Cryptography, Statistics and Pseudo-Randomness (Part I)

Stefan Brands[*] Richard Gill[†]

Abstract

In the classical approach to pseudo-random number generators, a generator is considered to perform well if its output sequences pass a battery of statistical tests that has become standard. In recent years, it has turned out that this approach is not satisfactory. Many generators have turned out to seriously bias the outcome of some simulation experiments in which they were put to use. From a theoretical point of view, the classical approach does not at all explain in what way a completely deterministic algorithm can be said to simulate randomness.

Much less known is that cryptographers, who have a need for pseudo-random numbers of very high quality, have developed a theory that actually explains why a pseudo-random number generator can simulate randomness. Our aim in this two-part paper is to make this theory more accessible for mathematical statisticians and probabilists.

1 Introduction.

The classical approach to simulating randomness by purely deterministic means has proved to be effective in many applications where randomness is needed. However, as computers get faster and applications more sophisticated, the demand for more and more high quality pseudo-random numbers rapidly increases. In recent years, many of the classical pseudo-random number generators have turned out to exhibit statistical properties that can seriously bias the results of simulation experiments. Even 'state of the art' random number generators can turn out to be rather poor for some applications; see, e.g., Ferrenberg, Landau and Wong (1992).

Although the classical approach consists of a large body of useful information, it in our opinion somehow misses the point: in what way can a completely deterministic algorithm be said to simulate randomness? In fact, 'probability theory' is notably absent in treatments of the classical approach to pseudo-random number generation. Most of the literature is devoted to finding long periods of iteratively and deterministically determined integers which over a

[*] Centre for Mathematics and Computer Science, Kruislaan 413, 1098 SJ Amsterdam, Netherlands.

[†] Mathematical Institute, University Utrecht, Budapestlaan 6, 3584 CD Utrecht, Netherlands.

complete period have nice uniformity properties. If sequences produced by a generator pass a standard set of several statistical tests, then the generator is generally accepted as satisfactorily pseudo-random. See Knuth (1981) for the classical theory; and Marsaglia and Zaman (1991) for more recent developments.

In cryptography there is a need for specially reliable random number generators. The reason for this is that a secret key, used for encryption of messages or other purposes, is hardest to guess by adversaries when it is chosen uniformly at random from the set of all allowed keys. For effective use in practice the key often must be produced deterministically, by a pseudo-random number generator. However, by studying the observable effects (e.g., encrypted messages) of using the secret key in a particular cryptographic application, an adversary may for example guess the secret key with significant probability of success. In particular, in many applications the algorithm used to generate the keys cannot be assumed to remain known only to the legitimate user. Therefore, pseudo-random number generators useful for cryptographic applications must satisfy very strong requirements concerning their statistical properties.

To this end, cryptographers have not just invented their own pseudo-random number generators but even developed an elaborate and elegant theory, containing nice probabilistic and statistical ideas, which actually *explains* why a pseudo-random number generator can simulate randomness. In the cryptographic approach, a generator is considered to be pseudo-random if the amount of information that can be computed from it in a feasible amount of time is essentially the same as what can be got out of a truly random sequence. The algorithm that is to extract such information may know the method of generation; in fact, only the seed is supposed to be unknown to it. To formalise this approach, intriguing notions based on algorithmic complexity theory are needed, such as one-way permutations and hard-core bits.

In this article and its sequel Brands and Gill (1996), we aim to make the cryptographic theory more accessible for mathematical statisticians and probabilists. We will moreover argue that it is highly relevant to the actual use of pseudo-random number generators in statistical simulation experiments, bootstrapping, and so on. Further background material including results on statistical testing of the new generators is given in Brands (1991).

This Part I is an essayistic introduction to the cryptographic theory. It much improves an earlier version which appeared as (Gill, 1994, section 14). In Part II (Brands and Gill, 1996) we provide the mathematical analysis of the QR-generator, one of the most efficient of the new generators. The results are not new, but they are scattered over many papers and the proofs are often not very accessible for non-experts in the field.

