Introduction to Cryptography

Build your first own encryption algorithm

Oliver Snellman*

November 26, 2023

Introduction

Encryption is a process that transforms a message into a coded message. The idea is to make it easy for a friend to recover the original message, but very hard for anyone else to do the same. The coded message can be transformed back to the original message only with a private key, a process called decryption.

The great mathematician Alan Turing managed to crack the German Enigma encryption system during the Second World War. Could a modern day genius do the same with contemporary encryption algorithms? Surprisingly no - it is not enough to figure out how the encryption algorithm works. Actually, the functioning of the popular Secure Hash Algorithm (SHA-256), used for example in many blockchain applications, is common knowledge. How can you not, then, deduce the input that went in to the algorithm, based on the observable output and detailed understanding of what the algorithm does?

Reversing the modern encryption algorithms is not possible, because they introduce pseudo-randomness to break the one-to-one mapping between the message and the coded message. Therefore, knowing the rules of the algorithm doesn't help, if the pseudo-randomness can't be negated, which is only possible with something called the "private key".

Let's build a simple encryption system from scratch to gain intuition on how securing messages happens nowadays. The main building blocks of our algorithm are 1) a one-to-one mapping between characters and numbers and 2) pseudo-randomness to break the one-to-one mapping.

Mapping

Consider a function that maps the 26 letters in the English alphabet into the first 26 natural numbers, creating a lookup table. More technically, this function is an injection from the domain (message space) to the range (coded message space).

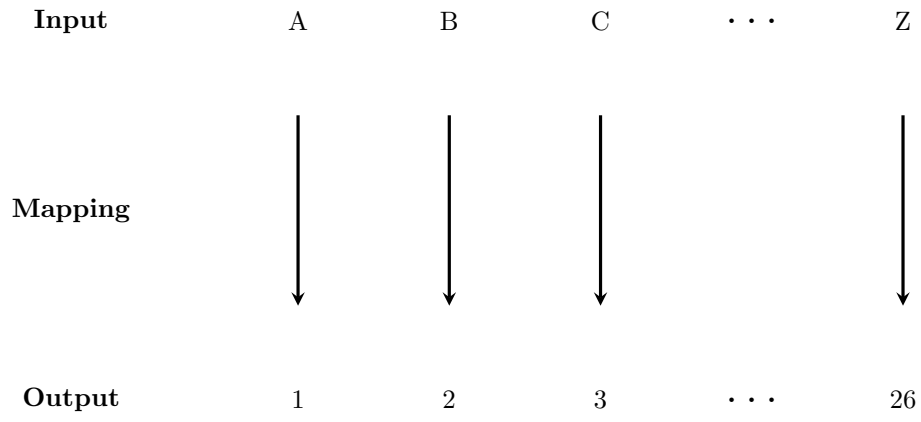

Figure 1: Simple encryption with one-to-one mapping.

*University of Helsinki, Helsinki Graduate School of Economics, Email: oliver.snellman@helsinki.fi

Using this rule, any text can be encrypted into an unreadable sequence of numbers.

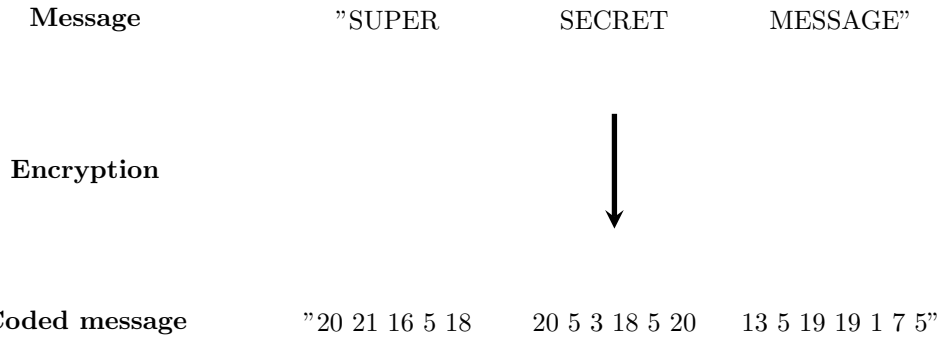

Figure 2: Transforming a sentence into numbers with a one-to-one mapping.

If there are many coded messages encrypted with this scheme, it is possible to observe patterns, use probabilities of how commonly different words incur in language and deduce the logic behind the encryption. Technically this amount to finding the inverse of the injective encryption function. For example, it might be easy to deduce the mapping for short words like "I"="9" or "AND"="1 14 4" and use those to deduce the mapping for rest of the letters. If someone does figure out the lookup table in Figure 1 then it is easy to decrypt all messages which were encrypted using the same method.

