Pseudorandomness is the most important concept in cryptography. No fewer than three Turing Awards were given out for studying it, at least in part: Manuel Blum's (1995), Andrew Chi-Chih Yao's (2000), and Avi Wigderson's (2023). Michael Rabin might well have received a fourth one, had he not already collected his in 1976 for inventing finite automata.

Pseudorandomness is a tricky notion. The name is somewhat misleading. Merriam-Webster defines *pseudo* as "being apparently rather than actually as stated." In the context of pseudorandomness, this means something that *appears* random but is not actually random. That definition captures part of the idea, but it is not quite right for cryptography. Nevertheless we'll take it as our starting point.

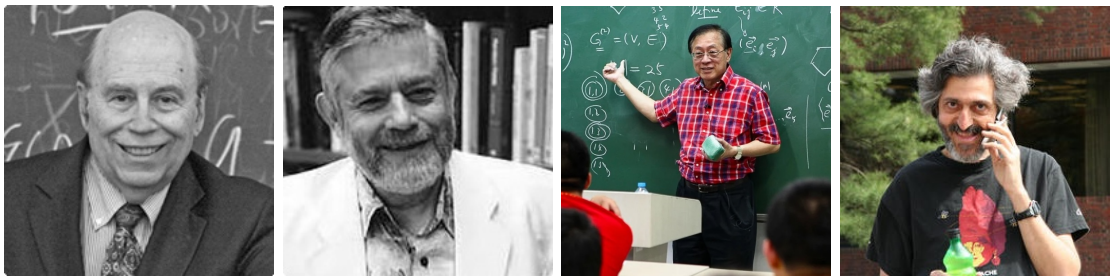

FIGURE 1: Michael Rabin, Manuel Blum, Andrew Yao, and Avi Wigderson. (From ACM, *South China Morning Post*, and IAS.)

At this point we change convention. We will use bits rather than English letters to write messages and ciphertexts. Bit representations are convenient not only because they are more friendly to computers. Unlike letters of the alphabet, bits are indivisible. This makes them easier to reason about.

1 Pseudorandom sequences

Here is a trick question. Which of these three bit sequences is random, and which is not?

$$r_1 = 1110100001101110111010011001111110100010110001001101000010010001$$
$$r_2 = 1111111111111111111111111111111111111111111111111111111111111111$$
$$r_3 = 1001100010111100010110111011000001100011001110011110100000101111$$

The first and third sequences look pretty random to me. The second one definitely doesn't. Is that correct?

A cynic might argue that all three are equally random. If we drew a 64-bit sequence out of all possibilities, each of these three outcomes would have had the same probability, namely 2^{-64}. It would be unfair to favor any one over any other as more random.

This argument is logically flawless, but doesn't align with our intuition about randomness. Sequence r_2 possesses way too much "order" to come out random. But how can we tell apart ordered sequences from orderless ones?

Mathematician Andrei Kolmogorov came up with an appealing criterion. Ordered sequences admit a short description, whereas random ones do not. The sequence r_2, for instance, can be succinctly described as 64 consecutive ones, or the string 1^{64}. In contrast, r_1 doesn't seem to have any such short description. Or does it?

I don't know for sure about r_1, but r_3 certainly does. r_3 happens to consist of the leading digits of the binary representation of $\pi = 3.1415\ldots$ Although r_3 may look random, it is far from it.

Kerchoff's postulate demands that I reveal the algorithm I used to obtain all these sequences. Now that you know the algorithm I used to produce r_3, you can tell that is not random.

How about r_1? To obtain r_1, I asked the website `random.org` for four random numbers between 0 and $65535 = 2^{16} - 1$ and glued together their bit representations. Assuming that `random.org` furnishes pure randomness as advertised, sequence r_1 is truly random.

For cryptographic purposes, r_1 would serve as a fantastic one-time pad. The objective of today's lecture, however, is to overcome the principal inconvenience of the one-time pad: its long description. Sequence r_1 is too perfect to qualify as pseudorandom.

In conclusion, none of these three sequences are pseudorandom. We will have to go through some trials and tribulations to see an example of one that is. Here is a summary of the criteria that a pseudorandom sequence must meet:

Definition 1. A bit sequence is *pseudorandom* if

1. The sampler that produced it is known to everyone,
2. No distinguisher that expends reasonable effort can tell it apart from pure random bits,
3. It is not truly random: Some distinguisher can tell it apart from pure random bits.

Criteria 2 and 3 appear contradictory. One asks that no distinguisher can tell pseudorandom apart from random, while the other asks that some distinguisher can do that. The difference is in the effort. In principle, a pseudorandom sequence can always be distinguished from a truly random one. Any method that does so, however, imposes unreasonable effort on the distinguisher.

