const BN_ULONG cofactor;/* the cofactor */
            const unsigned char *seed;/* the seed (optional) */
            size_t       seed_len;
            const char *comment;      /* a short (less than 80 characters)
                                       * description of the curve */
} EC_CURVE_DATA;
```

Ein Beispiel einer standardisierten EC_CURVE_DATA:

```
/* the nist prime curves */
static const unsigned char _EC_NIST_PRIME_192_SEED[] = {
            0x30,0x45,0xAE,0x6F,0xC8,0x42,0x2F,0x64,0xED,0x57,
            0x95,0x28,0xD3,0x81,0x20,0xEA,0xE1,0x21,0x96,0xD5};
static const EC_CURVE_DATA _EC_NIST_PRIME_192 = {
            NID_X9_62_prime_field,
            "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF",
            "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFC",
            "64210519E59C80E70FA7E9AB72243049FEB8DEECC146B9B1",
            "188DA80EB03090F67CBF20EB43A18800F4FF0AFD82FF1012",
            "07192b95ffc8da78631011ed6b24cdd573f977a11e794811",
            "FFFFFFFFFFFFFFFFFFFFFFFF99DEF836146BC9B1B4D22831",1,
            _EC_NIST_PRIME_192_SEED, 20,
            "192 bit prime curve from the X9.62 draft"
            };

typedef struct _ec_list_element_st {
            int          nid;
            const EC_CURVE_DATA *data;
            } ec_list_element;
```

12.4 EC- Funktionen

Für das grobe Verständnis der EC-Implementation sind folgende Funktionen von Belang:

```
EC_GROUP *EC_GROUP_new_by_nid(int nid);

EC_KEY *EC_KEY_new(void);

int EC_GROUP_set_curve_GFp(EC_GROUP *group, const BIGNUM *p, const BIGNUM *a, const BIGNUM *b, B
            N_CTX *ctx);

int EC_GROUP_set_curve_GF2m(EC_GROUP *group, const BIGNUM *p, const BIGNUM *a, const BIGNUM *b,
            BN_CTX *ctx)x

int EC_KEY_generate_key(EC_KEY *eckey);

int ECDH_compute_key(unsigned char *key, const EC_POINT *pub_key, EC_KEY *eckey)
```

Die wichtigsten EC-Grundfunktionen sind:

```
EC_POINT *EC_POINT_new(const EC_GROUP *group)

int EC_POINT_add(const EC_GROUP *group, EC_POINT *r, const EC_POINT *a, const EC_POINT *b, BN_CTX
            *ctxr)

int EC_POINT_dbl(const EC_GROUP *group, EC_POINT *r, const EC_POINT *a, BN_CTX *ctx)

int EC_POINT_mul(const EC_GROUP *group, EC_POINT *r, const BIGNUM *g_scalar, const EC_POINT *point,
            const BIGNUM *p_scalar, BN_CTX *ctx)

int EC_POINT_invert(const EC_GROUP *group, EC_POINT *a, BN_CTX *ctx)
```

12.5 Algorithmus-Auswahl über Funktionen-Pointer

Bei der Funktion

```
int EC_POINT_add(const EC_GROUP *group, EC_POINT *r, const EC_POINT *a, const EC_POINT *b, BN_CTX
```

```
*ctx),
```
die sich in ec_lib.c befindet, ist es beispielsweise sinnvoll, mehrere unterschiedliche Implementationen zu haben. Momentan wird diese Funktion auf

```
int ec_GFp_simple_add(const EC_GROUP *group, EC_POINT *r, const EC_POINT *a, const EC_POINT *b,
BN_CTX *ctx)
```
gemappt, die sich in ecp_smpl.c befindet.

Die Funktionen-Pointer können benutzt werden, um während der Laufzeit eine geeignete Implementation bzw. Algorithmus dynamisch auswählen zu können.

Man könnte diese Aufgabe auch mit Precompiler-Anweisungen lösen, was auch einige Vorteile, wie die kleinere Code-Grösse, hat.

Die Funktion EC_POINT_add() sieht beispielsweise wie folgt aus:

```
int EC_POINT_add(const EC_GROUP *group, EC_POINT *r, const EC_POINT *a, const EC_POINT *b, BN_CTX
*ctx)
            {
            if (group->meth->add == 0)
                        {
                        ECerr(EC_F_EC_POINT_ADD, ERR_R_SHOULD_NOT_HAVE_BEEN_CALLED);
                        return 0;
                        }
            if ((group->meth != r->meth) || (r->meth != a->meth) || (a->meth != b->meth))
                        {
                        ECerr(EC_F_EC_POINT_ADD, EC_R_INCOMPATIBLE_OBJECTS);
                        return 0;
                        }
            return group->meth->add(group, r, a, b, ctx);
            }
```

Sie ruft nur eine Fuktion auf, deren Funktions-Pointer in „group->meth->add“ gespeichert ist. Bei einer EC_GROUP befindet sich unter EC_METHOD eine Liste von Funktionen-Pointern:

```
typedef struct ec_method_st {
            ....................
            /* used by EC_POINT_add, EC_POINT_dbl, ECP_POINT_invert: */
            int (*add)(const EC_GROUP *, EC_POINT *r, const EC_POINT *a, const EC_POINT *b, BN_CTX
                        *);
            int (*dbl)(const EC_GROUP *, EC_POINT *r, const EC_POINT *a, BN_CTX *);
            int (*invert)(const EC_GROUP *, EC_POINT *, BN_CTX *);
            ....................
}  EC_METHOD ;
```

Diese Liste kann man nun mit Funktionen-Pointern befüllen:

```
const EC_METHOD *EC_GFp_mont_method(void)
            {
            static const EC_METHOD ret = {
                        .............
                        ec_GFp_simple_add,
                        .............
            return &ret;
            }
```

Der Aufruf von group->meth->add() führt somit zum Aufruf der Funktion

```
int ec_GFp_simple_add(const EC_GROUP *group, EC_POINT *r, const EC_POINT *a, const EC_POINT *b,
                        BN_CTX *ctx)
```
über den Fuktionen-Pointer „int (*add)(...)**“.**

12.6 Bignum-Strukturen

Die Zentrale Datenstruktur in der Bignumber-Library ist die „BIGNUM“:

typedef struct bignum_st

```
{
BN_ULONG *d; /* Pointer to an array of 'BN_BITS2' bit chunks. */
int top;     /* Index of last used d +1. */
/* The next are internal book keeping for bn_expand. */
int max;     /* Size of the d array. */
int neg;     /* one if the number is negative */
int flags;
} BIGNUM;
```

Die eigentlichen Daten werden durch den Pointer *d adressiert.

12.7 Bignum-Funktionen

Die EC-Grundfunktionen basieren auf den Funktionen der OpenSSL-Bignumber-Library.
In der Funktion

```
int ec_GFp_simple_add(const EC_GROUP *group, EC_POINT *r, const EC_POINT *a, const EC_POINT *b,
                          BN_CTX *ctx)
```

wird über den Funktionen-Pointer „field_mul“im Struct „EC_RECORD“ die Funktion

```
int ec_GFp_mont_field_mul(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, const BIGNUM *b, BN_CTX
                          *ctx)
```

aufgerufen, diese ruft ihrerseits die

```
int BN_mod_mul_montgomery(BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_MONT_CTX *mont, BN_CTX *ctx)
```

auf. Diese modulare Multiplikation befindet sich in der Bignum-Library und baut auf einfacheren Bignum-Grundfunktionen auf. Das geht runter bis zur Word-Multiplikation. Diese Funktion sind für neuere Prozessoren optimiert, weswegen die Schleife in allen vierfach aufgefächert wird. Auf dem 186er bringt es nichts, aber es entstehen keine Leistungseinbussen dadurch. Dafür aber wird der Code umfangreicher. Ein Beispiel:

```
BN_ULONG bn_mul_add_words(BN_ULONG *rp, BN_ULONG *ap, int num, BN_ULONG w)
/**********************************************************************/
{
             BN_ULONG c1=0;

             assert(num >= 0);
             if (num <= 0) return(c1);

             while (num&~3)
                          {
                          mul_add(rp[0],ap[0],w,c1);
                          mul_add(rp[1],ap[1],w,c1);
                          mul_add(rp[2],ap[2],w,c1);
                          mul_add(rp[3],ap[3],w,c1);
                          ap+=4; rp+=4; num-=4;
                          }
             if (num)
                          {
                          mul_add(rp[0],ap[0],w,c1); if (--num==0) return c1;
                          mul_add(rp[1],ap[1],w,c1); if (--num==0) return c1;
                          mul_add(rp[2],ap[2],w,c1); return c1;
                          }

             return(c1);
}
```

Die Funktion verwendet auf der letzten Stufe Makros um die Stackbedieung zu sparen.

```
#define mul_add(r,a,w,c) { \
```

```
            BN_ULLONG t; \
            t=(BN_ULLONG)w * (a) + (r) + (c); \
            (r)= Lw(t); \
            (c)= Hw(t); \
            }
```

Trotzdem kann der Borland C++-Compiler diesen Code nur sehr ineffizient übersetzen. Ein Problem bei der Programmierung in C ist, dass man auf das Carry-Flag nicht direkt zugreifen kann. Man muss den Code anders schreiben.

Zuerst haben wir die Makros in Assembler geschrieben.Das obere Makro sieht nun wie folgt aus:

```
#define mul_add(r,indexr,a,indexa,w,c) { \
asm{\
   les bx,a;                  /*       den wert von a in es:bx einlesen, dann a[indexa] in BX */\
   mov ax,es:[bx+indexa*2];\
   mul w;                    /* w * (a)         (high word im dx, low word im ax) */\
   xor cx,cx;\
   les bx,r;                  /* + (r)           ( r[indexr]   zu AX addieren) */\
   add ax,es:[bx+indexr*2];\
   adc dx,cx;\
   add ax,c;                  /* + (c);          */\
   adc dx,cx;                  /* (c)= Hw(t);*/\
   mov c,dx;\
   mov es:[bx+indexr*2],ax;      /* (r)= Lw(t); */\
}\
}
```

Dieser Code konnte auf der Stufe „Makro“ nicht mehr optimiert werden. Uns waren die ständigen Adress-Dereferenzierungen mit LES ein Dorn im Auge. Um den Code weiter zu Optimieren mussten wir eine Stufe nach oben in der Funktionen-Aufrufkette gehen.

```
BN_ULONG bn_mul_add_words(BN_ULONG *r, BN_ULONG *a, int n, BN_ULONG w)
/**********************************************************************/
{
            BN_ULONG c=0;       //liegt auf dem Stack bei -2
            assert(n >= 0);
            if (n <= 0) return(c);
asm{
   push si
   push di
   push ds

   lds di,r                   ;//r[   in ds:di
   les si,a                   ;//a[   in es:si
   xor ax,ax              ;//accu =Low-Word
   xor bx,bx              ;//(n+=2)
   mov cx,n                ;//n--
   xor dx,dx               ;//High-Word

}goonx:asm{                   //c:r0 = w*a0+c+r0           0,1,2,...

   mov ax,es:[si+bx]
   mul w                      ;// a * w         (high word im dx, low word im ax)

   add ax,c                   ;// + c            altes carry
   adc dx,0

   add ax,ds:[di+bx]          ;// + r
   adc dx,0

   mov ds:[di+bx],ax          ;//r <=     ergebnis zurückschreiben (carry bleibt im dx)
   mov c,dx

   inc bx
   inc bx
   dec cx                             ;//if (--n <= 0) break;
   jg goonx                           ;//   jump on greater  (signed values)

   pop ds
   pop di
   pop si
   mov c,dx
}
            return(c);
}
```

Der Overhead durch den Funktions-Aufruf ist enorm. Der Call ist dabei nur nebensächlich. Das Pushen der Parameter auf den Stack und das Abholen der Rückgabeparameter benötigen sehr viel Zeit.

Auf einem hohen Level, dort wo die Funktionen sehr mächtig geworden sind, nur noch selten aufgerufen werden und lange laufen, ist es nicht mehr notwendig den Code in Assembler zu optimieren. Ein Ort, an dem man meinen könnte etwas einsparen zu können, aber praktisch nichts zu gewinnen ist:

Die Funktion ec_GPp_mont_field_mu(...) ruft eine andere über einen Funktionen-Pointer auf:

Die Auflösung des Funktionen-Pointers benötigt zusätzliche Zeit:

```
field_mul = group->meth->field_mul;
```

```
0000:fb04      c4 5e 06         les          bx,ss:[06+bp]
0000:fb07      26 c4 1f         les          bx,es:[bx]
0000:fb0a      26 8b 47 7c      mov          ax,es:[7c+bx]
0000:fb0e      26 8b 57 7a      mov          dx,es:[7a+bx]
0000:fb12      89 46 fe         mov          ss:[-02+bp],ax
0000:fb15      89 56 fc         mov          ss:[-04+bp],dx
```

Der eigentliche Funktionsaufruf ist noch viel länger (Rückgabe kommt noch dazu):

```
       field_mul(...)
0000:fd0a      ff 76 14         push         ss:[14+bp]
0000:fd0d      ff 76 12         push         ss:[12+bp]
0000:fd10      ff 76 e6         push         ss:[-1a+bp]
0000:fd13      ff 76 e4         push         ss:[-1c+bp]
0000:fd16      ff 76 ee         push         ss:[-12+bp]
0000:fd19      ff 76 ec         push         ss:[-14+bp]
0000:fd1c      ff 76 ea         push         ss:[-16+bp]
0000:fd1f      ff 76 e8         push         ss:[-18+bp]
0000:fd22      ff 76 08         push         ss:[08+bp]
0000:fd25      ff 76 06         push         ss:[06+bp]
0000:fd28      ff 5e fc         call far     ss:[-04+bp]
0000:fd2b      83 c4 14         add          sp,14
0000:fd2e      0b c0            or           ax,ax
0000:fd30      75 03            jnz          fd35
0000:fd32      e9 ad 03         jmp          00e2
```

Auf einem tiefen Level würde man nun zumindest versuchen, den Funktionen-Pointer durch einen normalen Aufruf abzulösen. Diese Funktion ist aber schon so weit oben, dass es keine spürbare Performance-Verbesserung dadurch gäbe. Man muss sich um die Low-Level-Funktionen kümmern, wie Word-Multiplikation, -Addition, -Shift und noch viele weitere. Sogar bei der Polynom-Multiplikation war mit Assembler viel zu holen. Nur durch die Kodierung dieser Funktion in Assembler wurde eine Berechnungszeitverbesserung für eine 192er-EC-Kurve von 19 auf 14.5 Sekunden erzielt. Die Polynom-Multiplikation kann jedoch nicht so effizient wie die normale Multiplikation realisiert werden, weil es dafür keine entsprechende Operation im Prozessor gibt und diese über Shift-Befehle und XOR-Additionen gelöst werden muss.

12.8 Software-Tests

Es wurden fortlaufend kleinere Tests durchgeführt.

Der neu erstellte SSL-Handshake wurde mit unterschiedlichen Cipher-Suites getestet. Die wichtigsten Ciphers waren:

- RSA_RC4_MD5
- AECDH_RC4_SHA
- ECDH_RSA_RC4_SHA

Die Verschlüsselungs-Funktionen wurden zuerst einzeln und dann im SSL-Layer eingebunden getestet.

Die Berechnungszeiten wurden mit der Funktion

```
void RTX_Get_System_Ticks(unsigned long far * ticks);
```

aus der RTOS-Library gemessen. Diese Funktion misst die Zeit mit einer Auflösung von 1ms.

Die Bignum-Funktionen wurden nach der Codierung in Assembler auf die Richtigkeit der Ergebnisse getestet. Zusätzlich wurden fortlaufend die Berechnungszeiten überprüft um herauszufinden ob die Änderung sich gelohnt hat und sich weitere dieser Art lohnen würden.

Einige Schlusstests mit Debug-Meldungen auf dem Server und dem Client werden im Anhang aufgelistet. Aus diesen ist u.a. die Richtigkeit der Berechnungen herauslesbar. Die Entscheidungen in der Handshake-Statemachine können bspw. auch mit verfolgt werden.

13 Software-Bedienung

13.1 EC-Zertifikat-Generierung

Ab OpenSSL v0.98 wird EC vollständig integriert sein. Auf dem FTP-Server von openssl.org steht jedoch immer ein aktueller Snapshot zur Verfügung. Dieser ist bereits in der Lage EC-Zeritifikate zu generieren. Die Online-Dokumentation und die Man-Pages enthalten jedoch noch keine Anleitung zur EC-Zertifikat-Generierung, weshalb wir dies hier kurz erläutern.

Im Nachfolgenden wird die Zertifikat-Generierung schrittweise erklärt. Ausserdem wird gezeigt wie man den S_Client von OpenSSL konfigurieren muss, um eine Verbindung zum EC-fähigen Server aufzubauen.

Zuerst muss die Certification-Authority-Infrastruktur kreiert werden. Diese haben wir einfachheitshalber auf dem Rechner aufgesetzt, auf dem der SSL_Client dann später laufen sollte.

