Digital Cash

cand. mach. cand. inform. Jens Trotzky PGDipSc.

1. Dezember 2006

Inhaltsverzeichnis

1 Einführung

Dieser Seminarvortrag soll ein einfaches Modell zur Repräsentation von Geld beschreiben. Dabei soll es um ein System gehen, daß Eigenschaften, die wir von heutigem Geld her kennen, in mathematischer Form beschreibt und Möglichkeiten aufzeigen, wie man Geld in einer digitalen Welt abbilden kann. Der Vortrag basiert hauptsächlich auf Kapitel 4.5 [1].

1.1 Eigenschaften von Geld

Als erstes betrachten wir Eigenschaften unseres heutigen Geldes, wie es überall in der Welt benutzt wird. Die Eigenschaften führen zu einer sehr allgemein gültigen Aussage: Geld ist allgemein akzeptiert, d.h., daß ein Verkäufer, ein Kunde und eine Bank sich darauf verlassen können, daß sie mit dem Geld untereinander Geschäfte und Transaktionen abwickeln können. Dies resultiert daraus, daß ein jeder, am Geschäft Beteiligter, darauf vertraut, daß er mit dem vorhandenen Geld auch mit anderen Entitäten Handel betreiben kann. Zudem erkennen wir, daß folgende Dinge gelten:

- Geld ist sicher
- Geld ist teilbar
- Geld ist anonym

1.2 Transaktionen

1.2.1 Online Transaktion

Unter einer Online Transaktion verstehen wir ein Geschäft, bei dem während der Transaktion die Bank mit einbezogen wird. Das Schema läuft im Normalfall wie folgt ab:
Der Kunde möchte ein Produkt erwerben und authentifiziert sich, z.B. mit Hilfe einer Karte (ec-Karte, Kreditkarte, etc.) und einer dazugehörigen PIN Nummer (Personal Identification Number). Die Daten werden dann elektronisch mit der Bank verifiziert (Im Allgemeinen wird gleichzeitig auch die Bonität des Kunden überprüft). Im positiven Falle wird das Konto um den Betrag belastet und die Transaktion ist abgeschlossen. Der Kunde hat damit das Produkt erworben. Dem Verkäufer wird das entsprechende Geld auf sein Konto gutgeschrieben. Der Nachteil hierbei ist, daß die Bank jede Transaktion mitprotokollieren kann und somit weiss, was der Kunde wann und wo gekauft hat.

1.2.2 Offline Transaktion

Eine Offline Transaktion entspricht typischem Bargeld. Dazu hebt ein Kunde von seinem Konto eine bestimmte Menge Geldes ab und erhält diese in Form von Münzen und Banknoten. Anschliessend kann er mit diesem Bargeld bezahlen. Der Verkäufer bringt dieses Geld nach dem Kauf zu seiner Bank, um es einzuzahlen. Zu beachten ist hier, daß besondere Maßnahmen ergriffen werden müssen, um ein Kopieren der Münzen und Banknoten zu unterbinden. Bei Bargeld funktioniert dies durch die Fälschungssicherheit der Banknoten.

2 Grundlagen

2.1 Primzahlen

Primzahlen sind in der Kryptographie von besonderer Bedeutung. Unter Primzahlen verstehen wir diejenigen ganzen positiven Zahlen, die sich nur durch 1 und sich selbst teilen lassen. Die ersten Primzahlen lauten: $[1], 2, 3, 5, 7, 11,$ Es gibt unendlich viele Primzahlen [1] [Delfs, Seite 268, Theorem A.62]

Primzahlen kommt eine so besondere Bedeutung zu, weil es relativ einfach ist, daß Produkt zweier Primzahlen zu berechnen, umgekehrt aber sehr schwer ist bei gegebener Zahl die beiden Primfaktoren zu ermitteln. Es gibt einige Verfahren in der Kryptographie, die auf dieser Tatsache basierend, Sicherheit ermöglichen.

2.1.1 Sichere Primzahlen

Der folgende Quelltext beschreibt einen einfachen Algorithmus, mit dem Sichere Primzahlen erzeugt werden können.
Die Primzahleigenschaft wird dabei mit dem Miller-Rabin Primzahltest überprüft. Dieser wird im folgenden Abschnitt kurz beschrieben.

```
def secureprim(bits):
    isprim=0
    while isprim==0:
        q=longprim(bits-1)
        p=q*2+1
        isprim=millerrabin(p)
    return p

def longprim(bits):
    from random import randint
    isprim=0
    while isprim==0:
        p=1
        for a in range(1,bits):
            p=p+randint(0,1)*pow(2,a)
        isprim=millerrabin(p)
    return p
```

2.1.2 Primzahltest nach Rabin und Miller

Der Primzahltest von Miller und Rabin ist ein probabilistischer Primzahltest. Die Aussage, daß eine Zahl n eine Primzahl ist, ist nicht immer wahr. Eine Aussage, daß eine gegebene Zahl n keine Primzahl ist, ist hingegen immer richtig. Der Primzahltest nach Miller und Rabin ist eine schnelle Möglichkeit, eine Zahl auf ihre Primzahleigenschaften hin zu testen.
Der folgende Quelltext zeigt eine Implementierung des Algorithmus. Er basiert auf [6].

```
def millerrabin(n):
    from random import randint
    m=n-1
    l=0
    while m%2==0:
        m=m/2
        l=l+1
    x=0
    a=0
    while x<=n/2:
        a=a+1
        x=x+randint(0,1)*pow(2,a)
    x0=pow(x,m,n)
```

```
    if x0==1 or x0==n-1:
        return 1
    for i in range(1, l-1):
        x1=pow(x0,2,n)
        if x1==n-1:
            return 1
        elif x1==1:
            return 0
        x0=x1
    return 0
```

2.2 Gruppen

Eine Gruppe ist ein mathematisches Konstrukt mit folgenden Charakteristika. [7]
Eine Gruppe besteht aus einer Menge $\mathbb{G}$ zusammen mit einer Veknüpfung $\cdot$, die je zwei Elementen aus $\mathbb{G}$ wieder ein Element aus $\mathbb{G}$ zuordnet. Weiterhin gelten folgende 3 Axiome:

1. Assoziativität: Die Verknüpfung $\cdot$ ist assoziativ: $(g \cdot h) \cdot k = g \cdot (h \cdot k) \forall g, h, k \in \mathbb{G}$
2. Existenz eines neutralen Elements: $\exists e \in \mathbb{G} | e \cdot g = g \forall g \in \mathbb{G}$
3. Existenz eines inversen Elements: $\forall g \in \mathbb{G} \exists g^{-1} | g^{-1} \cdot g = e$

2.3 Generator

```
def findgenerator(p,q):
    from random import randint
    isgenerator=0
    while isgenerator==0:
        g=1
        a=0
        while g<=p/2:
            a=a+1
            g=g+randint(0,1)*pow(2,a)
        g=g%p
        if pow(g,q,p)==1:
            isgenerator=1
    return g
```

2.4 Hashfunktionen

Eine Hashfunktion h erfüllt nach [6] folgende Eigenschaften:

1. h bildet Eingaben einer beliebigen Bitlänge auf Ausgaben $h(x)$ einer festen Bitlänge ab. $h(x)$ wird auch Fingerabdruck (fingerprint) von x genannt.
2. Es seien x und h gegeben, dann ist $h(x)$ leicht zu berechnen.

Für den Fall einer 128-Bit Hashfunktion schreiben wir kurz: $h : \{0,1\}^* \rightarrow x \in \mathbb{N} | x \leq 2^{128}$.
Zudem verlangen wir in unseren Protokollen jeweils, daß die Hashfunktion stark kollisionsfrei ist. Die ist gleichbedeutend mit der Aussage:
Eine Hashfunktion ist stark kollisionsfrei, wenn es berechnungsmässig praktisch unmöglich ist, Nachrichten M und M' mit $M \neq M'$ und $h(M') = h(M)$ zu finden.

2.5 Diskreter Logarithmus

Unter dem diskreten Logarihmus einer Zahl a versteht man folgende Berechnung:
$a = b^c \, mod \, d \Rightarrow c = log_b a \, mod \, d$
Das Problem besteht dabei darin, daß dieser schwer zu berechnen ist. Auf der Komplexität dieser Berechnung basiert letzlich die Sicherheit der folgenden Protokolle und damit des Fair Electronic Cash Systems.

3 Zero Knowledge Protokolle

Zero Knowledge Protokolle dienen dem Beweis von Wissen, ohne dieses explizit preiszugeben. Ein typisches Beispiel lässt sich wie folgt konstruieren:
Person A möchte Person B überzeugen, daß sie die Kombination zu einem Tresor kennt. Allerdings möchte Person A nicht die Kombination zum Öffnen der Tür preisgeben. Ein einfaches Zero Knowledge Protokoll würde nun wie folgt funktionieren:
Während der Tresor offen ist unterschreibt Person B ein Blatt Papier und legt es in den Tresor. Im Beisein von Person A und B wird der Tresor verschlossen. Die Person B kann sich davon überzeugen, daß der Tresor verschlossen ist. Nun verlässt Person B den Raum. Da Person A die Kombination zu dem Tresor kennt, kann sie den Tresor öffnen und das unterschriebene Blatt herausholen und Person B geben. Person B kann sich anschlissend über die Unversehrtheit des Tresors überzeugen. Damit kann A beweisen, daß sie die Kombination kennt ohne diese preiszugeben. In ähnlicher Weise funktionieren auch die folgenden mathematisch basierten Zero Knowledge Protokolle.