2 Distinguishers

Paraphrasing Tolstoy,

> All random sequences are alike; each non-random sequence is not random in its own way.

Table 1: Independent outputs of three samplers.

Batch A	Batch B	Batch C
10100110	11110101	01101000
00010111	11100000	10011001
10100001	00111101	11000101
00010000	10111010	00011010
10001100	11110110	10011010
00010001	01111111	00110111
00010011	01101110	11001110
00011111	00100100	10101111
00101111	10000110	01000101
00001011	10011111	10111100
00001001	10100110	01111001
10010000	11100110	11010110
10001010	10111001	10110100
00001000	01111101	11001100
10101001	10100111	00001000
00111011	11001110	10100100
00111110	11110010	01011110
00111111	11111101	10111110
10111110	10111111	01110000
00101110	11000110	01111010

To appreciate the many ways in which a sequence can fail to be pseudorandom, let's look at the examples in Table 1. Each column contains 20 independent outputs of some (unknown for now) sampler. None of them is pseudorandom. To explain why, we must demonstrate how to distinguish each of them from pure random bits.

Let's start with **batch A**. If you look closely, the second bit is zero in all samples. Indeed, the sequences were sampled using the following code:

```
def A():
    x = random_bits()
    x[1] = 0
    return x
```

Let's now consider the alternative hypothesis that the twenty samples are purely random. Could the second bit end up being zero in all of them? Such an outcome is possible, but exceedingly unlikely. If batch A was truly random, the probability of all second bits being zero would have been 2^{-20}.

The distinguisher D_A

```
def D_A(x):
    return x[1] == 0
```

accepts samples of A with probability 1, but accepts truly random strings with probability 1/2 only. Consequently, it distinguishes outputs of A from truly random strings with advantage 1/2.

This may not sound like much, but it is. For twenty samples, the advantage grows to a sizeable $1 - 2^{20} \approx 0.9999990$, and for 200 samples it is effectively 1. This example illustrates a general principle: If Eve has even a small but noticeable advantage in distinguishing a single sample from a truly random one, her advantage becomes enormous for a large batch of samples.

Batch B, in contrast, does not have a fixed bit. Each of its bits is sometimes zero and sometimes one. How often exactly? Here are the statistics:

bit i	1	2	3	4	5	6	7	8
$\Pr(x_i = 1)$	0.75	0.55	0.80	0.55	0.50	0.80	0.65	0.45

If the samples were truly random, each of these probabilities should be around one half. We would expect some fluctuation around this half owing to variance. Some of the bit biases in batch B are quite large. Can these be accounted for by variance? Or does sampler B produce truly atypical outcomes?

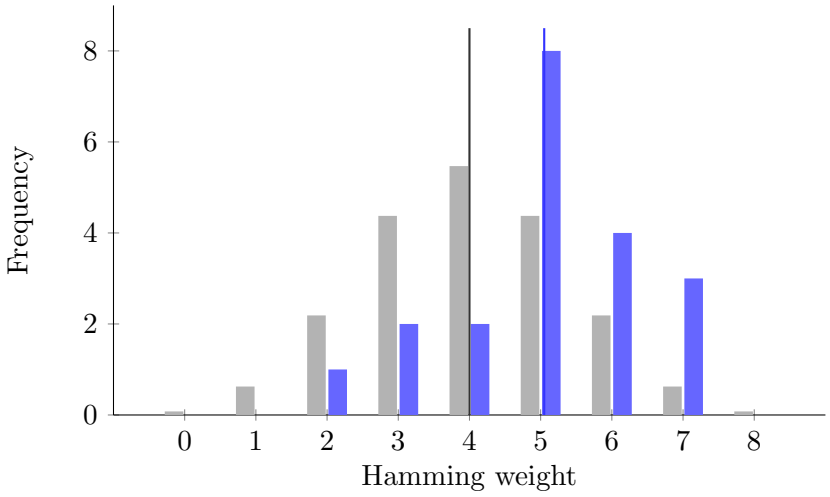

FIGURE 2: Hamming weight histogram of batch B samples (blue) and purely random (grey).

Addressing this question properly requires somewhat advanced knowledge of statistics. Barring that, we can gain some insight by looking at the data. Figure 2 shows two histograms. The blue one represents the Hamming weight (the total number of ones) among the samples in batch B. The gray one shows the Hamming weights that we would expect to see in a large sample of pure random sequences. The blue histogram exhibits a clear shift to the right.

