Großer Beleg
Zufallszahlen unter L4/DROPS
Technische Universität Dresden
Fakultät Informatik

Hans Marcus Krüger

Dezember 2004

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

Dieser Beleg befasst sich mit der Thematik der Erstellung von Zufallszahlen unter DROPS[22]. Die Arbeit startet mit theoretischen Betrachtungen, Vorstellung einiger kryptographischer Algorithmen und der Analyse von Zufallsquellen unter Linux. Anschließend werden einige Zufallspools untersucht und schließlich wird ein Zufallszahlengenerator (in Folge auch RNG für Random Number Generator) und Entropiepool für die DROPS vorgestellt.
Die zentrale Frage, die beantwortet werden soll ist, wie kann ein Zufallszahlenserver unter DROPS aussehen und welche Probleme müssen dafür gelößt werden.
Die Quelltexte, Daten, Hilfsprogramme und Sonstiges, die bei der Ausführung der Arbeit entstanden sind, können unter
http://www.inf.tu-dresde.de/∼hk407852/
abgerufen werden.
Zudem möchte ich mich bei Dr. A. Westfeld, Cristian Helmut, Dr. Cl.-J. Hamann, Dr. H. Klimant und Natalia Krasowska für die umfangreiche Unterstützung die sie mir bei dieser Arbeit geleistet haben recht herzlich bedanken.

Kapitel 2

Zufall

2.1 Wozu wird Zufall gebraucht

Zufall wird in der modernen Informatik an den verschiedensten Stellen gebraucht. Die Anforderungen sind unterschiedlich und zum Teil sogar widersprüchlich. Während das Qualitätsmanagement zum Testen von Softwaremodulen eine einfach zu reproduzierende Pseudo-Bit-Zufalls-Kette benötigt, um zum Beispiel Monte-Carlo-Tests zu realisieren[1] und Fehler effizienter zu beseitigen, könnte diese Eigenschaft für die Kryptographie ein großes Risiko bedeuten.
Eine ausführliche Liste der Anwendungsfelder für Zufallszahlen kann unter [2] gefunden werden. Der Beleg befasst sich jedoch hauptsächlich mit der Erstellung von kryptographisch geeignetem Zufall.
Aber auch im kryptographischen Umfeld sind die Anforderungen unterschiedlich, vor allem wenn es um die Menge des erzeugten Zufalls geht. Am Heim-PC wird nur relativ wenig Zufall benötigt. Große Server, wie zum Beispiel die Rechner einer großen Bank, benötigen sehr große Mengen an Zufall. In einigen Fällen ist „weniger guter“, aber schnell produzierter Zufall besser, als wenn die Anwendungen mehrere Sekunden blockieren. In anderen steht die Unvorhersehbarkeit im Vordergrund.

[1] Bei dieser Testmethode werden zufällige Eingaben erzeugt und das Programm darauf untersucht, ob es korrekt auf diese Eingaben reagiert. Wichtige Merkmale dabei sind die Berechnung eines korrekten Ergebnisses bei einer gültigen Eingabe oder die korrekte Fehlerbehandlung.

2.2 Zufallszahlen-Generatoren

Lange Zeit waren die Linear-Congruental-Algorithms als Generatoren sehr beliebt. Diese können durchaus sehr lange Sequenzen erzeugen, aber keine ist für eine kryptographische Anwendung geeignet.Ein Beispiel für solch einen Generator, aus den Unix Manual-Pages für rand(3), ist:

Beispiel 1 *Ein Linear-Congruental-Algorithms-Generator*

```
static unsigned long next = 1;
/* RAND_MAX assumed to be 32767 */
int myrand(void) {
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}
```

Dieser Generator erzeugt immer die gleiche Folge, die einem Zufallsstrom ähnelt. Wenn eine solche Sequenz alleine von den vorherigen Zuständen abhängt, das heißt, keine äußeren Einwirkungen vorhanden sind und diese Sequenz dennoch nicht von einem wirklichen zufälligen Strom unterschieden wird, so ist diese Zufallssequenz pseudo-zufällig[1].
Solche Generatoren sind deterministisch. Sie erstellen bei gleichem Anfangszustand immer die gleiche Ausgabe. Sie unterscheiden sich im Erzeugungsalgorythmus, den notwendigen Ressourcen und auch in ihrer Anwendbarkeit im Feld der Kryptographie, vor allem wenn es um die Frage der Sicherheit[2] dieser Erzeuger geht.
Generatoren, die solche deterministischen Ströme erzeugen, sind sehr interessant für die Informatik. Sie sind auch geeignet für die Kryptographie und in einigen Gebieten sogar explizit erwünscht[3]. Analog zu den Chifriersystemen können auch solche Generatoren das Kerckhoffs-Prinzip[20] umsetzen: Der Generator implementiert einen öffentlichen Algorithmus, besitzt aber einen internen Zustand, den Schlüssel. Solange dieser geheim bleibt, kann ein Angreifer die Sequenz nicht reproduzieren.
Das Problem bei solchen Generatoren ist, dass wenn der interne Zustand einmal

[2]Wenn in der Kryptographie Sicherheit durch Anwendung von Zufall gewährleistet wird, so ist es notwendig, dass ein Angreifer nicht in Kenntniss der Ausgabe des Generators kommt, sei es durch Erraten, Errechnen oder sonstige Methoden.

[3]Bei dem Pseudo-One-Time-Pad wird diese Eigenschaft ausgenutzt, um die Menge des Schlüsselmaterials gering zu halten[1, 20].

ermittelt worden ist, so lassen sich alle zukünftigen Ausgaben errechnen. Lassen sich die Operationen des Generators invertieren, so lassen sich sogar die Werte zurückrechnen, die in der Vergangenheit ausgegeben wurden.
Die Größe des Zustandsraums solcher Generatoren ist sehr wichtig. Sollte dieser zu klein sein, so kann ein *Electronic Code Book* (ECB)[4] erstellt werden und der Generator wird dadurch *gebrochen*.
Der Angriff mit dem ECB kann dadurch entschärft werden, indem solche Generatoren äußerliche Einflüsse aufnehmen, und damit ihren internen Zustand manipulieren. Ist diese externe Einwirkung erlaubt (das heißt, der Generator wird nicht in einen unerlaubten Zustand überführt) und zufällig in Zeit und Wert, so springt der Generator zu einem neuen Zustand.
Somit kann versucht werden, zufällige Ereignisse der Umwelt aufzunehmen und dabei die Entropie des Ereignisses in den Generator zu projizieren.
Sind diese äußerlichen Einwirkungen aber zu gering oder vorhersagbar, so kann der Angreifer eine Abschätzung über den ursprünglichen Zustand und das Ereignis machen und durch Probieren, seinen Generator in den gleichen Zustand zu überführen, womit dieser dann die gleiche Ausgabe erzeugt. Durch diese Methode muss der Angreifer nicht mehr alle theoretischen Möglichkeiten durchprobieren und könnte somit signifikante Einsparungen in Zeit und Resourcen in seiner Suche erzielen.
Infolge von (zu oft "durch"klingt nicht schön) solchen Schwachstellen in den Generatoren entstand, spätestens als kryptographische Applikationen auf Zufallszahlen angewiesen waren, ein ernsthaftes Sicherheitsproblem. Entweder verwendeten die Anwendungen diese Funktionen trotz ihrer Einschränkungen weiterhin, oder jede Applikation entwickelte ihren eigenen Pool. Auch die zweite Lösung ist inakzeptabel, da bei so vielen Implementierungen entsprechend viele Implementierungsfehler vorhanden sind. Zudem fehlt den meisten Programmierern das nötige Wissen und die Zeit, um korrekte Pools zu erstellen. Außerdem ist damit noch immer nicht das Problem der Einspeisung von Zufall gelöst. Eine Reihe von kritischen Sicherheitslücken waren die Folge.
Beispielsweise verwendete Netscape 4.7 eine Routine zur Erstellung von 128 Bit Schlüsseln, die nur einen beschränkten Wertebereich besaß, womit der Entropiegehalt der Schlüssel letztendlich nur 47 Bit betrug. Innerhalb weniger

[4]Bei dieser Methode erstellt der Angreifer ein Nachschlagewerk, indem zu jeder Ausgabe der entsprechende Zustand zugeordnet wird. Nach einigen Ausgaben kann der Angreifer mit sehr hoher Wahrscheinlichkeit den internen Zustand tabellarisch ermitteln.

Minuten konnte jede verschlüsselte Verbindung „gebrochen“ werden [15, 16]. [14] listet weitere in der Einleitung auf.
Ein Lösungsansatz ist die Bereitstellung von einem Pool durch das Betriebssystem. Dieser Pool kann von mehreren Spezialisten untersucht werden. Alle Quellen fließen in diesen Pool und alle Anwendungen beziehen von dort ihren Zufall. Linux, beispielsweise, verfolgt dieses Prinzip. Standards wie das PKCS#12 der RSA Security[21] sollen zudem Schnittstellen definieren, die Plattform übergreifend verwendet werden können.

2.3 Wie kann Zufall gewonnen werden?

Zufall kann aus mehreren Quellen gewonnen werden. Linux verwendet hauptsächlich die Intervallzeiten zwischen den Unterbrechungen (Interrupts) und der Interaktion Mensch-Computer.
Vorteil dieser Quellen ist, dass sie implizit in allen modernen Rechnern vorhanden sind. Nahezu alle Modelle besitzen eine Maus und eine Festplatte. Die meisten auch schon einen Netzwerkanschluss. Jedoch müssen noch weitere Faktoren berücksichtigt werden. Zum Beispiel werden alle Bewegungen der Maus dem X-Server gemeldet und können dort von fremden Programmen überwacht werden. Sollte der Kernel aus Optimierungsgründen die Maus „pollen“, so ist erst recht nicht mit Variationen der Intervalle zu rechnen. Aber auch unter Verwendung von Unterbrechungen ist Vorsicht geboten, sollte die Plattform über keine hochauflösende Uhr verfügen. Unter Linux wird, beispielsweise, gerne die Zeitzählung mit Hilfe der *Jiffies* verwendet. Dabei ist ein *Jiffy* 1/100s lang. Im Vergleich zu den üblichen 5ms bis 15ms Plattenzugriffszeiten ist diese Zeitzählung eher ungeeignet, wird aber dennoch verwendet[5].
Ein Problem bilden auch Puffer und Read-Ahead, das sich vor allem auf Betriebssystemebene bezieht. Damit, beispielsweise, sensitive Daten erstellt werden können, muss zuvor das Programm geladen werden. Ist der Pool gefüllt, bleibt dieser unverändert von den Zugriffen auf die Festplatte. Das Programm befindet sich nun im Puffer. Greift es nun auf dem Pool zu, wird dieser wieder von den Daten der Maus gefüllt, da keine weiteren Zugriffe auf die Festplatte stattfinden. Bei solchen komplexen Gebilden wie ein Betriebssystem können noch viele weitere

[5]Für Plattformen, die den Assemblerbefehl *rdtsc*, der einer hochauflösenden Uhr entspricht, nicht kennen wird, stattdessen, die Zählung mit Hilfe der Jiffies realisiert.

solcher Nebenwirkungen auftreten.
Dabei sind vor allen Festplatten sehr interessante Quellen. [5] ist eine interessante Studie über die Gewinnung von Zufall unter der Verwendung von Festplatten. Sie besagt, dass Festplatten für die Erstellung von Zufall geeignet sind, da sich Variationen in den Zugriffszeiten auf wohl untersuchte physikalische Ereignisse zurückführen lassen, die als zufällig gelten.
Weiterhin interessant sind auf Hardware basierende Zufallszahlengeneratoren. Solche Generatoren verwenden Einflüsse der Umwelt, in der sie sich befinden. Dabei wird bei diesen Geräten auch unterschieden, ob der Zufall noch aufbereitet werden muss, oder ob dies im Gerät selbst geschieht [6]. Sollte dies nicht so sein, so muss der Anwender oder das Betriebssystem diese Nachverarbeitung durchführen.
Leider sind solche Geräte noch eine Seltenheit in Systemen der Endanwender. Jedoch könnten aktuelle Entwicklungen dies in den nächsten Jahren ändern[6]. Aber ein solcher Generator löst nicht unbedingt alle Probleme. Auch solche Generatoren können fehleranfällig sein. Zudem müsste dem Hersteller des Generators voll und ganz vertraut werden. Eine Alternative wäre das Zusammenführen mehrerer Quellen. Techniken dafür werden in Kapitel 4 vorgestellt.
Es sind aber noch weitere Quellen erdenklich. Zum Beispiel bilden TCP-Sequenznummern eine interessante Zufallsquelle. Alle gängigen Betriebssysteme wählen zur Bootzeit oder zu jedem Aufbau einer neuen TCP-Verbindung eine zufällige Zahl, die fortlaufend inkrementiert wird. Besitzt der Rechner Zugriff auf einen ausreichend ausgelasteten Netzwerkknoten, so könnten diese Daten dazu beitragen, den Zufallspool zu manipulieren. Auch die Daten der Pakete können Entropie beisteuern. Dabei muss noch nicht einmal der komplette Netzwerkverkehr verarbeitet werden; besitzen die Protokolle Prüfsummen, so spiegelt sich, wie in Kapitel 4 genauer beschrieben wird, die Entropie der Daten in diesen Prüfsummen wider.
Auch Quellen, die zu einem bestimmten Wert tendieren, sind akzeptabel. Mit Hilfe von unterschiedlichen Techniken kann dennoch ein ausgeglichener Strom erzeugt werden. [7] nennt dafür einige Lösungen.
Wichtig, in allen Fällen, ist nur: Die Entropie darf niemals überbewertet werden. Solange eine sichere Abschätzung der Entropie und eine korrekte Aufarbeitung erfolgt, sind alle erdenklichen Quellen erlaubt.
Gesucht wird somit ein Generator, der möglichst alle oben erwähnten Anforde-

[6]Sowohl die Entwicklung von DRM und TCPA, in denen kryptographische Operationen die Grundlage bilden und auf Zufall angewiesen sind, wie auch die neueren „Chipsets“ mit eingebauten Hardware-Zufallszahlengenerator werden in den nächsten Jahren dieses Bild verändern.

rungen entspricht und gleichzeitig die genannten Schwachstellen nicht besitzt.

Kapitel 3

Fallbeispiel: Zufallsquellen unter Linux

Dieser Abschnitt befasst sich mit der Analyse von den unterschiedlichen Quellen, die unter Linux für die Entropiegewinnung verwendet werden.
Die zu Grunde liegenden Daten der folgenden Auswertung stammen aus einem Acer Travelmate 290 Laptop, Intel Centrino, 1.5GHz, 512 MB Ram, Debian Sarge, Linux 2.6.7. Die Daten wurden während des normalen Betriebs gesammelt. Eine einzelne, zeitliche Zuordnung zu laufender Anwendung wurde nicht vorgenommen. Im Laufe von 3 Tagen wurde die Eingabe in den Entropiepool dupliziert. Dabei wurden 14.000 Proben genommen, die insgesamt 141.000 Bit Entropie enthalten sollten. Insgesamt konnten vier Quellen ausgemacht werden: Die Festplatte, die Maus, die Tastatur und eine vierte unbekannte Quelle. Dabei sorgen die Maus und die Festplatte für 96% der gesammelten Entropie.

