# Praktikum: Grundlagen und Implementierung moderner Public-Key-Algorithmen

Prof. Dr. Michael Braun

Sommersemester 2010

**Aufgabe 1. (Effiziente 32-Bit Multiplikation)**
Algorithmus 1 realisiert die Multiplikation zweier 32-Bit Wörter $a$ und $b$ mittels vier Multiplikationen von 16-Bit Wörtern und liefert das 64-Bit Ergebnis $(u, v)$ in Ansi C.

**Algorithmus 1** 32-Bit Multiplikation mit vier 16-Bit-Multiplikationen

```
#define HI(x) (x >> 16)
#define LO(x) (x & 0x0000FFFF)

void mulc(uint32_t *u, uint32_t *v, uint32_t a, uint32_t b) {
  uint32_t t;
  t = LO(a) * LO(b);
  *v = LO(t);
  t = HI(t) + HI(a) * LO(b);
  *u = HI(t);
  t = LO(t) + LO(a) * HI(b);
  *v |= LO(t) << 16;
  *u += HI(t) + HI(a) * HI(b);
}
```

Algorithmus 2 realisiert auch eine Multiplikation zweier 32-Bit Wörter $a$ und $b$ mittels Assembler-Unterstützung. Das Ergebnis ist die 64-Bit-Quantität $(u, v)$.

**Algorithmus 2** Multiplikation von 32-Bit Zahlen auf Intel-x86

```
void mula(uint32_t *u, uint32_t *v, uint32_t a, uint32_t b) {
  __asm {
    mov eax, a
    mul b
    mov *u, edx
    mov *v, eax
  }
}
```

a. Implementieren Sie die Multiplikationsroutine für 32-Bit-Zahlen (i) in Ansi C und (ii) mit Assembler-Unterstützung.

b. Verifizieren Sie die Richtigkeit der beiden Routinen, indem Sie zufällige Zahlen mit beiden Routinen multiplizieren und die Ergebnisse vergleichen.

c. Implementieren Sie danach einen geeigneten Performanz-Test. Wieviel Prozent schneller ist die Assembler unterstützte Implementierung?

**Aufgabe 2. (Bitoperationen)**
Auf einer 32-Bit-Plattform wird ein $m$-Bit-Datenwort $a = (a_{m-1}, \ldots, a_1, a_0)$ als Array $A$ von 32-Bit-Datenworten $A[i]$, $0 \leq i < t$, mit $t = \lceil m/32 \rceil$ dargestellt:

$$A = (A[t-1], \ldots, A[1], A[0]) = (0, \ldots, 0, a_{m-1}, \ldots, a_1, a_0)$$

Die vordersten Nullbits werden dabei nicht verwendet.

a. Implementieren Sie eine Ansi C Routine

```
void f2m_rand(
  uint32_t t,
  uint32_t m,
  uint32_t *A
);
```

welche ein zufälliges $m$-Bit-Datenwort $a = (a_{m-1}, \ldots, a_1, a_0)$ erzeugt und in ein Array $A$ der Länge $t$ schreibt. Der Speicher für $A$ soll bereits allokiert sein.

b. Implementieren Sie ein Ansi C Routine

```
void f2m_print(
  uint32_t t,
  uint32_t *A
);
```

welche das Array als hexadezimale Folge $A[t-1], A[t-2], \ldots, A[1], A[0]$ auf den Bildschirm ausgibt.

c. Implementieren Sie eine Ansi C Routine

```
void f2m_assign(
  uint32_t t,
  uint32_t *A,
  uint32_t *B
);
```

welche ein Array $A$ der Länge $t$ in ein Array $B$ der Länge $t$ kopiert. Der Speicher für $B$ soll bereits allokiert sein.

d. Implementieren Sie eine Ansi C Routine

```
void f2m_and(
  uint32_t t,
  uint32_t *A,
  uint32_t *B,
  uint32_t *C
);
```

welche eine AND-Operation von zwei Arrays $A$ und $B$ der Länge $t$ realisiert und das Ergebnis in $C$ speichert. Der Speicher für $C$ soll bereits allokiert sein.