This is, in essence, what Alan Turing did. Enigma was naturally much more complex than the example above. The rule by which letters were mapped to symbols also changed every day according to another rule. Turing, with his team, managed to figure out the logic of the Enigma's encryption and how it changed daily. The breakthrough was the phrase that repeated in the end of every message, praising the leader.

This would not be possible with the modern encryption algorithms, because they are not injective one-to-one mappings between the message and coded message.

Adding pseudo-randomness

Modern encryption algorithms break the one-to-one mapping in a way that can be reversed only with the help of a private key. Let's start by looking at two examples of encryption, which break the one-to-one mapping.

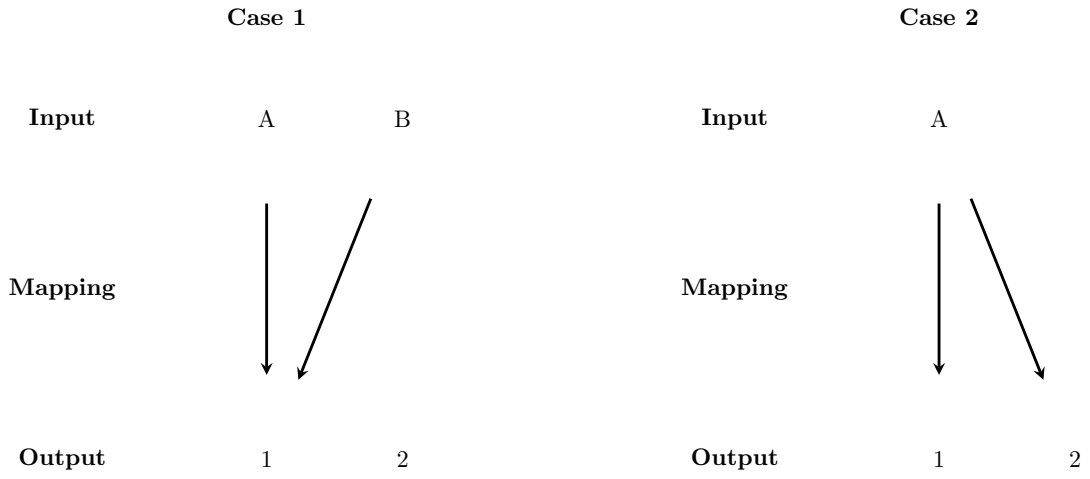

Figure 3: The encryption in Case 1 allows multiple inputs to map into the same output (non-injective). The encryption in Case 2 allows an input to map into different outputs at different times (correspondence). An encryption can allow both to happen at the same time.

One way to break the one-to-one mapping like in the Figure 3 is by introducing pseudo-randomness. **Pseudo-randomness** looks completely random, but the same exact numbers can be replicated with the private key (seed).

Let's say Harry wants to encrypt the word "MOON" and send it to Remus. Consider the same mapping as before in Figure 1, but now we add a second step: To each of the numbers resulting from the mapping we add randomly either -1, 0, or +1, each with 1/3 probability.

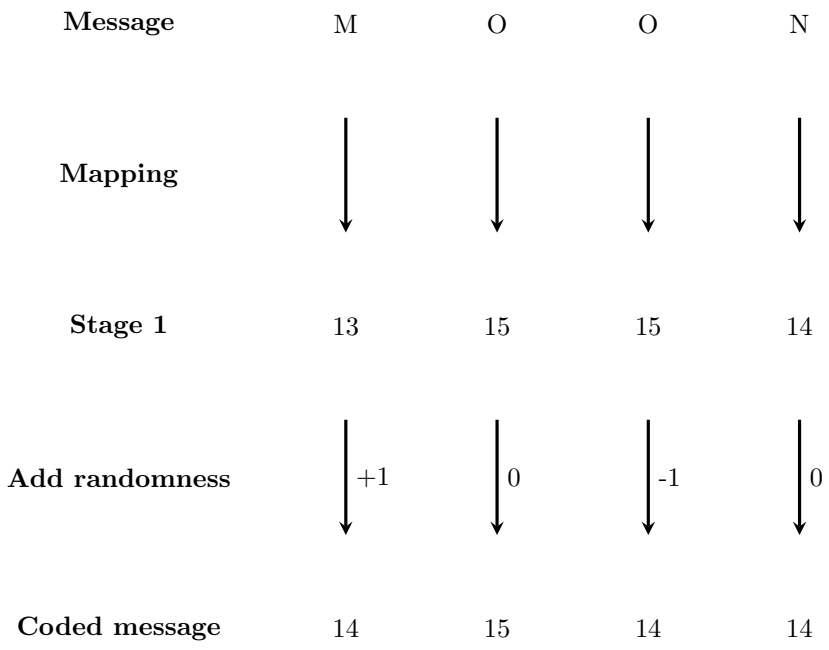

Figure 4: Encryption with pseudo-randomness, controlled with a private Key.