3.1 Vorraussetzungen

Bevor wir uns weiter Systeme und Algortihmen im Bereich der Zero Knowledge Protokolle ansehen können, müssen wir zuerst einige Vereinbarungen und Vorraussetzungen treffen bzw. festlegen. Die folgenden Algortihmen und Protokolle laufen im folgenden beschriebenen System.
Nach [1] benötigen wir für unsere Zero Knowledge Protokolle folgende Schritte:

1. Eine Primzahl p (Vorzugsweise suchen wir eine sichere Primzahl p (vgl. 2.1.1))
2. Eine Primzahl q, die $p-1$ teilt.
3. Eine Zahl g, die eine Untergruppe der Ordnung q in $\mathbb{Z}_p$ erzeugt.
4. Eine Hashfunktion h, die stark kollisionsfrei ist.
5. Eine Zahl x, die unser privater Schlüssel wird.
6. Eine Zahl y, die sich aus x berechnen lässt und unseren öffentlichen Schlüssel darstellt.

3.2 *ProofLog*

Ziel dieses Protokolls ist es, einer Person zu beweisen, daß man, in dem vorrausgesetzten System, einer anderen Person beweisen kann, daß man den privaten Schlüssel x kennt, ohne diesen preiszugeben.
Ausführung des Protokoll's

Beweisende Person	Datenaustausch	Überprüfende Personen
Bekannt sind p, q, y		
Wählt $r \in \{0, \ldots, q-1\}$ Berechnet $a = g^r$		
	$a \rightarrow$	
		Wählt $c \in \{0, \ldots, q-1\}$
	$\leftarrow c$	
Berechnet $b = r - c \cdot x$		
	$b \rightarrow$	
		Überprüft, ob $a = g^b \cdot y^c$

Beweis der Richtigkeit der Verifikationsbedingung
Es ist zu zeigen, daß $a = g^r = g^b \cdot y^c$ gilt.

$$\begin{aligned} a &= g^r && (1)\\ &= g^{b+c\cdot x} && (2)\\ &= g^b \cdot g^{c\cdot x} && (3)\\ &= g^b \cdot y^c && (4) \end{aligned}$$

3.3 $ProofLog_h$

Dieses Protokoll dient dazu, einer Person zu beweisen, daß einem der Logarithmus von $log_g y$ bekannt ist.

Ausführung des Protokolls

Beweisende Person	Datenaustausch	Überprüfende Personen
Bekannt sind p, q, y, sowie die Nachricht m und die Hashfunktion h		
Wählt $r \in \{0, \ldots, q-1\}$ Berechnet $a = g^r$ Berechnet $c = h(m \parallel a)$ Berechnet $b = r - c \cdot x$	$c, b \rightarrow$	Überprüft, ob $c = h(m \parallel g^b \cdot y^c)$ gilt.

Beweis der Richtigkeit der Verifikationsbedingung

Es ist zu zeigen, daß $c = h(m \parallel a) = h(m \parallel g^b \cdot y^c)$ gilt.

$$
\begin{aligned}
c &= h(m \parallel a) && (5)\\
&= h(m \parallel g^r) && (6)\\
&= h(m \parallel g^{b+c \cdot x}) && (7)\\
&= h(m \parallel g^b \cdot g^{c \cdot x}) && (8)\\
&= h(m \parallel g^b \cdot y^c) && (9)
\end{aligned}
$$

4 Signaturverfahren

Der Begriff einer Signatur ist eng mit dem Begriff der Unterschrift verknüpft. Eine Unterschrift ist dabei ein persönliches Merkmal, welches eine Person im Idealfall von jeder anderen Person unterscheidet. Im Bereich der Kryptographie dient eine Unterschrift dazu, Daten zu markieren. Die unterschreibende Person kann damit den Inhalt des Dokuments bestätigen etc. Wichtig ist hierbei, daß der Inhalt des Dokuments nicht geändert werden kann bzw. die Unterschrift nicht gefälscht werden kann. Im folgenden geht es um digitale Signaturen, die für die folgenden Prtokolle von Bedeutung sind.

4.1 Digitale Signaturen

Im Allgemeinen versteht man unter einer digitalen Signatur die Bildung eines Hashwertes einer gegebenen Nachricht M und deren gmeinsame Veröffentlichung mit $h(M)$. Da die Hashfunktion h stark kollisionsfrei ist, ist die Nachricht M vor einer Veränderung geschützt, da es berechnungsmässig schwer ist, eine Nachricht $M_2 \neq M_1$ zu finden, für die $h(M_2) = h(M_1)$ gilt. Im Allgemeinen arbeitet man in einem Public-Key Kryptosystem. Eine Person, die im Besitz des privaten Schlüssels und damit der privaten Transformation D_A ist, kann eine Nachricht M wie folgt signieren: Sie versendet $(M, D_A(h(M)))$. Ein Gegenüber kann dann mithilfe von der öffentlichen Transformation E_A die Signatur überprüfen, da $E_A(D_A(h(M))) = h(M)$. Die öffentliche Transformation kann nur dann gelingen, wenn die Nachricht $h(M)$ durch die authorisierte Person mit D_A verschlüsselt wurde. Die Sicherheit beruht in der Geheimhaltung von D_A.

4.2 Blindes Signaturverfahren

Blinde Signaturverfahren dienen dem digitalen signieren von Daten, wobei die unterschreibende Partei keine Kenntnis über den eigentlichen Inhalt der Nachricht ist. Ein Beispiel für diese Art von Signatur sieht wie folgt aus:
Die unterzeichnende Partei versiegelt einen Briefumschlag mit einem persönlichen Siegel, kennt jedoch den Inhalt des Briefes nicht.

4.2.1 Ausführung von $BlindLogSig_h$

Beweisende Person	Datenaustausch	Überprüfende Personen
Bekannt sind p, q, y, sowie die Nachricht m und die Hashfunktion h		
Wähle $\overline{r} \in \{0, \ldots, q-1\}$ Berechne $\overline{a} = g^{\overline{r}}$		
	$\overline{a} \rightarrow$	
		Wähle zufällige $u, v, w \in \{0, \ldots, q-1\}, u \neq 0$ Berechne $a = \overline{a}^u \cdot g^v \cdot y^w$ Berechne $c = h(m \parallel a)$ Berechne $\overline{c} = (c - w) \cdot u^{-1}$
	$\leftarrow \overline{c}$	
Berechne $\overline{b} = \overline{r} - \overline{c} \cdot x$		
	$\overline{b} \rightarrow$	
		Überprüfe, ob $\overline{a} = g^{\overline{b}} \cdot y^{\overline{c}}$ gilt. Berechne $b = u \cdot \overline{b} + v$

4.2.2 Beweis der Richtigkeit der Verifikationsbedingung von $BlindLogSig_h$

Zu zeigen ist, daß $c = h(m \parallel a) = h(m \parallel g^b)$ gilt.
Es gilt:

$$
\begin{aligned}
c &= h(m \parallel a) && (10)\\
&= h(m \parallel \overline{a}^u \cdot g^v \cdot y^w) && (11)\\
&= h(m \parallel (g^{\overline{b}} \cdot y^{\overline{c}})^u \cdot g^v \cdot y^w) && (12)\\
&= h(m \parallel (g^{\frac{b-v}{u}} \cdot y^{(c-w)\cdot u^{-1}})^u \cdot g^v \cdot y^w) && (13)
\end{aligned}
$$

$$
\begin{aligned}
&= h(m \parallel g^{b-v} \cdot y^{c-w} \cdot g^v \cdot y^w) && (14)\\
&= h(m \parallel g^{b-v+v} \cdot y^{c-w+w}) && (15)\\
&= h(m \parallel g^b \cdot y^c) && (16)
\end{aligned}
$$

4.2.3 Vorraussetzungen für $BlindLogEqSig_h$

Es gelten für den Algorithmus $BlindLogEqSig_h$ folgende Vorraussetzungen:

$$
\begin{aligned}
a_1 &= \overline{a_1}^u \cdot g^v \cdot y^w && (17)\\
a_2 &= (\overline{a_2}^u \cdot \overline{m}^v \cdot \overline{z}^w)^s \cdot (\overline{a_1}^u \cdot g^v \cdot y^w)^t && (18)\\
c &= u \cdot \overline{c} + w \Rightarrow \overline{c} = \frac{c-w}{u} && (19)\\
b &= u \cdot \overline{b} + v \Rightarrow \overline{b} = \frac{b-v}{u} && (20)\\
m &= \overline{m}^s \cdot g^t && (21)\\
z &= \overline{z}^s \cdot y^t && (22)\\
y &= g^x && (23)\\
\widetilde{y} &= \widetilde{g}^x && (24)\\
z &= m^x && (25)\\
a_1 &= g^b \cdot y^c && (26)\\
a_2 &= m^b \cdot z^c && (27)\\
a &= (a_1, a_2) && (28)
\end{aligned}
$$