Quelle	Proben		Entropie		∅-Ent.	Approx.	∅-Approx.
Festplatte	6834	46%	66480	47%	9,728	1852	0,27
Maus	7101	48%	68011	48%	9,578	95904	13,51
Tastatur	439	3%	4386	3%	9,990	2286	5,21
Unbekannt	223	1%	2162	2%	9,695	178	0,80
Summe:	14597	100%	141039	100%	9,662	103884	7,12

Eine genauere Analyse wird unter Anhang A durchgeführt.
Nun stellt sich die Frage, wie viel Entropie wirklich in den Proben enthalten ist. Diese Frage ist schwer zu beantworten. Vielmehr soll sich der Rest dieses Abschnitts mit zwei Abschätzungen befassen. Zuerst wird eine Abschätzung nach oben durchgeführt. Ist der ermittelte Wert geringer als die Abschätzung, so er-

folgt im Fallbeispiel auf jeden Fall eine Überbewertung der Entropie.
Anschließend wird eine vorsichtige Abschätzung untersucht: Wie viel Entropie ist mindestens enthalten? Die vorsichtigste Abschätzung setzt diesen Wert stets auf Null, was aber für die Praxis nicht interessant ist. Davon abgesehen müsste jede Quelle einzeln untersucht werden und diese Untersuchungen könnten durchaus sehr detaillierte Kenntnisse über die Systeme erfordern. Diese genauen Untersuchungen sollen hier nicht geführt werden. Vielmehr soll eine Basis für spätere Korrekturen des Entropiegehalts gewonnen werden.
Für die erste Betrachtung dieser Zahlen soll die Tatsache zur Hilfe genommen werden, dass sich, im Mittel, bei zwei aufeinander folgenden Zufallszahlen die Hälfte aller Bits verändern. Diese Annahme wurde für die sehr einfach gewählte Approximation gewählt und ist eine Abschätzung nach „oben“. Bessere Abschätzungen würden zusätzlich noch lineare Abhängigkeiten und andere mehr untersuchen, um eine anständige Aussage über den Gehalt an Zufall in der Zahl zu bestimmen.
Die approximierte Entropie wurde aus der doppelten Hemmingdistanz von zwei aufeinander folgenden Zahlen ermittelt, wobei dieser Wert 32 nicht überschreiten durfte:

$$A(x) = min(2Hem(x_i, x_{i-1}), 32)$$

Unter Anbetracht der Tatsache, dass die Eingaben relativ konstant sind, wie das Diagramm der dritten Quelle in Anhang A verdeutlicht, muss auch angenommen werden, dass die Entwickler des Linux-Pools den Zeitpunkt des Auftretens eines Ereignisses als zufällig auffassen, und dies dementsprechend bei der Berechnung der Entropie berücksichtigen. Dies ist durchaus sinnvoll, jedoch muss darauf hingewiesen werden, dass sich hier nur die logische Zeitrechnung anbietet: Das einzig Wichtige ist die Reihenfolge, in der die Ereignisse auftreten, nicht aber die absolute Zeit in der (sie?) geschieht.
Entschärft wird die großzügige Abschätzung der Entropie durch das Falten am Ende der Erstellung des Zufalls, da somit die doppelte Menge an Entropie für die eigentliche Erzeugung benötigt wird. Wie zuvor angedeutet werden alle Bewegungen der Maus an den Anwender gemeldet und auch der Zugriff auf die Festplatte lässt sich womöglich abschätzen. Werden diese Tatsachen noch genauer beachtet, so muss die Approximation nochmals nach unten korrigiert werden. Es deutet also stark darauf hin, dass der reale Zufallsgehalt der Eingaben unter Linux nochmals genauer ermittelt werden muss. Diese detaillierteren Betrachtungen wurden in diesem Beleg jedoch nicht durchgeführt.

Kapitel 4

Kryptographische Grundlagen

In diesem Abschnitt sollen einige kryptographische Verfahren und Algorithmen vorgestellt werden, die zum Verständnis der nachfolgenden Abschnitte beitragen.

4.1 Erstellen eines ausgeglichenen Stroms

Sollte ein Strom nicht die erforderliche Eigenschaft besitzen, dass Nullen und Einsen gleichmäßig auftreten, so kann dies durch das bitweise Verknüpfen mit der Exklusiv-Oder-Operation erreicht werden. Dieses Verfahren wird in [7] beschrieben. Dies ist aber noch keine Aussage darüber, ob das Endprodukt zufällig ist. Jedoch es gilt als Folge von [7]: Ist ein Strom in endlich vielen Stellen zufällig, tendiert jedoch zu einer bestimmten Zahl, so kann daraus ein zufälliger, ausgeglichener Strom erzeugt werden. Das Verfahren eignet sich auch für das Zusammenführen von unterschiedlichen Quellen.
Das vorgestellte Verfahren der exklusiven Verknüpfung hat aber einen großen Nachteil. So ist die Ausgaberate direkt von der langsamsten Quelle abhängig. Ein anderer, viel versprechender Ansatz ist die Anwendung von Hash-Funktionen, wie in Folge erläutert wird. Es können aber auch symmetrische Verschlüsselungssysteme oder andere Funktionen verwendet werden. Wichtig ist nur, dass die Funktion gewisse Eigenschaften besitzt, die anschließend betrachtet werden.

4.2 Zufall in Hash-Funktionen

Eine Hashfunktion F bildet einen Eingabevektor M der *beliebigen* Länge m auf einen Ausgabevektor D der *konstanten* Länge l ab. Die wichtigsten Eigenschaften,

die für die Erstellung von Zufall gebraucht werden, sind:

1. Die Ausgabe ist statistisch uniform, unabhängig von der statistischen Verteilung der Eingabe.

2. Die Funktion besitzt den *Avalanche-Effekt*[1]

3. Die Hashfunktion ist eine *Einwegfunktion*. Das bedeutet, dass diese Abbildung nicht eindeutig und auch nicht invertierbar ist.

4. Die Hashfunktion besitzt ein *Gedächniss*. Das Endprodukt der Hashfunktion ist abhängig von allen Eingaben und der Reihenfolge, in der diese Eingaben getätigt wurden.

Somit eignen sich diese Funktionen hervorragend, um Zufall aufzubereiten. Sollte ein Strom bis auf endlich viele Stellen vorhersagbar sein, kann dieser Strom beliebig lange durch die Hashfunktion verarbeitet werden, bis ein akzeptables Ergebnis vorliegt. Dazu sind folgende Randbedingungen notwendig:

1. Der Angreifer kann über die Eingaben nur Annahmen treffen, diese aber nicht direkt bestimmen

2. Der Angreifer kann keine Zwischenergebnisse einsehen

Ein Angreifer kann eine Eingabe mit einer Wahrscheinlichkeit p_e korrekt bestimmen. Besitzt diese Eingabe endlich viele endlichen Teilketten, die unvorhersehbar sind, so setzt sich p_e aus den Wahrscheinlichkeiten p_t der einzelnen unvorhersehbaren Teilketten T zusammen. Sind im Eingabestrom n solcher Teilketten vorhanden und wird der Eingabestrom als Ganzes betrachtet, so kann der Angreifer diese Eingabe mit der Wahrscheinlichkeit von

$$p_e = p_t^n \tag{4.1}$$

[1]Der Avalanche-Effekt besagt, dass bei jeder Änderung der Eingabe sich durchschnittlich die Hälfte der Ausgabebits verändern[1, 12].

korrekt vorhersagen[2].

Beispiel 2 *Angenommen, dass ein Eingabestrom aus 510 Einsen und anschließend zwei Bits bestehe, die mit den Wahrscheinlichkeiten von je 25% 00, 10, 01 und 11 ergeben. Die ersten 510 Stellen sind immer konstant, die letzten aber zufällig. Somit kann der Angreifer mit der Wahrscheinlichkeit von 25% die verwendete Eingabe erraten. Da die Hashfunktion deterministisch ist, weiss der Angreifer mit der gleichen Sicherheit, welche Ausgabe die Funktion produzieren wird.*

Wenn solche unvorhersehbare Teilketten durch eine Hashfunktion verarbeitet werden, so spiegelt sich der Zufall im erzeugtem Hashwert wider. Die Wahrscheinlichkeit, mit der eine bestimmte Ausgabe durch die Hashfunktion erstellt wird ist aber auch abhängig von dem Wertebereich des Hashwerts und beträgt bei idealem Hashverfahren den Wert $p_h = \frac{1}{2^l}$, wobei l die Länge in Bits des Hashwerts ist. Dieser Wert kann, trivialerweise, nicht unterschritten werden, womit, am Ende, die Wahrscheinlichkeit p_a der Ausgabe durch

$$p_a = max(p_e, p_h) \tag{4.2}$$

gegeben wird.

Beispiel 3 *Die Hashfunktion SHA1 [17] besitzt eine Ausgabe der Länge $l = 160$ Bits. Ist die Wahrscheinlichkeit p_e für die Eingabe größer als $p_h = \frac{1}{2^{160}}$, das heißt, es sind weniger als 160 Bit Zufall in der Eingabe vorhanden, so ist die Wahrscheinlichkeit der Ausgabe mit der Wahrscheinlichkeit der Eingabe P_e zu bestimmen. Besitzt die Eingabe weit mehr als 160 Bit Zufall, so ist die Wahrscheinlichkeit der Ausgabe konstant $p_h = \frac{1}{2^{160}}$, da der Hashwert nur 160 Bit Entropie aufnehmen kann.*

Zusätzlich können mehrere Quellen zu einem Strom konkateniert und durch die Hashfunktion verarbeitet werden. Sind die Quellen statistisch unabhängig, so ist p_e

[2]In Wirklichkeit ist diese Berechnung sehr viel komplexer. So besteht die komplette Kette aus Teilketten T_i, die je eine betimmte Wahrscheinlichkeit $P(T_i)$ besitzen. Konkateniert müsste die gesamte Wahrscheinlichkeit durch Anwendung der Baye'schen Formel für bedingte Wahrscheinlichkeit errechnet werden. Wenn angenommen wird, dass die unverhersehbaren Teilketten T_i^* unabhängig voneinander sind, d.h. $P(T_i^*|T_j^*) = P(T_i^*)$, der Einfachheit halber allen Teilketten die gleiche Wahrscheinlichkeit zugeordnet wird ($P(T_i^*) = P(T_j^*)$) und der Rest der Kette vorhersagbar ist, so kann die Gleichung 4.1 als gültige Approximation gewählt werden, da $p_e \leq p_t^n$ gilt.

das Produkt der Wahrscheinlichkeiten aller Quellen. Für mehrere Quellen entsteht somit

$$p_a = max(p_h, \prod_i p_{ei}) \tag{4.3}$$

De Folge ist, wenn der Angreifer auch nur über eine einzige Quelle keine Aussage machen kann, kann dieser die Ausgabe nicht bestimmen, mit einer Wahrscheinlichkeit größer als die der unbestimmten Eingabe. Somit kann die Funktion auch sehr gut zur Kombination von Quellen verwendet werden. Zudem können Quellen unterschiedlicher Geschwindigkeiten kombiniert werden.

4.3 General Feedback Shift Register

General Feedback Shift Register (GFSR) waren lange Zeit sehr beliebte Pseudo-Zufalls-Bit-Generatoren. Sie ließen sich sehr einfach in Hardware implementieren. Der nachfolgende Zustand wird aus dem aktuellen erstellt, indem das Register geschoben und das neue Bit aus dem vorherigen Zustand errechnet wird [1]. Diese Berechnung erfolgt mittels eines Polynoms $F(x)$.
Dabei kann ein Register der Länge l, 2^l verschiedene Werte annehmen bzw. 2^l unterschiedliche interne Zustände besitzen. Der Generator besitzt somit einen Zustandsraum Z, einen aktuellen Zustand z und ein charakteristisches Polynom $F(x)$.

1. $z \in \{0,1\}^l$
2. $F : \{0,1\}^l \rightarrow \{0,1\}^l$
3. $z_{i+1} = F(z_i)$

Der Generator $GFSR(F, z_0)$ erzeugt eine Halbordnung S. Die Länge dieser Sequenz kann maximal 2^l betragen. Für die Erstellung von Zufallszahlen ist das wünschenswert, weil der Generator erst nach 2^l Schritten anfangen würde, die Sequenz zu wiederholen. Aber bei der falschen Wahl von F oder z_0 kann die Sequenz zu einem Zyklus kleinerer Länge degenerieren. Wenn jedoch $F(x)$ als primitives Polynom über GF(2) gewählt wird, so ist die Sequenz $2^l - 1$ lang[2, 1]. Sind die Zustände als natürliche Zahlen kodiert, so ist jede Zahl zwischen 1 und $2^l - 1$ in der Sequenz S enthalten und somit gültig für $GFSR = (F, z_0)$. Der große Vorteil ist, dass jede Zahl dieses Wertebereichs als Startwert verwendet werden kann. Mit diesen Vorgaben ist der Generator nur noch von $F(x)$ abhängig und dadurch das

Initialisierungsproblem gelöst. Die Null fällt aus S raus, da sie das neutrale Element in GF(2) bildet.
Leider sind GFSR nicht sicher für kryptographische Anwendungen. Jedes GFSR besitzt eine lineare Komplexität n. Die lineare Komplexität ist der Grad des „kleinsten“ GFSR, der die Ausgabe reproduzieren kann und dieser Wert existiert immer und ist endlich. Mittels des Algorithmus *Berleykamp-Massey* kann, nach Analyse von $2n$ Ausgabebits, ein solches minimales GFSR errechnet werden[1].
Man könnte auf die Idee kommen, den Grad des Polynoms einfach so groß zu wählen, dass eine solche Analyse nicht mehr realistisch ist. Jedoch ist das Errechnen solcher primitiven Polynome aufwendig und erfordert die Kenntnis der Primfaktoren von $2^l - 1$.
Dennoch werden GFSRs für unterschiedliche kryptographische Anwendung verwendet. Vor allem als *Streamciphers* sind sie sehr beliebt. Man behilft sich verschiedener Konstruktionen, um sie resistenter zu machen.

4.4 Twisted General Feedback Shift Register

Twisted General Feedback Shift Register (TGFSR) bauen auf GFSRs mit primitiven Polynomen über GF(2) auf. Es adressiert jedoch zwei wesentliche Probleme der GFSR:

1. Die kurze Periode der Sequenz
2. Die Ineffizienz der in Software implementierten Generatoren für größere Polynome

Die Idee ist einfach und wurde zuerst von [9, 10] vorgeschlagen. Man nehme einfach mehrere GFSRs und fasse sie als einen Vektorraum auf. Auf Maschinen mit $w = 32$ Bit Wordbreite bietet sich an, 32 GFSRs zu verwenden. Das wird im Speicher spaltenweise abgelegt. Somit befinden sich (idealerweise) in einem Maschienenword nur Operanden des gleichen Grads. Die Operation F wird immer über alle Register gleichzeitig ausgeführt. Entscheidend ist das Mischen der Sequenzen am Ende, denn wenn dieser Schritt nicht wäre, würden alle GFSR parallel laufen und alle hätten stets den gleichen Zustand. Die Ausgabe wäre immer 0 oder $2^w - 1$ (alles Eins).
Das Mischen erfolgt dabei nach folgendem Prinzip:

1. Der Vektor $x_0 := (x_{w-1,0}, ..., x_{00})$, der die niederwertigsten Bits aller GFSRs darstellt, wird nach links gerollt.

