CS 198 Codebreaking at Cal
Spring 2023 Course Notes

Note 5

Introduction to Asymmetric Encryption

Recall from before that asymmetric encryption involves two parties securely communicating despite not having a shared key. This supposedly magical concept is made possible by **hard problems in mathematics**, specifically **trapdoor functions.**

Definition 1: Trapdoor Function

A **trapdoor function** is defined as some function $F_K(x)$ with has some "key" K. $F_K(x)$ is efficiently computable for anyone, even for those that do not know K. However, its inverse $F_K^{-1}(x)$ is **computationally hard** to compute if one does not know K, and easy to compute if one does know K.

Let's consider the use case of asymmetric encryption before we jump into the details. Say you wanted to talk with someone on the other side of the world without being eavesdropped. In this situation, you aren't easily able to trade symmetric keys. However, if you know their **public key**, which is an identifier akin to an address, you can "mail" them a message using said public key. Only the owner of said public key will be able to decrypt it, because they have the **private key** corresponding to it.

RSA

The most ubiquitous form of asymmetric encryption today and throughout history is **RSA**, named after its creators Rivest, Shamir, and Adleman. The workings of RSA rely on modular arithmetic, also known as the **field** of integers modulo some other integer.

1. Alice picks two **prime numbers** p, q (where $p \neq q$) and sets $N = pq$. These two primes **must be kept secret**!
2. Alice picks the public exponent e as some integer coprime to $(p-1)(q-1)$. 65537 is a common choice, as is 3 (but not anymore for reasons we will see shortly).
3. Alice finds the private key d as $d = e^{-1} \mod (p-1)(q-1)$, where e^{-1} is the modular inverse of $e \mod (p-1)(q-1)$.
4. Alice publishes (e, N) as the public key and keeps d as the private key.

To encrypt, Bob does the following:

1. Bob encodes his plaintext message m as some integer $\mod N$.
2. Bob finds $E(x) = m^e \mod N$ using efficient modular exponentiation and sends this to Alice.

To decrypt, Alice finds $(E(x))^d = x \mod N$.

Why does this work out? We can prove the correctness of RSA using Euler's Theorem!

Theorem 1: Euler's Theorem

$a^{\varphi(n)} \equiv 1 \mod n$ if $\gcd(a,n) = 1$, where φ is Euler's totient function (represents number of positive integers coprime to and less than n).

We note that $\varphi(N) = (p-1)(q-1)$, since $\varphi(pq) = \varphi(p)\varphi(q) = (p-1)(q-1)$. Therefore, $x^{(p-1)(q-1)} \equiv 1 \mod N$ for all x.

First, let's revisit d. We define d as the inverse of e, so by definition $ed = 1 \mod (p-1)(q-1)$. As a result,

$$\begin{aligned}
&(x^e)^d \mod N \\
&\equiv x^{ed} \mod N \\
&\equiv x^{1+k(p-1)(q-1)} \mod N \\
&\equiv x(x^{(p-1)(q-1)})^k \mod N \\
&\equiv x \mod N
\end{aligned}$$

as desired. We now see why p, q must be secret: if an attacker learns either, they will be able to find d for themselves and decrypt normally. The problem of finding the prime factors of a large integer is known as the **integer factorization problem**, and it is believed to be considerably hard for large integers. More on this next week.

Remark 1: Malleability of RSA

If an attacker is able to modify the ciphertext in-transit, they will be able to predictably modify the underyling plaintext. This property is known as **malleability**, and is *sometimes* desirable (lookup homomorphic encryption) and sometimes not (when we don't want people tampering with our messages).

Consider the situation in which you known Bob is sending Alice a bill for dinner as a plain number. You can deterministically modify this to be 2 times as larger, or whatever factor you want!

To do so, simply multiply the ciphertext by k^e in order to multiply the plaintext by a factor of k. This accomplishes the following:

$$m^e \cdot k^e \mod N \tag{1}$$
$$(km)^e \mod N \tag{2}$$

and therefore Alice will decrypt km as the message!

RSA Broadcast Attack

Recall from earlier that we stated $e = 3$ is a technically valid but often poor choice. The reason for this is the broadcast attack, which makes use of the Chinese Remainder Theorem.

Definition 2: Chinese Remainder Theorem

The **Chinese Remainder Theorem** (aka CRT) states that for coprime $n_1, n_2, \cdots n_k$, the following system of modular equations has a unique solution for $x \pmod{n_1 \cdot n_2 \cdot \cdots n_k}$.

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

such that $x \equiv b \pmod{\prod_{i=1}^{k} n_i}$ for some unique b.

