4.1 Der Blum-Blum-Shub-Generator

Der Blum-Blum-Shub-**Generator** oder **BBS-Generator** setzt bei der in Kapitel III vorgestellten Quadratrest-Vermutung an und funktioniert so: Als ersten Schritt wählt man eine große zufällige Blum-Zahl m; weitere Einschränkungen werden – wie auch beim RSA-Verfahren – bei einer wirklich zufälligen Wahl nicht mehr für nötig gehalten, schaden aber auch nicht.

Als zweites wählt man dann einen (zufälligen) Startwert $x_0 \in [\lceil\sqrt{m}\,\rceil \ldots m - \lceil\sqrt{m}\,\rceil]$, der zu m teilerfremd ist. [Falls x_0 nicht zu m teilerfremd ist, hat man m per Zufall faktorisiert. Dass das vorkommt, ist äußerst unwahrscheinlich. Auch die Einschränkung $\sqrt{m} < x_0 < m - \sqrt{m}$ ist bei wirklich zufälliger Wahl unnötig und wird für die folgende Theorie nicht verwendet.]

Nun kann man an die Erzeugung einer Zufallsfolge gehen: Man bildet die Folge $x_i = x_{i-1}^2 \bmod m$. Ausgegeben wird aber *nur das letzte Bit* $b_i = \mathrm{lsb}(x_i)$. Außer eventuell dem Startwert x_0 sind alle x_i Quadratreste.

Die Zahlen p und q werden nur zur Bildung von m gebraucht und dann vernichtet; insbesondere sind sie als Geheimnis des Generators zu behandeln. Ebenso bleiben alle nicht ausgegebenen Bits der Folgenglieder x_i geheim.

Das Programm zur Parameter-Erzeugung für den BBS-Generator besteht aus folgenden Prozeduren:

Prozedur BlumPrime

[Erzeugt die kleinste Primzahl $p \geq x$, für die auch $\frac{p-1}{2}$ prim ist.]
Eingabeparameter:
x = Ausgangswert.
Ausgabeparameter:
p = kleinste Primzahl $\geq x$ mit $\frac{p-1}{2}$ prim.
Anweisungen:
Setze $p = x$.
Falls p gerade, erhöhe p um 1.
Falls $\frac{p-1}{2}$ gerade, erhöhe p um 2.
Solange ($\frac{p-1}{2}$ nicht prim) und (p nicht prim)
    erhöhe p um 4.

Aus dem Abschnitt über den Primzahlsatz wissen wir, dass es wohl sogar sehr viele der gesuchten Zahlen gibt. Der Algorithmus ist somit „empirisch“ auch ohne eine künstliche Abbruchbedingung korrekt. Im übrigen wird für ‘BlumPrime’ ein Primzahltest benötigt, wie sie auch schon behandelt wurden.

Prozedur BlumRandomPrime

[Erzeugt eine zufällige Primzahl mit n Bits, für die auch $\frac{p-1}{2}$ prim ist.]
Eingabeparameter:
n = Zahl der gewünschten Bits.
Ausgabeparameter:
p = eine Primzahl mit $2^{n-1} < p < 2^n$ und $\frac{p-1}{2}$ prim.
Anweisungen:
Bilde eine Zufallszahl x mit $2^{n-1} \leq x < 2^n$.
Setze $p = \text{BlumPrime}(x)$.
Falls $p \geq 2^n$, setze $x = 2^{n-1}$ und $p = \text{BlumPrime}(x)$.

