This unit is a quick look at some of the interesting and important ways random numbers are used in applied cryptography in order to secure information and computation.

1 Cryptographic and Randomness Primitives

So far in the class, we’ve spent considerable time talking about and using three basic “building blocks” of randomized algorithms and data structures:

1) True random number generators
2) Pseudorandom number generators (PRNGs)
3) Hash functions

Each of these has a number of different implementations, as we have seen, with various different properties of quality vs cost or speed.

In the context of cryptography, these are called “cryptographic primitives”, the basic building blocks upon which cryptographic protocols are built. There are many cryptographic primitives, but here are the five we will be using in this class:

1) Cryptographic true random number generators
2) Cryptographic PRNGs
3) Cryptographic Hash functions
4) Symmetric encryption (a.k.a. secret-key encryption)
5) Asymmetric encryption (a.k.a. public-key encryption)

You can see that the first three are the same kinds of things we’ve been talking about already, but with a new adjective “cryptographic”. In what we’ve been using so far, it’s not a big deal if some human can tell that we are, for example, using a Lehmer LCG or a Carter & Wegman hash. And if someone were able to inspect our program and predict the next random number as it’s running, who cares? We just needed our random numbers to be good enough to ensure good running times on valid data sets.

But in the context of cryptography, there are more restrictions in order to ensure security. Random numbers are used for things like generating secret keys, which no one should ever be able to guess even if they know exactly how the key is being generated. There is a deep field of study in defining and proving the security of these cryptographic primitives, but it boils down to this:

A cryptographic PRNG or hash function should be indistinguishable from true randomness.

What the word “indistinguishable” here means is usually one of two things. The first thing it means is “information-theoretic security”, which means that the distribution is exactly the same, so no one would ever be able to tell the difference. This is hard (sometimes impossible) to achieve, but when we can, it’s the best security guarantee possible.

The second thing “indistinguishable” could mean is a bit weaker, but still very useful. “Computational” security means that any polynomial-time algorithm (that is, any reasonably efficient program) would never
be able to tell the difference. In other words, roughly, someone might be able to figure out your PRNG seed or unwind your hash function, but they would have to run an exponential-time algorithm to do it. So if you make your seed length or hash size large enough, you can achieve security.

We’ll see that these two kinds of security can also be applied to protocols and other things we build on top of these “primitives”. You’ll also look at a few specific examples of these crypto primitives in the upcoming problems.

For completeness, I’m going to give a quick and dirty definition of symmetric and asymmetric encryption, which you should all already be familiar with from your other classes.

**Symmetric Encryption**

A symmetric encryption scheme consists of three parts: a key generator, an encryption routine, and a decryption routine.

The key generator uses a random number source or PRNG and produces a secret key $k$. We write $\text{kgen}() \rightarrow k$.

Encryption takes the secret key $k$ and a plaintext message $M$ and produces an encrypted ciphertext $E$: $\text{Enc}_k(M) \rightarrow E$.

Decryption takes the same secret key $k$ and a ciphertext $E$ and recovers the original plaintext: $\text{Dec}_k(E) \rightarrow M$.

**Asymmetric Encryption**

An asymmetric encryption scheme is the same as symmetric encryption except that the key generator produces a pair of keys $k_{\text{pub}}$ and $k_{\text{pri}}$. The encryption algorithm makes use of the public key, and the decryption algorithm makes use of the private key.

The public and private keys are for this reason sometimes called the encryption key and decryption key, respectively.

## 2 Probabilistic Encryption

Most encryption algorithms, in their basic formulation, are deterministic. That means if you take the same key, and the same plaintext input, you’ll get the same ciphertext output every time.

This is problematic, because that’s exactly what happens in many situations. Imagine, for example, a professor sending in her student’s grades to the registrar one by one. Each grade is encrypted, to protect confidentiality. But anyone, even without the decryption key, could still see which students have the same grade! Even worse, because there are only 5 possible letter grades, in a predictable distribution, it will not take too long for an attacker to learn exactly what each encrypted grade really means.

The solution is to add some randomness to the encryption process, so that even encrypting the same exact message with the same exact key will produce a very different looking ciphertext. Specifically, the property we want can be stated as follows:

In a probabilistic encryption scheme, any two encryptions - even of the exact same message - should be indistinguishable by anyone that does not have the decryption key.

### 2.1 One-Time Pad

There is an “easy” solution to the probabilistic encryption problem, and it’s one that we’ll use in some other protocols as well. The idea is to use a completely random key that has the same length or size as your plaintext, and use this to scramble (and unscramble) the message.
If this is done correctly, you get the property that every possible ciphertext is equally likely, based only on the randomness of the one-time pad (OTP) key and not on the randomness of the plaintext message itself. Because every possible ciphertext is equally likely, you get very strong security - indistinguishability of the "information-theoretic" type. Even given an infinite amount of computing time, someone without the key can't do anything better than randomly guess as to what message was sent.

In practice, this is usually implemented computationally with the XOR (exclusive-or) operation. If my plaintext message is \( n \) bits long, I choose a length-\( n \) string of random bits, and then bitwise XOR the two in order to get the encrypted ciphertext.

