7 Probabilistic Encryption

[References to "the paper" in this section are to "Probabilistic Encryption," in *Journal of Computer & System Sciences* 28, pp. 270–299. I have also used *Primality and Cryptography*, by E. Kranakis]

So far, the public key systems have been functions f such that the message M presumably cannot be computed from the encoding $f(M)$. A further concern arises as to whether, even if the adversary cannot identify M exactly, he may be able to obtain some partial information about M, for example tell whether M is an even number, a square, a power of 2, etc.

An extreme case of this would be a scenario in which the adversary knows the message is one of two possibilities, M_1 or M_2. Since we have been assuming that the function f is easy to calculate, all the adversary needs to do is compare $f(M_1)$ and $f(M_2)$ with the ciphertext.

Probabilistic encryption is a system designed to avoid these problems. Instead of $f(M)$ being a single number, the calculation of $f(M)$ involves the sender doing some things randomly during the calculation, so that M has many different encryptions. Indeed, the probability should be very close to 1 that if the same message is sent twice, the encryptions should be different.

7.1 The Goldwasser-Micali encryption system

As in many previously discussed systems, the person receiving messages chooses two primes ($\sim$ 100 digits) p, q and announces $n = pq$. This system is concerned with whether, for a given number a, there is x with $x^2 \equiv a \pmod{n}$. Such a are called *squares* or (in most books and papers) *quadratic residues*. For technical reasons, when we refer to squares mod n, we will exclude a which are divisible by p or q. The following facts are easy to prove, in some cases using primitive roots.

Lemma 20 *If a, b are squares, then ab is a square. If a is a square and b is not a square, then ab is not a square.*

Lemma 21 *a is a square mod n if and only if it is a square mod p and a square mod q.*

Lemma 22 *Let $h = \frac{p-1}{2}$. If a is a square mod p, $a^h \equiv 1 \pmod{p}$. If a is not a square, $a^h \equiv -1$.*

This implies that, if p and q are known, it is easy to decide whether a is a square. The encryption system depends on the assumption (called QRA in the paper [p. 294]) that this problem is very difficult if p, q are unknown.

Lemma 23 $1/2$ *of the numbers from 1 to* $p-1$ *are squares mod* p*. Take the numbers from 1 to* n *and leave out those divisible by* p *or by* q*. Divide the remaining* $(p-1)(q-1)$ *numbers into four groups according to whether they are squares or not mod* p *and also mod* q*. There are* $(p-1)(q-1)/4$ *numbers in each group.*

The numbers which are not squares mod p and also not squares mod q are called *pseudo-squares*. Example: If $p = 5$, $q = 7$, the squares mod 35 are 1, 4, 9, 16, 29, 11 ($29 \equiv 8^2$, $11 \equiv 9^2$; note we don't include 25 and 14, because they're divisible by p, q). The pseudo-squares must be congruent to 2 or 3 mod 5 and to 3, 5, or 6 mod 7. Thus the pseudo-squares are 17, 12, 27, 3, 33, 13.

The encryption system is primarily concerned with the union of the set of squares and pseudo-squares— this set is unfortunately denoted both by Z_n^1 (p. 291) and by Z_n^{+1}. Since exactly half the members of Z_n^1 are squares, the crude idea of saying "this is a square" all the time will only be right half the time. (QRA) says that no algorithm that runs in a reasonable amount of time can do much better than this. [the precise definitions of "reasonable" and "much better" are what require the concepts of circuits of size k and "ϵ-approximating"]

In addition to announcing n, the person receiving messages announces one pseudo-square y. To send a sequence of 0's and 1's, the sender converts them into numbers as follows: for each number in the sequence, an x is chosen *at random*. 0 is converted into x^2 mod n, 1 is converted into yx^2. Each 0 or 1 in the sequence can be converted (depending on the choice of x) into one of $(p-1)(q-1)/4$ different numbers. If the message is of length 500 (about one line of ordinary text), and $p, q \approx 10^{100}$, the message can be encoded into $(1/4)10^{100000}$ different possible ciphertexts.

By Lemma 20, 0's are converted to squares, 1's are converted to pseudo-squares. Since the receiver knows p, q, Lemmas 21 and 22 show he can efficiently decode the message.

In the subsequent sections, we will give the essential ideas of Goldwasser & Micali's proof that (assuming QRA) this system will prevent the adversary from obtaining any partial information about the plaintext.

7.2 Weak laws of large numbers

Both the encryption algorithm and the hypothetical algorithms used by the adversary involve random events. We will need a theorem that says that, if an event with probability p is tried r times, the chance that the number of successes is not close to pr is small. The paper uses[7] (p. 293)

Lemma 24 *Let S_r be the number of successes in r tries. For any ψ*

$$\Pr\left(\left|\frac{S_r}{r} - p\right| > \psi\right) < \frac{1}{4r\psi^2}$$

Proof: S_r is a random variable, which is the sum of r independent random variables, each having value 0 or 1. Let V be the variance of S_r. Each of the 0–1 variables has variance $\le 1/4$, so

$$r^2\psi^2 \Pr(|S_r - rp| > r\psi) < V \le \frac{r}{4}$$

Lemma 24 provides a very rough estimate of the probability. An improvement requiring much more work is:

Lemma 25 *With the same notation as Lemma 24,*

$$\Pr\left(\frac{S_r}{r} \ge p + \psi\right)$$
$$\le \frac{1}{\sqrt{2\pi r(p+\psi)(1-p-\psi)}}\left(\frac{(1-p)(p+\psi)}{\psi}\right)\exp\left(-\frac{r\psi^2(1+\psi)}{2(1-p)(p+\psi)}\right) \qquad (*)$$

For comparison, if $p = .5$, $r = 1000$, the probability that there are ≥ 520 successes is .1087. Lemma 24 gives[8] an upper limit of .3125, while Lemma 25 gives .1498. (these figures courtesy of Mathematica)

One reason the paper does not use Lemma 25 is that it does not give a simple formula for how large r would have to be in terms of the other quantities. We will not use this result later, and you should skip to section 7.3 unless you like to manipulate formulas.

[7] The usual central limit theorem cannot be used because it does not tell you how large r must be for the normal distribution to give a good estimate.

[8] We divide by 2 to eliminate the probability of ≤ 480.

Proof: We will assume $pr + r\psi$ is integer. From the binomial theorem:

$$\begin{aligned}
\Pr(S_r \geq rp + r\psi) &= \sum_{i \geq pr+r\psi} \binom{r}{i} p^i (1-p)^{r-i} \\
&\leq \binom{r}{pr+r\psi} p^{pr+r\psi}(1-p)^{r-pr-r\psi}(1+\alpha+\alpha^2+\ldots) \\
&\quad \text{where } \alpha = \frac{p(r-pr-r\psi)}{(1-p)(pr+r\psi+1)}
\end{aligned}$$

$p + \psi \leq 1$ implies $p - p\psi - p^2 > 0$ and

$$\sum \alpha^i = \frac{1}{1-\alpha} = \frac{(1-p)(pr+r\psi+1)}{r\psi+1-p} \leq \frac{(1-p)(p+\psi)}{\psi}$$

which gives the second factor of $(*)$. We use Stirling's formula on the binomial coefficient and group it with the powers of p and $1-p$ to obtain:

$$\left(\frac{1}{\sqrt{2\pi r(p+\psi)(1-p-\psi)}}\right)\left(\frac{p}{p+\psi}\right)^{pr+r\psi}\left(\frac{1-p}{1-p-\psi}\right)^{r-pr-\psi r} \qquad (**)$$

The first factor of $(**)$ is the first factor of $(*)$. We obtain upper bounds on the rest of $(**)$, using

$$-A - \frac{A^2}{2(1-A)} \leq \ln(1-A) \leq -A - \frac{A^2}{2}$$

(the lower bound on $\ln(1-A)$ involves a geometric series)

$$\begin{aligned}
(pr+r\psi)\ln\left(1-\frac{\psi}{p+\psi}\right) &\leq -r\psi - \frac{r\psi^2}{2(p+\psi)} \\
(pr+\psi r-r)\ln\left(1-\frac{\psi}{1-p}\right) &\leq \frac{(r-pr-r\psi)\psi}{1-p} + \frac{(r-pr-r\psi)\psi^2(1-p)}{2(1-p)^2(1-p-\psi)} \\
&= r\psi - \frac{\psi^2 r}{1-p}\left(-1+\frac{1}{2}\right)
\end{aligned}$$

Adding these and using exp gives the remaining factor of $(*)$.

7.3 The magic of sampling

We have 10^6 envelopes. Inside each envelope is a piece of paper with 0 or 1 written on it. If we want to know exactly how many envelopes have each number, we have to open them all. Suppose we want to estimate the fraction of the envelopes of each kind, and we want the proportion to be accurate to within .05. Now we need only open $9(10^5)$ envelopes.

The situation changes dramatically if we only want to estimate the proportion with high probability. If we are willing to accept a .01 probability of an error $> .05$, Lemma 24 implies we only need to open a randomly chosen sample of 10^4 envelopes[9].

The special feature of problems involving squares and pseudo-squares is that sampling is possible. We saw in our discussion of the Rabin system that every number mod n has four square roots. Thus if we choose one of the $(p-1)(q-1)$ numbers x not divisible[10] by p or q and compute x^2 mod n, each square has a $(p-1)(q-1)/4$ chance of being chosen. It is also important that it is possible to sample from Z_n^1 (the set of squares and pseudo-squares) even if p, q are not known.

Lemma 26 *There is an efficient algorithm for deciding if $a \in Z_n^1$.*

The proof of this is difficult, involving "quadratic reciprocity" and the "Jacobi symbol." The algorithm itself is not that complicated, and is given in the RSA paper.

Given this lemma, we can sample in Z_n^1 by choosing x at random and testing if it is in the set. If not, another x is chosen. Since roughly half of $1 \leq x \leq n$ is in Z_n^1, this won't take too long.

The different sampling possibilities we have discussed so far have all assumed that only n was known. If we are given a single pseudo-square y, we can sample among all pseudo-squares by calculating yx^2 for x randomly chosen.

The possibility of doing these various kinds of sampling is closely related to properties 2(a) and (c) in the paper (p. 277).

[9]Lemma 25 and Mathmatica suggest 400 envelopes are enough.

[10]Even though p, q are unknown, the gcd of x, n can be computed.

