Lecture 5: RSA Cryptosystem

Lecture 5

Objectives

By the end of this lecture you should be able to understand

1. Public Key Cryptography
2. The elements of RSA cryptosystem
3. Basic attacks for RSA cryptosystem

Note: This lecture is adapted from Coursera Number Theory and Cryptography course [1], Computer and Network Security Course [2], and Burton's textbook

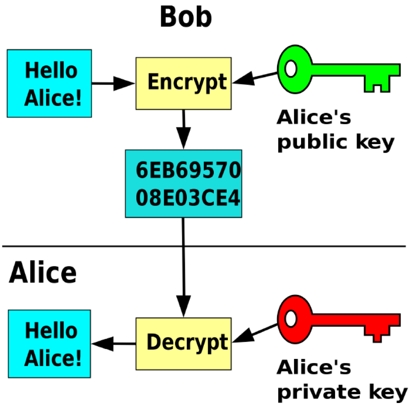
[1] https://www.coursera.org/learn/number-theory-cryptography/home/welcome
[2] https://engineering.purdue.edu/kak/compsec/NewLectures/

Outline

Public Key Cryptography

- Encryption and decryption are carried out using two different keys: Public key and Private key
- Public-key cryptography is also known as asymmetric-key
- This is different from the symmetric key cryptography where the encryption and decryption keys are the same, and of course, they are both private
- All members interested in secure communication publish their public keys
  - SSH protocol: each server publish on its port 22 the public key stored for your login id on the server.
- This solves the problem of key distribution associated with symmetric-key cryptography.

Asymmetric Encryption



Asymmetric Encryption Protocol

- Bob generates two random keys: public key E and private key D
- Bob publishes E for anyone to access
- Anyone can encrypt/cipher message for Bob using E
- Only Bob can decrypt/decipher an encrypted message using D
- The encryption algorithm is public, so actually anyone can decrypt by trying all possible keys, but with known algorithms, it would take hundreds of years or more

RSA Cryptosystem

- The RSA algorithm is named after Ron Rivest, Adi Shamir, and Leonard Adleman.
- RSA is the most commonly used asymmetric cryptography algorithm at present.
- The starting point of RSA cryptosystem is Euler's theorem

Theorem

If $n \geq 1$ *and* $\gcd(a, n) = 1$, *then* $a^{\phi(n)} \equiv 1 \pmod{n}$

- i.e., when a and n are coprimes, the exponents will behave modulo the totient $\phi(n)$:

$$a^k \equiv a^{k \bmod \phi(n)} \pmod{n}$$

Basic Idea of RSA

- Assume modular arithmetic in modulo n
- Assume $m < n$ is an integer representation of the message, where m and n are coprimes: $\gcd(m, n) = 1$
- Assume e is the enciphering exponent such that $\gcd(e, \phi(n)) = 1$
- **Encryption**: transform m to ciphertext c:

$$c \equiv m^e \pmod{n}$$

- **Decryption**: transform c to m using a deciphering exponent d:

$$c^d \equiv m^{ed} \equiv m^{1+k\phi(n)} \equiv m \pmod{n}$$

- **Recovery condition**: d is the multiplicative inverse of e in modulo $\phi(n)$:

$$ed \equiv 1 \pmod{\phi(n)} \quad \text{or} \quad ed - k\phi(n) = 1$$

RSA Protocol

- Generate two big random primes p and q and compute $n = pq$
- Generate random enciphering exponent e coprime with $\phi(n)$, where

$$\phi(n) = pq(1 - 1/p)(1 - 1/q) = (p - 1)(q - 1)$$

- Public Key E is the pair (n, e) – known by everyone
- Private Key D is the pair (p, q) – only known by trusted members
- Knowing the pair (p, q), compute the deciphering exponent d using extended Euclid algorithm to solve:

$$ed \equiv 1 \pmod{\phi(n)} \quad \text{or} \quad ed + k\phi(n) = 1$$

- Pre-compute d right after generating (p, q, e)
- Encrypt and decrypt using fast modular exponentiation.

Why RSA is Secure?

- n is publicly known, but its factorization is secret!
- RSA relies on the difficulty of factorization of n in short time
- Why do we need the factors p and q? To compute $\phi(n)$
- If someone invent a fast factorization algorithm, RSA will immediately become insecure

How to select primes p and q?

- Choose the size B (in bits) of the modulus integer n so that
  - B is big enough to make the algorithm secure
  - B is big enough to make the message $m < n$
  - Typically, B is around 200 digits each so that n would have around 400 digits
- Generate random prime integers p and q
  - Use an RNG to gerenate random number of size $B/2$
  - Set the LSB to 1 to make the number odd
  - Set the highest 2 bits to 1 to make sure the number is big enough
  - Use a primality test to check if it is prime (e.g., Miller-Rabin)
  - if not, increment the integer by 2 and repeat
- If $p = q$, throw away one of them and repeat

How to choose the public exponent e?