2 Neyman and random number generators.

In this special issue of *Probability and Mathematical Statistics* it is appropriate to emphasize a fundamental connection between our subject and the work of J. Neyman. The theory of reliable pseudo-random number generators presented below is based on the idea that a good pseudo-random number generator should ‘pass every (feasible) statistical test’ of the generator. Passing a statistical test means: the power of the test to detect a departure from randomness of the given generator is not larger than the size of the test—the probability it rejects a truly random sequence.

The Neyman-Pearson theory of statistical testing, and in particular the concepts of size and power of a statistical test, are in fact at the heart of this theory.

Neyman made much use of pseudo-random number generators in his work with Betty Scott. We are grateful for the following remarks made for us by L. Le Cam.

“In the early fifties, Neyman and Scott worked on a cluster model for galaxies. To see whether the model had a chance they generated a lot of ‘random galaxies’ (the random sample was not clustered enough).

“Another time Neyman was worried about the power of one of his $c(\alpha)$ tests. He and Betty generated thousands and thousands of random data and tried the tests. They also tried t-tests and found to their dismay that the t-distributions varied periodically along the generated samples. They talked to Dick Lehmer [inventor of the linear congruential generator] who gave them another generating algorithm.

“Neyman was often quite cautious in applications. Unless there was sound theory, he tried his test or estimates on random data. When I met him around April 1950 (in Paris) he asked me to provide him with a hundred samples of size 12 from a uniform distribution, just to illustrate the point that confidence intervals based on a maximum are shorter and better than those based on the means.”

3 The classical approach.

A classical pseudo-random number generator is an algorithm which, on given a starting number called the seed, produces a sequence of numbers according to a simple deterministic recursion. Usually the numbers are integers in a given, finite range, hence the sequence eventually becomes periodic. For instance, the well-known linear congruential generator, starting with an integer seed x_0, produces a sequence of integers $(x_n)_{n\geq 0}$ according to the rule

$$x_n = ax_{n-1} + b \bmod m \tag{1}$$

where the integers a, b and m are integer parameters chosen beforehand. By $x \bmod m$, we mean the least positive remainder of x when divided by m. Most reseach on this method has been aimed at finding values of the parameters a, b and m such that the resulting sequence (on given a specific seed x_0 as input) passes the statistical tests thought to be representative for pseudo-randomness. Scores of papers have been written on this subject, establishing period-lengths for various choices of the parameters and the seed, and reporting on results of statistical tests. Much attention has been paid to the question whether the choice $b \neq 0$ is better or worse than $b = 0$; indeed it has been shown that essentially there is no difference. The choice of b is important only in that a bad choice may shorten the length of the period. The same is true for the seed value. For an appropriate choice of a, b and m, it can be guaranteed that the numbers x_n follow a cycle which is actually a permutation of the set $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$ of the integers modulo m (see for example Knuth, 1981).

On the other hand it is known that if one uses n-tuples of these numbers to represent points in the unit cube in n-space, one finds that the points, rather than appearing to be randomly spread throughout the n-dimensional cube, fall on a lattice with relatively large spacing compared to what is expected under the null-hypothesis of true randomness.

For this reason, it might be worthwhile to investigate whether the performance can be improved by outputting only part of the bits of x_n. The numbers

$$u_n = x_n/m$$

seem to behave reasonably like independent uniform$[0, 1)$ random variables and

$$y_n = \lfloor 2u_n \rfloor$$

as independent Bernoulli $(\frac{1}{2})$ variables (fair coin tosses). Note that y_n is the most significant bit of the number u_n expressed as binary fraction. Since a uniform$[0, 1)$ random variable is approximated on the computer by a number of fixed, finite precision, and since the successive bits in a uniform$[0, 1)$ random variable are independent Bernoulli $(\frac{1}{2})$ variables, a pseudo-random bit generator which produces independent Bernoulli$(\frac{1}{2})$ variables is all we really need.

4 The cryptographic approach.

The classical approach in fact is concerned with the randomness of single sequences: a sequence is considered random enough if it passes several statistical tests. This seems somewhat illogical. Furthermore, the battery of tests that a sequence ought to pass is rather arbitrary. The cryptographic approach is concerned with the randomness of *collections* of sequences of bits.

