# Lecture 14: Public-Key Cryptography

These scribe notes were written by students in CSE 127 Winter 2021. They have been lightly edited but may still contain errors.

## 1 Introduction

In the previous lecture, we discussed symmetric cryptography and how it is used to securely send information between parties. This system hinges on the involved parties having a secret and symmetric key to encrypt and decrypt each other's messages. This leads to the issue of key distribution, as those communicating need some way to agree on a key without meeting in person or sending it over an insecure channel. Here we discuss public-key cryptography, which gives among other things a method to solve this issue of symmetric key distribution. We will go over key exchange, public-key encryption, and digital signatures.

## 2 Key Exchange

### 2.1 Asymmetric Cryptography

Asymmetric cryptography refers to using both a public key and a private key. The public key is known to everyone and is used to encrypt messages and verify digital signatures. The private key is known only to the user and is used to decrypt the ciphertext or generate digital signatures.

$$\text{Encrypt}_{\text{public key}}(\text{message}) = \text{ciphertext} \tag{1}$$
$$\text{Decrypt}_{\text{private key}}(\text{ciphertext}) = \text{message} \tag{2}$$

Since the public key is safe to publish and available to everyone, anyone is able to send a secret message to a user. Intercepting the secret message would also not be an issue, as they would be unable to decrypt the secret message without the user's private key. This is called asymmetric cryptography because the public key and private key differ. Public-key encryption must also follow the correctness property

$$\text{Decrypt}_{\text{private key}}(\text{Encrypt}_{\text{public key}}(\text{message})) = \text{message} \tag{3}$$

### 2.2 Modular Arithmetic

Several of the public-key cryptography algorithms that are in use today are based off of modular arithmetic. Since these are the easiest to understand, we will cover them in this class, so we need to review modular arithmetic.
Recall how division and remainders work, with * meaning "normal" integer multiplication.

$$n = (q * d) + r$$
$$n \equiv r \bmod d$$

In this example, d is the modulus and r is the remainder.

$$\text{Addition: } (a \bmod d) + (b \bmod d) \equiv (a + b) \bmod d \tag{4}$$
$$\text{Subtraction: } (a \bmod d) - (b \bmod d) \equiv (a - b) \bmod d \tag{5}$$
$$\text{Multiplication: } (a \bmod d) * (b \bmod d) \equiv (a * b) \bmod d \tag{6}$$
$$\text{Exponentiation: } (g^a \bmod d) \equiv (g * g ... * g) \bmod d \tag{7}$$

| Operation | Value 1 (a) | Value 2 (b) | Modulus (d) | Remainder (r) |
|---|---|---|---|---|
| Addition | 11 | 16 | 6 | $27 \bmod 6 \equiv 3$ |
| Subtraction | 9 | 3 | 4 | $6 \bmod 4 \equiv 2$ |
| Multiplication | 7 | 3 | 6 | $21 \bmod 6 \equiv 3$ |
| Exponentiation | 3 | 4 | 4 | $81 \bmod 4 \equiv 1$ |

Table 1: Modular Arithmetic Examples

From the above equations we see that addition, subtraction, multiplication, and exponentiation work as you expect even with modular reduction. Modular exponentiation is defined analogously to what you expect from the integers ($g^a$ means multiply $g$ by itself $a$ times). This is not a computationally efficient algorithm, but there are computationally efficient algorithms to compute modular exponentiation. We run into some issues when trying to define an analogue of "division" for modular arithmetic. It turns out the right way to think about division is called a modular inverse.

$$b * b^{-1} \equiv 1 \bmod d \tag{8}$$

In this instance, $b^{-1}$ is the modular inverse of $b$, and is defined as the value such that if you multiply it by $b$ you get 1 mod $d$. However, the modular inverse is not well defined for all possible values of $b$ and $d$. In order for a modular inverse to be well defined, the greatest common divisor between b and d must be 1 (relatively prime). For our purposes, this tells us that any integer between 1 and $p-1$ will have a modular inverse modulo a prime $p$.

$$\begin{aligned} \text{Example:} \quad & b = 4 \text{ and } d = 7 \\ 4 * (4^{-1} \bmod 7) &\equiv 1 \bmod 7 \\ 4 * 2 &\equiv 1 \bmod 7 \\ \text{The modular inverse is } 2 & \end{aligned}$$