- Recall: Recovery condition is to have $\gcd(e, \phi(n)) = 1$ to have a multiplicative inverse in modulo $\phi(n)$.
- Since $\phi(n) = (p-1)(q-1)$, the condition is equivalent to

$$\gcd(e, p-1) = 1 = \gcd(e, q-1)$$

- For computational efficiency, choose e to be prime and has few 1's to make the modular exponentiation fast.
- Typical values of e are 17 and 65537

A Toy Example

- The public key is $n = 2701$, $e = 47$
- The private key is $p = 37$, $q = 73$, then $\phi(n) = 36 \times 72 = 2592$
- $e = 47$ is a valid enciphering exponent since $\gcd(47, 2592) = 1$
- The message to be encrypted is: *NO WAY TODAY*
- Translate it into an integer: $m = 131426220024261914030024$
- Split it into four-digit blocks: 1314 2622 0024 2619 1403 0024
- Ciphered text is obtained by $c_i \equiv m_i^{47} \pmod{2701}$
- Deciphering exponent is obtained by Extended Euclid Algorithm

$$47 \times 1103 + 2592 \times -20 = 1$$

- Deciphered text is obtained by $m_i = c_i^{1103} \pmod{2701}$

Breaking The RSA Cryptosystem

- There have been many trials to break RSA for decades
- A reliable algorithm has many details to make it robust to attacks
- Missing these details might lead to a breakable cipher

Simple Attacks – Finite Set of Messages

- Assume a scenario where your message belongs to a finite set or even a yes/no binary message
- For example: $m = 1$ means “Attack" and $m = 0$ means “Don’t Attack"
- Then, you encrypt m with RSA to get a ciphertext c
- Remember: Every one has the public key!
- An attacker can encrypt $m = 0$ and $m = 1$ messages to find their equivalent ciphertexts
- This applies for any small set of messages!

Defense for Finite Set of Messages

- Solution: Use randomness!
- For example, for a 256-bit block message:
  - use the first 128 for the real message
  - use the last 128 bits for random meaningless message
- Receiver will simply ignore the last 128 bits
- Attacker will have to search in a larger space of more than 2^{128} possible messages

Simple Attacks – Small Prime Factor p or q

- What if p or q is less than $1,000,000$?
- Number of prime numbers less than $1,000,000$ is not large!
- An attacker can simply do an exhaustive search for this small factor, then get the other one
- One typical solution: generate random primes for the secret key uniformly among very large, 2048-bit numbers

Small Difference $|p - q|$

- What is the difference $|p - q|$ is small?
- Assume $q > p$, since $n = pq$, therefore

$$p < \sqrt{n} < q$$
$$0 < \sqrt{n} - p < q - p = r$$

- Therefore, $\sqrt{n} - r < p < \sqrt{n}$
- Try all integers between $\sqrt{n} - r$ and $\sqrt{n}$ to get p
- Even faster, we can write n as

$$n = pq = \left(\frac{p+q}{2} + \frac{p-q}{2}\right)\left(\frac{p+q}{2} - \frac{p-q}{2}\right) = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2$$

- Keep adding squares of integers to n and each time check if the result is exact square of an integer or not. Then stop when you get an exact square of an integer. This square is $\left(\frac{p+q}{2}\right)^2$. Then we have 2 equations in 2 unknowns p and q: Their sum and product.

Defense for Small Difference $|p - q|$

- Generate p and q
- if $|p - q|$ is small, regenerate
- Repeat until $|p - q|$ is large enough

Insufficient Randomness

- For generating p and q, what if RNG seed isn't random enough?
- OpenSSL RSA key generation: keys are generated by the router immediately after startup, no incoming network packets to get randomness from yet.

```
rng = RandomNumberGenerator()
rng.seed(12345) # Same RNG seed!
p = rng.bigRandomPrime()
rng.addRandomness(bits)
q = rng.bigRandomPrime()
n = p×q
```

- Problem: Same p can be generated for different q's on different devices! This is dangerous, why?

Combine Public Keys

- If public keys n_1 and n_2 are generated using the same p, but different q.
- Then, $\gcd(n_1, n_2) = p$ and we can use extended Euclid algorithm to get this common p by solving the diophantine

$$n_1 x + n_2 y = p$$

- This company might have the same issue for many devices: can compromise more keys $n_1, n_2, \ldots$ with common p.
- Experiment resulted in 0.4% factored HTTPS keys!
- Solution: Make sure the RNG is properly seeded
- Some computer programs ask the user to move mouse for some time to get randomness for the RNG seed

Hastad's Broadcast Attack I

- What if the sender broadcast the same message m to several receivers?
- same message m is sent using different public keys
- Assume $e_1 = e_2 = e_3 = 3$ as a simple case

$$c_1 \equiv m^3 \pmod{n_1}$$
$$c_2 \equiv m^3 \pmod{n_2}$$
$$c_3 \equiv m^3 \pmod{n_3}$$

- As discussed before, $\gcd(n_i, n_j) = 1$ for $i \neq j$
- Attacker can use CRT to get c such that $0 \leq c < n_1 n_2 n_3$ and

$$c \equiv c_i \pmod{n_i} \; \forall i \in \{1, 2, 3\}$$

Hastad's Broadcast Attack II

- We can use CRT to solve

$$c \equiv c_1 \pmod{n_1}$$
$$c \equiv c_2 \pmod{n_2}$$
$$c \equiv c_3 \pmod{n_3}$$

- where according to CRT, we have

$$c \equiv m^3 \pmod{n_1 n_2 n_3}$$

- So, $c = m^3$ and the attacker can decipher m as $m = \sqrt[3]{c}$
- This cubic root can be easily done in FP32 arithmetic followed by rounding to integer numbers.
- Solution: add random padding to m before encryption