7.4 Determining algorithm performance by sampling

We are interested in algorithms for deciding whether a given number is or is not a square. As with the algorithm in Section 6, there is some probability that, for a given input a, the algorithm may give the wrong answer.

Let p_a be the probability that a given algorithm gives the correct answer for input a. We are also interested in p_S, which is the average of p_a over all squares a, and p_{PS}, the average over all pseudo-squares, and p_Z, the average of p_a over all $a \in Z_n^1$.

If we are given an algorithm, we can easily determine p_S by running it with input $a = x^2$ on a sample of randomly chosen x and counting the number of times the algorithm answers "this is a square."

The procedure for determining p_Z is more elaborate. Suppose we have an algorithm for which $p_S = .6$. Using Lemma 26, generate a sample of 100 members of Z_n^1, and run the algorithm on each of them. Suppose we get the answer "this is a square" 65 times. There are ~ 50 squares in the sample, on which there have been $.6(50)$ correct responses and 20 incorrect. Pseudo-squares have been identified as squares $65 - 30 = 35$ times, which suggests $p_{PS} \approx 15/50$. Finally $p_Z = (p_S + p_{PS})/2 \approx .45$.

Lemma 24 or 25 can be used to determine the probability that these estimates come within a specified amount.

7.5 Two versions of QRA

1. There is no efficient algorithm for distinguishing squares from pseudo-squares with $p_a > 1 - \epsilon$ for all $a \in Z_n^1$.
2. There is no efficient algorithm with $p_Z > .5 + \epsilon$

It would seem that (1) is not as strong as (2). Note that (2) would rule out an algorithm with $p_S = .9$ and $p_{PS} = .2$. This would be something that says "this is a square" most of the time, occasionally correctly identifying a pseudo-square. However, the paper (p. 293) shows that (1) implies (2).

Suppose we are given an algorithm. We estimate p_S, p_{PS}, p_Z with high probability using the techniques in Section 7.4. To take a specific example, we will assume we find $p_S = .6$, $p_{PS} = .45$. We want to test whether a is a square. Run the algorithm on ax^2 for 1000 randomly chosen x. If a is a square, the algorithm will say "this is a square" ≈ 600 times. If a is a pseudo-square, the answer will be "this is a square" ≈ 550 times.

7.6 Knowing a pseudo-square does not help much

QRA talks about the ability to identify squares when only n is known. In the proposed encryption system, a pseudo-square y is also announced. The paper shows (p. 295) that this does not make the problem easier.

Suppose we have an algorithm which takes as input a, y and tries to decide if a is a square. Assume $p_Z = .55$ whenever y is a pseudo-square. Choose $y \in Z_n^1$ at random, then use the techniques from Section 7.4 to estimate p_Z. Since half the numbers in Z_n^1 are pseudo-squares, you will quickly find a y for which $p_Z = .55$.

7.7 The inability to distinguish two plaintexts

Theorem 5.1 of the paper addresses the issue we mentioned at the beginning of section 7. It shows that if we have an algorithm which can identify messages m_1 and m_2 and efficiently tell the difference between an encryption of m_1 and an encryption of m_2, then we could construct an algorithm which efficiently distinguishes squares from pseudo-squares. Thus (QRA) implies we cannot tell the difference between m_1 and m_2.

Proof:[11] Suppose we are trying to decide whether $a \in Z_n^1$ is a square and that the two distinguishable messages are

$$\begin{aligned} m_1 &= 01001011 \\ m_2 &= 11101101 \end{aligned}$$

Choose 8 x_i randomly and consider the sequences

$$\begin{array}{rrrrrrrr} x_1^2 & ax_2^2 & x_3^2 & x_4^2 & ax_5^2 & x_6^2 & ax_7^2 & ax_8^2 \\ ax_1^2 & ax_2^2 & ax_3^2 & x_4^2 & ax_5^2 & ax_6^2 & x_7^2 & ax_8^2 \end{array}$$

If a is a pseudo-square, these will be randomly chosen encodings of m_1 and m_2. In this case, the performance of our assumed algorithm on the two

[11]The argument we give is a simplification of the one in the paper, in that we do not use the "sampling walk." The more complicated argument seems to be necessary to analyze encryption systems in general, as opposed to those based on squares and pseudo-squares.

sequences (averaged over repeated random choices of x_i) will be different. If a is a square, both sequences will be randomly chosen encodings of the message consisting of all 0's, so the algorithm's response on average to the two sequences will be identical.

7.8 Semantic Security

Theorem 5.2 of the paper shows that there is no property of the plaintext message which can be efficiently estimated by looking at the ciphertext. Typical properties might be "the last bit of the plaintext is 0" or "the number of 1's is twice as much as the number of 0's." In general, a property is defined in the paper as the value of a function $f(m)$ which takes a message as input and gives a number as output. If $f(m)$ is constant for all m, prediction of $f(m)$ is trivial. Similarly, if $f(m)$ is almost constant for almost all m, there is a simple algorithm which will be close to right with high prob

We wish to show that, except in the special cases we've mentioned, there is no efficient algorithm which will predict $f(m)$ from the ciphertext for m. If there were, we could run our algorithm to estimate $f(m)$ on the ciphertext from randomly generated m until we found m_1, m_2 on which the algorithm behaved differently. But this would contradict the result of the previous section.