e. Implementieren Sie eine Ansi C Routine

```
void f2m_or(
  uint32_t t,
  uint32_t *A,
  uint32_t *B,
  uint32_t *C
);
```

welche eine OR-Operation von zwei Arrays $A$ und $B$ der Länge $t$ realisiert und das Ergebnis in $C$ speichert. Der Speicher für $C$ soll bereits allokiert sein.

f. Implementieren Sie eine Ansi C Routine

```
void f2m_xor(
  uint32_t t,
  uint32_t *A,
  uint32_t *B,
  uint32_t *C
);
```

welche eine XOR-Operation von zwei Arrays $A$ und $B$ der Länge $t$ realisiert und das Ergebnis in $C$ speichert. Der Speicher für $C$ soll bereits allokiert sein.

g. Implementieren Sie eine Ansi C Routine

```
void f2m_shiftleft(
  uint32_t t,
  uint32_t *A,
  uint32_t *B
);
```

welche einen Links-Shift eines Arrays $A$ der Länge $t$ realisiert und das Ergebnis in $B$ speichert. Der Speicher für $B$ soll bereits allokiert sein. Es soll auch ein Aufruf „`f2m_shiftleft(t,A,A)`“ möglich sein.

h. Implementieren Sie eine Ansi C Routine

```
void f2m_shiftright(
  uint32_t t,
  uint32_t *A,
  uint32_t *B
);
```

welche einen Rechts-Shift eines Arrays $A$ der Länge $t$ realisiert und das Ergebnis in $B$ speichert. Der Speicher für $B$ soll bereits allokiert sein. Es soll auch ein Aufruf „`f2m_shiftright(t,A,A)`“ möglich sein.

i. Implementieren Sie eine Ansi C Routine

```
uint32_t get_bit(
  uint32_t U,
  uint32_t i
);
```

welche das Bit $U_i$ aus dem 32-Bit-Wort $U = U_{31} \ldots U_1 U_0$ zurück gibt.

j. Implementieren Sie eine Ansi C Routine

```
uint32_t f2m_bitm(
  uint32_t t,
  uint32_t m,
  uint32_t *A
);
```

welche das Bit $a_{m-1}$ aus dem Array $A$ der Länge $t$ zurück gibt.

**Aufgabe 3. (Arithmetik in $\mathbb{F}_{2^m}$)**
Auf einer 32-Bit-Plattform wird ein Element $a(z) = a_{m-1}z^{m-1} + \ldots + a_1 z + a_0$ aus $\mathbb{F}_{2^m}$ als Array $A$ von 32-Bit-Datenworten $A[i]$, $0 \leq i < t$, mit $t = \lceil m/32 \rceil$ dargestellt:

$$A = (A[t-1], \ldots, A[1], A[0]) = (0, \ldots, 0, a_{m-1}, \ldots, a_1, a_0)$$

Die vordersten Nullbits werden dabei nicht verwendet.

a. Implementieren Sie eine Ansi C Routine

```
uint32_t f2m_biti(
  uint32_t t,
  uint32_t i,
  uint32_t *A
);
```

welche das Bit $a_i$ aus dem Array $A$ der Länge $t$ zurück gibt.

b. Implementieren Sie eine Ansi C Routine

```
void f2m_one(
  uint32_t t,
  uint32_t *A
);
```

welche das Polynom $a(z) = 1$ speichert. Der Speicher für $A$ soll bereits allokiert sein.

c. Implementieren Sie eine Ansi C Routine

```
void f2m_zero(
  uint32_t t,
  uint32_t *A
);
```

welche das Polynom $a(z) = 0$ speichert. Der Speicher für $A$ soll bereits allokiert sein.

d. Implementieren Sie eine Ansi C Routine

```
uint32_t f2m_is_equal(
  uint32_t t,
  uint32_t *A,
  uint32_t *B
);
```

welche 1 zurück gibt, falls das Array $A$ der Länge $t$ den gleichen Inhalt wie $B$ besitzt und 0 sonst.

e. Implementieren Sie eine Ansi C Routine

```
uint32_t f2m_is_one(
  uint32_t t,
  uint32_t *A,
);
```

welche 1 zurück gibt, falls das Array $A$ der Länge $t$ das Polynom $a(z) = 1$ darstellt und 0 sonst.

f. Implementieren Sie eine Ansi C Routine

```
uint32_t f2m_is_zero(
  uint32_t t,
  uint32_t *A,
);
```

welche 1 zurück gibt, falls das Array $A$ der Länge $t$ das Polynom $a(z) = 0$ darstellt und 0 sonst.

