RSA in Real World

Name 何泽安 (He Zean)

SID # 12011323

CS201 DISCRETE MATHEMATICS

2021 FALL

FINAL PROJECT

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPT. OF COMPUTER SCIENCE AND TECHNOLOGY

Contents

1 Introduction

For centuries, the method of encrypting information has invariably been using a traditional cryptosystem, which is a single-cipher key system that requires group members participating in the encrypted communication to either obtain the key in advance, or to find a way to distribute the key through a secure channel. However, as more and more information is transmitted over the Internet, this traditional way may not provide a good trade-off between security and the cost of protecting the key, therefore, the need to encrypt and protect information becomes a stimulus to develop a new type of cryptosystem. While official messages can be distributed among users using secure channels through dedicated messengers with keys, making up the larger majority that ordinary users who need to achieve secure interactions in insecure networks. A technique that completely separates encryption from decryption, a.k.a. *Public Key Cryptography (PKC)*, has emerged as the best solution to this problem nowadays [1].

The RSA algorithm is considered as the most influential PKC algorithm currently, which is resistant to all known cryptographic attacks to date, and has been recommended by ISO as the standard for public key data encryption. In addition, by using the RSA algorithm, one can attach a *digital signature* to the end of the message, which can be easily verified by anyone, but cannot be forged [2]. In this project, we will focus on the application of RSA algorithm which can always successfully help people to peotect their imformation in the Internet, a simple attempt to attack RSA is also included.

2 Theorem

2.1 Number Theory: Fundamentals

Throughout the lecture in which we learnt about RSA, the professor emphasized the difficulty of factoring large integers. Actually, the reliability of RSA algorithm just depends on this fact, in other words, the more difficult it is to factor a very large integer, the more reliable the RSA algorithm is. If someone were to find a fast factorization algorithm, the reliability of RSA would be extremely reduced, but it is very unlikely that such an algorithm would be found.

Though it is easy to check whether a number is a composite number (basically, the $O(\sqrt{n})$ way is quick enough), according to CLRS [2], up to the date of 2009, it was still not feasible to factorize any 1024-bit number using the best supercomputers and the best algorithms available at the time. It is commonly considered that the 1024-bit RSA key is *basically safe*, and the 2048-bit RSA key is *extremely safe*.

2.2 Attack

When the RSA algorithm uses the well selected p and q as mentioned in the next section, attacking it is really a hard problem. However, we can attempt to attack some "weak" RSAs which may not obey the ISO standard, for example, when the $n = pq$ is too small, or $|p - q|$ is too large or small.

2.2.1 Brute Force

Our first step is to factorize n. ① Applying Miller–Rabin primality test [3] and quick power, in each term we need to check whether a number is a prime, we use $O(k \log^3 n)$, and even if we check the sumterm (say, p, q), we can get an $O(n \cdot k^2 \log^{1.5} n)$, still acceptable for a "rather samll" n; ② when n is less than 768-bit long (all the possible combinations within this range have been calculated and are saved into FactorDB [4]), we can simply check the database and thus get p and q.

```
int64_t quickPow(int a, int b, int r) {
    int64_t ans = 1, buff = a;
    while (b) {
        if (b & 1) ans = (ans * buff) % r;
        buff = (buff * buff) % r;
        b >>= 1;
    }
    return ans;
}

bool millerRabbin(int n, int a) {
    int r = 0, s = n - 1;
    if (n % a == 0) return false;
```

```
    while (!(s & 1)) {
        s >>= 1;
        r++;
    }
    int64_t k = quickPow(a, s, n);
    if (k == 1) return true;
    for (int i = 0; i < r; i++, k = k * k % n)
        if (k == n - 1) return true;
    return false;
}

bool isPrime(int n) {
    int tester[] = {2, 3, 5, 7};
    if (n == 1) return false;
    for (int i = 0; i < 4; i++) {
        if (n == tester[i]) return true;
        else if (!millerRabbin(n, tester[i])) return false;
    }
    return true;
}
```

Trivally, in the next step, with $d \cdot e \equiv 1(\mod \phi(n))$ (here $\phi(n) = p \cdot q$), we can use the Extended Euclidean Algorithm to find the modular inverse of e, a.k.a. d, which only costs $O(\log(\min(p, q)))$.

```
void exgcd(int a, int b, int &g, int &x, int &y) {
    if (!b) g = a, x = 1, y = 0;
    else exgcd(b, a % b, g, y, x), y -= x * (a / b);
}

inline int modinv(int num, int64_t mod) {
    int g, x, y;
    exgcd(num, mod, g, x, y);
    return ((x % mod) + mod) % mod;
}
```

2.2.2 Low Cryptographic Index Attack

Suppose that when e is extreme small (certainly this does not obey the RSA standard), attacker should manully attempt the case of $e = 1$ and $e = 2$, also $e = 3$ can be tried: ① if the original message is not large enough, a.k.a. $m^3 < n$, we can straightly get $m = \sqrt[3]{n}$; ② if $\sqrt[3]{n}$

makes no sense, one can try several small integer k, s.t. $\sqrt[3]{C \pm k \cdot n}$ can be identified as the plaintext.

