

## Lecture Notes, Week 12

### 1 Coin-Flipping

Alice and Bob are in the process of getting divorced and are trying to decide who gets custody of their pet cat, Fluffy. They both want the cat, so they agree to decide by flipping a coin: heads Alice wins; tails Bob wins. Bob has already moved out and does not wish to be in the same room with Alice. The feeling is mutual, so Alice proposes that she flip the coin and telephone Bob with the result.

This proposal of course is not acceptable to Bob since he has no way of knowing whether Alice is telling the truth when she says that the coin landed heads. “Look Alice,” he says, “to be fair, we both have to be involved in flipping the coin. We’ll each flip a private coin and XOR our two coins together to determine who gets Fluffy. You should be happy with this arrangement since even if you don’t trust me to flip fairly, your own fair coin is sufficient to ensure that the XOR is unbiased.” This sounds reasonable to Alice, so she lets him go on to propose the protocol of Figure 1. In this protocol, 1 means “heads” and 0 means “tails”.

Alice	Bob
1. Choose random bit $b_A \in \{0, 1\}$	$\xrightarrow{b_A}$
2.	$\xleftarrow{b_B}$ Choose random bit $b_B \in \{0, 1\}$ .
3. Coin outcome is $b = b_A \oplus b_B$ .	Coin outcome is $b = b_A \oplus b_B$ .

Figure 1: Distributed coin flip protocol requiring honest parties.

After Alice considers Figure 1 for awhile, she objects. “This isn’t fair. You get to see my coin flip before I see yours, so now you have complete control over the value of  $b$ .” She suggests that she would be happy if the first two steps were reversed, so that Bob flipped his coin first, but Bob balks at that suggestion.

They then both remember last week’s lecture and decide to use blobs to prevent either party from controlling the outcome. They agree on the protocol of Figure 2. At the completion of step 2, both Alice and Bob have each other’s commitment (something they failed to achieve in the past, which is why they’re in the middle of a divorce now), but neither know the other’s private bit. They each learn each other’s bit at the completion of the respective open protocols.

While this protocol appears to be completely symmetric, it really isn’t quite, for one of the parties completes step 3 before the other one does. Say Bob completes opening  $c_B$  first. At that point, Alice knows  $b_B$  and hence the coin outcome  $b$ . If it turns out that she lost, she might decide to stop the protocol and refuse to complete her part of step 3.

We haven’t really addressed the question for any of these protocols about what happens if one party quits in the middle or one party detects the other party cheating. We have only been concerned until now with the possibility of undetected cheating. But in any real situation, one party might feel that he or she stands to gain by cheating, even if the cheating is detected. That in turn raises

Alice		Bob
1. Choose random bit $b_A \in \{0, 1\}$ .		Choose random bit $b_B \in \{0, 1\}$ .
2. <b>commit</b> ( $b_A$ ).	$\longleftrightarrow$	<b>commit</b> ( $b_B$ ).
3. <b>open</b> ( $c_A$ ).	$\longleftrightarrow$	<b>open</b> ( $c_B$ ).
4. Coin outcome is $b = b_A \oplus b_B$ .		Coin outcome is $b = b_A \oplus b_B$ .

Figure 2: Distributed coin flip protocol using blobs.

complicated questions as to what happens next. Does a third party Carol become involved? If so, can Bob prove to Carol that Alice cheated? What if Alice refuses to talk to Carol? It may be instructive to think about the recourse that Bob has in similar real-life situations and to consider the reasons why such situations rarely arise. For example, what happens if someone fails to follow the provisions of a contract or if someone ignores a summons to appear in court?

## 2 Locked Box Paradigm

Protocols for coin flipping and for dealing a poker hand from a deck of cards can be based on the intuitive notion of locked boxes. This idea in turn can be implemented using commutative cryptosystems.

### 2.1 Coin-flipping using locked boxes

We discussed the coin-flipping problem in section 1 and presented a protocol based on bit-commitment. Here we present a coin-flipping protocol based on the idea of locked boxes.