g. Implementieren Sie eine Ansi C Routine

```
void f2m_mul(
  uint32_t t,
  uint32_t m,
  uint32_t *A,
  uint32_t *B,
  uint32_t *F,
  uint32_t *C,
  uint32_t *T1,
  uint32_t *T2
);
```

welche den Shift-and-Algorithmus (siehe Alg. 3) zur modularen Multiplikation

$$c(z) = a(z)b(z) \mod f(x)$$

realisiert. Die Register $T1$ und $T2$ sind Hilfsregister, die sie intern verwenden können, ohne in der Funktion Hilfsregister allokieren zu müssen. Alle Hilfsregister und das Register $C$ müssen bereits allokiert sein. Es soll auch ein Aufruf „`f2m_mul(t,m,A,B,F,A,T1,T2)`“ möglich sein.

**Algorithmus 3** Shift-and-Add

Eingabe: $a(z), b(z) \in \mathbb{F}_2[z]$ mit Grad $\leq m-1$, $f(z) = z^m + g(z)$ irreduzibel
Ausgabe: $c(z) = a(z)b(z) \mod f(z)$

(1) $t^{(1)} \leftarrow b$
(2) $t^{(2)} \leftarrow 0$
(3) **for** $i \leftarrow 0$ **to** $m-1$ **do**
(4) **if** $t_m^{(1)} = 1$ **then** $t^{(1)} \leftarrow t^{(1)} \oplus f$
(5) **if** $a_i = 1$ **then** $t^{(2)} \leftarrow t^{(2)} \oplus t^{(1)}$
(6) $t^{(1)} \leftarrow t^{(1)} \ll 1$
(7) $c \leftarrow t^{(2)}$
(8) **return** $c$

h. Implementieren Sie eine Ansi C Routine

```
void f2m_inv(
  uint32_t t,
  uint32_t m,
  uint32_t *A,
  uint32_t *F,
  uint32_t *B,
  uint32_t *U,
  uint32_t *V,
  uint32_t *T1,
  uint32_t *T2
);
```

welche den binären Algorithmus zur multiplikativen Inversion (siehe Alg. 4)

$$b(x) = a(x)^{-1} \mod f(z)$$

realisiert. Die Register $U$, $V$, $T1$ und $T2$ sind Hilfsregister, die sie intern verwenden können, ohne in der Funktion Hilfsregister allokieren zu müssen. Alle Hilfsregister und das Register $B$ müssen bereits allokiert sein. Es soll auch ein Aufruf „`f2m_inv(t,A,F,A,U,V,T1,T2)`“ möglich sein.

**Algorithmus 4** Inversion

Eingabe: $a(z) \in \mathbb{F}_2[z] \setminus \{0\}$ mit Grad $\leq m-1$, $f(z) = z^m + g(z)$ irreduzibel
Ausgabe: $b(z) = a(z)^{-1} \mod f(z)$

(1) $u \leftarrow a$, $v \leftarrow f$
(3) $t^{(1)} \leftarrow 1$, $t^{(2)} \leftarrow 0$
(5) **while** ($u \neq 1$ **and** $v \neq 1$) **do**
(6) **while** ($u_0 = 0$ **and** $u \neq 0$) **do**
(7) $u \leftarrow u \gg 1$
(8) **if** ($t_0^{(1)} = 0$ **and** $t^{(1)} \neq 0$) **then** $t^{(1)} \leftarrow t^{(1)} \gg 1$
(9) **else** $t^{(1)} \leftarrow (t^{(1)} \oplus f) \gg 1$
(10) **while** ($v_0 = 0$ **and** $v \neq 0$) **do**
(11) $v \leftarrow v \gg 1$
(12) **if** ($t_0^{(2)} = 0$ **and** $t^{(2)} \neq 0$) **then** $t^{(2)} \leftarrow t^{(2)} \gg 1$
(13) **else** $t^{(2)} \leftarrow (t^{(2)} \oplus f) \gg 1$
(14) **if** $\deg(u) > \deg(v)$ **then:** $u \leftarrow u \oplus v$, $t^{(1)} \leftarrow t^{(1)} \oplus t^{(2)}$
(15) **else:** $v \leftarrow u \oplus v$, $t^{(2)} \leftarrow t^{(1)} \oplus t^{(2)}$
(16) **if** $u = 1$ **then** $b \leftarrow t^{(1)}$; **else** $b \leftarrow t^{(2)}$
(17) **return** $b$

i. Schreiben Sie Routinen, welche die Korrektheit von Multiplikation und Inversion überprüfen und testen Sie diese für den Körper $\mathbb{F}_{2^{163}}$ mit dem irreduziblen Polynom $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$.