```
cd /usr/local/ssl/
./misc/CA.sh -newca
```

Ein RSA-Schlüsselpaar kreieren (mit dem Private-Key wird die CA das Zertifikat signieren)

```
openssl genrsa -out ./demoCA/private/cakey.pem 2048
```

Ein selbstsiginiertes Zertifikat für die CA ausstellen.

```
openssl req -new -x509 -days 730 -key ./demoCA/private/cakey.pem -out ./demoCA/cacert.pem
```

Das CA-Zertifikat wird nun ins PKCS12-Format gewandelt, damit es von Browsern eingelesen werden kann.

```
openssl pkcs12 -export -in cacert.pem -out cacert.p12 -inkey ./private/cakey.pem
```

Einen EC-Schlüssel generieren. Dieser kann für das EC-Diffie-Helman-Verfahren gebraucht werden. Der Schlüssel besteht aus einem Secret-Key (eine Bignumber) und einem Startpunkt G (ein Punkt auf der EC-Kurve im Feld). Aus diesen wird ein Public-Key generiert ($G^{SecretKey}$). Der Public-Key ist auch ein Punkt auf der EC-Kurve. Das Zertifikat enthält den Kurventyp, den Startpunkt G und den Public-Key $G^{SecretKey}$.

```
openssl ecparam -genkey -name prime192v1 -out ec_key.pem
```

Einen Certificate Request an die CA erstellen.

```
openssl req -new -key ec_key.pem -out ec_key_req.pem
```

Bei der CA ausgeführte Signierung des EC-Schlüssels

```
openssl ca -keyfile ./demoCA/private/cakey.pem -in ec_key_req.pem -out ec_cert.pem
```

Das Zertifikat muss für die Embedded-Umgebung vom PEM-Format zum DER-Format gewandelt werden. Weil das Zertifikat im DER-Format übertragen wird, kann somit die Wandlung dem Embedded-System erspart bleibten.

```
openssl x509 -in ec_cert.pem -out ec_cert.der -outform DER
```

Der EC Private-Key muss auch in ein Format gewandelt werden, welches vom Embedded-System einfach eingelesen werden kann.

```
./fswcert -k --type ec ec_key.pem > ec_key.hst
```

Hierfür musste das Tool „fswcert“ erweitert werden, damit es das EC-Private-Key-Format versteht. Darauf wird nicht weiter eingegangen. Der Quellcode vom angepassten fswcert befindet sich auf der beiliegenden CD-ROM.

Das EC-Zertifikat und der EC-Privat-Key müssen nun per FTP auf das Embedded-System übertragen werden und richtig umbenannt werden (siehe die vordefinierten Dateinamen in „mssl.c“). Dann muss das mSSL gestartet werden.

13.2 Zertifikat-Formate

- Plaintext-ASN1

  ASN1 heisst Abstract Syntax Notation One und ist eine Strukturbeschreibungssprache. Eine andere Strukturbeschreibungssprache ist z.B. das viel bekanntere XML.

  In diesem Format ist die ASN1 in ASCII-Zeichen ausgeschrieben:

  Wir definieren einen Datentyp Point:

  „Point ::= SET {x INTEGER, y INTEGER}“

  Nun definieren wir eine Variable p des Datentyps Point:

  p ::= { x 9, y 33}

- DER-Format

  Das ist immer noch ASN1, diesmal ist es aber mit Encoding-Regeln binär kodiert worden. Dieses Format ist viel kompakter als Plaintext. Das obere Beispiel im DER-Format würde wie folgt aussehen (dargestellt mit dem hexadezimalen Zahlensystem):

  p ::= { x 9, y 33} => 31'06' 02'01'09' 02'01'21

  Das erste Byte „31“ steht für das Schlüsselwort SET. Das zweite Byte „06“ bezeichnet die Länge des Inhaltes von SET. „02“ heisst INTEGER mit der Länge „01“ und Inhalt „09“. Und noch ein Integer mit dem Inhalt „21“ was in dezimaler Schreibweise 33 bedeutet.

- PEM-Format

  Das DER-Format wird BASE64-kodiert. Damit ist es immer noch viel kompakter als Plaintext-ASN1.

Das Zertifikat muss im DER-Format gespeichert werden, damit die Konvertierung aus dem PEM-Format nicht jedesmal vom Embedded-System erledigt werden muss.

Der Schlüssel muss in dem sehr primitiven HST-Format gespeichert werden, damit auf die ASN1-Library auf dem Embedded-System verzichtet werden kann.

13.3 Aufbau einer SSL-Session mit dem OpenSSL-Client

Nach dem Starten von MSSL kann nun vom Client aus eine SSL-Session angefordert werden. Wenn ECC verwendet werden soll, geht das nur mit dem S_CLIENT von OpenSSL.

```
openssl s_client -connect 160.85.162.77:443 -ssl3 -cipher ECDH-RSA-RC4-SHA
```

Ein Browser kann nicht verwendet werden, weil noch keiner die EC-Algorithmen unterstützt.

Falls der S_Client das Zertifikat als ungültig bezeichnet, kann lokal mit

```
openssl verify -purpose sslclient ./ec_cert.pem
```

geprüft werden.

Falls dies nicht klappt, sollte noch ein Pfad zum CA-Zertifikat angegeben werden.

```
openssl verify -CAfile ./demoCA/cacert.pem -purpose sslclient ./ec_cert.pem
```

Die „CAfile“-Option kann auch beim S_Client angegeben werden, wenn im Konfigurationsfile von OpenSSL der Pfad falsch gesetzt ist.

```
openssl s_client -connect 160.85.162.77:443 -ssl3 -cipher ECDH-RSA-RC4-SHA
```

Beim S_Client kann auch noch eine „-debug“ Option angegeben werden, womit begrenzt Debugging möglich ist.

Nun können auch andere Ciphers ausprobiert werden.

```
openssl s_client -connect 160.85.162.77:443 -ssl3 -cipher ECDH-ECDSA-RC4-SHA
```

14 Schlusswort

Am Anfang dieser Diplomarbeit standen wir vor einer nicht überwindbar scheinenden Hürde, diese stellte der mathematische Hintergrund der elliptischen Kurven dar. Diese Hürde zu bewältigen war eine extrem herausfordernde Aufgabe, die sehr viel Zeit verschlang. Jedoch war diese Grundlagenarbeit unumgänglich, um gewisse Entscheidungen treffen zu können.

Ein weiterer grosser Teil war die Einarbeitung in OpenSSL, dazu konnten wir zwar die in einer Projektarbeit gewonnen Kenntnisse von einem Open Source Bluetooth-Stack in einigen Bereichen wieder einsetzen, doch der Programmierstil ist halt doch von Projekt zu Projekt sehr verschieden. Als besonderen Leckerbissen offenbarte uns der OpenSSL-Code die Einsicht, dass auch in normalem C eine Art objektorientierte Programmierung möglich ist.

Als wir mit der Einarbeitung in OpenSSL soweit waren, dass wir mit einer eigenen Implementation loslegen konnten, überraschte uns Sun Microsystems mit der Schenkung ihrer EC-Implementation. Da diese die beiden wichtigsten Feldtypen und eine grosse Anzahl an Standard-Kurven unterstützt, wurde der ECC-Code ebenfalls auf das Zielsystem portiert. Die ersten Messungen brachten dann aber eher ernüchternde Resultate, weshalb wir einen Blick in verschiedene Multi-Precision-Integer-Implementationen warfen.

Durch diesen Streifzug entstanden quasi als Nebenprodukt einfache EC-Implementationen für die Mozilla-MPI- und die PGP-Bignumber-Bibliotheken. Mit den neuen Erkenntnissen bewaffnet waren wir dann auch in der Lage, die Schwachpunkte der OpenSSL-Bignumber-Bibliothek auszumachen und an neuralgischen Stellen durch cleveren Assembler-Code auszumerzen.

Die wichtigste Schlussfolgerung konnten wir deswegen erst in der zweitletzten Woche ziehen: Ohne effiziente Multi-Precision-Bibliothek ist keine effiziente EC-Implementation möglich. Dies ist auch der Punkt, den es in einem nächsten Schritt anzupacken gilt. Wir sind überzeugt, dass sich die Berechnungszeiten noch weiter senken lassen, wenn eine einfache Multi-Precision-Bibliothek komplett in Assembler geschrieben wird und eventuell einige Einschränkungen in Kauf genommen werden.

Rückblickend war diese Diplomarbeit sehr herausfordernd und interessant. Wir konnten unser Know-How in der Kryptographie, einem sehr spannenden und wichtigen Bereich der Informatik, erweitern und vertiefen.

Das Gebiet der Datensicherheit und Privatsphäre ist gerade in der heutigen Zeit, in der in Folge von Terrorismus Gesetze erlassen werden, ohne eine politische Diskussion zu führen, aktuell wie nie zuvor.

15 Anhang

15.1 Referenzen

[Blake99] I. Blake, G. Seroussi & N. Smart, Elliptic Curves in Cryptography, Cambridge University Press, 1999

[CeSec] Certicom, Remarks on the Security of the Elliptic Curve Cryptosystem, Juli 2000

[ECCT] Certicom, ECC Online Tutorial, http://www.certicom.com/resources/ecc_tutorial/ecc_tutorial.html

[Gor98] D. Gordon, A survey of fast exponentiation methods, Journal of Algorithms, 1998

[HAC96] A. Menezes, P. van Oorschot, S. Vanstone, Handbook of Applied Cryptography, CRC Press, 1996

[Hank00] D. Hankerson, J. Hernandez, A. Menezes, Software Implementation of Elliptic Curve Cryptography Over Binary Fields, 2000

[Hase99] T. Hasegawa, J. Nakajima, M. Matsui, A Small and Fast Software Implementation of Elliptic Curve Cryptosystems over GF(p) on a 16-Bit Microcomputer, IEICE Trans. Fundamentals, Vol. E82A, Januar 1999

[IETF-ECC02] TLS Working Group, Internet-Draft, ECC Cipher Suites for TLS, August 2002

[Möll01] B. Möller, Algorithms for Multi-exponentiation, TU Darmstadt, 2001

[P1363] IEEE P1363/D13 (Draft Version 13), Standard Specifications for Public Key Cryptography, November 1999

[X9-62] ANSI X9.63-1998, Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), September 998

[X9-63] ANSI X9.63-199x, Public Key Cryptography For The Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography, Januar 1999

[SEC1] Certicom Research, SEC 1: Elliptic Curve Cryptography, Standards for Efficient Cryptography Group (www.secg.org), September 2000

[1] Certicom Research, SEC 2: Recommended Elliptic Curve Domain Parameters, Standards for Efficient Cryptography Group (www.secg.org), September 2000

[2] Network Working Group, RFC 2246, The TLS Protocol 1.0, Januar 1999

[3] TLS Working Group, The SSL Protocol 3.0, November 1996

[4] A. Steffen, Secure Network Communication Part I: Introduction to Cryptography, Zürcher Hochschule Winterthur, März 2002

[5] A. Steffen, Secure Network Communication Part II: Introduction to Cryptography, Zürcher Hochschule Winterthur, März 2002

[6] A. Steffen, The Elliptic Curve Cryptosystem, Zürcher Hochschule Winterthur, Juli 2002

[7] A. Steffen, Secure Communications in Distributed Embedded Systems, Zürcher Hochschule Winterthur

[8] Certicom, Current Public-Key Cryptographic Systems, Juli 2000

[9] RSA Security Inc., Übersetung von Lutz Donnerhacke, Kryptographie FAQ 3.0, http://www.iks-jena.de/mitarb/lutz/security/cryptfaq/

[10] Dr. K. Reinhardt, Interaktives Kryptologie-Skript, Universität Tübingen, http://www-fs.informatik.uni-tuebingen.de/~reinhard/krypto/

[11] Certicom Research, The Elliptic Curve Digital Signature Algorithm (ECDSA)

[12] D. Johnson, A.Menezes, The Elliptic Curve Digital Signature Algorithm

(ECDSA), Februar 2000
[13] D. Bailey, C.Paar, Optimal Extension Fields for Fast Arithmetic in Public-Key Algorithms, Crypto '98, 1998
[14] A. Miyaji, T. Ono and H. Cohen, Effcient elliptic curve exponentiation, Matsushita
[15] H. Baier, Effcient Algorithms for Generating Elliptic Curves over Finite Fields Suitable for Use in Cryptography, TU Darmstadt, 2002
[16] Certicom, The Elliptic Curve Cryptosystem for Smart Cards, Mai 1998
[17] A. Woodbury, Effcient Algorithms for Elliptic Curve Cryptosystems on Embedded Systems, Worcester Polytechnic Institute, September 2001
[18] A. Gathani, Implementation Of Elliptic Curve Cryptography In Embedded System, November 2001
[19] H. Baier, Elliptic Curves of Prime Order over Optimal Extension Fields for Use in Cryptography, TU Darmstadt, 2001
[20] C. Paar, Implementation Options for Finite Field Arithmetic for Ellitpic Curve Cryptosystems, 1999
[21] J.H. Park, J.H. Cheon, S.G. Hahn, New Type of Optimal Extension Fields and its Applications
[22] M. Schimmler, V. Bunimov, B. Tolg, Area-Time-Efficient Montgomery Modular Multiplication, TU Braunschweig
[23] C. Koç, T. Acar, Montgomery Multiplication in GF(2k), April 1998
[24] E. Savaş, C Koç, The Montgomery Modular Inverse – Revisited, IEEE Transactions on Computers, Vol. 49, No. 7, Juli 2000
[25] D. J. Guan, Montgomery Algorithm for Modular Multiplication, September 2001
[26] S.C. Shantz, From Euclid's GCD to Montgomery Multiplication to the Great Divide, Juni 2001
[27] Uwe Krieger, Elliptische Kurven - Basis für ein alternatives Public Key Kryptosystem, cv cryptovision
[28] R. Schroeppel, H. Orman, S. O'Malley, Fast Key Exchange with Elliptic Curve Systems, University of Arizona, März 1995
[29] J. Hills, Introduction to Elliptic Curve Cryptography: A Mathematical Overview, Oktober 2001
[30] P. Karu, J. Loikkanen, Practical Comparison of Fast Public-key Cryptosystems, Helsinki University of Technology
[31] M. Brown, D. Hankerson, J. López, A. Menezes, Software Implementation of the NIST Ellitpic Curves Over Prime Fields
[32] J.W. Chung, S.G. Sim, P.J. Lee, Fast Implementation of Elliptic Curve Defined over $GF(p^m)$ on CalmRISC with MAC2424 Coprocessor
[33] D. Hankerson, J. López Hernandez, A. Menezes, Software Implementation of Ellitpic Curve Cryptography Over Binary Fields
[34] M. Aydos, T. Yanık, C. Koç, High-Speed Implementation of an ECC-based Wireless Authentication Protocol on an ARM Microprocessor, Oregon State University
[35] C.H. Lim, H.S. Hwang, Fast Implementation of Elliptic Curve Arithmetic in $GF(p^n)$
[36] J. López, R. Dahab, Improved Algorithms for Ellitpic Curve Arithmetic in $GF(2^n)$, Oktober 1998

[37] J. López, R. Dahab, Fast Multiplication on Elliptic Curves over GF(2^m) without Precomputation
[38] F. Lemmermeyer, Elliptische Kurven I, September 1999
[39] C. Koç, Elliptic Curve Cryptosystems, Oregon State University
[40] A. Miyaji, T. Ono and H. Cohen, Effcient elliptic curve exponentiation
[41] K. Fong, D. Hankerson, J. López, A. Menezes, M. Tucker, Performance comparisons of elliptic curve systems in software, Oktober 2001
[42] J. López, R. Dahab, Performance of Elliptic Curve Cryptosystems, Mai 2000
[43] S.M. Thompson, An Introduction to Elliptic Curve Cryptosystems, 1998
[44] http://www.securitytechnet.com/crypto/algorithm/ecc.html
[45] T. Müller, H. Käser, R. Gübeli, R. Klaus, Technische Informatik I, vdf Lehrbuch, September 1999
[46] M. Brill, Mathematik für Informatiker, Hanser Verlag, 2001
[47] B. Stroustrup, The C++ Programming Language, Second Edition, Addison Wesley, 1991

15.2 Source-Code

15.2.1 EC-Grundfunktionen