There is also an inverse operation for modular exponentiation, which is called discrete log.

$$\text{Suppose } b^a \equiv y \bmod d \tag{9}$$
$$\text{Then } \log_b y = a \tag{10}$$

In this instance, a is the discrete log. There is no known polynomial-time algorithm to compute this value.

$$\begin{aligned} \text{Example: } b = 5,\ d = 7, \text{ and } y = 4 \\ 5^a \equiv 4 \bmod 7 \\ 5^2 \equiv 4 \bmod 7 \\ \text{The discrete log is } 2 \end{aligned}$$

## 3 Key Exchange

### 3.1 Diffie-Hellman Key Exchange

In Diffie-Hellman key exchange, two parties Alice and Bob wish to exchange public information visible to a passive eavesdropper who can see all the messages that were exchanged, and at the end obtain a mutual "shared secret" that the attacker cannot compute even though she has seen every message that was transmitted. Diffie-Hellman works as follows:

1. The public, global parameters decided on in advance and known to all parties are some prime number $p$ and some integer $g$ which is a primitive root modulo $p$
2. Alice chooses an integer exponent $a$ and sends Bob $g^a \bmod p$
3. Bob chooses an integer exponent $b$ and sends Alice $g^b \bmod p$

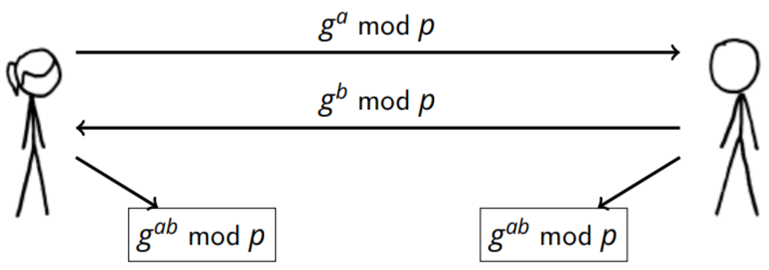


Figure 1: Diffie-Hellmann Key Exchange

Using modular exponentiation, we know that $(g^a \bmod p)^b$ and $(g^b \bmod p)^a$ are the same. Thus, Alice and Bob share the same secret, $g^{ab} \bmod p$ while any passive adversary can only know $g^a \bmod p$ and $g^b \bmod p$, from which it seems to be hard to obtain $a$ and $b$.

In particular, Diffie-Hellman security relies on the assumption that the discrete log problem is a hard problem for some values of $p$ and $g$, and that we are dealing with a passive, not active, adversary. In the case that we pick a non optimal $p$ and $g$, the discrete log may become trivial and a passive adversary could easily obtain the secret key that only Alice and Bob are supposed to share, which is why you should never try to implement Diffie-Hellman yourself even though it looks mathematically simple. In the case of an active adversary we present the following scenario.

Say we have an active adversary who acts as a man-in-the-middle, Mallory. When Alice sends Bob $g^a \bmod p$, since Mallory is an active adversary, she intercepts the message, comes up with some exponent $m$, and sends Bob $g^m \bmod p$. Similarly she will intercept Bob's message to Alice, coming up with some exponent $n$, and sending $g^n \bmod p$. Alice and Bob will not know that their messages have been intercepted, so in the end Alice will think their shared secret is $g^{an} \bmod p$ and Bob will think their shared secret is $g^{bm} \bmod p$. When Alice sends a message encrypted using what she thinks is their shared secret, Mallory can now decrypt the message since she has the original $g^a \bmod p$ and the exponent $n$. She can also do this with encrypted messages sent by Bob since she has the exponent $m$.