4.1 Motivation.

The new generators from cryptography are not much different from the classical generators. For example, the so-called *quadratic-residue* or QR-generator is defined as follows: given suitably chosen integers x_0 and m, define

$$x_n = \begin{cases} x_{n-1}^2 \bmod m & \text{if } x_{n-1}^2 \bmod m < m/2; \\ m - (x_{n-1}^2 \bmod m) & \text{otherwise.} \end{cases} \tag{2}$$

and let

$$y_n = \text{lsb}(x_n)$$

denote the least significant bit of x_n. Then it can be shown that the y_n can excellently approximate fair Bernoulli trials. The theorem which guarantees this (under a certain unproven but generally accepted assumption) is an asymptotic theorem, for the case that the binary length, k, of the modulus converges to infinity. The assumption is that it is infeasible to factor the modulus m if it is the product of two distinct primes of approximately equal length that are both congruent to 3 modulo 4 (such composites are called Blum integers). The problem of factoring composites is widely assumed to be an infeasible task, and Blum integers are thought to be among the hardest composites to factor. At this point, it will be unclear in what sense the infeasibility of factoring

Blum integers relates to the pseudo-randomness of the QR-generator. After discussing the general cryptographic theory, we will return in detail to the QR-generator in Part II of this paper.

A minor difference from the classical generators is that what would be a fixed parameter m is now also considered part of the seed. The only parameter of the QR-generator is in fact the chosen length k of the numbers x_n produced inside the generator.

The basic idea in the cryptographic approach is that a pseudo-random number generator is not a device for creating randomness but rather a device for amplifying randomness. If we consider the seed as truly random, then the output sequence $(y_n)_{n\geq 0}$ is also random, and we may ask how close its distribution is to the distribution of fair Bernoulli trials.

For simplicity, let us suppose that the seed is a k-bit integer, chosen uniformly at random from all possible bit strings of this length. Clearly, the joint distribution of the output sequence $(y_n)_{n\geq 0}$ is highly degenerate, especially if it is long. Suppose we generate $y_1, \ldots, y_{l(k)}$ where $l(\cdot)$ is some polynomial in k of degree at least 2. There are only 2^k possible, equally likely values for the whole sequence (assuming they are all different) out of an enormous $2^{l(k)}$ equally likely values of a truly random binary sequence of length $l(k)$. However the degeneracy can be so well hidden that we are not aware of it. And this must hold for the classical random number generators which are routinely used by statisticians and others at exactly the kind of scale described here.

Obviously the degeneracy can be found if one looks for it. A powerful statistical test for determining whether $y_1, \ldots, y_{l(k)}$ are truly random or only pseudo-random consists of checking if the sequence is one of the 2^k sequences produced by the generator or one of the other $2^{l(k)} - 2^k$ sequences possible with a truly random sequence. This constitutes a statistical test with size approaching zero if k is increased (more specifically, the size is $2^{k-l(k)}$) and power equal to one. There is a big drawback to this test however: it takes a lot of computing time. Suppose for example that the seed is one hundred bits long, and that $l(k) = k^4$. Then we are talking about using one hundred fair coin tosses to simulate one hundred million. Producing a single sequence of 100 000 000 bits for the statistical simulation experiment is very feasible, but producing all 2^{100} possible sequences is definitely not feasible; it would take a million supercomputers that can each produce a million sequences per second about forty billion years to get the job finished! So the just mentioned statistical test is certainly infeasible. However there might exist statistical tests which only require feasible computational resources yet just as conclusively detect pseudo-randomness from true randomness.

4.2 Founding the theory on algorithmic complexity theory.

The aim of the cryptographic theory is to construct pseudo-random number generators such that no *feasible* statistical test can show up the difference between a deterministically generated sequence and a truly random sequence. The use of the word ‘feasible’ sounds vague but can be made completely precise through standard notions of algorithmic complexity theory. Feasible in algorithmic complexity theory means computable in *polynomial time* by an algorithm (which can be formalised using, e.g., Turing machines). That is, the running time of the algorithm used to compute the test is at most polynomial in the size of the input (here, we measure the size of numbers by their binary length). Note that feasibility is formalised in an asymptotic sense: only asymptotically (in the size of a given problem) can one distinguish between problems that are feasible to solve and problems that are infeasible to solve. The formal distinction between feasible and infeasible problems is generally agreed to correspond rather closely to the practical distinction between problems which, as their size gets larger, remain tractable or become untractable.