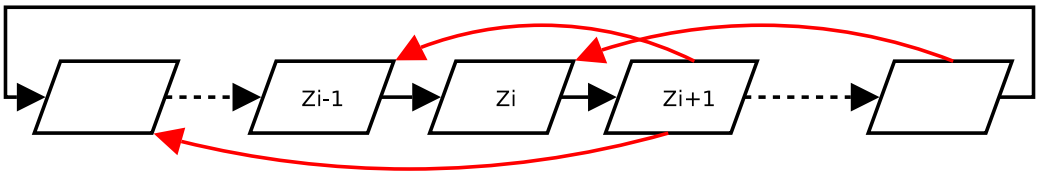

Abbildung 4.1: Auswirkungen von externen Einflüssen auf ein TGFSR

2. Wenn $x_{w-1,0}$ (das höchswertige Bit von x_0) eine 1 war, so ist $x_0 = x_0 \oplus A$, wobei A ein konstanter Vektor aus $\{0,1\}^w$ ist.

Die Autoren behaupten, dass die Sequenz eine Länge von $2^{lw}-1$ besitzt, und dass der Generator alle gängigen statistischen Tests erfolgreich bestanden hat.
Das Initialisierungsproblem ist hier ebenfalls gelöst: Jede Belegung des Felds außer dem neutralen Element ist ein gültiger Zustand. Somit eignen sich die TGFSRs auch hervorragend, um externe Einflüsse aufzunehmen. Wird der neue Wert x_0 zuvor mit einem zufälligen Wort verknüpft, so „springt“ der Generator an einem ganz anderen, jedoch gültigen Platz im Zustandsraum (Abb. 4.1). Diese Verknüpfung kann mit der Operation Exklusiv-Oder erfolgen. Hierbei muss aber beachtet werden, dass eine Kollision erzeugt werden kann: Durch die geschickte Eingabe kann der 2^{lw}. Zustand erreicht werden. Der Pool befindet sich dann in einem Zyklus der Länge 1. Bei der nächsten Eingabe kann dieser aber wieder aus diesem Zyklus befreit werden, womit die Wahrscheinlichkeit, dass sich daraus ein Problem ergibt, sehr gering ist. Ein Angreifer könnte sich dies jedoch zu nutze machen. Er braucht jedoch die Kenntnis über den aktuellen Zustand, um diese Kollision zu erzeugen. Diese zufälligen Manipulationen bleiben dadurch erhalten, dass die Sequenz an einer anderen Stelle des Zustandsraums fortgesetzt wird. Sie dienen somit auch dazu, den Generator in eine unbestimmte Ausgangsstellung zu überführen, von wo an die Erzeugung ohne externe Einflüsse fortgeführt werden kann.
Abbildung 5.2 auf Seite 24 verdeutlicht, wie dieses Verfahren unter Linux zum Einsatz kommt.

Kapitel 5

Bereits vorhandene Entropiepools

5.1 Linux

5.1.1 Grobstruktur

Der Linux-Pool ist in zwei Pools untergliedert, den primären und den sekundären Pool. Der Zufall kann in beide Pools geschrieben werden. Unter normalen Bedingungen wird die Erzeugung immer über den sekundären Pool realisiert. Ist in diesem Pool nicht genug Entropie vorhanden, so wird er zuvor noch mit der Entropie des primären Pools aufgefüllt.
Linux bezieht die Entropie aus mehreren Quellen. Dabei wurde bei der Konzipierung des Pools auch darauf geachtet, dass nicht zwingend gute Zufallsquellen vorhanden sind. Mit der verwendeten Architektur geschieht dennoch eine angemessene Aufbereitung des Zufalls. Die Erzeugung geschieht durch das Hashen des sekundären Pools.
Vorweg sei erwähnt, dass der Begriff „Entropie“ in diesem gesamten Abschnitt nicht den realen Zufallsgehalt zum Ausdruck bringt, sondern den von Linux ermittelten Wert darstellt. An den Stellen, wo dies nicht zutrifft, wird explizit darauf hingewiesen.

5.1.2 Entropiegewinn

Es existieren zwei Schnittstellen, um Zufall einzugeben. Der Anwender kann Zufall über die Geräteschnittstelle /dev/random einspeisen und mittels eines

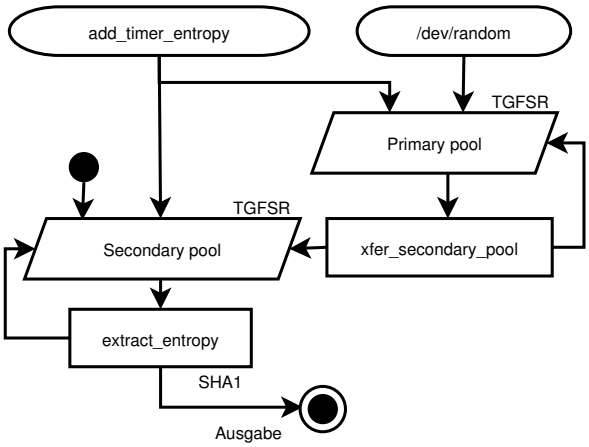

Abbildung 5.1: Linux-Pool

IO-Controll-Befehls den Entropie-Zähler des Pools beeinflussen. Zusätzlich wird Zufall von eingebauten Gerätetreibern eingegeben. Zufallsquellen sind Zugriffszeiten, Bewegungen mit der Maus, Tastatureingaben und noch weitere.
Die Funktion add_timer_randomness bildet dabei den zentralen Eintrittspunkt für die zweite Art der Zufallseingabe. Diese Funktion führt Buch über die letzten Vorkommnisse der Ereignisse und errechnet anhand dieser Daten die kleinste Differenz der letzten drei Intervallzeiten. Dieser Wert stellt die Basis für die Bestimmung der Entropie dar, die in der jeweiligen Probe enthalten ist, geht aber selber nicht in die Manipulation des Pools mit ein. Bei der Maus und der Tastatur werden die ausgelesenen Daten in den Pool gegeben, bei Festplatten, die konstant bleibende Geräte-Nummer(Major, Minor) und bei sonstigen Unterbrechungen die Nummer der Unterbrechung (IRQ 1 bis 15). Zusätzlich wird dieser Wert mit einer Uhr verknüpft. Bei Systemen, die den *rdtsc*-Befehl kennen, wird diese Uhr verwendet, sonst verwendet Linux den eigene Zeitzählung, die *Jiffy*.
Der primäre Pool wird immer zuerst gefüllt. Ist dieser voll, so wird der sekundäre Pool aufgefüllt. Anschließend wird nur noch jede 4096. Probe abwechselnd in jeden Pool eingefügt.
Bei solchen Eingaben ist es natürlich sehr wichtig, dass keine Überschätzung der Entropie stattfindet. Leider hat die Auswertung unter Kapitel 2 aber das Gegenteil angedeutet.

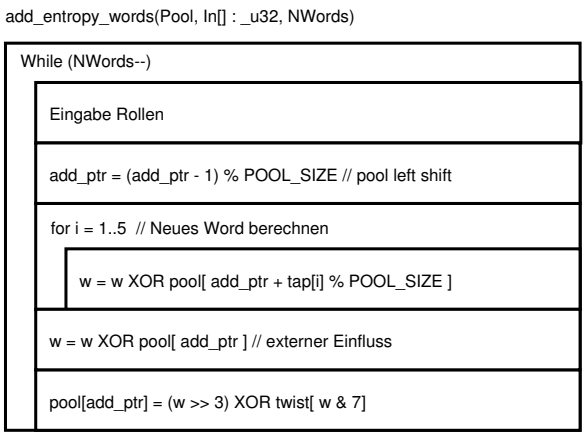

Abbildung 5.2: Die TGFSR-Routine add_entropy_words

5.1.3 Die Pools

Das zentrale Element der Pools bilden zwei unabhängige Twisted General Feedback Shift Register (TGFSR). Diese Register sind normalerweise 512 Wörter groß. Somit besitzen sie einen Zyklus $l = 2^{nw} - 1$, wobei w die Wortlänge ist. Für den Pool, unabhängig der Architektur, beträgt $w = 32$ Bit. Somit ist $l = 2^{16384} - 1$, was lang genug ist, um einen ECB-ähnlichen Angriff auf den Pool zu verhindern. Zusätzlich besitzt jeder Pool seinen eigenen Entropiezähler, in dem die Menge an Entropie festgehalten wird, die sich im Pool befindet.

Ein größeres Problem stellt die Vunerabilität von GFSR dar. Sollte für die TGFSR die gleiche Vunerabilität gelten, wie bei den GFSR, was wegen ihrer Ähnlichkeit anzunehmen ist, so ist der interne Zustand mit einer relativ geringen Menge von produzierten Ausgabewörter berechenbar [1]. Deshalb ist es unumgänglich, die Ausgabe nochmal so zu transformieren, dass Rückschlüsse auf beliebige Teile der TGFSR ausgeschlossen sind. Unter Linux wird das durch die Anwendung von SHA1[17] oder MD5[18] gewährleistet.

Gleichzeitig ist auch der Zustandsraum groß genug, um viele Zufallszahlen zu erzeugen und trotzdem eine Wiederholung der Sequenz zu vermeiden. Dies ist ein großer Vorteil, wenn es nicht darum geht, sehr guten Zufall zu erzeugen, oder wenn Entropiegewinn durch die Hardware ganz ausgeschlossen ist. In diesem Fall muss natürlich das Problem der Initialisierung der Pools gelöst werden.

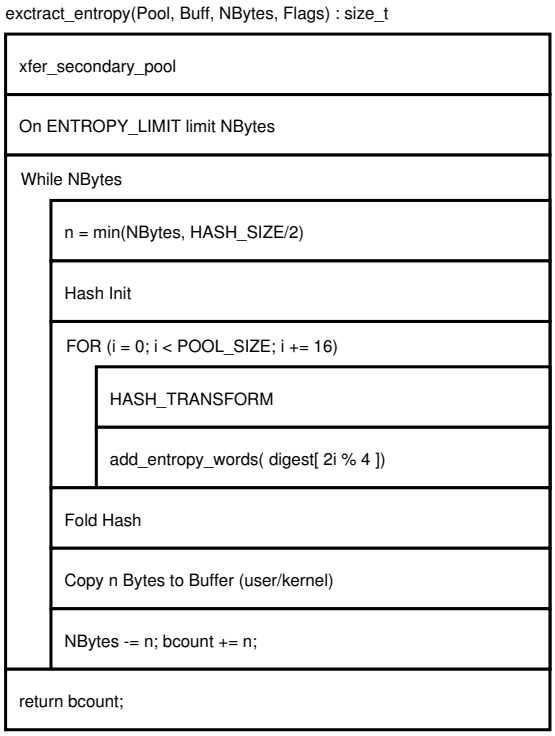

Abbildung 5.3: Routine zur Zufallszahlenerstellung extract_entropy

5.1.4 Zufallsgenerierung

Die Erstellung von Zufallszahlen geschieht im Wesentlichen in der Funktion extract_entropy (Abb. 5.3). Als erstes wird, falls nicht genug Entropie im sekundären Pool vorhanden ist, mittels der Funktion xfer_entropy_pool Entropie vom primären in den sekundären Pool geladen. Diese Routine wird immer ausgeführt. Sie beendet aber vorzeitig, wenn der Zielpool genug Entropie besitzt, um den Auftrag zu erfüllen. Somit wird nur dann ein Transfer von Entropie stattfinden, wenn der sekundäre Pool nicht genug Entropie enthält. Im zweiten Schritt wird anhand des vorrätigen Zufalls die Anzahl der Bytes bestimmt, die in der nachfolgenden Routine erstellt werden können. Anschließend wird der Entropiezähler des Pools entsprechend decrementiert und es beginnt die eigentliche Erzeugung.
Es werden immer $s = \frac{HASH\_SIZE}{2}$ zufällige Bytes erstellt, indem der Pool komplett gehascht wird. Jedesmal wenn ein solcher Block transformiert wurde, wird aus dem Resultat ein Word genommen und wieder, mittels der Funktion add_entropy_word, der aktuelle Pool manipuliert.
Anschließend wird das Resultat in einer XOR-Operation gefaltet. Dazu werden

beide „Hälften“ des Hash exklusiv miteinander verknüpft. Dieser neue Wert der Länge $s = \frac{HASH\_SIZE}{2}$ ist der endgültige Zufall, der dann in den Zielpuffer kopiert wird. Müssen weniger als s Bytes übertragen werden, so werden die restlichen Bytes aus dem Produkt ignoriert.
Für die Produktion des nächsten Werts wird der Pool wieder gehascht und gefaltet. An dieser Stelle sei jedoch ausdrücklich darauf hingewiesen, dass kein neuer Transfer von Entropie aus den primären in den sekundären Pool stattfindet.

5.1.5 „Guter“ und „schlechter“ Zufall

Sind beide Pools nicht ausreichend mit Zufall gefüllt, so führt dies im zweiten Schritt der Erzeugung von Zufall dazu, dass bei der Anforderung von „gutem“ Zufall die Anzahl der angeforderten Bytes nach unten korrigiert wird. Bei der Anfrage nach „schlechtem“ Zufall findet keine Korrektur statt, womit der Ausgang der Länge n-Bits durch einen Pool erzeugt wurde, der zuvor mit $m < n$ Bit in einem unvorhersehbare Zustand überführt wurde. Somit nicht zwingend vorausgesetzt, dass „guter“ Zufall aus dem primären Pool kommt.
Die Unterscheidung erfolgt über die Wahl der Geräteschnittstelle. Bei /dev/random erfolgt eine Korrektur und über /dev/urandom kann unbegrenzt ein Strom aus Pseudo-Bit-Zufallszahlen ausgelesen werden.

5.1.6 Einfluss der Länge de Pools

Die Pools besitzen im Normalfall eine Länge von 512 Wörtern. Bei der Erstellung von Zufall muss der komplette Pool gehascht werden. Dies ergibt 32 Iterationen. Zu beachten ist jedoch, dass bezüglich der Sicherheit die Größe des Pools keine Auswirkung auf den erstellten Hashwert hat. Sowohl MD5 als auch SHA1 haben, genau genommen, zwei Eingaben: den Eingabeblock und den Teil-Hash der bisherigen Transformationen. Diese zweite Eingabe ist der interne Zustand der Hashfunktion. Wenn die Erzeugung von mehreren Zufallszahlen beobachtet wird, so werden immer nur an einer bestimmten Stelle p_i und folgende des schnellen Pools $P = (p_0, p_1, \ldots, p_{i-1}, p_i, \ldots, p_{n-1})$ Änderungen eingefügt. Ein großer Teil des Pools, von p_0 bis p_{i-1}, wird nicht verändert und trägt dementsprechend auch nicht maßgeblich zur Erzeugung der neuen Zufallszahl bei, da dieser konstant ist und somit bis zu p_i der Hashwert identisch zu dem vorherigen Lauf ist. Erst ab dem Einbeziehen des veränderten Blocks verändert sich die Ausgabe. Deshalb ist es theoretisch nicht notwendig, diesen ersten Teil bei der Erstellung zu

berücksichtigen. Alternativ könnte der Pool auf einen einzigen Block verkleinert werden. Einzig die Länge des Zyklus würde darunter leiden. Der Zeitgewinn wäre jedoch nicht zu unterschätzen.