[The paper points out that it is not assumed that $f(m)$ is an easily computable function. I think this is a minor issue. The theorem really discusses the capabilities of a an easily computable program for estimating f.]

7.9 How to play poker over the telephone

We will not analyze an entire game of poker, but just the task of each player [we will assume only two players] getting dealt cards so that (i) each player gets his cards at random, with all cards equally likely (ii) neither player knows what his opponent has (iii) the players cannot get the same cards. You will probably appreciate the procedure more if you first try to devise a way of doing this yourself.

Several previous attempts to use cryptographic devices for this purpose were flawed[12]. The elaborate procedure we describe is based on some number-

[12]R. Lipton, "How to cheat at mental poker," *Proceedings of AMS Short Course on Cryptography*

theory tools developed in section 3.3 and earlier in this section:

1. If $n = pq$ and a is a square mod n, it has four square roots. If we know roots r_1, r_2 with $r_1 \not\equiv \pm r_2$, we can find p, q.
2. If $p \equiv 3 \pmod 4$, a is a square mod p if and only if $-a$ is not a square (Lemma 22). If we also have $q \equiv 3 \pmod 4$, then $a \in Z_n^1$ if and only if $-a \in Z_n^1$.
3. We can test whether or not $a \in Z_n^1$ without knowing p, q.

Two techniques are used repeatedly. They are also of interest in other applications.

Theorem 27 (random numbers) *B can generate a random number so that A does not know its value now, but can verify it later.*

A "first try" might be for B to generate a random number and give an encryption of it to A, with the key revealed for verification later. This does not work, since A cannot be sure that B chose his number at random.

To insure randomness, A gives B a second number (which A is supposed to choose at random) after receiving B's encryption, and the number used by B is the "exclusive or" of the two:

$$\begin{array}{r} \text{A chooses } 0110001 \\ \text{B chooses } 1011011 \\ \hline \text{B uses } 1101010 \end{array}$$

Even if one of the players does not choose his number at random, the result will be random as long as the other player does.

Theorem 28 *B can ask A a question related to n. The answer to this question may or may not allow B to factor n. At the time the question is asked, A cannot tell whether the answer he gives B is useful or useless, but this can be verified later.*

Proof: A chooses primes $p, q \equiv 3 \pmod 4$, and announces $n = pq$. Using the technique of Theorem 27, B generates a random x, and will ask A for a square root of $a \equiv x^2$. At the time the question is asked, A will know a but not x. B is allowed to specify whether the square root A gives him is or is not in Z_n^1.

If $x \in Z_n^1$ and B specifies that the square root is in Z_n^1, A will give B $\pm x$, which is useless. B can get useful information by specifying that the square root is not in Z_n^1. If $x \notin Z_n^1$, the square root in Z_n^1 will be useful, and the other will be useless.

Since x is randomly chosen, and half the possible x are in Z_n^1 and half are not, A will not be able to guess right more than half the time whether he is being asked for useful or useless information.

The procedure

1. A announces $n_1, \ldots n_{52}$, each of which is a product of two large primes $\equiv 3 \pmod 4$. He encodes the names of the different cards using different n_i and also announces these. [if B finds the factors of one of the n_i, it does not help him identify the other cards] B does the same thing using $m_1, \ldots m_{52}$.

2. To get a card, B asks A one question for each n_i, using the procedure of Theorem 28. 51 of the questions will be useless. The useful question allows B to decode the name of the card he receives. [it is crucial that A will be able to verify the uselessness of the other 51 questions after the game.]

3. B deletes the m_i corresponding to the card he received (this ensures A will not get this card).

4. A gets a card by asking 51 questions about the remaining m_i, of which 50 are useless. He deletes the n_i corresponding to this card.

5. If B gets a second card, he asks 51 questions. He avoids getting the same card twice by not asking a useful question about the same n_i as the first time.

This procedure is too cumbersome to be practical, but it is a good example of the kinds of things that can be done using cryptographic procedures. Current research focusses on other tasks involving exchanges of encrypted and partially encrypted information between two players.

8 Pseudo-random number generators

[This section is based on Blum, Blum, & Shub, "A simple unpredictable pseudo-random number generator," *SIAM J. Computing* 15, 364–383.]

Many programs (e. g., simulations, one-time-pads) make use of numbers that are supposed to be random. A genuine source of randomness might be a subroutine that made calls on something like a built-in Geiger counter. We will be concerned with algorithms that produce a sequence of numbers (usually 0's and 1's) which appears random (precise definition will be given later).

A typical example of such an algorithm is the function `rand()` in the C programming language. Each call updates an internally maintained N using the formula

$$N = N * 1103515245 + 12345 \quad \mathrm{mod}\ 4294967296 = 2^{32}$$

with the output given by $2^{-16}N \bmod 2^{15}$.

I recently wrote a program to roll dice which involved using `rand()` mod 6. In over 100 calls, it never happened that the same number occurred on two consecutive rolls, even though this should have happened about 1/6(100) times! This suggests this particular generator has some problems.[13]

In this section, we will present random number generators for which it can be proved (given assumptions like (QRA)) that such problems will not occur.