With this encryption algorithm and with the particular random numbers (+1, 0, -1, 0), the message "MOON" transforms to the coded message "14 15 14 14". Now there are three different instances of "14" in the coded message, each corresponding to a different letter in the original message, demonstrating the Case 1 from Figure 3. The letter "O" in the original message also now corresponds to two different symbols, "15" and "14", demonstrating the Case 2.

If we were to augment the original message by writing the word twice "MOON MOON", the second word "MOON" could look different from the first one after the encryption. Therefore, in absence of one-to-one mapping, it is not possible to break the code and construct a lookup table to cipher the future messages.

But how can anyone, even the correct recipient, then, recover the original message? The answer lies in the private key. It is not possible for computers to come up with completely random numbers, but instead they generate the numbers deterministically using some complex function, which is very sensitive to its input values[1]. This function is supplied with some predetermined list of input values, which create the random looking outcomes. By setting a "seed" we specify that a particular list of input values is to be used in the pseudorandom number generation. With the same seed, the same numbers will appear every time. Hence, the random numbers only appear to be random, but they can be recreated with the seed. The seed is the private key in this example.

If two people have agreed to always use the number "2039" as the private key alongside with a certain function to create the random looking numbers, then afterwards they can send each others encrypted messages in public, which only they can decrypt using the private key. The following R-code demonstrates the issue.

[1] Just in case it's unclear, these input values are conceptually different from the inputs (letters) to the encryption algorithm.

No seed	**Seed**
	set.seed(2039)
sample(c(0,-1,1), 4, replace=TRUE)	sample(c(0,-1,1), 4, replace=TRUE)
[1] 0 0 -1 1	[1] 1 0 -1 0
	set.seed(2039)
sample(c(0,-1,1), 4, replace=TRUE)	sample(c(0,-1,1), 4, replace=TRUE)
[1] 1 1 -1 0	[1] 1 0 -1 0

The recipient of the encrypted message "14 15 14 14" can now generate the same four random numbers (+1, 0, -1, 0), deduct them from the encrypted message as we are now going the opposite way, and then apply the simple mapping from Figure 1 in reverse. This procedure yields the original message "MOON".

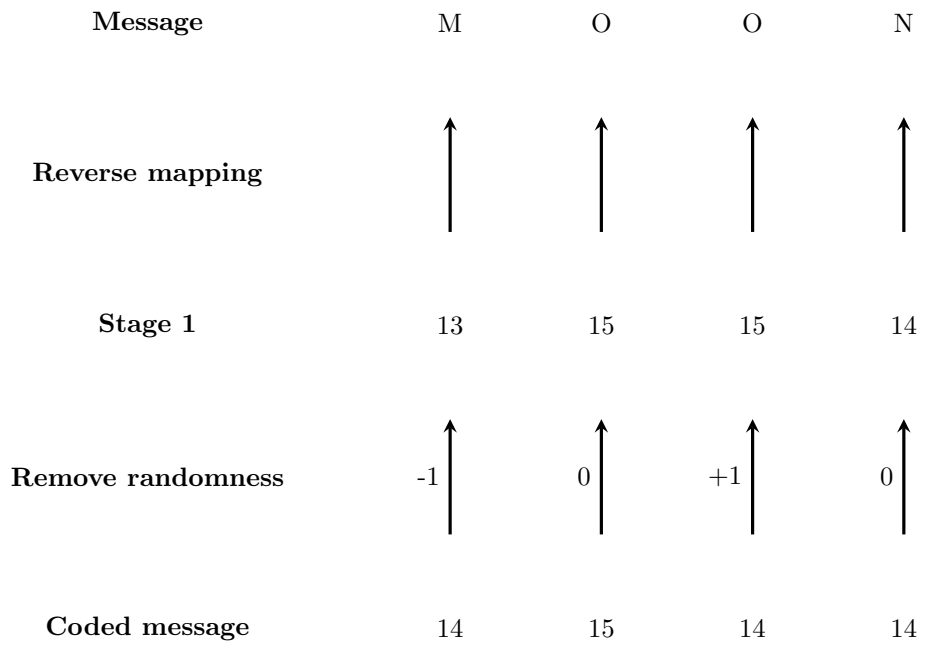

Figure 5: decryption using the private key "2039".