4.2.4 Ausführung von $BlindLogEqSig_h$

Beweisende Person	Datenaustausch	Überprüfende Personen
Bekannt sind p, q, y		
		Wähle $s, t \in \{0, \ldots, q-1\}, s \neq 0$
		Berechne $\overline{m} = m^{\frac{1}{s}} \cdot g^{-\frac{t}{s}}$
	$\leftarrow \overline{m}$	
Wähle $\overline{r} \in \{0, \ldots, q-1\}$		
Berechne $\overline{z} = \overline{m}^x$		
Berechne $\overline{a} = (g^{\overline{r}}, \overline{m}^{\overline{r}})$		
	$\overline{a}, \overline{z} \rightarrow$	
		Wähle $u, v, w \in \{0, \ldots, q-1\} \mid u \neq 0$
		Berechne $a_1 = \overline{a_1}^u \cdot g^v \cdot y^w$
		Berechne $a_2 = (\overline{a_2}^u \cdot \overline{m}^v \cdot \overline{z}^w)^s \cdot (\overline{a_1}^u \cdot g^v \cdot y^w)^t$
		Berechne $z = \overline{z}^s \cdot y^t$
		Berechne $c = h(m \parallel z \parallel a_1 \parallel a_2)$
		Berechne $\overline{c} = (c-w) \cdot u^{-1}$
	$\leftarrow \overline{c}$	
Berechne $\overline{b} = \overline{r} - \overline{c} \cdot x$		
	$\overline{b} \rightarrow$	
		Berechne $\overline{a_1} = g^{\overline{b}} \cdot y^{\overline{c}}$
		Berechne $\overline{a_2} = \overline{m}^{\overline{b}} \cdot \overline{z}^{\overline{c}}$
		Berechne $b = u \cdot \overline{b} + v$
		Damit erhält man (z, c, b)

4.2.5 Beweis der Richtigkeit der Verifikationsbedingung von $BlindLogEqSig_h$

Zu zeigen ist, daß $a_1 = g^b \cdot y^c$ gilt.
Es gilt:

$$
\begin{aligned}
a_1 &= \overline{a_1}^u \cdot g^v \cdot y^w && (29)\\
&= (g^{\overline{b}} \cdot y^{\overline{c}})^u \cdot g^v \cdot y^w && (30)\\
&= (g^{\overline{r}-\overline{c}\cdot x} \cdot y^{\overline{c}})^u \cdot g^v \cdot y^w && (31)\\
&= (g^{\overline{r}})^u \cdot g^v \cdot y^w && (32)\\
&= (g^{\overline{b}+\overline{c}\cdot x})^u \cdot g^v \cdot y^w && (33)\\
&= (g^{\frac{b-v}{u}+\frac{c-w}{u}\cdot x})^u \cdot g^v \cdot y^w && (34)\\
&= g^{b-v+c\cdot x-w\cdot x} \cdot g^v \cdot y^w && (35)\\
&= g^{b-v+c\cdot x-w\cdot x} \cdot g^v \cdot g^{w\cdot x} && (36)\\
&= g^{b-v+c\cdot x-w\cdot x+v+w\cdot x} && (37)\\
&= g^{b+c\cdot x} && (38)\\
&= g^b \cdot y^c && (39)
\end{aligned}
$$

Des weiteren ist zu zeigen, daß $a_2 = m^b \cdot z^c$ gilt.
Es gilt:

$$
\begin{aligned}
a_2 &= (\overline{a_2}^u \cdot \overline{m}^v \cdot \overline{z}^w)^s \cdot (\overline{a_1}^u \cdot g^v \cdot y^w)^t && (40)\\
&= ((\overline{m}^{\overline{b}} \cdot \overline{z}^{\overline{c}})^u \cdot \overline{m}^v \cdot \overline{z}^w)^s \cdot (\overline{a_1}^u \cdot g^v \cdot y^w)^t && (41)\\
&= ((\overline{m}^{\overline{b}} \cdot \overline{z}^{\overline{c}})^u \cdot \overline{m}^v \cdot \overline{z}^w)^s \cdot ((g^{\overline{b}} \cdot y^{\overline{c}})^u \cdot g^v \cdot y^w)^t && (42)\\
&= ((\overline{m}^{\overline{b}} \cdot (\overline{m}^x)^{\overline{c}})^u \cdot \overline{m}^v \cdot (\overline{m}^x)^w)^s \cdot ((g^{\overline{b}} \cdot y^{\overline{c}})^u \cdot g^v \cdot y^w)^t && (43)\\
&= ((\overline{m}^{\overline{b}} \cdot (\overline{m}^x)^{(\frac{c-w}{u})})^u \cdot \overline{m}^v \cdot (\overline{m}^x)^w)^s \cdot ((g^{\overline{b}} \cdot y^{(\frac{c-w}{u})})^u \cdot g^v \cdot y^w)^t && (44)\\
&= ((\overline{m}^{(\frac{b-v}{u})} \cdot (\overline{m}^x)^{(\frac{c-w}{u})})^u \cdot \overline{m}^v \cdot (\overline{m}^x)^w)^s \cdot ((g^{(\frac{b-v}{u})} \cdot y^{(\frac{c-w}{u})})^u \cdot g^v \cdot y^w)^t && (45)\\
&= (((m^{\frac{1}{s}} \cdot g^{-\frac{t}{s}})^{(\frac{b-v}{u})} \cdot ((m^{\frac{1}{s}} \cdot g^{-\frac{t}{s}})^x)^{(\frac{c-w}{u})})^u \cdot (m^{\frac{1}{s}} \cdot g^{-\frac{t}{s}})^v \cdot && (46)\\
&\quad ((m^{\frac{1}{s}} \cdot g^{-\frac{t}{s}})^x)^w)^s \cdot ((g^{(\frac{b-v}{u})} \cdot y^{(\frac{c-w}{u})})^u \cdot g^v \cdot y^w)^t\\
&= (((m^{\frac{b-v}{s\cdot u}} \cdot g^{-\frac{t\cdot b-t\cdot v}{s\cdot u}}) \cdot (m^{\frac{x}{s}} \cdot g^{-\frac{t\cdot x}{s}})^{(\frac{c-w}{u})})^u \cdot (m^{\frac{1}{s}} \cdot g^{-\frac{t}{s}})^v \cdot && (47)\\
&\quad m^{\frac{x\cdot w}{s}} \cdot g^{-\frac{t\cdot x\cdot w}{s}})^s \cdot ((g^{(\frac{b-v}{u})} \cdot y^{(\frac{c-w}{u})})^u \cdot g^v \cdot y^w)^t\\
&= ((m^{\frac{b-v}{s}} \cdot g^{-\frac{t\cdot b-t\cdot v}{s}}) \cdot (m^{\frac{x}{s}} \cdot g^{-\frac{t\cdot x}{s}})^{c-w} \cdot (m^{\frac{1}{s}} \cdot g^{-\frac{t}{s}})^v \cdot && (48)\\
&\quad m^{\frac{x\cdot w}{s}} \cdot g^{-\frac{t\cdot x\cdot w}{s}})^s \cdot (g^{b-v} \cdot y^{c-w} \cdot g^v \cdot y^w)^t\\
&= (m^{b-v} \cdot g^{-t\cdot b+t\cdot v}) \cdot (m^x \cdot g^{-t\cdot x})^{c-w} \cdot (m \cdot g^{-t})^v \cdot && (49)\\
&\quad m^{x\cdot w} \cdot g^{-t\cdot x\cdot w} \cdot (g^{b-v} \cdot y^{c-w} \cdot g^v \cdot y^w)^t\\
&= m^{b-v} \cdot g^{-t\cdot b+t\cdot v} \cdot m^{c\cdot x-w\cdot x} \cdot g^{-t\cdot c\cdot x+t\cdot w\cdot x} \cdot m^v \cdot g^{-t\cdot v} \cdot && (50)\\
&\quad m^{x\cdot w} \cdot g^{-t\cdot x\cdot w} \cdot g^{b\cdot t-v\cdot t} \cdot y^{c\cdot t-w\cdot t} \cdot g^{v\cdot t} \cdot y^{w\cdot t}\\
&= m^{b-v+c\cdot x-w\cdot x+v+x\cdot w} \cdot g^{-t\cdot b+t\cdot v-t\cdot c\cdot x+t\cdot w\cdot x-t\cdot v-t\cdot x\cdot w+b\cdot t-v\cdot t+v\cdot t} \cdot y^{c\cdot t-w\cdot t+w\cdot t} && (51)\\
&= m^{b+c\cdot x} \cdot g^{-t\cdot c\cdot x} \cdot y^{c\cdot t} && (52)\\
&= m^{b+c\cdot x} \cdot g^{-t\cdot c\cdot x} \cdot g^{c\cdot t\cdot x} && (53)\\
&= m^{b+c\cdot x} \cdot g^{-t\cdot c\cdot x+c\cdot t\cdot x} && (54)\\
&= m^{b+c\cdot x} && (55)\\
&= m^b \cdot z^c && (56)
\end{aligned}
$$

5 Fair Electronic Cash System

Zuerst müssen wir die allgemeienn Vorraussetzungen für ein Elektronisches Geldsystem erstellen. Im Anschluss betrachten wir zwei verschiedene Ausführungsarten eines Systems. Das eine ist ein Online System, welches zwar einfacher zu implemntieren ist, bei der Bezahlung aber immer eine Verbindugn zur Bank benötigt, um die Gültigkeit der Münze zu überprüfen. Umgekehrt braucht das Offline System diese Verbindung nicht, benötigt aber im Gegenzug mehr Aufwand bei der Implementierung.

