Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bitcoin Wallet Hersteller - Vertrauen ist gut, Kontrolle ist besser

Masterarbeit

Von Raphael Adam

Betreuer: Dr. Bernd Borchert

Gutachter: Prof. Dr. Peter Hauck
Apl. Prof. Klaus Reinhardt

Zusammenfassung

Es werden in dieser Masterarbeit Verfahren für eine veränderte Schlüsselerzeugung des ElGamal-, RSA- und Lamport-Signaturverfahrens sowie für den ECDSA vorgestellt und auf ihre Sicherheit untersucht. Die geänderten Verfahren funktionieren über ein Protokoll zwischen zwei Parteien, wobei eine Partei ein Security-Token und die andere Partei der Benutzer des Tokens ist. Ziel der Protokolle ist es, dass der Hersteller eines Security-Tokens nicht die Schlüsselerzeugung von digitalen Signaturverfahren manipulieren kann. So könnte z.B. ein Hersteller von Bitcoin Wallets nicht wirklich neue Schlüssel auf den Wallets erzeugen lassen, sondern stattdessen feste Werte verwenden. Die Grundlage für diese Arbeit bilden die Verfahren ElGX [1] und RSAX [2] von Dominik Reichl. Das Protokoll für den ECDSA wurde für diese Arbeit außerdem auf einer Chipkarte implementiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Sicherheitsrelevante Operationen werden häufig auf sogenannte Security-Tokens, wie z.B. Chipkarten, ausgelagert. Diese Tokens bieten einen Schutz vor unbefugtem Zugriff auf Daten. Eine typische Operation sieht dabei so aus, dass eine Nachricht an das Security-Token geschickt wird, welches diese dann mit einem privaten Schlüssel signiert. Dies ist auch die grundlegende Arbeitsweise von Bitcoin Wallets. Das Wallet speichert die privaten Bitcoin-Schlüssel und signiert Transaktionen.

Ein entscheidender Grund für den Einsatz von Security-Tokens ist, dass der private Schlüssel dort sicher gespeichert werden kann. Es stellt sich dann die Frage, wie der private Schlüssel erzeugt werden soll. Eine Möglichkeit ist es, den Schlüssel extern zu generieren und dann an das Token zu schicken. Dieses Vorgehen bietet aber Angriffsmöglichkeiten. Auf dem Rechner, der den Schlüssel erzeugt, könnte sich beispielsweise Malware befinden. Der Schlüssel wäre damit kompromittiert.

Manche Tokens, wie z.B. Chipkarten, bieten die Möglichkeit, einen Schlüssel direkt auf der Karte zu erzeugen. In diesem Fall muss der private Schlüssel niemals das Token verlassen. Ein Nutzer hat so aber keine Möglichkeit, die Qualität des Schlüssels zu beobachten. Durch eine mangelhafte Implementierung oder durch böswillige Absichten des Herstellers könnte die Schlüsselerzeugung zu einem schwachen Schlüssel führen. Entscheidend ist bei der Generierung des Schlüssels, dass dieser wirklich zufällig gewählt wird.

Einem böswilligen Hersteller für Bitcoin Wallets (oder jemandem, der bei der Entwicklung beteiligt war) könnte eine Beeinflussung der Schlüsselerzeugung die Möglichkeit geben, auf die Bitcoins der Nutzer zuzugreifen. So könnte das Wallet beispielsweise die Schlüsselerzeugung nur vortäuschen und stattdessen einen vorberechneten, dem Hersteller bekannten Schlüssel verwenden. Der Hersteller könnte dann für die Bitcoin-Adressen der Nutzer Transaktionen signieren und somit auf die Bitcoins zugreifen.

Dominik Reichl hat für das ElGamal- und RSA-Signaturverfahren Abwandlungen entwickelt, um dieses Problem zu lösen [1][2]. Die Schlüsselerzeugung erfolgt dabei in einem Protokoll zwischen dem Benutzer und dem Security Token. Es soll dadurch sichergestellt werden, dass nur das Token den privaten Schlüssel kennt und dass

dieser zufällig erzeugt wird.

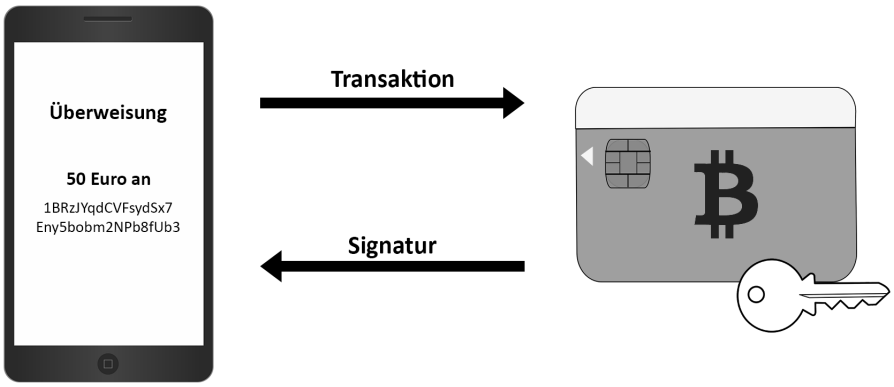

Abbildung 1.1: Ein Bitcoin Wallet signiert mit dem privaten Schlüssel eine Transaktion.

1.2 Ziele der Arbeit

Es sollen die genannten Verfahren von Dominik Reichl auf ihre Sicherheit untersucht werden. Zudem soll das Prinzip der Verfahren von Dominik Reichl noch auf andere Signaturverfahren übertragen werden. Konkret sind dies der ECDSA und das Lamport-Signaturverfahren. Auch für diese Abwandlungen soll die Sicherheit betrachtet werden.

Der ECDSA ist das Signaturverfahren, das bei Bitcoin zum Einsatz kommt. Das im Rahmen dieser Arbeit entwickelte Protokoll zur Schlüsselerzeugung und zur Erzeugung von Signaturen für den ECDSA soll in Form eines Prototyps implementiert werden. Das Security-Token soll dabei mit einer Chipkarte realisiert werden. Die Kommunikation soll über NFC drahtlos mit einem Android-Smartphone erfolgen.

1.3 Aufbau der Arbeit

Es werden in Kapitel 2 zunächst die Signaturverfahren beschrieben, für welche Abwandlungen entwickelt worden sind. In Kapitel 3 wird aufgezeigt, auf welchem gemeinsamen Prinzip die Protokolle in dieser Arbeit beruhen. Die darauf folgenden Kapitel behandeln einzeln die für die verschiedenen Signaturverfahren entwickelten Protokolle und deren Sicherheit. Danach folgt in Kapitel 8 ein Vergleich der Verfahren dieser Arbeit mit einer realen Implementierung eines Bitcoin Wallets. In Kapitel 9 werden die Details zur Implementierung des Prototyps beschrieben. Im letzten Kapitel erfolgt dann eine abschließende Betrachtung.

2 Grundlagen

2.1 Digitale Signaturverfahren

Eine digitale Signatur hat den Zweck, für eine Nachricht aussagen zu können, dass der angebliche Urheber die Nachricht in der vorliegenden Form tatsächlich erstellt hat. Der Urheber besitzt dafür einen privaten Schlüssel K_P, welchen er geheim halten muss. Mit dem privaten Schlüssel kann für eine Nachricht eine digitale Signatur erstellt werden. Die Signatur kann mit dem zu K_P gehörenden öffentlichen Schlüssel $K_{\text{Ö}}$ verifiziert werden. $K_{\text{Ö}}$ muss somit jedem, der eine Nachricht verifizieren will, bekannt sein. Es liegt nahe, dass es praktisch nicht möglich sein darf, K_P aus $K_{\text{Ö}}$ zu berechnen. Ansonsten ließe sich eine Signatur nicht mehr dem Besitzer von K_P zurechnen. [3, S. 425]

Im Folgenden werden die Signaturverfahren vorgestellt, die für diese Arbeit relevant sind.

2.1.1 ElGamal

Das ElGamal-Signaturverfahren [4] basiert darauf, dass es vermutlich schwer ist, das diskrete Logarithmus Problem (siehe [3, S. 103]) zu lösen. In diesem Fall bedeutet das, dass für eine Primzahl p, eine zugehörige Primitivwurzel[1] g und eine Zahl $y \in \mathbb{Z}_p^*$ nicht einfach eine Zahl $x \in \mathbb{Z}_p^*$ bestimmt werden kann, so dass $g^x \bmod p = y$.

Schlüsselerzeugung

Mit folgendem Algorithmus wird ein neues Schlüsselpaar erzeugt: [3, S. 454]

1. Wähle zufällig eine (große) Primzahl p und eine Primitivwurzel g mod p. Um effizient eine Primitivwurzel zu finden, kann p so gewählt werden, dass $p := 2q+1$, wobei p und q Primzahlen sind. Es ist jetzt die Faktorisierung von $p-1$

[1] Eine Primitivwurzel mod p (mit p Primzahl) ist eine Zahl $g \in \mathbb{Z}_p$, für die gilt, dass es für jede Zahl $x \in \mathbb{Z}_p^*$ einen Exponenten $e \in \mathbb{Z}_p^*$ gibt, so dass $x = g^e \bmod p$. [5, S. 38]

bekannt. Wähle nun solange Zahlen für g, bis $g^2 \bmod p \neq 1$ und $g^q \bmod p \neq 1$. Sind diese Eigenschaften erfüllt, so ist g eine Primitivwurzel mod p. [3, S. 163]

2. Ziehe zufällig eine Zahl[2] $a \in [1, p-2]$.

3. Berechne $x := g^a \bmod p$.

Der öffentliche Schlüssel ist nun (p, g, x), der private Schlüssel ist a.

Erzeugung einer Signatur

Es sei $H : \{0,1\} \to \mathbb{Z}_p$ eine kollisionsresistente[3] Hashfunktion. Mit dem folgenden Algorithmus kann für eine Nachricht m eine Signatur erzeugt werden: [3, S. 454]

1. Ziehe zufällig eine Zahl $k \in [1, p-2]$ mit $\mathrm{ggT}(k, p-1) = 1$.

2. Berechne $r := g^k \bmod p$.

3. Berechne $s := (H(m) - ar)k^{-1} \bmod (p-1)$. Es ist k^{-1} das Inverse zu k bezüglich mod $(p-1)$.

Die Signatur für die Nachricht m ist das Paar (r, s).

Verifikation einer Signatur

Die Gültigkeit einer Signatur (r, s) für eine Nachricht m wird mit dem öffentlichen Schlüssel folgendermaßen überprüft: [3, S. 454 f.]

1. Überprüfe, ob $r \in [1, p-1]$.

2. Überprüfe, ob $x^r r^s \equiv g^{H(m)} \pmod p$.

Beweise

Satz 2.1. Wurde eine Signatur für eine Nachricht m wie oben beschrieben erzeugt, so gilt $x^r r^s \equiv g^{H(m)} \pmod p$.

[2] In dieser Arbeit sind immer ganze Zahlen gemeint.

[3] Es sei mit kollisionsresistent in dieser Arbeit die starke Kollisionsresistenz gemeint. D.h. es ist praktisch nicht möglich, zwei unterschiedliche Werte x,x' zu finden, so dass $H(x) = H(x')$. [3, S. 324]

Beweis. Es sind $r \equiv g^k \pmod{p}$ und $s \equiv (H(m)-ar)k^{-1} \pmod{(p-1)}$. Multipliziert man beide Seiten von s mit k, so ergibt sich $ks \equiv H(m) - ar \pmod{(p-1)}$. Dies lässt sich zu $H(m) \equiv ar + ks \pmod{(p-1)}$ umstellen. Somit: $g^{H(m)} \equiv g^{ar+ks} \equiv x^r r^s \pmod{p}$. [3, S. 455] □

2.1.2 ECDSA

Der ECDSA (*Elliptic Curve Digital Signature Algorithm*) [6] funktioniert nach einem ähnlichem Prinzip wie das ElGamal-Signaturverfahren, wobei jedoch statt in der Gruppe $\mathbb{Z}_p^*$ in einer Gruppe der Punkte einer elliptischen Kurve gerechnet wird.

Eine Elliptische Kurve über einem endlichen Körper $\mathbb{F}_p$ ist durch eine Gleichung $y^2 \equiv x^3+ax+b \pmod{p}$ bestimmt. Die zugehörige Gruppe besteht aus allen Punkten (x, y) mit $x, y \in \mathbb{F}_p$, die diese Gleichung erfüllen und aus einem zusätzlichen Punkt $\mathcal{O}$ (Fernpunkt). Für $a, b \in \mathbb{F}_p$ muss gelten, dass $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$. Es lässt sich auf dieser Menge dann eine Operation $\oplus$ definieren, die die Gruppeneigenschaften erfüllt[4]. [7, S. 6 f.]

Die Sicherheit von ECDSA beruht darauf, dass es vermutlich schwierig ist, den diskreten Logarithmus in der Punktgruppe einer elliptischen Kurve zu berechnen. D.h. für einen Punkt P auf dieser elliptischen Kurve und einen Punkt $Q \in \langle P \rangle$ muss es schwer sein, ein $l \in \mathbb{F}_p$ zu finden, so dass $Q = l \cdot P$ (wobei $l \cdot P$ die skalare Multiplikation ist, z.B. $3 \cdot P = P \oplus P \oplus P$). [8, S. 153]

Schlüsselerzeugung

Die Kurvenparameter seien wie folgt gegeben:

- Eine Primzahl p, die den endlichen Körper $\mathbb{F}_p$ bestimmt.
- $a, b \in \mathbb{F}_p$, die die elliptische Kurve $y^2 = x^3 + ax + b$ bestimmen.
- Ein Punkt G auf dieser Kurve mit der Ordnung n.

Folgendermaßen wird ein neues Schlüsselpaar erzeugt: [8, S. 180]

1. Ziehe zufällig eine Zahl $d \in [1, n-1]$.
2. Berechne $Q := d \cdot G$.

Der öffentliche Schlüssel ist Q, der private Schlüssel ist d.

[4] Siehe [7, S. 8] für eine Definition dieser Operation.