There are two reasons why XOR is used here. First, it's reversible (for decryption) using exactly the same process. If you XOR the ciphertext with the same key used for encryption, the two XOR's essentially "cancel out" and you get back the same original message. Mathematically, it looks like this:

\[
(M \oplus K) \oplus K = M \oplus (K \oplus K) = M \oplus 0 = M
\]

The second reason why XOR is used is that if you XOR any arbitrary string with a string of random bits, the result is completely random. To see this, check out the truth table for XOR:

<table>
<thead>
<tr>
<th>( x \oplus y )</th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1</td>
<td>1 0</td>
</tr>
</tbody>
</table>

Looking across each row of the truth table, you see an equal number of 0’s or 1’s in the result of the XOR. This means that if I have the first bit, \( x \), coming from some non-random source (my original message), and if the second bit, \( y \), is equally likely to be 0 or 1, then \( x \oplus y \) is equally likely to be 0 or 1, no matter what \( x \) was.

OTPs offer a "perfect" solution to the probabilistic encryption algorithm. But there's a big problem: the key can only be used once and has to be as long as the message itself. Then you have to ask, how do I transmit the key itself? And you're back to the same problem as you had originally.

(Note: OTPs are actually used in some situations where you want absolute secrecy and you have ample time to prepare ahead of time for the secure transmission. One famous example is the SIGSALY system used in WWII for Churchill and Roosevelt to communicate. Their audio was mixed with records of random noise before transmission, and on the other end an identical record of random noise was used to descramble.)

### 2.2 More efficient probabilistic encryption

You can mix the idea of OTPs with existing symmetric or asymmetric encryption to get a very nice solution to the probabilistic encryption problem. The basic idea is this:

- Use a OTP to encrypt the data
- Use your existing encryption algorithm to encrypt the OTP key
- Send both the OTP-encrypted message, and the regular-encrypted OTP key as the full (probabilistic) encryption.
- Decryption involves first decrypting the OTP key, then using XOR to decrypt the actual message.

Mathematically, we might write this encryption as follows:

\[
\text{Enc}_{\text{probabilistic}}(M) \mapsto (\text{Enc}_{\text{deterministic}}(K) \| K \oplus M),
\]

where \( K \) is a randomly-chosen OTP key. (Note, the \( \| \) operator here just means concatenation.)

This method doubles the message length but makes it perfectly probabilistic: Since every OTP key is equally likely to be chosen, the first part of the ciphertext \( \text{Enc}(K) \) is just an encryption of a random value, which itself is therefore random even if your encryption algorithm is deterministic. And the second part of the ciphertext \( K \oplus M \) is a OTP encryption, which as discussed above is indistinguishable from random.
The idea just mentioned works great in theory and is useful for understanding how probabilistic encryption works. In practice, it’s still not good enough because of the doubling of the message length.

For symmetric ciphers, the OTP key is called an “initialization vector” or IV, and there are various modes of operation that specify how to re-use the same randomness over and over throughout a long encryption. This way you only need a string of random bits as long as one encrypted block (typically something like 256 bits), rather than the entire message length.

For asymmetric ciphers, a different technique is used, whereby the original message is “padded” with some random bits before encryption. The details actually depend on which kind of public-key encryption algorithm you use. For example, the standard for RSA encryption is called OAEP.

3 Oblivious Transfer

Encryption is useful for two parties, say Alice and Bob, to communicate in such a way that no third party “eavesdropper” that views all their encrypted communication can tell what is going on. Informally, Alice and Bob get security from some eavesdropper Eve, but they get no privacy from each other: Alice and Bob end up both learning the entire secret message, and Eve learns nothing.

Sometimes we want even more privacy than this, where Alice and Bob also maintain some secrets from each other. A very useful first step towards such privacy is a scheme called 1-2 Oblivious Transfer. Informally, it’s a way that Alice (the “sender”) can send just one of two messages to Bob (the “receiver”), without knowing which message Bob received, and without Bob knowing the other message that he didn’t choose to receive.

Here’s a more formal definition:

1-2 Oblivious Transfer

Alice starts with two messages $m_0$ and $m_1$.

Bob starts with a single bit $b \in \{0, 1\}$.

Bob should learn the single message $m_b$ selected by his bit $b$.

Bob should not learn anything about the other message $m_{1-b}$, and Alice should not learn Bob’s bit $b$.

As a somewhat contrived example, say Bob is donating blood at a blood bank. A real problem with blood donations is keeping HIV-positive blood out of the system, without stigmatizing or ostracizing potential donors who are HIV-positive. You can imagine a screening question, “Is there a chance that you have contracted HIV?” Depending on your answer, you receive one of two label codes to attach to your blood donation. This could be implemented as a 1-2 OT, where the yes/no answer is Bob’s bit $b$, and the two messages $m_0$ and $m_1$ are the two codes (one for safe blood and one for potentially HIV+ blood).