8.1 The Quadratic Generator

Let $n = pq$, where p, q are primes $\equiv 3 \pmod 4$. For each prime, a is a square if and only if $-a$ is not a square (Lemma 22). This implies that, if $x \equiv \pm a_1 \pmod p$ and $x \equiv \pm a_2 \pmod q$, there will be exactly one choice which makes x a square mod n. Hence, if b is a square mod n, exactly one of its four square roots will also be a square. This *principal* square root will be denoted by $\sqrt{b}$.

[13]Knuth suggests that a better way to obtain a random number between 0 and $k-1$ is to use k `rand()`$/M$, where M is the maximum value of `rand()`.

The quadratic generator uses a randomly chosen square x (called the *seed*) not divisible by p or q to generate a sequence of 0's and 1's (*bits*). The sequence is a_i mod 2, where $a_0 = x$ and $a_{i+1} \equiv \sqrt{a_i} \pmod{n}$:

$$x \bmod 2 \quad \sqrt{x} \bmod 2 \quad \sqrt{\sqrt{x}} \bmod 2 \quad \ldots$$

(from a practical point of view, it is simpler to generate the sequence starting with the last number and squaring)

As a small example with $n = 589 = 19(31)$ and $x = 81$, the sequence of a_i is

$$81 \quad 9 \quad 586 \quad 175 \quad 112 \quad 443 \quad 214 \quad 237\ldots$$

(note that $\sqrt{9} = -3$, not 3) which gives the sequence of bits 11010101.

8.2 The Next Bit Theorem

It would certainly be undesirable if there were an efficient algorithm which took as input the first k bits of the sequence from the generator and guessed the $(k+1)$-st bit with probability much greater than 1/2. We say a generator satisfies the *Next Bit Condition* if there is no such algorithm.

Theorem 29 *If (QRA) is true, the quadratic generator satisfies the Next Bit Condition.*

Proof: We will show that an algorithm that could predict the $(k+1)$-st bit could be used to distinguish squares from pseudo-squares mod n.

Let $b \in Z_n^1$. The sequence of length k

$$b^{2^k} \quad b^{2^{k-1}} \ldots b^4 \quad b^2$$

can be considered as coming from the quadratic generator with seed the first term of the sequence. If we take this sequence mod 2 and give it to our predictor, we would get a guess as to whether

$$\sqrt{b^2} \equiv 0 \text{ or } 1 \pmod{2}$$

which has probability $> 1/2$ of being right. The principle square root of b^2 is b if b is a square, $n - b$ if b is a pseudo-square. Since $b \not\equiv n - b$ mod 2, the information from the predictor gives us a guess as to whether b is a square.

8.3 The Efficient Test Theorem

When we are given a sequence of bits from a pseudo-random number generator, we often test the quality of the generator by doing things like counting the fraction of 0's, the fraction of subsequences of the form 111, etc.

A *test* is defined to be an efficiently computable function T which takes as input a sequence of bits of length m and gives as output a number between 0 and 1. Define

$$\begin{aligned} A_r &= \text{Average over all sequences } s\ \{T(s)\} \\ A_g &= \text{Average over } s \text{ from the generator } \{T(s)\} \end{aligned}$$

These averages both involve finite operations— A_r involves adding up $T(s)$ over the 2^m possible s and dividing. Similarly A_g deals with an average over all possible seeds (presumably the number of possible seeds is much less than 2^m).

It would take too much time to calculate A_r, A_g exactly, but they can be estimated with high probability using the sampling ideas in section 7.3.

A generator is said to *satisfy* the test T if A_g is close to A_r, i. e., T cannot tell the difference between sequences from the generator and genuinely random sequences. [we are being deliberately vague about the precise definition of "close."]

Theorem 30 *If a generator satisfies the Next Bit Condition, it satisfies all efficiently computable tests T.*

Proof: We will show that, if we had T with A_r significantly different from A_g, then for some k, T could be used to predict the $(k+1)$-st bit from the first k bits with probability somewhat larger than $1/2$. This would contradict the Next Bit Condition.

If s is a sequence of i bits, let f_s be the fraction of all possible seeds whose first i bits are s. For some s, we may have $f_s = 0$. Note that

$$A_g = \sum_s f_s T(s)$$

where the sum is over all s of length m.

The proof involves two steps:

1. Identify a $0 \leq k \leq m-1$ such that the behavior of $T(s)$ depends in a significant way on the $(k+1)$-st bit of s.
2. Use T to make a prediction for the $(k+1)$-st bit.

The proof of step 1 uses ideas similar to the "sampling walk" used to prove Theorem 5.1 in the Goldwasser-Micali paper. Define

$$A_i = \sum_{s,t} f_s 2^{i-m} T(s \circ t)$$

where the sum is over all s of length i and t of length $m-i$, with $\circ$ meaning to combine s and t to create a sequence of length m. A_i is the expected value of T applied to a sequence in which the first i bits come from the generator (using a randomly chosen seed), with the remaining bits coming from a genuinely random source.

Note that $A_0 = A_r$, $A_m = A_g$, and that all A_i can be estimated with high probability using sampling. Since

$$|A_r - A_g| \leq \sum_{1}^{m} |A_i - A_{i-1}| \text{ there is } k \text{ with } |A_{k+1} - A_k| \geq |A_r - A_g|/m \quad (2)$$

This completes step 1.[14]

In step 2, we are concentrating on a specific sequence s of length k, where k satisfies (2). We wish to use the behavior of T to predict whether the $(k+1)$-st bit should be 0 or 1. Intuitively, we ask T which of the two possibilities would make the sequence look more random.