Erzeugung einer Signatur

Es seien die Kurvenparameter wie bei der Schlüsselerzeugung gegeben. Außerdem sei H eine kollisionsresistente Hashfunktion mit der Eigenschaft, dass die Bitlänge der Ausgabe nicht größer ist als die Bitlänge von n. Eine Signatur für eine Nachricht m wird dann wie folgt erzeugt: [8, S. 184]

1. Ziehe zufällig eine Zahl $k \in [1, n-1]$.
2. Berechne $k \cdot G := (x_1, y_1)$.
3. Berechne $r := x_1 \bmod n$. Falls $r = 0$, gehe zu Schritt 1.
4. Berechne $z := H(m)$.
5. Berechne $s := k^{-1}(z + rd) \bmod n$. Falls $s = 0$, gehe zu Schritt 1.

Die Signatur ist das Paar (r, s).

Verifikation einer Signatur

Eine Signatur (r, s) kann folgendermaßen mit dem öffentlichem Schlüssel Q überprüft werden: [8, S. 184]

1. Stelle sicher, dass $r, s \in [1, n-1]$.
2. Berechne $z := H(m)$.
3. Berechne $w := s^{-1} \bmod n$.
4. Berechne $u_1 := zw \bmod n$ und $u_2 := rw \bmod n$.
5. Berechne $(x_1, y_1) := u_1 \cdot G \oplus u_2 \cdot Q$.
6. Stelle sicher, dass $r = x_1 \bmod n$.

Wenn keine der Überprüfungen fehlschlägt, so ist die Signatur gültig.

Beweise

Satz 2.2. Die beschriebene Verifikation einer Signatur funktioniert.

Beweis. Es gilt $s \equiv k^{-1}(z + rd) \pmod{n}$. Dies lässt sich zu $k \equiv s^{-1}(z + rd) \equiv s^{-1}z + s^{-1}rd \equiv wz + wrd \equiv u_1 + u_2 d \pmod{n}$ umformen. Somit ist $(x_1, y_1) = u_1 \cdot G \oplus u_2 \cdot Q = (u_1 + u_2 d) \cdot G = k \cdot G$ und damit $r = x_1 \bmod n$. [8, S. 185] □

2.1.3 RSA

Das RSA-Kryptosystem [9] kann sowohl zum Signieren als auch zum Verschlüsseln eingesetzt werden. Die Sicherheit von RSA ist davon abhängig, dass es schwer sein muss, große Zahlen zu faktorisieren [9, S. 11 ff.].

Schlüsselerzeugung

Mit dem folgendem Algorithmus wird ein neues Schlüsselpaar erzeugt: [3, S. 434]

1. Wähle zufällig zwei (große) unterschiedliche Primzahlen p und q. Es sei $n := pq$.
2. Berechne die Eulersche Phi-Funktion für n. Die Phi-Funktion ist definiert als $\varphi(n) = |\{a \in \mathbb{N} \mid 1 \leq a \leq n \wedge \mathrm{ggT}(a, n) = 1\}|$. Für Primzahlen p und q gilt: $\varphi(pq) = (p - 1)(q - 1)$.
3. Wähle eine Zahle e mit $1 < e < \varphi(n)$, so dass $\mathrm{ggT}(e, \varphi(n)) = 1$.
4. Berechne d, so dass $ed \equiv 1 \pmod{\varphi(n)}$.

Der öffentliche Schlüssel ist nun das Paar (e, n), der private Schlüssel ist d.

Erzeugung einer Signatur

Um eine Nachricht $m \in [0, n - 1]$ zu signieren (m kann auch der Hashwert einer Nachricht sein), wird die Signatur s folgendermaßen berechnet: $s := m^d \bmod n$.

Verifikation einer Signatur

Jeder, der Zugriff auf den öffentlichen Schlüssel (e, n) hat, kann für eine Nachricht m und zugehöriger Signatur s überprüfen, ob die Signatur gültig ist. Dazu muss sichergestellt werden, dass $m = s^e \bmod n$.

Beweise

Satz 2.3. Wenn $s := m^d \bmod n$, dann gilt, dass $m = s^e \bmod n$.

Beweis. Aus $ed \equiv 1 \pmod{\varphi(n)}$ folgt, dass es eine Zahl k gibt mit $ed = 1 + k \cdot \varphi(n)$. Ist jetzt $\text{ggT}(m, p) = 1$, so folgt mit dem kleinen Satz von Fermat[5], dass $m^{p-1} \equiv 1 \pmod{p}$. Nimmt man beide Seiten hoch $k(q-1)$ und multipliziert sie dann mit m, so erhält man $m^{1+k(p-1)(q-1)} \equiv m \pmod{p}$. Diese Kongruenz ist aber auch in dem Fall gültig, wenn $\text{ggT}(m, p) = p$. Somit gilt in allen Fällen, dass $m^{ed} \equiv m \pmod{p}$. Über die gleiche Argumentation erhält man auch $m^{ed} \equiv m \pmod{q}$. Da p und q unterschiedliche Primzahlen sind, folgt damit, dass $m^{ed} \equiv m \pmod{n}$. Ist nun $s := m^d \bmod n$, so ist damit $s^e \bmod n = m^{ed} \bmod n = m$. [3, S. 286] □

2.1.4 Lamport

Die Lamport-Signatur [10] ist eine digitale Signatur, die mithilfe einer Einwegfunktion (z.B. einer Hashfunktion) realisiert ist. Mit jedem Schlüsselpaar kann nur eine einzelne Nachricht signiert werden.

Das Verfahren funktioniert nach dem folgendem Prinzip: Es wird zuerst ein privater Schlüssel erzeugt. Dies geschieht so, dass zugehörig zu jedem Bit einer gehashten Nachricht zwei zufällige Zahlen erzeugt werden. Eine Zahl dafür, dass dieses Bit 0 ist und eine Zahl dafür, dass es 1 ist. Der öffentliche Schlüssel besteht aus den Hashwerten all dieser zufälligen Zahlen. Soll eine Nachricht m signiert werden, so wird zuerst der Hashwert $H(m)$ gebildet. Für jedes Bit von $H(m)$ besteht die Signatur aus der zu diesem Bit zugehörigen zufälligen Zahl. Mithilfe des öffentlichen Schlüssels lässt sich dann überprüfen, ob diese Zahlen die Urbilder der Hashwerte sind.

Das Lamport-Verfahren soll in dieser Arbeit in der im Folgenden beschriebenen Definition verwendet werden.

Schlüsselerzeugung

Es seien:

- $k \in \mathbb{N}$
- H eine Einwegfunktion mit $H : \{0,1\}^k \to \{0,1\}^k$

[5] Der kleine Satz von Fermat besagt, dass $a^{r-1} \equiv 1 \pmod{r}$, wenn r eine Primzahl ist und wenn a nicht durch r teilbar ist. [3, S. 69]

Wähle $x_{i,j} \in \{0,1\}^k$ zufällig für alle $i \in \{1,2,...,k\}$ und $j \in \{0,1\}$. Die Werte $x_{i,j}$ stellen den privaten Schlüssel dar. Der öffentliche Schlüssel besteht aus den Werten $y_{i,j} := H(x_{i,j})$. [11, S. 3]

Erzeugung einer Signatur

Es sei $m := (m_1, m_2, ..., m_k) \in \{0,1\}^k$ der Hashwert einer Nachricht. Die Signatur für diese Nachricht ist dann $(x_{1,m_1}, x_{2,m_2}, ..., x_{k,m_k}) := (s_1, s_2, ..., s_k)$. [11, S. 3]

Verifikation einer Signatur

Mithilfe des öffentlichen Schlüssels wird überprüft, ob $y_{i,m_i} = H(s_i)$ für alle $i \in \{1,2,...,k\}$. Falls ja, so ist die Signatur gültig. [11, S. 3]

3 Gemeinsames Prinzip der Verfahren

Alle in den folgenden Kapiteln beschriebenen Protokolle sind Abwandlungen der in Kapitel 2 vorgestellten digitalen Signaturverfahren. Diese Abwandlungen dienen hauptsächlich dazu sicherzustellen, dass der private Schlüssel auf einem Security-Token wirklich zufällig erzeugt wird. Ein Hersteller eines Security-Tokens soll nicht in der Lage sein, eine Schlüsselerzeugung zu implementieren, bei der er sich Kenntnis über private Schlüssel verschaffen kann. Die Abwandlungen sind alle so benannt, dass ein X an den ursprünglichen Namen des Signaturverfahrens angehängt wird. Aus RSA wird beispielsweise RSAX.

Um die Zufälligkeit des privaten Schlüssels sicherzustellen, kommen bei der Schlüsselerzeugung der abgeänderten Verfahren zwei Parteien zum Einsatz. Der private Schlüssel wird auf dem Security-Token generiert. Die Partei des Security-Tokens wird mit S bezeichnet. Nur S soll den privaten Schlüssel kennen. S muss nun bei der Schlüsselerzeugung Nachrichten mit der zweiten Partei, A (für Alice), austauschen. Diese Partei repräsentiert den Benutzer des Security-Tokens. A wird, im Gegensatz zu S, vertraut. Es wird aber auch der umgekehrte Fall betrachtet, d.h. wenn A misstraut und S vertraut wird. Dieses Szenario könnte sich z.B. durch Malware ergeben. Ist A ein PC oder ein Smartphone, so sind dort prinzipiell Schadprogramme möglich. Bei einem Security-Token ist dies nicht ohne Weiteres machbar.

Die Verfahren ElGX, ECDSAX und LamportX verwenden den folgenden Ablauf, um eine zufällige Zahl $z \in \{0, 1, ..., n-1\}$ zu erzeugen, die nur S kennen soll:

Es sei F eine Einweg- oder Verschlüsselungsfunktion. F soll stark kollisionsresistent sein.

1. S zieht zufällig eine Zahl $s \in \{0, 1, ..., n-1\}$ und schickt $F(s)$ an A.

2. A zieht zufällig eine Zahl $a \in \{0, 1, ..., n-1\}$ und schickt a an S.

3. S berechnet $z := a + s \bmod n$ und schickt $F(z)$ an A.

4. A stellt sicher, dass S für z auch wirklich $a + s \bmod n$ verwendet.

Abhängig davon, wie F genau aussieht, werden verschiedene Methoden eingesetzt

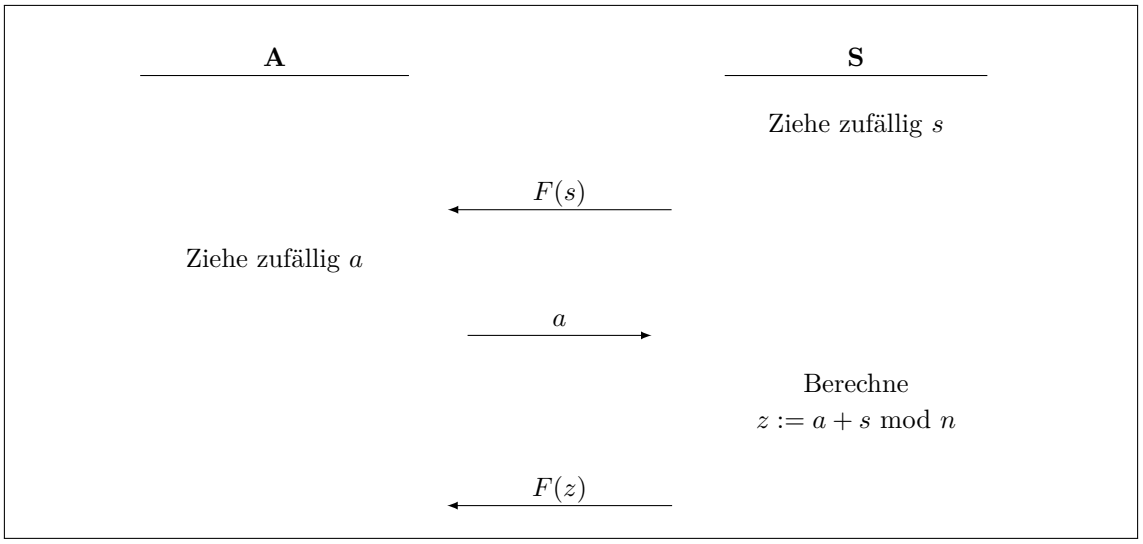

Abbildung 3.1: Prinzipieller Ablauf bei der Erzeugung einer zufälligen Zahl z.

um sicherzustellen, dass S auch wirklich $a + s \bmod n$ für z verwendet. Bei den Verfahren ElGX und ECDSAX kann ohne Offenlegung von s oder z überprüft werden, ob $z = a + s \bmod n$. Es ist bei ElGX $F(x) = g^x \bmod p$ (mit p Primzahl und g Primitivwurzel mod p) und $n := p - 1$. Es muss dann $F(z) = F(s) \cdot F(a)$ gelten. Da bei ElGX der öffentliche Schlüssel $F(z)$ ist und z der private Schlüssel, kann A durch diese Berechnung nun sicherstellen, dass der private Schlüssel richtig erzeugt wurde.

Bei den Verfahren RSAX und LamportX ist eine solche Überprüfung nicht möglich, es kommt dort eine andere Methode zum Einsatz. Es werden mit dem obigen Ablauf mehr zufällige Zahlen z_i erzeugt, als für die Schlüsselerzeugung eigentlich notwendig ist. A bestimmt dann zufällig durch die Wahl der i, welche z_i für den Schlüssel verwendet werden. Für alle anderen z_i muss S danach die zugehörigen s_i offenlegen. A ist nun in der Lage zu überprüfen, ob diese z_i von S richtig berechnet wurden. Sollte S also ein bestimmtes z_i nicht ordnungsgemäß berechnen, so wird A dies mit einer bestimmten Wahrscheinlichkeit bemerken.

Angenommen, die Partei A verhält sich bei dem obigen Protokoll regelgemäß und wählt ihre Zahl a unabhängig von s und zufällig aus der Gleichverteilung[1] auf $\{0, 1, ..., n-1\}$. Nehme weiter an, dass S versucht, auf die Erzeugung von z Einfluss zu nehmen und ihre Zahl s aus einer unbekannten Verteilung wählt. So könnte z.B. S eine feste Zahl für s verwenden. Satz 3.1 zeigt dann, dass die resultierende Zahl z trotzdem eine auf $\{0, 1, ..., n-1\}$ gleichverteilt zufällig gewählte Zahl ist.