At the same time, 'feasible' is formulated in a probabilistic sense: the algorithm must run in (expected) polynomial time. The probability aspect stems from the fact that the algorithm is allowed to be probabilistic, i.e., it is allowed to make choices based on flipping a fair coin.

'Showing up the difference' between a generated sequence and a truly random sequence can also be made precise. We have a statistical testing problem with, as null hypothesis, true randomness; as alternative, the distribution induced by the generated sequence from the distribution of the seed. A given statistical test shows up a difference if there is a difference between the size and power of the test. Again, this has to be formulated in an asymptotic sense.

Note that if the size and power of a given statistical test are different, one can independently repeat the test a number of times and build a new test whose power and size lie even further apart. In fact, if the power and size differ by at least one divided by a polynomial in the size of the input to the test, then only a polynomial number of replications of the test suffice to bring the size close to zero and the power close to one. Thus, 'failing a feasible statistical test' in the weak sense of power being just 'slightly' bigger than size means that there exists a much more conclusive feasible test which the generator also fails.

In Part II we will give a fairly detailed overview of the proof that the QR-generator is pseudo-random in this sense, provided it is true (as most people believe) that factoring Blum integers is infeasible on average. The proof consists of converting a statistical test which shows up the nonrandomness of this pseudo-random number generator into a probabilistic polynomial-time algorithm for factoring the modulus m.

The linear congruential generator, as defined by (1), can be shown not to be pseudo-random in this sense: one can essentially recover the seed from the sequence in polynomial time, and hence come up with a statistical test that overwhelmingly rejects its randomness; see Boyar (1989). It is an interesting open question whether modifications exist that do pass all polynomial-time statistical tests, although several partial results suggest that this is unlikely to be the case.

4.3 Historical perspective.

The idea to classify a generator as pseudo-random if no polynomial-time statistical test can distinguish its outputs from truly random sequences is in fact a very natural one, as is reflected in the history of pseudo-random number generators. Around 1938, a sequence output by a number generator was considered fine if it passed the four tests devised by Kendall and Babbington (being the frequency test, the serial test, the poker test and the gap test). Then some other tests were added; for quite a while the series of tests described in Knuth (1981) was thought to be very representative. Nowadays, it seems that this set of tests is not stringent enough for sophisticated applications. This has lead to the development of some extra tests that are more difficult to pass by sequences that are not good enough, yet pass all the tests in Knuth (1981). For details about such stringent tests, see Marsaglia (1984). From this perspective, the cryptographic theory is a natural next (big) step forwards, considering *all* feasible statistical tests, with the interpretation of feasible in terms of polynomial time making the theory very robust under future technological developments.

5 The relevance of the cryptographic approach.

Before embarking on the cryptographic theory we should pay some more attention to its relevance. In practice, does it make sense to suppose that the seed of a pseudo-random number generator is chosen at random? What has ‘passing all feasible statistical tests’ got to do with how a generator is actually used in practice?

As an example, let us consider the statistical simulation experiments carried out in Nielsen, Gill, Andersen and Sørensen (1992) which aimed to show that a generalised likelihood ratio test, in a certain semiparametric model for bivariate data estimated by non-parametric maximum likelihood, has the same asymptotic properties as in the parametric case. During the simulations the nominal P-value of the log likelihood ratio test, assuming an asymptotic chi-square distribution to be applicable under the null-hypothesis, was calculated for a large number of large samples from the null model. If the conjectured asymptotic theory is true and if the chosen sample size is large enough to make it a reasonable approximation, these P-values should be approximately a sample from a uniform distribution on $[0, 1]$. The simulations were summarised by plots of the empirical distribution function of the observed P-values. At larger sample sizes, the plots looked very much indeed like pictures of the uniform empirical distribution function, supporting the conjectured results.

