Number Theory 2: The Rivers-Shamir-Adleman (RSA) Cryptosystem (draft)

The RSA Cryptosystem

Some cryptography basics. Before we get to the RSA, we need to understand why public-key systems are needed. The general problem is like this: we have two agents, called Alice and Bob, and Alice wants to send some private information to Bob over a public channel. The cryptographers sometimes introduce another party, called Eve (eavesdropper), that listens to the public channel and tries to figure out what Alice and Bob are saying to each other.

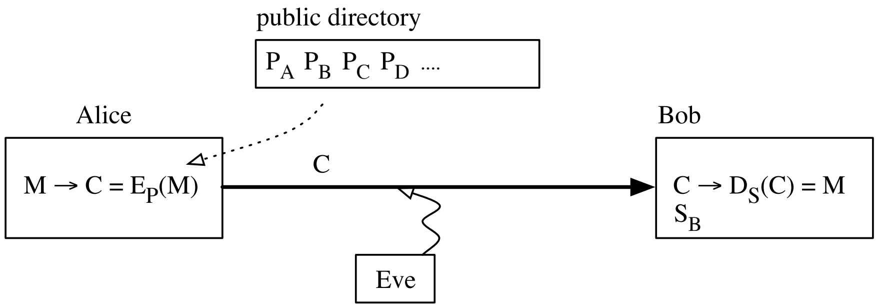
Secret algorithms. One approach is to use a *secret algorithm* system, in which Alice has a secret encryption algorithm E and Bob has a secret decryption algorithm D, such that $D(E(M)) = M$ for each M. Thus, D is the inverse of E, $D = E^{-1}$. This method has two drawbacks. First, it is completely infeasible in a large system like, for example, Internet. How do Alice and Bob agree on a common algorithm? Most parties that want to exchange private messages do not have time, desire, nor sufficient knowledge of cryptography to design their own secret algorithms. Another issue is security. Somewhat paradoxically, a secret-algorithm system may well be less secure than an algorithm with a public algorithm (defined below), because it has not been properly verified.

Public algorithm with secret key. We address these concerns by using a *public algorithm* with *secret-key.* The encryption has two arguments: a message M and a secret key K. For different keys the encrypted message $C = E_K(M)$ will be different. Bob decodes the message using the secret key, $M = D_K(C)$. Of course, as before, we need that for each key K, D_K is the inverse of E_K.

The main advantage of this system is that now the users can use standard, well tested, algorithms. The most commonly used standard is DES (Data Encryption Standard) set in 1977 by the National Bureau of Standards. DES uses 56-bit keys (plus 8 parity bits) for enciphering 64-bit blocks of data. DES consists of a sequence of rather involved transformations that can be very efficiently implemented in hardware. Longer messages are broken into 64-bit blocks. A naive approach, called the ECB mode (Electronic Code-Book), is to encipher each block separately. This method is not safe and is not recommended. Instead, one can use techniques that use feedback to make the encoding of each block depend on all previous blocks. DES is still heavily used, for example by banks (ATMs, inter-bank transactions). Other currently used block ciphers are IDEA (used in PGP) and Blowfish.

One problem still remains though: How do Alice and Bob agree on what key to use? We get into a chicken-and-egg problem here: In order to exchange secret messages over a public channel, they need to agree on a secret key, but this needs to be done over a public channel too! Sounds impossible, huh. As we shall soon see, it can be done. There are protocols for key exchange over a public channel. Another option is to use a public-key system. Read on.

Public-key systems. The trick behind public-key systems is to have *two* keys for each user, one public and one secret. The public key is used for encryption and the secret key for decryption. Alice's public key is P_A, and her secret key is S_A. For brevity, we will write $P_A(M)$ instead of $E_{P_A}(M)$, and $S_A(C)$ instead of $D_{S_A}(C)$. As before, the encryption and decryption algorithms need to be the inverses of each other, that is $S_A(P_A(M)) = M$.



In order for this approach to work, we need to make sure that the secret keys cannot be computed from the public keys. What is exactly meant by "cannot"? After all, one could just try all possible keys and see if one of them works. However, when the keys are large enough, say 100 digits long or so, this brute-force approach is not feasible. The key length (or, in general, the strength of the system) should be such that the (expected) cost of computing the secret key substantially exceeds potential benefits of breaking the system.

The idea of public-key systems was introduced by Diffie and Hellman in 1976. It has several implementations. One, developed by Merkle and Hellman (and patented) was based on the knapsack problem. Their algorithm was subsequently broken by Shamir and others. Other methods include the RSA and ElGamal systems.

The RSA cryptosystem. The RSA (Rivest-Shamir-Adleman) is the most popular implementation of public-key systems currently in use. Although not provably secure, a lot of work has been done trying to break this system, and no efficient method is known.

In RSA, we think of each message M as a number of some fixed length. If the message is longer, it is broken into smaller pieces. We describe the system in three parts: initialization, encryption, and decryption.

Initialization: Bob picks $n = pq$, where p, q are two large distinct primes, say 100 digits long. Then he selects a small integer e relatively prime to $\phi(n) = (p-1)(q-1)$. Let $d = e^{-1} \bmod \phi(n)$, the inverse of e modulo $\phi(n)$. He publishes $P = (e, n)$ as the public key and hides $S = d$, his secret key.