5.1 Systemvereinbarungen

Bank	Vertrauenspartei	Kunde	Verkäufer
Allgemein bekannt sind wie in vorangegangenen Algorithmen p, q			
Wählt eine primitive Wurzel g der Ordnung q in $\mathbb{Z}_p^*$	Wählt eine primitive Wurzel g_2 der Ordnung q in $\mathbb{Z}_p^*$	Wählt eine primitive Wurzel g_1 der Ordnung q in $\mathbb{Z}_p^*$	Wählt eine primitive Wurzel g_1 der Ordnung q in $\mathbb{Z}_p^*$
Wählt $x_B \in \{1, \ldots, q-1\}$	Wählt $x_T \in \{1, \ldots, q-1\}$	Wählt $x_C \in \{1, \ldots, q-1\}$	Wählt $x_S \in \{1, \ldots, q-1\}$
Berechnet $y_B = g^{x_B}$	Berechnet $y_T = g^{x_T}$	Berechnet $y_C = g^{x_C}$	Berechnet $y_S = g^{x_S}$
Veröffentlicht y_B	Veröffentlicht y_T	Veröffentlicht y_C	Veröffentlicht y_S

5.2 Kontoeröffnung

Wenn die Kundin ein Konto bei der Bank eröffnet, so beweisst sie der Bank mit Hilfe des Protokolls aus 3.2 ihre Identität. Dazu wird das Protokoll als $ProofLog(g_1, y_C)$ ausgeführt. Bei akzeptierter Identität speichert die Bank die Daten der Kundin zusammen mit der Kontonummer und dem öffentlichen Schlüssel y_C der Kundin.

5.3 Online Electronic Cash System

Das Online Electronic Cash System zeichnet sich durch die Eigenschaft aus, daß das Geschäft während der Verkaufstransaktion mit der Bank verbunden sein muß, um die Gültigkeit der Münze zu überprüfen.

5.3.1 Transaktion: Abheben einer Münze

Bank	Vertrauenspartei	Kunde
Wählt $\overline{r} \in \{0, \ldots, q-1\}$ Berechnet $\overline{a} = g^{\overline{r}}$ Sendet $\overline{a}$ an den Kunden		
		Wählt zufällige $u, v, w \in \{0, \ldots, q-1\}, u \neq 0$ Berechnet $a = \overline{a}^u \cdot g^v \cdot y^w$ Berechnet $c = h(a)$ Berechnet $\overline{c} = (c - w) \cdot u^{-1}$ Verschlüsselt (u, v, w) mit y_T Sendet $\overline{c}, E_{y_T}((u, v, w))$ an die Bank
Sendet $\overline{a}, \overline{c}, E_{y_T}((u, v, w))$ and die Vertrauenspartei		
	Überprüft ob $u \cdot \overline{c} + w = h(\overline{a}^u \cdot g^v \cdot y^w)$ Sendet Ergebnis an die Bank	
Wenn Ergebnis erfolgreich: Berechnet $\overline{b} = \overline{r} - \overline{c} \cdot x$ Belastet das Kundenkonto Sendet $\overline{b}$ an die Kundin		
		Überprüft, ob $\overline{a} = g^{\overline{b}} \cdot y^{\overline{c}}$ gilt. Berechnet $b = u \cdot \overline{b} + v$ Erhält die Münze als Signatur $\sigma = (c, b)$

5.3.2 Transaktion: Zahlung und Einzahlung auf das Konto

Kunde	Laden	Bank
Sendet $\sigma = (c, b)$ an den Laden		
	Berechnet $h(g^b \cdot y^c)$ Verifiziert ob $c = h(g^b \cdot y^c)$ gilt Sendet im positiven Fall die Münze an die Bank	
		Berechnet $h(g^b \cdot y^c)$ Verifiziert ob $c = h(g^b \cdot y^c)$ gilt Überprüft, ob die Münze bereits aufgetaucht ist. Im negativen Falle speichert sie die Münze und schreibt sie dem Kundenkonto gut.

5.3.3 Transaktion: Münzverfolgung und Feststellung des Eigentümers

Im Falle, daß die Münze bei der Überprüfung durch die Bank als Doublette entlarvt wird (vgl. 5.3.2) wird folgendes Protokoll ausgeführt:

Bank	Vertrauenspartei
Sendet doppelte Münze (c, b) an Vertrauenspartei	
	Berechnet $\overline{c} = (c - w) \cdot u^{-1}$
	Berechnet $\overline{b} = (b - v) \cdot u^{-1}$
	Berechnet $\overline{a} = g^{\overline{b}} \cdot y^{\overline{c}}$
	Ermittelt anhand von u, v, w den Kunden.

Das beschriebene Verfahren erfordert, daß die Vertrauenspartei für jeden gespeicherten Satz u, v, w die drei Berechnungen durchführt. Um dies zu vermeiden kann die Vertrauenspartei bereits bei der Abhebung der Münze den Wert $c = \overline{c} \cdot u + w$ berechnen. Die Suche nach c würde sich dadurch beschleunigen und die Berechnungen nach dem oben genannten Protokoll müssten nur für die gefundenen c durchgeführt werden, um die Identität des Kunden zu verifizieren.

5.3.4 Anonymität des Kunden

Die Anonymität des Kunden beruht auf der Tatsache, daß die Münze blind signiert wird und auf der Sicherheit des Verschlüsselung der Verschleierungsfaktoren u, v, w. Der Vertrauenspartei ist es jedoch möglich, diese Anonoymität mit Wissen der Münzdaten, die die Bank erhält, aufzuheben.

5.4 Offline Electronic Cash System

5.4.1 Zusatzprotokoll: $ProofLogEq_h$

Vorraussetzungen

$$y_1 = g_1^x \quad (57)$$
$$y_2 = g_2^x \quad (58)$$

Ausführung des Protokolls

Beweisende Person	Datenaustausch	Überprüfende Personen
Bekannt sind p, q, y_1, y_2, g_1, g_2, sowie die Hashfunktion h		
Wählt $\overline{r} \in \{0, \ldots, q-1\}$ Berechnet $a = (a_1, a_2) = (g_1^r, g_2^r)$ Berechnet $c = h(g_1 \parallel y_1 \parallel g_2 \parallel y_2 \parallel a_1 \parallel a_2)$ Berechnet $b = r - c \cdot x$	$c, b \rightarrow$	Überprüft, ob $c = h(g_1 \parallel y_1 \parallel g_2 \parallel y_2 \parallel g_1^b \cdot y_1^c \parallel g_2^b \cdot y_2^c)$

Beweis der Richtigkeit der Verifikationsbedingung
Es ist zu zeigen, daß $c = h(g_1 \parallel y_1 \parallel g_2 \parallel y_2 \parallel a_1 \parallel a_2) = h(g_1 \parallel y_1 \parallel g_2 \parallel y_2 \parallel g_1^b \cdot y_1^c \parallel g_2^b \cdot y_2^c)$ gilt.
Es gilt:

$$\begin{aligned} c &= h(g_1 \parallel y_1 \parallel g_2 \parallel y_2 \parallel a_1 \parallel a_2) && (59)\\ &= h(g_1 \parallel y_1 \parallel g_2 \parallel y_2 \parallel g_1^r \parallel g_2^r) && (60)\\ &= h(g_1 \parallel y_1 \parallel g_2 \parallel y_2 \parallel g_1^{b+c\cdot x} \parallel g_2^{b+c\cdot x}) && (61)\\ &= h(g_1 \parallel y_1 \parallel g_2 \parallel y_2 \parallel g_1^b \cdot g_1^{c\cdot x} \parallel g_2^b \cdot g_2^{c\cdot x}) && (62)\\ &= h(g_1 \parallel y_1 \parallel g_2 \parallel y_2 \parallel g_1^b \cdot y_1^c \parallel g_2^b \cdot y_2^c) && (63) \end{aligned}$$

5.4.2 Münzverfolgung aktivieren

Ausführung des Protokolls

Kunde	Bank
Wählt $s \in \{1, \ldots, q-1\}$ Berechnet $m = g_{Kunde} \cdot g_{Vertrauenspartei}^s$ Berechnet $d = y_{Vertrauenspartei}^s$ Berechnet $\overline{m} = m^{\frac{1}{s}} = g_{Kunde}^{\frac{1}{s}} \cdot g_{Vertrauenspartei}$ Berechnet (c, b) $= ProofLogEq_h(\frac{\overline{m}}{g_{Vertrauenspartei}}, g_{Kunde}, y_{Vertrauenspartei}, d)$ Sendet $(c, b, \frac{\overline{m}}{g_{Vertrauenspartei}}, g_{Kunde}, y_{Vertrauenspartei}, d)$	
	Verifiziert $(c, b) = ProofLogEq_h$ Speichert im positiven Fall d

Anmerkung
Das Protokoll $ProofLogEq_h$ wird hier mit dem gewählten Wert s anstelle des privaten Schlüssel's x verwendet.

5.4.3 Abheben einer Münze

Ausführung des Protokolls

Kunde	Bank
Wählt $r \in \{0, \ldots, q-1\}$ Berechnet $c_{Kunde,\text{Muenze}} = g^r_{Kunde}$ Berechnet $(z, c_1, b_1) =$ $BlindLogEqSig_h(c_{Kunde,\text{Muenze}}, g_{Kunde}, y_{Kunde}, m)$	
Die Münze ist $(c_1, b_1, c_{K,M}, g_{Kunde}, y_{Kunde}, m, z)$	Partizipiert in der Ausführung von $BlindLogEqSig_h$

5.4.4 Zahlung und Einzahlung auf das Konto

In einem Offline System ist es nicht möglich, die doppelte Zahlung zu verhindern, jedoch ist es möglich, diese zu entdecken. Wir betrachten deshalb zunächst den Zahlungsprozess.