5.1.7 Pseudo-Bit-Ketten in der Zufallszahlenerzeugung

Eine unschöne Eigenschaft des Linux-Pools ist, es wird zwischen der Erstellung von zwei nacheinander folgenden Zufallszahlen keine neue Entropie in den Pool gespeist. Der Pool wird zu Beginn eines Auftrags mit Zufallsbit gefüllt. Anschließend folgt die Berechnung, ohne dass neuer Zufall einen Einfluss auf den Pool haben muss.
Die Hashfunktion H^* manipuliert den Pool P_0 bei der Erstellung einer Zufallszahl z_0, indem Zwischenergebnisse in den Pool gegeben werden. Das Resultat von H^* ist somit eine Zufallszahl und ein neuer Pool, der für die Erstellung der nächsten Zufallszahl z_1 verwendet wird.

$$(z_0, P_1) = H^*(P_0)$$

Alle Operationen sind deterministisch. Somit kann ein Angreifer, sollte dieser P_0 erraten können, die komplette Folge errechnen. Somit entspricht diese Erzeugung einem Pseudo-Bit-Zufallszahlengenerator, der jedesmal neu initialisiert wird.
Das Errechnen von vorangegangenen Zufallszahlen ist aber nicht ohne weiteres möglich. Dies setzt die Umkehrung der Hashfunktion H^* voraus.

5.1.8 Hardware-RNGs und Linux

Das Ziel des Zufallszahlenerzeugung unter Linux war es, ohne unterstützende Hardware eine Zufallszahlensequenz zu erstellen, die ausreichend sicher für die Anwendung starker kryptographischer Verfahren ist, ohne dabei spezielle Hardware zur Erzeugung von Zufall verwenden zu müssen.
In den letzten Zeiten ist der Zugang zur hardwareunterstützten Zufallszahlenerzeugung jedoch erschwinglicher geworden, womit zu rechnen ist, dass mittelfristig solche Geräte auch Einzug in den Massenmarkt erhalten werden. Dadurch stellt sich zwangsläufig die Frage, wo diese Eingabe der Zufallszahlen stattfinden soll.
Am wahrscheinlichsten ist die Tatsache, dass der Entropiepool an Bedeutung verliert, denn der Zufall könnte direkt von den Geräten bezogen werden. Dies würde die aufwendigen Berechnungen ersparen. Bei diesem Ansatz muss jedoch

vollständig dem Gerät vertraut werden. Ein weiteres Problem könnte die Menge des erzeugten Zufalls darstellen.
Alternativ könnte der Zufallszahlengenerator einfach in den primären Pool einspeisen. Der große Vorteil dieses Ansatzes ist, dass nun nicht mehr komplett dem Gerät vertraut werden muss, sondern es dominiert stets der zufälligste Wert. Diese Variante ist zudem sehr hilfreich, wenn nicht genug Zufall erzeugt werden kann, denn es können mehrere langsame Quellen dazu beitragen, Zufall in der benötigten Größenordnung zu generieren.
Für Linux ist die Geräteschnittstelle /dev/random und die Anwendung von IO-Control-Befehlen für die Einspeisung von zusätzlichen Quellen vorgesehen, was genau mit dem oberen Vorschlag übereinstimmt.

5.1.9 Dominanz einzelner Quellen

Bei der Einspeisung von Entropie wird diese solange aufgenommen, bis beide Pools gefüllt sind. Anschließend werden nur noch sehr wenige Eingaben aufgenommen. Eine Ausnahme bildet die Eingabe über die Geräteschnittstelle /dev/random. Besonders zur Boot-Zeit kann dies ein Problem darstellen. So sind die meisten Quellen noch nicht vorhanden, da ihre Treiber noch nicht aktiviert wurden und somit wird der Pool von wenigen Quellen gefüllt. Es ist hier deshalb von sehr großer Wichtigkeit, dass Linux, im Bootvorgang, sowohl einen „Seed“ in den Pool schreibt, wie auch Entropie ausliest[1]. Das gibt dem Pool die Möglichkeit, einen unvorhersehbaren Zustand ein- und Entropie von, hoffentlich bereits initialisierten, weiteren Quellen aufzunehmen.

5.1.10 Speicher- und Zeitbedarf

Der Linux-Pool wird in 2480 Zeilen implementiert. Diese Zeilen beinhalten aber weit mehr Funktionalität als für den eigentlichen Linux-Pool notwendig ist. Es werden noch Routinen zur sicheren Erzeugung von TCP/IP Sequenznummern und andere bereitgestellt.
Der Speicherbedarf des Linux-Pools wird maßgeblich durch die Größe der beiden Pools bestimmt. In der Standardausführung sind diese je 512 Wörter lang, was insgesamt 4096 Bytes entspricht.
Die zeitliche Betrachtung ist etwas aufwendiger: Für n Bytes werden $k = \lceil \frac{2n}{HASH\_SIZE} \rceil$ Durchläufe durch den sekundären Pool benötigt. Zusätzlich,im

[1] Unter Sarge Debian Linux geschieht das durch das Programm /etc/rcS.d/S55urandom

schlimmsten Fall,noch weitere 2k Durchläufe durch den primären. Bei jedem Durchlauf werden 32 Hashblöcke verarbeitet und 32 TGFSR-Operationen ausgeführt. Diese Operationen sind, in dieser Menge, zeitaufwendig.

5.1.11 Ein Schlusswort zum Linux-Pool

Anfänglich war es nicht sehr einfach die Beweggründe komplett zu verstehen. Der komplette Pool ist in einer einzigen Datei implementiert, worunter die Leserlichkeit leidet. Zudem sind die zum Teil komplexen Interaktionen nicht einfach nachzuvollziehen.
Zum Schluss muss jedoch eingestanden werden, dass sehr gute Überlegungen in den Entwurf eingeflossen sind. Für die Erfordernisse der normalen Anwender ist der Linux-Pool ausreichend.

5.2 Der Zufallsgenerator Yarrow-160

Yarrow ist ein von John Kelsey, Bruce Schneier und Niels Ferguson entworfener Pseudo-RNG [13]. Dabei wurde Yarrow auf die bestehenden Angriffe hin entworfen. Das Dokument nennt dabei folgende Angriffe:

1. Entropie-Überbewertung: Nach Angaben der Autoren bildet das die größte Gefahr.
2. Falsche Initialisierung oder unvorsichtiger Umgang mit Initialisierungsvektoren.
3. Fehler in der Implementierung.
4. Kryptoanalytische Angriffe.
5. Neben-Kanäle. Angriffe, die sich Eigenschaften der Algorithmen wie Stromkonsum und Zeitverbrauch zu Nutze machen.
6. Kontrolle über Zufallsquellen: Der Angreifer kann Zufallsquellen vorhersagen, abhören, steuern oder sogar ersetzen.

Nicht immer ist es möglich, sich gegen alle Angriffe zu schützen. Fehler können dabei folgende Konsequenzen haben:

1. Berechnen der Zukunft: Wenn einmal der interne Zustand bekannt geworden ist, kann der Angreifer alle Zufallszahlen errechnen, die in Zukunft noch erstellt werden (Beispiel: GFSR).

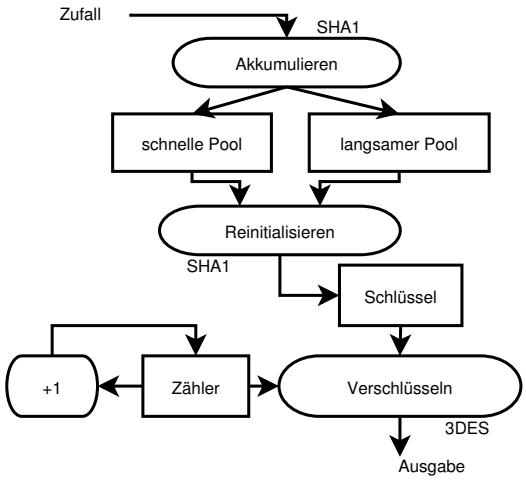

Abbildung 5.4: Yarrow-160 Entropie Pool

2. Iteratives Raten (Iterative Guessing Attacks): Wenn der Pool alle n Bit, m Bits Entropie hinzufügt und $m \ll n$, dann kann der Angreifer diese m Bit an eingeflossener Entropie erraten, beziehungsweise durchprobieren und mit der Ausgabe des Pools testen, ob seine Wahl korrekt war. Anschließend kennt der Angreifer alle Zahlen, die in diesem Zeitraum erstellt wurden.

3. Das Errechnen von vorhergegangenen Zahlen (zum Beispiel bei GFSR).

Der Yarrow-Generator(Abb. 5.4) besteht im wesentlichen aus zwei Teilen, dem der Zufallsaufbereitung und der Zufallszahlenerzeugung. Die Aufbereitung sammelt die eingehenden Zufallsströme und erstellt damit zwei Pools: den schnellen und den langsamen Pool. Aus beiden Pools wird ein Schlüssel erstellt, womit ein Zähler mit einer symmetrischen Operation verschlüsselt wird. Das Chifrat ergibt den Zufall.
Der schnelle Pool wird jedesmal aktualisiert, wenn neuer Zufall dazu gekommen ist. Der langsame Pool hingegen ist konservativer, was seine Aktualisierung angeht. Dieser sammelt sehr viel Entropie, weit über den 160 Bit, die ein SHA1-Hashwert aufnehmen kann und vorzugsweise aus mehreren Quellen.
Der langsame Pool wird auch nach einer gewissen Zeit ungültig, womit eine Erzeugung eines neuen Pools erzwungen wird. Dadurch sollen Auswirkungen eines bekannt gewordenen internen Zustands eingeschränkt werden. Sicherheitskritische Anwendungen sollen zudem den langsamen Pool vorzeitig invalidieren können.
Der Akkumulator sammelt den Zufall unter der Verwendung der Hashfunktion

SHA1. Ist genug Zufall vorhanden, so bildet der Hashwert den neuen langsamen oder schnellen Pool.
Der neue Schlüssel, wiederum, setzt sich aus dem alten Schlüssel, dem langsamen und dem schnellen Pool zusammen.
Der Zufall wird durch die symmetrische Verschlüsselung eines Zählers mit dem Schlüssel erstellt. Der Schlüsseltext ist der Zufall. Periodisch wird der Schlüssel durch die eigene Ausgabe ersetzt. In diesem Schritt betont das Papier, dass dadurch kein neuer Zufall hinzugefügt wurde.
Schneider verweist darauf, dass nur maximal 160 Bit Entropie in diesem SHA1-Hash vorhanden sein können. Zudem wird dieser Hashwert noch auf den Schlüssel abgebildet, der für die symmetrische Verschlüsselung verwendet wird (zum Beispiel verwendet DES nur 56 Bit). Da das Dokument 3DES vorschlägt, besteht an dieser Stelle keine Einschränkung.
Der Speicheraufwand ist, im Vergleich zu dem Linux-Pool, relativ gering. Mit einigen dutzenden Bytes ließe sich dieser Pool realisieren. Auch der zeitliche Aufwand sollte weit unterhalb des Linux-Pool liegen. Alleine die Reaktionsmöglichkeit auf „Bursts“ ist eingeschränkt. Der Vorrat an Entropie ist durch das Design auf 160 Bit limitiert.

5.3 Schlussfolgerungen

Im Vergleich besitzen beide Pools ihre Stärken und Schwächen. Yarrow hat den entschiedenen Nachteil, keinen Puffer zu haben, um größere Mengen Entropie vorrätig zu speichern. Zudem ist die Eingabe von Zufall mit einer Hashoperation verbunden. Ist der Pool nicht korrekt implementiert, so fällt die im Vergleich zur TGFSR-Operation viel aufwendigere Hash-Operation des Akkumulators zu lasten des einspeisenden Programms.
Auf der anderen Seite ist Yarrow kleiner, die Erzeugung schneller und die Tatsache, dass der langsame Pool konservativ bezüglich der Entropie ist, sicherer.
Zuletzt sei nochmals explizit erwähnt, dass der primäre Pool von Linux nicht dem langsamen Pool aus Yarrow-160 gleicht. Beide Pools verhalten sich unterschiedlich. Der Linux-Pool implementiert eher eine zweistufige Erzeugung, wobei das nicht notwendigerweise stimmt, da der sekundäre Pool ebenfalls mit Entropie direkt aufgefüllt wird. Ist im sekundären Pool genug Entropie enthalten, dann wird der primäre Pool ignoriert.

Kapitel 6

Der L4 Entropie-Pool

Anhand der gesammelten Erkenntnisse soll in Folge ein Pool für DROPS entworfen werden. Der neue Pool ist eine vollständige neue Entwicklung. Die Gründe dafür werden in Kürze erläutert. Einige wenige Codesegmente wurden aus dem Linux-Pool (Kernel 2.6.7) entnommen.

6.1 Warum eine neue Implementierung?

Dies hat im Wesentlichen folgende Gründe.

1. Linux macht keine Unterscheidung des Zufalls. Jeder Zufall wird gleich bewertet. Der L4-Pool soll zur Bestimmung der Güte des einfließenden Zufalls die Quelle mit berücksichtigen können. Dazu werden Entropieklassen eingeführt.

2. Der Aufbau des Linux-Pools besitzt keinen zentralen Eintrittspunkt für das Einschleusen von Entropie. Das Manipulieren des Pools und das Bestimmen, wie viel Zufall vorhanden sind, sind komplett voneinander getrennt. Der L4-Pool soll es jedoch ermöglichen, dass Kontrollinstanzen für den einfließenden Zufall eingebaut werden können.

3. Der Pool soll das von Bruce Schneier vorgeschlagene Konzept des langsamen und schnellen Pools realisieren. Dies ist vor allem auch wegen der Architektur wichtig. Systeme unter DROPS können sehr minimalistisch ausgelegt werden. Solche Annahmen, dass bei dem Bootprozess der Pool durch ein Programm in einen unvorhersehbaren Zustand überführt wird, können nicht gemacht werden.

4. Reine Pseudo-Bit-Zufallsketten sollen, wenn möglich, vermieden werden.

5. Es soll die Möglichkeit bestehen, den Pool sehr ressourcensparend zu verwenden, damit auch in eingebetteten Systeme eine mögliche Anwendung stattfinden kann.

Die Implementierung stützt sich im wesentlichen auf Yarrow-160 und Linux-Pool aus Kernel 2.6.7.

