Kryptographische Systeme auf Basis des diskreten Logarithmus

Inhaltsverzeichnis

1 Einführung

Neben der Faktorisierungsannahme gibt es weitere Annahmen, die für kryptographische Systeme eine wichtige Rolle spielen. So z.B. auch die **Diskrete-Logarithmus-Annahme**, welche darauf beruht, dass Potenzieren in zyklischen Gruppen „leicht“ ist, dagegen das Logarithmieren deutlich „schwieriger“ ist.

Praktische Bedeutung erhielten die in diesem Versuch vorgestellten kryptographischen Verfahren, aufgrund der Möglichkeit sie auf effizientere Gruppen (z.B. den elliptischen Kurven) anzuwenden.

Einführend wird erläutert, was sich hinter der diskreten Logarithmus-Annahme verbirgt und anschließend gezeigt, wie sie im Diffie-Hellman-Schlüsselaustausch-Verfahren und dem ElGamal-Konzelations- und Signaturverfahren zum Einsatz kommt.

1.1 Potenzieren

Potenzieren im Bereich natürlicher, ganzer, reeller oder sogar komplexer Zahlen ist eine grundlegende Operation der Mathematik:

$$g^e = y \qquad (g, e, y \in \mathbb{N}, \mathbb{Z}, \mathbb{R}, \mathbb{C}) \tag{1}$$

1.2 Logarithmieren

Die Umkehrfunktion zum Potenzieren in Bezug auf das Auffinden der Exponenten ist das Logarithmieren.

$$\log_g y = e \qquad (g, e, y \in \mathbb{N}, \mathbb{Z}, \mathbb{R}, \mathbb{C}) \tag{2}$$

Hierbei ist es nicht unüblich, dass dieser Exponent außerhalb des Definitionsbereichs liegt (z.B. $\log_{10} 8 \approx 0,9$). Nichtsdestotrotz lässt sich dieser Exponent mit Hilfe von Computern in sehr schneller Zeit (näherungsweise) errechnen. Beispielhaft wäre da die Berechnung über die Potenzreihenentwicklung.

1.3 Potenzieren mod p

Durch die Einschränkung des Wertebereichs in den Restklassenring modulo p lässt sich das Problem der Potenzberechnung vereinfachen, indem der im Praktikum „Zahlentheoretische Algorithmen“ beschriebene Square-and-Multiply-Algorithmus verwendet wird. Somit lässt sich festhalten, dass das Berechnen der Potenz im Restklassenring modulo p „leicht“ ist.

1.4 Logarithmieren mod p

Die Einschränkung, die das Berechnen der Potenz erleichtert hat, erschwert nun das Berechnen des Logarithmus im Restklassenring modulo p. Denn Näherungsverfahren wie die Potenzreihenentwicklung sind hier ungeeignet, da keine Funktion wie bei den reellen Zahlen vorliegt.

Definition 1. Sei G eine endliche zyklische Gruppe der Ordnung p. Sei g ein Generator in G und $y \in G$. **Der diskrete Logarithmus von y zur Basis g**, kurz $\log_g y$, ist die kleinste natürliche Zahl x, mit $0 \leq x \leq p-2$, so dass $y = g^x$.

Die Parametereinschränkungen für p und g werden in Abschnitt 3 näher erläutert.

Abb. 1 verbildlicht die Auswirkung der Beschränkung auf den Restklassenring modulo p.

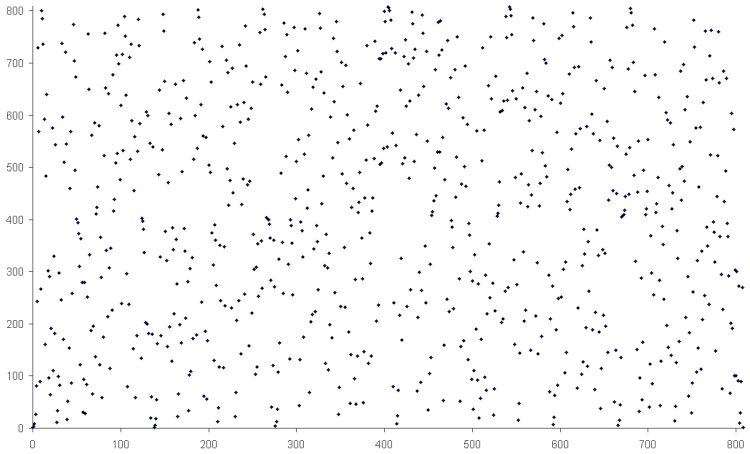

Abbildung 1: Verteilung des diskreten Logarithmus für $p = 809$ und $g = 3$

Durch diese „Willkürlichkeit“ in der Verteilung der diskreten Logarithmen stellt sich das Berechnungsproblem des diskreten Logarithmus als „schwer“ dar.

2 Diffie-Hellman-Schlüsselaustausch

Als erstes Beispiel für die Anwendung des diskreten Logarithmus in kryptographischen Systemen wird der Diffie-Hellman-Schlüsselaustausch betrachtet. Hierbei ist die Zielstellung, dass ein symmetrischer Schlüssel sicher an 2 Parteien verteilt wird, ohne dass diese sich persönlich dafür treffen müssen. Das heißt, dass der Schlüssel über einen öffentlichen, ungesicherten Kanal übertragen werden muss. Mit Hilfe des diskreten Logarithmus kann allerdings gewährleistet werden, dass außer diesen beiden Parteien niemand den Schlüssel erhält. Wie dies funktioniert, soll Abbildung 2 veranschaulichen.

