Why RSA? A Pedagogical Comment

Pedro Barragan Olague[1], Olga Kosheleva[2],
and Vladik Kreinovich[1]
[1]Department of Computer Science
[2]Department of Teacher Education
University of Texas at El Paso
500 W. University
El Paso, TX 79968, USA
pabarraganolague@miners.utep.edu
olgak@utep.edu, vladik@utep.edu

Abstract

One of the most widely used cryptographic algorithms is the RSA algorithm in which a message m encoded as the remainder c of m^e modulo n, where n and e are given numbers – forming a public code. A similar transformation c^d mod n, for an appropriate secret code d, enables us to reconstruct the original message. In this paper, we provide a pedagogical explanation for this algorithm.

1 RSA Algorithm: A Pedagogical Puzzle

RSA algorithm: a brief reminder. In many computer transaction, the communicated message is encoded, to avoid eavesdropping. This happens, e.g., every time a credit card information is passed over to some website. In most of such cases, a special *RSA* algorithm is used to encode the message m; see, e.g., [1].

In this algorithm, two specially selected and publicly available numbers n and e are used to encode the message. The encoded message c has the form of the remainder $c = m^e$ mod n.

The number n is usually at least 100 decimal digits long, and the number e is similarly large. For such large numbers, it is not feasible to compute m^e simply as $m \cdot \ldots \cdot m$, by starting with m and $e - 1$ times multiplying the result by m. Instead, the following much faster algorithm is performed.

First, the number e is represented in the binary form, as the sum of powers of two: $e = 2^{k_1} + 2^{k_2} + \ldots + 2^{k_p}$ for some $k_1 > k_2 > \ldots > k_p$. For example, 11_{10} is represented as

$$1011_2 = 2^3 + 2^1 + 2^0 = 8 + 2 + 1.$$

In general, $m^e = m^{2^{k_1}} \cdot m^{2^{k_2}} \cdot \ldots m^{2^{k_p}}$. For example, for $e = 11_{10}$, we have

$$m^{11} = m^{2^3} \cdot m^{2^1} \cdot m^{2^0} = m^8 \cdot m^2 \cdot m^1.$$

So, to compute m^e, we:

- first compute $m^2 = m \cdot m \bmod m$,
- then compute $m^4 = m^2 \cdot m^2 \bmod n$,
- $\ldots$,
- for each k, we compute $m^{2^{k+1}} = m^{2^k} \cdot m^{2^k} \bmod n$,
- $\ldots$, and,
- finally, we compute $m^{2^{k_1}} = m^{2^{k_1-1}} \cdot m^{2^{k_1-1}} \bmod n$.

After that, we sequentially multiply the powers $m^{2^{k_i}}$ until we get the desired value m^e.

For example, for $n = 35$, $e = 11$, and $m = 3$:

- first, we compute

$$m^2 = m \cdot m = 3 \cdot 3 = 9 = 9 \bmod 35,$$

- then, we compute

$$m^4 = m^2 \cdot m^2 = 9 \cdot 9 = 81 = 11 \bmod 35,$$

- finally, we compute

$$m^8 = m^4 \cdot m^4 =$$

$$11 \cdot 11 = 121 = 16 \bmod 35.$$

After that, we compute

$$c = m^{11} = m^8 \cdot m^2 \cdot m^1 = 16 \cdot 9 \cdot 3 =$$

$$(16 \cdot 9) \cdot 3 = 144 \cdot 3 = 4 \cdot 3 = 12 \bmod 35.$$

Once the encoded signal c is received, it can be decoded as $m = c^d \bmod n$, for an appropriate *secret code* d. For the reconstruction to be possible, we must take d for which $e \cdot d \equiv 1 \bmod \varphi(n)$, where $\varphi(n)$ denotes the number of positive integers which are smaller than n and mutually prime with n. For example, if n is a prime number, then $\varphi(n) = n - 1$. If n is a product of two prime numbers $n = p \cdot q$, then $\varphi(n) = (p - 1) \cdot (q - 1)$, etc. To be able to find such a d, we need to select e which is mutually prime with $\varphi(n)$, i.e., for which the greatest common divisor $\gcd(e, \varphi(n)) = 1$.

In RSA, usually, the value n is selected as a product of two large prime numbers.

In the above example $n = 35 = 5 \cdot 7$, we have $\varphi(35) = (5-1) \cdot (7-1) = 4 \cdot 6 = 24$. Here clearly, $\gcd(e, \varphi(n)) = 1$ and thus, there exists a secret code d for which $e \cdot d \bmod \varphi(n) = 1$: indeed, we can take $d = 13$ (please note that this is a simplified pedagogical example; in real life, $d \neq e$). So, to reconstruct m, we compute $c^d = c^8 \cdot c^2 \cdot c^1$. Here,

$$c^2 = c \cdot c = 12 \cdot 12 = 144 = 4 \bmod 35,$$

$$c^4 = c^2 \cdot c^2 = 4 \cdot 4 = 16 \bmod 35, \text{ and}$$

$$c^8 = c^4 \cdot c^4 = 16 \cdot 16 = 256 = 11 \bmod 35.$$

Thus,

$$c^{11} = c^8 \cdot c^2 \cdot c^1 = 11 \cdot 4 \cdot 12 =$$

$$(11 \cdot 4) \cdot 12 = 44 \cdot 12 = 9 \cdot 12 = 108 = 3 \bmod 35.$$

So, we have indeed reconstructed the original message $m = 3$.

