PSEUDOZUFÄLLIGE FUNKTIONEN

Konrad Hilse
hilse@informatik.hu-berlin.de

Seminar: Perlen der Theoretischen Informatik
Dozenten: Prof. Johannes Köbler, Olaf Beyersdorff
Wintersemester 2002/2003

21. Januar 2003

1. Motivation

Pseudozufallsgeneratoren ermöglichen es, eine große Anzahl von pseudozufälligen Werten aus wenigen Zufallsbits zu generieren. Es genügt, n zufällige, also gleichverteilt gewählte, Bits zu erzeugen und zu speichern, wenn $\text{poly}(n)$ pseudozufällige Bits benötigt werden. Für viele Anwendungen ist das ausreichend, wenn man z. B. das one-time-pad zum Verschlüsseln verwenden will, genügt es, für Klartexte der Länge $\text{poly}(n)$ einen Schlüssel der Länge n zu erzeugen und zu übertragen.

Da jeder effiziente Algorithmus nur polynominiell viele (Pseudo-)zufallswerte verwenden kann, könnte man annehmen, dass es für jede Anwendung ausreicht, wenn ein Pseudozufallsgenerator polynominiell viele Pseudozufallswerte zur Verfügung stellt. Dies gilt allerdings nur für Anwendungen, die diese Pseudozufallswerte in einer konstanten Reihenfolge verwenden. Es gibt jedoch auch Algorithmen, die in Abhängigkeit von der Eingabe bestimmte Zufallswerte benötigen, d. h. es ist vorher nicht bekannt, in welcher Reihenfolge die Zufallszahlen benötigt werden. Verwendet man Pseudozufallsgeneratoren für solche Algorithmen, müssen n Zufallsbits erzeugt und $\text{poly}(n)$ viele gespeichert werden. Verwendet man stattdessen pseudozufällige Funktionen, so genügt es, n Zufallsbits zu erzeugen und *auch nur diese n Bits zu speichern.*

2. Definition

Pseudozufällige Funktionen sind Funktionen, die kein effizienter Algorithmus von zufälligen Funktionen unterscheiden kann. Dabei ist es einem solchen Algorithmus erlaubt, Funktionswerte für beliebige Argumente zu erfragen, dies kann auch adaptiv geschehen, das heißt weitere Anfragen können auch in Abhängigkeit von bisher erhaltenen Antworten gestellt werden. Da der Algorithmus effizient sein soll, ist er in der Laufzeit polynominiell beschränkt, kann daher auch höchstens polynominiell viele Funktionswerte erfragen.

Genauer gesagt: Es ist für keinen effizienten Algorithmus möglich, zu unterscheiden, ob die Antworten von einer Funktion aus dem pseudozufälligen

Funktionsensemble oder dem gleichverteilten Funktionsensemble geliefert wurden. Im folgenden werden daher zuerst Ensembles von Funktionen definiert, um dann später pseudozufällige Funktionsensembles formal definieren zu können. Der Einfachheit halber werden hier nur längenerhaltende Funktionen (d. h. $|f(x)| = |x|$) betrachtet.

Definition 2.1. (Funktionsensemble):
Sei $l : \mathbb{N} \to \mathbb{N}$

Ein l**-bit Funktionsensemble** *ist eine Folge* $F = \{F_n\}_{n\in\mathbb{N}}$ *von Zufallsvariablen, wobei jede Zufallsvariable* F_n *Werte aus der Menge der Funktionen annimmt, die von* $l(n)$*-Bit langen Strings auf* $l(n)$*-Bit lange Strings abbilden, d. h. aus der Menge der Funktionen mit Definitions- und Wertebereich* $\{0,1\}^{l(n)}$*. Im* **gleichverteilten** l**-bit Funktionsensemble**, *bezeichnet mit* $H = \{H_n\}_{n\in\mathbb{N}}$, *ist* H_n *gleichverteilt über der Menge der Funktionen mit Definitions- und Wertebereich* $\{0,1\}^{l(n)}$.

Da es $(2^n)^{(2^n)}$ Funktionen gibt, die von n-bit-Strings auf n-bit-Strings abbilden, kann für $l(n) = n$ die Zufallsvariable H_n also $2^{n\cdot 2^n}$ Werte annehmen.