Suppose the plot is based on 1000 replications at sample size 1000. We see 1000 points, lying closely along the diagonal in the unit square. Each point represents a single P-value based on a sample of size 1000 from some bivariate distribution. Thus, supposing real numbers were represented by strings of 30 bits, about 60 million simulated fair coin tosses are needed to draw the graph. In fact the 60 million coin tosses are the completely deterministic output of a pseudo-random number generator, starting with a seed represented as a string of about 100 bits. The seed was the result left at the end of the previous simulation experiment; alternatively one may let the system choose the seed itself in some way (using the system clock, perhaps) or the user can set it: perhaps with real fair coin tosses but more likely using a coding of his or her birthday or bank account number or just with the first string of numbers which came to mind. In any of these cases it seems completely justified to consider the initial seed, for this experiment, as truly random and even uniformly distributed on its range. (If one carries out a number of simulation experiments at the same workstation using subsequent segments of the same cycle of pseudo-random numbers, different experiments are not independent of one another, however this does not change the interpretation of what is going on in one given experiment.)

As we discussed before, the joint distribution of the whole output sequence does not remotely look like what it is supposed to simulate. However, one is not interested in the joint distribution of the whole output sequence but just in the distribution of a few numerical statistics, perhaps even a single zero-one valued statistic. The statistics of interest depend on the application for which the random numbers are needed. For instance, the conclusion drawn from the plot we have discussed is ‘this looks like a uniform sample’. One could quantify this impression by calculating some measure of distance of the plotted curve from the diagonal, or just carry out a Kolmogorov-Smirnov test at the 5% level (with as conclusion ‘O.K.’).

Also in applications of the bootstrap and in other statistical applications, it is common to use a hundred or so essentially random, fair coin tosses, to generate several million up to several billion, and then compute from these just a few numerical quantities. For example, the result of a bootstrap experiment is the measurement of one or two empirical quantiles, to be used in the construction of a confidence interval. The only important thing about these observed quantiles (based on several thousand replicates of a statistic computed on samples of one hundred or a thousand observations) is that they lie with large probability, under pseudo-randomness, in the same small interval (about ‘the true bootstrap quantile’) as under true randomness.

The gist of this is that, even if we produce millions of random numbers in a statistical simulation experiment, we are only interested in the outcome of a few zero-one variables computed from all of them. Use of the simulation is based on a reliance that these variables have essentially the same distribution under pseudo-randomness as under true randomness: in other words, they should be of no use as a statistical test of randomness against pseudo-randomness. If the distributions were different and known in advance, we could even use our simulation experiment as a statistical test of our generator. It would be the most sensible test to use since it tests exactly the aspect of the generator which is important for the application! However, the probabilities in question are not known in advance and cannot be easily calculated, which is after all exactly the reason for doing a simulation experiment in the first place. Note also that even if the simulation experiment of Nielsen, Gill, Andersen and Sørensen is large, it still gets finished in reasonable time and if necessary could be repeated a few times. The statistical test of randomness which the use of the experiment represents, is a polynomial-time test.

The point is that a pseudo-random number generator which passes *all* feasible tests is a number generator which we can safely use for *all* practical applications. Of course, since the requirements for the cryptographic generators are stated in asymptotic terms, in practice the length of the seed to the generator must be chosen with lower bound which is assumed to be sufficient to approximate the asymptotic results.

6 The cryptographic theory.

Now back to cryptography. Two notions are central to the theoretical construction of pseudo-random number generators: one-way permutations and hard-core predicates.

6.1 One-way permutations.

Informally, a *one-way permutation* is a permutation which is easy to compute, while computing its inverse is difficult on average. ‘Easy’ and ‘difficult’ mean here: in polynomial time, and not in polynomial time, respectively. The notion is therefore an asymptotic notion and we are really applying it to a sequence of permutations $f_k(\cdot)$, typically defined on a subset of the set of binary strings of length k.