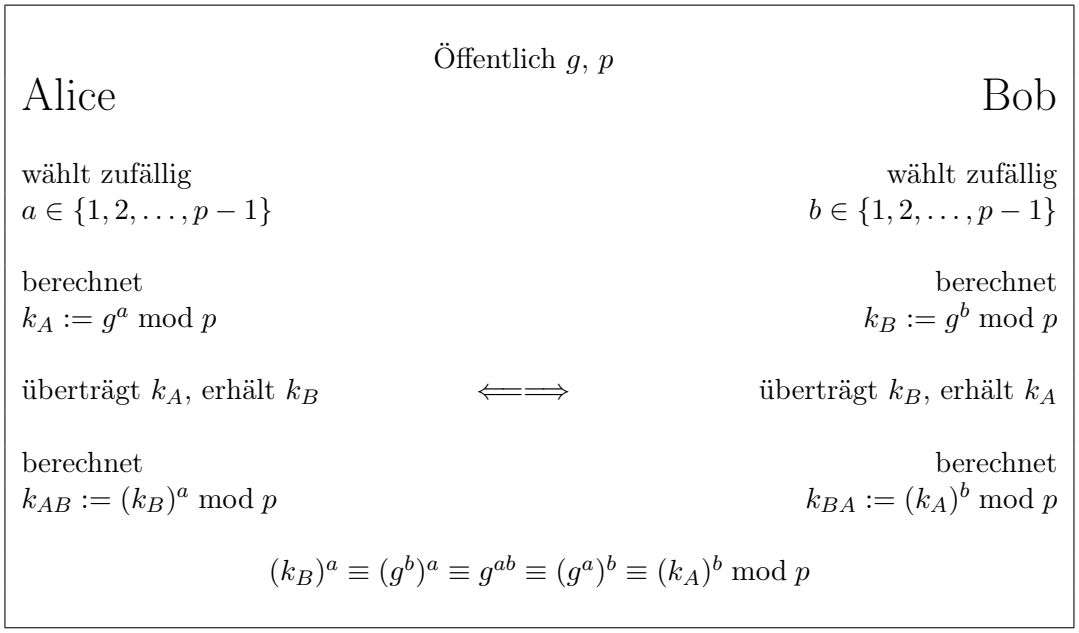

Abbildung 2: Diffie-Hellman-Schlüsselaustausch-Protokoll

Beide Parteien wählen zufällig einen Wert (a bzw. b) und erzeugen einen Teilschlüssel (k_A bzw. k_B) der über einen ungesicherten Kanal übertragen wird. Jede Partei berechnet nun aus dem erhaltenen Teilschlüssel und dem eigenen gewählten Wert den eigentlichen Schlüssel $k_{AB} \equiv_p (k_B)^a \equiv_p (k_A)^b$. Da die beiden gewählten Werte (a, b) nur jeweils den Erzeugern bekannt sind, sollte niemand, der die Übertragung von k_A und k_B abhört, in der Lage sein k_{AB} zu berechnen, solange das Berechnen der diskreten Logarithmen von k_A bzw. k_B schwer ist.

Ist ein Angreifer in der Lage diskrete Logarithmen zu berechnen, dann kann er ebenso den Diffie-Hellman-Schlüsselaustausch unterminieren. Ist ein Angreifer in der Lage aus den öffentlichen Parametern und den abgehörten Teilschlüsseln (k_A, k_B) den Schlüssel k_{AB} zu berechnen (Diffie-Hellman-Problem genannt), so ist allerdings bisher nicht bewiesen, dass er dadurch ebenso diskrete Logarithmen berechnen kann. Damit ist der Diffie-Hellman-Schlüsselaustausch gegenüber passiven Angriffen höchstens so sicher wie das Berechnen des diskreten Logarithmus schwer ist.

Das **Diffie-Hellman-Problem** soll nun etwas genauer betrachtet werden. Gegeben seien die öffentlichen Parameter g, p und die Teilschlüssel $k_A = g^a \bmod p$, $k_B = g^b \bmod p$. Gesucht ist der Schlüssel $k_{AB} = g^{a \cdot b} \bmod p$. Das Diffie-Hellman-Problem (bzw. Diffie-Hellman-Annahme) besagt, dass das Berechnen des geheimen Schlüssels k_{AB} mit den gegebenen Werten „schwer“ ist. Bisher ist nicht bekannt, dass das Diffie-Hellman-Problem ohne Berechnung des diskreten Logarithmus gelöst werden kann.

Dieses Protokoll hat in dieser Form allerdings noch eine wichtige Schwachstelle. Denn auch ohne das Wissen über die geheimen Werte und den symmetrischen Schlüssel ist ein Angrei-

fer durch den **Man-in-the-Middle-Angriff** [1] in der Lage den Inhalt einer Kommunikation, basierend auf einem Diffie-Hellman-Schlüssel, abzuhören. Hierzu schaltet sich der Angreifer bereits während des Schlüsselaustausches zwischen beide Parteien und gibt sich gegenüber jedem der beiden als der andere aus. Somit kann er einen symmetrischen Schlüssel zu Alice aufbauen und einen anderen Schlüssel zu Bob. Während der Kommunikation fängt er die Datenpakete ab, entschlüsselt sie mit dem entsprechenden Schlüssel und verschlüsselt sie anschließend für die Weiterleitung an den echten Empfänger. Somit ist er in der Lage die Kommunikationsinhalte sowohl abzuhören als auch zu manipulieren.

Alice	Mallory	Bob
erzeugt Schlüssel k_{AB} $\{= k_{AM}\}$	erzeugt Schlüssel k_{AM}, k_{BM}	erzeugt Schlüssel $\{k_{BM} =\}$ k_{BA}
verschlüsselt $c_1 = f_{k_{AM}}(m)$	ent-/verschlüsselt $m = f^{-1}_{k_{AM}}(c_1)$ $c_2 = f_{k_{BM}}(m)$	entschlüsselt $m = f^{-1}_{k_{BM}}(c_2)$

Abbildung 3: Man-in-the-Middle-Angriff beim Diffie-Hellman-Schlüsselaustausch

AUFGABE 1

a) Berechnen Sie beispielhaft den symmetrischen Schlüssel auf Basis des Diffie-Hellman-Schlüsselaustausches für $p = 809$, $g = 3$, $a = 5$ und $b = 17$. Berechnen Sie hierbei sowohl k_{AB} als auch k_{BA}.

b) Mit welchen (evtl. kryptographischen) Möglichkeiten kann der Man-in-the-Middle-Angriff wirkungslos gemacht werden?

c) Zeigen Sie, dass wenn es einen effizienten Algorithmus zum Berechnen von diskreten Logarithmen gibt, das Diffie-Hellman-Problem gelöst werden kann.