Beispiel 2.2. Sei $l(n) = n$ und $n = 2$. Dann kann H_2 $4^4 = 256$ Werte annehmen, es gibt 256 mögliche Funktionen, jeweils mit dem Definitions- und Wertebereich $\{00, 01, 10, 11\}$. Sei f eine Realisierung von H_2, d. h. f wurde unter Gleichverteilung aus diesen 256 Funktionen gewählt. Betrachtet man jetzt f an einem beliebigen Punkt des Definitionsbereiches, z. B. $f(10)$, so kann man über den Funktionswert keine Vorhersage treffen: Es gibt jeweils 64 Funktionen, für die $f(10) = 00, 01, 10$ *bzw.* 11 gilt. Kennt man schon andere Funktionswerte, z. B. $f(01) = 00$, so kann man trotzdem keine Vorhersage über $f(10)$ treffen, selbst wenn man alle anderen Funktionswerte kennt, z. B. sei $f(00) = f(01) = f(11) = 00$, sind für $f(10)$ immer noch alle Werte gleich wahrscheinlich: Es gibt 4 Funktionen mit $f(00) = f(01) = f(11) = 00$, jeweils eine, für die $f(10) = 00, 01, 10$ *bzw.* 11 gilt.

Diese Eigenschaft zufälliger Funktionen nennt man auch **Nichtvorhersagbarkeit**. Eine formale Definition der Nichtvorhersagbarkeit folgt in Definition 2.7, da dafür noch andere Definitionen erforderlich sind. Dann wird sich auch zeigen, dass pseudozufällige Funktionen ebenfalls nicht vorhersagbar sind.

Für eine formale Definition von pseudozufälligen Funktionen wird der Begriff **Orakelmaschine** verwendet. Dies ist eine modifizierte Turingmaschine, die zusätzlich eine Orakelfunktion befragen kann.

Definition 2.3. (Orakelmaschine)
Eine probabilistische Orakelmaschine ist eine probabilistische Turingmaschine mit einem zusätzlichen Band, Orakelband *genannt, und zwei besonderen Zuständen,* Orakelaufruf *und* Orakelantwort *genannt. Die Berechnung einer solchen Orakelmaschine* M *bei Eingabe* x *und Verwendung des Orakels* f :

$\{0,1\}^* \to \{0,1\}^*$ *wird durch folgende Zustandsüberführungsfunktion beschrieben:*

Für Konfigurationen mit einem Zustand verschieden von „Orakelaufruf" sind die möglichen Folgekonfigurationen wie üblich definiert. Sei γ eine Konfiguration, deren Zustand „Orakelaufruf" ist, und der Inhalt des Orakelbandes sei q. Dann ist die Folgekonfiguration die gleiche wie γ, nur der Zustand ist „Orakelantwort" und der Inhalt des Orakelbandes ist $f(q)$. q wird Anfrage *der Maschine M genannt, $f(q)$* Orakelantwort*. Im folgenden werden nur probabilistische, zeitlich polynominiell beschränkte Orakelmaschinen betrachtet und diese nur mit „Orakelmaschine" bezeichnet.*

Damit ist jeder effiziente Algorithmus, der Funktionswerte für beliebige Argumente erfragen kann, durch eine Orakelmaschine darstellbar (und natürlich auch umgekehrt), d. h. pseudozufällige Funktionen können mit Hilfe von Orakelmaschinen definiert werden.

Definition 2.4. (pseudozufällige Funktionsensembles)
Ein l-Bit-Funktionsensemble $F = \{F_n\}_{n\in\mathbb{N}}$ heißt **pseudozufällig**, *wenn für jede zeitlich polynominiell beschränkte, probabilistische Orakelmaschine M, jedes Polynom $p(\cdot)$ und alle genügend großen n gilt:*

$$\left|\text{Prob}\left[M^{F_n}(1^n) = 1\right] - \text{Prob}\left[M^{H_n}(1^n) = 1\right]\right| < \frac{1}{p(n)}$$

wobei $H = \{H_n\}_{n\in\mathbb{N}}$ das gleichverteilte l-bit-Funktionsensemble ist.

Um pseudozufällige Funktionen praktisch verwenden zu können, ist es notwendig, dass sie effizient berechnet werden können. Dazu müssen alle Funktionen in dem Funktionsensemble eine kurze (also nur polynominiell lange) Beschreibung haben, diese muss effizient auszuwählen und die zugehörige Funktion muss effizient zu berechnen sein.