Recall our RSA scheme in which Alice encrypts her message m to an end user by sending $m^e \pmod{N}$ for their N. Now, let's consider what happens when she encrypts the *same message* to multiple people using the same exponent $e = 3$:

$$C_1 = m^3 \pmod{n_1}$$
$$C_2 = m^3 \pmod{n_2}$$
$$C_3 = m^3 \pmod{n_3}$$

Notice that this looks similar to our CRT setup. We then set $x = m^e$ and re-arrange:

$$x = C_1 \pmod{n_1}$$
$$x = C_2 \pmod{n_2}$$
$$x = C_3 \pmod{n_3}$$

By the Chinese Remainder Theorem, we now know the value of $m^e \pmod{n_1 \cdot n_2 \cdot n_3}$. By itself, this doesn't seem like we've made much progress. However, there is a crucial difference between this and something like $x^e \pmod{n_1}$: *x^3 is strictly less than $n_1 \cdot n_2 \cdot n_3$.*

Why does this matter? Well, there is no currently known algorithm for efficiently finding x given $x^e \pmod{n}$, mostly because x^e by itself could be greater than N, which means we cant just take the e-th root. We will expand upon why this is considered difficult next week. In this example, however, it is **guaranteed** that N is greater than x^e.

This can be shown by expanding x^3 to $x \cdot x \cdot x$, and noting that $x < n_1$, $x < n_2$, and $x < n_3$ since x is reduced modulo each. It follows that $x * x * x < n_1 \cdot n_2 \cdot n_3$. Knowing this, it is actually legal to take the cube root and recover x:

$$\sqrt[3]{x} \equiv x \pmod{n_1 \cdot n_2 \cdot n_3}$$

This x is our original message – we have successfully broken RSA in the case of a low exponent and a message repeatedly sent to different people. That might sound like a bit of a far-fetched case, but it is still a serious flaw that is often overlooked in older implementations. If (god forbid) you used RSA to encrypt a password to a secret group chat and sent it to your friends using a low exponent, it would be trivial for an eavesdropper to learn the key themselves.

This attack is most often mitigated by using $e = 65537$, which is often prohibitively high to perform this attack. Unless we had 65537 repeated messages, there would be no easy way to take the e-root of x and be sure it was correct.

Diffie-Hellman Key Exchange

Instead of using RSA to share a symmetric encryption key, there exists a much better scheme named the Diffie-Hellman Key Exchange (it even has key exchange in the name)! This scheme is

similar, but not quite equal to RSA. It uses the properties of modular arithmetic to share a value between two parties while keeping it safe from an eavesdropper.

1. Alice and Bob must first agree on three values: g and N, where g is the generator (base) and N is the modulus. We usually require that $\text{ord}(g)$ is large.
2. Alice and Bob each choose a separate, private number a and $b \mod N$ respectively.
3. Alice computes $g^a \pmod n$ and sends it to Bob.
4. Bob computes $g^b \pmod n$ and sends it to Alice.

Eve now knows the values $g, N, g^a \mod N$, and $g^b \mod N$. Alice and Bob know all these values, plus their own secrets (a and b respectively).

1. Alice computes $(g^b)^a \equiv g^{ab} \mod N$
2. Bob computes $(g^a)^b \equiv g^{ab} \mod N$

Now Alice and Bob have a shared secret value $g^{ab} \mod N$! Eve cannot easily find this value, since $g^a \cdot g^b \equiv g^{a+b} \mod N$, not $g^{ab} \mod N$. We will discuss some more security properties of Diffie-Hellman next week, but the crux of this system is the **discrete logarithm problem**.

Definition 3: Discrete Logarithm Problem

The **discrete logarithm problem** is the problem of finding x in the equation $g^x \pmod N$, given g and N. It is considered computationally intractable for large values of N.

Were Eve able to compute the discrete log, she could find a or b and construct $g^{ab} \pmod n$ herself.

The current record for discrete logarithm is of a 1024-bit prime (very large number), so it is not impossible to break, just very expensive. The authors of Logjam estimated that a discrete log over a 1024-bit prime could cost roughly $100MM, which is certainly feasible for nation-state actors. This is why most systems use at least 2048-bit, and sometimes higher. There exist various methods of "cherrypicking" easy/hard primes for discrete logarithm as well, which we will see next week.

Diffie-Hellman Assumptions

On top of the security of the discrete logarithm problem, Diffie-Hellman relies on the **decision Diffie-Hellman assumption**.

Definition 4: Decisional Diffie-Hellman Problem

The **decisional Diffie-Hellman problem** is the problem of distinguishing between $g^{ab} \mod n$ and some random element $\mod n$, given $g^a \mod n$ and $g^b \mod n$.

This in effect says that the final shared secret should not be able to be guessed at a higher probability than random + some negligible value.

Diffie-Hellman also indirectly relies on the **Diffie-Hellman problem**, which is the problem of finding g^{ab} knowing only g^a, g^b (all mod p).