```
/* Diplomarbeit Sna 02/3
 * Einfache EC-Implementation über ein Primfeld
 * unter Verwendung projektiver Koordinaten.
 *
 * Datum: 28.10.2002
 * Autor: D. Wild
 */

#include <stdlib.h>
#include <ec/ec.h>
#include <bignum/bn.h>
#include <bignum/bnprint.h>


// Alloziert Platz für einen EC-Punkt
ecpoint* ecpoint_new(void) {
      ecpoint *point;
      point = malloc(sizeof *point);

      bnBegin(&point->X);
      bnBegin(&point->Y);
      bnBegin(&point->Z);
      bnSetQ(&point->Z, 1);
      point->Z_is_one = TRUE;
      return point;
}

// Gibt Platz von EC-Punkt frei
void ecpoint_free(ecpoint *point) {
      if(point == NULL) {
            return;
      }

      bnEnd(&point->X);
      bnEnd(&point->Y);
      bnEnd(&point->Z);
      free(point);
}

// Kopiert einen EC-Punkt
int ecpoint_copy(ecpoint *result, ecpoint *point) {
      if(result==NULL) {
            return 0;
      }

      bnCopy(&result->X,&point->X);
      bnCopy(&result->Y,&point->Y);
      bnCopy(&result->Z,&point->Z);
      result->Z_is_one = point->Z_is_one;
      return 1;
}

// Konvertiert projektive Koordinaten eines Punktes in affine Koordinaten
int ecpoint_toaffine(const eccurve *curve, ecpoint *point) {
      BigNum temp1, temp2;
      int res=1;

      if(point->Z_is_one) {
            return 1;
      }
      if(ecpoint_isatinfinity(point)) {
            return 0;
      }

      bnBegin(&temp1);
      bnBegin(&temp2);

      //X = X / Z^2
      if(bnSquareMod(&temp2,&point->Z,&curve->field) <0) //T2=Z^2
            return -1;
      if(bnInv(&temp1,&temp2,&curve->field) <0) //T1=1/T2
            return -1;
      if(bnMulMod(&point->X,&point->X,&temp1,&curve->field) <0) //X=X*T1
            return -1;

      //Y = Y / Z^3
      if(bnMulMod(&temp1,&temp2,&point->Z,&curve->field) <0 ) //T1=T2*Z
            return -1;
      if(bnInv(&temp1,&temp1,&curve->field) <0) // T1=1/T1
            return -1;
      if(bnMulMod(&point->Y,&point->Y,&temp1,&curve->field) <0) //Y=Y*T1
```

```
            return -1;

      bnSetQ(&point->Z,1);
      point->Z_is_one = 1;

      bnEnd(&temp1);
      bnEnd(&temp2);
      return res;
}

// Testet, ob ein Punkt in der Unendlichkeit liegt
int ecpoint_isatinfinity(const ecpoint *point) {
      return !bnCmpQ(&point->Z,0);
}

// Testet, ob ein Punkt auf der Kurve liegt
int ecpoint_isoncurve(const eccurve *curve, const ecpoint *point) {
      BigNum temp1, temp2, temp3;
      int res;

      if(ecpoint_isatinfinity(point)) {
            return 1;
      }

      bnBegin(&temp1);
      bnBegin(&temp2);
      bnBegin(&temp3);

      /* We have a curve defined by a Weierstrass equation
       *      y^2 = x^3 + a*x + b.
       * The point to consider is given in Jacobian projective coordinates
       * where  (X, Y, Z)  represents  (x, y) = (X/Z^2, Y/Z^3).
       * Substituting this and multiplying by  Z^6  transforms the above equation into
       *      Y^2 = X^3 + a*X*Z^4 + b*Z^6.
       */
      bnSetQ(&temp2,3);
      bnExpMod(&temp1,&point->X,&temp2,&curve->field); //T1=X^3

      if(!point->Z_is_one) {
            //projective:
            bnSetQ(&temp3,4);
            bnExpMod(&temp2,&point->Z,&temp3,&curve->field); //T2=Z^4
            bnMulMod(&temp2,&point->X,&temp2,&curve->field); //T2=X*T2

            bnMulMod(&temp2,&curve->a,&temp2,&curve->field); //T2=a*T2
            bnAddMod(&temp1,&temp2,&curve->field); //T2=T1+T2

            bnSetQ(&temp3,6);
            bnExpMod(&temp2,&point->Z,&temp3,&curve->field); //T2=Z^6
            bnMulMod(&temp2,&curve->b,&temp2,&curve->field); //T2=b*T2

            bnAddMod(&temp1,&temp2,&curve->field); //T1=T1+T2
            //X^3 + a*X*Z^4 + b*Z^6
      }
      else {
            //affine:
            if(curve->a_minus3) {
                  bnCopy(&temp2,&point->X);
                  bnLShift(&temp2,1);
                  bnAddMod(&temp2,&point->X,&curve->field); //T2=T2+X
                  bnSubMod(&temp1,&temp2,&curve->field); //T1=T1-T2
            }
            else {
                  bnMulMod(&temp2,&curve->a,&point->X,&curve->field); //T2=a*X
                  bnAddMod(&temp1,&temp2,&curve->field); //T1=T1+T2
            }
            bnAddMod(&temp1,&curve->b,&curve->field); //T1=T1+b
            //T1 = X^3 + a*X + b
      }

      bnSquareMod(&temp2,&point->Y,&curve->field); //T2=Y^2

      res = !bnCmp(&temp1,&temp2);

      bnEnd(&temp1);
      bnEnd(&temp2);
      bnEnd(&temp3);

      return res;
}

// Führt auf einer elliptischen Kurve eine Punktaddition durch
int ecpoint_add(const eccurve *curve, ecpoint *result, ecpoint *point1, ecpoint *point2) {
      BigNum temp1, temp2;
      int res=0;

      if(result==NULL) {
            return 0;
```

```
}

if(point1==point2) {
      return ecpoint_dbl(curve,result,point1);
}
if(ecpoint_isatinfinity(point1)) {
      if(result!=point2) {
            return ecpoint_copy(result,point2);
      }
      return 1;
}
if(ecpoint_isatinfinity(point2)) {
      if(result!=point1) {
            return ecpoint_copy(result,point1);
      }
      return 1;
}

bnBegin(&temp1);
bnBegin(&temp2);

/* Implmentiert das Verfahren beschrieben in "A Small and Fast Software
 * Implementation of Elliptic Curve Cryptosystems over GF(p) on a 16-Bit
 * Microcomputer" von T. Hasegawa, J. Nakajima und M. Matsui erschienen
 * in IEICE Trans. Fundamentals, Vol. E82-A, No. 1 January 1999.
 */
if(!point2->Z_is_one) {
      res=bnSquareMod(&temp1,&point2->Z,&curve->field); //T1=Z2*Z2
      res=bnMulMod(&result->X,&point1->X,&temp1,&curve->field); //X=X1*T1
      res=bnMulMod(&temp1,&point2->Z,&temp1,&curve->field); //T1=Z2*T1
      res=bnMulMod(&result->Y,&point1->Y,&temp1,&curve->field); //Y=Y1*T1
}
else if(result!=point1) {
      bnCopy(&result->X,&point1->X);
      bnCopy(&result->Y,&point1->Y);
}

res=bnSquareMod(&temp1,&point1->Z,&curve->field); //T1=Z1*Z1
res=bnMulMod(&temp2,&point2->X,&temp1,&curve->field); //T2=X2*T1
res=bnMulMod(&temp1,&point1->Z,&temp1,&curve->field); //T1=Z1*T1
res=bnMulMod(&temp1,&point2->Y,&temp1,&curve->field); //T1=Y2*T1

res=bnSubMod(&result->Y,&temp1,&curve->field); //Y=Y-T1
res=bnLShift(&temp1,1); //T1=2*T1 (+nächster Befehle)
res=bnMod(&temp1,&temp1,&curve->field);
res=bnAddMod(&temp1,&result->Y,&curve->field); //T1=T1+Y
res=bnSubMod(&result->X,&temp2,&curve->field); //X=X-T2
res=bnLShift(&temp2,1); //T2=2*T2 (+nächster Befehle)
res=bnMod(&temp2,&temp2,&curve->field);
res=bnAddMod(&temp2,&result->X,&curve->field); //T2=T2+X

if(!point2->Z_is_one) {
      res=bnMulMod(&result->Z,&point1->Z,&point2->Z,&curve->field); //Z=Z1*Z2
}
else if(result!=point1) {
      res=bnCopy(&result->Z,&point1->Z); //Z=Z1
}
res=bnMulMod(&result->Z,&result->Z,&result->X,&curve->field); //Z=Z*X

res=bnMulMod(&temp1,&temp1,&result->X,&curve->field); //T1=T1*X
res=bnSquareMod(&result->X,&result->X,&curve->field); //X=X*X
res=bnMulMod(&temp2,&temp2,&result->X,&curve->field); //T2=T2*X
res=bnMulMod(&temp1,&temp1,&result->X,&curve->field); //T1=T1*X

res=bnSquareMod(&result->X,&result->Y,&curve->field); //X=Y*Y
res=bnSubMod(&result->X,&temp2,&curve->field); //X=X-T2

res=bnSubMod(&temp2,&result->X,&curve->field); //T2=T2-X
res=bnSubMod(&temp2,&result->X,&curve->field); //T2=T2-X
res=bnMulMod(&temp2,&temp2,&result->Y,&curve->field); //T2=T2*Y
bnCopy(&result->Y,&temp2);
res=bnSubMod(&result->Y,&temp1,&curve->field); //Y=T2-T1

if(!(bnLSWord(&result->Y) & 0x0001)) { // wenn gerade
      bnRShift(&result->Y,1); //Y=Y/2
}
else {
      res=bnCopy(&temp1,&curve->field); //T1=P
      res=bnAddQ(&temp1,1); //T1=T1+1
      bnRShift(&temp1,1); //T1=T1/2 (T3 ist jetzt das Inverse von 2)
      res=bnMulMod(&result->Y,&result->Y,&temp1,&curve->field); //Y=Y*T1
}

result->Z_is_one = FALSE;

bnEnd(&temp1);
bnEnd(&temp2);
```

```
      return !res;
}

// Führt auf einer elliptische Kurve eine Punktverdopplung durch
int ecpoint_dbl(const eccurve *curve, ecpoint *result, ecpoint *point) {
      BigNum temp1, temp2, temp3;
      int res=0;

      if(result==NULL) {
            return 0;
      }

      if(ecpoint_isatinfinity(point)) {
            // Ein Punkt in der Unendlichkeit mal sich selber,
            // ergibt wieder sich selber: P*O=O also O*O=O
            bnSetQ(&result->Z,0);
            result->Z_is_one = FALSE;
            return 1;
      }

      if(result!=point) {
            ecpoint_copy(result,point);
      }

      bnBegin(&temp1);
      bnBegin(&temp2);
      bnBegin(&temp3);

