Cryptography in Practice, continued

Sergey Gorbunov

Some parts of this manuscript are summaries of sections from “Cryptography Engineering” by Niels Ferguson, Bruce Schneier, Tadayoshi Kohno.

1 RSA Public-Key Encryption Scheme

Invented in 1978 by Rivest, Shamir and Adleman [4], it is still the most popular public-key encryption algorithm!

- **Keys Generation:**
  1. Randomly choose two large primes p, q. Compute $n = p \cdot q$.
  2. Sample two exponents e, d such that $ed = 1(mod\ \phi(n))$.
  3. Output e as the public key, and d as the secret key.
- **Encryption**(e, m)**:**
  1. To encrypt a message m, compute $c = m^e(mod\ n)$
- **Decryption**(d, c)**:**
  1. To decrypt a ciphertext c, compute $m = c^d(mod\ n)$.

Correctness Note that during decryption, the user computes

$$c^d = (m^e)^d = m^{ed} = m^{k\phi(n)+1} = m^{k\phi(n)} \cdot m^1 = m(mod\ n)$$

as claimed. The last equality holds due to Euler's Theorem, which states that if m and n are coprime positive integers, then

$$m^{\phi(n)} = 1(mod\ n)$$

One great advantage of RSA is that it can be used to sign messages. In particular, the owner of the secret key d can sign a message m by computing $s = m^d(mod\ n)$. The pair (m, s) corresponds to a signature. To verify the signature, anyone can compute $s^e(mod\ n)$ and verify that the result equals to m. The correctness follows similarly:

$$s^e = (m^d)^e = m^{de} = m^{k\phi(n)+1} = m(mod\ n)$$

Problems of "Black-Box" RSA First of all, given two signed messages (m_1, s_1, m_2, s_2), where $s_1 = m_1^d (mod\ n)$ and $s_2 = m_2^d (mod\ n)$, an attacker can simply compute a **valid** signature of a product of the two messages: $s_1 \cdot s_2 = (m_1 \cdot m_2)^d (mod\ n)$.

Next we note that it is always desirable to keep the public key exponent e as small as possible to improve the efficiency of the encryption algorithm. Now, suppose $e = 5$ and we wish to encrypt a message $m < \sqrt[5]{n}$. Then, $m^e = m^5 < n$. Hence, there is no modulus reduction that will take place upon encryption! Therefore, an attacker can just take a fifth root of the ciphertext c to recover the message, which is easy to do since there is modulus reduction.

The underlying problem with using a "black-box" implementation of RSA is that there is connection between mathematical structure and the message. A typical solution in practice is to use a special encoding function. In particular, an encoding function is first applied to a message the result of which is then encrypted. After performing the RSA decryption, the message is recovered by applying a decoding function. There are many standards to encoding functions available for use (for example PKCS #1 $v2.1$ [2])

Also, encryption using RSA algorithm is very inefficient. Instead, we typically choose a key k for a fast (and secure) block cipher and use RSA to encrypt k. Then, we use the block cipher and the key k to encrypt the message m. The ciphertext then consists of two encryptions: one corresponding to the RSA encryption of k, and another corresponding to the block cipher encryption of m.

For signatures, using a simple encoding is however not sufficient. For one, the message m can be arbitrary long. To solve this, and at the same time to break the structure of the message m, we can apply a hash function h to m first (for example, SHA-256). However, SHA-256 hashes arbitrary messages to a fixed output size: 256 bits, which is *much* smaller than the modulo n. Again, this is a problem (think about the attack described for the encryption). Hence, we need to map $h(m)$ into a *large* number. For this, we can use a pseudo-random number generator r to map into a larger fixed length output. By combining these two steps, we break the structure of the message m and map it to a necessary and sufficiently large integer $(mod\ n)$.