Definition 2.5. (effizient berechenbare Funktionsensembles)
Ein l-Bit-Funktionsensemble $F = \{F_n\}_{n\in\mathbb{N}}$ heißt **effizient berechenbar**, *wenn folgende Bedingungen erfüllt sind:*

a) effizient indizierbar: *Es existiert ein probabilistischer, zeitlich polynominiell beschränkter Algorithmus I und eine Abbildung ϕ von Strings auf Funktionen, so dass $\phi(I(1^n))$ und F_n identisch verteilt sind. Die dem String i zugeordnete Funktion heißt dann f_i (d. h. $f_i := \phi(i)$)*
b) effizient berechenbar: *Es existiert ein zeitlich polynominiell beschränkter Algorithmus V, so dass $V(i,x) = f_i(x)$ für alle $i \in I(1^n)$ und alle $x \in \{0,1\}^{l(n)}$*

Es muss also in polynominieller Zeit möglich sein, Funktionen aus einem effizient berechenbaren Funktionsensemble auszuwählen und zu berechnen. Dazu muss der Algorithmus I eine Repräsentation der Funktion ausgeben, diese Repräsentation kann also auch nur höchstens polynominiell lang sein. Ein effizient

berechenbares Funktionsensemble kann somit nur $2^{poly(n)}$ viele Funktionen der $(2^n)^{(2^n)}$ möglichen Funktionen enthalten. Folglich ist $\{H_n\}_{n\in\mathbb{N}}$ nicht effizient berechenbar.

In Definition 2.5 wurde zur Indizierung ein probabilistischer Algorithmus I verwendet. Dieser gibt bei Eingabe 1^n Strings der Länge n aus, welche auf Funktionen abgebildet werden. Dabei bestimmt die Eingabe 1^n nur die Länge, welcher String ausgegeben wird, ist abhänging von „Münzwürfen“, die I durchführt. Daher kann auch das Ergebnis dieser „Münzwürfe“ als Funktionsbeschreibung verwendet werden. Zusammen mit Definition 2.4 erhält man dann folgende alternative Definition pseudozufälliger Funktionen:

Definition 2.6. (Effizient berechenbare pseudozufällige Funktionsensembles):
Ein **effizient berechenbares pseudozufälliges Funktionsensemble** *ist eine Menge von endlichen Funktion*

$$\left\{ f_s : \{0,1\}^{l(|s|)} \to \{0,1\}^{l(|s|)} \right\}_{s\in\{0,1\}^*}$$

mit $l : \mathbb{N} \to \mathbb{N}$*, die folgende Bedingungen erfüllt:*

a) Effiziente Berechenbarkeit: *Es existiert ein zeitlich polynominiell beschränkter Algorithmus, der bei Eingabe* s *und* $x \in \{0,1\}^{l(|s|)}$ $f_s(x)$ *zurückgibt.*
b) Pseudozufälligkeit: *Das Funktionsensemble* $F = \{F_n\}_{n\in\mathbb{N}}$*, definiert so, dass* F_n *gleichverteilt ist über* $\{f_s\}_{s\in\{0,1\}^n}$*, ist pseudozufällig.*

Nachdem jetzt Orakelmaschinen und effizient berechenbare pseudozufällige Funktionsensembles definiert wurden, soll zum Schluss dieses Abschnitts noch die **Nichtvorhersagbarkeit** definiert werden und gezeigt werden, dass effizient berechenbare pseudozufällige Funktionsensembles ebenfalls nicht vorhersagbar sind. Dazu werden hier nur Ensembles von Booleanfunktionen betrachtet, also Ensembles $\{F_n\}_{n\in\mathbb{N}}$, deren Zufallsvariablen F_n Werte aus dem Bereich der Funktionen $\{0,1\}^n \to \{0,1\}$ annehmen.

Definition 2.7. Nichtvorhersagbare Funktionsensembles
Sei F *ein Ensemble von Booleanfunktionen. Dann heißt* F **nicht vorhersagbar**, *wenn für alle Orakelmaschinen* M *und alle Polynome* $p(\cdot)$ *gilt:*

$$\text{Prob}\left[\text{test}^{F_n}\left(M^{F_n}(1^n)\right) = 1\right] < \frac{1}{2} + \frac{1}{p(n)}$$

wobei die Orakelmaschine M *bei bei Eingabe* 1^n *ein Tupel* $(x,\sigma) \in \{0,1\}^n \times \{0,1\}$ *ausgibt, das aus einem* x *aus dem Definitionsbereich von* F_n *und einer Vorhersage für den Funktionswert von von* x *besteht. Die Orakelmaschine darf beliebige Berechnungen durchführen und beliebige Anfragen stellen, aber natürlich nicht nach dem Funktionswert von* x *fragen. Weiterhin sei* $\text{test}^f(x,\sigma) := 1$ *genau dann, wenn* $f(x) = \sigma$ *gilt.*

Somit ist die Wahrscheinlichkeit, für ein von M gewähltes x den Funktionswert vorherzusagen, verschwindend nahe an $\frac{1}{2}$.