6.2 Designprinzipien

Folgende Eigenschaften sollen erfüllt sein:

1. Der Pool soll einfach zu verwenden sein. Auch Programmierer mit wenig Erfahrung mit der Kryptographie sollen in der Lage sein, den Pool fehlerfrei zu verwenden.

2. Die Auswirkungen der Überbewertung der eingespeisten Entropie, sowie vorhersagbare Eintrittspunkte des Pool sollten minimiert werden.

3. Initialisierung des Pools soll einfach sein.

4. Aus einem kompromittierten Entropie-Pool dürfen nicht zuvor erstellte Zufallszahlenfolgen errechnet werden können.

5. Ein kompromittierter Pool darf nur endliche Zeit vorhersagbar sein. Periodisch oder auf Anfrage soll eine neue Generierung des internen Zustandes erfolgen, wobei aus dem aktuellen Zustand keine Vorhersagen über den neuen abgeleitet werden können.

6. Der Pool sollte möglichst die Qualität des Zufalls nicht verschlechtern. Wenn eine Quelle eine gute Folge von Zufall erstellt, so sollte der Pool möglichst keine Pseudo-Bit-Zufallsfolge erstellen.

7. Der Pool soll Schnittstellen und Design-Prinzipien beinhalten, die mit der L4-Architektur vereinbar sind.

6.3 Übersicht

Der Pool soll in Umgebungen eingesetzt werden, wo es viel, wenig oder gar keinen Zufall gibt. Er besteht aus einem Akkumulator, einem schnellen Pool (Fast

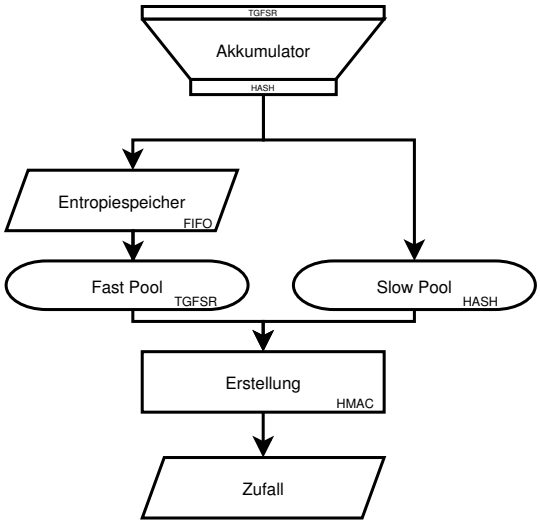

Abbildung 6.1: L4 Entropie Pool

Pool), einem langsamen Pool (Slow Pool) und einem Entropiespeicher. Die detaillierte Beschreibung der einzelnen Komponenten erfolgt in kürze. Abbildung 6.1 dient zur bessern Orientierung. Die eingespeiste Entropie wird zuerst im Akkumulator aufgearbeitet. Dieser Akkumulator erstellt einen Hashwert (MD5 oder SHA1), von dem ausgegangen wird, dass jedes Wort L4RANDOM_ENTROPY_QUEUE_THRESHOLD = 32 Bit Zufall enthält. In der Implementierung entspricht dies dem Wert 32. Dieses Resultat kann, je nach Bedarf, in den Entropiespeicher abgelegt oder als neuer langsamer Pool verwendet werden.
Die Erzeugung von Zufall erfolgt durch das Erstellen des Hashwerts über den schnellen Pool. Anschließend wird der produzierte Hashwert gefaltet und als Zufall herausgegeben. Die Hashwert-Erstellung selbst ähnelt einem HMAC [19]. Was das bedeutet, wird im Laufe der Beschreibung verdeutlicht. Dabei wird die Hashfunktion mit einem zusätzlichen Schlüssel verknüpft. Der Schlüssel für die Erstellung des Zufalls ist der langsame Pool. Somit ist die erstellte Zufallsfolge vom beiden Pools abhängig.

6.4 Entropieklassen

Der L4-Pool implementiert die Überlegung von B.Schneier, Quellen in unterschiedliche Klassen zu unterteilen[13]. Der Grund ist, dass die Quellen sich sehr voneinander in ihrem Zufallsgehalt unterscheiden können. So folgt die Eingabe über die Tastatur einem bestimmten Muster. Sowohl zeitliche Abstände wie auch die

Reihenfolge der Tasten sind nicht sehr zufällig. Ein auf Hardware basierender Zufallszahlengenerator, sei hier zum Vergleich genannt, der einen konstanten ausgeglichenen Strom erzeugt.
Einige Klassen sind EC_DISK, EC_KEYBOARD und EC_MOUSE, für Festplatten, Tastatur und Maus. In der Datei l4random.h werden alle Klassen definiert. Wie die Klassen EC_ANY, EC_SET und EC_EMPTY sich verhalten, wird im nächsten Abschnitt erläutert.

6.5 Der Akkumulator

Der Akkumulator besteht aus einem TGFSR, einem Hashwert und einem Entropie-Buchhaltungssystem. Die Eingabe wird zuerst in den TGFSR der Länge von 16 Wörtern geschrieben, der als Eingabevektor für die Hashfunktion dient. Statt eines gewöhnlichen Puffers zu verwenden, wurde hier die Entscheidung getroffen, einen TGFSR zu benutzen, weil dieser die Eingabe besser einmischt. Sobald der TGFSR einmal komplett durchgeschoben wurde, wird der TGFSR gehascht. Wenn genug Entropie eingeflossen wird, wird der TGFSR nochmals gehascht und der Hashwert wird in den Entropiespeicher gegeben oder als langsamer Pool verwendet.
Die Besonderheit des Akkumulators ist, dass komplexe Angaben über die Art des Zufalls gemacht werden können, die notwendig sind, um eine gute Ausgabe zu erzeugen. So kann dem Akkumulator mitgeteilt werden, dass die Ausgabe erst dann die angestrebte Güte besitzt, wenn mehrere Quellen mit unterschiedlicher Wichtung eingeflossen sind. Dazu wird dem Akkumulator ein Feld accData des Typs acc_policy_t[] mitgegeben. Jeder Eintrag in diesem Feld besitzt eine Klasse, einen Schwellwert und ein Konto(credits):

```
typedef struct {
  e_class_t    class;      // original class
  e_credits_t  threshold;
  e_credits_t  credits;
  e_class_t    _class;     // needed for EC_SET
} acc_policy_t;
```

Besonders erwähnenswert sind die Klassen EC_ANY und EC_SET. Erstere bedeutet, dass jede Klasse die Bedingung erfüllt; letztere bedeutet, dass bei dem ersten

Auftreten einer nicht eingetragenen Klasse das entsprechende Konto fest dieser neuen Klasse zugeordnet wird.

Beispiel 4 *Beispiel für einen langsamen Pool*

```
acc_policy_t accData = {
    { EC_KEYBOARD, 320 },
    { EC_DISK,     720 },
    { EC_SET,      160 },
    { EC_ANY,     1440 },
    { EC_EMPTY,      0 }
};
```

Dieser Pool benötigt mindestens drei unterschiedliche Quellen: EC_KEYBOARD, EC_DISK und einer zusätzlichen beliebigen Quelle, jedoch nicht einer der ersten beiden. Die zugehörigen Werte bedeuten die Menge an Entropie, die von den entsprechenden Klassen benötigt wird.

Der Folgende Algorithmus entscheidet dann, ob noch weiterer Zufall nötig ist:

Algorithmus 1 *Entropiezählung im Akkumulator*

1. *Die Eingabe besteht aus einem Feld* enropy *der Länge* cnt*. Der Zufall besitzt* credits *Zufallsbits und gehört der Entropieklasse* class *an. Informationen zur Buchhaltung werden in* accData *gehalten.*

2. *Füge den Zufall* entropy *dem TGFSR-Pool hinzu.*

3. *Überprüfe, ob die angegebene Klasse* class *in dem Feld* accData *vorhanden ist. Falls ja, dann überprüfe, ob noch Zufallsbits in dieser Klasse gebraucht werden. Wenn ja, dann addiere* credits *zu dem entsprechenden Konto. Wenn nicht, suche nach dem ersten Konto der Klasse* EC_ANY *und addiere dort* credits *dazu. Gehe zu 6.*

4. *Wenn nicht die Klasse, sondern mindestens ein Konto der Klasse* EC_SET *vorhanden ist, dann ordne dieses Konto der aktuellen Klasse zu und addiere credits. Damit wird fortan diese Klasse in 1. behandelt. Gehe zu 6.*

5. *Wenn weder 3. noch 4. zutreffen, dann suche nach dem ersten Konto der Klasse* EC_ANY *und addiere* credits.

6. *Teste, ob das aktuelle Konto den Schwellwert in diesem Aufruf überschritten hat. Wenn ja, dann decrementiere den Zähler* unsatisfiedSources *für „unerfüllte“ Konten.*

7. *Wenn dieser Zähler 0 erreicht hat, gebe* ACC_READY *zurück, sonst* ACC_NEED_MORE_ENTROPY

Zurück zu dem obigen Beispiel: Wegen Schritt 5 kann das letzte Konto mit Entropie beliebiger Klassen aufgefüllt werden, und wegen 3. kann das auch Entropie der Klassen EC_KEYBORAD, EC_DISK oder der dritten Klasse sein. Dieses Verhalten ist deshalb Wünschenswert, damit die Aufbereitung der Entropie in Systemen mit langsamen Quellen nicht zu lange dauert.
Und aus Gründen der Implementierung muss das Feld accData immer mit dem Konto EC_EMPTY enden, damit eine variable Anzahl der Konten angegeben werden kann.
Ein einziger Akkumulator wird für die Aufbereitung der Eingaben für den langsamen, wie auch für den schnellen Pool, verwendet. Es ändert sich lediglich das übergebene Feld *accData*.
An dieser Stelle kann auch sehr deutlich gezeigt werden, wie ein konservativer langsamer Pool zu realisieren ist: Die erzeugte Ausgabe ist HASH_SIZE Wörter lang, was genau der Länge des Hashwerts entspricht. Dabei wird angenommen, das jedes erzeugte Wort L4RANDOM_ENTROPY_QUEUE_THRESHOLD Bit Entropie besitzt. Für die Erzeugung dieser Ausgabe waren mindestens 2640 Bits notwendig und zudem aus unterschiedlichen Quellen. Diese 2640 Bits werden auf 128 (bei MD5) Bits abgebildet. Der schnelle Pool hingegen besitzt in der aktuellen Implementierung nur das Konto EC_ANY mit einem Schwellwert von 160 Bit.
Der Akkumulator wird somit zum zentralen Steuerelement für den Pool. Die Überbewertung der Entropie kann dadurch korrigiert werden, dass die Schwellwerte angepasst werden.
Wenn der Angreifer $n-1$ Quellen kontrollieren, erraten oder vortäuschen kann, so kann er dennoch nichts über die erzeugten Zufallszahlen aussagen, da der langsame Pool genauso in die Erzeugung der Zahlen einfließt wie der schnelle Pool. Der langsame Pool beinhaltet jedoch soviel Entropie von der unabhängigen Quelle, dass der Angreifer mindestens die 160 Bit Entropie des langsamen Pools erraten muss.
Dem Wert in L4RANDOM_ENTROPY_QUEUE_THRESHOLD gebührt noch eine nähere Betrachtung. In der Implementierung wird davon ausgegangen, dass

das Wort der Länge 32 Bit ebenfalls 32Bit Entropie besitzt. Das ist wahrscheinlich falsch, da dieses Wort ein Produkt der Hashtransformation ist. Wegen den vorhandenen Kollisionen muss davon ausgegangen werden, dass weniger Zufall im Produkt vorhanden ist [13]. Da die Implementierung jedoch am Ende den Hash nochmals faltet, und somit nur die Hälfte an Bits heraus gibt, wird dieser Verlust hier vernachlässigt.

6.6 Der Entropiespeicher

Der Entropiespeicher ist ein einfacher FIFO als Ringpuffer realisiert. Er wird vom Akkumulator bei der Aufbereitung des Zufalls für den schnellen Pool gefüllt. Vor jeder Generierung von Zufallszahlen durch die Bibliothek werden mindestens zweimal so viele Wörter aus dem Speicher in den schnellen Pool eingetragen wie am Ende herausgegeben werden.
Die wichtigste Begründung für den Speicher ist die Entkoppelung des Eingangs- und des Ausgangsstroms. Sollten die Anfragen in „Bursts“ auftreten und im Durchschnitt geringer sein als die Eingaben, so werden Wartezeiten minimal gehalten.
Zusätzlich entkoppelt der Speicher die Entropie-Aufwertung von der Zufallszahlenerstellung. Beide bilden im Design zwei getrennte Module. Unter Linux sind Aufwertung, Speicherung und Erzeugen eine unzertrennliche Einheit.
Der Nachteil ist natürlich, wenn der interne Zustand des Entropiepools bekannt geworden ist,dann führt eine lange Schlange dazu, dass ein Angreifer eine größere Menge von Zufallszahlen errechnen kann. Ein vorzeitiges Invalidieren des langsamen Pools schränkt dies jedoch wieder ein.

6.7 Der langsame Pool

Wie schon zuvor beschrieben, trägt der langsame Pool maßgeblich zur Sicherheit bei. In der aktuellen Implementierung ist dieser folgendermaßen definiert:

```
typedef union {
  u_int32_t pool[HASH_BLK_SIZE];
  struct {
    size_t reKeyIn;
    unsigned int  digest[HASH_SIZE];
    acc_policy_t * accPolicy;
```

```
    time_t lastReKey;
  } ctrlData;
} slow_pool_t;
```

Der Hashwert wird über der Variable *pool* gebildet, die sich mit *crtlData* den gleichen Speicherplatz teilt. In *digest* kommt die Ausgabe des Akkumulators und in *lastReKey* die Zeit, wann der langsame Pool erzeugt wurde. Zusätzlich wird in $reKeyIn$ noch die Anzahl der Bytes eingetragen, für die der langsame Pool noch gültig ist. Nach ungefähr $reKeyIn$ Bytes wird der langsame Pool ungültig und ein neuer langsamer Pool muss erstellt werden. Bei einem „ReKeyRequest“ wird dieser Wert ebenfalls auf 0 gesetzt, was die vorzeitige Invalidierung des langsamen Pools zur Folge hat. Dadurch, dass dieser Wert nach jeder Anforderung von Zufall dekrementiert wird, ist für zwei aufeinander folgende Anforderungen der langsame Pool niemals gleich. Die zeitliche Beschränkung des Pools hat den Vorteil, dass es einen Angreifer beschränkt, sollte dieser den internen Zustand des Pool in Erfahrung gebracht haben.