Another similar case is *broadcast attack*[5]. When the chosen encryption index is low and the same message is sent to a group of receivers using the same encryption index, then a broadcast attack can be performed to get the plaintext. In this case, we get different n (modulo) and c (ciphertext) for different messages, but they share the same m (plaintext) and e (encryption index). In this case, the CRT could be applied to solve the only solution. Suppose we got k groups of m and e:

$$\begin{cases} C_1 \equiv m^e (\text{mod } n_1) \\ C_2 \equiv m^e (\text{mod } n_2) \\ \cdots \\ C_k \equiv m^e (\text{mod } n_k) \end{cases}$$

Since $n_1 \cdots n_k$ are pairwise prime, the congruence equation group has the only solution:

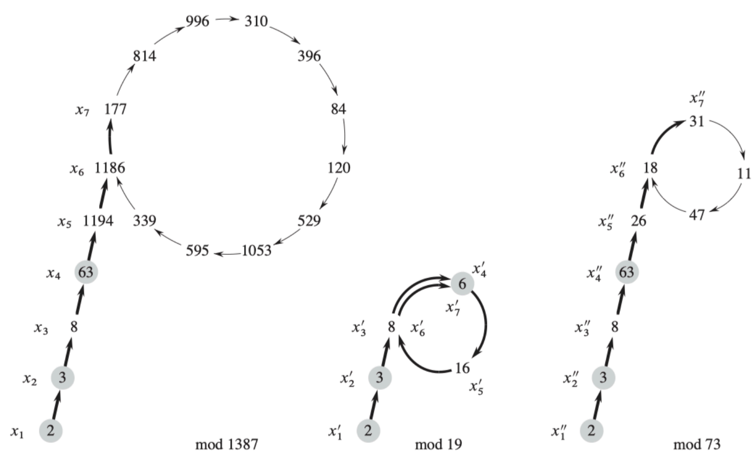
$$m^e \equiv \Sigma_{i=1}^{k} C_i M_i y_i (\text{mod } M)$$

where $M = \Pi_{j=1}^{k} n_j$, $M_i = \frac{M}{n_i}$ and $M_i y_i \equiv 1(\text{mod } n_i)$. Then we can almost get the m^e, only small integer k need to be applied that $m^e = \Sigma_{i=1}^{k} C_i M_i y_i + kM$.

A similar case is *low decryption index attack*, where e is large and n seems difficult to be factorized, formally we say $d < \frac{1}{3} n^{1/4}$ and $q < p < 2q$. Wiener's attack[6] can find the solution in polynomial time.

2.2.3 Pollard's rho heuristic

When the *improper* paragrams p and q are selected, say, $|p - q|$ is either too large or too small, we can apply *the Format method* or *the Pollard rho method* to decompose n quickly. The Pollard's rho heuristic is an effective algorithm to factorize large numbers, consider an integer R, the traditional way of trial division by all integers up to R is able to factorize any number less than R^2, while Pollard's rho heuristic can (probably) find the solution for integers up to R^4 (for our target n, it seems Pollard's rho takes $O(n^{1/4})$, however, considering the input length, a.k.a. $\ell = \log_2 n$, we say it takes $O(2^{\ell/4})$, which is exponent), using the same time complexity [2,7]. This algorithm first randomly choose a small integer x_1 as a seed, then run a function (s.t. $f(x_1) = x_1^2 + a$) to find x_2. If $\gcd(x_1 - x_2, n) \neq 1$, the result is recorded as one factor of n, otherwise, we back the first step and take x_2 as the previous n.



(a) (b) (c)