- Imagine two sturdy boxes with hinged lids that can be locked with a padlock. Alice writes “heads” on a slip of paper and “tails” on another and places one of these slips in each box. She puts a padlock on each box for which she holds the only key. She then gives both locked boxes to Bob, in some random order.
- Bob cannot open the boxes and does not know which box contains “heads” and which contains “tails”. He chooses one of the boxes and locks it with his own padlock, for which he has the only key. Now the box has two locks on it, one belonging to Alice and one to Bob. He gives the doubly-locked box back to Alice.
- Alice removes her lock and returns the box to Bob.
- Bob removes his lock, opens the box, and learns the outcome of the coin toss. He gives the slip of paper from the unlocked box back to Alice.
- Alice verifies that it is her slip of paper, with her handwriting on it, that she prepared at the beginning. She sends her key to Bob.
- Bob removes Alice’s lock from the other box and verifies that she carried out her protocol correctly. (In particular, he checks that the slip of paper in the other box contains the other coin value.)

## 2.2 Commutative cryptosystems

Alice and Bob can carry out this protocol electronically using any *commutative* cryptosystem, that is, one in which  $E_A(E_B(m)) = E_B(E_A(m))$  for all messages  $m$ . RSA is commutative for keys with a common modulus  $n$ , so we can use RSA in an unconventional way. Rather than making the encryption exponent public and keeping the factorization of  $n$  private, we turn things around. Alice and Bob jointly chose primes  $p$  and  $q$ , and both compute  $n = pq$ . Alice then chooses an RSA key pair  $A = ((e_A, n), (d_A, n))$ , which she can do since she knows the factorization of  $n$ . Similarly, Bob chooses an RSA key pair  $B = ((e_B, n), (d_B, n))$  using the same  $n$ . Alice and Bob both keep their key pairs private (until the end of the protocol, when they reveal them to each other to verify that there was no cheating).

We note that this scheme may have completely different security properties from usual RSA. In RSA, there are three different secrets involved with the key: the factorization of  $n$ , the encryption exponent  $e$ , and the decryption exponent  $d$ . We have seen previously that knowing  $n$  and any two of these pieces of information allows the third to be reconstructed. Thus, knowing the factorization of  $n$  and  $e$  lets one compute  $d$  (easy). We also showed in [section 1.3 of lecture notes week 6](#) how to factor  $n$  given both  $e$  and  $d$ .

The way RSA is usually used, only  $e$  is public, and it is believed to be hard to find the other quantities. Here we propose making the factorization of  $n$  public but keeping  $e$  and  $d$  private. It may indeed be hard to find  $e$  and  $d$ , even knowing the factorization of  $n$ , but if it is, that fact is not going to follow from the difficulty of factoring  $n$ . Of course, for security, we need more than just that it is hard to find  $e$  and  $d$ . We also need it to be hard to find  $m$  given  $c = m^e \bmod n$ . This is reminiscent of the discrete log problem, but of course  $n$  is not prime in this case.

## 2.3 Coin-flipping using commutative cryptosystems

Assuming RSA used in this new way is secure, we can implement the locked box protocol as shown in Figure 3. Here we assume that Alice and Bob initially know large primes  $p$  and  $q$ . In step (2), Alice chooses a random number  $r$  such that  $r < (n - 1)/2$ . This ensures that  $m_0$  and  $m_1$  are both in  $\mathbb{Z}_n$ . Note that  $i$  and  $r$  can be efficiently recovered from  $m_i$  since  $i$  is just the low-order bit of  $m_i$  and  $r = (m_i - i)/2$ .

To see that the protocol works when both Alice and Bob are honest, observe that in step 3,  $c_{ab} = E_B(E_A(m_j))$  for some  $j$ . Then in step 4,  $c_b = D_A(E_B(E_A(m_j))) = E_B(m_j)$  by the commutativity of  $E_A$  and  $E_B$ . Hence, in step 5,  $m = m_j$  is one of Alice's strings from step 2.

A dishonest Bob can control the outcome of the coin toss if he can find two keys  $B$  and  $B'$  such that  $E_B(c_a) = E_{B'}(c'_a)$ , where  $C = \{c_a, c'_a\}$  is the set received from Alice in step 2. In this case,  $c_{ab} = E_B(E_A(m_j)) = E_{B'}(E_A(m_{1-j}))$  for some  $j$ . Then in step 4,  $c_b = E_B(m_j) = E_{B'}(m_{1-j})$ . Hence,  $m_j = D_B(c_b)$  and  $m_{1-j} = D_{B'}(c_b)$ , so Bob can obtain both of Alice's messages and then send  $B$  or  $B'$  in step 5 to force the outcome to be as he pleases.