Ausführung des Protokolls

Kunde	Laden
Bildet $M = y_{Laden} \parallel \mathsf{Zeit} \parallel (c_1, b_1)$ Berechnet $(c_2, b_2) =$ $ProofLog_h(M, \frac{m}{g_{Kunde}}, g_{Vertrauenspartei})$ Berechnet Muenze = $((c_1, b_1, c_{Kunde,\text{Muenze}}, g_{Kunde}, y_{Kunde}, m, z),$ $(c_2, b_2, M, \frac{m}{g_{Kunde}}, g_{Vertrauenspartei}))$	
	Der Laden überprüft: Ob M die korrekte Form hat. Ob $c_2 = h(M \parallel c_{Kunde,\text{Muenze}})$ Ob $(z, c_1, b_1) =$ $BlindLogEqSig_h(c_{Kunde,\text{Muenze}}, g_{Kunde}, y_{Kunde}, m)$ Ob $c_2 = h(M \parallel (\frac{m}{g_{Kunde}})^{b_2} g^{c_2}_{Vertrauenspartei})$ Im positiven Fall akzeptiert der Laden die Münze. und leitet diese an die Bank weiter. Die Bank überprüft die Münze erneut und sucht nach einer gleichen Münze. Wenn die Münze Bereits existiert führt sie das folgende Protokoll aus.

5.4.5 Münzverfolgung und Feststellung des Eigentümers

Es gelten folgende Operationen:
Die Bank stellt der Vertrauenspartei $d = y^s_{Vertrauenspartei}$ zur Verfügung.
Mit Hilfe von

$$g_{Kunde} \cdot d^{\frac{1}{x_{Vertrauenspartei}}} \quad = \quad g_K \cdot g^s_{Vertrauenspartei} = m \tag{64}$$

kann die Münze identifiziert werden.
Mit Hilfe von

$$\frac{m}{g_{Kunde}}^{x_{Vertrauenspartei}} \quad = \quad (g^s_{Vertrauenspartei})^{x_{Vertrauenspartei}} = y^s_{Vertrauenspartei} = d \tag{65}$$

entsprechend der Besitzer der Münze.

6 Bewertung

Dieser Abschnitt soll die Umsetzung des in [1] beschriebene Fair Electronic Cash System bewerten. Folgende Punkte sollen hier besonders herausgehoben werden:

1. Kommunikation zwischen den Protokollparteien
2. Datenaufkommen
3. Sicherheit und Anonymität

Kommunikation zwischen den Protokollparteien ist im Allgemeinen sehr groß. Die Menge der einzelnen auszuführenden Protokolle erfordert es, daß die einzelnen Partein häufig Daten austauschen. Die folgende Tabelle verdeutlicht dies für das Online Protokoll.

Bank	Vertrauenspartei	Kunde	Laden	Transfer
$\overline{r}, \overline{a}$				$\overline{a}$
		u, v, w, a, c		$\overline{c}, E_{y_T}((u, v, w))$
	u, v, w			$\overline{a}, \overline{c}, E_{y_T}((u, v, w))$
$\overline{b}$				Verify
		$\overline{a}, b$		$\overline{b}$
			$c = h(g^b y^c)$	$\sigma = (c, b)$
$c = h(g^b \cdot y^c)$				(c, b)
	$\overline{c}, \overline{b}, \overline{a}$			(c, b)

Im Laufe der Duzrchführung des Protokolls findet dabei 7 mal ein Datenaustausch statt.
Das **Datenaufkommen** während eines vollständigen Zyklus ist abhängig von der Größe der gewählten Zahlen und der Hashfunktionen. Folgende Werte sind eine erste Gute Näherung:
Folgende Daten müssen gespeichert werden:
Bei der Bank (2 Werte): $Konto_{Kunde}, Kunde_{Laden}$
Bei allen Parteien (40 Werte): $p, q, g_{Bank}, g_{Vertrauenspartei}, g_{Kunde}, g_{Laden}$ sowie
$y_{Bank}, y_{Vertrauenspartei}, y_{Kunde}, y_{Laden}$
Bei der jeweiligen Partei (4 Werte): $x_{Bank}, x_{Vertrauenspartei}, x_{Kunde}, x_{Laden}$
Während der Ausführung der Protokolle zusätzlich:
Temporär: $r, a, c, b\ \overline{r}_{Bank}, \overline{a}_{Bank}, u_{Kunde}, v_{Kunde}, w_{Kunde}, a_{Kunde}, c_{Kunde}, \overline{c}_{Kunde}$
$k_{u_{Kunde}}, k_{v_{Kunde}}, k_{w_{Kunde}}, a_{u_{Kunde}}, b_{u_{Kunde}}, a_{v_{Kunde}}, b_{v_{Kunde}}, a_{w_{Kunde}}, b_{w_{Kunde}}$
$z_{u_{Kunde}}, z_{v_{Kunde}}, z_{w_{Kunde}}, u_{Kunde}, v_{Kunde}, w_{Kunde}$
$\overline{b}_{Bank}, \overline{a}, b_{Kunde}, \sigma, \overline{c}, \overline{b}, \overline{a}$
Wir sehen, daß zusammen 82 Werte gespeichert werden müssen. Geht man von einem System von 1024 Bit Zahlen aus, so errechnet sich der Speicherplatzbedarf für eine Münze zu $82 \cdot 1024 = 10496$ Byte.
Weiter ist zu beachten, daß die Bank Informationen über jede Münze speichern muss und die Vertrauenspartei die Werte u, v, w für jede Münze.
Eine Münze ist durch das gegebene Verfahren dabei nicht äquivalent zu einer Münze, wie wir sie heute kennen. Jede neue Transaktion bedarf einer neuen Münze, da es sonst möglich ist Informationen über den privaten Schlüssel zu gewinnen.
Die Menge der Heute im Umlauf befindlichen Münzen liegt bei weit über 100000000. Geht man davon aus, daß eine Münze pro Tag 1 mal ausgegeben wird, so bedeetet dies einen Jahresbedarf von $365 \cdot 100000000 = 36500000000$ Münzen. Man erkennt, daß damit der Gesamtspeicherbedarf der Münzen stark anwächst. 365 Terabyte sind ein erster Ansatz.
Der Punkt **Sicherheit** ist von besonderer Bedeutung. Die Anonymität eines Kunden kann hier immer von der Vertrauenspartei widerrufen werden: Dies ist gerade im Hinblick auf Anonymität des Kunden

problematisch. Die Vertrauenspartei besitzt immer genügend Informationen, um für jede beliebige Münze, unabhängig ihrer doppelten Ausgabe, eine Verknüpfung von Münze und Kunden herzustellen. Dies ist sicherlich nicht im Interesse des Kunden, da sich hier besonders die Frage stellt, wer die Rolle der Vertrauenspartei einnehmen sollte.

7 Fazit

Zusammenfassend lässt sich sagen, daß das vorgestellte System von einigem wissenschaftlichen Interesse ist, jedoch viele Fragen und Probleme offen lässt. Gerade im Hinblick auf Sicherheit ist es doch sehr fraglich, ob dieses System jemals von Kunden angenommen werden würde.
Probleme, die bei der Implementierung aufkommen, lassen sich in Zukunft sicherlich einfacher handhaben (z.B. das Datenaufkommen), andere jedoch bleiben immer erhalten. Auch ist bei diesem System bis jetzt nur davon ausgegangen worden, daß es eine Bank gibt und eine Münze in genau dem Zyklus Bank -> Kunde -> Verkäufer -> Bank bewegt wird. Die nötige Flexibilität des Systems ist hier bei weitem nicht gegeben. Der Aufwand von bis zu 10 interaktiven Transaktionen und der Datenaustausch von bis zu 80 Variablen ist für kleine Summen Geldes wohl kaum gerechtfertigt.
Die Sicherheit ist ein weiterer Punkt der mit großer Wahrscheinlichkeit für Aufregung sorgt. Die Tatsache, daß die Anonymität so einfach widerrufen werden kann ist ein grosses Manko.

Literatur

[1] **Introduction to Crypthography - Principles and Applications**, *Hans Delfs, Helmut Knebl*, Springer Verlag 2002, ISBN: 3540422781

[2] **Stefan Brands' System of Digital Cash**, *J. Orlin Grabbe*, http://orlingrabbe.com/stefbrdc.htm

[3] **Zero Knowledge Protocols and Small Systems**, *Hannu A. Aronsson*, Department of Computer Science, Helsinki University of Technology, http://www.tml.tkk.fi/Opinnot/Tik-110.501/1995/zeroknowledge.html

[4] **Algorithms and Computation**, *Tetsuo Asano*, Springer Verlag 1996, ISBN: 3540620486, Seite 436

[5] **Electronic Commerce**, *Ravi Kalakota*, Addison-Wesley Professional 1996, ISBN: 0201880679, Seite 161

[6] **Kryptographie - Grundlagen, Algorithmen, Protokolle**, *Dietmar Wätjen*, Spektrum Akademischer Verlag 2004, ISBN: 3827414318

[7] **Lineare Algebra - Eine Einführung in die Wissenschaft der Vektoren, Abbildungen und Matrizen**, *Albrecht Beutelspacher*, Vieweg Verlag 2003, 6. Auflage, ISBN: 352856508X

[8] **Verteilte System - Grundlagen und Paradigmen**, *Andrew S. Tanenbaum, Maarten Van Stehen*, Pearson Studium 2003, ISBN: 3827370574, Seite 544 - 546