3 Wahl von g und p

Damit die diskrete Logarithmus-Annahme zutrifft, sind gewisse Einschränkungen in der Wahl der Parameter zu beachten.

Sei im Folgenden p eine Primzahl, wobei $p-1$ mindestens einen sehr großen Faktor hat. Damit ist $\mathbb{Z}_p$ ein endlicher Körper und $\mathbb{Z}_p^* = \mathbb{Z}_p \backslash \{0\}$, unter Einbeziehung der Multiplikation, eine Gruppe mit $\left|\mathbb{Z}_p^*\right| = \phi(p) = p - 1$ Elementen. Somit hat diese Gruppe eine Ordnung von $p - 1$.

[1] auch Janusangriff genannt

Nach dem kleinen Fermatschen Satz gilt also für diese Gruppe die zyklische Eigenschaft:

$$g^{p-1} \equiv 1 (\text{mod } p) \tag{3}$$

Für den Diffie-Hellman-Schlüsselaustausch genügt eine beliebige zyklische Gruppe mit ausreichend Elementen. Um eine möglichst hohe Sicherheit zu bieten, sollte die Basis g eine Gruppe mit ausreichend hoher Ordnung erzeugen. Das heißt, durch Potenzierung von g sollen möglichst viele Elemente der Gruppe erzeugt werden können. Somit darf mathematisch betrachtet die Basis g beliebig gewählt werden, für den praktischen Gebrauch sollte jedoch als Basis ein erzeugendes Element (im Folgenden **Generator** genannt), welches die gesamte Gruppe erzeugt, gewählt werden.

$$< g > = \{g^0, g^1, g^2, \ldots, g^{p-1}\} = \mathbb{Z}_p^* \tag{4}$$

Bedingt durch die zyklische Eigenschaft dieser Gruppe muss mindestens ein Generator g in dieser Gruppe existieren. Es gibt allerdings bedeutend mehr als nur einen Generator, denn alle i-ten Potenzen von einem Generator g mit $ggT(i, p-1) = 1$, sind ebenfalls Generatoren. Das heißt also, dass es genau $\phi(p-1)$ Generatoren gibt. Bewiesen wird dies z.B. in [BNS05, S. 136].

Elemente, die keine Generatoren sind erzeugen, durch Potenzierung Untergruppen von $\mathbb{Z}_p^*$, deren Ordnung stets Teiler der Ordnung von $\mathbb{Z}_p^*$ sind. Diese Tatsache wird genutzt um Generatoren zu finden. Wie dies von statten geht, ist im Anhang A.1 beschrieben.

Warum werden Generatoren benötigt?
Wird als Basis eine Zahl genutzt, die kein Generator ist, so werden die Ergebnisse von Potenzierungen stets in der erzeugten Untergruppe liegen. Da jedoch die gesamte Gruppe ausgenutzt werden soll, sollten Generatoren als Basis verwendet werden.

Sei also von nun an p prim und g ein Generator in $\mathbb{Z}_p^*$.

4 ElGamal

4.1 Einführung

Betrachtet wird nun ein Verschlüsselungs- und ein Signatur-System auf Basis des diskreten Logarithmus. Taher ElGamal publizierte 1984 [ElG85] diese Verfahren, wobei sich der asymmetrische Verschlüsselungsalgorithmus aus dem Diffie-Hellman-Schlüsselaustausch ableitet.

4.2 Verschlüsselungsalgorithmus

4.2.1 Schlüsselerzeugung

Der Empfänger verschlüsselter Nachrichten (hier Bob) erzeugt zuerst einen geheimen und einen öffentlichen Schlüssel. Der gewählte, geheime Schlüssel $k_d \in \mathbb{Z}_{p-1}$ [2], entspricht beim

[2] Dies ist der Bereich, der eine Verschlüsselung möglich macht, allerdings sollten u.A. die Werte 0 und 1 nicht verwendet werden.

Diffie-Hellman-Schlüsselaustausch dem a bzw. b. Der öffentliche Schlüssel besteht neben der Primzahl p und der Basis g ebenso aus der Potenz:

$$k_c := g^{k_d} \bmod p \tag{5}$$

4.2.2 Verschlüsselung

Um eine Nachricht $m \in \mathbb{Z}_p$ zu verschlüsseln wählt die sendende Instanz (hier Alice) eine Zufallszahl $r \in \mathbb{Z}_p$. Wobei $r = 0$ und $r = 1$ zum Schutz der Nachricht nicht verwendet werden sollten. Nun berechnet Alice mit Hilfe des öffentlichen Schlüssels (k_c, g, p) von Bob den Schlüssel

$$K = {k_c}^r \bmod p \tag{6}$$

Diesen Schlüssel K verwendet Alice nun als symmetrischen Schlüssel für die Verschlüsselungsfunktion f.

$$c_2 := f_K(m) \tag{7}$$

Betrachtet wird hier nur die üblichste Funktion $f_K(m) = m \cdot K \bmod p$. Andere invertierbare Funktionen sind ebenso möglich, z.B. Addition $\bmod p$.

Um Bob die gewählte Zufallszahl r mitzuteilen, wird diese als $c_1 := g^r \bmod p$ übertragen. Damit ergibt sich als verschlüsselte Nachricht ein Tupel aus der maskierten Zufallszahl r und dem eigentlichen Schlüsseltext c_2, die Multiplikation aus dem symmetrischen Schlüssel K und der Nachricht m.

$$(c_1 := g^r \bmod p \ , \ c_2 := m \cdot K \bmod p) \tag{8}$$

Für eine Nachricht $m \leq p - 1$ ist somit der Schlüsseltext doppelt so lang wie der Klartext. Doch muss für jeden Klartextblock ein unterschiedlicher Zufallswert r verwendet werden. Denn ansonsten gilt $m_1 \cdot m_2^{-1} = c_{2,1} \cdot c_{2,2}^{-1} \bmod p$, wobei durch Kenntnis eines Klartextblockes jeder weitere Klartextblock berechnet werden könnte.