Okay, so that example feels a bit “forced”. The real case for 1-2 OT is that it’s a key building block in the protocols we’re going to develop later on. For now, let’s just focus on the technical challenge of how to implement a 1-2 OT protocol.

There have been multiple proposed solutions to this problem, but the current “best practices” solutions rely on things like trapdoors and hardcore predicates that we don’t have time to get into. So I’m going to describe a simpler (and somewhat flawed) OT method here, that gets the main idea across.

The intuition behind the protocol is as follows: Bob chooses a one-time pad and sends it to Alice, along with another random value that means nothing to him. Alice encrypts both messages, one with Bob’s OTP key, and the other with the random meaningless thing. But Alice doesn’t know which is which, so she doesn’t learn Bob’s bit. Then Bob receives both encryptions but can only decrypt the one corresponding to his actual OTP key. Alice uses a public-key encryption algorithm to make sure Bob doesn’t “cheat” and choose two OTP keys and therefore learn both messages.
Here’s that intuition written as a step-by-step protocol. In class we also wrote this in typical protocol 3-column syntax to show what Alice knows, what Bob knows, and what information is transferred between them.

1. Alice has the two messages $m_0$ and $m_1$.
   - Alice generates a pair of encryption and decryption keys for asymmetric (“public-key”) encryption, $\text{Enc}$ and $\text{Dec}$.
   - Alice generates two random bit-strings (“nonces”) $x_0$ and $x_1$.
   - Alice sends $x_0$, $x_1$, and the public key $\text{Enc}$ to Bob.
2. Bob has his secret bit $b \in \{0, 1\}$.
   - Bob chooses a secret OTP key $y$, which is again just a random bit-string.
   - Bob computes $u = \text{Enc}(y) \oplus x_b$.
   - Bob sends $u$ back to Alice.
3. Alice computes two values using her private decryption key and the two nonces she chose from step 1:
   
   
   $$v_0 = \text{Dec}(u \oplus x_0) \oplus m_0$$
   $$v_1 = \text{Dec}(u \oplus x_1) \oplus m_1$$
   
   (Since either $\text{Dec}(u \oplus x_0)$ or $\text{Dec}(u \oplus x_1)$ equals Bob’s nonce $y$, either $v_0$ or $v_1$ will be the message Bob should receive, encrypted with his OTP. But Alice doesn’t know which $v_i$ is decryptable by Bob and which one isn’t!)
   - Alice sends both messages $v_0$ and $v_1$ to Bob.
4. Bob picks the message $v_b$ corresponding to his bit and computes
   
   $$m_b = v_b \oplus y$$
   
   using the secret nonce $y$ he chose in step 2.

At the end, Bob learns his message $m_b$ but not the other message, and Alice knows that he’s received one of the two messages but doesn’t know which one.

How does Bob know he’s safe? Well, since $y$ is a random nonce, the encryption $\text{Enc}(y)$ is also a random value, which essentially acts as a one-time pad for his secret key $y$. Since Alice doesn’t know $y$, she couldn’t possibly tell the difference between $\text{Enc}(y) \oplus x_0$ or $\text{Enc}(y) \oplus x_1$.

And how does Alice know Bob doesn’t learn the other message? This actually relies on some strong property of the public-key encryption algorithm $\text{Enc}$, namely that it’s impossible for Bob to learn anything about one decryption even if he knows the decryption of a related value. From this property, Alice can infer that Bob only knows either $\text{Dec}(u \oplus x_0)$ or $\text{Dec}(u \oplus x_1)$ but not both; the decryption that Bob doesn’t know will be just like a random string of bits. So when Bob receives $v_0$ and $v_1$, at least one of them will appear to him as a message that’s been XOR’d with a random string of bits - i.e., a message encrypted with a OTP that he doesn’t have. So at least one of the messages will not be recoverable by Bob.

(Caution: This is all rather informal, with the main goal of conveying the idea of 1-2 OT and where randomness gets used within it. Please don’t use this as an actual secure protocol since you haven’t seen a security proof. The details of how you implement this correctly will depend strongly on the details of the public-key encryption algorithm that gets used. Namely, the “flaw” that I’ve swept under the rug is that Bob has to make sure that $\text{Enc}(y) \oplus x_0 \oplus x_1$ is decryptable by Alice, or else she’ll learn which message he has chosen.)
4 Oblivious Equality Check

One of the coolest and simplest uses of 1-2 OT is to solve the following problem:

**Oblivious Equality Check**

Alice and Bob each have secret bit-strings $A$ and $B$, both of the same length $n$.

They want to learn whether $A = B$ or not, without revealing anything about their secret when the strings are not equal.

As a silly example, imagine a couple staring deeply into each other eyes. One person says, “Are you thinking what I’m thinking?” The other responds, “I think so!” But in reality, one of them is thinking of the string MARRIAGE and the other is thinking of the string BURRITOS. They would like to learn the simple fact that they are not, in fact, thinking the same thing, without the potential embarrassment of saying what was actually on their mind - without revealing even that their two strings matched up in 3 out of the 8 positions.