The vertical lines represent the mean values of the two data sets. The mean Hamming weight is 5.05 for batch B and 4 for truly random sequences. A sensible distinguisher D_B might test if the average Hamming weight exceeds, say 4.5.

Sampler B produces a random output conditioned on a majority of the bits x_1, x_3, x_6 being set to one. The conditioning biases those bits towards one, resulting in outputs with higher than normal Hamming weight. Distinguisher D_B has advantage about 0.215 for a single sample. The advantage grows to about 0.751 for 20 samples and 0.998 for 200 samples.

There are two lessons to be drawn from this example. First, a pseudorandom sequence must avoid not only perfectly regular patterns, but any observable statistical irregularity. Second, such irregularities can often be detected even if the code of the sampler is censored, provided the distinguisher has access to enough samples. Pseudorandomness is challenging to achieve not because the sampler is known to the distinguisher, but because the distinguisher has a virtually infinite menu of statistical tests to try.

Finally, let's look at **batch C**. Here are the bit biases.

bit i	1	2	3	4	5	6	7	8
$\Pr(x_i = 1)$	0.55	0.50	0.50	0.60	0.65	0.60	0.45	0.30

Bit x_8 seems to be unusually biased. Is this a statistical fluke? One way to find out is to look at more samples. After taking another 200 samples, the probability of x_8 being one shoots up to 0.49. That is not unusual.

Another statistic we can examine is the frequency of digrams. In a purely random bit sequence, all four digrams `00`, `01`, `10`, and `11` are equally likely at any pair of positions. Here are the frequences for 200 samples of C:

bits	12	23	34	45	56	67	78
`00`	0.280	0.275	0.330	0.265	0.240	0.265	0.275
`01`	0.235	0.260	0.190	0.310	0.230	0.230	0.260
`10`	0.255	0.245	0.245	0.205	0.255	0.270	0.235
`11`	0.230	0.220	0.235	0.220	0.275	0.235	0.230

All observed frequencies are between 0.2 and 0.3. There appears to be no unusual deviation from the expected 1/4.

All digrams of sampler C are in fact equally likely. Yet, its output is not pseudorandom: x_2, x_3, and x_6 always add up to zero modulo two. For a pure random string, this would only happen with probability half. The two are therefore distinguishable with advantage 1/2.

A function that computes the modulo-2 sum of a subset of bits is called a *parity*. There are as many parities as there are subsets of bits: 2^8 for an 8-bit sequence, and in general 2^n for an n-bit sequence. These include the parity of the empty subset, which is the constant 0.

Parities occupy a special place among distinguishers. If x is purely random, all parities save the empty one are unbiased. They are equally likely to evaluate to `0` and to `1`. It turns out that pure random sequences are the only ones with this property. In the contrapositive, if a sequence can be distinguished from random, then one of the parities can do the distinguishing. (The catch is that the distinguishing advantage of this parity could be quite small.) For this reason, the behavior of parities is important throughout cryptography.

Parities are also useful for cryptanalysis. Suppose Eve suspected that some parity in batch C is fixed, but she doesn't know which one. There is an algorithm called modular Gaussian elimination that will

find this parity for her. Any sampler of pseudorandom bits must thwart this distinguishing strategy. In a pseudorandom sequence, no subset of bits can have a fixed parity.

There is another lesson to be drawn from this example. Statistical analyses often look for *local correlations* in data. One of the bits may be biased. If not, perhaps some pair of bits is correlated. For instance, if bit x_2 indicates smoking and bit x_5 indicates cancer, an unusual frequency of pairs of `11`s among x_2x_5 might support a correlation between smoking and cancer. Sampler C exhibits no such correlations: Every pair of outputs is equally likely in any pair of positions. Yet, C does not produce pseudorandom sequences. Avoiding local correlations is necessary, but not sufficient, for pseudorandomness.

After seeing all these non-examples of pseudorandom sequences, you must be dying to know how one looks. I will delay satisfaction for two reasons.

First, all examples I know are rather complicated. One possible starting point is the failures of sampler C. To get around them, we can fix some parities to ensure that the sequence is not truly random, but then flip each bit with some small probability to disable modular Gaussian elimination. The precise implementation details are not relevant for us.

There is a second, conceptual reason that makes pseudorandomness so elusive. Pseudorandom sequences are widely believed to exist, but this is not known with mathematical certainty. The most famous open problem in the theory of computing is the P versus NP question. The question asks whether for all problems whose solutions can be *verified* without much effort, the solutions can also be *found* with comparable effort. Most computer scientists believe not, but there are contrarians. If the contrarians are right, pseudorandom sequences cannot exist, and all encryption schemes are as useless as Caesar's.