The **YAFU** project is a practical tool which implements several algorithms to factor large integers. The simple test demonstrates dectypting a RSA-512 encrypted message:

```
-----BEGIN RSA PRIVATE KEY-----
MIIBOQIBAAJAbq4Z0mGxWeeJPaCcWlNNxZV1SJVQ10IxAAnASG7Zk1B38q3X3Uj+
tUYdsU2PcHWJvxu1E0K9+ETHTMqk/Zye1wIDAQABAkBcndG7y8Y13ltot5K+vwDQ
ex3WrCQmzRvl1UMTGbd13n+8ExvmvWokSNbK+wSuhE8oSTkBeALgo4AzuZ+mVY1B
AiEAvNqMqFiVdrMfQWCze0IkFCDEupOpVa4qEyXqxXVwArcCIQCWCDGUc5FvvGqB
O56ydSpDcGBfuL7NKt5iUL0Bffuk4QIhAKK+Q1AfZk2v9lNEneauDKE7y8xsyxQG
zkNJ/ZLDrQ7pAiBnd90hgRYi1fEpkQFgF3d/LOf5+8HyYocdjIrclZLPYQIgDbTD
czwDs36QYNnV/hheqxaHPsYLBjeXW8MfzWa09ZE=
-----END RSA PRIVATE KEY-----

-----BEGIN PUBLIC KEY-----
MFswDQYJKoZIhvcNAQEBBQADSgAwRwJAbq4Z0mGxWeeJPaCcWlNNxZV1SJVQ10Ix
AAnASG7Zk1B38q3X3Uj+tUYdsU2PcHWJvxu1E0K9+ETHTMqk/Zye1wIDAQAB
-----END PUBLIC KEY-----

SUSTech CS201  ==Encrypted==>  MHo0LqFCa3eRlsC+BZvf5ZPle7Nu4ge1Xef/7VnfgewLDO+klKmc6gJfGs6+
    M260lJKpBHrRCyvKvSau7JCrmg==
```

```python
def pollard_rho(self, n, seed=2, p=2, c=1):
    if n & 1 == 0:
        return 2
    if n % 3 == 0:
        return 3
```

```
    if n % 5 == 0:
        return 5
    if is_prime(n):
        return n
    f = lambda x: x ** p + c
    x, y = seed, seed
    while True:
        x = f(x) % n
        y = f(f(y)) % n
        d = gcd((x - y), n)
        if d > 1:
            return d
```

The code is borrowed from RsaCtfTool [8], the slow version uses the above python code, and the fast version uses YAFU.

2.3 Standards

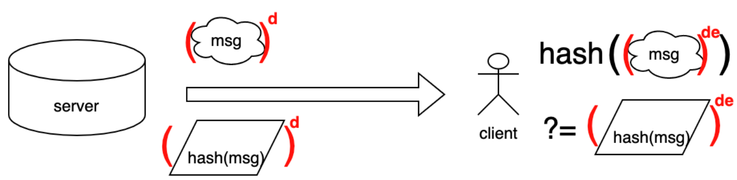
As the several possible attacks we discussed above, we learn that the RSA algorithm highly depends on a pair of properly selected p and q. They should not only long enough, but also different properly in their values. The *PKCS #1* (Public-Key Cryptography Standards) defined in RFC 3447 [9] specified the mathematical definitions and properties that RSA public and private keys must have, which should basically avoid the cases discussed above that may make attacking it easier.

Basically, a standard RSA key should include at least there features: ① the n id at a length of $2^{8(k-1)} \leq n < 2^{8k}$, where the length k of the modulus must be at least 12 octets to accommodate the block formats in PKCS #1; ② multi-prime keys are allowed since version 2.1, which lets $n = \Pi_{i=1} \geq 2ri$ thus making n harder to be cracked.

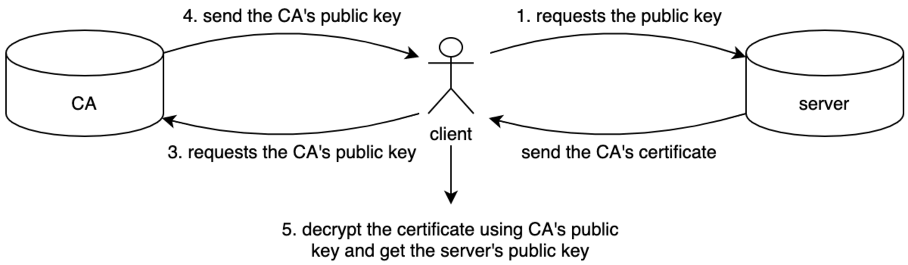
3 Application: HTTPS

When it comes to the Internet, it is natural to mention the *HTTPS* protocol. The HTTP protocol uses plaintext, which means that when a packet is transmitted using the HTTP protocol and is unfortunally cut off in the middle, the data inside will be completely exposed. Therefore, if the packet contains the user's account and password, it could be assumed that the user's account number and password have been leaked. The HTTPS protocol uses SSL to encrypt the data, so even if the data is intercepted, the user's data cannot be known without the decryption (private) key.

When clients are communicating with the server, each client uses the same public key of the server, which should make only the server that can decrypt the information. However, when the server sending data to the client, it is also necessary to let the client know the data it received keeps unchanged from the server, in this case, we should introduce the digital signatures. In short, the digital signature is to form a digest of the sent data using a hash function and then encrypt it with the server's private key; when the client gets the data, it apply the same hash function to the data it receives and compares it with the digest decrypted with the public key, if the two results have no difference, it means that the data transmitted in the middle is not tampered.