6.8 Der schnelle Pool

```
typedef struct {
  u_int32_t pool[L4RANDOM_FAST_POOL_SIZE];
  tgfsr_t tgfsr;
  acc_policy_t * accPolicy;
} fast_pool_t;
```

Der schnelle Pool ist ein einfaches TGFSR. In der zugrunde liegenden Implementierung wird ein Polynom des Grades 32 mit 5 Koeffizienten gewählt, deren Abstand RMS-Normal verteilt ist. Dieses Polynom ist somit nicht mehr koeffizientenarm (Sparse), und sollte somit für eine gute Sequenz ausreichend sein. Die Tatsache, dass die Koeffizienten nahezu äquidistant zueinander liegen, hat die Auswirkung, dass die Veränderung des Zustandes gleichmäßig vom ganzen Vektorraum abhängt.
Die Polynome des Grads 32 oder höher wurden aus der Implementierung von Linux genommen und nicht weiter auf ihre Primitivität geprüft.
Der TGFSR sorgt dafür, dass wenn der Pool ohne externe Eingaben betrieben wird, funktioniert der Generator dennoch als guter Pseudo-Bit-Zufallszahlengenerator.

Anders als unter Linux ist der TGFSR nur 32 Wörter lang. Unter Linux sind es 512. Der Vorteil sind die Einsparungen von Speicher und die Tatsache, dass nur noch zwei Iterationen der Hashfunktion durchzuführen sind, was im Gegensatz zu den mindestens 32 Durchläufen unter Linux steht.
Dafür besitzt der TGFSR unter der L4-Implementierung einen weitaus geringeren Zyklus. Jedoch sollte die Zykluslänge von $2^{1024} - 1$ ausreichend sein.
Aus diesen Gründen wurde für L4 ein kürzerer und schnellerer Pool gewählt.

6.9 Die Zufallszahlengenerierung

Die Zufallszahlen werden in der Funktion l4RandomGetRandomBytes(Abb. 6.2) erstellt. Die wesentlichen Unterschiede zu der Implemetierung unter Linux sind grau hervorgehoben. Als erstes wird überprüft, ob ein langsamer Pool vorhanden ist. Sollte kein gültiger Pool vorhanden sein, so bricht die Erzeugung sofort ab. Diese Abfrage kann mittels der Übergabe von L4RANDOM_IGN_SLOW_POOL überbrückt werden. Ist ein Pool vorhanden, so wird die Hashfunktion initialisiert und der langsame Pool gehascht[1].
Anschließend werden in einer Schleife je HASH_SIZE/2 zufällige Wörter erzeugt. In jedem Schleifendurchlauf werden HASH_SIZE Wörter aus dem Entropiespeicher in den schnellen Pool geschrieben. Der schnelle Pool wird gehascht und produziert einen Hashwert, dessen erste Hälfte mit der zweiten Hälfte mittels XOR verknüpft wird (Faltung). Das Resultat wird als Zufall in den Ausgabepuffer geschrieben und die Schleife wird erneut ausgeführt, wobei die Hashfunktion nicht neu initialisiert wird.
Sollte kein Zufall mehr im Speicher vorhanden sein, so bricht die Funktion vorzeitig ab. Der Rückgabewert gibt über die Anzahl der erzeugten Zufallsbytes Auskunft. Bei der Angabe von L4RANDOM_FAST wird auf ein Transfer von Entropie aus dem Speicher in den schnellen Pool komplett verzichtet. In diesem Fall ist auch eine Erzeugung ohne Entropie möglich. Durch den fehlenden Zufall im Eingabevektor der Hash-Funktion handelt es sich bei der Ausgabe um eine reine Pseudo-Bit-Zufallsfolge.

[1] An diese Stelle wird deutlich, warum die Erstellung von Zufall einem HMAC[19] ähnelt. Der langsame Pool ist äquivalent zu dem unter HMAC verwendeten Schlüssel.

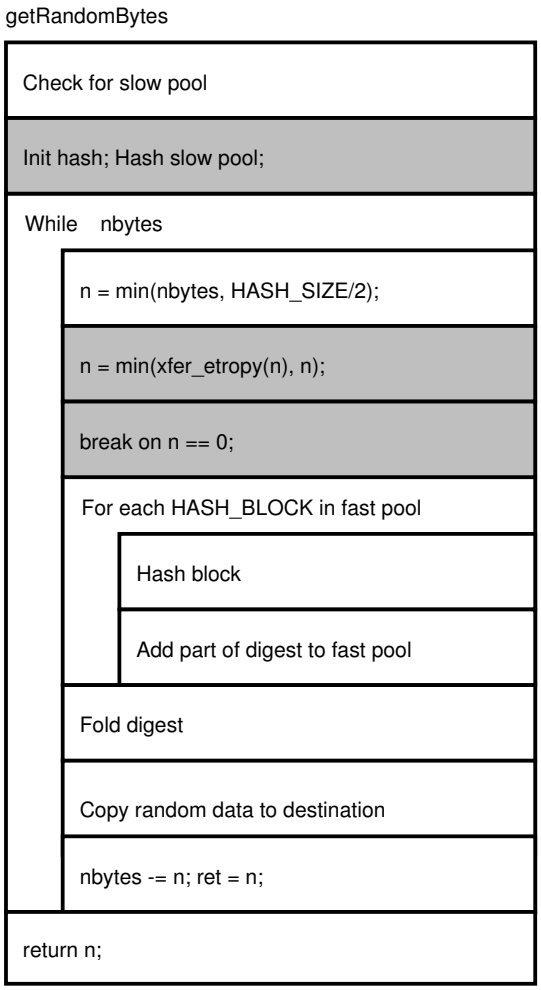

Abbildung 6.2: Routine zur Zufallszahlenerstllung l4RandomGetRandomBytes

6.10 Die Entropieaufnahme

Der Pool stellt eine zentrale Schnittstelle für die Eingabe von Entropie zur Verfügung: l4RandomAddRawEntropy. Diese Funktion implementiert dabei folgend Algorithmus:

Algorithmus 2 *Die Entropieaufnahme*

1. *Überprüfe, ob Zufall benötigt wird. Wenn nicht, dann verlasse die Funktion und gebe L4RANDOM_OK zurück.*

2. *Überprüfe, welchen Regeln der Akkumulator verfolgt. Ist momentan der schnelle Pool aktiv und der langsame Pool ungültig, so setze den langsamen Pool als Ziel. Gehe zu 4.*

3. *Ist der langsame Pool fast leer (< L4RANDOM_SLOW_POOL_LIMIT), der schnelle jedoch noch ausreichend gefüllt (> L4RANDOM_FAST_POOL_-LIMIT), so setze den langsamen Pool ebenfalls als Ziel.*

4. *Füge den Zufall hinzu.*

5. *Wenn die hinzugefügte Entropie ausreichend war, um die Akkumulierung erfolgreich zu beenden, dann installiere den Zufall. Wenn das Ziel der schnelle Pool ist, dann wird das Produkt in den Entropiespeicher geschrieben. Sonst wird der langsame Pool erneuert und das Ziel wieder auf den schnellen Pool gesetzt. In beiden Fällen wird der Akkumulator anschließend zurückgesetzt.*

6. *Sollte immer noch Entropie benötigt werden, gebe L4RANDOM_-NEED_ENTROPY oder L4RANDOM_NO_SLOWPOOL zurück. Sonst L4-RANDOM_OK.*

Mit den Schwellwerten unter 3. soll erreicht werden, dass sich die Erneuerung des langsamen Pools nicht negativ auf das System auswirkt. Die genauen Werte sind noch nicht korrekt ermittelt. Das Ziel ist, diese Werte so zu wählen, das die Erstellung eines neuen langsamen Pools zum spätmöglichsten Zeitpunkt beginnt, so dass zu keinem Zeitpunkt das System wegen eines ungültigen langsamen Pools die Erzeugung verweigern muss. Dies gilt natürlich nur für den normalen Betrieb und nicht für den Fall, dass eine vorzeitige Invalidierung des Pools stattgefunden hat.

6.11 Resourcen Bewertung

Der Pool besteht aus ca. 1700 Zeilen Code. Der Speicherbedarf beträgt ungefähr 1k Byte, wobei der Entropiespeicher, der in seiner Größe veränderbar ist, 512 Bytes belegt. Der langsame Pool wurde mit 84 und der schnelle mit 128 Bytes implementiert. Für Systeme mit weniger Speicher lässt sich hier noch ca. die Hälfte der Bytes einsparen, ohne dabei Veränderungen am Design vornehmen zu müssen.
Im zeitlichen Vergleich sind zwei Fälle zu unterscheiden: Bei der Eingabe der Entropie wird unter Linux eine einfache TGFSR-Operation durchgeführt. Unter L4Random muss hin und wieder der Eingabevektor durch die Hashfunktion verarbeitet werden. Dies ist aufwendiger. Mit einer geschickten Programmierung des L4Random-Servers kann dennoch erreicht werden, dass dieser Mehraufwand nicht zu lasten des einspeisenden Programms geht.
Bei der Erstellung von Zufall ist der L4-Pool klar im Vorteil, da sehr viel weniger Operationen ausgeführt werden. Für n Bytes werden ebenfalls $k = \lceil 2n/HASH\_SIZE \rceil$ Durchläufe benötigt. Bei jedem Durchlauf werden aber nur zwei Hashblöcke verarbeitet und zwei TGFSR-Operationen ausgeführt. Zusätzlich werden noch $2n$ TGFSR Operationen für die Einspeisung von Zufall in den Pool benötigt und eine einzige Ausführung der Hashfunktion für die Anwendung des langsamen Pools. Vor allem der „kurze“, schnelle Pool sorgt für einen Geschwindigkeitsgewinn.

6.12 Auswahl der Parameter

Anhand der zu Beginn geführten Analysen über den Eingangsstrom unter Linux wird für den schnellen Pool folgende Politik gewählt:

```
acc_policy_t FastPoolAccPolicy = {
    { EC_ANY,   160 }
    { EC_EMPTY,   0 }
}
```

Die soll eine möglichst schnelle Erzeugung von Zufall im normalem Betrieb ermöglichen. Der langsame Pool sollte in jedem Falle die Probleme der Überbewertung der Quellen entgegenwirken. Dazu werden die Abschätzungen auf Kapitel 3 berücksichtigt. Damit langsame Quellen nicht dazu führen, dass der Pool blockiert und die Tatsache, dass nicht vorweg gesagt werden kann, ob das System

eine Maus oder sonstiges besitzt, beschränkt sich die Angabe nur darauf, festzulegen, dass mindestens drei Quellen vorhanden sein müssen. Die 160 Bit sollten auch durch langsame Quellen zu erbringen sein. Anschließend können die schnellen Quellen die restlichen 1440 Bit auffüllen.

```
acc_policy_t SlowPoolAccPolicy = {
    { EC_SET,    160 },
    { EC_SET,    160 },
    { EC_SET,    160 },
    { EC_ANY,   1440 },
    { EC_EMPTY,    0 }
}
```

6.13 Statistische Tests

Der Pool wurde durch wenige statistische Tests geprüft. Diese Tests waren alle erfolgreich. Aber die statistischen Auswertungen der Zufallszahlenfolge können nach Anwendung der Hashfunktion höchstens grobe Programmierfehler offenbaren. Dies begründet die Tatsache, warum nur einige wenige Tests durchgeführt wurden. Anhang B führt einige Graphiken und Erläuterungen dazu.
Wie angedeutet, müssen diese Ergebnisse mit Vorsicht betrachtet werden. Zwar suggerieren sie, dass ein Angreifer diese Folge nicht von reinem Zufall unterscheiden könnte, doch die Folge, die mit

```
z = (char)HASH( (int)j++ );
```

erzeugt wird, tut dies ebenfalls. Dieser Generator ist aber auf jeden Fall nicht sicher. Sehr viel maßgeblicher für die Sicherheit des Entropiepools (und auch alle weiteren Pools, die hier angegeben wurden) ist die Verhinderung von Fehlern in der Implementierung, die korrekte Einschätzung der Entropie und die Auswahl der Quellen.
Eine weitere wichtige Betrachtung ist die Ausgabe des schnellen Pools. Sie ist maßgeblich für den anschließend erzeugten Hashwert. [9, 10] behauptet schon die Eignung des TGFSR als Generator und stützt sich dabei auf statistische Tests.

6.14 Mögliche Veränderungen

Die Implementierung besitzt mehrere Möglichkeiten, den Speicherbedarf und zum Teil den Zeitbedarf zu steuern. Diese Regler folgen bestimmten Rahmenbedingungen, die im Quelltext beschrieben werden. Die angewendeten Algorithmen sind ausschlaggebend für die Werte, die diese Regler annehmen können. So kann zum Beispiel der Speicherbedarf des Akkumulators nicht unter der Blockgröße der Hashfunktion fallen.
Abhilfe würde hier das Ersetzen der Algorithmen schaffen. Andere Einwegfunktionen oder sogar symmetrische Verschlüsselungsalgorithmen mit weniger Platzbedarf könnten zum Einsatz kommen. Beispielsweise könnte die Hashfunktion durch den AES-Algorithmus ausgetauscht werden. Die Blockgröße von 128 Bit hätte eine Reduzierung um ca. den vierfachen Wert des minimalen Speicherplatzbedarfs des schnellen Pools und der TGFSR der Akkumulators. Die Akkumulatoren könnten die Entropie zu einem Schlüssel sammeln und einen festen Wert verschlüsseln. Bei der Erstellung der Zufallszahlen bildet ein Pool den Schlüssel, der andere den Klartext.
Solche Ersetzungen erfordern jedoch eine gründliche Planung. Tiefer gehende Betrachtungen wurden jedoch nicht ausgeführt.
Außer den schon angesprochen Speichereinsparungen bedeutet dies, dass für den Fall, dass wenn die Sicherheit eines Algorithmus nicht mehr gegeben ist, der Pool, unter Wiederverwendung des größten Teils des Quelltextes den neuen Gegebenheiten angepasst werden kann.