      /* Implmentiert das Verfahren beschrieben in "A Small and Fast Software
       * Implementation of Elliptic Curve Cryptosystems over GF(p) on a 16-Bit
       * Microcomputer" von T. Hasegawa, J. Nakajima und M. Matsui erschienen
       * in IEICE Trans. Fundamentals, Vol. E82-A, No. 1 January 1999.
       */
      result->Z_is_one = FALSE;
      res=bnSquareMod(&temp1,&result->Z,&curve->field); //T1=Z*Z
      res=bnMulMod(&result->Z,&result->Y,&result->Z,&curve->field); //Z=Y*Z
      res=bnLShift(&result->Z,1); //Z=2*Z (+nächster Befehle)
      res=bnMod(&result->Z,&result->Z,&curve->field);

      if(curve->a_minus3) {
            res=bnCopy(&temp2,&result->X); //T2=X
            res=bnSubMod(&temp2,&temp1,&curve->field); //T2=T2-T1
            res=bnAddMod(&temp1,&result->X,&curve->field); //T1=T1+X
            res=bnMulMod(&temp2,&temp1,&temp2,&curve->field); //T2=T1*T2
            res=bnCopy(&temp1,&temp2); //T1=T2
            res=bnLShift(&temp1,1); //T1=2*T1 (+nächster Befehle)
            res=bnMod(&temp1,&temp1,&curve->field);
            res=bnAddMod(&temp1,&temp2,&curve->field); //T1=T1+T2
      }
      else {
            res=bnSquareMod(&temp1,&temp1,&curve->field); //T1=T1*T1
            res=bnMulMod(&temp1,&curve->a,&temp1,&curve->field); //T1=a*T1
            res=bnSquareMod(&temp2,&result->X,&curve->field); //T2=X*X
            res=bnAddMod(&temp1,&temp2,&curve->field); //T1=T1+T2
            res=bnLShift(&temp2,1); //T2=2*T2 (+nächster Befehle)
            res=bnMod(&temp2,&temp2,&curve->field);
            res=bnAddMod(&temp1,&temp2,&curve->field); //T1=T1+T2
      }

      res=bnLShift(&result->Y,1); //Y=2*Y (+nächster Befehle)
      res=bnMod(&result->Y,&result->Y,&curve->field);
      res=bnSquareMod(&result->Y,&result->Y,&curve->field); //Y=Y*Y
      res=bnSquareMod(&temp2,&result->Y,&curve->field); //T2=Y*Y


      if(!(bnLSWord(&temp2) & 0x0001)) { // wenn gerade
            bnRShift(&temp2,1); //T2=T2/2
      }
      else {
            res=bnCopy(&temp3,&curve->field); //T3=P
            res=bnAddQ(&temp3,1); //T3=T3+1
            bnRShift(&temp3,1); //T3=T3/2 (T3 ist jetzt das Inverse von 2)
            res=bnMulMod(&temp2,&temp2,&temp3,&curve->field); //T2=T2*T3
      }

      res=bnMulMod(&result->Y,&result->Y,&result->X,&curve->field); //Y=Y*X

      res=bnSquareMod(&result->X,&temp1,&curve->field); //X=T1*T1
      res=bnSubMod(&result->X,&result->Y,&curve->field); //X=X-Y
      res=bnSubMod(&result->X,&result->Y,&curve->field); //X=X-Y

      res=bnSubMod(&result->Y,&result->X,&curve->field); //Y=Y-X
      res=bnMulMod(&result->Y,&result->Y,&temp1,&curve->field); //Y=Y*T1
      res=bnSubMod(&result->Y,&temp2,&curve->field); //Y=Y-T2

      bnEnd(&temp1);
      bnEnd(&temp2);
      bnEnd(&temp3);
```

```
      return !res;
}

// Findet das an der x-Achse gespiegelte Negative eines Punktes
int ecpoint_invert(const eccurve *curve, ecpoint *point) {
      int res=0;

      if(ecpoint_isatinfinity(point) || !bnCmpQ(&point->Y,0))
            return 0; // der Punkt ist sein eigenes Inverses

      bnSub(&point->Y,&curve->field);     //Y=Field-Y

      return !res;
}

// Führt auf einer elliptischen Kurve eine Skalarmultiplikation durch
int ecpoint_mul(const eccurve *curve, ecpoint *result, ecpoint *point, BigNum *scalar) {
#define R 5 //spezifiziert die maximale Fenstergrösse (von Skalar zu Skalar verschieden)
                        // ist R=1 wird die Binary Methode verwendet
      int length, i;
#if R>1
      /* Implementiert die Sliding Window Methode zur Punktmultiplikation.
       * Algorithmus IV.4 in Elliptic Curves in Cryptography von
       * I. Blake, G. Seroussi und N. Smart.
       */

      int arrsize = 1<<R;     // berechnet die Grösse des Precompute-Arrays
      int j, t, h;
      ecpoint **Points;

      if(result==NULL) {
            return 0;
      }

      Points=malloc(arrsize*sizeof(Points[0]));  //der Array für die vorberechneten Punkte

      for(i=0;i<arrsize;i++) {
            Points[i]=NULL;
      }
      Points[1]=ecpoint_new();
      ecpoint_copy(Points[1],point); //P1 = P
      Points[2]=ecpoint_new();
      ecpoint_dbl(curve,Points[2],point); //P2 = 2*P
      for(i=2;i<arrsize-1;i+=2) {   //Punkte vorberechnen: P3=3*P, P5=5*P, ...
            Points[i+1]=ecpoint_new();
            ecpoint_add(curve,Points[i+1],Points[i-1],Points[2]);
      }

      bnNorm(scalar);
      length = bnBits(scalar);
      j=length-1;

      // Q = O    (in Unendlichkeit)
      bnSetQ(&result->Z,0);
      result->Z_is_one = FALSE;

      while(j>=0) {
            //testet, ob das i-te Bit nicht gesetzt ist (nur für 16Bit):
            if(!(((unsigned short *)scalar->ptr)[j/(sizeof(unsigned short)*8)]
                  & (0x0001 << (j%(sizeof(unsigned short)*8))))) {
                  ecpoint_dbl(curve,result,result);
                  j--;
            }
            else {
                  t=j-R+1; //Scanposition ist auf der rechten Seite des Fensters (j-R+1)
                  h=0;

                  //nach links Fenster verkleinern solange Bit an Position t = 0
                  while((j>=t) && !(((unsigned short *)scalar->ptr)[t/(sizeof(unsigned short)*8)]
                        & (0x0001 << (t%(sizeof(unsigned short)*8))))) {
                        t++;
                  }
                  //Fenster in h speichern
                  for(i=j;i>=t;i--) {
                        ecpoint_dbl(curve,result,result);
                        h = h<<1 | ((((unsigned short *)scalar->ptr)[i/(sizeof(unsigned short)*8)]
                              & (0x0001 << (i%(sizeof(unsigned short)*8))))?1:0);
                  }
                  if(Points[h]!=NULL) {
                        ecpoint_add(curve,result,result,Points[h]);
                  }
                  else {
                        printf("\nSpeicherfehler: Points[%d]",h);
                        return -1;
                  }

                  j=t-1;
            }
```

```
	}

	for(i=0;i<arrsize;i++) {
		if(Points[i]!=NULL)
			ecpoint_free(Points[i]);
	}
	free(Points);
#else
	/* Implementiert die binäre Methode zur Punktmultiplikation.
	 * Algorithmus IV.1 in Elliptic Curves in Cryptography von
	 * I. Blake, G. Seroussi und N. Smart.
	 */
	if(result==NULL) {
		return 0;
	}
	bnNorm(scalar);
	length = bnBits(scalar);

	// Q = O    (in Unendlichkeit)
	bnSetQ(&result->Z,0);
	result->Z_is_one = FALSE;

	for(i=length-1;i>=0;i--) {
		ecpoint_dbl(curve,result,result);
		//testet, ob das i-te Bit gesetzt ist (nur für 16Bit):
		if(((unsigned short *)scalar->ptr)[i/(sizeof(unsigned short)*8)]
			& (0x0001 << (i%(sizeof(unsigned short)*8)))) {
			ecpoint_add(curve,result,result,point);
		}
	}
#endif

	return 0;
}

// Alloziert Platz für eine neue Kurve
eccurve* eccurve_new(void) {
	eccurve *curve;
	curve = malloc(sizeof *curve);

	bnBegin(&curve->field);
	bnBegin(&curve->a);
	bnBegin(&curve->b);

	return curve;
}

// Gibt Platz einer Kurve frei
void eccurve_free(eccurve *curve) {
	if(curve == NULL) {
		return;
	}

	bnEnd(&curve->field);
	bnEnd(&curve->a);
	bnEnd(&curve->b);

	if(curve->generator != NULL) {
		ecpoint_free(curve->generator);
	}

	free(curve);
}
```

15.2.2 Testfunktion

```
/* DA Sna 02/3
 * Testsoftware für EC-Implementation
 */

#include <ec/ec.h>
#include <bignum/bn.h>
#include <bignum/bnprint.h>

// Konfiguration:
#define CURVE160 // Auswahl der Kurve
#define EMBEDDED // damit kann ein Test für die Embedded Umgebung realisiert werden


#ifdef EMBEDDED
#include <rtos.h>

#define TASK_STACKSIZE     5120 // WORDS => 10240 Bytes

static unsigned int mssl_stack[TASK_STACKSIZE];
```

```
static int mssl_id;

void huge mssl_task(void);

static TaskDefBlock  mssl_taskdefblock =
{
  mssl_task,
  "mssl",                        // a name: 4 chars
  &mssl_stack[TASK_STACKSIZE],  // top of stack
  TASK_STACKSIZE*sizeof(int),   // size of stack
  0,                             // attributes, not supported now
  0,                            // priority 20(high) ... 127(low)
  0,                             // time slice (if any), not supported now
  0,0,0,0                        // mailbox depth, not supported now
};

void main(void) {
             RTX_Create_Task(&mssl_id , &mssl_taskdefblock);
}



void huge mssl_task(void) {
             unsigned long start, stop;

#else
void main(void) {
#endif
             int i,bit;
             BigNum test1,test2,test3,prime,scalar;
             ecpoint *point, *point1, *point2, *point3;
             eccurve *curve;

#ifdef CURVE160
// Kurven-Paramter 160-Bit
             unsigned char a[] = {
                          0xFE, 0x57, 0xFF, 0xFF, 0xFF,
                          0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
                          0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
                          0xFF, 0xFF, 0xFF, 0xFF, 0xFC,
             };
             unsigned char b[] = {
                          0x6C, 0xDB, 0x99, 0x78, 0x27,
                          0x59, 0xFF, 0x69, 0x05, 0xC0,
                          0xCC, 0x9B, 0xCB, 0xC9, 0xC4,
                          0xFC, 0x13, 0x62, 0x6F, 0x76,
             };
             unsigned char p[] = {
                          0xFE, 0x57, 0xFF, 0xFF, 0xFF,
                          0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
                          0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
                          0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
             };
             unsigned char Px[] = {
                          0xA0, 0x6E, 0xA3, 0x1F, 0x4E,
                          0xB5, 0x51, 0x75, 0xD6, 0x38,
                          0x4F, 0x65, 0xC5, 0x4D, 0x4E,
                          0x24, 0x63, 0xCE, 0xA4, 0xDA,
             };
             unsigned char Py[] = {
                          0x19, 0x13, 0x78, 0x9F, 0x59,
                          0xFF, 0x19, 0x58, 0xF2, 0xFA,
                          0x6E, 0x63, 0x74, 0x5B, 0x87,
                          0x9C, 0x7D, 0x21, 0xE2, 0x5A,
             };
             unsigned char k[] = {
                          0x60, 0x1E, 0xDD, 0xB9, 0x30,
                          0xF4, 0x70, 0x4F, 0x39, 0x0B,
                          0x12, 0x2D, 0x9D, 0x5F, 0x2D,
                          0xE5, 0xA6, 0xE5, 0x25, 0xC1,
             };
#else
// Kurven-Paramter 192-Bit
             unsigned char a[] = {
                          0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
                          0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE,
                          0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC,
             };
             unsigned char b[] = {
                          0x64, 0x21, 0x05, 0x19, 0xE5, 0x9C, 0x80, 0xE7,
                          0x0F, 0xA7, 0xE9, 0xAB, 0x72, 0x24, 0x30, 0x49,
                          0xFE, 0xB8, 0xDE, 0xEC, 0xC1, 0x46, 0xB9, 0xB1,
             };
             unsigned char p[] = {
                          0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
                          0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE,
                          0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
             };
```

```
            unsigned char Px[] = {
                        0x18, 0x8D, 0xA8, 0x0E, 0xB0, 0x30, 0x90, 0xF6,
                        0x7C, 0xBF, 0x20, 0xEB, 0x43, 0xA1, 0x88, 0x00,
                        0xF4, 0xFF, 0x0A, 0xFD, 0x82, 0xFF, 0x10, 0x12,
            };
            unsigned char Py[] = {
                        0x07, 0x19, 0x2b, 0x95, 0xff, 0xc8, 0xda, 0x78,
                        0x63, 0x10, 0x11, 0xed, 0x6b, 0x24, 0xcd, 0xd5,
                        0x73, 0xf9, 0x77, 0xa1, 0x1e, 0x79, 0x48, 0x11,
            };
            unsigned char k[] = {
                        0x60, 0x1E, 0xDD, 0xB9, 0x30, 0xF4, 0x70, 0x4F,
                        0x39, 0x0B, 0x12, 0x2D, 0x9D, 0x5F, 0x2D, 0xE5,
                        0xA6, 0xE5, 0x25, 0xC1, 0xE5, 0x11, 0x45, 0xF5,
            };
#endif

            point = ecpoint_new();
            point1 = ecpoint_new();
            point2 = ecpoint_new();
            point3 = ecpoint_new();
            bnBegin(&scalar);
            curve = eccurve_new();
            curve->a_minus3=FALSE;

            bnInsertBigBytes(&point->X, Px, 0, sizeof(Px));
            bnInsertBigBytes(&point->Y, Py, 0, sizeof(Py));
```

```
bnInsertBigBytes(&curve->a, a, 0, sizeof(a));
bnInsertBigBytes(&curve->b, b, 0, sizeof(b));
bnInsertBigBytes(&curve->field, p, 0, sizeof(p));
bnInsertBigBytes(&scalar, k, 0, sizeof(k));
```

```
#ifdef EMBEDDED
            RTX_Get_System_Ticks(&start);
#endif
            ecpoint_dbl(curve,point1,point);
#ifdef EMBEDDED
            RTX_Get_System_Ticks(&stop);
            printf("\nPointDbl: %dms",stop-start);
#endif
            printf("\nPoint^2: %d",ecpoint_isoncurve(curve,point1));
            ecpoint_toaffine(curve,point1);
            bnPrint(stdout,"\nX:     ",&point1->X,"\n");
            bnPrint(stdout,"Y:     ",&point1->Y,"\n");
            bnPrint(stdout,"Z:     ",&point1->Z,"\n");

#ifdef EMBEDDED
            RTX_Get_System_Ticks(&start);
#endif
            ecpoint_add(curve,point1,point1,point);
#ifdef EMBEDDED
            RTX_Get_System_Ticks(&stop);
            printf("\nPointAdd: %dms",stop-start);
#endif
            printf("\nPoint^9: %d",ecpoint_isoncurve(curve,point1));
            ecpoint_toaffine(curve,point1);
            bnPrint(stdout,"\nX:     ",&point1->X,"\n");
            bnPrint(stdout,"Y:     ",&point1->Y,"\n");
            bnPrint(stdout,"Z:     ",&point1->Z,"\n");