4.2.3 Entschlüsselung

Bob erhält nun das Tupel (c_1, c_2). Damit er die Nachricht entschlüsseln kann, benötigt er Informationen über die Zufallszahl von Alice, welche er aus $c_1 := g^r \bmod p$ erhält. Durch die Kenntnis des Empfängers über seinen geheimen Schlüssel k_d ist er in der Lage daraus ${c_1}^{k_d} \equiv_p (g^r)^{k_d} \equiv_p (g^{k_d})^r \equiv_p {k_c}^r \equiv_p K$ zu berechnen. Nun kann er den Term $c_2 := m \cdot K \bmod p$ durch Multiplikation mit K^{-1} demaskieren und erhält den Klartext m.

$$m = {f_{k_A}}^{-1}(c_2) = c_2 \cdot K^{-1} = c_2 \cdot {c_1}^{-k_d} \bmod p \tag{9}$$

Zur Vereinfachung kann statt $-k_d$ auch der positive Wert $p - 1 - k_d$ im Exponenten stehen, da $x^{p-1} \equiv 1 \bmod p$.

Ohne die Kenntnis von k_d ist ein Angreifer gezwungen ${k_c}^r = K$ nur aus $c_1 = g^r$ zu errechnen, welches dem Diffie-Hellman-Problem gleich kommt.

4.2.4 Bezug zum Diffie-Hellman-Schlüsselaustausch

Abb. 4 veranschaulicht den Zusammenhang zum Diffie-Hellman-Schlüsselaustausch.

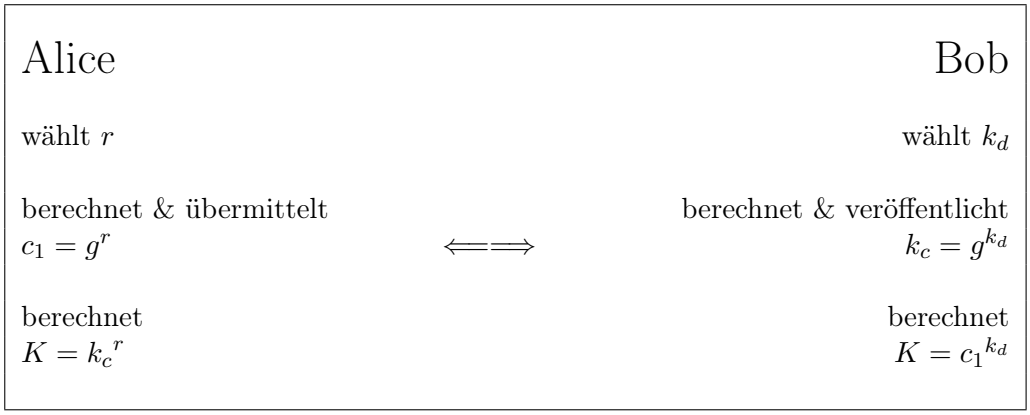

Abbildung 4: Analogie von ElGamal und Diffie-Hellman-Schlüsselaustausch

4.2.5 Eigenschaften

Wie sicherlich bereits aufgefallen ist, ist die ElGamal-Verschlüsselung durch die Wahl von r **indeterministisch**. Somit führt die Verschlüsselung von 2 gleichen Klartexten zu unterschiedlichen Schlüsseltexten.

Eine weitere Eigenschaft ist die Möglichkeit der **Re-Encryption**. Hierbei kann aus einem Schlüsseltext ein weiterer Schlüsseltext zum selben Klartext erzeugt werden, ohne dass dafür dieser Klartext bekannt sein muss. Hierzu wird eine weitere Zufallszahl s gewählt und sowohl $g^s \bmod p$ als auch ${k_c}^s \bmod p$ berechnet.

$$\begin{aligned} &(c_1 \cdot g^s \bmod p\ , \quad && c_2 \cdot\ {k_c}^s \bmod p) \\ &(g^r \cdot g^s \bmod p\ , \quad && m \cdot\ {k_c}^r \cdot\ {k_c}^s \bmod p) \\ &(g^{r+s} \bmod p\ , \quad && m \cdot\ {k_c}^{r+s} \bmod p) \end{aligned} \tag{10}$$

Ebenso wichtig ist die Eigenschaft des **multiplikativen Homomorphismus** von ElGamal-Schlüsseltexten. Werden die Elemente (c_1, c_2) zweier ElGamal-Schlüsseltext-Tupel paarweise multipliziert so ergibt die Entschlüsselung dieses Tupels das Produkt der beiden ursprünglichen Klartexte.

Diese Eigenschaften können u.U. für Angriffe auf spezielle Protokolle, die ElGamal-Verschlüsselung verwenden, angewandt werden. Aber sie ermöglichen auch die Entwicklung von weiteren kryptographischen Protokollen, wie z.B. dem Idemix[3].

[3] Idemix (oder auch „Identity Mixer“) ist ein Open-Source-Identitätsmanager, der es ermöglicht, Online-Aktivitäten unter Pseudonymen durchzuführen. Dabei werden ausschließlich wirklich notwendige Informationen preisgegeben, wie z.B. die Erlaubnis ein Fahrzeug der Klasse B zu fahren. Mehr dazu unter `http://www-05.ibm.com/ch/think/archiv/22007/inhalt5.shtml`. Zum Protokoll [CL01] und zur Implementierung [CH02].

4.2.6 Sicherheit

Offensichtlich ist ein Angreifer durch die Möglichkeit diskrete Logarithmen zu berechnen auch in der Lage die ElGamal-Verschlüsselung zu brechen und sogar den geheimen Schlüssel zu kompromittieren. Bisher ist allerdings nicht bewiesen, dass durch Brechen der ElGamal-Verschlüsselung diskrete Logarithmen berechnet werden können. Daraus ergibt sich, dass die ElGamal-Verschlüsselung höchstens so schwer ist wie die Berechnung von diskreten Logarithmen.