The idea behind this protocol is that Alice chooses a pair of numbers for each of the $n$ bits in their messages. One number in the pair is associated with a 0 bit value, and the other number is associated with that bit being 1. Alice sets things up so that the XOR of all the numbers corresponding to her secret message $A$ is a string of all 0 bits, indicating a match, whereas the XOR of any other selection will just be some random value (not all 0’s). Bob selects the numbers corresponding to his secret message and XOR’s them together; if he gets all 0’s, then he knows their messages are the same.

More formally now:

1. Alice has an $n$-bit message $A = a_1a_2\cdots a_n$.
   
   Alice chooses $2^n - 1$ random nonces, $x_{i,b}$ for each $1 \leq i \leq n$ and $b \in \{0, 1\}$, except for the very last bit corresponding to her message $x_{n,a_n}$.

   Alice computes the XOR of the first $n - 1$ nonces corresponding to her message and saves that as the last **non-random** nonce:
   
   $$x_{n,a_n} = x_{1,a_1} \oplus x_{2,a_2} \oplus \cdots \oplus x_{n-1,a_{n-1}}$$

2. For each of his secret bits $b_1, \ldots, b_n$, Bob selects one of Alice’s nonces for that bit using 1-2 OT.

   So Bob receives a list of $n$ values $x_{1,b_1}, x_{2,b_2}, \ldots, x_{n,b_n}$.

   Bob XORs all these values together to compute
   
   $$y = x_{1,b_1} \oplus x_{2,b_2} \oplus \cdots \oplus x_{n,b_n}$$

   If $y = 0$, then the strings (probably) match and Bob reports EQUAL back to Alice.

   If $y \neq 0$, then the strings definitely don’t match and Bob reports UNEQUAL back to Alice.

Why does this work? Well, the way Alice has set up the messages, she knows that

$$x_{1,a_1} \oplus x_{2,a_2} \oplus \cdots \oplus x_{n,a_n} = 0$$

That’s determined entirely by her non-random setting of the last value $x_{n,a_n}$. So if Bob’s bits $b_i$ are all the same as Alice’s bits $a_i$, then the result of his massive XOR computation will definitely come out to be 0.

But if Bob’s bits are not the same - if they’re off by even one bit - then what does he get when he computes $y$? Let’s define the set $S \subseteq \{1, 2, \ldots, n\}$ of indexes where Alice’s and Bob’s bits don’t match. That is,

$$S = \{i | a_i \neq b_i\} = \{s_1, s_2, \ldots, s_m\}$$

where $m$ is the number of bits where Alice’s and Bob’s messages do not match up.

Now remembering that $x_{1,a_1} \oplus \cdots \oplus x_{n,a_n} = 0$, we can see that Bob’s computed XOR value $y$ equals

$$y = y \oplus 0$$
\[(x_{1,b_1} \oplus \cdots \oplus x_{n,b_n}) \oplus (x_{1,a_1} \oplus \cdots \oplus x_{n,a_n}) \]
\[(x_{s_1,0} \oplus x_{s_1,1}) \oplus (x_{s_2,0} \oplus x_{s_2,1}) \oplus \cdots \oplus (x_{s_m,0} \oplus x_{s_m,1}) \]

In other words, Bob’s XOR computation \(y\) is the XOR of a bunch of pairs of nonce values that Alice chose. From the way Alice chose them, at least one (and usually both) number in each pair is a totally random nonce, meaning that the XOR of each pair is indistinguishable from random. So the XOR of all the pairs’ XORs - which equals Bob’s computed value \(y\) - is also completely indistinguishable from random unless \(m = 0\). That is, unless the two strings match up completely, Bob just gets some random meaningless value \(y\), which is almost certainly not going to equal 0.

### 4.1 Honest but curious

There is a rather obvious security flaw in the above protocol: What if Bob, after performing his \(n\) oblivious transfers and computing the correct XOR value \(y\) to learn the real answer, simply lies to Alice on the very last step when he reports back whether the two strings were equal?

This vulnerability is really just an underlying assumption in the protocol above, what’s called the honest but curious or “semi-honest” model. In this model, both parties of the protocol always follow the protocol exactly and do everything it says, but afterwards they remember everything and try to learn any additional information they might be able to.

So we know the protocol above is secure in this semi-honest model, but is not secure in the “malicious” setting where either Alice or Bob might deviate from the protocol in order to trick the other person.

We’re not going to focus much on this stronger “malicious security” setting, but it’s useful to point out that there are ways to achieve this higher level of security, and they usually involve putting a few extra cryptographic “checks” in place to make sure both parties are following the protocol correctly.

In the case of the oblivious equality check protocol, an easy way to get a higher level of security is for Alice to set it up so that the XOR of all the values

\[x_{1,a_1} \oplus \cdots \oplus x_{n,a_n}\]

is not exactly equal to an all 0’s string, but also includes a random “challenge” as part of it. For example, Alice might set it up so that this XOR equals a string of half 0’s and half random 1’s and 0’s according to some secret challenge she chooses. Then if Bob reports EQUAL at the end of the protocol, he must also send back the challenge string so that Alice can check he’s actually got it right.