## 2.4 Card dealing using locked boxes

The same locked box paradigm can be used for dealing a 5-card poker hand from a deck of cards. Alice takes a deck of cards, places each card in a separate box, and locks each box with her lock. She arranges the boxes in random order and ships them off to Bob. Bob picks five boxes, locks each with his lock, and send them back. Alice removes her locks from those five boxes and returns them to Bob. Bob unlocks them and obtains the five cards of his poker hand. Further details are left to the reader.

	Alice		Bob
1.	Choose RSA key pair $A$ with modulus $n = pq$ .		Choose RSA key pair $B$ with modulus $n = pq$ .
2.	Choose random $r \in \mathbf{Z}_{(n-1)/2}$ . Let $m_i = 2r + i$ , for $i \in \{0, 1\}$ . Let $c_i = E_A(m_i)$ for $i \in \{0, 1\}$ . Let $C = \{c_0, c_1\}$ .	$\xrightarrow{C}$	Choose $c_a \in C$ .
3.		$\xleftarrow{c_{ab}}$	Let $c_{ab} = E_B(c_a)$ .
4.	Let $c_b = D_A(c_{ab})$ .	$\xrightarrow{c_b}$	
5.			Let $m = D_B(c_b)$ . Let $i = m \bmod 2$ . Let $r = (m - i)/2$ . If $i = 0$ outcome is “tails”. If $i = 1$ outcome is “heads”.
6.	Let $m = D_B(c_b)$ . Check $m \in \{m_0, m_1\}$ . If $m = m_0$ outcome is “tails”. If $m = m_1$ outcome is “heads”.	$\xleftarrow{B}$	
7.		$\xrightarrow{A}$	Let $c'_a = C - \{c_a\}$ . Let $m' = D_A(c'_a)$ . Let $i' = m' \bmod 2$ . Let $r' = (m' - i')/2$ . Check that $i' \neq i$ and $r' = r$ .

Figure 3: Distributed coin flip protocol using locked boxes.

### 3 Oblivious Transfer

In the locked box coin-flipping protocol, Alice has two messages  $m_0$  and  $m_1$ . Bob gets one of them. Alice doesn't know which (until Bob tells her). Bob can't cheat to get both messages. Alice can't cheat to learn which message Bob got. The *oblivious transfer problem* abstracts these properties from particular applications such as coin flipping and card dealing,

#### 3.1 Oblivious transfer of a secret

Alice has a secret  $s$ . In an oblivious transfer protocol, half of the time Bob learns  $s$  and half of the time he learns nothing. Afterwards, Alice doesn't know whether or not Bob learned  $s$ . Bob can do nothing to increase his chances of getting  $s$ , and Alice can do nothing to learn whether or not Bob got her secret. Rabin proposed an oblivious transfer protocol based on quadratic residuosity, shown in Figure 4, in the early 1980's.

Alice can carry out step 3 since she knows the factorization of  $n$  and can find all of the square roots of  $a$ . However, she has no idea which  $x$  Bob used to generate  $a$ . Hence, with probability  $1/2$ ,

	Alice	Bob
1.	Secret $s$ . Choose primes $p, q$ , and let $n = pq$ . Generate RSA public key $(e, n)$ . Compute $c = E_{(e,n)}(s)$ .	$\xrightarrow{(e,n,c)}$
2.		Choose random $x \in \mathbb{Z}_n^*$ . Compute $a = x^2 \bmod n$ .
3.	Check $a \in \text{QR}_n$ . Choose $y$ at random from among the 4 square roots of $a \pmod{n}$ .	$\xleftarrow{a}$ $\xrightarrow{y}$
4.		Check $y^2 \equiv a \pmod{n}$ . If $y \not\equiv \pm x \pmod{n}$ , use $x, y$ to factor $n$ and decrypt $c$ to obtain $s$ .

Figure 4: Rabin's oblivious transfer protocol.

$y \equiv \pm x \pmod{n}$  and with probability  $1/2$ ,  $y \not\equiv \pm x \pmod{n}$ . If  $y \not\equiv \pm x \pmod{n}$ , then the two factors of  $n$  are  $\gcd(x - y, n)$  and  $n / \gcd(x - y, n)$ , so Bob factors  $n$  and decrypts  $c$  in step 4. However, if  $y \equiv \pm x \pmod{n}$ , he learns nothing, and Alice's secret is as secure as RSA itself.