RSA algorithm: a pedagogical puzzle. While the RSA algorithm is very efficient and effective, its origins are usually explained by a stroke of genius, as a clever trick that its authors came up with.

It may have been a stroke of genius, but from the pedagogical viewpoint, it is desirable to also have a natural explanation for this algorithm. Natural explanations help remember and understand the materials better than presenting it as a random-sounding combination of seemingly unrelated difficult-to-explain tricks.

2 RSA Algorithm: A Pedagogical Explanation

What do we want from encoding? We want an encoding algorithm which is difficult to crack. This means, in particular, that the encoded message c should be as far from the original message m as possible.

On the other hand, we want the encoding and decoding to slow down the communication process as little as possible. Thus, both encoding and decoding should be as fast as possible.

Let us show that these two natural requirements lead to RSA encoding.

Need for speed necessitates the absence of branching. In the computer, each algorithm consists of a sequence of elementary arithmetic operations. In some algorithms, this sequence is fixed, but in general, the selection of the next operation may depend on the results of the previous operations – i.e., we have *branching*.

It is known that, in general, branching slows down computations (see, e.g., [2]): without branching, the computer can start preparing for the next arithmetic operation while finishing the previous one, while with branching, we do not know which operation to prepare for until the previous operation is completed.

Since we would like the encoding operations to be as fast as possible, it therefore makes sense to only consider encoding algorithms without branching, for which, for each code, the sequence of arithmetic operations is the same for all the messages.

Which arithmetic operations should we use? To decide which elementary arithmetic operations should be included in the encoding algorithm, let us use the second requirement – that the result of encoding should be as different from the original message as possible.

The best way to achieve that is to make sure that each of the arithmetic operations that forms the desired encoding algorithm changes the previous result as much as possible. Which arithmetic operations have this property? There are four arithmetic operations: addition $a + b$, subtraction $a - b$, multiplication $a \cdot b$, and integer division a/b.

Without losing generality, let us assume that $a \le b$ (messages are natural numbers). Then,

- the difference between $a + b$ and a is $|(a + b) - a| = b \le a$;
- the difference between $a - b$ and a is $|(a - b) - a| = b \le a$;
- the difference between $a \cdot b$ and a is $|a \cdot b - a| = a \cdot (b - 1)$, and
- the difference between a/b and a is $|(a/b) - a| = a - (a/b) < a$.

We see that for addition, subtraction, and division, the difference between the result of the arithmetic operation and one of its arguments always does not exceed a, while for the product, with the exception of the cases $b = 0$, $b = 1$, and $b = 2$, the difference is larger than a.

Thus, to make sure that the result of each operation in our algorithm is as far away from the inputs as possible, it makes sense to require that all these operations are multiplications. Thus, in our algorithm, each operation consists is a multiplication – either the product of two previous result, or the product of a previous result and a constant, or the product of two constants. (By the way, from the computational viewpoint, the third cases does not make much sense: if we want to compute a product of two constants, why not multiply them beforehand instead of multiplying them again and again every time when need to encode a new message.)

This leads to the RSA encoding. Let us show that the above idea indeed leads, in effect, to the RSA encoding. Indeed, in the beginning, we have the original message m; we can view this message as the result r_0 of the 0-th step. On each step k, as r_k, we take either the product of two previous results r_i and r_j, $i, j < k$, or the product of one of the previous results r_i and a constant c_k.

Let us show, by induction over k, that the result r_k of each computational step has the form $a_k \cdot m^{b_k}$ for some values a_k and b_k.

The base statement is trivially true: for $k = 0$, we have $r_0 = m = a_0 \cdot m^{b_0}$, where $a_0 = 1$ and $b_0 = 1$.

Let us now assume that we have proved this statement for all the steps $i, j < k$. Then, if $r_k = r_i \cdot r_j$, then from the already-proven expressions $r_i = a_i \cdot m^{b_i}$ and $r_j = a_j \cdot m^{b_j}$, we conclude that $r_k = r_i \cdot r_j = a_k \cdot m^{b_k}$, where $a_k = a_i \cdot a_j$ and $b_k = b_i + b_j$. Similarly, if $r_k = c_k \cdot r_i$, then, from $r_i = a_i \cdot m^{b_i}$, we get $r_k = a_k \cdot m^{b_k}$, where $a_k = c_k \cdot a_i$ and $b_k = b_i$.

The statement is proven, and thus, the final result of the algorithm – i.e., the encoded message – also has the form $c = a \cdot m^b$, for some a and b.

Of course, in the computers, we do not perform operations with unlimited integers. In effect, we perform all the computations modulo some large number n. Thus, we conclude that $c = a \cdot m^b \bmod n$.

The only difference from the RSA encoding is that we also have a multiplicative factor a – but multiplying by a and dividing by a is easy, so sending an encoded message $a \cdot m^e$ is, in effect, equivalent, to sending an easier-to-compute message m^e.

Thus, we indeed get an explanation for the RSA encoding.

Acknowledgments

This work was supported by the National Science Foundation grants HRD-0734825 and HRD-1242122 (Cyber-ShARE Center of Excellence) and DUE-0926721, and by an award "UTEP and Prudential Actuarial Science Academy and Pipeline Initiative" from Prudential Foundation.

References

[1] Th. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 2009.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, Waltham, Massachusetts, 2012.