            //bnSetQ(&scalar,5);
#ifdef EMBEDDED
            RTX_Get_System_Ticks(&start);
#endif
            ecpoint_mul(curve,point3,point,&scalar);
#ifdef EMBEDDED
            RTX_Get_System_Ticks(&stop);
            printf("\nPointMul: %dms",stop-start);
#endif
            ecpoint_toaffine(curve,point3);
            printf("\nPoint3: %d",ecpoint_isoncurve(curve,point3));
            bnPrint(stdout,"\nX2:    ",&point3->X,"\n");
            bnPrint(stdout,"Y2:    ",&point3->Y,"\n");
            bnPrint(stdout,"Z2:    ",&point3->Z,"\n");

            ecpoint_invert(curve,point3);
            printf("\nPoint3: %d",ecpoint_isoncurve(curve,point3));

            eccurve_free(curve);
            ecpoint_free(point);
            ecpoint_free(point1);
            ecpoint_free(point2);
            ecpoint_free(point3);
            bnEnd(&scalar);

#ifdef EMBEDDED
RTX_Delete_Task(mssl_id);
#endif
```

15.3 Software-Testresultate

Die Tests wurden in follgender Umgebung durchgeführt:

Der SSL_CLIENT ist der s_client von Openssl 0.9.8 und läuft auf einem PC unter Linux.

Der erweiterte mSSL_SERVER läuft auf dem Embedded-System mit einer RTOS.

Protokoliert wurden

- AECDH_RC4_SHA
- ECDH_RSA_RC4_SHA
- RSA_RSA_RC4_MD5

15.3.1 SSL_CLIENT: Anonymous ECDH mit RC4, SHA

```
[root@localhost root]# cd /usr/local/ssl/
[root@localhost ssl]# openssl s_client -connect 160.85.162.77:443 -ssl3 -cipher AECDH-RC4-SHA -debug
CONNECTED(00000003)
write to 081A3F18 [081AF8A0] (50 bytes => 50 (0x32))
0000 - 16 03 00 00 2d 01 00 00-29 03 00 3d ba cc 4f 24   ....-...)..=..O$
0010 - 60 fb e2 d1 94 66 06 2f-08 5a 36 62 42 15 0d 9b   `....f./.Z6bB...
0020 - 00 c8 2c a5 67 8d ba 41-16 56 ad 00 00 02 00 56   ..,.g..A.V.....V
0030 - 01                                                .
0032 - <SPACES/NULS>
read from 081A3F18 [081AB090] (5 bytes => 5 (0x5))
0000 - 16 03 00 00 4a                                    ....J
read from 081A3F18 [081AB095] (74 bytes => 74 (0x4A))
0000 - 02 00 00 46 03 00 34 03-61 4b a8 3a ce e0 92 9c   ...F..4.aK.:....
0010 - ff 03 be d3 c5 25 a2 ec-61 85 b1 ea 93 bf a0 5e   .....%..a......^
0020 - a9 79 1c 8a ed 16 20 00-00 00 00 00 00 00 00 00   .y.... .........
0030 - 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0040 - 00 00 00 00 00 00 01 00-56                        ........V
004a - <SPACES/NULS>
read from 081A3F18 [081AB090] (5 bytes => 5 (0x5))
0000 - 16 03 00 00 3c                                    ....<
read from 081A3F18 [081AB095] (60 bytes => 60 (0x3C))
0000 - 0c 00 00 34 03 13 31 04-83 c0 94 e7 29 be 24 b4   ...4..1.....).$.
0010 - 6b 3a 0e 0f c0 1a 73 4f-ba 3d 63 ca fb 0c cd 6f   k:....sO.=c....o
0020 - 1d 8e 88 d5 61 41 3c 97-56 93 0c ef 66 96 7f 22   ....aA<.V...f.."
0030 - 64 18 d2 f0 a8 a6 fb 35-0e                        d......5.
003c - <SPACES/NULS>
write to 081A3F18 [081B51B8] (59 bytes => 59 (0x3B))
0000 - 16 03 00 00 36 10 00 00-32 31 04 77 8d 6b f9 db   ....6...21.w.k..
0010 - d5 da 5a 7c 30 13 1a 5c-ab 90 13 42 70 6f fc b3   ..Z|0..\...Bpo..
0020 - 24 d4 4f bb b7 c8 d1 3c-cf 08 5f 31 ac cc de d5   $.O....<.._1....
0030 - 11 7d 0e 6d 06 3c fd e0-c2 3a fd                  .}.m.<...:.
write to 081A3F18 [081B51B8] (6 bytes => 6 (0x6))
0000 - 14 03 00 00 01 01                                 ......
write to 081A3F18 [081B51B8] (65 bytes => 65 (0x41))
0000 - 16 03 00 00 3c 8c b0 27-fd 46 71 1c 0c a4 e9 04   ....<..'.Fq.....
0010 - 48 9b fa 6e 5d 0f e7 b0-54 53 3c 21 60 39 27 ab   H..n]...TS<!`9'.
0020 - 13 e9 94 73 02 da be 72-01 97 c5 e5 93 b7 43 de   ...s...r......C.
0030 - c6 36 d1 b8 0b ee f6 3b-21 f4 cf ae 4a 9c 4f 2e   .6.....;!...J.O.
0040 - ac                                                .
read from 081A3F18 [081AB090] (5 bytes => 5 (0x5))
0000 - 14 03 00 00 01                                    .....
read from 081A3F18 [081AB095] (1 bytes => 1 (0x1))
0000 - 01                                                .
read from 081A3F18 [081AB090] (5 bytes => 5 (0x5))
0000 - 16 03 00 00 3c                                    ....<
read from 081A3F18 [081AB095] (60 bytes => 60 (0x3C))
0000 - 4c a3 41 5c 60 a7 cc 95-94 a4 f8 81 34 b0 6e c3   L.A\`.......4.n.
0010 - 7d 16 5e cf 8f 2c 65 28-f4 01 db f2 ee 46 40 95   }.^..,e(.....F@.
0020 - e6 ad 36 1e 10 9e af 4f-10 0a 1d 92 9e 8a d1 ad   ..6....O........
0030 - e6 50 38 6d 61 11 ef 2e-e7 c3 50 b1               .P8ma.....P.
---
no peer certificate available
---
No client certificate CA names sent
```

```
---
SSL handshake has read 215 bytes and written 180 bytes
---
New, TLSv1/SSLv3, Cipher is AECDH-RC4-SHA
SSL-Session:
    Protocol  : SSLv3
    Cipher    : AECDH-RC4-SHA
    Session-ID: 0000000000000000000000000000000000000000000000000000000000000001
    Session-ID-ctx:
    Master-Key:
4B550D21ABF42E1C40FC38E550D06369EEDD4DA05A236FE12E1762AF5E509CBD3DE6D14DBEDA2EAA07F6F2CBDBD2315B
    Key-Arg   : None
    Start Time: 1035652175
    Timeout   : 7200 (sec)
    Verify return code: 0 (ok)
---
```

15.3.2 mSSL_SERVER: Anonymous ECDH mit RC4, SHA

```
[root@localhost root]# telnet 160.85.162.77
Trying 160.85.162.77...
Connected to 160.85.162.77.
Escape character is '^]'.

SC12 Telnet session

Username: tel
Password: ***
User logged in
A:\>msslnd

A:\>
mssl v0.1 started

Init EXT with 8-O-1 and 2400 baud
Baud divider: 521

Using RTS/CTS flow control

NQC: opcode: 71 00
MSSL: read file '\s_cert.der' (938 bytes)
MSSL: read file '\s_key.hst' (1166 bytes)
DH-Params generated
MSSL-sub:
dh.p:F5D42B6911098F081CC0D316E9689FB2E86C6B327E7EC543E6B393DF69F427A1353E868F44928A635507664C6D63C8EA
967EE41FBE267F5103821E67FE806403
MSSL-sub: dh.g:2
MSSL-sub: dh.priv_key:13F0BA025019FD2EC62EE7E016FF0076C01AE22D0945348B
MSSL-sub:
dh.pub_key:C8C745D58283D37B72998959C13F07B1AF0745D4A75EC4DA36198CFE74BC41D80DEDCE704BEFFD0E45C40111C4
1E056EBD23F562766B43C13FB308E76A12082B
MSSL: read file '\ec_cert.der' (782 bytes)
MSSL: read file '\ec_key.hst' (49 bytes)
MSSL: ec_private_key readed from file: 44 45 33 44 41 44 45 35 33 30 35 30 43 42 41 36 36 46 32 33 42
43 31 38 32 34 32 37 46 35 39 41 33 44 46 37 42 38 39 44 32 45 35 41 41 41 41 33
MSSL-sub: creating ec.pub_key..

MSSL-sub:  Time for ECDH-Key-generation [ms]: 9193
MSSL-sub: ec_priv_key eingelesen [BIGNUM]:B96335414D131E095F5ED8741B57C3E6714EA03D6149646D
MSSL-sub: ec.pri_key:
B96335414D131E095F5ED8741B57C3E6714EA03D6149646D
MSSL-sub: ec.pub_key:
83C094E729BE24B46B3A0E0FC01A734FBA3D63CAFB0CCD6F,
1D8E88D561413C9756930CEF66967F226418D2F0A8A6FB35
MSSL: socket open successful
MSSL: socket bind successful
MSSL: socket listen successful
MSSL: waiting for connection..
MSSL: connection from 160.85.162.78:772
MSSL-sub: fetched next Paket
MSSL-sub: read one RECORD from the received TLS-Paket
MSSL-sub:protocol = 22 version SSLv3 length = 45
MSSL-sub: It's a TLS_HANDSHAKE
MSSL-sub: handshake = 1 length = 41
MSSL: ClientHelloRecord: 03 00 3D BA CC 4F 24 60 FB E2 D1 94 66 06 2F 08 5A 36 62 42 15 0D 9B 00 C8
2C A5 67 8D BA 41 16 56 AD 00 00 02 00 56 01 00
MSSL-sub: cipher suite to compare: 86 ECDH anon / 128 bit RC4 / SHA
MSSL-sub: cipher suite: ECDH anon / 128 bit RC4 / SHA
MSSL: server hello msg
02 00 00 46 03 00 34 03 61 4B A8 3A CE E0 92 9C FF 03 BE D3 C5 25 A2 EC 61 85 B1 EA 93 BF A0 5E A9 79
1C 8A ED 16 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 01 00 56 00
MSSL-sub: send server_hello_msg
MSSL-sub: Session is new (not resumed)
MSSL-sub: server_key_exchange_msg with Curve_Type,Curve_id,encodelen,ECDH.pub_key
```

```
  msg-len: 56
MSSL: dump of server_key_exchange_msg:
0C 00 00 34 03 13 31 04 83 C0 94 E7 29 BE 24 B4 6B 3A 0E 0F C0 1A 73 4F BA 3D 63 CA FB 0C CD 6F 1D 8E
88 D5 61 41 3C 97 56 93 0C EF 66 96 7F 22 64 18 D2 F0 A8 A6 FB 35
MSSL-sub: send server_hello_done_msg
    =>TLS_CLIENT_KEY_EXCHANGE
MSSL-sub: fetched next Paket
MSSL-sub: read one RECORD from the received TLS-Paket
MSSL-sub:protocol = 22 version SSLv3 length = 54
MSSL-sub: It's a TLS_HANDSHAKE
MSSL-sub: handshake = 16 length = 50
MSSL: beginning of client_key_exchange (ECDH) ...client_key_exchange message-len: 50
MSSL-sub: client_key_exchange.encoded_point:
04 77 8D 6B F9 DB D5 DA 5A 7C 30 13 1A 5C AB 90 13 42 70 6F FC B3 24 D4 4F BB B7 C8 D1 3C CF 08 5F 31
AC CC DE D5 11 7D 0E 6D 06 3C FD E0 C2 3A FD
MSSL-sub:  Time for ECDH-Key-computation [ms]: 9364
MSSL-sub: alen(ECDH_size of ec_key ist immer 20): 20
MSSL-sub: länge des computed_ec_key in DEZ und HEX : 20    14
MSSL: ECDH: g^Xc^Xs succeeded (see premaster_secret):

MSSL: ECDH-common-key:
09 C3 CD EB 3C 24 D3 F7 61 C4 42 8F 4A 26 50 87 86 B4 7C A0
MSSL-sub: =>Client_Key_Exchange processed, =>TLS_CLIENT_CHANGE_CIPHER_SPEC
MSSL-sub: fetched next Paket
MSSL-sub: read one RECORD from the received TLS-Paket
MSSL-sub:protocol = 20 version SSLv3 length = 1
MSSL-sub: =>use new
 Client_Cipher_Suite from now:
MSSL-sub-specs:   session_id 1
MSSL-sub-specs: client_random  3D BA CC 4F 24 60 FB E2 D1 94 66 06 2F 08 5A 36 62 42 15 0D 9B 00 C8
2C A5 67 8D BA 41 16 56 AD
MSSL-sub-specs: server-random  34 03 61 4B A8 3A CE E0 92 9C FF 03 BE D3 C5 25 A2 EC 61 85 B1 EA 93
BF A0 5E A9 79 1C 8A ED 16
MSSL-sub-specs: master_secret  4B 55 0D 21 AB F4 2E 1C 40 FC 38 E5 50 D0 63 69 EE DD 4D A0 5A 23 6F
E1 2E 17 62 AF 5E 50 9C BD 3D E6 D1 4D BE DA 2E AA 07 F6 F2 CB DB D2 31 5B
MSSL-sub-specs: secret_keys  52 1B B1 DD 02 12 CA 1B E2 8A A3 C6 21 F8 4E 21 B7 30 71 E2 BD 78 26 B5
35 2E 98 62 2E 2C 55 4D 84 90 9E 27 7D BD B7 49 5F E8 45 73 9F F2 A0 6E 66 DB 59 78 7D 22 F7 05 F8 6B
88 D7 50 01 EC 60 1A B4 CD 60 DC 31 1C D2 2B 63 16 5C 99 A0 46 1E F1 5C 21 30 C5 34 01 BB 76 7F 64 9E
11 9C C4 C9
MSSL-sub-specs: client_MAC_secret  52 1B B1 DD 02 12 CA 1B E2 8A A3 C6 21 F8 4E 21 B7 30 71 E2
MSSL-sub-specs: server_MAC_secret  BD 78 26 B5 35 2E 98 62 2E 2C 55 4D 84 90 9E 27 7D BD B7 49
=>TLS_CLIENT_FINISHED
MSSL-sub: read one RECORD from the received TLS-Paket
MSSL-sub:protocol = 22 version SSLv3 length = 60
MSSL: SHA MAC: F8 08 3B E5 22 51 A1 BA E9 82 F8 44 BD 3F 7D 39 3F C1 0F C3
MSSL-sub: It's a TLS_HANDSHAKE
MSSL-sub: handshake = 20 length = 36
MSSL: handshake = 20 length = 36
MSSL-sub: process client_finished_msg
MSSL-sub: =>client_finished processed, =>TLS_ACCEPT
MSSL-sub: send change_cipher_spec
MSSL-sub: send server_finished_msg
MSSL-sub: =>new session, so take new specs,