More specifically, a permutation $f(\cdot)$ is (strongly) one-way if, on the one hand, $f(\cdot)$ can be computed in time polynomial in the length of its argument, but on the other hand, for all polynomials $p(\cdot)$ and (probabilistic) polynomial-time algorithms M,

$$\mathrm{P}_k\Big(M(f(x)) = x\Big) < \frac{1}{p(k)}$$

for all sufficiently large k. By $M(y)$ we mean the result of applying the algorithm M to the input y. The probability distribution P_k is taken over $\{0,1\}^k$ and the internal coin tosses of M. Note that the inverting algorithm M is allowed to be probabilistic. The distribution of x over $\{0,1\}^k$ in general will not be the uniform distribution, but usually will assign the uniform distribution on an appropriate subset D_k of $\{0,1\}^k$.

The assumption that one-way permutations exist is at least as strong as the famous $NP \neq P$ conjecture of algorithmic complexity theory. Namely, if $NP = P$, then the inverse under $f(\cdot)$ can be correctly guessed in polynomial time. This implies, because the NP versus P question is unresolved, that permutations can only be proven one-way under some unproven assumption. In fact, since a one-way permutation is defined to be difficult to invert on the average, whereas the complexity class NP considers only the worst-case complexity of problems, $NP \neq P$ does not imply the existence of a one-way permutation.

Note that a necessary but not sufficient condition for a one-way permutation is that the number of elements in the domain D_k grows super-polynomially with k. If this were not the case, then a simple polynomial-time inverting algorithm exists that successively evaluates the permuted value of each element in the domain until the inverse is found. Likewise there must exist a probabilistic polynomial-time algorithm that can sample elements from the domain according to some specified distribution (usually the uniform distribution).

6.2 Hard-core predicates.

If a permutation $f(\cdot)$ is one-way, then for the majority of elements in its range it cannot be the case that all the bits of the inverse can be computed in polynomial time; at least one bit of x must be hard to predict given $f(x)$. Indeed, there may exist more, perhaps even all of the bits of x are hard to predict. More generally, there must exist at least one polynomial-time computable function from a subset of the bits of the argument of $f(\cdot)$ to $\{0,1\}$ that is at least somewhat hard to compute given $f(x)$. Intuitively, the idea of the cryptographic approach is to extract this bit and use it to construct the output of the generator. This is formalised through the notion of a *hard-core predicate*.

Formally, a predicate $B(\cdot)$ is hard-core for the permutation $f(\cdot)$ if, on the one hand, $B(f(\cdot))$ can be computed in polynomial time, but on the other, for all polynomials $p(\cdot)$ and probabilistic polynomial time algorithms M,

$$\Big|\mathrm{P}_k\Big(M(x) = B(x)\Big) - \frac{1}{2}\Big| < \frac{1}{p(k)}$$

for all sufficiently large k. Again, the probability distribution is defined over the set $D_k \subseteq \{0,1\}^k$ and the internal coin tosses of M.

Intuitively, one should think of $B(\cdot)$ as assigning to x a particular hard to compute bit of its inverse under the mapping defined by $f(\cdot)$. Knowing x, it is hard to compute its inverse under $f(\cdot)$ because $f(\cdot)$ is one-way. Some bits of such an element may be easily computable but by definition not the hard-core bit. A hard-core predicate maintains in a very strong sense the one-way property of the permutation $f(\cdot)$.

As an example, consider a one-way permutation that is defined on a set that consists purely of odd integers. Although it is infeasible on the average to compute x from $f(x)$, the least significant bit of the argument of $f(\cdot)$ is certainly not hard-core, since it is always a 1.

The only known method to rigorously prove that a predicate $B(\cdot)$ is hard-core for a permutation $f(\cdot)$ is to show that the existence of a feasible algorithm that computes it would imply the existence of a feasible algorithm to invert $f(\cdot)$, which would be a contradiction with $f(\cdot)$ being one-way. This is called a (probabilistic) polynomial-time reduction. Probabilistic reductions often involve ingenious 'oracle' sampling techniques, based on statistical techniques; a good example of such a reduction will be given in Part II of this paper, where we discuss the proof that the least significant bit of the function displayed in (2) is hard-core.

We next set up the definitions needed to discuss pseudo-random number generators.