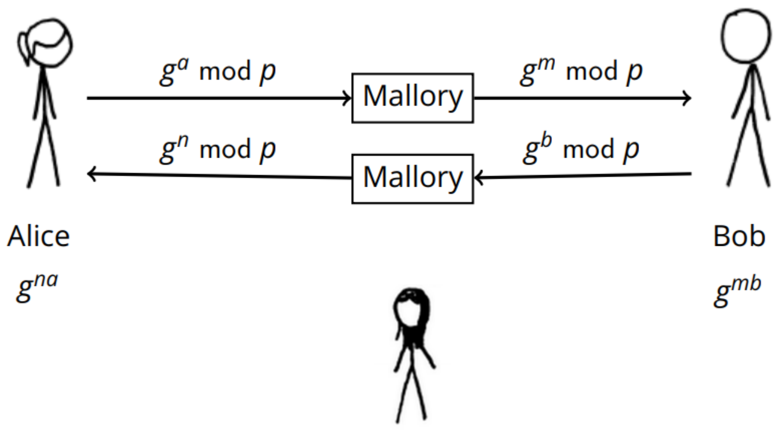


Figure 2: Diffie-Hellmann Key Exchange

Thus Diffie-Hellman is not secure against active adversaries, and fixing this weakness requires additional means such as a way to authenticate messages sent and/or ways to keep messages from being modified in transit.

## 4 Public Key Encryption

In the last section, we've seen how the Diffie-Hellman key exchange is used to have two parties agree on a shared key. Once Alice and Bob have agreed on a shared key, they can then use a symmetric cipher to encrypt the data they wish to transmit.

There is a second idea in public-key cryptography that is called public-key encryption. Public-key encryption means the algorithm has two keys: one public and one private. In this section, we explore public key encryption and the RSA encryption algorithm, named after the algorithm's inventors Ron Rivest, Adi Shamir, and Len Adleman.

## 4.1 RSA Encryption

Similar to how the Diffie-Hellman key exchange relies on discrete logarithms being computationally difficult, RSA encryption relies on prime factorization of large numbers being computationally difficult. We want the calculation to be easy to perform in one direction, but difficult to invert if you don't have the proper key.

For the algorithm to help Bob communicate with Alice, Bob would need to create a public key for Alice to encrypt messages to, and a private key to help Bob decrypt Alice's messages. RSA key generation works as follows:

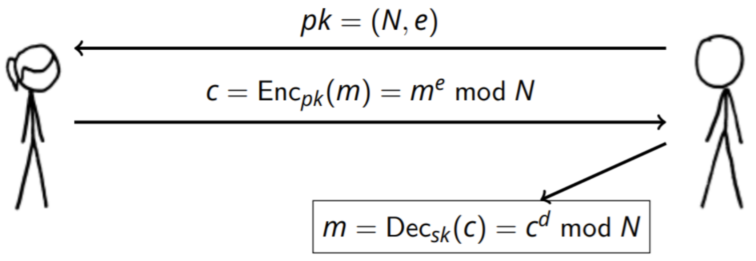
1. Take two large prime numbers, $p$ and $q$, and multiply them to get a number $N$. Keep $p$ and $q$ secret.
2. Calculate the totient of $N$: $\phi(N) = (p-1)(q-1)$
3. Find a positive integer $e$ that is less than $\phi(N)$ and is co-prime with $\phi(N)$, meaning the greatest common divisor between $e$ and $\phi(N)$ is 1.
4. Calculate the number $d = e^{-1} \bmod \phi(N)$. This means that $e \cdot d \equiv 1 \bmod \phi(N)$, so $d$ and $e$ are modular inverses of one another.

The pair of integers $(N, e)$ is Bob's public key and will be shared by everyone who wants to communicate with him. The numbers $p$, $q$, and $d$ are part of Bob's secret key that will be kept to himself.
So, if Alice wants to send a message $m$ and encrypt it into ciphertext $c$ to send over to Bob, she would perform the calculation:

$$\text{Enc}(m) = m^e \bmod N = c \tag{11}$$

Then, Bob can decrypt Alice's message by doing the following:

$$\text{Dec}(c) = c^d \bmod N = m \tag{12}$$



$pk = (N, e)$
$c = \text{Enc}_{pk}(m) = m^e \bmod N$
$m = \text{Dec}_{sk}(c) = c^d \bmod N$

Figure 3: RSA encryption workflow for Alice to send a message to Bob

The asymmetric encryption correctness property holds in that $\text{Dec}(\text{Enc}(m)) = m^{ed} \bmod N \equiv m \bmod N$ due to the fact that we chose $d$ and $e$ to be multiplicative inverses of each other. The correctness of RSA encryption can be verified using Euler's theorem, which we will not detail here.