AUFGABE 2

a) Zeigen Sie, dass durch Entschlüsselung eines ElGamal-Schlüsseltextes der Klartext m erhalten wird. Lösen Sie hierzu die Gleichung (9) auf.

b) Zeigen Sie, wie durch die Entschlüsselung des Produktes zweier ElGamal-Schlüsseltexte das Produkt der beiden Klartexte entsteht.

c) Als symmetrische Operation in Formel (7) wurde die Multiplikation genutzt. Wird die Sicherheit des Verfahrens beeinträchtigt, wenn stattdessen eine XOR-Verknüpfung verwendet wird?

d) Anstatt alle Blöcke einer Nachricht jeweils mit unterschiedlichen Zufallszahlen zu verschlüsseln, kann ein effektiveres Verfahren eingesetzt werden. Welches?

e) Implementieren Sie den Ver- und Entschlüsselungsteil des ElGamal-Konzelationssystems in den Klassen `ElGamal_Encrypter` und `ElGamal_Decrypter`.
Probieren Sie anschließend anstelle der Multiplikation als symmetrische Operation ebenso die Addition mod p.

4.3 Signaturalgorithmus

Mit der Veröffentlichung der ElGamal-Verschlüsselung wurde auch ein digitales Signatursystem vorgestellt, welches allerdings, abgesehen von der Parameterwahl, keine besondere Ähnlichkeit zum Verschlüsselungssystem hat.

4.3.1 Schlüsselerzeugung

Der Signierer einer Nachricht wählt als öffentliche Parameter die Primzahl p, einen Generator g und als geheimen Signierschlüssel $k_s \in \mathbb{Z}_{p-1}$, woraus er den öffentlichen Testschlüssel k_t berechnet:

$$k_t = g^{k_s} \bmod p \tag{11}$$

4.3.2 Signatur erstellen

Bei der Erstellung einer Signatur zu einer Nachricht m mit $0 \leq m \leq p-1$ wählt der Signierer zuerst eine Zufallszahl r in $\mathbb{Z}^*_{p-1}$ mit $ggT(r, p-1) = 1$ und berechnet $s_1 = g^r \bmod p$.
Gesucht ist nun eine Signatur, die mit Hilfe des öffentlichen Schlüssels und der öffentlichen Parameter verifiziert werden kann, ein Angreifer mit diesen Werten hingegen keine Signatur

erstellen oder fälschen kann.
Die Signatur besteht aus $s = (s_1, s_2)$, so dass die folgende Gleichung wahr ist:

$$g^m \equiv k_t^{s_1} \cdot {s_1}^{s_2} (\text{mod } p) \tag{12}$$

Dafür berechnet der Signierer die folgende Kongruenz:

$$s_2 = (m - s_1 \cdot k_s) \cdot r^{-1} \bmod (p-1) \tag{13}$$

Daraus ergibt sich nun folgende Signatur:

$$s = (s_1,\ s_2) = (g^r \bmod p,\ (m - s_1 \cdot k_s) \cdot r^{-1} \bmod (p-1)) \tag{14}$$

Aus Sicherheitsgründen[4] und da üblicherweise Nachrichten nicht im Bereich $0 \leq m \leq p-1$ liegen, wird ausschließlich der Hashwert der Nachricht $h(m)$ signiert, wobei die verwendete Hashfunktion eine Einwegfunktion ist.

4.3.3 Signatur prüfen

Der Empfänger der Nachricht (m, s_1, s_2) prüft, ob die Nachricht unverändert ist, indem er die folgende Gleichung testet:

$$g^m \equiv k_t^{s_1} \cdot {s_1}^{s_2} (\text{mod} p) \tag{15}$$

Stimmt diese Gleichung so ist die Nachricht korrekt signiert, denn es gilt:

$$k_t^{s_1} \cdot {s_1}^{s_2} \equiv g^{k_s s_1 + s_2 r} \equiv g^{k_s s_1 + m - k_s s_1} \equiv g^m (\text{mod } p) \tag{16}$$

Anmerkung: Zur Berechnung von s_2 wird $\bmod (p-1)$ verwendet, da s_2 zum Testen im Exponenten steht, wobei gilt: $a^b \bmod p \equiv a^{b \bmod (p-1)}$

Im Gegensatz zur Signatur mittels RSA lässt sich die Nachricht selbst nicht aus der Signatur berechnen (message recovery), zudem expandiert eine signierte Nachricht um die doppelte Schlüssellänge $|p|$ (m, s_1, s_2).

4.3.4 Sicherheit

Ebenso wie das Verschlüsselungsverfahren ist für die digitale Signatur mittels ElGamal nur bewiesen, dass sie höchstens so sicher ist, wie das Berechnen von diskreten Logarithmen schwer ist. Dennoch sei hier gezeigt, dass, wenn ein Angreifer eine Signatur fälschen möchte, er sowohl s_1 als auch s_2 passend wählen muss, damit gilt:

$$g^m \equiv k_t^{s_1} \cdot s_1^{s_2} (\text{mod} p) \tag{17}$$

Wählt er hierbei zuerst s_1, so muss er einen diskreten Logarithmus berechnen um s_2 zu berechnen. Wird s_2 zuerst gewählt so muss eine komplizierte Gleichung gelöst werden, die wiederum einen diskreten Logarithmus enthält. Wählt er allerdings s_1 und s_2 simultan, so müsste er zum Bestimmen der passenden Nachricht m wiederum einen diskreten Logarithmus

[4] existenzielles Brechen beschrieben in [ElG85]

berechnen. Dennoch ist nicht bewiesen, dass es keine bessere Herangehensweise zum Fälschen der Signatur von gewählten Nachrichten gibt.

Ähnlich wie bei der Verschlüsselung darf der **Zufallswert** r **nie mehrmals** zur Signierung verwendet werden. Angenommen einem Angreifer sind 2 Paare von Nachrichten und deren Signaturen bekannt, so kann er für die Gleichung (13) ein lineares Gleichungssystem aufstellen. Da allerdings der Wert r unterschiedlich sein sollte ist dieses Gleichungssystem unterbesetzt. Bei konstantem r ließe sich dagegen aus dem Gleichungssystem der geheime Schlüssel k_s berechnen.