Satz 3.1. Es seien X, Y zwei unabhängige diskrete Zufallsvariablen auf $\{0, 1, ..., n-1\}$, wobei X gleichverteilt ist. Es ist dann $Z = X + Y \bmod n$ auch gleichverteilt auf $\{0, 1, ..., n-1\}$, egal wie die Verteilung von Y aussieht.

[1] Bei einer diskreten Gleichverteilung [12, S. 5] ist jedes mögliche Ergebnis gleich wahrscheinlich.

Beweis. Es sei $z \in \{0, 1, ..., n-1\}$. Für die Wahrscheinlichkeitsfunktion für Z gilt dann:[2]

$$P(Z = z) = \sum_{i=0}^{n-1} P(Y = i)P(X = z - i \bmod n)$$

Da X gleichverteilt ist, gilt für beliebige $x \in \{0, 1, ..., n-1\}$, dass $P(X = x) = \frac{1}{n}$. Somit:

$$P(Z = z) = \frac{1}{n} \sum_{i=0}^{n-1} P(Y = i) = \frac{1}{n}$$

Z ist also gleichverteilt. □

Wollen sich, wie oben beschrieben, zwei Parteien auf eine zufällige Zahl einigen, so ist es für die Sicherheit von Vorteil, wenn die Partei, der misstraut wird (hier S), sich zuerst auf ihre Zahl festlegen muss. Im umgekehrten Fall wäre es sonst nicht möglich Satz 3.1 anzuwenden. Würde zuerst A ihre Zahl a wählen und $F(a)$ an S schicken, so könnte S ihre Zahl s in Abhängigkeit von a wählen. $F(a)$ hängt deterministisch von a ab, somit könnte S für eine abhängige Wahl z.B. einfach $s := F(a)$ an A schicken. Jetzt kann zwar nicht mehr garantiert werden, dass z gleichverteilt zufällig gewählt wird, was aber nicht unbedingt heißt, dass dies S tatsächlich einen Angriff ermöglicht. Über die Verteilung von z lässt sich aber keine Aussage mehr machen.

[2] Beweisidee aus [13].

4 ElGX

ElGX [1] ist eine von Dominik Reichl erdachte Abwandlung des ElGamal-Signaturverfahrens (siehe Kapitel 2.1.1). Im Vergleich zu ElGamal wurden dabei die Schlüsselerzeugung und die Erzeugung von Signaturen geändert. Das Format der Schlüssel und der Signaturen ist jedoch dasselbe wie bei ElGamal und eine Signatur wird auch auf dieselbe Weise wie bei ElGamal verifiziert.

Die Erzeugung des Schlüsselpaares und von Signaturen geschieht bei ElGX mit zwei Parteien. Die Partei S kennt den privaten Schlüssel. Diese Partei stellt das Security-Token dar. Damit sichergestellt ist, dass der private Schlüssel und die Signaturen zufällig erzeugt werden, ist eine zweite Partei A (Alice) involviert. Diese Partei steht für den Benutzer des Security-Tokens. In der Praxis könnte A z.B. eine Smartphone Applikation sein und die Kommunikation mit S über NFC stattfinden.

Im Folgenden wird zuerst ein Protokoll mit der Bezeichnung NMRNG vorgestellt. Dieses wird für die ersten beiden optionalen Schritte der Schlüsselerzeugung benötigt.

4.1 Nicht manipulierbare Erzeugung von zufälligen Zahlen

Das Protokoll NMRNG (von *Non-Manipulable Random Number Generation*) stammt von Dominik Reichl [1]. Es wurden hier jedoch ein paar Details geändert. So wird z.B. bei Reichl eine Verschlüsselungsfunktion verwendet. Hier wird jedoch mit einer Hashfunktion gearbeitet.

NMRNG(b_l, b_u) ist ein Protokoll zwischen A und S, das die folgenden Eigenschaften erfüllen soll:

- Die Ausgabe des Protokolls ist eine zufällige Zahl z aus der Gleichverteilung auf $[b_l, b_u - 1]$.
- Sowohl A als auch S sollen z nach dem Protokolldurchlauf kennen.

Um dies umzusetzen, soll folgendes Protokoll verwendet werden:

Es sei H eine kollisionsresistente Hashfunktion.

1. S zieht eine zufällige Zahl x aus der Gleichverteilung auf $[0, b_u - b_l - 1]$. S schickt $\tilde{x} := H(x)$ an A.

2. A zieht eine zufällige Zahl y aus der Gleichverteilung auf $[0, b_u - b_l - 1]$ und schickt sie an S.

3. S schickt x an A

4. A überprüft, ob $\tilde{x} = H(x)$. Falls nein, so bricht A das Protokoll ab.

5. Das Ergebnis des Protokolls ist $z := b_l + (x + y \bmod (b_u - b_l))$. Dies können A und S beide berechnen.

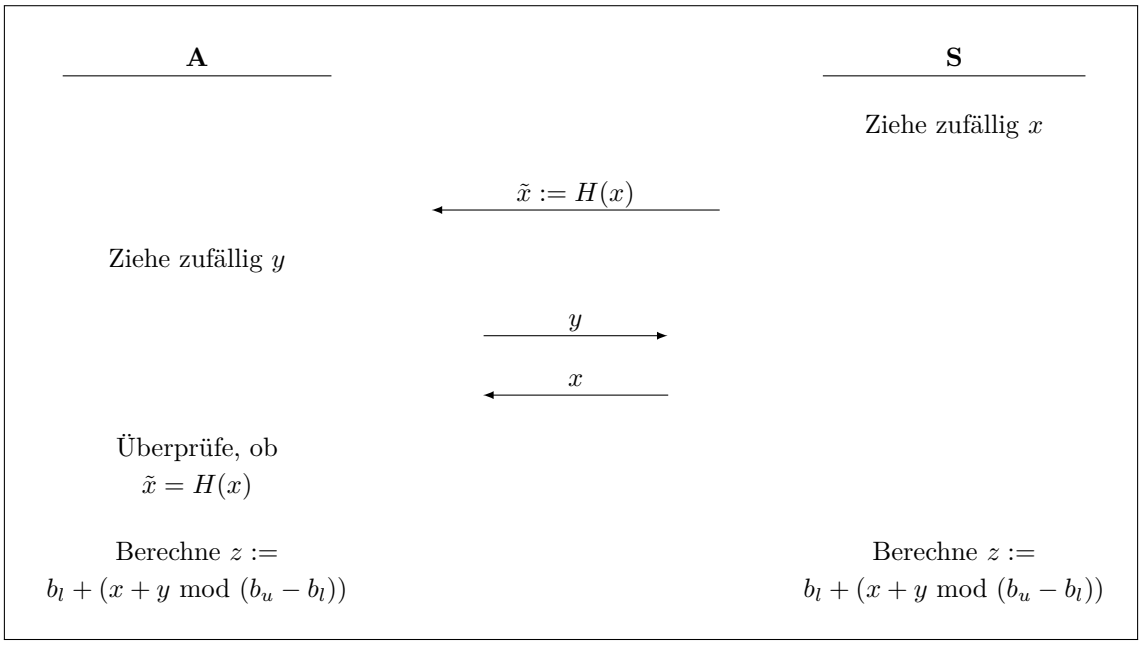

Abbildung 4.1: Ablauf des NMRNG Protokolls.

4.1.1 Sicherheit

Es wird hier und auch bei allen anderen Protokollen davon ausgegangen, dass nur die Parteien A und S involviert sind. Es wird also nicht betrachtet, ob das Protokoll gegen Angreifer von außen oder gegen Mithörer sicher ist. Eine Sicherheit gegenüber solchen Angriffen ist in der Praxis vermutlich nicht notwendig, da die Kommunikation zwischen A und S nicht über ein Netzwerk stattfindet. Ein Security-Token befindet sich normalerweise in unmittelbar physischer Nähe des Nutzers. Der Datenaustausch zwischen A und S kann z.B. über NFC (Smartphone und Chipkarte) oder über einen Kartenleser (PC und Chipkarte) stattfinden.

Es werden nun die beiden Szenarien untersucht, in denen einer Partei getraut wird und die andere Partei versucht zu betrügen. D.h. diese Partei versucht so vom Protokoll abzuweichen, dass die Ausgabe z nicht gleichverteilt zufällig gewählt wird. Im schlimmsten Fall würde das bedeuten, dass z von einer Partei frei gewählt werden kann. Der Fall, dass S die betrügerische Partei ist, steht in dieser Arbeit im Zentrum. Verhalten sich beide Parteien böswillig, so ist bei keinem der Protokolle in dieser Arbeit eine Sicherheit gegeben.

Szenario: *S* versucht zu betrügen

Nach der Erzeugung von z muss A noch in irgendeiner Form sicherstellen, dass S auch wirklich den richtigen Wert für z verwendet. NMRNG wird später dazu eingesetzt, Teile des öffentlichen Schlüssels bei ElGX zu erzeugen. Es würde für S an dieser Stelle keinen Sinn machen, einen anderen Wert für z zu benutzen. A würde dann immer noch mit dem richtigen z rechnen.

Es sei angenommen, dass A sich gemäß dem Protokoll verhält und für y eine von x unabhängige und gleichverteilt zufällige Zahl zieht. S muss sich auf x festlegen, noch bevor A den Wert y gewählt hat. Zu diesem Zeitpunkt kann S also x nicht in Abhängigkeit von y wählen. Nachdem S der Wert y von A zugeschickt wurde, bleibt S nichts anderes übrig, als x an A zu schicken. Da H eine kollisionsresistente Hashfunktion ist, ist es S nicht möglich einen Wert $x' \neq x$ zu finden, so dass $H(x') = H(x)$. Da nun also A und S ihre Zahlen unabhängig voneinander wählen und da A ihre Zahl gleichverteilt zufällig wählt, sind die Voraussetzungen für Satz Satz 3.1 gegeben. Das Ergebnis z wird gleichverteilt zufällig gewählt. S hat somit keine Möglichkeit zu betrügen.

Szenario: *A* versucht zu betrügen

Nehme nun an, dass sich S korrekt verhält. A muss y erst wählen, nachdem S sich schon auf x festgelegt hat und $H(x)$ an A geschickt hat. Für A ist es also möglich, y in Abhängigkeit von x zu wählen. So kann A z.B. $H(x)$ als Zahl interpretieren und für y verwenden (eventuell müssen noch irgendwelche Operationen vorgenommen werden, so dass $y \in [0, b_u - b_l - 1]$). Die Voraussetzungen für Satz 3.1 sind somit nicht gegeben. Wie früher schon angemerkt, muss dies nicht unbedingt heißen, dass A in der Praxis einen erfolgreichen Angriff irgendeiner Art durchführen kann. Es kann nur nicht mehr garantiert werden, dass z gleichverteilt zufällig gewählt wird.

Eine andere Möglichkeit, das Verfahren zu beeinflussen, ergibt sich in Schritt 4. A hat hier die Möglichkeit, das Protokoll abzubrechen. Theoretisch könnte A das Protokoll so oft abbrechen und neu starten, bis z einen geeigneten Wert ergibt. Für eine einfachere Betrachtung wird in dieser Arbeit aber angenommen, dass ein Protokoll

bei einem Abbruch nicht erneut ausgeführt wird. Sollte ein Neustart möglich sein, so steht dies explizit in dem Protokoll.

4.2 Schlüsselerzeugung

Das Protokoll zur Schlüsselerzeugung stammt ebenfalls von Dominik Reichl [1]. Es wurden aber auch hier kleine Änderungen vorgenommen.

Das hier vorgestellte Protokoll soll diese Eigenschaften erfüllen:

- Der private Schlüssel a wird zufällig aus der Gleichverteilung auf $[2, p-2]$ gezogen.
- Nur S kennt den privaten Schlüssel.

A und S verwenden das folgende Protokoll, um ein neues ElGamal Schlüsselpaar zu erzeugen. Die ersten beiden Schritte sind dabei optional. Es ist auch möglich, dass sich die Parteien auf feste Werte für p und g einigen.

1. A und S erzeugen eine zufällige Zahl $q' := \mathrm{NMRNG}(2, 2^{4095})$ und wählen q als die kleinste Primzahl größer oder gleich q', für die gilt, dass $p := 2q + 1$ auch eine Primzahl ist.

2. Da die Faktorisierung von $p-1$ bekannt ist, können A und S effizient entscheiden, ob eine Zahl eine Primitivwurzel mod p ist. A und S erzeugen nun solange zufällige Zahlen durch Ausführung von $\mathrm{NMRNG}(2, p-1)$, bis sie eine Primitivwurzel g gefunden haben.

3. S zieht eine zufällige Zahl a' aus der Gleichverteilung auf $[2, p-2]$, berechnet $x' := g^{a'} \bmod p$ und schickt x' an A.

4. A zieht eine zufällige Zahl d aus der Gleichverteilung auf $[0, p-2]$ und schickt d an S.

5. S berechnet $a := a' + d \bmod (p-1)$. Falls $a \leq 1$: Schicke a' an A und gehe zurück zu Schritt 3.

   S berechnet $x := g^a \bmod p$ und schickt x an A.

6. A stellt sicher, dass $x = x' \cdot g^d \bmod p$.

Der öffentliche Schlüssel ist (p, g, x), der private Schlüssel ist a.