## 4.2 RSA Security

Observe that as the key generation algorithm is defined in the previous section, if the attacker could factor the integer modulus $N$ in the public key into its prime factors $p$ and $q$, the attacker could efficiently compute the secret key the same way that the owner of the key did in the first place. Thus RSA is only secure if integer factorization is hard for some kinds of integers.

The structure of the RSA modulus is chosen to avoid some obvious and less-obvious attacks via factoring algorithms. For example, making $N$ the product of two equal-sized primes $p$ and $q$ avoids some known attacks. In modern usage, we want $N$ to be greater than 2048 bits to avoid currently known factoring algorithms. And much like the case of Diffie-Hellman, there are many other subtle properties of RSA key generation and decryption that can lead to vulnerabilities if they are not implemented properly, even though the algorithm looks simple.

As an example of how the RSA algorithm we just presented above is vulnerable, RSA is totally insecure if the message is not padded. For example, we can modify a message sent from Alice and send the forged message for Bob to read. This property is called malleability. For example, let $c$ be a valid ciphertext generated by Alice that is the encryption of some message $m$. Mallory can generate a ciphertext that is the encryption of $m \cdot a$ for any $a$ of Mallory's choosing even though Mallory may have no idea what the message $m$ is:

1. Mallory intercepts Alice's encrypted message $c = m^e \bmod N$
2. Mallory chooses a random number $a$
3. Mallory calculates $c' = c \cdot a^e \bmod N$ since $c \cdot a^e \bmod N \equiv m^e a^e \bmod N \equiv (ma)^e \bmod N = \text{Enc}(ma)$
4. Mallory sends the forged ciphertext $c'$ to Bob.

We can also perform an attack on an unpadded RSA encryption algorithm to recover an original plaintext $m$ sent from Alice to Bob, assuming that Bob is willing to decrypt some messages or reveal information about plaintexts that don't seem sensitive. This attack is known as a chosen ciphertext attack and works as follows:

1. Mallory performs the same malleability attack explained above, obtaining $c' = \text{Enc}(ma) = c \cdot a^e \bmod N$ for a chosen random number $a$, and sending it to Bob.
2. Bob calculates the plaintext $ma = m' = (c')^d \bmod N$.
3. Bob sends $m' = ma$ to Mallory.
4. Mallory calculates original message $m = ma \cdot a^{-1} \bmod N$.

The reason why unpadded RSA encryption is insecure is that the algorithm is homomorphic under multiplication, meaning we can make computations on the ciphertext without needing the key. Thus RSA must always be used with a padding scheme, which means that the raw message is *padded* with some value before the textbook RSA encryption method is applied to it, and the padding is stripped off in the course of decryption before the raw message is returned.

### 4.2.1 RSA Padding

One method of padding is RSA PKCS #1 v1.5. It pads a message $m$ with the following function:

$$\text{pad}(m) = 00\ 02\ [\text{random padding string}]\ 00\ [m]$$

With this, Alice can send her message to Bob by padding the message first and then performing textbook RSA encryption as described above, and Bob can apply textbook RSA decryption to get Alice's padded message and strip off the padding to recover the original message.

There are RSA padding methods that are provably secure, but this method is not one of them. Nevertheless RSA PKCS #1 v1.5 encryption padding is widely used today despite the fact that it is very difficult to secure against a class of chosen ciphertext attacks called padding oracle attacks.

# 5 Digital Signatures

A digital signature is a public key algorithm used to validate the authenticity of a message. A digital signature is a value bound to a particular message that can be verified to ensure two properties: only the holder of the private key could have generated a valid signature on the message, and that the message has not been modified from the time it was signed. A digital signature consists of the following:

1. **Signing algorithm:** (secret key, message) $\rightarrow$ signature

$$\text{Sign}_{sk}(\text{message}) = s \tag{13}$$

2. **Signature verification algorithm:** (public key, message, signature) $\rightarrow$ bool
   Either accepts or rejects the message's claim of authenticity based on the inputs: public key, message string, and signature tag.

$$\text{Verify}_{pk}(\text{message}, s) = \text{true} \mid \text{false} \tag{14}$$

3. **Key generation algorithm:** Generates a private key and a corresponding public key.