**Aufgabe 4. (Elliptische Kurven Arithmetik)**

a. Implementieren Sie eine Ansi C Routine (siehe Algorithmus 5)

```
void f2m_montgomery(
  uint32_t t,
  uint32_t m,
  uint32_t *k,
  uint32_t *xP,
  uint32_t *F,
  uint32_t *b,
  uint32_t *X1,
  uint32_t *Z1,
  uint32_t *X2,
  uint32_t *Z2,
  uint32_t *T,
  uint32_t *S,
  uint32_t *M1,
  uint32_t *M2
);
```

welche aus einer $x$-Koordinate $x_P$ eines Punktes $P$ auf einer elliptischen Kurve $E : y^2 + xy = x^3 + ax^2 + b$ die $x$-Koordinate $x_1$ von $k \cdot P$ und die $x$-Koordinate $x_2$ von $(k+1) \cdot P$ berechnet. Die Koordinate $x_1$ wird dabei als $(X_1, Z_1)$ mit $x_1 = X_1/Z_1$ repräsentiert und $x_2$ durch $(X_2, Z_2)$ mit $x_2 = X_2/Z_2$.

Der Parameter $b$ ist der $b$-Parameter der elliptischen Kurve ($a$ wird nicht benötigt!) und $F$ speichert das irreduzible Polynom des endlichen Körpers $\mathbb{F}_{2^m}$.

Die Register $T, S$ sind Hilfsregister und $M_1, M_2$ sind Register, die für die Multiplikation im endlichen Körper $\mathbb{F}_{2^m}$ verwendet werden.

Der Speicher für die Register $X_1, Z_1, X_2, Z_2, T, S, M_1, M_2$ muss bereits allokiert sein.

**Algorithmus 5** Montgomery Skalarmultiplikation

Eingabe: $x$-Koordinate $x_P$ eines Punktes $P$, Parameter $b$ der elliptischen Kurve, $k$ natürliche Zahl

Ausgabe: $X_1, Z_1, X_2, Z_2$, wobei $x_1 = X_1/Z_1$ die $x$-Koordinate des Punktes $k \cdot P$ ist und $x_2 = X_2/Z_2$ die $x$-Koordinate des Punktes $(k+1) \cdot P$.

(1) $X_1 \leftarrow 1$, $Z_1 \leftarrow 0$
(2) $X_2 \leftarrow x_P$, $Z_2 \leftarrow 1$
(3) **for** $i \leftarrow m-1$ **downto** $0$ **do**
(4) **if** $k_i = 1$ **do**
(5) $T \leftarrow X_1 * Z_2$
(6) $S \leftarrow X_2 * Z_1$
(7) $X_1 \leftarrow T * S$
(8) $S \leftarrow T \oplus S$
(9) $Z_1 \leftarrow S * S$
(10) $T \leftarrow x_P * Z_1$
(11) $X_1 \leftarrow X_1 \oplus T$
(12) $T \leftarrow X_2 * X_2$
(13) $S \leftarrow Z_2 * Z_2$
(14) $Z_2 \leftarrow S * T$
(15) $S \leftarrow S * S$
(16) $S \leftarrow S * b$
(17) $X_2 \leftarrow T * T$
(18) $X_2 \leftarrow X_2 \oplus S$
(19) **else do**
(20) $T \leftarrow X_2 * Z_1$
(21) $S \leftarrow X_1 * Z_2$
(22) $X_2 \leftarrow T * S$
(23) $S \leftarrow T \oplus S$
(24) $Z_2 \leftarrow S * S$
(25) $T \leftarrow x_P * Z_2$
(26) $X_2 \leftarrow X_2 \oplus T$
(27) $T \leftarrow X_1 * X_1$
(28) $S \leftarrow Z_1 * Z_1$
(29) $Z_1 \leftarrow S * T$
(30) $S \leftarrow S * S$
(31) $S \leftarrow S * b$
(32) $X_1 \leftarrow T * T$
(33) $X_1 \leftarrow X_1 \oplus S$
(34) **return** $(X_1, Z_1, X_2, Z_2)$