This doesn’t unfortunately prevent Bob from lying about the UNEQUAL case, but at least prevents him from claiming to share the same secret message when they really do not.

### 5 Secure Function Evaluation

The oblivious equality check in the previous section is really just one specific example of a more general problem: Alice and Bob want to compute some kind of function based on their own secret inputs, without revealing those secret inputs to the other party.

For another silly example, say Alice and Bob are out to dinner and want to decide who’s going to pay for the meal. They decide the fair thing is for whoever makes more money to pay for it, so they want to figure out which of them has a higher salary. But they don’t want to reveal exactly how much either of them makes, just to be sure who makes more. This is basically saying that they want to compute the “greater-than” function on two inputs, without revealing their input to the other person.

Here’s the formal problem statement:
Secure Function Evaluation

Let $f$ be an arbitrary function $f(a, b) \mapsto \{0, 1\}$ that takes two input bit-strings $a$ and $b$, possibly of different lengths, and computes a single bit output.

The SFE problem is that Alice, who knows the input $a$, and Bob, who knows the input $b$, want to work together to compute $f(a, b)$ without revealing anything else about $a$ or $b$ to each other.

This is useful not just for the paying-for-meals example above, but in a lot of scenarios you can imagine. One real-world instance of this being used is in sugar beet auctions in Denmark. Yes, really. In any kind of auction or negotiation situation, it can be important for both parties to agree on something (such as a fair price or fair compromise) without revealing all the details of their positions to each other to avoid “gaming the system”.

Just like with OTPs for encryption, there is an easy and “perfect” solution here: get a “trusted third party” to do the computation. That is, Alice and Bob could each give their input to Trudy, then Trudy performs the computation and tells the answer back to Alice and Bob. (If Alice and Bob want a little extra security afterwards, Trudy’s life might be in danger.) Obviously the problem here is that in real-life negotiation situations or online transactions, there frequently isn’t any “middleman” such as Trudy that both parties trust completely.

The solution that doesn’t involve any trusted third party was first proposed in the 1980s by computer scientist Andy Yao. It’s called “Yao’s Garbled Circuits” and relies on both symmetric encryption as well as 1-2 Oblivious Transfer. The basic outline is that Alice (the “creator”) creates an encrypted boolean circuit that Bob (the “evaluator”) evaluates using his secret inputs and Alice’s encrypted inputs. We’ll go through the circuit creation process, the circuit evaluation process, and finally the protocol which brings it all together.

Both of these assume that the function $f$ is represented as a boolean circuit with wires and gates. Some of the wires are “inputs” corresponding to the bits in Alice’s or Bob’s original secret inputs, and one of the wires is designated as the “output” from the entire circuit. For simplicity, we’ll assume each gate has 2 inputs and exactly 1 output.

5.1 Alice: Garbling the circuit

Similarly to the oblivious equality check, Alice is going to choose 2 random nonces for each wire, corresponding to whether that wire’s value is 0 or 1. Bob will learn only one out of each pair, then he does his computation to figure out the output of the entire circuit.

Actually, Alice chooses random values for almost every wire in the circuit. The one which is not randomly chosen is the very last wire, the output from the entire circuit. This wire’s true/false values will not be set randomly, but can be set to simply 0 and 1 for true and false. That way, when Bob finishes evaluating the circuit, he knows the output from the function.

The tricky part for Alice is how to encode the gates in the circuit. Since Bob won’t have the actual true/false values of the input wires, he can’t just evaluate the gate’s operation like normal. And even if he could, he’s not trying to learn the true/false value of the output wire, but rather one of the two secret values for the output wire.

The trick here is for Alice to encrypt the truth table of each gate in the circuit. Specifically, for each gate that has two input wires and one output wire, Alice encrypts the proper output wire for each of the four possible settings of the input wires, by encrypting the output wire value (twice) using the input wire values as keys for the symmetric encryption.

For example, supposed we have an AND gate with two inputs $x$ and $y$ and output $z$. We’ll write $x^F, x^T$ to represent the two random values representing true/false for that wire; similarly for the other wires $y$ and $z$. The actual truth table for this AND gate looks like this:
Each entry in the encrypted truth table is the double-encryption of one of the $z$ values, using one of the $x$ values and one of the $y$ values as the keys:

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>Encrypted truth table entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x^F$</td>
<td>$y^F$</td>
<td>$\text{Enc}<em>{x^F} (\text{Enc}</em>{y^F} (z^F))$</td>
</tr>
<tr>
<td>$x^F$</td>
<td>$y^T$</td>
<td>$\text{Enc}<em>{x^F} (\text{Enc}</em>{y^T} (z^F))$</td>
</tr>
<tr>
<td>$x^T$</td>
<td>$y^F$</td>
<td>$\text{Enc}<em>{x^T} (\text{Enc}</em>{y^F} (z^F))$</td>
</tr>
<tr>
<td>$x^T$</td>
<td>$y^T$</td>
<td>$\text{Enc}<em>{x^T} (\text{Enc}</em>{y^T} (z^T))$</td>
</tr>
</tbody>
</table>

What Bob will see is just the list of these four encrypted truth table values, in no particular order so that he doesn’t know what any of them means. As we’ll see next, Bob will know one pair of the input wire values, which allows him to successfully perform one of the four decryptions and get one of the values for the output wire.