Es ist allerdings möglich für einen Angreifer, durch Wissen über eine Nachricht und deren Signatur weitere Signaturen und passende Nachrichten zu generieren. Doch ist es nicht möglich, auf diesem Wege eine zu einer bestimmten Nachricht passende Signatur zu erstellen und deshalb wird dies nicht als Angriff gegen das System angesehen.

4.3.5 Varianten (DSS)

Es gibt verschiedene Varianten der ElGamal-Signatur, die entweder eine andere Kongruenz aufstellen, statt der gesamten Gruppe nur eine Untergruppe von $\mathbb{Z}_p^*$ nutzen, oder sogar auf anderen Gruppen wie z.B. den elliptischen Kurven operieren.

Die wohl wichtigste Variante stellt der **DSA** (Digital Signature Algorithm) dar. Er wurde 1991 vom U.S. National Institute of Standards and Technology (NIST) vorgeschlagen und später als **DSS** (Digital Signature Standard) standardisiert.

Im Folgenden werden die Grundzüge vom DSS vorgestellt. Als öffentliche Parameter wählt der Signierer eine Primzahl p mit etwa 512 bis 1024 Bit, eine weitere Primzahl q mit 160 Bit welche Teiler von $p-1$ ist und einen Generator $g \in \mathbb{Z}_p^*$ welcher eine Gruppe erzeugt die die Ordnung q besitzt. Anschließend wählt er zufällig seinen geheimen Signierschlüssel $k_s \in \mathbb{Z}_q^*$ und berechnet den öffentlichen Testschlüssel $k_t = g^{k_s} \bmod p$. Um nun eine Nachricht zu signieren, führt er folgende Schritte aus:

- wähle Zufallszahl $r \in \mathbb{Z}_q^*$ und berechne $s_1 = (g^r \bmod p) \bmod q$
- bestimme $r^{-1} \bmod q$
- berechne $s_2 = r^{-1} \cdot (h(m) + k_s \cdot s_1) \bmod q$, wobei h ein Einweg-Hashfunktion ist
- Die Signatur besteht aus $s = (s_1, s_2)$

Für die Verifikation der Signatur werden die folgenden Schritte ausgeführt.

- berechne $u_1 = s_2^{-1} \cdot h(m) \bmod q$ und $u_2 = s_2^{-1} \cdot s_1 \bmod q$
- prüfe ob $s_1 = (g^{u_1} \cdot k_t^{u_2} \bmod p) \bmod q$

Stimmt die letzte Gleichung, so ist die Signatur gültig, denn:

$$\begin{aligned} s_1 &= g^{u_1} \cdot k_t^{u_2} \bmod q \\ &= g^{s_2^{-1} \cdot h(m)} \cdot g^{k_s \cdot s_2^{-1} \cdot s_1} \bmod q \\ &= g^{s_2^{-1}(h(m)+k_s \cdot s_1)} \bmod q \\ &= g^{r(h(m)+k_s \cdot s_1)^{-1} \cdot (h(m)+k_s \cdot s_1)} \bmod q \\ s_1 &= g^r \bmod q \end{aligned} \tag{18}$$

AUFGABE 3

a) Zeigen Sie, wie ausgehend von der Formel (12) der Wert für s_2 berechnet werden kann. Vergleichen Sie Ihre Formel mit der Formel (13). Tipp: Ersetzen Sie die Variablen in Formel (12) durch die Variablen, die der Signierer zur Verfügung hat.

5 Verfahren zum Berechnen des diskreten Logarithmus

Im Folgenden werden ein paar Verfahren vorgestellt die es erlauben den diskreten Logarithmus effizienter zu berechnen als durch eine vollständige Suche, bei der die Basis bis zu gesuchten Potenz potenziert wird.

Gegeben sei eine Primzahl p und $g, y \in \mathbb{Z}_p^*$.
Gesucht ist $\log_g y =: x$, ergeben aus $y := g^x \bmod p$.

5.1 Der Babystep-Giantstep-Algorithmus

Der Babystep-Giantstep-Algorithmus (auch Shanks' Algorithmus) unterteilt die vollständige Suche in 2 Abschnitte, die die Laufzeit deutlich verringern, dafür allerdings den Speicherbedarf stark erhöhen.

Als Erstes wird der konstante Wert t mit $t = \left\lceil \sqrt{|G|} \right\rceil = \lceil \sqrt{p-1} \rceil$ berechnet. Der diskrete Logarithmus von y lässt sich so zerlegen in $x = u \cdot t + v$ mit $0 \leq u \leq t$ und $0 \leq v < t$, wobei sich die Ausgangsformel $y := g^x \bmod p$ umformen lässt in:

$$y \equiv g^x \equiv g^{u \cdot t + v} (\mathrm{mod} p) \Leftrightarrow y \cdot g^{-v} = g^{u \cdot t} \bmod p \tag{19}$$

Gesucht sind nun also u und v, so dass $y \cdot g^{-v} = g^{u \cdot t} \bmod p$ gilt. Dafür werden 2 Listen berechnet:

$$\begin{aligned} \textbf{Babystep-Liste:} \quad & B = \{ \ (\ i,\ y \cdot g^{-i} \bmod p\ ) \quad | \quad 0 \leq i < t \ \} \\ \textbf{Giantstep-Liste:} \quad & G = \{ \ (\ j,\ (g^t)^j \bmod p\ ) \quad | \quad 0 \leq j \leq t \ \} \end{aligned} \tag{20}$$

Hierbei durchläuft der Zähler i alle möglichen Werte für v und der Zähler j alle Werte für u. Berechnet wird zuerst die Babystep-Liste und anschließend wird die Giantstep-Liste rekursiv berechnet, wobei nach jedem Schritt das Ergebnis mit den Elementen in der Babystep-Liste verglichen wird. Gibt es eine Werteübereinstimmung in den Listen, so ergeben die Indizes der

betreffenden Listenelemente die Werte für u und v. Damit kann $x = u \cdot t + v \bmod (p-1)$ berechnet werden.