There is one potential problem with this protocol. A cheating Bob in step 2 might send a number  $a$  which he generated by some other means than squaring a random  $x$ . In this case, he learns something new no matter which square root Alice sends him in step 3. Perhaps that information, together with what he already learned in the course of generating  $a$ , is enough for him to factor  $n$ . We don't know of any method by which Bob could find a quadratic residue without also knowing one of its square roots. We certainly don't know of any method that would produce a quadratic residue  $a$  and some other information  $\Xi$  that, combined with  $y$ , would allow Bob to factor  $n$ . But we also cannot prove that no such method exists.

We can fix this protocol by inserting between steps 2 and 3 a zero knowledge proof that Bob knows a square root of  $a$ . This is essentially what the simplified Feige-Fiat-Shamir protocol does, but we have to reverse the roles of Alice and Bob. Now Bob is the one with the secret square root  $x$ . He wants to prove to Alice that he knows  $x$ , but he does not want Alice to get any information about  $x$ , since if she learns  $x$ , she could choose  $y = x$  and reduce his chances of learning  $s$  while still appear to be playing honestly. Again, details are left to the reader.

### 3.2 One-of-two oblivious transfer

In the *one-of-two oblivious transfer*, Alice has two secrets,  $s_0$  and  $s_1$ . Bob always gets exactly one of the secrets. He gets each with probability  $1/2$ , and Alice does not know which he got.

The locked box protocol is one way to implement one-of-two oblivious transfer. Another is based on a public key cryptosystem (such as RSA) and a symmetric cryptosystem (such as AES). This protocol, given in Figure 5, does not rely on the cryptosystems being commutative.

In step 2, Bob encrypts a randomly chosen key  $k$  for the symmetric cryptosystem using one of the encryption keys that Alice sent him in step 1. One of the  $k_i$  Alice computes in step 3 is Bob's key  $k$ , but because  $k$  is random, she can't tell which it is. However, the key that is different from  $k$

	Alice	Bob
1.	Choose two PKS key pairs $(e_0, d_0)$ and $(e_1, d_1)$ .	
		$\xrightarrow{e_0, e_1}$
2.		Generate random key $k$ for a symmetric cryptosystem $(\hat{E}, \hat{D})$ . Choose $e \in \{e_0, e_1\}$ . Compute $c = E_e(k)$ .
		$\xleftarrow{c}$
3.	Compute $k_i = D_{d_i}(c)$ for $i = 0, 1$ . Compute $c_i = \hat{E}_{k_i}(s_i)$ for $i = 0, 1$ .	$\xrightarrow{c_0, c_1}$
4.		Compute $s_i = \hat{D}_k(c_i)$ for $i = 0, 1$ . One of the $s_i$ is correct and the other is garbage.

Figure 5: One-of-two oblivious transfer using two PKS key pairs.

cannot be computed by Bob since Bob doesn't have the corresponding decryption key  $d_i$ .

Note that this protocol depends on Bob being able to distinguish good secrets from random-looking garbage. To make it work for arbitrary secrets, Alice can add some known redundancy to the secrets that Bob can recognize. Another possibility would be for her to encrypt  $c \cdot s_i$  in step 3 instead of just  $s_i$ . Since Bob knows  $c$ , he could then check the two possible decryptions to see which one began with  $c$ .

## 4 Pseudorandom Sequence Generation

We mentioned pseudorandom sequence generation in section 3.3 of [lecture notes week 11](#). In a little more detail, a *pseudorandom sequence generator*  $G$  is a function from a domain of *seeds*  $\mathcal{S}$  to a domain of strings  $\mathcal{X}$ . We will generally find it convenient to assume that all of the seeds in  $\mathcal{S}$  have the same length  $m$  and that all of the strings in  $\mathcal{X}$  have the same length  $n$ , where  $m \ll n$  and  $n$  is polynomially related to  $m$ .

Intuitively, we want the strings  $G(s)$  to “look random”. But what does that mean? Chaitin and Kolmogorov proposed ways of defining what it means for an individual sequence to be considered random. While philosophically very interesting, these notions are somewhat different than the statistical notions that most people mean by randomness.

