**Schnorr signature:**
Public parameters: g which is a generator of Gq, p which is a safe prime, and q. To sign, we have a function \( S = \text{Sig}_x(m, r) \). x is the secret key. m is the message being signed. r is a random factor.

Key generation is only done once. x is the secret key in \( \mathbb{Z}_q \). Public key \( y = g^x \mod p \).

Signature \( \{s_1, s_2, s_3\} = \{g^r, H(m|s_1), s_2x+r\} \). To verify, we have a function verify(m, s, y). Takes the message m, the signature \( \{s_1, s_2, s_3\} \) s, and takes y which is the public key of the person that signed it. You want to check if \( s_1y^{s_2} = g^{s_3} \). You also want to check if \( s_2 = H(m|s_1) \). You can also drop s2 by checking if \( s_1y^{H(m|s_1)} = g^{s_3} \) instead. This is exactly the same as a zero knowledge proof, except the message is included in the hash.

**Schnorr Variants:**
S3 = S2x + r is what we'll use in the class. x is the secret key. s3 and s2 are in the signature. In order to not be able to solve for the secret key with 1 equation, we need 2 unknowns. We know gx and gr, but can't solve for x or r because of the DLP.

There are different ways of writing out S3, which are variants of Schnorr.
- \( S_3 = s_2x + r \) (In-class method)
- \( S_3 = s_2x - r \) (textbook method, makes verification more simple)
- \( S_3 = r^{-1}(s_2 + xs_1) \) (Method used in practice, DSA, basically Schnorr)

Schnorr came with zero-knowledge proof, Fiat-Shamir modified Schnorr, DSA was built separately.

It may be possible to solve for x if you use the same r in multiple messages. You can extract the private key out if you have 2 messages with the same r.

**CCA-Secure Public-Key Encryption:**
Recall ElGamal is CPA secure because it can be reduced to the DDH assumption, but not CCA secure because Alice can send m1, m2, and Oracle sends back cb, but Alice can calculate cb’ Enc(1), or you can rerandomize it. In order to be CCA secure, it should also be infeasible to pick random values and get a valid ciphertext. The naive approach is to do CPA-secure encryption and then sign it. ElGamal + Schnorr Signature, you would output \( \{c_1, c_2, s_1, s_2, s_3\} \), but cryptographers don't like it because it needs 2 keys, and involves sending 5 values.

Another method (First one) that was provably CCA-secure was Cramer-Shoup, and had 4 elements and 1 key, but the key was much bigger.

Another method that is CCA-secure is Twin Elgamal, which is a few years old, unlike the other things which are like 40 years old.

Twin Elgamal has 3 public parameters p, q, g. Secret key is \( \{x_1, x_2\} \). Public key \( y_1 = g^{x_1} \mod p \). \( y_2 = g^{x_2} \mod p \). Enc pk(m, r) = \( <g^r, \text{SEnc}_x(m, IV)> \). It can use any CCA-secure mode of operation. \( k = H(g^r, y_1^r, y_2^r) \). To decrypt, \( \text{Dec}_k(<c_1, c_2>) = \text{SDec}_k(c_2) \). We know the \( k = H(c_1, c_1^{x_2}, c_1^{x_2}) \). This is equivalent to \( H(gr, (g^{x_1})^r, (g^{x_2})^r) \). It is hard to pick a random value that is valid too. Note SEnc is any method of encryption that uses a symmetric key.
**Diffie-Hellman:**
Is a key exchange protocol. It is secure unless you have an active adversary that can change values rather than just eavesdrop. Vulnerable to man-in-the-middle attack where Adversary decrypts from one and reencrypts to send the other, not letting them know they aren't. In order to solve this, we assume Alice knows Bob's public key, and Bob knows Alice's public key. Called the Station-to-Station protocol, is the Diffie Hellman with signed values, developed by Prof here at Carleton. Alice has Public key PKA, generates a € [1,q], Ya = gᵃ mod p. Alice sends Ya, Bob has Public key PKB generates Yb = gb mod p, sends it to Alice, signing with sign_ab(Yb, Ya). We can also symmetrically encrypt the signed thing, forcing Alice to decrypt to ensure she got the right key. Alice then sends back the same thing, signing with her key: SEnck(Sign_skA(YA, YB)). Bob can decrypt with the public key, followed by verifying the signature with PKA.

If adversary changes YA, then Bob sends back with changed YA and then Alice can not decrypt properly.

**How do Alice and Bob learn each other's public keys?**

Lets say we only have symmetric keys. Say we have a bunch of people that want to share messages. Alice has to share a key with Carol, Alice has to share a key with Bob, etc. So we need n² keys in total, with 1 for each pair, where n is the number of people. Now if we use public keys, each person has their own keys, so we have only O(n) keys total, instead of O(n²)

Certificates allow us to use a trusted third party, called a certificate authority. Lets say the certificate authority is Carleton University. Prof wants to send a message to Alice, sends it to the certificate authority instead who will sign it then. Say Bob wants Alice's public key, Bob requests Alice's public key, Alice sends her PKA and her Cert. Then Bob can verify the Cert with the public key of the certificate authority. Now if an attacker wants to change the PKA, he needs to get a certificate from the certification authority, who we trust to only give public keys out to people that should get them. This is called a centralized public-key infrastructure. This is how the real-world internet works. In order to know the keys of the certificate authorities, they are built into the browser. Also, each key can be used to sign other keys. So basically there are around 800 people that are certificate authorities that can sign things that shouldn't be signed.

**RSA Setting**
RSA is the first public-key setting. We will work mod N where N = pq, and p, q are large safe primes.

An element a of Zn is invertible iff gcd(a, n) = 1. p, 2p,...(q-1)p are not invertible. q, 2q,...(p-1)q are not invertible. Everything else is invertible. Zⁿ* excludes non-invertible numbers. The order of an element a in Zⁿ* is from (p-1)(q-1), so it is divisible by {1, 2, 4, ^p, ^q, ^p ^q, (p-1)(q-1)}. The order of any element n is equal to the euler function of n, Ord(a) = φ(n), which is all the numbers less than n that are coprime to n. φ(p) = p-1. φ(n) = (p-1)(q-1). The number of elements in Zⁿ* is (p-1)(q-1).