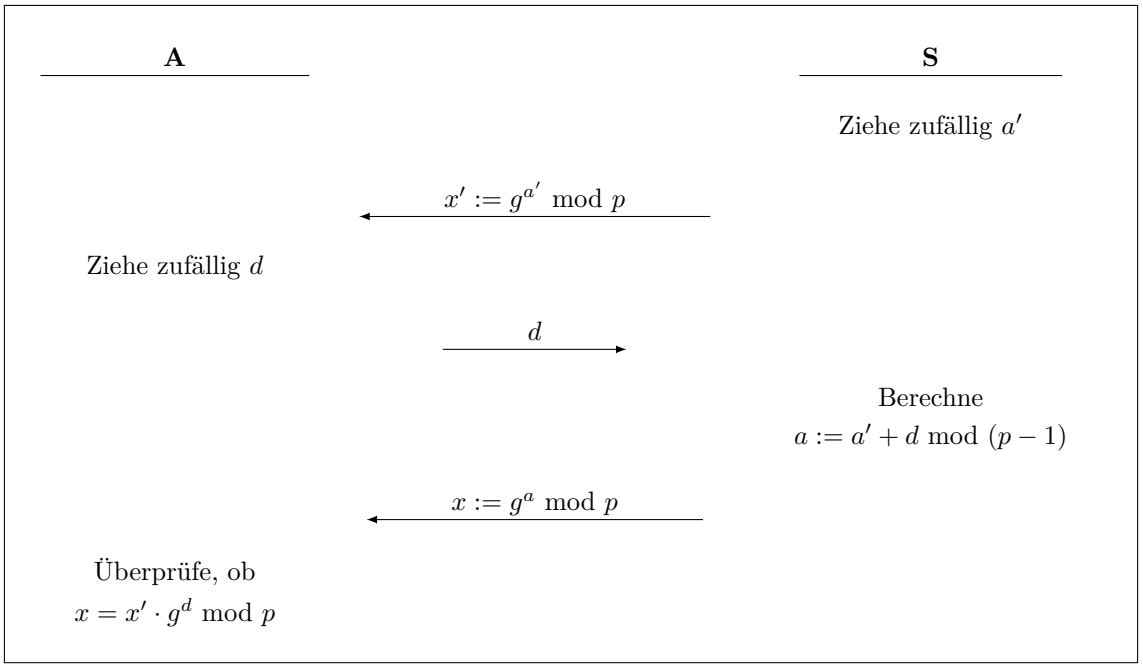

Abbildung 4.2: Vereinfachter Ablauf der ElGX-Schlüsselerzeugung.

4.2.1 Sicherheit

Es sei hier angenommen, dass die Werte p und g feststehen und nicht über das Protokoll erzeugt werden. Die ersten beiden optionalen Schritte der ElGX-Schlüsselerzeugung werden also nicht in die Überlegungen zur Sicherheit mit einbezogen.

Szenario: *S* versucht zu betrügen

Es sei angenommen, dass A sich gemäß dem Protokoll verhält (A zieht d also unabhängig von a' und gleichverteilt zufällig). S versucht hingegen die Schlüsselerzeugung so zu beeinflussen, dass der private Schlüssel a nicht gleichverteilt zufällig gewählt wird.

Es muss hier wieder S zuerst ihre zufällige Zahl a' wählen und $x' := g^{a'} \bmod p$ an A schicken. Die zufällige Zahl d von A steht zu diesem Zeitpunkt noch nicht fest. S kann a' also nicht in Abhängigkeit von d wählen. Nachdem S den Wert d von A erhalten hat, muss sie $x := g^a \bmod p$ mit $a = a' + d \bmod p - 1$ an A schicken. Nehme an, dass S $a \in [2, p-2]$ so wählen will, dass $a \neq a' + d \bmod (p-1)$. Damit A diese Manipulation nicht bemerkt, muss gelten, dass $g^a \bmod p = g^{a'} \cdot g^d \bmod p = g^{a'+d} \bmod p$. D.h. $a \equiv a' + d \pmod{p-1}$. Es muss somit $a = a' + d \bmod (p-1)$ sein.

Da S ihre Zahl a' unabhängig von der Zahl d von A wählen muss und da A ihre Zahl gleichverteilt zufällig und unabhängig wählt, sind die Voraussetzungen für

Satz 3.1 gegeben. Damit folgt, dass $a = a' + d \bmod p - 1$ gleichverteilt zufällig ist. Außerdem kann A durch Überprüfung, ob $x = x' \cdot g^d \bmod p$, sicherstellen, dass S für den privaten Schlüssel a den richtigen Wert verwendet (es ist x der öffentliche Schlüssel).

In Schritt 5 der Schlüsselerzeugung muss das Protokoll wiederholt werden, falls $a \leq 1$. Indem davor a' an A geschickt wird, kann A sicherstellen, dass dieser unwahrscheinliche Fall tatsächlich aufgetreten ist (durch Überprüfung, ob $a'+d \bmod p-1 \leq 1$ und $x' = g^{a'} \bmod p$). S wird hier also keine Möglichkeit gegeben, die Schlüsselerzeugung beliebig oft zu wiederholen. [1]

Szenario: *A* versucht zu betrügen

Es sei nun angenommen, dass S vertraut werden kann und dass A versucht, entweder Kenntnis über den privaten Schlüssel zu erlangen oder die Schlüsselerzeugung so zu beeinflussen, dass der private Schlüssel a nicht gleichverteilt zufällig gewählt wird.

Will A nach der Schlüsselerzeugung den privaten Schlüssel a berechnen, so kann sie entweder aus $x' := g^{a'} \bmod p$ versuchen a' zu bestimmen und damit a zu berechnen. Oder sie berechnet a direkt aus $x := g^a \bmod p$. In beiden Fällen müsste A in der Lage sein, den diskreten Logarithmus zu lösen. [1]

Da A ihre zufällige Zahl d erst wählen muss, nachdem die zufällige Zahl a' von S schon feststeht, kann A ihre Zahl in Abhängigkeit der Zahl von S wählen. So kann A z.B. $d := x'$ an S schicken. Satz 3.1 lässt sich also nicht anwenden.

Das folgende Beispiel zeigt außerdem, dass es hier für A möglich ist, aus $x' := g^{a'} \bmod p$ Informationen über a' zu gewinnen. Mit einer passenden Wahl von d kann die Verteilung von a dann so beeinflusst werden, dass a nur ungerade Werte ergibt:

Liegt eine Zahl x' vor, mit $x' := g^{a'} \bmod p$, so kann einfach bestimmt werden, ob a' gerade oder ungerade ist. Es muss nur überprüft werden, ob x' ein quadratischer Rest modulo p ist (d.h. es gibt eine Zahl r mit $r^2 \equiv x' \pmod p$). Falls ja, so ist a' gerade [14, S. 38]. Es ist x' ein quadratischer Rest, wenn ${x'}^{\frac{p-1}{2}} \bmod p = 1$ [14, S. 34]. A kann also feststellen, ob a' gerade oder ungerade ist, bevor sie d wählt. Je nachdem wählt sie für d dann entweder 0 oder 1, sodass a eine ungerade Zahl ergibt.

Es ist trotzdem fraglich, ob A in der Lage ist, die Verteilung von a soweit zu beeinflussen, dass sie Kenntnis von dem privaten Schlüssel erlangen kann.

4.3 Erzeugung einer Signatur

Das Protokoll zur Erzeugung einer Signatur (von Dominik Reichl [1]) funktioniert analog zu der Schlüsselerzeugung. Im Vergleich zu dem Protokoll von Reichl wurden hier die gleichen Änderungen wie bei der Schlüsselerzeugung vorgenommen.

Das Protokoll soll diese Eigenschaften erfüllen:

- Der Wert k wird zufällig aus der Gleichverteilung auf $[2, p-2]$ gezogen.
- Nur S kennt k.

A kann zusammen mit S mit dem folgenden Protokoll eine Nachricht m signieren. Es sei H eine kollisionsresistente Hashfunktion.

1. A schickt $H(m)$ an S.
2. S zieht eine zufällige Zahl k' aus der Gleichverteilung auf $[2, p-2]$, berechnet $r' := g^{k'} \bmod p$ und schickt r' an A.
3. A zieht eine zufällige Zahl d' aus der Gleichverteilung auf $[0, p-2]$ und schickt d' an S.
4. S berechnet $k := k' + d' \bmod (p-1)$. Falls $\mathrm{ggT}(k, p-1) \neq 1$: Schicke k' an A und gehe zurück zu Schritt 2.

   S berechnet $r := g^k \bmod p$ und $s := (H(m) - ar)k^{-1} \bmod (p-1)$. Falls $s = 0$: Melde einen Fehler und breche das Protokoll ab.

   S schickt die Signatur (r, s) an A.
5. A stellt sicher, dass $r = r' \cdot g^{d'} \bmod p$.

4.3.1 Sicherheit

Für die Sicherheit ist es erforderlich, dass beim Erzeugen einer Signatur der Wert k zufällig erzeugt wird und dass dieser die Karte nicht verlässt. Wäre dem Kartenhersteller dieser Wert bekannt, so könnte er den privaten Schlüssel einfach berechnen. Werden zudem zwei Nachrichten mit dem gleichen k und dem gleichem privaten Schlüssel signiert, so kann der private Schlüssel ebenfalls berechnet werden [4].

Der Wert k wird hier genau wie der private Schlüssel erzeugt. Es können also zu k die gleichen Aussagen zur Sicherheit wie zu dem privaten Schlüssel gemacht werden.

Im Gegensatz zur Schlüsselerzeugung gibt es bei der Erzeugung einer Signatur für S die Möglichkeit, das Protokoll abzubrechen (wenn $s = 0$). Um A glaubhaft zu überzeugen, dass dieser Fall eingetreten ist, müsste S den privaten Schlüssel veröffentlichen. Der Fall, dass $s = 0$ ist, sollte jedoch sehr selten vorkommen. Es wird hier deshalb davon ausgegangen, dass S in diesem Fall das Protokoll nicht neu starten darf.

5 ECDSAX

ECDSAX ist eine Abwandlung des ECDSA (*Elliptic Curve Digital Signature Algorithm*, siehe Kapitel 2.1.2). Der ECDSA wird z.B. bei Bitcoin zum Signieren von Transaktionen eingesetzt. Es werden dort die standardisierten Kurvenparameter mit der Bezeichnung *secp256k1* [15, S.9] verwendet.

ECDSAX ist dem Verfahren ElGX sehr ähnlich, da der DSA (*Digital Signature Algorithm*) eine Variante des ElGamal Signaturverfahrens ist. Der ECDSA ist wiederum eine Variante des DSA, wobei in einer Gruppe der Punkte einer elliptischen Kurve gerechnet wird.

Es wurden auch hier nur die Schlüsselerzeugung und die Erzeugung von Signaturen im Vergleich zum ECDSA geändert. Das Format der Schlüssel und der Signaturen ist unverändert. Eine Signatur wird auch auf dieselbe Weise verifiziert wie beim ECDSA. Die Erzeugung der Schlüssel und der Signaturen geschieht wieder zwischen zwei Parteien, S (dem Security-Token) und A (dem Benutzer).

Ein ähnliches System wie ECDSAX, um Bitcoin gegen betrügerische Hardware-Hersteller abzusichern, wurde schon 2013 in einem Blog [16] veröffentlicht. Zum Zeitpunkt der Entwicklung von ECDSAX hatte ich aber keine Kenntnis von diesem Blogeintrag. Ich habe das Prinzip von ElGX auf den ECDSA übertragen.

5.1 Schlüsselerzeugung

Es seien:

- p eine Primzahl, die den endlichen Körper $\mathbb{F}_p$ bestimmt.
- $a, b \in \mathbb{F}_p$, die die elliptische Kurve $y^2 = x^3 + ax + b$ bestimmen.
- G ein Punkt auf dieser Kurve mit der Ordnung n.

Das Protokoll zur Schlüsselerzeugung soll diese Eigenschaften erfüllen:

- Der private Schlüssel d wird zufällig aus der Gleichverteilung auf $[1, n-1]$ gezogen.

- Nur S kennt den privaten Schlüssel.

A und S verwenden das folgende Protokoll, um ein neues Schlüsselpaar zu erzeugen:

1. Auf die verwendeten Kurvenparameter wird sich vorher geeinigt, z.B. secp256k1.

2. S zieht eine zufällige Zahl d' aus der Gleichverteilung auf $[1, n-1]$, berechnet $Q' := d' \cdot G$ und schickt Q' an A.

3. A zieht eine zufällige Zahl t aus der Gleichverteilung über $[0, n-1]$ und schickt t an S.

4. S berechnet $d := d' + t \bmod n$. Wenn $d = 0$: schicke d' an A und gehe zurück zu Schritt 2.

   S berechnet $Q := d \cdot G$ und schickt Q an A.

5. A stellt sicher, dass $Q = Q' \oplus t \cdot G$.

Der öffentliche Schlüssel ist dann Q, der private Schlüssel ist d.

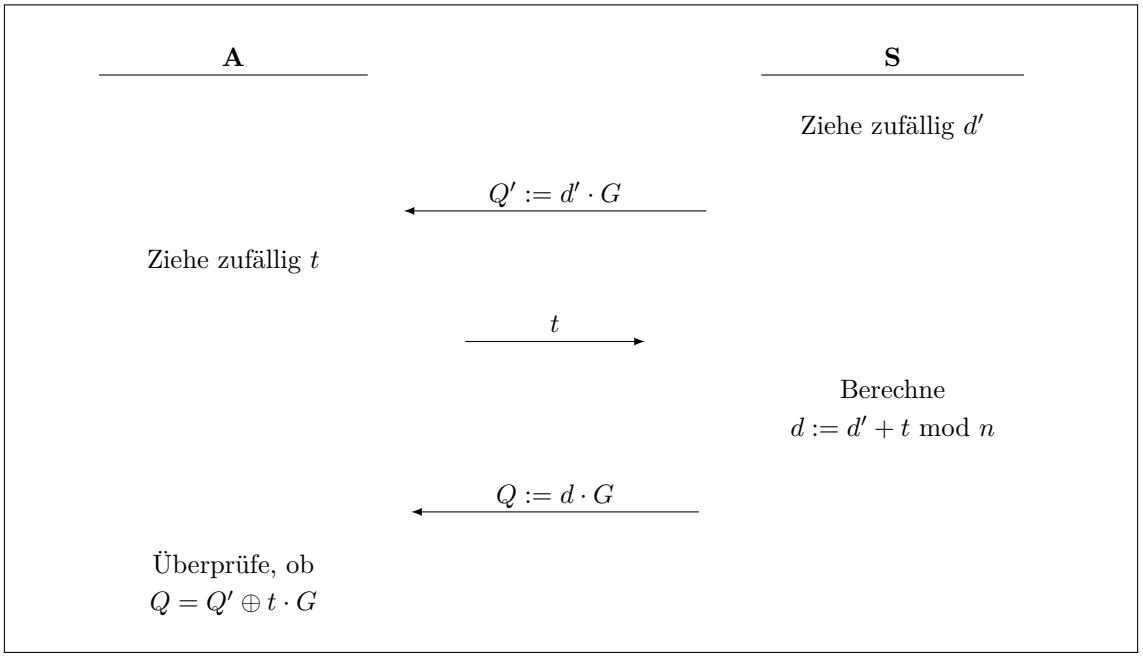

Abbildung 5.1: Vereinfachter Ablauf der ECDSAX-Schlüsselerzeugung.