We take a different tack. We assume that the seeds are chosen uniformly at random from  $\mathcal{S}$ , that is, we consider a uniformly distributed random variable  $S$  over  $\mathcal{S}$ . Then  $X = G(S)$  is a random variable over  $\mathcal{X}$ . For  $x \in \mathcal{X}$ ,

$$\text{prob}[X = x] = \frac{|\{s \in \mathcal{S} \mid G(s) = x\}|}{|\mathcal{S}|}.$$

That is, the probability is the fraction of seeds that give rise to  $x$ . Because  $m \ll n$ ,  $|\mathcal{S}| = 2^m$ , and  $|\mathcal{X}| = 2^n$ , most strings in  $\mathcal{X}$  are not in the range of  $G$  and hence have probability 0. If  $G$  happens to be one-to-one, then the remaining strings each have probability  $1/2^m$ .

We also consider the uniform random variable  $U \in \mathcal{X}$ , where  $U = x$  with probability  $1/2^n$  for every  $x \in \mathcal{X}$ .  $U$  is what we usually mean by a “truly random” variable on  $n$ -bit strings.

We will say that  $G$  is a *cryptographically strong* pseudorandom sequence generator if  $X$  and  $U$  are *indistinguishable* to all probabilistic polynomial Turing machines. We have already seen that the probability distributions of  $X$  and  $U$  are quite different. Nevertheless, they are indistinguishable if there is no feasible algorithm to determine whether random samples come from  $X$  or from  $U$ .

Before going further, let me describe some functions  $G$  for which  $G(S)$  is readily distinguished from  $U$ . Suppose every string  $x = G(s)$  has the form  $b_1b_1b_2b_2b_3b_3\dots$ , for example 0011111100001100110000.... An algorithm that guesses that  $x$  came from  $G(S)$  if  $x$  is of that form, and guesses that  $x$  came from  $U$  otherwise, will be right almost all of the time. True, it is possible to get a string like this from  $U$ , but it is extremely unlikely.

Formally speaking, a *judge* is a probabilistic Turing machine  $J$  that takes an  $n$ -bit string as input and produces a single bit  $b$  as output. Because it is probabilistic, it actually defines a random function from  $\mathcal{X}$  to  $\{0, 1\}$ . This means that for every input  $x$ , there is some probability  $p_x$  that the output is 1. If the input string is itself a random variable  $X$ , then the probability that the output is 1 is the weighted sum over all possible inputs that the judge outputs 1, where the weights are the probabilities of the corresponding inputs occurring. Thus, the output value is itself a random variable which we denote by  $J(X)$ .

Now, we say that two random variables  $X$  and  $Y$  are  $\epsilon$ -indistinguishable by judge  $J$  if

$$|\text{prob}[J(X) = 1] - \text{prob}[J(Y) = 1]| < \epsilon.$$

Intuitively, we say that  $G$  is *cryptographically strong* if  $G(S)$  and  $U$  are  $\epsilon$ -indistinguishable for suitably small  $\epsilon$  by all judges that do not run for too long. A careful mathematical treatment of the concept of indistinguishability must relate the length parameters  $m$  and  $n$ , the error parameter  $\epsilon$ , and the allowed running time of the judges, all of which is beyond the scope of this course.

[Note: The topics covered by these lecture notes are presented in more detail and with greater mathematical rigor in [handout 18](#).]

## 5 BBS Pseudorandom Sequence Generator

We present a cryptographically strong PRSG due to Blum, Blum, and Shub. The generator is based on the difficulty of determining, for a given  $a \in \mathbf{Z}_n^*$  with Jacobi symbol  $(\frac{a}{n}) = 1$ , whether or not  $a$  is a quadratic residue, i.e., whether or not  $a \in \text{QR}_n$ . Recall from [lecture notes week 7](#), that this is the property upon which the security of the Goldwasser-Micali probabilistic encryption system relies. The BBS generator further requires  $n$  to be a certain kind of composite number called a Blum integer. Blum primes and Blum integers were introduced in [lecture notes week 11](#). We review their properties here.

### 5.1 Blum integers

A *Blum prime* is a prime number  $p$  such that  $p \equiv 3 \pmod{4}$ . A *Blum integer* is a number  $n = pq$ , where  $p$  and  $q$  are Blum primes. Blum primes and Blum integers have the important property that every quadratic residue  $a$  has a square root  $y$  which is itself a quadratic residue. We call such a  $y$  a *principal square root* of  $a$  and denote it by  $\sqrt{a} \pmod{n}$  or simply by  $\sqrt{a}$  when it is clear that  $\pmod{n}$  is intended.