Encryption: Each message is represented by an integer M between 0 and $n-1$. If Alice wants to send M to Bob, she computes $C = E_P(M) = M^e \operatorname{rem} n$, and sends C to Bob.

Decryption: Upon receiving a ciphertext C, Bob computes $D_S(C) = C^d \operatorname{rem} n$. The value of $D_S(C)$ is the decrypted message.

Theorem 1 *RSA is correct.*

Proof: We need to show that $S(P(M)) = M$, that is $M^{ed} = M \pmod{n}$. Recall that $\phi(n) = (p-1)(q-1)$, because $n = pq$. Since $e = d^{-1} \pmod{\phi(n)}$, we have

$$ed = 1 + k(p-1)(q-1),$$

for some k and, therefore, if $M \neq 0 \pmod{p}$, by Fermat's Little Theorem we get:

$$M^{ed} \; = \; M(M^{p-1})^{k(q-1)} \; = \; M \pmod{p}$$

If $M = 0 \pmod{p}$ then $M^{ed} = M \pmod{p}$ as well. Similarly we get $M^{ed} = M \pmod{q}$. Thus $M^{ed} - M$ is a multiple of both p and q. Since p, q are prime, $M^{ed} - M$ must be also a multiple of $pq = n$. This implies that $M^{ed} = M \pmod{n}$, and we are done. □

Example. Suppose that Bob picks $p = 7$ and $q = 11$. Then $n = pq = 77$, and $\phi(n) = (p-1)(q-1) = 60$. Bob picks $e = 13$, and $d = 13^{-1} = 37 \pmod{60}$. Bob's public key is $P_B = (13, 77)$ and his secret key is $S_B = (37, 77)$. The encryption and decryption algorithms are:

$$P_B(M) \; = \; M^{13} \bmod 77 \qquad\qquad S_B(C) \; = \; C^{37} \bmod 77$$

Now, let's suppose that Alice wants to send a message $M = 5$ to Bob. She sends him

$$\begin{aligned} C \; = \; P_B(5) \; &= \; 5^{13} \bmod 77 \\ &= \; 5 \cdot ((5 \cdot 5^2)^2)^2 \bmod 77 \\ &= \; 5 \cdot ((48)^2)^2 \bmod 77 \\ &= \; 5 \cdot (71)^2 \bmod 77 \\ &= \; 5 \cdot 36 \bmod 77 \\ &= \; 26 \end{aligned}$$

Bob receives $C = 26$ and decodes it using his secret key:

$$\begin{aligned} M \; = \; S_B(26) \; &= \; 26^{37} \bmod 77 \\ &= \; 26 \cdot ((26 \cdot ((26^2)^2)^2)^2)^2 \bmod 77 \\ &= \; \cdots \\ &= \; 5 \end{aligned}$$

Security. We need to decide first what exactly we mean by "security", or, what does it mean to "break the system". For our purpose, assume that by breaking the cryptosystem we mean computing the secret key S_A from the public key P_A. One way to do it is by exhaustive search: Given P_A, generate some pair (M, C), where M is a message and $C = P_A(M)$ is a ciphertext, and then try all keys S, and for each compute $M' = S(C)$, until $M' = M$. Then $S_A = S$ for which $S(C) = M$. Of course, this brute-force approach is impractical for large keys.

Ideally, we would like to formally *prove* that our cryptosystem cannot be broken, that is, that there is no polynomial-time algorithm that computes the secret key from the public key. Unfortunately, noone knows how to prove such impossibility results for natural computational problems. The next best thing would be to show the following: if we could break the cryptosystem, then we could also solve some other computational problem which is believed to be hard. One such notoriously hard problem is factorization.

Notice that if you can factor fast, you can break the RSA: Given n, compute its factorization $n = pq$. Then you can compute $\phi(n) = (p-1)(q-1)$, and finally, you compute $d = e^{-1} \bmod \phi(n)$ using Extended Euclid's algorithm, and we're done. This does not prove, however, that breaking the RSA is as hard as factoring, since there could be other ways of computing $\phi(n)$ that do not use factorization, or there could be a way to compute d without computing $\phi(n)$. Nevertheless, in spite of intensive research in this area, no such method has been yet discovered.

Efficiency. Of course, for the cryptosystem to be useful, it's important that all computation in the initialization, encryption and decryption steps be efficient. This is not an algorithm's class, so we will only briefly address this issue.

In the initialization stage, we need to first find large primes. This can be done by simply guessing large numbers and verifying their primality. The Prime Number Theorem implies that we only need to make a small number of guesses, in expectation, and primality can be verified efficiently. We also need to compute an inverse of e modulo $\phi(n)$. This can be done using the Extended Euclid's Algorithm.

The encryption/decryption stages involve computing large powers modulo n. The public exponent e is often small, but the private exponent d could be the same order of magnitude as n, that is 200 digits or more. If we simply multiplied C by itself C times (each time taking the remainder modulo n), this would involve some 10^{200} multiplications, which is hopeless.

But there is a better way, called the squaring algorithm. Suppose you want to calculate $3^{32} \bmod 7$. You can write it as $((((3^2)^2)^2)^2)^2 \bmod 7$, so instead of 31 multiplications, this computation will only require 5 multiplications. If the exponent is not a power of 2, this can be slightly modified. For example, $3^{35} \bmod 7 = 3 \cdot (3 \cdot (((3^2)^2)^2)^2)^2 \bmod 7$.