Satz 2.8. *Sei F ein effizient berechenbares pseudozufälliges Funktionsensemble. Dann ist F nicht vorhersagbar.*

Beweis von Behauptung 2.8

Angenommen, F sei vorhersagbar. Dann existiert eine Orakelmaschine M, die F vorhersagen kann, d. h.: es gilt:

$$\text{Prob}\left[\text{test}^{F_n}\left(M^{F_n}(1^n)\right) = 1\right] > \frac{1}{2} + \frac{1}{p(n)}$$

für unendlich viele n.

Dann lässt sich aus M ein M' konstruieren, so dass M' F und H unterscheiden kann. M' führt die gleichen Berechnungen und Orakelanfragen durch wie M, stoppt aber nicht, wenn M stoppt, sondern befragt noch einmal das Orakel mit der Anfrage x. Ist die Orakelantwort gleich σ, gibt M' 1 aus, sonst 0. M' überprüft also die Vorhersage von M. Damit gilt aufgrund der Annahme $\text{Prob}\left[M'^{F_n}(1^n) = 1\right] > \frac{1}{2} + \frac{1}{p(n)}$. Andererseits ist H das gleichverteilte Funktionsensemble, daher gilt $\text{Prob}\left[M'^{H_n}(1^n) = 1\right] = \frac{1}{2}$. Dann kann M' H und F unterscheiden, denn es gilt

$$\text{Prob}\left[M'^{F_n}(1^n) = 1\right] - \text{Prob}\left[M'^{H_n}(1^n) = 1\right] > \frac{1}{2} + \frac{1}{p(n)} - \frac{1}{2} = \frac{1}{p(n)}$$

Somit ist F nicht pseudozufällig. Dies ist ein Widerspruch zur Behauptung, d. h. die Annahme, F sei vorhersagbar, muss falsch sein. □

3. Konstruktion

Effizient berechenbare pseudozufällige Funktionsensembles (mit $l(n) = n$) können folgendermaßen aus Pseudozufallsgeneratoren konstruiert werden:

Konstruktion 3.1. *Sei G ein Pseudozufallsgenerator, der bei Eingaben der Länge n Strings der Länge $2n$ ausgibt. $G_0(s)$ bezeichne die erste Hälfte der Ausgabe von G, $G_1(s)$ die zweite Hälfte, so dass $G(s) = G_0(s)G_1(s)$ und $|s| = |G_0(s)| = |G_1(s)|$. Weiterhin sei für jedes $s \in \{0,1\}^n$ eine Funktion $f_s : \{0,1\}^n \to \{0,1\}^n$ definiert:*

Für $x = \sigma_1\sigma_2\sigma_3 \ldots \sigma_n \in \{0,1\}^n$ sei

$$f_s(x) = f_s(\sigma_1\sigma_2\sigma_3 \ldots \sigma_n) = G_{\sigma_n}(\cdots(G_{\sigma_2}(G_{\sigma_1}(s)))\cdots)$$

Somit wird $f_s(x)$ folgendermaßen berechnet:

```
y := s;
for i := 1 to n do
  y := G_σi(y);
endfor;
output y;
```

Dei F_n eine Zufallsvariable, die gleichverteilt ist über der Menge der Funktionen f_s mit $s \in \{0,1\}^n$. Daraus ergibt sich das Funktionsensemble $\{F_n\}_{n\in\mathbb{N}}$.

s ist somit ein „Startwert“, auf den der Pseudozufallsgenerator nach obigem Algorithmus n-mal angewendet wird. Dieser Startwert wird aus allen 2^n möglichen Startwerten gleichverteilt gewählt. Betrachtet man die Berechnung aller möglicher Funktionswerte von f_s, erhält man einen vollständigen Binärbaum der Tiefe n, wobei die Wurzel den String s enthält. Enthält ein Knoten den String r, so enthält der linke Kindknoten den String $G_o(r)$, der rechte den String $G_1(r)$. Die Wurzel wird auch als Level-0-Knoten bezeichnet, ihre Kindknoten als Level-1-Knoten, usw. Die Blätter des Baumes sind entsprechend Level-n-Knoten. Die Berechnung von $f_s(x)$ entspricht dann einem Weg von der Wurzel bis zu einem Blatt, wobei der Inhalt des erreichten Blattes der Wert $f_s(x)$ ist. Mit $x = \sigma_1\sigma_2 \dots \sigma_n$ wird, beginnend von der Wurzel, jeweils der linke Kindknoten eines Level-(i-1)-Knoten gewählt, wenn $\sigma_i = 0$ ist und der rechte, wenn $\sigma_i = 1$. Die Zufallsvariable F_n erhält als Wert einen solchen Baum aus der Menge der 2^n möglichen Bäume, die durch 2^n möglich Wurzelinhalte entstehen.