Finally, we note some of the latest attacks on RSA cryptosystem. We note that RSA would be broken if we had an efficient factoring algorithm. Although we currently do not have one, there are some major implementation issues that have been observed. In particular, a substantial percentage of RSA keys are generated using common randomness. In this report [1], the authors show that many RSA cryptosystems share a single large factor in common. Given two modulo $n_1 = p * q_1, n_2 = p * q_2$, it is easy to recover p by using a GCD algorithm. In particular, out of 11.4 million RSA moduli, 26965 were found to be vulnerable. This offers *at most* security guarantee of 99.8% and affects RSA itself and PKI described below.

2 Public-Key Infrastructure (PKI)

This infrastructure allows you to recognize and validate which public keys belong to whom. On the very high level it works as follows: A central authority (*Certificate Authority*) generates a pair of public and secret RSA keys. We assume *everyone* knows the public key of the authority. Now, say Alice generates her own pair of public and secret keys. To allow everyone to validate her "identity" on the Internet, she comes to the CA with her public key and asks for a signature. The CA verifies Alice's "identity" and outputs a signature saying: "this public key belongs to Alice". Now, anyone can verify the validity of Alice's public key (using CA's public key) and use it for secure communication with Alice. Bob can do the same to establish his pair of public. This process can be used recursively establishing a *chain of trust*.

Some important considerations:

- Expiration date: no public-key should be valid forever.
- Validation of the Identity. What is an identity?
- Key-Management vs PKI.

3 Storing Secrets

There are various places where we can store secret keys: on disk, mobile device, memory and so on. Storing a secret key on disk is a good solution, but it *assumes* that the computer is kept secure from the attackers. In addition, it becomes inaccessible from remote machines. Alternatively, we could store the secret on a mobile device: it becomes very portable, but we are much more likely to lose it. Finally, storing a secret key in memory is not very infeasible: can you remember at least 256 random bits? A good solution would be to use a **long** sentence as a password which you can easily remember, and *derive* as key from it using a *Password-Based Key Derivation Function* [3]. On the very high level, it applies a hash function multiple types to your password. If you do not want to use a password-based derivation, then write your key on a piece of paper and keep it in your wallet. It is probably more secure than any of your electronic devices.

Secret Sharing can help you both provide reliability and security in keeping your secrets. On the high level, given a secret s, the algorithm generates shares $s_1, \ldots, s_n$ (for any desirable n) such that:

- given all n shares, we can recover the original secret s.
- given *any* subset of less than n shares, the adversary learns nothing about s.

Hence, we could store a share on a remove server. Unless all but n servers collude (pull their shares together), they learn nothing about n. However, assuming we have access to each of the servers, we can recover the secret. To share a bit s_j of the secret s we do the following:

- Choose random bits $r_1, \ldots, r_{n-1}$. Each player $i = 1, \ldots, n-1$ is given a share $c_i = r_i$.
- The last player n, is given $c_n = r_1 \oplus r_2 \oplus \ldots \oplus r_{n-1} \oplus s_j$.

Clearly, given shares $c_1, \ldots, c_n$, $s_j = c_1 \oplus c_2 \oplus \ldots \oplus c_n$. Given any subset of less than n shares, s_j is information theoretically hidden. A much more useful tool is a *threshold secret sharing scheme (TSSS)*: A (t, n)-TSSS satisfies the following properties.

- There is an algorithm $Share_{t,s}(s)$ that takes a threshold value t, number of shares n and a secret s and outputs shares $c_1, \ldots, c_n$.
- There is an algorithm $Reconstruct(c_{i_1}, \ldots, c_{i_t})$ that takes t shares and recovers the secret s.
- No subset of less than t shares reveals anything about about the secret s.

See Shamir's Secret Sharing as an example `http://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing`.

References

[1] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung and Christophe Wachter, *Ron was wrong, Whit is right*, `http://eprint.iacr.org/2012/064.pdf`

[2] *Public-Key Cryptography Standards (PKCS)* #1*: RSA Cryptography, Specifications Version* 2.1 `http://tools.ietf.org/html/rfc3447`

[3] *PKCS* #5*: Password-Based Cryptography Specification Version* 2.0 `http://www.ietf.org/rfc/rfc2898.txt`

[4] Ronald L. Rivest, Adi Shamir and Leonard M. Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, In Commun. ACM, 1978.