Kapitel 7

Einbindung in L4

7.1 Die Bibliothek

Die gesamte Funktionalität des Pools ist in einer Bibliothek, l4randomlib, zusammengefasst. Die verwendete Programmiersprache ist C. Diese Bibliothek besitzt die Funktionen für die Manipulation des TGFSR, des Akkumulators, des Entropiespeichers, die Hashfunktion und den Pool an sich. Die wesentlichen Dateien sind tgfsr.[ch], accumulator.[ch], hash.[ch], buffer.h und l4random.[ch]. Die Dokumentation ist in den Dateien eingebettet und zusammen besteht der Pool aus ungefähr 1700 Zeilen Code. Diese Zeilen enthalten zudem Test-Routinen, Routinen zur Sammlung statistischer Daten und „Debug“-Zeilen, die allesamt entfernt werden können, ohne dabei den Pool zu beeinträchtigen.
Bei der Implementierung wurde keine besondere Rücksicht auf Portabilität genommen. Jedoch sollte dies nicht weiter problematisch sein, da keine architekturspeziefische Codesegmente eingefügt wurden. Zudem wurde Wert auf die Verwendung von Typen mit fester Bitbreite gelegt, wo es sonst zu Problemen kommen könnte. Die Bibliothek greift auf nur zwei externe Funktionen zu: printf(3) und memset(3). Erstere ist nur dann notwendig, wenn die Debug-Ausgabe aktiviert ist. Letztere kann durch eine eigene Implementierung unter missing.c ersetzt werden. Speicherallokationen finden nicht statt. Der Bibliothek ist auch ein Testprogramm beigelegt, welches die Anwendung verdeutlicht. Die folgende Dokumentation soll nur eine grobe Übersicht vermitteln. Für eine detaillierte Beschreibung sei auf die Dokumentation im Quelltext verwiesen.
Zusätzlich ist noch dringlichst darauf hinzuweisen, dass die Bibliothek noch nicht gegen eventuelle konkurrierende Zugriffe (Race Conditions) geschützt ist. Der

Quelltext enthält zwar Kommentare, wo solche Wettlaufsituationen auftreten können, aber die entsprechenden Auschlussverfahren wurden nicht implementiert. Die unterschiedlichen Parameter lassen sich in den Dateien l4random.[ch] einstellen. Lediglich die Wahl der Hashfunktion muss in der Datei hash.h erfolgen. Es besteht die Auswahl zwischen MD5 und SHA1.
Die Schnittstelle des Pools wird aus folgenden Funktionen gebildet:

```
l4random_status_e l4RandomInitPool(void);
void   l4DestroyPool(void);
```

Diese beiden Funktionen initialisieren und zerstören den Pool. Bei der Zerstörung wird darauf geachtet, dass anschließend keine kritischen Informationen mehr im Speicher abgelegt sind. Der Aufruf von l4RandomInitPool ist obligatorisch. Bei der Initialisierung ist das Ausführen von Testroutinen vorgesehen. Schlägt eine Testroutine fehl, so wird der Wert L4RANDOM_SELF_TEST_FAILED zurück gegeben.

```
l4random_status_e l4RandomAddRawEntropy(
  size_t bcount,
  const unsigned int * words,
  int class,
  int credits);
```

Diese Funktion speist *bcount* Wörter Zufall der Klasse *class* in den Pool ein. Dem gesamten Feld wird eine Entropie von insgesamt *credits* zugeordnet. Der Rückgabewert ist L4RANDOM_OK, wenn kein weiterer Zufall benötigt wird. Bei L4RANDOM_NO_SLOWPOOL und L4RANDOM_NEED_ENTROPY muss der Pool weiterhin mit Zufall versorgt werden.

```
l4random_status_e l4RandomSeedPool(
  size_t count,
  const unsigned int * words );
```

initialisiert den schnellen Pool, indem die Eingabe direkt in den TGFSR geschrieben wird. Es ist empfohlen, einen Initialisierungsvektor der Länge L4RANDOM_FAST_POOL_SIZE Wörter zu verwenden.

```
size_t l4RandomGetRandomBytes(size_t bcount,
  unsigned char * buff,
  int flags);
```

erzeugt maximal *bcount* zufällige Bytes und legt sie in *buff* ab. Sollte kein langsamer Pool oder nicht genug Entropie vorhanden sein, so bricht die Routine vorzeitig ab. Es wird immer die Anzahl der erzeugten Bytes zurück gegeben. Ist in *flags* L4RANDOM_FAST angegeben, so wird kein Zufall für die Erzeugung verwendet. Mit L4RANDOM_IGN_SLOW_POOL ist eine Erzeugung auch ohne gültigen langsamen Pool möglich.

```
l4random_status_e l4RandomReKey(void);
```

erzwingt die vorzeitige Invalidierung des aktuellen langsamen Pools. Die Erzeugung neuer Zufallszahlen ist bis zum Ersatz des langsamen Pools nicht mehr möglich. Die Funktion beendet sofort. Vor allem wartet sie nicht darauf, dass ein neuer Pool erzeugt wird.
Als Rückgabewerte sind L4RANDOM_OK für Erfolg und L4RANDOM_REKEY_DENIED vorgesehen. Letzteres muss vom Zufallszahlenserver implementiert werden. Da es zudem sinnvoll ist, nicht jede Aufforderung zu gewähren, sollte der Server eine Überprüfung durchführen. Jedoch verfügt der aktuelle Server nicht über eine solche Funktionalität.

```
l4random_status_e l4RandomStatus(
  l4random_status_t * dst );
```

füllt *dst* mit einigen wichtigen Informationen zum Pool aus, wie enthaltene Entropie, Status und andere.

```
int l4RandomRand(void);
```

ist als Ersatz für rand(3) gedacht. Die erzeugten Zahlen werden mit L4RANDOM_FAST und L4RANDOM_IGN_SLOW_POOL erzeugt. Aus diesem Grund ist diese Funktion für kryptographische Anwendungen unbrauchbar. Sie ist dennoch sicherer als rand(3).

7.2 Der Server

Bei dem Server handelt es sich um eine sehr einfache Implementierung, in der nur die nötigsten Routinen eingebaut wurden. Die IPC-Schnittstelle wird in IDL in

der Datei l4random.idl beschrieben. Die Umsetzung der Schnittstellenspezifikation erfolgte mit Dice[22]. Eine Bibliothek l4random abstrahiert den Zugriff auf den Server soweit, dass keine Unterscheidungen zwischen dem direkten Anwenden der l4randomlib-Bibliothek und dem Zugriff über den Server für den Programmierer ersichtlich sind. Der Code sieht in beiden Fällen identisch aus. Entscheidet sich der Programmierer den zentralen Pool über den Server zu verwenden, so „linkt“ dieser den Pool gegen l4random. Möchte man eine eigene Instanz des Pools verwenden, so muss die Bibliothek l4randomlib angegeben werden.

Kapitel 8

Unbeantwortete Fragen

Dieser Beleg kommt zum seinem Schluss, ohne einige wichtige Detailfragen wirklich beantwortet zu haben, die sich im Laufe der Ausarbeitung ergeben haben. Diese Antworten sind jedoch abhängig von weiterführenden Studien und auch von Erfahrungswerten, die nur in der praktischen Anwendung gesammelt werden können.

8.1 Entropiegewinn unter L4

Eine wichtige Frage ist, wie die Entropie unter L4 gewonnen werden soll. Der Ansatz, der unter Linux verwendet wird, an einer zentralen Stelle die Interrupts abzufangen, würde viel für die Sammlung von Zufall tun: Die Änderungen wären nur an einer einzigen Stelle im Kern notwendig. Die Binärkompatibilität wäre dadurch gewährleistet.
Es stellt sich nur die Frage, ob dieser Ansatz wünschenswert ist, wenn die Devise ist, den Kern von unnötigen Routinen frei zu halten. Außerdem sollte man sich fragen, ob dieses System selbst nicht neue Sicherheitsbedenken hervorruft? Wenn ein Programm von allen Unterbrechungen informiert wird, so kann nicht mehr von einer strikten Trennung zwischen den Applikationen gesprochen werden.
Ein weiterer Ansatz ist, in Dice eine Option einzufügen, die dafür sorgt, dass dem Versenden von Nachrichten transparent eine Weiterleitung als Ganzes oder partiell an den Entropiepool geht. Somit könnten die Programmierer Zufall beisteuern, ohne sich mit dem Pool auseinander gesetzt zu haben. Aber auch hier sind die obigen Befürchtungen zum Ausdruck zu bringen.
Was wohl unumgänglich bleiben wird, ist das Erstellen von Programmen, die

RNGs in Hardware auslesen und den Zufall einspeisen.
Zusätzlich sollte mittelfristig das L4/Linux-Modul random.c so umgebaut werden, dass die Existenz eines Zufallszahlenserver überprüft und dieser dann verwendet wird. Auch hierfür wurden im Rahmen des Belegs erste Ansätze unternommen. Eine Portierung des Pools auf Linux wurde eingeleitet: Der Quelltext enthält die nötigen Anpassungen, um das Bauen des Pools in der Linux-Kernel-Umgebung zu erlauben.

8.2 Quellen sicher bestimmen

In den meisten Fällen wird der Zufall durch andere Programme eingegeben. Aber wie soll der Server zuverlässig entscheiden können, welches Programm „gut“ oder „böse“ ist. Momentan „glaubt“ der Pool alles, was ihm vorgegeben wird. Es müssen Mechanismen eingebaut werden, um die Quelle zuverlässig zu bestimmen. Secure-Booting könnte die Lösung für dieses Problem sein. Damit könnten Module, die als glaubwürdig eingestuft wurden, vom Kernel bestätigt werden. Der Zufallszahlenserver könnte mit Hilfe dieser Authentifikation eine Klassifizierung vornehmen.

8.3 Bestimmen der Entropie

Zudem muss noch die Überlegung angestellt werden, wer letztendlich den Entropiegehalt einer Probe bestimmt. Sind es die Applikationen selbst oder macht dies der Server, und nach welchen Methoden soll dies geschehen? Werden aufwendige statistische Analysen durchgeführt oder erfolgt die Bestimmung unter Einbeziehung des sendenden Programms?

8.4 Vertrauen in die Hardware

In wie fern können neue Technologien die Umsetzung der Sicherheit verhindern? Direct-Memory-Access(DMA) erlaubt den direkt Zugriff auf den Speicher und somit die Umgehung der Isolation durch die Adressräume. Über eine Sicherheitslücke im Festplattentreiber könnte der Pool manipuliert werden.
Wie sehr können Fehler in der Hardware und neue Technologien die Sicherheit beeinflussen? Die Buglisten der Mikroprozessoren sind lang. Jedes „Stepping“ korrigiert Fehler und deckt neue auf.
Jedoch stehen solche Fragen nicht direkt im Zusammenhang mit dem Pool. Viel-

mehr sind es grundsätzliche Probleme, die bei der Anwendung der Mikro-Kernel-Architektur auftreten.

Kapitel 9

Schlusswort

Abschließend kann gesagt werden, dass das Thema der Zufallszahlenerstellung unter Rechnersystemen noch nicht abgeschlossen ist. Auch bei monolithischen Systemen sind noch einige Fragen offen. Mikro-Kernel-Systeme stellen aber eine andere Umgebung dar. Annahmen, die für ihre großen „Geschwister“ gelten, dürfen nicht ohne weiteres übernommen werden. Die einfache Portierung des Linux-Pools wäre unangebracht gewesen.
Die kryptographischen Grundlagen bleiben auf beiden Systemen gleich. Aber die Umsetzung dieser Vorgaben muss in beiden Fällen anders realisiert werden.
Bis für Mikro-Kerne ein guter Zufallszahlenserver vorhanden ist, muss noch viel Arbeit getätigt werden. Jedoch spricht nichts dagegen. Dieser Beleg zeigt einen Ansatz und der vorherige Abschnitt erwähnt die bestehenden größten Schwierigkeiten und führt auch Lösungsansätze auf. Zudem kann der existierende Pool mit mittlerem Aufwand an zwei bis drei Quellen direkt gekoppelt werden, um als Notlösung für ein spezielles Anwendungsgebiet eingesetzt zu werden.

Anhang A

Zufallsquellen unter Linux

Im Anschluss befinden sich 4 Abblidungen, die eine genauere Betrachtung der Quellen unter Linux erlauben. Abbildung A.1 verdeutlicht die Häufigkeiten der unterschiedlichen Quellen. Maus und Festplatte tragen am häufigsten zum Entropiegewinn bei. Unter Linux wird jeder Probe, die in den Pool eingespeist wird, eine Entropie zugeordnet. Das Histogramm dieser Werte ist in Abbildung A.2 dargestellt. Die beiden senkrechten Linien zeigen das Mittel dieser Werte (rechts) und das Mittel der Approximierungsfuktion, wie sie in Kapitel 3 vorgestellt wurde (links).
Die Approximation ist eine sehr einfache Berrechnung mit Anwendung der Hemmingdistanz. Sie ist eine obere Grenze, da bessere Approxiamtionen noch weitere Korrelationsfaktoren berücksichtigen würden. Ist dieser Wert überschritten, wie es in diesem Beispiel geschieht, so findet höchswahrscheinlich eine starke Überbewertung der realen Entropie statt.
Den realen Entropiegehalt zu ermitteln, ist nahezu unmöglich. Dies erfordert genauste Kenntnisse über die Quellen, die Umgebung (das Betriebssystem) und die Nebenerscheinungen, die auftreten können. Für die Ausführung dieses Belegs sollen die Ergebnisse des Unixprogramms `ent(1)` [24] herangezogen werden, das statistisch einen Entropiegehalt ermittelt. Die Ausgabe ist unter Abbildung A zu sehen. Ganz besonders ist der Wert Entropie = 4.453733, und die Tatsache, dass dieser kleiner ist als die Hälfte der von Linux zugeordneten durchschnittlichen Entropie von $9,662$. Das hat eine sehr schwerwiegende Folge: Bei der Erstellung des Zufalls wird durch den Erstellungprozess am Ende eine impliziete Halbierung der eingespeisten Entropie durchgeführt. Das heißt, es wird doppelt soviel Entropie benötigt wie Zufall erstellt wird. `ent(1)` verdeutlicht jedoch, dass die

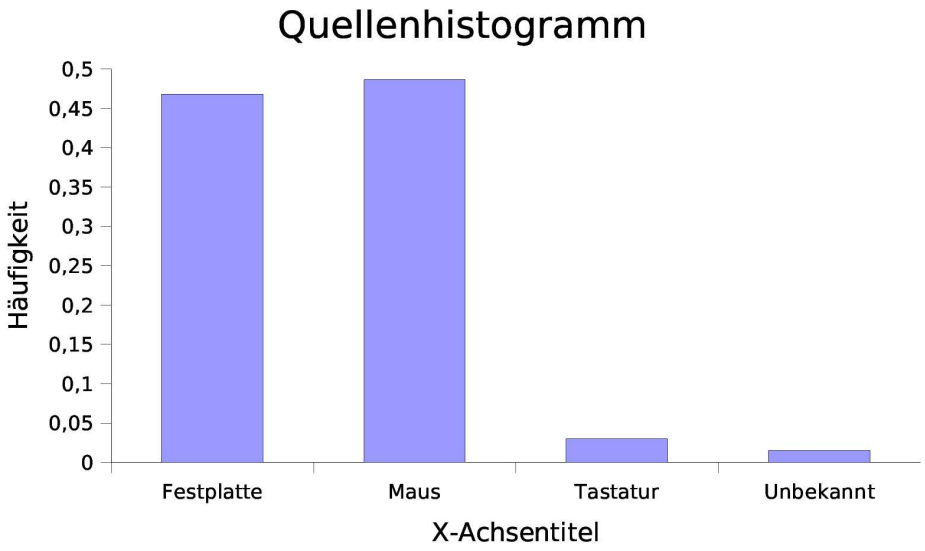

Abbildung A.1: Histogramm der Quellen unter Linux

Überbewertung so dermaßen hoch ist, dass auch diese Halbierung am Ende nicht ausreichend ist.
Zuletzt wird in Abblidung A.4 die die Verteilungsfunktion der Proben gezeigt. Sie gibt Auskunft darüber, dass die Proben sich wesentlich in zwei Wertebereichen aufhalten, die den beiden nahezu senkrechten Anstiegen entsprechen. Anhand dieser Tatsache kann verdeutlicht werden, dass eine Aufarbeitung dieses Zufalls notwendig ist.