We will need to look at the analogues of the averages A_k and A_{k+1}, restricting attention to those sequences which begin with s:

$$\begin{aligned} A_k(s) &= \sum_t 2^{k-m} T(s \circ t) \\ A_{k+1}(s) &= \sum_t (f_{s\circ 0}/f_s) 2^{k+1-m} T(s \circ 0 \circ t) + \\ &\quad \sum_t (f_{s\circ 1}/f_s) 2^{k+1-m} T(s \circ 1 \circ t) \end{aligned}$$

[14]Instead of estimating all the A_i, we could begin by estimating $A_{.5m}$. We would next estimate either $A_{.75m}$ or $A_{.25m}$, depending on whether $A_{.5m}$ was closer to A_0 or A_m.

The definition of $A_{k+1}(s)$ is based on the idea that $s \circ 0$ and $s \circ 1$ are the only sequences of length $k+1$ which begin with s. Note that, for $i = k$ or $k+1$, $A_i = \sum_s f_s A_i(s)$, where the sum is taken over all s of length k.

$$\begin{aligned} \text{Define} \quad A_{s,0} &= \sum_t 2^{k+1-m} T(s \circ 0 \circ t) \\ A_{s,1} &= \sum_t 2^{k+1-m} T(s \circ 1 \circ t) \end{aligned}$$

These are the expected values of T for a sequence which begins with s, has either 0 or 1 as its $(k+1)$-st term, and continues randomly. They can be estimated by sampling. Let p_s be the fraction of the seeds which give s as the first k bits which give 0 as the $(k+1) - st$ bit (thus $p_s = f_{s \circ 0}/f_s$). Then

$$\begin{aligned} A_k(s) &= \frac{1}{2}A_{s,0} + \frac{1}{2}A_{s,1} && (3) \\ A_{k+1}(s) &= p_s A_{s,0} + (1 - p_s)A_{s,1} && (4) \end{aligned}$$

If we could estimate p_s from (4), it would be simple to predict the next generated bit after s. Unfortunately, we cannot efficiently estimate $A_{k+1}(s)$. The problem is that we would have to sample among the seeds which generate s, and there is no easy way to find such seeds. Instead, we must find a way to use the information that the average of $A_{k+1}(s)$ is A_{k+1}, which we can estimate.

The (far from obvious) idea will be to have the prediction of the $(k+1)$-st bit itself be random. As we will see below, the probabilities can be assigned to the two possible predictions can be chosen so that the expected number of correct guesses looks like the right-hand side of (4).

We will assume $A_{k+1} > A_k$ [remember, we chose k so that the difference between the two is significant]. The other case can be handled similarly. If $A_{s,0} > A_k(s) > A_{s,1}$, we would expect sequences beginning with $s \circ 0$ to look more like things from the generator than sequences beginning with $s \circ 1$. Our prediction for the next bit following s will be random, given by

$$\text{Predict} \begin{cases} 0 \text{ with probability } \frac{1}{2} + A_{s,0} - A_k(s) \\ 1 \text{ with probability } \frac{1}{2} + A_{s,1} - A_k(s) \end{cases}$$

[The probabilities add to 1 by equation (3).]

The probability that the prediction for input s is correct is

$$\begin{aligned} p_s\left(\frac{1}{2}+A_{s,0}-A_k(s)\right)+(1-p_s)\left(\frac{1}{2}+A_{s,1}-A_k(s)\right) &= \\ \frac{1}{2}+p_sA_{s,0}+(1-p_s)A_{s,1}-A_k(s) &= \frac{1}{2}+A_{k+1}(s)-A_k(s) \end{aligned}$$

When we average over all seeds resulting in all possible s, we get a correct prediction with probability $1/2+A_{k+1}-A_k$, which, by (2), is significantly greater than 1/2. [15]

8.3.1 A consequence involving symmetry

The Next Bit Condition was stated in a way that clearly distinguished the beginning of the pseudo-random sequence from the end. By contrast, the Efficient Test Theorem treats a pseudo-random sequence in a completely symmetrical way. From that point of view, it does not matter which end of the sequence is used to start the construction. This leads to

Corollary 31 *Let $n = pq$. Start with a random $1 \le x \le n-1$ not divisible by p or q. Let $a_0 = x$, $a_{i+1} \equiv a_i^2 \pmod n$. The sequence of bits given by a_i mod 2 satisfies all efficient tests.*

9 Further results on pseudo-random generators

The two main results of the preceding section were the Next Bit Theorem and the Efficient Tests Theorem. The former depended on (QRA) and facts about squares and pseudo-squares. The latter was an abstract result about properties of arbitrary generators. Our first result is an abstract version of the Next Bit Theorem.

[15] Thanks to R. Sengupta for pointing out the importance of the expression for $A_{k+1}(s)-A_k(s)$.