We need two other facts about Blum integers  $n$ .

**Claim 1** Let  $a \in \text{QR}_n$ . Then  $(\frac{a}{n}) = (\frac{-a}{n}) = 1$ .

Let  $\text{lsb}(x)$  be the least significant bit of integer  $x$ . That is,  $\text{lsb}(x) = (x \bmod 2)$ .

**Claim 2** Let  $x \in \mathbf{Z}_n$ . Then  $\text{lsb}(x) \oplus \text{lsb}(-x) = 1$ .

Claim 1 actually holds for all  $n$  which are the product of two distinct odd primes. Claim 2 holds whenever  $n$  is odd.

## 5.2 BBS algorithm

The Blum-Blum-Shub generator BBS is defined by a Blum integer  $n = pq$  and an integer  $\ell$ . It maps strings in  $\mathbf{Z}_n^*$  to strings in  $\{0, 1\}^\ell$ . Given a seed  $s_0 \in \mathbf{Z}_n^*$ , we define a sequence  $s_1, s_2, s_3, \dots, s_\ell$ , where  $s_i = s_{i-1}^2 \bmod n$  for  $i = 1, \dots, \ell$ . The  $\ell$ -bit output sequence is  $b_1, b_2, b_3, \dots, b_\ell$ , where  $b_i = \text{lsb}(s_i)$ .

## 5.3 Bit-prediction

One important property of the uniform distribution  $U$  on bit-strings  $b_1, \dots, b_\ell$  is that the individual bits are statistically independent from each other. This means that the probability that a particular bit  $b_i = 1$  is unaffected by the values of the other bits in the sequence. This implies that any algorithm that attempts to predict  $b_i$ , even knowing other bits of the sequence, will be correct only 1/2 of the time. We now translate this property of unpredictability to pseudorandom sequences.

### 5.3.1 Next-bit prediction

One property we would like a pseudorandom sequence to have is that it be difficult to predict the next bit given the bits that came before.

We say that an algorithm  $A$  is an  $\epsilon$ -next-bit predictor for bit  $i$  of a PRSG  $G$  if

$$\text{prob}[A(b_1, \dots, b_{i-1}) = b_i] \geq \frac{1}{2} + \epsilon$$

where  $(b_1, \dots, b_i) = G_i(S)$ . To explain this notation,  $S$  is a uniformly distributed random variable ranging over the possible seeds for  $G$ .  $G(S)$  is a random variable (i.e., probability distribution) over the output strings of  $G$ , and  $G_i(S)$  is the corresponding probability distribution on the length- $i$  prefixes of  $G(S)$ .

Next-bit prediction is closely related to indistinguishability, introduced in section 4. We will show later that if  $G(S)$  has a next-bit predictor for some bit  $i$ , then  $G(S)$  is distinguishable from the uniform distribution  $U$  on the same length strings, and conversely, if  $G(S)$  is distinguishable from  $U$ , then there is a next-bit predictor for some bit  $i$  of  $G(S)$ . The precise definitions under which this theorem is true are subtle, for one must quantify both the amount of time the judge and next-bit predictor algorithms are permitted to run as well as how much better than chance the judgments or predictions must be in order to be considered a successful judge or next-bit predictor. We defer the mathematics for now and focus instead on the intuitive concepts that underly this theorem.

### 5.3.2 Building a judge from a next-bit predictor

Suppose a PRSG  $G$  has an  $\epsilon$ -next-bit predictor  $A$  for some bit  $i$ . Here's how to build a judge  $J$  that distinguishes  $G(S)$  from  $U$ . The judge  $J$ , given a sample string drawn from either  $G(S)$  or from  $U$ , runs algorithm  $A$  to guess bit  $b_i$  from bits  $b_1, \dots, b_{i-1}$ . If the guess agrees with the real  $b_i$ , then  $J$  outputs 1 (meaning that he guesses the sequence came from  $G(S)$ ). Otherwise,  $J$  outputs 0. For sequences drawn from  $G(S)$ ,  $J$  will output 1 with the same probability that  $A$  successfully

predicts bit  $b_i$ , which is at least  $1/2 + \epsilon$ . For sequences drawn from  $U$ , the judge will output 1 with probability exactly  $1/2$ . Hence, the judge distinguishes  $G(S)$  from  $U$  with advantage  $\epsilon$ .