**In summary**: Alice chooses 2 random nonces for every wire except the output wire, and computes 4 encrypted values for every gate using the wire values as keys.

### 5.2 Bob: Evaluating the garbled circuit

Bob will receive from Alice the full list of encrypted truth tables. Bob will also have one value for each input wire, corresponding to his and Alice’s actual inputs. Crucially, *Bob doesn’t know what the wire values mean* (except for his original input bits). So Bob has one of the two possible wire values, but he doesn’t know if that value indicates “true” or “false”.

Bob’s job of evaluating the circuit boils down to computing the output wire each gate, given the input wires to that gate and the garbled truth table from the gate that he received from Alice. From the way Alice encrypted the truth table, since Bob has one pair of input wire values, he should be able to decrypt one of the four encrypted truth table entries and get the proper output wire value. But how does he know which value to decrypt?

The answer is, he doesn’t! Bob can just try decrypting all four truth table values, and whichever one decrypts properly, must be the correct output wire value. (In practice, symmetric encryption schemes have what amounts to “check bits” to let you know whether the decryption worked properly. For the example we did in class, we said it works properly if the output was less than 10.)

Specifically, given the four garbled truth table entries $G_1, G_2, G_3, G_4$, and the two input wire values $X$ and $Y$, Bob tries to compute four decryptions:

- $\text{Dec}_Y (\text{Dec}_X (G_1))$
- $\text{Dec}_Y (\text{Dec}_X (G_2))$
- $\text{Dec}_Y (\text{Dec}_X (G_3))$
- $\text{Dec}_Y (\text{Dec}_X (G_4))$

If his input wires are correct, and the garbled circuit is correct, then one of these decryptions will work correctly and yield the proper output wire value $Z$. Bob then uses that as input to the next gate, and so on for all the gates until getting the final output wire value from the entire circuit. Since the last output wire value is not encrypted by Alice, Bob now knows the result of the secure function evaluation.
5.3 The protocol

Now let’s put these pieces together to describe the entire protocol for SFE using Yao’s garbled circuits.

1. Alice writes down the boolean circuit and labels all the wires and gates. She chooses two random nonce values for every wire except the final output wire, which is assigned non-random values 0 and 1.

   Alice then encrypts the truth table for each gate, using the input and output wire values for that gate.

   Alice sends to Bob the following:
   - A description of the circuit (which wires are connected to which gates and so forth)
   - The four values in the encrypted truth table for each gate
   - One of the two wire values for each wire representing Alice’s input bits.

   (Note: Alice does not send Bob any wire values other than the values for her input bits.)

2. Bob receives the garbled circuit from Alice. This includes the encrypted wire values for Alice’s inputs.

   To get the wire values for Bob’s inputs, Bob executes a 1-2 OT protocol for each of his input wires. This way Alice sends him his input wire values, without him learning the other wire value and without Alice learning his secret input bits.

   Bob then evaluates all the gates in the circuit to compute all the encrypted wire values. Finally, he computes the wire value of the final output, which reveals to him the output of the entire function.

   Bob shares this final output wire value with Alice so she learns the output as well.

   The security is maintained because of the properties of 1-2 OT, as well as the assumption that Bob never learns both values for any wire in the circuit. This guarantees security in the “honest but curious model”. It is also fairly straightforward to modify the output wire values to make sure Bob doesn’t lie at the end, just like with the equality check protocol.

5.4 In-class example

Here is a pdf of our in-class example of garbled circuits.

6 Oblivious RAM

Secure computation is a big deal. We just spent some time talking about multi-party secure computation, where Alice and Bob (and maybe others!) want to work together in order to compute some function, without each of them learning the other person’s private input.

When you get down to the hardware level, even single-person secure computation becomes significant. What this means in practice is something called a cryptoprocessor which is like a limited-purpose CPU that has some encryption/decryption capabilities and which is also considered tamper-resistant for extra security. For example, the chip in your CAC is a cryptoprocessor, and recent iPhones have a special CPU for crypto called “Secure Enclave”. Most desktops and laptops sold within the last decade also have a cryptoprocessor built in to the motherboard, which is called a TMP or “Trusted Platform Module”.

The main vulnerability if you actually want to use any of these things to run a program, for example, is that at some point you’re going to rely on something outside of your trusted processor. In particular, you’re going to read and write from main memory (RAM).

Oblivious RAM is a way of hiding (encrypting, essentially), not only the contents of memory but also the access pattern, which is the sequence of which memory addresses are accessed, and in which order.
Here is a terrible ASCII art picture of what’s going on. Notice that we’re going to use the term label to mean the secret internal addresses of each block of memory as the CPU see it, and address to mean the scrambled, publicly-viewable external actual addresses of where the encrypted data is being stored.

```
    CPU <----- ORAM <----- Memory
    \  load/store    \  load/store    \    
    by "label"        by address
```

Formally, you can think of ORAM as a “middle-man” that sits between the processor and memory. It takes each instruction from the processor and translates it into some series of memory instructions for the outside. So the processor just thinks it’s accessing a normal RAM, but someone who is eavesdropping on the actual memory has no idea what’s going on.