Wird der Suchaufwand beim Vergleichen der Listenelemente vernachlässigt, so ergibt sich eine **Komplexität von** $O(\sqrt{p-1})$. Dies ist zwar eine erhebliche Verbesserung gegenüber der vollständigen Suche mit einer Komplexität von $O(p-1)$, dennoch wäre ein polynomieller Aufwand in $\log p$ wünschenswert, denn dieser Algorithmus ist ab $|G| > 2^{160}$ nicht mehr praktikabel einsetzbar, siehe [Buc04].

Ein Vorteil dieses Verfahrens gegenüber anderen ist die **generische** Eigenschaft, die es erlaubt, es auf jede beliebige Gruppe anzuwenden.

AUFGABE 4

a) Implementieren Sie den Babystep-Giantstep-Algorithmus in der Klasse `BsGs_Attack`. Beachten Sie hierbei ebenfalls den Anhang A.2.

b) Auf welche Engpässe stoßen Sie bei größeren Schlüssellängen ? Vergleichen Sie die Zeitaufwände der Algorithmus-Abschnitte (Nutzen Sie das `benchmark`-Objekt). Welche Verbesserungen würden Sie vornehmen um die Engpässe zu verringern? Implementieren Sie diese. Sie sollten mit Ihrem optimierten Algorithmus bei einem 36-Bit-Schlüssel im Schnitt zwischen 1200 und 2500ms benötigen.

5.2 Weitere Verfahren

Index-Calculus-Algorithmus

Der wohl effizienteste Algorithmus zum Berechnen von diskreten Logarithmen ist der Index-Calculus-Algorithmus. Dieser lässt sich allerdings nur auf zyklische Gruppe und endlichen Körpern über $GF(p^a)$ anwenden. Besonders effizient ist er, wenn p relativ „klein“ ist.

Im Folgenden wird die Vorgehensweise für eine zyklische Gruppe G kurz erläutert. Als Grundlage dient eine relativ kleine Untergruppe S von G, die möglichst viele Elemente aus G als Produkt von Elementen aus S darstellen kann. Zu jedem dieser Elemente p_i aus S wird der diskrete Logarithmus zur Basis g berechnet und in einer Datenbank gehalten. Dieser Schritt kann unabhängig vom gegebenen Wert y durchgeführt werden. Der Hauptschritt der Berechnung besteht grob aus folgenden Schritten:

- wähle einen Wert $k \in \mathbb{Z}_p$ und berechne $y \cdot g^k$
- schreibe $y \cdot g^k$ als Produkt von Elementen p_i aus S.

$$y \cdot g^k = \prod_{i=1}^{t} p_i^{d_i} \bmod p \quad , \; d_i \geq 0, \; p_i \in S, \; t \leq |S| \tag{21}$$

  Gelingt dies nicht, wiederhole den ersten Schritt

- Ziehe den Logarithmus von beiden Seiten:

$$\begin{aligned} \log_g y + k &= \sum_{i=1}^{t} d_i \log_g p_i \bmod p \\ x = \log_g y &= \sum_{i=1}^{t} d_i \log_g p_i - k \bmod p \end{aligned} \tag{22}$$

Pollard's Rho-Algorithmus

Pollard's Rho-Algorithmus hat eine ähnliche Laufzeit wie der Babystep-Giantstep-Algorithmus, allerdings benötigt er deutlich weniger Speicher.

Als erstes wird die Gruppe G in 3 disjunkte Untergruppen S_1, S_2, S_3 aufgeteilt. Dies kann beliebig geschehen, doch ein paar Bedingungen müssen eingehalten werden, z.B. $1 \notin S_2$. Als nächster Schritt wird eine Reihe $x_0, x_1, x_2, \ldots$ an Werten angelegt ($x_0 = 1$):

$$x_{i+1} = \begin{cases} y \cdot x_i, & \text{wenn } x_i \in S_1 \\ x_i^2, & \text{wenn } x_i \in S_2 \\ g \cdot x_i, & \text{wenn } x_i \in S_3 \end{cases} \tag{23}$$

Für diese Reihen werden parallel 2 weitere Reihen a_i und b_i berechnet, so dass $x_i = g^{a_i} y^{b_i} \bmod p$ gilt. Hier ist $a_0 = 0$ und $b_0 = 0$:

$$a_{i+1} = \begin{cases} a_i, & \text{wenn } x_i \in S_1 \\ 2a_i \bmod p, & \text{wenn } x_i \in S_2 \\ a_i + 1 \bmod p, & \text{wenn } x_i \in S_3 \end{cases} \tag{24}$$

$$b_{i+1} = \begin{cases} b_i + 1 \bmod p, & \text{wenn } x_i \in S_1 \\ 2b_i \bmod p, & \text{wenn } x_i \in S_2 \\ b_i, & \text{wenn } x_i \in S_3 \end{cases} \tag{25}$$

Nun werden 2 gleiche Gruppenelemente gesucht, $x_s = x_t$. Daraus ergibt sich $g^{a_s} y^{b_s} = g^{a_t} y^{b_t} \Leftrightarrow y^{b_s - b_t} = g^{a_t - a_s} \pmod p$. Da $y := g^x \bmod p$, werden beide Seiten logarithmiert und ergeben:

$$(b_s - b_t) \cdot \log_g y = (a_t - a_s) \bmod (p-1) \tag{26}$$

Für den Fall, dass $b_s \neq b_t$, kann auf diese Weise der diskrete Logarithmus berechnet werden.

Pohlig-Hellman-Algorithmus

Der Pohlig-Hellman-Algorithmus, eine Variante des Babystep-Giantstep-Algorithmus, lässt sich nur auf Gruppen mit hauptsächlich „kleinen" Teilern von $p-1$ anwenden. Dabei wird versucht das Problem von der Gruppe G auf eine zyklische Gruppe G_{p^k} zu reduzieren. Der Aufwand wird dabei von $O(\sqrt{p-1})$ auf $O(\sqrt{p_b})$ herabgesetzt, wobei p_b der größte Primfaktor von $p-1$ sei. Näheres hierzu in [PH78].