It follows that no cryptographically strong PRSG can have an  $\epsilon$ -next-bit predictor. In other words, no algorithm that attempts to predict the next bit can have more than a “small” advantage  $\epsilon$  over chance.

## 5.4 Previous-bit prediction

Previous-bit prediction, while perhaps less natural, is analogous to next-bit prediction. An  $\epsilon$ -previous-bit predictor for bit  $i$  is an algorithm that, given bits  $b_{i+1}, \dots, b_\ell$ , correctly predicts  $b_i$  with probability at least  $1/2 + \epsilon$ .

As with next-bit predictors, it is the case that if  $G(S)$  has a previous-bit predictor for some bit  $b_j$ , then some judge distinguishes  $G(S)$  from  $U$ . Again, I am being vague with the exact conditions under which this is true. The somewhat surprising fact follows that  $G(S)$  has an  $\epsilon$ -next-bit predictor for some bit  $i$  if and only if it has an  $\epsilon'$ -previous-bit predictor for some bit  $j$  (where  $\epsilon$  and  $\epsilon'$  are related but not necessarily equal).

To give some intuition into why such a fact might be true, we look at the special case of  $\ell = 2$ , that is, of 2-bit sequences. The probability distribution  $G(S)$  can be described by four probabilities

$$p_{u,v} = \text{prob}[b_1 = u \wedge b_2 = v], \text{ where } u, v \in \{0, 1\}.$$

Written in tabular form, we have

		$b_2$	
		0	1
$b_1$	0	$p_{0,0}$	$p_{0,1}$
	1	$p_{1,0}$	$p_{1,1}$

We describe an algorithm  $A(v)$  for predicting  $b_1$  given  $b_2 = v$ .  $A(v)$  predicts  $b_1 = 0$  if  $p_{0,v} > p_{1,v}$ , and it predicts  $b_1 = 1$  if  $p_{0,v} \leq p_{1,v}$ . In other words, the algorithm chooses the value for  $b_1$  that is most likely given that  $b_2 = v$ . Let  $a(v)$  be the value predicted by  $A(v)$ .

**Theorem 1** *If  $A$  is an  $\epsilon$ -previous-bit predictor for  $b_1$ , then  $A$  is an  $\epsilon$ -next-bit predictor for either  $b_1$  or  $b_2$ .*

**Proof:** Assume  $A$  is an  $\epsilon$ -previous-bit predictor for  $b_1$ . Then  $A$  correctly predicts  $b_1$  given  $b_2$  with probability at least  $1/2 + \epsilon$ . We show that  $A$  is an  $\epsilon$ -next-bit predictor for either  $b_1$  or  $b_2$ .

We have two cases:

*Case 1:*  $a(0) = a(1)$ . Then algorithm  $A$  does not depend on  $v$ , so  $A$  itself is also an  $\epsilon$ -next-bit predictor for  $b_1$ .

*Case 2:*  $a(0) \neq a(1)$ . The probability that  $A(v)$  correctly predicts  $b_1$  when  $b_2 = v$  is given by the conditional probability

$$\text{prob}[b_1 = a(v) \mid b_2 = v] = \frac{\text{prob}[b_1 = a(v) \wedge b_2 = v]}{\text{prob}[b_2 = v]} = \frac{p_{a(v),v}}{\text{prob}[b_2 = v]}$$

The overall probability that  $A(b_2)$  is correct for  $b_1$  is the weighted average of the conditional probabilities for  $v = 0$  and  $v = 1$ , weighted by the probability that  $b_2 = v$ . Thus,

$$\begin{aligned} \text{prob}[A(b_2) \text{ is correct for } b_1] &= \sum_{v \in \{0,1\}} \text{prob}[b_1 = a(v) \mid b_2 = v] \cdot \text{prob}[b_2 = v] \\ &= \sum_{v \in \{0,1\}} p_{a(v),v} \\ &= p_{a(0),0} + p_{a(1),1} \end{aligned}$$