Here is a somewhat more precise definition:

An **Oblivious RAM (ORAM) protocol** is a way of translating any (sequence of) LOAD/STORE commands on internal labels, to a sequence of LOAD/STORE commands on external addresses, with the following properties:

1. (correctness) The results from the ORAM, from the internal view, are exactly the same as they would be from a regular, unencrypted RAM.

2. (security) Anyone who sees the entire sequence of external commands, and all the data that passes between the ORAM and main memory, cannot determine anything about the internal commands other than (possibly) how many operations occurred.

There are more formal, detailed definitions than this in research papers but this gives the main idea of what ORAMs are all about.

### 6.1 ORAM 0 (Transfer everything)

As a warm-up to see how ORAM works, you can imagine an ORAM that just copies the entire memory contents, and re-encrypts them, on every single access.

That is, any internal LOAD or STORE command causes the entire contents of external memory to be first downloaded and decrypted. Then the relevant LOAD/STORE command is actually executed, which involves either looking something up in the unencrypted data or changing part of it, and finally the entire contents of memory are re-encrypted and written back to external memory.

This satisfies both conditions of an ORAM, but in a really inefficient way. If the total size of memory is $n$, doing it this way means every option costs $O(n)$ time and requires $O(n)$ amount of data transfer as well. So it works, but it’s horribly inefficient.

### 6.2 ORAM 1 (Square root ORAM)

The first efficient ORAM solution was proposed by Oded Goldreich and Rafail Ostrovsky in the 1990s. The main research paper they wrote is Software protection and simulation on oblivious RAMs, published in the Journal of the ACM in 1996.

Here’s the main idea: you keep a “stash” of size $O(\sqrt{n})$ that holds the most recently accessed memory locations. Every time there is another operation, you first check if that label is in the stash, and if so you just deal with it in the stash and just read some random-looking location from the rest of memory. Otherwise, if it’s not in the stash, then you have to go ahead and look it up in memory wherever it’s stored, and then you store it in the stash for the next look-up.

There are a few difficulties with this basic idea, that have to be solved:
1. **How do you make the non-stash memory locations look random?**

Remember, an eavesdropper gets to see every external address that the ORAM reads or writes to. So if those addresses have *any* relationship with the internal labels, that’s going to leak out some secret information.

The solution is to use hashing with cryptographically-secure hash functions, like we’ve been learning about. Specifically, because the non-stash memory locations are written all at once in batches, we can use any *perfect static hashing* routine. We learned about the FKS hashing algorithm which will work perfectly well; cuckoo hashing is another solution that has also been suggested by some recent research papers.

2. **What about when the stash gets full?**

After $O(\sqrt{n})$ operations to different locations, the stash will become totally full, so you have to somehow “flush” it back to the rest of memory.

Actually, you *have to do this even if the stash isn’t actually full*, because if you didn’t it would reveal that you accessed the same place in memory twice!

The “flushing” operation involves three stages. First, you read in everything from memory and combine it with whatever’s in the stash. Since the stash contains all the more recent changes, you probably have to add/remove/update some contents of the main memory.

Then, you pick a new random hash function and re-do the perfect static hashing algorithm (such as FKS) to move everything to a new, random-looking location. Then you write everything back to those locations, and clear the stash.

Now you’re ready for a new “round” of $O(\sqrt{n})$ more operations before the next flush and re-hashing occurs. Importantly, within each round, the same memory address will never be accessed twice, and there is zero correlation between the memory addresses from one round to the next one. This is what gives the security properties that ORAM needs.

3. **What “dummy” memory location do you look at when the accessed item is already in the stash?**

You *have* to access some memory location outside of stash, because otherwise that would reveal that the thing you’re looking up was already in the stash, which would leak information about the access pattern to an eavesdropper.

The solution is to add $O(\sqrt{n})$ “dummy” locations to main memory, and you access them sequentially whenever you lookup something that’s already in the stash. This way, you never access the same place in memory twice, and they all look equally random to any observer.

Overall, you get basically the following algorithm:

```plaintext
count = 0
for each internal LOAD(label) or STORE(label) operation:
    Download and decrypt entire stash.
    If label is in stash:
        read the next dummy location, at address hash(n + count)
    else if label is not in stash:
        read the label from memory at address hash(label)
    Update contents if necessary, and add label to stash
    count = count + 1
    if count == sqrt(n):
        Download and decrypt entire memory contents
        Incorporate stash into memory contents.
        Re-hash and write updated, encrypted memory to new locations
        Clear the stash
```
count = 0
else:
    Encrypt stash and store entire stash

For most operations, the cost is dominated by the loading and storing of the entire stash, which costs \( O(\sqrt{n}) \). Once every \( \sqrt{n} \) operations, you also have to re-hash the entire memory, which costs \( O(n) \). So the total amortized cost over any sequence of \( \sqrt{n} \) operations is

\[
\frac{\text{cost of ops}}{\text{number of ops}} = \frac{\sqrt{n} \cdot \sqrt{n} + n}{\sqrt{n}} = O(\sqrt{n})
\]

Hence the name, square root ORAM.