MSSL-sub:=>TLS_ACCEPT
```

15.3.3 SSL_CLIENT: ECDH,RSA mit RC4, SHA

[root@localhost ssl]# openssl s_client -CAfile ./demoCA/cacert.pem -connect 160.85.162.77:443 -ssl3 -cipher ECDH-RSA-RC4-SHA -debug

```
CONNECTED(00000003)
write to 081A3F18 [081AF8A0] (50 bytes => 50 (0x32))
0000 - 16 03 00 00 2d 01 00 00-29 03 00 3d ba cf 99 20   ....-...)..=...
0010 - a2 a4 a9 f7 2e d4 5b e4-6f 31 9e bf 1a 2f ee f0   ......[.o1.../..
0020 - 45 36 95 53 89 e3 66 1f-78 25 a2 00 00 02 00 4e   E6.S..f.x%.....N
0030 - 01                                                .
0032 - <SPACES/NULS>
read from 081A3F18 [081AB090] (5 bytes => 5 (0x5))
0000 - 16 03 00 00 4a                                    ....J
read from 081A3F18 [081AB095] (74 bytes => 74 (0x4A))
0000 - 02 00 00 46 03 00 34 03-61 4b a8 3a ce e0 92 9c   ...F..4.aK.:....
0010 - ff 03 be d3 c5 25 a2 ec-61 85 b1 ea 93 bf a0 5e   .....%..a......^
0020 - a9 79 1c 8a ed 16 20 00-00 00 00 00 00 00 00 00   .y.... .........
0030 - 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0040 - 00 00 00 00 00 00 01 00-4e                        ........N
004a - <SPACES/NULS>
read from 081A3F18 [081AB090] (5 bytes => 5 (0x5))
0000 - 16 03 00 03 1c                                    .....
read from 081A3F18 [081AB095] (796 bytes => 796 (0x31C))
0000 - 0b 00 03 14 00 03 11 00-03 0e 30 82 03 0a 30 82   ..........0...0.
0010 - 01 f2 a0 03 02 01 02 02-01 01 30 0d 06 09 2a 86   ..........0...*.
0020 - 48 86 f7 0d 01 01 04 05-00 30 4f 31 0b 30 09 06   H........0O1.0..
0030 - 03 55 04 06 13 02 43 48-31 0b 30 09 06 03 55 04   .U....CH1.0...U.
0040 - 08 13 02 5a 48 31 0c 30-0a 06 03 55 04 0a 13 03   ...ZH1.0...U....
0050 - 5a 48 57 31 25 30 23 06-03 55 04 03 13 1c 45 43   ZHW1%0#..U....EC
```

```
0060 - 43 2d 44 41 20 43 65 72-74 69 66 69 63 61 74 65   C-DA Certificate
0070 - 20 41 75 74 68 6f 72 69-74 79 30 1e 17 0d 30 32    Authority0...02
0080 - 31 30 31 30 31 31 30 36-32 37 5a 17 0d 30 33 31   1010110627Z..031
0090 - 30 31 30 31 31 30 36 32-37 5a 30 46 31 0b 30 09   010110627Z0F1.0.
00a0 - 06 03 55 04 06 13 02 43-48 31 0b 30 09 06 03 55   ..U....CH1.0...U
00b0 - 04 08 13 02 5a 48 31 0c-30 0a 06 03 55 04 0a 13   ....ZH1.0...U...
00c0 - 03 5a 48 57 31 1c 30 1a-06 03 55 04 03 13 13 70   .ZHW1.0...U....p
00d0 - 79 67 6d 79 2e 73 74 72-6f 6e 67 73 65 63 2e 63   ygmy.strongsec.c
00e0 - 6f 6d 30 49 30 13 06 07-2a 86 48 ce 3d 02 01 06   om0I0...*.H.=...
00f0 - 08 2a 86 48 ce 3d 03 01-01 03 32 00 04 8d c8 e5   .*.H.=....2.....
0100 - 8a 4e 6c 1d 6d 61 43 98-d3 14 c2 7d 62 b2 b0 b9   .Nl.maC....}b...
0110 - 10 dc 29 f9 99 a4 5b 95-cc ac ae 50 48 1c 45 b1   ..)...[....PH.E.
0120 - b4 c9 3e 2f e5 e9 e0 ca-e3 8d a5 9e 85 a3 81 d4   ..>/............
0130 - 30 81 d1 30 09 06 03 55-1d 13 04 02 30 00 30 2c   0..0...U....0.0,
0140 - 06 09 60 86 48 01 86 f8-42 01 0d 04 1f 16 1d 4f   ..`.H...B......O
0150 - 70 65 6e 53 53 4c 20 47-65 6e 65 72 61 74 65 64   penSSL Generated
0160 - 20 43 65 72 74 69 66 69-63 61 74 65 30 1d 06 03    Certificate0...
0170 - 55 1d 0e 04 16 04 14 82-4d 2e 5f 14 60 5c c1 0e   U.......M._.`\..
0180 - 28 57 0e 1d ef ca 1d 1d-ab 4b 75 30 77 06 03 55   (W.......Ku0w..U
0190 - 1d 23 04 70 30 6e 80 14-30 db 6e 58 f2 1f 1b ce   .#.p0n..0.nX....
01a0 - 40 aa c8 f6 18 bf d5 b3-ec 21 89 45 a1 53 a4 51   @........!.E.S.Q
01b0 - 30 4f 31 0b 30 09 06 03-55 04 06 13 02 43 48 31   0O1.0...U....CH1
01c0 - 0b 30 09 06 03 55 04 08-13 02 5a 48 31 0c 30 0a   .0...U....ZH1.0.
01d0 - 06 03 55 04 0a 13 03 5a-48 57 31 25 30 23 06 03   ..U....ZHW1%0#..
01e0 - 55 04 03 13 1c 45 43 43-2d 44 41 20 43 65 72 74   U....ECC-DA Cert
01f0 - 69 66 69 63 61 74 65 20-41 75 74 68 6f 72 69 74   ificate Authorit
0200 - 79 82 01 00 30 0d 06 09-2a 86 48 86 f7 0d 01 01   y...0...*.H.....
0210 - 04 05 00 03 82 01 01 00-98 18 de dc a0 df f8 7e   ...............~
0220 - 77 85 4b 7e 4c 03 f4 23-dc 44 f4 d5 54 1a 0b 2d   w.K~L..#.D..T..-
0230 - 8d e1 e1 f4 f1 eb a9 68-f2 9c 7a 87 0e 3f 70 74   .......h..z..?pt
0240 - d0 de dd a3 6e cd c7 4d-0b f0 a7 bc 46 19 3a c6   ....n..M....F.:.
0250 - 56 a9 69 8e 4b 5b fa c5-94 cf 09 23 7a 3a e5 1d   V.i.K[.....#z:..
0260 - 9e 76 2c 62 e3 12 29 23-c2 50 c3 70 eb 6d 0e 86   .v,b..)#.P.p.m..
0270 - 36 bd a9 2b 1f f2 5b e6-e8 e6 34 95 c5 68 ad 66   6..+..[...4..h.f
0280 - 1a 4d 97 f9 c5 dd a4 5a-df db c9 94 fc d3 12 6c   .M.....Z.......l
0290 - 9e 68 6b 0d 54 d3 7c aa-86 0c 11 c6 63 36 32 fb   .hk.T.|.....c62.
02a0 - 4c 01 da f0 f8 4d 41 1f-5a 33 02 63 11 30 fe c4   L....MA.Z3.c.0..
02b0 - e6 f9 d9 1d 12 57 48 1b-dc f9 bd 45 eb 1d c4 44   .....WH....E...D
02c0 - 3f f3 67 3c 83 ce e6 60-e1 ae 23 5b 22 2e c7 11   ?.g<...`..#["...
02d0 - 1d 63 f1 26 e6 74 61 0e-a1 42 75 5a 1e 44 57 80   .c.&.ta..BuZ.DW.
02e0 - ef 99 2c d8 26 fb 6d cd-f5 a6 1f 2f 8a e4 69 81   ..,.&.m..../..i.
02f0 - 40 37 38 16 76 61 72 df-39 80 ca 04 8d a1 14 c0   @78.var.9.......
0300 - 7b 5a bb de 19 b4 da 97-78 0c b7 22 f1 3e ed 30   {Z......x..".>.0
0310 - 22 2a f3 5b d4 ba 47 00-0e                        "*.[..G..
031c - <SPACES/NULS>
depth=1 /C=CH/ST=ZH/O=ZHW/CN=ECC-DA Certificate Authority
verify return:1
depth=0 /C=CH/ST=ZH/O=ZHW/CN=pygmy.strongsec.com
verify return:1
write to 081A3F18 [081B51B8] (59 bytes => 59 (0x3B))
0000 - 16 03 00 00 36 10 00 00-32 31 04 86 a1 66 8d 57   ....6...21...f.W
0010 - 33 0e 09 dc 73 d1 22 8c-18 29 c9 b7 e4 21 41 d9   3...s."..)...!A.
0020 - ed 9a 3f 15 19 4e 31 84-91 14 97 c7 1f b3 5b 1f   ..?..N1.......[.
0030 - 3e dd 90 10 34 0a f9 8c-10 3e 81                  >...4....>.
write to 081A3F18 [081B51B8] (6 bytes => 6 (0x6))
0000 - 14 03 00 00 01 01                                 ......
write to 081A3F18 [081B51B8] (65 bytes => 65 (0x41))
0000 - 16 03 00 00 3c 13 15 55-f9 5d e9 77 27 5d 1b dc   ....<..U.].w']..
0010 - 1b 19 06 8e 09 ef 80 be-c5 dc d5 56 c9 cc 8a b8   ...........V....
0020 - 21 1b fc dc c5 88 47 d5-76 6c 9f 98 1d 5e 65 5f   !.....G.vl...^e_
0030 - f8 97 d9 af f0 ff e4 6c-c9 6e a0 1d 66 36 f5 27   .......l.n..f6.'
0040 - eb                                                .
read from 081A3F18 [081AB090] (5 bytes => 5 (0x5))
0000 - 14 03 00 00 01                                    .....
read from 081A3F18 [081AB095] (1 bytes => 1 (0x1))
0000 - 01                                                .
read from 081A3F18 [081AB090] (5 bytes => 5 (0x5))
0000 - 16 03 00 00 3c                                    ....<
read from 081A3F18 [081AB095] (60 bytes => 60 (0x3C))
0000 - dd ce 70 b9 d5 16 fa cb-46 ed 7f 0f e5 fe 02 e5   ..p.....F.......
0010 - 5a 73 1c 02 aa bc 22 90-d1 bb 0a d1 4f c0 14 3b   Zs....".....O..;
0020 - 83 90 0d 70 12 ca eb 96-5c 4c 38 8a 25 6e 78 35   ...p....\L8.%nx5
0030 - 8e cb 6a 35 3c 06 cf 3a-a7 c8 c6 7d               ..j5<..:...}
---
Certificate chain
 0 s:/C=CH/ST=ZH/O=ZHW/CN=pygmy.strongsec.com
   i:/C=CH/ST=ZH/O=ZHW/CN=ECC-DA Certificate Authority
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIDCjCCAfKgAwIBAgIBATANBgkqhkiG9w0BAQQFADBPMQswCQYDVQQGEwJDSDEL
MAkGA1UECBMCWkgxDDAKBgNVBAoTA1pIVzElMCMGA1UEAxMcRUNDLURBIENlcnRp
ZmljYXRlIEF1dGhvcml0eTAeFw0wMjEwMTAxMTA2MjdaFw0wMzEwMTAxMTA2Mjda
MEYxCzAJBgNVBAYTAkNIMQswCQYDVQQIEwJaSDEMMAoGA1UEChMDWkhXMRwwGgYD
VQQDExNweWdteS5zdHJvbmdzZWMuY29tMEkwEwYHKoZIzj0CAQYIKoZIzj0DAQED
MgAEjcjlik5sHW1hQ5jTFMJ9YrKwuRDcKfmZpFuVzKyuUEgcRbG0yT4v5engyuON
pZ6Fo4HUMIHRMAkGA1UdEwQCMAAwLAYJYIZIAYb4QgENBB8WHU9wZW5TU0wgR2Vu
ZXJhdGVkIENlcnRpZmljYXRlMB0GA1UdDgQWBBSCTS5fFGBcwQ4oVw4d78odHatL
```

```
dTB3BgNVHSMEcDBugBQw225Y8h8bzkCqyPYYv9Wz7CGJRaFTpFEwTzELMAkGA1UE
BhMCQ0gxCzAJBgNVBAgTAlpIMQwwCgYDVQQKEwNaSFcxJTAjBgNVBAMTHEVDQy1E
QSBDZXJ0aWZpY2F0ZSBBdXRob3JpdHmCAQAwDQYJKoZIhvcNAQEEBQADggEBAJgY
3tyg3/h+d4VLfkwD9CPcRPTVVBoLLY3h4fTx66lo8px6hw4/cHTQ3t2jbs3HTQvw
p7xGGTrGVqlpjktb+sWUzwkjejrlHZ52LGLjEikjwlDDcOttDoY2vakrH/Jb5ujm
NJXFaK1mGk2X+cXdpFrf28mU/NMSbJ5oaw1U03yqhgwRxmM2MvtMAdrw+E1BH1oz
AmMRMP7E5vnZHRJXSBvc+b1F6x3ERD/zZzyDzuZg4a4jWyIuxxEdY/Em5nRhDqFC
dVoeRFeA75ks2Cb7bc31ph8viuRpgUA3OBZ2YXLfOYDKBI2hFMB7WrveGbTal3gM
tyLxPu0wIirzW9S6RwA=
```


```
-----END CERTIFICATE-----
subject=/C=CH/ST=ZH/O=ZHW/CN=pygmy.strongsec.com
issuer=/C=CH/ST=ZH/O=ZHW/CN=ECC-DA Certificate Authority
---
No client certificate CA names sent
---
SSL handshake has read 951 bytes and written 180 bytes
---
New, TLSv1/SSLv3, Cipher is ECDH-RSA-RC4-SHA
Server public key is 192 bit
SSL-Session:
    Protocol  : SSLv3
    Cipher    : ECDH-RSA-RC4-SHA
    Session-ID: 0000000000000000000000000000000000000000000000000000000000000001

    Session-ID-ctx:
 Master-Key:
94F8CACB9F14BA276774BFD49EF166C2766FBFB266D5F96028DE79F58ED990B8C69AEC47A08FA2C7EB0F3A0E0CE3679C
 Key-Arg   : None
 Start Time: 1035653017
 Timeout   : 7200 (sec)
 Verify return code: 0 (ok)
---
```

15.3.4 mSSL_SERVER: ECDH,RSA mit RC4, SHA

```
[root@localhost root]# telnet 160.85.162.77
Trying 160.85.162.77...
Connected to 160.85.162.77.
Escape character is '^]'.