Um eine Funktion aus diesem Ensemble zu spezifizieren, genügt es, ihren „Startwert“ s zu spezifieren, daher ist auch nur ein n-Bit-String zufällig zu wählen und zu speichern, um eine solche Funktion auszuwählen und zu speichern.

Satz 3.2. *Das in Konstruktion 3.1 konstruierte Funktionsensemble F ist ein effizient berechenbares, pseudozufälliges Funktionsensemble.*

Da, wie in einem früheren Referat gezeigt, die Existenz von Einwegfunktionen die Existenz von Pseudozufallsgeneratoren impliziert, folgt

Korollar 3.3. *Wenn Einwegfunktionen existieren, dann existieren auch pseudozufällige Funktionen.*

Aus einem pseudozufälligen Funktionsensemble lässt sich auch ein Pseudozufallsgenerator konstruieren, falls $l(n) > \log_2(n)$ und ein pseudozufälliges Funktionsensemble nach Definition 2.6 gegeben ist. Man wählt unter Gleichverteilung ein $s \in \{0,1\}^n$ und damit eine Funktion $f_s(x)$. Dieses s ist der seed des Pseudozufallsgenerators, alle Funktionswerte, also $f_s(0^{l(n)})f_s(0^{l(n)-1}1)\cdots f_s(1^{l(n)})$, die Ausgabe. Da f_s aus einem pseudozufälligen Funktionsensemble gewählt wurde, ist f_s nach Satz 2.7 nicht vorhersagbar, daher ist die Ausgabe pseudozufällig. Die Ausgabe hat die Länge $l(n) \cdot 2^{l(n)} > \log_2(n) \cdot n$ und ist damit länger als die Eingabe, der Expansionsfaktor ist $\log_2(n)$.

Somit folgt

Korollar 3.4. *Pseudozufällige Funktionen (für $l(n) > \log_2(n)$) existieren genau dann, wenn Pseudozufallsgeneratoren existieren.*

Beweis von Behauptung 3.2: Offensichtlich ist F effizient berechenbar. Auch die Pseudozufälligkeit erscheint einleuchtend: Wenn s zufällig gewählt und G ein Pseudozufallsgenerator ist, so ist $G(s)$ und somit auch $G_0(s)$ und $G_1(s)$ nicht effizient von Zufallswerten zu unterscheiden, somit auch nicht $G_0(G_0(s))$ usw. Dies bestätigt der folgende Beweis, es wird angenommen, F wäre nicht pseudozufällig, es gäbe also eine Orakelmaschine, die F von dem gleichverteilten Funktionsensemble H unterscheiden kann. Daraus wird ein Algorithmus

konstruiert, der $G(U_n)$ und U_{2n}, also die Ausgabe des Zufallsgenerators G und Zufallsstrings, unterscheiden kann.

Hierbei wird die sogenannte *Hybridtechnik* angewendet. Dazu werden zwischen zwei Ensembles, die von einem gegebenen Algorithmus unterschieden werden können, polynominiell viele „Zwischenensembles“, Hybriden genannt, definiert. Da die Extreme unterschieden werden können, muss es einen nicht verschwindenden Unterschied in der Akzeptanzwahrscheinlichkeit geben, wenn der Algorithmus die Extreme als Eingabe erhält. Da es nur polynominiell viele Hybriden gibt, kann nicht zwischen allen benachbarten Hybriden ein verschwindender Unterschied in der Akzeptanzwahrscheinlichkeit bestehen, also kann dieser Algorithmus dann auch benachbarte Hybriden unterscheiden.

In diesem Fall sind die Extreme F_n nach Konstruktion 3.1 und H_n. Unter der Annahme, dass F_n nicht pseudozufällig ist, gibt es eine Orakelmaschine, die F_n und H_n unterscheiden kann. Die Hybriden werden folgendermaßen konstruiert. Die k-te Hybride entsteht, indem man die Level-k-Knoten mit gleichverteilt gewählten Strings versieht und die Beschriftung aller Knoten größerer Tiefe wie in Konstruktion 3.1 unter Verwendung des Pseudozufallsgenerators berechnet. Somit erhält man als 0-te Hybride F_n, da die Wurzel mit einem Zufallsstring versehen wird, als n-te Hybride erhält man H_n, da die Blätter mit Zufallsstrings versehen werden, G also nicht angewendet wird.