8 Anhang

8.1 Quellcode in Python

```
#Liefert einen Text in ASCII Darstellung für eine gegebene Zahl. (Umkehrung von numberastext)


def textasnumber(m):
    g=0
    for i in range(len(m)):
        g=g+ ord(m[i:i+1])*pow(256,i)
    return g

#Liefert einen Zahlendarstellung eines Strings (basierend auf einem Text in ASCII Darstellung)

def numberastext(n):
    s=""
    while n>=1:
        n1=n
        n2=n1%256
        st=chr(n2)
        s=s + st
        n=n/256
    return s

#Berechnet den ggT mit Hilfe des euklidischen Algorithmus (3.2)

def ggt(a,n):
    g0=n
    g1=a
    i=1
    while g1<>0:
        g2=g0%g1
        g0=g1
        g1=g2
    return g0

#Berechnet as Inverese a^-1mod n mit Hilfe des erweiterten euklidischen Algorithmus (3.3)

def inva(a,n):
    g0=n
    g1=a
    v0=0
    v1=1
    while g1<>0:
        y=g0/g1
        g2=g0-y*g1
        v2=v0-y*v1
        g0=g1
        g1=g2
        v0=v1
        v1=v2
    x=v0
    if x<=0:x=x+n
    return x

#Liefert 1, falls n eine Primzahl ist, 0 falls n keine Primzahl ist. Der Test basiert auf Rabin
```

```
Miller und ist probabilistisch (5.6).

def millerrabin(n):
    from random import randint
    m=n-1
    l=0
    while m%2==0:
        m=m/2
        l=l+1
    x=0
    a=0
    while x<=n/2:
        a=a+1
        x=x+randint(0,1)*pow(2,a)
    x0=pow(x,m,n)
    if x0==1 or x0==n-1:
        return 1
    for i in range(1, l-1):
        x1=pow(x0,2,n)
        if x1==n-1:
            return 1
        elif x1==1:
            return 0
        x0=x1
    return 0

#Liefert eine zufällige Primzahl der Bitlänge n oder kleiner

def longprim(n):
    from random import randint
    isprim=0
    while isprim==0:
        p=1
        for a in range(1,n):
            p=p+randint(0,1)*pow(2,a)
        isprim=millerrabin(p)
    return p

#Liefert eine sichere Primzahl der Bitlänge n oder kleiner (7.3)

def secureprim(n):
    isprim=0
    while isprim==0:
        q=longprim(n-1)
        p=q*2+1
        isprim=millerrabin(p)
    return p

#Liefert ein zufälligen Generator g für den gilt g^q mod p=1

def findgenerator(p,q):
    from random import randint
    isgenerator=0
    while isgenerator==0:
        g=1
        a=0
        while g<=p/2:
```

```
            a=a+1
            g=g+randint(0,1)*pow(2,a)
        g=g%p
        if pow(g,q,p)==1:
            isgenerator=1
    return g

#Liefert einen möglichen geheimen Schlüssel

def findsecretkey(q):
    from random import randint
    x=0
    a=0
    while x<=q-1:
        a=a+1
        x=x+randint(0,1)*pow(2,a)
    x=x%(q-1)
    return x

#Berechnet bei gegebenen Generator, dem geheimen Schlüssel und der Primzahl p einen öffentlichen
Schlüssel.

def getpublickey(g,x,p):
    y=pow(g,x,p)
    return y

#Liefert den Hashwert nach MD5 einer gegebenen Nachricht m.

def hash(m):
    import md5
    h=long(md5.new(m).hexdigest(),16)
    return h

#Liefert eine Zufallszahl, die kleiner als die gegebene Zahl q ist.

def randomnumbersmaller(q):
    from random import randint
    x=0
    a=0
    while x<=q-1:
        a=a+1
        x=x+randint(0,1)*pow(2,a)
    x=x%(q-1)
    return x

#Liefert eine Zufallszahl, die kleiner als die gegebene Zahl q ist.

def rns(q):
    from random import randint
    x=0
    a=0
    while x<=q-1:
        a=a+1
        x=x+randint(0,1)*pow(2,a)
    x=x%(q-1)
    return x
```

```
#Liefert eine Zufallszahl, die kleiner als die gegebene Zahl q ist und grÃűsser gleich 1 ist.

def rnspos(q):
    x=0
    while x==0:
        x=rns(q)
    return x

#(c,b)=ProofLogEq_h(g1,y1,g2,y2)
#r = Random number < q
#p = Prime number p
#q = Prime number q
#x = Secret x

def prooflogeqh(g1,y1,g2,y2,r,p,q,x):
    print "ProofLogEq_h"
    print "g1="+str(g1)
    print "y1="+str(y1)
    print "g2="+str(g2)
    print "y2="+str(y2)
    print "r="+str(r)
    print "p="+str(p)
    print "q="+str(q)
    print "x="+str(x)
    print "---"
    a1=pow(g1,r,p)
    a2=pow(g2,r,p)
    m=str(g1)+str(y1)+str(g2)+str(y2)+str(a1)+str(a2)
    c=hash(m)%q
    b=(r-c*x)%q
    cblist=[]
    cblist.append(c)
    cblist.append(b)
    print "(c,b)="
    print cblist
    print "ProofLogEq_h End"
    return cblist

# Protocol 4.17
#(c,b)=ProofLog(g,y)
#r = Random number < q
#c = Random number < q
#p = Prime number p
#q = Prime number q
#x = Secret key x | y=g^x

def prooflog(g,y,r,c,p,q,x):
    print "ProofLog"
    print "g="+str(g)
    print "y="+str(y)
    print "r="+str(r)
    print "c="+str(c)
    print "p="+str(p)
    print "q="+str(q)
    print "x="+str(x)
    print "---"
    a=pow(g,r,p)
```

```
    b=(r-c*x)%q
    cblist=[]
    cblist.append(c)
    cblist.append(b)
    print "(c,b)="
    print cblist
    print "ProofLog End"
    return cblist

#Protocol 4.17a
#(c,b)=ProofLog_h(g,y,m)
#m = Message
#r = Random number < q
#p = Prime number p
#q = Prime number q
#x = Secret key x | y=g^x

def prooflogh(g,y,m,r,p,q,x):
    print "ProofLog_h"
    print "g="+str(g)
    print "y="+str(y)
    print "m="+str(m)
    print "r="+str(r)
    print "p="+str(p)
    print "q="+str(q)
    print "x="+str(x)
    print "---"
    a=pow(g,r,p)
    message=str(m)+str(a)
    c=hash(message)
    b=(r- c*x)%q
    cblist=[]
    cblist.append(c)
    cblist.append(b)
    print "(c,b)="
    print cblist
    print "ProofLog_h End"
    return cblist

#Protocol 4.17b
#(c,b)=ProofLog_h(g,y)
#r = Random number < q
#p = Prime number p
#q = Prime number q
#x = Secret key x | y=g^x

def prooflogh(g,y,r,p,q,x):
    print "ProofLog_h"
    print "g="+str(g)
    print "y="+str(y)
    print "r="+str(r)
    print "p="+str(p)
    print "q="+str(q)
    print "x="+str(x)
    print "---"
    a=pow(g,r,p)
    message=str(a)
```

```
    c=hash(message)
    b=(r- c*x)%q
    cblist=[]
    cblist.append(c)
    cblist.append(b)
    print "(c,b)="
    print cblist
    print "ProofLog_h End"
    return cblist

#Protocol 4.18
#(c,b)=BlindLogSig_h(m)
#m = Message
#r = Random number r_quer < q
#u = Random number u < q and u <> 0
#v = Random number v < q
#w = Random number w < q
#p = Prime number p
#q = Prime number q
#x = Secret key x | y=g^x
#g = Generator g | y=g^x
#y = Public key y | y=g^x

def blindlogsigh(m,r,u,v,w,p,q,g,x,y):
    print "BlindLogSig_h"
    print "m="+str(m)
    print "r="+str(r)
    print "u="+str(u)
    print "v="+str(v)
    print "w="+str(w)
    print "p="+str(p)
    print "q="+str(q)
    print "g="+str(g)
    print "x="+str(x)
    print "y="+str(y)
    print "---"
    rq=r
    aq=pow(g,rq,p)
    a=(pow(aq,u,p)*pow(g,v,p)*pow(y,w,p))%p
    message=str(m)+str(a)
    c=hash(message)
    cq=((c-w)*inva(u,q))%q
    bq=(rq-cq*x)%q
    b=(u*bq+v)%q
    cblist=[]
    cblist.append(c)
    cblist.append(b)
    print "(c,b)="
    print cblist
    print "BlindLogSig_h End"
    return cblist

#Protocol 4.19
#(z,c,b)=BlindLogEqSig_h(g,y,m)
#m = Message
#s = Random number s < q and s <> 0
#t = Random number t < q
```

```
#r = Random number r_quer < q
#u = Random number u < q and u <> 0
#v = Random number v < q
#w = Random number w < q
#p = Prime number p
#q = Prime number q
#x = Secret key x | y=g^x

def blindlogeqsigh(g,y,m,s,t,r,u,v,w,p,q,x):
    print "BlindLogEqSig_h"
    print "g="+str(g)
    print "y="+str(y)
    print "m="+str(m)
    print "s="+str(s)
    print "t="+str(t)
    print "r="+str(r)
    print "u="+str(u)
    print "v="+str(v)
    print "w="+str(w)
    print "p="+str(p)
    print "q="+str(q)
    print "x="+str(x)
    print "---"
    rq=r
    mn=textasnumber(m)
    s_1=inva(s,q-1)
    mq=(pow(mn,s_1,p)*pow(pow(inva(g,p),t,p),s_1,p))%p
    zq=pow(mq,x,p)
    a1q=pow(g,rq,p)
    a2q=pow(mq,rq,p)
    a1=(pow(a1q,u,p)*pow(g,v,p)*pow(y,w,p))%p