Final step

The encryption typically maps input characters into output characters, instead of numbers. The problem with numbers is the need to use more than one digit to represent some inputs.

We add the same initial mapping but in reverse to the end, to get the coded message in characters. The decryption happens similarly as in Figure 5, with the additional beginning step of first reverse mapping the coded message to numbers.

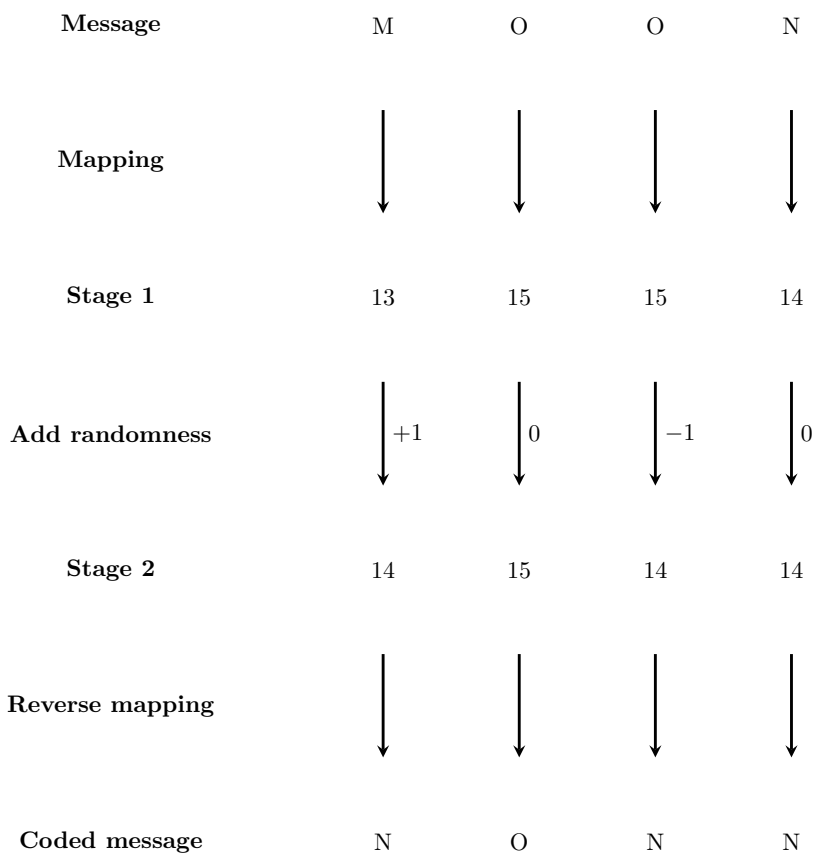

Figure 6: The full encryption algorithm.

Further

In our encryption algorithm there are three main components: First, mapping from letters to numbers, then adding pseudo-randomness to those numbers and finally a second mapping back from numbers to letters. So far we have specified quite simple solutions to all of these steps.

The mapping between characters into numbers can be improved by adopting for example the Unicode Transformation Format (UTF-8). In R the function utf8ToInt() maps letters and special characters (and even numbers) to numbers. Also the space " " between words in the message should be encrypted and the empty space should be added to the output options, to break the sentence structure in encryption. The inverse of this function is intToUtf8(), but the second mapping does not need to be the inverse of the first one.

The pseudo-randomness can also be added in much more versatile ways. For example, we could allow any input to be mapped into any of the output options, instead of just the adjacent ones. There can be many private keys and multiple stages of adding more mappings and randomness.

Hashing algorithm is another popular cryptographic tool. Unlike encryption algorithms, hashing algorithm is not meant to be reversed even by the friendly recipient. Suppose there is a publicly observed document and a sequence of numbers called the public key. When a hashing algorithm is used on the document with the private key, it outputs the public key. Only someone who knows the private key can use the hashing algorithm to re-create the public key, thereby having the power to verify or claim ownership for the document. Probably the best known example of this is the Secure Hash Algorithm (SHA-256), developed by the NSA, and used in blockchains behind many cryptocurrencies. Without the private key the best way is to guess what the private key could be and see if it produces the public key. The modern encryption algorithms have been designed in such a way that it is infeasible even with the best computers to obtain the private key this way.