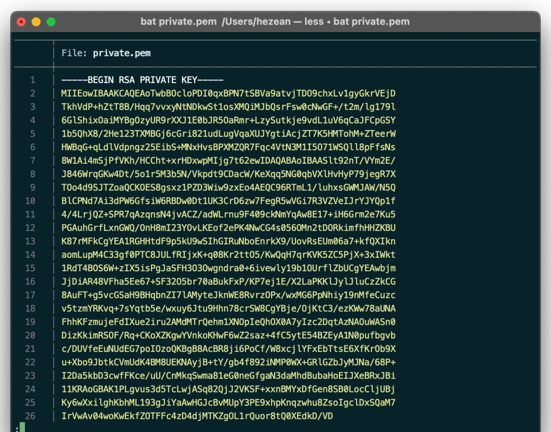


Still there exists one quistion: how to make sure that we are communicating with the "real" server (that is, to prevent an attacker from intercepting a client's request at the proxy server level, redirecting it to his own server, and giving the client the public key of that server, so that the client mistakenly thinks he is interacting with the correct server)? For this problem, the HTTPS protocol built a concept: *authoritative third-party authority (CA)*. When a server sends its public key and some private information to the CA, the CA encrypts these data with its own private key, this is called the digital certificate (*SSL certificate*). When the server sends data to a client, it also sends the certificate downloaded from CA to the local, when the client gets the certificate, again it decrypts the certificate with CA's public key to get the real server's public key. The client-server interaction process under the https protocol can be summarized as follows:

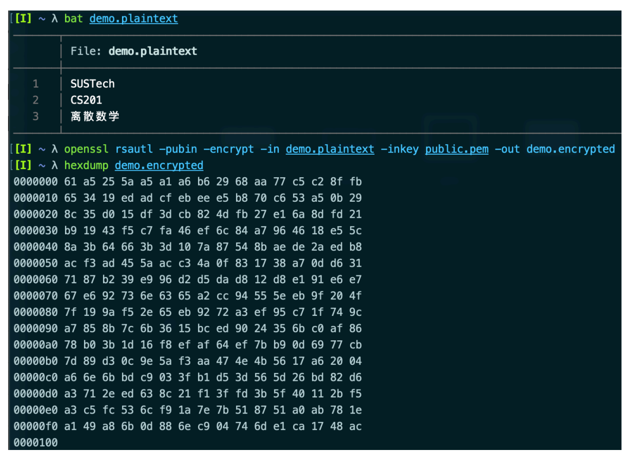


As for the SSL part, another famous tool is OpenSSL, which of course supports several operations about RSA.

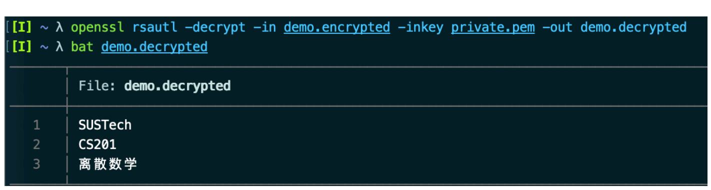
(a) Using OpenSSL to generate RSA public and private keys



(b) Private key generated in last step



(c) Applying the public key to encrypt the plaintext, then we can view the result in hex form



(d) Use the private key to decrypt the encrypted file

References

[1] A. Konheim, *Computer Security and Cryptography*. Hoboken, NJ, United States: Wiley, 2007.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[3] Wikipedia contributors, "Miller–Rabin primality test," 12 2021. [Online]. Available: https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test

[4] "factordb.com." [Online]. Available: http://factordb.com

[5] N1Key, "RSA 常见攻击," 08 2021. [Online]. Available: http://cn1nja.com/posts/rsa/#font-coloref6c00font

[6] Pablocelayes, "rsa-wiener-attack: A Python implementation of the Wiener attack on RSA public-key encryption scheme," 2016. [Online]. Available: https://github.com/pablocelayes/rsa-wiener-attack

[7] Wikipedia contributors, "Pollard's rho algorithm," 09 2021. [Online]. Available: https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm

[8] Ganapati, "RsaCtfTool: RSA attack tool (mainly for ctf) - retreive private key from weak public key and/or uncipher data," 2021. [Online]. Available: https://github.com/Ganapati/RsaCtfTool

[9] J. Jonsson, B. Kaliski, and RSA Laboratories, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1," 02 2003. [Online]. Available: http://tools.ietf.org/html/rfc3447

Disclaimer: All references used in this project are only for the purpose of guiding the direction of exploration and ensuring correctness, any code appearing in the referenced web pages has not been used directly, unless explicitly stated.