SC12 Telnet session

Username: tel
Password: ***
msslnd

A:\>
mssl v0.1 started

Init EXT with 8-O-1 and 2400 baud
Baud divider: 521

Using RTS/CTS flow control

NQC: opcode: 71 00
MSSL: read file '\s_cert.der' (938 bytes)
MSSL: read file '\s_key.hst' (1166 bytes)
 sdh: alle var deklariert
 sdh: dh_new()...
 sdh: p,g,priv_key: chunk to bignum ende.
DH-Params generated
MSSL-sub:
dh.p:F5D42B6911098F081CC0D316E9689FB2E86C6B327E7EC543E6B393DF69F427A1353E868F44928A635507664C6D63C8EA
967EE41FBE267F5103821E67FE806403
MSSL-sub: dh.g:2
MSSL-sub: dh.priv_key:13F0BA025019FD2EC62EE7E016FF0076C01AE22D0945348B
MSSL-sub:
dh.pub_key:C8C745D58283D37B72998959C13F07B1AF0745D4A75EC4DA36198CFE74BC41D80DEDCE704BEFFD0E45C40111C4
1E056EBD23F562766B43C13FB308E76A12082B
MSSL: read file '\ec_cert.der' (782 bytes)
MSSL: read file '\ec_key.hst' (49 bytes)
MSSL: ec_private_key readed from file: 44 45 33 44 41 44 45 35 33 30 35 30 43 42 41 36 36 46 32 33 42
43 31 38 32 34 32 37 46 35 39 41 33 44 46 37 42 38 39 44 32 45 35 41 41 41 41 33
MSSL-sub: creating ec.pub_key..

MSSL-sub:  Time for ECDH-Key-generation [ms]: 9106
MSSL-sub: ec_priv_key eingelesen [BIGNUM]:DE3DADE53050CBA66F23BC182427F59A3DF7B89D2E5AAAA3
MSSL-sub: ec.pri_key:
DE3DADE53050CBA66F23BC182427F59A3DF7B89D2E5AAAA3
MSSL-sub: ec.pub_key:
8DC8E58A4E6C1D6D614398D314C27D62B2B0B910DC29F999,
A45B95CCACAE50481C45B1B4C93E2FE5E9E0CAE38DA59E85
MSSL: socket open successful
MSSL: socket bind successful
MSSL: socket listen successful
MSSL: waiting for connection..
MSSL: connection from 160.85.162.78:1540
MSSL-sub: fetched next Paket
MSSL-sub: read one RECORD from the received TLS-Paket
MSSL-sub:protocol = 22 version SSLv3 length = 45
MSSL-sub: It's a TLS_HANDSHAKE
MSSL-sub: handshake = 1 length = 41
MSSL: ClientHelloRecord: 03 00 3D BA CF 99 20 A2 A4 A9 F7 2E D4 5B E4 6F 31 9E BF 1A 2F EE F0 45 36
95 53 89 E3 66 1F 78 25 A2 00 00 02 00 4E 01 00
MSSL-sub: cipher suite to compare: 78 ECDH RSA / 128 bit RC4 / SHA
MSSL-sub: cipher suite: ECDH RSA / 128 bit RC4 / SHA
MSSL: server hello msg
02 00 00 46 03 00 34 03 61 4B A8 3A CE E0 92 9C FF 03 BE D3 C5 25 A2 EC 61 85 B1 EA 93 BF A0 5E A9 79
1C 8A ED 16 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 01 00 4E 00
MSSL-sub: send server_hello_msg
MSSL-sub: Session is new (not resumed)
MSSL-sub: send certificate_msg
MSSL-sub: send server_hello_done_msg
    =>TLS_CLIENT_KEY_EXCHANGE
MSSL-sub: fetched next Paket
MSSL-sub: read one RECORD from the received TLS-Paket
MSSL-sub:protocol = 22 version SSLv3 length = 54
MSSL-sub: It's a TLS_HANDSHAKE
MSSL-sub: handshake = 16 length = 50
MSSL: beginning of client_key_exchange (ECDH) ...client_key_exchange message-len: 50
MSSL-sub: client_key_exchange.encoded_point:
04 86 A1 66 8D 57 33 0E 09 DC 73 D1 22 8C 18 29 C9 B7 E4 21 41 D9 ED 9A 3F 15 19 4E 31 84 91 14 97 C7
1F B3 5B 1F 3E DD 90 10 34 0A F9 8C 10 3E 81
MSSL-sub:  Time for ECDH-Key-computation [ms]: 9299
MSSL-sub: alen(ECDH_size of ec_key ist immer 20): 20
MSSL-sub: länge des computed_ec_key in DEZ und HEX : 20    14
MSSL: ECDH: g^Xc^Xs succeeded (see premaster_secret):
```

```
MSSL: ECDH-common-key:
1F C4 CA EB 9D D3 83 43 B3 08 80 F2 4D F7 A7 4D 07 81 E4 E8
MSSL-sub: =>Client_Key_Exchange processed, =>TLS_CLIENT_CHANGE_CIPHER_SPEC
MSSL-sub: fetched next Paket
MSSL-sub: read one RECORD from the received TLS-Paket
MSSL-sub:protocol = 20 version SSLv3 length = 1
MSSL-sub: =>use new Client_Cipher_Suite from now:
MSSL-sub-specs:   session_id 1
MSSL-sub-specs: client_random  3D BA CF 99 20 A2 A4 A9 F7 2E D4 5B E4 6F 31 9E BF 1A 2F EE F0 45 36
95 53 89 E3 66 1F 78 25 A2
MSSL-sub-specs: server-random  34 03 61 4B A8 3A CE E0 92 9C FF 03 BE D3 C5 25 A2 EC 61 85 B1 EA 93
BF A0 5E A9 79 1C 8A ED 16
MSSL-sub-specs: master_secret  94 F8 CA CB 9F 14 BA 27 67 74 BF D4 9E F1 66 C2 76 6F BF B2 66 D5 F9
60 28 DE 79 F5 8E D9 90 B8 C6 9A EC 47 A0 8F A2 C7 EB 0F 3A 0E 0C E3 67 9C
MSSL-sub-specs: secret_keys  76 21 1C EF BD B9 75 AC A1 7E 1C E8 1B E4 38 3F ED 0E B0 4F A7 4F 86 B7
CA 98 D0 BC 53 2A 77 DA 51 F9 18 B2 C1 02 E6 A7 37 F0 55 0E F6 13 A1 83 49 33 DB D1 84 E9 43 95 65 69
B2 19 02 31 65 19 DA 2E D9 C1 3D 08 D6 C6 D1 4E BA 49 4A EB 8A FA 98 B9 32 8F D8 3D 89 CC D6 2B FB 02
C0 96 07 4C
MSSL-sub-specs: client_MAC_secret  76 21 1C EF BD B9 75 AC A1 7E 1C E8 1B E4 38 3F ED 0E B0 4F
MSSL-sub-specs: server_MAC_secret  A7 4F 86 B7 CA 98 D0 BC 53 2A 77 DA 51 F9 18 B2 C1 02 E6 A7
=>TLS_CLIENT_FINISHED
MSSL-sub: read one RECORD from the received TLS-Paket
MSSL-sub:protocol = 22 version SSLv3 length = 60
MSSL: SHA MAC: BC D8 9E BD CE E0 3F C2 15 BD 5A EA A1 40 A2 29 8E D7 6A 0A
MSSL-sub: It's a TLS_HANDSHAKE
MSSL-sub: handshake = 20 length = 36
MSSL: handshake = 20 length = 36
MSSL-sub: process client_finished_msg
MSSL-sub: =>client_finished processed, =>TLS_ACCEPT
MSSL-sub: send change_cipher_spec
MSSL-sub: send server_finished_msg
MSSL-sub: =>new session, so take new specs,

MSSL-sub:=>TLS_ACCEPT
```

15.3.5 SSL_CLIENT: RSA,RSA mit RC4, MD5

```
[root@localhost ssl]# openssl s_client -connect 160.85.162.77:443 -ssl3 -cipher RC4-MD5 -debug
CONNECTED(00000003)
write to 081A3F18 [081AF8A0] (50 bytes => 50 (0x32))
0000 - 16 03 00 00 2d 01 00 00-29 03 00 3d ba d2 e8 f2   ....-...)..=....
0010 - 11 49 d5 c2 6f 4b 14 8a-9a d5 d6 05 48 06 af fe   .I..oK......H...
0020 - be 9c a8 72 97 aa dd 19-d0 9c f5 00 00 02 00 04   ...r............
0030 - 01                                                .
0032 - <SPACES/NULS>
read from 081A3F18 [081AB090] (5 bytes => 5 (0x5))
0000 - 16 03 00 00 4a                                    ....J
read from 081A3F18 [081AB095] (74 bytes => 74 (0x4A))
0000 - 02 00 00 46 03 00 34 03-61 4b a8 3a ce e0 92 9c   ...F..4.aK.:....
0010 - ff 03 be d3 c5 25 a2 ec-61 85 b1 ea 93 bf a0 5e   .....%..a......^
0020 - a9 79 1c 8a ed 16 20 00-00 00 00 00 00 00 00 00   .y.... .........
0030 - 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0040 - 00 00 00 00 00 00 01 00-04                        .........
004a - <SPACES/NULS>
read from 081A3F18 [081AB090] (5 bytes => 5 (0x5))
0000 - 16 03 00 03 b8                                    .....
read from 081A3F18 [081AB095] (952 bytes => 952 (0x3B8))
0000 - 0b 00 03 b0 00 03 ad 00-03 aa 30 82 03 a6 30 82   ..........0...0.
0010 - 02 8e a0 03 02 01 02 02-01 32 30 0d 06 09 2a 86   .........20...*.
0020 - 48 86 f7 0d 01 01 04 05-00 30 42 31 0b 30 09 06  H........0B1.0..
0030 - 03 55 04 06 13 02 43 48-31 17 30 15 06 03 55 04   .U....CH1.0...U.
0040 - 0a 13 0e 73 74 72 6f 6e-67 53 65 63 20 47 6d 62   ...strongSec Gmb
0050 - 48 31 1a 30 18 06 03 55-04 03 13 11 73 74 72 6f   H1.0...U....stro
0060 - 6e 67 53 65 63 20 52 6f-6f 74 20 43 41 30 1e 17   ngSec Root CA0..
0070 - 0d 30 32 30 38 30 37 31-35 31 30 33 31 5a 17 0d   .020807151031Z..
0080 - 30 33 30 38 30 37 31 35-31 30 33 31 5a 30 5f 31   030807151031Z0_1
0090 - 0b 30 09 06 03 55 04 06-13 02 43 48 31 17 30 15   .0...U....CH1.0.
00a0 - 06 03 55 04 0a 13 0e 73-74 72 6f 6e 67 53 65 63   ..U....strongSec
00b0 - 20 47 6d 62 48 31 19 30-17 06 03 55 04 0b 13 10    GmbH1.0...U....
00c0 - 31 30 32 34 20 62 69 74-20 52 53 41 20 6b 65 79   1024 bit RSA key
00d0 - 31 1c 30 1a 06 03 55 04-03 13 13 70 79 67 6d 79   1.0...U....pygmy
00e0 - 2e 73 74 72 6f 6e 67 73-65 63 2e 63 6f 6d 30 81   .strongsec.com0.
00f0 - 9f 30 0d 06 09 2a 86 48-86 f7 0d 01 01 01 05 00   .0...*.H........
0100 - 03 81 8d 00 30 81 89 02-81 81 00 dc 00 86 83 3e   ....0..........>
0110 - cb 1b b2 4a b0 8f 53 c9-fa c3 cc f8 69 20 e4 c0   ...J..S.....i ..
0120 - f9 d8 c1 9a 3b 56 e4 7d-ea 0a 9e 22 1b 71 3a 30   ....;V.}...".q:0
0130 - 9b 0c 5e a1 13 e7 c0 fe-2a 5f e1 c6 d8 64 4f c9   ..^.....*_...dO.
0140 - 5c 8f 5c f6 d0 36 e4 0f-68 2c 2c 91 52 e2 a5 5a  \.\..6..h,,.R..Z
0150 - db 1c 33 be 01 6e 56 54-2c 73 df 2c 82 70 bf ea   ..3..nVT,s.,.p..
0160 - 1d de 57 e7 d8 f5 18 fb-b9 6d 57 14 fc a6 ba 3a   ..W......mW....:
0170 - af f9 99 f2 d3 be fa e1-d1 3b b8 03 0c 1d f4 25   .........;.....%
0180 - 32 b7 29 ce 79 d6 c0 19-9b 79 97 02 03 01 00 01   2.).y....y......
0190 - a3 82 01 0c 30 82 01 08-30 09 06 03 55 1d 13 04   ....0...0...U...
01a0 - 02 30 00 30 11 06 09 60-86 48 01 86 f8 42 01 01   .0.0...`.H...B..
01b0 - 04 04 03 02 06 40 30 0b-06 03 55 1d 0f 04 04 03   .....@0...U.....
```