### 6.3 ORAM 1.1: Hierarchical ORAM

The 1996 paper by Goldreich and Ostrovsky actually goes beyond the \( O(\sqrt{n}) \) ORAM construction above and describes something better and more complicated called hierarchical ORAM.

The idea is that the square root ORAM has 2 levels: the stash, and the rest of memory. Well you could extend this idea to 3 levels, where you would have a small stash, then an intermediate stash, and then the rest of memory. You still read and write the entire small stash on every access, then when that gets full you have to re-hash and re-write the intermediate stash. When the intermediate stash gets full, you have to re-hash and re-write the entire memory.

But why stop at 3 levels? If you take this to its mathematical conclusion, you end up with \( O(\log n) \) levels of intermediate stashes, all of which have their own randomly-chosen perfect hash functions and which are re-hashed with different frequencies depending on their size.

If you use FKS or cuckoo hashing, the total cost is essentially \( O(\log n) \) amortized lookups per operation, which is also essentially optimal.

But it’s important to point out one of the biggest shortcomings here (as well as with the square root ORAM): The re-hashing and “shuffling” of the entire memory, requires \( O(n) \) local storage space for when you’re choosing new hash functions. This is fine in some settings, but in other contexts (such as a secure cryptoprocessor with limited cache and registers), it would seem to defeat the entire purpose of having external memory in the first place. There are complicated ways around this issue (namely, oblivious sorting algorithms), but are a bit more than what we have time to cover, and they add extra complication as well as asymptotic cost to the protocol.

Instead, we’ll focus on a more recent improved construction that achieves a similar cost without the need for lots of local storage:

### 6.4 ORAM 2: Path ORAM

Path ORAM is an idea first developed by Emil Stefanov and Elaine Shi who (at the time) were both at UC Berkeley. Their main, detailed but nicely-written paper, is Path ORAM: An Extremely Simple Oblivious RAM Protocol.

The main idea, as we saw in class, is to organize the external memory into buckets of a complete binary tree, stored in an array like is done with binary heaps. Each item is assigned a random position corresponding to one of the leaf nodes, and this position is randomly re-assigned every time that item is accessed.

The item itself does not have to be stored exactly in the leaf node corresponding to its position, but it has a little more flexibility. The key property, maintained at all times, is that every item is stored somewhere along the path from the root node to the leaf node indicated by that item’s position. Hence the name, Path ORAM.
Whenever you access some item, you first have to look up its position (more on that a bit later). Then you do three things:

1. Remove all the items stored along the path from the root to that position’s leaf node.
2. Randomly reassign the item’s position to some other, randomly-chosen leaf node.
3. Re-write everything along the old, original path, moving everything as far down towards the leaf nodes as possible.

Basically, as we saw in class, what happens is that when you do an operation, the one item you’re looking up will likely move up in the path towards the root, and all the other items along the path will move down towards the leaf. Note that all the other items, since they have randomly-assigned positions, have nothing to do with the item you’re actually accessing. They’re just sort of “along for the ride”.

This is a beautiful, simple idea with turns out to be very effective in practice as well. There are really just two issues with it:

First, the path might get full. Even if the entire path isn’t full, some top portion of the path might get full (since not all items along the path can move all the way down to the leaf). This is called overflow, and it’s solved by creating another stash! Any items that can’t be stored along the path go into stash temporarily, and are removed from stash on some later operation as soon as there’s room.

The second issue is where do you store the positions. Unlike with the square root ORAM, you’re not shuffling the entire contents of memory all at once, but just one item at a time. So it’s not so easy as just using a hash function and periodically re-hashing. Instead, we have to store a map of the positions within the ORAM itself. In class, we saw how a B-tree could be used to do this as long as each block of storage is large enough, like $10 \log n$ bits or something like that.

Now how do we know all this will work? If things go wrong, who’s to say that the root bucket as well as the entire stash won’t get full, causing data loss and/or loss of privacy? This is called a stash overflow, and to avoid it, there are two key parameters that have to be chosen very, very carefully:

- Stash size, the number of items that you hold temporarily when they overflow out of the root node of the tree.
- Bucket size, the number of items that can go in each node of the tree.

If you have a larger bucket size, then there’s less chance that the tree will overflow into stash, but it also makes every operation that much more expensive. And if you have a larger stash, that decreases the chance of stash overflow but also makes every operation more costly since you have to read and write the entire stash every time.

It turns out things are optimized for low probability of stash overflow when you set the stash size to $O(\log N)$, where $N$ is the total number of operations, and the bucket size to some constant that’s at least 5. The detailed proof is in the paper linked above, and we also discussed some of the main ideas of the proof in class.