    a2=((pow((pow(a2q,u,p)*pow(mq,v,p)*pow(zq,w,p))%p,s,p))*pow((pow(a1q,u,p)*pow(g,v,p)*pow(y,w,p
    z=(pow(zq,s,p)*pow(y,t,p))%p
    message=str(m)+str(z)+str(a1)+str(a2)
    c=hash(message)
    cq=((c-w)*inva(u,q))%q
    bq=(rq-cq*x)%q
    a1q2=(pow(g,bq,p)*pow(y,cq,p))%p
    a2q2=(pow(mq,bq,p)*pow(zq,cq,p))%p
    b=(u*bq+v)%q
    zcblist=[]
    zcblist.append(z)
    zcblist.append(c)
    zcblist.append(b)
    print "(z,c,b)="
    print zcblist
    print "BlindLogEqSig_h End"
    return zcblist

#Protocol 4.19 Variation
#(z,c,b)=BlindLogEqSig_h(M,g,y,m)
#M = Message 1
#m = Message 2
#s = Random number s < q and s <> 0
#t = Random number t < q
#r = Random number r_quer < q
#u = Random number u < q and u <> 0
```

```
#v = Random number v < q
#w = Random number w < q
#p = Prime number p
#q = Prime number q
#x = Secret key x | y=g^x

def blindlogeqsigh(mm,g,y,m,s,t,r,u,v,w,p,q,x):
    print "BlindLogEqSig_h"
    print "M="+str(mm)
    print "g="+str(g)
    print "y="+str(y)
    print "m="+str(m)
    print "s="+str(s)
    print "t="+str(t)
    print "r="+str(r)
    print "u="+str(u)
    print "v="+str(v)
    print "w="+str(w)
    print "p="+str(p)
    print "q="+str(q)
    print "x="+str(x)
    print "---"
    rq=r
    mn=textasnumber(str(m))
    s_1=inva(s,q-1)
    mq=(pow(mn,s_1,p)*pow(pow(inva(g,p),t,p),s_1,p))%p
    zq=pow(mq,x,p)
    a1q=pow(g,rq,p)
    a2q=pow(mq,rq,p)
    a1=(pow(a1q,u,p)*pow(g,v,p)*pow(y,w,p))%p
    a2=((pow((pow(a2q,u,p)*pow(mq,v,p)*pow(zq,w,p))%p,s,p))*pow((pow(a1q,u,p)*pow(g,v,p)*pow(y,w,p
    print "a2="+str(a2)
    z=(pow(zq,s,p)*pow(y,t,p))%p
    print z
    print pow(y,t,p)
    message=str(mm)+str(m)+str(z)+str(a1)+str(a2)
    print message
    c=hash(message)
    cq=((c-w)*inva(u,q))%q
    bq=(rq-cq*x)%q
    a1q2=(pow(g,bq,p)*pow(y,cq,p))%p
    a2q2=(pow(mq,bq,p)*pow(zq,cq,p))%p
    b=(u*bq+v)%q
    zcblist=[]
    zcblist.append(z)
    zcblist.append(c)
    zcblist.append(b)
    print "(z,c,b)="
    print zcblist
    print "BlindLogEqSig_h End"
    return zcblist

#(g,x,y)=get_gxy(p,q)
#Returns a set of g,x,y | y=g^x
#p = Secure prime Number p
#q = Prime Number q | p=2q+1
```

```
def get_gxy(p,q):
    print "get_gxy"
    print "p="+str(p)
    print "q="+str(q)
    print "---"
    g=findgenerator(p,q)
    x=findsecretkey(q)
    y=getpublickey(g,x,p)
    gxylist=[]
    gxylist.append(g)
    gxylist.append(x)
    gxylist.append(y)
    print "(g,x,y)="
    print gxylist
    print "get_gxy End"
    return gxylist

#(a,b)=elgamalencrypt(k,m,g,y,p)
#k=Random number smaller p-2
#m=Message
#g=Generator | y=g^x
#y=Public Key | y=g^x
#p=Prime number p

def elgamalencrypt(k,m,g,y,p):
    print "ElGamalEncrypt"
    print "k="+str(k)
    print "m="+str(m)
    print "g="+str(g)
    print "y="+str(y)
    print "p="+str(p)
    print "---"
    a=pow(g,k,p)
    b=(m*pow(y,k,p))%p
    ablist=[]
    ablist.append(a)
    ablist.append(b)
    print "(a,b)="
    print ablist
    print "ElGamalEncrypt End"
    return ablist

#m=elgamaldecrypt(a,b,x,p)
#(a,b)=Encrypted message
#x=Private Key | y=g^x
#p=Prime number p

def elgamaldecrypt(a,b,x,p):
    print "ElGamalDecrypt"
    print "a="+str(a)
    print "b="+str(b)
    print "x="+str(x)
    print "p="+str(p)
    print "---"
    z=inva(pow(a,x,p),p)
    m=(z*b)%p
    print "m="
```

```
    print m
    print "ElGamalDecrypt End"
    return m

#Protocol 4.20
#(c,b)=Withdrawal(gb,xb,yb,gv,xv,yv,uk,vk,wk,rqb,p,q)
#gb=Generator g_Bank | y=g^x
#xb=Private key x_Bank | y=g^x
#yb=Public key y_Bank | y=g^x
#gv=Generator g_Vertrauenspartei | y=g^x
#xv=Private key x_Vertrauenspartei | y=g^x
#yv=Public key y_Vertrauenspartei | y=g^x
#uk=Random number u < q and u <> 0 (Kunde)
#vk=Random number v < q (Kunde)
#wk=Random number w < q (Kunde)
#rgb=Random number r_quer_Bank < q
#p=Prime number p
#q=Prime number q | 2q+1=p

def onlinewithdrawal(gb,xb,yb,gv,xv,yv,uk,vk,wk,rqb,p,q):
    print "Online Withdrawal"
    print "gb="+str(gb)
    print "xb="+str(xb)
    print "yb="+str(yb)
    print "gv="+str(gv)
    print "xv="+str(xv)
    print "yv="+str(yv)
    print "uk="+str(uk)
    print "vk="+str(vk)
    print "wk="+str(wk)
    print "rqb="+str(rqb)
    print "p="+str(p)
    print "q="+str(q)
    print "---"
    aqb=pow(gb,rqb,p)
    ak=(pow(aqb,uk,p)*pow(gb,vk,p)*pow(yb,wk,p))%p
    ck=hash(str(ak))
    cqk=((ck-wk)*inva(uk,q))%q
    eku=randomnumbersmaller(p-2)
    ekv=randomnumbersmaller(p-2)
    ekw=randomnumbersmaller(p-2)
    abuk=elgamalencrypt(eku,uk,gv,yv,p)
    abvk=elgamalencrypt(ekv,vk,gv,yv,p)
    abwk=elgamalencrypt(ekw,wk,gv,yv,p)
    mu=elgamaldecrypt(abuk[0],abuk[1],xv,p)
    mv=elgamaldecrypt(abvk[0],abvk[1],xv,p)
    mw=elgamaldecrypt(abwk[0],abwk[1],xv,p)
    vucquerpw=((mu*cqk)+mw)%q
    vhaqugvyw=hash(str((pow(aqb,mu,p)*pow(gb,mv,p)*pow(yb,mw,p))%p))%q
    bqb=(rqb-cqk*xb)%q
    aqk=(pow(gb,bqb,p)*pow(yb,cqk,p))%p
    bk=((uk*bqb)+vk)%q
    cblist=[]
    cblist.append(ck)
    cblist.append(bk)
    print "(c,b)="
    print cblist
```

```
    print "Online Withdrawal End"
    return cblist

#(bool)=onlinepayment(c,b,gb,yb,p)
#(c,b)=Signature of coin
#gb=Generator g_Bank | y=g^x
#yb=Public key y_Bank | y=g^x
#p=Prime number p

def onlinepayment(c,b,gb,yb,p):
    print "Online Payment"
    print "c="+str(c)
    print "b="+str(b)
    print "gb="+str(gb)
    print "yb="+str(yb)
    print "p="+str(p)
    print "---"
    c2=hash(str((pow(gb,b,p)*pow(yb,c,p))%p))
    istrue=0
    if c2==c:
        print "Münze ist gültig"
        istrue=1
    print "(bool)="
    print istrue
    print "Online Payment End"
    return istrue

#(aq,bq,cq)=onlinetracecoin(c,b,u,v,w,gb,yb,p,q)
#(c,b)=Signature of coin
#u=Random number u < q and u <> 0 (Kunde)
#v=Random number v < q (Kunde)
#w=Random number w < q (Kunde)
#gb=Generator g_Bank | y=g^x
#yb=Public key y_Bank | y=g^x
#p=Prime number p
#q=Prime number q | 2q+1=p

def onlinetracecoin(c,b,u,v,w,gb,yb,p,q):
    print "OnlineTraceCoin"
    print "c="+str(c)
    print "b="+str(b)
    print "u="+str(u)
    print "v="+str(v)
    print "w="+str(w)
    print "gb="+str(gb)
    print "yb="+str(yb)
    print "p="+str(p)
    print "q="+str(q)
    print "---"
    cq=((c-w)*inva(u,q))%q
    bq=((b-v)*inva(u,q))%q
    aq=(pow(gb,bq,p)*pow(yb,cq,p))%p
    abclist=[]
    abclist.append(aq)
    abclist.append(bq)
    abclist.append(cq)
    print "(aq,bq,cq)="
```

```
    print abclist
    print "OnlineTraceCoin End"
    return abclist

def verify_prooflogeqh(c,b,g1,y1,g2,y2,p,q):
    hm1=(pow(g1,b,p)*pow(y1,c,p))%p
    hm2=(pow(g2,b,p)*pow(y2,c,p))%p
    hm=str(g1)+str(y1)+str(g2)+str(y2)+str(hm1)+str(hm2)
    print hm
    valid=0
    if hash(hm)%q==c:
        valid=1
    return valid

def verify_blindlogeqsigh(z,c,b,mm,m,g,y,p,q):
    hm1=(pow(g,b,p)*pow(y,c,p))%p
    hm2=(pow(m,b,p)*pow(z,c,p))%p
    hm=str(mm)+str(m)+str(z)+str(hm1)+str(hm2)
    print hm
    valid=0
    if hash(hm)%q==c:
        valid=1
    return valid

#Hauptprogramm

bits=128
#Definiere Kunden und Verkäufer Konten
konto_kunde=100
konto_laden=100
p=secureprim(bits)

print "p="+str(p) q=(p-1)/2 print "q="+str(q)

#Finde Generatoren für Bank, Vertrauenspartei, Kunde und Verkäufer
gxy_b=get_gxy(p,q)
gxy_k=get_gxy(p,q)
gxy_v=get_gxy(p,q)
gxy_l=get_gxy(p,q)

r=rns(q)
c=rns(q)
prooflog(gxy_k[0],gxy_k[2],r,c,p,q,gxy_k[1])

rqb=rns(q)
uk=rns(q)
vk=rns(q)
wk=rns(q)

coinsignature=onlinewithdrawal(gxy_b[0],gxy_b[1],gxy_b[2],gxy_v[0],gxy_v[1],gxy_v[2],uk,vk,wk,rqb
validcoin=onlinepayment(coinsignature[0],coinsignature[1],gxy_b[0],gxy_b[2],p)
trace=onlinetracecoin(coinsignature[0],coinsignature[1],uk,vk,wk,gxy_b[0],gxy_b[2],p,q)
```