5.1.1 Sicherheit

Die Schlüsselerzeugung funktioniert bei ECDSAX analog zu der Schlüsselerzeugung bei ElGX. Es wird nur in einer anderen Gruppe gerechnet. Solange es schwer ist, den diskreten Logarithmus in dieser Gruppe zu berechnen, können die Aussagen über die Sicherheit von ElGX hier übernommen werden.

Mit der Überprüfung, ob $Q = Q' \oplus t \cdot G$, kann A sicherstellen, dass S für den privaten Schlüssel d auch den richtigen Wert verwendet (aus $Q = Q' \oplus t \cdot G$ folgt, dass $d \equiv d' + t \pmod{n}$). S hat hier wieder keine Möglichkeit zu betrügen, wenn A sich korrekt verhält. Es kann dann Satz 3.1 angewendet werden. Versucht A zu betrügen, so sind die Voraussetzungen für den Satz wieder nicht gegeben (A kann t in Abhängigkeit von d' wählen).

5.2 Erzeugung einer Signatur

Es seien:

- H eine kollisionsresistente Hashfunktion.
- L_n die Bitlänge der Ordnung n von G.

Das Protokoll zur Erzeugung einer Signatur soll diese Eigenschaften erfüllen:

- Der Wert k wird zufällig aus der Gleichverteilung auf $[1, n-1]$ gezogen.
- Nur S kennt k.

A kann zusammen mit S mit dem folgendem Protokoll eine Nachricht m signieren.

1. A schickt z an S, wobei z die L_n höchstwertigen Bits von $H(m)$ sind.
2. S wählt eine zufällige Zahl k' aus der Gleichverteilung auf $[1, n-1]$, berechnet $P' := k' \cdot G$ und schickt P' an A.
3. A zieht eine zufällige Zahl t aus der Gleichverteilung auf $[0, n-1]$ und schickt t an S.
4. S berechnet $k := k' + t \bmod n$ und $P := k \cdot G = (x_1, y_1)$. Es sei $r := x_1 \bmod n$. Wenn $r = 0$, so schicke k' an A und gehe zu Schritt 2.

   S berechnet $s := k^{-1}(z + rd) \bmod n$. Wenn $s = 0$, melde einen Fehler und breche das Protokoll ab.

   S schickt P und s an A.

5. A stellt sicher, dass $P = P' \oplus t \cdot G$. A berechnet $r = x_1 \bmod n$. Die Signatur ist dann (r, s).

5.2.1 Sicherheit

Im ursprünglichen ECDSA Algorithmus wird nur r an A ausgegeben. Hier kennt A den Punkt $P = k \cdot G$. Dies liefert A aber keine wesentlichen neuen Informationen. Bei z.B. secp256k1 ist n annähernd so groß wie p. D.h. mit hoher Wahrscheinlichkeit ist $r = x1$. Hätte A nur r und somit den x-Wert von P zur Verfügung, könnte A mit der Kurvengleichung ($y^2 = x^3 + ax + b$) zwei mögliche Lösungen für y_1 berechnen.

Wie bei dem ElGamal-Signaturverfahren ist es für die Sicherheit hier wieder notwendig, dass der Hersteller des Security Tokens keine Möglichkeit hat, Kenntnis von k zu erlangen. Es dürfen auch wieder nicht mehrere Signaturen mit demselben k erzeugt werden. Der Wert k wird hier auf die gleiche Weise wie der private Schlüssel generiert. Somit können die gleichen Aussagen zur Sicherheit gemacht werden wie bei der Schlüsselerzeugung.

6 RSAX

RSAX [2] wurde wie auch ELGX von Dominik Reichl entwickelt. Es handelt sich dabei um eine Abwandlung des RSA-Verfahrens (siehe Kapitel 2.1.3). Bei RSAX wurde nur die Schlüsselerzeugung im Vergleich zu RSA geändert. Das erzeugte Schlüsselpaar kann auf die herkömmliche Weise sowohl zum Signieren wie auch zum Verschlüsseln eingesetzt werden. Die RSAX-Schlüsselerzeung geschieht wieder zwischen den zwei Parteien A (dem Benutzer) und S (dem Security-Token).

Die Sicherheit des RSAX-Verfahrens hängt von dem Parameter m ab. S kann bei RSAX ohne großen rechnerischen Aufwand versuchen zu betrügen (d.h. vom Protokoll abzuweichen, um ein nicht zufälliges Schlüsselpaar zu erzeugen). Es hängt dann vom Zufall ab, ob dieser Betrugsversuch von A erkannt wird. Es wird gezeigt, dass die Wahrscheinlichkeit, dass ein solcher Betrugsversuch erfolgreich ist und von A nicht erkannt wird, bei $\frac{2}{m}$ liegt.

6.1 Schlüsselerzeugung

Im Vergleich zur ursprünglichen RSAX-Schlüsselerzeugung von Dominik Reichl [2] wurden hier wieder kleine Änderungen vorgenommen. Es wurde unter anderem eine Verschlüsselungsfunktion durch eine Hashfunktion ersetzt.

Es seien:

- b_u, b_l feste Zahlen. Die erzeugten Primzahlen liegen in dem Intervall $[b_l, b_u - 1]$.
- $m \in \mathbb{N}_{\geq 2}$ eine feste Zahl (der Sicherheitsparameter).
- H eine kollisionsresistente Hashfunktion.
- $T : \mathbb{N} \to \mathbb{Z}_2$ eine Funktion, die überprüft, ob eine gegebene natürliche Zahl eine akzeptable Primzahl für das RSA-Verfahren ist. T gibt 1 aus, genau dann, wenn die gegebene Zahl akzeptabel ist.
- $P : [0, b_u - b_l - 1] \to [b_l, b_u - 1]$ eine Funktion, die für eine Eingabe x eine Primzahl $y := b_l + (x + i \bmod (b_u - b_l))$ mit $i \in \mathbb{N}$ zurückgibt, so dass $T(y) = 1$ und i minimal.

Das Protokoll zur Schlüsselerzeugung soll diese Eigenschaften erfüllen:

- Der Wert n ist das Produkt von zwei gleichverteilt zufällig gewählten Primzahlen.
- Nur S kennt die Primzahlen, aus denen n besteht.

A und S verwenden das folgende Protokoll um ein Schlüsselpaar zu erzeugen:

1. S zieht m zufällige Zahlen $a_1, ..., a_m$ aus der Gleichverteilung über $[0, b_u - b_l - 1]$ und schickt $(H(a_1), H(a_2), ..., H(a_m))$ an A.

2. A zieht m zufällige Zahlen $d_1, ..., d_m$ aus der Gleichverteilung über $[0, b_u - b_l - 1]$ und schickt $(d_1, ..., d_m)$ an S.

3. S berechnet für jedes $1 \leq i \leq m$: $p'_i := a_i + d_i \mod (b_u - b_l)$ und $p_i := P(p'_i)$. Es sei $\psi := \prod_{i=1}^{m} p_i$. S schickt ψ an A.

4. A erzeugt zufällig ein Paar (t_1, t_2) mit $t_1, t_2 \in \{1, 2, ..., m\}$ und $t_1 < t_2$. A schickt (t_1, t_2) an S.

5. Es sei $I := \{1, 2, ..., m\} \setminus \{t_1, t_2\}$. Für alle $i \in I$ schickt S a_i an A.

   S verwendet die Primzahlen p_{t_1} und p_{t_2}, um ein RSA-Schlüsselpaar zu erzeugen und schickt den öffentlichen Schlüssel (n, e) (wobei $n = p_{t_1} \cdot p_{t_2}$) an A.

6. Für alle $i \in I$ stellt A sicher, dass die Hashwerte der in Schritt 5 erhaltenen a_i den in Schritt 1 erhaltenen Werten entsprechen.

   A berechnet $z'_i := a_i + d_i \mod (b_u - b_l)$ für jedes $i \in I$ und $z_i := P(z'_i)$.

   Es sei $\gamma := \prod_{i \in I} z_i$. A stellt sicher, dass $n = \frac{\psi}{\gamma}$.

   A überprüft außerdem, ob $b_l^2 \leq n \leq b_u^2$ und ob $c \nmid n \ \forall c \in \{y \in \mathbb{P} \mid y \leq (\frac{b_u}{b_l})^2\}$.

6.1.1 Sicherheit

Es sei hier angemerkt, dass mit $P(p')$ eine Primzahl nicht gleichverteilt zufällig aus der Menge der akzeptablen Primzahlen gezogen wird, auch wenn p' gleichverteilt zufällig gewählt wird. Befindet sich vor einer Primzahl ein relativ großer Abstand zur vorhergegangenen Primzahl, so ist es wahrscheinlicher, dass $P(p')$ diese Primzahl ausgibt, als wenn dieser Abstand vergleichsweise kurz ist. Da Primzahlen aber relativ gleichmäßig verteilt sind [3, S. 134 f.], sollte die sich ergebende Verteilung der Primzahlen nahe genug an der Gleichverteilung sein. Eine andere Möglichkeit wäre es, P mit einem Pseudozufallszahlengenerator zu realisieren.

Szenario: *S* versucht zu betrügen

Hält sich A an das Protokoll und wählt für $i \in [1, m]$ die d_i gleichverteilt zufällig und unabhängig von den a_i, so sind die p'_i mit Satz 3.1 auch gleichverteilt zufällig. S muss die a_i wieder zuerst ziehen und kann diese somit nicht abhängig von den d_i wählen. Es kann hier jedoch nicht mit Sicherheit ausgesagt werden, dass S den Schlüssel mit den richtigen Werten berechnet hat.

Wie schon angemerkt wurde, kann S bei diesem Protokoll versuchen zu betrügen. Durch Überprüfung, ob $n = \frac{\psi}{\gamma}$, kann A kann für alle $i \in I$ sicherstellen, dass p_i von S richtig berechnet wurde. A kann jedoch nicht überprüfen, ob p_{t_1} und p_{t_2} von S richtig berechnet wurden. S hat also die Möglichkeit, in Schritt 3 für $j \in [1, m]$ ein p_j falsch zu berechnen (damit der Schlüssel kompromittiert ist, reicht es ein einzelnes p_j zu manipulieren). Es ist also $p_j \neq P(p'_j)$. S muss dann hoffen, dass A entweder $t_1 = j$ oder $t_2 = j$ wählt. Nehme o.B.d.A. an, dass $j = 1$. Die Anzahl der Möglichkeiten, um (t_1, t_2) zu wählen, liegt bei $\binom{m}{2}$. Die Anzahl der Möglichkeiten, um (t_1, t_2) so zu wählen, dass $t_1 \neq 1$, beträgt $\binom{m-1}{2}$. Also ist die Wahrscheinlichkeit für eine nicht erkannte Manipulation: [2]

$$1 - \frac{\binom{m-1}{2}}{\binom{m}{2}} = 1 - \frac{\frac{(m-1)\cdot(m-2)}{2}}{\frac{m\cdot(m-1)}{2}} = 1 - \frac{(m-1)\cdot(m-2)}{m\cdot(m-1)} = 1 - \frac{m-2}{m} = \frac{2}{m}$$

Die letzten beiden Tests, die A in Schritt 6 der Schlüsselerzeugung durchführt, sollen sicherstellen, dass S nicht zusätzlich zu p_{t_1} und p_{t_2} einen weiteren Faktor in n eingerechnet hat. Mit der Überprüfung, ob $n = \frac{\psi}{\gamma}$, kann A dies nicht erkennen, da S sowohl n als auch ψ um diesen Faktor erweitern kann. Also stellt A zuerst sicher, dass $b_l^2 \leq n \leq b_u^2$. Wenn das gilt, dann liegt n innerhalb des möglichen Wertebereichs von $p_{t_1} \cdot p_{t_2}$. Ein zusätzlicher (von A unentdeckter) Faktor kann jetzt maximal $\frac{b_u^2}{b_l^2} = (\frac{b_u}{b_l})^2$ groß sein. Diese möglichen Faktoren werden durch die Überprüfung, ob n durch kleine Primzahlen teilbar ist, ausgeschlossen. Es ist somit S nicht möglich, n um einen zusätzlichen Faktor zu erweitern. [2]

Der Wert e (Teil des öffentlichen Schlüssels) sollte hier nach Möglichkeit ein fester Wert sein (es muss jedoch $\text{ggT}(e, \varphi(n)) = 1$ gelten). Ist e für S frei wählbar, so könnte S für den privaten Schlüssel d einen beliebigen Wert wählen und e abhängig von d berechnen ($e \cdot d \equiv 1 \pmod{\varphi(n)}$). Der Hersteller von S könnte so Kenntnis von d erlangen.

Szenario: *A* versucht zu betrügen

Nehme an, A versucht nach der Schlüsselerzeugung auf die Werte p_{t_1} und p_{t_2} zu schließen. Alle Informationen, die A vorliegen und mit denen p_{t_1} und p_{t_2} berechnet

werden können, sind $H(a_{t_1})$, $H(a_{t_2})$ und n (und ψ). A müsste also in der Lage sein, entweder aus $H(x)$ einen Wert x zu bestimmen oder n zu faktorisieren.

Es lässt sich hier Satz 3.1 wieder nicht anwenden. A hat die Möglichkeit, für $i \in [1, m]$ die Werte d_i in Abhängigkeit der a_i zu bestimmen. Für die p'_i kann also keine Gleichverteilung garantiert werden.