Für alle $0 \leq k \leq n$ wird die hybride Verteilung H_n^k, die als Werte Funktionen $f : \{0,1\}^n \to \{0,1\}^n$ annimmt, folgendermaßen definiert:
Es wird für alle $s_1, s_2, \ldots, s_{2^k} \in \{0,1\}^n$ eine Funktion
$f_{s_1,s_2,\ldots,s_{2^k}} : \{0,1\}^n \to \{0,1\}^n$ definiert:

$$f_{s_1,s_2,\ldots,s_{2^k}} := G_{\sigma_n}(\cdots(G_{\sigma_{k+2}}(G_{\sigma_{k+1}}(s_{idx(\sigma_k\ldots\sigma_1)})))\cdots)$$

$idx(\alpha)$ sei dabei der Index in der lexikographischen Reihenfolge der Binärstrings der Länge $|\alpha|$. Somit sind die $s_1, s_2, \ldots, s_{2^k}$ die Werte, die den Knoten der Level-k-Ebene zugewiesen werden, und zwar in der durch $idx(\alpha)$ festgelegten Reihenfolge. Zur Berechnung von $f_{s_1,s_2,\ldots,s_{2^k}}(x)$ wird mit den ersten k Bits von x ein s_j ausgewählt, danach (n-1)-mal der Pseudozufallsgenerator G angewendet, wobei die restlichen Bits von x bestimmen, welche der Funktionen G_0 oder G_1 verwendet wird. Die Zufallsvariable H_n^k sei gleichverteilt über alle $(2^n)^{2^k}$ solche Funktionen.

Da angenommen wurde, dass F im Widerspruch zur Behauptung nicht pseudozufällig ist, existiert eine Orakelmaschine M und und ein Polynom $p(\cdot)$, so dass für unendlich viele n gilt:

$$\Delta(n) := \left|\text{Prob}\left[M^{F_n}(1^n) = 1\right] - \text{Prob}\left[M^{H_n}(1^n) = 1\right]\right| > \frac{1}{p(n)}$$

Da M zeitlich polynominiell beschränkt ist, kann sie auch nur polynominiell viele Orakelanfragen stellen, daher existiert ein Polynom $t(\cdot)$, so dass M höchstens $t = t(n)$ Anfragen stellt. Mit Hilfe von M wird jetzt ein Algorithmus D konstruiert, der t Eingaben $G(U_n^{(1)}), G(U_n^{(2)}), \ldots G(U_n^{(t)})$ von t Eingaben

$U_{2n}^{(1)}, U_{2n}^{(2)}, \ldots U_{2n}^{(t)}$ unterscheiden kann, also die Ausgabe des Pseudozufallsgenerators von echten Zufallszahlen unterscheiden kann.

Algorithmus D: Der Algorithmus erhält $t = t(n)$ Eingaben $\alpha_1, \ldots, \alpha_t \in \{0,1\}^n$. Zuerst wählt D einen sogenannten „Checkpoint", ein k gleichverteilt aus $\{0, 1, \ldots, n-1\}$. Danach wird die Orakelmaschine M mit Eingabe 1^n aufgerufen, wobei D die Orakelanfragen von M wie folgt beantwortet.

Wie auch schon zur Konstruktion der Hybriden wird hier eine Abwandlung von Konstruktion 3.1 verwendet. Es werden nicht alle Funktionswerte berechnet, sondern nur die, nach denen die Orakelmaschine fragt, anders wäre es auch effizient nicht möglich, da es 2^n mögliche Funktionswerte gibt. Es werden Strings in die Level-k+1-Knoten plaziert, allerdings nicht zufällige, sondern jeweils die beiden Hälften eines Eingabestrings $\alpha_1, \ldots, \alpha_t \in \{0,1\}^{2n}$ in die beiden Kindknoten eines Level-k-Knotens.