b. Verifizieren Sie Ihren Algorithmus anhand der folgenden Testdaten:

$$
\begin{aligned}
f(z) &= z^{163} + z^7 + z^6 + z^3 + 1 \\
b &= \texttt{0x00000002 0A601907 B8C953CA 1481EB10 512F7874 4A3205FD} \\
x_P &= \texttt{0x00000003 F0EBA162 86A2D57E A0991168 D4994637 E8343E36} \\
y_P &= \texttt{0x00000000 D51FBC6C 71A0094F A2CDD545 B11C5C0C 797324F1} \\
k &= \texttt{0x00000001 367D5331 3A3A028C 0BF1F5B4 34B7C146 627801B3}
\end{aligned}
$$

Ergebnis:

$$
\begin{aligned}
X_1/Z_1 &= \texttt{0x00000000 C6F057F8 7A2904D1 A92F5AAC 224A3097 2956F3A9} \\
X_2/Z_2 &= \texttt{0x00000001 E94CC140 0477F984 493BE049 5646250C C4A82A12}
\end{aligned}
$$

c. Schreiben Sie eine Ansi C Routine

```
uint32_t f2m_reconstruct_point(
  uint32_t t,
  uint32_t m,
  uint32_t *F,
  uint32_t *xP,
  uint32_t *yP,
  uint32_t *X1,
  uint32_t *Z1,
  uint32_t *X2,
  uint32_t *Z2,
  uint32_t *xQ,
  uint32_t *yQ,
  uint32_t *A1,
  uint32_t *A2,
  uint32_t *A3,
  uint32_t *A4
);
```

welche aus dem Punkt $P = (x_P, y_P)$ und den Koordinaten $(X_1, Z_1)$ und $(X_2, Z_2)$, wobei $X_1/Z_1$ die $x$-Koordinate des Punktes $k \cdot P$ und $X_2/Z_2$ die $x$-Koordinate des Punktes $(k+1) \cdot P$ ist, die Koordinaten von $Q = (x_Q, y_Q) = k \cdot P$ berechnet. Ist $Q = \infty$ der unendlich ferne Punkt, dann wird 0 zurückgegeben, ansonsten 1.

Die Register $A_1, A_2, A_3, A_4$ sind Hilfsregister, die für Multiplikation und Inversion verwendet werden. Der Speicher für die Hilfsregister muss bereits allokiert sein.

**Algorithmus 6** Punkt-Rückgewinnerung

Eingabe: Koordinaten $(x_P, y_P)$ eines Punktes $P$, $(X_1, Z_1, X_2, Z_2)$ wobei $X_1/Z_1$ die $x$-Koordinate des Punktes $k \cdot P$ ist und $X_2/Z_2$ die $x$-Koordinate des Punktes $(k+1) \cdot P$.
Ausgabe: Koordinaten $(x_Q, y_Q)$ des Punktes $Q = k \cdot P$.
Rückgabewert: 0 falls $Q = \infty$ und 1 sonst.

(1) **if** $Z_1 = 0$ **then return**$(0)$
(2) **if** $Z_2 = 0$ **then** $x_Q \leftarrow x_P$; $y_Q \leftarrow x_P \oplus y_P$; **return**$(1)$
(3) $x_Q \leftarrow Z_1 * Z_2$
(4) $y_Q \leftarrow x_P * x_P$
(5) $y_Q \leftarrow y_Q \oplus y_P$
(6) $y_Q \leftarrow x_Q * y_Q$
(7) $x_Q \leftarrow x_Q * x_P$
(8) $x_Q \leftarrow x_Q^{-1}$
(9) $Z_2 \leftarrow Z_2 * x_P$
(10) $Z_2 \leftarrow Z_2 \oplus X_2$
(11) $X_2 \leftarrow Z_1 * x_P$
(12) $X_2 \leftarrow X_2 \oplus X_1$
(13) $Z_2 \leftarrow X_2 * Z_2$
(14) $y_Q \leftarrow y_Q \oplus Z_2$
(15) $y_Q \leftarrow y_Q * x_Q$
(16) $x_Q \leftarrow Z_1^{-1}$
(17) $x_Q \leftarrow x_Q * X_1$
(18) $Z_2 \leftarrow x_P \oplus x_Q$
(19) $y_Q \leftarrow y_Q * Z_2$
(20) $y_Q \leftarrow y_Q \oplus y_P$
(21) **return**$(1)$