6.2 Implementierbarkeit

Security-Tokens haben normalerweise nur sehr beschränkte Rechenkraft und ein Primzahltest für eine große Zahl ist eine relativ aufwendige Operation. Hier soll betrachtet werden, inwieweit sich RSAX auf einer Chipkarte implementieren lässt.

Es sei angenommen, dass die Berechnung von $P(p'_i)$ für ein zufälliges p'_i auf einer Chipkarte 5 Sekunden dauert. Nehme außerdem an, dass die Schlüsselerzeugung nicht länger als 5 Minuten dauern soll und dass die Rechenzeiten für die anderen Operationen zu vernachlässigen sind. Das heißt, dass m einen Wert von maximal 60 haben kann. In der Praxis dürfte dieser Wert derzeit bei der gegebenen Zeitbeschränkung jedoch noch deutlich niedriger liegen [17]. Mit $m = 60$ folgt, dass die Wahrscheinlichkeit für eine unentdeckte Manipulation von n bei $\frac{2}{60} \approx 3,33\%$ liegt.

Hätte ein erkannter Betrugsversuch keine weiteren Folgen, so wäre dieser Wert wohl deutlich zu hoch. Es wäre aber denkbar, RSAX noch mit anderen Sicherungsmaßnahmen zu kombinieren. Man könnte das Verfahren so implementieren, dass ein erkannter Manipulationsversuch zu einer Sperrung der Karte in der Software von A führt. Würde eine Karte dann versuchen, die Schlüsselerzeugung zu beeinflussen, so besteht bei $m = 60$ eine Wahrscheinlichkeit von $\frac{58}{60} \approx 96,67\%$, dass die Karte dadurch gesperrt würde.

Eine andere möglicherweise problematische Operation bei RSAX ist die Summe $\psi := \prod_{i=1}^{m} p_i$, da hier viele große Zahlen multipliziert werden müssen. Auch die Datenmenge, die übertragen werden muss, kann relativ groß werden. Will man einen Betrug mit einer Wahrscheinlichkeit von $99,99\%$ entdecken, so muss $m = 20.000$ sein. Bei einer angenommen Bitlänge der a_i und d_i von 2048 Bit müssen dann allein in Schritt 2 ca. $4,88$ Megabyte übertragen werden.

7 LamportX

LamportX ist eine Modifikation der Lamport-Signatur (siehe Kapitel 2.1.4). Es wurde nur die Schlüsselerzeugung geändert. Das Format einer Signatur ist unverändert, ebenso die Verifikation einer Signatur. Wie bei den anderen Verfahren erfolgt die Schlüsselerzeugung zwischen den Parteien A (dem Benutzer) und S (dem Security-Token).

LamportX hat Ähnlichkeit mit RSAX. Die Sicherheit ist hier auch von einem Parameter m abhängig. Die Karte hat die Möglichkeit, einen Betrug zu versuchen. Je größer der Wert m ist, desto wahrscheinlicher ist es, dass der Betrugsversuch von A erkannt wird.

Die Erzeugung des privaten Schlüssels folgt bei LamportX dem folgenden Prinzip: Es werden für jedes Bit der gehashten Nachricht nicht wie bei der Lamport-Signatur 2, sondern $2m$ zufällige Zahlen erzeugt. Für den privaten Schlüssel werden schließlich aber wieder nur 2 Zahlen verwendet. Für alle nicht verwendeten Zahlen ist es der kontrollierenden Partei A möglich sicherzustellen, dass diese richtig berechnet wurden.

7.1 Schlüsselerzeugung

Es seien:

- $k \in \mathbb{N}$
- $n := 2^k$
- H eine kollisionsresistente Hashfunktion mit $H : \{0,1\}^k \to \{0,1\}^k$
- $m \in \mathbb{N}_{\geq 2}$

Das Protokoll zur Schlüsselerzeugung soll diese Eigenschaften erfüllen:

- Die Werte $x_{i,j}$ des privaten Schlüssels werden gleichverteilt zufällig gewählt.
- Nur S kennt den privaten Schlüssel.

A und S verwenden das folgende Protokoll, um ein Schlüsselpaar für die Lamport-Signatur zu erhalten:

1. S zieht $a_{i,j,l}$ zufällig aus der Gleichverteilung auf $[0, n-1]$ für alle $i \in \{1, 2, ..., k\}$, $j \in \{0, 1\}$ und $l \in \{1, 2, ..., m\}$. Dies sind $2 \cdot k \cdot m$ Elemente.

   S schickt $y'_{i,j,l} := H(a_{i,j,l})$ an A für jedes $a_{i,j,l}$.

2. A zieht auf die gleiche Weise $2 \cdot k \cdot m$ zufällige Elemente $b_{i,j,l}$ aus der Gleichverteilung auf $[0, n-1]$ und schickt sie an S.

3. S berechnet $x_{i,j,l} := a_{i,j,l} + b_{i,j,l} \bmod n$ und schickt $y_{i,j,l} := H(x_{i,j,l})$ an A für alle $i \in \{1, 2, ..., k\}$, $j \in \{0, 1\}$ und $l \in \{1, 2, ..., m\}$.

4. A erzeugt zufällig $2k$ Werte $t_{i,j} \in \{1, 2, ..., m\}$ für alle $i \in \{1, 2, ..., k\}$ und $j \in \{0, 1\}$ und schickt sie an S.

5. Für alle $i \in \{1, 2, ..., k\}$, $j \in \{0, 1\}$ und $l \in \{1, 2, ..., m\} \setminus \{t_{i,j}\}$ schickt S $a_{i,j,l}$ an A.

6. Für feste i und j kann A jetzt für alle $l \in \{1, 2, ..., m\} \setminus \{t_{i,j}\}$ überprüfen, ob $x_{i,j,l}$ von S richtig berechnet wurde. Dazu stellt A zuerst sicher, dass $H(a_{i,j,l}) = y'_{i,j,l}$. Dann berechnet A jeweils $z_{i,j,l} := a_{i,j,l} + b_{i,j,l} \bmod n$ und überprüft, ob der Hashwert von $z_{i,j,l}$ mit dem in Schritt 3 von S erhaltenen $y_{i,j,l}$ übereinstimmt.

Der private Schlüssel besteht dann aus $x_{i,j} := x_{i,j,t_{i,j}}$, der öffentliche Schlüssel aus $y_{i,j} := y_{i,j,t_{i,j}}$ für alle $i \in \{1, 2, ..., k\}$ und $j \in \{0, 1\}$.

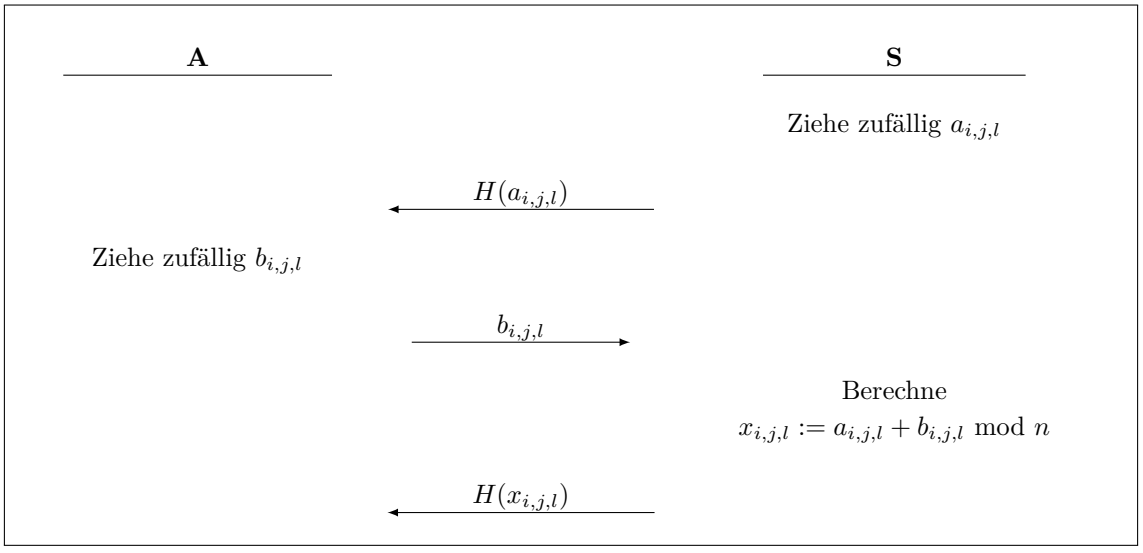

Abbildung 7.1: Die ersten drei Schritte der LamportX-Schlüsselerzeugung.

7.1.1 Sicherheit

Szenario: *S* versucht zu betrügen

Nehme hier wieder an, dass sich A korrekt verhält und ihre Zahlen $b_{i,j,l}$ unabhängig von den $a_{i,j,l}$ und gleichverteilt zufällig wählt. S muss wie bei den anderen Protokollen die Zahlen $a_{i,j,l}$ zuerst wählen. Somit können diese nicht von den $b_{i,j,l}$ abhängen. Mit Satz 3.1 folgt somit, dass die $x_{i,j,l}$ gleichverteilt zufällig gewählt werden. Analog zu RSAX kann hier aber nicht garantiert werden, dass S auch die richtigen Werte für den privaten Schlüssel verwendet. Sollte S versuchen, ein oder mehrere $x_{i,j,l}$ falsch zu berechnen, so wird dies mit einer bestimmten Wahrscheinlichkeit von A entdeckt werden.

Nehme an, dass S den Wert x_{i_0,j_0,l_0} falsch berechnet. A kann für alle $i \in \{1, 2, ..., k\}$, $j \in \{0, 1\}$ und $l \in \{1, 2, ..., m\} \setminus \{t_{i,j}\}$ überprüfen, ob $x_{i,j,l}$ von S richtig berechnet wurde. S kann also nur hoffen, dass $l_0 = t_{i_0,j_0}$. Für feste i, j kann S höchstens ein $x_{i,j,l}$ manipulieren. Die Wahrscheinlichkeit, dass die Manipulation eines einzelnen $x_{i,j,l}$ von A nicht erkannt wird, liegt bei $\frac{1}{m}$. Führt S erfolgreich einen solchen Betrug durch, so wäre damit nur ein einzelnes $x_{i,j}$ des privaten Schlüssel kompromittiert. Die Wahrscheinlichkeit, dass S es schafft, alle $x_{i,j}$ unentdeckt zu manipulieren, liegt bei $\frac{1}{m^{2k}}$.

Szenario: *A* versucht zu betrügen

Will A eines der $x_{i,j}$ des privaten Schlüssels bestimmen, so kann A nur versuchen von $H(a_{i,j})$ auf $a_{i,j}$, oder von $H(x_{i,j})$ auf $x_{i,j}$ zu schließen. Auch andere Urbilder von $H(x_{i,j})$ als $x_{i,j}$ wären als Teil des privaten Schlüssels verwendbar.

Wie immer in diesem Szenario lässt sich Satz 3.1 nicht anwenden, da A ihre Werte $b_{i,j,l}$ ziehen muss, nachdem sich S schon auf ihre Werte $a_{i,j,l}$ festgelegt hat. A kann also die $b_{i,j,l}$ in Abhängigkeit der $a_{i,j,l}$ wählen.

7.1.2 Implementierbarkeit

Bei einer Implementierung von LamportX könnte sich die große Menge an Zahlen, mit denen gearbeitet wird, als Problem erweisen. Ein typischer Wert für k ist 256. Will man eine Manipulation von S an einem einzelnen $x_{i,j}$ mit einer Wahrscheinlichkeit von 99% entdecken, so muss $m = 100$ sein. Mit diesen Werten müssten in dem Protokoll dann insgesamt über 6 Megabyte an Daten übertragen werden. Das Security-Token müsste zudem mehrere Megabyte an Daten zwischenspeichern. Auf Chipkarten ist besonders Speicher eine knappe Ressource.

8 Vergleich mit einem realen Bitcoin Wallet

Die in dieser Arbeit vorgestellten Verfahren sollen an dieser Stelle mit dem Bitcoin Wallet CoolWallet [18] des Herstellers CoolBitX verglichen werden. Dieses Wallet hat die Form einer herkömmlichen Chipkarte und besitzt ein Display und einen Knopf. Über Bluetooth kann das Wallet mit einer Smartphone Applikation kommunizieren.

Die Schlüsselerzeugung funktioniert beim CoolWallet so, dass zuerst zufällig ein Seed erzeugt wird. Aus dem Seed wird dann deterministisch das Schlüsselpaar generiert. Der Seed soll auf Papier abgeschrieben werden, um den privaten Schlüssel bei Verlust des Wallets wiederherstellen zu können. Es gibt zwei Möglichkeiten, den Seed zu erzeugen:

1. Der Seed wird von der Karte generiert und auf deren Display angezeigt.

2. Der Seed wird von der Smartphone Applikation des Herstellers erzeugt, dort angezeigt und dann zur Karte übertragen.

Es ist mit Programmen von anderen Herstellern möglich zu überprüfen, ob der verwendete Seed wirklich zu dem erzeugtem öffentlichen Schlüssel führt. Wäre dies nicht so, würde auch ein Wiederherstellen des privaten Schlüssel aus dem Seed nicht funktionieren. Es sei auch angenommen, dass die Funktion, die den Seed auf einen privaten Schlüssel abbildet, nicht eine kleinere Bildmenge hat als die Menge aller möglicher Seeds.

Hätte der Hersteller bösartige Absichten, so würden beide Methoden, den Seed zu erzeugen, keinen wirklichen Schutz bieten. Bei Möglichkeit 1 könnte die Karte durch den Hersteller so manipuliert worden sein, dass der Seed nicht gleichverteilt zufällig erzeugt wird. Die Menge der möglichen Schlüsselpaare könnte damit viel kleiner sein und vom Hersteller komplett berechnet werden. Bei der 2. Methode könnte diese Manipulation von der Smartphone Applikation des Herstellers vorgenommen werden.