5.3 Schlussfolgerung

Es wurden 4 Verfahren zum Berechnen von diskreten Logarithmen vorgestellt. Doch kann keiner von ihnen eine polynomale Laufzeit aufweisen, wodurch die diskrete Logarithmus-Annahme erfüllt bleibt.

Dennoch sei erwähnt, dass wenn ein Algorithmus existiert, der zu einer bestimmten Basis g den diskreten Logarithmus effizient berechnet, dieser Algorithmus dazu verwenden werden kann diskrete Logarithmen zu beliebigen Generatoren zu berechnen [MOV96]:

Seien g und h Generatoren in $\mathbb{Z}_p^*$, $y \in \mathbb{Z}_p^*$ und sei $a = \log_g y$, $b = \log_h y$ und $c = \log_g h$. Dann gilt $g^a = y = h^b = (g^c)^b$.
Daraus ergibt sich $a = b \cdot c \bmod p$ und

$$\begin{aligned} b &= a \cdot c^{-1} \bmod p \\ \log_h y &= (\log_g y)(\log_g h)^{-1} \bmod p \end{aligned} \tag{27}$$

Literatur

[BNS05] Beutelspacher, Albrecht ; Neumann, Heike B. ; Schwarzpaul, Thomas: *Kryptographie in Theorie und Praxis.* Friedr. Vieweg und Sohn Verlag/GWV Fachverlag GmbH, 2005. – ISBN 3–528–03168–9

[Buc04] Buchmann, Johannes A.: *Introduction To Cryptography.* Second. Springer-Verlag NY, LLC, 2004. – ISBN 0–387–20756–2

[CH02] Camenisch, Jan ; Herreweghen, Els V.: *Design and Implementation of the Idemix Anonymous Credential System.* Research Report RZ 3419, IBM Research Division. Also appeared in ACM Computer and Communication Security 2002, 2002

[CL01] Camenisch, Jan ; Lysyanskaya, Anna: *Efficient Non-transferable Anonymous Multi-show Credential System with Optional Anonymity Revocation.* In Birgit Pfitzmann, Advances in Cryptology, EUROCRYPT 2001 Volume 2045, 2001. – 93–118 S.

[ElG85] ElGamal, Taher: *A public-key cryptosystem and a signature scheme based on discrete logarithms.* IEEE Transactions on Information Theory 31, 1985. – 469–472 S.

[MOV96] Menezes, A. ; Oorschot, P. van ; Vanstone, S.: *Handbook of Applied Cryptography.* Fifth. CRC Press, 1996 `http://www.cacr.math.uwaterloo.ca/hac`. – ISBN 0–8493–8523–7

[PH78] Pohlig, Stephen ; Hellman, Martin: *An Improved Algorithm for Computing Logarithms over GF(p) and its Cryptographic Significance.* IEEE Transactions Information Theory 24, 1978. – 106–110 S.

A Anhang

A.1 Finden von Generatoren

Um einen Generator in $\mathbb{Z}_p^*$ zu finden, wird die Tatsache genutzt, dass Elemente, die keine Generatoren sind, Untergruppen erzeugen, deren Ordnung ein Teiler von $p-1$ ist. Deswegen werden zuerst eben diese Teiler ermittelt, indem $p-1$ faktorisiert wird.

$$p - 1 = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_k^{e_k} \tag{28}$$

Anschließend wird zufällig ein Element h aus $\mathbb{Z}_p^*$ gewählt und getestet, ob dessen Potenz mit jedem Primfaktor p_i ein Teiler von $|\langle h \rangle|$ ist.

$$\forall i \; h^{\frac{p-1}{p_i}} \neq 1 \bmod p \; , \; 1 \leq i \leq k \tag{29}$$

Ergibt $h^{\frac{p-1}{p_i}} \bmod p$ für alle Primfaktoren p_i nie 1, so ist die Ordnung von $\langle h \rangle$ folglich $p-1$. Dies bedeutet, dass mit h die gesamte Gruppe erzeugt werden kann und ein Generator $g = h$ gefunden wurde. Sollte jedoch $h^{\frac{p-1}{p_i}} \bmod p$ je 1 ergeben, so muss ein neues Element h gewählt und getestet werden.

Da das Finden der Primfaktoren für große Schlüssellängen sehr aufwändig ist, empfiehlt es sich für p eine „sichere Primzahl“ zu wählen. Das heißt, dass sich p aus $2 \cdot q + 1$ berechnet, wobei q prim und das ergebende p ebenfalls prim sein muss. Somit steht q bereits als einziger Primfaktor fest.

A.2 JVM Heap vergrößern

Bei der Implementierung des Babystep-Giantstep-Algorithmus kann bei größeren Schlüssellängen an die Grenzen des von der JVM zugewiesenen Heap gestoßen werden. Dies äußert sich durch die Fehlermeldung:

```
Exception in thread "AWT-EventQueue-0"
   java.lang.OutOfMemoryError: Java heap space
```

Um den Heap zu vergrößern sollte die Anwendung wie folgt gestartet werden:

```
java -XX:+AggressiveHeap gui.BsGs_gui
```

Sollte dies Probleme bereiten, gibt es die Alternative die Heap-Größe über die Optionen `-Xms` und `-Xmx` zu definieren:

```
java -Xms256m -Xmx512m gui.BsGs_gui
```

Für den seltenen Fall, dass der Garbage Collector mehr als 98% der Rechenzeit beansprucht (`java.lang.OutOfMemoryError: GC overhead limit exceeded`) kann mit der Option `-XX:-UseGCOverheadLimit` dieses Limit aufgehoben werden.

In Eclipse-Projekten können die JVM-Argumente folgendermaßen eingerichtet werden:
Run → Run ... → Launch Konfiguration auswählen oder neu erstellen → Arguments → VM Arguments → `-Xms256m -Xmx512m`