Even if the contrarians happen to be wrong there is an important message in all of this. The encryption schemes that we will study in the next lectures are by and large believed to be extremely secure. Internet commerce and the wealth of nations depends on that. We cannot rule out, however, that one day some brilliant mind will find a clever way to break each and every one of them. The dustbin of history overflows with broken cryptosystems.

3 Pseudorandom generators

In theory, once we have a sampler for pseudorandom sequences, we can use this sampler to build secure encryption. In practice, it is convenient that the sampler is of a specific type—a pseudorandom generator.

Recall that a sampler is a deterministic device that converts pure randomness into an output sequence. To describe pseudorandom generators we will restrict the amount of pure randomness that is available to the sampler. We may therefore view the sampler as a *function* that takes ℓ purely random bits of input and produces n bits of output. The random input to a pseudorandom generator is called its *seed*.

Definition 2. A *pseudorandom generator* is a sampler with two properties:

1. Its seed is shorter than its output, i.e., $\ell < n$, and
2. No distinguisher can tell apart its output from n pure random bits with reasonable effort.

As usual, the algorithm executed by the pseudorandom generator is known to the distinguisher.

There appears to be a curious discrepancy between Definitions 1 and 2. Both require effective indistinguishability from random bits. Definition 1, however, also posits that pseudorandom sequences are in principle distinguishable from pure randomness. There is no such provision in Definition 2. Why is that?

The reason is that this condition is met by design: The output sequence of a pseudorandom generator can never be purely random. It is worth understanding why.

To be specific, let's suppose the pseudorandom generator G converts $\ell = 3$ bits of seed into $n = 4$ bits of output. The behavior of G can be completely specified by listing its outputs in some predetermined order, for example

k	000	001	010	011	100	101	110	111
$G(k)$	1011	0001	1100	1100	0100	1001	0000	0101

No matter how we choose G, there can be at most $2^3 = 8$ distinct sequences in the second row. (In this example there are 7 because 1100 repeats.) On the other hand, there are $2^4 = 16$ possible 4-bit sequences. This leaves $2^4 - 2^3 = 8$ sequences that never occur as values of G.

This provides a strategy for Eve to distinguish outputs of G from pure random sequences. Given a test sample x, if it occurs in the list of possible values of G, Eve accepts it. Otherwise, she rejects it.

If x were a sample of G, it would certainly be present in the list of possible values. Eve accepts it with probability 1. If x was pure randomness, however, it is in the list with probability at most $2^3/2^4 = 1/2$: There are at most 2^3 items in the list, but 2^4 possible choices for x. In conclusion, Eve's advantage in distinguishing outputs of G from random is at least $1 - 1/2 = 1/2$. So the output of G *can* be distinguished from random.

To execute her strategy, however, Eve had to search over all $2^3 = 8$ possible outputs of G as a potential match for x. This is all good for $\ell = 3$ bits of seed, but quite impractical when, say, $\ell = 1024$, which is a common parameter choice in applications. Eve would then need to check membership into a list of size 2^{1024}, an enormous number.

One striking aspect of Eve's brute-force attack is that her effort is dramatically affected by the seed length, but virtually independent of the output length. Whether G extends 1024 bits of seed into 1025 bits of output or into a million bits, Eve would be searching over the same number 2^{1024} of possible outputs.

Length extension

Cryptographic theory explains why the output length of the pseudorandom generator is effectively irrelevant. The quantity that really matters is its security, that is the effort that it takes Eve to distinguish its samples from random.

Suppose that we have somehow managed to get our hands on a pseudorandom generator G. To be specific, suppose it converts its $\ell = 100$ bit seed into $n = 400$ output bits. We can then apply this G to obtain as many pseudorandom bits as we want.

Here is how. The seed has 100 bits. After running G, we have 400 pseudorandom bits. We output the first 300 bits. The remaining 100 are used to seed G again, giving out 400 fresh pseudorandom bits. The total output length has now increased from 400 bits to 700 bits.

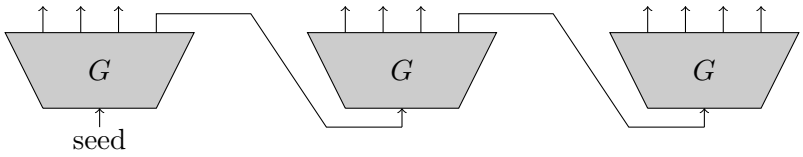

FIGURE 3: Length extension.

There is no reason to stop at a single iteration. We can keep going for as long as pseudorandom bits are needed (see Figure 3). Any pseudorandom generator, even one that expands its seed by a single bit, can be length extended into a virtually infinite pseudorandom sequence without changing the seed.