Two main signature properties are required. First, the correctness property for a digital signature means that a digital signature generated using some private key should correctly verify using the corresponding public key. This allows anyone with the public key to verify the authenticity of a signed message. For example, Bob wants to verify Alice's signature using only a public key.

$$\text{Verify}_{pk}(\text{message}, \text{sign}_{sk}(\text{message})) = \text{true} \tag{15}$$

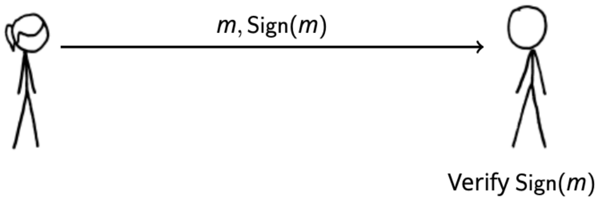


Figure 4: Signature Verification Workflow

The security property we want from a digital signature is that it should not be computationally feasible for an attacker to generate a valid signature for a message $m$ that verifies with the public key without knowing the corresponding private key.

## 5.1 Textbook RSA Signatures

The RSA signature algorithm is a digital signature algorithm that uses the same mathematical properties as RSA encryption. The textbook RSA signature algorithm uses a public key $pk$ and a secret key $sk$, both constructed in the same way as encryption. The secret key $sk$ consists of two primes $p$ and $q$, and a secret exponent $d$ which can be computed with the function $d = e^{-1} \bmod (p-1)(q-1)$. The public key pk contains the modulus $N = pq$, and $e$, the public exponent.

The correctness property for RSA signatures holds for a similar reason that correctness holds for RSA encryption:

$$S^e \bmod N = (m^d)^e \bmod N = m \bmod N$$

An example of RSA signature verification is as follows:

1. Alice sends Bob a message $m$ and a signature $s$. $s = \text{Sign}(m) = m^d \bmod N$.
2. Alice will also send Bob the public key $pk = (N, e)$

3. Bob receives the message and verifies it by computing $m = s^e \bmod N$.

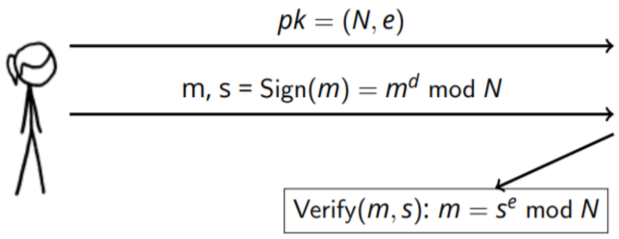

$pk = (N, e)$
m, s = Sign($m$) = $m^d$ mod $N$
Verify($m, s$): $m = s^e$ mod $N$

Figure 5: RSA signature verification

To summarize, if Alice generates her signature using the secret key, Bob would then verify her signature using the public key.

Textbook RSA signatures are very insecure against signature forgery attacks. Here is an example of an attack that takes advantage of the fact that RSA is multiplicatively homomorphic that allows an attacker to forge a valid signature on a message of their choice, if the victim is willing to sign seemingly innocuous messages:

- Attacker wants Sign($x$)
- Attacker computes an integer $z = xy^e \bmod N$ for some $y$
- Attacker asks signer for $S = \text{Sign}(z) = z^d \bmod N$
- Attacker computes $\text{Sign}(x) = sy^{-1} \bmod N$

#### 5.1.1 RSA Signature Padding

The defense against the above class of attacks is to use a padding scheme with RSA signatures, and to sign the cryptographic hash of the message rather than the message itself. The most common implementation choice is the RSA PKCS #1 v1.5 signature padding.

$$\text{pad}(m) = \text{00 01 [FF FF FF ... FF] 00 [data H}(m)]$$

The PKCS signature padding begins with a 00 byte and a 01 byte, followed by some FF bytes and another 00 byte. After that is some metadata identifying the hash algorithm, followed by the hash of the message. The signer hashes and pads the message and then signs the padded message using the RSA private key. The verifier verifies using the RSA public key and strips off the padding to recover the hash of the message, and then verifies that the hash of $m$ matches their own freshly computed hash over $m$.
PKCS #1 v1.5 signature padding is actually provably secure, but an improperly implemented signature verification procedure that does not correctly verify all of the padding can be vulnerable to an interesing signature forgery attack.