Die Korrektheit dieses Algorithmus ist ebenfalls nur empirisch gesichert; der Fall, dass keine Primzahl gefunden wird, kommt in der Praxis aber nicht vor. Ein ernstzunehmender Einwand ist allerdings, dass der Algorithmus die verschiedenen Primzahlen mit ungleicher Wahrscheinlichkeit ausspuckt. Die Wahrscheinlichkeit für eine bestimmte Primzahl ist nämlich proportional zur Differenz zur nächstkleineren derartigen Primzahl (wobei die Differenz $\bmod 2^{n-1}$ zu bilden ist, wenn man den Übergang am Ende von 2^n zu 2^{n-1} auch noch angemessen berücksichtigen will). Auch ändert sich die Dichte der Primzahlen zwischen 2^{n-1} und 2^n fast um den Faktor 2, wie die Überlegungen zum Primzahlsatz gezeigt haben. Da aber kein Verfahren bekannt ist, diese Ungleichverteilung zuungunsten des BBS-Generators oder ähnlicher Verfahren auszunutzen, soll der Algorithmus hier nicht verkompliziert werden.

Für den Algorithmus 'BlumRandomPrime' wird außerdem ein Zufallsgenerator zur Bildung des Ausgangswerts x benötigt; hierfür sollte man auf einen willkürlichen Wert, einen „echten" Zufallswert zurückgreifen, z. B. einen aus einer genügend langen, vom Benutzer eingegeben Passphrase gebildeten.

Prozedur BlumNumber

[Erzeugt eine zufällige Blum-Zahl mit n oder $n+1$ Bits,
die schwer zu faktorisieren ist.]
Eingabeparameter:
n = Zahl der gewünschten Bits.
Ausgabeparameter:
m = eine Blum-Zahl.
Anweisungen:
Setze $k = \lfloor \frac{n}{2} \rfloor$.
Setze $l = n - k + 1$.
Setze $p = \text{BlumRandomPrime}(k)$.
Setze $q = \text{BlumRandomPrime}(l)$.
Setze $m = p \cdot q$.

Damit sind die Prozeduren zur Parameter-Erzeugung komplett. Eine BLUM-Zahl mit $n = 1025$, die mit diesem Verfahren erzeugt wurde, ist in Tabelle 2 abgedruckt (sie hat 309 Dezimalstellen). Im Hinblick auf den Fortschritt der Faktorisierungsalgorithmen solte man allerdings lieber BLUM-Zahlen in der Größenordnung ab 2048 Bit verwenden.

```
 4506 15286 74466 50249 26225 14044 26383 22616 74480 10227
69340 10344 80414 96318 08671 21639 63710 30387 17602 25696
53909 02080 09976 45161 76261 91025 59480 62175 49124 86394
40823 70452 14981 62658 94574 67753 74945 83135 16199 61782
07594 51105 16833 44889 30109 66289 10763 64987 90309 41852
27681 66632 02722 32988 57145 85172 07427 89442 30004 31819
83739 34537
```

Tabelle 2: Eine Blum-Zahl mit 1025 Bits

Die eigentliche Zufallserzeugung geht nun so: Man setzt den Modul m als globale Konstante und den Startwert x als globale Variable, die mit einem „echt“ zufälligen Wert im Intervall $[\lceil\sqrt{m}\,\rceil \ldots m - \lceil\sqrt{m}\,\rceil]$ vorbesetzt wird. Mit der folgenden Prozedur erzeugt man dann eine Bitfolge der gewünschten Länge:

Prozedur BBSrandomBit

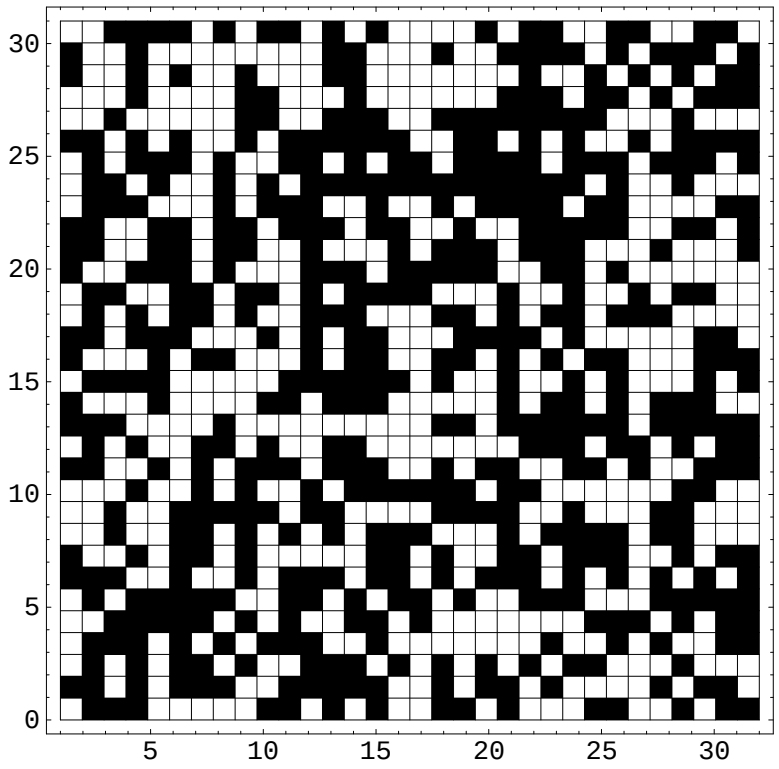
[Erzeugt eine Folge von n Pseudozufalls-Bits.]
Eingabeparameter:
n = Zahl der gewünschten Bits.
Ausgabeparameter:
blist = eine Liste von Bits.
Anweisungen:
Für $i = 1, \ldots, n$
ersetze x durch $x^2 \bmod m$,
setze $b = x \bmod 2$,
hänge b an blist an.

Mit dem oben erzeugten Modul und einem (gemäß der Konvention geheimgehaltenen) geeigneten Startwert wurde mit dieser Prozedur die Folge von 1024 Bits erzeugt, die in Tabelle 3 steht.

Einen optischen Eindruck von der Zufälligkeit der Folge vermittelt das folgende Bild.

1000 1111 1001 0101 1001 0111 0011 0100 0010 1000 1100 0001
1010 0101 1110 1001 1010 1001 0110 0010 1010 1010 0111 0111
1000 1010 1000 1101 1111 1101 1010 1100 1100 0001 0101 1001
0111 1111 0001 0100 1010 0000 1100 1010 0101 1000 1110 0000
0001 1011 0100 0100 1010 0010 1010 1010 0110 1001 0111 1100
1011 0010 0011 0100 1101 1001 0101 0100 0111 0100 0010 0111
1101 1000 0010 0111 1000 0110 1110 0111 1110 1101 0110 1000
0001 0011 1111 0011 0011 0101 0001 0001 1010 0110 0101 1000
1010 1100 1011 0011 1111 1000 1001 0100 0001 1110 1111 1111
1001 0000 0010 0000 0111 0111 1001 0001 1111 0100 1010 0011
1000 0111 1100 0000 1011 0110 1011 1010 0111 0100 1110 1001
1001 0101 0011 1000 0010 0011 1010 1001 1100 0010 1111 1001
1010 1001 0110 0011 1001 0100 1000 1111 1001 1001 0010 1000
0111 0110 1101 0011 0110 0010 1110 0010 0000 1100 1011 1111
0011 0010 0110 1110 1000 1000 1110 1110 0011 0010 0100 0100
1101 1000 0011 0010 1000 1110 1000 1101 1010 0001 0011 1100
1001 0110 1010 0000 0000 0000 1011 0111 1010 0010 1100 1010
0100 0010 0010 0010 0010 1011 0100 0000 1100 1010 1101 0000
1101 1111 0011 0001 1000 0000 0111 0111 1110 1111 0011 1011
1111 0001 0010 1000 0110 1011 0111 0011 1111 1011 0101 0100
0110 1111 1111 0011 1011 0000 1010 0010 1100 0010 1001 0101
1110 1001 1001 1001

Tabelle 3: 1024 „perfekte“ Zufallsbits


30
25
20
15
10
5
0
5
10
15
20
25
30