6.3 Statistical tests.

A bit generator $f(\cdot)$ is actually a sequence of polynomial time computable functions $f_k(\cdot)$ mapping, say, $\{0,1\}^k$ to $\{0,1\}^{l(k)}$ for some polynomial function $l(\cdot)$. The domain is called the seed domain and given an appropriate probability distribution. A feasible statistical test T of a generator $f(\cdot)$ is a sequence of (probabilistic) polynomial-time algorithms $T_{l(k)}$ that take inputs from $\{0,1\}^{l(k)}$, and output a zero or a one. The outcome should be interpreted as 'accept' or 'reject', and the algorithm $T_{l(k)}$ is in fact a statistical test of the null hypothesis that the output sequences follow the uniform distribution on strings of length $l(k)$ against the alternative that they have the distribution induced through the generator by some distribution over strings of length k. We say that $f(\cdot)$ passes the statistical test T if, for all polynomials, the difference between the power and size of the statistical test is eventually smaller than one divided by that polynomial, which implies that this difference cannot be 'significantly' amplified by polynomially many repetitions of the test. A generator is *pseudo-random* if it passes all feasible statistical tests.

Intuitively, this criterion for pseudo-randomness can be visualised by viewing the sequences of length $l(k)$ produced by the generator as points scattered throughout the set of all bit strings of length $l(k)$. A *feasible* statistical test can be thought of as inducing a 'very simple' partition (such as a line) of the set of all strings of length $l(k)$ into two categories: those strings that it accepts and those that it rejects. The test is passed if the pseudo-random points are divided in approximately the same proportion as the set of all points.

6.4 Unpredictability.

An apparently less stringent criterion of a generator is (next-bit) *unpredictability*. This means that for each position from 1 up to $l(k)-1$ in the output sequence, no feasible probabilistic algorithm exists which predicts, with success probability 'significantly' exceeding a half, the next output bit of the sequence, given the preceding bits up to this position. If a generator is predictable then for some position in the output sequence one can, with non-negligible success probability, feasibly predict the next bit from the preceding ones.

An important theorem of Yao (1982) states that the property of being unpredictable is actually equivalent to being pseudo-random. In other words, passing all feasible next-bit statistical tests implies passing *all* feasible statistical tests. Since pseudo-randomness does not depend on whether the output bits are taken in their usual order or in reverse order, we have the corollary: forwards unpredictability is equivalent to backwards unpredictability. Thus, the notion of unpredictability of next bits (or previous bits) forms a ‘universal’ test of randomness with respect to polynomial-time computation. The relevance of this is most of all that it is much easier in general to prove (perhaps under some assumption) that a generator is unpredictable than to show directly that it is pseudo-random. Indeed, the proof of correctness of the general construction discussed in the next subsection depends on this fact.

Here is a sketch of the proof of the theorem. It is easy to see that if a generator is predictable then it is not pseudo-random, since we can obviously construct a feasible statistical test, that the generator fails, from a successful prediction of one of its output bits. For the converse, suppose the generator is not pseudo-random. This means that there exists a feasible statistical test whose size and power are ‘significantly’ different from one another. Denote the output sequence of the generator by $y = (y_1, \ldots, y_{l(k)})$ and let $y^* = (y_1^*, \ldots, y_{l(k)}^*)$ denote a truly random sequence. From these two consider all the ‘cross-over’ combined sequences:

$$y^{(n)} = (y_1, \ldots, y_{n-1}, y_n^*, \ldots, y_{l(k)}^*), \quad n = 1, \ldots, l(k).$$

Apply the statistical test to both of $y^{(n)}$ and $y^{(n+1)}$. Since there is an appreciable difference between the probabilities of rejecting y^* and rejecting y, there has to be somewhere, at least, some ‘significant’ difference between the probabilities of rejecting $y^{(n)}$ and $y^{(n+1)}$, since the first element of the first of these pairs is y^* and the second element of the last of the pairs is y. Here we use the fact that $l(k)$ is a polynomial in k: a probability which is larger than one divided by some polynomial, also has this property when divided by $l(k)$ (pigeonhole principle!).