8.2 Beispiel: Online Electronic Cash System

Wahl der Primzahlen q, p

$$p = 437283104004656906268158377384770524 87$$
$$q = 218641552002328453134079188692385262 43$$

Die Bank wählt:

$$(g, x, y) =$$
$$41478105403780723041241862255085661924,$$
$$20814560258663031719184224103261819544,$$
$$37279760573303776369176024013875303803$$

Der Kunde wählt:

$$(g, x, y) =$$
$$40772352138427768345793757880802376 78,$$
$$14181484444369094049539890782867593598,$$
$$24294138801712989646858496538901562349$$

Die Vertrauenspartei wählt:

$$(g, x, y) =$$
$$32458827877281731233893560938140947969,$$
$$43907678176974499004319307602851643 50,$$
$$62701290545701093748319683729012023 07$$

Der Laden wählt:

$$(g, x, y) =$$
$$34184959763056268713447403936277617521,$$
$$18776106158095696703340149081359682908,$$
$$32881418409226631359941994874906222132$$

Zur Kontoeröffnung führen die Bank und der Kunde das Protokoll $ProofLog_h$ aus:

$$g = 40772352138427768345793757880802376 78$$
$$y = 24294138801712989646858496538901562349$$
$$r = 90682195637676830893209580750084331 38$$
$$c = 17266635994746656792623632824895366568$$
$$p = 43728310400465690626815837738477052487$$
$$q = 21864155200232845313407918869238526243$$
$$x = 14181484444369094049539890782867593598$$
$$(c, b) =$$
$$17266635994746656792623632824895366568,$$
$$59014758427921608017600710380865187 7$$

Der Kunde hebt die Münze von seinem Konto ab:

$$
\begin{aligned}
gb &= 41478105403780723041241862255085661924 \\
xb &= 20814560258663031719184224103261819544 \\
yb &= 37279760573303776369176024013875303803 \\
gv &= 32458827877281731233893560938140947969 \\
xv &= 43907678176974499004319307602851643 50 \\
yv &= 62701290545701093748319683729012023 07 \\
uk &= 93144914337485145563440377462648694 08 \\
vk &= 41463444199398386573694968739282898 36 \\
wk &= 10861736086037713971498946182525584186 \\
rqb &= 43083866470371470021178515171864703 10 \\
p &= 43728310400465690626815837738477052487 \\
q &= 21864155200232845313407918869238526243
\end{aligned}
$$

Der Kunde verschlüsselt seinen Blendungsfaktor u mithilfe des ElGamal Verfahrens:

$$
\begin{aligned}
k &= 23348357707503538227199579973361780000 \\
m &= 93144914337485145563440377462648694 08 \\
g &= 32458827877281731233893560938140947969 \\
y &= 62701290545701093748319683729012023 07 \\
p &= 43728310400465690626815837738477052487 \\
&\qquad (a, b) = \\
&\quad 23771262264845457384097857517648014102 \\
&\quad 39998965684368905572008766958733578665
\end{aligned}
$$

Der Kunde verschlüsselt seinen Blendungsfaktor v mithilfe des ElGamal Verfahrens:

$$
\begin{aligned}
k &= 23486492678233481772706798066248229888 \\
m &= 41463444199398386573694968739282898 36 \\
g &= 32458827877281731233893560938140947969 \\
y &= 62701290545701093748319683729012023 07 \\
p &= 43728310400465690626815837738477052487 \\
&\qquad (a, b) = \\
&\quad 30815119520216007042916868085066720498 \\
&\quad 19567718565837359464710612078229175429
\end{aligned}
$$

Der Kunde verschlüsselt seinen Blendungsfaktor w mithilfe des ElGamal Verfahrens:

$$
\begin{aligned}
&\qquad ElGamalEncrypt \\
k &= 26379215799504644764443170680155966838 \\
m &= 10861736086037713971498946182525584186 \\
g &= 32458827877281731233893560938140947969 \\
y &= 62701290545701093748319683729012023 07 \\
p &= 43728310400465690626815837738477052487 \\
&\qquad (a, b) = \\
&\quad 32903758029140234633796653135855882556 \\
&\quad 26999968978704549198951051648733716787
\end{aligned}
$$

Die Vertrauenspartei entschlüsselt den Blendungsfaktor u mithilfe des ElGamal Verfahrens:

$$\begin{aligned}
a &= 23771262264845457384097857517648014102 \\
b &= 39998965684368905572008766958733578665 \\
x &= 43907678176974499004319307602851643 50 \\
p &= 43728310400465690626815837738477052487 \\
m &= 93144914337485145563440377462648694 08
\end{aligned}$$

Die Vertrauenspartei entschlüsselt den Blendungsfaktor v mithilfe des ElGamal Verfahrens:

$$\begin{aligned}
a &= 30815119520216007042916868085066720498 \\
b &= 19567718565837359464710612078229175429 \\
x &= 43907678176974499004319307602851643 50 \\
p &= 43728310400465690626815837738477052487 \\
m &= 41463444199398386573694968739282898 36
\end{aligned}$$

Die Vertrauenspartei entschlüsselt den Blendungsfaktor w mithilfe des ElGamal Verfahrens:

$$\begin{aligned}
a &= 32903758029140234633796653135855882556 \\
b &= 26999968978704549198951051648733716787 \\
x &= 43907678176974499004319307602851643 50 \\
p &= 43728310400465690626815837738477052487 \\
m &= 10861736086037713971498946182525584186
\end{aligned}$$

Der Kunde berechnet die von der Bank signierte Münze:

$$\begin{aligned}
(c, b) = 21894797945867248950039698693758305 8828, \\
21521581392631802722539273948824161976
\end{aligned}$$

Der Kunde bezahlt im Laden mit seiner Münze:

$$\begin{aligned}
c &= 21894797945867248950039698693758305 8828 \\
b &= 21521581392631802722539273948824161976 \\
gb &= 41478105403780723041241862255085661924 \\
yb &= 37279760573303776369176024013875303803 \\
p &= 43728310400465690626815837738477052487
\end{aligned}$$

Der Vorgang ist beendet.
Im Falle einer doppelten Münze in der Datenbank der Bank wird die Münze zum Besitzer zurückverfolgt:

$$\begin{aligned}
c &= 21894797945867248950039698693758305 8828 \\
b &= 21521581392631802722539273948824161976 \\
u &= 93144914337485145563440377462648694 08 \\
v &= 41463444199398386573694968739282898 36 \\
w &= 10861736086037713971498946182525584186 \\
gb &= 41478105403780723041241862255085661924
\end{aligned}$$

$$yb = 372797605733037763691760240138753038 03$$
$$p = 437283104004656906268158377384770524 87$$
$$q = 218641552002328453134079188692385262 43$$
$$(aq, bq, cq) =$$
$$264535598847733895201974420834851372 26,$$
$$152862551887988793209537591582095311 21,$$
$$658419674751721176163196669538554128 5$$

Die Vertrauenspartei verknpft die signierte Münze mit der doppelten Münze in der Bank.