d. Testen Sie die Routine, indem Sie die Koordinaten von $Q = (x_Q, y_Q) = k \cdot P$ für die Eingabewerte aus Teilaufgabe b bestimmen. Das Ergebnis ist:

$$x_Q = \texttt{0x00000000 C6F057F8 7A2904D1 A92F5AAC 224A3097 2956F3A9}$$
$$y_Q = \texttt{0x00000001 7BDF0F38 F1624A38 BF6AB32A E8591187 9F1FC7C8}$$

**Aufgabe 5. (Diffie-Hellman Protokoll)**
Schreiben Sie ein Programm, welches für die Parameter

$$
\begin{aligned}
f(z) &= z^{163} + z^7 + z^6 + z^3 + 1 \\
b &= \texttt{0x00000002 0A601907 B8C953CA 1481EB10 512F7874 4A3205FD} \\
x_P &= \texttt{0x00000003 F0EBA162 86A2D57E A0991168 D4994637 E8343E36} \\
y_P &= \texttt{0x00000000 D51FBC6C 71A0094F A2CDD545 B11C5C0C 797324F1}
\end{aligned}
$$

und zufällige Werte $k, d$ die Gleichung

$$k \cdot (d \cdot P) = d \cdot (k \cdot P)$$

überprfüft.

**Das Praktikum gilt als erfolgreich bestanden, wenn dieser Test funktioniert!**

**Lernkontrollaufgaben 1. (Kryptographie)**

a. Ordnen Sie den Schutzzielen (Vertraulichkeit, Integrität, Datenauthentizität, Teilnehmerauthentizität, Verbindlichkeit) die Mechanismen (asymmetrische und symmetrische Verschlüsselung, Hashfunktionen, Message Authentication Codes, Challenge-Response-Verfahren, Digitale Signatur) zu, welche diese Ziele realisieren.

b. Wie unterscheiden sich RSA, DL und ECC Systeme in der Schlüsselerzeugung? Bei welchem Verfahren ist sie am aufwändigsten? Erläutern Sie warum!

c. Wieviel Bits an Sicherheit bei ECC benötigt man, um ein äquivalentes Sicherheitsniveau zu erhalten wie AES-256?

d. Beweisen Sie, dass der DSA-Algorithmus wirklich funktioniert!

e. Eine schnelle Exponentiation „$x^d$ mod $n$“ wird sowohl für RSA- als auch für DL-Systeme benötigt. Skizzieren Sie den „Square-and-Multiply-Algorithmus“ und geben Sie den Aufwand an.

**Lernkontrollaufgaben 2. (Endliche Körper und Arithmetik)**

a. Welche Arten zur Konstruktion endlicher Körper gibt es? Erläutern Sie ausführlich diese Konstruktionen. Wie funktionieren jeweils Multiplikation und Addition?

b. Beschreiben Sie den euklidischen Algorithmus. Wie funktioniert der erweiterte euklidische Algorithmus. Wie kann man damit in Primkörpern Elemente multiplikativ invertieren?

c. Ein binäres Polynom $a(z)$ kann auch als binäre Folge dargestellt werden, so dass der höchste Koeffizient ganz links und der niedrigste Koeffizient ganz rechts steht. Wie lässt sich hier ein Links- bzw. Rechts-Shift mathematisch interpretieren?

d. Wie lässt sich ein Links- bzw. Rechts-Shift eines Elementes eines Primkörpers mathematisch interpretieren?

**Lernkontrollaufgaben 3. (Elliptische Kurven)**

a. Erläutern Sie das Diskrete Logarithmusproblem für Elliptische Kurven und erklären Sie wie man damit asymmetrische Kryptographie realisieren kann.

b. Erläutern Sie die Funktionsweise des Square-and-Multiply-Algorithmus und begründen Sie, warum der Algorithmus funktioniert.