Man könnte die Schlüsselerzeugung gegen diese Angriffsmöglichkeit absichern, indem eine Applikation verwendet wird, die nicht vom gleichen Hersteller wie die Karte stammt. Der Seed müsste dann in dieser Anwendung erzeugt werden. Würde

man dieser Applikation vertrauen, so wäre sichergestellt, dass der Seed gleichverteilt zufällig erzeugt wird. Außerdem könnte dann direkt überprüft werden, ob der öffentliche Schlüssel korrekt aus dem Seed berechnet wurde.

Es wäre ebenfalls möglich, die Applikation des Kartenherstellers zu verwenden und den Seed vom Benutzer selber wählen zu lassen. Dieser könnte mit einem anderen Programm überprüfen, ob der öffentliche Schlüssel korrekt berechnet wurde. Ein Problem dabei wäre, dass die Benutzer ihre Seeds wahrscheinlich nicht gleichverteilt zufällig wählen. Daraus könnten sich wieder Angriffsmöglichkeiten ergeben.

Bei beiden Lösungsvorschlägen wird jedoch der Seed vom Smartphone gesehen. Dies stellt ein Problem dar, da ein Angreifer theoretisch mithilfe von Malware auf dem Gerät in Besitz des privaten Schlüssels gelangen könnte.

Ein Vorteil der Schlüsselerzeugung des CoolWallets ist, dass es eine Möglichkeit gibt, den Schlüssel von einer verlorenen oder defekten Karte wiederherzustellen. Dies ist bei den in dieser Arbeit vorgestellten Verfahren nicht möglich. Der private Schlüssel ist bei allen diesen Verfahren nur der Karte bekannt.

9 Implementierung

Die in Kapitel 5 vorgestellte Abwandlung des ECDSA wurde im Rahmen dieser Masterarbeit in Form eines Prototyps implementiert. Die Partei A wurde dabei mit einer Smartphone Applikation, die Partei S mit einer Chipkarte realisiert. Gemeinsam mit beiden Geräten ist es möglich, neue Schlüsselpaare zu erzeugen und beliebige Zeichenketten zu signieren.

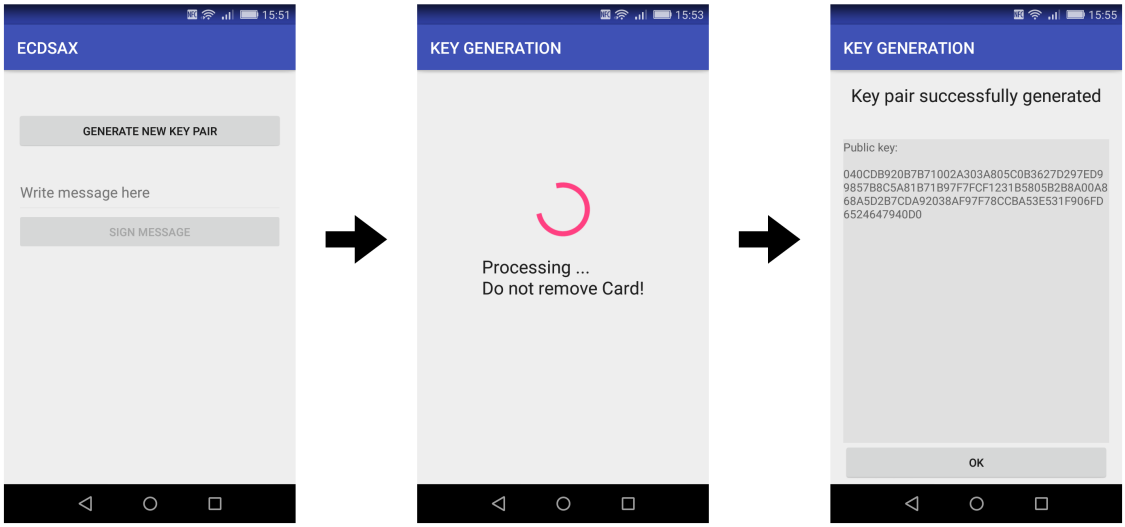

Abbildung 9.1: Anzeige des Android Smartphones beim Erzeugen eines Schlüsselpaares.

In dieser Implementierung wurden die gleichen Kurvenparameter verwendet, die auch bei Bitcoin zum Signieren von Transaktionen zum Einsatz kommen. Diese Standarisierung wird mit secp256k1 [15, S.9] bezeichnet. Die Bitlänge beträgt 256 Bit, die Kurvengleichung ist $y^2 = x^3 + 7$. Die vorgestellte Implementierung ließe sich theoretisch noch zu einem Bitcoin Wallet erweitern. Es können aber natürlich auch andere Parameter verwendet werden.

9.1 Verwendete Technologien

Die Partei des Benutzers A wurde mit einer Android Applikation in Java umgesetzt. Die Kommunikation mit der Chipkarte erfolgt über NFC (*Near Field Communication*). Das bedeutet, die Karte muss für eine Datenübertragung unmittelbar an das

Smartphone gehalten werden. Die gewählte Chipkarte arbeitet mit Java Card. Konkret wurde als Chipkarte das Modell J3D081 von NXP verwendet. Es kommt auf dieser Karte die Java Card Version 3.0.1 Classic zum Einsatz. Aufgrund des kleinen Formfaktors und der niedrigen Kosten einer Chipkarte sind Ressourcen wie Speicher und Rechenleistung sehr beschränkt. Für aufwendige Rechenoperationen ist in der Regel eine Hardwareunterstützung notwendig. Häufig werden kryptographische Operationen wie AES oder RSA unterstützt.

9.1.1 Java Card

Java Card stellt eine Untermenge von Java dar, die auf Chipkarten verwendet werden kann. Auf einer Chipkarte mit Java Card können sich mehrere Anwendungen befinden und unabhängig voneinander ausgeführt werden. Die Anwendungen werden als Applets bezeichnet. Die Syntax von Java Card ist dieselbe wie bei Java. Viele höhere Konzepte wie z.B. Threads existieren bei Java Card jedoch nicht. Oft wird noch nicht einmal der Datentyp Integer unterstützt. [19, S. 2.1 ff.]

Wie auch bei Java werden Java Card Applets auf einer Karte in einer Laufzeitumgebung (*Java Card Runtime Environment*, JCRE) ausgeführt. Sie besteht unter anderem aus der *Java Card Virtual Machine* (JCVM) und der Programmierschnittstelle (API). Damit ein Applet auf einer Karte ausgeführt werden kann, muss dieses zuerst installiert werden. Danach kann das Applet über einen Befehl ausgewählt werden. [20, S. 3.1 ff.]

Die Kommunikation zwischen einem Lesegerät und einer Karte erfolgt nach dem Master-Slave Prinzip. Es wird ein Befehl an die Karte gesendet, welche daraufhin eine Antwort schickt. Die Befehle werden als command APDUs (*Application Protocol Data Unit*), Antworten als response APDUs bezeichnet. Der Aufbau der APDU-Nachrichten ist in Abbildung 9.2 gezeigt. [21]

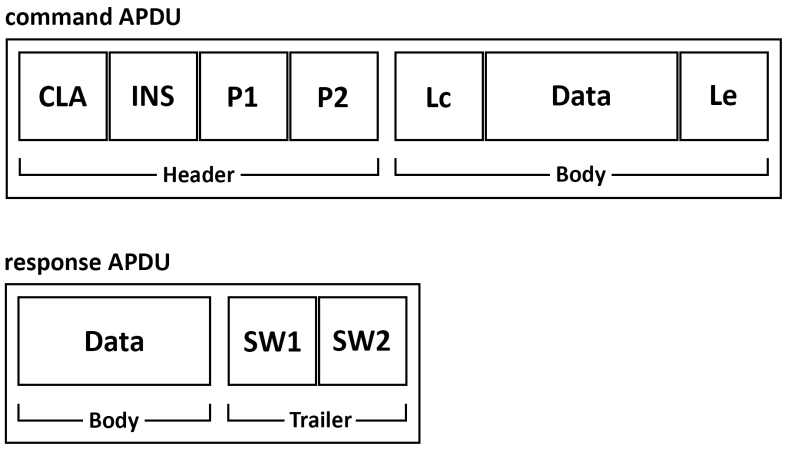

Abbildung 9.2: Aufbau der command APDUs und response APDUs.

Die Felder einer command APDU haben die folgende Bedeutung [21]:

- **CLA**: Klasse von Befehlen.
- **INS**: Gibt den Befehl an.
- **P1, P2**: Parameter für den Befehl.
- **Lc**: Anzahl der Bytes im Data-Feld.
- **Data**: Daten, die optional übermittelt werden können.
- **Le**: Die maximale Anzahl erwarteter Bytes der Daten in der response APDU.

Werden keine Daten übermittelt, so kann der Body einer command APDU weggelassen werden. Eine response APDU kann ebenfalls optional Daten enthalten. Außerdem werden immer zwei Bytes für den Status übermittelt (SW1 und SW2). Ein erfolgreich ausgeführter Befehl wird beispielsweise mit den Status-Bytes SW1=0x90[1] und SW2=0x00 bestätigt. [21]

Objekte werden bei Java Card in nichtflüchtigem Speicher gehalten. Eine automatische Speicherbereinigung (*Garbage Collection*) ist zudem normalerweise nicht verfügbar [19, S. 2.5]. Ein Entwickler muss somit sparsam in der Erzeugung von Objekten wie Arrays sein, um den geringen Speicher nicht überzubelegen. Das hier verwendete Chipkartenmodell hat 80 Kilobyte nichtflüchtigen Speicher (*Electrically Erasable Programmable Read-Only Memory*, EEPROM).

9.2 Kommunikation

Im Folgenden wird beschrieben, welche konkreten Daten das Smartphone und die Karte austauschen. Da die Schnittstelle von Java Card nicht Zugang zu allen benötigten Rechenoperationen liefert, musste bei der tatsächlichen Implementierung von dem hier vorgestellten Protokoll leicht abgewichen werden. Diese Anpassungen werden in Kapitel 9.3 beschrieben. Es ist technisch jedoch möglich, die Kommunikation wie unten stehend umzusetzen. Dazu ist jedoch ein Zugriff auf tiefer liegende Funktionen der Chipkarte notwendig.

Bei Java Card wird ein Applet durch eine AID (*Application Identifier*) identifiziert. Diese kann frei gewählt werden und sei hier z.B. durch die Folge dieser Bytes gegeben:

```
APPLET_AID = 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0xFF
```

[1] Mit dem Präfix 0x werden hexadezimale Werte gekennzeichnet.

Außerdem werden für das Applet noch die Bytes für eine Klasse (CLA) und für zwei Befehle (INS) festgelegt:

```
CLA_APP = 0x80
INS_NEWKEY = 0x00
INS_SIGN = 0x02
```

Das Befehl-Byte `INS_NEWKEY` wird bei dem Protokoll zum Erzeugen eines neuen Schlüsselpaares in den APDUs angegeben, das Byte `INS_SIGN` dementsprechend bei dem Protokoll zum Erzeugen einer Signatur. Durch die P1- und P2-Bytes werden die Nachrichten innerhalb der Protokolle unterschieden.

9.2.1 Schlüsselerzeugung

Damit das Smartphone mit dem Applet auf der Chipkarte kommunizieren kann, muss das Applet zuerst ausgewählt werden. Dazu schickt das Smartphone die folgende APDU an die Karte:

```
0x00 0xA4 0x04 0x00 0x0A APPLET_AID 0xFF
```

Die Felder CLA, INS etc. sind bei diesem Befehl von Java Card vorgeschrieben.

Das Smartphone initiiert dann mit dem folgendem Befehl eine Schlüsselerzeugung (diese APDU enthält keine Daten und damit keinen Body):

```
CLA_APP INS_NEWKEY 0x00 0x00
```

Daraufhin zieht die Karte eine zufällige Zahl d' und schickt als Antwort $Q' := d' \cdot G$ an das Smartphone (siehe Kapitel 5.1).

Nach Erhalt der Antwort erzeugt das Smartphone ebenfalls eine zufällige Zahl t und schickt sie an die Karte:

```
CLA_APP INS_NEWKEY 0x01 0x00 T_LENGTH T 0xFF
```

Die Karte berechnet dann $d := d' + t \bmod n$ und überprüft, ob $d = 0$. Falls ja, so wird mit einer Fehlermeldung geantwortet und das Protokoll abgebrochen. Falls nein, so berechnet die Karte $Q := d \cdot G$ und schickt Q an das Smartphone. Für die Karte ist das Schlüsselpaar (Q, d) damit erfolgreich erzeugt.

Das Smartphone überprüft, ob $Q = Q' + t \cdot G$. Falls ja, so ist auch für das Smartphone die Schlüsselerzeugung erfolgreich abgeschlossen. Der öffentliche Schlüssel Q wird dauerhaft gespeichert.

9.2.2 Erzeugung einer Signatur

Für eine Nachricht soll eine Signatur erstellt werden. Als erstes muss wieder über einen Befehl die Applikation auf der Chipkarte ausgewählt werden. Dann berechnet

das Smartphone den Hashwert der Nachricht. Dieser wird mit folgender APDU an die Karte geschickt:

```
CLA_APP INS_SIGN 0x00 0x00 HASH_LENGTH HASH 0xFF
```

Die Karte erzeugt dann eine zufällige Zahl k' und schickt als Antwort $P' := k' \cdot G$.

Das Smartphone erzeugt nun zufällig t und schickt es an die Karte:

```
CLA_APP INS_SIGN 0x01 0x00 T_LENGTH T 0xFF
```

Mit diesem Wert kann die Karte dann $k := k' + t \bmod n$ berechnen. Als nächstes berechnet die Karte $P := k \cdot G = (x_1, y_1)$. Falls $r := x_1 \bmod n = 0$, so wird k' an das Smartphone geschickt und das Protokoll wiederholt. Schließlich wird $s := k^{-1}(z + rd) \bmod n$ berechnet. Falls $s = 0$, so wird mit einer Fehlermeldung abgebrochen. Die Karte schickt P und s zum Smartphone.

Es wird nun vom Smartphone sichergestellt, dass $P = P' + t \cdot G$. Falls ja, so ist (r, s) mit $r = x_1 \bmod n$ eine gültige Signatur für die Nachricht.