Private-key encryption

We now have the main ingredient in place for private-key encryption: A long pseudorandom sequence. Alice and Bob can use this sequence in lieu of their one-time pad. To agree on the sequence, they no longer have to share it in its entirety in advance. It is enough for them to remember the seed.

The scheme is shown in Figure 4. Messages and keys are bit sequences, and addition is modulo 2. The main component is the pseudorandom generator G. The key is its seed k. To encrypt, Alice adds her

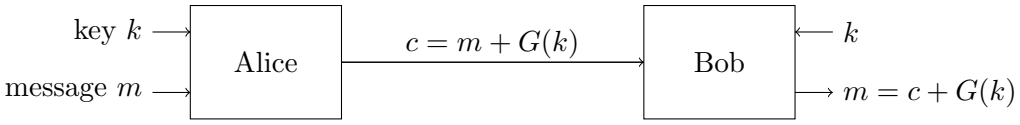

FIGURE 4: Private-key encryption with a pseudorandom generator G.

message (bitwise modulo 2) to the *output* of G. To decrypt, Bob does the same.[1]

In this scheme, the key is shorter than the message. The former is the input to the pseudorandom generator, while the latter is as long as its output. Shannon told us that this scheme cannot be perfectly secure. Yet, for all practical purposes, it is fantastically secure.

To understand why let's recall message indistinguishability. Eve wants to distinguish encryptions of some two messages m_1 and m_2. These have the form $c_1 = m_1 + G(k)$ and $c_2 = m_2 + G(k)$. If she can distinguish these two, she can distinguish at least one of them from a pure random sequence r. Suppose for instance that she can tell c_1 from r. Then she can distinguish $c_1 + m_1$ from $r + m_1$. But $c_1 + m_1$ equals $G(k)$, while $r + m_1$ is a purely random sequence, no matter what m_1 was. So she can distinguish $G(k)$, the output of the pseudorandom generator, from pure randomness. G could not have been pseudorandom in the first place.

The details of this reduction are not important for us. The key point is that by replacing the one-time pad by a long sequence that is pseudorandom but can be derived from a fairly short key—the output of a pseudorandom generator—Alice and Bob give away little in security.

At this point we should verify that no crime was committed. The scheme is compatible with the postulates of cryptography. Eve gets to see all algorithms, including the pseudorandom generator G, as required by Kerchoff's postulate. This does not help her distinguish encryptions. The honest parties' effort consists of evaluating G, while the adversary has to distinguish its output from random. The latter task is substantially more taxing, as posited by the Relative Effort postulate. Finally, Alice and Bob avail themselves of true randomness to jointly choose the key k, as permitted by the Free Randomness postulate.

4 Summary

We started out this lecture wanting to build an encryption scheme for long messages with short keys. We saw how to do it, provided Alice and Bob can get hold of a pseudorandom generator. The skeptics among you might wonder what we have gained from three hours of discussion. We have pushed one problem—building encryption—to another one—building a pseudorandom generator.

Building a pseudorandom generator turned out to be challenging. There are all sorts of ways in which a sequence can fail to be pseudorandom, and no guarantee that one is. Yet pseudorandomness must surely be important if so many Turing Awards were given out for it. Why is it important?

The theory of cryptography explains why. Pseudorandom generators are the heart and soul of private-key cryptography. They make most of it possible. Without them, none of it would exist. More advanced cryptography like public-key encryption and homomorphic encryption is also based on pseudorandom generators with special additional properties. In short, pseudorandomness is an extremely useful abstraction.

There is also a pragmatic case for pseudorandomness. Indistinguishability from pure randomness is a well-defined criterion for security that can be subjected to a suite of tests. While we can never know for sure whether a given proposal is a secure pseudorandom generator, we can test it for flaws: biases of individual bits, local correlations, biases of parities, dependencies detected via modular Gaussian elimination, and so on. Such tests are important for cryptographic engineers to evaluate the strengths and weaknesses of their constructions.

Like many profound scientific ideas, pseudorandomness is on the borderline of nonsense. A pseudorandom generator expands a real random seed into a random-looking but not really random output. If we are

[1] Bob should in fact subtract $G(k)$, but addition and subtraction are the same modulo two.

after randomness, why not take the output to be random in the first place? Doesn't the Free Randomness postulate give us that?

In cryptography, the answer is that it is not randomness that we are after. We want data sequences that look random to an adversary, but follow a prescribed deterministic logic from the honest parties' perspective. Pseudorandomness provides precisely this ability.