Sei q_i eine Anfrage von M. Da D alle vorherigen Anfragen gespeichert hat, kann D feststellen, ob eine andere Anfrage in den ersten k Bits mit q_i übereinstimmt. Wenn nicht, so wird in Abhängkeit vom (k+1)-ten Bit die erste oder die zweite Hälfte der Eingabe α_i verwendet. Mit dieser beginnend wird die Berechnung wie in 3.1 fortgesetzt, d. h. beginnend vom (k+2)-ten Bit anhand von q_i entschieden, ob G_0 oder G_1 zu verwenden ist. Die Anfrage von M wird mit dem Inhalt des dabei erreichten Blattes beantwortet. Sei $P_0(\alpha)$ die erste Hälfte von α, $P_1(\alpha)$ die zweite. Dann antwortet D mit

$$G_{\sigma_n}(G_{\sigma_{n-1}}(\cdots(G_{\sigma_{k+2}}(P_{\sigma_{k+1}}(\alpha_i)))\cdots))$$

Stimmt dagegen q_i in den ersten k Bits mit einer vorherigen Anfrage überein, so würde in Konstruktion 3.1 die Berechnung beider Werte mindestens bis zu einem Level-k-Knoten identisch sein. Sei q_j die erste Anfrage, die in den ersten k Bits mit q_i übereinstimmt. Dann wird anstatt α_i erneut α_j verwendet und analog dem ersten Fall die Berechnung bis zu einem Blatt fortgesetzt. D antwortet also mit

$$G_{\sigma_n}(G_{\sigma_{n-1}}(\cdots(G_{\sigma_{k+2}}(P_{\sigma_{k+1}}(\alpha_j)))\cdots))$$

Da Maschine M höchstens $t(n)$ viele Anfragen stellt, stehen genug Eingaben zur Verfügung. Nachdem M angehalten hat, gibt D die Ausgabe von M zurück.

Was passiert nun, wenn D zufällige bzw. pseudozufällige Werte als Eingabe erhält?

- Erhält D Eingaben $\alpha_i \in U_{2n}$, also gleichverteilt gewählte Strings der Länge $2n$, und platziert jeweils Hälften davon in Level-(k+1)-Knoten, so ist dies das gleiche, als wenn zufällige Strings der Länge n in Level-(k+1)-Knoten platziert würden. Daher verhält sich M so, als ob die Hybride H_n^{k+1} als Orakelfunktion verwendet würde.
- Erhält D Eingaben $\alpha_i \in G(U_n)$ und platziert jeweils Hälften dieser pseudozufälligen Strings in Level-(k+1)-Knoten, so ist das das gleiche, als ob gleichverteilt gewählte Strings der Länge n in Level-k-Knoten platziert würden, da dann ebenfalls G angewendet würde und die entstehenden

pseudozufälligen Strings in Level-(k+1)-Knoten platziert würden. Daher verhält sich M so, als ob die Hybride H_n^k als Orakelfunktion verwendet würde.

Dies führt zu der Behauptung:
Sei $n \in \mathbb{N}$ und $t := t(n)$. Sei K eine Zufallsvariable, die die Checkpoint-Wahl des Algorithmus D (bei Eingabe von t $2n$-Bit-Strings) beschreibt. Dann gilt für alle $0 \leq k < n$:

$$\begin{aligned} \text{Prob}\left[D(G(U_n^{(1)}),\ldots,G(U_n^{t(n)})) = 1 | K = k\right] &= \text{Prob}\left[M^{H_n^k}(1^n) = 1\right] \\ \text{Prob}\left[D(U_{2n}^{(1)},\ldots,U_{2n}^{t(n)})) = 1 | K = k\right] &= \text{Prob}\left[M^{H_n^{k+1}}(1^n) = 1\right] \end{aligned}$$

wobei die $U_n^{(i)}$ und $U_{2n}^{(j)}$ unabhängige, über $\{0,1\}^n$ bzw. $\{0,1\}^{2n}$ gleichverteilte Zufallsvariablen sind.

Auf einen Beweis soll hier verzichtet werden, da dieser ziemlich aufwändig wäre und die Behauptung auch relativ offensichtlich ist.

Zusammen mit $\Delta(n) := \left|\text{Prob}\left[M^{F_n}(1^n) = 1\right] - \text{Prob}\left[M^{H_n}(1^n) = 1\right]\right| > \frac{1}{p(n)}$ folgt

$$\begin{aligned} & \text{Prob}[D(G(U_n^{(1)}),\ldots,G(U_n^{(t)})) = 1] - \text{Prob}[D(U_{2n}^{(1)},\ldots,U_{2n}^{(t)}) = 1] \\ = & \left(\tfrac{1}{n}\sum_{k=0}^{n-1}\text{Prob}\left[M^{H_n^k}(1^n) = 1\right]\right) - \left(\frac{1}{n}\sum_{k=0}^{n-1}\text{Prob}\left[M^{H_n^{k+1}}(1^n) = 1\right]\right) \\ = & \frac{1}{n}\left(\text{Prob}\left[M^{H_n^0}(1^n) = 1\right]\right) - \frac{1}{n}\left(\text{Prob}\left[M^{H_n^n}(1^n) = 1\right]\right) \\ = & \frac{\Delta(n)}{n} \end{aligned}$$