```
01c0 - 02 05 e0 30 5d 06 09 60-86 48 01 86 f8 42 01 0d   ...0]..`.H...B..
01d0 - 04 50 16 4e 43 65 72 74-69 66 69 63 61 74 65 20   .P.NCertificate
01e0 - 69 73 73 75 65 64 20 62-79 20 73 74 72 6f 6e 67   issued by strong
01f0 - 53 65 63 20 47 6d 62 48-2c 20 53 77 69 74 7a 65   Sec GmbH, Switze
0200 - 72 6c 61 6e 64 2c 20 68-74 74 70 3a 2f 2f 77 77   rland, http://ww
0210 - 77 2e 73 74 72 6f 6e 67-73 65 63 2e 63 68 2f 63   w.strongsec.ch/c
0220
 - 61 2f 30 21 06 03 55 1d-11 01 01 ff 04 17 30 15   a/0!..U.......0.
0230 - 82 13 70 79 67 6d 79 2e-73 74 72 6f 6e 67 73 65   ..pygmy.strongse
0240 - 63 2e 63 6f 6d 30 35 06-03 55 1d 1f 04 2e 30 2c   c.com05..U....0,
0250 - 30 2a a0 28 a0 26 86 24-68 74 74 70 3a 2f 2f 77   0*.(.&.$http://w
0260 - 77 77 2e 73 74 72 6f 6e-67 73 65 63 2e 63 6f 6d   ww.strongsec.com
0270 - 2f 63 61 2f 63 65 72 74-2e 63 72 6c 30 22 06 09   /ca/cert.crl0"..
0280 - 60 86 48 01 86 f8 42 01-0c 04 15 16 13 70 79 67   `.H...B......pyg
0290 - 6d 79 2e 73 74 72 6f 6e-67 73 65 63 2e 63 6f 6d   my.strongsec.com
02a0 - 30 0d 06 09 2a 86 48 86-f7 0d 01 01 04 05 00 03   0...*.H.........
02b0 - 82 01 01 00 45 7b e7 55-5a a8 7c a3 a1 dc 20 fd   ....E{.UZ.|... .
02c0 - c3 78 77 af a7 e6 4f 22-80 33 fb 1b 16 79 84 33   .xw...O".3...y.3
02d0 - 4c 31 a8 82 99 e5 a6 23-c7 8b a9 a4 8d 6f e8 34   L1.....#.....o.4
02e0 - 51 2d 44 4d 86 7c 77 68-d5 57 18 b8 1d 2f 72 d1   Q-DM.|wh.W.../r.
02f0 - 32 dd 43 70 fa 22 f6 93-e7 19 a9 8c d1 0e 54 fe   2.Cp."........T.
0300 - a3 96 a8 98 81 f9 73 ca-18 6c c0 51 c5 78 eb 96   ......s..l.Q.x..
0310 - 25 74 1c 84 47 4e 17 a3-10 47 8b 0f e8 3c a4 b2   %t..GN...G...<..
0320 - 79 3e fb 1b fc b2 63 ad-fb db 08 41 b9 ff 5a 5d   y>....c....A..Z]
0330 - 8c f4 c9 de 5d 57 c3 3d-dd 2a d9 18 94 47 58 77   ....]W.=.*...GXw
0340 - a7 7c 85 95 c8 0a 5a c2-f5 5b a5 c9 0d f7 e6 29   .|....Z..[.....)
0350 - d9 fa 13 c8 9f 6f 11 d9-5a 70 a4 19 71 96 f7 6a   .....o..Zp..q..j
0360 - ef 64 7e 87 a6 50 e0 2d-38 cc aa 2e 04 49 6b bb   .d~..P.-8....Ik.
0370 - 8b 51 d0 c4 a0 b7 2d 6d-9c 9a c3 93 fc 32 bf 3b   .Q....-m.....2.;
0380 - 70 16 b6 f7 54 f1 91 64-de 1d 11 18 de bd c1 48   p...T..d.......H
0390 - f3 a9 a7 8d 4d 85 14 ad-6b aa a0 b7 25 eb 38 80   ....M...k...%.8.
03a0 - f4 b8 10 7d 44 35 22 15-3d fc e6 42 2c 46 1b 23   ...}D5".=..B,F.#
03b0 - 4b 0e 6c 76 0e                                    K.lv.
03b8 - <SPACES/NULS>
depth=0 /C=CH/O=strongSec GmbH/OU=1024 bit RSA key/CN=pygmy.strongsec.com
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=0 /C=CH/O=strongSec GmbH/OU=1024 bit RSA key/CN=pygmy.strongsec.com
verify error:num=27:certificate not trusted
verify return:1
depth=0 /C=CH/O=strongSec GmbH/OU=1024 bit RSA key/CN=pygmy.strongsec.com
verify error:num=21:unable to verify the first certificate
verify return:1
write to 081A3F18 [081B51B8] (137 bytes => 137 (0x89))
0000 - 16 03 00 00 84 10 00 00-80 8f bd 04 4d 52 d8 aa   ............MR..
0010 - 30 70 bd 59 88 c2 96 8a-b7 01 2d 5f a1 98 41 b5   0p.Y......-_..A.
0020 - 6b 5e ac 3e a5 83 97 da-4f c5 7a 09 04 98 5e fb   k^.>....O.z...^.
0030 - 60 af a0 49 65 36 bc 6b-5d e7 8e bd 31 14 9d be   `..Ie6.k]...1...
0040 - 3d 6f 99 d1 8c ca 69 d1-0d 4e f2 9c bf 03 5d 1c   =o....i..N....].
0050 - 3b 99 33 d6 c9 34 b2 85-f9 e8 51 95 be 73 13 25   ;.3..4....Q..s.%
0060 - 00 fc 95 53 f8 7e 3e 11-db 8e b5 03 a5 51 03 91   ...S.~>......Q..
0070 - 27 06 e5 21 65 af 63 ba-a7 2c bc e3 2b 5d a9 e6   '..!e.c..,..+]..
0080 - f2 c9 f9 8e be 77 59 aa-67                        .....wY.g
write to 081A3F18 [081B51B8] (6 bytes => 6 (0x6))
0000 - 14 03 00 00 01 01                                 ......
write to 081A3F18 [081B51B8] (61 bytes => 61 (0x3D))
0000 - 16 03 00 00 38 46 17 d4-d4 ef c6 c1 18 6b de a8   ....8F.......k..
0010 - 9c 39 17 7e ff ec 9d f5-28 af a1 c3 13 87 ef 95   .9.~....(.......
0020 - f4 f1 15 0b 75 84 69 49-12 da ab 09 59 b1 fd ba   ....u.iI....Y...
0030 - 58 a2 14 18 31 b2 3f 84-e7 1a ca c8 c8            X...1.?......
read from 081A3F18 [081AB090] (5 bytes => 5 (0x5))
0000 - 14 03 00 00 01                                    .....
read from 081A3F18 [081AB095] (1 bytes => 1 (0x1))
0000 - 01                                                .
read from 081A3F18 [081AB090] (5 bytes => 5 (0x5))
0000 - 16 03 00 00 38                                    ....8
read from 081A3F18 [081AB095] (56 bytes => 56 (0x38))
0000 - a5 23 1d bb 44 4f fe e0-52 31 13 5e 31 6e 02 5b   .#..DO..R1.^1n.[
0010 - 21 31 04 e8 4d 80 86 20-93 9a 3c 2b 87 91 6b 3d   !1..M.. ..<+..k=
0020 - 39 e9 98 2f 9b 19 3b 8d-19 ec b1 7c 87 e5 ca a5   9../..;....|....
0030 - 8b 76 27 da c2 f0 5c c0-                          .v'...\.
---
Certificate chain
 0 s:/C=CH/O=strongSec GmbH/OU=1024 bit RSA key/CN=pygmy.strongsec.com
   i:/C=CH/O=strongSec GmbH/CN=strongSec Root CA
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIDpjCCAo6gAwIBAgIBMjANBgkqhkiG9w0BAQQFADBCMQswCQYDVQQGEwJDSDEX
MBUGA1UEChMOc3Ryb25nU2VjIEdtYkgxGjAYBgNVBAMTEXN0cm9uZ1NlYyBSb290
IENBMB4XDTAyMDgwNzE1MTAzMVoXDTAzMDgwNzE1MTAzMVowXzELMAkGA1UEBhMC
Q0gxFzAVBgNVBAoTDnN0cm9uZ1NlYyBHbWJIMRkwFwYDVQQLExAxMDI0IGJpdCBS
U0Ega2V5MRwwGgYDVQQDExNweWdteS5zdHJvbmdzZWMuY29tMIGfMA0GCSqGSIb3
DQEBAQUAA4GNADCBiQKBgQDcAIaDPssbskqwj1PJ+sPM+Gkg5MD52MGaO1bkfeoK
niIbcTowmwxeoRPnwP4qX+HG2GRPyVyPXPbQNuQPaCwskVLipVrbHDO+AW5WVCxz
3yyCcL/qHd5X59j1GPu5bVcU/Ka6Oq/5mfLTvvrh0Tu4Awwd9CUytynOedbAGZt5
lwIDAQABo4IBDDCCAQgwCQYDVR0TBAIwADARBglghkgBhvhCAQEEBAMCBkAwCwYD
VR0PBAQDAgXgMF0GCWCGSAGG+EIBDQRQFk5DZXJ0aWZpY2F0ZSBpc3N1ZWQgYnkg
```

```
c3Ryb25nU2VjIEdtYkgsIFN3aXR6ZXJsYW5kLCBodHRwOi8vd3d3LnN0cm9uZ3Nl
Yy5jaC9jYS8wIQYDVR0RAQH/BBcwFYITcHlnbXkuc3Ryb25nc2VjLmNvbTA1BgNV
HR8ELjAsMCqgKKAmhiRodHRwOi8vd3d3LnN0cm9uZ3NlYy5jb20vY2EvY2VydC5j
cmwwIgYJYIZIAYb4QgEMBBUWE3B5Z215LnN0cm9uZ3NlYy5jb20wDQYJKoZIhvcN
AQEEBQADggEBAEV751VaqHyjodwg/cN4d6+n5k8igDP7GxZ5hDNMMaiCmeWmI8eL
qaSNb+g0US1ETYZ8d2jVVxi4HS9y0TLdQ3D6IvaT5xmpjNEOVP6jlqiYgflzyhhs
wFHFeOuWJXQchEdOF6MQR4sP6Dyksnk++xv8smOt+9sIQbn/Wl2M9MneXVfDPd0q
2RiUR1h3p3yFlcgKWsL1W6XJDffmKdn6E8ifbxHZWnCkGXGW92rvZH6HplDgLTjM
qi4ESWu7i1HQxKC3LW2cmsOT/DK/O3AWtvdU8ZFk3h0RGN69wUjzqaeNTYUUrWuq
oLcl6ziA9LgQfUQ1IhU9/OZCLEYbI0sObHY=
-----END CERTIFICATE-----
subject=/C=CH/O=strongSec GmbH/OU=1024 bit RSA key/CN=pygmy.strongsec.com
issuer=/C=CH/O=strongSec GmbH/CN=strongSec Root CA
---
No client certificate CA names sent
---
SSL handshake has read 1103 bytes and written 254 bytes
---
New, TLSv1/SSLv3, Cipher is RC4-MD5
Server public key is 1024 bit
SSL-Session:
    Protocol  : SSLv3
    Cipher    : RC4-MD5
    Session-ID: 0000000000000000000000000000000000000000000000000000000000000001
    Session-ID-ctx:
    Master-Key:
E1DB26A8729E2A40BC22FAF88C7289A25D39B93FF7D549D130F4AB779AD3A28A59A033CA7A3B1DDB0C06661A4960E5E0
    Key-Arg   : None
    Start Time: 1035653864
    Timeout   : 7200 (sec)
    Verify return code: 21 (unable to verify the first certificate)
---
```

15.3.6 mSSL_SERVER: RSA,RSA mit RC4, MD5

```
A:\>
MSSL: connection from 160.85.162.78:2820
MSSL-sub: fetched next Paket
MSSL-sub: read one RECORD from the received TLS-Paket
MSSL-sub:protocol = 22 version SSLv3 length = 45
MSSL-sub: It's a TLS_HANDSHAKE
MSSL-sub: handshake = 1 length = 41
MSSL: ClientHelloRecord: 03 00 3D BA D2 E8 F2 11 49 D5 C2 6F 4B 14 8A 9A D5 D6 05 48 06 AF FE BE 9C
A8 72 97 AA DD 19 D0 9C F5 00 00 02 00 04 01 00
MSSL-sub: cipher suite to compare: 4 RSA / 128 bit RC4 / MD5
MSSL-sub: cipher suite: RSA / 128 bit RC4 / MD5
MSSL: server hello msg
02 00 00 46 03 00 34 03 61 4B A8 3A CE E0 92 9C FF 03 BE D3 C5 25 A2 EC 61 85 B1 EA 93 BF A0 5E A9 79
1C 8A ED 16 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 01 00 04 00
MSSL-sub: send server_hello_msg
MSSL-sub: Session is new (not resumed)
MSSL-sub: send certificate_msg
MSSL-sub: send server_hello_done_msg
    =>TLS_CLIENT_KEY_EXCHANGE
MSSL-sub: fetched next Paket
MSSL-sub: read one RECORD from the received TLS-Paket
MSSL-sub:protocol = 22 version SSLv3 length = 132
MSSL-sub: It's a TLS_HANDSHAKE
MSSL-sub: handshake = 16 length = 128
MSSL: decrypting premaster secret ...
RSA: data = 128 bytes, modulus = 128 bytes
MSSL-sub:  Time for RSA-decryption1 [ms]: 560b
MSSL-sub:  Time for RSA-decryption2 [ms]: aa8f
MSSL-sub:  Time for RSA-decryption2-1 [ms]: 21636
MSSL: decryption succeeded
MSSL-sub: process client_key_exchange_msg (RSA)
MSSL-sub: =>Client_Key_Exchange processed, =>TLS_CLIENT_CHANGE_CIPHER_SPEC
MSSL-sub: fetched next Paket
MSSL-sub: read one RECORD from the received TLS-Paket
MSSL-sub:protocol = 20 version SSLv3 length = 1
MSSL-sub: =>use new Client_Cipher_Suite from now:
MSSL-sub-specs:   session_id 1
MSSL-sub-specs: client_random  3D BA D2 E8 F2 11 49 D5 C2 6F 4B 14 8A 9A D5 D6 05 48 06 AF FE BE 9C
A8 72 97 AA DD 19 D0 9C F5
MSSL-sub-specs: server-random  34 03 61 4B A8 3A CE E0 92 9C FF 03 BE D3 C5 25 A2 EC 61 85 B1 EA 93
BF A0 5E A9 79 1C 8A ED 16
MSSL-sub-specs: master_secret  E1 DB 26 A8 72 9E 2A 40 BC 22 FA F8 8C 72 89 A2 5D 39 B9 3F F7 D5 49
D1 30 F4 AB 77 9A D3 A2 8A 59 A0 33 CA 7A 3B 1D DB 0C 06 66 1A 49 60 E5 E0
MSSL-sub-specs: secret_keys  BD B0 45 F4 8C E5 F1 F2 8A A7 87 D0 45 88 8B 91 1E BB 9D 6D D6 31 95 F7
9E 6B 1B CE B7 EC 3E 34 0D CB 08 EF 9E 1D 6A 54 02 12 AD 26 2D 84 C0 0D B9 B1 60 71 56 AE ED 6B 1A B4
27 B6 C3 3A D1 20 02 B8 A0 74 A8 92 FD A7 B1 FA A1 D7 DA 2C 55 61 DD B7 F9 91 AC 53 4A ED CC A5 E3 9A
26 56 0A 87
MSSL-sub-specs: client_MAC_secret  BD B0 45 F4 8C E5 F1 F2 8A A7 87 D0 45 88 8B 91
MSSL-sub-specs: server_MAC_secret  1E BB 9D 6D D6 31 95 F7 9E 6B 1B CE B7 EC 3E 34
=>TLS_CLIENT_FINISHED
MSSL-sub: read one RECORD from the received TLS-Paket
MSSL-sub:protocol = 22 version SSLv3 length = 56
MSSL-sub: It's a TLS_HANDSHAKE
MSSL-sub: handshake = 20 length = 36
MSSL: handshake = 20 length = 36
MSSL-sub: process client_finished_msg
MSSL-sub: =>client_finished processed, =>TLS_ACCEPT
MSSL-sub: send change_cipher_spec
MSSL-sub: send server_finished_msg
MSSL-sub: =>new session, so take new specs,

MSSL-sub:=>TLS_ACCEPT
```