#### 5.1.2 Bleichenbacher's RSA Signature Forgery Attack

There are two conditions that make the victim vulnerable to this attack:

- The padding check is done lazily (checks padding format but not length)
- The RSA signature uses a low exponent such as $e = 3$.

When these conditions are met, an attacker can craft a signature for any message that can validate against the target by following these steps:

1. Construct a perfect cube $x$ over the integers, ignoring $N$, such that $x = [\text{pad}(m')]$ [garbage] and $m'$ is the forged message. It is important that $x$ is exactly the length of the RSA key used for signing.
2. Compute $s$ such that $s^3 = x$. An easy way to do this is to set garbage to 0 and take the ceiling of the cube root of $x$, i.e., $s = \lceil x^{\frac{1}{3}} \rceil$.

This attack does not work if $e = 65537$ because for the 2048 or 4096-bit RSA key sizes currently used in practice $m^e > N$. ($e = 65537$ is the most common RSA exponent used in practice.) To protect against this attack, implementations must properly verify PKCS #1v1.5 padding.

## 6 Putting It All Together

Now we will look at how this exchange works in practice and construct a baby version of a cryptographic protocol handshake. Let's imagine a scenario where two parties, Alice and Bob, want to communicate in the presence of a man-in-the-middle attacker who might modify messages in transit.

- They begin with a Diffie-Hellman key exchange. Alice sends $g^a \bmod p$ to Bob, and Bob sends $g^b\ mod\ p$ to Alice.
- Bob sends the signature $s = Sign(g^a \bmod p, g^b \bmod p)$ to Alice so that Alice can verify that neither of the Diffie-Hellman key exchange values were modified in transit.
- Assuming Alice has already retrieved Bob's public key, she can validate the signature and ensure she is talking to Bob and not the man-in-the-middle.
- Bob and Alice independently derive the symmetric key $k = g^{ab} \bmod p$
- They can now share symmetrically encrypted messages using the key $k$

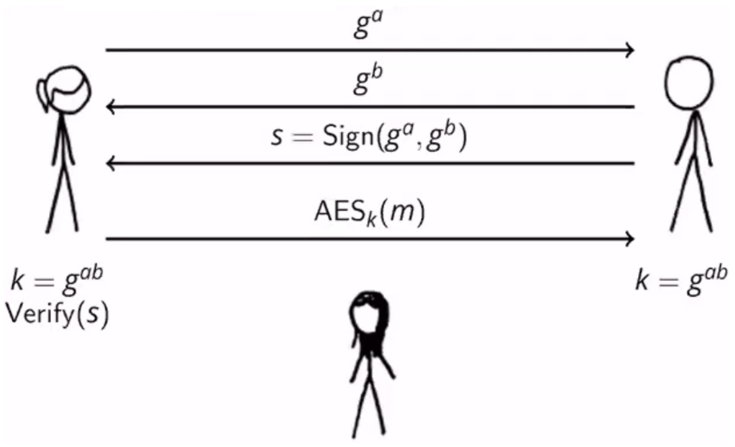


Figure 6: Public-key Cryptography in Practice

### 6.1 Public-key Cryptography and Quantum Computers

Currently, all public-key cryptography used in the real world relies on the use of the following computationally hard problems:

- Factoring
- Discrete log mod primes
- Elliptic curve discrete log

These can all be solved efficiently with quantum computers, meaning more secure replacements will be needed in the coming years as quantum computing technology improves. To combat this, there are some new replacement key exchange and digital signature algorithms under development that are based off of different types of mathematical assumptions. A few examples of these new algorithms include:

- Lattice-based cryptography
- Multivariate crypotgraphy
- Hash-based signatures
- Supersingular isogeny Diffie-Hellman

## 7 References

[1] Nadia Heninger (2021) Public-Key Cryptography pp.1-41 UCSD