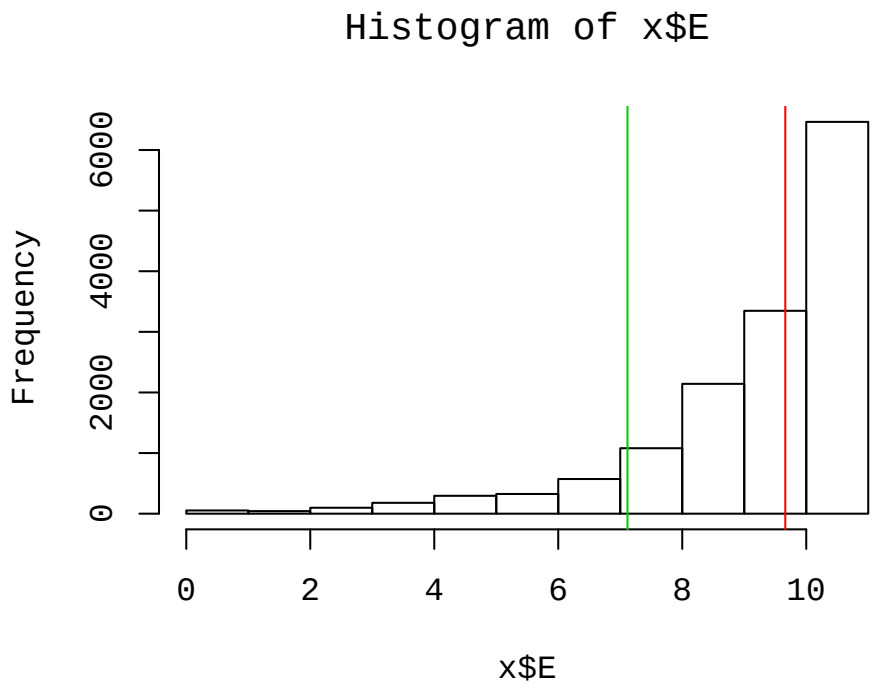

Abbildung A.2: Entropie Histogramm

```
Entropy = 4.453733 bits per byte.

Optimum compression would reduce the size
of this 58392 byte file by 44 percent.

Chi square distribution for 58392 samples is 2463657.39, and randomly
would exceed this value 0.01 percent of the times.

Arithmetic mean value of data bytes is 47.2379 (127.5 = random).
Monte Carlo value for Pi is 3.649815043 (error 16.18 percent).
Serial correlation coefficient is 0.328397 (totally uncorrelated = 0.0).
```

Abbildung A.3: Ausgabe von ent(1)

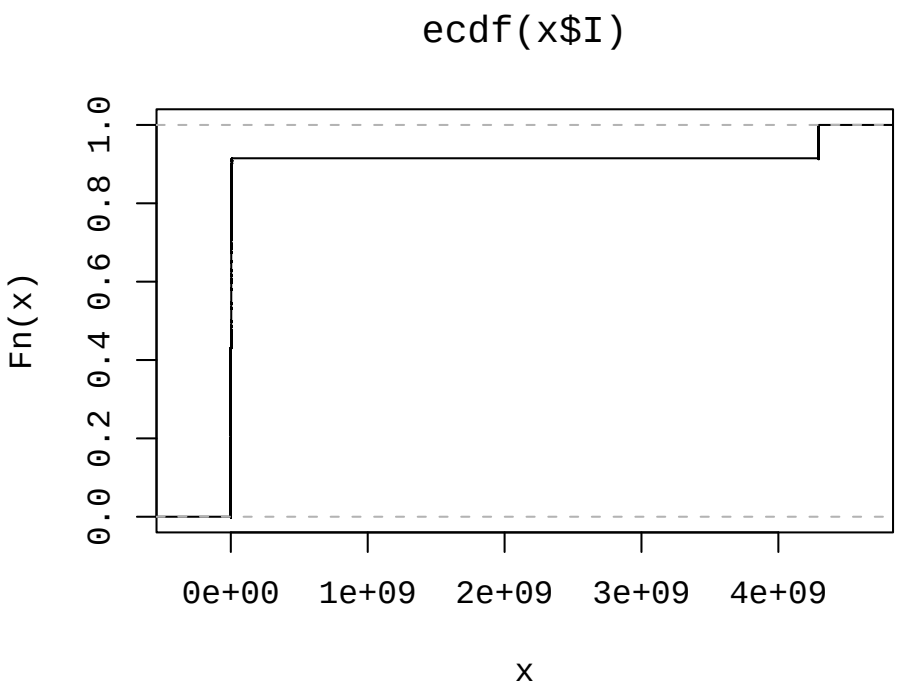

Abbildung A.4: Verteilungsfunktion der Quellen

Anhang B

L4Random Pool Ausgabe

Es wurden keine Aufwendigen Test durchgeführt, da die Ausgabe ein Produkt einer Hashfunktion ist. Die wenigen Tests haben dies bestätigt. Viel wichtiger war es, eventuelle groben Prgrammierfehler zu entdecken und den Monte-Carlo-Test mit mehreren tausend Durchläufen und die Untersuchung des Pools an einigen, zufällig ausgewählten Stellen.
Ausgabe von ent(1):

```
Entropy = 7.999980 bits per byte.
Optimum compression would reduce the size
of this 10000000 byte file by 0 percent.
Chi square distribution for 10000000 samples is 271.13, and randomly
would exceed this value 25.00 percent of the times.
Arithmetic mean value of data bytes is 127.5225 (127.5 = random).
Monte Carlo value for Pi is 3.140850056 (error 0.02 percent).
Serial correlation coefficient is -0.000416 (totally uncorrelated = 0.0).
```

Einige statistische Abbildungen: Die Abbildung B.1 zeigt die Verteilungsfunktion von drei Proben. Die beiden kleinen Abbildungen beziehen sich auch insgesammt auf 25 fortlaufende Proben, die zufällig an einer Stelle des Ausgebestroms entnommen wurden. Die größere Abbildung bezieht sich auf eine solche Sequenz der Länge 100.000. Die rote Gerade in jeder Abbildung zeigt den Erwartungswert. Wie es zu erwarten ist, weichen die Proben kürzerer Länge mehr oder weniger stark von dem Ideal ab. Bei der Probe der Länge 100.000 sind beide Linien nahezu identisch.
Dies bedeutet, dass die Ausgabe des Pools der erwünschten uniformen Verteilung entspricht.

Diese Messungen sind lokal. Um eine Aussage machen zu können, ob die Abweichungen von der Ideallinie statistischen Erwartungen entsprechen, müssten mehrere dieser Test durchgeführt und anschließend mit dem Chi-Quadrat-Test oder mit anderen Methoden bewertet werden.
Eine weitere Methode ist, die globale Bithäufigkeit zu bestimmen. Diese Tests beziehen sich auf alle 100.000 Proben. Weiterhin wird in den nachfolgenden Abbildungen die Bithäufigkeit gemessen. Dabei werden die 100.000 Proben in Teilsequenzen der Länge 25, beziehungsweise 250 aufgeteilt und die Anzahl der Einsen für jede Teilkette ermittelt. Erwartet wird, dass im ersten Fall die Summe durchschnittlich $12,5$ und im zweiten 125 ergibt. Im ersten Fall wurde zudem die Betrachtung auf jeweils das zweite Bit der Ausgabe beschränkt.
Die rote Linie verdeutlich wieder den Erwartungswert. Die Punkte sind die Häufigkeiten, dass die Summe (Anzahl der Einsen) kleiner ist oder gleich den zugeordneten X-Wert annimmt.
In beiden Fällen lässt sich sehr gut sehen, dass im Durchschnitt die Bitverteilung den Erwartungen entspricht. Auch für weitere Tests unterschiedlicher Länge und beschränkt auf unterschiedliche Bits entsprach das Ergebniss immer den Erwartungen.
Wie schon in Kapitel 6.13 erwähnt, handelt es sich hier um die Ausgabe einer Hashfunktion. Es wurden also keine anderen Ergebnisse erwartet.

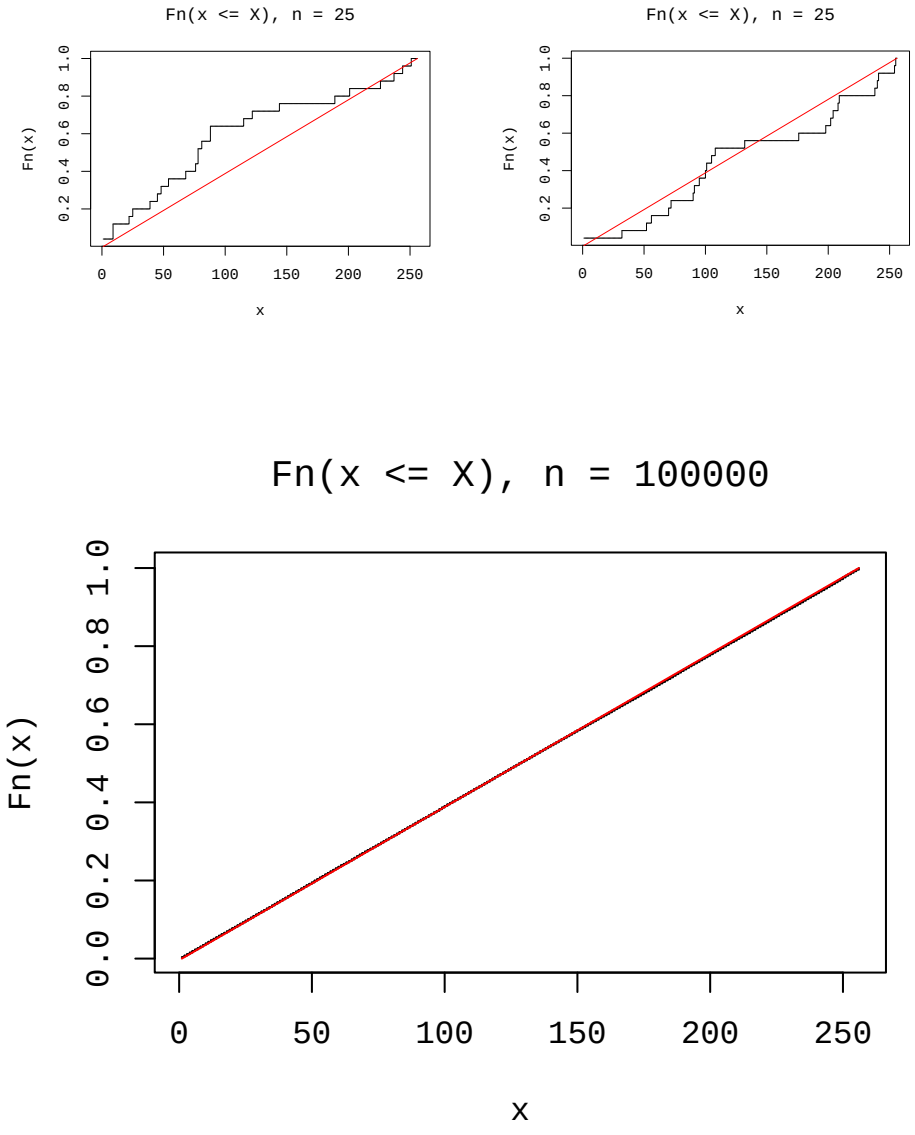

Abbildung B.1: Verteilungfunktionen

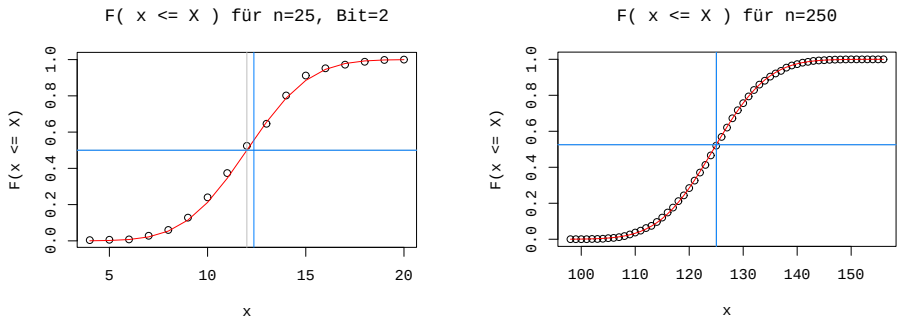

Abbildung B.2: Bitverteilung

Literaturverzeichnis

[1] Bruce Schneier, „Applied Cryptography 2nd Edition“, John Wiley and Sons, 1996

[2] D. Knuth, „The Art of Computer Programming: Volume 2, Seminumerical Algorithms“, Addison-Wesley, 1997

[3] Linux Sourcecode 2.6.7

[4] Juan Soto, „Statistical Testing of Random Number Generators“, National Intitute of Standarts & Technology

[5] D. Davis, R. Ihaka, P. Fenstermacher, „Cryptographic Ramdomness from Air Turbulence in Disk Drives“, Springer-Verlag Lecture Notes in Computer Sience, Nr. 839, 1994

[6] W. Killmann, W. Schindler, „Funktionsklassen und Evaluationsmethologie für physikalische Zufallszahlengeneratoren“, Bundesamt für Sicherheit in der Informationstechnick(BSI), Bonn, 2001

[7] RFC1750, „Randomness Recommendations for Security“, Network Working Group, 1994

[8] J.S. Coron, D. Naccache, „An Accurate Evaluation of Maurer's 'universal Test“, Frankreich

[9] M. Matsumoto, Y. Kurita, „Twisted GFSR Generators“, 1992

[10] M. Matsumoto, Y. Kurita, „Twisted GFSR Generators II“, 1992

[11] S. Gorenstein, „Testing a Random Number Generator“, New York

[12] A. Menezes, P. van Oorschot, S. Vanstone, „Handbook of Applied Cryptography“, CRC Press, 1996

[13] J. Kelsey, B. Schneier, N. Ferguson, „Yarrow 160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator“, Counterpane Systems

[14] P. Gutmann, „Software Generation of Practical Strong Random Numbers“, 7th USENIX Security Symposium, 1998

[15] I. Goldberg, D. Wagner, „Randomness and the Netscape Browser“, Dr. Dobbs Journal, Januar 1996

[16] J. Sandberg, „Netscape's Internt Software Contains Flaw that Jeopardizes Security of Data“, The Wall Street Journal, 18.09.1995

[17] „RFC 3174 - US Secure Hash Algorithm 1 (SHA1)“, Network Working Group, September 2001

[18] R. Rivest, „The MD5 Message-Digest Algorithm“, Network Working Group, April 1992

[19] H. Krawczyk, M. Bellare, R. Canetti, „RFC 2104 - HMAC: Keyed-Hashing for Message Authentication“, Network Working Group, Februar 1997

[20] A. Pfitzmann, „Datensicherheit und Kryptographie“, Tu Dresden, 2000

[21] PKCS Standards, RSA Laboratories

[22] „Dresden Real Time Operating System“, Technische Universität Dresdem Fakultät Informatik, `http://os.inf.tu-dresden.de/drops/`

[23] μ-Sina (Mikro-Sina), `http://os.inf.tu-dresden.de/mikrosina/`

[24] Unixprogramm ent(1),pseudorandom number sequence test, John Walker, `http://www.fourmilab.ch/`