9.1 Hard-Core Predicates and Pseudo-Random Numbers

We have seen several functions f with the property that $f(x)$ was easy to compute but f^{-1} was difficult. Such an f is called a *one-way function*.[16] A *predicate* is a property B which is true or false for any x. Typical examples might be "is an odd number" or "is ≤ 50." A *hard-core predicate* for a function f is a predicate such that

1. there is an efficient algorithm for deciding whether $B(x)$ is true
2. if we are given $f(x)$, there is no efficient algorithm for guessing whether $B(x)$ is true which has a probability much greater than 1/2 of being right.

The example we used in the quadratic number generator was

$$f(x) \equiv x^2 \bmod n \qquad B(x) = \text{ "}x\text{ is odd"}$$

where we are looking only at those x which are squares.

Theorem 32 *If B is a hard-core predicate for f, the random number generator in which a seed x is chosen at random, giving the sequence of bits*

$$B(x) \quad B(f^{-1}(x)) \quad B(f^{-1}(f^{-1}(x)))\ldots$$

satisfies the Next Bit Condition.

As a corollary, the sequence $B(x)$, $B(f(x))$,... satisfies all efficient tests, using the arguments in the preceding section.

Proof: If there were a program which took as input a sequence of bits and could guess the next bit, we could get a good guess on $B(x)$ with $f(x)$ known by asking the next-bit predictor what would occur next in the sequence

$$B(f^{(k)}(x)) \quad B(f^{(k-1)}(x)\ldots B(f(f(x))) \quad B(f(x))$$

(this is really the same proof as in the case of the quadratic generator).

[16] In later work, we will be defining a one-way function to be such that there is no efficiently computable g with $g(x) = f^{-1}(x)$ for most x.

9.2 Construction of hard-core predicates

It seems much harder to find examples of hard-core predicates than of one-way functions. The latter can be obtained from discrete logarithms, subset-sum problems, and elsewhere. It is difficult to prove that a property cannot be guessed with probability much larger than 1/2. This makes the following result[17] interesting.

Theorem 33 *Let f be a one-way function which maps sequences of bits of length n to sequences of length n. For each $S \subset \{1, \dots n\}$ and n-bit x define $B(S, x)$ to be true if the number of 1's in positions in x specified by S is even. There is no efficient algorithm which can guess $B(S, x)$ given S and $f(x)$ with probability much greater than $1/2$.*

This result does not rule out the possibility that, for some specific S, it may be possible to guess $B(S, x)$ in an efficient way. However, it would not be possible to do something like guess $B(S, x)$ with probability .6 for 10% of all possible S, and probability .5 for all other S. [this would imply we would be correct about $B(S, x)$ overall with probability .51]

To prove this, we shall consider the following scenario: We are trying to determine an unknown $x \in \{0, 1\}^n$. We have an oracle $A(S)$ which is supposed to tell us $B(S, x)$. The oracle may lie, but must tell the truth more than half the time. Theorem 34 says we can efficiently enumerate a not-too-large set which probably includes x.

Theorem 34 *Suppose that $A(S) = B(S, x)$ for at least $1/2 + \epsilon$ of all $S \subset \{1, \dots n\}$. We can enumerate $U \subset \{0, 1\}^n$ in time polynomial in ϵ^{-1} such that $x \in U$ with probability close to 1.*

To deduce Theorem 33 from Theorem 34, suppose we had an efficient algorithm for guessing $B(S, x)$. We obtain U, which cannot be too large given the time bound. If we are given the value of $f(x)$, we can evaluate f for all $x \in U$ to identify the correct x with high probability, which would mean f is not a one-way function.

The construction of U proceeds in stages. At stage k ($1 \leq k \leq n$), we enumerate $U_k \subset \{0, 1\}^k$ which includes (with probability close to 1) the first k digits of x.

[17]O. Goldreich and L. Levin, "A Hard-Core Predicate for Every One-Way Function," *Symposium on Theory of Computation* (1989)

We will consider k fixed for the rest of this section. Define $L = \{1, \dots k\}$ and $R = \{k+1, \dots n\}$. If α and β are true or false, $\alpha == \beta$ is defined to be 1 if α and β are both true or both false, 0 otherwise. If h is defined for all $S \subset L$, we will use $\underset{S}{\text{Avg}}\ h(S)$ to represent the average value of h, in other words, $2^{-k} \sum_S h(S)$, with similar definitions for other collections of S. The hypothesis of Theorem 34 can be written as

$$\underset{S \subset \{1, \dots n\}}{\text{Avg}} \quad (A(S) == B(S, x)) \geq 1/2 + \epsilon$$

Let $v \in \{0,1\}^k$. If there is a $w \in \{0,1\}^{n-k}$ with $x = v \circ w$ [i.e., v is the first k bits of x], then

$$1/2 + \epsilon \leq \underset{D \subset R}{\text{Avg}} \left(\underset{C \subset L}{\text{Avg}} \Big(A(C \cup D) == B(C \cup D, v \circ w) \Big) \right)$$

$$\text{Define} \quad T(v, D) = \underset{C \subset L}{\text{Avg}} \Big(A(C \cup D) == B(C, v) \Big)$$

Recall that $B(S, x)$ is true if and only if the sum of the bits of x corresponding to S is even. This implies

$$\underset{C}{\text{Avg}} \Big(A(C \cup D) == B(C \cup D, v \circ w) \Big) = \begin{cases} T(v, D) & \text{if } B(D, w) \text{ is true} \\ 1 - T(v, D) & \text{if } B(D, w) \text{ is false} \end{cases}$$

Thus, if $x = v \circ w$, then

$$\underset{D \subset R}{\text{Avg}} \Big(|T(v, D) - 1/2| \Big) \geq \epsilon \tag{5}$$

To test whether a given v satisfies (5) requires looking at all possible C and D, which would take too much time. However, as in section 7.3, we can take not-too-large random samples from all possible C and D, with a high probability of correctly deciding whether v satisfies (5). Let N be the number of different D used in the sampling.