Dies ist aber größer als $\frac{1}{p(n)}$. Somit folgt, das D, ein probabilistischer, zeitlich polynominiell beschränkter Algorithmus, polynominiell viele Werte aus $G(U_n)$ von polynominiell vielen Werten aus U_{2n} unterscheiden kann, d. h. G kann kein Pseudozufallsgenerator sein. Dies ist aber ein Widerspruch zur Voraussetzung. Daher muss die Annahme, F sei nicht pseudozufällig, falsch sein, daraus folgt die Behauptung 3.2. □

4. Anwendung

Da pseudozufällige Funktionen im Gegensatz zu zufälligen Funktionen eine kurze Beschreibung haben, ist es möglich, sie effizient zu speichern und auszutauschen. Mit Hilfe einer pseudozufälligen Funktion können alle, die diese kennen, jederzeit gegebenen Parametern den gleichen, zufällig erscheinenden Wert zuordnen. Dies ist für jeden anderen, der die verwendete pseudozufällige Funktion nicht kennt, zumindest nicht effizient möglich.

Ein anschauliches Anwendungsbeispiel ist eine Gruppe von Menschen, die eine nur Mitgliedern bekannte pseudozufällige Funktion benutzen kann, um Mitglieder zu identifizieren. Dies läuft dabei folgendermaßen ab: Person A wählt einen Zufallswert der Länge n, also aus dem Definitionsbereich der pseudozufälligen Funktion, und übermittelt diesen der zu identifizierenden Person B. Diese antwortet mit dem Funktionswert.

Auf diese Art und Weise kann A überprüfen, ob B wirklich zu der Gruppe gehört, ohne B irgendetwas Relevantes über die geheime Funktion zu verraten, und keiner, der die geheime pseudozufällige Funktion nicht kennt, kann sich als Gruppenmitglied ausgeben, selbst wenn er beliebig viele (natürlich höchstens polynominiell viele) Gruppenmitglieder zur Identifikation auffordert, also zu beliebigen Werten seiner Wahl den Funktionswert erfährt.

Auf diese Art und Weise funktioniert auch das *Challenging Handshake Authentication Protocol (CHAP)*. Dieses wird als Teil des *Point-to-Point Protokolls (PPP)* zur Benutzeranmeldung verwendet. Der Server schickt an den Benutzer, der sich anmelden will, einen Zufallswert (CHAP-Challenge), der Benutzer berechnet mit dem MD5-Hashverfahren aus einem nur ihm und dem Server bekannten Schlüssel und dem Zufallswert einen Hashwert und schickt diesen an den Server. Der Server führt die gleiche Berechnung durch und vergleicht die Hashwerte, bei Übereinstimmung ist die Anmeldung erfolgreich. Die für diese Anwendung erforderliche Eigenschaft der verwendeten Funktion, bzw. des verwendeten Funktionsensembles mit dem Schlüssel als Funktionsbeschreibung und MD5 als effizientem Auswertungsalgorithmus, ist die Unverhersagbarkeit, d. h. es darf nicht möglich sein, ohne Kenntnis der Funktionsbeschreibung (des Schlüssels) für ein gegebenes x irgendwelche Eigenschaften des Funktionswertes vorherzusagen. Verwendet man ein pseudozufälliges Funktionsensemble, so ist dies nach Satz 2.7 gewährleistet. Allerdings gilt diese Aussage nur für polynominiell beschränkte Algorithmen nicht aber für solche mit expotentieller Laufzeit. Daher muss man n genügend groß wählen.

5. Zusammenfassung

Pseudozufällige Funktionen sind Funktionen, deren Funktionswerte für alle effizienten Algorithmen zufällig erscheinen. Sie können aus Pseudozufallsgeneratoren konstruiert werden und implizieren die Existenz von Pseudozufallsgeneratoren. Damit existieren pseudozufällige Funktionen genau dann, wenn auch Pseudozufallsgeneratoren und Einwegfunktionen existieren.

Während es Pseudozufallsgeneratoren ermöglichen, Folgen von Pseudozufallszahlen zu erzeugen, erlauben pseudozufällige Funktionen den direkten Zugriff auf alle Elemente einer solchen Folge von Pseudozufallszahlen. Daher können pseudozufällige Funktionen echte Zufallsfunktionen in allen effizienten Anwendung ersetzen, und damit insbesondere auch in der Kryptologie.

Literatur

[1] Goldreich, Oded. *Foundations of Cryptography.* Cambridge University Press, 2001