Now, since  $a(0) \neq a(1)$ , the function  $a$  is one-to-one and onto. A simple case analysis shows that either  $a(v) = v$  for  $v \in \{0, 1\}$ , or  $a(v) = \neg v$  for  $v \in \{0, 1\}$ . That is,  $a$  is either the identity or the complement function. In either case,  $a$  is its own inverse. Hence, we may also use algorithm  $A(u)$  as a predictor for  $b_2$  given  $b_1 = u$ . By a similar analysis to that used above, we get

$$\begin{aligned} \text{prob}[A(b_1) \text{ is correct for } b_2] &= \sum_{u \in \{0,1\}} \text{prob}[b_2 = a(u) \mid b_1 = u] \cdot \text{prob}[b_1 = u] \\ &= \sum_{u \in \{0,1\}} p_{u,a(u)} \\ &= p_{0,a(0)} + p_{1,a(1)} \end{aligned}$$

But

$$p_{a(0),0} + p_{a(1),1} = p_{0,a(0)} + p_{1,a(1)}$$

since either  $a$  is the identity function or the complement function. Hence,  $A(b_1)$  is correct for  $b_2$  with the same probability that  $A(b_2)$  is correct for  $b_1$ . Therefore,  $A$  is an  $\epsilon$ -next-bit predictor for  $b_2$ .

In both cases, we conclude that  $A$  is an  $\epsilon$ -next-bit predictor for either  $b_1$  or  $b_2$ . ■

## 5.5 Security of BBS generator

We now show that if BBS has a previous-bit predictor, then there is an algorithm for testing quadratic residues whose running time and accuracy are related to the running time and accuracy of the BBS predictor. Thus, if quadratic-residue-testing is “hard”, then so is previous-bit prediction for BBS. See [handout 18](#) for further results on the security of BBS.

**Theorem 2** *Let  $A$  be an  $\epsilon$ -previous-bit predictor for  $\text{BBS}(S)$ . Then we can find an algorithm  $Q$  for testing whether a number  $x$  with Jacobi symbol 1 is a quadratic residue, and  $Q$  will be correct with probability at least  $1/2 + \epsilon$ .*

**Proof:** Assume that  $A$  predicts  $b_j$  given the  $k$  bits  $b_{j+1}, \dots, b_{j+k}$ . Then  $A$  also predicts  $b_1$  given  $b_2, \dots, b_k$  with the same accuracy. This follows from the fact that the mapping  $x \mapsto x^2 \bmod n$  is a permutation on  $\text{QR}_n$ . Hence,  $s_0$  and  $s_{j-1}$  are identically distributed, so the  $k$ -bit prefixes of  $\text{BBS}(s_0)$  and  $\text{BBS}(s_{j-1})$  are likewise identically distributed.

Algorithm  $Q(x)$ , shown in Figure 6, tests whether or not a number  $x$  with Jacobi symbol 1 is a quadratic residue modulo  $n$ . It outputs 1 to mean  $x \in \text{QR}_n$  and 0 to mean  $x \notin \text{QR}_n$ .

To  $Q(x)$ :

1. Let  $s_2 = x^2 \bmod n$ .
2. Let  $s_i = s_{i-1}^2 \bmod n$ , for  $i = 3, \dots, k$ .
3. Let  $b_1 = \text{lsb}(x)$ .
4. Let  $b_i = \text{lsb}(s_i)$ , for  $i = 2, \dots, k$ .
5. Let  $c = A(b_2, \dots, b_k)$ .
6. If  $c = b_1$  then output 1; else output 0.

Figure 6: Algorithm  $Q$  for testing  $x \in \text{QR}_n$ .

Since  $\left(\frac{x}{n}\right) = 1$ , then either  $x$  or  $-x$  is a quadratic residue. Let  $s_0$  be the principal square root of  $x$  or  $-x$  and consider the first  $k$  bits of  $\text{BBS}(s_0)$ . We have two cases:

If  $x \in \text{QR}_n$ , the sequence of seeds is  $s_0, x, s_2, \dots, s_k$  and the corresponding sequence of output bits is

$$b_1, b_2, \dots, b_k.$$

If  $-x \in \text{QR}_n$ , the sequence of seeds is  $s_0, -x, s_2, \dots, s_k$  and the corresponding sequence of output bits is

$$\neg b_1, b_2, \dots, b_k.$$

Hence, if the predicted first bit  $c$  is correct, then

$$c = b_1 \quad \text{iff} \quad x \in \text{QR}_n.$$

Since the predicted bit is correct with probability at least  $1/2 + \epsilon$ , algorithm  $Q$  is correct with probability at least  $1/2 + \epsilon$ . ■