At stage $k - 1$, we have obtained U_{k-1} which includes the first $k - 1$ bits of x with high probability. To create U_k, we take each member of U_{k-1}, add 0 and 1 to it, and identify using sampling which of the resulting k-bit strings satisfy (5).

This process would take too much time if the number of strings doubled at each step. To complete the proof, we use Lemma 35 to show that, for every D, the number of v with $|T(v,D)-1/2| \geq \epsilon$ is not too large, specifically there are at most $(2\epsilon)^{-2}$ such v. In order to to be included in U_k, v must satisfy $|T(v,D)-1/2| \geq \epsilon$ for at least one of the sets D used in the sample, which gives a bound of $N(2\epsilon)^{-2}$ on $|U_k|$.

Lemma 35 *For any D, $\sum_v (T(v,D)-1/2)^2 = 1/4$, where the sum is over all $v \in \{0,1\}^k$.*

Proof: D is a constant and may be ignored. We will argue by induction on k. Define $L' = \{1, \ldots k-1\}$. The function A may be represented by A_1, A_2 with

$$A_1(C \cup D) = A(C \cup D) \quad A_2(C \cup D) = A(C \cup \{k\} \cup D) \quad \text{for all } C \subset L'.$$

$$\begin{aligned}
\text{Define} \quad T_i(v,D) &= \underset{C \subset L'}{\text{Avg}} \Big(A_i(C \cup D) == B(C,v)\Big) \quad i = 1,2 \\
\text{Then} \quad T(v,D) &= \frac{1}{2}\left(T_1(v,D) + T_2(v,D)\right) \quad \text{if} \quad v_k = 0 \\
&= \frac{1}{2}\Big(T_1(v,D) + (1 - T_2(v,D)\Big) \quad \text{if} \quad v_k = 1
\end{aligned}$$

We divide $\{0,1\}^k$ according to whether $v_k = 0$ or 1 to obtain

$$\begin{aligned}
\sum_v (T(v,D)-1/2)^2 &= \sum_{v_k=0} \left(\frac{1}{2}T_1(v,D) + \frac{1}{2}T_2(v,D) - \frac{1}{2}\right)^2 \\
&\quad + \sum_{v_k=1} \left(\frac{1}{2}T_1(v,D) + \frac{1}{2}\left(1 - T_2(v,D)\right) - \frac{1}{2}\right)^2 \\
&= \sum \frac{1}{2}\left[\left(T_1(v,D) - \frac{1}{2}\right)^2 + \left(T_2(v,D) - \frac{1}{2}\right)^2\right] \\
&= \frac{1}{2}\left[\frac{1}{4} + \frac{1}{4}\right] \quad \blacksquare
\end{aligned}$$

9.3 A recent pseudo-random number generator

R. Impagliazzo and M. Naor[18] proposed a pseudo-random number generator that involved only addition:

Choose $1 \leq a_i \leq 2^n$ randomly, for $1 \leq i \leq k$, with $k < n < (1+\epsilon)k$. Choose $S \subset \{1, \ldots k\}$ randomly. Add the a_i corresponding to S mod 2^n to obtain a sequence of n bits. Use the first $n-k$ bits as output from the generator. The remaining k bits give you a new S, for which you obtain a new sum.

This generator is more efficient than the one in section 8.1, since it uses simpler operations and produces more than one bit of pseudo-random output per iteration. The paper establishes connections between the existence of efficient tests (in the sense of section 8.3) and the average-case difficulty of subset-sum problems. We will not try to give a precise statement of the result, but will try to indicate some of the main ideas.

Suppose there were an efficient algorithm $\mathcal{A}$ which looked at the a_i and at $b \in \{0,1\}^n$ and guessed whether b came from a sum of the a_i with probability greater than 1/2. We will show that such an algorithm could be used to determine whether b came from a sum with probability close to 1.

In the case in which $b = \sum_{i \in S} a_i$ we show that $\mathcal{A}$ can be used to guess, for each $R \subset \{1, \ldots, n\}$, whether $|R \cap S|$ is even or odd with probability greater than 1/2. Theorem 34 can then be used to identify S with probability close to 1.

To guess the parity of $|R \cap S|$, we make a guess j for the size of this set and choose a random x. We have $\mathcal{A}$ look at a problem in which b is replaced by $b - jx$ and a_i is replaced by $a_i - x$ for each $i \in R$. If j is a correct guess, $\mathcal{A}$ will see a sum of a subset. Otherwise, $b - jx$ will be a random number, and we assumed $\mathcal{A}$ could distinguish between these two possibilities with probability greater than 1/2.

[18] "Efficient Cryptographic Schemes Provably as Secure as Subset Sum," *Symposium on Foundations of Computer Science*, 1989