Now if one can distinguish between $y^{(n)}$ and $y^{(n+1)}$ for some n, it seems plausible that one can predict, with some success, y_n from $(y_1, \ldots, y_{n-1})$, since the only difference between $y^{(n)}$ and $y^{(n+1)}$ is whether the nth bit contains the deterministically formed y_n or the fair coin toss y_n^*. Indeed, one can show that a feasible probabilistic prediction algorithm can be built on comparing the results of the statistical test applied to the two sequences: $(y_1, \ldots, y_{n-1}, 0, y_{n+1}^*, \ldots, y_l^*)$ and $(y_1, \ldots, y_{n-1}, 1, y_{n+1}^*, \ldots, y_l^*)$. Note that the resulting next-bit test is probabilistic because it has to supply the fair coin tosses $(y_{n+1}^*, \ldots, y_l^*)$.

6.5 The construction of Blum and Micali.

To conclude our overview of the general cryptographic theory, we show that given any one-way permutation $f(\cdot)$ with a hard-core predicate $B(\cdot)$, one can construct a pseudo-random generator $g(\cdot)$ by iterating $f(\cdot)$, and outputting successive values of $B(f(\cdot))$ (both of which are easy to do). That is,

$$g(x) = (B(f(x)), B(f(f(x))), \ldots, B(f^{l(k)}(x))),$$

where the seed x is sampled from the same distribution as that defined for the arguments of $f(\cdot)$. This elegant and important construction is due to Blum and Micali (1984).

To see why this results in a bit generator that is pseudo-random, we show that $g(\cdot)$ is not backwards predictable by a feasible statistical test. If $g(\cdot)$ were backwards predictable, we could feasibly guess, with some degree of success, the value of say $B(f^n(x))$ given the values of $B(f^{n+1}(x)), \ldots, B(f^{l(k)}(x))$. This latter set of values can be feasibly computed from $f^n(x)$. So given $f^n(x)$ one can apparently guess $B(f^n(x))$ with success probability that 'significantly' exceeds 1/2. But this contradicts $B(\cdot)$ being hard-core, since the fact that $f(\cdot)$ is a permutation implies that $f^n(x)$ has the same distribution as $f(x)$.

7 Bibliography

M. Blum and S. Micali (1984), How to generate cryptographically strong sequences of pseudo-random bits, *SIAM J. Comp.* **13**, 850–864.

J. Boyar (1989), Inferring sequences produced by pseudorandom number generators, *J. ACM* **36**, 129–141.

S.A. Brands (1991), *The Cryptographic Approach to Pseudorandom Bit Generation*, Master's thesis, Dept. Math., Univ. Utrecht.

S.A. Brands and R.D. Gill (1996), Cryptography, Statistics and Pseudo-Randomness (Part II), *Probability and Mathematical Statistics*, to appear.

A.M. Ferrenberg, D.P. Landau, and Y.J. Wong (1992), Monte Carlo simulations: hidden errors from 'good' random number generators, *Phys. Rev. Letters* **69**, 3382–3384.

R.D. Gill (1994), Lectures on Survival Analysis, in: P.Bernard (ed.), *Lectures on Probability Theory, Ecole d'Eté de Probabilités de Saint Flour XXII–1992*, Springer Lecture Notes in Mathematics 1581.

D.E. Knuth (1981), *The Art of Computer Programming, vol. 2: Seminumerical Algorithms*, Addison-Wesley.

G. Marsaglia (1984), A current view of random number generators, *Proceedings, Computer Science and Statistics, 16th Symposium on the Interface*, Atlanta.

G. Marsaglia and A. Zaman (1991), A new class of random number generators, *Ann. Appl. Probab.* **1**, 462–480.

G.G. Nielsen, R.D. Gill, P.K. Andersen, and T.I.A. Sørensen (1992), A counting process approach to maximum likelihood estimation in frailty models, *Scand. J. Statist.* **19**, 25–43.

A.C. Yao (1982), Theory and applications of trapdoor functions, *Proc. 23rd IEEE Symp. Found. Comp. Sci.*, 458–463.