9.3 Probleme und Lösungen

Wie schon angemerkt wurde, liefert Java Card nur beschränkte Möglichkeiten für Berechnungen, die nicht durch die API zur Verfügung gestellt werden. Viele grundlegende Rechenschritte, wie z.B. die Addition von großen Zahlen, mussten von Hand implementiert werden. Solche großen Zahlen werden hier direkt als Byte-Arrays repräsentiert. Für manche Berechnungen, wie die modulare Multiplikation von großen Zahlen, wäre eine eigene Implementierung ohne Hardwareunterstützung zu langsam gewesen. Es wurde in solchen Fällen dann versucht, mithilfe von Funktionen der API und durch Rechentricks eine effiziente Implementierung zu realisieren. Da dies aber auch nicht immer möglich war, musste teilweise das ursprüngliche Protokoll leicht abgeändert werden. Alle hier benötigten Operationen, die nicht von der API zur Verfügung gestellt werden, werden jedoch im Grunde von der Hardware der Karte unterstützt. Theoretisch lässt sich das Protokoll also ohne die in diesem Kapitel beschriebenen Änderungen auf einer solchen Chipkarte implementieren. Die Änderungen gelten somit nur für den hier vorgestellten Prototyp und werden auch nicht tief gehend auf ihre Sicherheit untersucht.

9.3.1 Skalarmultiplikation von Punkten

Die Skalarmultiplikation für Punkte auf elliptischen Kurven wird von Java Card nicht direkt unterstützt. Dies muss im Protokoll aber an mehreren Stellen von der Karte berechnet werden. Die Berechnung von $Q' := d' \cdot G$ bei der Schlüsselerzeugung und von $P' := k' \cdot G$ bei der Erzeugung einer Signatur lässt sich problemlos

bewerkstelligen. Die Werte d' beziehungsweise k' sind hier zufällig. Somit kann die in der API zur Verfügung stehende Funktion zur Erzeugung eines neuen zufälligen ECDSA-Schlüsselpaares verwendet werden. Der resultierende öffentliche Schlüssel entspricht dann Q' beziehungsweise P'.

Anders sieht die Situation bei der Berechnung von $Q := d \cdot G$ bei der Schlüsselerzeugung und bei Berechnung von $P := k \cdot G$ bei der Erzeugung einer Signatur aus. Die Zahlen d beziehungsweise k sind hier nicht zufällig, sondern vorgegebene Werte. Es ist mit der Java Card API hier nicht möglich, Q beziehungsweise P vollständig und effizient zu berechnen. Mithilfe der Funktionalität für die Diffie-Hellman-Schlüsselvereinbarung für elliptische Kurven ist es aber möglich, den x-Wert einer solchen Skalarmultiplikation effizient zu berechnen. Die Protokolle der Schlüsselerzeugung und der Erzeugung einer Signatur wurden deshalb so abgeändert, dass die Karte jeweils nur den x-Wert der Punkte Q und P an das Smartphone schickt. Bei der Schlüsselerzeugung überprüft das Smartphone dann, ob der x-Wert des Punktes $Q' + t \cdot G$ dem von der Karte erhaltenem x-Wert entspricht. Diese Änderung wurde analog bei der Erzeugung einer Signatur vorgenommen. Da das Smartphone den y-Wert selber korrekt berechnet, kann die Karte den öffentlichen Schlüssel (und damit den privaten Schlüssel) nicht abweichend vom Protokoll festlegen.

Damit die Karte auch in Besitzt des vollständigen öffentlichen Schlüssels ist, wird der y-Wert von Q am Ende des Protokolls zur Schlüsselerzeugung noch vom Smartphone an die Karte geschickt:

```
CLA_APP INS_NEWKEY 0x02 0x00 Y_LENGTH Y 0xFF
```

Bei gegebenen x-Wert kann y zwei unterschiedliche Werte annehmen. Diese sind durch die Kurvengleichung $y^2 = x^3 + 7$ bestimmt. Das Smartphone hätte keinen Nutzen davon, einen falschen y-Wert an die Karte zu schicken. Der private und der öffentliche Schlüssel würden dann nicht zusammenpassen.

9.3.2 Modulare Multiplikation

Bei der Erzeugung einer Signatur muss die Chipkarte den Wert $s := k^{-1}(z + rd) \bmod n$ berechnen. Das erfordert zwei modulare Multiplikationen. Eine eigene Implementierung ist hier ineffizient. Es wurde deshalb die Funktionalität zur Verschlüsselung mit RSA der API zur Hilfe genommen. Damit ist es möglich, für eine Zahl x den Wert $x^2 \bmod n$ zu berechnen. Es soll nun für zwei Zahlen a und b der Wert $ab \bmod n$ berechnet werden. Mit Hilfe der binomischen Formel wird dies mit der folgenden Idee umgesetzt: $(a+b)^2 - a^2 - b^2 = 2ab$. Addition und Subtraktion sind effizient möglich. Somit muss nun nur noch durch 2 geteilt werden. Da Modulo n gerechnet wird, geht das nicht immer durch einen Bit-Shift. Der genaue Algorithmus funktioniert daher folgendermaßen (in Pseudocode, `sqr(x)` berechnet $x^2 \bmod n$):

```
multiply(a,b) {
    if (a is even) then
```

```
        return sqr(shift_right(a)) + sqr(b) - sqr(shift_right(a) -
            b);
    else if (b is even) then
        return sqr(a) + sqr(shift_right(b)) - sqr(a -
            shift_right(b));
    else
        return sqr(shift_right(a+1)) + sqr(b) - sqr(b -
            shift_right(a+1)) - b;
}
```

9.4 Performance

Es ergeben sich folgende Ausführungszeiten für die beiden Protokolle (Durchschnitt aus 10 Ausführungen):

- **Schlüsselerzeugung:** $0,71\ s$
- **Erzeugung einer Signatur:** $8,67\ s$

Für den Prototypen musste gewisse Funktionalität, die nicht über die API verfügbar ist, selbst implementiert werden. D.h. diese Berechnungen werden nicht auf spezialisierter Hardware ausgeführt. Das macht sich in der Performance bemerkbar. Die Erzeugung einer Signatur dauert relativ lange, da auf umständlichem Weg die modularen Multiplikationen berechnet werden müssen. Im Gegensatz zur Schlüsselerzeugung wird die Operation zur Erzeugung eine Signatur in der Praxis häufig ausgeführt. Eine Verbesserung der Ausführungszeit wäre also wünschenswert.

Würde man die Hardware einer Chipkarte speziell für das implementierte Protokoll entwickeln oder hätte man Zugriff auf eine umfangreichere API, so wäre eine performante Implementierung möglich. Die Rechenschritte unterscheiden sich nicht wesentlich vom herkömmlichem ECDSA, welcher auf der verwendeten Chipkarte schnell ausgeführt werden kann.

10 Fazit

10.1 Zusammenfassung

In dieser Arbeit wurden Abwandlungen für vier verschiedene Signaturverfahren vorgestellt. Die Schlüsselerzeugung funktioniert bei diesen geänderten Verfahren über ein Protokoll zwischen einem Benutzer und einem Security-Token. Bei ElGX und bei ECDSAX musste zusätzlich auch die Erzeugung einer Signatur im Vergleich zum ElGamal-Signaturverfahren bzw. ECDSA geändert werden. Dies hat den Grund, dass dort die Signaturen mithilfe eines zufälligen Wertes erzeugt werden.

Ziel all dieser Protokolle sollte es sein, einem Hersteller keine Möglichkeit zu geben, eine Schlüsselerzeugung auf einem von ihm hergestelltem Security-Token zu implementieren, die zu Schlüsseln von schlechter Qualität führt. Das bedeutet, dass die zufälligen Werte, die bei der Schlüsselerzeugung verwendet werden, auch wirklich zufällig erzeugt werden müssen.

Wird auf einem Security-Token ElGX oder ECDSAX eingesetzt, so hat der Hersteller des Tokens keine Möglichkeit die Verfahren so zu implementieren, dass er Kenntnis von den privaten Schlüsseln der Nutzer erlangen kann. Bei RSAX und LamportX kann der Hersteller versuchen die Schlüsselerzeugung auf dem Security-Token zu manipulieren. Dies wird jedoch mit einer bestimmten Wahrscheinlichkeit von dem Benutzer entdeckt. Diese Wahrscheinlichkeit kann über einen Parameter gesteuert werden. Je niedriger aber die Wahrscheinlichkeit eines erfolgreichen Betrugsversuches, desto höher ist der Rechenaufwand und die Menge der zu übertragenden Daten.

Die Relevanz des Themas von betrügerischen Herstellern wurde anhand des Beispiels von Bitcoin Wallets näher gebracht. Da der ECDSA bei Bitcoin zum Einsatz kommt, wurde somit in dieser Arbeit das ECDSAX Verfahren implementiert. Das Security-Token wurde über eine Chipkarte mit Java Card realisiert. Für den Part des Benutzers wurde eine Android-Applikation programmiert. Die Kommunikation zwischen dem Smartphone und der Karte erfolgt über NFC. Da auf einer Chipkarte nur mit starken Beschränkungen programmiert werden kann, gab es dort bei der Implementierung einige Schwierigkeiten. Über Rechentricks konnten einige dieser Hindernisse überwunden werden. Teilweise mussten für den Prototyp die Protokolle aber leicht abgeändert werden, um eine effiziente Umsetzung zu ermöglichen.

10.2 Ausblick

Die Ergebnisse in dieser Arbeit führen zu folgenden Überlegungen bezüglich Verbesserung und Erweiterung der Protokolle und der Implementierung:

- Für einen realen Einsatz sind RSAX und LamportX wahrscheinlich noch nicht ausgereift genug. Damit ein Betrug von S annähernd ausgeschlossen ist, muss der Sicherheitsparameter einen relativ hohen Wert haben. Das hat bei den Verfahren aber zur Folge, dass die Menge der zu sendenden Daten und die Zahl der Rechenschritte für heutige Tokens vermutlich zu hoch ist. Eine interessante Frage wäre also, ob für das RSA- und das Lamport-Verfahren bessere Protokolle möglich sind. Kurz vor Fertigstellung dieser Arbeit hat Dominik Reichl noch das Verfahren RSAX_2 [22] entwickelt. Eine Untersuchung müsste zeigen, ob dieses Protokoll die genannten Probleme für RSA lösen kann.

- Bei allen vorgestellten Protokollen hat A keine Kenntnis von dem privaten Schlüssel. Sollte das Security Token verlorengehen, so hat hier der Nutzer keine Möglichkeit, den Schlüssel wiederherzustellen. Wird z.B. der ECDSAX für ein Bitcoin Wallet eingesetzt, so könnten hier große Geldbeträge unwiederbringlich verlorengehen. Es bleibt die Frage, ob eine Backup-Lösung möglich ist, ohne dass A den privaten Schlüssel kennt. Denkbar wäre eine Lösung, bei der der gleiche Schlüssel auf zwei oder mehr Tokens erzeugt werden kann.

- Bei der Implementierung des ECDSAX in dieser Arbeit dauert die Erzeugung einer Signatur noch deutlich zu lange. Die Ursache war hier, dass bestimmte Funktionalität nicht über die Java Card API verfügbar war. Hier wäre eine effiziente Realisierung wünschenswert.

Literaturverzeichnis

[1] Dominik Reichl. ElGX - An ElGamal-Based Digital Signature System, 2015.

[2] Dominik Reichl. RSAX, 2015.

[3] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[4] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In GeorgeRobert Blakley and David Chaum, editors, *Advances in Cryptology*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer Berlin Heidelberg, 1985.

[5] Carl Friedrich Gauss. *Untersuchungen über höhere Arithmetik*. Hermann Maser, Julius Springer, Berlin, 1889.

[6] American National Standards Institute Standards Committee Financial Services. *Public Key Cryptography for the Financial Services Industry - the Elliptic Curve Digital Signature Algorithm (ECDSA): ANSI American National Standard for Financial Services, ANS X9.62-2005*. American national standard / ANSI. Accredited Standards Committee X9, Incorporated, 2005.

[7] Certicom Research. Standards for efficient cryptography, SEC 1: Elliptic curve cryptography, 2009. Version 2.0.

[8] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[9] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[10] Leslie Lamport. Constructing digital signatures from one way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, 1979.

[11] Johannes Buchmann, Erik Dahmen, and Michael Szydlo. Hash-based digital signature schemes, 2008.

[12] U. Krengel. *Einführung in die Wahrscheinlichkeitstheorie und Statistik*. vieweg studium; Aufbaukurs Mathematik. Vieweg+Teubner Verlag, 2015.

[13] drhab (`http://math.stackexchange.com/users/75923/drhab`). Sum modulo of two random variables with one uniformly distributed. Mathematics Stack Exchange, März 2016. `http://math.stackexchange.com/q/1683240`.

[14] Richard Rex McKnight. Individual bit security of the discrete logarithm: Theory and implementation using oracles. Master thesis, Ludwig-Maximilians-Universität München, 2007.

[15] Certicom Research. Standards for efficient cryptography, SEC 2: Recommended elliptic curve domain parameters, 2010. Version 2.0.

[16] No subliminal Channel. `https://web.archive.org/web/20150602025255/http://firmcoin.com/?p=52`, Juni 2013.

[17] Chenghuai Lu, Andre L. M. dos Santos, and Francisco R. Pimentel. Implementation of fast rsa key generation on smart cards. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, SAC '02, pages 214–220, New York, NY, USA, 2002. ACM.

[18] CoolBitX CoolWallet. `https://coolbitx.com/coolwallet/`, März 2016.

[19] Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Classic Edition*, 2009.

[20] Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Classic Edition*, 2009.

[21] ISO/IEC 7816-4:2013 Identification cards - Integrated circuit cards - Part 4: Organization, security and commands for interchange. Standard, International Organization for Standardization, Geneva, CH, 2013.

[22] Dominik Reichl. RSAX_2, 2016.

Erklärung

Hiermit erkläre ich, dass ich diese schriftliche Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe.

Tübingen, den 28. März 2016

_____

Raphael Adam